## ZAB协议

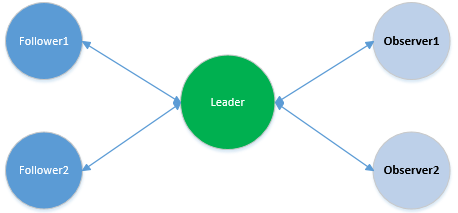
ZAB（Zookeeper Atomic Broadcast）是为Zookeeper 设计的支持崩溃恢复和原子广播的一致性协议。

### ZK集群中节点的三种角色

**Leader**：一个ZK集群中同一时间只有一个Leader，负责读写请求的处理。

**Follower**：一个ZK集群中可以存在多个Follower，可直接处理读请求，转发写请求给Leader，对Leader广播的写请求进行投票。当Leader下线后Follower将参与Leader竞选。

**Observer**：除了不参与投票，其它都与Follower一样。可以在不影响集群写请求处理时间的情况下增加集群读请求的处理能力。



### ZK集群节点的状态

ZK集群中节点有三种状态：

* LOOKING

Leader选举阶段

* LEADER

Leader服务器作为主进程领导状态

* FOLLOWING

Follower服务器与Leader服务器保持同步状态

Zookeeper集群中的节点首次启动的时候都是LOOKING状态，此时集群中不存在LEADER状态的节点，所有处于LOOKING状态的节点都会试图选出一个新的Leader，随后如果进程发现已经选出一个Leader， 则它就会进入到FOLLOWER状态，并开始和Leader服务器保持同步。我们将处于LEADER状态的节点称为leader，处于FOLLOWING状态的节点称为follower。在集群运行过程中leader随时可能会下线，当leader下线后，集群中的follower节点就会变为LOOKING状态，重新选举出leader。

**术语解释**

**Quorum**：过半机器数，这个是整合Leader选举算法中最重要的一个术语，我们可以理解为一个量词，指的是ZooKeeper集群中过半的机器数，公式为Quorum=(n/2+1)。例如：如果集群机器总数为3，那么Quorum就是2

### ZAB协议介绍

ZAB协议由消息广播和崩溃恢复两个模式组成，其中崩溃恢复模式又分为Leader选举和数据同步，接下来就介绍一下这两种模式。

### · 消息广播

1. 当Leader接收到一个写请求会将请求封装成Proposal
2. 将事务操作写入到本地日志文件，当写入完成后会向Proposal投票收集器发送ACK反馈。
3. 将Proposal发送给集群中过半的Follower，然后进入等待
4. Follower收到Leader广播的Proposal后，(1)写入日志。（2）日志写入完成后响应给Leader一个ACK。（3）等待Leader的commit命令
5. Leader接收Follower的ACK响应，Proposal投票收集器进行统计，当收到的ACK数过半时，（1）将事务操作写入内存数据库。（2）向Follower发送commit命令。（3）响应客户端。
6. Follower收到commit命令后对Proposal进行提交，写入日志文件。

如图1所示消息广播流程示意图：

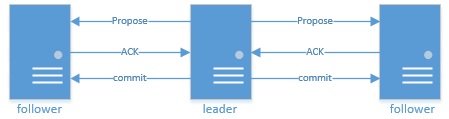


图1

可以看出，ZAB协议的消息广播类似于二阶段提交，不同于二阶段提交的是：

**·** 消息广播过程没有回滚机制，Follower要么响应Leader，要么抛弃Leader

**·** 因为消息广播过程没有回滚机制，所以Leader只需要获得超过半数的来自Follower的ACK即可提交事务操作，而不必等待所有Follower响应。

当然，在这种简化的二阶段提交模型下，是无法处理Leader宕机而带来的数据不一致问题，因此ZAB协议就增加了另一个崩溃恢复模式来解决这个问题。另外，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，所以很容易保证消息广播过程中消息发送与接收的顺序性。

### · 崩溃恢复

当Leader因为宕机、网络分区等各种原因无法与过半的Follower保持通信的时候，集群就会进入崩溃恢复模式，选举新的Leader。

在消息广播模式中提到了Leader宕机可能会带来数据不一致问题，我们先看一下会有哪些数据不一致的情况：

1. 如图2所示，

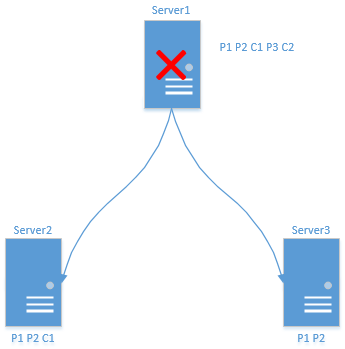


图2

Server1作为leader节点，先后发送了消息P1、P2、C1、P3、C2给follower节点Server2、Server3，其中P1、P2、P3是发送的提案，而对应的C1、C2是对之前发送的提案P1、P2的commit，当leadedr发送了C2消息后出现故障立即退出了，此时Server2收到了P1、P2、C1，Server3收到了P1、P2。对于这种情况就需要保证P2最终能在所有服务器上被提交，否则就会出现不一致。

