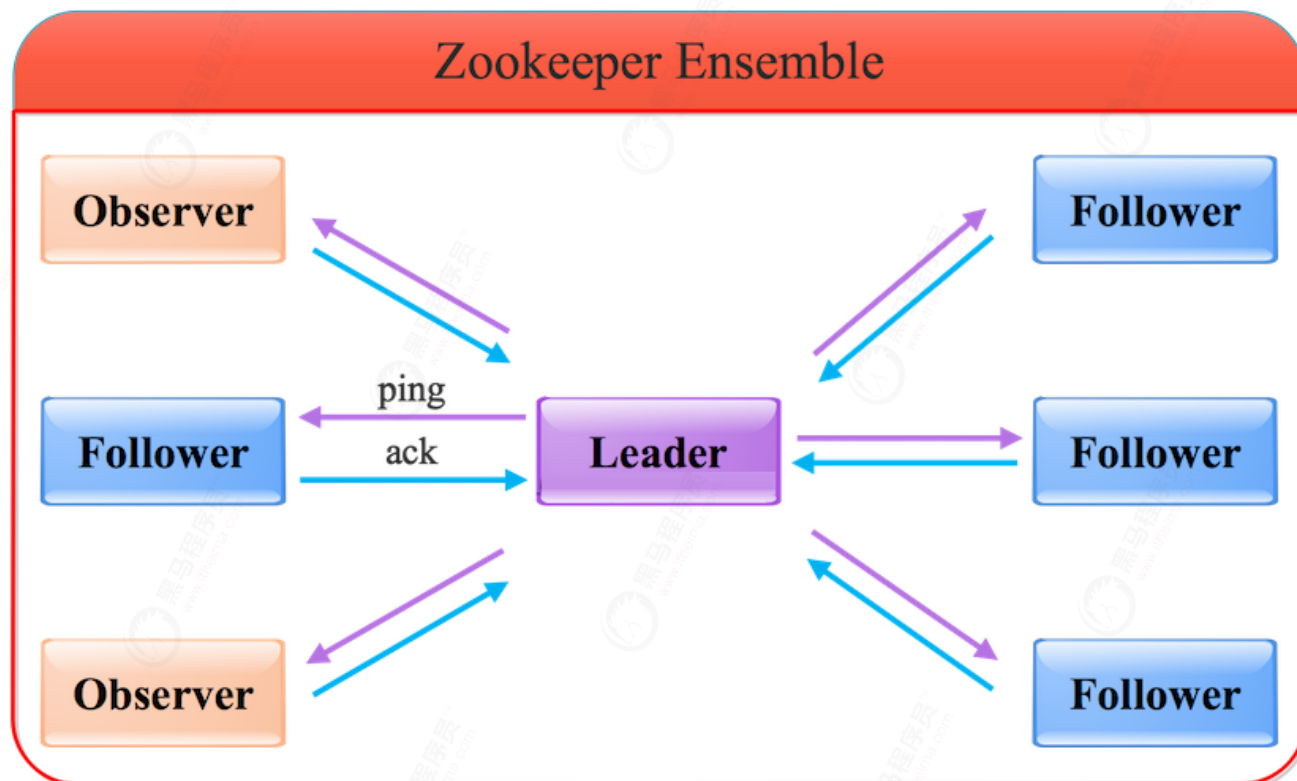


深入RPC原理（下）

1. Zookeeper深入剖析

1.1 ZK特性与集群架构设计

ZK集群架构设计：



ZK主要分为三种角色：

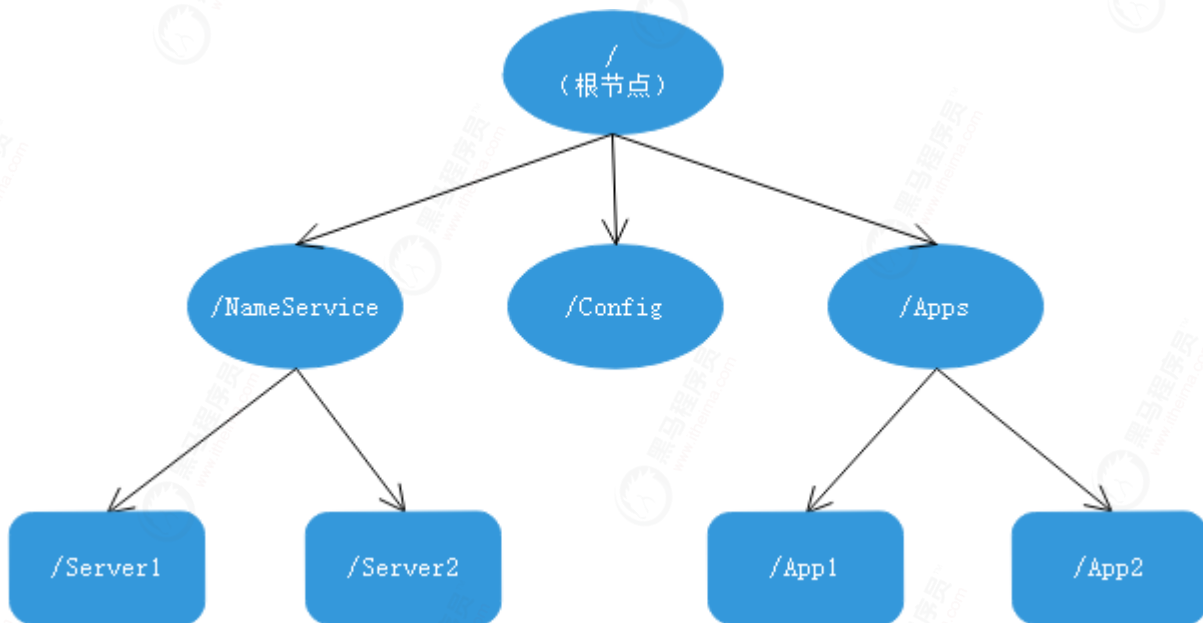
- Leader（领导者）：一个Zookeeper集群同一时间只会有一个实际工作的**Leader**，它会发起并维护与各Follower及Observer间的心跳。所有的写操作必须要通过**Leader**完成再由**Leader**将写操作广播给其它服务器。
- Follower（跟随者）：一个Zookeeper集群可能同时存在多个**Follower**，它会响应Leader的心跳。Follower可以处理客户端的读请求，但写请求转发给**Leader**处理，并且负责参与新**leader**的选举、响应 leader 的提议。
- Observer（观察者）：无投票权，不参加选举，也不响应提议。Observer不需要将事务持久化到磁盘，如果重启需要从 leader 重新同步整个名字空间。

ZK特性：

- 顺序一致性
- 实时性
- 原子性

1.2 ZK数据结构与存储

1.2.1 ZK数据结构模型



节点存储容量不能超过1M。

ZNode节点类型：

Znode的分为四类：

持久节点（persistent node）：节点会被持久化处理，执行命令：`create /apps/app1 "order"`。

临时节点（ephemeral node）：客户端断开连接后，ZooKeeper会自动删除临时节点，执行命令：`create -e /apps/app1 "order"`。

顺序节点（sequential node）：每次创建顺序节点时，ZooKeeper都会在路径后面自动添加上10位的数字，从1开始，最大值为2147483647（ $2^{32}-1$ ），每个顺序节点都有一个单独的计数器，并且单调递增的，由leader实例维护，执行命令：`create -s /apps/app1 "order"`。

临时顺序节点（EPHEMERAL_SEQUENTIAL）：基本特性与临时节点一致，创建节点的过程中，zookeeper会在其名字后自动追加一个单调增长的数字后缀，作为新的节点名。

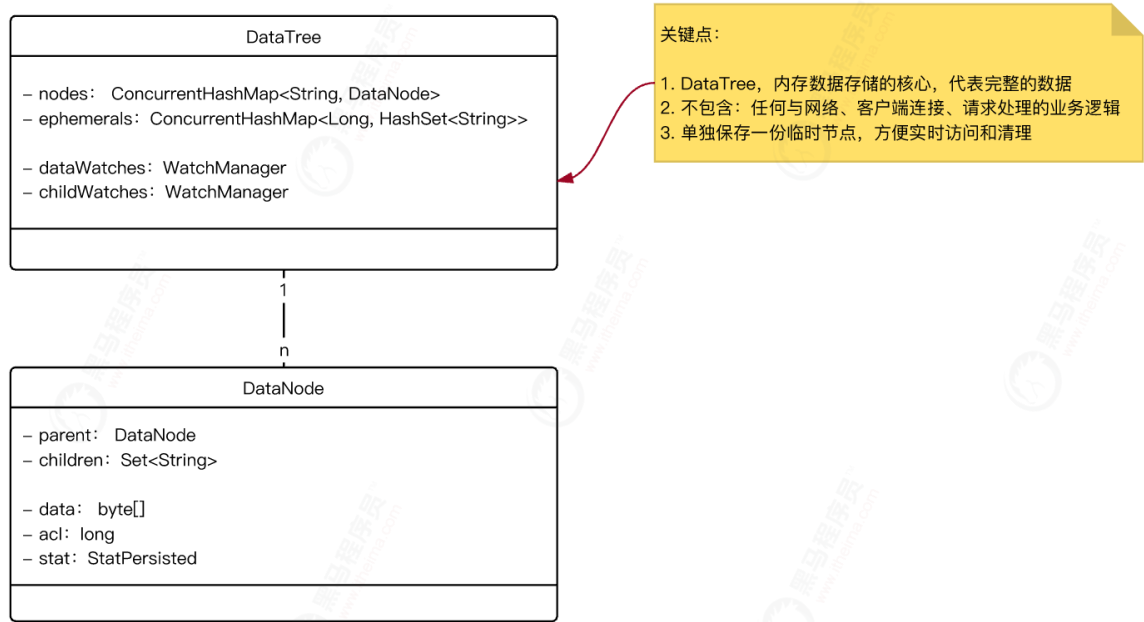
ZNode节点属性：

```
[zk: localhost:2181(CONNECTED) 1] get /testNode
test
cZxid = 0x2
ctime = Fri Aug 06 22:28:23 CST 2020
mZxid = 0x2
mtime = Fri Aug 06 22:28:23 CST 2020
pZxid = 0x2
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 4
numChildren = 0
```

1.2.2 ZK数据存储方式

数据存储方式分为三类：

1. 内存数据

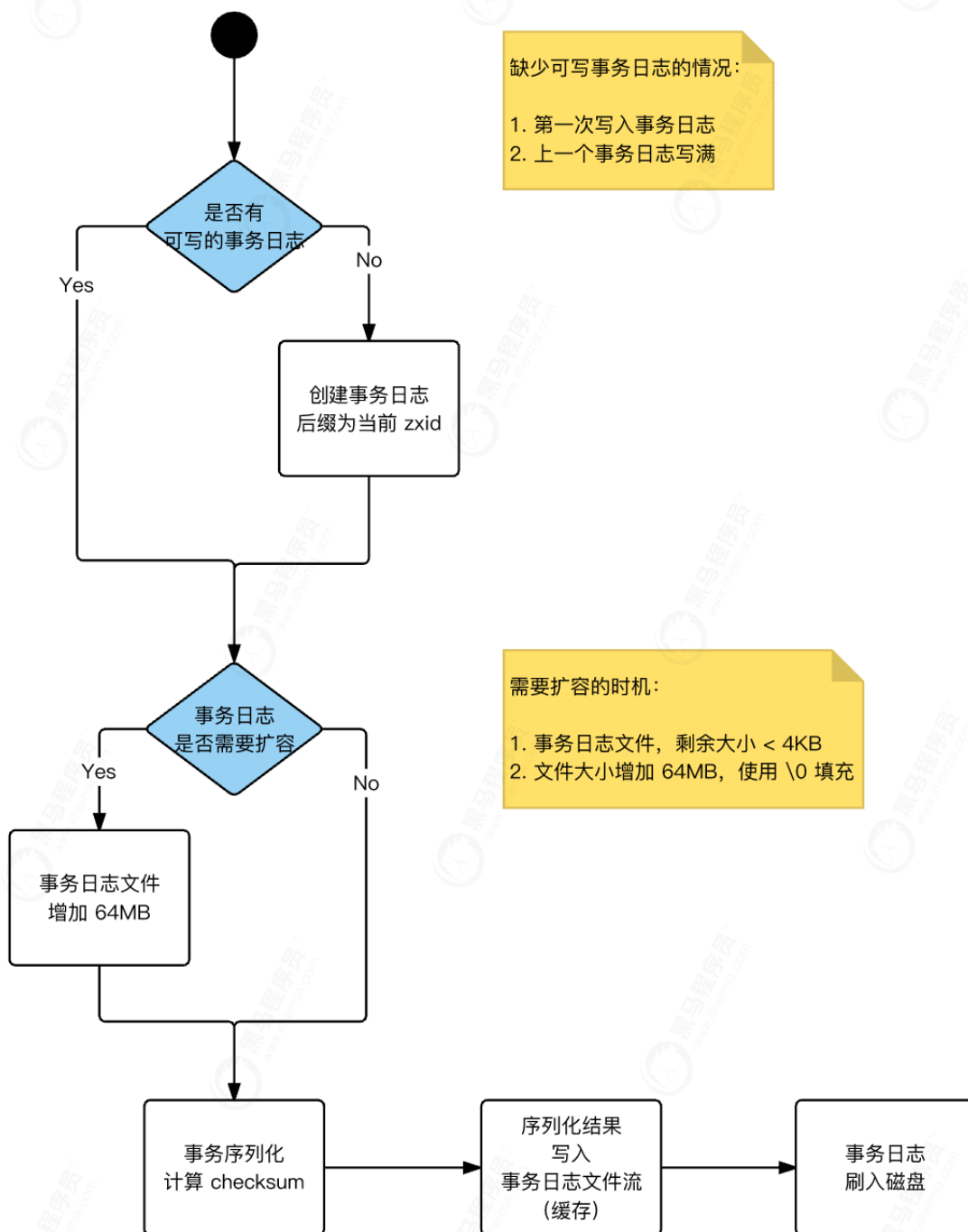


内存数据结构分为三类：

- 1. **DataTree**: 内存数据存储的核心
- 2. **DataNode**: 数据存储的最小单元
- 3. **ZKDatabase**: ZK的内存数据库

2. 事务日志

事务日志处理流程：



事务日志文件内容示例:

查看日志命令:

```
java -classpath ../lib/slf4j-api-1.7.25.jar:./zookeeper-3.4.14.jar  
org.apache.zookeeper.server.LogFormatter /data/zookeeper/version-2/log.100000001 > log1.log
```

产生的日志内容:

```
3/23/20 8:55:19 PM EDT session 0x20003a778cc0012 cxid 0xa9 zxid 0x1000001bc delete '/lock-namespace/shared_lock/order/W-0000000016

3/23/20 8:55:29 PM EDT session 0x20003a778cc0012 cxid 0xb0 zxid 0x1000001bd delete '/lock-namespace/shared_lock/order/W-0000000017

3/23/20 9:46:18 PM EDT session 0x20003a778cc0012 cxid 0xb1 zxid 0x1000001be create '/lock-namespace/shared_lock/order/W-0000000018,#3139322e3136382e3132332e313033,v{s{31,s{'world','anyone'}}},T,19

3/23/20 9:46:38 PM EDT session 0x20003a778cc0012 cxid 0xb4 zxid 0x1000001bf delete '/lock-namespace/shared_lock/order/W-0000000018
```

3. 数据快照 (snapshot)

数据快照用来记录Zookeeper服务器上某一时刻的全量内存数据内容，并将其写入指定的磁盘文件中。

Zookeeper在进行若干次事务日志记录后，将内存数据库的全量数据**Dump**到本地文件中，这个就是数据快照

快照查看命令：

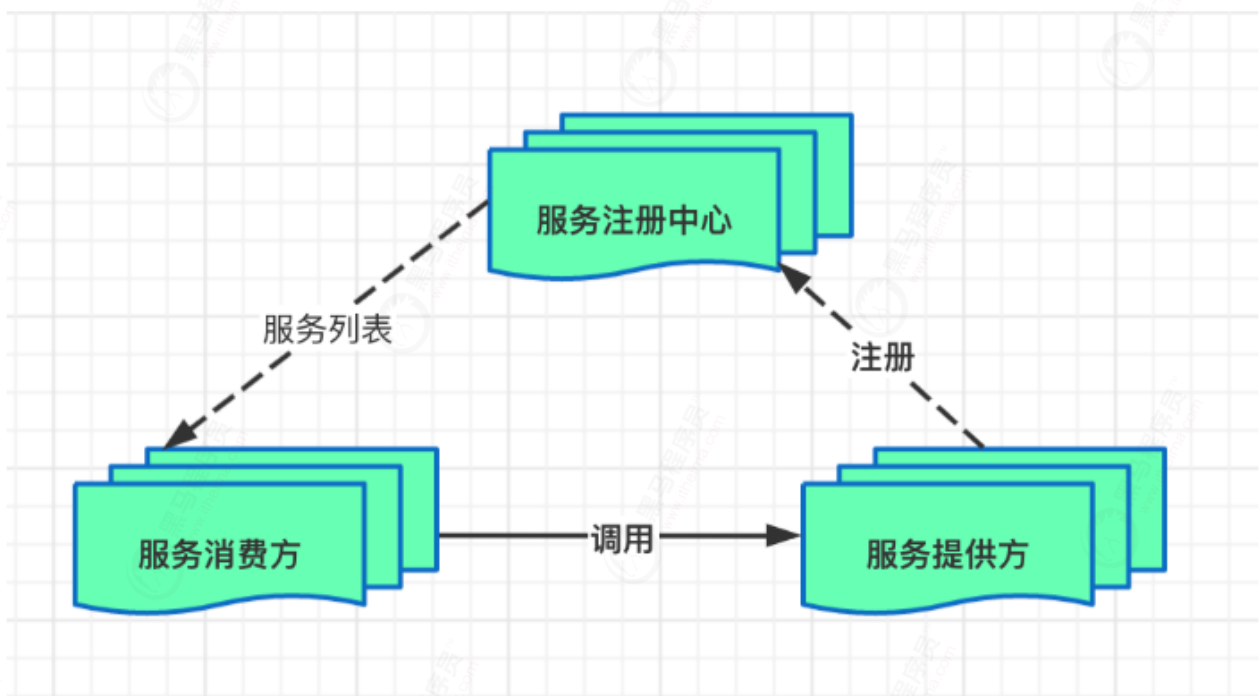
```
java -classpath ../lib/slf4j-api-1.7.25.jar:./zookeeper-3.4.14.jar
org.apache.zookeeper.server.SnapshotFormatter /data/zookeeper/version-2/snapshot.100000000
> snap1.log
```

