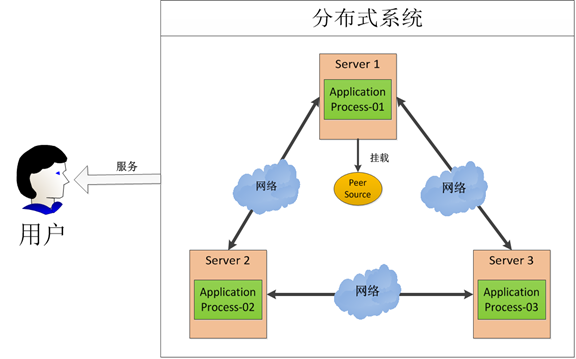
# 概述

## 1.1分布式协调技术

在学习zookeeper之前，我们先来了解一下分布式协调技术。分布式协调技术主要用来解决分布式环境当中多个进程之间的同步控制，让他们有序的去访问某种临界资源，防止出现"脏数据"。可能你会说了，为什么要使用分布式协调技术呢？我们直接写一个调度算法来协调各个进程，保持进程同步不就可以了吗？当然，在单机运行的情况下，这样做是没问题的，问题是现在处在分布式环境下，这些进程极有可能是跑在不同的机器上的，这就给我们的进程同步带来了困难



上图中进程01，02，03分别是跑在服务器1，2，3上的，各个进程之间通过网络进行连接构成一个完整的系统来为用户提供服务，这就是一个典型的分布式系统。当然，分布式系统在提高了系统性能的同时，也带来了一些令人头痛的问题

假设我们在Server2上挂载了一个不可共享的(排他性)的资源，在同一时刻只允许一个进程访问该资源，然而三个进程都想竞争这个资源，那么我们就需要一个协调机制来帮助进程有序访问该资源，这个协调机制就是锁，不过这个锁和我们常说的锁有所不同，它是分布式锁。当进程01在使用该资源的时候，会先去获得锁，进程01获得锁以后会对该资源保持**独占**，这样其他进程就无法访问该资源，进程01用完该资源以后就将锁释放掉，让其他进程来获得锁，那么通过这个锁机制，我们就能保证了分布式系统中多个进程能够有序的访问该临界资源。这种分布式环境下的锁被叫做**分布式锁，**分布式锁也就是我们**分布式协调技术**实现的核心内容

问题到了这儿，似乎变得简单了起来，我们只需要将单机环境下的锁，通过网络实现在分布式环境中。是的，这一切说起来很简单，但问题是，网络是不可靠的。比如，在同一台机器上，你对一个进程服务的调用如果成功，那就是成功，如果调用失败，比如抛出异常那就是调用失败。但是在分布式环境中，由于网络的不可 靠，你对一个服务的调用失败了并不表示一定是失败的，可能是执行成功了，但是响应返回的时候失败了。还有，A和B都去调用C服务，在时间上 A先发送调用请求，B后发送调用请求，但是由于网络延迟不一，并不一定是A先B后。麻烦的问题又来了，如何解决我们暂且放下，我们再来看另一个场景

淘宝卖家在后台上架一件的商品，通过服务器A提交到主数据库，假设刚提交后立马就有用户去通过应用服务器B去从数据库查询该商品，就会出现一个现象，卖家已经上架成功了，然而买家却看不到；而经过一段时间后，主数据库的数据同步到了从数据库，买家就能查到了。(关于数据库主从复制，自行查询资料)



在这一过程中：

假设卖家更新成功之后买家立马就能看到卖家的更新，则称为**强一致性**；

如果卖家更新成功后买家不能立刻看到卖家更新的内容，则称为**弱一致性**；

而卖家更新成功后，买家经过一段时间最终能看到卖家的更新，则称为**最终一致性**。

以上两个问题，都可以通过zookeeper来解决

## 1.2 Zookeeper

通过将一个整体业务分拆多个子业务，部署在不同的服务器上，可以缩短单个任务的执行时间，进一步提高业务处理效率，这就是**分布式**应用。分布式应用正在运行的单一子系统可以部署**集群**，即同一个子业务，部署在多个服务器上，通过提高单位时间内执行的业务数量来提升处理效率，而在集群中运行的每台机器被称为**节点**。