1. 如图3所示，

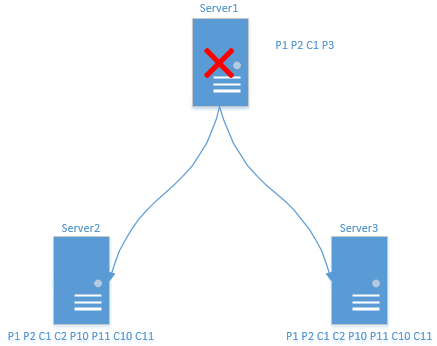


图3

Server1作为leader节点发送P3后发生故障立即退出了，导致Server2和Server3

都未能收到事务P3。于是当Server1重启后再次加入集群时，需要确保丢弃P3这个事务。

**· Leader选举**

所以ZAB协议选举Leader就需要提供两点保证：

**·** 确保已经被Leader提交的Proposal最终也能被所有服务器提交

**·** 确保丢弃那些只在Leader上被提出的Proposal

针对这些要求，如果让Leader选举算法保证新选举出来的Leader拥有集群所有机器中编号(即ZXID)最大的事务Proposal，那么就可以保证新选举出来的Leader一定拥有所有已经提交的事务Proposal(TCP连接的FIFO特性)，更为重要的是，如果让拥有最高编号事务Proposal的机器成为Leader，就可以省去Leader服务器检查Proposal的提交和丢弃这一步操作了。

**· 数据同步**

当Leader选举完成后，在集群正式开始工作之前(即接收客户端的请求，提出新的事务Proposal)，需要Leader确认事务日志中的所有Proposal是否被过半的Follower提交了，也就是说集群中过半的Follower是否和Leader保持同步状态，下面我们看下ZAB协议的数据恢复过程。

集群中的所有机器要么成为Leader，要么成为Follower与Leader保持同步，Leader需要确保所有Follower能够收到每一条Proposal，并将已经提交的Proposal应用到内存数据库中。具体的，Leader会为每一个Follower维护一个队列，并将那些没有被各Follower同步的事务以Proposal的形式放入队列中最终发送给Follower，并在每一个Proposal后面紧跟着一个Commit消息。等到Follower将所有未同步的事务Proposal从Leader服务器上同步过来并且成功应用到内存数据库后，Leader会将这些Follower放到真正可用的Follower列表中，并开始之后的其它流程。

上面讲到的是正常情况下的数据同步流程，下面来看下ZAB是如何处理那些需要被丢弃的事务Proposal的。在ZAB协议的事务Proposal的设计中，事物编号ZXID是一个64位的数字，其低32位可以简单看做是一个单调递增的计数器，针对每一个客户端事务，Leader在产生新的事物Proposal的时候都会对该计数器进行+1操作，高32位代表Leader周期epoch的编号，每当选举出一个新的Leader，就会从这个Leader服务器上取出其事务日志中最大事物Proposal的ZXID，并从ZXID中解析出epoch值然后+1，之后就以此值作为新的epoch，并将低32位置0来生成新的ZXID。ZAB协议中通过这样设计的ZXID来区分不同的Leader周期的策略，可以有效的避免不同的Leade服务器错误的使用相等的ZXID编号提出不一样的事务Proposal的异常情况，这对于识别在Leader崩溃恢复前后生成的Proposal非常有帮助，大大简化和提升了数据恢复流程。

基于这样的ZXID设计，当一个包含了上一个Leader周期中尚未提交过的事物Proposal的节点启动时，其肯定无法成为Leader，原因很简单，因为当前集群中肯定包含一个Quorum集合，该集合中的机器一定包含了更高epoch的事务Proposal，因此这台机器的事务Proposal编号肯定不是最高的，自然也就无法成为Leader了。当这台机器以Follower的角色加入到集群中时，Leader就会根据自己服务器上最后被提交的Proposal来和Follower中的Proposal进行比对，比对结果当然是让Follower服务器进行回退操作，回退到一个已经被集群中过半节点提交的最新的Proposal。举个例子，图3中当Server1重启后连接上集群中的Leader，Leader就会要求Server1去除P3。

