1 概述

zookeeper 是 Hadoop 的正式子项目，是一个针对大型分布式系统的可靠协调系统。zookeeper 是 ZDH 系统中的一个主要组件，它能为 HDFS、HBase、MapReduce、Yarn、Hive 等 ZDH 组件提供重要的功能支撑。在分布式应用中，通常需要 Zookeeper 来提供可靠的、可扩展的、分布式的、可配置的协调机制来统一各系统的状态。

1.1 什么是 zookeeper

zookeeper 英文名为动物园管理员. 这是因为 Hadoop 生态系统中很多组件的名字都是动物，hadoop 本身就是小象的意思，还有 hive 小蜜蜂，pig, 而 zookeeper 就是为这些组件提供协同服务的能力.

zookeeper 为分布式系统提供了高效可靠且易于使用的协同服务，它还可以为分布式应用提供相当多的服务，诸如统一命名服务，配置管理，状态同步和组服务等。

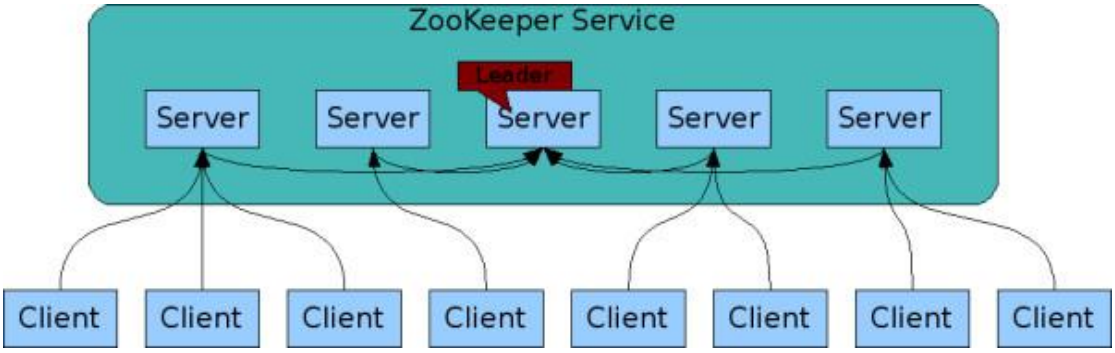
1.2 zookeeper 的架构

Zookeeper 中的角色主要有以下三类，如下表所示：

角色	描述
领导者(leader)	负责投票的发起和决议，更新系统状态

学习者(learner)	跟随者(follower)	接收客户请求，并向客户端返回结果，在选举 leader 过程中参与投票
	观察者(Observer)	可以接收客户端连接，将写请求发送给 leader。Observer 不参与投票，只同步 leader 数据。设置 observer 的目的是为了扩展系统，提高系统的读性能，
客户端(Client)		请求发起方

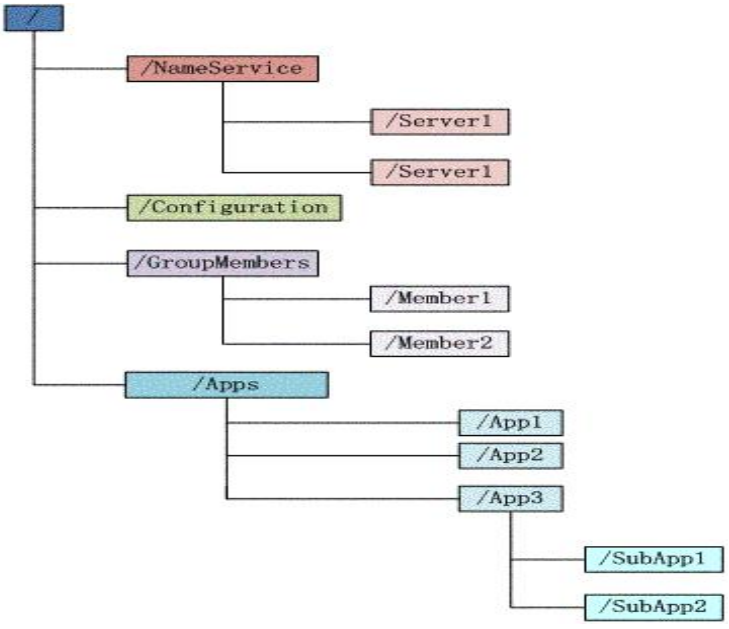
系统模型如图所示：



1. 客户端可以连接到每个 server，每个 server 的数据完全相同;
2. 每个 follower 都和 leader 有连接，接受 leader 的数据更新操作;
3. Leader 只有一个，宕机后会重新选出一个 Leader;
4. Server 记录事务日志和快照到持久存储;
5. 过半数 server 可用，整体服务就可用;
6. 读写操作的速率比为 10:1， Zookeeper 设计的读写频率比是 2:1;

1.3 zookeeper 的数据模型

数据模型如下示：



解读：

1、基于树型结构的命名空间，与文件系统类似；

2、每个子目录项如 NameService 都被称作为 znode，这个 znode 是被它所在的路径唯一标识，如 Server1 这个 znode 的标识为/NameService/Server1；

3、znode 节点不支持重命名；

4、znode 可以有子节点目录，并且每个 znode 可以存储数据，注意 EPHEMERAL 类型的目录节点不能有子节点目录；

5、数据大小不超过 1MB；

6、数据读写要保证完整性；

7、znode 可以是临时节点。 Zookeeper 的客户端和服务器通信采用长连接方式，每个客户端和服务器通过心跳来保持连接，这个连接状态称为 session，如果 znode 是临时节点，当这个 session 失效，znode 也就删除了；

8、znode 的目录名可以自动编号，创建的时候指定 SEQUENTIAL 方式,如 App1 已经存在的话，将会自动命名为 App2；

9、znode 可以被监控，包括这个目录节点中存储的数据的修改，子节点目录的变化等，一旦变化可以通知设置监控的客户端，这个是 Zookeeper 的核心特性，Zookeeper 的很多功能都是基于这个特性实现的，通过这个特性可以实现的功能包括配置的集中管理，集群管理，分布式锁等等。

1.4 zookeeper 能做什么

- **Master-Work 架构的协同服务**:
  - **Master 选举**: 让可用的 Master 分配任务给 Worker 的过程是至关重要的。
  - **崩溃检测**: Master 必须能检测 Worker 什么时候崩溃或者不可连接。
  - **组成员管理**: Master 必须能知道哪些 Worker 可以执行任务。
  - **元数据管理**: Master 和 Worker 必须能以可靠的方式存储任务分配和执行状态的信息。

  Master-Worker 架构，如下三个问题至关重要：
  1. Master 崩溃: Master 一旦崩溃，系统将不能在分配新任务或者重新分配来自 Worker 的任务。
  2. Worker 崩溃: Worker 一旦崩溃，那么分配给它的任务将不能被完成。
  3. 通信失败: Master 和 Worker 之间不能交换信息，Worker 将不能学习分配给它的新任务。

- **分布式锁**: 这个主要得益于 zookeeper 为我们保证了数据的强一致性。锁服务可以分为两类，一个是保持独占，另一个是控制时序。
  - **独占锁**: 就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法是把 Zookeeper 上的一个 znode 看作是一把锁，通过 create znode 的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这

把锁。

- **时序锁**：就是所有视图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局时序了。做法和上面基本类似，只是这里/distribute_lock 已经预先存在，客户端在它下面创建临时有序节点（这个可以通过节点的属性控制：CreateMode.EPHEMERAL_SEQUENTIAL 来指定）。zookeeper 的父节点（/distribu te_lock）维持一份 sequence,保证子节点创建的时序性，从而也形成了每个客户端的全局时序。

- **配置同步**：各客户端设置 watch 监视存放配置数据的 znode 节点. 一旦数据有变化时,所以监视的 watch 都会收到消息,从而触发配置数据更新.

- **服务发现**：一个分布式的 SOA 架构中，服务是一个集群提供的，当消费者访问某个服务时，就需要采用某种机制发现现在有哪些节点可以提供该服务(这也称之为服务发现，比如 Alibaba 开源的 SOA 框架 Dubbo 就采用了 Zookeeper 作为服务发现的底层机制)。还有开源的 Kafka 队列就采用了 Zookeeper 作为 Cosnumer 的上下线管理。

基于 zookeeper 实现的服务发现功能存在一个隐患：在 ZooKeeper 中，网络分区中的客户端节点无法到达 Quorum 时，就会与 ZooKeeper 失去联系，从而也就无法使用其服务发现机制。
虽然可以通过客户端缓存和其它技术弥补这种缺陷，但这并不能从根本上解决问题，如果 Quorum 完全不可用，或者集群分区和客户端都恰好连接到了不属于这个 Quorum 但仍然健康的节点，那么客户端状态仍将丢失。
服务发现建议可以考虑具备尽可能高的可用性和恢复能力的 Eureka.

1.5 zookeeper 不能做什么

- **不适合用于大容量存储**：建议 zookeeper 只存放协调和控制的数据，业务数据可以考虑使用数据库或者分布式文件系统等。
- **不适合对服务可用性要求高的场景**：zookeeper 不能保证服务的高可用性,其在极端情况(比如网路分割)下可能会丢弃一些请求,消费者程序需要重新请求才能获得结果.

分布式系统领域有个著名的 CAP 定理(C- 数据一致性;A-服务可用性;P-服务对网络分区故障的容错性，这三个特性在任何分布式系统中不能同时满足，最多同时满足两个);ZooKeeper 是个 CP 的，即任何时刻对 ZooKeeper 的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性;但是它不能保证每次服务请求的可用性.

1.6 zookeeper 的分布式能力

分布式系统的好处是:
1. 一台服务器的性能不足以提供足够的能力服务于所有网络请求;
2. 防止服务器宕机，造成服务不可用或是数据丢失.

分布式系统的 CAP 理论的三个特性：

分布式系统的 CAP 理论的三个特性，这三个特性在任何分布式系统中不能同时满足，最多同时满足两个：

- **一致性(Consistency)**：任何一个读操作总是能读取到之前完成的写操作结果，也就是在分布式环境中，多点的数据是一致的。
- **可用性(Availability)**：每一个操作总是能够在确定的时间内返回，也就是系统随时都是可用的。在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（可用性不仅包括读，还有写）
- **分区容忍性(Partition tolerance)**：　集群中的某些节点在无法联系后，集群整体是否还能继续进行服务。

ZooKeeper 是个 CP 的系统，即任何时刻对 ZooKeeper 的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性;但是它不能保证每次服务请求的可用性.

为了保证数据的一致性，zookeeper 实现的是一个"**写任意**"，"**立即复制**" 的分布式系统.

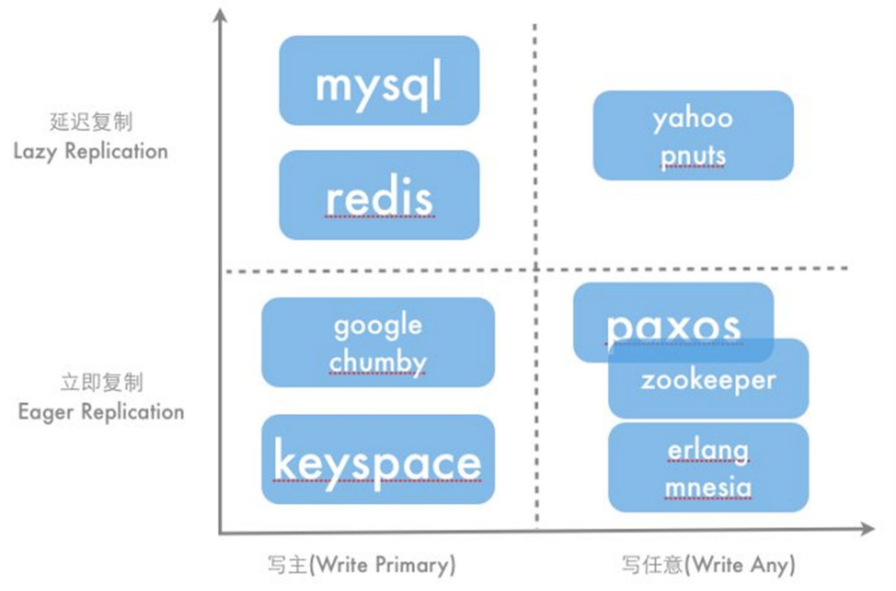
数据复制集群系统的分类

从客户端读写访问的透明度来看：

1. **写主 (WriteMaster)**：数据的修改提交给指定的节点。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称读写分离；
2. **写任意 (Write Any)**：对数据的修改可提交给任意的节点，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

从客户端访问的响应速度来看：

1. **延迟复制 (Lazy Replication)**：节点完成修改操作后，直接给用户返回成功的消息。修改的数据延迟复制到别的节点上，客户端不需要为数据复制过程付出等待时间。当节点分布在不同的地域上的时候，客户端就近提交，很快就能返回，响应速度是最快的；
2. **立即复制 (Eager Replication)**：节点完成修改后，还要等别的节点都完成修改才给用户返回成功的消息。客户端需要为数据的复制过程付出等待时间。当节点分布在不同的地域上的时候，客户端虽能就近提交，但仍享受不到就近提交的快速响应的好处了；



数据复制集群系统的一致性：

1. **从客户端来看**：一致性主要指的是多并发访问时更新过的数据如何获取的问题.
   a) **强一致性**：系统中的某个数据被成功更新(事务成功返回)后，后续任何对该数据的读取操作都得到更新后的值。这是传统关系数据库提供的一致性模型，也是关系数据库深受人们喜爱的原因之一。
   b) **弱一致性**：系统中的某个数据被更新后，后续对该数据的读取操作得到的不一定是更新后的值。
   c) **最终一致性**：属于弱一致性，不同的是，最终一致性的系统会在后台异步地更新所有的备份，所以最终所有的备份都会是最新的数据。这种情况下通常有个“不一致性时间窗口”(inconsistency window)存在：即数据更新完成后在经过这个“不一致性时间窗口”，后续读取操作就能够得到更新后的值.
2. **从服务端来看**：一致性是指更新如何复制分布到整个系统，以保证数据最终一致。
   N — 数据复制的份数
   W — 更新数据是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
   a) **强一致性**：如果 W+R>N，写的节点和读的节点重叠，则是强一致性。例如对于典型的一主一备同步复制的关系型数据库，N=2,W=2,R=1，则不管读的是主库还是备库的数据，都是一致的.
   b) **弱一致性**：如果 W+R<=N，则是弱一致性。例如对于一主一备异步复制的关系型数据库，N=2,W=1,R=1，则如果读的是备库，就可能无法读取主库已经更新过的数据，所以是弱一致性。

   对于分布式系统，为了保证高可用性，一般设置 N>=3。不同的 N,W,R 组合，是在可用性和一致性之间取一个平衡，以适应不同的应用场景。

   对于 zookeeper 系统，为了提供性能，采用的是 N=server 数，W=N/2+1，R=1.

zookeeper 采用的 zab 协议是基于 paxos 算法的,算是"**强一致性**"的同步.

