# 基础篇

### 1. 分布式理论

* **集中式系统**：由一台或多台计算机组成中心节点，数据集中存储于这个中心节点中。
* **分布式系统**：是一个硬件或软件组件分布在不通的网络计算机上，彼此之间仅仅是通过消息传递进行通信和协调的系统
* 为了对外提供高可用服务，避免单点故障，方便水平扩展，就产生从集中式到分布式的转变需求，但是分布式系统会出现以下问题，而这几个问题又相互影响，比如网络问题和并发问题会导致数据不一致
* 网络问题

分布式系统一大特点是通过消息传递进行通信和协调就不可避免网络问题，网络问题包括网络延迟、脑裂(split-brain)、网络不可达(unreachable)

单机内存访问延时在纳秒级，而网络通信需要0.1-1ms，如果经过过多网络设备会延时更多，丢包和延迟非常普遍

当网络发生异常，不管是硬件或者软件，都有可能发生"网络分区",俗称"脑裂(split-brain)",比如5台服务器，2台可以互相通信，另外3台可以互相通信，而这前2台和后3台之间无法通信，就形成两个小规模集群，这两个小集群很有可能分布在不同机架上或者在不同的数据中心

由于网络异常，更会普遍发生的情况是数据传输超时或者建立连接超时，此时消息发送端并不知道消息是否成功发送

* 数据一致性

为了保证高可用，数据会多副本分布在不同服务器上，当提供服务的副本挂了以后，需要将服务切换至其它副本，副本之间的数据需要保证完整性和一致性

* 并发问题缺乏全局时钟

分布式系统的多个节点在同一时刻并发操作共享的资源；在分布式系统中，很难定义两个事件谁先谁后，原因就是因为分布式系统缺乏一个全局时钟序列控制

* **数据一致性**

一致性有以下几种等级

* **强一致性**：任何client在写入的数据，都能在之后被人任何client既完整又一致的看到，不会出现看到旧数据，也不会看到不一致的数据。
* **弱一致性**：写入数据成功后，并不能保证立即可以看到读入的值，能保证在某个时间级别达到一致性，如果能保证在一定时间内(比如zk是保证任何存活的节点在syncLimit\*ticktime时间范围内达到数据一致)达到数据一致，就叫做**最终一致性**

如果细分**，**弱一致性分为**会话一致性和用户一致性，**zookeeper可以保证会话一致性和用户一致性(client重连时会拿最后一次zxid去验证，如果新连的server存在zxid那么就建立会话)

* ACID

在传统单机数据库中，我们能很容易实现满足ACID特性的事务处理系统

事务具有以下特性，A--原子性，C--一致性，I-隔离性(read uncommitted,read committed,repeatable read,serializable)

D---持久性

* 分布式事务

典型的分布式事务场景：一个跨银行的转账操作涉及调用两个异地银行服务，其中一个是本地银行提供的取款服务，另一个则是目标银行提供的存款服务，两个操作必须都成功要么都失败。

* **我所理解的CAP理论**

CAP原文(摘自wiki):

CAP定理（CAP theorem），又被称作布鲁尔定理（Brewer's theorem），它指出对于一个[分布式计算系统](http://zh.wikipedia.org/wiki/%E5%88%86%E5%B8%83%E5%BC%8F%E8%AE%A1%E7%AE%97)来说，不可能同时满足以下三点：[[1]](http://zh.wikipedia.org/wiki/CAP%E5%AE%9A%E7%90%86#cite_note-Lynch-1)[[2]](http://zh.wikipedia.org/wiki/CAP%E5%AE%9A%E7%90%86#cite_note-2)

一致性（[Consistency](http://en.wikipedia.org/wiki/Consistency_(database_systems%EF%BC%89))（等同于所有节点访问同一份最新的数据副本）

可用性（[Availability](http://en.wikipedia.org/wiki/Availability)）（对数据更新具备高可用性）

容忍网络分区（[Partition tolerance](http://en.wikipedia.org/wiki/Network_partitioning)）（以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择[[3]](http://zh.wikipedia.org/wiki/CAP%E5%AE%9A%E7%90%86#cite_note-3)。）

由于CAP理论是2000年提出的，每项定义很模糊，我所理解的CAP理论如下：

* 一致性C:操作总是能读取到之前完成的写操作结果；
* 可用性A:读写操作总是能够在很短的时间内返回，即使某台机器发生了故障，也能够

通过其它副本正常执行，而不需要等到机器重启或者机器上的服务分配给其它机器以后才能成功；

* 分区可容忍性P:无法做到机器宕机，机房停电或者出现机房之间网络故障等异常情况的容错；

以上三点不能同时满足，

目前**满足CA**的数据库系统有：mysql，postgres，因为是单机的无法满足P，单机访问也不存在数据不一致和长时间无法响应的问题(在无法容忍机器宕机、网络异常等情况)

**满足AP**的：Cassandra(基于gossip协议达到最终一致性)，zookeeper

**满足**CP的：Mongodb(可以设置写请求被所有节点响应后才返回)

因为为了对外可靠的服务避免宕机及网络故障，那么我们必须保证P，要能容忍分区出错（包括宕机、网络异常），那么就需要在A可用性和C强一致性进行取舍，假如我们保证强一致性，每次写操作就要将数据同步到其它所有节点，要么每次读操作读到所有节点，取最新的数据，要么读的节点数+写的操作数总和大于总副本数，无论哪种情况，都会增加多次网络开销造成可用性(在一定时间内返回响应)变差。假如我们为了高可用性，所有副本接收到数据请求时立即返回给客户端，而这时又恰好在数据同步的时间窗口中，就会造成不同客户端看到的数据不一致。

* **放弃C保P和A**:比如一对夫妻在异地同时在酒店网站订房间，丈夫看到419房间可以预定，妻子看到419房间已经被人预定
* **放弃A保P和C**：由于每次数据都要同步到所有副本，造成某个用户下单购物等待超过1分钟
* **放弃P保A和C**：放弃P相当于对于无法对机器宕机、机房停电、网络异常的容错，那么就使用单台机器，或者数据分片存放在多台机器没有副本，因为没有数据同步操作数据也只有一副，那么就能很容易满足A和C，但是一旦遇到宕机、网络异常，整个服务将不可用

假设

* N — 数据复制的份数
* W — 更新数据时需要保证写完成的节点数
* R — 读取数据的时候需要读取的节点数

如果W+R>N，写的节点和读的节点重叠，则是强一致性，如果W+R<N则是弱一致性的。

* **BASE理论**

BASE是Basically Available(基本可用)、Soft state(软状态)、Eventually consistent(最终一致性)三个短语的缩写。

* 基本可用：响应上的损失(出现故障情况响应时间可以延长)、功能上的损失(引导到降级页)
* 弱状态：允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许副本直接数据有同步延迟
* 最终一致性：最终一致性强调的是系统中所有的数据副本在经过一段时间的同步后，最终能够达到一个一致的状态。
* 最终一致性的五大变种

根据业务场景，要求的一致性也不一样

* 因果一致性：进程A更新了某个数据通知B，那么进程B之后对该数据访问应该能够访问到A更新后的最新值，如果B进行数据更新的话务必基于进程A更新的最新值

比如zk就可以实现因果一致性，A修改完数据发送给B，B执行sync()，然后得到version，再拿着version去setData

* 读己之所写：进程A更新一个数据项之后，它总能访问到更新过的最新值，而不会看到旧值。比如论坛中，用户可以及时看到被自己编辑过后的帖子也可以看到自己修改后的个人资料，但是用户可以不用实时看到别人修改过的帖子，mongodb可以设置读写primary，也可以设置写primary，读secondary，所以mongodb可以满足这一一致性需求。

zk也满足了读己之所写，在连接不重连的情况下，由于只连一个zk sever只建立一个连接，该zk server commit数据更新返回给client之后，client每次读都不会读到旧值。在连接重连的情况下，由于client每次连新的zk server都会验证zxid(事务id)不能比自己提交过的zxid(事务id)小，所以一定也能看到自己写入的值

* 会话一致性：和“读己之所写”类似，只是“己”变成了“会话”，用户申请一个会话，即使连接变了需要重新建立，但如果还是在一个会话中，那么看到数据不能是之前写入的旧值

zk满足会话一致性

* 单调读一致性：当一个进程从系统中读到一个数据项的值，那么后续任何对该数据项的访问都不该返回比这更旧的值

zk满足单调读

* 单调写一致性：

单调写一致性是指，一个系统需要保证一个进程写操作被顺序的执行

zk满足单调写

### 2. 一致性协议从paxos到zab

为了解决一致性问题，涌现了一大批经典的一致性协议和算法，最著名的就是二阶段提交、三阶段提交、Paxos算法了。

* 2PC

是Two-Phase Commit的缩写，即二阶段提交，目前绝大多数的关系型数据库都是采用二阶段提交协议来完成分布式事务处理的

阶段一：提交事务请求

阶段二：执行事务提交

举个例子：

**Phase 1**

**Prepare:** 组织者(coordinator)打电话给所有参与者(participant) ，同时告知参与者列表。

**Proposal:** 提出周六2pm-5pm举办活动。

**Vote:** participant需vote结果给coordinator：accept or reject。  
**Block:** 如果accept, participant锁住周六2pm-5pm的时间，不再接受其他请求。

**Phase 2**

**Commit:** 如果所有参与者都同意，组织者coodinator通知所有参与者commit, 否则通知abort，participant解除锁定。

优点：原理简单，实现方便

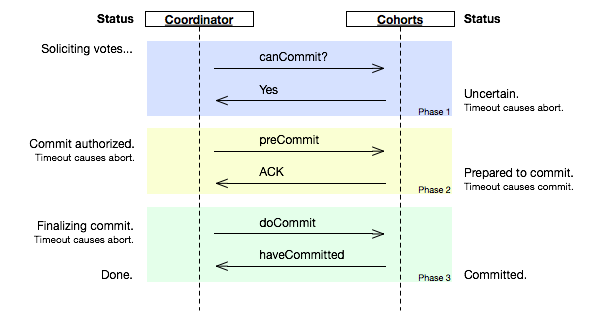
缺点：同步阻塞、单点故障、网络分区等故障无法容错

* 3PC

阶段一：CanCommit

阶段二：PreCommit

阶段三：doCommit



cohorts(participant)收到preCommit之后，如果没收到commit, 默认也执行commit, 即图上的timeout cause commit。

如果coodinator发送了一半preCommit crash, watchdog接管之后通过query, 如果有任一节点收到commit, 或者全部节点收到preCommit, 则可继续commit, 否则abort。

优点：允许发生单点故障后继续达成一致

缺点：网络分区问题，比如preCommit消息发送后突然两个机房断开，这时候coodinator所在机房会abort, 另外剩余replicas机房会commit

* Paxos

Paxos算法是[莱斯利·兰伯特](http://zh.wikipedia.org/wiki/%E8%8E%B1%E6%96%AF%E5%88%A9%C2%B7%E5%85%B0%E4%BC%AF%E7%89%B9)（Leslie Lamport，就是 [LaTeX](http://zh.wikipedia.org/wiki/LaTeX) 中的"La"，此人现在在[微软研究院](http://zh.wikipedia.org/wiki/%E5%BE%AE%E8%BB%9F%E7%A0%94%E7%A9%B6%E9%99%A2)）于1990年提出的一种基于消息传递且具有高度容错特性的一致性算法。

P1a. 每次Paxos实例执行都分配一个编号，编号需要递增，每个replica不接受比当前最大编号小的提案

P2. 一旦一个 value v 被replica通过，那么之后任何再批准的 value 必须是 v，即没有拜占庭将军(Byzantine)问题。拿上面请客的比喻来说，就是一个参与者一旦accept周六2pm-5pm的proposal, 就不能改变主意。以后不管谁来问都是accept这个value。

一个proposal只需要多数派同意即可通过。因此比2PC/3PC更灵活，在一个2f+1个节点的集群中，允许有f个节点不可用

Google Chubby是一个大名鼎鼎的分布式锁服务，从中可以了解到Paxos算法是如何在实际工程中得到应用的

* **quorum算法**

在2PC中，leader需要将数据复制到所有的集群中副本上，有一个副本响应慢(僵死)接下来的写操作就会一直等待，这种等待是无法忍受的。所以就必须寻找一个数值N，使得同步到N个副本即可返回响应，这个N数值就是quorum要满足的投票数量，假如我们设置的很小，比如有5(s1-s5)台机器，我们选择2，s1,s2副本完成复制并对复制请求(create /znode)作出了响应，而此时突然s1，s2与s3，s4，s5网络不可达，由于quorum num等于2，在s1,s2满足了这个quorum条件，形成一个集群，而s3，S4,s5也形成了一个集群，造成两个集群，而s3，S4,s5永远看不到/znode的节点，就形成了数据丢失，导致数据不一致。

所以quorum被设置成大多数，比如集群数量是2N+1，quorum所满足的投票数值是N+1，zk集群可以最大容忍N个server出现故障，超过N个就不能正常提供服务。

ZAB的原子消息广播和投票选举都使用quorum，quorum是保证zk数据一致性的基础。

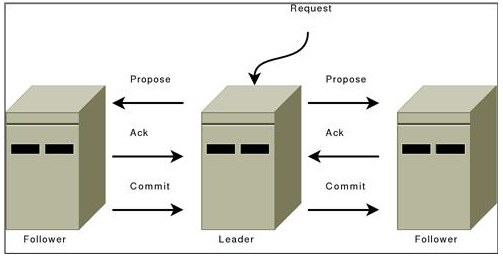
* ZAB

ZooKeeper并不是遵循Paxos协议，而是基于自身设计并优化的一个2 phase commit的协议---ZAB,Zookeeper Atomic Broadcast，它是一种特别为Zookeeper设计的崩溃可恢复的原子消息广播算法。

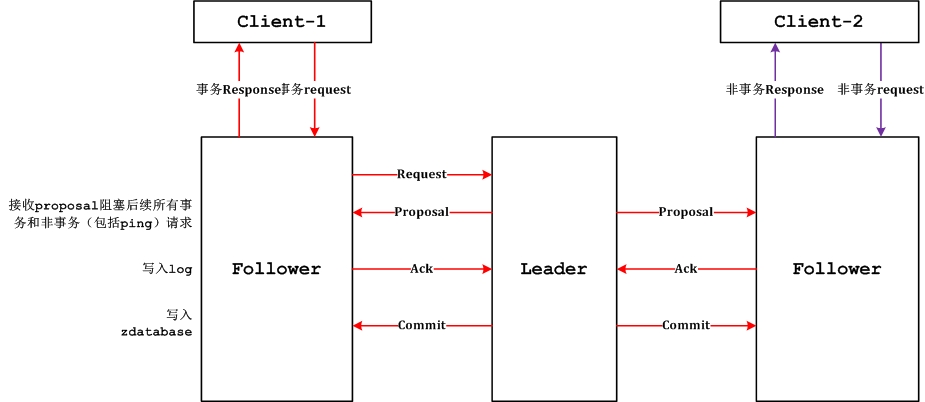
ZAB协议的核心是定义了对于那些改变zk服务器数据状态的事务请求的处理方式。

ZAB包括三个阶段模式：**消息广播--崩溃恢复---数据同步**，无论哪个节点在这三个阶段的循环重复

* **消息广播**：所有事务请求必须是由一个全局唯一的服务器来协调处理，这样的服务器被称为Leader服务器，而余下的其它服务器则称为Follower服务器。Leader服务器负责将一个客户端事务转成一个事务Proposal，并将该proposal分发给集群中所有的Follower服务器。之后leaer服务器需要等待所有follower服务器的反馈，一旦超过半数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分发commit消息，要求其将前一个Proposal进行提交



上图表示了在确定leader和follower情况下消息广播的事务处理流程，client会只会连接一个zookeeper服务器，在失去连接重连的情况下才会尝试连接ip列表的下一个server，对于非事务请求，zk server直接自己处理读取自己的zookeeper datatree数据或者对于ping请求返回ping ack，但是对于事务请求会转发到leader



蓝线是非事务请求，红线是事务请求。首先应该知道的是任何client与zk-server只保持一个连接，任何follower和leader之间也仅仅只保持一个连接；在此前提下zk所提供的顺序一致性才能有保证，在上图中，当一个proposal到达发起事务的follower的时候，会阻塞后续接到的所有事务和非事务请求，直到收到对应的commit消息，于是保证了读己一致性、会话一致性

* 崩溃恢复

ZAB对2PC主要的改进就是崩溃恢复，使得zk能应对各种各样异常问题(宕机、网络异常)，以下是我总结的集中崩溃恢复的几种情况。

①leader故障，leader由于宕机、网络异常发生的故障

在此情况下，所有follower由于失去leader的连接或者在synclimit\*ticktime时间范围内没有读到来自leader的ping请求，会从follower状态转为looking状态，同时停止对外服务进入投票选举阶段选出一个新leader，参与选举的server 成员，向各个节点发送自己的(epoch,zxid)pair，如果对方比自己的大(先比较epoch,再比较zxid，再比较myid)，那么就更新投票继续发送，直到选择一个大多数都赞成的(epoch,zxid) ，如果选择不出来就一直无限投票持续下去。

在leader崩溃之前，只要它已经发出proposal（leader发完proposal才写入自己的事务日志）并且该proposal已经被写入事务日志，那么在下次选举结束后被同步到各个follower，不会丢失该事务消息

②follower故障follower故障(宕机、网络异常、僵死)。出现follower故障后，会导致向它发送ping请求和proposal请求的leader，在synclimit\*ticktime没有收到proposal的ack(proposal已发出的情况下)或者没有在synclimit\*ticktime时间段内收到返回的PING请求，那么leader将follower移除forwarding follower列表，停止向它发送proposal。

