ZooKeeper

0、前言

ZooKeeper 是一个**分布式数据一致性解决方案**，分布式应用程序可以基于 ZooKeeper 实现诸如**数据发布/订阅、负载均衡、命名服务、Master 选举、分布式锁**和分布式队列、分布式协调/通知等功能。

【背景】

> **领导：今天下午3-4点大会议室开会，收到请回复1。**
>
> **而领导会看有多少人回复1，如果回复的人太少，他可能会再次进行通知，如果大多数人都回复了，那么会议才可以进行。他并不会统计有多少人回复。**
>
> **等到下午3点，所有同时一起做规划。**

1、概念

【Zookeeper的角色】

通过命令 **zookeeper-server status** 可查看

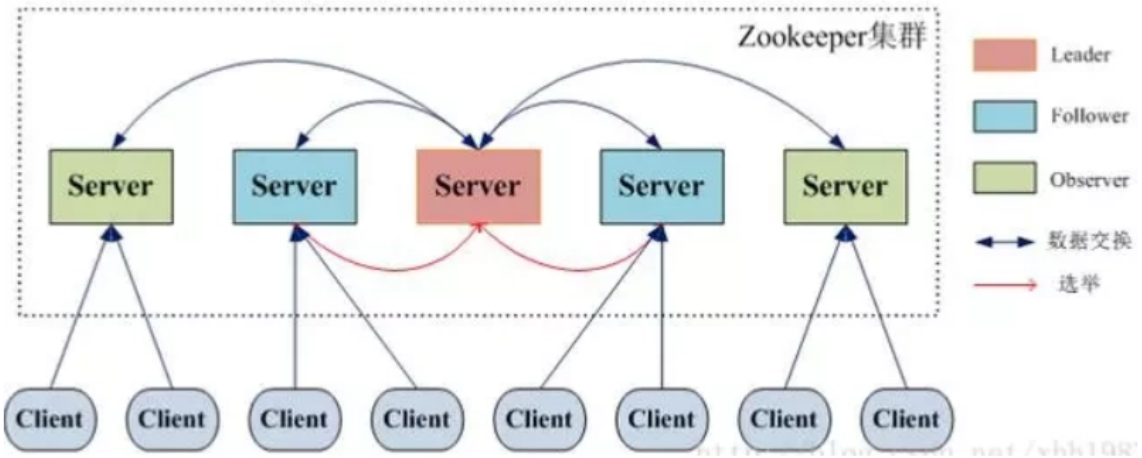
```
[root@node-20-103 ~]# zookeeper-server status
JMX enabled by default
Using config: /etc/zookeeper/conf/zoo.cfg
Mode: follower
```

```
[root@node-20-104 ~]# zookeeper-server status
JMX enabled by default
Using config: /etc/zookeeper/conf/zoo.cfg
Mode: leader
```

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

  为了支持更多的客户端，需要增加更多Server；Server增多，投票阶段延迟增大，影响性能；权衡伸缩性和高吞吐率，引入Observer。Observer不参与投票，接受客户端的连接，并将写请求**转发**给leader节点。
- 客户端（client），请求发起方。

> **leader发起投票，分发提议，发起提交**
>
> **follower只读不写**
>
> **observer为了避免当节点过多投票延迟过大影响性能而进行的水平扩展，只转发请求和同步状态给leader，提高读取速度**



• Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是**恢复模式（选主）和广播模式（同步）**。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

【事务id - zxid】

ZooKeeper状态的每一次改变，都对应着一个事务id，原子递增。

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

所有的提议（proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的统治时期。低32位用于递增计数。

> **事务id用来表示数据节点发生改变的唯一递增标识**

【会话（Session）】

Session 是指客户端会话，在讲解客户端会话之前，我们先来了解下客户端连接。在 ZooKeeper 中，一个客户端连接是指客户端和 ZooKeeper 服务器之间的TCP长连接。

ZooKeeper 对外的服务端口默认是**2181**，客户端启动时，首先会与服务器建立一个TCP连接，从第一次连接建立开始，客户端会话的生命周期也开始了，通过这个连接，客户端能够通过**心跳检测和服务器保持有效的会话**，也能够向 ZooKeeper 服务器**发送请求并接受响应**，同时还能通过该连接接收来自服务器的 **Watch 事件通知。**

Session 的 SessionTimeout 值用来设置一个客户端会话的超时时间。当由于服务器压力太大、网络故障或是客户端主动断开连接等各种原因导致客户端连接断开时，只要在 SessionTimeout 规定的时间内能够重新连接上集群中任意一台服务器，那么之前创建的会话仍然有效。

> **会话的作用是维持心跳、请求和响应、监听回调等；有会话维持时间**

【版本】

Zookeeper 的每个 ZNode 上都会存储数据，对应于每个ZNode，Zookeeper 都会为其维护一个叫作 **Stat** 的数据结构，Stat中记录了这个 ZNode 的三个数据版本，分别是version（当前ZNode的版本）、cversion（当前ZNode子节点的版本）和 cversion（当前ZNode的ACL版本）。

> **节点数据包含了当前节点和当前节点子节点的版本信息和权限控制信息**

【Watcher】

Watcher（事件监听器），是Zookeeper中的一个很重要的特性。Zookeeper允许用户在指定节点上注册一些Watcher，并且在一些特定事件触发的时候，ZooKeeper服务端会将事件通知到感兴趣的客户端上去，该机制是Zookeeper实现分布式协调服务的重要特性。

> **观察者模式，客户端监听节点，zk回调通知，客户端再反应；反应器模式**

【ACL】

Zookeeper采用ACL（AccessControlLists）策略来进行权限控制，类似于 UNIX 文件系统的权限控制。Zookeeper 定义了如下5种权限。

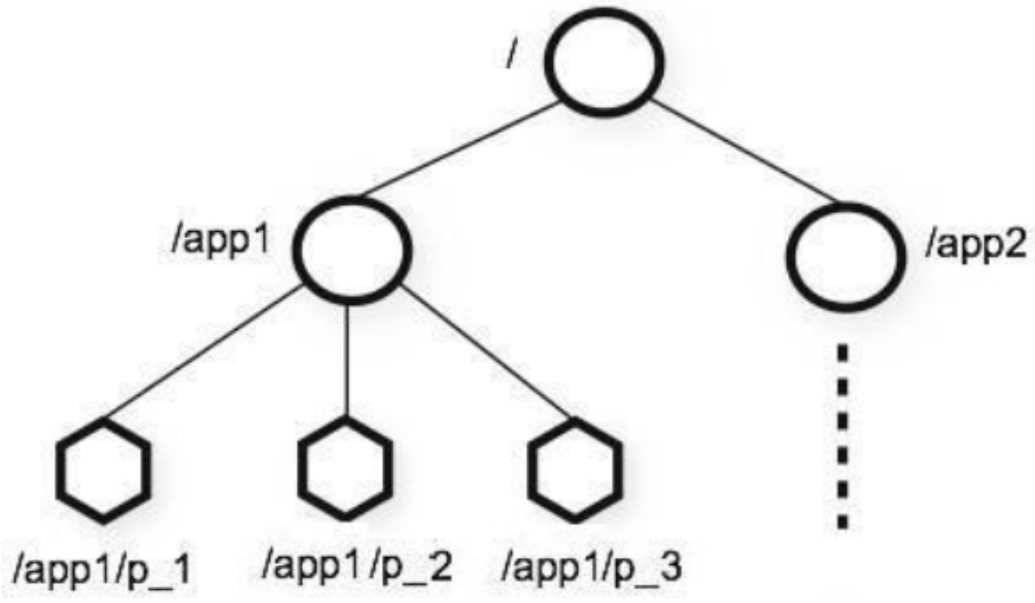
- **CREATE**：创建子节点的权限。
- **READ**：获取节点数据和子节点列表的权限。
- **WRITE**：更新节点数据的权限。
- **DELETE**：删除子节点的权限。
- **ADMIN**：设置节点 ACL 的权限。