因为 Zookeepr 对写请求进行投票表决的时候是过半数就算表决成功，可以处理并返回给 client 的. 在 client 收到写成功回应后可能有小部分 server 尚未更新完数据(如网络延迟较大). 因此在当需要最新数据时，应该在读数据之前调用 sync 同步一下. 但是一旦 client 收到成功回应，那可以确认至少有过半数的节点已完成数据的更新，一旦集群宕机,数据也是能正常恢复的. 而且这过半数节点中 Leader 和 client 所连接的节点必定已经完成了数据的更新.

Zookeeper 的一致性要强于最终一致性，而稍弱于强一致性.

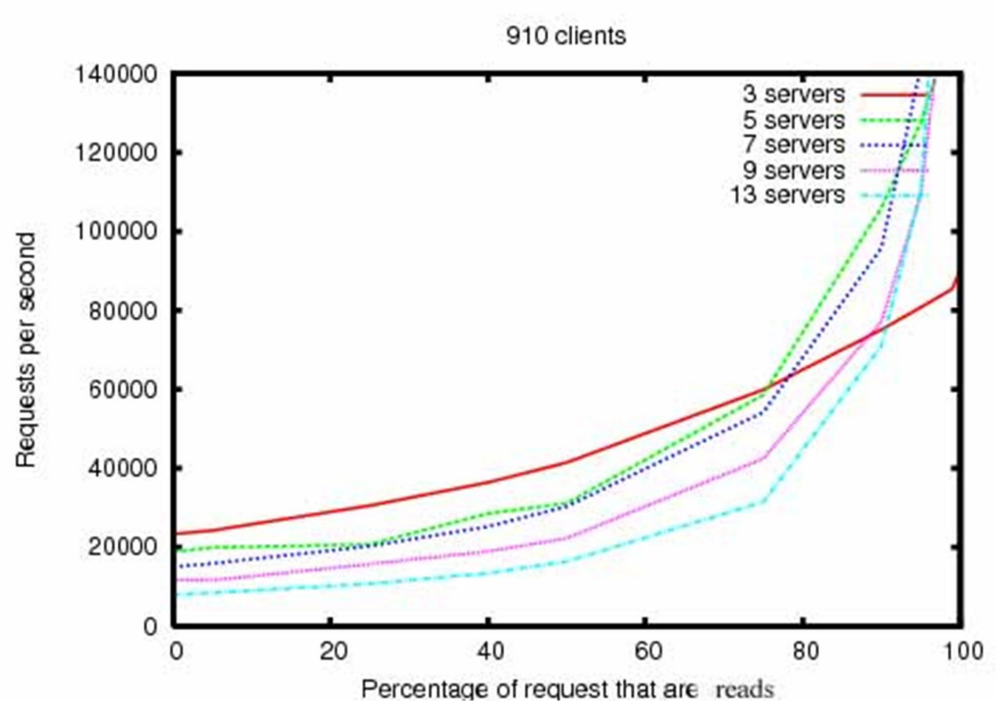
1.7 zookeeper 的特性

Zookeeper 有如下基本特性：

- **强一致性**：Client 不论连接到哪个 Server，展示给它的都是同一个视图，这是 Zookeeper 最重要的性能。
- **可靠性**：具有简单、健壮、良好的性能，如果消息 Message 被一台服务器接受，那么它将被所有的服务器接受。
- **实时性**：Zookeeper 保证客户端在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。考虑到网络延时等原因，在需要最新数据时，应该在读数据之前调用 sync 实现。
- **等待无关（wait-free）**：慢的或者失效的 Client 不得干预快速的 Client 的请求，使得每个 Client 都能有效的等待。
- **原子性**：更新只能成功或者失败，没有中间状态。
- **顺序性**：包括全局有序和因果顺序两种。
  - 全局有序是指如果在一台服务器上消息 a 在消息 b 前发布，则在所有 Server 上消息 a 都将在消息 b 前被发布；
  - 因果顺序是指如果一个消息 b 在消息 a 后被同一个发送者发布，a 必将排在 b 前面被处理。

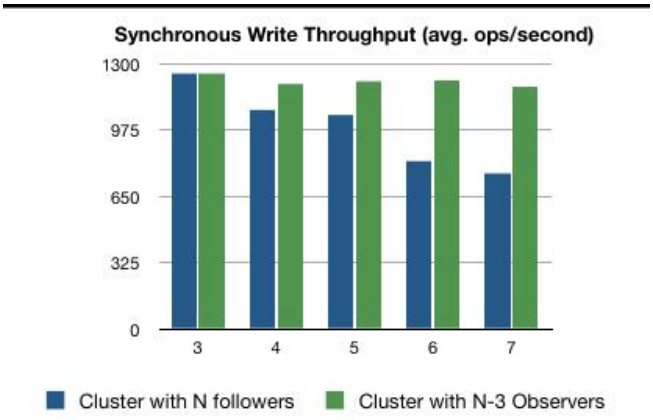
1.8 Zookeeper 的性能

zookeeper 在读多于写(通常是协调服务)的情景下性能很高



通过增加机器，它的读吞吐能力和响应能力扩展性非常好；

而写操作，随着机器的增多吞吐能力肯定下降（这也是它建立 observer 的原因），决策群的大小决定着决议收敛的快慢. 为此需要限制 FOLLOWERS 的数量，以 OBSERVER 代替.



1.9 zookeeper 小结

Zookeeper 是一个高性能，分布式的，开源分布式应用协调服务。

它提供了简单原始的分布式类小容量文件系统，分布式应用可以基于它实现更高级的服务，比如同步，配置管理，集群管理，命名空间等。

2 设计思想

zookeeper 是使用的 paxos 算法来解决分布式节点数据一致性的问题.

2.1 paxos 算法

paxos 算法解决的问题是：让每一个处于正常工作的服务端都执行一个相同的命令序列.

在分布式系统中，有多个 Client 和多个 Server。 每个 client 都可以和每个 server 进行交互。假设每个 Client 此时各发一个请求，分别是 01，02，03，04。那么在每个服务器上看到的命令顺序可能是不一样的。
Paxos 算法就是要保证每个服务端接受的命令顺序是完全一样的。
Paxos 考虑的问题是，在异步通信过程中，发送的数据可能会被丢失(lost)、延长(delayed)、重复(duplicated)，但不会出现被篡改。同时任意一个服务端不会出现拜占庭将军问题（Byzantine failure），可以简单地理解为结点群在决定命令序列的过程中没有结点受病毒、黑客影响。

2.1.1 算法的逻辑角色

- proposer: 提出提案，提案信息包括提案编号和提议的 value;
- acceptor: 收到提案后可以接受(accept)提案;

- learner：只能"学习"被批准的提案；

2.1.2 算法保持一致性的基本语义

- 决议(value)只有在被 proposers 提出后才能被批准(未经批准的决议称为"提案(proposal)")；
- 在一次 Paxos 算法的执行实例中，只批准(chosen)一个 value；
- learners 只能获得被批准(chosen)的 value；

2.1.3 算法的四个约束

由上面的三个语义可演化为四个约束：

- P1：一个 acceptor 必须接受(accept)第一次收到的提案；

P1 是不完备的。如果恰好一半 acceptor 接受的提案具有 value A，另一半接受的提案具有 value B，那么就无法形成多数派，无法批准任何一个 value。
语义 2 并不要求只批准一个提案，暗示可能存在多个提案。只要提案的 value 是一样的，批准多个提案不违背约束 2。于是可以产生约束 P2：

P2：一旦一个具有 value v 的提案被批准(chosen)，那么之后批准(chosen)的提案必须具有 value v。

通过某种方法可以为每个提案分配一个编号，在提案之间建立一个全序关系，所谓“之后”都是指所有编号更大的提案。
如果 P1 和 P2 都能够保证，那么约束 2 就能够保证。
批准一个 value 意味着多个 acceptor 接受(accept)了该 value. 因此,可以对 P2 进行加强：

- P2a：一旦一个具有 value v 的提案被批准(chosen)，那么之后任何 acceptor 再次接受(accept)的提案必须具有 value v；

由于通信是异步的，P2a 和 P1 会发生冲突。如果一个 value 被批准后，一个 proposer 和一个 acceptor 从休眠中苏醒，前者提出一个具有新的 value 的提案。根据 P1，后者应当接受，根据 P2a，则不应当接受，这中场景下 P2a 和 P1 有矛盾。于是需要换个思路，转而对 proposer 的行为进行约束：

- P2b：一旦一个具有 value v 的提案被批准(chosen)，那么以后任何 proposer 提出的提案必须具有 value v；

由于 acceptor 能接受的提案都必须由 proposer 提出，所以 P2b 蕴涵了 P2a，是一个更强的约束。
但是根据 P2b 难以提出实现手段。因此需要进一步加强 P2b。
假设一个编号为 m 的 value v 已经获得批准(chosen)，来看看在什么情况下对任何编号为 n（n>m）的提案都含有 value v。因为 m 已经获得批准(chosen)，显然存在一个 acceptors 的多数派 C，他们都接受(accept)了 v。考虑到任何多数派都和 C 具有至少一个公共成员，可以找到一个蕴涵 P2b 的约束 P2c：

- P2c:如果一个编号为 n 的提案具有 value v，那么存在一个多数派，要么他们中所有人都没有接受(accept)编号小于 n 的任何提案，要么他们已经接受(accpet)的所有编号小于 n 的提案中编号最大的那个提案具有 value v;

要满足 P2c 的约束，proposer 提出一个提案前，首先要和足以形成多数派的 acceptors 进行通信，获得他们进行的最近一次接受(accept)的提案（prepare 过程），之后根据回收的信息决定这次提案的 value，形成提案开始投票。当获得多数 acceptors 接受(accept)后，提案获得批准(chosen)，由 proposer 将这个消息告知 learner。这个简略的过程经过进一步细化后就形成了 Paxos 算法。

2.1.4 决议的提出与批准

通过一个决议分为两个阶段:

- **prepare 阶段:**
  - proposer 选择一个提案编号 n 并将 prepare 请求发送给 acceptors 中的一个多数派;
  - acceptor 收到 prepare 消息后，如果提案的编号大于它已经回复的所有 prepare 消息，则 acceptor 将自己上次接受的提案回复给 proposer，并承诺不再回复小于 n 的提案;

优化: 在 prepare 阶段中，如 acceptor 发现存在一个更高编号的提案，则需要通知提案人，提醒其中断这次提案。

- **批准阶段:**
  - 当一个 proposer 收到了多数 acceptors 对 prepare 的回复后，就进入批准阶段。它要向回复 prepare 请求的 acceptors 发送 accept 请求，包括编号 n 和根据 P2c 决定的 value（如果根据 P2c 没有已经接受的 value，那么它可以自由决定 value）。
  - 在不违背自己向其他 proposer 的承诺的前提下，acceptor 收到 accept 请求后即接受这个请求，如果发现自己有一个更大的提案号，就拒绝修改。

这个过程在任何时候中断都可以保证正确性。

2.1.5 决议的发布

一个显而易见的方法是当 acceptors 批准一个 value 时，将这个消息发送给所有 learner。但是这个方法会导致消息量过大。

由于假设没 Byzantine failures，learners 可以通过别的 learners 获取已经通过的决议。因此 acceptors 只需将批准的消息发送给指定的某一个 learner，其他 learners 向它询问已经通过的决议。这个方法降低了消息量，但是指定 learner 失效将引起系统失效。

因此 acceptors 需要将 accept 消息发送给 learners 的一个子集，然后

由这些 learners 去通知所有 learners。

但是由于消息传递的不确定性，可能会没任何 learner 获得了决议批准的消息。当 learners 需要了解决议通过情况时，可以让一个 proposer 重新进行一次提案。注意一个 learner 可能兼任 proposer。

2.1.6 快速 paxos 算法

Paxos 算法可能出现死循环，就是在两个 Proposer 总是在交替 prepare。并且，Paxos 算法在出现竞争的情况下，其收敛速度很慢，甚至可能出现活锁的情况，例如当有三个及三个以上的 proposer 在发送 prepare 请求后，很难有一个 proposer 收到半数以上的回复而不断地执行 prepare。因此，为了避免竞争，加快收敛的速度，在算法中引入了一个 Leader 这个角色，在正常情况下同时应该最多只能有一个参与者扮演 Leader 角色，而其它的参与者则扮演 Acceptor 的角色，同时所有的人又都扮演 Learner 的角色。

快速优化算法内容:

- 只有 Leader 可以提出议案，从而避免了竞争使得算法能够快速地收敛而趋于一致，此时的 paxos 算法在本质上就退变为两阶段提交协议。
- 在异常情况下，系统可能会出现多 Leader 的情况，这并不会破坏算法对一致性的保证，此时多个 Leader 都可以提出自己的提案，优化的算法就退化成了原始的 paxos 算法。

一个 Leader 的工作流程主要有分为三个阶段：

- **学习阶段**: 向其它的参与者学习自己不知道的数据(决议);当一个参与者成为了 Leader 之后，它应该需要知道绝大多数的 paxos 实例，因此就会马上启动一个主动学习的过程。假设当前的新 Leader 早就知道了 1-134、138 和 139 的 paxos 实例，那么它会执行 135-137 和大于 139 的 paxos 实例的第一阶段。如果只检测到 135 和 140 的 paxos 实例有确定的值，那它最后就会知道 1-135 以及 138-140 的 paxos 实例。
- **同步阶段**: 让绝大多数参与者保持数据(决议)的一致性;此时的 Leader 已经知道了 1-135、138-140 的 paxos 实例，那么它就会重新执行 1-135 的 paxos 实例，以保证绝大多数参与者在 1-135 的 paxos 实例上是保持一致的。至于 139-140 的 paxos 实例，它并不马上执行 138-140 的 paxos 实例，而是等到在服务阶段填充了 136、137 的 paxos 实例之后再执行。这里之所以要填充间隔，是为了避免以后的 Leader 总是要学习这些间隔中的 paxos 实例，而这些 paxos 实例又没有对应的确定值。
- **服务阶段**: 为客户端服务，提案;Leader 将用户的请求转化为对应的 paxos 实例，当然，它可以并发的执行多个 paxos 实例，当这个 Leader 出现异常之后，就很有可能造成 paxos 实例出现间断。

2.2 ZAB 协议

zookeeper 的核心是原子广播，这个机制保证了各个 Server 之间信息的同步。实现这一机制的协议叫做 Zab 协议，通信协议为 TCP/IP。

zab 协议有两种模式，它们分别是恢复模式(选主)和广播模式(同步)。

- **恢复模式**：当服务启动或者在领导者崩溃后，进入了恢复模式，将集群中节点恢复到一致的状态；当领导者被选举出来，且大多数 Server 完成了和 leader 的状态同步后，恢复模式结束。
- **广播模式**：恢复模式结束后进入广播模式。 所有的变更请求都提交到 Leader 上。 Leader 发起广播提议，当过决策群中过半数响应后在广播决议。该模式保证了 leader 和 learner 具有相同的状态变化。

ZAB 协议最重要的就是"**过半可用**"：
恢复模式中使用 paxos 进行选举 leader 时，多数派为过半 server. 当超过一半的 server 认为某个 server 为 leader 时则决议被批准；
广播模式中 paxos 退化为两阶段提交，决议需要得到过半数的 server 许可才能得到批准.

