Zookeeper

今日重点：

1. Zookeeper的特性
2. Zookeeper的数据模型
3. Zookeeper的节点类型

## Zookeeper基本知识

### ZooKeeper概述

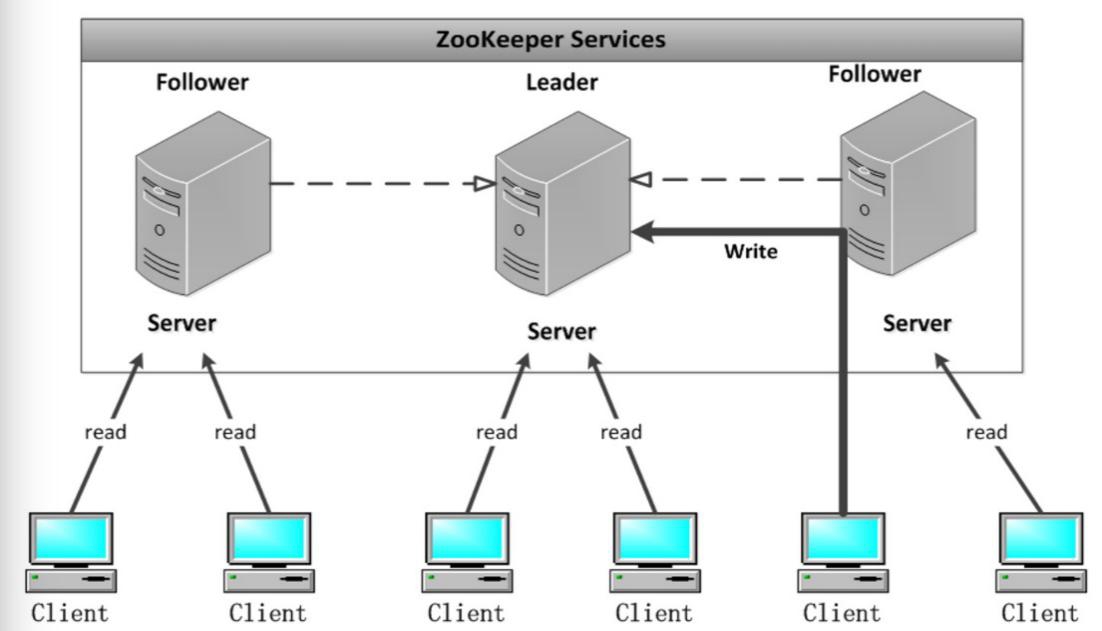
Zookeeper是一个分布式协调服务的开源框架。主要用来解决分布式集群中应用系统的一致性问题。

ZooKeeper本质上是一个**分布式的小文件存储系统**。提供基于类似于文件系统的目录树方式的数据存储，并且可以对树中的节点进行有效管理。从而用来维护和监控你存储的数据的状态变化。通过监控这些数据状态的变化，从而可以达到基于数据的集群管理。

### ZooKeeper特性

1. **全局数据一致**：集群中每个服务器保存一份相同的数据副本，client无论连接到哪个服务器，展示的数据都是一致的，这是最重要的特征；
2. 可靠性：如果消息被其中一台服务器接受，那么将被所有的服务器接受。
3. 顺序性：包括全局有序和偏序两种：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；偏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。
4. 数据更新原子性：一次数据更新要么成功（半数以上节点成功），要么失败，不存在中间状态；
5. 实时性：Zookeeper保证客户端将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。

### ZooKeeper集群角色



**Leader:**

Zookeeper集群工作的核心

事务请求（写操作）的唯一调度和处理者，保证集群事务处理的顺序性；

集群内部各个服务器的调度者。

对于create，setData，delete等有写操作的请求，则需要统一转发给leader处理，leader需要决定编号、执行操作，这个过程称为一个事务。

**Follower:**

处理客户端非事务（读操作）请求，转发事务请求给Leader；

参与集群Leader选举投票。

此外，针对访问量比较大的zookeeper集群，还可新增观察者角色。

**Observer:**

观察者角色，观察Zookeeper集群的最新状态变化并将这些状态同步过来，其对于非事务请求可以进行独立处理，对于事务请求，则会转发给Leader服务器进行处理。

不会参与任何形式的投票只提供非事务服务，通常用于在不影响集群事务处理能力的前提下提升集群的非事务处理能力。

## ZooKeeper集群搭建

Zookeeper集群搭建指的是ZooKeeper分布式模式安装。通常由2n+1台server组成。这是因为为了保证Leader选举（基于Paxos算法的实现）能过得到多数的支持，所以ZooKeeper集群的数量一般为奇数。