> **权限控制包括：节点权限的增删改查**

【客户端请求】

- 事务性请求：create、set、delete
- 非事务性请求：get、exist

2、数据模型



在谈到分布式的时候，我们通常说的“节点"是指组成集群的每一台机器。然而，在Zookeeper中，“节点"分为两类，第一类同样是指构成集群的机器，我们称之为机器节点；第二类则是指数据模型中的数据单元，我们称之为数据节点——ZNode。

Zookeeper将所有**数据存储在内存中，数据模型是一棵树**（Znode Tree），由斜杠（/）的进行分割的路径，就是一个Znode，例如/foo/path1。每个上都会保存自己的数据内容，同时还会保存一系列属性信息。

> **数据以树形结构存储在内存中**
>
> **节点是目录或者文件**
>
> **节点保存数据内容和节点属性（事务id、创建时间、版本）**

【节点类型】

1、PERSISTENT-持久化目录节点

客户端与zookeeper断开连接后，该节点依旧存在

2、PERSISTENT_SEQUENTIAL-持久化顺序编号目录节点

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

3、EPHEMERAL-临时目录节点

客户端与zookeeper断开连接后，该节点被删除

4、EPHEMERAL_SEQUENTIAL-临时顺序编号目录节点

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

【节点的工作状态】

- LOOKING：当前Server不知道leader是谁，正在搜寻。
- LEADING：当前Server即为选举出来的leader。
- FOLLOWING：leader已经选举出来，当前Server与之同步。

3、ZAB协议

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

在Zookeeper中，ZAB有两种模式，一种是**恢复模式**，用来当**集群启动时、或者leader挂起时、或者leader的follower不足半数时**发起新leader的选举，当选举完成，并且已有**过半**的节点和leader的**数据同步完成**后，ZAB退出恢复模式。

另一种模式是当集群已有过半follower节点和leader节点数据同步完成以后，此时集群可以对外服务了，ZAB协议进入**广播模式。**当客户端有请求过来时，首先请求会统一转发交给**leader**来处理，leader会为请求生成一个**提案和事务id**，提案写入本地的**事务日志**，然后leader将提案和事务id广播给follower。follower收到以后，将提案写入事务日志，然后回复**ack。**当有过半follower节点回复ack后，leader会**提交事务**，然后将内存中的数据节点进行更新，同时将已提交的事件**广播**给其他follower，其他节点收到以后也会进行事务的提交，这就是ZAB的广播模式。

> **选主、同步（服务启动、运行时中断）**

> **预提交（事务日志），收到半数ack，提交（更新内存数据 - 树节点），广播提交**

4、选举流程

这篇主要分析leader的选择机制，zookeeper提供了三种方式：

- LeaderElection
- AuthFastLeaderElection
- FastLeaderElection（默认）

> **FastLeaderElection默认，其余两种官方说之后会废弃**

Leader选举时机

- 集群启动
- Leader挂起
- Leader发现Follower已不足半数（因为zk有很多机制需要follower半数获胜机制，如果follower不足半数，则无法让机制正常进行）

> **启动、leader挂起、follower不足半数**

【FastLeaderElection选举机制】

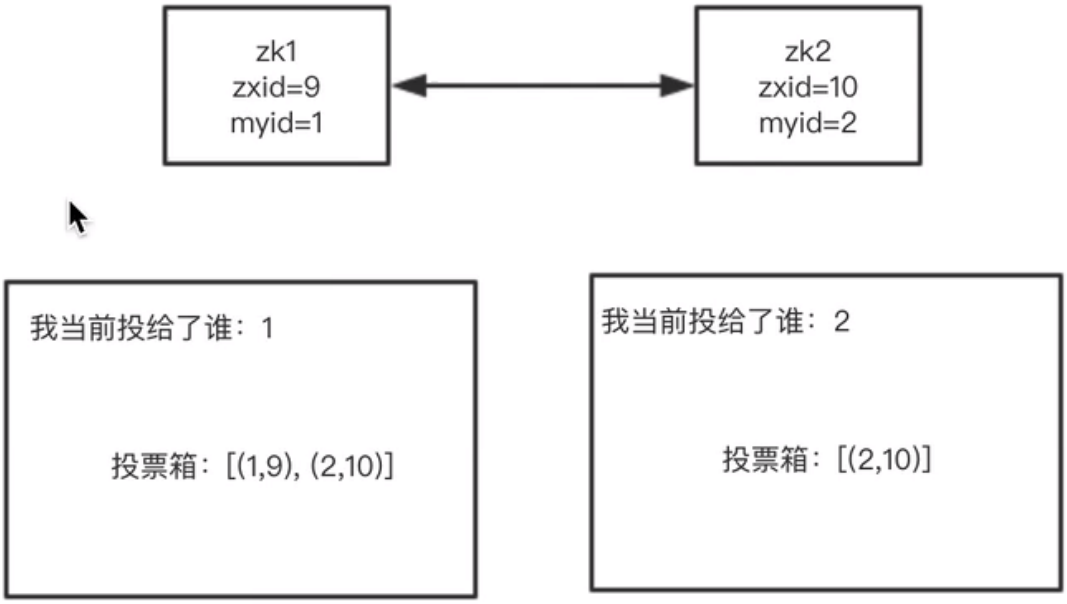
【集群第一次启动】

目前有5台服务器，每台服务器均没有数据，它们的编号分别是1,2,3,4,5,按编号依次启动，它们的选择举过程如下：

- 服务器1启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器1的状态一直属于Looking。
- 服务器2启动，给自己投票，同时与之前启动的服务器1交换结果，由于服务器2的编号大所以服务器2胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是LOOKING。
- 服务器3启动，给自己投票，同时与之前启动的服务器1,2交换信息，由于服务器3的编号最大所以服务器3胜出，此时投票数正好大于半数，所以服务器3成为领导者，服务器1,2成为小弟。
- 服务器4启动，给自己投票，同时与之前启动的服务器1,2,3交换信息，尽管服务器4的编号大，但之前服务器3已经胜出，所以服务器4只能成为小弟。
- 服务器5启动，后面的逻辑同服务器4成为小弟。