对 Zab 设计而言很关键的一点是：

对于每个状态变化的观察，都是基于上一个状态的增量变化。这也就意味着对于状态变化，必须依赖于其更新的次序。

状态改变过程是不能乱序的。只要增量的发送是有序的，状态的更新便是幂等的，即使某个增量变化被反复执行多次。

因此，对于消息系统而言，保证 at-least once 语义已经足够，以此来简化整个 Zab 的实现。

状态变化的有序分为全序和因果顺序：

- **全序顺序**：TCP 协议保证了消息的全序特性(先发先到)；
- **因果顺序**：引入 Leader 和 zxid 解决了因果顺序问题(变更消息上由 zxid 排序，leader 保证了 zxid 的顺序).

2.2.1 保证因果顺序的 Zxid

Zxid 是 zookeeper 中为了保持一致性而引入的一个重要概念.

其为每个变更操作都添加了一个标识. 而且后发的变更操作是 zxid 比之前的 zxid 大. 通过 zxid,Follower 能很容易发现请求的顺序，从而拒绝老的请求.

zxid 为一 64 位数字：

- 高 32 位为 leader 信息又称为 epoch，每次 leader 转换时递增；
- 低 32 位为消息编号，Leader 处理写消息时应该从 0 开始递增.

后 32 位溢出时候，leader 会通知所有节点中断服务，进入 looking 状态，进行新一次的 leader 选举。

选举后 Epoch 产生的策略：
在启动 zookeeper 时，根据最新的 snapshot 目录下有两个文件：acceptedEpoch、currentEpoch，其中 acceptedEpoch 表示节点接收的推荐 leader 的 epoch，currentEpoch 为上一次接收的 leader 的 Epoch。初始状态均为 0。 在稳定运行的集群中，acceptedEpoch 和 currentEpoch 是一样的。在进行下一次投票后，currentEpoch 是 acceptedEpoch 和 currentEpoch 间的最大值+1。在角色选举完成后，zxid 的前 32 为取 currentEpoch 值，后 32 为置为 0。开始接收客户端的请求。当一个新的 epoch 生成，新的 leader 会被激活。

3 工作流程概述

Zookeeper 为提供分布式协同服务,需要解决如下几个问题:

1. server 上电是如何恢复历史数据?
2. 集群中的 servers 如何在某个时候能达到一致的状态?
3. 集群中的 servers 在达到一致后,执行修改数据的操作还能保持一致?
4. 如何实现 EPHEMERAL 节点的功能, 使得连接断开时该 znode 自动删除?
5. znode 数据变化时 client 如何感知?

zookeeper 集群工作的过程包括如下两步:
1. **Recovery 过程**: 这个过程泛指集群服务器的启动和恢复,因为恢复也可以理解为另一种层面上的"启动"——需要恢复历史数据的启动。
2. **Broadcast 过程**: 这是启动完毕之后,集群中的服务器开始接收客户端的连接一起工作的过程,如果客户端有修改数据的改动,那么一定会由 leader 广播给 follower, 所以称为"Broadcast".

展开来说, zookeeper 集群大概是这样工作的:
1. 首先每个服务器读取配置文件和数据文件。加载 zoo.cfg 配置文件的内容，知道本机对应的配置。在实质启动阶段先把 dataTree 加载进内存，获取最新的 zxid，通过获取的 zxid 值判断 epoch 值 epochOfZxid，读取 currentEpoch 文件得 currentEpoch，读取 acceptedEpoch 文件得到 acceptedEpoch 值，比较 epochOfZxid、currentEpoch、acceptedEpoch，取其中的最大值 epochMax 值，用 epochMax 值写 currentEpoch、acceptedEpoch 文件，以备调用。在初次启动时候，epochOfZxid、currentEpoch、acceptedEpoch 初始状态均为 0。
2. 在上述准备工作完毕，开始进入选举阶段。选举阶段先是要初始选票，读入准备好的 zxid 值、currentEpoch、acceptedEpoch 值，读入合法的 sid 值，组装成选票，开始投票选举 leader。集群中的服务器开始根据前面给出的 quorum port 端口, 监听集群中其他服务器的请求,并且把自己选举的 leader 也通知其他服务器。循环数次,选举出集群的 leader。
3. 选举完 leader 其实还不算是真正意义上的”leader”。因为到了这里 leader 还需要与集群中的其他服务器同步数据。如果这一步出错,将返回步骤 2 中重新选举 leader。在 leader 选举完毕之后,集群中的其他服

务器称为”follower”,也就是都要听从 leader 的指令。

4. 集群中的所有服务器，不论是 leader 还是 follower，大家的数据一致了,可以开始接收客户端的连接了。如果是读类型的请求,那么直接返回就是了,因为并不改变数据；否则,都要向 leader 汇报。leader 收到这个修改数据的请求之后,将会广播给集群中其他 follower，当超过一半数量的 follower 有了回复，那么就相当于这个修改操作被批准了。这时 leader 可以反馈被请求的 follower 节点，可以给客户端一个回应。

上面 1，2，3 步骤对应的 recovery 过程,而 4 对应的 broadcast 过程。
步骤 1 解决问题"server 上电是如何恢复历史数据？"；
步骤 3 解决问题"集群中的 servers 如何在某个时候能达到一致的状态?"；
步骤 4 解决问题"集群中的 servers 在达到一致后,执行修改数据的操作还能保持一致?".

zookeeper 集群为提供给业务使用时,提供了 client 这样一个对外提供 znode 数据查询修改的接口.

其通过 NIO 长连接连接到 server 上，并对 znode 进行操作.
当 client 和 server 建立连接的时候设置好超时时间 sessionTimeout. 然后 client 发送请求或周期性 ping 以推迟 session 的过期时间. 一旦 server 发现 session 已经过期就通知 client，并将该 session 对应的 EPHEMERAL 节点删除.
client 可以针对 znode 路径设置 watcher，并通知 server，当该 znode 有变化时将通知 client，client 收到通知消息后将会触发设置 watcher 的处理.

其中只存活于 session 生命周期内的 EPHEMERAL 节点功能，可以解决问题"如何实现 EPHEMERAL 节点的功能，使得连接断开时该 znode 自动删除？"；watcher 机制用于解决问题" znode 数据变化时 client 如何感知?".

4 数据持久化

zookeeper 主要持久化了两类数据:

- **snapshot**: 存放某个时刻的内存数据快照;
- **txnlog**: 存放某段时间与修改数据相关的操作.

zookeeper 为了对分布式集群提供协同服务，存储了一定的集群控制数据.当 zookeeper 集群节点异常恢复时需要保证之前表决过的数据不能丢失. 为此设计了 snapshot + txnlog 相结合来恢复数据的方式.

- zookeeper 总数据量不大，所以其可以很快的作出一个 snapshot;
- 当需要恢复数据时,只需要读取最近的一个 snapshot，并将在 snapshot 时间点之后的 txnlog 中的操作全部重跑一遍，就可以得到最新的数据;
- snapshot 生成时未将内存的数据锁死，在该过程中仍然是可以更新数据的. 这样 snapshot 其实并不完全是该时间点的内存数据，但是由于 ZAB 协议中状态变化是幂等的，所以状态变化按顺序执行的话是可以保

证数据一致性的.

生成快照时,为了提高 server 的性能，并未对内存数据 Datatree 和 sessionTracker 加锁.

4.1 内存数据树

zookeeper 的数据模型是一个树形的命名空间，为此在内存中也是以树形结构存储的.

持久化模块 ZKDataBase 是 zookeeper 维护在内存和日志中的数据，含如下几部分内容：

- DataTree（内存树）：zookeeper 所有数据节点(dataNode)路径映射出来的 hash 表，和数据节点的树状架构。用户访问的是 hash 表，hash 表和树状结构都会被持久化到硬盘中:

DataTree 数据模型:

- DataNode: (1:n)所有数据节点的 hash 表
- dataWatchManager: (1:1)处理 node 节点的变更事件,发送 Watcher
- childWatchManager: (1:1)处理 node 子节点的变更事件,发送 Watcher
- Ephemerals：会话信息节点，hash 表。在每次会话建立，zookeeper 会给客户端分配一个 sessionID，如果建立一个 ephemeral 节点，当会话超时或 session 结束，zookeeper 会删除 ephemeral 节点。

DataNode 模型:

- Parent: 父节点，数据类型 DataNode
- data byte[]: 本数据节点记录的数据信息，数据类型 bytes[]
- acl: 访问控制信息
- stat: 审计信息. 记录了本数据节点一些属性信息，如：建立时间、建立时刻 zxid、更改时间、更改时刻 zxid、版本等
- children: 点的子节点，为 DataNode 列表

- committedLog: 默认存储 500 条变更记录(该数据不持久化)，用于加速 Leader 和 Learners 同步(当 learner 的 zxid 在内存 committedLog 中时直接发送需要更新的事务列表，而不需要序列化当前 Datatree). 只在内存中

- SnapShot: 定期对 DataTree 的数据做一个本地备份。

- TxnLog: 是一些历史的版本变更日志(每次有写事件变化，就会写入到该日志中)。

SnapShot 和 TxnLog 两类文件用于 server 恢复时能将 DataTree 恢复到某个 zxid 对应的数据状态.

4.2 快照持久化格式

生成时采用DataTree的lastZxid作为文件名后缀(类似于snapshot.zxid), 并遍历DataTree, 将其中每个节点依次保存.

- FileHeader: 快照文件头, 包含如下属性

```
magic(int): "ZKSN"的int值
version(int): 默认为2
dbid(Long): 默认为-1
```

- snapEntry: 快照体, 包含如下属性

```
sessionCount(Int): session个数
sessionsEntry: 会话数据, 按session个数依次存放
    id(Long): session Id
    timeout(Int): session timeout
aclMapSize(Int): aclMap的大小
aclMapEntry: acl映射数据
    aclId(Long):acl标识
    acl(List{perms(Int), scheme(String), id(String)}): acl数据
dataNodeEntry: 节点实例.遍历DataTree, 将每个节点依次保存.
    path(String): 节点路径(对于根路径"/", 保存为"").
    dataNodeData
        data(Buffer): 节点上保存的数据信息
    acl(Long): acl Id,可从aclMap得出真实acl
    StatPersisted: 审计信息
        czxid(Long): 节点创建zxid
        mzxid(Long): 节点修改zxid
        ctime(Long): 节点创建时间(ms)
        mtime(Long): 节点修改时间(ms)
        version(Int): 节点变更版本
        cversion(Int): 子节点变更版本
        aversion(Int): acl变更版本
        ephemeralOwner(Long): 非临时节点为0,
                              临时节点为节点的sessionId
       pzxid(Long): 子节点变更zxid
EOR(String): 结束符, 固定字符串"/"
```

- entryCRC(Long): 采用算法Adler32得到FileHeader+snapEntry的校验码.
- EOR (String): 结束符,固定字符串"/"

```
查看快照:
java -Djava.ext.dirs=../lib -cp ../zookeeper-3.4.5.jar
org.apache.zookeeper.server.SnapshotFormatter ../../data/version-2/snapshot
```

```
.300000001
```

4.3 事务日志持久化格式

每次有写事务通过 SyncRequestProcessor 时, 将该写事务记录到文件中. 文件初始 64M(可通过 zookeeper.preAllocSize 配置). 顺序写文件, 当剩余容量小于 4K 时以 64M 进行扩展.

日志文件创建时采用第一个事务 txn 的 zxid 作为文件名后缀(类似于 log.zxid).

- FileHeader: 日志文件头, 包含如下属性

```
magic(int): "ZKLG"的 int 值
version(int): 默认为 2
dbid(Long): 默认为 0
```

- txnEntry: 日志体. 其中一个事务日志文件中会包含多个 txnEntry.

```
txnEntryCRC(Long): 采用算法 Adler32 得到 txnEntryData 的校验码
txnEntryLen(Int): txnEntryData 内容的长度
txnEntryData
    txnHeader: 事务头数据
          clientId(Long),
          cxid(Int),
          zxid(Long),
          time(Long),
          type(Int)
    txn: 事务内容数据. 如下的变更事务中某个事务.
        CheckVersionTxn:
            path(String),
            version(Int)
        CreateSessionTxn:
            timeOut(int)
        CreateTxn:
            path(String),
            data(Buffer),
            acl(List{perms(Int), scheme(String), id(String)}),
            ephemeral(bool),
            parentCVersion(Int)
        CreateTxnV0:
            path(String),
            data(Buffer),
            acl(List{perms(Int), scheme(String), id(String)}),
            ephemeral(bool)
        DeleteTxn:
            path(String)
```

```
        ErrorTxn:
            err(Int)
        MultiTxn:
            txns(List{Txn})
        SetACLTxn:
            path(String),
            acl(List{perms(Int), scheme(String), id(String)}),
            version(Int)
        SetDataTxn:
            path(String),
            data(Buffer),
            version(Int)
        SetMaxChildrenTxn:
            path(String),
            max(Int)
EOR (byte):结束符,固定的一个字节作为事务结束符,默认为'B'
```

查看事务日志:

```
java -Djava.ext.dirs=../lib -cp ../zookeeper-3.4.5.jar
org.apache.zookeeper.server.LogFormatter ../../dataLog/version-2/log.500000
001
```

4.4 数据的生成

生成 snapshot 是通过扫描保存在内存中的 Datatree 和 sessionTracker 来把状态持久化的.
生成 txnlog 是通过将 SyncRequestProcessor.queuedRequests 中的请求写到文件中来实现的.

snapshot 和 txnlog 的生成时机有如下几种:

- server 启动时读取持久化的 snapshot 和 txnlog 并 merge 以获得该 server 停止前数据状态. 当加载完成后会生成一个新的 snapshot;
- server 从 looking 切换为 follower/observer 角色时,会和 leader 同步数据, 当数据和 leader 同步一致后会生成一个新的 snapshot;
- server 中 SyncRequestProcessor 中请求队列 queuedRequests 缓存发生的变更请求. 一旦有空闲或 1000 个 request, 将吧缓存的请求 flush 到硬盘生成 txnlog. txnlog 文件最大 64M, 一旦容量超过就会生成一个新的 txnlog 文件.
- 每隔 snapCount/2+随机数(0~snapCount/2)个 request 会重新生成一个 snapshot 并滚动一次 txnlog. 这样可以避免热点,保证服务质量.