当在synclimit\*ticktime时间范围内leader没有收到足够quorum个数的PING请求，那么leader就变成LOOKING角色，重新进行leader选举。

如果leader中有任何一个proposal没有在synclimit\*ticktime时间内获得quorum个数的ack(比如某个follower刷磁盘很慢)，但是PING心跳是正常的，这个时候该follower和leader会阻塞所有来自client的读写请求，造成服务不可用，由于client向server发送的ping(间隔1/3个sessiontimeout发一次)也被阻塞，造成client主动断开连接，所有请求都将收到

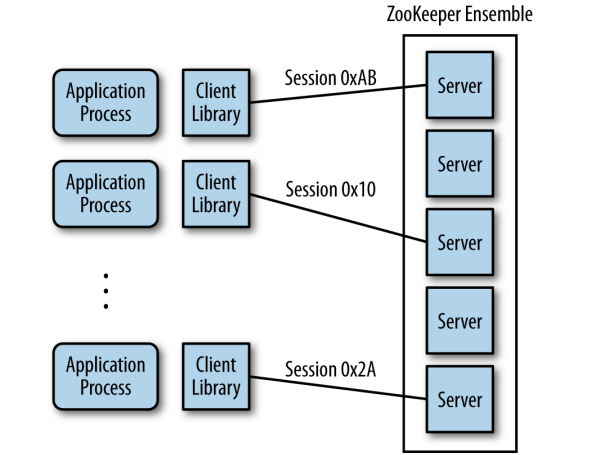
org.apache.zookeeper.KeeperException$ConnectionLossException: KeeperErrorCode，由此可见等待ack事务的数量应该监控起来。

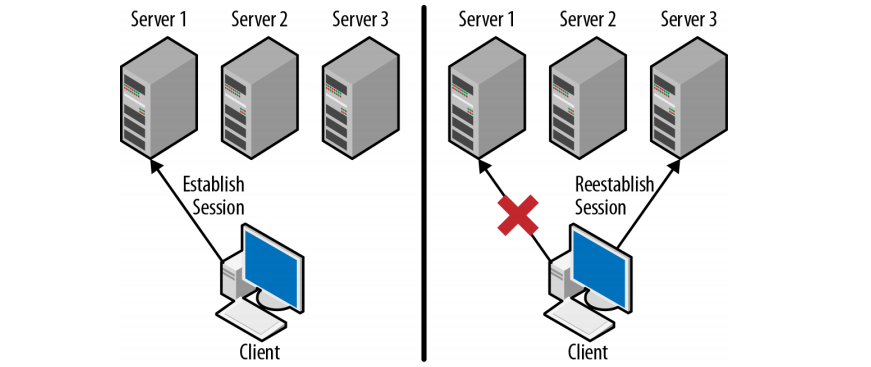
quorum个数的zk server运行正常是整个zookeeper集群运行正常对外提供正常稳定服务的保证，超过quorum数量的server出现故障（宕机、僵死、超时响应、网络异常）都会导致整个zookeeper集群无法继续提供服务。

③刚才我们说的是zk server的故障对集群和事务请求的影响，这次是看zk server异常对client的影响。当zk server出现故障后，client会重新对ip列表的下个server发送建立连接的请求，如果之前申请过会话，那么重连后还是建立在上一次会话中(通过发送sessionid)。

每次zk client的事务操作都会得到一个响应，表示成功或失败，一旦在等待响应的过程中，出现了连接丢失等异常，zk client是无法知道上次事务请求是否成功，只能下次建立连接时进行验证(curator通过withProtection来避免在创建节点时出现这种情况导致创建多个重复节点)。

由于zk不是强一致性的，各个zk server的数据在同一时刻不是同一视图，这就需要zk client在重连时发送最后它获得zxid来验证自己所连的zk server不能比上次连的zk server数据旧，从而保证会话内的一致性。



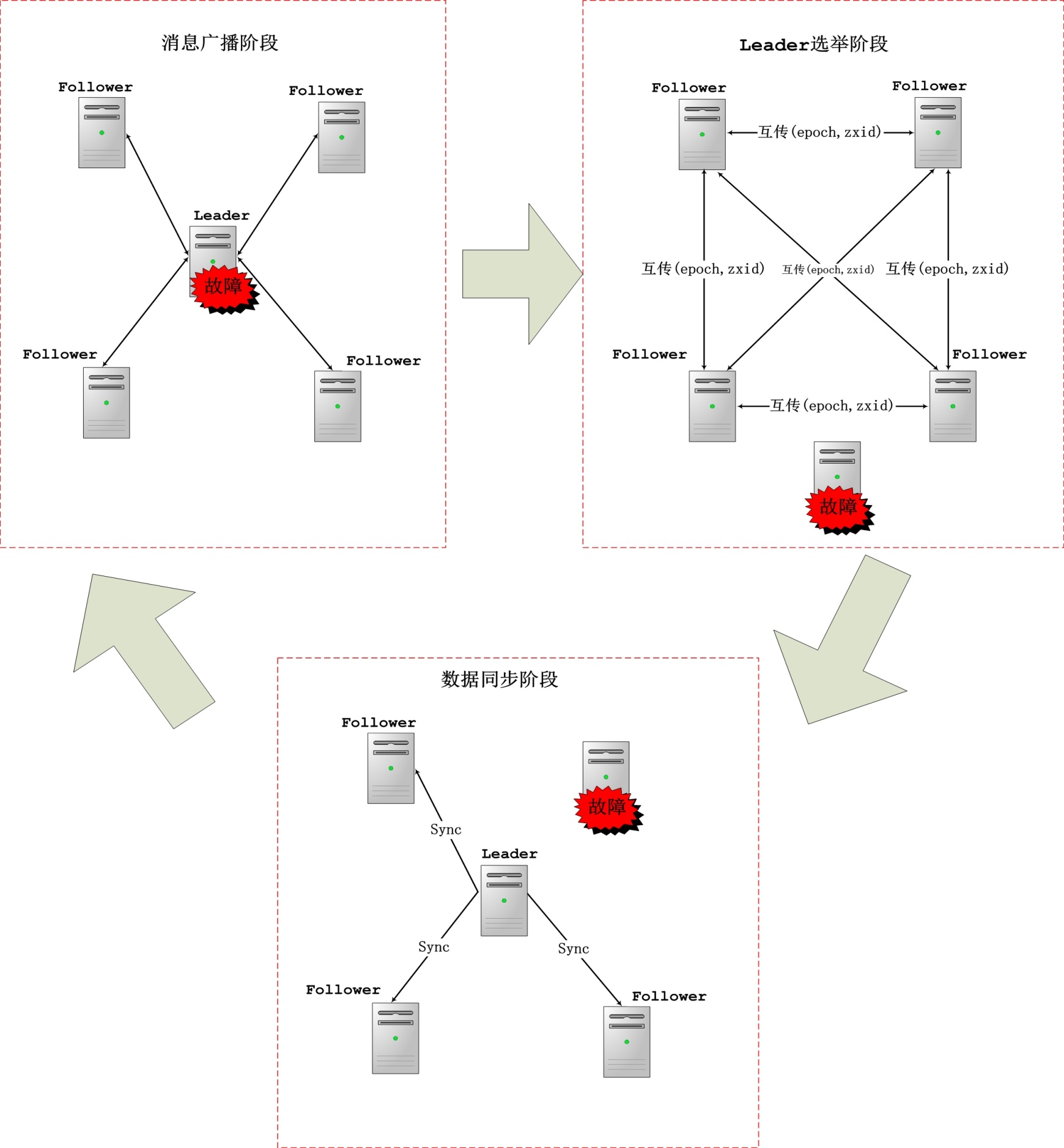


上图中client用同样的sessionid重连时会带上最后一次收到的zxid，所连的server的last zxid >= zxid,才能建立session会话

* 数据同步

当leader被选举出来后或者有新的follower加入quorum集群后，leader会向follower同步数据，当数据都已经达到follower之后，leader最后会发送一个UPTODATE的消息给follower代表数据已经完成同步，follower启动zookeeper，开始对外提供服务，但此时follower还没有将数据完全应用到自己zookeeper database里面，所以在数据同步阶段，client会看到旧数据。

三阶段转换图：



三阶段点评：三阶段是循环转换的，集群刚启动时是在leader选举阶段，如上图，在leader选举阶段即便有一台server宕机了，但是如果达到选举leader的条件，必须至少3个server的投票达到一致，集群的总数还是5而不是4.leader选举的具体细节后续会详细介绍。

### 3. Zookeeper特性

* **zookeeper定义**：ZooKeeper is a centralized service for

maintaining configuration information,

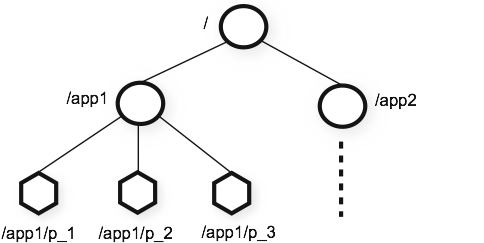
naming,

providing distributed synchronization,

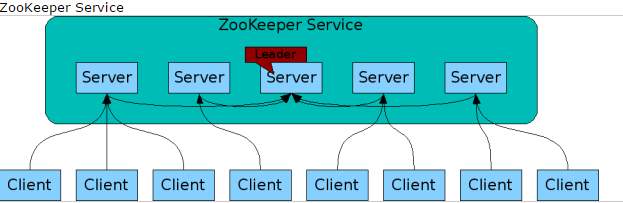
and providing group services.

* **zookeeper用途**：可以用zookeeper进行发布/订阅，Mater选举，分布式协调/通知，分布式锁，分布式队列，命名服务，集群管理
* **Zookeeper设计目标**

1、简单的数据模型



2、可以构建集群



3、**顺序访问**

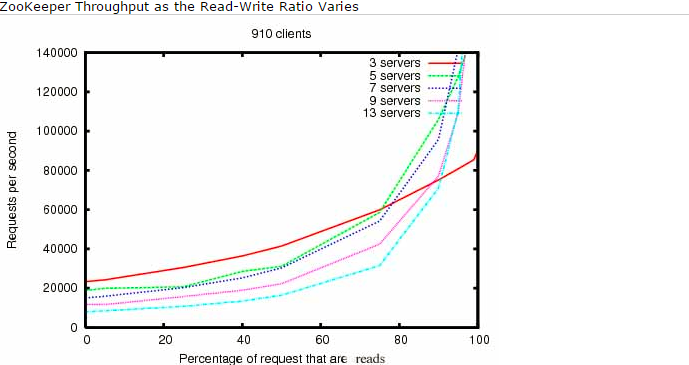
zk对每个事务请求都会分配一个zxid，基于此，可以用zk来实现高级的同步原语比如分布式锁、分布式队列、barrier、发布订阅

4、**高性能**

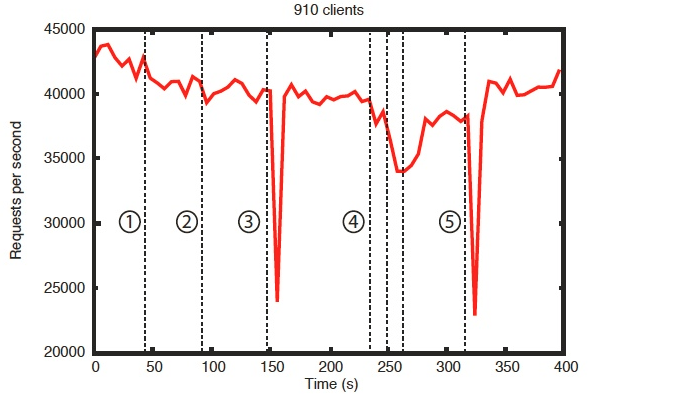
由于zk所有数据都是放在内存中，读请求也不需要太多cpu操作，所以响应非常快，但是写性能不高，读比写要快十倍。

因为写操作需要发起事务，事务发起方和leader在收到该事务对应commit消息之前会阻塞读请求，在写操作过多的情况下，读请求会受到影响，以下是官方给出的性能测试图。

由于所有数据都是放入内存中提供访问(快照和事务日志只是提供数据恢复)，所以zk不适合大数据量的存储。



故障恢复性能测试，以下是官网给出的在故障恢复情况下的性能情况,和上图同样的压力7台机器，30%的写操作



横轴是时间线(单位秒),纵轴是qps，①虚线是一个follower从故障到恢复的时刻

②虚线是另外一个follower从故障到恢复的时刻

③虚线是将leader出故障的时刻

④虚线是将两个follower从出故障到恢复的时刻

⑤虚线是将另外一个leader出故障的时刻

从图中可以看出leader出现故障的影响比follower出现故障的影响要大，因为leader故障需要经历投票选举的阶段，官方指出，Zookeeper花费了200ms做leader选举(在出现某几台机器网络故障时，ticktime越大，选举的时间窗口越大，leader启动同步服务端口如果慢的话也会导致重新选举拉长选举时间窗口)。

follower从故障中恢复会提升整个Zookeeper集群的处理能力。

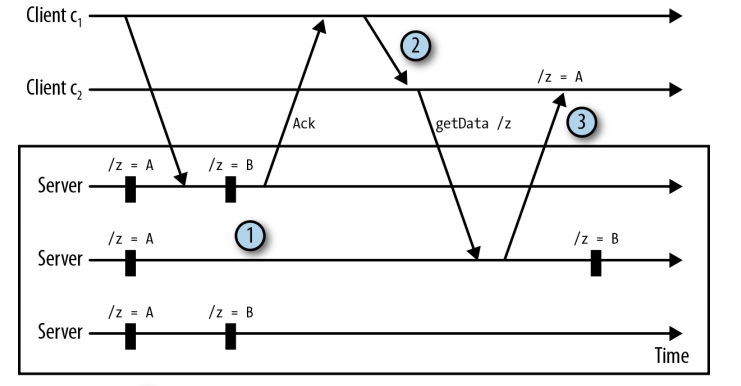
* **Zookeeper一致性保证**
* **顺序一致性(Seuential Consistency)**

①从一个client发出的更新请求会被顺序的应用到Zookeeper中去。基于client-server单一连接，follower-leader单一连接，follower、leader内部线程之间用FIFO队列传递数据；才能实现所有的更新操作都是顺序执行的，每次事务请求都会申请一个自增的zxid(事务id)。

②zk可以保证一致性的五大变种即读己之所写，会话一致性，单调读一致性，单调写一致性，因果一致性(需要调用sync方法)。

zk唯一不能保证的一致性问题是：两个客户端同一时刻看到的数据不一定是一样的。

但是通过sync方法可以保证因果一致性



1)client1发起setData请求将/z数据改成"B",并得到ACK，跟新成功应用到server1,和server3中，但是由于proposal还是网络传输没用应用到server2

2)这时client1发消息告知client2，/z已经被我更新为"B"

3)此时client2向server2发起读请求，但是proposal还在数据传输，未到达server2，与server2向client2返回旧数据"A"

为了避免这一情况client2只需要在getData之前发送一次sync()请求，sync请求会转发到leader，保证sync之前的proposal都会commit执行才返回给client2，所以能保证client2看到最新数据。

* **原子性(Atomicity)**

所有事务请求的处理结果在整个集群中所有机器的应用情况是一致的，不会出现某些部分机器应用了该事务，而其余机器没有应用该事务。并且可以保证的是同一时刻quorum数量的机器都应用集群最后一个zxid

对于发出proposal的leader写入了该事务日志但是突然网络中断，其余的机器未能收到proposal，其余机器选举了新的leader，更新了epoch，之前的老leader虽然zxid最大，但是epoch低，在老leader重新加入集群中，会删除最后一次未能成功propose到其余机器的事务

* **单一视图(Single System Image)**

所有机器存储的都是同一份数据，client不管连接到哪台zk，看到的都是同一份数据(虽然数据不保证完全一致)

* **可靠性(Reliability)**

所有事务请求只要client收到成功code或者非事务请求查询到了某次更新，那么这次更新就肯定成功应用到集群中了。无论后续遇到任何崩溃，都不会回滚到更新之前的数据状态

* **实时性(Timeliness)**

所有client读到的Zookeeper数据保证在一定时间内是最新的(synclimit\*ticktime)。任何数据更新要么在这一时间范围内被看到，要么超过时间范围client 检测到服务不可用(所连的zk server)

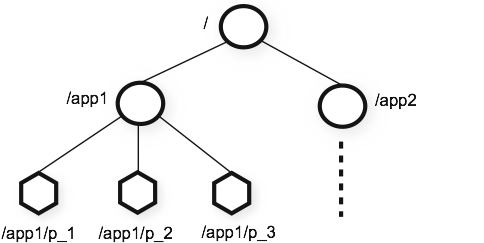
### 4. Zookeeper概念

* **Zookeeper数据模型(Zookeeper Data Model)**

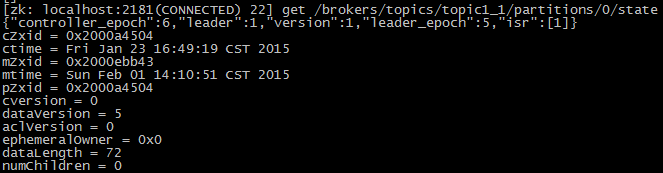
Zookeeper的数据模型是一个树形结构(Znode Tree),znode是该数据模型中的数据单元，/是整棵树的root节点，每个znode上都会保持自己的数据并持有一份stat属性信息，分为两种类型临时节点(Ephemeral Nodes)和永久节点(Persistent Nodes)。

每种节点上都可以设置一个特殊属性--SEQUENTIAL。znode名字是有约束的，尽量不要用非字符串和包含.的字符串作为path名字，/zookeeper是zk特有节点，不能被用户创建。