Zookeeper作为一个高可靠、高可用的分布式协调服务，为我们提供了完美的解决方案，它提供了一项基本服务：分布式锁服务。由于ZooKeeper的开源特性，后来我们的开发者在分布式锁的基础上，摸索了出了其他的使用方法：配置维护、组服务、分布式消息队列、分布式通知/协调等。

Zookeeper具有相当的可靠性，不会因为某个分布式节点的错误而崩溃，它严格的序列访问控制意味着它拥有强大的分布式锁的能力，zookeeper一般提供以下服务：

* **命名服务** - 按**名称**标识集群中的**节点**。它类似于DNS域名解析服务，但命名服务仅为**节点**和**名称**关联映射关系，而DNS是为ip地址和域名关联映射关系。
* **配置管理** – 管理节点的系统配置信息，比如数据库密码等配置，在集群环境下，只修改一台服务器的密码，如果一台一台地修改是不现实的，这就需要用到zookeeper的配置管理服务。
* **集群管理** - 实时地在集群和节点状态中加入/离开节点。
* **选举算法** - 选举一个节点作为协调目的的leader。
* **锁定和同步服务** - 为了允许在分布式系统中对共享资源进行有序的访问，可能需要实现分布式互斥，同时zookeeper也提供了同步服务
* **高度可靠的数据注册表** - 即使在一个或几个节点关闭时也可以获得数据。

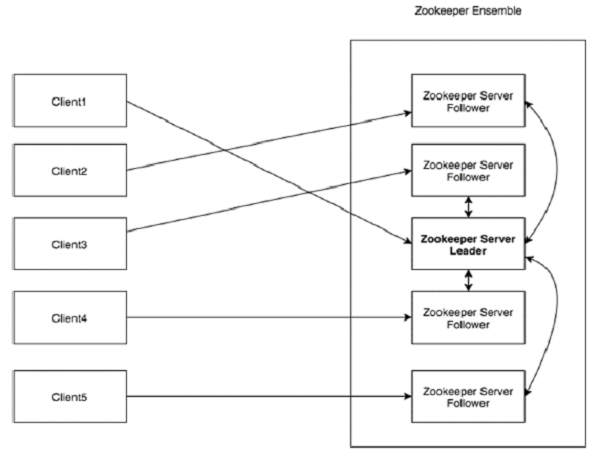
## 1.3 Zookeeper在dubbo中的作用

Dubbo是阿里开源的一款分布式微服务框架，具体请参见《Dubbo学习笔记》

Zookeeper为Dubbo提供服务注册功能。对于一个服务框架，注册中心是其核心中的核心，虽然暂时挂掉并不会导致整个服务体系出问题，但是一旦挂掉，整体风险就很高。Dubbo底层将zookeeper常用的客户端zkclient和curator封装成为ZookeeperClient