4.5 数据的恢复

由于 snapshot 和 txnlog 并非完全同步，通常情况下，txnlog 信息会多于 snapshot，比如 snapshot 记录到了第 80 条事务，但 txnlog 可能记录了 150 条事务，因此在加载的时候，就应该联合 snapshot 和 txnlog.

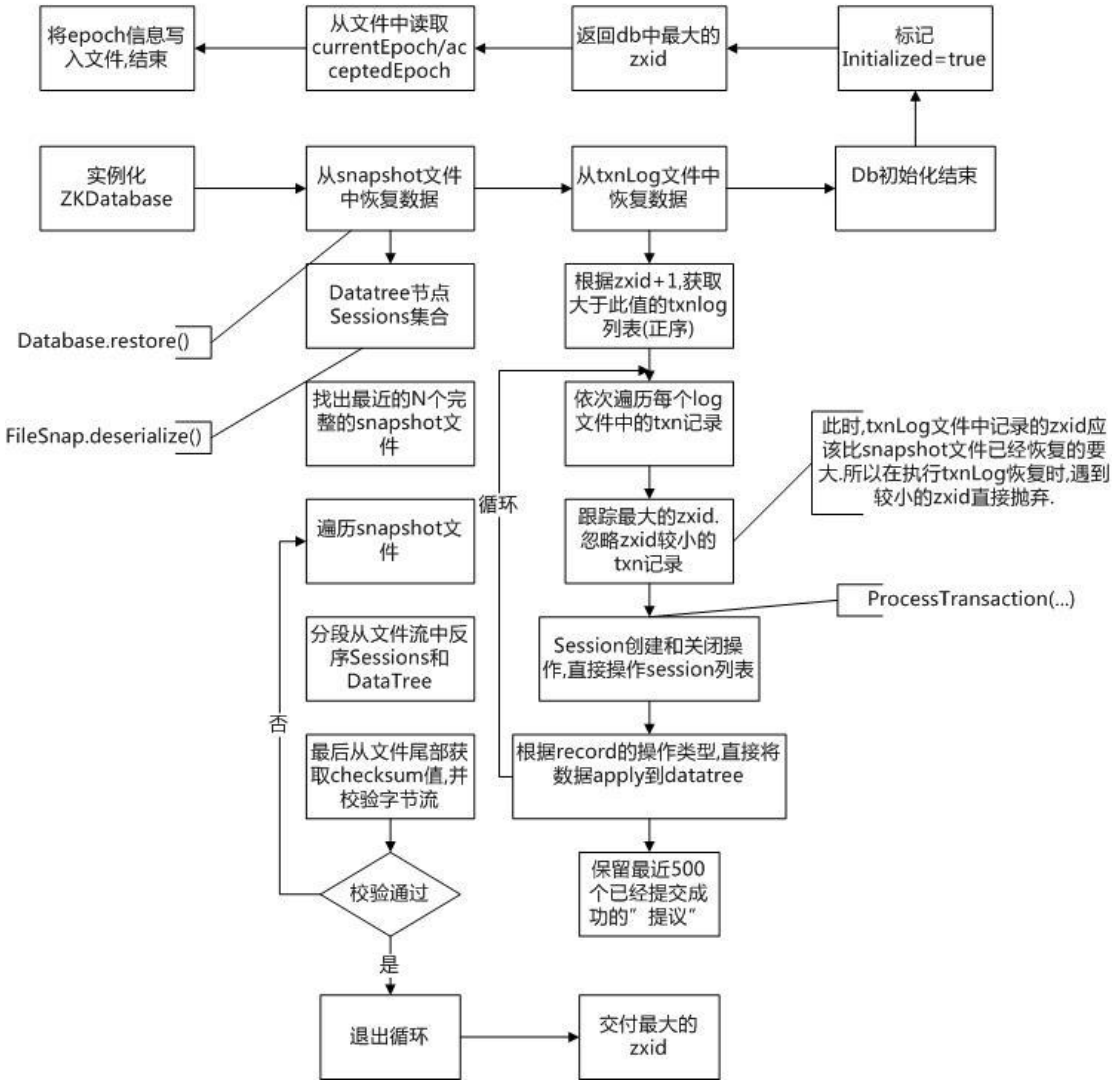
如何联合？举个例子，就拿上面的例子来说，假设共有 150 条事务，并产生了 3 个日志文件，log.1,log.51,log.100

snapshot 记录到了第 80 条,那么就还应该需要 log.51 和 log.100 来做合并，程序会从 log.51 中第 81 条事务开始加载

主要流程:

- 从最新的 100 个 snapshot 中加载可用的最新一个到 DataTree 中, 更新 DataTree.lastProcessedZxid;
- 加载最新的多个 txnlog 文件(一个小于等于 lastProcessedZxid 的 log, 外加其余大于 lastProcessedZxid 的 log);
- 从文件中依次读出事务 txn, 并将大于 lastProcessedZxid 事务代表的变更操作更新到 DataTree 中;
- 更新的同时将事务添加到 committedLog 中.

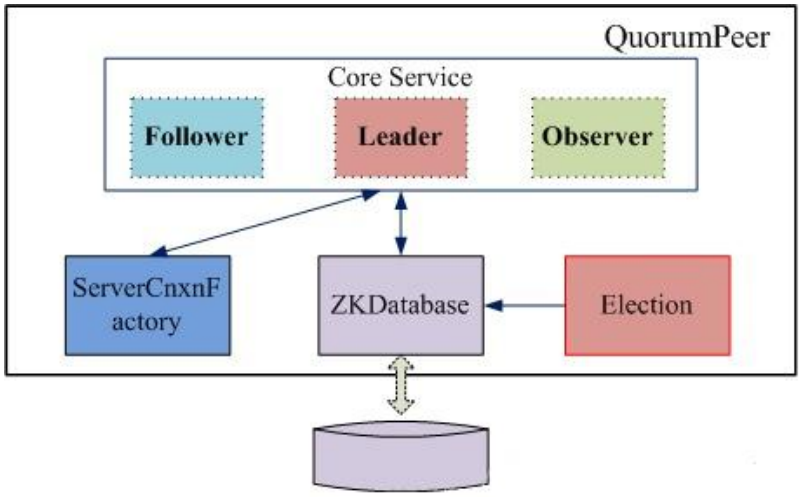
(1)QuorumPeer加载zkdatabase过程



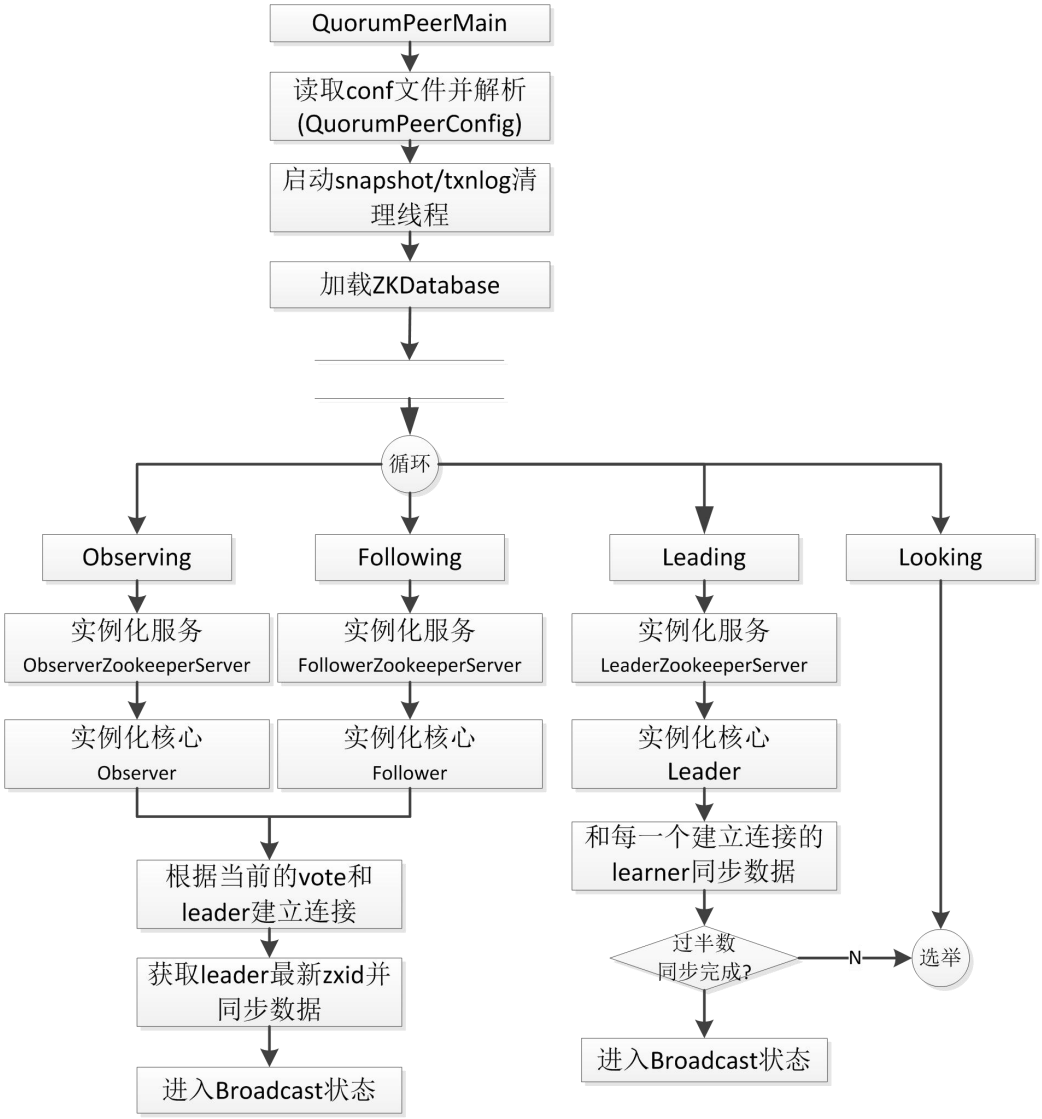
5 Recovery阶段

zookeeper 对于每个节点 QuorumPeer 主要如下四个组件:

- **客户端请求处理器 ServerCnxnFactory**: 负责维护与客户端的连接(接收客户端的请求并发送相应的响应);
- **数据引擎 ZKDatabase**: 负责存储/加载/查找数据(基于目录树结构的 KV+操作日志+客户端 Session);
- **选举器 Election**: 负责选举集群的一个 Leader 节点;
- **Leader/Follower/Observer**: 一个 QuorumPeer 节点应该完成的核心职责;



Server 启动时先通过 snapshot 和 txnlog 加载持久化数据到 ZKDataBase 中. 再进行选举, 并同步数据, 最后根据选举出的角色实例化角色服务.



5.1 数据恢复流程

加载快照日志进行数据恢复的流程介绍见 4.5 节.

5.2 选举流程

选举的时机:

- **Leader 失效**: 当 Leader 和 Follower 发送 ping 时,遇到"多数派"的 Follower 无法响应时(可能多数 Follower 已经离群,或者 Leader 离群),此时 Leader 进入 LOOKING 模式,开始选举.
- Follower 认为 Leader"失效":
  - Follower 首次加入集群时无法确定 Leader 则尝试选举;
  - Follower 和 Leader 之间的网络问题,导致 Follower 离群,该情况会一直处于选举只到网络恢复;

选票数据结构:
提议 leader 的　　sid　　zxid　　Epoch 发送 server 的　　sid　　state　　Epoch
sid:　　唯一标识一台主机,由用户手动写入 data 目录下的 myid 文件,结合 zoo.cfg 配置文件配置使用。在节点启动时，会判断该数据的合法性; zxid:　　选举 leader 用，是本机目前所见的最大的 ID 值; epoch:　　同 logicalclock，标识选举的轮数; State:　　标识本机的状态信息，含：looking、leading、following 和 observing。

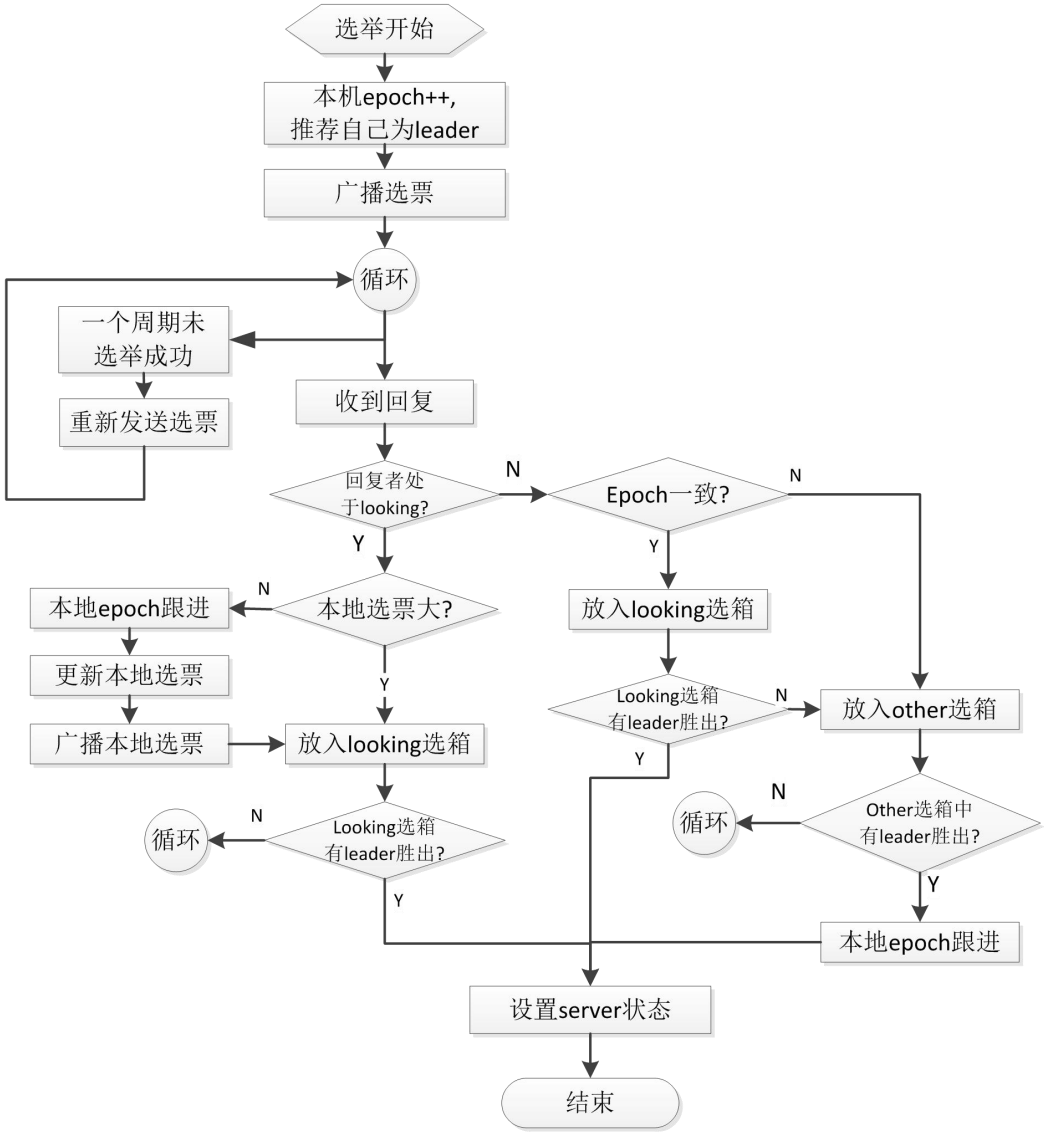
5.2.1 FastLeaderElection 快速选举

选举流程概述:

1. 选举线程首先向所有 Server 发起一次投票(包括自己);
2. 接收 server 处于 looking 时说明也在发起选举流程. 会和该 server 交互,将选票统一成按 epoch,zxid,sid 顺序最大的作为 leader.并广播该选票;
3. 接收 server 未处于 looking 时说明该 server 处于正常状态, 其未发起选举流程. 因此只是将自己所知的 leader 信息打包为投票传给处于 looking 状态 server.
4. looking 状态的 server 统计选票:
   a) 如果 Followers 发回的投票 leader 过半数的话说明集群正常运行,直接采用其推荐的 leader 即可.
   b) 否则如果 Followers 和 Looking 状态 servers 发回的投票中有 leader 过半数, 并且之后所有选票中没有比该 leader 更新的(按 epoch,zxid,sid 顺序, 该 leader 最大), 则说明 leader 选举成功.
   c) 否则说明未选举成功, 继续等待更多选票
5. leader 选举成功后还要和 followers 进行数据同步.在 leader 正式上任

后做的第一件事情就是根据当前保存的 zxid 值设置 epoch。随后 leader 会创建 NEWLEADER 包，通知 follower 目前 leader 所持有的 zxid，检验 follower 是否需要同步。Leader 在收到半数以上 follower 反馈，标识同步数据完毕，leader 正式成为 leader。Leader 初始化 zxid。正式对集群行驶管理权。Zookeeper 集群启动成功。否则 leader 的服务线程重启，再重新选举.

6. leader 和 followers 在稳步运行状态会定时去 ping，一旦连接的 servers 未超过半数，则服务线程重启，再重新选举.

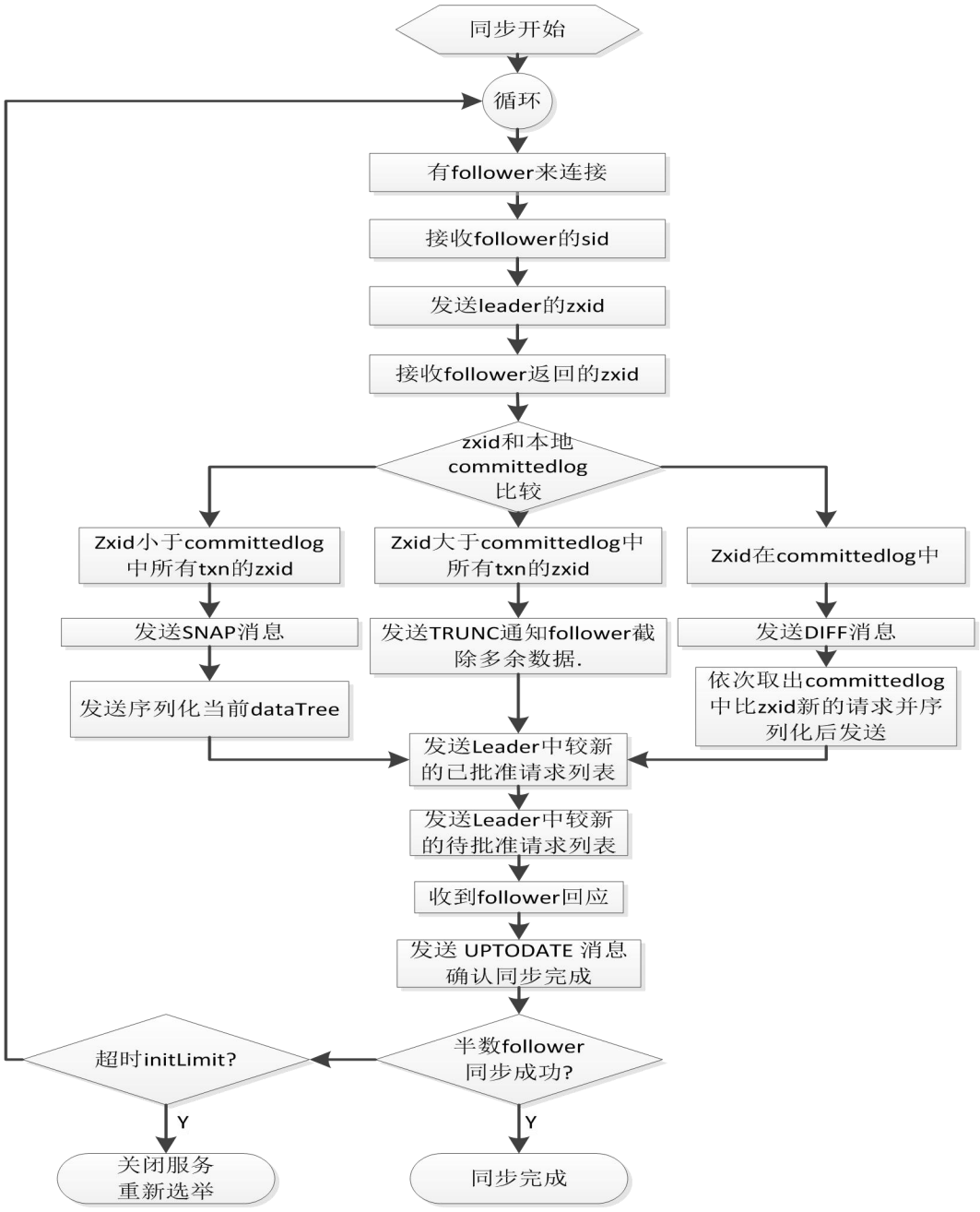


5.3 数据同步流程

当 zookeeper 选出 leader 时，由于各 server 间数据可能不一致,所以需要将 server 间的数据同步一致.

5.3.1 同步中 Leader 流程

QuorumPeer 在经过 Leader 选举，获悉当前 server 是 Leader 时，其实还不算是真正意义上的”leader”。因为到了这里 leader 还需要与集群中的其他服务器同步数据。如果这一步出错,将返回关闭服务，并重新选举 leader。



Leader 数据同步工作如下:

1. leader 接收到的来自某个 follower 发上来的 FOLLOWERINFO 数据包,该信息包告知了 follower 节点保存的 zxid。leader 会根据这个 zxid 与自己保存的 zxid 进行比较:
   a) 如果数据完全一致,则发送 UPTODATE 包告知 follower 当前数据就是最新的了;

b) 如果不一致，则判断这一阶段之内有没有已经被提交的提议值,如果有,那么:

i. 如果有部分数据没有同步,那么会发送 DIFF 数据包申请将有差异的数据同步过去。同时将 follower 没有的数据逐个打包成 COMMIT 包并发送给 follower 节点;

ii. 如果 follower 数据 id 更大,那么会发送 TRUNC 封包告知截除多余数据;

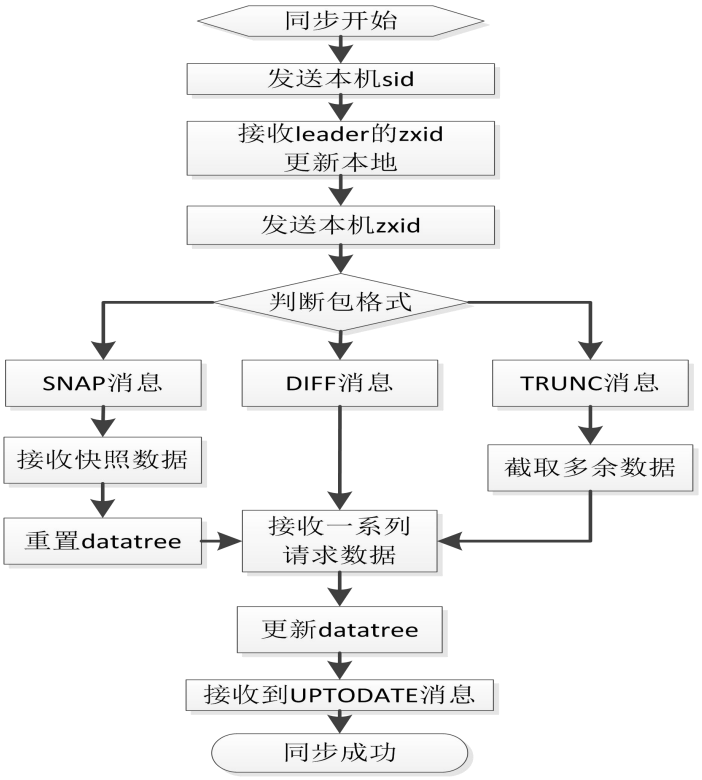
iii. 如果这一阶段内没有新提交的提议值,直接发送 SNAP 包将快照同步发送给 follower;

2. 消息验证完毕之后,发送UPTODATE包告知follower当前数据是最新的了

以上过程中,任何情况出现的错误,服务器将自动将选举状态切换到 LOOKING 状态,重新开始进行选举。

5.3.2 同步中 Follower 流程

QuorumPeer 经过 Leader 选举后发现自己是 Follower, 则先向 Leader 同步数据. 然后才能接受客户端的连接.



Follower 数据同步工作如下：

1. 尝试与 leader 建立连接。如果一定时间内没有连接上,就报错退出,重新回到选举状态;
2. follower 发送 FOLLOWERINFO 包给 leader,该包中带上自己的最大 id,也就是会告知 leader 本机保存的最大 zxid;
3. leader 根据不同的情况发送 DIFF,UPTODATE,TRUNC,SNAP,此时 follower 跟 leader 的数据也就同步上了。

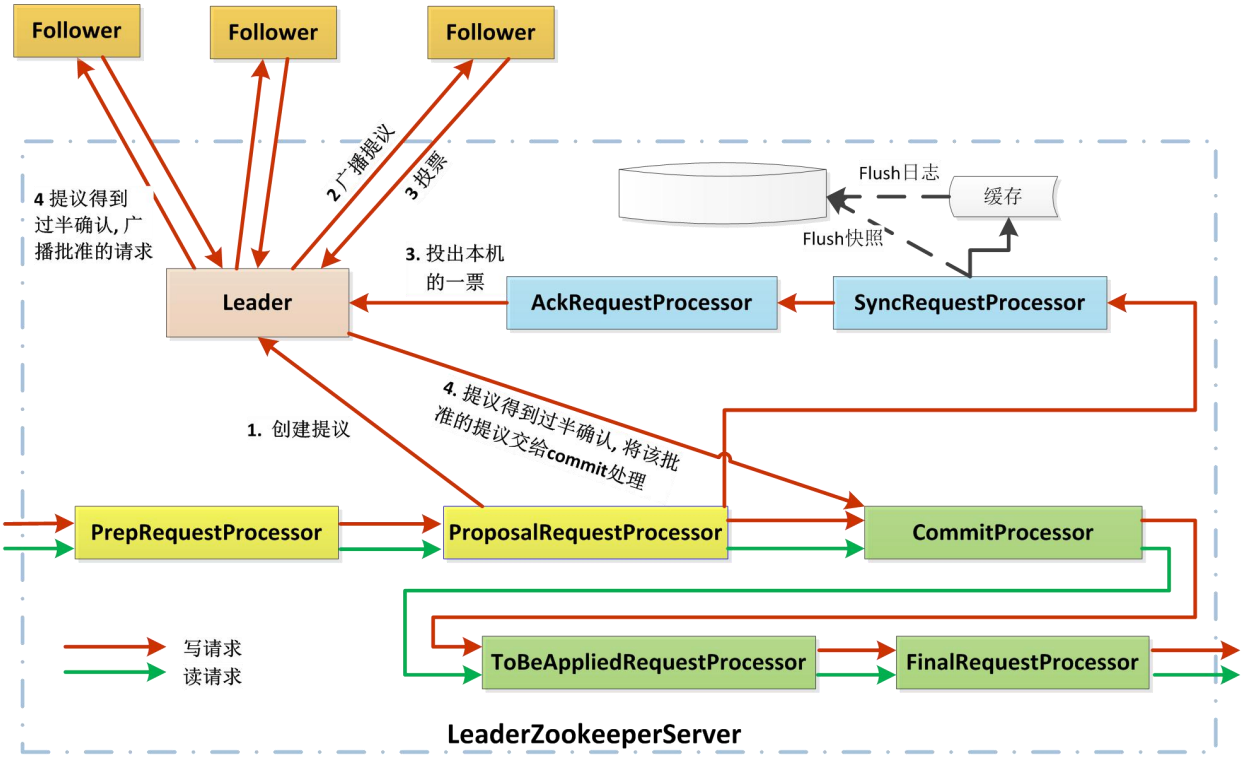
4. 由于 leader 端发送的最后一个包是 UPTODATE,因此在接收到这个包之后 follower 结束同步数据过程,发送 ACK 封包回复 leader.

以上过程中,任何情况出现的错误,服务器将自动将选举状态切换到 LOOKING 状态,重新开始进行选举。

6 Broadcast阶段

6.1 Broadcast 阶段中 Leader 流程

Leader 可以接收 client 的读写请求, 也可以接收 Learner 转发过来的写请求. 对于读请求会在 Leader 本机完成; 对于写请求需要新建提议在整个集群的决策群中广播, 只有过半数 Follower 回应收到提议, 才表示提议通过, 然后对 Follower 发布 commit 消息, 对 Observer 发布 inform 消息, 处理写请求并输出对客户端回
应.



