# 概述

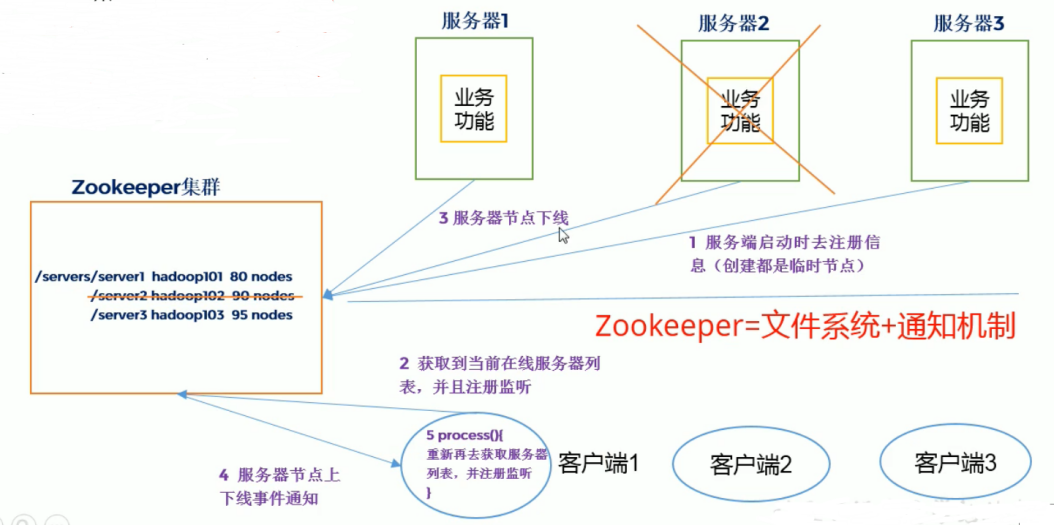
Zookeeper是一个典型的分布式数据一致性的解决方案，是由雅虎研究院开发，是Google Chubby的开源实现，后者托管到Apache。

分布式应用程序可以基于它实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、master选举和分布式队列等功能。著名的Hadoop、Kafka、Dubbo都是基于Zookeeper而构建的。

## 概述

**Zookeeper=文件系统+监听通知机制**。

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。



Zookeeper从设计模式角度来理解：是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后观察者的注册，一旦这些数据的状态发生变化，zookeeper就将负责通知已经在zookeeper上注册的那些观察者做出相应的反应。

## 特点

**zookeper的保证：**

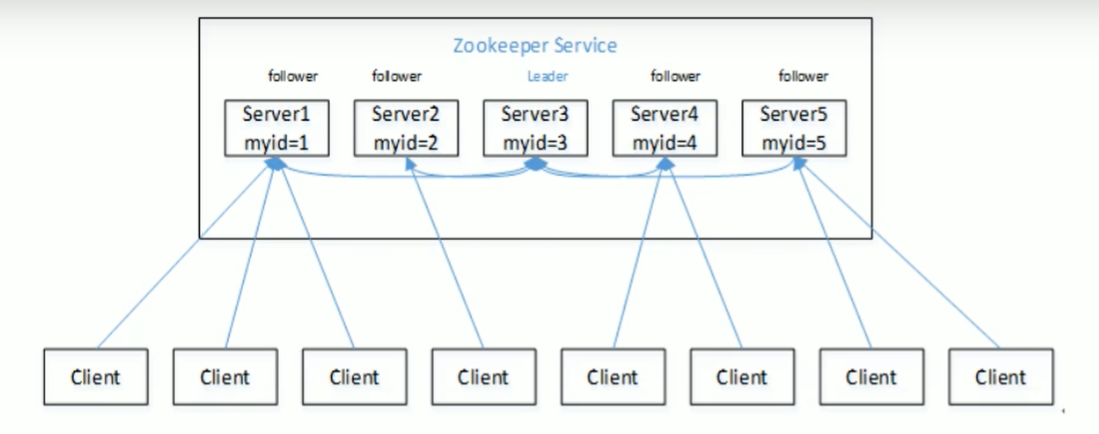
顺序一致性：按提交顺序应用；

原子性：事物操作要么成功，要么失败；

单一视图：同一客户端，无论连接哪个zk，视图一致；

可靠性：事物多副本持久化；

实时性：保证在一定时间内，客户端能够读到最新数据。



1. zookeeper：一个领导者（leader），多个跟随者（follower）组成的集群；
2. 集群只要有半数以上节点存活，zookeeper集群就能正常服务；
3. 全局数据一致性：每个server保存一份相同的数据副本，client无论连接到哪个server，数据都是一致的；
4. 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；
5. 数据更新原子性，一次数据更新要么成功，要么失败；
6. 实时性，在一定时间范围内，client能读到最新数据。

### 优点

非阻塞全部快照（达成最终一致）

高效的内存管理

高可靠

API 简单

连接管理可以自动重试

ZooKeeper recipes的实现是经过完整良好的测试的。

有一套框架使得写新的ZooKeeper recipes非常简单。

支持监听事件

发生网络分区时，各个区都会开始选举 leader，那么节点数少的那个分区将会停止运行

### 缺点

zookeeper是java写的，那么自然就会继承java的缺点，例如GC暂停。

**如果开启了快照，数据会写入磁盘，此时zookeeper的读写操作会有一个暂时的停顿**。

**对于每个watch请求，zookeeper都会打开一个新的socket连接，这样zookeeper就需要实时管理很多socket连接，比较复杂**。

实际使用上的问题暴露出的是ZK系统设计层面上的缺陷，总结下可以归纳为：

API事务能力不足，不支持客户端发起事务性的多步骤操作

服务器无仲裁能力，ZK服务器端不能做基本的判断逻辑，必须都在客户端进行

回调机制受限，Znode上面watch仅支持触发一次回调，不支持定时过期失效

可扩展性不足，ZK集群不支持在线动态添加机器或替换机器

Zab协议，协议在选举和发送环节都有优化空间

**etcd与ZK的不同之处与优势总结如下：**

使用Raft协议取代Zab协议，实现相比ZK更简单，不过跨机房问题仍然存在

使用go语言实现，性能对比ZK提升明显，仍然存在GC问题

客户端使用restful API，客户端支持类型更多，易用性加强很多

支持过期节点与续约机制，支持批量事务和多watch设置

支持运行时机器变动

## zk vs etcd

zookeeper是用java开发的，被Apache很多项目采用。

etcd是用go开发的，主要是被Kubernetes采用。

zookeeper非常稳定，是一个著名的分布式协调系统，etcd是后起之秀，前景广阔。

因为etcd是用go写的，现在还没有很好的java客户端库，需要通过http方式调用。

而zookeeper在这方面就成熟很多，对于java之外的其他开发语言都有很好的客户端库。

具体选择zookeeper还是etcd，需要根据您的需求结合它们各自的特性进行判断，还有您所使用的开发语言。

## 设计目标

Zookeeper致力于为分布式应用提供一个高性能、高可用，并且具有严格顺序访问控制能力的分布式协调服务。

1. 高性能

Zookeeper将全量数据存储在内存中，并直接服务于客户端的所有非事务请求，尤其适用于以读为主的应用场景。

1. 高可用

Zookeeper一般以集群的方式对外提供服务，一般3~5台机器就可以组成一个可用的zookeeper集群，没台机器都会在内存中维护当前的服务器状态，并且每台机器之间都相互保持着通信。只要集群中超过一半的机器都能够正常工作，那么整个集群就能够正常对外服务。