其处理步骤如下：

- 1) 检查是否需要进行数据快照，每一次事务日志记录之后，Zookeeper都会检测是否需要进行数据快照，考虑到数据快照对于Zookeeper机器的影响，需要尽量避免ZK集群中的所有机器在同一时刻进行数据快照。采用过半随机策略进行数据快照操作。
- 2) 切换事务日志文件，表示当前的事务日志已经写满，需要重新创建一个新的事务日志。
- 3) 创建数据快照的异步线程，创建单独的异步线程来进行数据快照以避免影响Zookeeper主线程的运行状态。
- 4) 获取全量数据和会话信息，从ZKDatabase数据库中获取到**DataTree**和会话信息。
- 5) 生成快照数据文件名，ZK根据当前已经提交的最大ZXID来生成数据快照文件名。
- 6) 数据序列化，首先序列化文件头信息，然后再对会话信息和DataTree分别进行序列化，同时生成一个Checksum校验文件，一并写入快照文件中。

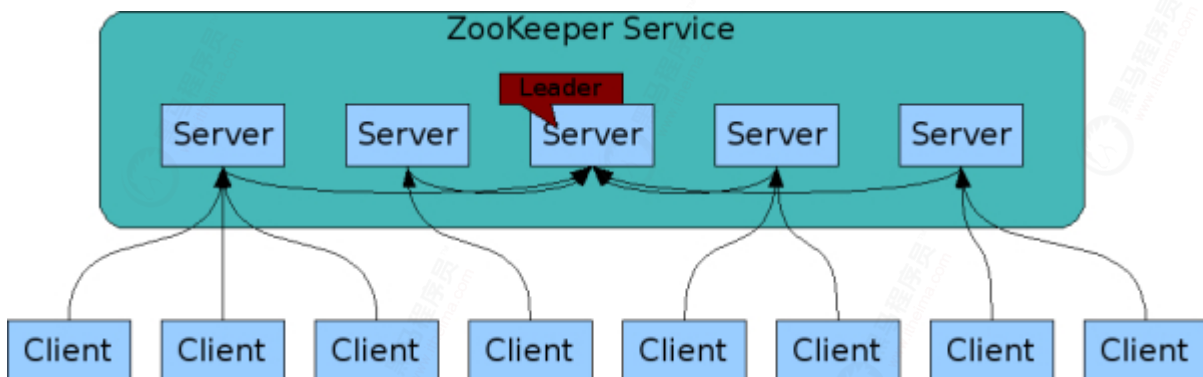
1.3 ZK的应用场景

1.3.1 服务注册发现



分布式服务架构中，服务的注册与发现是最核心的基础服务之一，注册中心可以看做是分布式服务架构的通信中心。

Zookeeper集群，通过监听机制，实现服务信息的订阅：



Zookeeper是采用ZAB协议保证了数据的强一致性。**ZAB**协议的实现原理是怎样？ZK是如何实现选举，**Paxos**算法又是如何运用的？

1.3.2 分布式锁

分布式锁的实现需要注意的：

同步访问共享资源。

- 锁的可重入性：递归调用不会被阻塞、不会发生死锁(**synchronized methodA()**, **ReentrantLock**)
- 锁的超时：避免死锁、死循环等意外情况
- 锁的阻塞：保证原子性等 -> 下单（范围广，包含商品，账户资金，下单信息生成，将粒度细化。）
- 锁的特性支持：阻塞锁、可重入锁、公平锁、联锁、信号量、读写锁

分布式锁的使用需要注意：

- 分布式锁的开销，能不能用就不用
- 加锁的粒度
- 加锁的方式（并发量大，缓存做实现）

基于ZK的分布式锁实现：

整体思想：每个客户端发起加锁时，生成一个唯一的临时有序节点。如何判断锁是否创建呢？只需要判断是否存在临时节点（若存在，则根据序号判断）。

ZK的分布式锁的优缺点

优点：可以有效的解决单点问题，不可重入问题，非阻塞问题以及锁无法释放的问题。

缺点：性能上不如基于缓存实现的分布式锁。ZK的强一致性，一个事务的操作在所有节点都同步完成。

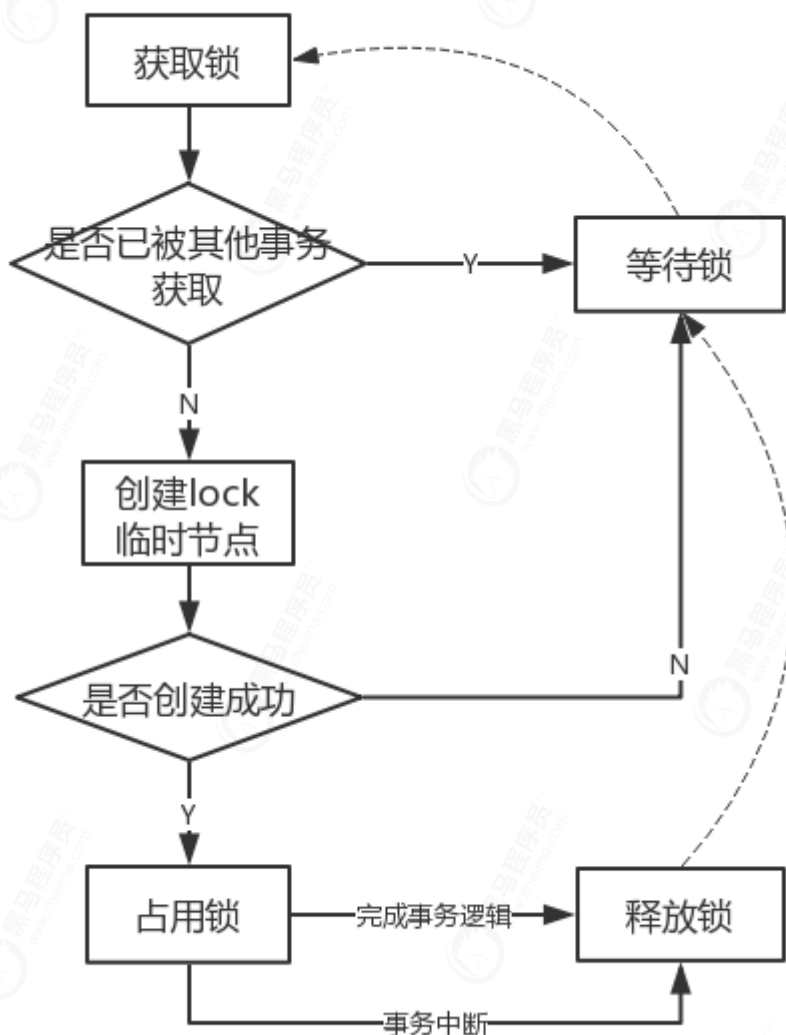
ZK是如何实现分布式锁？

1. 排它锁

定义：如果某个事务(**Transaction**)对数据对象(**Object**)加上了排他锁，那么在整个加锁期间，只允许该事务对数据进行读取或更新操作，其他任何事务不能对该数据对象做任何操作，直到该事务释放了排他锁。

排它锁实现流程：

Zookeeper 的强一致性特性，能够很好地保证在分布式高并发情况下节点的创建一定能够保证全局唯一性，即Zookeeper将会保证客户端无法重复创建一个已经存在的数据节点。可以利用Zookeeper这个特性，实现排他锁。



实现步骤:

- 定义锁: 通过Zookeeper上的数据节点来表示一个锁
- 获取锁: 客户端通过调用 `create` 方法创建锁的临时节点, 创建成功的客户端代表获取锁, 没有获得锁的客户端在该节点上注册Watcher监听, 以便实时获取lock节点的变更情况

- 释放锁

符合以下两种情况都可以让锁释放

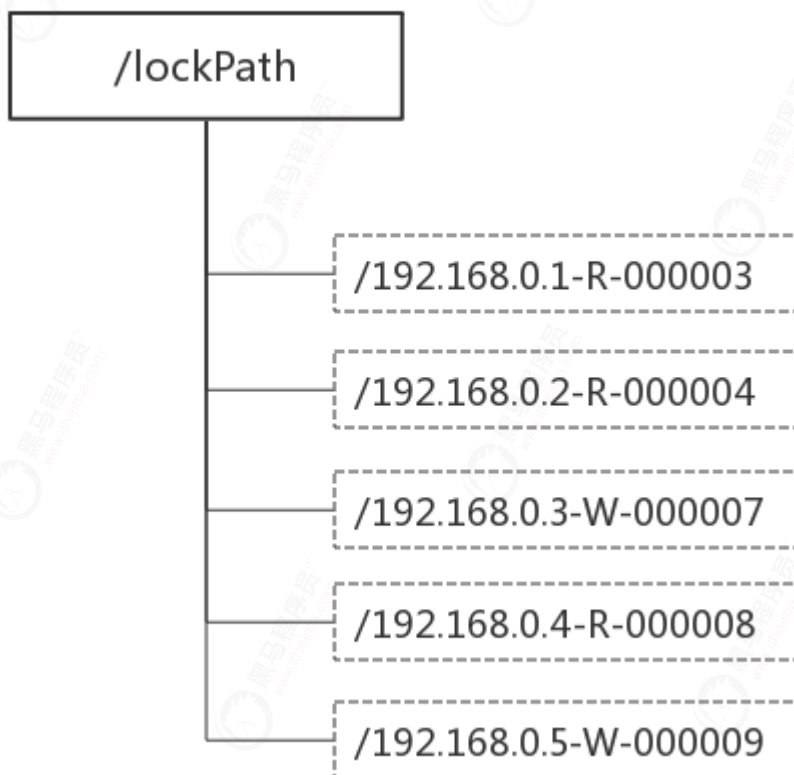
- 当前获得锁的客户端发生宕机或异常, 那么Zookeeper上这个临时节点就会被删除
- 正常执行完业务逻辑, 客户端主动删除自己创建的临时节点

2. 共享锁

定义: 如果事务Transaction对数据对象Object加上了共享锁, 在不是写操作事务下, 其他事务仍可以对Object进行读取操作。

共享锁实现流程:

- 1) 定义锁



2) 获取锁

如果是读请求，则创建 `/lockpath/[hostname]-R-序号` 节点，如果是写请求则创建 `/lockpath/[hostname]-W-序号` 节点。

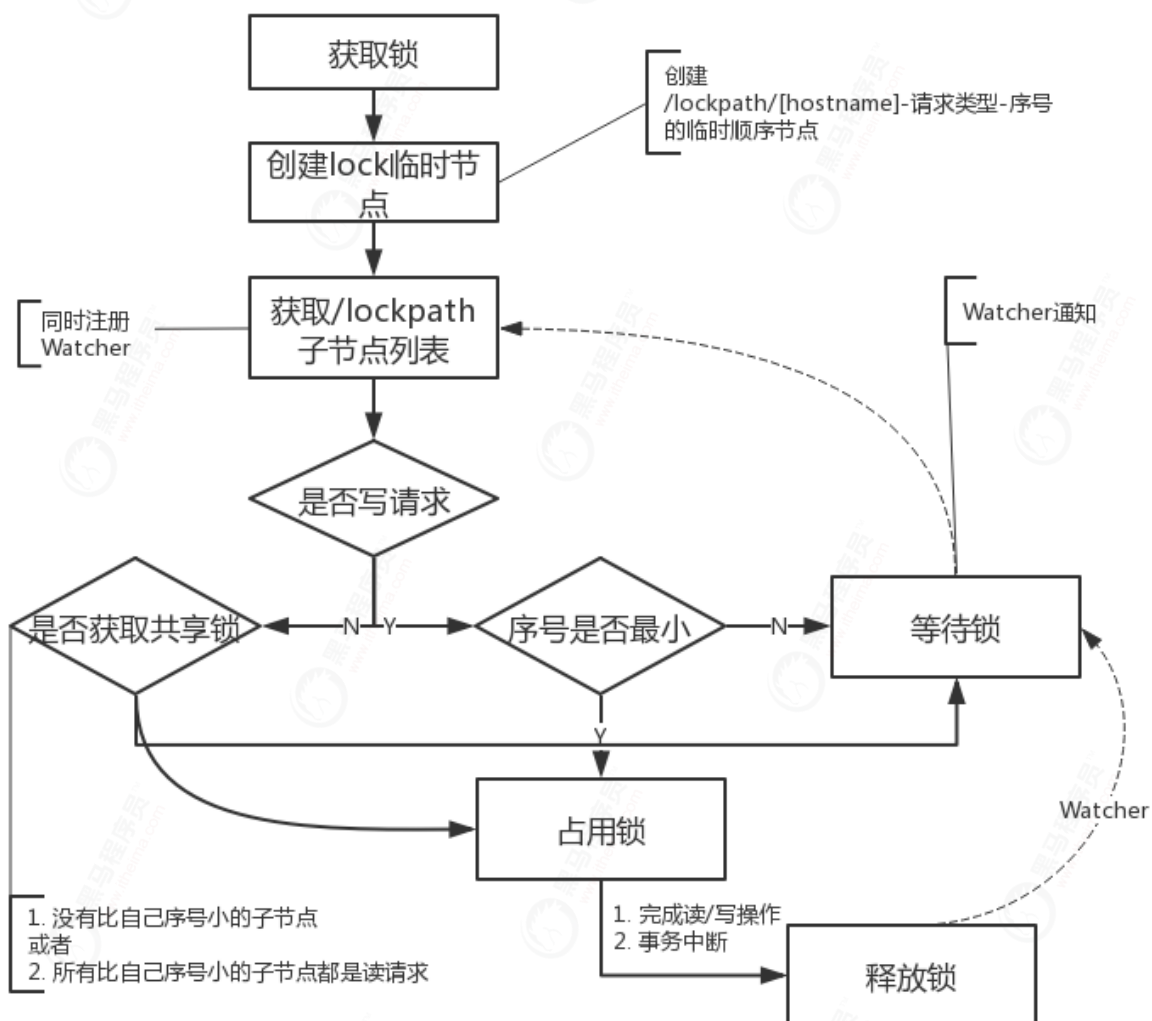
3) 读写顺序判断处理

- 创建完节点后，获取 `/lockpath` 节点下的所有子节点，并对下面所有注册的子节点变更进行 **Watcher** 监听；
- 确定自己的节点序号在所有子节点中的顺序，针对顺序的大小，处理读写请求流程；
- 对于读请求：1. 如果没有比自己序号更小的子节点，或者比自己序号小的子节点都是读请求，那么表明可以成功获取到了共享锁；2. 如果有比自己序号小的子节点有写请求，那么先进行等待。
- 对于写请求，如果存在比自己更小的节点，那么进行等待；
- 接收到 **Watcher** 通知后，再次处理上面的读写请求流程。

4) 释放锁

与排它锁逻辑一致。

基于ZK的共享锁实现流程：



3. 共享锁产生的羊群效应解决方案

羊群效应该如何解决呢？

其实只需要改进Watch的监听处理流程：

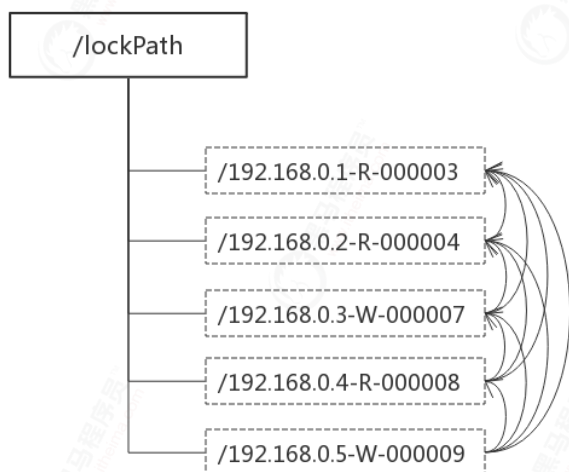


图1. 改进前

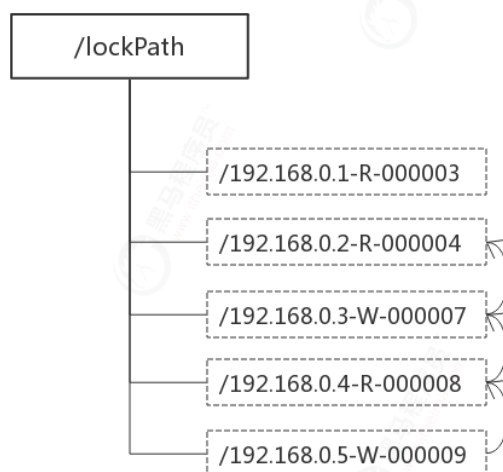


图2. 改进后

1.3.3 利用ZK实现公平选举

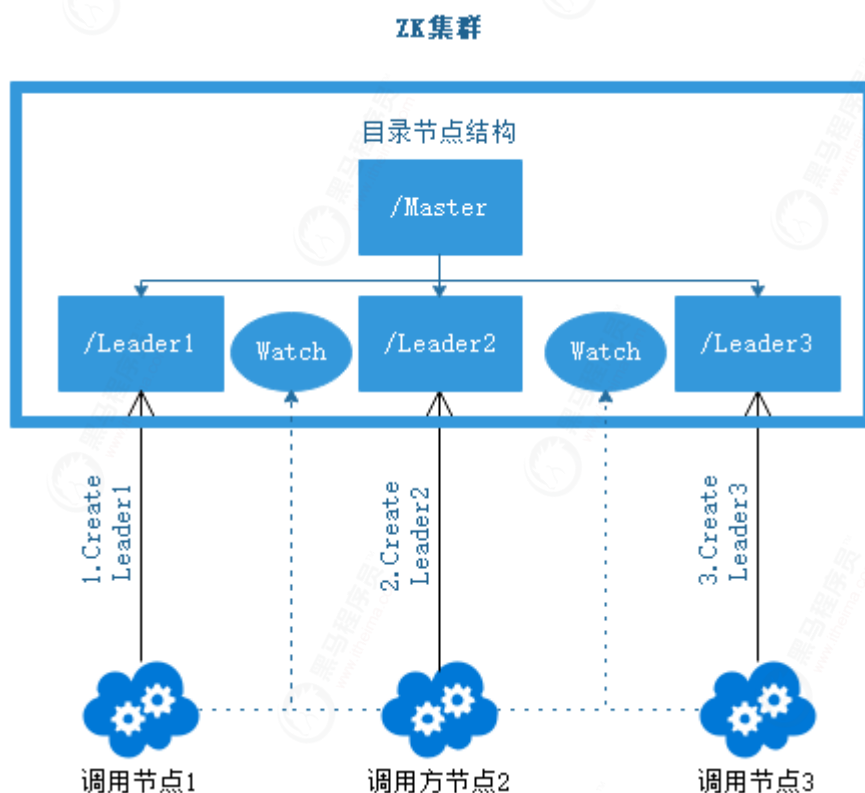
ZK是如何实现集群选举呢？

1. 基于**ZK**的**Watch**机制，ZK的所有节点的读取操作，都可以附带一个Watch，一旦数据有变化，Watch就会被触发，通知客户端数据发生变动。
2. 基于ZK实现的分布式锁，这里的锁是指排它锁，任意时刻，最多只有一个进程可以获取锁。