## 1.4 Zookeeper的基本原理

### 1.4.1 架构



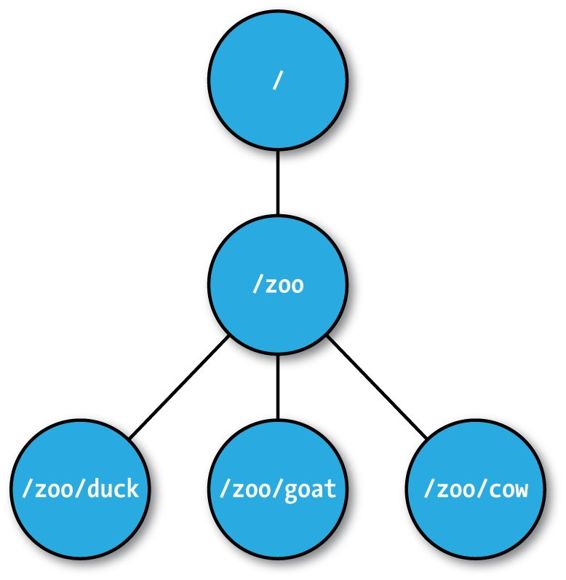
从上图可以看出，ZooKeeper是“客户端-服务器”架构。

**客户端**是我们的分布式应用集群中的一个节点，从服务器访问信息。每个客户端都会定期向它所连接的服务器发送消息以使服务器知道客户端是活跃的。被询问的 ZooKeeper **服务器**会进行响应，表示服务器也处于活动状态。如果客户端在指定时间内没有收到服务器的确认，那么客户端会连接到集合体中的另一台服务器，而且客户端会话会被透明地转移到新的 ZooKeeper 服务器。

**ZooKeeper Ensemble**是zookeeper服务器组，一般最少需要三个节点(这样可靠性上允许一台宕机)，节点数最好为单数，原因请参照1.4.3节半数以上投票

### 1.4.2 数据模型

Zookeeper本质上是一个类似Linux文件系统的树型分布式小文件存储系统，但是它没有文件和文件夹的概念，只有一个叫做znode的概念，它是树的节点，它既是数据的容器，也是其他节点的容器(有点类似于文件夹，文件夹既是文件的容器又是其他文件夹的容器)。每个znode由一个名称标识，并用路径(/)序列分隔。



需要注意的是存储在 znode 中的数据的默认最大大小为 1 MB。因此既是其类似于文件系统，也不能作为文件系统使用

### 1.4.3 特性

半数以上投票通过：客户端的增删改操作无论访问到了哪台zookeeper服务器，最终都会被转发给leader服务器，再由leader服务器分给zookeeper集群中所有follower服务器去投票（投票指的是在内存中做增删改操作），半数投票通过就被认为操作可执行（commit），否则不可执行

这一特性导致

3台服务器，至少2台正常运行才行（3的半数为1.5，半数以上最少为2），正常运行可以允许1台服务器挂掉

4台服务器，至少3台正常运行才行（4的半数为2，半数以上最少为3），正常运行可以允许1台服务器挂掉

5台服务器，至少3台正常运行才行（5的半数为2.5，半数以上最少为3），正常运行可以允许2台服务器挂掉

6台服务器，至少3台正常运行才行（6的半数为3，半数以上最少为4），正常运行可以允许2台服务器挂掉

通过以上可以发现，3台服务器和4台服务器都最多允许1台服务器挂掉，5台服务器和6台服务器都最多允许2台服务器挂掉

但是明显4台服务器成本高于3台服务器成本，6台服务器成本高于5服务器成本。因此选择单数台服务器进行集群部署，这是由于半数以上投票通过决定的。

1.**全局一致性：**每个zookeeper服务器都保存一份相同的数据副本(即相同的文件树)，无论客户端访问哪台服务器，展示的数据都是一致的

2.**可靠性**：如果消息(对文件系统中znode节点的增删改查)被zookeeper集群中的任意一台服务器接受，那么该消息也将被所有的服务器接受，这是为了保证所有服务器上的数据一致

3.**顺序性：**分为全局有序和

**全局有序：**如果在一台服务器上，消息a在消息b之前发布，则在所有服务器上a都在b之前发布，这也是为了保证全局一致性(这里的消息仍是对文件系统中znode节点的增删改查，后面不再赘述)

**全局偏序：**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(对文件系统中znode节点的增删改查)要么成功(半数以上服务器节点更新成功，就是投票机制)，要么失败，不存在中间状态，类似事务

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。但由于网络延时等原因，Zookeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用sync()接口。