* **临时节点**:znode生命周期与客户端会话绑定，一旦客户端会话失效，那么该客户端会话期间创建所有临时节点都将被删除
* **永久节点**：除非主动进行针对该znode的删除操作，无论客户端会话是否失效，都会删掉该znode
* **顺序属性**(SEQUENTIAL)：一旦创建节点时带上该属性，那么该节点后缀都会加上一个由父节点维度下的自增整形数字，这样能有效防止两个client创建同一个名字的znode而导致其中一个client收到创建错误的响应。



* **zk状态信息(Zookeeper Stat Structure)**

****

|  |  |
| --- | --- |
| cxid | 该数据节点被创建时的事务id |
| ctime | 该节点被创建时的时间 |
| mzxid | 节点最后一次被更新时的事务id |
| mtime | 节点最后一次被更新的时间 |
| pzxid | 该节点子列表最后一次被修改时的事务id，子节点内容的变化不会影响pzxid，从上图我们看出，由于没有子节点，所以pzxid==cxid |
| cversion | 当前数据节点子节点变更的版本号，每变一次数值+1 |
| aversion | 当前数据节点ACL变更版本号，每变一次数值+1 |
| dataversion | 当前数据内容变更的版本号，每变一次数值+1 |
| emphemeralOwner | 创建该临时节点的会话sessionId，如果是永久节点那么值为0 |
| datalength | 数据内容的长度 |
| numchildren | 当前节点的子节点个数 |

* **Version版本**

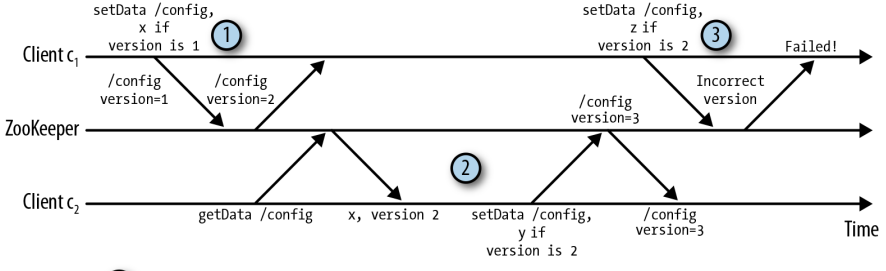
每个节点都有一个版本号，每次数据更新都会加1.基于此，可以实现各个不同客户端CAS原子更新。

像redis、memcached都支持CAS操作来实现数据原子操作，数据库基于CAS实现乐观锁，java concurret 包基于CAS实现高性能并发操作。

zk的API中有两种操作支持基于版本做CAS操作----setData和delete操作。如果传递版本-1，则忽略原子操作强制进行更新和修改。

如果delete传版本-1，对于具有ACL权限控制的节点可以强制删除，对于具有ACL权限控制的子节点不能强制删除，只能给自己添加权限的才能删除

setData传版本-1，对于具有ACL权限控制的节点无法更新



1)client1 更新/config成功，version变成2

2)client2 获取/config的version，并且

setData(/config,data="",version=2),version变成3，并返回给client2

3)client1 传递version2来更新/config,结果返回错误，因为版本不对

* **ZXID 事务id**

每次对Zookeeper状态的改变都会生成一个自增id叫做zxid来表示该次更新的事务id，高32位代表leader的epoch，低32位是一个自增id。zxid是uniqe的，如果zxid1>zxid2,那么zxid1肯定发生在zxid2之后。

事务操作包括(要申请zxid)

**create**

**delete**

**setData**

**multi**

**setACL**

**createSession**

**closeSession**

非事务操作包括

**sync(唯一一个需要转发给leader的非事务操作)**

**exists**

**getData**

**getACL**

**getChildren**

**getChildren2**

**ping**

**auth(每次重连时发送)**

**setWatchs(每次重连时发送)**

事务操作不管成功与否，都会申请一个zxid，都会记录到事务日志文件中。所有事务ID都是leader自增分配的。

<https://issues.apache.org/jira/browse/ZOOKEEPER-1277>

由于zxid低32位是自增id，假如每秒写入500次，3个月后，zxid将使高32位的epoch发生rollover，这是不允许的，于是zk修复了这个bug，leader在发现zxid低位达到0xffffffffL时就shutdown强制做一次leader选举。

* **EPOCH ---Leader周期**

epoch中文是纪元世纪的意思，实际上很形象，每次选举出一个新leader，就是开启一个新纪元，直到下次遇到故障重新进行leader选举，这段时间所有的事务id的高32位都是该epoch。下次leader选举成功后，epoch加1形成新的epoch。

epoch在各个分布式系统都有用到，用于做leader选举（先比较epoch）以及fencing，比如kafka中基于epoch做fencing(防止集群间过期消息造成影响)。

Zookeeper由于只保持一个连接，并且只有一个线程处理集群间的消息传递，故不需要使用epoch做集群间的fencing，只在leader选举时用。

* **Watcher--数据变更通知**

|  |  |  |  |
| --- | --- | --- | --- |
| **KeeperState** | **EventType** | **触发条件** | **说明** |
| **SyncConnected** | **None(-1)** | **客户端与服务端建立会话成功** | **客户端与服务器处于连接状态** |
| **NodeCreated(1)** | **Watcher监听的对应数据节点被创建** |
| **NodeDelete(2)** | **Watcher监听的对应数据节点被删除** |
| **NodeDataChanged(3)** | **Watcher监听的对应数据节点数据内容发生变更** |
| **NodeChildrenChanged**  **(4)** | **Watcher监听的对应数据节点的子节点列表发生变化** |
| **Disconnected** | **None(-1)** | **客户端与服务器断开连接** | **客户端未及时发送ping请求或者在**  **读超时的时间内未能接收到来自server的响应那么就关闭连接(不会关闭session)**  **所有pending request收到**  **SessionTimeout**  **Exception响应** |
| **Expired** | **None(-1)** | **会话超时** | **收到SessionExpired**  **Exception** |
| **AuthFailed** | **None(-1)** | **使用错误的schema**  **或者SASL权限检查失败** | **通常也会收到**  **AuthFailed**  **Exception** |

使用默认watcher，建立会话成功、Disconnected、Expired、Auth\_failed通知都会收到，其中遇到Expired和AUTH\_FAILED，会话会失效是不可恢复的。

只能重新建立新session，client使用新的Zookeeper实例

**watch存在放client端，server端只存放哪些client要对哪些path做哪种类型watch(datawatch和childrenwatch)。**

**getData注册节点删除、内容更新**

**exsit 注册节点创建、删除、内容更新**

**getChildren注册节点删除、子列表变化**

**watcheEvent包括KeeperState，eventTpye，path**

**watch有如下特性**

1、一次触发

client每次设置watch，收到通知后必须再次注册watch才能收到下次事件通知，比如

getData("/path",xxWatcher),收到通知以后需要再次注册watch

getData("/path",xxWatcher),可以在每次通知时注册上watch并获取一次最新数据，所以不会漏掉任何一次更新。

2、顺序执行

所有的watch的处理都在event线程顺序的执行

3、轻量

网络开销和内存开销很廉价

watch导致的羊群效应

由于多个client都会watch同一个path。当该path遇到事件发生时，所有client都将收到通知，并同时发送getData或getChildren等读请求，导致server端压力变大，集群数量没有太大的情况是不会出现,避免羊群效应的方法是不让所有client监听同一个path，比如分布式锁每次抢占锁创建一个临时有序znode，每个client只需要监听比自己小的那个znode，如果比自己小的那个znode得到释放，那么就获取锁

**watch的触发机制**

1、watcher对象保存在client中，注册器保存了多个不同事件类型的map，map中kv结构是(path->wacher)

2、client所连的server会保存(path--事件类型)的映射关系，绑定到连接对象。

3、当client进行重连时，由于client有watch的缓存，在发送connectRequest请求前会发送setWatch请求，将watch注册到server，在server端保存映射关系

与此类似的是auth

**watchEvent不一定是server发起的**

watchEvent除了notifcation，还有client自己也可以发起，比如disconnect事件是在client发现server网络通信超时。

还有auth\_failed等

* **Session--会话**

zk client在与zk server建立连接以后，会发送一个connectRequest请求来建立一个会话，用初始的lastzxid，sessiontimeout去与zk server协商，zk server返回一个sessionTimeout、sessionId和sessionPasswd，以后连接断开client都会带上sessionId和sessionPasswd来重新激活会话，发ping请求能让会话延后过期。server将socket连接与sessionId绑定，就无需进行每次传递sessionId了。

sessionTimeout时间范围是(2->40)\*tickTime,最大是40倍的ticktime时间，最小是2倍的ticktime时间，所以如果用户传的sessiontimeout时间如果在这个范围就不用变，否则使用最大或者最小值。

* client会定期向zk server发ping请求，如果在2/3个sessionTimeout时间内没有收到响应，那么就发布Connecting\_loss事件，然后进行在iplist进行随机重连（使用之前申请的sessionId和sessionPasswd）。client主动发起的PING请求不仅可以保持会话存活，也可以及时侦测到zk server的是否正常，如果不正常就选择其他server进行重连.
* 有三种情况会导致会话永久关闭:

1)client主动发起closeSession

2)AUTH\_FAILED一般在addauth遇到错误schema时出现

3)最常见的就是Session\_Expired,server端发起的。注意session都存放在leader服务器上，过期通知过期session清理都是leader进行触发

* SessionId的创建

64位，前8位是创建sessionId的机器id，即myid文件里存放的机器id，低56位是部分时间戳。事务id只有leader可以生成，而sessionId是由创建请求接收server创建的。

比如ephemeralOwner = 0x14d298c7cad000c

转换成二进制是

101001101001010011000110001111100101011010000000000001100

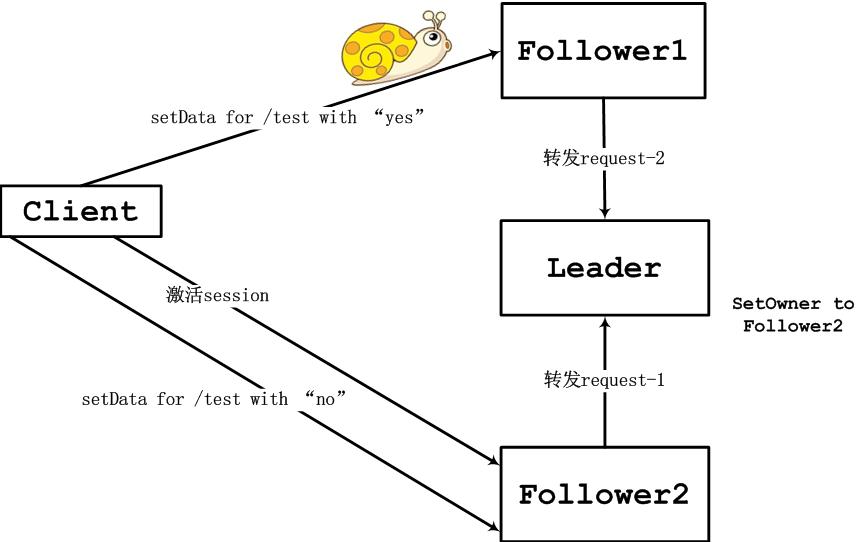
低位移除时间相关的56位，得到1,1就是机器id

* createSession操作根据是否传sessionId可以分为首次创建session和激活session两种，每种都会重新设置该session的owner即follower到leader的socket连接，而其它事务操作一经发现是不同的follower转发给leader的请求，那么该事务就会得到SessionMovedException的异常。这是zk做的一种fencing策略。

首先应该知道client socket的readTimeout是2/3个sessionTimeout,连超时是sessionTimeout除以iplist size

readTimeout = negotiatedSessionTimeout \* 2 / 3;

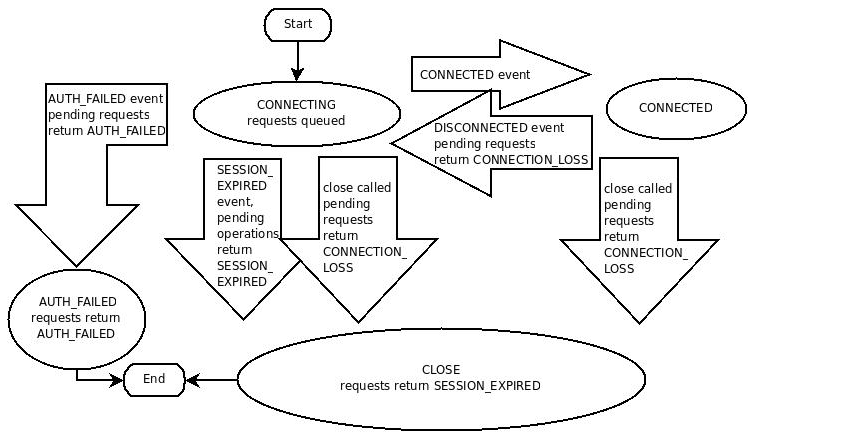
connectTimeout = negotiatedSessionTimeout / hostProvider.size();



Client首先向Follower1发送setData请求(request-2)，但是网络传输很慢，导致client读超时，然后client转向Follower2建立连接、激活session、并发送setData 为no的请求，Follower2将request-1转发到leader。修改完成后/test的数据变成no，但是假如这时request-2请求到达了Follower1这时，Follower1又转发request-2到leader，导致/test数据被覆盖变成了旧数据yes。

为了避免这个问题，zk在3.2.1中对session增加一个owner属性，前面我们讨论过，session和client连接绑定，每个Follower向leader转发事务请求都会带上client端的sessionId。如果遇到createSession请求(激活也发这个)，那么更改Owner，后续的同一个sessionId请求如果发现owner已经不是该Follower了，request对应的Response状态码会变成error，异常是SessionMovedException

以下是Session状态转换图



session涉及的操作

|  |  |  |
| --- | --- | --- |
| 类型 | 触发条件 | 说明 |
| createSession | Zookeeper client初始化时， | 建立连接成功后就做申请创建session，协商session相关属性sessionTimeout、sessionId等。  这是一次事务请求 |
|  | 重连的时候 | Client会带上之前缓存的  sessionId和sessionPasswd去激活session。  Follower会向leader发送激活请求，leader作出激活请求的响应  当发送激活请求session过期时，  Leader会验证session是否过期，之后Follower接收到响应会将sessionTimeout设置为0，client会触发sessionExpired事件  这是非事务请求 |
| closeSession | Leader发现session失效后，会提交closeSession请求，Follower会先发一个closeSession的响应然后关闭client连接 |  |
|  | Zookeeper client 关闭的时候可以发送closeSession请求，不用等到超时 |  |
| 收到KeeperState.Expired | 所有转发到leader的请求都有可能收到session Expired的响应。 |  |
|  | 在进行重连时候，有可能session已经过期，这时sever会将sessionTimeout时间设置为0，client判断sessionTimeout<=0,就发布SessionExpired事件。 |  |

* **Zookeeper API**
* **Creat /path data**

String create(**final** String path, **byte** data[], List<ACL> acl,

CreateMode createMode)

**Void**create(**final** String path, **byte** data[], List<ACL> acl,

CreateMode createMode, StringCallback cb, Object ctx)

ACL是给节点添加权限控制，createMode包括四种，临时有序临时无须永久有序永久无序

* **Delete /path**

void delete(final String path, int version)

delete(final String path, int version, VoidCallback cb, Object ctx)

* **Exists /path**

Stat exists(final String path, Watcher watcher)

**void** exists(**final** String path, Watcher watcher,

StatCallback cb, Object ctx)

Stat exists(String path, boolean watch)

Watch设置为true使用默认的Watcher

* **SetData /path data**

Stat setData(final String path, byte data[], int version)

Stat setData(fin**al** String path, **byte** data[], **int** version,

StatCallback cb, Object ctx)

* getData /path

public byte[] getData(final String path, Watcher watcher, Stat stat)

byte[] getData(String path, boolean watch, Stat stat)

* getChildren /path

List<String> getChildren(final String path, Watcher watcher)

List<String> getChildren(String path, boolean watch)

* setACl /path acl

设置znode节点的权限

* sync

同步操作，保证数据是最新的

* close

closeSession操作

* multi

zk的事务操作，所有操作批量执行

* **Zookeeper角色**

Zookeeper有三种角色：分别是Leader、Follower、OBSERVER,没有leader的时候，所有zk服务器都处于LOOKING状态，不处于任何角色。一旦leader选举成功，各个角色才各自归位。

Leader：做事务请求的验证，发送PROPOSAL,收集ACK,发PING维护Follower和Observer的存活列表，维护session的生命周期，分配事务id

Follower：接收client的读写操作，将写操作转发到leader，接收Proposal，参与proposal投票对leader作出ACK，自己处理读操作，注册watch，触发通知。响应leader发送的PING请求，检测leader的存活状态

Observer：不参与任何投票，leader不会向Observer发proposal，也不参与leader的选举，它接收读写请求，并转发写请求给leader，它的存在并不能增加宕机容忍性，而是提升集群的非事务处理能力。响应leader发送的PING请求，检测leader的存活状态