- 什么是公平选举？

公平选举是要遵循公平性，大家都遵循规则，依照请求先后顺序，参与选举。

- 选举处理流程



三台节点向ZK集群创建**Sequence**类型节点，每个节点所创建的序号不一样，他们会判断自己所创建的节点序号是否为最小，这个与顺序有关，如果是最小，则选取为**Leader**，否则为**Follower**角色。

如果Leader出现问题如何处理？

Leader 所在进程如果意外宕机，其与 ZooKeeper 间的 **Session** 结束，由于其创建的节点为**Ephemeral**类型，故该节点会自动被删除。

Follower角色节点是如何感知的？

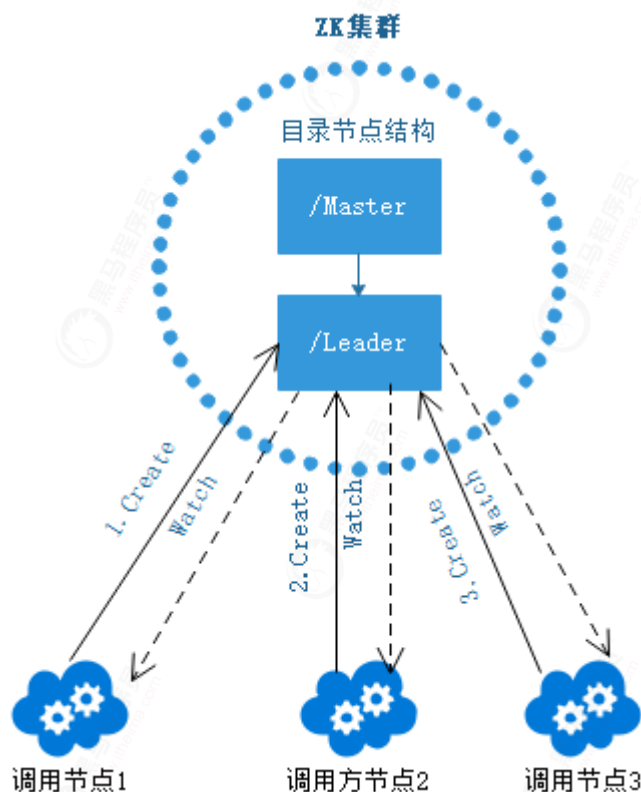
在公平模式下，每个Follower都会 Watch 序号刚好比自己序号小的节点。在上图中，调用方节点2会Watch节点/Master/Leader1，调用方节点3会Watch节点/Master/Leader2。如果Leader宕机，/Master/Leader1删除，调用方节点2就能得到通知。节点2先判断自己的序号 2 是不是当前最小的序号，在该场景下，其序号为最小，所以节点2成为新的Leader。

1.3.4 利用ZK实现非公平选举

- 什么是非公平选举？

非公平选举就是没有遵循选举的公平性，仍然沿用上面的例子：村子里要选举村长，领导通知大家在明早7点前排队在前十的人就可以参与选举，这个时候有人晚到，但借关系插队排在前面，这个就是非公平选举。

- 选举处理流程



三台调用节点向ZK集群创建Non-sequence节点，但只会有一个调用节点创建成功，谁能够抢占资源在ZK集群创建成功，与顺序无关，则竞选为Leader，其他客户端则创建失败，成为Follower角色。

1.4 深入理解Leader选举

1.4.1 PAXOS选举算法

1. Paxos算法概述

背景：

主流分布式一致性算法包括Paxos，Raft和ZAB，它们之间有怎样的区别与关系？

Google Chubby的作者Mike Burrows说过，世上只有一种一致性算法，那就是**Paxos**，所有其他一致性算法都是**Paxos**算法的不完整或衍生版。

什么是**Paxos**？

Paxos算法是基于消息传递且具有高度容错特性的一致性算法，是目前公认的解决分布式一致性问题最有效的算法之一，其解决的问题就是在分布式系统中如何就某个值（决议）达成一致。

Paxos的作用：

常见的分布式系统中，总会发生机器宕机或网络异常（包括消息的延迟、丢失、重复、乱序）等情况。

Paxos 算法是分布式一致性算法用来解决一个分布式系统如何就某个值(决议)达成一致性的问题。

2. 拜占庭问题

拜占庭将军问题是一个共识问题：一群将军想要实现某一个目标，必须是全体一致的决定，一致进攻或者一致撤退；但由于叛徒的存在，将军们不知道应该如何达成一致。

举例来说：

假设有三个拜占庭将军，分别为A、B、C，他们要讨论的只有一件事情：明天是进攻还是撤退。为此，将军们需要依据“少数服从多数”原则投票表决，只要有两个人意见达成一致就可以。

如果A和B投进攻，C投撤退，那么传递结果：

1. 那么A的信使传递给B和C的消息都是进攻；
2. B的信使传递给A和C的消息都是进攻；
3. 而C的信使传给A和B的消息都是撤退。

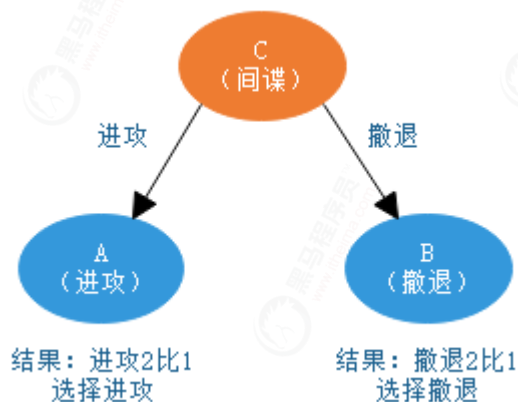
通过以上决策，三个将军就都知道进攻和撤退占比是 **2:1**。显而易见，进攻方胜出，第二天大家都要进攻，三者行动最终达成一致。

但是，如果稍微做一个改动：三个将军中出了一个叛徒呢？叛徒的目的是破坏忠诚将军间一致性的达成，让拜占庭的军队遭受损失。

假设**A和B**是忠诚将军，A投进攻，B投撤退，如果你是**C**这个叛徒将军，那么你该做些什么，才能在第二天让两个忠诚的将军做出相反的决定呢？

进攻方和撤退方现在是 **1:1**，无论C投哪一方，都会变成 **2:1**，一致性还是会达成。但是，作为叛徒的C，你必然不会按照常规出牌，于是你让一个信使告诉**A**的内容是你“要进攻”，让另一个信使告诉**B**的则是你“要撤退”。

至此，A将军看到的投票结果是：进攻方：撤退方 = **2:1**，而B将军看到的是 **1:2**。第二天，忠诚的A冲上了战场，却发现只有自己一支军队发起了进攻，而同样忠诚的B，却早已撤退。最终，A的军队败给了敌人，拜占庭的军队遭受损失。



拜占庭问题情形在计算机世界中也会出现，如果三个节点中有一个异常节点，那么最坏情况下两个正常节点之间是无法保证一致性的。那么你之前听说过的 **etcd** 这样的系统可以保证三个节点有一个宕机的情况下依然可以对外提供一致性服务是为什么呢？因为这类系统并没有考虑拜占庭故障，在他们的假设里故障节点可能会停止服务，可能会超时，但是不会发送异常消息。尽管拜占庭的“幽灵”很难处理，但在实际工作应用中，却并不需要过分去考虑它，因为对于大多数系统来说，内部环境里，硬件故障导致消息错误的概率本身就很低，还要按照拜占庭叛徒的策略来处理故障就更为困难了。

3. Paxos角色

Paxos将系统中的角色分为三种：

Proposer：提议者，提出提案Proposal（未经批准的决议）信息，它包括提案编号 (Proposal ID) 和提议的值 (Value)。

Acceptor：决策者，参与决策，回应**Proposers**的提案。收到Proposal后可以接受提案，若Proposal获得多数Acceptors的接受，则标识该Proposal为批准。

Learner：最终决策学习者，不参与决策，从Proposers/Acceptors学习最新达成一致的决议 (Value)。

在具体的实现中，一个进程可能同时充当多种角色。比如一个进程可能既是Proposer又是Acceptor或Learner。Proposer负责提出提案，Acceptor负责对提案作出裁决（accept与否），Learner负责学习提案结果，Acceptor告诉Learner哪个value被选定，Learner则无条件认为相应的value被选定。

为了避免单点故障，会有一个Acceptor集合群，Proposer向Acceptor集合群发送提案，Acceptor集合群中，只有一半以上的成员同意了这个提案（Proposal），就认为该提案被接受选定了。

4. Paxos算法详解

Paxos包含三种算法：Basic Paxos，Multi Paxos和Fast Paxos。

1. Basic Paxos

Basic Paxos 执行过程分为两个阶段：

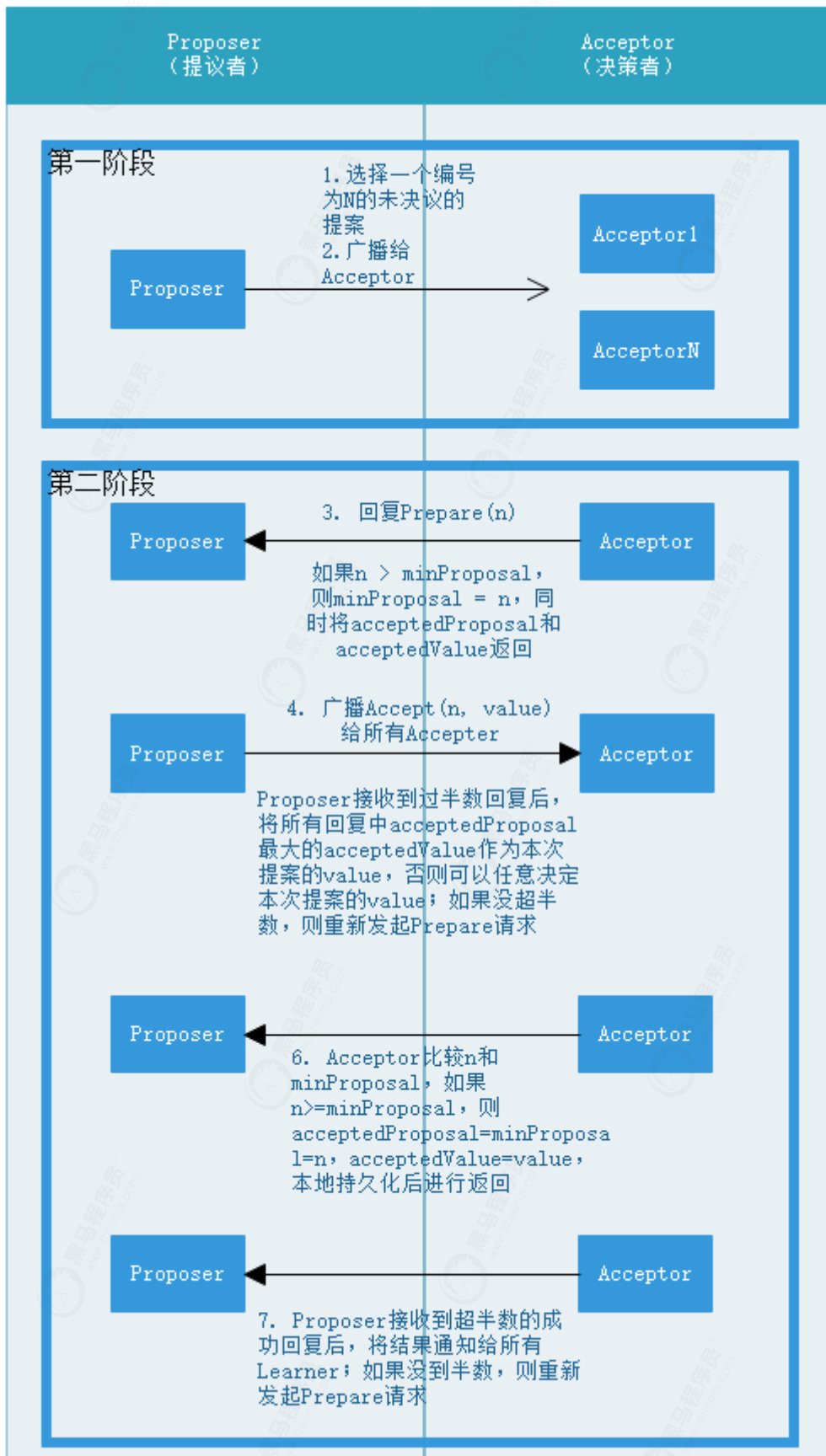
◦ 阶段一（Prepare阶段）：

1. Proposer选择一个提案Proposal编号为N，然后向半数以上的Acceptor发送编号为N的Prepare请求。
2. 如果某个Acceptor收到一个编号为N的Prepare请求，如果小于它已经响应过的请求，则拒绝。
3. 如果N大于该Acceptor已经响应过的所有请求的编号，那么它就会将它已经接受过（已经经过第二阶段accept的提案）的编号最大的提案作为响应反馈给Proposer，如果还没有的accept提案的话返回{p0k, null, null}空信息，同时该**Acceptor**承诺不再接受任何编号小于N的提案。

◦ 阶段二（accept阶段）：

1. 如果一个Proposer收到半数以上Acceptor对其发出的提案响应，那么它就会发送一个针对[N,V]提案的Accept请求给半数以上的Acceptor。注意：N是提案的编号，V就是该提案的决议，该决议是响应编号中最大提案的value。如果响应中不包含任何提案，那么V值就由Proposer自己决定。
2. 如果Acceptor收到一个针对编号为N的提案的Accept请求，只要该Acceptor没有对编号大于N的Prepare请求做出过响应，那么它就接受该提案。如果N小于Acceptor以及响应的prepare请求，则拒绝，不回应或回复error（当proposer没有收到过半的回应，那么它会重新进入第一阶段，递增提案号，重新提出prepare请求）。

整体流程：



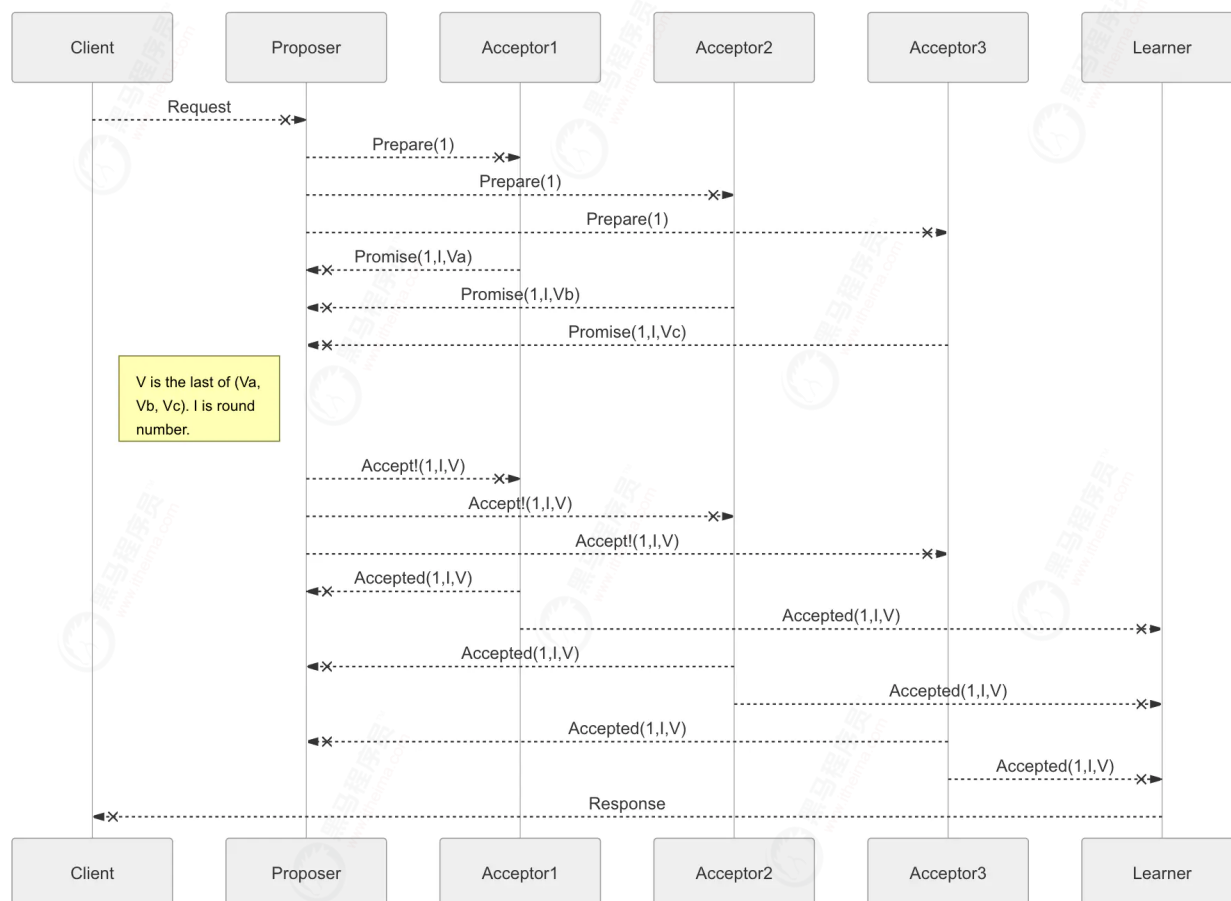
上述介绍了Basic Paxos算法, 但在算法运行过程中, 可能还会存在一种极端情况, 当有两个proposer依次提出一系列编号递增的议案, 那么会陷入死循环, 无法完成第二阶段, 也就是无法选定一个提案。由此产生了活锁问题, 如何解决? 再看下面的Multi Paxos算法。