### 1.4.4 角色

**Zookeeper中有三种角色：**

**1.Leader(总统)**作为整个ZooKeeper集群的主节点，负责响应所有对ZooKeeper状态变更的请求。它会将每个状态更新请求进行排序和编号。它是事务请求的唯一调度和处理者，保证集群事务处理的顺序性，也是集群内部各个服务器的调度者，一个 ZooKeeper 集群同一时刻只会有一个 Leader，其他都是 Follower 或 Observer。

那么什么是**状态更新请求(写操作)**呢？它是一种事务性请求(不同于数据库事务)，我们都知道，zookeeper本质上是一个小文件存储系统，其状态更新请求，实际上就是创建节点(create)、修改节点中的数据(setData)、和删除节点(delete)这几个操作。多个状态更新请求必须有序执行，设想一下，如果我们对某个znode同时发起两个请求(setData和delete)，如果delete先执行，那么setData就会失败，如果setData先执行，那么后面的delete必然会删除节点，影响到setData的后续操作，这就需要leader对其进行排序和编号来保证事务处理的顺序性

**这里需要注意的一点是：虽然客户端对某台Follower节点发起了状态更新请求(事务性请求，写操作)，但Follower无权处置，必须将其转发至leader，由leader进行全局编号排序和调度，这样就保证了顺序性。然后再有leader对所有节点发起事务性请求，如果半数以上写入成功，则视为事务性请求成功。**

**2.Follower(议员)**是leader节点的追随者。它主要负责处理客户端非事务性请求(读操作)，转发事务性请求给Leader服务器，参与事务请求Proposal的投票，参与Leader选举投票。但是随着在集合体中的Follower节点数量的增加，投票成本随之增加，写入性能必然会下降，因为必须将写入内容写入到更多的服务器中，并在更多服务器之间进行协调。

**leader和follower构成ZooKeeper集群的法定人数，也就是说，只有他们才参与新leader的选举投票、事务请求投票，而Observer无权投票，只能跟随。**

**3.Observer(观察者)**和Follower唯一的区别在于，Observer不参与任何形式的投票，包括事务请求Proposal的投票(半数以上修改成功视作成功)和Leader选举投票，除了不参与投票其他都和follower一样，收到事务性请求也会转发给leader处理，即使Observer宕机也不会对集群的可用性有任何影响。简单地讲，Observer服务器只提供非事物服务(读)，可以在不影响集群事务处理能力(写能力)的前提下提升集群的非事物处理能力(读能力)，因为它不参与投票，它仅用作提高读取操作的吞吐量

为了安全起见，可以在集合体中使用五个法定节点(leader+follower)。五个节点的集合体让我们拿出一台服务器进行维护或滚动升级，并能够在不中断服务的情况下承受第二台服务器的意外故障。因此，在 ZooKeeper 集合体中，3、5、7是最典型的节点数量

**但是需要注意：ZooKeeper 集合体的大小与分布式系统中的节点大小没有什么关系。分布式系统中的节点将是 ZooKeeper 集合体的客户端**

### 1.4.5 会话Session

会话对于ZooKeeper的操作非常重要。会话中的请求按FIFO(先入先出)顺序执行。一旦客户端连接到服务器，将建立会话并向客户端分配会话ID 。

以下是会话的五种状态，ZooKeeper会话在整个运行期间的生命周期中，会在不同的会话状态之间进行切换

|  |  |
| --- | --- |
| CONNECTING | 连接中 |
| CONNECTED | 已连接 |
| RECONNECTING | 重新连接中 |
| RECONNECTED | 已重新连接 |
| CLOSE | 关闭 |