### 5. Zookeeper配置

|  |  |  |
| --- | --- | --- |
| 名称 | 说明 | 备注 |
| ticktime | Zk的计量时间的基本单位时间，单位是毫秒。3.4.6默认是2000 | 比如session超时的时间是，2-20倍的ticktime，具体多少倍由client传递一个初始值和server进行协商；除此以外Follower与server之间的心跳是synclimit个ticktime时间，client与server，Follower与leader之间的连超时、读超时、发ping间隔(f-l是1/2个)都是以ticktime为单位的 |
| dataDir | 存储快照文件目录 | 如果没有配置dataLogDir  那么事务日志也会存放在dataDir |
| clientPort | 一般2181 | 提供client读写服务的端口 |
| dataLogDir | 用于配置Zookeeper服务器存储事务日志文件的目录 | Zk每次写操作都会进行事务日志文件的写入(只在恢复时读),所以选择高性能磁盘至关重要，一定不能和存快照的dataDir放在一个磁盘块上，因为快照生成会极大的使用io资源，从而影响到事务日志文件的写入 |
| initLimit | 数据同步阶段leader允许Follower多少个ticktime时间完成初始化过程。默认10 | 根据集群规模相应增大这个值 |
| syncLimit | 默认5，用于配置leader和Follower直接的心跳检测最多忍受多少个ticktime时间得不到响应；以及最多忍受多少次ticktime得不到proposal ACK的响应 | 如果部署的网络环境比较差，可以调大这个数 |
| snapCount | 默认100000，  Logcount=snapCount/2+  random(snapcount/2)  配置写入logcount个事务日志重新生成一个快照，生成快照时下一次事务到来时，事务日志也会重新创建一个。 | 之所以用随机数为了避免所有  Zk server同时执行快照生成操作导致造成集群大规模网络延迟，snapcount要和下面的preAllocaSize对应设置，由于事务日志是预分配大小的，不够的时候会自动扩容，为了避免每次扩容，将snapcount\*平均每个事务大小稍微  小于preAllocSize是最佳。 |
| preAllocSize | 单位KB,默认是65535，用于配置Zookeeper事务日志文件预分配的磁盘大小默认64M. | 预分配可以省去扩容的开销。和snapcount一起使用，比如我们设置snapcount是500，最大一个事务是1K，那么我们将preAllocSize设置成500 |
| minSessionTimeout | 默认2倍ticktime | 如果客户端设置的超时时间不在该范围内，那么就会被强制设置为最大值或者最小值 |
| maxSessionTimeout | 默认20倍ticktime |
| maxClientCnxns | 单台ip的client与zk server所能建立的最大连接数 | 不会限制client总数，仅仅限制单台所建立的连接，一个Zookeeper实例只建立一个连接到zk server |
| jute.maxbuffer | 用于配置单个znode节点所能存储的最大数据量，默认是  4096 \* 1024，4M | 系统属性配置，如果想改这个默认值，client和server的系统配置都要增加这个属性 |
| server.id=host:port:port | id即myid文件存放的serverId  host就是服务器ip  第一个port是Follower和  Leader进行运行时通信和数据同步所用的端口，第二个是  进行leader选举过程中使用的投票端口  比如三台  例如  server.1=192.168.137.10:2888:3888  server.2=192.168.137.11:2888:3888  server.3=192.168.137.12:2888:3888 | 该配置指定了集群机器数量，比如左边例子中配置了3台，那么quorum数量就是2，超过1台机器出现故障，整个服务将不可用 |
| forceSync | 系统属性(zookeeper.forceSync)，默认yes  配置zk服务器是否在事务提交的时候，将日志同步强制刷到磁盘。  默认情况下，当事务不繁忙的时候(poll为null)，会将已经写到到filechannel事务记录批量刷入磁盘。如果设置为false，那么将不调用force方法，即不能保证内存数据刷入磁盘。 | 如果为no，那么  不能保证每次提交事务都写入磁盘（系统缓存相关） |
| GlobalOutstandingLimit | 系统属性(zookeeper.globalOutstandingLimit)  用于配置Zookeeper服务器最大请求堆积数量。防止资源耗尽限制请求数，默认是1000 |  |
| leaderServes | 系统属性(zookeeper.leaderServes)  默认yes情况下Leader可以接收和处理来自client的所有读写请求，但是为了让leader专注于分布式协调，可以设置为no |  |
| CntTimeout | 系统属性(zookeeper.cnxTimeout)  默认5000，单位毫秒，该参数用于配置在leader选举时，各个服务建立连接的超时时间 |  |
| autopurge.snapRetainCount | 默认3，  Zoo.cnf配置属性  用于配置自动清理时候需要保留的快照数据文件数量和对应的事务日志文件。 |  |
| Autopurge.purgeInterval | 默认0，zoo.cnf配置  配置该属性大于0，即开启了一个线程定时清理日志文件 |  |

后续例子中，我们以3888作为选举端口，2888作为数据同步+消息广播端口，2181作为对外服务端口

### 6. Zookeeper ACLs

ZK的ACL由三部分组成：权限模式(Schema)、授权对象(ID)和权限(Permission)

通常使用”schema:id:permission”来标识一个有效的ACL信息

* 权限模式

|  |  |  |
| --- | --- | --- |
| 模式 | 说明 | 授权ID例子 |
| ip | 比如ip:192.168.0.1/24表示对  192.168.0.\*这个ip段做控制 |  |
| Digest | 最常用的权限控制，类似于  Username:password  便于区分不同应用的权限控制,  Zk会对其先经过SHA-A编码再BASE-64编码。通过  DigestAuthenticationProvider.  generateDigest(String) | “username:BASE64(SHA-1(username:password))”  比如test:test  编码后是  test:  V28q/NynI4JI3Rk54h0r8O5kMug= |
| World | 开放的权限控制，对所有用户开放  只有一个id即”world:anynoe” | 只有一个anyone |
| Super | 超级用户的意思，一种特殊的Digest模式，超级用户可以对zk任意znode做任何操作 | 在zk启动的时候  添加如下系统属性  -Dzookeeper.DigestAuthenticationProvider.superDigest=test:  V28q/NynI4JI3Rk54h0r8O5kMug= |
| auth | 它不需要id, 只要是通过authentication的user都有权限（zookeeper支持通过kerberos来进行authencation, 也支持username/password形式的authentication) |  |

* 权限
* CREATE(C):允许授权对象在该数据节点下创建子节点
* DELETE(D)：允许授权对象删除该数据节点的子节点，也可以删除该节点
* READ(R)：允许读取该节点数据或者子节点列表
* WRITE(W)：允许对数据节点进行更新操作
* ADMIN(A)：允许对该节点ACL的设置操作
* 使用方式：通过addauth为自己添加权限，通过create或者setAcl方法对path设置权限
* Java API内置权限
* OPEN\_ACL\_UNSAFE

所有用户拥有该节点所有权限

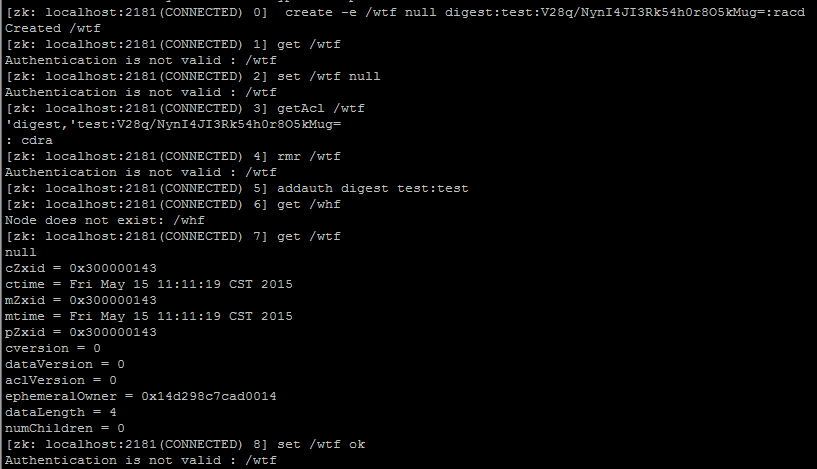
* READ\_ACL\_UNSAFE

所有用户只有读权限

* CREATOR\_ALL\_ACL

创建者有所有权限

* 例子



我们一开始创建了一个znode /test7,并控制权限在digest:test:test权限下有racd权限唯独没有w权限。

紧接着我们更新、读取、删除操作均不行

然后我们给自己添加权限

之后可以读取和删除，但是不能修改数据，原因是我们没有对digest：test:test添加w权限

# 协议篇

### 协议说明

* Client<-->ZkServer端口2181

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 说明 | Request | Response |
| ConnectRequest | 建立连接后  请求session  创建协商  sessionTimeout  sessionPasswd  或者在连接丢失时  重新激活session | ConnectRequest(lastZxidSeen,  sessiontimeout,  sessionId,passwd)  说明：无requestHeadre  第一次时lastZxidSeen为0  Sessiontimeout是参数指定的  sessionId、passwd为空  之后会将上面的信息保存下来激活时传递。  LastZxidSeen用于判断server的数据  是否是最新的，sessionId用于激活session，passwd用于验证是否伪造的请求。每次激活都会在leader端重新绑定该session与owner(follower) | ConnectResponse(protocolVersion  timeout,sessionId,passwd)  说明：无replyHeader |
| ping | Client定期向  Server发ping 请求，Follower保存到本地等到发给leader ping  请求的时候就会  将session map的快照发给leader，leader  让这些session renew，延长过期时间，保持活性 | RequestHeader(xid=-2,type=11)  type=11代表ping请求。  xid用于client验证顺序一致性的，假如传回server端传回的xid与pending队列的第一个请求xid对不上就抛错丢弃  说明：server收到ping，将sessionId放入sessionMap  无请求体 | ReplyHeader(xid=-2, LastProcessedZxid，error)  lastProccessedZxid是应用到zk database里面的最后一个事务id  ping响应的error值总是0，即成功  client会将lastZxid赋值到本地存起来  client收到响应，仅仅更新最后一次recv数据的时间，以供下次间隔时间发ping  无响应体 |
| CloseSession | Zookeeper.close  发起关闭session请求。 | RequestHeader(OpCode.closeSession) |  |
| CreateRequest | Client向server  发送创建znode的请求 | RequestHeader(xid=自增,  type=Opcode.create)  createRequest(path,data,acl,flags)  其中flags代表createMode，比如临时有序、永久有序  Xid用于验证得到的响应是否是顺序一致的 | ReplyHeader(xid,lastZxid,  error)  lastZxid最后应用到database的  zxid  createResponse(path) |
| deleteRequest | Client向  Server发送  删除znode的请求 | requsetHeader(xid=自  增,type=Opcode.create)  DeleteRequest(path,version)  强制删除version，version为-1 | ReplyHeader(xid,lastZxid,error)  响应体为空 |
| SetDataRequest | SetData请求 | RequestHeader(xid,type=Opcode.setData)  setDataRequest(path,data,version) | ReplyHeader(xid,lastZxid,error)  setDataResponse(Stat)  stat包括：  czxid,  mzxid,  ctime,  mtime,  version  cversion,  aversion,  ephemeralOwner,  dataLength,  numChildren,  pzxid |
| setACLRequest | SetACLRequest请求 | RequestHeader(xid,type=Opcode.setAcl)  setAclRequest(version,acl,path) | ReplyHeader(xid,lastZxid,error)  SetACLResponse(stat) |
| ExistsRequest | Exist请求 | RequestHeader(xid,type=Opcode.exists)  ExistsRequest(path,watch)  Watch是bool值 | ReplyHeader(xid,lastZxid,error)  ExistsResponse(Stat) |
| GetChildren  Request |  | RequestHeader(xid,type=Opcode.getChildren)  getChildrenRequest(path,watch) | ReplyHeader(xid,lastZxid,error)  getChildrenResponse(childrenList)  childrenList就是children path list |
| GetDataRequest |  | RequestHeader(xid,type=Opcode.getData)  getDataRequest(path,watch)  Watch是bool值 | ReplyHeader(xid,lastZxid,error)  getDataResponse(data,Stat) |
| Sync |  | RequestHeader(xid,type=Opcode.getData)  SyncRequest(path) | replyHeader(xid,lastZxid,error)  SyncResponse(path) |
| Multi |  | requestHeader(xid,type=Opcode.getData)  MultiTransactionRecord()  说明：multitransationRecord  由MultiHeader-RequestRecord对组成的List  MultiHeader(type,done,error)  requestRecord即相应操作的request | ReplyHeader(xid,lastZxid,error)  MultiResponse(List<OpResult>)  如果其中有一个操作出错，那么OPresult会对应ErrorResut |
| getAclRequest |  |  |  |
| addAuth | 为连接添加auth权限 | RequestHeader(-4, OpCode.auth)  AuthPacket(0, scheme, auth)  在重连和调用addAuth时发送 | new ReplyHeader(h.getXid(), 0, KeeperException.Code.OK/ AUTHFAILED.intValue())  如果schema出错，client会收到  Auth\_failed响应，server关闭连接，。 |
| setWatch | 只在重连时发送设置watch。 | RequestHeader(-8, pCode.setWatches)  SetWatches(lastZxid,  dataWatches,existsWatches,childWatches)  说明：watch的只会保存在client所连server中，和session不同，Follow不会将watch转发给leader，而是自己管理。因此  每次由于各自原因重连时(不是session过期)，server会发生改变。因此需要发送  SetWatch请求让新server保存 | ReplyHeader(request.cxid, lastZxid, err.intValue()) |

* Follower/OBSERVER<--->Leader 2888(事务接收和同步数据端口)初始化、数据同步阶段,FOLLOWER/OBSERVER我们统称learner

|  |  |  |
| --- | --- | --- |
| FOLLOWERINFO  Follower->leader | 向leader发送注册请求，告诉leader我是你的follower了 | QuorumPacket(type,zxid=(acceptedEpoch,0),data=sid)  Zxid是转义自acceptedEpoch，低32位为0  Data是serverid即sid  说明：leader对每一个follower/observer都会开启一个learnerHandler，通过sid标示每个learnerHandler属于哪个follower，通过  acceptedEpoch，来协商leader用最新的  epoch |
| OBSERVERINFO  Observer->leader | Observer到leader  向leader发送注册请求，告诉leader我是你的observer了 | QuorumPacket(type,zxid=(acceptedEpoch,0),data=sid)  Zxid是转义自acceptedEpoch，低32位为0  Data是serverid即sid  说明：leader对每一个follower都会开启一个learnerHandler，通过sid标示每个learnerHandler属于哪个follower，通过  acceptedEpoch，来协商leader用最新的  epoch.  对于OBSERVERINFO,learnerType  为OBSERVER |
| LEADERINFO  Leader->learner | 收到大部分的followinfo请求后  就会发送learner  LEADERINFO信息 | QuorumPacket(type,zxid(newEpoch,0),协议版本号)  说明：learner收到epoch，更新本地的epoch |
| ACKEPOCH  Learner-->leader | 将自己的最新zxid和  Epoch发送给leader | QuorumPacket(Leader.ACKEPOCH, lastLoggedZxid, epochBytes)  说明：两个作用1、告诉leader我的epoch已经同步  2、告诉leader的最后zxid是多少，为数据同步做准备 |
| DIFF  Leader->learner | 通知learner，将进行DIFF方式的数据同步 | QuorumPacket(DIFF, zxidToSend) |
| PROPOSAL  Leader->learner | 数据同步阶段  每个propoasal  顺序发送 | QuorumPacket(Leader.***PROPOSAL***, request.zxid,  Data)  Zxid是要同步的proposal 的zxid，  Data包括txHeader(clientId,cxid,zxid,time,type)  和Record(createReqeust/setDataReqesut等)  cxid不同于create zxid，它是client在发送过来的xid |
| COMMIT  Leader->learner | 数据同步阶段  每个propoasal  后都跟着COMMIT | 只包含一个要提交的zxid |
| SNAP  Leader->learner | 通知learner，将进行SNAP方式的全量数据同步 | QuorumPacket(SNAP, zxidToSend)  紧接着会有一个zkdatabase tree序列化后发送给follower |
| TRUNC  Leader->learner | 通知learner，将进行数据回滚 | QuorumPacket(TRUNC, zxidToSend) |
| NEWLEADER  Leader-->learner |  | QuorumPacket(NEWLEADER, zk.getZxid()  说明：  1、learner接收到newleader说明数据同步数据发送完毕，返回ack  即可对外提供服务  2、Learner接收到后NEWLEADER,生成一个SNAP快照，还未开始对外提供服务 |
| ACK  Learner->leader |  | QuorumPacket(Leader.ACK, zxid)  说明：  当收到大部分ACK后,启动zookeeper server,并开启对外服务 |
| UPTODATE  Leader-->learner |  | QuorumPacket(Leader.UPTODATE, -1  说明：  learnerHandler收到ACK以后，发送UPTODATE，表示数据同步完成  follower、observer开始向外提供服务 |
|  |  |  |
|  |  |  |

* Leader<----->Follower 2888(事务接收和同步数据端口)消息广播阶段

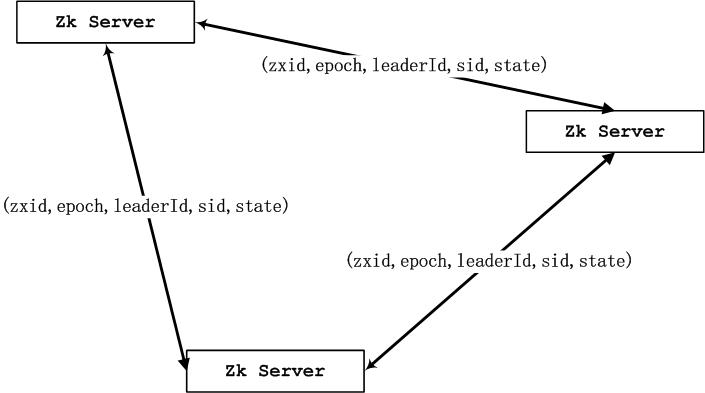
|  |  |  |
| --- | --- | --- |
| REQUEST  Learner->leader | Follower或者observer向leader转发事务请求。 | QuorumPacket(Leader.REQUEST, -1, data, request.authInfo)  Data包括sessionId,client的xid,type以及client过来的request  sessionId用于判断session是否过期，谁发起的操作 |
| Leader.REVALIDATE  follower->leader  Leader-->follower | 当client连接断开重连时(不一定是上一个server)，Follower发送到leader的session激活请求，leader接收后会将session激活并且将owner设置为来源Follower | QuorumPacket(Leader.REVALIDATE,data)  Data包括：sessionid,timeout |
| PROPOSAL  Leader->follower | Leader向follower发送  Proposal操作用于  记录到事务日志 | QuorumPacket(Leader.***PROPOSAL***, request.zxid,  Data)  Zxid是要同步的proposal 的zxid，  Data包括txHeader(clientId,cxid,zxid,time,type)  和Record(createReqeust/setDataReqesut等)  cxid不同于create zxid，它是client在发送过来的xid |
| ACK  Follower->leader | 写入到事务日志后  Follow向leader发出响应 | QuorumPacket(Leader.ACK, zxid) |
| COMMIT  Leader->follower | Leader向follower发送提交请求，follow将数据入库 | 只包含一个要提交的zxid |
| INFORM  Leader->observer | Leader向observer发送到数据+提交请求 |  |
| PING  Leader-->learner |  | new QuorumPacket(Leader.PING, lastProposedZxid)  说明：lastProposedZxid没有用 |
| PING  Learner->leader | Learner对leader发出的PING的响应 | QuorumPacket(Leader.PING，data)  说明：  Data：map(sessionId-timeout) |
| SYNC  Leader->learner | Follower向leader发送sync请求是以request的形式发送的。  Leader以SYNC形式响应Follower  专门用于client的  Sync请求 | new QuorumPacket(Leader.SYNC, 0) |