1. 严格顺序访问

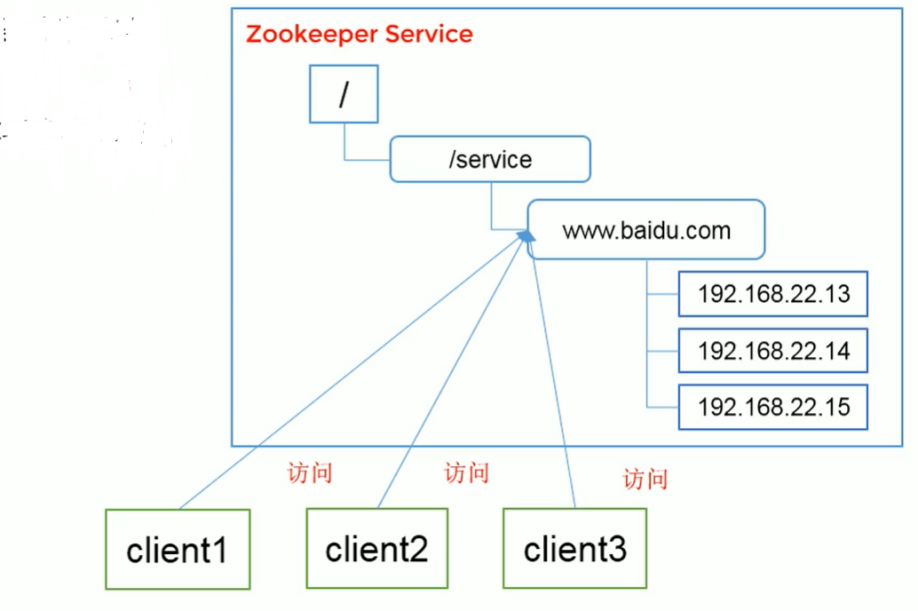
对于来自客户端的每个更新请求，zookeeper都会分配一个全局唯一的递增编号，这个编号反映了所有事务操作的先后顺序。

## 应用场景

提供服务包括：统一命名服务、统一配置管理、统一集群管理、服务器节点动态上下线、软负载均衡、分布式锁、分布式唯一ID等。

### 统一命名服务

在分布式环境下，经常需要对应用/服务进行统一命名，便于识别。例如：IP不容易记住，而域名容易记住。



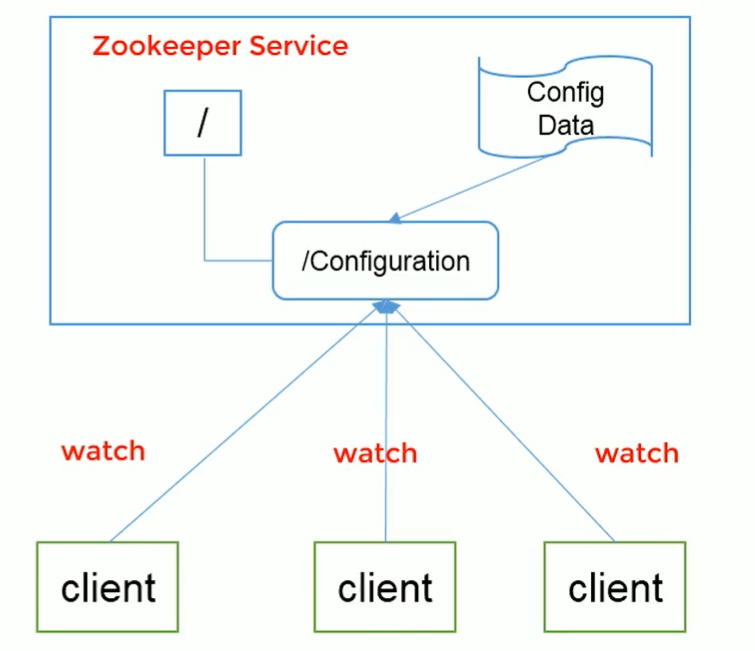
### 统一配置管理

在分布式环境下，配置文件同步非常常见：

1. 一般要求一个集群中，所有节点的配置信息是一致的，比如kafka集群；
2. 对配置文件修改后，希望能够快速同步到各个节点上；

配置管理可交由zookeeper实现：

1. 可将配置信息写入zookeeper上的一个ZNode；
2. 各个客户端服务器监听这个ZNode；
3. 一旦ZNode中的数据被修改，zookeeper将通知各个客户端服务器。