> **初次启动，对比节点id值大小，过半优选**

【具体流程】

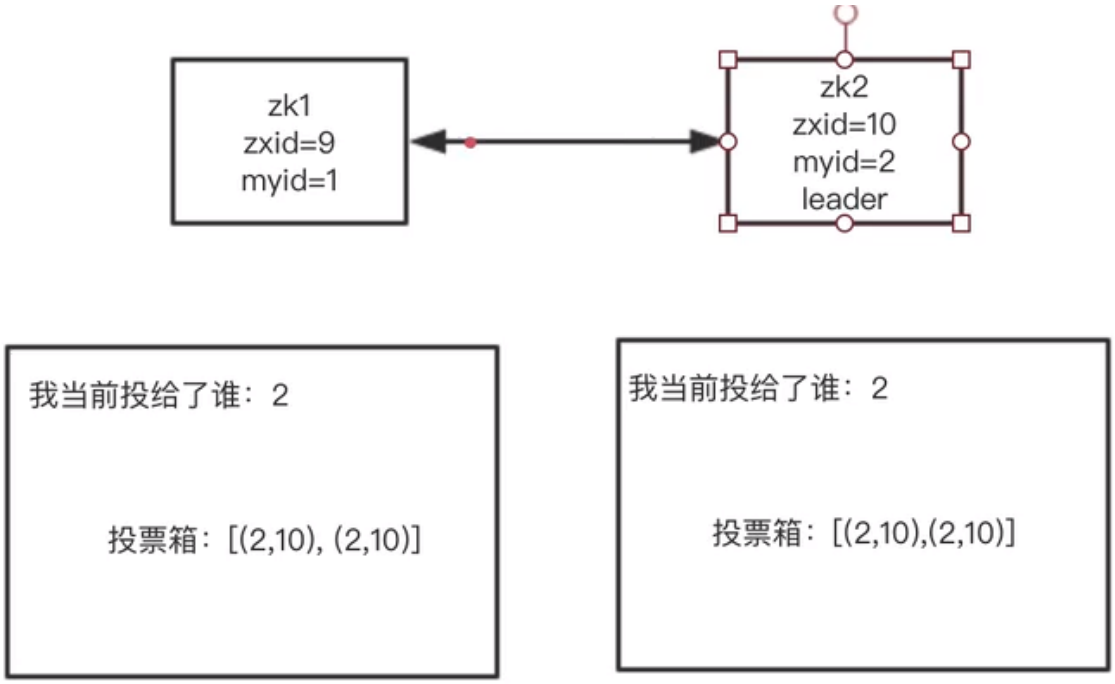


集群启动时，集群的每台机器都处于**Looking**状态，就是处于寻找Leader的状态。当发现此时没有leader的时候，就会进入选举。除了观察者（只负责把请求转发给leader）以外，每个候选人都会参与投票，首先会投票给**自己**，然后和其他的候选人进行**交流和对比**，对比他们各自的支持者哪个**更好**，就投票给谁。

【对比事务id】

在Zookeeper中，定义谁更好谁更强的标准有两个，一个是**事务的id**。这个事务的id，是原子递增的，当有客户端的请求发送给Zookeeper的一台服务器时，Zookeeper会为其生成一个事务id，id越大说明其数据越新，在Zookeeper看来就是其维护数据的能力越强，所以候选人会倾向于把票投给事务id越大的候选人。

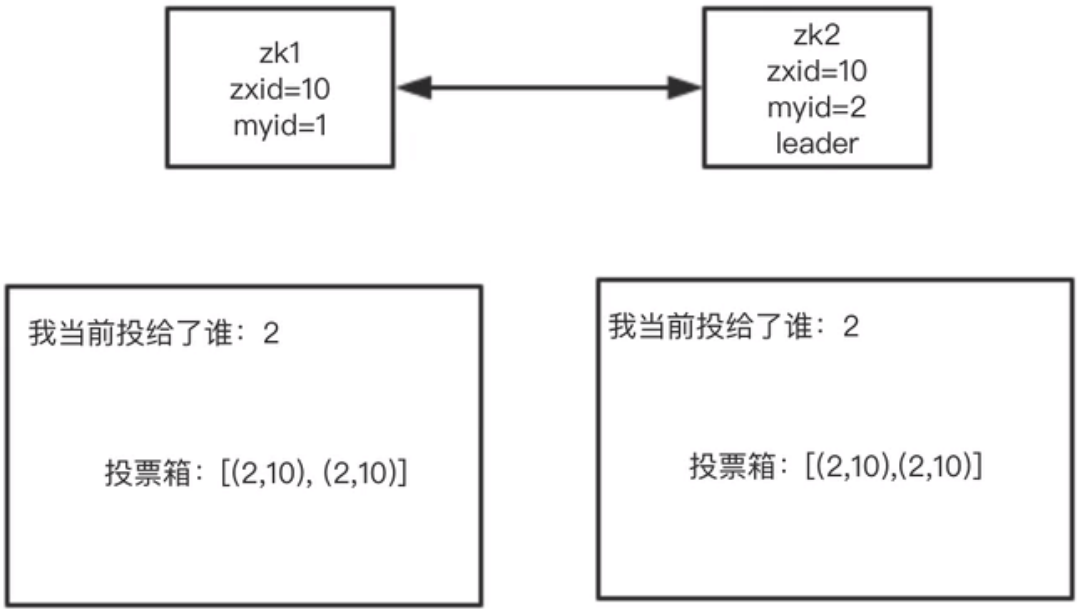
比如现在有两台机器zk1和zk2进行选举和对比。zk1的本机id是1，事务id是9，zk2的本机id是2，事务id是10。首先zk1和zk2会把票投给自己，每个候选人持有一个**投票箱**，在程序中的实现就是一个数组，这个数据存储的结构信息有两个，**一个是服务器的id，一个是该服务器当前的事务id。**



第一轮投票过后两个节点各自的投票箱就是（1,9）和（2,10）这样的情况。然后两个节点通过**心跳**进行数据通信，互发对方当前投票箱的数据，然后节点各自进行对比，再把票投给事务id更大的那个id，同时更新投票箱的数据，**同步**当前最新的选票，比如zk1的投票箱就变成是2,10和2,10，分别代表两个节点的投票情况，然后再把它的投票情况**广播**给其他节点，其他候选人再进行一轮对比和投票。直到某个节点拥有超过**半数**的选票后，即当选Leader，选举结束，进入数据的同步。