* Looking<---->Looking 3888(选举投票端口)选举投票阶段|或者新server加入

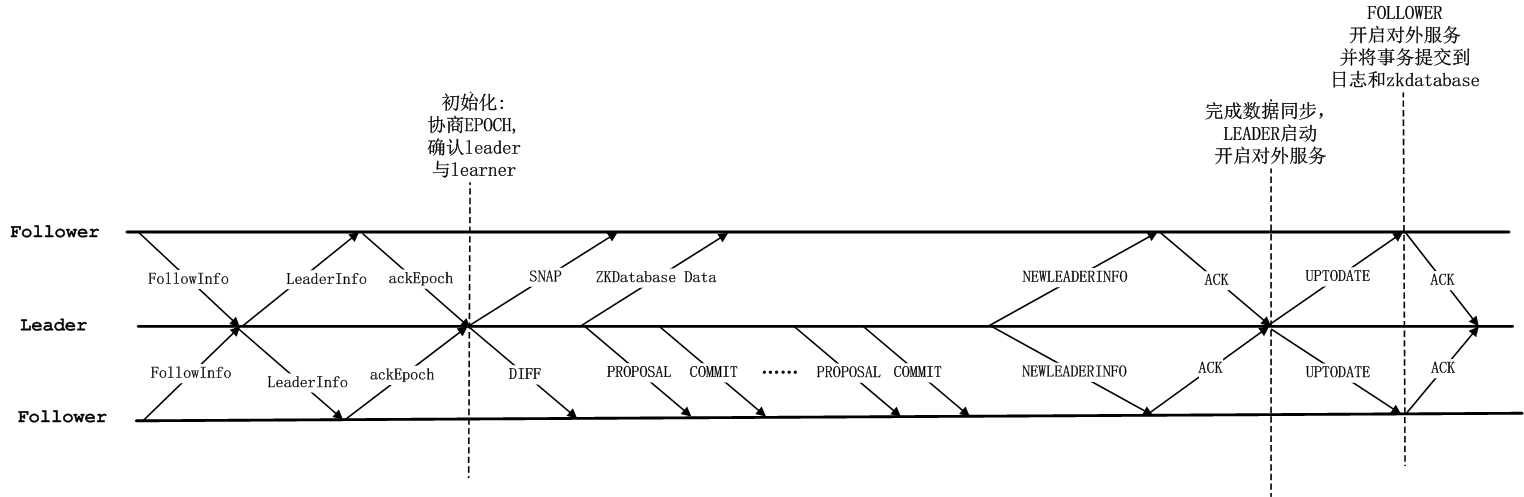
|  |  |  |
| --- | --- | --- |
| PEER <---> PEER | 处于LOOKING状态向所有server发送  投票。  处于F/L状态的只需要回应自己已经认同的投票 | ToSend(proposedleaderId,  proposedzxid,  proposedEpoch,sid,state)  sid:是对方serverId  state：是LOOING状态还是FOLLOWER\LEADER |

### 2、协议流程

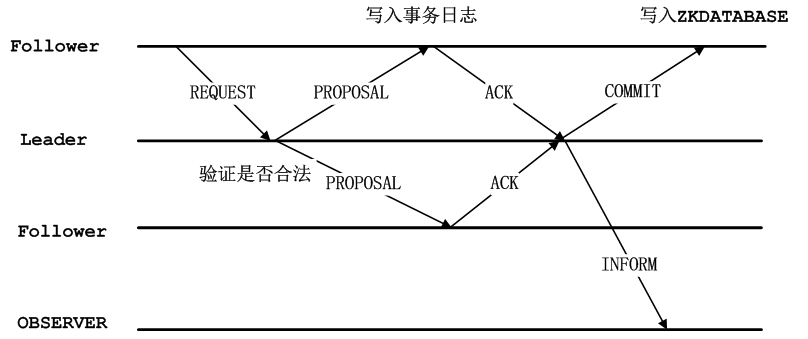
* 选举阶段



* 初始化、同步阶段



* 消息广播阶段



# 日志篇

### 1、ZKDatebase内存数据

事务日志和ZKDatebase快照文件是用于恢复数据到内存数据库的，有必要搞清楚ZKDataBase的数据结构。

数据单元DATANODE

|  |  |
| --- | --- |
| 类型 | 说明 |
| DataNode | Parent DataNode |
| Acl | acl |
| Byte[] | 数据 |
| Stat | Stat属性 |

### 2、事务日志

-rw-rw-r--. 1 whf whf 2 12月 16 17:00 acceptedEpoch

-rw-rw-r--. 1 whf whf 2 12月 16 17:00 currentEpoch

-rwxrw-r--. 1 whf whf 67108880 12月 14 14:47 log.100000001

-rw-rw-r--. 1 whf whf 67108880 12月 16 15:20 log.1800000001

-rw-rw-r--. 1 whf whf 67108880 12月 16 18:23 log.1900000001

-rwxrw-r--. 1 whf whf 67108880 12月 16 16:07 log.300000001

事务日志的文件后缀是文件第一条事务日志的zxid，高32位代表epoch，低32位代表自增序列号，我们从右边属8位，得到第一个文件epoch是1，第二个是24，第三个epoch是25，第四个文件epoch是3.

通过java LogFormatter 日志文件看看文件格式是怎么样的

14-12-16 上午10时35分45秒 session 0x14cbffda2220000 cxid 0x0 zxid 0x300000001 createSession 30000

14-12-16 上午10时36分20秒 session 0x14cbffda2220000 cxid 0x0 zxid 0x300000002 closeSession null

14-12-16 上午11时41分54秒 session 0x14cbffda2220001 cxid 0x0 zxid 0x300000003 createSession 4000

14-12-16 下午02时02分58秒 session 0x14cbffda2220001 cxid 0x2 zxid 0x300000004 closeSession null

14-12-16 下午03时09分24秒 session 0x14cbffda2220002 cxid 0x0 zxid 0x300000005 createSession 4000

14-12-16 下午03时11分31秒 session 0x14cbffda2220002 cxid 0x2 zxid 0x300000006 closeSession null

14-12-16 下午03时14分37秒 session 0x14cbffda2220003 cxid 0x0 zxid 0x300000007 createSession 4000

14-12-16 下午03时26分58秒 session 0x14cbffda2220003 cxid 0x0 zxid 0x300000008 closeSession null

14-12-16 下午03时28分58秒 session 0x14cbffda2220004 cxid 0x0 zxid 0x300000009 createSession 4000

14-12-16 下午03时53分04秒 session 0x14cbffda2220004 cxid 0x2 zxid 0x30000000a create '/hadoop-ha/whf/ActiveStandbyElectorLock,#a37768661236e6e311a47768663120ffffffd43e28ffffffd33e,v{s{31,s{'world,'anyone}}},T,1

14-12-16 下午03时55分17秒 session 0x14cbffda2220004 cxid 0x4 zxid 0x30000000b create '/hadoop-ha/whf/ActiveBreadCrumb,#a37768661236e6e311a47768663120ffffffd43e28ffffffd33e,v{s{31,s{'world,'anyone}}},F,2

14-12-16 下午04时07分26秒 session 0x14cbffda2220004 cxid 0x0 zxid 0x30000000c closeSession null

EOF reached after 12 txns.

可以看出

日志格式为：

时间|sessionId|cxid(client操作序列号)|zxid|opType(操作类型)|操作数据(事务记录比如SetDataTxn,createTxn)

其中内容是对每个字节十六进制打印，除此以外，每条事务日志都由一个4字节代表的checksum,用于验证数据是否完好无损，zk默认使用的是Adler32算法。

事务日志创建的时候会预分配空间，为的是避免磁盘Seek的频率提高IO效率，预分配的大小用系统属性zookeeper.preAllocSize来指定

### 3、快照文件

在每间隔写入一定数量事务日志后，就会异步全量创建一个快照文件，该快照文件序列化了整个zookeeper DataBase。文件名以当前zk提交的最大ZXID来作为后缀。

最后写入一个checkSum，写入快照文件中，以供反序列化时验证数据完整性使用，如果数据发生损坏就启动不了，这时只能通过从其它机器拷贝一份数据才能正常启动。

-rwxrw-r--. 1 whf whf 507 12月 16 10:10 snapshot.100000008

-rw-rw-r--. 1 whf whf 507 12月 6 21:54 snapshot.1600000004

-rw-rw-r--. 1 whf whf 594 12月 16 17:00 snapshot.180000000b

-rwxrw-r--. 1 whf whf 507 12月 4 23:56 snapshot.c00000002

执行java SnapshotFormat 快照文件

可以看到：

ZNode Details (count=7):

----

/

cZxid = 0x00000000000000

ctime = Thu Jan 01 08:00:00 CST 1970

mZxid = 0x00000000000000

mtime = Thu Jan 01 08:00:00 CST 1970

pZxid = 0x00001800000009

cversion = 2

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x00000000000000

dataLength = 0

----

/zk-book

cZxid = 0x00001800000009

ctime = Sat May 16 15:10:41 CST 2014

mZxid = 0x00001800000009

mtime = Sat May 16 15:10:41 CST 2014

pZxid = 0x00001800000009

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x00000000000000

dataLength = 3

----

/hadoop-ha

cZxid = 0x00000100000004

ctime = Sat Apr 11 11:57:41 CST 2014

mZxid = 0x00000100000004

mtime = Sat Apr 11 11:57:41 CST 2014

pZxid = 0x00000100000005

cversion = 1

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x00000000000000

dataLength = 0

----

/hadoop-ha/whf

cZxid = 0x00000100000005

ctime = Sat Apr 11 11:58:35 CST 2014

mZxid = 0x00000100000005

mtime = Sat Apr 11 11:58:35 CST 2014

pZxid = 0x00000100000005

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x00000000000000

dataLength = 0

----

/zookeeper

cZxid = 0x00000000000000

ctime = Thu Jan 01 08:00:00 CST 1970

mZxid = 0x00000000000000

mtime = Thu Jan 01 08:00:00 CST 1970

pZxid = 0x00000000000000

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x00000000000000

dataLength = 0

----

/zookeeper/quota

cZxid = 0x00000000000000

ctime = Thu Jan 01 08:00:00 CST 1970

mZxid = 0x00000000000000

mtime = Thu Jan 01 08:00:00 CST 1970

pZxid = 0x00000000000000

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x00000000000000

dataLength = 0

----

Session Details (sid, timeout, ephemeralCount):

最后一行是session信息，zkdatabase不仅存储datanode信息，还存储当前存活的session信息，包括sessionId，timeout，ephemeralCont

### 4、其它文件

rw-rw-r--. 1 whf whf 2 12月 16 17:00 acceptedEpoch

-rw-rw-r--. 1 whf whf 2 12月 16 17:00 currentEpoch

currentEpoch文件内容就一个数字，代表当前epoch，选举leader的时候使用currentEpoch发送自身的epoch，与leader协商后，会更新acceptedEpoch到acceptedEpoch文件中.

当作为数据同步以后，learner会更新currentEpoch，使其与acceptedEpoch一致。

所以accptedEpoch>=currentEpoch,acceptedEpoch是一个过渡文件。

# 流程篇

### 1. Zookeeper组件介绍

* **Leader**

负责同步数据、进行事务事务广播的zk实例

* **Learner**

Observer、Follower统称learner

* **LearnerHandler**

leader对每个learner分配一个learnerHandler来接收请求

### 2. 线程模型

* **线程类型说明**

**绿色代表client读写请求相关，蓝色代表leader选举相关，黄色代表数据同步相关，紫色消息广播相关**

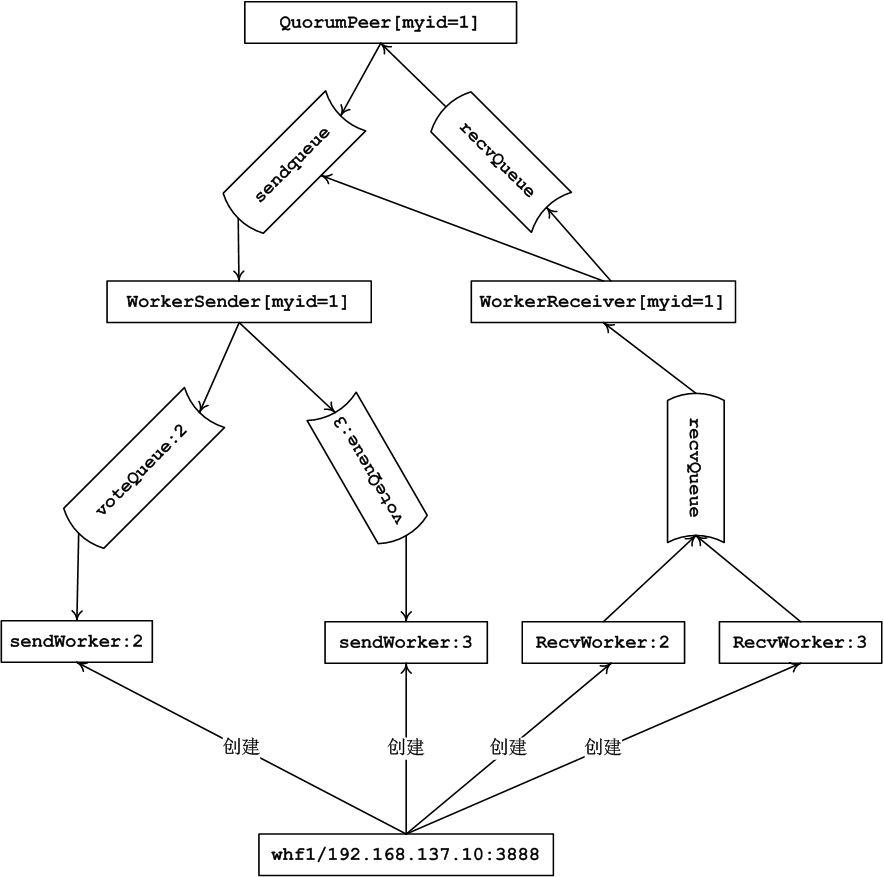
* **zk server**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **作用** | **角色** | **阶段** |
| **NIOServerCxn.Factory**  **:0.0.0.0/0.0.0.0:2181** | **对外接收client读写请求** | **Follower/leader/observer** | **消息广播/选举阶段/同步阶段** |
| **WorkerReceiver[myid=1]** | **用于leader选举；**  **将多个recvWorker收到的投票汇总起来，自身处于looking状态提供给quorumPeer进行投票判断，**  **自身处于learner和leader状态发出自己的投票交给workerSender(用于告诉重新加入的节点leader是谁)** | **Leader/follower/observer** | **消息广播/选举阶段/同步阶段** |
| **WorkerSender[myid=1]** | **用于leader选举；**  **负责将将要发出的投票**  **添加到各个serverid对应的发送队列中(每个serverid一个队列)** | **Leader/follower/observer** | **消息广播/选举阶段/同步阶段** |
| **whf1/192.168.137.10:3888** | **用于leader选举，监听选举端口，创建连接，创建**  **sendWorker和recvWorker**  **org.apache.zookeeper.**  **server.quorum.QuorumCnxManager**  **.Listener.run()** | **Leader/follower/observer** | **消息广播/选举/同步阶段** |
| **QuorumPeer[myid=1]**  **/0:0:0:0:0:0:0:0:2181** | **用于选举发送投票消息寻找leader，**  **Follower用于初始化数据同步，向leader注册自己**  **Leader用于初始化监控数据同步完成**  **Follower在消息广播用作接收leader过来的packet、返回ping**  **Leader在消息广播用作发ping并心跳检查**  **启动zookeeperserver，开启对外服务** | **Leader/follower/observer** | **消息广播/选举阶段/同步阶段** |
| **SendWorker:2** | **用于leader选举**  **将server2对应的队列数据依次发送给server2** | **Leader/follower/observer** | **消息广播/选举阶段/同步阶段** |
| **RecvWorker:2** | **用于leader选举**  **接收server2过来的投票，以队列方传送给workerReceiver线程** | **Leader/follower/observer** | **消息广播/选举阶段/同步阶段** |
| **CommitProcessor:1** | **用于等待commit消息，**  **处理事务请求，更新database**  **向client发出响应** | **Leader/follower/observer** | **消息广播** |
| **FollowerRequestProcessor:1** | **转发给commitProcessor线程处理，判断是否事务请求，是的话提交事务** | **Follower** | **消息广播** |
| **SyncThread:1** | **将事务记录到事务日志中，定期做日志快照，向leader响应ack** | **Leader/follower/observer** | **消息广播** |
| **LearnerHandler-**  **/192.168.137.10:57428** | **每个learnerr一个learnerHandler请求**  **用于数据同步阶段向learner初始化消息(leaderinfo)**  **接收来自learner的ack、ping、revalidate、request请求，同步处理ping;同步处理revalidate请求，并交给对应sender线程异步返回响应;**  **统计验证ack返回数量，交给所有同步Follower的sender异步发送commit消息**  **接收Request请求交给processor线程处理** | **Leader** | **消息广播** |
| **Sender-/192.168.137.10:57428** | **每个learner一个sender线程**  **用于向learner发送ping消息、同步阶段向learner发送proposal、commit消息、snap、diff等、消息广播阶段向Follower发送proposal**  **Commit消息，向observer发送info消息** | **Leader** | **消息广播** |
| **SessionTracker** | **定期检测session超时，修改session状态，对过期session提交**  **closeSession请求** | **Leader** | **消息广播** |
| **ProcessThread**  **(sid:2 cport:-1)** | **PreparedRequestProcessor**  **对于事务做预处理，用于验证操作是否合法。交给proposalRequestProcessor处理** | **Leader** | **消息广播** |
| **LearnerCnxAcceptor.run**  **（没有名字）端口2888** | **用于监听新learner的连接**  **，创建相应的learnerHandler线程。** | **Leader** | **消息广播** |