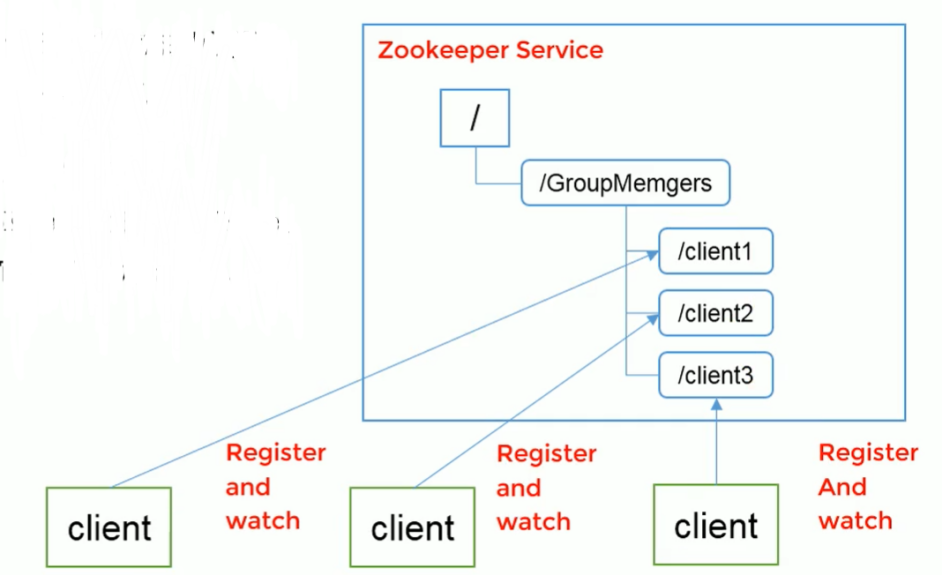


### 统一集群管理

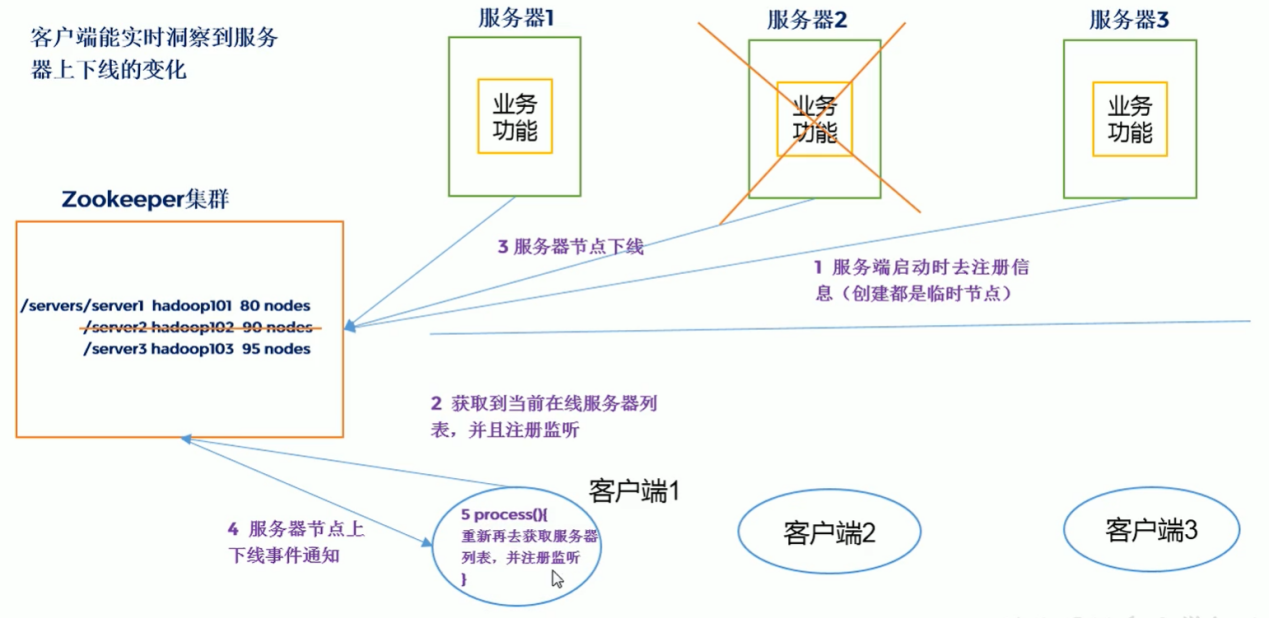
在分布式环境中，实时掌握每个节点的状态是必要的，可以根据节点实时状态做出一些调整。

Zookeeper可以实现实时监控节点状态变化：

1. 可将节点信息写入zookeeper上的一个ZNode；
2. 监听这个ZNode可获取它的实时状态变化。

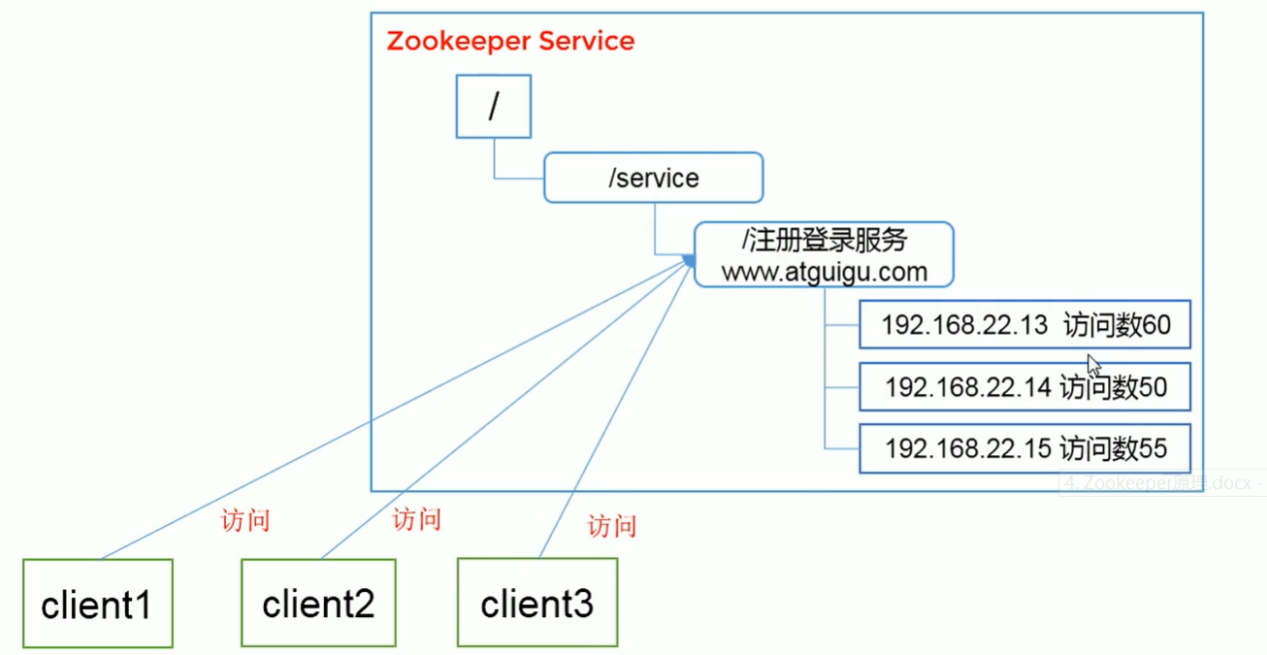


### 服务器节点动态上下线



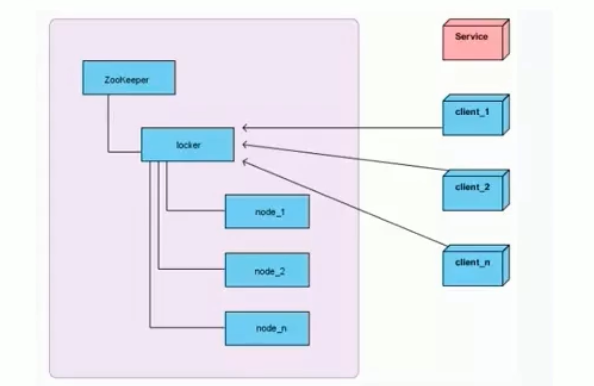
### 软负载均衡

在zookeeper中记录每台服务器的访问数，让访问数最少的服务器去处理最新的客户端请求。



### 分布式锁

一个集群是一个分布式系统，由堕胎服务器组成。为了提高并发度和可靠性，多台服务器上运行着同一服务。当多个服务在运行时就需要协调各个服务的进度，有时候需要保证某个服务在运行某个操作时，其他的服务都不能进行该操作，即对该操作进行加锁，如果当前机器挂掉后，释放锁并fail over到其他的机器继续执行该服务。



### 分布式唯一ID

在过去的单库单表型系统中，通常可以使用数据库字段自带的suto\_increment属性来自动为每条记录生成一个唯一的ID。但是分库分表后，就无法依靠数据库的auto\_increment属性来唯一标识一条记录了。此时，我们就可以用zookeeper的分布式环境下生成全局唯一ID。

做法如下：每次要生成一个新的ID时，创建一个持久顺序节点，创建操作返回的节点序号，即为新ID，然后把比自己节点小的删除即可。

# 安装配置

## 本地模式安装部署

1. 安装前准备
2. 安装JDK
3. 拷贝zookeeper安装包到Linux目录，并解压到指定目录
4. 配置修改
5. 将/opt/mudule/zookeeper-\*/conf路径下的zoo\_sample.cfg修改为zoo.cfg
6. 打开zoo.cfg，修改dataDir路径

dataDir=/opt/module/zookeeper-\*/zkData

1. 在/opt/mudule/zookeeper-\*目录下创建zkData文件夹
2. 操作zookeeper
3. 启动zookeeper：bin/zkServer.sh start
4. 查看进程是否启动：jps
5. 查看状态：bin/zkServer.sh status
6. 启动客户端：bin/zkCli.sh
7. 退出客户端：quit
8. 停止zookeeper：bin/zkServer.sh stop

