

专题19：Zab协议（史上最全、定期更新）

本文版本说明：V2

此文的格式，由markdown 通过程序转成而来，由于很多表格，没有来的及调整，出现一个格式问题，尼恩在此给大家道歉啦。

由于社群很多小伙伴，在面试，不断的交流最新的面试难题，所以，《Java面试红宝书》，后面会不断升级，迭代。

本专题，作为 《Java面试红宝书》专题之一，《Java面试红宝书》一共30个面试专题，后续还会增加

《Java面试红宝书》升级的规划为：

后续基本上，**每一个月，都会发布一次**，最新版本，可以扫描扫架构师尼恩微信，发送“领取电子书”获取。

尼恩的微信二维码在哪里呢？

具体可以百度搜索 **疯狂创客圈 总目录**

面试问题交流说明：

如果遇到面试难题，或者职业发展问题，或者中年危机问题，都可以来 疯狂创客圈社群交流，

加入交流群，加尼恩微信即可，

入交流群，加尼恩微信即可，发送“**入群**”

推荐：疯狂创客圈 高质量 博文

高并发 必读 的精彩博文	
nacos 实战（史上最全）	sentinel（史上最全+入门教程）
Zookeeper 分布式锁（图解+秒懂+史上最全）	Webflux（史上最全）
SpringCloud_gateway（史上最全）	TCP/IP（图解+秒懂+史上最全）
10分钟看懂，Java NIO 底层原理	Feign原理（图解）
更多精彩博文	请参见【 疯狂创客圈 高并发 总目录 】

zookeeper的使用场景

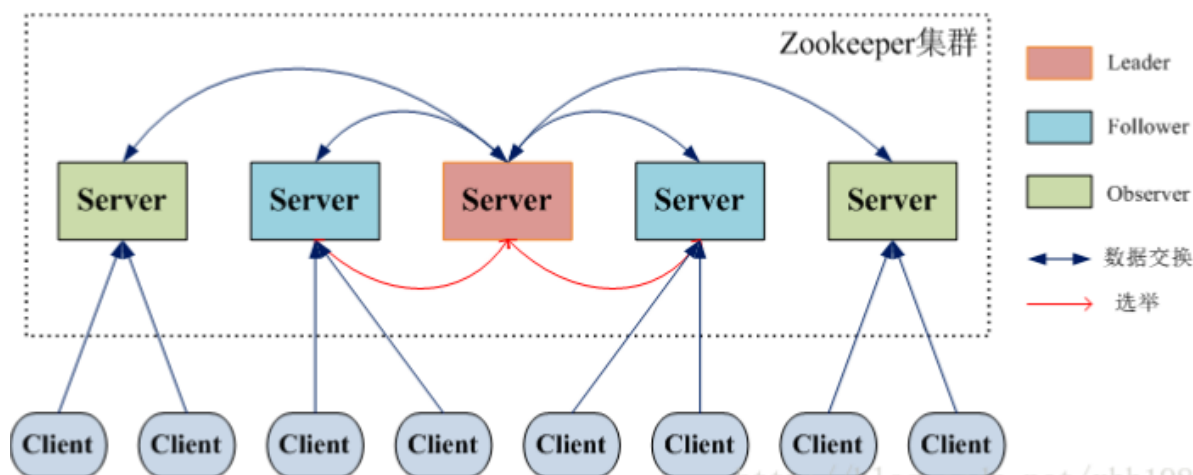
zookeeper是Apache开源的顶级项目，用于提供一个高性能、高可用，且具有严格的顺序访问控制能力（主要是写操作的严格顺序性）的分布式协调服务。可用于在分布式环境中保证数据的一致性，拥有非常广泛地使用场景。

主要的使用场景如下：

1. **实现数据的发布和订阅**：可以把一些可能变化的配置数据放入zookeeper服务器，客户端可以订阅并监听这个配置，一旦配置更新，就可以重新获取配置。
2. **命名服务**：有时分布式系统需要一些命名，比如提供RPC服务地址的名称、全局唯一不能重复的主键等，利用zookeeper节点不重名的特点可以进行命名服务。
3. **master选举**：有时分布式系统中存在多台服务器，必须选出一台master来执行特定的计算任务，master挂掉后，自动重新选出一台服务器担任master执行任务。这种情况可以使用zookeeper来实现，多个客户端创建同一个临时节点，只有一个可以成功，成功的节点就是master，其他客户端监听该节点，一旦被删除，说明master宕机，重新开始新的选举。
4. **负载均衡**：在消息队列的集群中，消息生产者需要比较均衡地将消息投递到不同的消息代理上，这里就涉及到负载均衡的使用。
5. **分布式锁**：后台接口分布式部署了以后，为了避免出现不同服务器的线程同时修改同一个数据引起并发问题，需要进行跨主机跨进程的线程同步，这是就用到分布式锁，主要基于zookeeper的临时有序节点来实现。
6. **分布式队列**：客户端提交的任务信息可以保存到zookeeper中，利用zookeeper有序节点的特性，实现一个先进先出的队列，zookeeper可以记录任务提交的拓扑信息并保持任务的有序调度。
7. **分布式协调和通知**：可以实现不同机器之间心跳监测（临时有序节点是否存在）、数据通信（向节点写入数据并监听变化）等场景。
8. **集群元数据管理**：每个集群的机器可以向zookeeper添加一个临时有序节点，只要节点存在表示机器存活。利用这个特点可以完成集群服务器的监控。还可以将主机的状态信息写入zookeeper的节点，监控中心订阅这些节点的数据来获得主机的实时信息。
9. 其他的需要分布式协调场景

Zookeeper的角色

- » 领导者（leader），负责进行投票的发起和决议，更新系统状态
- » 学习者（learner），包括跟随者（follower）和观察者（observer），follower用于接受客户端请求并想客户端返回结果，在选主过程中参与投票
- » Observer可以接受客户端连接，将写请求转发给leader，但observer不参加投票过程，只同步leader的状态，observer的目的是为了扩展系统，提高读取速度
- » 客户端（client），请求发起方



角色 ↗		描述 ↗
领导者 (Leader) ↗		领导者负责进行投票的发起和决议，更新系统状态 ↗
学习者 ↗ (Learner) ↗	跟随者 (Follower) ↗	Follower 用于接收客户请求并向客户端返回结果，在主过程中参与投票 ↗
	观察者 ↗ (Observer) ↗	Observer 可以接收客户端连接，将写请求转发给 leader 节点。但 Observer 不参加投票过程，只同步 leader 的状态。Observer 的目的是为了扩展系统，提高读取速度 ↗
客户端 (Client) ↗		请求发起方 ↗

每个Server在工作过程中有三种状态：

LOOKING：当前Server不知道leader是谁，正在搜寻

LEADING：当前Server即为选举出来的leader

FOLLOWING：leader已经选举出来，当前Server与之同步

什么是Observer？

zk3.3开始引入的角色，观察最新状态，并变更。与Follower不同只是不参与投票、选举，只提供非事务服务。

- Zookeeper需保证高可用和强一致性；
- 为了支持更多的客户端，需要增加更多Server；
- Server增多，投票阶段延迟增大，影响性能；
- 权衡伸缩性和高吞吐率，引入Observer
- Observer不参与投票；
- Observers接受客户端的连接，并将写请求转发给leader节点；
- 加入更多Observer节点，提高伸缩性，同时不影响吞吐率

Zab协议

Zookeeper 客户端会随机的连接到 zookeeper 集群中的一个节点。

注意，ZK集群会有很多节点，客户端在建立连接时，会随机挑选一个节点。

ZK集群对客户端的请求，按照类型（读、写两类）分开处理：

- 读请求

客户端直接从当前节点（其建立连接的节点）中读取数据；

- 写请求

这里涉及到了分布式事务。客户端就会向 Leader 提交事务，Leader 接收到事务提交，会广播该事务，只要超过半数节点写入成功，该事务就会被提交。

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议 的全称是 **Zookeeper Atomic Broadcast**（Zookeeper原子广播）。**Zookeeper 是通过 Zab 协议来保证分布式事务的最终一致性。**