> **启动时looking，无主则选举**
>
> **事务id对比，值越大则数据越新，胜率越大**
>
> **首轮投自己，之后对比其他，然后在投票箱中更新，最后过半优选**

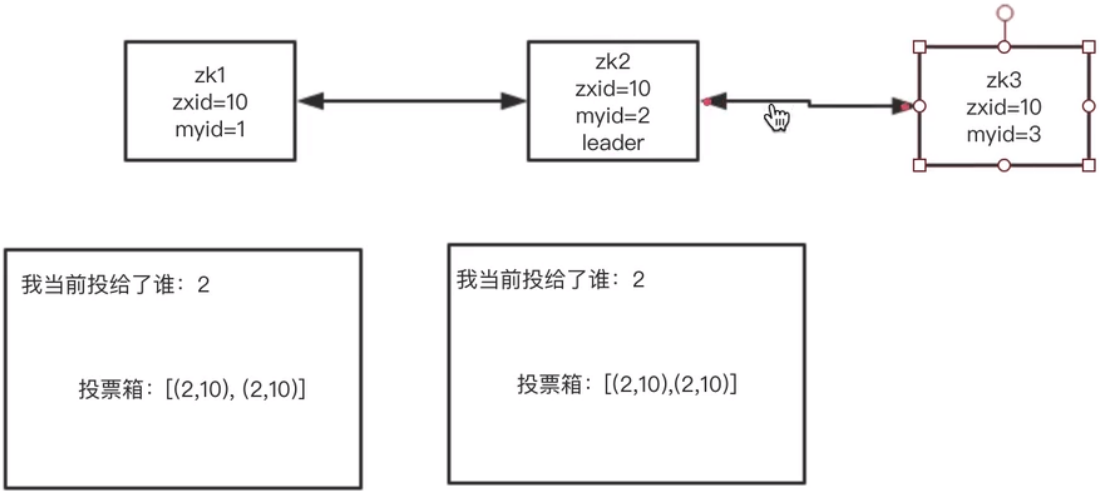
【对比节点id】



还有另外一种情况，就是当两个节点的事务id都一致的情况下，或者是当集群第一次启动后，还没有生成事务id的情况下，会对比另一个参数，就是各自的**节点id**的大小。在zookeeper集群中，每当有一个节点加入，就会在本地的配置文件中去声明一个标识本机身份的id，myid。在上述情况下，谁的myid的数值越大，则其他候选人会将票投给它。直到某个候选人拥有超过半数的选票后，即当选Leader，选举结束，进入数据的同步。

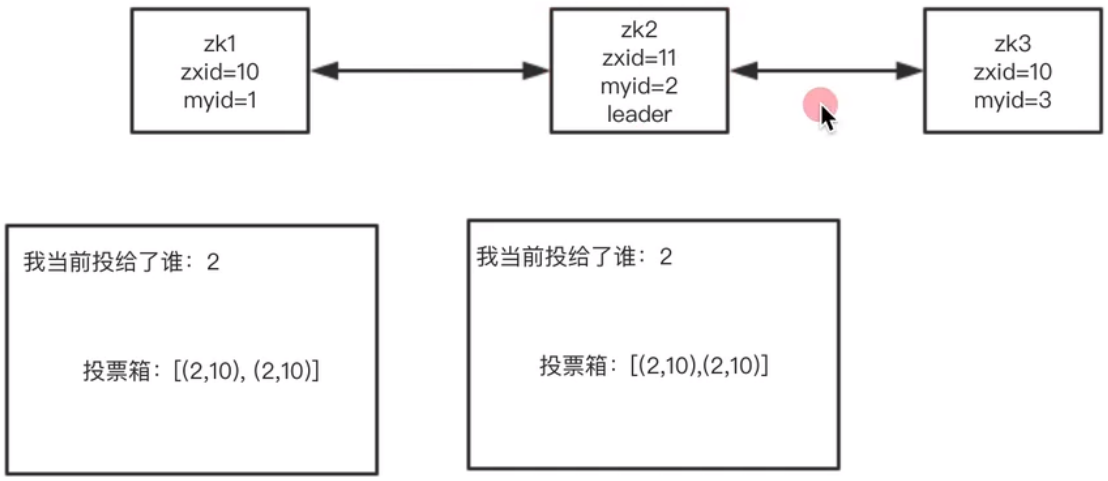
> **初次启动，对比节点id值大小**

【新节点加入】

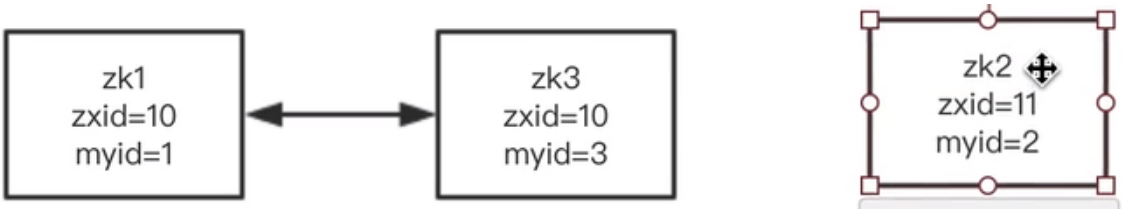


当有新的节点加入时，此时并不会触发leader的选举机制，而是会去查找集群中的leader，找到了就进行数据的同步。

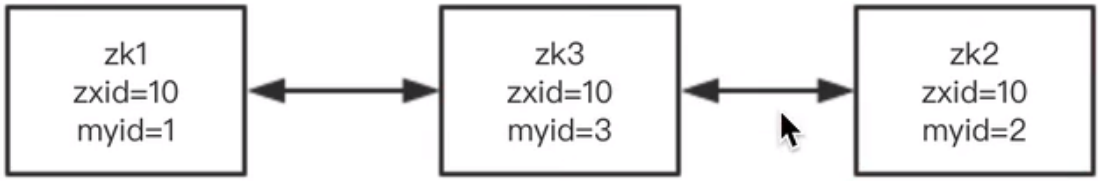
【同步出现异常】



zk2 leader 挂起，此时zk1和zk3重新发起选举，则zk3当选并同步数据。



此时zk2恢复并加入集群进行节点的注册，注册时zk2会将自己的myid和事务id发给leader，leader则进行对比，发现事务id不一致，且zk2的事务id超前，则会将leader自己的事务id发给zk2进行同步，相当于一个回滚的操作。