Zookeeper运行需要java环境，所以需要提前安装jdk。对于安装leader+follower模式的集群，大致过程如下：

* 配置主机名称到IP地址映射配置
* 修改ZooKeeper配置文件
* 远程复制分发安装文件
* 设置myid
* 启动ZooKeeper集群

如果要想使用Observer模式，可在对应节点的配置文件添加如下配置：

peerType=observer

其次，必须在配置文件指定哪些节点被指定为Observer，如：

server.1:node1:2181:3181:observer

其次，必须在配置文件指定哪些节点被指定为 Observer，如：  
server.1:localhost:2181:3181:observer

这里，我们安装的是leader+follower模式

|  |  |  |
| --- | --- | --- |
| 服务器IP | 主机名 | myid的值 |
| 192.168.88.161 | node1 | 1 |
| 192.168.88.162 | node2 | 2 |
| 192.168.88.163 | node3 | 3 |

### 第一步：下载zookeeeper的压缩包，下载网址如下

<http://archive.apache.org/dist/zookeeper/>

我们在这个网址下载我们使用的zk版本为3.4.6

下载完成之后，上传到我们的linux的/export/software路径下准备进行安装

### 第二步：解压

在node1主机上，解压zookeeper的压缩包到/export/server路径下去，然后准备进行安装

|  |
| --- |
| **cd** **/**export**/**software  **tar** **-**zxvf zookeeper-3.4.6.tar.gz -C **/**export**/**server**/** |

### 第三步：修改配置文件

在node1主机上，修改配置文件

|  |
| --- |
| **cd** **/**export**/**server**/**zookeeper-3.4.6**/**conf**/**  **cp** zoo\_sample.cfg zoo.cfg  **mkdir** -p **/**export**/**server**/**zookeeper-3.4.6**/**zkdatas**/**  **vim** zoo.cfg |

修改以下内容

|  |
| --- |
| #Zookeeper的数据存放目录  dataDir**=/**export**/**server**/**zookeeper-3.4.6**/**zkdatas  # 保留多少个快照  autopurge.snapRetainCount**=**3  # 日志多少小时清理一次  autopurge.purgeInterval**=**1  # 集群中服务器地址  server.1**=**node1**:**2888**:**3888  server.2**=**node2**:**2888**:**3888  server.3**=**node3**:**2888**:**3888 |

### 第四步：添加myid配置

在node1主机的/export/server/zookeeper-3.4.6/zkdatas/这个路径下创建一个文件，文件名为myid ,文件内容为1

|  |
| --- |
| **echo** 1 **>** **/**export**/**server**/**zookeeper-3.4.6**/**zkdatas**/**myid |

### 第五步：安装包分发并修改myid的值

在node1主机上，将安装包分发到其他机器

第一台机器上面执行以下两个命令

|  |
| --- |
| scp -r **/**export**/**server**/**zookeeper-3.4.6**/** node2**:/**export**/**server**/**  scp -r **/**export**/**server**/**zookeeper-3.4.6**/** node3**:/**export**/**server**/** |

第二台机器上修改myid的值为2

|  |
| --- |
| **echo** 2 **>** **/**export**/**server**/**zookeeper-3.4.6**/**zkdatas**/**myid |

第三台机器上修改myid的值为3

|  |
| --- |
| **echo** 3 **>** **/**export**/**server**/**zookeeper-3.4.6**/**zkdatas**/**myid |

### 第六步：三台机器启动zookeeper服务

三台机器分别启动zookeeper服务

这个命令三台机器都要执行

|  |
| --- |
| **/**export**/**server**/**zookeeper-3.4.6**/**bin**/**zkServer.sh start |