ZAB与Paxos联系&区别

两者设计目标不一样，ZAB主要用于构建高可用分布式系统，Paxos 算法用于构建一致性状态机器。所有会有细微差别。但是ZAB就是在Paxos保证一致性基础上设计出高可用的协议。

ZAB 的基本概念

epoch周期值（比喻：年号）

acceptedEpoch（比喻：接受的年号）：follower已经接受leader更改年号的（newepoch）提议。

currentEpoch（比喻：当前的年号）：当前的年号

history：当前节点接受到事务提议的log

lastZxid：history中最近接收到的提议zxid(最大的值)

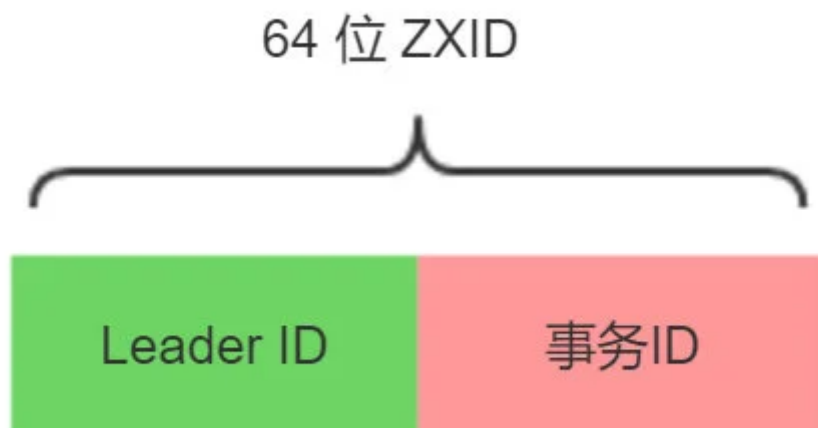
年号如何递增：

当产生新Leader的时候，就从这个Leader服务器上取出本地log中最大事务Zxid，从里面读出 epoch+1，作为一个新epoch，并将低32位置0（保证id绝对自增）

ZXID：事务的Proposal（提议）的id，可以简单理解为事务id

ZXID 是一个 64 位的数字，其中低 32 位可看作是计数器，Leader 服务器每产生一个新的事务 Proposal 的时候，都会该计数器进行加 1 操作。

ZXID 的高 32 位表示 Leader 周期 epoch 的编号，每当选举一个新的 Leader 服务器，就会从该服务器本地的事务日志中最大 Proposal 的 ZXID 中解析出对应的 epoch 值，然后对其加 1 操作，这个值就作为新的 epoch 值，并将低 32 位初始化为 0 来开始生成新的 ZXID。



Zookeeper 集群的模式

Zookeeper 集群的模式包括两种基本的模式：**崩溃恢复** 和 **消息广播**

崩溃恢复模式

当整个集群启动过程中，或者当 Leader 服务器出现网络中弄断、崩溃退出或重启等异常时，Zab协议就会进入**崩溃恢复模式**，选举产生新的Leader。

一但出现崩溃，会导致数据不一致，ZAB的崩溃恢复开始起作用。有如下两个确保：

1. ZAB协议需要确保已经在Leader提交的事务最终被所有服务器提交。
2. ZAB协议需要确保丢弃只在Leader服务器上被提出的事务。

针对上两个要求，如果Leader选举算法**保证新选举出来的Leader服务器拥有集群中所有机器最高编号（ZXID最大）的事务Proposal，那么就能保证新的Leader 一定具有已提交的所有提案**，更重要是，如果这么做，可以省去Leader服务器检查Proposal的提交和丢弃工作的这一步。

一旦Leader服务器出现崩溃，或者说网络原因导致Leader服务器失去了与过半的Follower的联系，那么就会进入崩溃恢复模式。为了保证程序的正常运行，整个恢复过程后需要选举一个新的Leader服务器。因此，ZAB协议需要一个高效可靠的Leader选举算法，从而确保能够快速的选举出新的Leader。同时，新的Leader选举算法不仅仅需要让Leader自己知道其自身已经被选举为Leader，同时还需要让集群中所有的其他机器也能够快速的感知选举产生的新的Leader服务器。

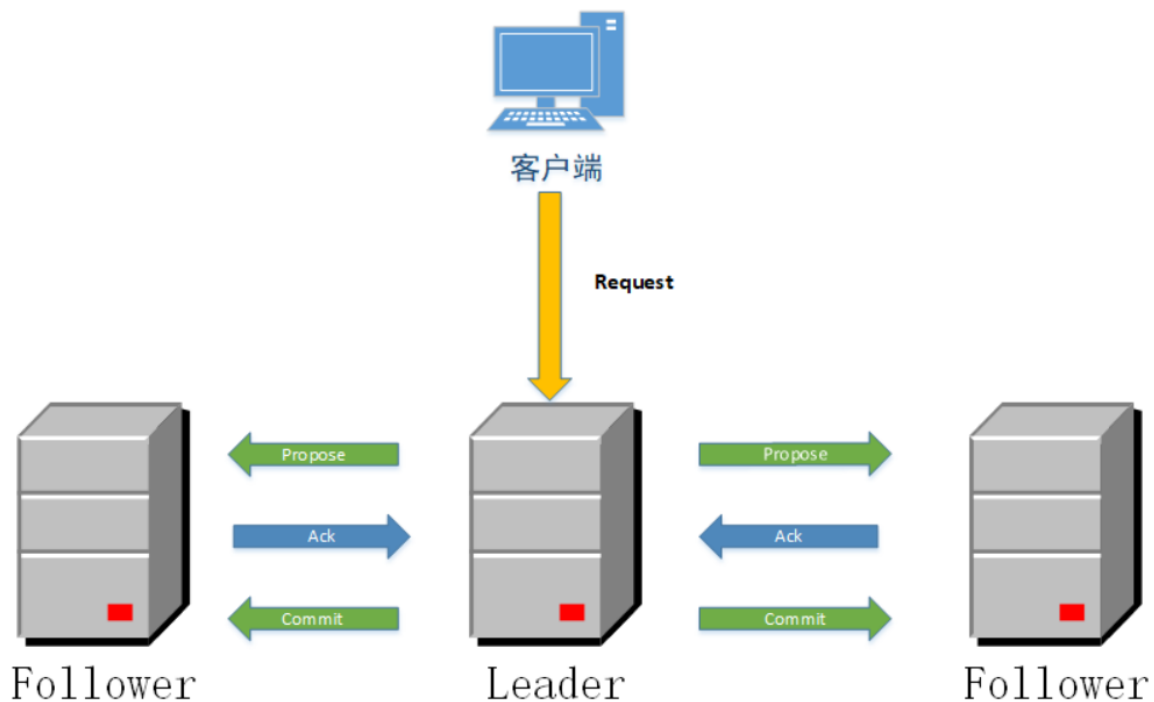
ZAB协议规定了如果一个事务Proposal在一台机器上被处理成功，那么应该在所有的机器上都被处理成功，哪怕机器出现崩溃。

消息广播模式

当新的Leader出来了，同时，已有过半机器完成同步之后，ZAB协议将退出恢复模式。**进入消息广播模式**。这时，如果有一台遵守Zab协议的服务器加入集群，因为此时集群中已经存在一个Leader服务器在广播消息，那么该新加入的服务器自动进入恢复模式：找到Leader服务器，并且完成数据同步。同步完成后，作为新的Follower一起参与到消息广播流程中。

如果集群中其他机器收到客户端事务请求后，那么会先转发Leader服务器，由Leader统一处理。

1. 在zookeeper集群中，数据副本的传递策略就是采用消息广播模式。zookeeper中数据副本的同步方式与二段提交相似，但是却又不同。二段提交要求协调者必须等到所有的参与者全部反馈ACK确认消息后，再发送commit消息。要求所有的参与者要么全部成功，要么全部失败。二段提交会产生严重的阻塞问题。
2. Zab协议中 Leader 等待 Follower 的ACK反馈消息是指“只要半数以上的Follower成功反馈即可，不需要收到全部Follower反馈”
3. 整个过程中，Leader为每个事务请求生产对应的Proposal，在广播前，为这个事务分配一个全局唯一ID，为ZXID（事务ID），必须按照递增的事务顺序进行处理。
4. 具体流程如下图。