## 配置参数

Zookeeper中的配置文件zoo.cfg中参数含义：

1. tickTime=2000：通信心跳数，zookeeper服务器与客户端心跳时间，单位毫秒

Zookeeper使用的基本时间，服务器之间或者客户端与服务器之间维持心跳的时间间隔，也就是每个tickTime时间就会发送一次心跳，时间单位为毫秒。

它用于心跳机制，并且设置最小的session超时时间为两倍心跳时间（session的最小超时时间是2\*tickTime）。

1. initLimit=10：LF初始通信时限

集群中的Fellower跟随者服务器与Leader领导者服务器之间初始连接时能容忍的最多心跳数（tickTime的数量），用它来限定集群中的zookeeper服务器连接到Leader的时限。

1. syncLimit=5：LF同步通信时限

集群中Leader与Fellower之间的最大响应时间单位，假如响应时间超过syncLimit\*tickTime，Leader认为Fellower死掉，从服务器列表中删除Fellower。

1. dataDir：数据文件目录+数据持久化路径

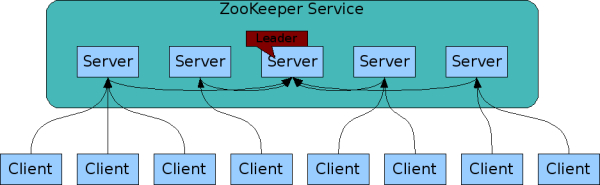
主要用于保存zookeeper中的数据。

1. clientPort=2181：客户端连接端口

监听客户端连接的端口。

# 原理

## 系统模型



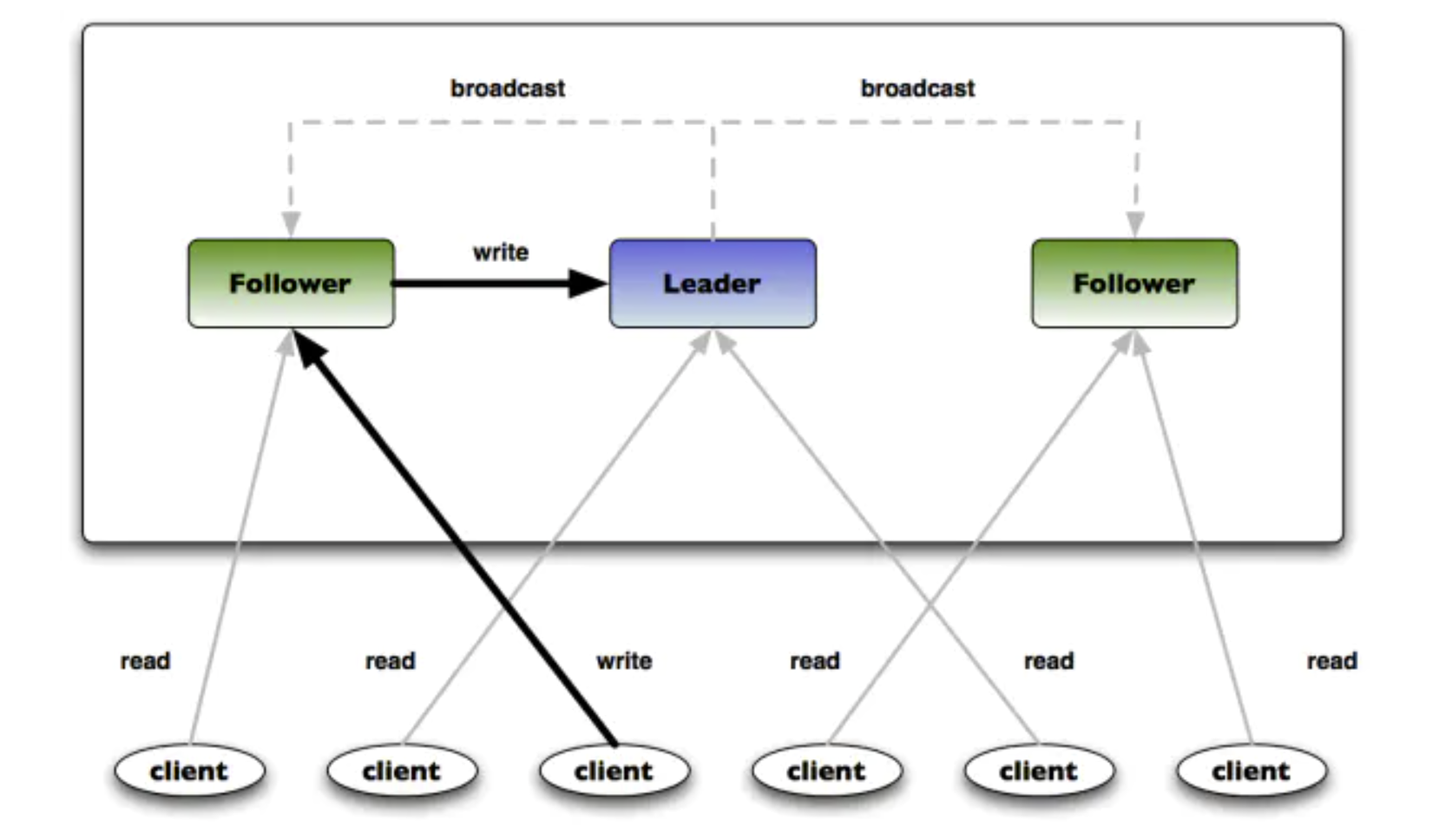
多个server构成的集群

客户端可以连接任意的server进行读写

leader角色，事务请求的唯一调度和处理者，保证事务的顺序性

follow角色，参与选举投票和事务提案投票

abserver角色，同步数据



zookeeper使用ZAB协议作为其一致性协议。zookeeper通过团队的形式工作，一组node一起工作，来提供分布式能力，这组node的数量需要是奇数。

第一个节点与其他节点沟通，选举出一个leader，获取多数票数的成为leader，这就是为什么需要奇数个node，其他节点被称为follower。

client连接 zookeeper时可以连接任何一个，client的读请求可以被任何一个节点处理，写请求只能被leader处理。所以，**添加新节点可以提高读的速度，但不会提高写的速度**。

对于CAP模型，zookeeper保障的是CP。

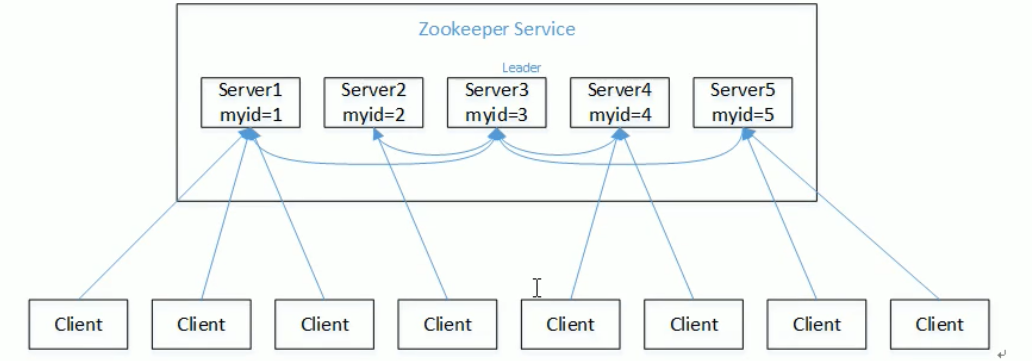
## 选举机制

### 概述

1. **半数机制**：集群中半数以上机器存活，集群可用，所以zookeeper适合安装**奇数台服务器**。
2. Zookeeper虽然在配置文件中并没有制定master和slave，但是zookeeper工作时，是有一个节点为Leader，其他则为Fellower，Leader是通过内部的选举机制临时产生的。