三台主机分别查看启动状态

|  |
| --- |
| **/**export**/**server**/**zookeeper-3.4.6**/**bin**/**zkServer.sh status |

### 配置Path环境变量

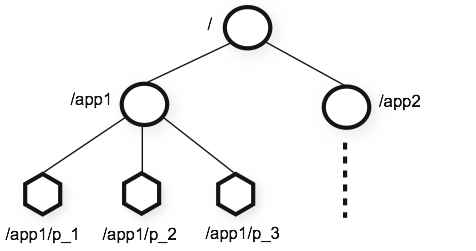
1：分别在三台中，修改/etc/proflie,添加以下内容

export ZOOKEEPER\_HOME=/export/server/zookeeper-3.4.6

export PATH=:$ZOOKEEPER\_HOME/bin:$PATH

2：分别在三台主机中,source /etc/profile

## Zookeeper数据模型



图中的每个节点称为一个Znode。 每个Znode由3部分组成:

ZooKeeper的数据模型，在结构上和标准文件系统的非常相似，拥有一个层次的命名空间，都是采用树形层次结构，ZooKeeper树中的每个节点被称为—Znode。和文件系统的目录树一样，ZooKeeper树中的每个节点可以拥有子节点。但也有不同之处：

1. **Znode兼具文件和目录两种特点,**既像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分，并可以具有子Znode。用户对Znode具有增、删、改、查等操作（权限允许的情况下）。
2. **Znode具有原子性操作**，读操作将获取与节点相关的所有数据，写操作也将替换掉节点的所有数据。另外，每一个节点都拥有自己的ACL(访问控制列表)，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作。
3. **Znode存储数据大小有限制,**ZooKeeper虽然可以关联一些数据，但并没有被设计为常规的数据库或者大数据存储，相反的是，它用来管理调度数据，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据的共同特性就是它们都是很小的数据，通常以KB为大小单位。ZooKeeper的服务器和客户端都被设计为严格检查并限制每个Znode的数据大小至多1M，当时常规使用中应该远小于此值。
4. **Znode通过路径引用，**如同Unix中的文件路径。路径必须是绝对的，因此他们必须由斜杠字符来开头。除此以外，他们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。在ZooKeeper中，路径由Unicode字符串组成，并且有一些限制。字符串"/zookeeper"用以保存管理信息，比如关键配额信息。
5. stat：此为状态信息, 描述该Znode的版本, 权限等信息

② data：与该Znode关联的数据 {“ip”：“192.168.23.45”，“status”:”running”}

③ children：该Znode下的子节点

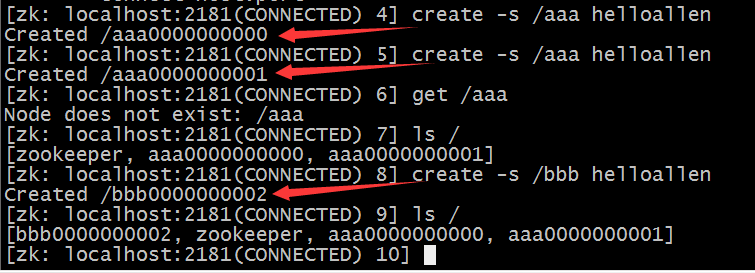
## Zookeeper节点类型

Znode有两种，分别为临时节点和永久节点。

节点的类型在创建时即被确定，并且不能改变。

临时节点：该节点的生命周期依赖于创建它们的会话。一旦会话结束，临时节点将被自动删除，当然可以也可以手动删除。临时节点不允许拥有子节点。  
　　永久节点：该节点的生命周期不依赖于会话，并且只有在客户端显示执行删除操作的时候，他们才能被删除。

Znode还有一个序列化的特性，如果创建的时候指定的话，该Znode的名字后面会自动追加一个不断增加的序列号。序列号对于此节点的父节点来说是唯一的，这样便会记录每个子节点创建的先后顺序。它的格式为“%10d”(10位数字，没有数值的数位用0补充，例如“0000000001”)。