> **follower事务id超前则回滚，同步回leader的事务id**

5、同步流程

选完Leader以后，zk就进入状态同步过程。

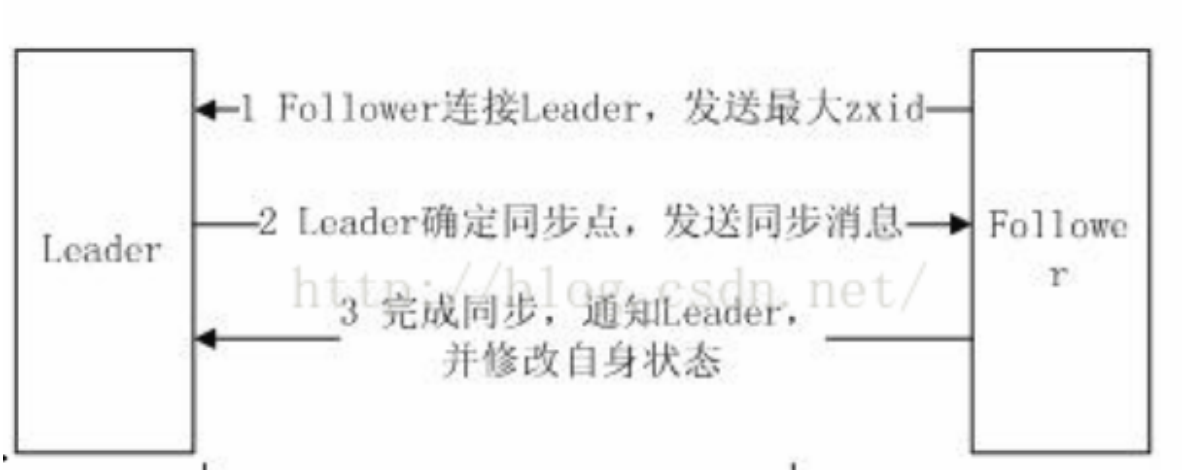
1.Leader等待server连接；

2 .Follower连接leader，将最大的zxid发送给leader；

3 .Leader根据follower的zxid确定同步点，将新数据发给follow；

4 .完成同步后通知follower已经成为uptodate状态；

5 .Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。



> **等待连接、发送事务id、开始同步、通知完成、恢复正常**

【确定同步点的流程】

leader把自己的事务id广播给follows，然后根据follower数量为每个follower创建一个LearnerHandler线程来处理同步请求，leader主线程阻塞，等待超过半数follower同步完数据之后成为正式leader。

follower接收到事务id后响应leader，告知自己的事务id。

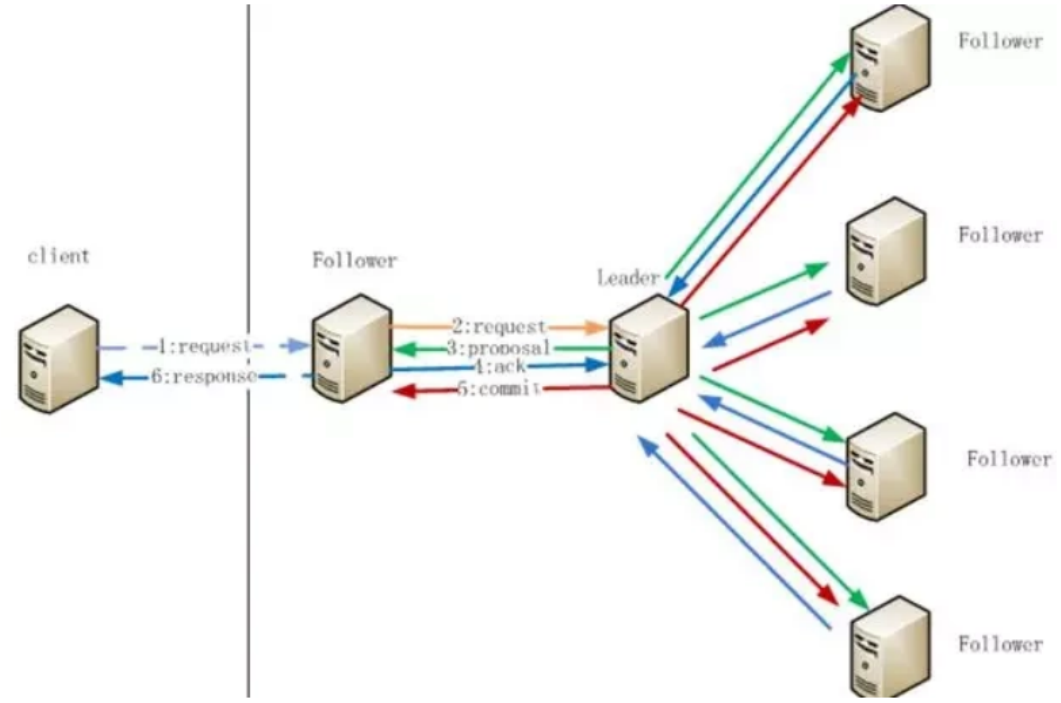
leader接收到回馈后开始判断：

1. 根据事务id判断，如果follower和leader数据一致，则直接发送DIFF告知已经同步，无需额外数据同步；
2. 如果数据不一致，判断是否有提交
   a）如果没有提交，则leader把快照发给follower进行同步
   b）如果有提交，则对比事务id
   如果follower事务id更大，则leader会要求follower删除多余的数据
   如果leader事务id更大，则leader将有差异的数据发送给follower进行同步

以上消息完毕后，LEADER发送UPTODATE包告知follower当前数据已同步，等待follower的ACK完成同步过程。

> **判断事务id、判断是否提交、判断事务id落地版本**

6、工作流程



1.Client向Follwer发出一个写的请求

2.Follwer把请求发送给Leader

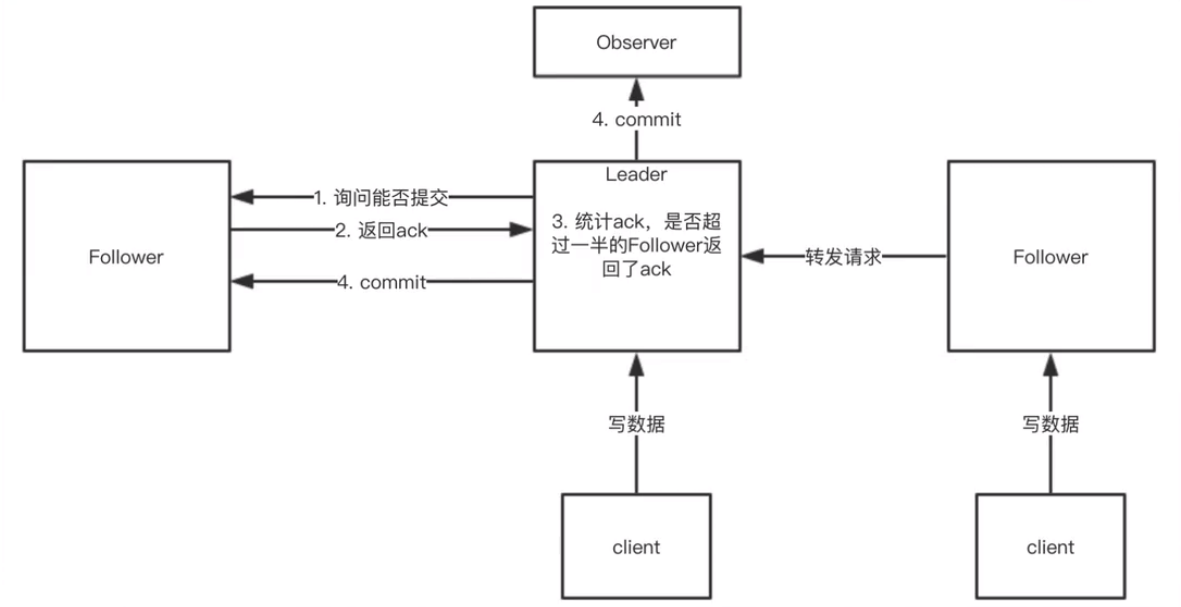
3.Leader接收到以后生成事务id，发起提议，写入本地事务日志（顺序写磁盘），并通知Follwer（预提交）