### 深入了解ZAB协议

我们已经了解了ZAB协议的大致内容，接下来将从系统模型、算法描述和运行分析三个方面来深入了解ZAB协议。

#### 系统模型

在深入讲解ZAB协议之前，我们先来抽象的描述一下ZAB协议需要构建的分布式系统模型。通常在由一组进程∏={P1,P2....Pn}组成的分布式系统中，每个进程都有自己的存储设备，各进程之间通过通信来进行消息传递。一般的在这样的分布式系统中，每一个进程都可能出现几次或多次的崩溃退出，并且在恢复后会重新加入到进程组∏中去。我们称能正常工作的进程处于UP状态，如果一个进程崩溃了我们称其处于DOWN状态。事实上当进程组中存在过半的处于UP状态的进程组成的一个子集时，就可以进行正常的消息广播了，我们将这样的一个进程子集称为Quorum(简称Q)，并假设这样的Q已经存在，其满足：



上述集合关系式表示，存在这样的进程子集Q，其必然是∏的子集；同时，存在这样的子集Q1和Q2，Q1和Q2的交集必定非空。

我们使用Pi和Pj来表示进程组∏中的两个进程，使用Cij表示进程Pi和Pj之间的通信通道，其满足如下两个特性：

**· 完整性**

如果Pi收到了来自Pj的消息m，那么Pj一定发送了消息m给Pi。

**· 前置性**

如果进程Pj收到了消息m，那么存在消息m’，如果消息m’是消息m的前置消息，那么进程Pj必须先收到消息m’，然后再收到消息m，我们将存在这种前置性的关系的两个消息表示为。前置性是整个协议设计中最关键的一点，因为对于ZK来讲，其每一个操作都是基于前一个操作进行的，所以所有消息都必须按照严格的先后顺序来执行。

#### 算法分析

整个ZAB协议包括崩溃恢复和消息广播两个过程，进一步可以细分为**发现、同步、广播**三个阶段，组成ZAB协议的每一个分布式进程都将循环的执行这三个进程，我们称这样一个循环代表一个主进程周期。

为了更好的进行ZAB协议各阶段的描述，我们先定义一些专有术语和标识：

|  |  |
| --- | --- |
| Fp | Follower f处理过的最后一个事务Proposal |
| Fzxid | Follower f处理过的最后一个事务Proposal的zxid |
| Hf | Follower f处理过的所有历史事务的序列 |
| Ie | 初始化历史记录，当准Leader完成阶段一后，此时它的Hf就被标记为Ie |

##### 阶段一：发现

阶段一的过程就是Leader选举的过程，用于在多个分布式进程中选出一个主进程，准Leader与Follower的工作流程如下：

1. 集群中的Follower将自己处理过的最后一个事务Proposal的epoch值CEPOCH(Fp)发送给准Leader
2. 当准Leader收到集群中过半的Follower发送的CEPOCH(Fp)后，取收到的最大的epoch然后+1得到e'，使用e'作为新一轮Leader的epoch编号，将e'发送给集群中过半的Follower。
3. 当Follower收到Leader发来的NEWEPOCH(e')后，如果检测其CEPOCH(Fp)小于e'，则赋值CEPOCH(Fp)为e'，同时向准Leader发送ACK响应。ACK响应(ACK-E(Fp，Hf))中包含了当前该Follower的epoch信息CEPOCH(Fp)和该Follower的历史事务集合Hf。

当准Leader接收到了过半Follower发送的ACK信息后，就会从这过半的Follower中选择一个Follower F，使用其的Hf作为准Leader的初始化事务集合Ie

关于这个Follower F的选择对于Quorum中的任意一个Follower F’，F需要满足以下两个条件中的一个：

CEPOCH(Fp) > CEPOCH(F'P)

CEPOCH(Fp) = CEPOCH(F'p) & (Fzxid > F'zxid || Fzxid = F'zxid)

##### 阶段二：同步

当完成发现阶段后，就进入了同步阶段。同步阶段主要完成Follower与准Leader的数据同步。

1. 准Leader将e'和Ie发送以NEWLEADER(e', Ie)形式发送给所有Quorum中的所有Follower。
2. Follower收到消息NEWLEADER(e', Ie)后，如果其CEPOCH(Fp)≠e'，则直接进入下一轮循环，因为此时Follower还处于上一轮选举周期，或是更上一轮，无法参与本轮同步。