ZAB协议消息广播流程示意图

<https://blog.csdn.net/crazymakercircle>

ZAB协议中涉及的二阶段提交和2pc有所不同。在ZAB协议的二阶段提交过程中，移除了中断逻辑，所有Follower服务器要么正常反馈Leader提出的事务Proposal，要么就抛弃Leader服务器。ZAB协议中，只要集群中过半的服务器已经反馈ACK，就开始提交事务了，不需要等待集群中所有的服务器都反馈响应。这种模型是无法处理Leader服务器崩溃退出而带来的数据不一致问题的，因此在ZAB协议中添加了另一个模式，即采用崩溃恢复模式来解决这个问题。此外，整个消息广播协议是基于具有FIFO特性的TCP协议来进行网络通信的，因此能够很容易保证消息广播过程中消息接受与发送的顺序性。

在整个消息广播过程中，Leader服务器会为每个事务请求生成对应的Proposal来进行广播，并且在广播事务Proposal之前，Leader服务器会首先为这个事务分配一个全局单调递增的唯一ID，我们称之为事务ID（即ZXID）。由于ZAB协议需要保证每一个消息严格的因果关系，因此必须将每一个事务Proposal按照其ZXID的先后顺序来进行排序与处理。

在消息广播过程中，Leader服务器会为每一个Follower服务器各自分配一个单独的队列，然后将需要广播的事务Proposal依次放入这些队列中，并且根据FIFO策略进行消息发送。每一个Follower服务器在接收到这个事务Proposal之后，都会首先将其以事务日志的形式写入到本地磁盘中去，并且在成功写入后反馈给Leader服务器一个ACK响应。当Leader服务器接收到超过半数Follower的ACK响应后，就会广播一个Commit消息给所有Follower服务器以通知其将事务进行提交，同时Leader自身也会完成事务的提交，而每一个Follower服务器收到Commit消息之后，也会完成对事务的提交。

ZAB的两种基本模式？

崩溃恢复：在正常情况下运行非常良好，一旦Leader出现崩溃或者由于网络原因导致Leader服务器失去了与过半Follower的联系，那么就会进入崩溃恢复模式。为了程序的正确运行，整个恢复过程后需要选举出一个新的Leader，因此需要一个高效可靠的选举方法快速选举出一个Leader。

消息广播：类似一个两阶段提交过程，针对客户端的事务请求，Leader服务器会为其生成对应的事务Proposal，并将其发送给集群中的其余所有机器，再分别收集各自的选票，最后进行事务提交。

哪些情况会导致ZAB进入恢复模式并选取新的Leader?

启动过程或Leader出现网络中断、崩溃退出与重启等异常情况时。

当选举出新的Leader后，同时集群中已有过半的机器与该Leader服务器完成了状态同步之后,ZAB就会退出恢复模式。

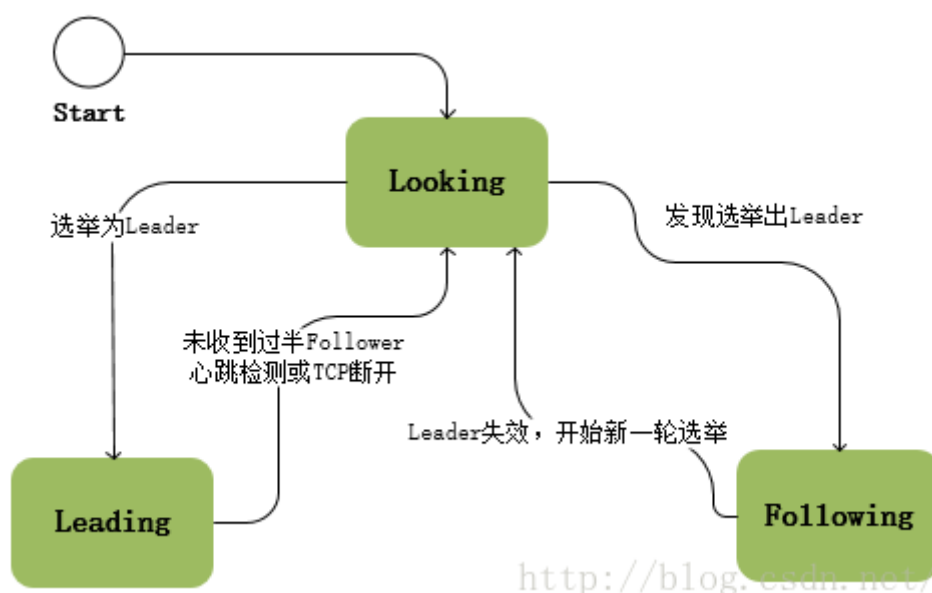
ZAB协议的选主：

ZAB协议中存在着**三种**状态，每个节点都属于以下三种中的一种：

1. Looking /election：系统刚启动时或者Leader崩溃后正处于选举状态
2. Following：Follower节点所处的状态，Follower与Leader处于数据同步阶段；
3. Leading：Leader所处状态，当前集群中有一个Leader为主进程；

在ZooKeeper的整个生命周期中，每个节点都会在Looking、Following、Leading状态间不断转换。

启动之初，ZooKeeper所有节点初始状态为Looking，这时集群会尝试选举出一个Leader节点，选举出的Leader节点切换为Leading状态；当节点发现集群中已经选举出Leader则该节点会切换到Following状态，然后和Leader节点保持同步；当Follower节点与Leader失去联系时Follower节点则会切换到Looking状态，开始新一轮选举；