Leader 对应的 ZooKeeperServer 是 LeaderZooKeeperServer. 其处理器职责链为:

1. PrepRequestProcessor -> ProposalRequestProcessor -> CommitProcessor -> ToBeAppliedRequestProcessor -> FinalRequestProcessor
2. SyncRequestProcessor -> AckRequestProcessor

- **PrepRequestProcessor 和 ProposalRequestProcessor 在一个线程中运行**: 主要用于操作合法性检查, 对写请求发起提议;
- **CommitProcessor, ToBeAppliedRequestProcessor, FinalRequestProcessor 在一个线程运行**: 主要用于对批准的提议进行处理, 有需要的话给客户端回应;

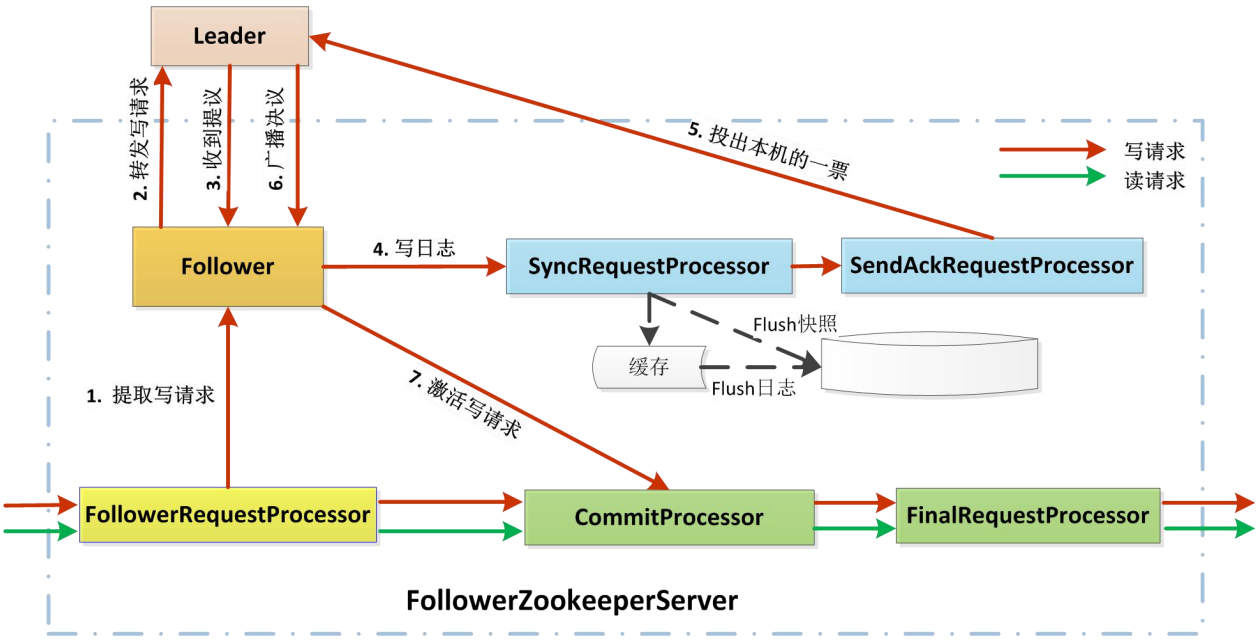
- **SyncRequestProcessor ， AckRequestProcessor 在一个线程运行**：主要用于将写请求刷入日志或快照，并对写请求的提议投出 Leader 的一票。

Leader 处理流程为：

1. 收到 client 发来的请求，或是 Follower 转发过来的写请求，提交到 Prep 处理器.
2. Prep 处理器对 session 和 acl 进行检查，发现异常的话设置错误码，再交给 Proposal 处理器.
3. Proposal 处理器对读请求直接交给 Commit 处理器. 对写请求生成提议广播给集群表决. 并且将请求放入 Sync 处理器和 Commit 处理器
4. Sync 处理器将写请求写日志，适时 flush 和写快照，再交给 Ack 处理器；
5. Ack 处理器投出本机的一票,并等待提议得到过半数响应后广播写请求,并将批准的请求交给 Commit 处理器；
6. Commit 处理器对于读请求直接交给 ToBeApplied 处理器. 对于写请求等待批准后的请求到达，并更新原请求后交给 ToBeApplied 处理器；
7. ToBeApplied 处理器直接交给 Final 处理器处理，处理完后从 toBeApplied 队列中移除该请求.
8. Final 处理器将写请求更新到内存树上，并返回读写操作的结果到 client.

6.2 Broadcast 阶段中 Follower 流程

Follower 接收 client 的读写请求. 对于读请求会在 Follower 本机完成;对于写请求转发给 Leader 提议表决. 待批准后由 Leader 通知处理.



Follower 对应的 ZooKeeperServer 是 FollowerZooKeeperServe. 其处理器职责链为：

1. FollowerRequestProcessor -> CommitProcessor -> FinalRequestProcessor
2. SyncRequestProcessor -> SendAckRequestProcessor

- **FollowerRequestProcessor 在一个线程中运行**：主要用于接受请求后，针对一些写动作发起一个 request，请求 leader 进行所有 server 的一致性的控制；
- **CommitProcessor，FinalRequestProcessor 在一个线程运行**：主要用于对批准的提议进行处理，有需要的话给客户端回应；
- **SyncRequestProcessor， SendAckRequestProcessor 在一个线程运行**：主要用于将写请求刷入日志或快照，并对写请求的提议投出 Leader 的一票。

Follower 处理流程为：

1. 收到 client 发来的请求,提交到 Follower 处理器.
2. Follower 处理器对读请求交给 commit 处理器. 对写请求转发给 leader 后再 commit 处理器.
3. Follower 收到 Leader 的提议后，交给 Sync 处理器；
4. Sync 处理器写日志后交给 SendAck 处理器；
5. SendAck 处理器投出本机的一票；
6. Follower 收到 leader 广播的决议，放入 Commit 处理器；
7. Commit 处理器对于读请求直接交给 Final 处理器. 对于写请求等待批准后的请求到达，并更新原请求后交给 Final 处理器；
8. Final 处理器将写请求更新到内存树上，并返回读写操作的结果到 client.

6.3 Broadcast 阶段中 Observer 流程

Observer 类似于 Follower，可以接收 client 的读写请求，但是不响应提议的表决.

6.4 数据如何保证一致性

Zookeeper 在新节点启动后，如何保证数据一致(对应 Recovery 阶段)：

1. 首先节点启动后，尝试读取本地的 SnapShot log 数据 (zkDb.loadDataBase())，反序列化为 DataTree 对象，并获取 last zxid。
2. learner 启动后会向 leader 发送自己的 last zxid
3. leader 收到 zxid 后，对比自己当前的 ZKDatabase 中的 last zxid
4. 如果当前 follower 的 zxid 在内存 committedLog 中，直接将内存中的 committedLog 提取出来进行发送，否则将当前的 DataTree 直接发送给 learner.(不再是发送变更记录)
5. 将 leader 当前已批准和待批准的请求发送给 learner；
6. 数据同步完成后，follower 会开始接收 request 请求

Zookeeper 在集群正常运行时，如何保证数据一致(对应 Broadcast 阶段)：

1. learner 在收到变更 request 时转发给 Leader;
2. leader 收到变更 request 时，先预处理. 再生成 proposal 提交给决策群投票;
3. 当 Leader 收到过半数的回应后. 表示该提议通过，给 followers 发 commit 消息，给 observers 发 inform 消息. Leader 进行后续处理
4. learners 收到消息后也各自对 dataTree 进行处理.

Zookeeper 最特别的一点是:
Leader 在发送 PROPOSAL 消息之前，和 Follower 接收到 PROPOSAL 消息之后，都会立即将消息记录到日志中。这样在收到过半的 ACK 之后，既可以确认消息已经在过半的 server 中保存过了。即使之后的 Commit 消息发送失败，也在事实上通过了消息。丢失 commit 消息的 follower 会在下一个事务中发现这一点，并自动退出。通过重启来重新取得一致性。

6.5 异常处理

Client 写请求验证不过如何处理?

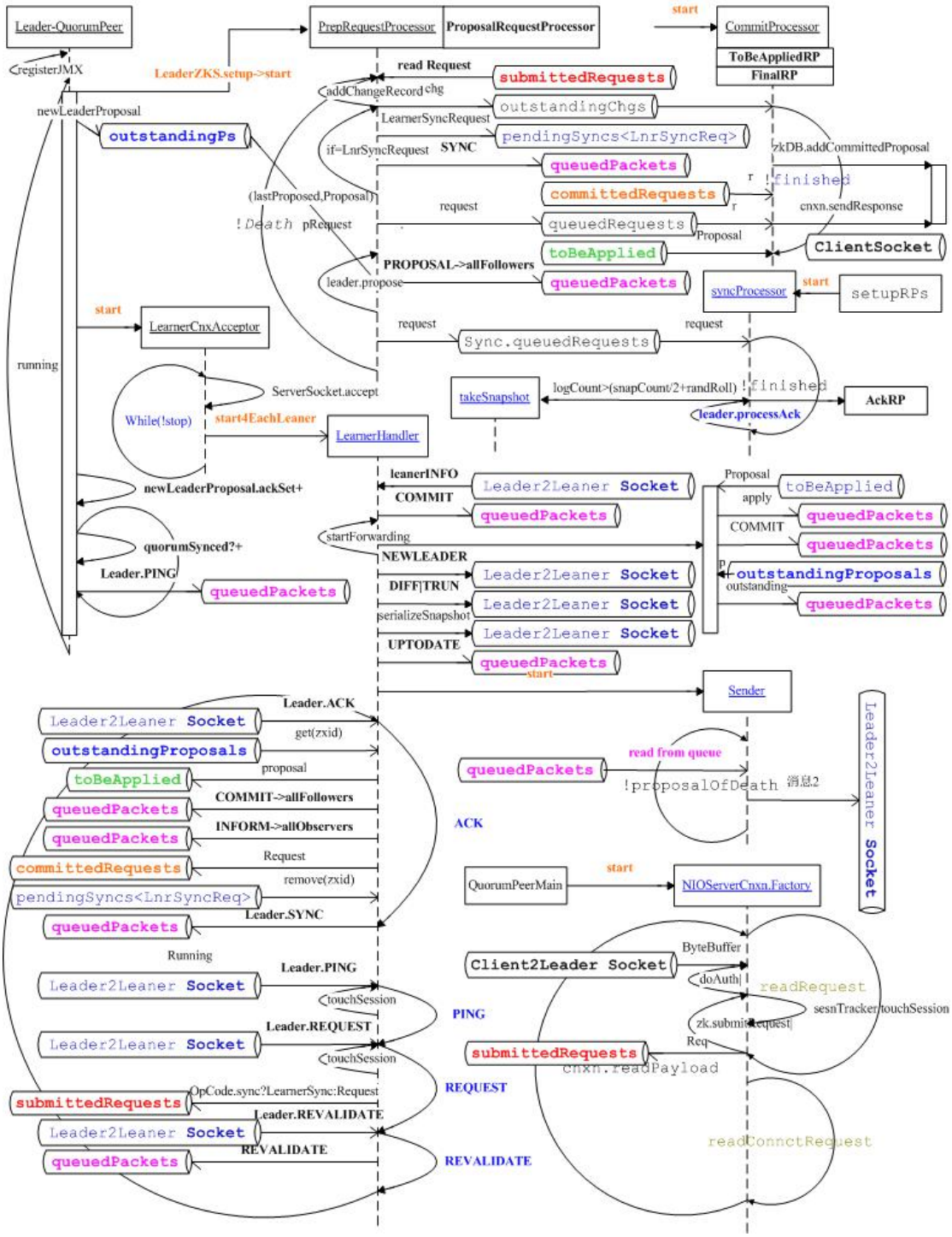
Client 写请求验证不通过，通常是由于 acl 验证不过，或路径逻辑有问题造成的. PrepRequestProcessor 发现验证不过的话会抛出异常，异常会填写到 Request 的错误码. 这样在 FinalRequestProcessor 处理时会将该错误码传给 client. Client 收到该响应发现操作失败码，会报异常.

Client 发请求过来，session 超期如何处理?

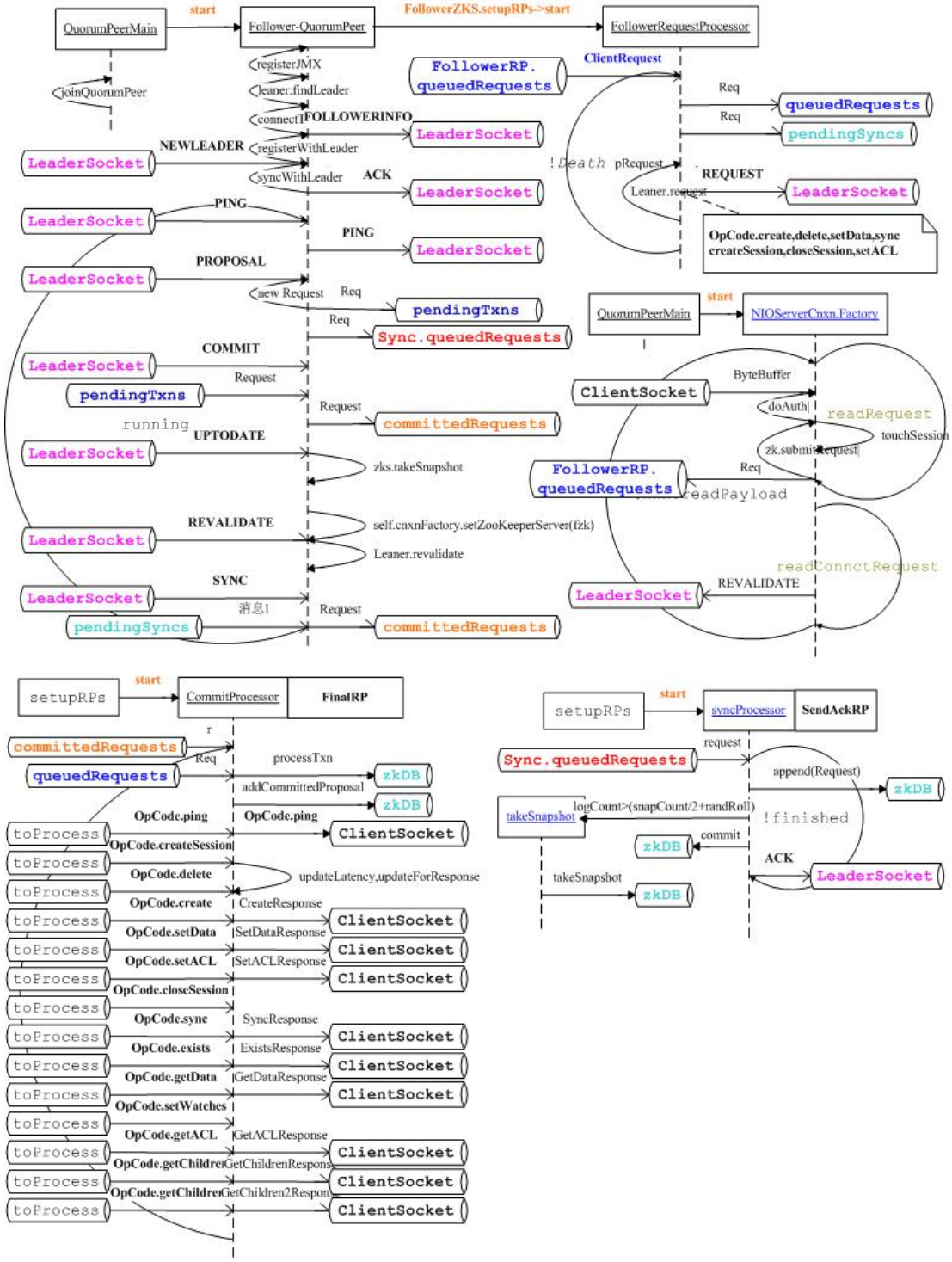
对于 Leader，由于在 PrepRequestProcessor 中已经对大部分请求做了超期验证. 发现超期的话会抛出异常，异常处理同上.