这样便会存在四种类型的Znode节点，分别对应：

PERSISTENT：永久节点

EPHEMERAL：临时节点

PERSISTENT\_SEQUENTIAL：永久节点、序列化

EPHEMERAL\_SEQUENTIAL：临时节点、序列化

## ZooKeeper的shell操作

### 客户端连接

运行 zkCli.sh –server ip 进入命令行工具。

|  |
| --- |
| zkCli.sh **-**server node1**:**2181 |

### shell基本操作

#### 操作命令

| **命令** | **说明** | **参数** |
| --- | --- | --- |
| create [-s] [-e] path data [acl] | 创建Znode | -s 指定是顺序节点<br>-e 指定是临时节点 |
| ls path [watch] | 列出Path下所有子Znode |  |
| get path [watch] | 获取Path对应的Znode的数据和属性 |  |
| ls2 path [watch] | 查看Path下所有子Znode以及子Znode的属性 |  |
| set path data [version] | 更新节点 | version 数据版本 |
| delete path [version] | 删除节点, 如果要删除的节点有子Znode则无法删除 | version 数据版本 |
| rmr path | 删除节点, 如果有子Znode则递归删除 |  |
| setquota -n|-b val path | 修改Znode配额 | -n 设置子节点最大个数<br>-b 设置节点数据最大长度 |
| history | 列出历史记录 |  |

#### 操作实例

1：创建普通永久节点

create /app1 hello

2: 创建永久顺序节点

create -s /app2 world

3:创建临时节点

create -e /tempnode world

4:创建顺序的临时节点

create -s -e /tempnode2 aaa

5:获取节点数据

get /app1

6:修改节点数据

set /app1 hadoop

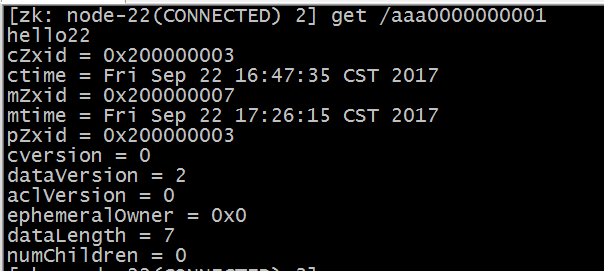
7:删除节点

delete /app1 删除的节点不能有子节点

rmr /app1 递归删除

#### 节点属性

每个znode都包含了一系列的属性，通过命令get，可以获得节点的属性。



dataVersion：数据版本号，每次对节点进行set操作，dataVersion的值都会增加1（即使设置的是相同的数据），可有效避免了数据更新时出现的先后顺序问题。

cversion ：子节点的版本号。当znode的子节点有变化时，cversion 的值就会增加1。

cZxid ：Znode创建的事务id。

mZxid ：Znode被修改的事务id，即每次对znode的修改都会更新mZxid。

对于zk来说，每次的变化都会产生一个唯一的事务id，zxid（ZooKeeper Transaction Id）。通过zxid，可以确定更新操作的先后顺序。例如，如果zxid1小于zxid2，说明zxid1操作先于zxid2发生，zxid对于整个zk都是唯一的，即使操作的是不同的znode。

ctime：节点创建时的时间戳.

mtime：节点最新一次更新发生时的时间戳.

ephemeralOwner:如果该节点为临时节点, ephemeralOwner值表示与该节点绑定的session id. 如果不是, ephemeralOwner值为0.

在client和server通信之前,首先需要建立连接,该连接称为session。连接建立后,如果发生连接超时、授权失败,或者显式关闭连接,连接便处于CLOSED状态, 此时session结束。

### ZooKeeper Watcher（监听机制）

ZooKeeper提供了分布式数据发布/订阅功能，一个典型的发布/订阅模型系统定义了一种一对多的订阅关系，能让多个订阅者同时监听某一个主题对象，当这个主题对象自身状态变化时，会通知所有订阅者，使他们能够做出相应的处理。