　一旦客户端开始创建Zookeeper对象，那么客户端状态就会变成CONNECTING状态，同时客户端开始尝试连接服务端，连接成功后，客户端状态变为CONNECTED，通常情况下，由于断网或其他原因，客户端与服务端之间会出现断开情况，一旦碰到这种情况，Zookeeper客户端会自动进行重连服务，同时客户端状态再次变成CONNCTING，直到重新连上服务端后，状态又变为CONNECTED，在通常情况下，客户端的状态总是介于CONNECTING和CONNECTED之间。但是，如果出现诸如会话超时、权限检查或是客户端主动退出程序等情况，客户端的状态就会直接变更为CLOSE状态。

**客户端会在会话超时时间过期范围内向服务端定时发送PING请求来保持会话的有效性（这就是所谓的心跳检测），同时，服务端需要不断地接收来自客户端的心跳检测，并且需要重新激活对应的客户端会话(TouchSession)**。如果ZooKeeper集合在超过服务器启动时设置的指定时间后（会话超时）都没有从客户端接收到心跳，则它会判定客户端死机。实际上，每次客户端有请求(包括读或写请求)发送到服务端，都会进行一次会话激活

会话超时通常以毫秒为单位。当会话由于任何原因结束时，在该会话期间创建的临时节点也会被删除。

Session会话包含以下4个基本属性：

|  |  |
| --- | --- |
| sessionID | 会话ID，用来标识一个会话。每次客户端创建新会话的时候，ZooKeeper都会为其分配一个全局唯一的SessionID |
| TimeOut | 会话超时时长。客户端在构建ZooKeeper实例的时候，会配置一个sessionTimeOut参数用于指定会话的超时时间。ZooKeeper客户端向服务端发送这个超时时间后，服务器会根据自己的超时时间限制最终确定会话的超时时间 |
| TickTime | 下次会话超时时间点。 |
| isClosing | 该属性用语标记一个会话是否已经被关闭。 |

### 1.4.6 znode

Znode作为zookeeper树的节点，由三部分组成

1. stat：存储znode的状态信息， 描述该Znode的版本， 权限等信息

致使ZooKeeper节点状态改变的每一个操作都将使节点接收到一个Zxid格式的时间戳，并且这个时间戳全局有序，在整个zookeeper中都是唯一的。也就是说，也就是说，每个对节点的改变都将产生一个唯一的Zxid。如果Zxid1的值小于Zxid2的值，那么Zxid1所对应的事件发生在Zxid2所对应的事件之前。实际上，ZooKeeper的每个节点维护者三个Zxid值，为别为：cZxid、mZxid、pZxid。

**czxid**：创建节点的事务的zxid

**mzxid**：对znode最近修改的zxid，每次对znode的修改都会更新mzxid

**ctime**：以距离时间原点(epoch)的毫秒数表示的znode创建时间

**mtime**：以距离时间原点(epoch)的毫秒数表示的znode最近修改时间

**version**：znode数据的修改次数，对节点的每一个写操作(一般是set)都将致使这个节点的版本号增加1

**cversion**：znode所拥有的子节点修改次数，当子节点变化时，cversion+1

**aversion**：znode的ACL被修改次数

**ephemeralOwner**：如果znode是临时节点，则其值为节点所有者的会话ID；如果不是临时节点，则为零。

**dataLength**：znode数据长度。

**numChildren**：znode子节点个数。

2.data：Znode存储的数据，这些数据用来管理调度数据，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，一般以KB为单位，最多不超过1MB

3.children：该Znode下的子节点

Znode有三种(也可以说是四种，因为顺序节点有两种，持久和临时)，节点类型创建时即被确定并且不可改变：

**1.持久节点** - 即使在创建该特定znode的客户端断开连接后，持久节点仍然存在。默认情况下，除非另有说明，否则所有znode都是持久的。

**2.临时节点** - 客户端活跃时，临时节点就是有效的。当客户端与ZooKeeper集群会话失效时，临时节点会自动删除，当然也可以手动删除。因此，只有临时节点不允许有子节点。如果临时节点被删除，则下一个合适的节点将填充其位置。临时节点在leader选举中起着重要作用。