2. Multi Paxos

原始的基础Paxos算法只能对一个值进行决议，而且每次决议至少需要两次网络来回，在实际应用中可能会产生各种各样的问题，所以不适用在实际工程中。因此，Multi Paxos基于Basix Paxos做了改进，可以连续确定多个值并提高效率：

1. 针对每一个要提议的值，运行一次Paxos算法实例（Instance），形成决议。每一个Paxos实例使用唯一的Instance ID标识。
2. 在所有Proposers中选主，选举一个Leader，由Leader唯一提交Proposal给Acceptors进行表决。这样没有Proposer竞争，解决了活锁问题。在系统中仅有一个Leader进行Value提交的情况下，Prepare阶段就可以跳过，从而将两阶段变为一阶段，提高效率。

示例：



在Basic Paxos协议中，每一次执行过程都需要经历Prepare->Promise->Accept->Accepted 这四个步骤，这样就会导致消息太多，影响性能。

在Multi Paxos的实际过程中：如果Leader足够稳定的话，Phase 1 里面的Prepare->Promise 完全可以省略掉，从而使用同一个Leader去发送Accept消息。

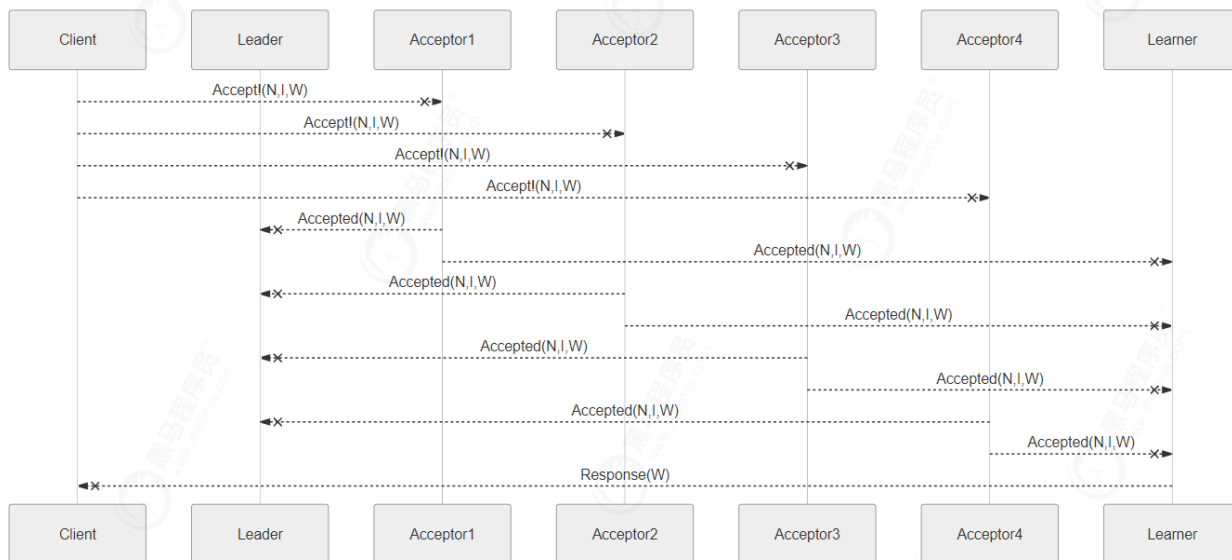
实质上，Multi Paxos模式下首先对所有Proposers进行Leader选举，再利用Basic Paxos来实现。

选出Leader后，只由Leader来提交Proposal，如果Leader出现宕机，则重新选举Leader，在系统中仅有一个Leader可以提交Proposal。Multi Paxos通过改变Prepare阶段的作用范围，从而使得Leader的连续提交只需要执行一次Prepare阶段，后续只需要执行Accept阶段，将两阶段变为一阶段，提高了效率。

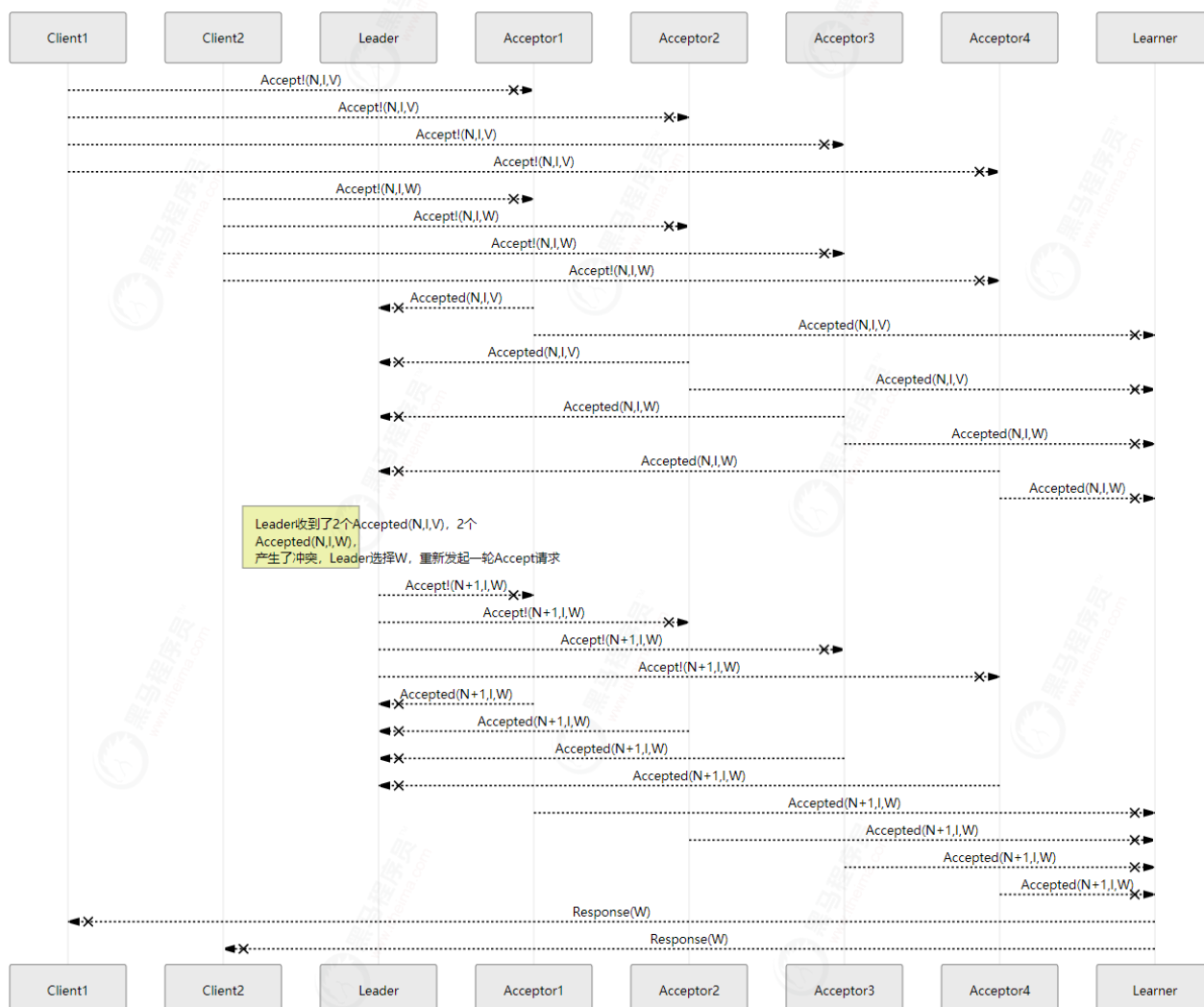
3. Fast Paxos

上述Paxos协议中，消息最后到达Learner一般都要经历 Client-->Proposer-->Acceptor-->Learner 多个步骤，实质上由Learner是真正执行任务，为了更快的让消息到达Learner，如果Proposer本身没有数据需要被确认的话，那么可以跳过Proposer这一步，直接将请求发送给Acceptor，由leader先进行检查，这样的操作叫做Fast Paxos。

Fast Paxos, non-conflicting:



Fast Paxos, conflicting proposals:



1. **Leader**检测到冲突之后，根据规定的算法从冲突中选择一个数据，重新发送Accept请求；
2. 当检测到冲突的时候，如果**Acceptors**自己就能解决冲突，那么就完全不需要**Leader**再次发送Accept请求了，这样就又减少了一次请求，节省开销。

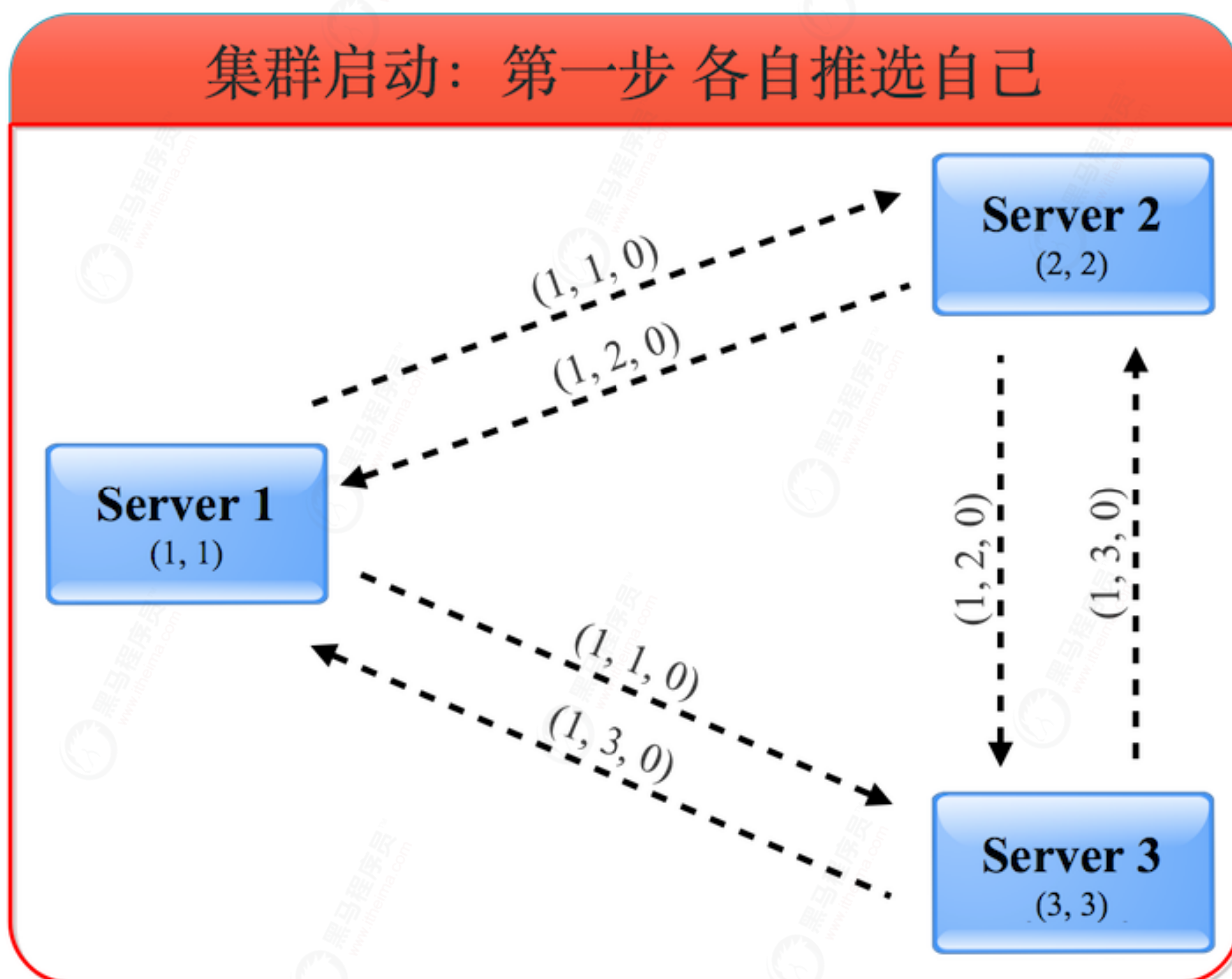
参考资料：

raft算法演示（对Paxos的一种改进。）：

<http://thesecretlivesofdata.com/raft/>

1.4.2 选主过程剖析（基于FastLeaderElection算法）

1. 第一步（初始化投票）



初始化投票，所有服务器的logicClock都为1，zxid都为0，每个节点都会投票给自己，投票信息（1，1，0）主要分为三部分：

第一位数代表服务器的logicalClock（自增的整数），即选举轮次，它表示这是该服务器发起的第多少轮投票；

第二位数代表被推荐的服务器的myid，它是ZooKeeper集群唯一的ID；

第三位数代表被推荐服务器的最大zxid，类似于RDBMS中的事务ID，实质上用于标识一次更新操作的事务Proposal（提议）ID。

ZXID组成：

zxid（Zookeeper Transaction Id）是ZAB协议的事务编号，其是一个64位的整数：

ZXID

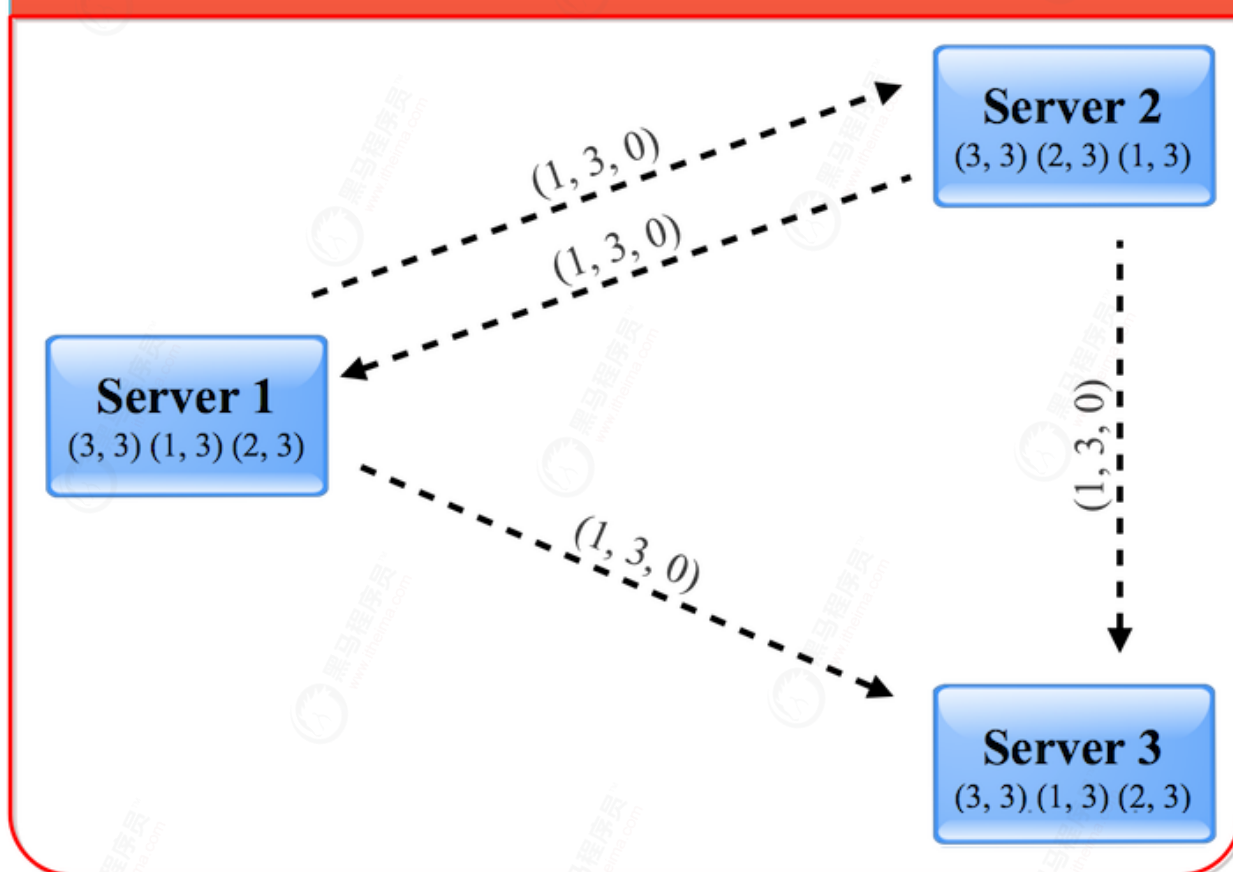


低 32 位是一个单调递增的计数器，每当 Leader 服务器产生一个新的事务 Proposal 时，递增 1。

高 32 位代表 Leader 的 epoch 编号，有点类似于 Raft 的任期(改朝换代)，每当选举一个新 Leader 时，就会从新 Leader 取出本地日志中的最大事务 Proposal 的 zxid，解析出 epoch 然后加 1，并将低 32 位置 0 来开始新的 zxid。