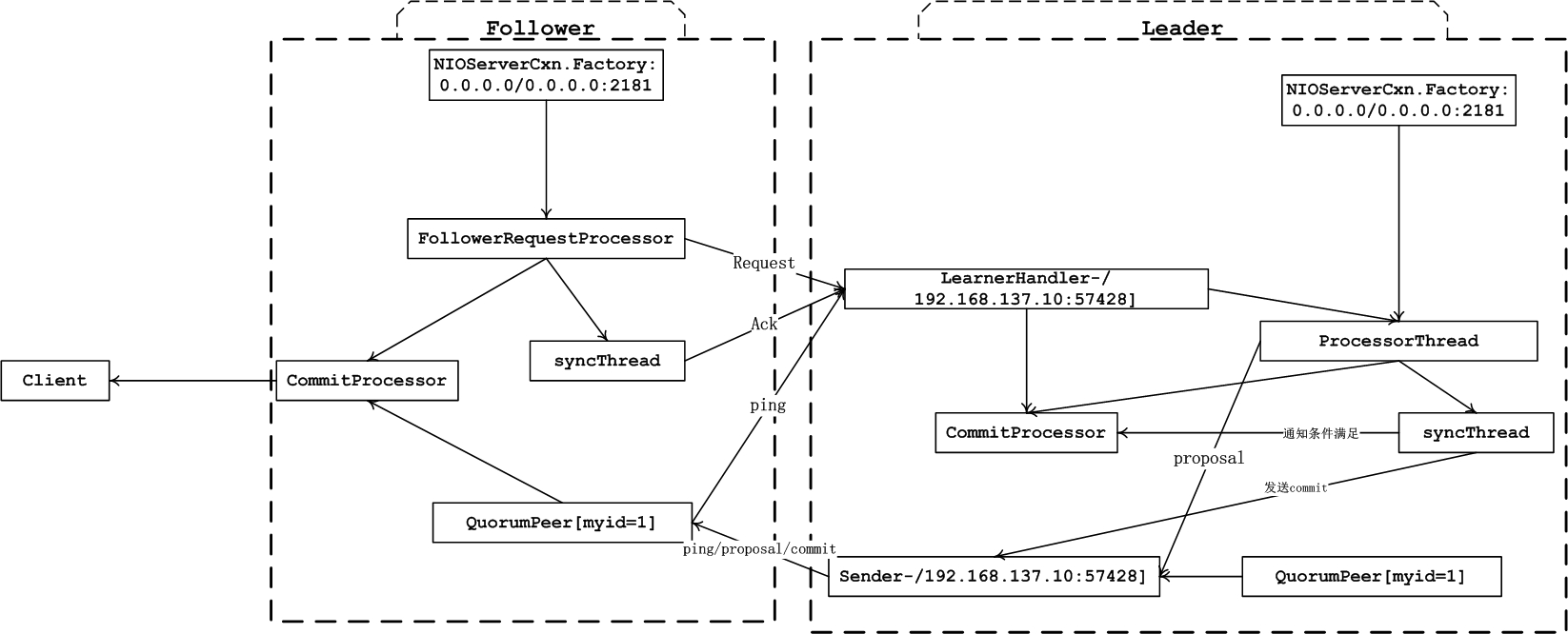
* **zkClient**

|  |  |  |
| --- | --- | --- |
| **名称** | **作用** | **角色/阶段** |
| **用户线程** |  |  |
| **sendThread** | **负责IO操作、发送请求接收响应、超时判断，**  **事件入队列，建立连接，建立session** |  |
| **evnetThread** | **处理事件和回调操作** |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

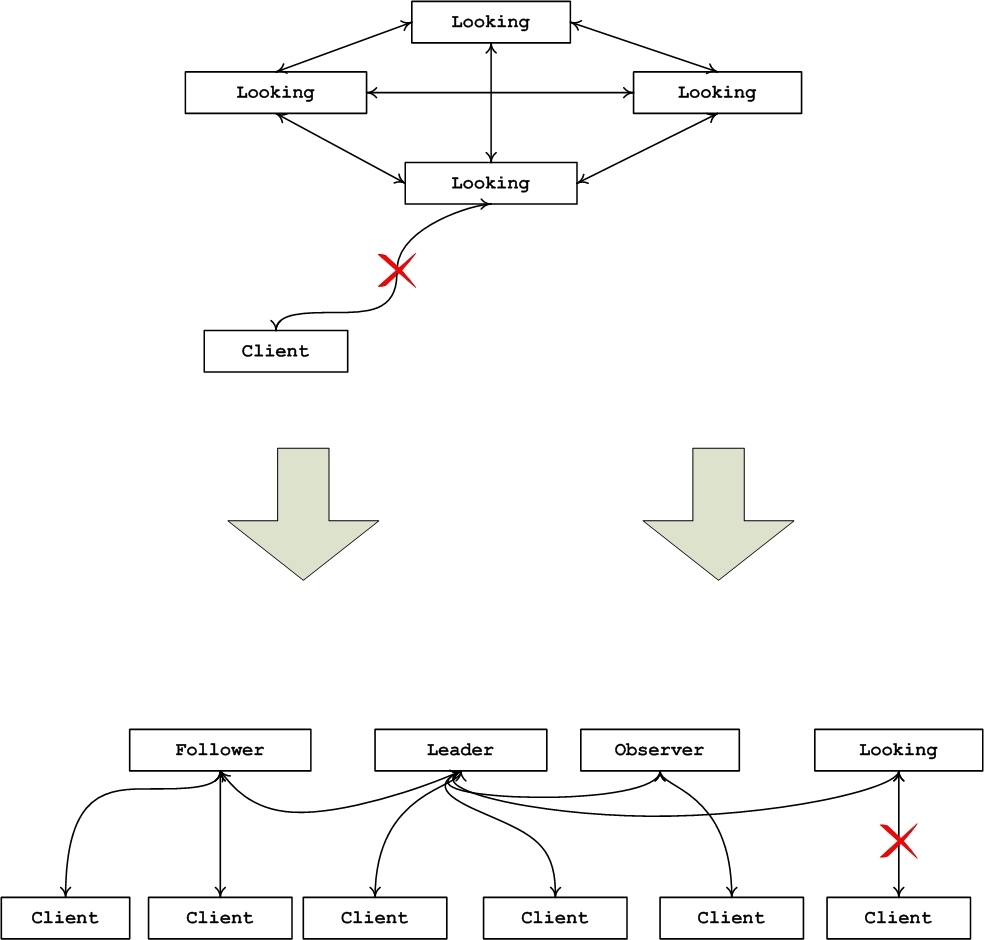
* **图解关联关系**
* **选举线程关系**

****

* **数据同步、消息广播线程**

****

### 3. 总体架构流程



* 点评：

当集群重启或失去leader时，所有参与选举leader的zk server都处于LOOKING状态，互发vote消息，由于每个server都有一个选举端口，两两之间必须只保持一个连接，为了达到这个目的，zk设定了一个规则即:sid大的去主动连接sid小的。

当client连接处于LOOKING状态的server时，server虽然可以建立连接，但是无法提供zookeeper服务。

当leader选举成功，状态从LOOKING变成leadering/followering/observering，

可以向外提供服务了，leader和learner之间以二阶段提交的形式处理事务请求，而learner可以单独处理非事务请求。当有宕机的zk server重新加入集群时，所有learner、leader向它返回当前的leader vote，looking收集投票进行验证通过后，角色变更为learner对外提供服务

### 4. zk启动过程



* 点评

启动过程做了几件事：1、从事务日志和快照日志恢复数据到内存数据库

2、启动对外服务端口

3、启动选举端口和相关线程，开始投票选举leader的过程

数据恢复是这样的：首先将最近的一次完好无损的快照文件全量解析到内存数据库，然后从事务日志中比快照文件生成的事务id的下一个事务id开始，依次应用事务到内存数据库当中，并加载currentEpoch以及lastZxid。

### 5. zk leader选举



* 点评：
* 选票格式:

(proposedLeaderId,proposedZxid,logicalclock,proposedEpoch)

初始投票是

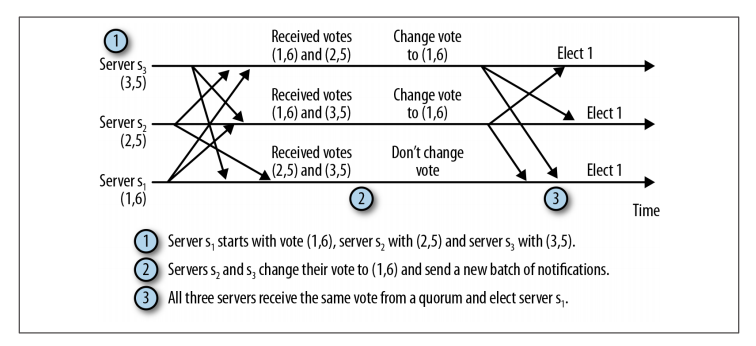
(自己的serverid,自己的lastZxid,自己的logicalClock,自己的currentEpoch)

说明：Logicalclock是选举次数，自进程启动以来经历的第几次选举，为避免因网络过慢导致上轮选举造成影响，故logicclock要达成一致，以最大的那个优先

* PK规则：先比较epoch，epoch相同的话比较zxid，zxid再相同的话比较serverid，值大的那个获胜，参与投票的server如果发现接收到的投票比自己大，那么会更新自己的投票再发送出去。
* 选举规则

1. 所有处于looking状态的server都向其它所有server发送投票，初始投票选自己。
2. 接收来自其它server的投票，接收不到就继续发送本地投票，如果PK失败则更新本地投票重新发出
3. 当收集的投票满足条件--->包括本地投票在内的quorum数量的投票都达成一致，等待200ms无异议时，当接收的投票来自于leadering或者following状态时还需要判断leader是否处于leadering状态，以避免建立连接不成重新进入looking状态
4. Follower和leader进入各自的角色，leader启动sync端口，follower向leader建立连接，follower向leader注册自己，leader发现follower不足够会重新进入选举阶段。

* 例子说明



如上图，假设epoch都是1，s3的sid=3,zxid为5，s2的sid=2，zxid为5，s3的sid=1，zxid=6.

s3,s2收到指向s1的投票后，修改投票重新发出，最终quorum数量达到投票一致，一致选举S1.

* **Falsely Election**

①还是上图的例子，s3,s2互发投票消息，由于网络原因都没有接到s1的投票,s3,s2判断投票多数达成一致都是选s3做leader，休眠200ms，s1投票依旧没有到来，于是s3成为leaer，s2成为follower。等s1投票消息到达s2,s3时epoch已经增1变成2了，s1将更新自己的投票选举s3.s1成为follower，在s1与leade数据同步时，s1将删除epoch=1时的事务6，造成数据丢失。

②还是s1网络故障，s1的投票没有达到s3,在200ms内到达了s2,s2将更新投票投s1,s1投票选s2,s3在休眠200ms后s1和s2新的投票消息仍然没有到达s3,s3判断s2为leader，试图与s2建立连接,但是s2认为s1是leader，于是不开启sync端口，s3建立连接超时，重新回到LOOKING状态。

finalWait控制在200ms以内，快速选举，这也就是Fast Leader Election命名的初衷吧

* 故障恢复

一个有2N+1个实例组成的zookeeper集群，最大能允许N个实例出现故障，超过N个就会造成服务不可用。那什么情况下会使得集群进入leader选举状态呢?

1、leader出现故障：当leader出现故障时所有的 Follower会进入LOOKING状态重新进行投票选举

2、当集群中有超过N个实例出现故障，这时leader将接收不到足够quorum数量的PING响应，这时leader 关闭同步端口，断开所有的learner连接，使得整个集群都进入LOOKING状态

3、当选举成功leader角色和Follower角色都已经建立之后，如果在Follower注册、Follower与leader建立连接、同步数据成功的个数，不足quorum数量，都会导致重新进入LOOKING状态重新进行leader选举。

4、learner和leader之间连接是BIO的，learner一直做阻塞读和阻塞写的循环，一旦发现读超时则退回到LOOKING状态重新寻找leader

### 6. 数据同步



* 点评

初始化和数据同步阶段主要完成这么几件事

Leader

1. 对于leader，启动同步端口，为每个learner分配一股learnerHandler和

Sender线程

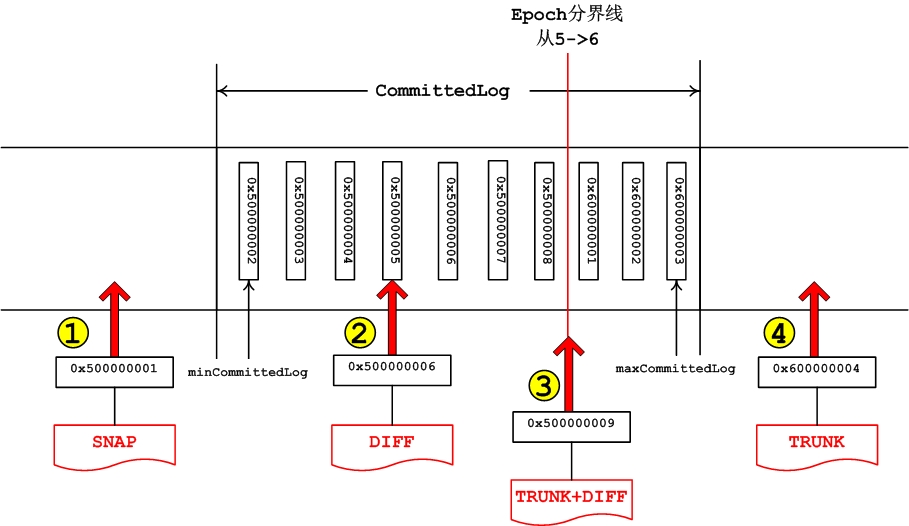
1. 等待超过集群机器半数的follower注册进入forwarding列表，

等待超过机器机器半数的follower接收到同步数据，完成数据同步。

1. 开启对外服务，启动处理事务的processor

Follower

1. 对于follower，建立与leader同步端口的连接
2. 同步epoch，向leader注册自己，完成数据同步
3. 启动处理事务的processor



* Zk server启动后会将最近最多500条事务保存在committedLog队列中，以便于leader->follower的增量同步。
* 同步策略说明

**SNAP**:leader将内存数据库快照序列化后发送给learner，learner全量替换本地的内存数据库

**DIFF**：以proposal+commit的方式，将committedLog的事务依次发给learner

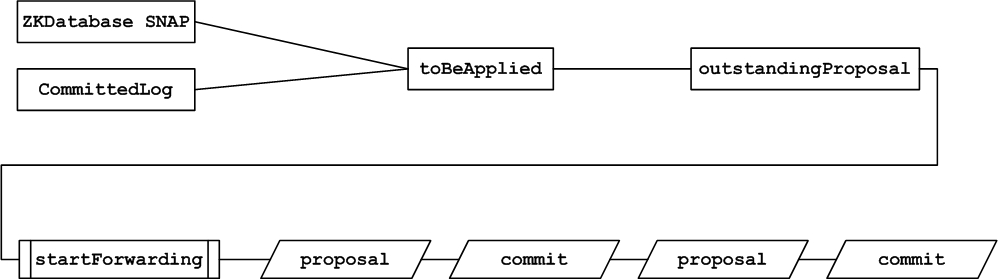
**TRUNK**:leader向learner发送(TRUNK+zxid)的消息，learner将日志删除到事务zxid位置

* 根据learner的lastZxid与committedLog的对应关系，将采用不同的同步策略：

1. lastZxid小于minCommittedLog，采用SNAP全量同步策略
2. lastZxid在minCommitedLog到maxCommittedLog之间，并包含在committedlog之中，走DIFF增量同步策略
3. lastZxid在minCommitedLog到maxCommittedLog之间，但不包含在committedlog之中(发生此情况的场景为:leader发送proposal成功之前，虽然记录到了本地但是其它follower并没有记录，导致数据丢失，但是client一样会认为操作失败)，走TRUNK+DIFF同步策略.
4. 这种情况几乎不能发生，我能想到的是follower首先记录了该事务日志，但突然宕机了，重启以后其它leader和follower依然还未执行记录该事务日志的操作。这种情况走TRUNK同步策略。

* committedLog是提交以后的事务日志，那么等待ack以及正在被finalRequestProceesor处理的是什么时候发送的呢？

答：Leader使用outstandingProposals等待ack的proposal，使用toBeApplied队列存放收到大多数ack但还没有被最终处理的proposal。在发送完数据库快照或者committedlog后，紧接着会发送toBeApplied和outstandingProposals，然后将follower添加到forwarding列表，开始参与事务投票(proposal-ack-commit)的过程，以此来保证learner不会丢失任何事务消息



### 7. server端事务处理



* 点评

1、**连接类型**

client-->server之间的连接是NIO的，learner--->leader之间的连接是BIO的，因为learner并不多，使用BIO的好处是读超时方便可控，learner和leader都很方便的判断对方是否读超时，另外leader为每个learner分配一个learnerHandler线程专门为一个learner服务，所以用不到NIO.