**3.顺序节点** - 顺序节点可以是持久的或临时的。每个父节点会为他的第一级子节点维护一份时序，会记录每个子节点创建的先后顺序。当一个新的znode被创建为一个顺序节点时，ZooKeeper通过将10位数字的序列号附加到原始名称来设置znode的路径。例如，如果将具有路径 /myapp 的znode创建为顺序节点，则ZooKeeper会将路径更改为 /myapp0000000001 ，并将下一个序列号设置为0000000002。如果两个顺序节点是同时创建的，那么ZooKeeper不会对每个znode使用相同的数字。顺序节点在锁定和同步中起重要作用。

Znode中一般有如下操作(包括事务性和非事务性)：



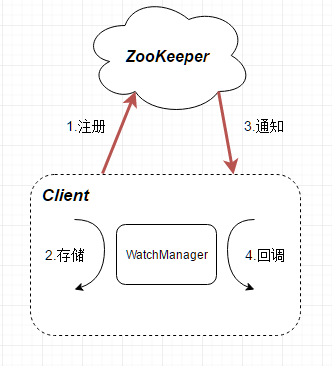
事务性ZooKeeper操作是有限制的。delete或setData必须明确要更新的Znode的版本号，我们可以调用exists找到。如果版本号不匹配，更新将会失败。

### 1.4.7 触发器watches

当Znode发生变化（Znode本身的增加，删除，修改，以及子Znode的变化），可以通过Watch机制通知到客户端

可以为所有的读操作设置watch，这些读操作包括：exists()、getChildren()及getData()。watch事件是一次性的触发器，当watch的对象状态发生改变时，将会触发此对象上watch所对应的事件，watch事件将被异步地发送给客户端。例如，如果客户端执行操作getData(“/znode1″， true)，而后/znode1 发生变更或是删除了，客户端都会得到一个/znode1 的watch事件。如果/znode1 再次发生变更，则在客户端没有设置新的watch的情况下，是不会再给这个客户端发送watch事件的。

客户端先向ZooKeeper服务端成功注册想要监听的节点状态，同时客户端本地会存储该监听器相关的信息在WatchManager中，当ZooKeeper服务端监听的数据状态发生变化时，ZooKeeper就会主动通知发送相应事件信息给相关会话客户端，客户端就会在本地响应式的回调相关Watcher的Handler。



Zookeeper提供了一个顺序保证，如果某个Znode设置了Watcher，且在Znode有变化的情况下通知到了客户端，但是在客户端接收到这个变化事件，但是还没有再次设置Watcher之前，如果其他客户端对该Znode做了修改，这种情况下，Znode第二次的变化客户端是无法收到通知的(因为watcher是一次性的)

当连接会话过期时，客户端将与服务器断开连接，相关的watches也将被删除。

Watcher分为两种：

数据观察 (data watches)：getData和exists负责设置数据watch

孩子观察 (child watches)：getChildren负责设置孩子watch

因此：

一个成功的setData操作将触发Znode的data watches

一个成功的create操作将触发Znode的data watches以及Znode父节点的child watches

一个成功的delete操作将触发Znode的data watches以及Znode父节点的child watches

该机制用于实现分布式发布订阅功能，该功能是一种一对多的关系，能够让多个订阅者监听一个主题对象，当主题对象发生改变时，通知所有订阅者

### 1.4.8工作流程

一旦ZooKeeper集合启动，它将等待客户端连接。客户端将连接到ZooKeeper集合中的一个节点。它可以是leader或follower节点。一旦客户端被连接，节点将向特定客户端分配会话ID并向该客户端发送确认。如果客户端没有收到确认，它将尝试连接ZooKeeper集合中的另一个节点。 一旦连接到节点，客户端将以有规律的间隔向节点发送心跳，以确保连接不会丢失。

如果客户端想要读取特定的znode，它将会向自己所连接的服务器发送读取请求，改服务器直接从本地返回所请求的znode。为此，在ZooKeeper集合中读取速度很快。