2. 第二步更新投票信息

集群启动：第二步 广播更新选票



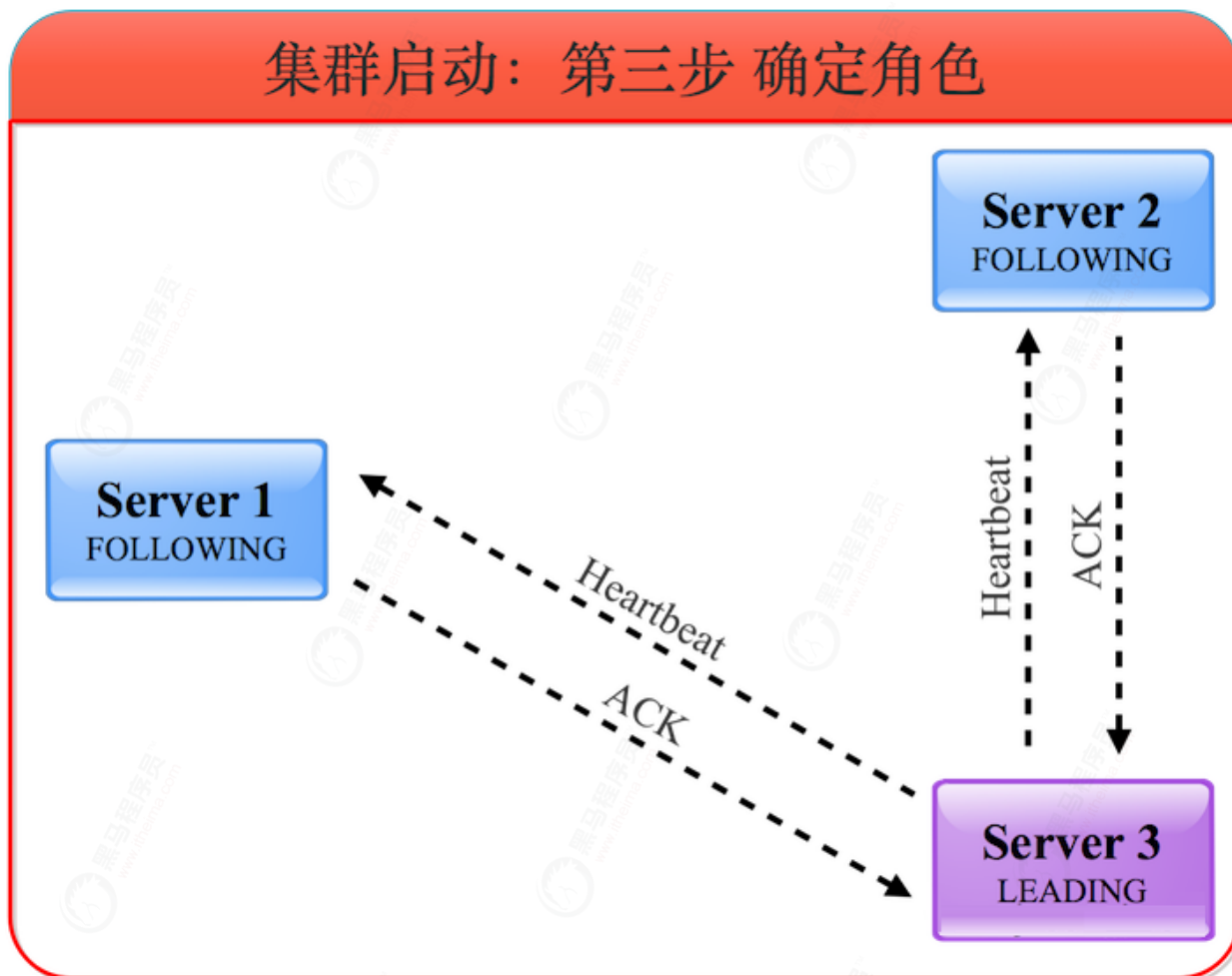
1. 服务器1收到服务器2的选票 (1, 2, 0) 和服务器的选票 (1, 3, 0) 后，由于所有的logicClock都相等，所有的zxid都相等，因此根据myid判断应该将自己的选票按照服务器的选票更新为 (1, 3, 0)。投票规则：

- 优先检查ZXID。ZXID比较大的服务器优先作为Leader。
- 如果ZXID相同，那么就比较myid。myid较大的服务器作为Leader服务器。

2. 服务器1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信息广播出去。此时服务器1票箱内的选票为1号选举3号(1, 3)，3号选举3号(3, 3)。
3. 服务器2收到服务器3的选票后，与服务器1的处理逻辑一样，也将自己的选票更新为 (1, 3) 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)。
4. 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)。

5. 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。

3. 第三步确定集群角色



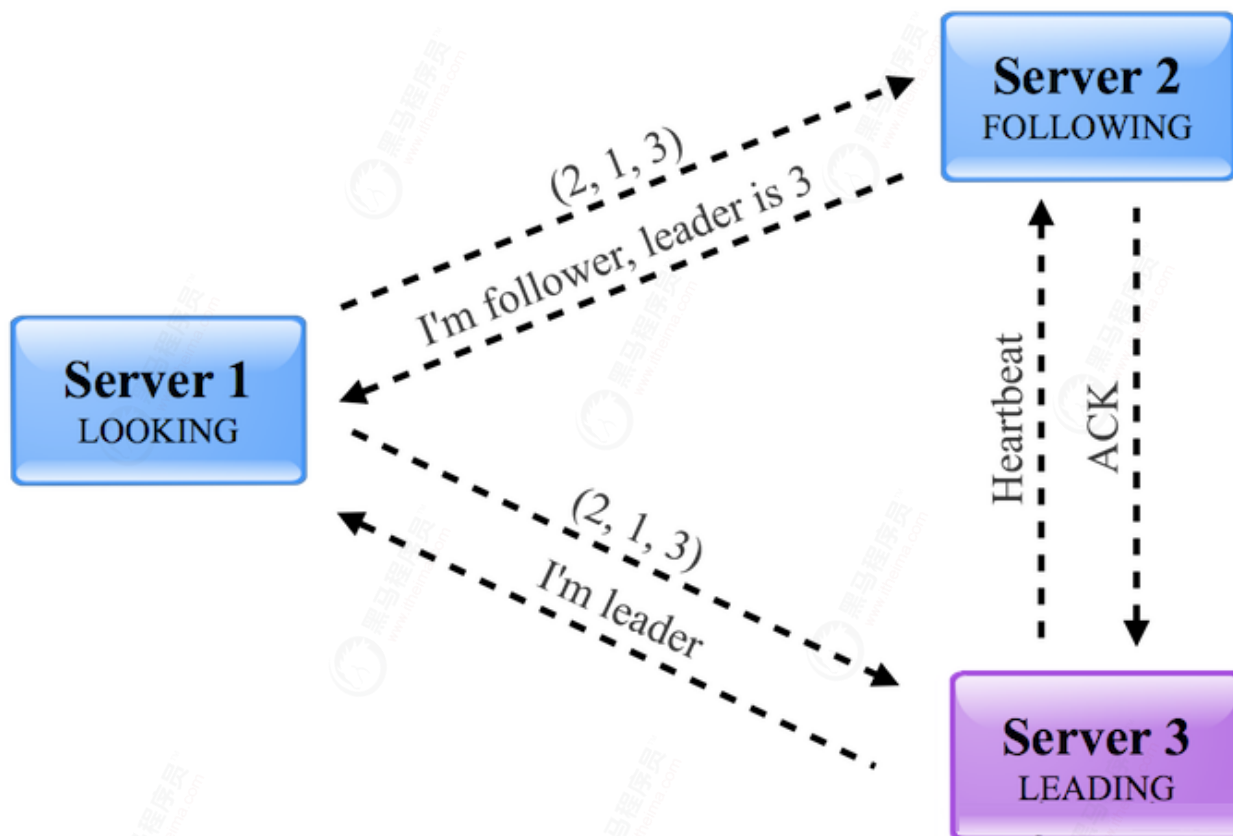
根据上述选票结果，三个服务器一致认为此时服务器3应该是Leader。因此服务器1和2都进入FOLLOWING状态，而服务器3进入LEADING状态。之后由Leader发起并维护与Follower间的心跳。

1.4.3 FOLLOW重启选举剖析

如果FOLLOW跟随者节点出现问题重启之后，是如何实现重新选举？

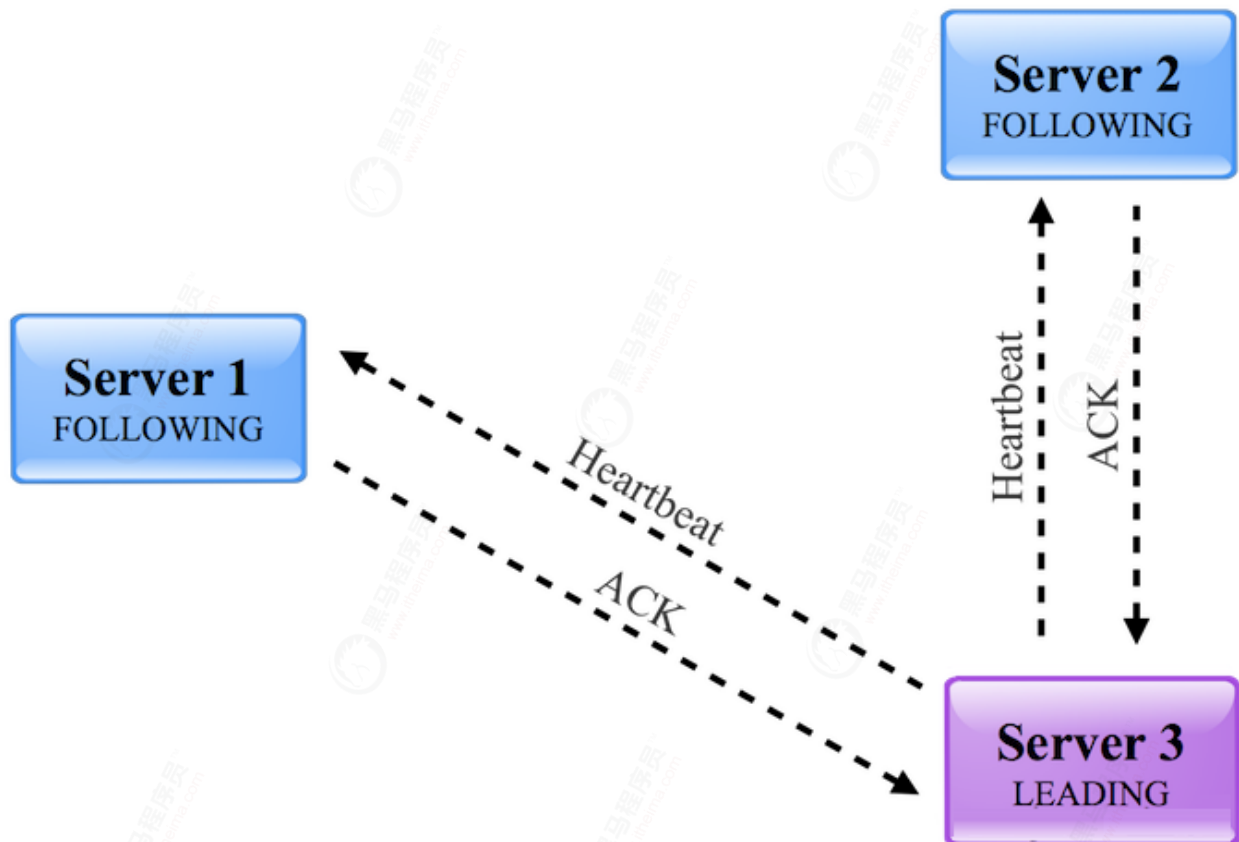
第一步：

Follower重启：投票给自己



第二步：

Follower重启：发现已有Leader，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3号主节点的选票信息返回给服务器1。

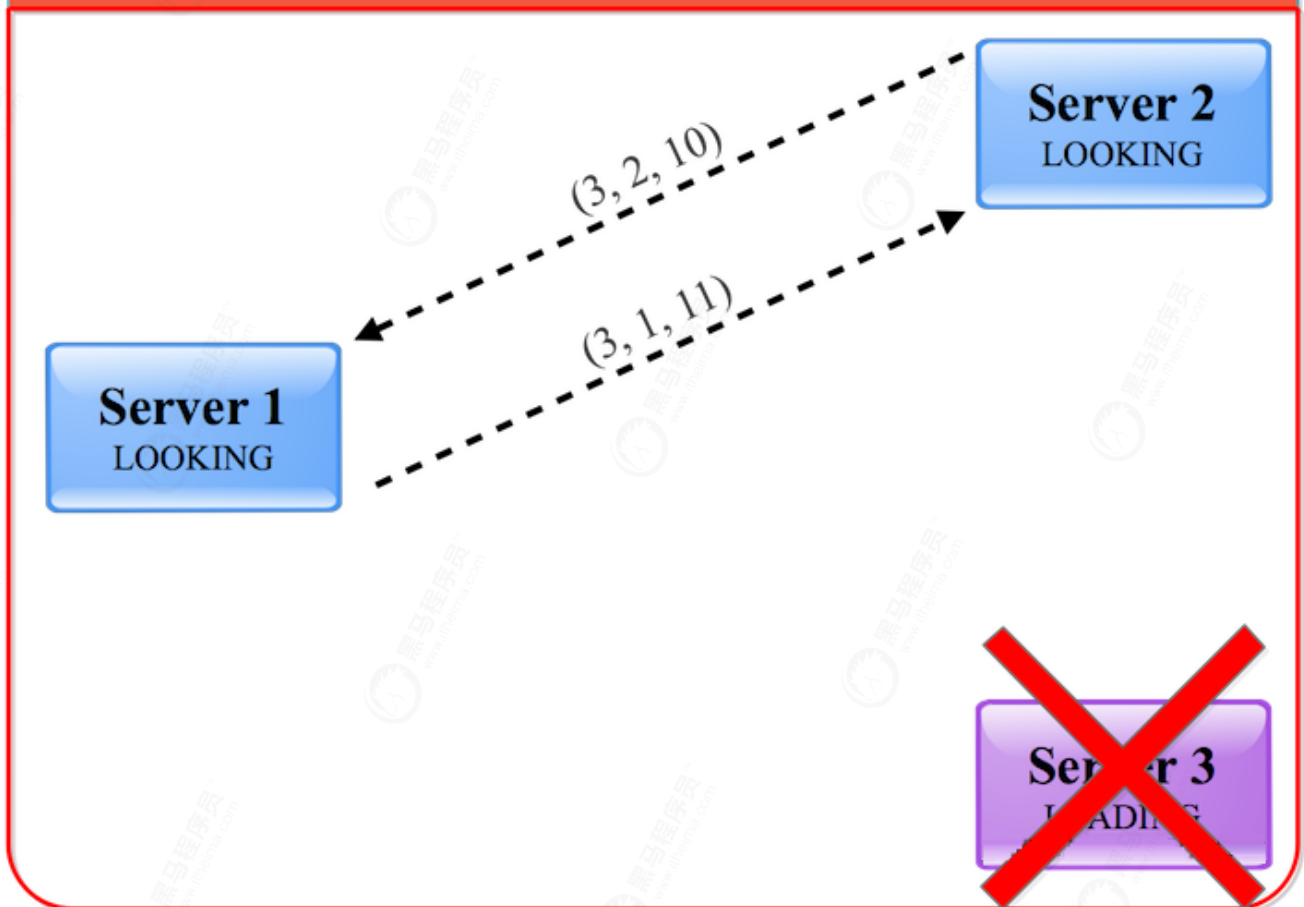
服务器1知道服务器3是Leader，并且通过服务器2与服务器3的选票可以证实服务器3确实得到了超过半数的选票。因此服务器1进入FOLLOWING状态。

1.4.4 LEADER重启选举剖析

Leader如果重启之后该如何选举？

第一步：

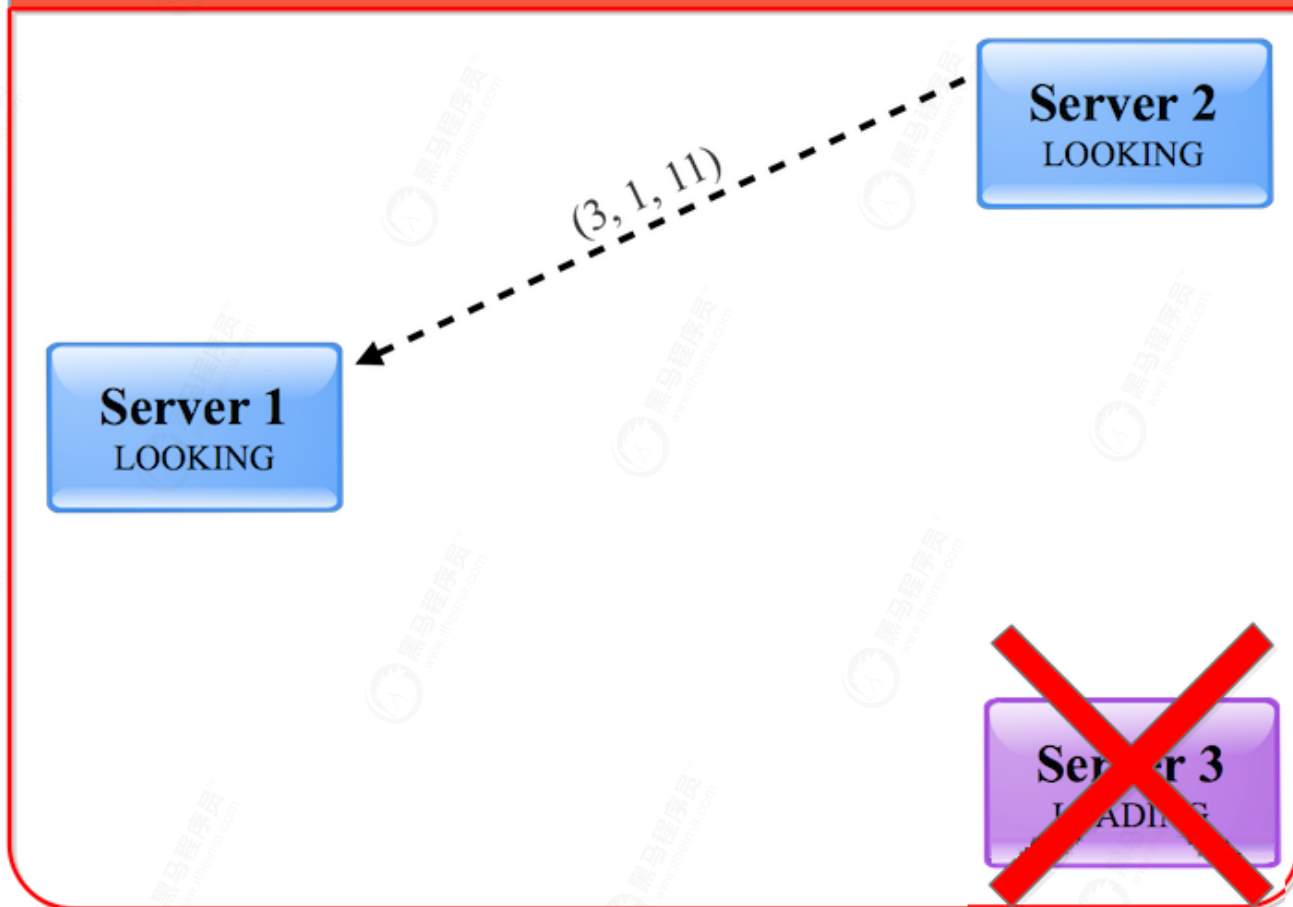
Leader重启：Leader宕机，Follower重新选主



Leader主节点（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状态并发起新一轮投票，并且都将投票选举为自己。