Zookeeper leader 选举图解

- 半数通过
 - 3台机器 挂一台 $2 > 3/2$
 - 4台机器 挂2台 $2! > 4/2$

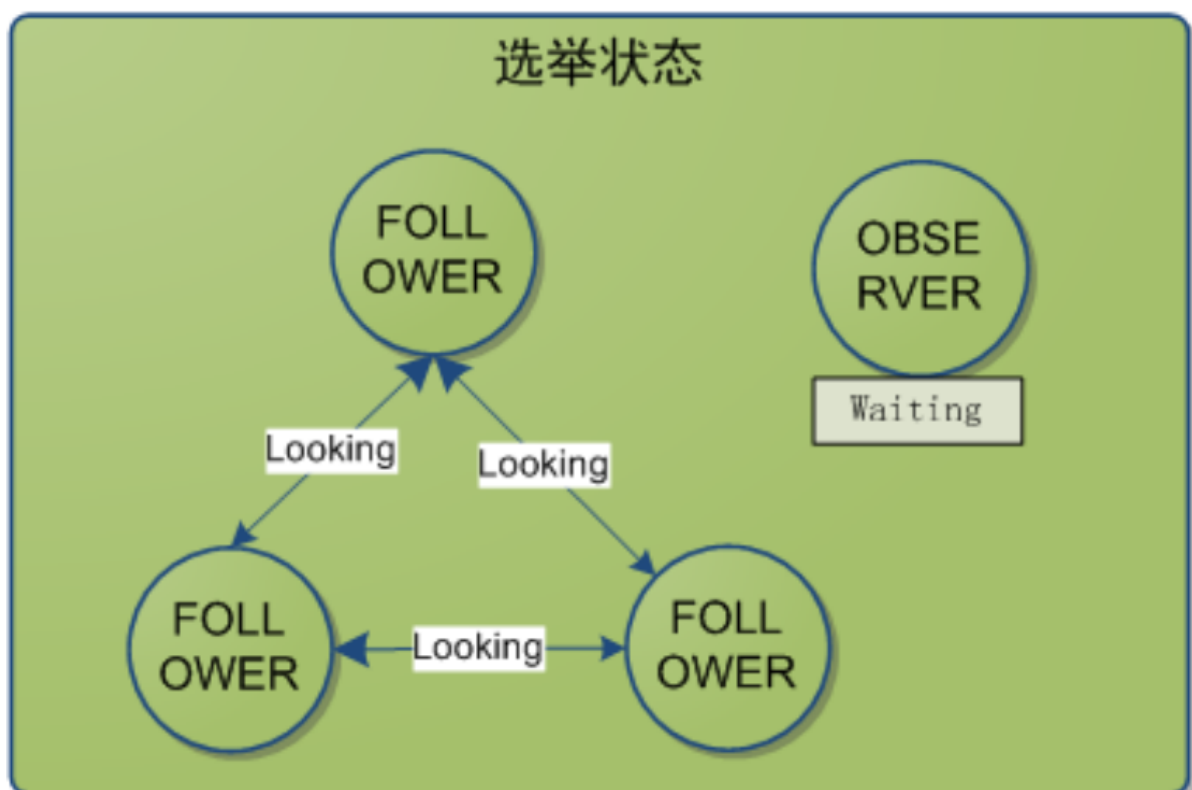
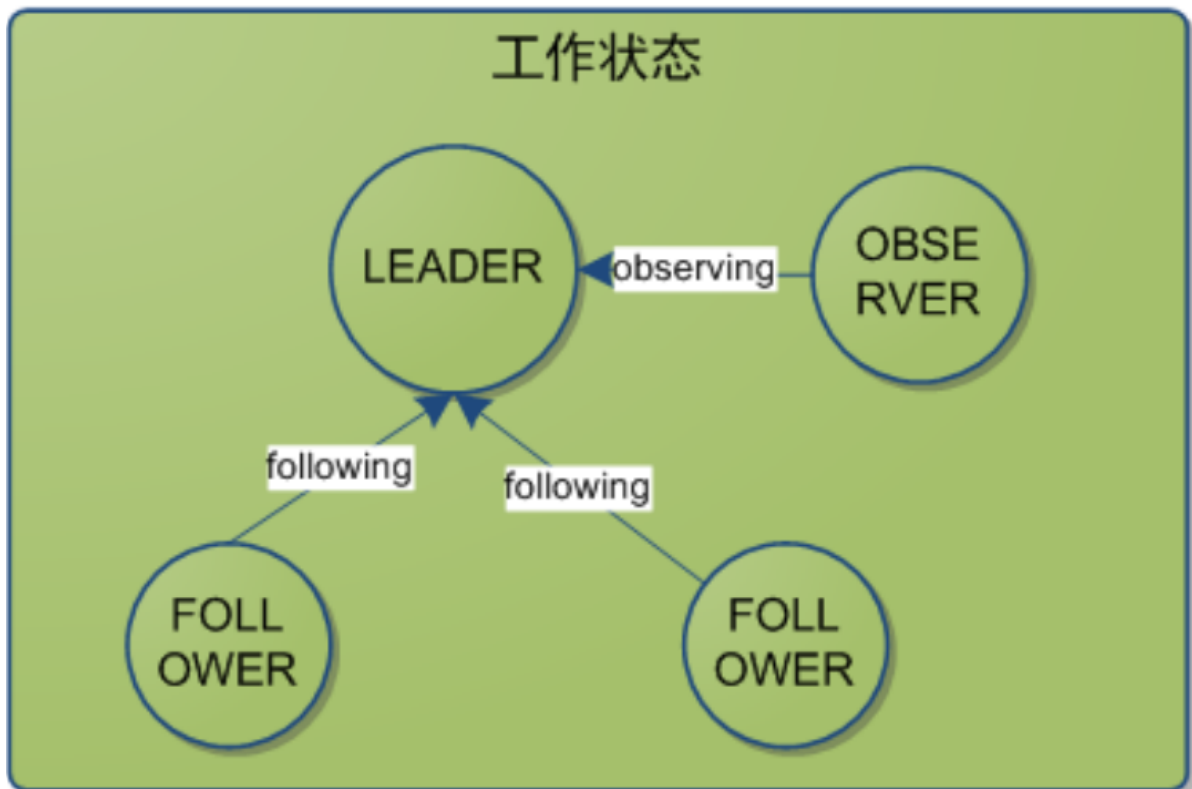
• A提案说，我要选自己，B你同意吗？C你同意吗？B说，我同意选A；C说，我同意选A。(注意，这里超过半数了，其实在现实世界选举已经成功了。

但是计算机世界是很严格，另外要理解算法，要继续模拟下去。)

• 接着B提案说，我要选自己，A你同意吗？A说，我已经超半数同意当选，你的提案无效；C说，A已经超半数同意当选，B提案无效。

• 接着C提案说，我要选自己，A你同意吗？A说，我已经超半数同意当选，你的提案无效；B说，A已经超半数同意当选，C的提案无效。

• 选举已经产生了Leader，后面的都是follower，只能服从Leader的命令。而且这里还有个细节，就是其实谁先启动谁当头。



选主的细分场景

分析以上两种细分场景：

1. 启动时期选举：

- 每个Server发出一个投票
- 接受来自各个服务器的投票
- 处理投票（优先检查ZXID，相同就比较myid）
- 统计投票（判断是否已经有过半的机器接收到相同的投票信息，所谓“过半”就是指大于集群机器数量的一半，即大于或等于 $(n/2+1)$ 。对于这里由3台机器构成的集群，大于等于2台即为达到“过半”要求。）
- 改变服务器状态：Leader->LEADING, Follower->FOLLOWING

2. 服务运行期间的Leader选举：

- 变更状态：Leader挂后，剩下的Follower都变成LOOKING，进入Leader选举
- 每个Server发出投票，第一轮都投自己，然后将自己投票发给所有机器
- 接收投票，与启动选举相同
- 处理投票，与启动选举相同
- 统计投票，与启动选举相同
- 改变服务器状态，与启动选举相同

为什么zookeeper集群的数目，一般为奇数个？

- Leader选举算法采用了Paxos协议；
- Paxos核心思想：当多数Server写成功，则任务数据写成功如果有3个Server，则两个写成功即可；如果有4或5个Server，则三个写成功即可。
- Server数目一般为奇数（3、5、7）如果有3个Server，则最多允许1个Server挂掉；如果有4个Server，则同样最多允许1个Server挂掉由此，

我们看出3台服务器和4台服务器的容灾能力是一样的，所以为了节省服务器资源，一般我们采用奇数个，作为服务器部署个数。

FastLeaderElection选主算法（选主算法可在zoo.cfg中配置）

ZAB 默认采用 TCP 版本的 FastLeaderElection 选举算法。在选举投票消息中包含了两个最基本的信息：所推举的服务器 SID 和 ZXID，分别表示被推举服务器的唯一标识（每台机器不一样）和事务 ID。假如投票信息为（SID, ZXID）的形式。在第一次投票的时候，由于还无法检测集群其他机器的状态信息，因此每台机器都将自己作为被推举的对象来进行投票。每次对收到的投票，都是一个对投票信息（SID, ZXID）对比的过程，规则如下：

- 1 逻辑时钟clock表示第几轮选举，重启时为0，选举时加1
- 2 启动时处于looking状态
- 3 选举细节

- 先选自己，选票为“clock(1)、looking、zxid、myid”
- 广播选票，收到其他节点选票
- 如果其他节点选票的clock大，更新自己的clock；
- 如果接收到的投票 ZXID 大于自己的 ZXID，就认可当前收到的投票，则更新自己选票的相应值为该节点值，表示选举该节点。
- 如果接收到的投票 ZXID 小于自己的 ZXID，那么就坚持的投票，不做任何变更。
- 如果接收到的投票 ZXID 等于自己的 ZXID，再对比 myid，比自己大，就认可当前收到的投票，表示选举该节点；如果比自己小，那就坚持自己的投票，不做变更。

4 重新发出选票

经过第二次投票后，集群中每台机器都会再次收到其他机器的投票，然后开始统计，如果一台机器收到了超过了半数的相同投票，那么这个投票对应的 myid机器即为 Leader。

简单来说，通常哪台服务器上的数据越新，那么越有可能成为 Leader。原因很简答，数据越新，也就越能够保证数据的恢复。当然，如果集群中有几个服务器具有相同的 ZXID，那么 myid较大的那台服务器成为 Leader。