举例：

假设有五台服务器组成的zookeeper集群，它们的id从1~5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依次启动，观察会发生什么：



1. 服务器1启动，此时只有它一台服务器启动了，它发出去的报文没有任何响应，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，互相交换自己的选举结果，由于两者都没有历史数据，所以id值较大的服务器2胜出，但是由于没有达到超过半数以上的投票。
3. 服务器3启动，同样投票给自己，id值较大的服务器3胜出，此时达到半数以上的投票，产生Leader。
4. 服务器4启动，但是此时已经产生Leader，作为Fellower。
5. 服务器5启动，同样为Fellower。

### Leader选举

**State:**

Looking

Following

Leading

Observing

**算法：**

1、每个Server发出一个投票：投自己

2、接受来自各个服务器的投票

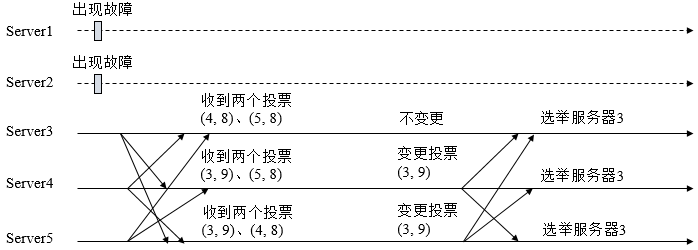
3、处理投票：针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK输了要变更投票，PK规则如下

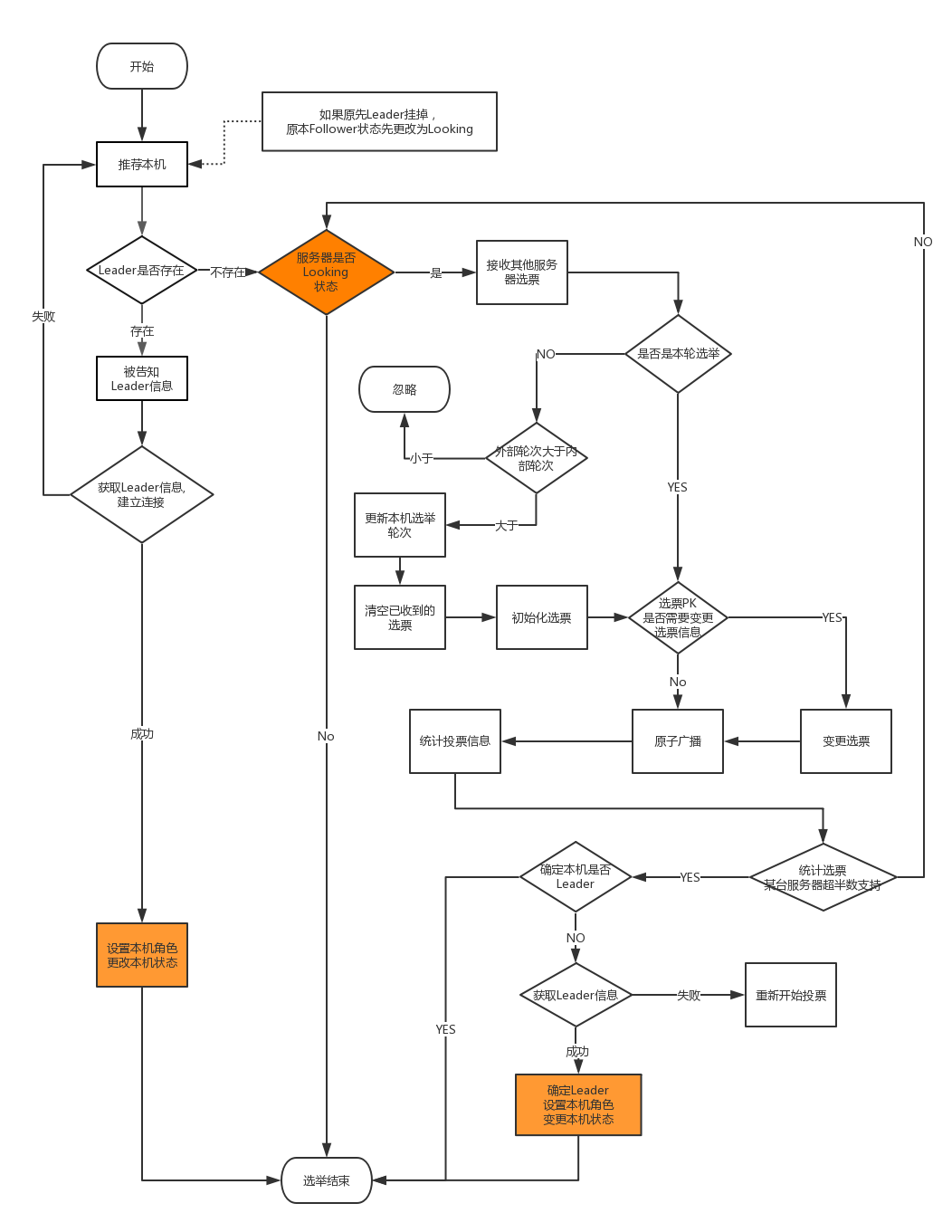
优先检查ZXID。ZXID比较大的服务器优先作为Leader。

如果ZXID相同，那么就比较myid。myid较大的服务器作为Leader服务器。

4、统计投票：每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息。

5、改变服务器状态。

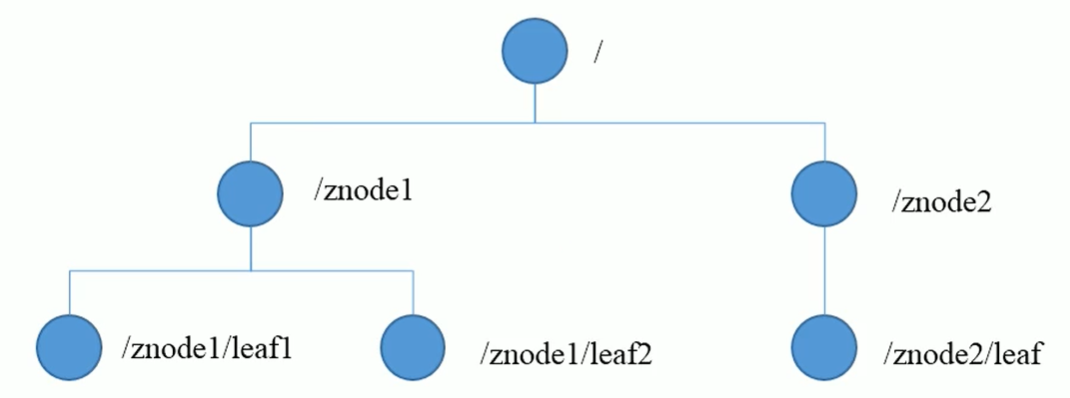




## 数据模型

### 数据结构ZNode

Zookeeper数据模型的结构与Unix文件系统很相似，整体上可以看作是一棵树，每个节点称作一个ZNode。**每一个ZNode默认能够存储1MB的数据，每个ZNode都可以通过其路径唯一标识**。



datatree：层次化结构组织，类似unix文件系统

znode：数据单元，可以挂载子节点

zxid：64位，事务性操作的唯一标识；

**节点类型：**

持久节点：持久化

持久顺序节点：持久化，维护第一级子节点创建顺序；

临时节点：生命周期与会话绑定，只能是叶子节点；

临时顺序节点：会话绑定，保持顺序

从图中我们可以看出ZooKeeper的数据模型，在结构上和标准文件系统的非常相似，都是采用这种树形层次结构，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：