4.Follwer收到后将提议写入事务日志，回复ACK

5.Leader收到过半ack后，提交提议，更新内存的数据节点，并将事件广播给follower

6.follower收到后也提交，更新内存的数据节点

7.Follwer把请求结果返回给Client



zookeeper将数据以树结构组织起来并保存在内存中，所以对于数据节点的读写都发生在内存中。为了保证数据的持久化，系统会存在一个事务日志，当节点挂起时可以通过事务日志来恢复数据，保证数据的安全性和可靠性。

每当有一个事务请求过来时，会先把请求发给leader，leader将事务写入事务日志，然后顺序写入磁盘。紧接着把事务发给其他的follower节点，这一步为预提交，follower接收到以后也顺序写入磁盘，然后给leader回复一个ack，当集群超过半数的follower回复了ack以后，leader就会进行事务提交，把数据更新到内存的节点中，然后广播通知其他的follower告诉他们leader已经同步完成，此时收到该通知的其他follower也会进行事务的提交，以此来保证集群数据的一致性。

7、特点

【性能】

- 支持高可用，集群部署，半数以上节点存活，则服务仍然可用
- 高性能，分布式读写，而且读写分离，降低单个节点的负载，提高读写性能

  数据保存在内存中，这也就保证了 高吞吐量和低延迟（但是内存限制了能够存储的容量不太大，此限制也是保持znode中存储的数据量较小的进一步原因）

【一致性】

- 数据的全局一致性，每个节点都有副本，通过事务的原子广播和事务日志的持久化保证数据的全局一致性
- 更新请求顺序进行，来自同一个client的更新请求按其发送顺序依次执行
- 数据更新原子性，一次数据更新要么成功，要么失败

【功能】

- 临时节点的特性，客户端可以实现基于观察者模式下的事件驱动，也可以基于节点的唯一性来实现原子操作，比如分布式锁

8、应用

数据发布与订阅（配置中心）

可以作为一个配置中心，把数据保存到ZooKeeper节点上，客户端对节点进行监听，如果配置有变动，则ZooKeeper会通知相应的客户端，客户端再重新进行配置的调整。实现配置信息的集中管理和动态更新。

比如机器列表信息（负载均衡）、数据库配置信息等。这些全局配置信息通常具备以下3个特性。

- 数据量通常比较小
- 数据内容在运行时动态变化
- 集群中各机器共享，配置一致

发布/订阅系统一般有两种设计模式，分别是**推（Push）和拉（Pull）**模式。

推：服务端主动将数据更新发送给所有订阅的客户端。

拉：客户端主动发起请求来获取最新数据，通常客户端都采用定时轮询拉取的方式。

ZooKeeper 采用的是推拉相结合的方式。如下：

客户端想服务端注册自己需要关注的节点，一旦该节点的数据发生变更，那么服务端就会向相应的客户端发送Watcher事件通知，客户端接收到这个消息通知后，需要主动到服务端获取最新的数据（推拉结合）。

命名服务(Naming Service)、服务的注册与发现

通过使用命名服务，客户端应用能够根据指定**名字来获取资源或服务的地址**，提供者等信息，比如dubbo服务的提供者把服务的名称和服务的地址注册到了ZooKeeper中，服务的调用者到ZooKeeper中通过服务的名称就可以得到服务的地址，再进行远程调用。

比如两个服务信息在zk的数据节点分别为/HelloWorldService/1.0.0/100.100.0.237:16888和/HelloWorldService/1.0.0/100.100.0.238:16888。注册与发现流程如下：

注册与发现流程如下（kafka再均衡）：

1.服务提供者启动时，会将其服务名称，ip地址注册到配置中心。

2.服务消费者在第一次调用服务时，会通过注册中心找到相应的服务的IP地址列表，并缓存到本地，以供后续使用。当消费者调用服务时，不会再去请求注册中心，而是直接通过负载均衡算法从IP列表中取一个服务提供者的服务器调用服务。

3.当服务提供者的服务发生变化时（服务或服务器新增或删除），注册中心会回调事件通知到注册的客户端，客户端收到通知后，主动来注册中心拉取最新的服务列表信息。

分布式协调/通知

不同的客户端都对ZK上同一个 ZNode 进行注册，监听 ZNode 的变化（包括ZNode本身内容及子节点的），如果 ZNode 发生了变化，那么所有订阅的客户端都能够接收到相应的Watcher通知，并做出相应的处理。这是基于**Watcher 注册与异步通知的典型的观察者模式**，可以实现例如**负载均衡、leader选举和事件驱动**等。

比如kafka的broker controller的选举和分区再均衡。

心跳检测

在传统的开发中，我们通常是通过主机直接是否可以相互PING通来判断，更复杂一点的话，则会通过在机器之间建立长连接，通过TCP连接固有的心跳检测机制来实现上层机器的心跳检测，这些都是非常常见的心跳检测方法。

基于ZK的临时节点的特性，可以让不同的进程都在ZK的一个指定节点下创建临时子节点，不同的进程直接可以根据这个临时子节点来判断对应的进程是否存活。**通过这种方式，检测和被检测系统直接并不需要直接相关联，而是通过ZK上的某个节点进行关联，大大减少了系统耦合。**

集群管理

集群环境下，如何知道有多少台机器在工作？是否有机器退出或加入？需要选举一个总管master，让总管来管理集群。

在父目录GroupMembers下为所有机器创建临时目录节点，然后监听父目录节点的子节点变化，一旦有机器挂掉，该机器与ZooKeeper的连接断开，其所创建的临时目录节点被删除，所有其他机器都会收到通知。当有新机器加入时也是同样的道理。

为所有机器创建临时顺序编号目录节点，给每台机器编号，然后每次选取编号最小的机器作为master。

Master选举

ZooKeepr可以保证在高并发的时候节点创建的唯一性，多个客户端请求创建同一个临时节点，只有一个客户端请求能够创建成功。利用这个特性，就能很容易地在分布式环境中进行 Master 选举了。成功创建该节点的客户端所在的机器就成为了 Master。