结束选主之后的进入消息广播模式

选举出Leader节点后ZAB进入原子广播阶段，Leader与Follower使用心跳检测来感知对方的存在：

- 1 当Leader节点在超时时间内能正常收到来自Follower的心跳，那Follower节点会一直与该节点保持连接；
- 2 若超时时间内Leader没有接收到来自过半Follower节点的心跳检测或TCP连接断开，那Leader会结束当前周期的领导，切换到Looking状态，所有Follower节点也会放弃该Leader节点切换到Looking状态，然后开始新一轮选举；

选主的过程

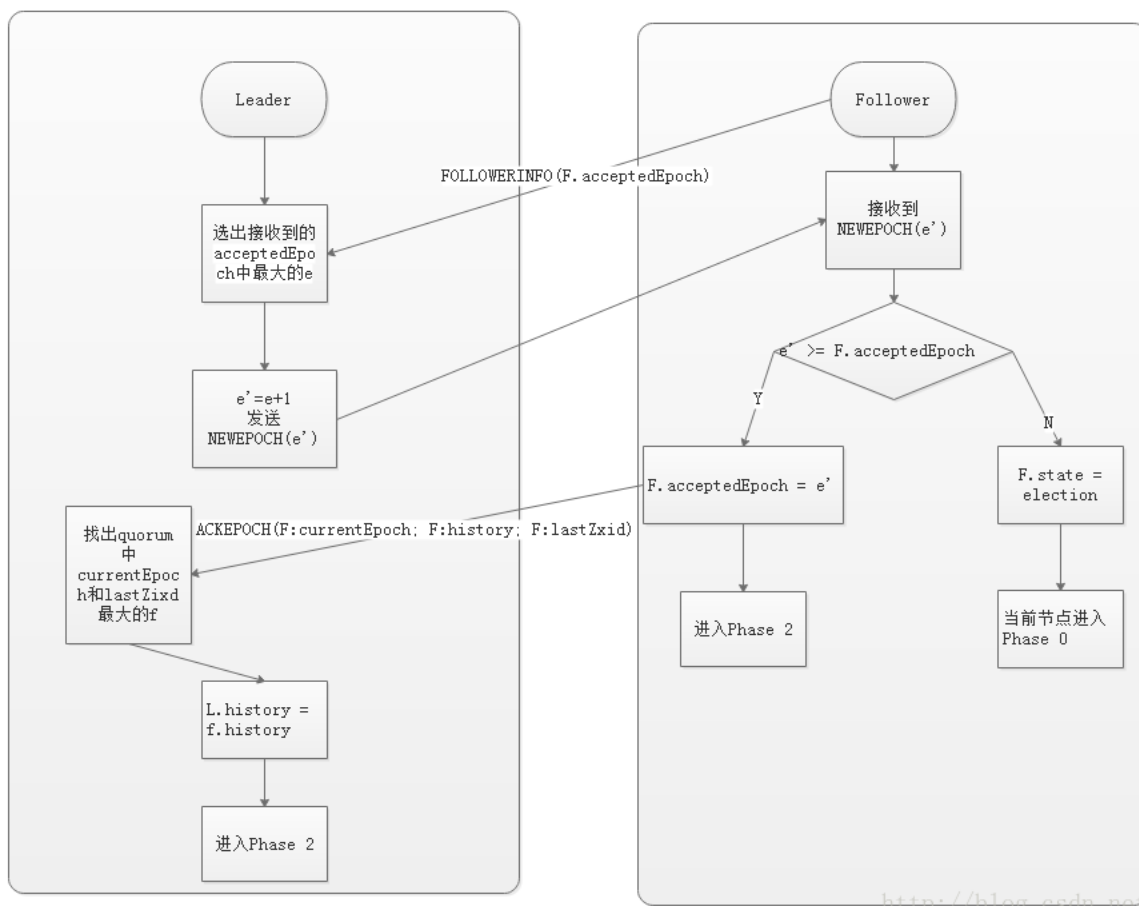
Phase 0: Leader election（选举阶段）

节点在一开始都处于选举阶段，只要有一个节点得到超半数节点的票数，它就可以当选准 leader。只有到达 Phase 3 准 leader 才会成为真正的 leader。这一阶段的目的是就是为了选出一个准 leader，然后进入下一个阶段。

协议并没有规定详细的选举算法，后面我们会提到实现中使用的 **Fast Leader Election**。

Phase 1: Discovery（发现阶段）

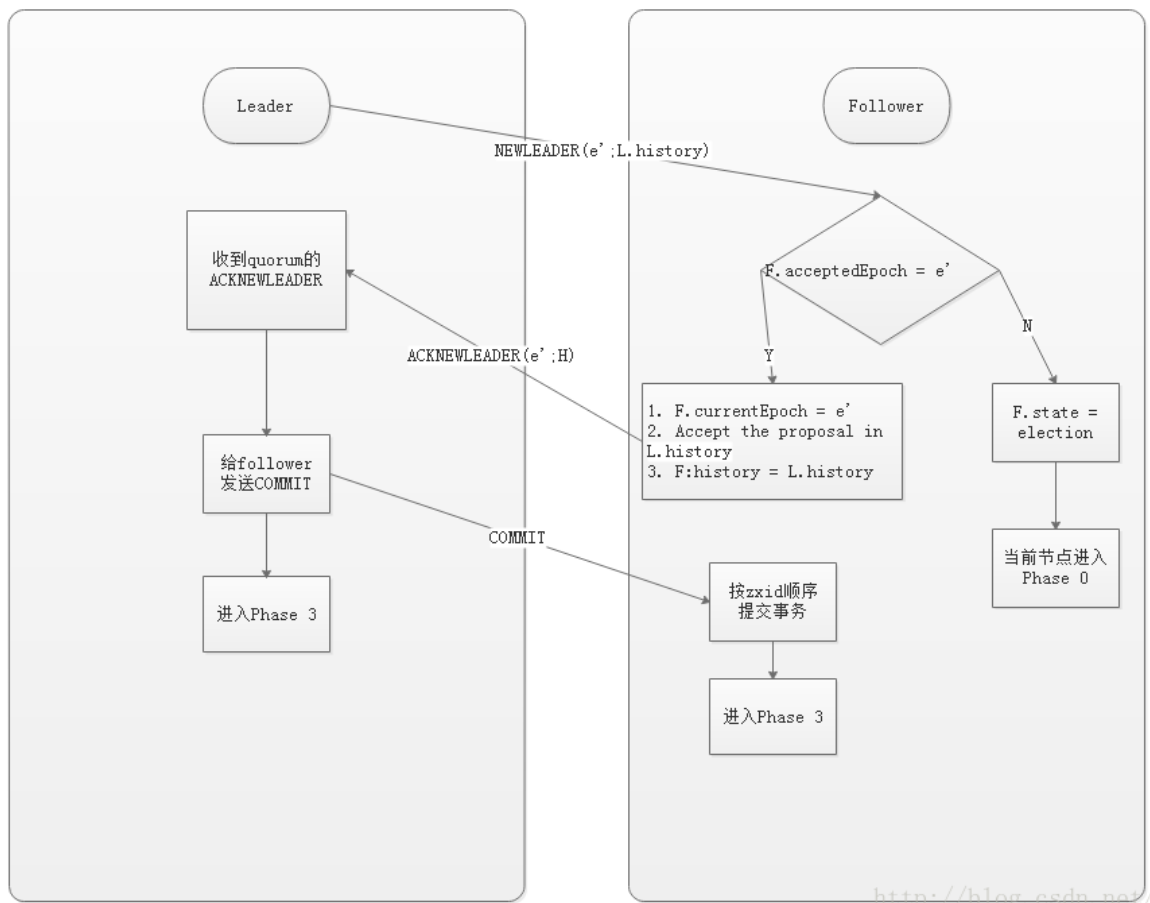
在这个阶段，followers 跟准 leader 进行通信，同步 followers 最近接收的事务提议。这个一阶段的主要目的是发现当前大多数节点接收的最新提议，并且准 leader 生成新的 epoch，让 followers 接受，更新它们的 accepted Epoch