ZooKeeper中，引入了Watcher机制来实现这种分布式的通知功能。ZooKeeper允许客户端向服务端注册一个Watcher监听，当服务端的一些事件触发了这个Watcher，那么就会向指定客户端发送一个事件通知来实现分布式的通知功能。

触发事件种类很多，如：节点创建，节点删除，节点改变，子节点改变等。

总的来说可以概括Watcher为以下三个过程：客户端向服务端注册Watcher、服务端事件发生触发Watcher、客户端回调Watcher得到触发事件情况

#### Watch机制特点

**一次性触发**

事件发生触发监听，一个watcher event就会被发送到设置监听的客户端，这种效果是一次性的，后续再次发生同样的事件，不会再次触发。

**事件封装**

ZooKeeper使用WatchedEvent对象来封装服务端事件并传递。

WatchedEvent包含了每一个事件的三个基本属性：

通知状态（keeperState），事件类型（EventType）和节点路径（path）

**event异步发送**

watcher的通知事件从服务端发送到客户端是异步的。

**先注册再触发**

Zookeeper中的watch机制，必须客户端先去服务端注册监听，这样事件发送才会触发监听，通知给客户端。

#### 通知状态和事件类型

同一个事件类型在不同的通知状态中代表的含义有所不同，下表列举了常见的通知状态和事件类型。

事件封装: Watcher 得到的事件是被封装过的, 包括三个内容 keeperState, eventType, path

| **KeeperState** | **EventType** | **触发条件** | **说明** |
| --- | --- | --- | --- |
|  | None | 连接成功 |  |
| SyncConnected | NodeCreated | Znode被创建 | 此时处于连接状态 |
| SyncConnected | NodeDeleted | Znode被删除 | 此时处于连接状态 |
| SyncConnected | NodeDataChanged | Znode数据被改变 | 此时处于连接状态 |
| SyncConnected | NodeChildChanged | Znode的子Znode数据被改变 | 此时处于连接状态 |
| Disconnected | None | 客户端和服务端断开连接 | 此时客户端和服务器处于断开连接状态 |
| Expired | None | 会话超时 | 会收到一个SessionExpiredExceptio |
| AuthFailed | None | 权限验证失败 | 会收到一个AuthFailedException |

其中连接状态事件(type=None, path=null)不需要客户端注册，客户端只要有需要直接处理就行了。

#### Shell 客户端设置watcher

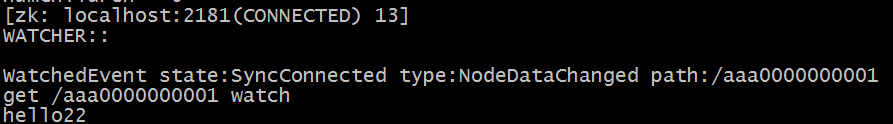
设置节点数据变动监听：



通过另一个客户端更改节点数据：



此时设置监听的节点收到通知：



## ZooKeeper Java API操作

这里操作Zookeeper的JavaAPI使用的是一套zookeeper客户端框架 Curator ，解决了很多Zookeeper客户端非常底层的细节开发工作 。