如果CEPOCH(Fp)=e'，那么Follower就会执行事务应用操作，具体的对于每个Proposal：<v,z>∈Ie，Follower都会接受<e',<v,z>>。当Follower处理完Ie中所有Proposal后，会给准Leader一个反馈，表示自己已经接受并处理完了Ie中所有的事务Proposal。

1. 当准Leader收到过半的Follower针对NEWLEADER(e', Ie)的反馈后，就会向所有Follower发送commit消息。至此准Leader完成阶段二，从准Leader成为正式的Leader。
2. 当Follower收到准Leader的commit消息后，就会以此提交在步骤2中Ie集合中的事物Proposal。支持Follower完成阶段二。

##### 阶段三：广播

当准Leader完成阶段二后就成为了正式的Leader，此时集群就可以对外提供服务了，对于客户端的每个写请求，Leader都会向集群中的所有Follower进行广播。

1. Leader对于接收到的每个写请求，都会转换为事务Proposal<e',<v,z>>，其中epoch(z)=e'，然后根据zxid的顺序依次发送给集群中的所有Follower，然后等待Follower的响应。
2. Follower根据接收到的事务Proposal的先后顺序依次处理，并将它们添加到历史事务集合Hf中，然后响应给Leader一个ACK。
3. Leader收到过半Follower针对事务Proposal<e',<v,z>>的ACK消息后，就会向集群中的所有Follower发送commit消息<e',<v,z>>，要求进行事务提交。
4. Follower收到来自Leader的commit<e',<v,z>>消息后，就会对事务Proposal<e',<v,z>>进行提交，应用到内存数据库中。需要注意的是，此时Follower必定已经提交了事务Proposal<v',z'>，其中z'∈Hf，z’z。

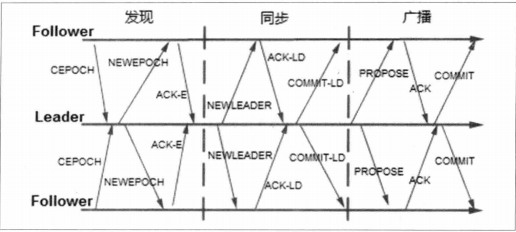


图4. ZAB协议描述示意图

以上就是ZAB协议的三个核心流程，在正常运行过程中会一直运行于阶段三反复的进行消息广播流程，当Leader崩溃退出或因为其他原因导致Leader缺失，则会进入到阶段一和二，选举出新的Leader。

当集群处于阶段三的时候，Leader会为每个Follower单独分配一个队列，Leader把接收到的客户端写请求转换为事务Proposal后按先后顺序放入每个Follower对应的队列，然后由单独的线程将队列中的事务Proposal发送给对应的Follower。这样根据TCP连接的FIFO特性，就可以保证Follower收到的事务Proposal的顺序性。

### FastLeaderElection算法

上面我们了解了ZAB协议，可以说Zookeeper集群是在ZAB协议的下运行的，其中Leader选举是Zookeeper集群运行中比较重要的一步，是保证分布式数据一致性的关键所在。

那么Leader选举算法具体是怎样实现的呢？接下来我们就了解一下ZAB协议中Leader选举具体实现算法。

ZooKeeper提供了三种选举算法，分别是LeaderElection、UDP版本的FastLeaderElection和TCP版本的FastLeaderElection，从3.4.0版本开始LeaderElection和UDP版本的FastLeaderElection就被废弃了。下文仅对TCP版本的FastLeaderElection算法进行介绍。

**术语解释**

**SID：服务器ID**

SID是一个数字，用来标识集群中的一台服务器，每台机器不能重复，和myid文件中的值一致。

**ZXID：事务id**

ZXID是由64位组成的数字，用来标识一次服务器状态的变更，可以理解成Leader生成的事务Proposal的编号。在前面的ZAB协议中有介绍到过。

**Vote：投票**

Leader选举，顾名思义要通过投票来完成。

接下来就开始深入了解一下Leader选举算法。

当集群中的一台服务器出现以下两种情况之一，就会进入Leader选举状态(也就是LOOKING状态)：

**·** 服务器初始化启动

**·** 服务器运行期间无法与Leader保持连接

而当一台服务器进入Leader选举状态，此时集群可能处于以下两种状态：

**·** 集群中本来已经存在Leader

**·** 集群中确实不存在Leader

针对第一种情况，通常是因为一台机器启动较晚，然后加入到集群中，在它加入到集群之前，集群中就已经可以正常工作了，也就是集群中已经存在Leader了。当该机器发起选举的时候就会被集群中的其它机器告知Leader的信息，然后该机器与Leader建立连接即可。