同时，其他没有成功创建该节点的客户端，都会在该节点上注册一个子节点变更的 Watcher，用于监控当前 Master 机器是否存活，一旦发现当前的Master挂了，那么其他客户端将会重新进行 Master 选举。

分布式锁

比如排他锁。ZooKeeper 上的一个 ZNode 可以表示一个锁。例如 /exclusive_lock/lock节点就可以被定义为一个锁。

把ZooKeeper上的一个ZNode看作是一个锁，获得锁就通过创建 ZNode 的方式来实现。所有客户端都去 /exclusive_lock节点下创建临时子节点 /exclusive_lock/lock。ZooKeeper 会保证在所有客户端中，最终只有一个客户端能够创建成功，那么就可以认为该客户端获得了锁。同时，所有没有获取到锁的客户端就需要到/exclusive_lock节点上注册一个子节点变更的Watcher监听，以便实时监听到lock节点的变更情况。

因为 /exclusive_lock/lock 是一个临时节点，因此在以下两种情况下，都有可能释放锁。当前获得锁的客户端机器发生宕机或重启，那么该临时节点就会被删除，释放锁。正常执行完业务逻辑后，客户端就会主动将自己创建的临时节点删除，释放锁。无论在什么情况下移除了lock节点，ZooKeeper 都会通知所有在 /exclusive_lock 节点上注册了节点变更 Watcher 监听的客户端。这些客户端在接收到通知后，再次重新发起分布式锁获取，即重复『获取锁』过程。

> 取锁：创建节点成功，其他客户端监听节点状态
>
> 解锁：锁主挂起或正常业务执行完，节点删除，其他客户端监听到，再次进行取锁

9、ZooKeeper的使用

【安装与部署】

Step1：配置JAVA环境，检验环境：java -version

Step2：下载并解压zookeeper

```
# cd /usr/local# wget
http://mirror.bit.edu.cn/apache/zookeeper/stable/zookeeper-3.4.12.tar.gz# tar -
zxvf zookeeper-3.4.12.tar.gz# cd zookeeper-3.4.12
```

Step3：重命名配置文件zoo_sample.cfg

```
# cp conf/zoo_sample.cfg conf/zoo.cfg
```

Step4：启动zookeeper

```
# bin/zkServer.sh start
```

Step5：检测是否成功启动，用zookeeper客户端连接下服务端

```
# bin/zkCli.sh
```

【命令操作】

1、使用 ls 命令来查看当前 ZooKeeper 中所包含的内容

```
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper]
```


2、创建一个新的 znode ， 使用 create /zkPro myData

```
[zk: localhost:2181(CONNECTED) 1] create /zkPro myData
Created /zkPro
```


3、再次使用 ls 命令来查看现在 zookeeper 中所包含的内容：

```
[zk: localhost:2181(CONNECTED) 3] ls /
[zookeeper, zkPro]
```


4、下面我们运行 get 命令来确认第二步中所创建的 znode 是否包含我们所创建的字符串：

```
[zk: localhost:2181(CONNECTED) 4] get /zkPro
myData
cZxid = 0x100000014
ctime = Wed Jul 11 22:21:07 CST 2018
mZxid = 0x100000014
mtime = Wed Jul 11 22:21:07 CST 2018
pZxid = 0x100000014
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
```


5、下面我们通过 set 命令来对 zk 所关联的字符串进行设置：

```
[zk: localhost:2181(CONNECTED) 5] set /zkPro myData123
cZxid = 0x100000014
ctime = Wed Jul 11 22:21:07 CST 2018
mZxid = 0x100000015
mtime = Wed Jul 11 22:24:47 CST 2018
pZxid = 0x100000014
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 9
numChildren = 0
```


6、下面我们将刚才创建的 znode 删除

```
[zk: localhost:2181(CONNECTED) 6] delete /zkPro
[zk: localhost:2181(CONNECTED) 7]
```


【集群部署】

Step1：配置JAVA环境，检验环境：java -version

Step2：下载并解压zookeeper

```
# cd /usr/local# wget
http://mirror.bit.edu.cn/apache/zookeeper/stable/zookeeper-3.4.12.tar.gz# tar -
zxvf zookeeper-3.4.12.tar.gz# cd zookeeper-3.4.12
```

Step3：重命名 zoo_sample.cfg文件

```
# cp conf/zoo_sample.cfg conf/zoo-1.cfg
```

Step4：修改配置文件zoo-1.cfg，原配置文件里有的，修改成下面的值，没有的则加上

```
# vim conf/zoo-1.cfgdataDir=/tmp/zookeeper-
1clientPort=2181server.1=127.0.0.1:2888:3888server.2=127.0.0.1:2889:3889server.3
=127.0.0.1:2890:3890
```

配置说明

- tickTime：这个时间是作为 Zookeeper 服务器之间或客户端与服务器之间维持心跳的时间间隔，也就是每个 tickTime 时间就会发送一个心跳。
- initLimit：这个配置项是用来配置 Zookeeper 接受客户端（这里所说的客户端不是用户连接 Zookeeper 服务器的客户端，而是 Zookeeper 服务器集群中连接到 Leader 的 Follower 服务器）初始化连接时最长能忍受多少个心跳时间间隔数。当已经超过 10个心跳的时间（也就是 tickTime）长度后 Zookeeper 服务器还没有收到客户端的返回信息，那么表明这个客户端连接失败。总的时间长度就是 10*2000=20 秒
- syncLimit：这个配置项标识 Leader 与 Follower 之间发送消息，请求和应答时间长度，最长不能超过多少个 tickTime 的时间长度，总的时间长度就是 5*2000=10秒
- dataDir：顾名思义就是 Zookeeper 保存数据的目录，默认情况下，Zookeeper 将写数据的日志文件也保存在这个目录里。
- clientPort：这个端口就是客户端连接 Zookeeper 服务器的端口，Zookeeper 会监听这个端口，接受客户端的访问请求。
- server.A=B：C：D：其中 A 是一个数字，表示这个是第几号服务器；B 是这个服务器的 ip 地址；C 表示的是这个服务器与集群中的 Leader 服务器交换信息的端口；D 表示的是万一集群中的 Leader 服务器挂了，需要一个端口来重新进行选举，选出一个新的 Leader，而这个端口就是用来执行选举时服务器相互通信的端口。如果是伪集群的配置方式，由于 B 都是一样，所以不同的 Zookeeper 实例通信端口号不能一样，所以要给它们分配不同的端口号。