**Curator包含了几个包：**

**curator-framework**：对zookeeper的底层api的一些封装

**curator-recipes**：封装了一些高级特性，如：Cache事件监听、选举、分布式锁、分布式计数器等

Maven依赖(使用curator的版本：2.12.0，对应Zookeeper的版本为：3.4.x，如果跨版本会有兼容性问题，很有可能导致节点操作失败)：

### 引入maven坐标

|  |
| --- |
| **<**dependencies**>**  **<**dependency**>**  **<**groupId**>**org.apache.curator**</**groupId**>**  **<**artifactId**>**curator-framework**</**artifactId**>**  **<**version**>**2**.**12**.**0**</**version**>**  **</**dependency**>**  **<**dependency**>**  **<**groupId**>**org.apache.curator**</**groupId**>**  **<**artifactId**>**curator-recipes**</**artifactId**>**  **<**version**>**2**.**12**.**0**</**version**>**  **</**dependency**>**  **<**dependency**>**  **<**groupId**>**com.google.collections**</**groupId**>**  **<**artifactId**>**google-collections**</**artifactId**>**  **<**version**>**1**.**0**</**version**>**  **</**dependency**>**  **<**dependency**>**  **<**groupId**>**junit**</**groupId**>**  **<**artifactId**>**junit**</**artifactId**>**  **<**version**>**RELEASE**</**version**>**  **</**dependency**>**  **<**dependency**>**  **<**groupId**>**org.slf4j**</**groupId**>**  **<**artifactId**>**slf4j-simple**</**artifactId**>**  **<**version**>**1**.**7**.**25**</**version**>**  **</**dependency**>**  **</**dependencies**>**  **<**build**>**  **<**plugins**>**  **<!--** java编译插件 **-->**  **<**plugin**>**  **<**groupId**>**org.apache.maven.plugins**</**groupId**>**  **<**artifactId**>**maven-compiler-plugin**</**artifactId**>**  **<**version**>**3**.**2**</**version**>**  **<**configuration**>**  **<**source**>**1**.**8**</**source**>**  **<**target**>**1**.**8**</**target**>**  **<**encoding**>**UTF-8**</**encoding**>**  **</**configuration**>**  **</**plugin**>**  **</**plugins**>**  **</**build**>** |

### 节点的操作

|  |
| --- |
| /\*  创建节点  \*/  @Test  public void createZnode**()** **throws** Exception **{**  //1:定制一个重试策略  /\*  param1: 重试的间隔时间  param2:重试的最大次数  \*/  RetryPolicy retryPolicy **=** **new** ExponentialBackoffRetry**(**1000**,**1**);**  //2:获取一个客户端对象  /\*  param1:要连接的Zookeeper服务器列表  param2:会话的超时时间  param3:链接超时时间  param4:重试策略  \*/  String connectionStr **=** "192.168.88.161:2181,192.168.88.162:2181,192.168.88.163:2181"**;**  CuratorFramework client **=** CuratorFrameworkFactory**.**newClient**(**connectionStr**,** 8000**,** 8000**,** retryPolicy**);**  //3:开启客户端  client**.**start**();**  //4:创建节点  /\*  节点类型:  CreateMode.PERSISTENT:永久节点  CreateMode.PERSISTENT\_SEQUENTIAL:永久序列化节点  CreateMode.EPHEMERAL:临时节点  CreateMode.EPHEMERAL\_SEQUENTIAL:临时序列化节点  /hello2 :节点路径  world :节点数据  \*/  client**.**create**().**creatingParentsIfNeeded**().**withMode**(**CreateMode**.**PERSISTENT**).**forPath**(**"/hello2"**,**"world"**.**getBytes**());**  //5:关闭客户端  client**.**close**();**  **}** |

## ZooKeeper选举机制

zookeeper默认的算法是FastLeaderElection，采用投票数大于半数则胜出的逻辑。

### 概念

服务器ID

比如有三台服务器，编号分别是1,2,3。

编号越大在选择算法中的权重越大。

选举状态

LOOKING，竞选状态。

FOLLOWING，随从状态，同步leader状态，参与投票。

OBSERVING，观察状态,同步leader状态，不参与投票。

LEADING，领导者状态。

数据ID

服务器中存放的最新数据version。

值越大说明数据越新，在选举算法中数据越新权重越大。

逻辑时钟

也叫投票的次数，同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加，然后与接收到的其它服务器返回的投票信息中的数值相比，根据不同的值做出不同的判断。

### 全新集群选举

假设目前有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。

### 非全新集群选举

对于运行正常的zookeeper集群，中途有机器down掉，需要重新选举时，选举过程就需要加入数据ID、服务器ID和逻辑时钟。

数据ID：数据新的version就大，数据每次更新都会更新version。

服务器ID：就是我们配置的myid中的值，每个机器一个。

逻辑时钟：这个值从0开始递增,每次选举对应一个值。 如果在同一次选举中,这个值是一致的。

这样选举的标准就变成：

1、逻辑时钟小的选举结果被忽略，重新投票；

2、统一逻辑时钟后，数据id大的胜出；

3、数据id相同的情况下，服务器id大的胜出；

根据这个规则选出leader。