一个 follower 只会连接一个 leader，如果有一个节点 f 认为另一个 follower p 是 leader，f 在尝试连接 p 时会被拒绝，f 被拒绝之后，就会进入 Phase 0。

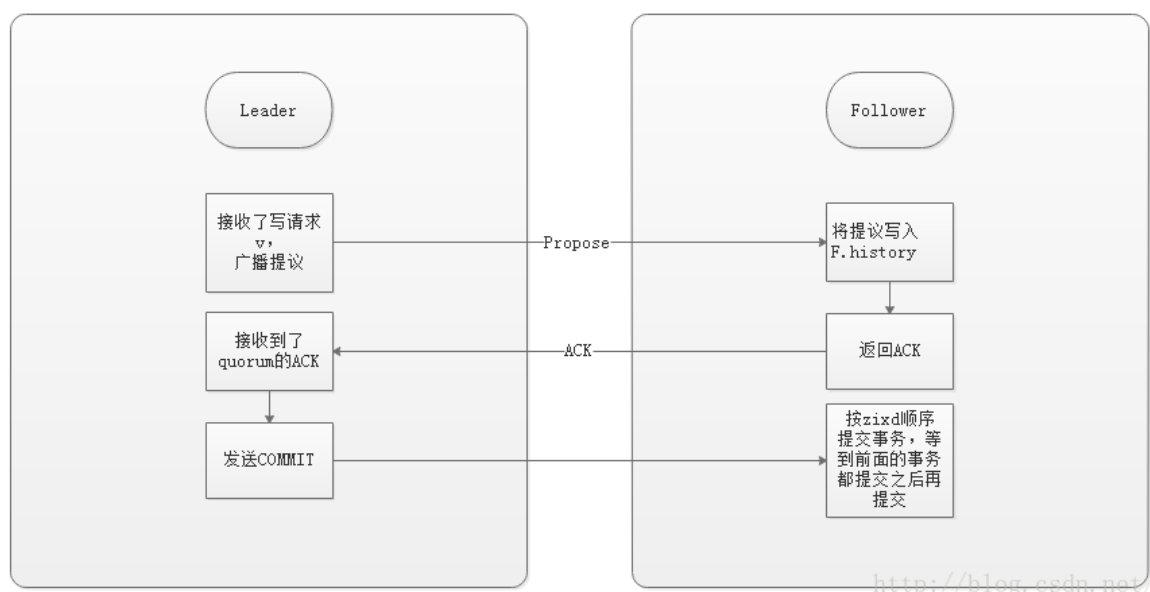
Phase 2: Synchronization (同步阶段)

同步阶段主要是利用 leader 前一阶段获得的最新提议历史，同步集群中所有的副本。只有当 quorum 都同步完成，准 leader 才会成为真正的 leader。follower 只会接收 zxid 比自己的 lastZxid 大的提议。



Phase 3: Broadcast (广播阶段)

到了这个阶段，Zookeeper 集群才能正式对外提供事务服务，并且 leader 可以进行消息广播。同时如果有新的节点加入，还需要对新节点进行同步。



值得注意的是，ZAB 提交事务并不像 2PC 一样需要全部 follower 都 ACK，只需要得到 quorum（超过半数的节点）的 ACK 就可以了。

Zookeeper节点的数据同步

完成Leader选举之后，在正式开始工作（即接收客户端的事务请求，然后提出新的提案）之前，Leader服务器会首先确认事务日志中的所有Proposal都已经被集群中过半的机器提交了，即是否完成数据同步。

集群中所有的正常运行的服务器，要么成为Leader，要么成为Follower并和Leader保持同步。Leader服务器需要确保所有的Follower服务器能够接收到每一条事务Proposal，并且能够正确地将所有已经提交的事务Proposal应用到内存数据库中。Leader服务器会为每一个Follower服务器都准备一个队列，并将没有被各Follower服务器同步的事务以Proposal消息形式逐个发送到Follower服务器，并在每一个Proposal消息后紧跟着再发送一个Commit消息，以表示这个事务已经被提交。等到Follower服务器将所有尚未同步的事务Proposal都从Leader服务器上同步过来并成功应用到本地数据库中后，Leader服务器就会将改Follower服务器加入到真正可用的Follower列表中，并开始之后的其他流程。

ZAB协议如何处理那些需要被丢弃的事务Proposal的？

在ZAB协议的事务编号ZXID设计中，ZXID是一个64位数字，其中低32位可以看做一个简单的单调递增的计数器，针对客户端的每一个事务请求，Leader服务器在产生一个新的事务Proposal的时候，都会对改计数器进行加一操作；而高32位则代表了leader周期epoch的编号，每当选举产生一个新的Leader服务器，就会从这个Leader服务器取出本地日志中最大事务Proposal的ZXID，并从ZXID中解析出对应的epoch值，然后再对其进行加1操作，之后的编号就会作为新的epoch，并将低32位置0来开始生成新的ZXID。ZAB协议中的这一通过epoch编号来区分Leader周期变化的策略，能够有效避免不同的Leader服务器错误的使用相同的ZXID编号提出不一样的事务的情况。

基于这样的策略，当一个包含了上一个Leader周期中尚未提交过的事务服务器启动时，肯定无法成为leader。因为当前集群中肯定包含一个Quorum集合，该集合中机器一定包含了更高的epoch的事务Proposal。

Zab协议的事务

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch（有点类似年代、年号）用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的统治时期。低32位用于递增计数。

那么什么是zxid呢？

ZooKeeper状态的每一次改变, 都对应着一个递增的Transaction id, 该id称为zxid. 由于zxid的递增性质, 如果zxid1小于zxid2, 那么zxid1肯定先于zxid2发生.

创建任意节点, 或者更新任意节点的数据, 或者删除任意节点, 都会导致Zookeeper状态发生改变, 从而导致zxid的值增加.

Zab协议的核心：

Zab协议的核心：**定义了事务请求的处理方式**

- 1) 所有的事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器被叫做 **Leader服务器**。其他剩余的服务器则是 **Follower服务器**。
- 2) Leader服务器 负责将一个客户端事务请求，转换成一个 **事务Proposal**，并将该 Proposal 分发给集群中所有的 Follower 服务器，也就是向所有 Follower 节点发送数据广播请求（或数据复制）
- 3) 分发之后，Leader服务器需要等待所有Follower服务器的反馈（Ack请求），**在Zab协议中，只要超过半数的Follower服务器进行了正确的反馈后**，那么 Leader 就会再次向所有的 Follower服务器发送 Commit 消息，要求其将上一个 事务proposal 进行提交。（注意：有点像2PC）

广播模式下只有主节点可以发送广播消息，如果某个从节点需要发送广播信息，也需要通过主节点进行。

Zab协议的事务特点：

消息广播阶段的数据写入策略，通过事务完成，有以下特点：

- 1) 在zookeeper集群中，数据副本的传递策略就是采用消息广播模式。zookeeper中数据副本的同步方式与二段提交相似，但是却又不同。二段提交要求协调者必须等到所有的参与者全部反馈ACK确认消息后，再发送commit消息。要求所有的参与者要么全部成功，要么全部失败。二段提交会产生严重的阻塞问题。
- 2) Zab协议中 Leader 等待 Follower 的ACK反馈消息是指“只要半数以上的Follower成功反馈即可，不需要收到全部Follower反馈”