第二步：

Leader重启：广播更新选票



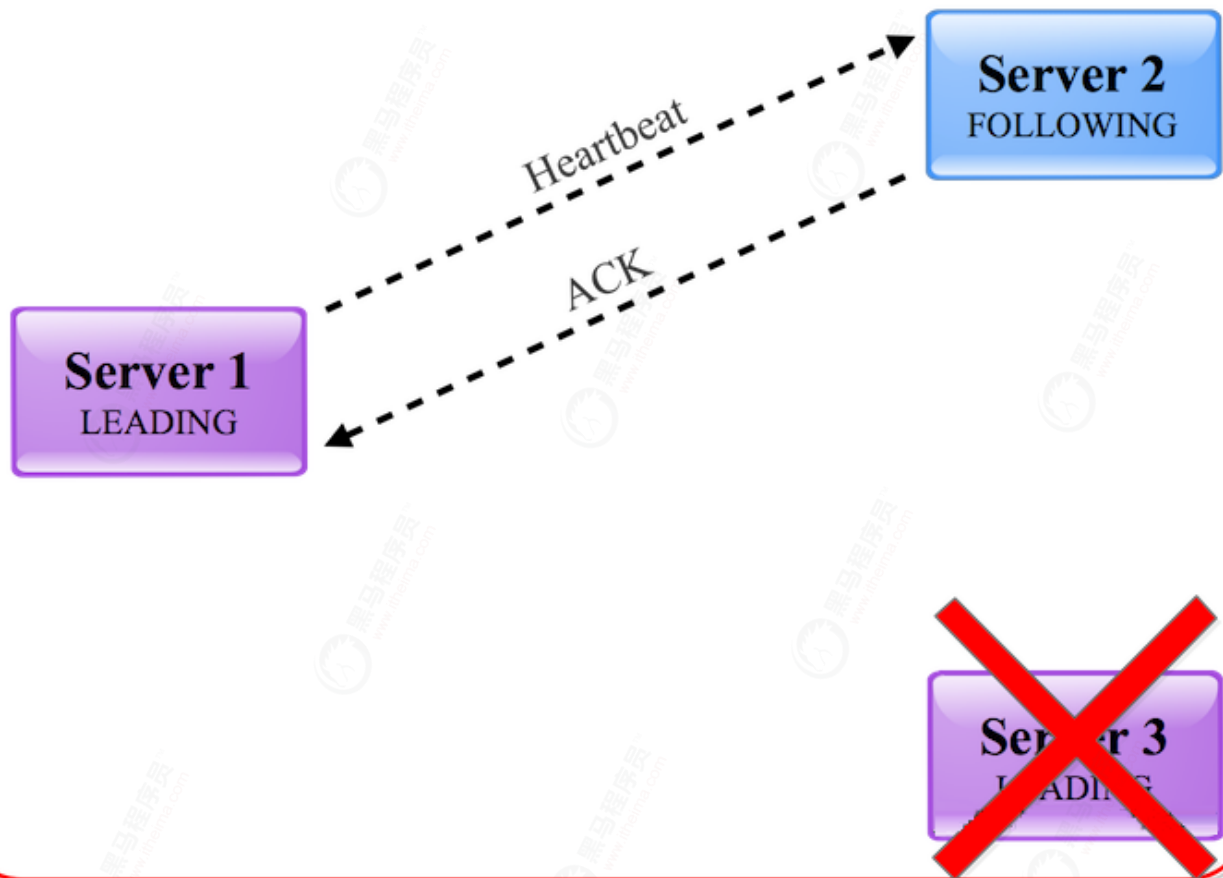
服务器1和2根据外部投票确定是否要更新自身的选票。这里就会出现两种情况：

1. 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完全同步，**zxid**一致。此时选票的更新主要取决于**myid**的大小，根据选举规则，**myid**越大的优先级越高。
2. 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。这个时候，服务器1和2可能其中有一个与旧Leader同步（即zxid与之相同），而另一个则没有同步（即**zxid**比之小）。这时选票的更新主要取决于谁的**zxid**较大，优先选取为主节点。

在第一步的图中所示，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）。

第三步：

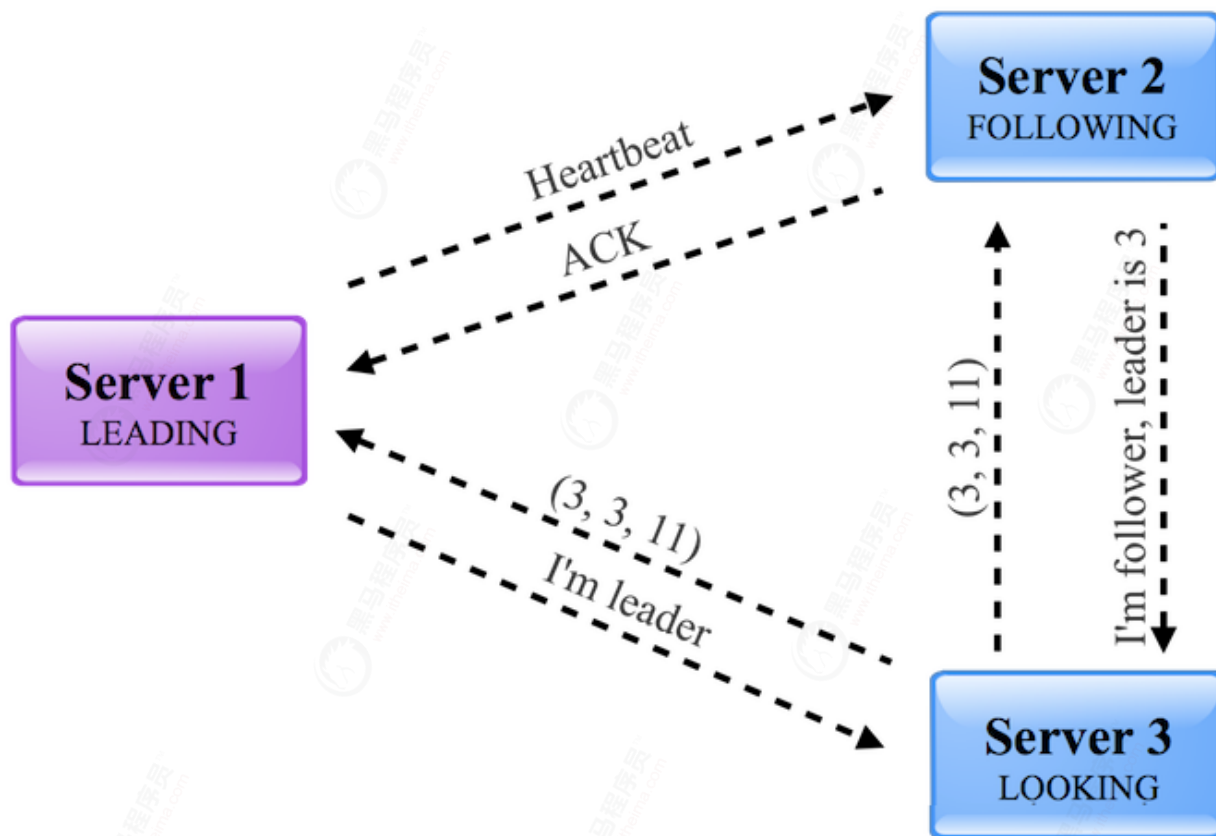
Leader重启：选出新Leader



经过第二步的选票更新后，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。

第四步：

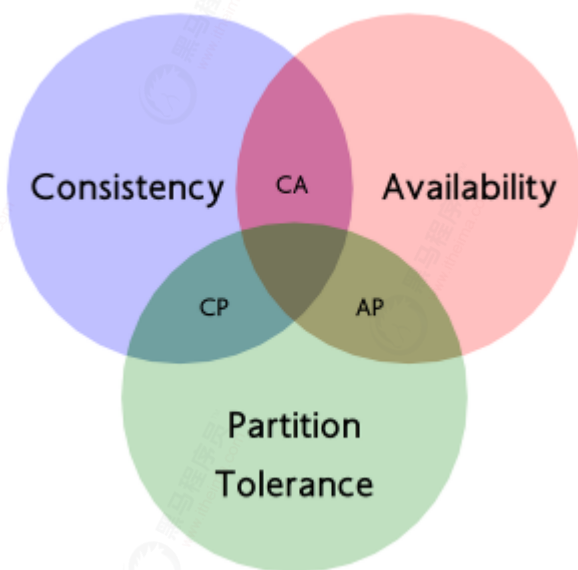
Leader重启：旧Leader恢复后发起选主



旧的Leader节点3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，旧的Leader节点3已经被新的Leader节点1取代，这就类似于上面所讲的FOLLOW重启选举的过程。

1.5 ZK一致性与同步原理

1.5.1 CAP定理



CAP定理，指的是在一个分布式系统中，Consistency（一致性）、Availability（可用性）、Partition tolerance（分区容错性）这三个基本需求，最多只能同时满足其中的2个。

- Consistency（一致性）：多个副本节点保持数据的一致性。
- Availability（可用性）：指系统一直处于可用的状态。
- Partition Tolerance（分区容错性）：遇到任何网络分区故障，仍然能够保证对外提供满足一致性和可用性的服务。

BASE理论？

思考：

- **ZK**为什么不能满足可用性呢？

集群中网络出现分割的故障，ZK将他们从自己的管理范围内踢出去，外界就不能访问这个节点。

- **ZK**在什么情况下是不能保证可用呢？

ZK在进行leader选举时，集群都是不可用的状态。选举leader的期间，持续的时间短的话是数百毫秒，长的话持续数秒。客户端加入重试机制来做补偿。

1.5.2 ZAB协议解析

- **ZAB**协议概述

ZAB（Atomic Broadcast Protocol）协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间数据一致性。

- **ZAB**与PAXOS的联系与区别

Paxos算法的目的在于设计分布式的一致性状态机系统。

ZAB协议的设计目的在于分布式的高可用数据主备系统。

ZAB借鉴了Paxos算法，做了相应的改进，ZAB协议除了遵从Paxos算法中的读阶段和写阶段，还有加入了同步阶段。

- **ZAB**协议两个过程：

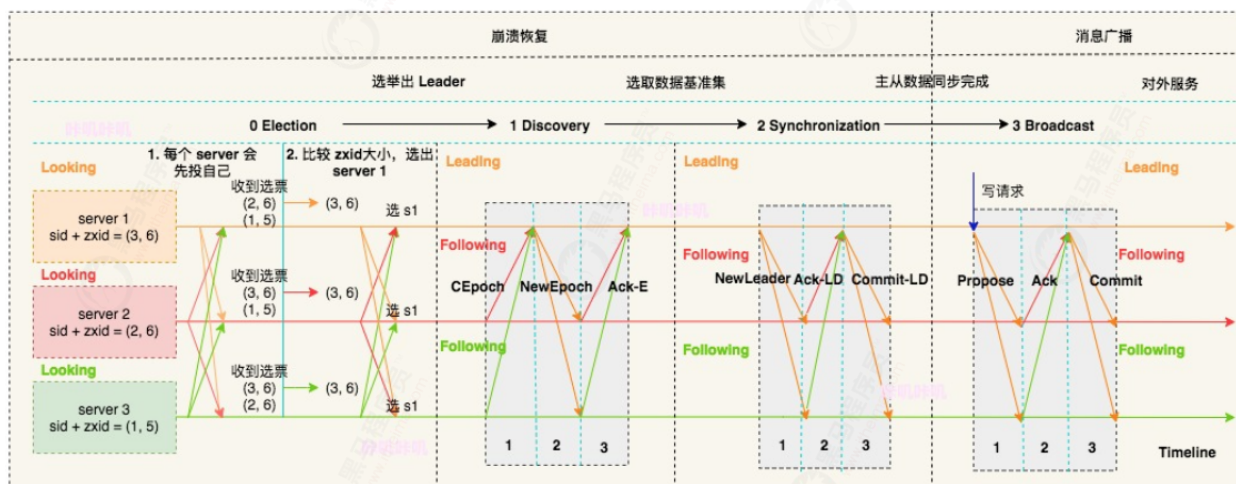
ZAB 协议主要包括两个过程：消息广播和崩溃恢复。和三个阶段：分为**Discovery**发现，**Synchronization**同步，**Broadcast**广播。

消息广播过程

Leader 节点接受事务提交，并且将新的 **Proposal** 请求广播给 **Follower** 节点，收集各个节点的反馈，决定是否进行 Commit。

崩溃恢复过程

如果在同步过程中出现 **Leader** 节点宕机，会进入崩溃恢复阶段，重新进行 **Leader** 选举，崩溃恢复阶段还包含数据同步操作，同步集群中最新的数据，保持集群的数据一致性。



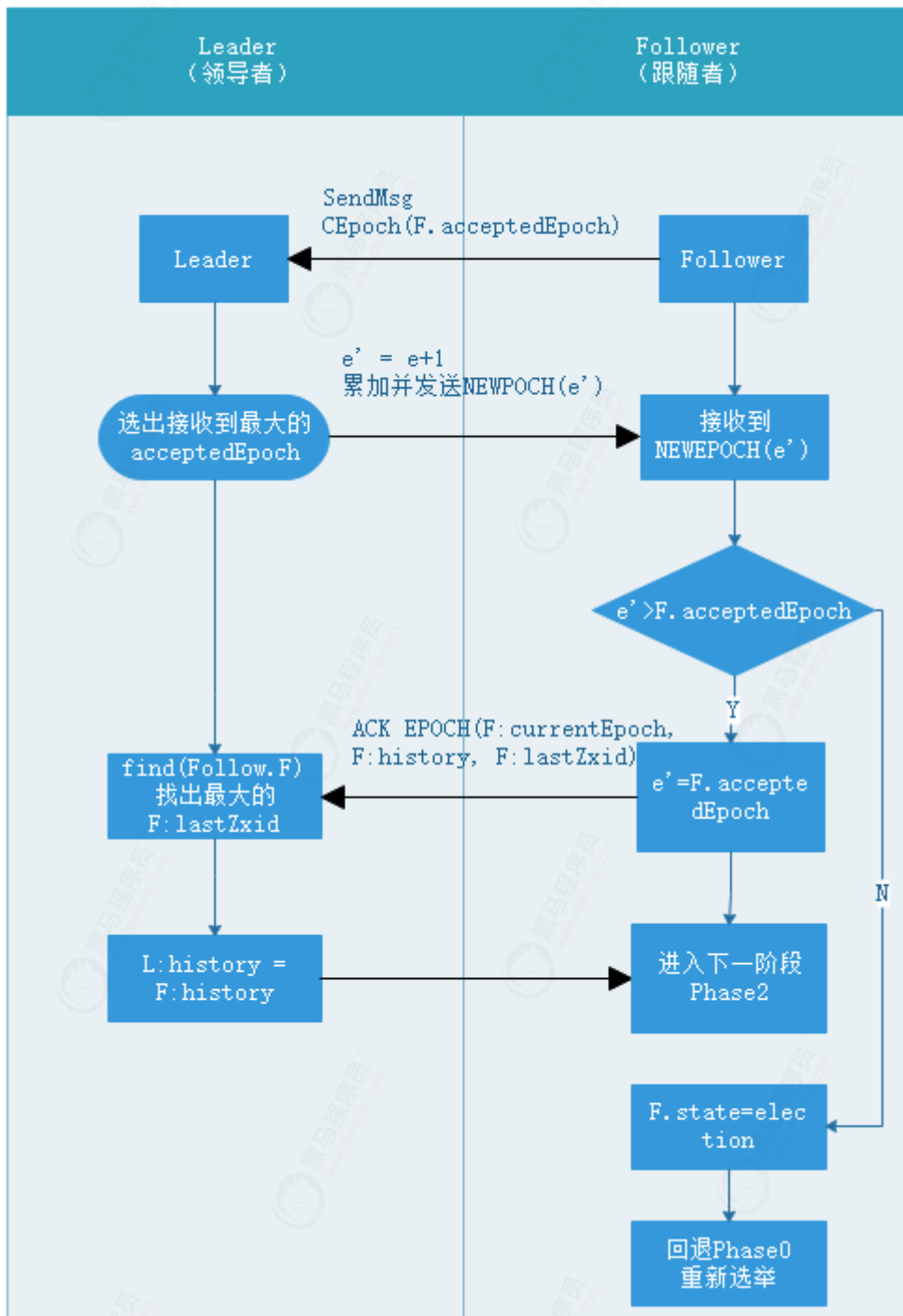
• ZAB协议三个阶段

ZAB存在三个个阶段：发现阶段、同步阶段和广播阶段。其中发现阶段等同于**Paxos**的读阶段，广播阶段等同**Paxos**的写阶段。

涉及专业术语：

- **CEpoch**: Follower 发送自己处理过的最后一个事务 Proposal 的 epoch 值。
- **NewEpoch**: Leader 根据接收 Follower 的 epoch，来生成新一轮 epoch 值。
- **Ack-E**: Follower 确认接收 Leader 的新 epoch。
- **NewLeader**: 确立领导地位，向其他 Follower 发送 NewLeader 消息。
- **Ack-LD**: Follower 确认接收 Leader 的 NewLeader 消息。
- **Commit-LD**: 提交新 Leader 的 proposal。
- **Propose**: Leader 开启一个新的事务。
- **Ack**: Follower 确认接收 Leader 的 Proposal。
- **Commit**: Leader 发送给 Follower，要求所有 Follower 提交事务 Proposal。