**2、session管理**

session生命周期从client发送connectRequest请求开始，所连server会分配一个sessionId以及sessionPasswd、协商生成sessionTimeout。而leader会对session进行超时管理，每隔1/2个ticktime，learner会接收到来自leader的ping请求，learner将本地保存的sessionId-timeout map返回给leader，并清空该map。leader会对收到的sessionId做移桶操作，延长过期deadline。

leader有个sessionTracker线程，每间隔ticktime将deadline桶内的session做expire操作，即发起closeSession请求，并将session置为close状态。

**3、watch管理**

watch是存放在各个server中的，而不是像session一样交给leader管理。

watch event有可能被所有事物操作触发，比如create/setData/closeSession(会删除临时节点),当server最后通过finalRequestProcessor进行处理的时候，会触发watch event，向client发送wath event消息。

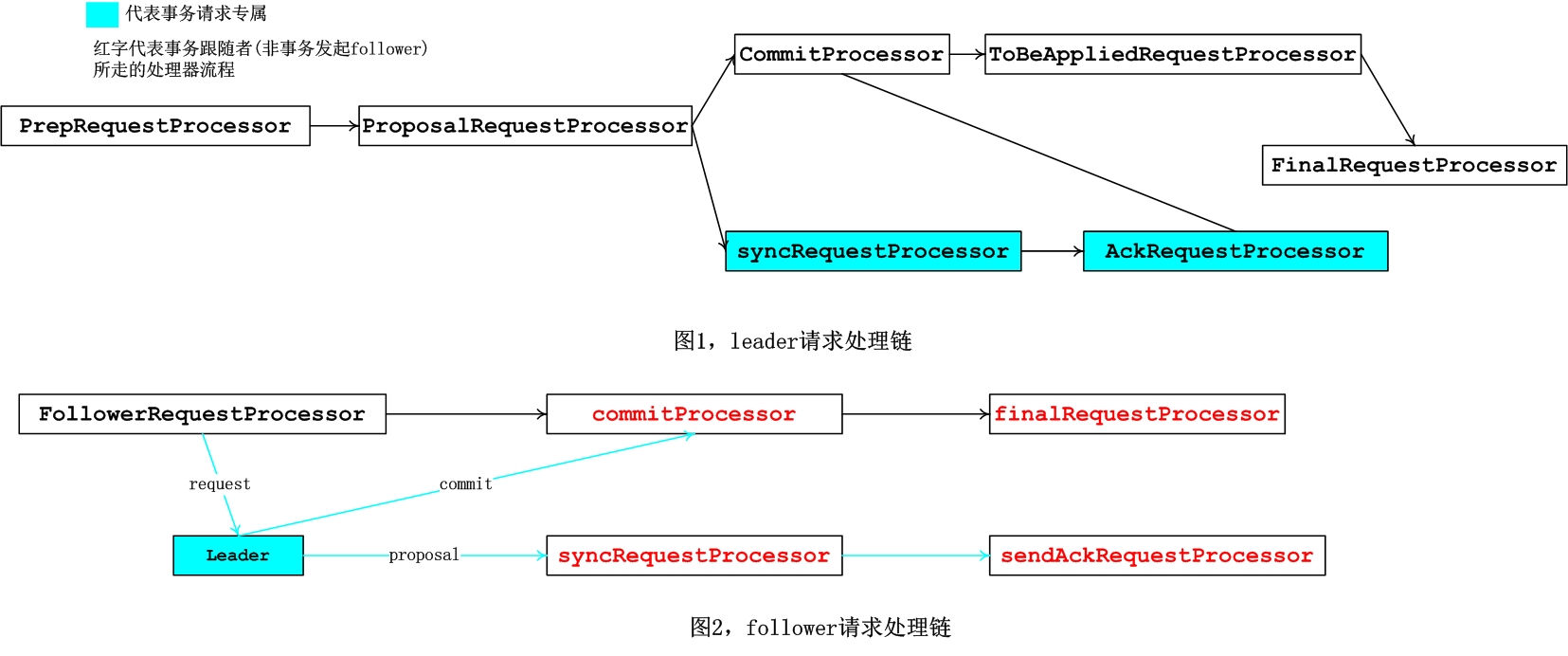
那么问题来了？如果存放client的Follower挂了是不是watch消失了，还需要用户在判断连接断开时重新将watch注册一遍？

答案是不需要，path-watch信息会保存在client一份，只要设置zookeeper.disableAutoWatchReset为true(默认true),每次在重连时会发送重新注册到新的zk server上，authinfo与此类似。

**4、事务处理器各组件介绍**

|  |  |  |
| --- | --- | --- |
| **类型** | **说明** | **所属角色** |
| **FollowerRequestProcessor** | 判断是否事务请求，是的话转发给leader  将请求交给commitProcessor | **follower** |
| **ObserverRequestProcessor** |  | Observer |
| **CommitProcessor** | 1对于非事务直接交给下一级处理器  2对于事务等待多数ack到达，然后交给下一级处理器 | Follower/observer/leader |
| **FinalRequestProcessor** | 1、将事务应用到内存数据库中，如果事务请求是属于该server对应的client发起的，还会组织返回结果响应给client  2、对于非事务获取请求结果数据返回给client | Follower/observer/leader |
| **prepRequestProcessor** | 判断当前是否事务请求，对于事务请求  创建请求事务头、事务体、版本检测、ACL检测、session检测 | leader |
| **SyncRequestProcess** | 记录事务日志到文件中，触发快照文件生成，事务日志文件滚动 | Follower/observer/leader |
| **SendAckRequestProcessor** | 返回ack给leader | Follower |
| **AckRequestProcessor** | 收集从follower过来的ack响应，判断达到多数条件唤醒commit线程  发送commit inform消息到learner | leader |
| **ToBeAppliedRequestProcessor** | 交给finalRequestProcessor，完事后从tobeapplied队列里面移除事务 | leader |
| **ProposalRequestProcessor** | 1. 对于非事务直接转发给commitProcessor 2. 对于事务请求发送proposal并交给syncRequestProcessor进行本地事务日志的记录 | leader |

**5、事务处理器之间的关联关系**

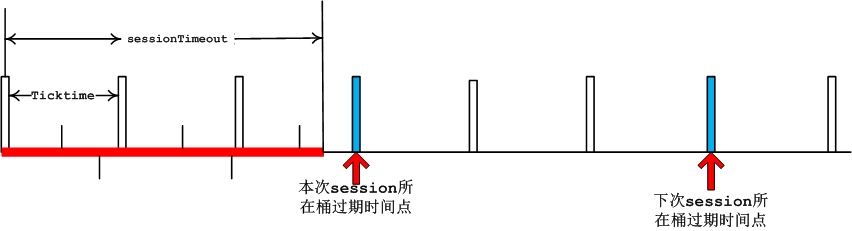
****

**6、session过期管理**

session创建之后就会交给leader来检测session的过期，发现过期时会发起closeSession事务。

session的生命周期的延续和ping请求密不可分，而ping请求不仅存在于client<->server还存在于leader<->learner。

client在没有zk操作的情况下平均间隔1/3 sessionTimeout时间(最大十秒)向server发送一次心跳(ping packet)，server需要立即返回响应,而leader每隔1/2 ticktime发送所有learner一次心跳(带上这段时间的发ping的来源session)。client到server，leader到learner，learner到leader，这三个环节共同组合维系session的生命周期，有一个环节出错都有可能造成session的失效。



如上图所示，红线代表sessionTimeout，蓝色长条代表session所在桶最后过期时间，由于leader在分配桶的时候是根据sessionstimeout做round计算，第二个蓝色条是下一次过期时间。client每隔1/3的sessiontime时间单位向server发送ping，server每隔1/2 ticktime向leader发送ping，leader接收到server的ping请求附带的sessionId，延续session的生命。

由此可见Zookeeper集群中所有server的conf配置都必须一样，不能出现ticktime有的大有的小。

### 8. client端请求流程



* 点评

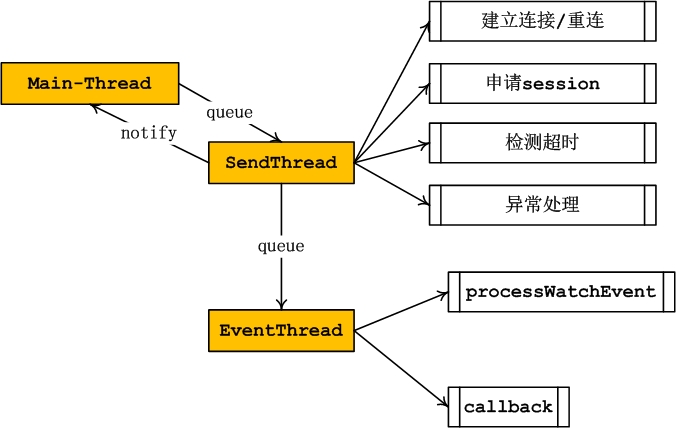
Client分为二个线程和多个用户线程，用户线程负责提交操作，sendThread负责读写，还有一个eventThread负责执行回调方法(如果选择了异步执行)和处理watch事件。

主线程提交操作请求后会挂起等待数据结果返回，SendThread得到结果后会唤醒主线程，返回结果如果有error code，它会翻译成异常抛出来。

SendThread负责读写、建立连接、重连、检测连接状态、检测超时、以及异常处理，比如遇到超时异常会清空所有等待packet，并唤醒用户线程，并且发布connectloss事件通过队列传输给eventThread。

EventThread负责处理watch event和执行回调方法。队列里面存放两种类型的object一类是sendThread产生的watch event一类是server返回的packet。

Watch event不一定是来源于server，也可以是sendThread生成的，比如检测到收包超时。



* syncConnected事件和connectloss事件会通知所有注册的watch
* 当出现不可恢复的异常时AUTH\_FAILED,SESSION\_EXPIRED,sendThread、eventThread都会退出。

遇到SESSION\_EXPIRED(一般发生在重连)，sendThread会往event队列里面queue一个特殊event---eventOfDeath，eventThread收到这个event后，会从while循环中退出，进而退出线程。

在显式调用Zookeeper.close方法时，会提交一个closeSession请求，然后close

sendThread，queue一个eventOfDeath给eventThread,

* new ZooKeeper("localhost:2181",5000,new xxxUserDefinedWatcher());

在构建Zookeeper Client的时候，构造方法里有一个默认watcher，在getData，

exist方法中如果没有指定watch，就会使用默认watch

* 在发生ConnectionLoss时，client会重连下一个serverIP，并带上sessionId、sessionPasswd重新激活会话，并将缓存中的auth、watch重新注册添加上。

# 应用篇

### 1、Zookeeper Example



### 2. zookeeper recipe

* **Name　Service**
* **Configuration**
* **Group Membership**

建立一个永久节点，小组成员在该节点下创建临时子节点。

* **分布式锁**
* 分布式独占锁

实现一：

1、调用create(),创建znode"/lock/uuid-lock-00000001",模式选择临时+有序，

2、调用getChildren()，不设置watch，获取节点路径集合排序，如果第一步创建的子节点序号是最小的，那么该client获取锁

3、否则调用exists(),设置watch ，监听比第一步小的那个znode

4、exist()返回false或者收到watch事件通知，进入第二步，否则继续等待通知

实现二：

1、client-A create()一个临时节点/lock，设置Ids.CREATOR\_ALL\_ACL权限控制，如果创建成功则成功获得锁

2、如果/lock已经存在就调用exists(),同时设置watch，如果返回true则进入等待，等待通知

3、如果返回false，则进入第1步

4、获取锁执行完操作，删除/lock

点评： 实现一：比较复杂，需要将所有子节点取回到本地然后排序，但避免了羊群效应

实现二：需要设置权限并且会出现羊群效应

zookeeper处理事务本来就是串行的，所以并不存在公平锁和非公平锁之分，两种实现都是公平锁。

* 分布式读写锁

实现一:

获取读锁

1、调用create，创建"\_locknode\_/read\_",模式选择临时+有序

2、调用getChildren,不设置watch，将结果按照序号正序排序

3、如果children结果集中没有一个以“write-”前缀比第一步创建的znode序号更小，那么该client获取锁.

4、否则调用exists，设置watch，监听下一个比client序号小并且以write-开头的znode。

5、如果返回false或者收到通知，跳到第2步，否则就一直等待通知

获取写锁

1、调用create创建节点"\_locknode\_/write-",临时+有序

2、调用getChildren，并将结果集排序

3、如果结果集中没有一个比第一步创建的节点小的，那么该client获取锁

4、否则调用exists，设置watch监听下一个比第一步创建znode序号小的path

5、如果返回false获取通知则进入第二步，否则等待通知

* **leader选举**

实现一

leader选举就是获取分布式独占锁，获取成功执行leader相关操作，获取失败等待锁释放通知

* 分布式队列
* 分布式优先队列
* 分布式屏障Barrier

### 3. zookeeper client的封装

* **ZkClient**

https://github.com/sgroschupf/zkclient，

kafka在用

* **Curator**
* **为什么要用curator**

原生Zookeeper Client有很多坏处，让开发者关注的点太多太底层，需要开发者处理很多情况，如果对Zookeeper使用的不好，很容易出现问题，以下列出我觉得原生Zookeeper client不好的地方

1、没有完善的重试机制，当server网络异常导致SendThread重连的时候，如果这个时候有Zookeeper api操作，统统返回connecttionLossException，而不是等待连接后重试

2、遇到SessionExpiredException、SessionMovedException，后续的zk操作将全部报错，除非你重新重建Zookeeper对象实例

3、由于watch是触发是一次性的，每次收到watch event都要重新注册watch，否则下次将无法监听到变更，对开发人员来说这种细节很容易遗漏

4、创建znode时，需要捕获nodeExistsException异常

5、既想监听子节点数量变化，又想监听子节点内容变化需要写很多代码

6、创建子节点需要先创建父节点，不能一次性一起创建，删除父节点需要先删干净子节点不能一次性一起删除

* **curator有什么好处，能解决哪些问题**

curator完美解决了原生API的以上所有问题，并且提供了很多强大的功能，方便的工具类，丰富的recipe实现，以及可读性很强的Fluent风格的API框架(链式风格)

* **curator概况**

**curator由以下几个部分组成**

Client: 是ZooKeeper客户端的一个高级封装, 提供了一些底层处理和相关的工具方法.

Framework: 用来简化ZooKeeper高级功能的使用, 并增加了一些新的功能, 比如管理到ZooKeeper集群的连接, 重试处理

Recipes: 通用ZooKeeper的recipe, 该组件建立在Framework的基础之上

Utilities:各种ZooKeeper的工具类

Errors: 理, 连接, 恢复等.