Zab协议的事务过程：

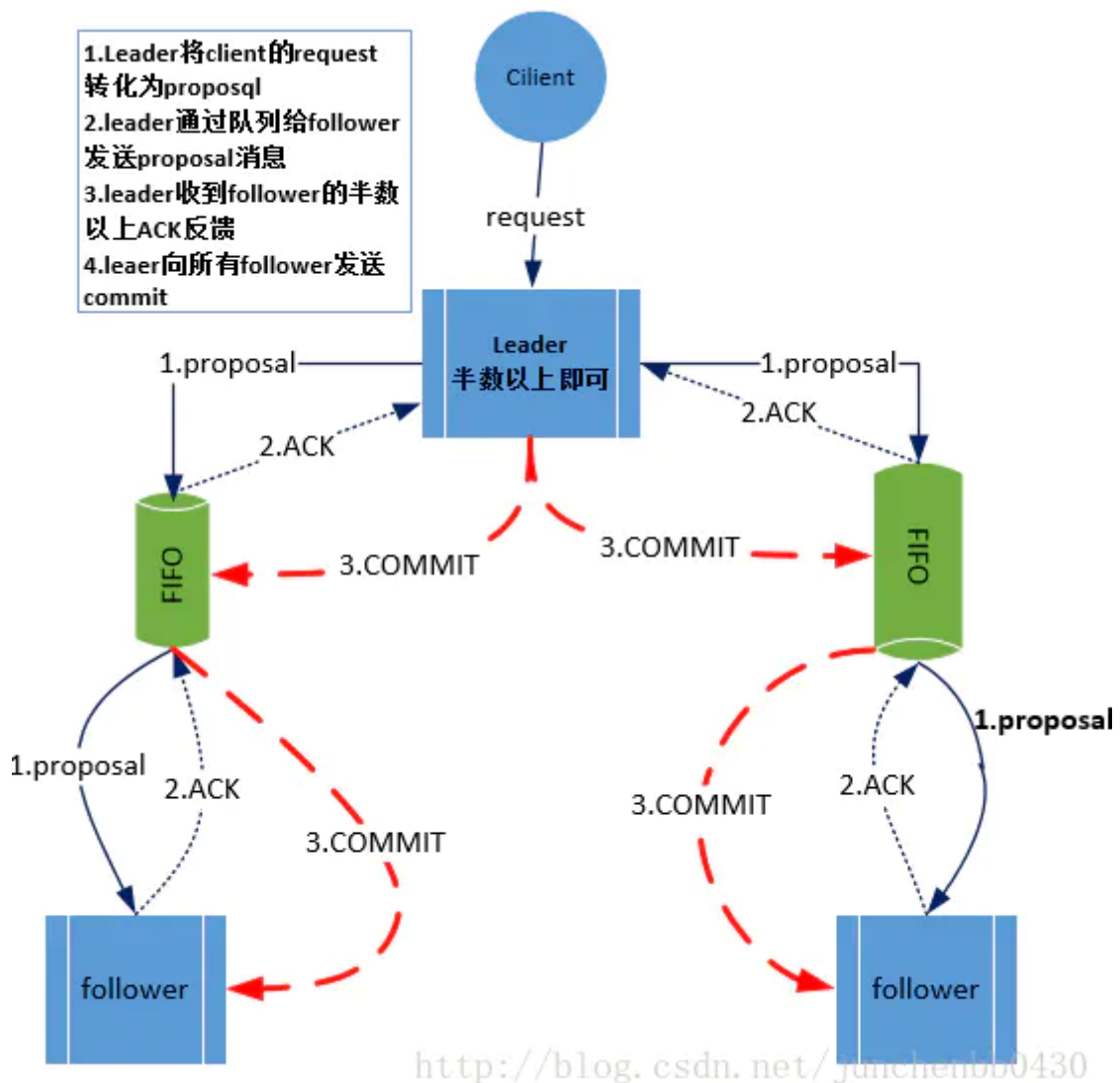
消息广播具体步骤如下：

- 1) 客户端发起一个写操作请求。
- 2) Leader 服务器将客户端的请求转化为事务 Proposal 提案，同时为每个 Proposal 分配一个全局的ID，即zxid。
- 3) Leader 服务器为每个 Follower 服务器分配一个单独的队列，然后将需要广播的 Proposal 依次放到队列中取，并且根据 FIFO 策略进行消息发送。
- 4) Follower 接收到 Proposal 后，会首先将其以事务日志的方式写入本地磁盘中，写入成功后向 Leader 反馈一个 Ack 响应消息。
- 5) Leader 接收到超过半数以上 Follower 的 Ack 响应消息后，即认为消息发送成功，可以发送 commit 消息。

6) Leader 向所有 Follower 广播 commit 消息，同时自身也会完成事务提交。Follower 接收到 commit 消息后，会将上一条事务提交。

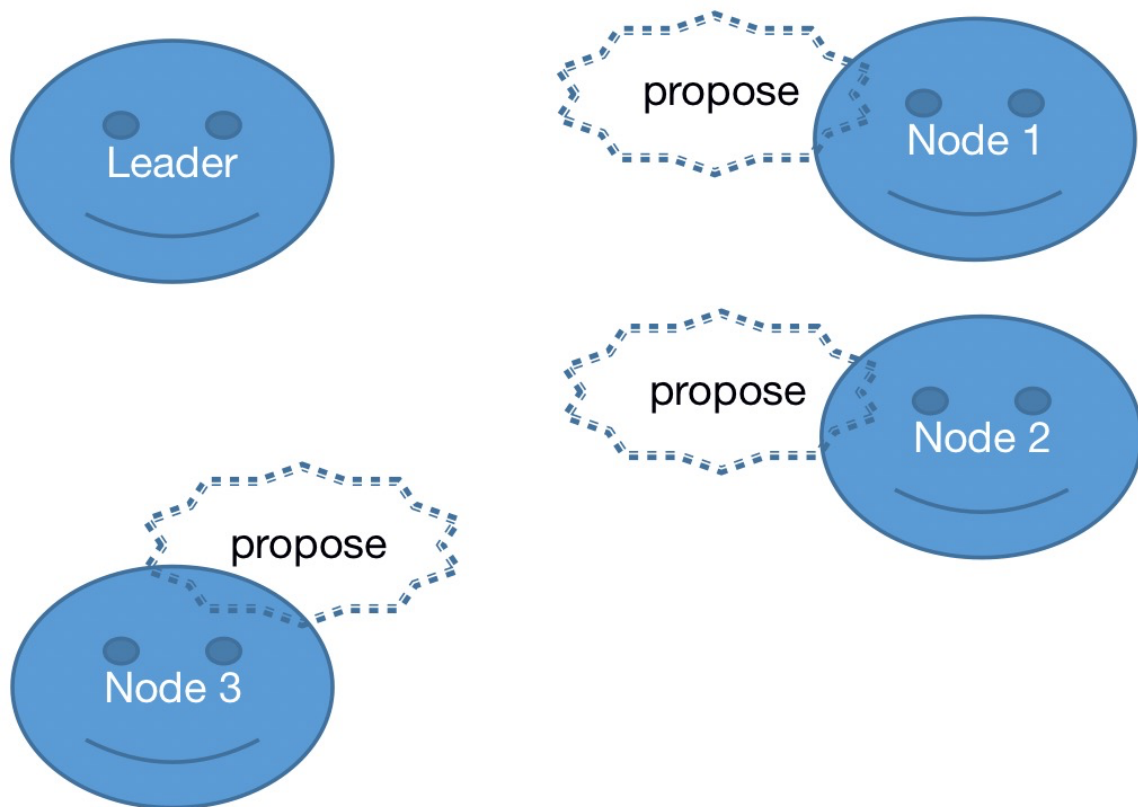
zookeeper 采用 Zab 协议的核心，就是只要有一台服务器提交了 Proposal，就要确保所有的服务器最终都能正确提交 Proposal。这也是 CAP/BASE 实现最终一致性的一个体现。

Leader 服务器与每一个 Follower 服务器之间都维护了一个单独的 FIFO 消息队列进行收发消息，使用队列消息可以做到异步解耦。Leader 和 Follower 之间只需要往队列中发消息即可。如果使用同步的方式会引起阻塞，性能要下降很多。



Zab协议的事务过程图解:

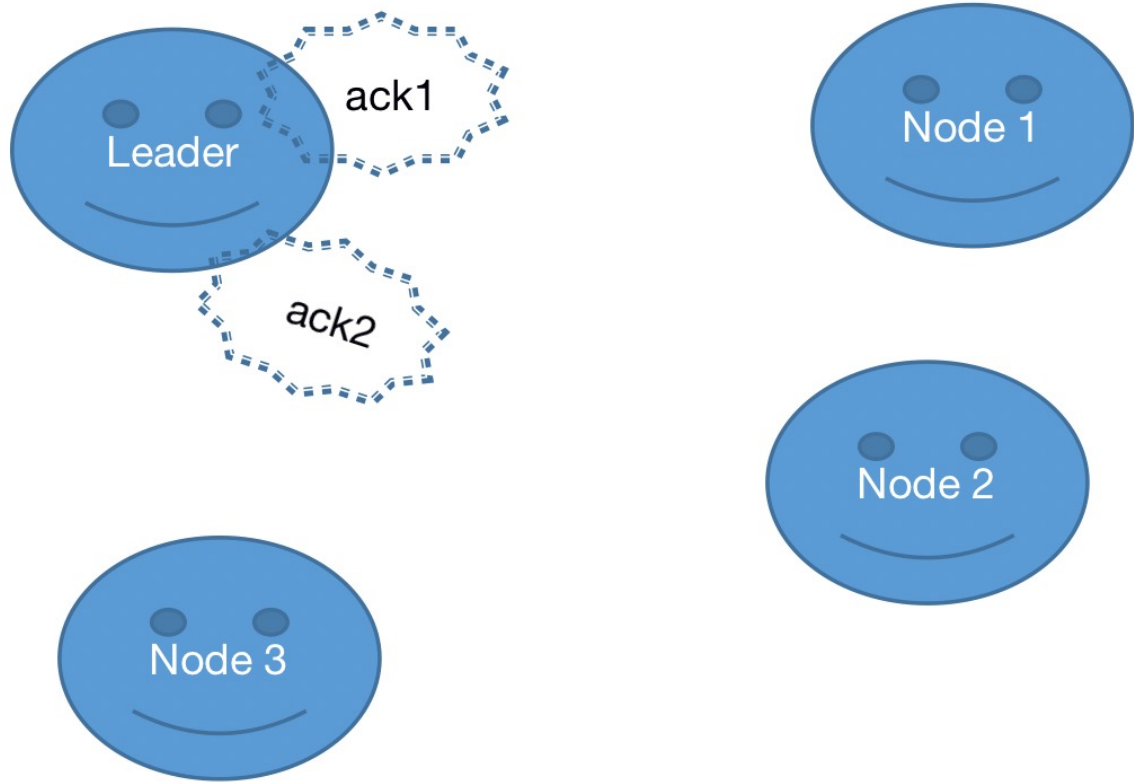
1、主节点广播发送事务提交提议



包括以下步骤：

- 针对客户端的事务请求，leader服务器会先将该事务写到本地的log文件中
- 然后，leader服务器会为这次请求生成对应的事务Proposal并且为这个事务Proposal分配一个全局递增的唯一的的事务ID，即Zxid
- leader服务器会为每一个follower服务器都各自分配一个单独的队列，将需要广播的事务Proposal依次放入队列中，发送给每一个follower

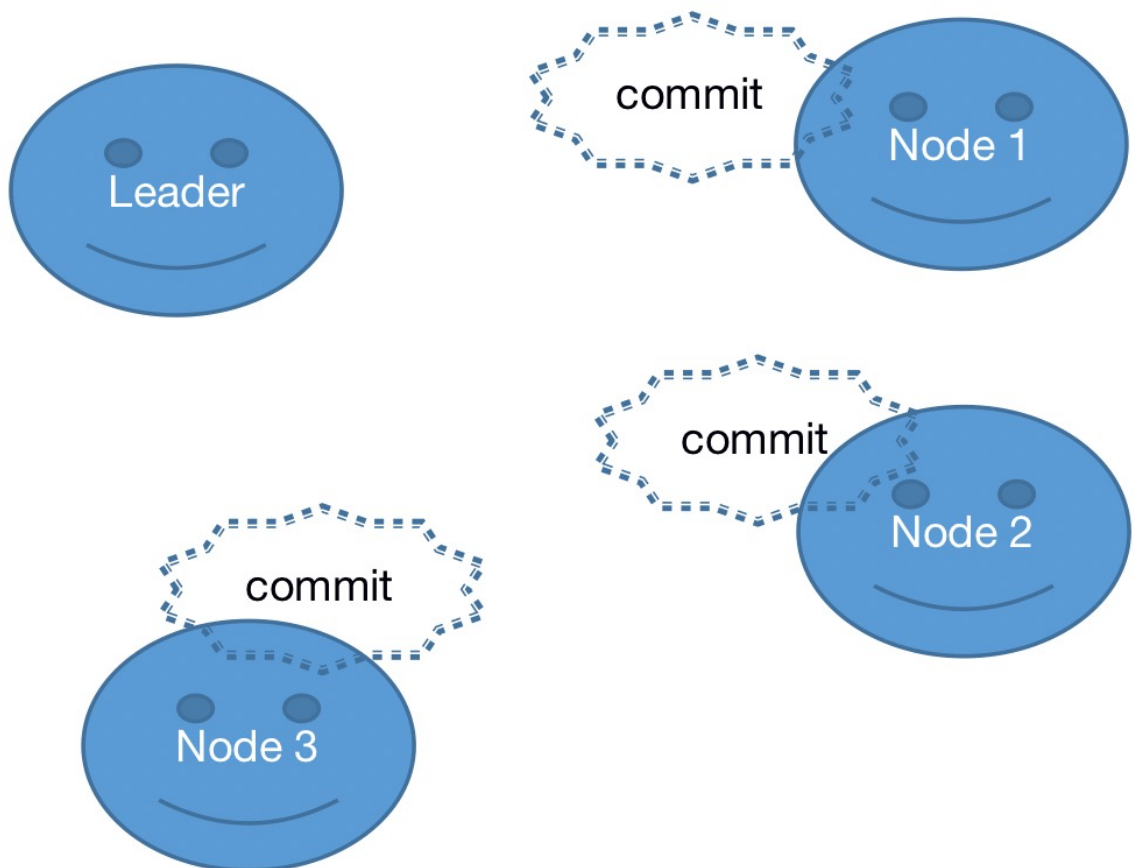
2、从节点接收到提议后，回复确认信息通知主节点



包括以下步骤：

- d. 每一个follower在收到队列之后，会从队列中依次取出事务Proposal，写入本地的事务日志中。如果写成功了，则给leader返回一个ACK消息

3、主节点接收到超过法定数量从节点确认信息后，广播发送事务提交命令到从节点



e. 当leader服务器接收到半数的follower的ACK相应之后，就会广播一个Commit消息给所有的follower以通知其进行事务提交，同时leader自身也进行事务提交

当然，在这种简化了的二阶段提交模型下，是无法处理Leader服务器崩溃退出而带来的数据不一致问题的，因此在ZAB协议中添加了另一个模式，即采用崩溃恢复模式来解决这个问题

整个消息广播协议是基于具有FIFO特性的TCP协议来进行网络通信的，因此能够很容易地保证消息广播过程中消息接收与发送的顺序性

zookeeper 服务实现中很重要的一点：顺序性。顺序请求，顺序响应；主节点事务顺序提交，从节点按顺序响应事务等等。

ZK性能问题

客户端对ZK的更新操作都是永久的，不可回退的，也就是说，一旦客户端收到一个来自server操作成功的响应，那么这个变更就永久生效了。为做到这点，ZK会将每次更新操作以事务日志的形式写入磁盘，写入成功后才会给予客户端响应。明白这点之后，你就会明白磁盘的吞吐性能对于ZK的影响了，磁盘写入速度制约着ZK每个更新操作的响应。

事务日志的写性能确实对ZK性能，尤其是更新操作的性能影响很大，为了尽量减少ZK在读写磁盘上的性能损失，可以考虑使用单独的磁盘作为事务日志的输出（使用单独的挂载点用于事务日志的输出）

ZK的事务日志输出是一个顺序写文件的过程，本身性能是很高的，所以尽量保证不要和其它随机写的应用程序共享一块磁盘，尽量避免对磁盘的竞争。

参考文献：

<https://blog.csdn.net/lu1005287365/article/details/52678400>

<http://www.voidcn.com/article/p-krmrvvfh-bnc.html>