7 Server的协作图

7.1 Leader 的协作图



7.2 Follower 的协作图



8 Client操作流程

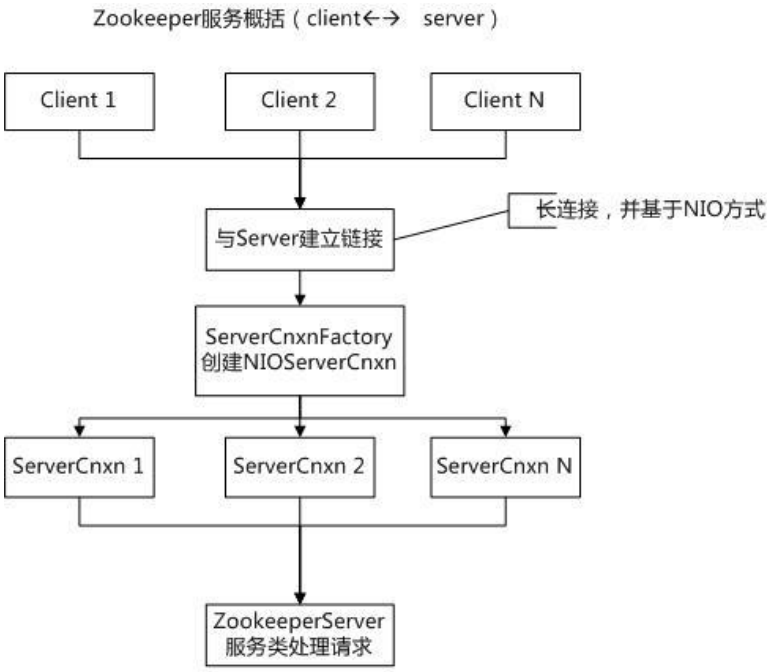
8.1 client 操作

8.1.1 client 与 server 端操作请求交互

zookeeper 是一个面向 client 服务类,通过此类可以创建 ZK client 实例,

以及向 server 提交请求,此类还间接的负责 Watcher 的注册和跟踪.

当用户创建一个 Zookeeper 实例以后，几乎所有的操作都被这个实例包办了,用户不用关心怎么连接到 Server, Watcher 什么时候被触发等等令人伤神的问题。可以写入一个或者多个 Zookeeper 的服务器地址，例如 127.0.0.1:3000, 127.0.0.1:3001, 127.0.0.1:3002″.客户端库会任意选择一个 ip:port 然后尝试连接。如果这个连接失败，或者这个客户端连接因为任何原因断掉了，这个客户端会自动选择下一个 ip:port 进行连接，直到连接建立起来为止.



8.1.2 client 操作 API

Zookeeper 支持的操作 API 如下

名称	同步	异步	watch	权限认证
Create	√	√		√
Delete	√	√		√
Exist	√	√	√	
getData	√	√	√	√
setData	√	√		√
getACL	√	√		
setACL	√	√		√
getChildren	√	√	√	√
Sync		√		
Multi	√			√

createSession	√			
closeSession	√			

8.1.3 四字命令

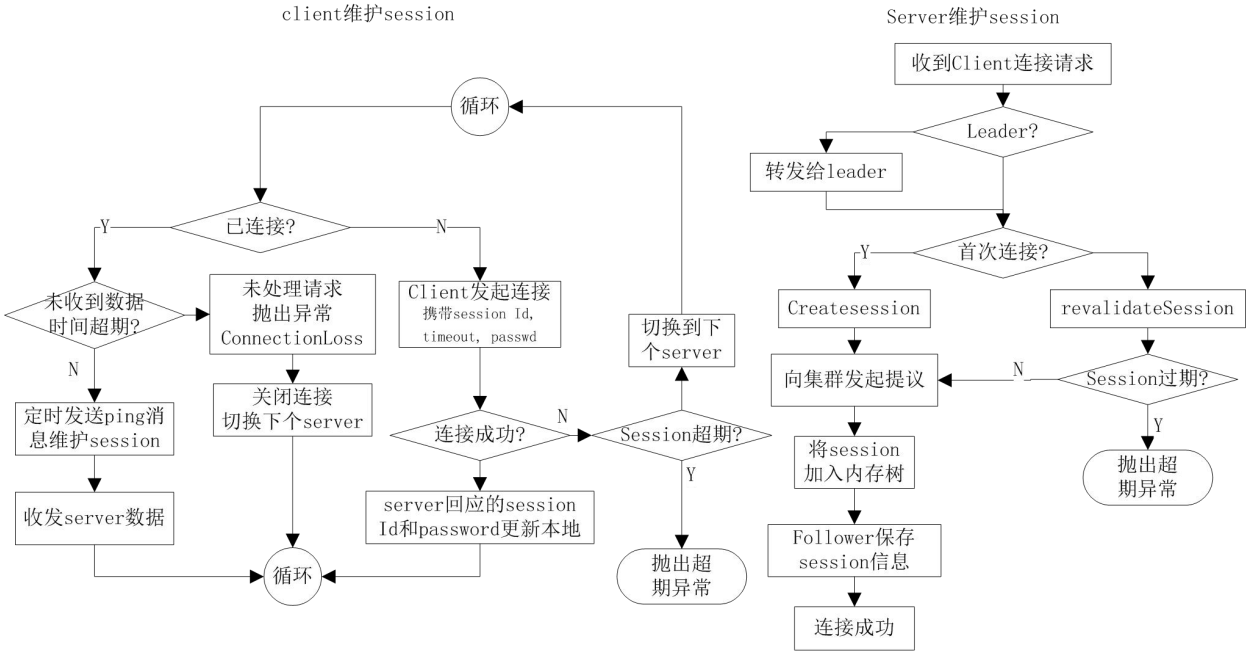
zookeeper 提供一些简单但是功能强大的 4 字命令. 这些四字命令不是通过 Zookeeper 生成实例来和服务端交互的. 而是直接连接服务端的监听端口, 并发送四字命令的 int 型值. 由 ServerCnxn 收到后获取当前 ZookeeperServer 的运行状态并打印出来.

例如: echo conf|nc localhost 2181

参数名	参数作用
conf	输出 server 的详细配置信息.
cons	输出指定 server 上所有客户端连接的详细信息，包括客户端 IP，会话 ID 等.
crst	重置所连接的统计信息.
dump	这个命令针对 Leader 执行，用于输出所有等待队列中的会话和临时节点的信息。
envi	用于输出 server 的环境变量。包括操作系统环境和 Java 环境。
ruok	用于测试 server 是否处于无错状态。如果正常，则返回“imok”,否则没有任何响应。 注意：ruok 不是一个特别有用的命令，它不能反映一个 server 是否处于正常工作。“stat”命令更靠谱。
stat	输出 server 简要状态和连接的客户端信息.
srvr	输出 server 简要状态信息.
srst	重置 server 的统计信息。
wchs	列出所有 watcher 数目等概要信息.
wchc	列出所有 watcher 信息，以 watcher 的 session 为归组单元排列，列出该会话订阅了哪些 path.
wchp	列出所有 watcher 信息，以 watcher 的 path 为归组单元排列，列出该 path 被哪些会话订阅者.
mntr	输出一些 ZK 运行时信息，通过对这些返回结果的解析，可以达到监控的效果.
isro	参看当前 server 是否只读. 即 java 系统属性 readonlymode.enabled 设置为 true 时, 输出 ro, 否则输出 rw.

8.2 Session 机制

8.2.1 Session 的实现



注意要点:

1. sessionTimeout 的设置

   客户端： sessionTimeout， 无默认值，创建实例时必填。

   服务端： minSessionTimeout（默认值为：tickTime*2），maxSessionTimeout（默认值为：tickTime*20），ticktime 的默认值为 3s，timeout 范围为 6s ~ 60s. 客户端设置的 sessionTimeout 到服务端后会取 minSessionTimeout 和 maxSessionTimeout 之间的值.

   如想客户端的 sessionTImeout 的超过该范围，需要同步修改服务端的配置.

2. Follower 只会维护 session 的相关信息. 而与之不同的是 Leader 会维护 session 超期时间，当超时后通知该 server 会话超期.

3. 重选 Leader 后，读出 session，会重新开始维护 session 超期时间. 实际上该超期时间是推迟了一些的.

8.2.2 Session 的异常

1. **SessionExpiredException 异常(和 EXPIRED 事件)**:
   a) **SessionExpiredException**: 通常是 zk 客户端与服务器的连接断了，试图连接上新的 zk 机器，这个过程如果耗时过长，超过 SESSION_TIMEOUT 后还没有成功连接上服务器，那么服务器认为这个 session 已经结束了（服务器无法确认是因为其它异常原因还是客户端主动结束会话），客户端会抛出异常. 需要重新实例 zookeeper 对象，重新操作所有临时数据

（包括临时节点和注册 Watcher）.

b) **EXPIRED**: 对于 EXPIRED 事件,则是在 zk 客户端重连接 server 时,server 发现 session 已经不存在,则会重置此次链接上的 session timeout 参数,如果 client 发现 server 返回的 timeout 时间为 0, 则发布一个本地的 Expired watchEvent.

2. **SessionTimeoutException 异常**:
这个异常属于链接异常,由 Client 端抛出;在 zk 客户端 socket 链接控制器 ClientCnxnSocket 会记录和 server 交互时间. SendThread 会不断轮询执行(执行发送请求 packet,读取响应,发送心跳等活动),当超过一段时间没有和 server 达成交互时, 将会导致 SessionTimeout 异常被抛出,此异常会被内部捕获,不需要客户端明确的去捕获或者进行其他措施,Client 会自动重连,直到 SessionExpired.
此异常不会直接导致 session 过期, 也不会清除当前 client 上注册的 watcher 集合. 尚未处理的响应会回以 CONNECTIONLOSS 错误(ReplyHeader,err 属性).最终报 ConnectionLossException.

3. **SessionMovedExcepion 触发机制**:
client 与 server1 的链接失效, client 和 server2 重建链接. client 发送给 server1 的请求,因为网络问题,延迟到达 server1,且在到达 server1 之前,client 与 server2 的 connect 请求已到达 server2.
该情况会触发 client 重连, 应用不会感知.

4. **ConnectionLossException 异常**:
因为 ZK Client 请求是被队列化的,这个队列化控制有每个客户端控制,如果 Client 在处理 Server 的响应结果时发现顺序有错乱,比如当前队列头部待确认的请求和 Server 交付的响应不是一个(通过 xid 做比较),那么 Client 认为在请求操作过程中,可能存在数据丢失的情况,将会触发此异常的发生;此异常会直接导致 Client 连接被关闭,重新建立连接.

5. **一直处于 connecting 异常**:
因为只有收到 server 的超时响应 client 才能知道异常. 所以当 server 集群未正常运行时, client 去链接会一直处于 connecting 状态.

8.3 Watcher 机制

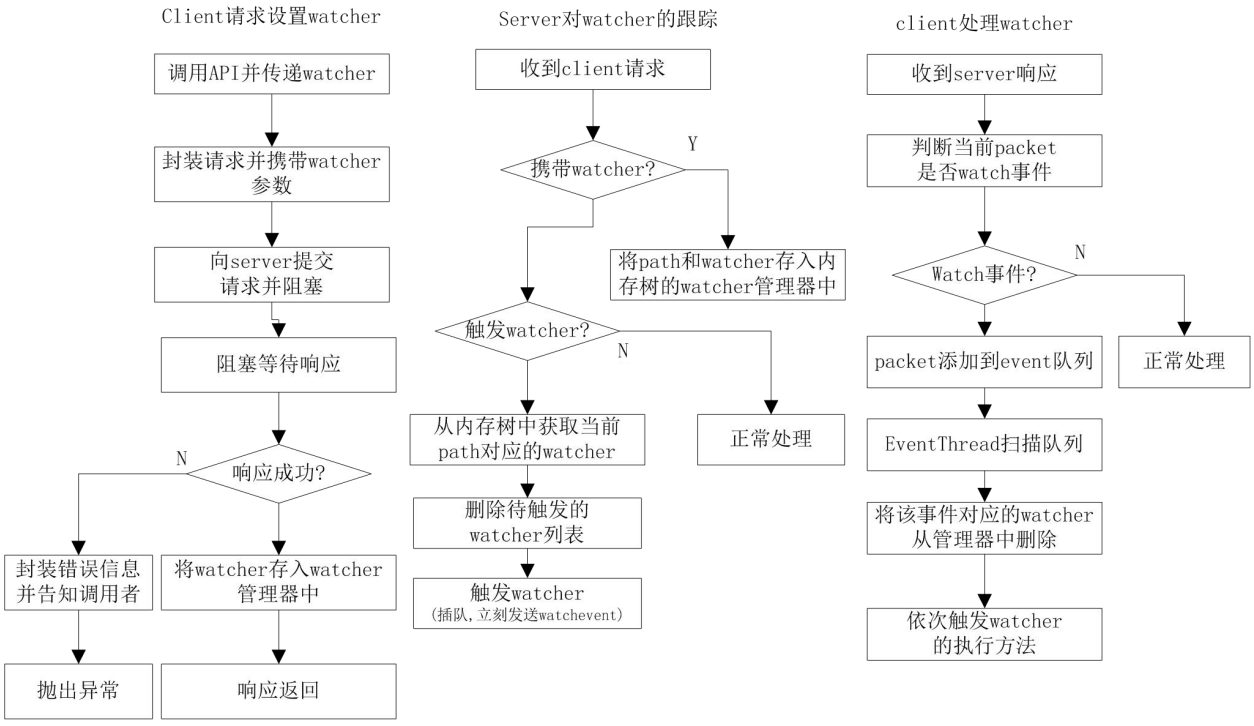
zookeeper 中最有特色的是监视(Watches)。客户端针对某个 znode 设置 Wathcer 后, 一旦该 znode 发生变化, 将会发消息通知客户端触发 watcher 的处理函数.