接下来我们重点来看在集群中不存在Leader的情况下，如何进行Leader选举。

**第一次投票**

集群中不存在Leader有两种情况，一种情况是集群处于刚启动状态，需要选举出一个Leader；另一种情况是集群处于运行状态，Leader因崩溃退出或无法与过半Follower保持连接等原因，需要重新选举Leader。无论哪种情况，此时集群中的服务器都处于试图选举出一个Leader的状态，我们把这种状态称为LOOKING状态。当一台服务器处于LOOKING状态时，就会通过向集群中的其它机器发送消息来选举Leader，我们称这个消息为“投票”。

投票消息中包含了两个最基本的信息SID和ZXID，分别表示被推举的服务器的唯一标识和被推举的服务器的最大事务ID。为了描述方便，我们使用(SID, ZXID)表示一个投票。

假设有一个由五个节点组成的ZooKeeper集群，SID分别为1、2、3、4、5，ZXID分别为2、2、2、1、1。并且此时SID为2的服务器为集群中的Leader，某一时刻SID为1和2的机器宕机了，此时集群中的其它机器就进入了LOOKING状态，重新选举Leader。

在进行第一次投票的时候，由于还无法检测集群中其它机器的状态信息，因此每台服务器都将自己作为推举对象发送给集群中其它服务器，在此例中SID为3、4、5的机器的投票情况分别是(3,2)、(4,1)、(5,1)。

**变更投票**

集群中的每台服务器发送完自己的选票后，也会接收其它服务器发来的选票，然后与自己当前投出的选票进行对比，决定是否变更投票。

先定义一些术语：

vote\_sid：接收到的投票中所推举的服务器的SID

vote\_zxid：接收到的投票中所推举的服务器的ZXID

self\_sid：当前服务器投出的投票中所推举服务器的SID

self\_zxid：当前服务器投出的投票中所推举服务器的ZXID

定义完一些术语之后，我们来看一下投票对比规则：

**·** vote\_zxid大于self\_zxid，更新当前服务器的投票为接收到的投票，然后再次将更新后的投票发送出去。

**·** vote\_zxid小于self\_zxid，当前服务器忽略接收到的投票，不做任何操作。

**·** vote\_zxid等于self\_zxid，那么就比较两者的SID，如果vote\_sid大于self\_sid，则更新当前服务器投票为接收到的投票，然后再次将更新后投票发送出去。

**·** vote\_zxid等于self\_zxid，并且vote\_sid小于self\_sid，当前服务器忽略接收到的投票，不做任何操作。

结合上面说到的5台机器组成的ZooKeeper集群，我们来分析一下投票变更的过程，如图5所示：

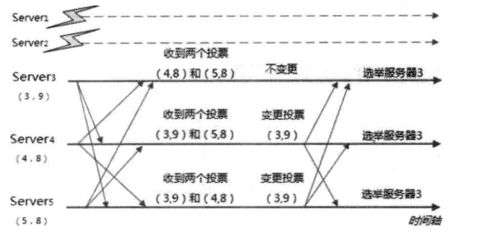


图5

对于Server3来说，接收到来自Server4和Server5的投票(4,8)和(5,8)，根据比对规则，Server3坚持自己的投票(3,9)，不做任何变更。

对于Server4来说，接收到了来自Server3和Server5的投票(3,9)和(5,8)，根据比对规则，Server4更新自己的投票为(3,9)，然后将更新后的投票广播出去。

对于Server5来说，接收到了来自Server3和Server4的投票(3,9)和(4,8)，根据比对规则，Server5更新自己的投票为(3,9)，然后将更新后的投票广播出去。

**确定Leader**

经过第二次投票后，集群中的每台机器都会再次受到其它机器的投票，然后开始统计投票。如果一台机器收到了超过半数的相同的投票，那么这个投票中SID所对应的机器就成为了Leader。

在上述的由5个机器组成的ZooKeeper集群中，需要3个或3个以上一致的投票即可。Server4和Server5都投票(3,9)，所以Server3成为了Leader。Server4和Server5会向Leader注册成为Follower，然后开始数据同步。

以上就是FastLeaderElection算法的介绍，简单的说哪台服务器的数据越新，哪台服务器就可以成为Leader。因为数据越新的服务器的ZXID越大，也就更能保证数据的恢复。如果集群中存在多个ZXID相同的机器，则SID最大的成为Leader。