#### 引用方式

Zonde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。

#### Znode结构

ZooKeeper命名空间中的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。图中的每个节点称为一个Znode。每个Znode由3部分组成：

1. 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
3. children：该Znode下的子节点

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，但常规使用中应该远小于此值。

#### 数据访问

ZooKeeper中的每个节点存储的数据要被原子性的操作。也就是说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。

#### 节点类型

ZooKeeper中的节点有两种，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。

1、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(Session)结束，临时节点将被自动删除，当然可以也可以手动删除。虽然每个临时的Znode都会绑定到一个客户端会话，但**他们对所有的客户端还是可见的**。另外，**ZooKeeper的临时节点不允许拥有子节点**。

2、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。

具体又可以划分：

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

1. PERSISTENT\_SEQUENTIAL：持久化顺序编号目录节点

客户端与zookee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

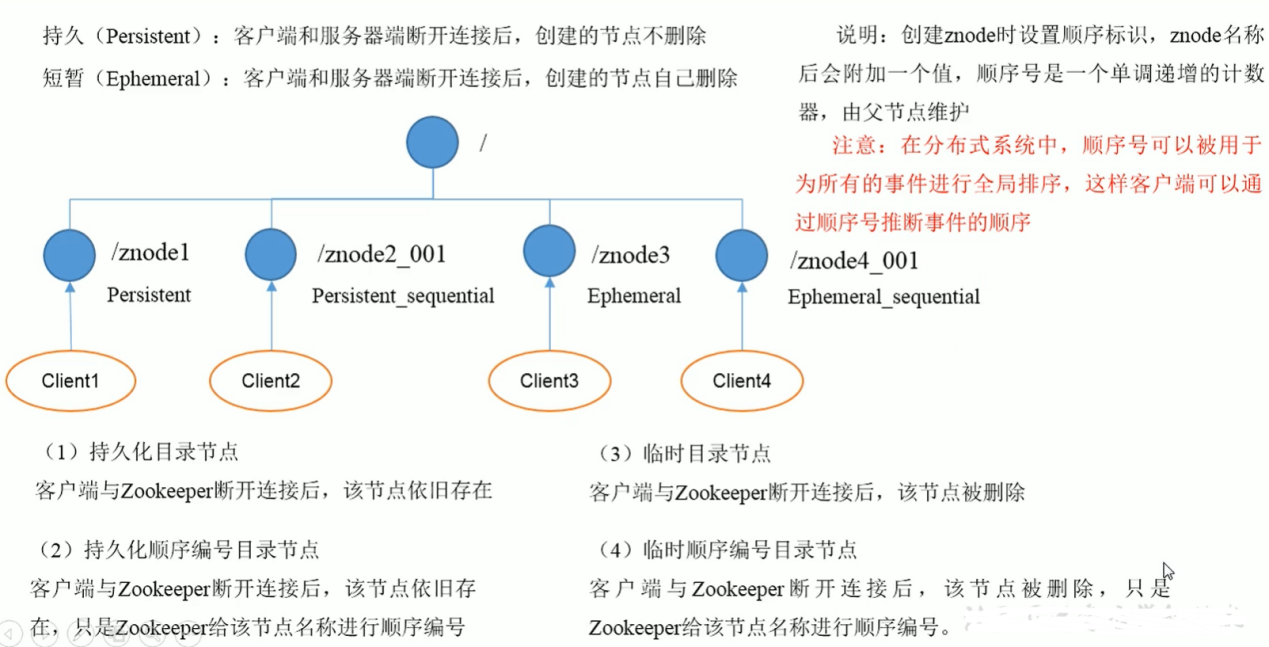
1. EPHEMERAL：临时目录节点

客户端与zookeeper断开连接后，该节点被删除

1. 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

注：对于持久节点和临时节点，同一个znode下，节点的名称是唯一的。



#### 顺序节点

当创建Znode的时候，用户可以请求在ZooKeeper的路径结尾添加一个递增的计数。这个计数对于此节点的父节点来说是唯一的，它的格式为"%10d"(10位数字，没有数值的数位用0补充，例如"0000000001")。当计数值大于232-1时，计数器将溢出。

#### 观察

客户端可以在节点上设置watch，我们称之为监视器。当节点状态发生改变时(Znode的增、删、改)将会触发watch所对应的操作。当watch被触发时，ZooKeeper将会向客户端**发送且仅发送一条通知**，因为watch只能被触发一次，这样可以减少网络流量。

### 时间

ZooKeeper有多种记录时间的形式，其中包含以下几个主要属性：

1、Zxid

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。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：是节点的创建时间所对应的Zxid格式时间戳。

mZxid：是节点的修改时间所对应的Zxid格式时间戳。

实现中Zxid是一个64为的数字，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。低32位是个递增计数。

2、版本号

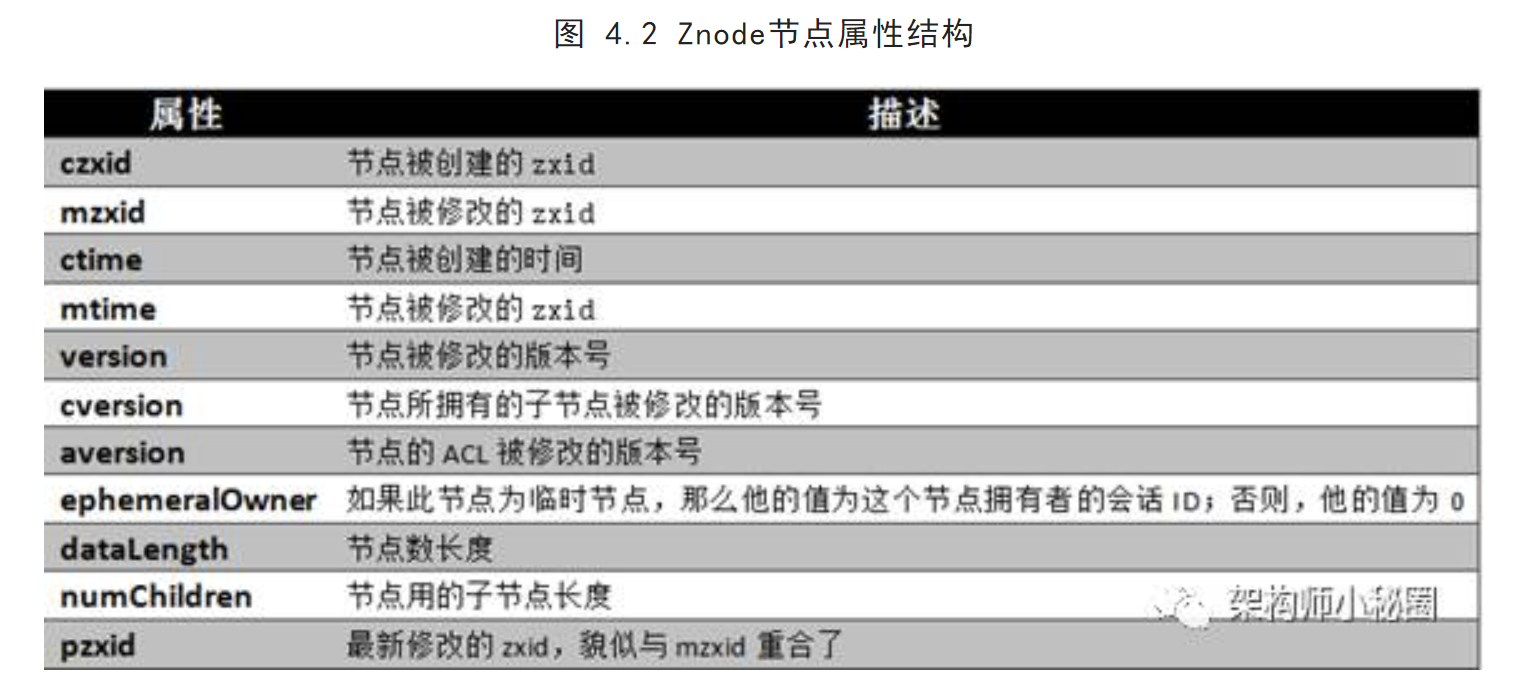
对节点的每一个操作都将致使这个节点的版本号增加。每个节点维护着三个版本号，他们分别为：