Extensions: recipe扩展

* **curator API 介绍**

1、client的创建、重试机制

**static** CuratorFramework *client* =CuratorFrameworkFactory.*builder*()

.connectString("192.168.137.10:2181") .retryPolicy(**new** ExponentialBackoffRetry(1000, 3)) .sessionTimeoutMs(50000).build();

client.start();//调用start才能开始使用

创建client，设置重试机制重试3次，每次休眠大约1秒

并不是遇到所有异常都将重试，只限于以下几个异常才会重试：

rc == KeeperException.Code.CONNECTIONLOSS.intValue())

(rc == KeeperException.Code.OPERATIONTIMEOUT.intValue()) ||

(rc == KeeperException.Code.SESSIONMOVED.intValue()) ||

(rc == KeeperException.Code.SESSIONEXPIRED.intValue());

当遇到sessionExpired时候，底层会重新创建一个Zookeeper实例，auth信息不会丢，但是非默认watch不会重新注册，随session消失的临时节点也不会重建

2、创建节点

client.create().creatingParentsIfNeeded().withProtection().withMode(CreateMode.EPHEMERAL\_SEQUENTIAL).forPath(path);

如果缺少父节点将首先递归创建父节点，模式临时有序节点，withProtection用在临时+有序模式。当client创建znode时有可能出现连接异常，这时client并不知道是否成功，使用withProtection会在创建节点path前面添加一个UUID，下次重试时首先查看是否存在这个节点，如果存在，说明上次创建成功退出重试并返回。

3、删除节点

client.delete().guaranteed().forPath(path)

如果删除失败，就继续后台删除直到删除成功或者session失效为止，在分布式锁情况下有用

client.delete().deletingChildrenIfNeeded().forPath(path)

如果有子节点就先删除子节点

client.delete().withVersion(version).forPath(path)

根据版本号删除节点

4、获取数据

client.getData().storingStatIn(stat).forPath(path)

读取一个节点的数据内容，同时获取到该节点的stat

5、更新数据

client.setData().withVersion(version).forPath(path)

更新一个节点的数据内容，强制指定版本进行删除

6、异步接口

public void processResult(CuratorFramework client,CuratorEvent event) throws Exception;

7、普通watch、全局listener的注册

client.getData().usingWatcher(new CuratorWatcher() {

@Override

public void process(WatchedEvent event) throws Exception {

System.out.println("path:" + event.getPath());

EventType eventType = event.getType();

if (eventType == EventType.None) {

KeeperState state = event.getState();

System.out.println(state.toString());

} else {

System.out.println(eventType.toString());

}

System.out.println(client.checkExists().watched().forPath("/just123"));

}

}).forPath("/just123");

以上watch的注册和原生的watch注册没什么区别

由于curator在SessionExpired情况下需要重建Zookeeper，所以以上注册在上一个Zookeeper实例中的watch将被垃圾回收。

client.getCuratorListenable().addListener(new CuratorListener() {

@Override

public void eventReceived(CuratorFramework client, CuratorEvent event) throws Exception {

System.out.println(client.checkExists().watched().forPath("/just123"));

}

});

全局listener，所有的事件通知都会经过listenable的listener，无论是连接丢失还是发生SessionExpired，该listener都不会消失，因为它一直绑定在默认watch中，默认watch在每次Zookeeper对象生成时都会传值进去

8、**NodeCache**

有了curator后，有了更方便的方法做事件监听了，节点监听使用nodeCache，NodeCache监听znode创建、删除、内容变更，不会随session过期而失效，无需重复注册，非常方便。

代码：

****

**9、PathChildrenCache**

nodeCache针对单个节点的创建、删除、内容变更，那么子节点列表的变更和子节点内容的监听就需要pathChildrenCache出马了，这货同样不需要重复注册，同样不会随session过期而失效，可以区分节点添加和节点删除事件



10、ensurePath工具类

EnsurePath ensurePath = **new** EnsurePath(*path*);

*client*.start();

System.***out***.println(*client*.usingNamespace("zk-book"));

ensurePath.ensure(*client*.getZookeeperClient());

使用EnsurePath无需捕获NodeExistsException了

**11、curator不好的地方**

curator支持SessionExpired之后重建Zookeeper继续进行client api操作，但是这时有可能临时节点都已经消失，显示注册的watch也已经失效，操作之前的临时节点时会抛出NoNodeExistsException。

同样curator支持sessionExpired之后重建Zookeeper申请新的session，有可能导致新session时看到的数据与老session时看到数据不一致。

* **curator 对recipe的实现**

**1、分布式锁**

****

**2、leader选举**

****

**3、分布式共享整形变量**

****

* **curator Example**

****

* **curator 官网地址**

**更多参考：http://curator.apache.org/index.html**

### 4. zookeeper 实践与应用

Hadoop，Hbase，kafka，Solr，等等

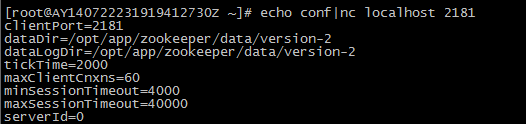
# 运维篇

### 1. zk常用命令

* 四字命令

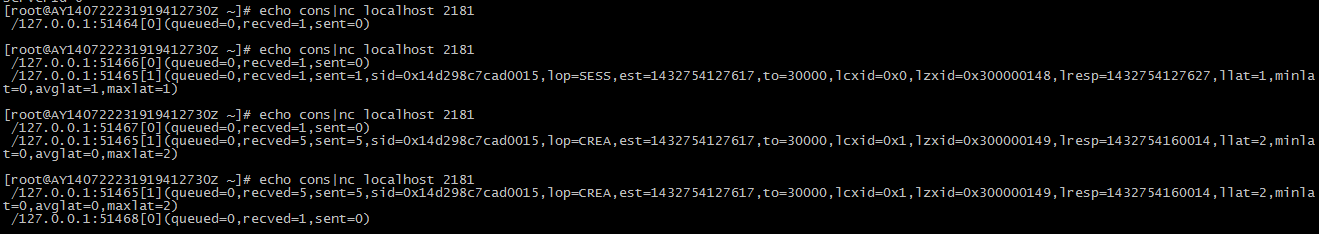
**1、conf**

打印该server中zoo.conf配置



**2 cons**

List full connection/session details for all clients connected to this server. Includes information on numbers of packets received/sent, session id, operation latencies, last operation performed, etc..



**3、crst**

Reset connection/session statistics for all connections

[root@AY140722231919412730Z ~]# echo crst|nc localhost 2181

Connection stats reset.

**4、dump**

Lists the outstanding sessions and ephemeral nodes. This only works on the leader.

[root@AY140722231919412730Z ~]# echo dump|nc localhost 2181

SessionTracker dump:

Session Sets (3):

0 expire at Thu May 28 03:21:32 CST 2013:

0 expire at Thu May 28 03:21:42 CST 2013:

1 expire at Thu May 28 03:21:52 CST 2013:

0x14d298c7cad0015

ephemeral nodes dump:

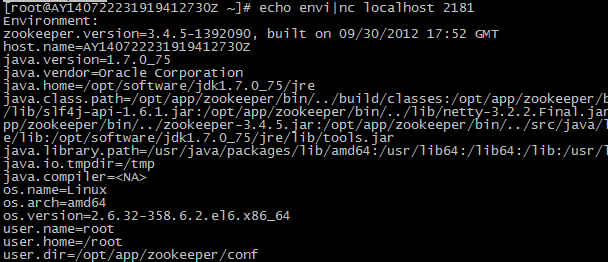
Sessions with Ephemerals (1):

0x14d298c7cad0015:

/dafs

**5、envi**

Print details about serving environment



**6、ruok**

Tests if server is running in a non-error state. The server will respond with imok if it is running. Otherwise it will not respond at all.

**A response of "imok" does not necessarily indicate that the server has joined the quorum,** just that the server process is active and bound to the specified client port. Use "stat" for details on state wrt quorum and client connection information.

[root@AY140722231919412730Z ~]# echo ruok|nc localhost 2181

imok

**7、srst**

Reset server statistics.

**[root@AY140722231919412730Z ~]# echo srst|nc localhost 2181**

**Server stats reset.**

**8、srvr**

 Lists full details for the server.

**[root@AY140722231919412730Z ~]# echo srvr|nc localhost 2181**

**Zookeeper version: 3.4.5-1392090, built on 09/30/2013 17:52 GMT**

**Latency min/avg/max: 0/0/1**

**Received: 5**

**Sent: 5**

**Connections: 2**

**Outstanding: 0**

**Zxid: 0x300000149**

**Mode: leader**

**Node count: 78**

**9、stat**

Lists brief details for the server and connected clients.

**[root@AY140722231919412730Z ~]# echo stat|nc localhost 2181**

**Zookeeper version: 3.4.5-1392090, built on 09/30/2013 17:52 GMT**

**Clients:**

**/127.0.0.1:51465[1](queued=0,recved=40,sent=40)**

**/127.0.0.1:51477[0](queued=0,recved=1,sent=0)**

**Latency min/avg/max: 0/0/1**

**Received: 15**

**Sent: 15**

**Connections: 2**

**Outstanding: 0**

**Zxid: 0x300000149**

**Mode: leader**

**Node count: 78**

**10、wchs**

Lists brief information on watches for the server.



**11、wchc**

Lists detailed information on watches for the server, by session. This outputs a list of sessions(connections) with associated watches (paths). Note, depending on the number of watches this operation may be expensive (ie impact server performance), use it carefully.



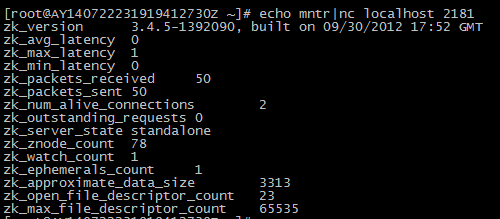
**12、wchp**

Lists detailed information on watches for the server, by path. This outputs a list of paths (znodes) with associated sessions. Note, depending on the number of watches this operation may be expensive (ie impact server performance), use it carefully.

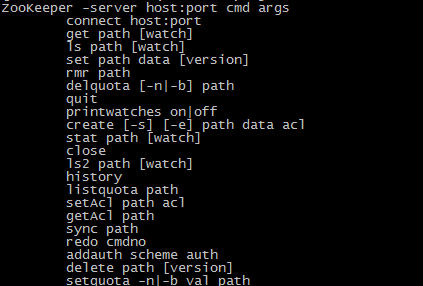


**13、mntr**

Outputs a list of variables that could be used for monitoring the health of the cluster.



* zkCli.sh



* https://github.com/outbrain/zookeepercli

一个写zk脚本很方便的工具，就是用它才得以删除带空格的znode的

$ zookeepercli --servers srv-1,srv-2,srv-3 -c create /demo\_only some\_value

# Default port is 2181. The above is equivalent to:

$ zookeepercli --servers srv-1:2181,srv-2:2181,srv-3:2181 -c create /demo\_only some\_value

$ zookeepercli --servers srv-1,srv-2,srv-3 --format=txt -c get /demo\_only

some\_value

# Same as above, JSON format output:

$ zookeepercli --servers srv-1,srv-2,srv-3 --format=json -c get /demo\_only

"some\_value"

# exists exits with exit code 0 when path exists, 1 when path does not exist

$ zookeepercli --servers srv-1,srv-2,srv-3 -c exists /demo\_only

true

$ zookeepercli --servers srv-1,srv-2,srv-3 -c set /demo\_only another\_value

$ zookeepercli --servers srv-1,srv-2,srv-3 --format=json -c get /demo\_only

"another\_value"

$ zookeepercli --servers srv-1,srv-2,srv-3 -c delete /demo\_only

$ zookeepercli --servers srv-1,srv-2,srv-3 -c get /demo\_only

2014-09-15 04:07:16 FATAL zk: node does not exist

$ zookeepercli --servers srv-1,srv-2,srv-3 -c create /demo\_only "path placeholder"

$ zookeepercli --servers srv-1,srv-2,srv-3 -c create /demo\_only/key1 "value1"

$ zookeepercli --servers srv-1,srv-2,srv-3 -c create /demo\_only/key2 "value2"

$ zookeepercli --servers srv-1,srv-2,srv-3 -c create /demo\_only/key3 "value3"

### 2. jmx

# FAQ

1、启动时事务日志有脏数据是否影响启动？

答：会影响，启动不成功，只能人工拷贝一份正常数据到该机器上，对应快照日志zk会找事务id最大的一个无损文件来load到内存，这此过程中遇到任何IO异常都会导致无法启动成功。

2、两个client同一时刻读同一个znode节点内容，会出现一个读到滞后的数据吗?

答：官方指出The clients view of the system is guaranteed to be up-to-date within a certain time bound.意思是follower落后最多这个时间段内(syncLimit\*ticktime)的事务消息，原因是在syncLimit\*ticktime内如果follower不响应来自leader的ping请求，或者在未能在syncLimit\*ticktime时间内将最早的一次proposal做出响应，follower将被移除learner成员列表。进而该follower无法继续提供服务。

上述已经证明，**①**不属于最近一次事务quorum成**员在数据上会出现数据滞后**，滞后范围最大是在syncLimit\*ticktime时间段内的事务数

**②**对于两个不同server属于最近一次事务quorum成员(最近一次事务成功返回ack)，如果这时两个client分别向这两个server发送读请求，**那么最多发生一次事务滞后**，有可能client-A获取比client-B更新的事务，是应用新事务的数据，而client-B获取的是没有应用新事务的数据，取决于收到commit的先后顺序。

**③**server-A提交了事务操作，server-B是leader，紧接着两个client分别向server-A和server-B发送读请求，两个client**将读到同样的数据**。因为事务发起方和leader在等待commit消息的时候会阻塞住后续的所有读写请求，导致该次commit到内存数据库操作一定发生在所有后续所有读写请求的前面，最后一次事务操作使得leader和server-B做了同步

这种情况有几种方法可以避免：

1做sync操作，A写完数据然后通知B我已经修改数据了，B发送一次sync操作然后会读到最新的数据。sync操作相当于一个"疏通器"，从leader那走一遍等sync返回响应时，之前的所有的事务消息都已经同步到client所连的server中了。不光sync操作，所有写操作都可以称为"疏通器"

2只保证写操作正确，用update by version，比如计数器

3自己只操作自己的节点，zk保证会话一致性

4允许延迟消息但是不想丢掉数据更新用watch

3、curator的连接管理，超时管理，ping请求怎么做的？

答：curator有一个线程专门做连接异常事件的处理，超时管理和ping请求管理都是底层ZOOKEEPER 做的

4、session move到新server会不会 形成新的sessionid

答：不会形成，leader在激活或者创建session时会记录每个session的owner，在follower中，sessionId与client连接绑定，在leader有具体的session对象负责存放sessionId，owner等信息，client在进行重连到其它server时会带上已经申请过的sessionId,leader发现是激活请求会为session对象设置新的owner对象，而不是重新创建一个session。

5、重连时会去检测zxid？

答：client在重连到其它server时，会带上lastSeenZxid参数，用于判断新的server是否包含最新的事务更新，如果不包含，client会重新寻找下一个server建立连接激活session。有可能重试次数过多耗时太长，leader将session过期，这时再发激活请求会收到SessionExpiredException的错误码。

6、选举时还会提供服务吗，这时会遇到什么问题？

答：不会提供服务了，所有服务都会失败。日志报告:ZooKeeperServer not running，客户端要等，server如果处于looking状态，服务是不可用的，即使处于leader或者follow而角色，启动zookeeper也需要quorum数量的server数据同步完成以后。

7、falsely election会不会发生？

答：会发生，前面已经说明过，搜Falsely Election

8、如果follower如果有尚未ack或者尚未commit的事务，会继续处理接下来的事务吗，会提前被ack或者commit吗？

答：可以回头看server端事务处理那一节的follow而处理链的图，ack和commit是处在不同的线程分支，当follow没有响应ack，并不影响它接收来自leader的proposal和commit消息，只不过后续的ack发送会等待(ping会定时检测ack是否即时)。当同一个事务对应的ack发生在commit之后，那么这个follower一定不是这个事务的quorum成员。

由此可见，写事务日志有可能发生在更新内存数据库的后面。

9、旧leader到新leader的过程有可能会发生时间空隙，如何规避时间交叉带来的影响？

答：这个时间窗口，zk集群在进行leader选举，所有server都将停止服务，而client会不停的轮询server来发起激活session的请求，但会一直报告ConnectionLosss的异常。

ticktime、syncLimit、initLimit不要设置太大，因为很多连超时、收包超时都是基于它们的，让集群server之间尽快收到其它server的异常通知，可以加快选举过程的等待的时间。

平时要做好监控，包括等待ack的时间，ack平均响应时间，写磁盘的速度等。ticktime、syncLimit、initLimt不要设置过大。之前遇到一次，将initLimit设置了很大，leader一直在

org.apache.zookeeper.server.quorum.Leader.getEpochToPropose(long, long)方法里等待，而follower早已经回到LOOKING状态发送投票。之前处在following状态的server由于重新回到了LOOKING状态，选举Epoch也增加了1，这个时候它会判断除了自己是否有大多数server跟随这个leader，由于没达到条件导致一直等待，直到持续syncLimit\*ticktime时间过期。这应该是zookeeper改进的地方

10、较小的zxid会在较大zxid之前提交到zookeeper database，这种情况会发生吗？

答：不可能，三点保证不会出现这种情况，事务请求都要经过leader的单一队列，leader向follow而顺序发送commit消息，follower存放待提交事务是单一队列的

11、如果AB两台已经做了follower和leader，第三台机器epoch很大，zk进程启动后会不会成为leader？会报告异常？

答： 第三台机器会一直死循环下去，java.io.IOException

12、client所有操作均报告connectionLoss异常，并且日志中发现一直在重连，这种情况是怎么导致的？

答：这表明zookeeper service不可用，有两种情况都会导致这一现象的发生，一是zookeeper 集群都在参与选举，所有zk server都无法提供服务，另外一种情况是某个事务较长时间内未能收到足够的响应，导致所有写请求阻塞，事务发起server接收到的读写请求阻塞。这是zookeeper应该改进的地方，至少让client知道server的状态。

13、zookeeper是否支持动态扩容？

答：稳定版3.4.6及之前的版本不支持，修改zoo.cfg文件只能重启才生效。zookeeper已经在3.5.0版本支持动态扩容了，但还属于内测版本

参考：https://issues.apache.org/jira/browse/ZOOKEEPER-107

设计文档：

https://issues.apache.org/jira/secure/attachment/12525379/zkreconfig-usenixatc-final.pdf

在3.4.6版本，扩容是这样的，比如原有3台机器

server.1=192.168.137.10:2888:3888

server.2=192.168.137.11:2888:3888

server.3=192.168.137.12:2888:3888

新加入4，5，配置如下

server.1=192.168.137.10:2888:3888

server.2=192.168.137.11:2888:3888

server.3=192.168.137.12:2888:3888

server.4=192.168.137.13:2888:3888

server.5=192.168.137.14:2888:3888

先将4加入启动，并执行echo stat|nc 192.168.137.13 2181执行看MOde是否是follewer，代表完成数据同步，之后以同样方法加入5。修改1，2，3机器的配置，同时重启1,2,3完成扩容过程。

如果防止在同时重启遇到未能预料到的问题，那么可以改用依次重启，比如3是leader，1，2是follower，先依次重启1，2，并使用echo stat|nc 192.168.137.xx 2181确保重启处于follower，然后关闭3，待1,2,4,5选举出一个leader以后再将3启动，最终确保所有server都处在follower或leader模式后结束整个扩容过程。之后就是告诉client更新server列表的事情了

相比同时重启，依次重启并不能避免leader选举，只不过能实时监控各个server的状态，缩短由重启(重启需要从日志中恢复数据到内存)带来的不可用时间窗口，避免同时重启过程中比如数据恢复阶段同时遇到问题加大整个集群的不可用时长。

# 参考

zookeeper官网

《Zookeeper Distributed Process Coordinatin》

《从PAXOS到Zookeeper：分布式一致性原理与实践》

《Zab：High-performance broadcast for primary-backup system》

<http://csrd.aliapp.com/?p=631>

http://timyang.net/tag/paxos/

https://cwiki.apache.org/confluence/display/ZOOKEEPER/FAQ