Step4：再从zoo-1.cfg复制两个配置文件zoo-2.cfg和zoo-3.cfg，只需修改dataDir和clientPort不同即可

```
# cp conf/zoo-1.cfg conf/zoo-2.cfg# cp conf/zoo-1.cfg conf/zoo-3.cfg# vim
conf/zoo-2.cfgdataDir=/tmp/zookeeper-2clientPort=2182# vim conf/zoo-
2.cfgdataDir=/tmp/zookeeper-3clientPort=2183
```

Step5：标识Server ID

创建三个文件夹/tmp/zookeeper-1，/tmp/zookeeper-2，/tmp/zookeeper-2，在每个目录中创建文件 myid 文件，写入当前实例的server id，即1.2.3

```
# cd /tmp/zookeeper-1# vim myid1# cd /tmp/zookeeper-2# vim myid2# cd
/tmp/zookeeper-3# vim myid3
```

Step6：启动三个zookeeper实例

```
# bin/zkServer.sh start conf/zoo-1.cfg# bin/zkServer.sh start conf/zoo-2.cfg#
bin/zkServer.sh start conf/zoo-3.cfg
```

Step7：检测集群状态，也可以直接用命令"zkCli.sh -server IP:PORT"连接zookeeper服务端检测

```
[root@centos-new zookeeper-3.4.12]# bin/zkServer.sh status conf/zoo-1.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo-1.cfg
Mode: follower
[root@centos-new zookeeper-3.4.12]# bin/zkServer.sh status conf/zoo-2.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo-2.cfg
Mode: leader
[root@centos-new zookeeper-3.4.12]# bin/zkServer.sh status conf/zoo-3.cfg
ZooKeeper JMX enabled by default
Using config: conf/zoo-3.cfg
Mode: follower
```


启动zookeeper

```
cd bin
./zkServer.sh start | stop | status | restart  # 启动|停止|查看状态|重启
```

客户端连接zookeeper

```
./zkCli.sh  # 启动客户端，默认连接本机的2181端口
或
./zkCli.sh -server 服务器地址：端口 # 连接指定主机、指定端口的zookeeper
quit  # 退出客户端
```

配置文件

配置项	含义	说明
tickTime=2000	心跳时间	维持心跳的时间间隔，单位是毫秒 在zookeeper中所有的时间都是以这个时间为基础单元，进行整数倍配置
initLimit=10	初始通信时限	用于zookeeper集群，此时有多台zookeeper服务器，其中一个为Leader，其他都为Follower
syncLimit=5	同步通信时限	在运行时Leader通过心跳检测与Follower进行通信，如果超过syncLimit*tickTime时间还未收到响应，则认为该Follower已经宕机
dataDir=../data	存储数据的目录	数据文件也称为snapshot快照文件
clientPort=2181	端口号	默认为2181
maxClientCnxns=60	单个客户端的最大连接数限制	默认为60，可以设置为0，表示没有限制
autopurge.snapRetainCount=3	保留文件的数量	默认3个
autopurge.purgeInterval=1	自动清理快照文件和事务日志的频率	默认为0，表示不开启自动清理，单位是小时
dataLogDir=	存储日志的目录	未指定时日志文件也存放在dataDir中，为了性能最大化，一般建议把dataDir和dataLogDir分别放到不同的磁盘上

常用命令

命令	作用	说明
help	查看帮助	查看所有操作命令
ls 节点路径	查看指定节点下的内容	
ls2 节点路径	查看指定节点的详细信息	查看所有子节点和当前节点的状态
create 节点路径 内容	创建普通节点	如果内容中有空格，则需要使用对双引号引起来

命令	作用	说明
get 节点路径	获取节点中的值	
create -e 节点路径 内容	创建临时节点	当连接断开后，节点会被自动删除
create -s 节点路径 内容	创建顺序编号节点	即带序号的节点
delete 节点路径	删除节点	只能删除空节点，即不能有子节点
rmr 节点路径	递归删除节点	remove recursion
stat 节点路径	查看节点状态	
set 节点路径 新值	修改节点内容	

集群在每台服务器的conf/zoo.cfg文件中添加如下内容：

```
server.20=192.168.4.20:2888:3888
server.21=192.168.4.21:2888:3888
server.22=192.168.4.22:2888:3888
```

```
格式： server.A=B:C:D
A表示这台服务器的编号ID，是一个数字
B表示服务器的IP地址或域名
C表示这台服务器与集群中的Leader交换信息时使用的端口
D表示执行选举Leader服务器时互相通信的端口
```

总结

概述

ZooKeeper 是一个分布式数据一致性的解决方案和框架，通过原子广播协议和事务日志记录的方式来保证当leader挂起时可以进行选举以及数据的同步和恢复等来保证数据的一致性。通过 ZooKeeper 可以实现的功能有服务的注册与发现，比如dubbo用到了zk作为注册中心；kafka 用到了 zk 作为几个客户端的元数据注册中心，分布式锁和负载均衡等功能。

角色

ZooKeeper 中有几个角色，leader发起投票，分发提议，发起提交，follower只读不写，observer为了避免当节点过多投票延迟过大影响性能而进行的水平扩展，只转发请求和同步状态给leader，提高读取速度。这种角色安排有实现了读写的分离，降低了单个节点的负载压力，提高了读写的性能。

属性（事务id、会话、版本、权限）

事务id用来表示数据节点发生改变的唯一递增标识（客户端请求导致数据变化）。会话的作用是维持心跳、处理请求和响应、承载监听回调等；有会话维持时间。节点数据包含了当前节点和当前节点子节点的版本信息和权限控制信息。权限控制包括节点权限的增删改查。

数据模型

数据模型是一棵树，数据以树形结构存储在内存中，节点是目录或者文件，节点保存数据内容和节点属性（事务id、创建时间、版本）。节点类型有四种，持久化、持久化顺序、临时、临时顺序节点等。

ZAB协议

ZAB有两种模式，恢复、同步（服务启动、运行时中断）。时机是启动、leader挂起、follower不足半数。同步机制，预提交（事务日志），收到半数ack，提交（更新内存数据 - 树节点），广播提交。

选举策略

初次启动，对比节点id值大小，过半优选。过程中选举，事务id对比，值越大则数据越新，胜率越大。首轮投自己，之后对比其他，然后在投票箱中更新，最后过半优选。

同步流程

分配线程、阻塞等待、

发送事务id、判断事务id、判断是否提交、对比事务id

完成同步、发送通知、恢复正常

参考资料

5分钟带你快速了解Zookeeper工作原理
https://segmentfault.com/a/1190000022431516

可能是全网把 ZooKeeper 概念讲的最清楚的一篇文章
https://zhuanlan.zhihu.com/p/44731983

深入浅析ZooKeeper的工作原理
https://www.jb51.net/article/139661.htm

为什么需要 Zookeeper
https://zhuanlan.zhihu.com/p/69114539?utm_source=wechat_session

zookeeper的数据存储和同步
https://www.iteye.com/blog/kavy-2119112

Zookeeper-数据同步
https://www.cnblogs.com/youngchaolin/p/13211752.html