version：节点数据版本号

cversion：子节点版本号

aversion：节点所拥有的ACL版本号

### 节点属性/Stat结构体



1. czxid - 引起这个znode创建的zxid，创建节点的事务的zxid
2. ctime - znode被创建的毫秒数(从1970年开始)
3. mzxid - znode最后更新的zxid
4. mtime - znode最后修改的毫秒数(从1970年开始)
5. pZxid-znode - 最后更新的子节点zxid
6. cversion - znode子节点变化号，znode子节点修改次数
7. dataversion - znode数据变化号
8. aclVersion - znode访问控制列表的变化号
9. ephemeralOwner - 如果是临时节点，这个是znode拥有者的session id。如果不是临时节点则是0
10. dataLength - znode的数据长度
11. numChildren - znode子节点数量

## 监听器/watcher

### 概述

ZooKeeper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，并且ZooKeeper为watch机制提供了有序的一致性保证。理论上，客户端接收watch事件的时间要快于其看到watch对象状态变化的时间。

### 类型

ZooKeeper所管理的watch可以分为两类：

1. 数据watch(data watches)：getData和exists负责设置数据watch
2. 孩子watch(child watches)：getChildren负责设置孩子watch

我们可以通过操作返回的数据来设置不同的watch：

1. getData和exists：返回关于节点的数据信息
2. getChildren：返回孩子列表

因此：

1. 一个成功的setData操作将触发Znode的数据watch
2. 一个成功的create操作将触发Znode的数据watch以及孩子watch
3. 一个成功的delete操作将触发Znode的数据watch以及孩子watch

### 注册与触发

1、exists操作上的watch，在被监视的Znode创建、删除或数据更新时被触发。

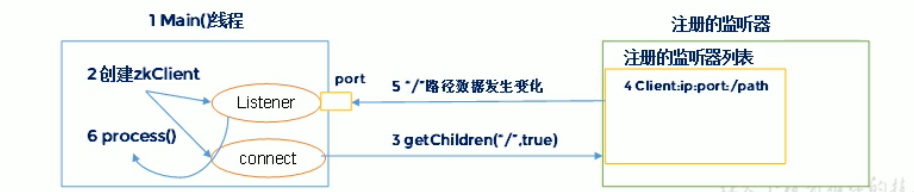
2、getData操作上的watch，在被监视的Znode删除或数据更新时被触发。在被创建时不能被触发，因为只有Znode一定存在，getData操作才会成功。

3、getChildren操作上的watch，在被监视的Znode的子节点创建或删除，或是这个Znode自身被删除时被触发。可以通过查看watch事件类型来区分是Znode，还是他的子节点被删除：NodeDelete表示Znode被删除，NodeDeletedChanged表示子节点被删除。

Watch由客户端所连接的ZooKeeper服务器在本地维护，因此watch可以非常容易地设置、管理和分派。当客户端连接到一个新的服务器时，任何的会话事件都将可能触发watch。另外，当从服务器断开连接的时候，watch将不会被接收。但是，当一个客户端重新建立连接的时候，任何先前 注册过的watch都会被重新注册。

### 原理

**原理：**



1. **监听原理：**
2. 首先要有一个main线程；
3. 在main线程中创建zookeeper客户端，这时就会创建两个线程，**一个负责网络连接通信（connect），一个负责监听（listener）**；
4. 通过connect线程将注册的监听事件发送给zookeeper；
5. 在zookeeper的注册监听器列表中将注册的监听事件添加到列表中；
6. Zookeeper监听到有效数据或路径变化，就会将这个消息发送给listener线程；
7. listener线程内部调用了process()方法。
8. **常见监听：**
9. 监听节点数据的变化：

get path [watch]

1. 监听子节点增减的变化：

ls path [watch]

**特性：**



### 会话状态迁移/一致性视图

1、注册只能确保一次消费

2、客户端串行执行

3、通知消息只包含通知状态、事件类型、节点路径。

NodeCreate

NodeDeleted

NodeDataChanged

NodeChildrenChanged

watch机制，带来的会话概念。

如果zk节点挂了，客户端要重连：客户端API。

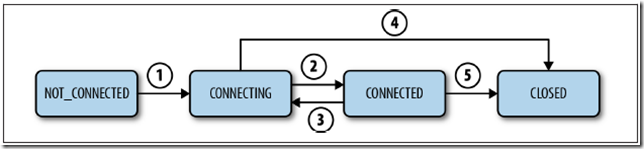
SessionID：唯一标识一个会话，每次客户端创建新会话时，zookeeper都会为其分配全局唯一的sessionID；

TimeOut：session超时时间；客户端初始化zookeeper实例时由客户端和服务端协商确定；

TickTime：下次会话超时的时间点；

IsClosing：用于标记一个会话是否关闭；

会话状态变迁：



一致性视图的问题

1、新客户端怎么保证看到一致性视图？

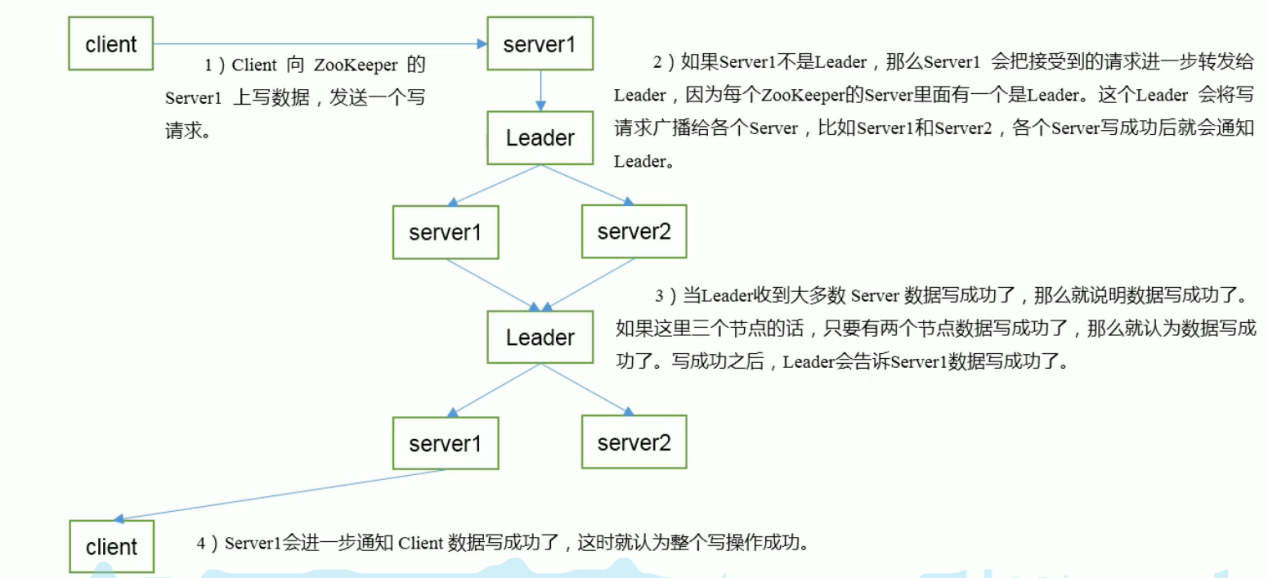
会随机选一个zk server建立会话，会话的创建过程属于事务型操作，创建成功后，会将sessionID返回给客户端，客户端缓存在内存中；