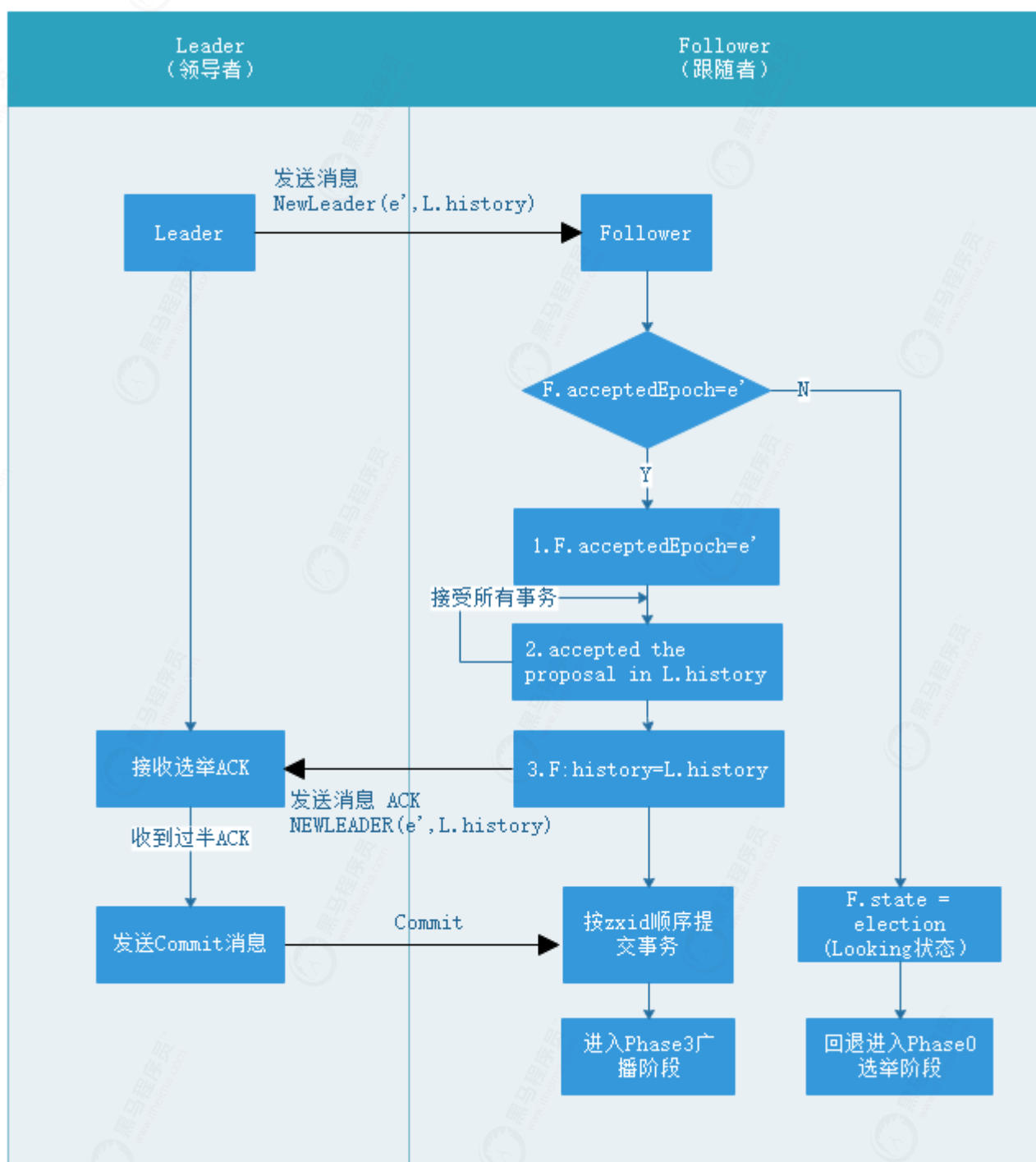
1) Discovery（发现阶段）



处理过程:

1. Follower 将自己最后处理的事务 **Proposal** 的 **epoch** 值发送给 **Leader**, 消息 `CEpoch(F.p)`, `F.p` 可以提取出 `zxid`。
2. 当 Leader 接收到过半的 **Follower** 的 **CEpoch** 消息后, Leader 生成 `NewEpoch(e')` 发送给这些过半的 **Follower**, e' 是比任何从 `CEpoch` 消息中收到的 `epoch` 值都要大。
3. Follower 一旦从 Leader 处收到 `NewEpoch(e')` 消息, 会先做判断, 如果 $e' < F.acceptedEpoch$, 并且 `F.state = election` 也就是 Looking 状态, 那么会重新回到 Leader 选举阶段。
4. Leader 一旦收到了过半的 **Follower** 的确认消息。它会从这些过半的 Follower 中选取一个 **F**, 并使用它作为初始化事务的集合 (用于同步集群数据), 然后结束发现阶段。既然要选择需要同步的事务集合, 必然要选择事务最全的, 所以, 须满足 **epoch** 是最大的且 **zxid** 也是最大的条件。

2) Synchronization (同步阶段)

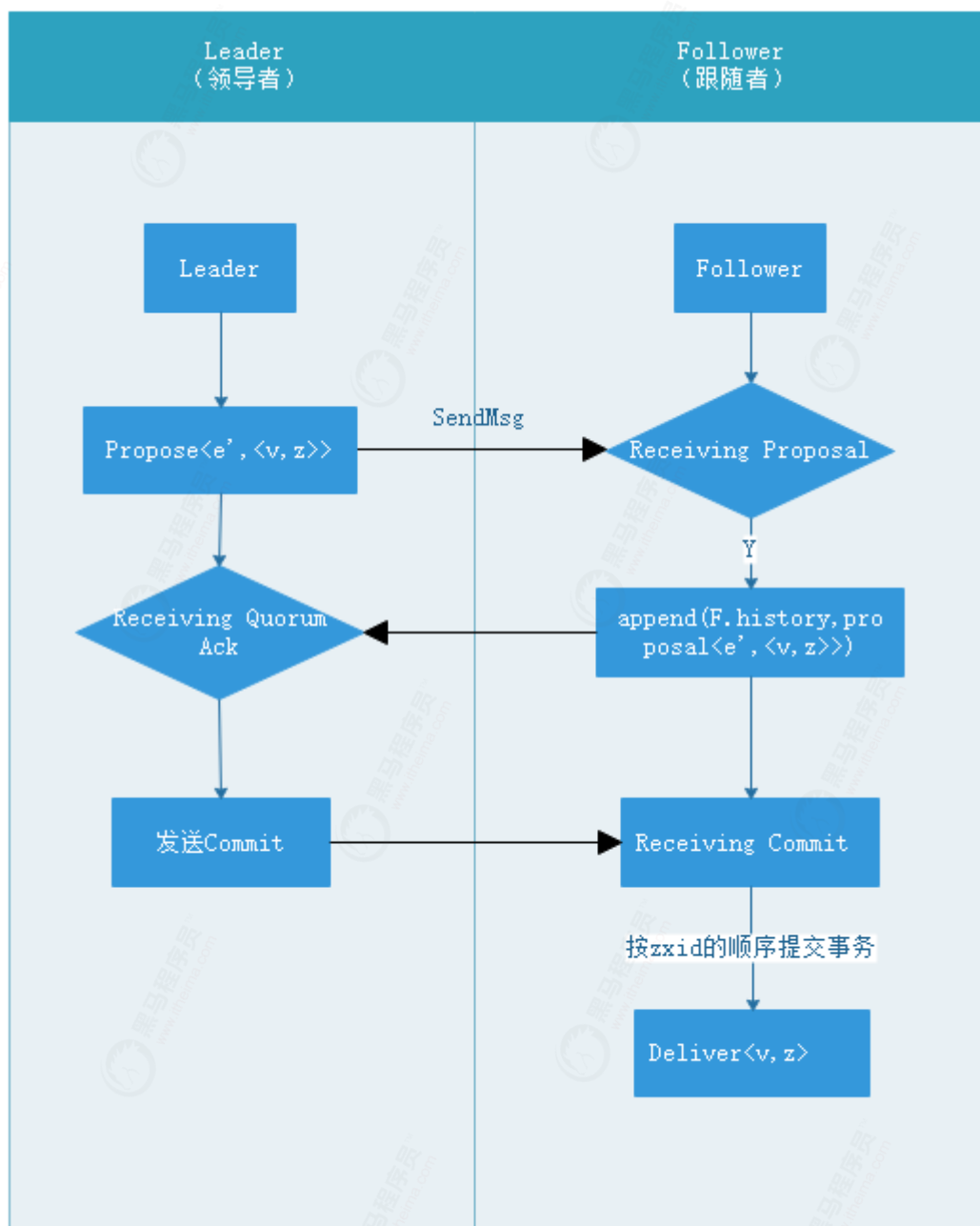


1. 第一个过程 NewLeader: Leader 将新 epoch 和 `S'` 以 `NewLeader(e', S')` 的消息形式发送给所有过半 (Quorum) 的 Follower。在上一阶段 `L.history = F.history`，所以 `S'` 就是流程图中的 `L.history`。
2. 第二过程ACK: 当 Follower 接收到 `NewLeader(e', S')` 消息后，做相应判断：
 1. 如果 Follower 的 epoch 等于 `e'`，也就是确认是不是该Follower的信息，因为前一阶段已经存储了最新的 `e'`。Follower 将会执行事务应用操作，将接收 `S'` 中的所有事务 **Proposal**，只是接收不作其他处理。

2. 如果 Follower 的 epoch 不等于 `e'`，即不是这一轮的 Follower 信息，直接回退至选举阶段。
3. Leader 在接收到过半的 Follower 的 Ack 消息后，发送 Commit 消息至所有的 Follower 进行同步，之后进入下一阶段即 **Broadcast**（消息广播）。

3. 第三个过程 Commit: 在收到 Leader 的 Commit 消息后，按顺序依次调用 `abdeliver()` 处理 `S'` 的每一个事务，随后结束这一阶段的处理。

3) Broadcast（广播阶段）

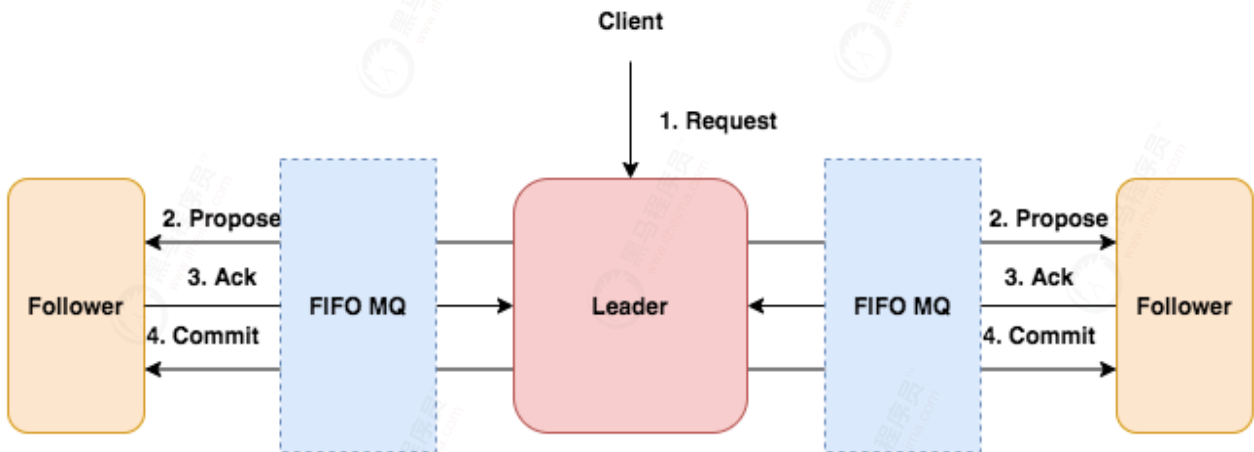


第一个过程 Propose: Leader 收到来自客户端新的事务请求后，会生成对应的事务 Proposal，并根据 zxid 的顺序（递增）向追随自己的所有 Follower 发送 `P<e', <v, z>>`，其中 `epoch(z) == e'`。

第二个过程 Ack: Follower 根据收到消息的次序来处理这些 Proposal，并追加到 H 中去，然后通知给 Leader。

第三个过程 Commit: 一旦 Follower 收到来自 Leader 的 Commit 消息，将调用 `abdeliver()` 提交事务。这里是按照zxid的顺序来提交事务。

广播请求处理流程:

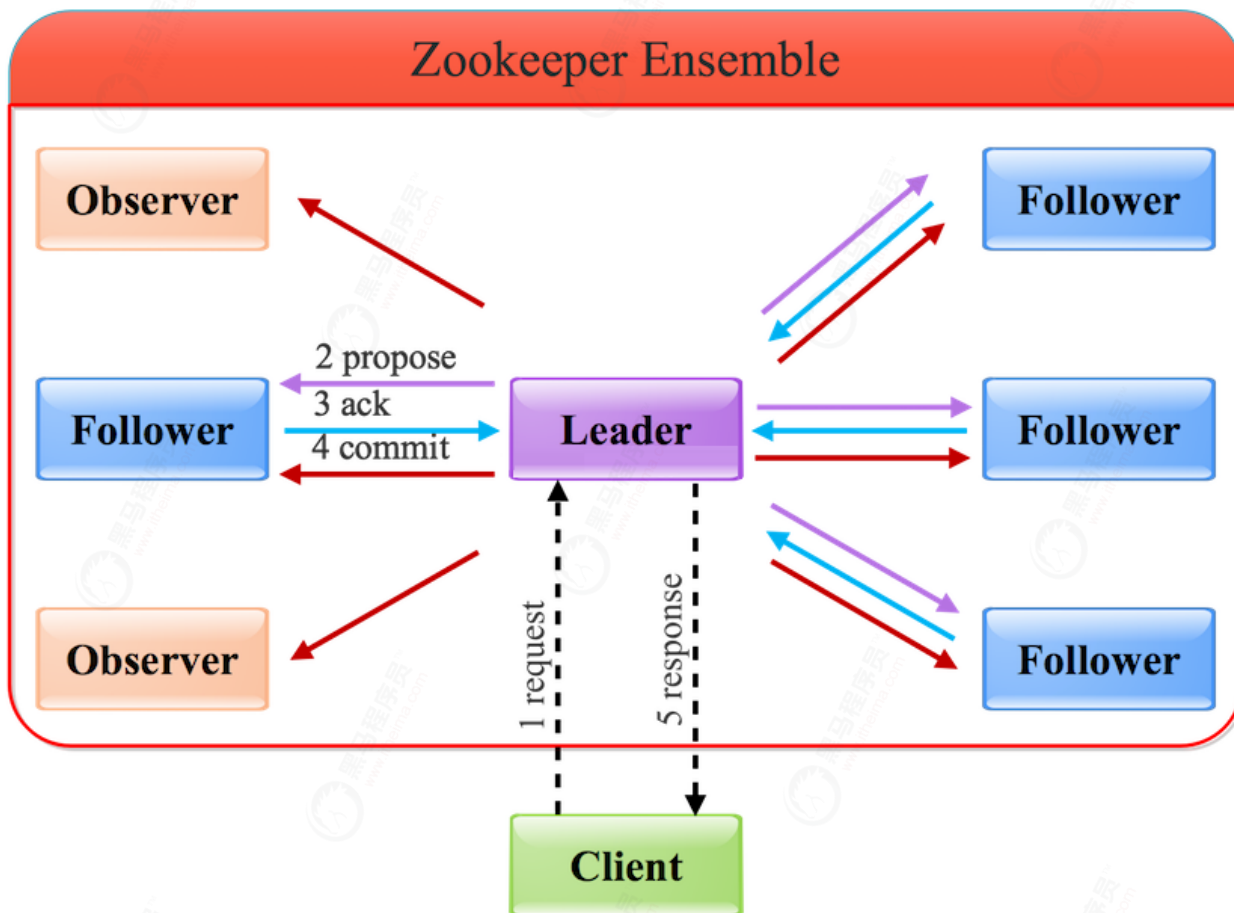


1. Leader（主）服务器接收 Client（客户端）的请求，为其生成 Proposal（提案）。
2. 然后将 Proposal 发送给所有 Follower（从）。主会为每一个从分配一个单独的队列，以保证消息的有序性。
3. 主节点等待所有从服务器反馈 Ack，当有过半的从服务器 Ack 之后，主节点会提交本地事务。然后广播 Commit 给所有，从节点接收到 Commit 之后完成提交。

1.6 ZK集群数据交互与同步原理（拓展）

1.6.1 集群数据交互流程

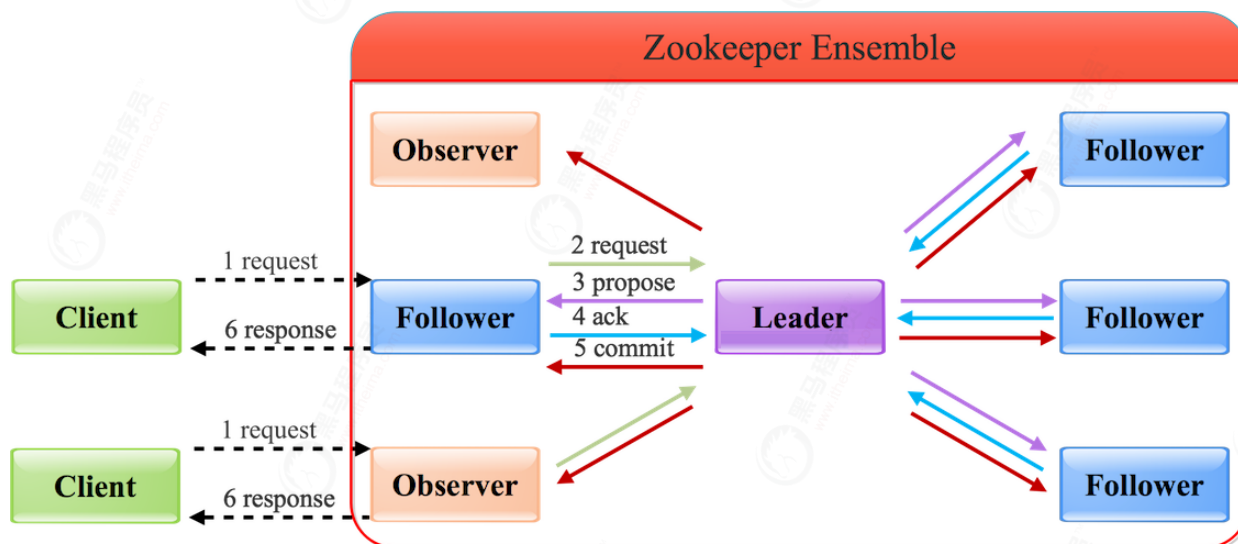
- 基于Leader的写操作流程:



处理流程:

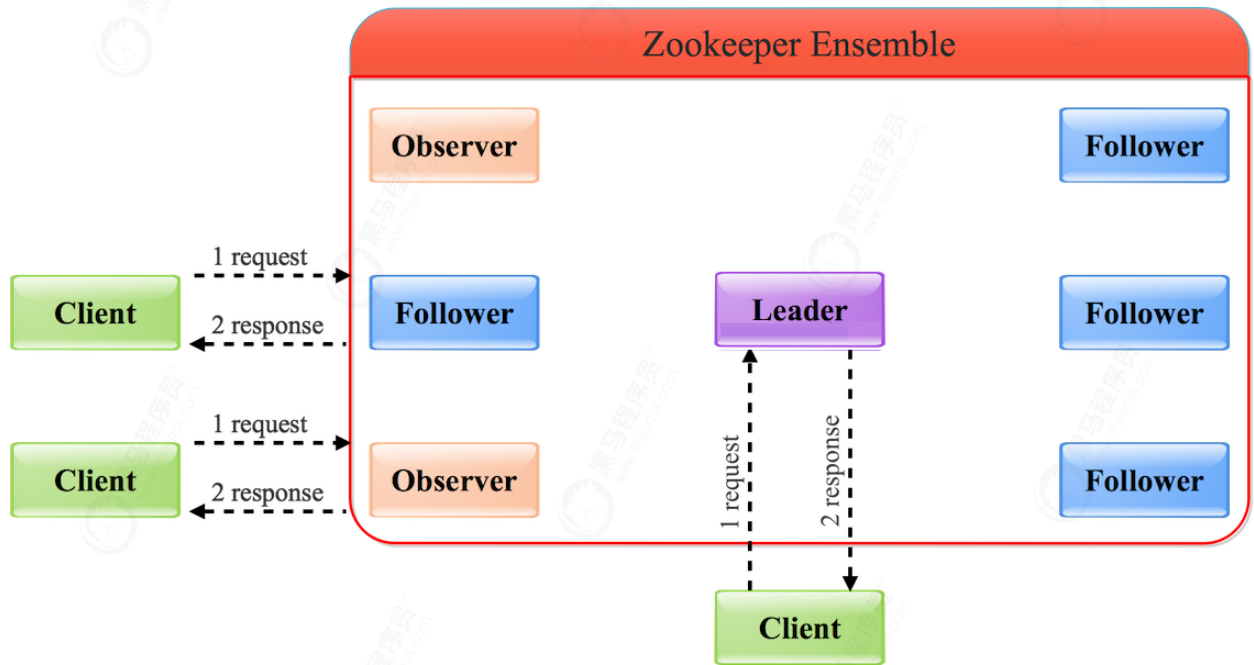
1. 客户端向Leader发起写请求
2. Leader将写请求以**Proposal**的形式发给所有**Follower**并等待**ACK**
3. Follower收到Leader的Proposal后返回ACK
4. Leader得到过半数的**ACK**（Leader对自己默认有一个ACK）后向所有的Follower和Observer发送Commit
5. Leader将处理结果返回给客户端

• 基于**Follower/Observer**的写操作流程:



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，然后按照基于Leader写操作流程处理数据。

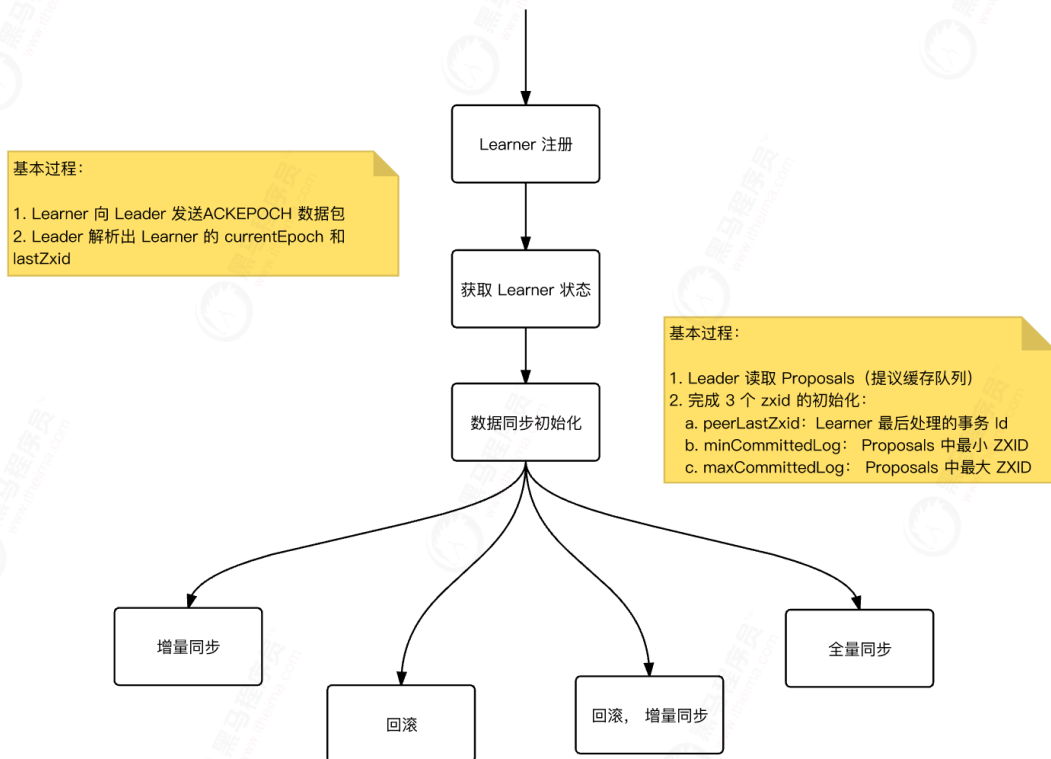
- 读操作流程:



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。

1.6.2 ZK集群数据同步原理

- 数据同步流程:



Leader服务器根据peerLastZxid、minCommittedLog、maxCommittedLog的值决定数据同步类型:

- o 差异化同步(DIFF同步)
- o 回滚同步(TRUNC同步)

- 先回滚再差异化同步(TRUNC + DIFF同步)
- 全量同步(SNAP同步)

参数解析:

- peerLastZxid: Learner服务器 (Follower或observer) 最后处理的zxid。
- minCommittedLog: Leader服务器proposal缓存队列committedLog中的最小的zxid。
- maxCommittedLog: Leader服务器proposal缓存队列committedLog中的最大的zxid。

follow节点代码:

```
void followLeader() throws InterruptedException {
    // 省略
    try {
        QuorumServer leaderServer = findLeader();
        try {
            /**
             * 1. follower 与 leader 建立连接
             */
            connectToLeader(leaderServer.addr, leaderServer.hostname);

            /**
             * 2. follower 向 leader 提交节点信息用于计算新的 epoch 值
             */
            long newEpochZxid = registerWithLeader(Leader.FOLLOWERINFO);

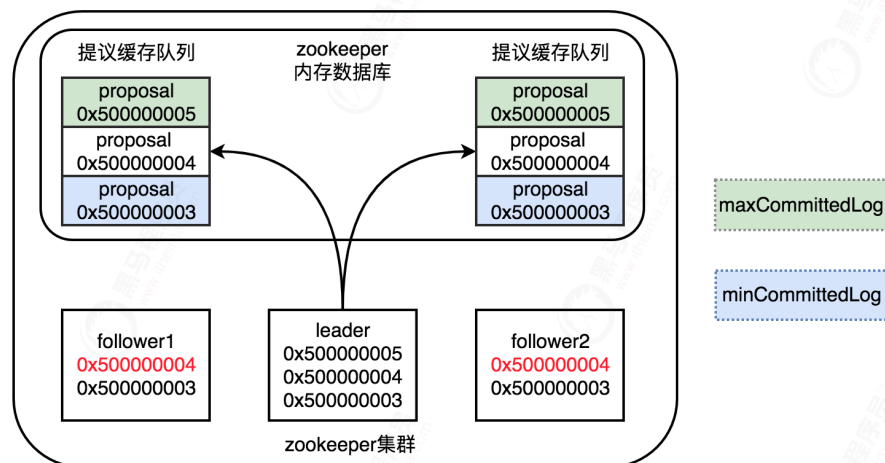
            /**
             * 3. follower 与 leader 数据同步
             */
            syncWithLeader(newEpochZxid);

            // 省略

        } catch (Exception e) {
            // 省略
        }
    } finally {
        // 省略
    }
}
```

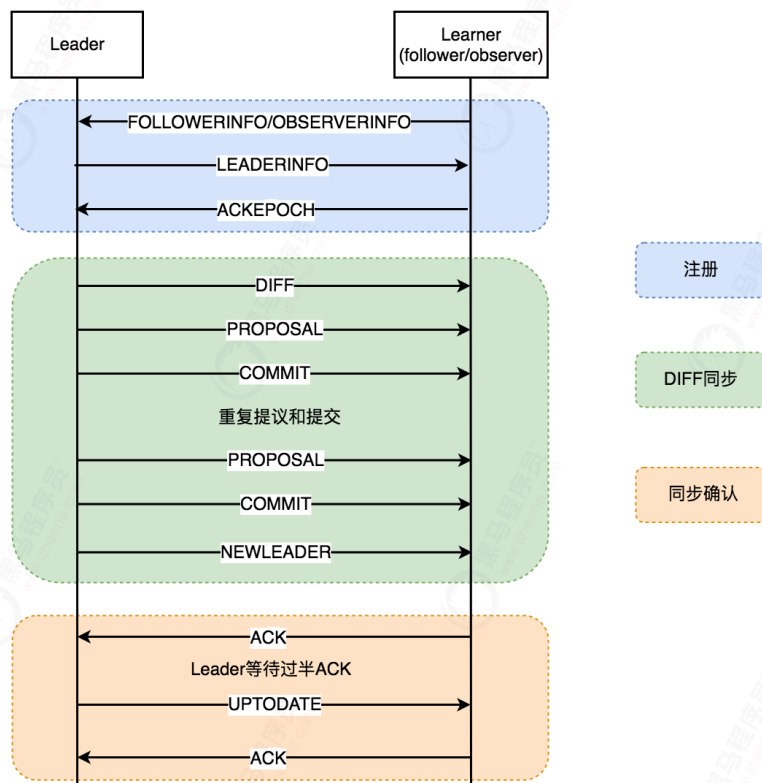
处理流程:

1. 请求与 leader 建立连接。
 2. 向leader提交节点信息计算新的 epoch 值。
 3. follow与leader进行数据同步。
- **DIFF** (差异化同步)



该图中Follower最后处理的zxid为**0x500000004**，minCommittedLog为**0x500000003**，maxCommittedLog为0x500000005，peerLastZxid在minCommittedLog和maxCommittedLog之间，就会走**DIFF**差异化同步。

Follower和Leader之间同步处理：

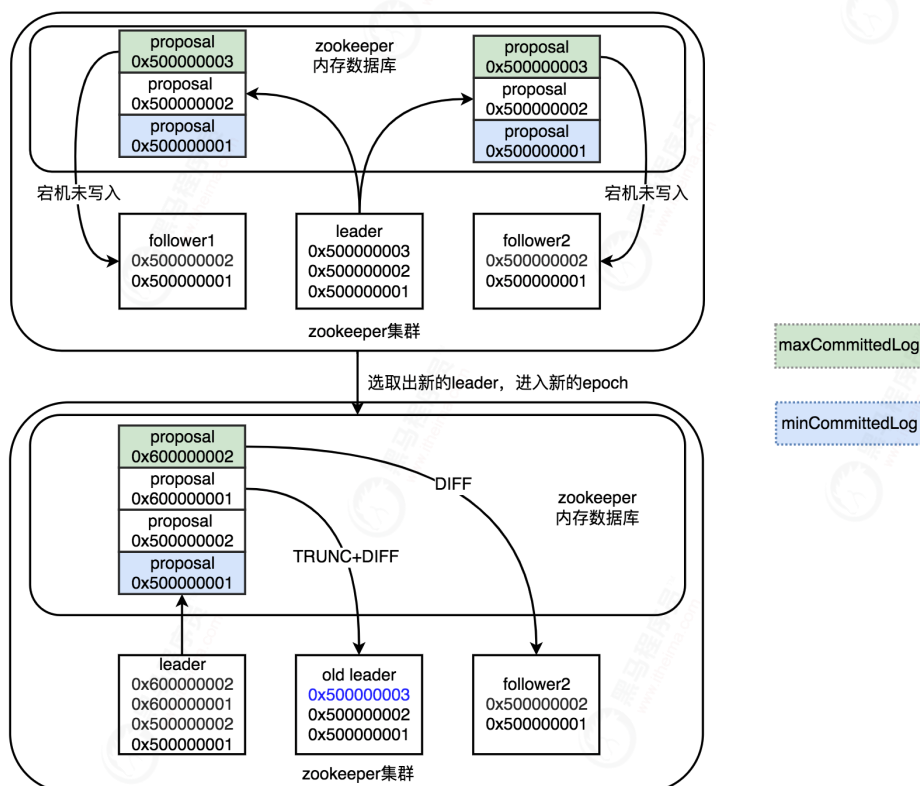


- Learner会给Leader发送**ACKEPOCH**数据包，将当前Learner的currentEpoch和最新事务序号**lastZxid**发送过去，并告诉Leader自己的状态。
- 在判断确认采用DIFF差异化同步后，Leader会发送**DIFF**指令给**Learner**，告诉它即将开始同步差异化的proposal（即Leader已提交但是Learner还未提交的proposal）。
- 对于上图情况，只有一个**proposal**需要同步，以此为例，Leader服务器会为一个proposal发送两个数据包给Learner来完成同步，分别是**PROPOSAL**内容数据包和**COMMIT**指令数据包，这两个一组拥有相同的**zxid**。如果有多个需要同步的**proposal**，就重复发送proposal对应的这两个包给Learner来实行同步直到最后完成所有的同步。
- 请求缓存队列中的proposal同步完成后，Leader会发送一个NEWLEADER指令到Learner。
- Learner收到NEWLeader指令后，会反馈一个**ACK**消息到**Leader**，表示自己确认完成请求缓存队列中proposal的同步。

- 以上是针对一个**Learner**的同步，会单独在一个**LearnerHandler**线程进行处理，其他的Learner也会有对应的LearnerHandler线程来处理，Leader主线程会等待Learner的同步结果。
- 如果是满足“过半策略”后，Leader服务器会向所有完成同步的**Learner**发送**UPTODATE**指令，告诉它们数据已经是最新了的，并且集群因为过半达到数据一致可以对外提供服务。
- 最后Learner在接收到**UPTODATE**指令后，会停止与Leader的数据同步，并再次反馈一个**ACK**消息。

• TRUNC+DIFF(回滚与差异化同步)

在实际运行当中，主节点leader可能会出现故障，比如重启，经过一段时间之后，数据是如何同步呢？当Leader将事务提交到本地事务日志中后，正准备将proposal发送给其他的Follower进行投票时突然宕机，这个时候Zookeeper集群会选取出新的Leader对外服务，并且可能提交了几个事务，经过一段时间之后，当旧的Leader再次上线，新Leader发现它身上有自己没有的事务，就需要回滚抹去旧的Leader上自己没有的事务，再让旧的Leader同步完自己新提交的事务，这个就是TRUNC+DIFF的应用场景。



- 当Leader准备将xid为0x500000003的proposal发送Learner投票就宕机了，导致Leader上会多出一条未在集群同步的数据。
- 此时选取了新的Leader，并且epoch在上次的基础上加1，Zookeeper集群进入了新一轮时代，并且新Leader提交了两个事务，xid分别为0x600000001和0x600000002。
- 当旧的Leader重新上线后，新Leader发现它身上有一个0x500000003事务记录是自己没有的，这个时候对于旧的Leader来说，peerLastZxid为0x500000003，而minCommittedLog为0x500000001，maxCommittedLog为0x600000002，peerLastZxid在minCommittedLog和maxCommittedLog之间。这个时候新Leader会发送TRUNC指令给这个旧的Leader(实质上是Learner，旧的Leader是便于理解)，让它截取一部分事务记录，这样老Leader会截取到最靠近peerLastZxid同时又存在于提议缓存队列的事务，即截取掉0x500000003的事务记录。
- 截取完成后，后面就是DIFF差异化同步了，流程和前面所讲一致。

• TRUNC(回滚同步)

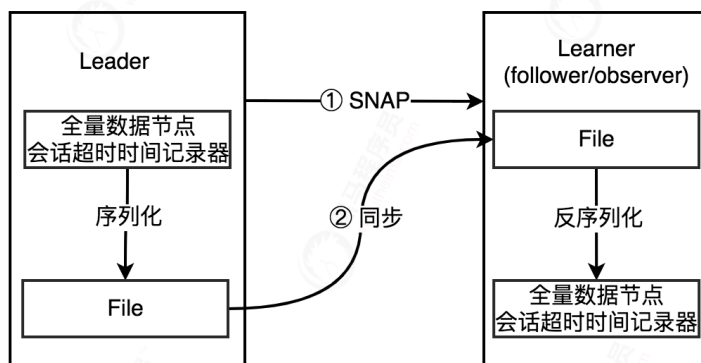
在什么场景下采用TRUNC？

如果Learner上的peerLastZxid的值，比maxCommittedLog还要大，这样只需要截取多余的部分事务记录，回滚至 maxCommittedLog就可以了，无需进行DIFF差异化同步。

这个就是TRUNC模式，该场景可以认为是 **TRUNC+DIFF** 的简化模式。

- **SNAP(全量同步)**

如果follower 的 peerLastZxid 小于 leader 的 minCommittedLog 或者 leader 节点上不存在提案缓存队列时，将采用 SNAP 全量同步方式。



该模式下 leader 首先会向 follower 发送 SNAP 报文，随后从内存数据库中获取全量数据序列化传输给 follower，follower 在接收全量数据后再进行反序列化加载到内存数据库中。

leader 在**SNAP**全量数据同步之后，会向 follower 发送 **NEWLEADER** 报文，在收到过半的 follower 响应的 ACK 之后，说明过半的节点已经完成了数据同步，接下来 leader 就会向 follower 发送 **UPTODATE** 报文告知 follower 节点可以对外提供服务了，此时 leader 会启动 zk server 开始对外提供服务。