如果客户端想要将数据存储在ZooKeeper集合中(写请求)，则会将znode路径和数据发送到服务器。连接的服务器将该请求转发给leader，然后leader将向所有的follower重新发出写入请求。如果只有半数以上节点成功响应，而写入请求成功，则成功返回代码将被发送到客户端。 否则，写入请求失败。集群节点数量越多，写入代价越大，推荐五台服务器作为leader和follower

## 1.5 leader选举

**Looking**：寻找Leader状态，当Server处于该状态时，此Server会认为当前集群中没有Leader，需要进入Leader选举状态。

**Following**： 跟随者状态，表明该Server角色为Follower。

**Leading**： 领导者状态，表明当前服务器角色是Leader。

**Observing**： 观察者状态，表明当前服务器角色是Observer。

Leader选举是保证分布式数据一致性的关键所在。当Zookeeper集群中的一台服务器出现以下两种情况之一时，需要进入Leader选举。

**(1) 服务器初始化启动。**

**(2) 服务器运行期间无法和Leader保持连接。**

### 1.5.1初始化选举

若进行Leader选举，则至少需要两台机器，但一般至少为3台，我们这里假设有5台，并且依次启动

1) 服务器1启动，此时只有它一台服务器启动了，它一台服务器无法单独完成选举，所以它的选举状态一直是LOOKING状态

2) 服务器2启动，它与最开始启动的服务器1进行通信，互相交换自己的投票结果。由于是初始情况，Server1和Server2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的myid和ZXID，使用(myid, ZXID)来表示，SID即Server的id，ZXID是事务id，ZXID越大，代表Server的事务操作越新，因为是初始化，所以都为0。

此时Server1的投票为(1, 0)，Server2的投票为(2, 0)，然后各自将这个投票发给集群中其他机器(只有俩，当然是互发啦)。

3)收到投票后，集群中的每个Server先判断投票有效性，如检查是不是本轮的投票，是不是来Looking状态的服务器投的票等，然后在对投票进行处理，投票规则如下：

**1.首先对比ZXID。ZXID大的服务器优先作为Leader**

**2.若ZXID相同，比如初始化的时候，每个Server的ZXID都为0，就会比较myid，myid大的选出来做Leader。**

对于Server1，它自己的投票是(1, 0)，接收Server2的投票为(2, 0)，因此服务器2胜出，于是Server1更新自己的投票为(2, 0)，然后重新投票，对于Server2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

4)每次投票后，服务器都会统计投票数，判断是否已经有过半机器接受到相同的投票信息，对于Server1、Server2而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息（即server2获得2票），但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3，2<3)，所以服务器1，2还是继续保持Looking状态.

5) 服务器3启动，根据前面的理论分析，服务器3成为服务器1，2，3中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的leader，Leader将状态更新为Leading，Follower将状态更新为Following。

6) 服务器4启动，根据前面的分析，理论上服务器4应该是服务器1，2，3，4中最大的，但是由于前面已经有半数以上的服务器选举了服务器3，所以它只能接收当小弟的命了.

7) 服务器5启动，同4一样，当小弟.

### 1.5.2服务恢复选举

如果在服务中，leader宕机，那么集群将暂停对外服务，在内部重新选举leader，这里以3台服务器为例，server2为leader

1）变更状态。其他的非Observer服务器将自己的状态改变为Looking，开始进入Leader选举。

2） 每个Server发出一个投票（myid，ZXID），由于此集群已经运行过(非初始化)，所以每个Server上的ZXID可能不同。假设Server1的ZXID为145，Server3的为122，第一轮投票中，Server1和Server3都投自己，票分别为（1，145）、（3,122）,将自己的票发送给集群中所有机器。

3） 每个Server接收接收来自其他Server的投票，不断被别的服务器”说服”，接下来的步骤与启动时步骤相同。

## 1.6 原子广播