2、客户端内部重连怎么保证一致性视图？

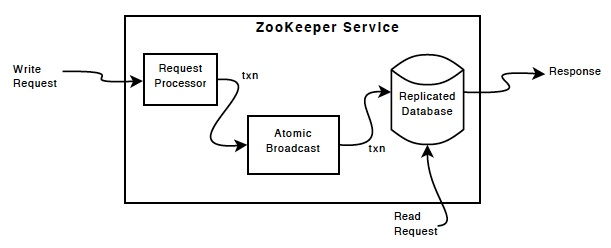
API内部会重新自动选择一个zk server，携带之前sessionID和最新zxid发起会话重连；

如果连接到的zk server最新zxid比客户端zxid小，zk server会拒绝客户端连接，客户端会重新选择一个server进行连接。

## 写数据流程



## 请求处理



request processor:对请求进行检验处理。

atomic broadcast:原子广播，保证消息的一致性。

repliacated database:副本数据库，维护整个数据视图。

## 数据与存储

**1、内存数据**

key-value

**2、事务日志**

强制刷盘（默认）

预分配空间

文件后缀为第一个zxid

1. **数据快照**

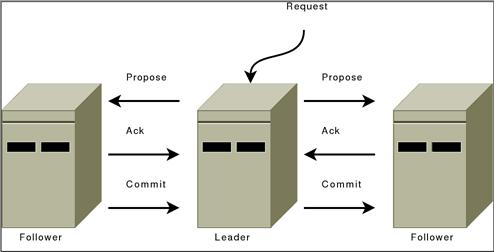
内存全量

异步刷盘

snapcount/2 + rand

文件后缀为最后一个zxid

## 原子广播



Leader发起Propose，收到半数节点以上（Quorums）的ACK，再发送commit

## follow节点故障

集群模式下半数以下的follow节点故障没有影响。

**数据恢复：**快照+事务日志

**数据追平:**

DIFF：minCommittedLog<=peerLastZxid<=maxCommittedLog;

TRUNK：peerLastZxid>maxCommittedLog;

SNAP：peerLastZxid<minCommittedLog;

# 部署方式

# ACL权限控制

## 概述

Zookeeper类似文件系统，client可以创建节点、更新节点、删除节点，那么如何做到节点的权限控制呢？Zookeeper的access control list访问控制列表可以做到这一点。

ACL权限控制，使用scheme:id:permission来标识，主要涵盖三个方面：

权限模式（scheme）：授权的策略

授权对象（id）：授权的对象

权限（permission）：授予的权限

其特征如下：

1. zookeeper的权限控制是基于每个znode节点的，需要对每个节点设置权限
2. 每个znode支持设置多种权限控制方案和多个权限
3. 子节点不会继承父节点的权限，客户端无权访问某节点，但可能可以访问它的子节点

## 权限模式



### world授权模式

命令：setAcl <path> world:anyone:<acl>

### ip授权模式

命令：setAcl <path> ip:<ip>:<acl>

### auth授权模式

命令：

addauth digest <user>:<password> #添加认证用户

setAcl <path> auth:<user>:<acl>

### digest授权模式

命令：setAcl <path> digest:<user>:<password>:<acl>

### 多种授权模式

### ACL超级管理员

## 授权对象

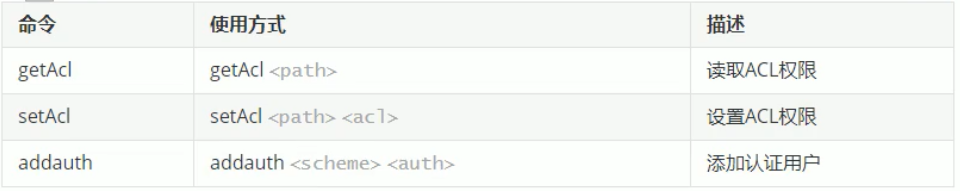
给谁授予权限，授权对象ID是指，权限赋予的实体，例如：IP地址或用户。

## 授予权限

create、delete、read、write、admin也就是增、删、改、查、管理权限，这5种权限简写为cdrwa，注意，这5种权限中，delete是指对子节点的删除权限，其他4种权限指对自身节点的操作权限。



## 相关命令



# ZAB协议

Zookeeper使用的是一种被称为Zookeeper Atomic Broadcast（ZAB，Zookeeper原子广播协议）作为其数据一致性别的核心算法。ZAB协议的核心是定义了对于那些会改变Zookeeper服务器数据状态的事务请求的处理方式，即：

所有事务请求必须由一个全局唯一的服务器来协调处理（即Leader服务器），其他服务器则成为Follower服务器。Leader负责将一个客户端事务请求转换为一个事务proposal，并将该proposal分发给集群中的所有的Follower服务器。之后Leader需要等待所有Follower服务器的反馈，一旦超过半数的Follower服务器进行了正确的反馈后，那么Leader就认为再次向所有的Follower服务器分发commit消息，要求其将前一个proposal进行提交。

特性：保证在leader上提交的事务最终被所有的服务器提交，保证丢弃没有经过半数检验的事务。

组成Zookeeper集群的每台机器都会在内存中维护当前的服务器状态，并且每台机器间都互相保持通信，只要集群中超过存在一半的机器能够正常工作，那么整个集群就能正常对外提供服务。

# 常用指令

## 启动ZK服务

命令：./zkServer.sh start

## 查看ZK服务状态

命令：./zkServer.sh status

## 停止ZK服务

命令：./zkServer.sh stop

## 重启ZK服务

命令：./zkServer.sh restart

## 连接ZK服务

命令：./zkCli.sh start -server ip:port

## 新增节点/目录

通过客户端对象的create方法创建节点。

## 修改节点/目录

## 删除节点/目录

命令：

delete /zk #此命令不可以删除有子节点的节点

rmr /zk #该命令可以删除有子节点的节点

## 查看节点/目录和状态

命令：ls / #使用ls查看当前Zookeeper中所包含的内容

## 返回节点链表

## 获取文件内容

命令：get /zk #确认znode是否包含我们所创建的字符串

## 修改文件内容

命令：set /zk “ssss” #对zookeeper所关联的字符串进行设置

## 退出客户端

命令：quit

## 帮助

命令：help

# 编程接口

# 客户端

## 开源客户端curator

## 图形化客户端ZooInspector

# 应用

Zk在TDSQL中应用：

1. 配置维护和通知：watch配置节点
2. 任务发布和资源发现：在父节点上watch子节点变化
3. 进程主备：**临时顺序节点，**在父节点上watch子节点变化

/xxxx/ip1\_00001

/xxxx/ip2\_00002

1. **分布式锁：屏障**