8.3.1 Watcher 消息与操作对应关系

各种写操作可以触发的 watcher，如下表所示：

	"/path"			"/path/child"		
	exists	getData	getChildren	exists	getData	getChildren
create("/path")	√	√				
delete("/path")	√	√	√			
setData("/path")	√	√				
create("/path/child")			√	√	√	
delete("/path/child")			√	√	√	√
setData("/path/child")				√	√	

8.3.2 Watcher 异步调用原理



注意要点:

1. "注册 watch"的操作有: exist, getChildren, getData;
   触发"watcher"的操作有: create, setData, delete;
   - 对于 setData,exist,getChildren 操作,可以接收 boolean 类型的 watcher 标识和 Watcher 对象,boolean 类型告知请求使用 defaultWatcher 对象注册事件.
   - 在创建 Zookeeper 实例时,允许接收一个 watcher 参数,此参数将会赋值给 watchMnanger.defaultWatcher,成为当前客户端的默认 Watcher.
2. 客户端并不会把用户创建的 watcher 对象传递给 Server,而是传递给 server 一个标记(boolean 值), server 此请求所涉及到的 patch 上是否有 watcher.

Watcher 信息并未在集群内同步，只存在于与 client 对接的 server.

3. watcher 的 process 方法是 client 用户自己实现的. 不宜进行大数据量的处理，因为所有 watcher 的处理和请求应答的处理都是在一个线程中的. 否则会影响吞吐的性能.
4. watcher 是一次性的，不能依赖 wathcer 来全权检测数据的变更,因为网络断开可能会导致事件通知的丢失.
   - 当事件被触发之后,server 端将删除事件,即使 client 端再次注册 watcher,那么"上一次事件"和"重新注册事件"这段事件内,仍然有可能数据已经变更.
5. 对于 Server 端遇到 session 关闭,连接关闭等异常时,都会清空和此连接(ServerCnxn)关联的 watch 列表.不过对于 Client 端却做了"弥补": 重连后会将本地已有的 watcher 列表全部发送给 Server.
6. 即使 path 节点目前不存在，依然可以设置 existWather. 只是需等到 path 节点存在后才会触发.

8.3.3 Watcher 异常处理

可能存在的异常问题：

1. **watcher 事件丢失**:
   client 向连接的 server 提交了 watcher 事件后,对应的 server 还未来得及提交给 leader 就直接出现了 jvm crash，这时对应的 watcher 事件会丢失。(理论上正常关闭 zookeeper server，不会存在该问题，需要客户端进行重试处理).
2. **ConnectionLossException 异常**:
   client 与其中的一台 server socket 链接出现异常.
   可以通过重试进行处理，在 ClientCnxn 会根据你初始化 ZooKeeper 时传递的服务列表，自动尝试下一个 server 节点，client 将本地已有的 watcher 列表全部发送给 Server.
3. **SessionExpiredException 异常**:
   client 的 session 超过 sessionTimeout 未进行任何操作.
   不能通过重试进行解决. 需要应用重新 new Zookeeper()，创建一个新的客户端，包括重新初始化对应的 watcher，EPHEMERAL 节点等。

8.4 ACL 机制

8.4.1 ACL 机制概述

在 Zookeeper 中，znode 的 ACL 是没有继承关系的，是独立控制的。

一个 ACL 对象就是一个 Id 和 permission 对，用来表示哪个/哪些范围的 Id（Who）在通过了怎样的鉴权（How）之后，就允许进行那些操作（What）：

- **permission**(What): 就是一个 int 表示的位码，每一位代表一个对应操作的允许状态。
- **Id** 由 scheme(Who)和一个具体的字符串鉴权表达式 id(How)构成，用来

描述哪个/哪些范围的 Id 应该怎样被鉴权。

- Scheme 事实上是所使用的鉴权插件的标识。
- id 的具体格式和语义由 scheme 对应的鉴权实现决定。

8.4.2 ACL 的机制

Zookeeper 的 ACL，同样可以从三个维度来理解：一是 scheme；二是 user；三是 permission，通常表示为 scheme:id:permissions，下面从这三个方面分别来介绍：

- **Permission** (What)：就是一个 int 表示的位码，每一位代表一个对应操作的允许状态。类似 unix 的文件权限，不同的是共有 5 种操作：
  - **CREATE** (c)：创建权限，可以在在当前 node 下创建 child node；
  - **READ** (r)：读权限，可以获取当前 node 的数据，可以 list 当前 node 所有的 child nodes；
  - **WRITE** (w)：写权限，可以向当前 node 写数据；
  - **DELETE** (d)：删除权限，可以删除当前的 node；
  - **ADMIN** (a)：管理权限，可以设置当前 node 的 permission.
- **scheme** (who)：scheme 对应于采用哪种方案来进行权限管理. zookeeper 实现了一个 pluggable 的 ACL 方案，可以通过扩展 scheme，来扩展 ACL 的机制。zookeeper 缺省支持下面几种 scheme：
  - **world**：它下面只有一个 id，叫 anyone. world:anyone 代表任何人，zookeeper 中对所有人有权限的结点就是属于 world:anyone 的.
  - **auth**：它不需要 id，只要是通过 authentication 的 user 都有权限 (zookeeper 支持 username/password 形式的 authentication)．没有对应的 id，或者只有一个空串""id。这个 scheme 没有对应的鉴权实现。语义是当前连接绑定的适合做创建者鉴权的 authInfo(通过调用 authInfo 的 scheme 对应的 AP 的 isAuthenticated()得知)都拥有对应的权限。遇到这个 auth 后，Server 会根据当前连接绑定的符合要求的 authInfo 生成 ACL 加入到所操作 znode 的 acl 列表中。
  - **digest**：使用 username:password 格式的字符串生成 MD5 hash 作为 ACL ID。 具体格式为：username:base64(SHA1(password)).对应内置的鉴权插件：DigestAuthenticationProvider.
  - **ip**：它对应的 id 为客户机的 IP 地址，设置的时候可以设置一个 ip 段，比如 ip:192.168.1.0/16，表示匹配前 16 个 bit 的 IP 段.用 IP 通配符匹配客户端 ip。对应内置鉴权插 IPAuthenticationProvider
  - **super**：在这种 scheme 情况下，对应的 id 拥有超级权限，可以做任何事情(cdrwa)
- **id**：id 与 scheme 是紧密相关的，具体的情况在上面介绍 scheme 的过程都已介绍，这里不再赘述。

8.4.3 ACL 的使用

客户端使用 acl 鉴权的时候需要按如下步骤:

1. 设置 acl 列表的时候只有三种鉴权方案可设("world","auth","ip");其中"auth"必须绑定一个有效的 AuthInfo;
2. auth 鉴权方式在操作 api 之前调用 addAuthInfo 添加鉴权信息(一般是设置"digest");

自定义的鉴权插件:

不管是内置还是自定义的鉴权插件都要实现 AP 接口(以下简称 AP)。

可以通过下面两种方式把新扩展的 AP 注册到 ProviderRegistry:

- **配置文件**：在 zookeeper 的配置文件中，加入 authProvider.$n=$classname
- **JVM 参数**：启动 Zookeeper 的时候，通过如下方式，把 AP 传入 -Dzookeeper.authProvider.$n=$classname

在上面的配置中, $n 是为了区分不同的 provider 的一个序号，只要保证不重复即可，没有实际的意义，通常用数字 1，2，3 等.

8.4.4 ACL 的实现

如前所述，在 zookeeper 中提供了一种 pluggable 的 ACL 机制。具体来说就是每种 scheme 对应于一种 ACL 机制，可以通过扩展 scheme 来扩展 ACL 的机制。

1. 在具体的实现中，每种 scheme 对应一种 AuthenticationProvider(以下简称 AP)。每种 AP 实现了当前机制下 authentication 的检查，通过了 authentication 的检查，然后再进行统一的 permission 检查，如此便实现了 ACL。所有的 AP 都注册在 ProviderRegistry 中，新扩展的 AP 可以通过配置注册到 ProviderRegistry 中。
2. AP 接口的 getScheme()方法定义了其对应的 scheme
3. 客户端与 Server 建立连接时，会将 ZooKeeper.addAuthInfo()方法添加的每个 authInfo 都发送给 ZKServer。
4. ZKServer 接受到 authInfo 请求后，首先根据 scheme 找到对应的 AP，然后调用其 handleAuthentication()方法将 auth 数据传入。对应的 AP 将 auth 数据解析为一个 Id，将其加入连接上绑定的 authInfo 列表（List）中。Server 在接入客户端连接时，首先会自动在连接上加上一个默认的 scheme 为 ip 的 authIndo：authInfo.add(new Id(“ip“, client-ip));
5. 鉴权时调用 AP 的 matches()方法判断进行该操作的当前连接上绑定的 authInfo 是否与所操作的 znode 的 ACL 列表匹配。

8.4.5 ACL 的超级用户

zookeeper 可以开启超级用户,使其能做任何事情. 开启步骤如下:

1. 服务启动时开启 superDigest 模式。

配置如下启动参数，然后重启 server(不能用 zkServer.sh restart 方式重启，先杀掉 zookeeper 进程，重启后调用 jps -v 检查启动参数是否设置正确)
"-Dzookeeper.DigestAuthenticationProvider.superDigest=super:/7ahZf2EjED/untmtb2NRkHhVlA="
启动参数中的超级用户密码/7ahZf2EjED/untmtb2NRkHhVlA=是对明文密码的 MD5Hash 加密的 base64 编码. 可以通过如下方法获取:
java -Djava.ext.dirs=../lib -cp $CLASSPATH:../zookeeper-3.4.5.jar org.apache.zookeeper.server.auth.DigestAuthenticationProvider super:yinshi.nc-1988

2. 在 java 代码中进行 digest 模式的授权:

```
zkClient.addAuthInfo( "digest", "super:yinshi.nc-1988".getBytes() );
```

9 部署注意事项

zookeeper 能够提供高可用分布式协调服务，是要基于以下两个条件:

1. **集群中只有少部分的机器不可用**:
   这里说的不可用是指这些机器或者是本身 down 掉了，或者是因为网络原因，有一部分机器无法和集群中其它绝大部分的机器通信。例如，如果 ZK 集群是跨机房部署的，那么有可能一些机器所在的机房被隔离了。
2. **部署节点有足够的磁盘存储空间以及良好的网络通信环境**:
   zookeeper 由于需要持久化事务日志和快照. 为了保证数据的可靠性, 其是在数据 flush 到磁盘后才会继续后续的业务. 所以磁盘的性能对 zookeeper 的性能有着很重要的影响.
   网络延迟,网络孤岛等情况都会对 zookeeper 的服务带来一定的影响, 导致其连接的 session 超期.
3. **部署集群不存在时间跳变**:
   连接 zookeeper 的 session 是有指定存活期的,一旦超过该存活期,session 就会中断, 需要业务再重新连接. 如果集群中存在时间跳变的话, 很容易导致连接到 zookeeper 的 session 频繁超期.

因此 zookeeper 部署时需要注意如下事项:

1. **集群维度**:
   - **部署节点数尽量是奇数**:
     zookeeper 服务存在"过半 server 存活即可用"的特性. 整个集群如果对外要可用的话，那么集群中必须要过半的机器是正常工作并且彼此之间能够正常通信。基于这个特性，那么如果想搭建一个能够允许 F 台机器 down 掉的集群，那么就要部署一个由 2xF+1 台机器构成的 ZK 集群。因此，一个由 3 台机器构成的 ZK 集群，能够在 down 掉一台机器后依然正常工作，而 5 台机器的集群，能够对两台机器 down 掉的情况容灾。注意，如果是一个 6 台机器构成的 ZK 集群，同样只能够 down 掉两台机器，因

为如果 down 掉 3 台，剩下的机器就没有过半了。

- **尽量每个节点配置对立的硬件环境**:
  为了尽可能地提高 ZK 集群的可用性,应该尽量避免一大批机器同时 down 掉的风险.为此最好能跨机架,跨机房部署.
- **保持 Server 地址列表一致**:
  - 集群中每个 server 的 zoo.cfg 中配置机器列表必须一致;
  - 集群中各节点的 hosts 配置一致;
  - 客户端使用的 server 地址列表必须和集群所有 server 的地址列表一致。(如果客户端配置了集群机器列表的子集的话，也是没有问题的，只是少了客户端的容灾)

2. **单机维度**: 对于 zookeeper 来说，如果在运行过程中，需要和其它应用程序来竞争磁盘，CPU，网络或是内存资源的话，那么整体性能将会大打折扣。由于客户端对 ZK 的更新操作都是永久的，不可回退的，也就是说，一旦客户端收到一个来自 server 操作成功的响应,那么这个变更就永久生效了。为做到这点，ZK 会将每次更新操作以事务日志的形式写入磁盘，写入成功后才会给予客户端响应。因此磁盘写入速度制约着 ZK 每个更新操作的响应。为了尽量减少 ZK 在读写磁盘上的性能损失

- **独立的事务日志和快照输出挂载盘**:
  对于每个更新操作，ZK 都会在确保事务日志已经落盘后，才会返回客户端响应。因此事务日志的输出性能在很大程度上影响 ZK 的整体吞吐性能。强烈建议是给事务日志的输出分配一个单独的磁盘。
- **配置合理的 JVM 堆大小，尽量避免内存与磁盘空间的交换**:
  确保设置一个合理的 JVM 堆大小，如果设置太大，会让内存与磁盘进行交换，这将使 ZK 的性能大打折扣。例如一个 4G 内存的机器的，如果你把 JVM 的堆大小设置为 4G 或更大,那么会使频繁发生内存与磁盘空间的交换，通常设置成 3G 就可以了。当然，为了获得一个最好的堆大小值，在特定的使用场景下进行一些压力测试。