# ======知识点======

## 什么是Zookeeper？

Zookeeper是一个高效的分布式协调服务，它暴露了一些公共服务，比如命名/配置/管理/同步控制/群组服务等。我们可以使用ZK来实现比如达成共识/集群管理/leader选举等。

**Zookeeper是一个高可用的分布式管理与协调框架，基于ZAB算法（原子消息广播协议）的实现**。该框架能够很好地保证分布式环境中数据的一致性。也正是基于这样的特性，使得Zookeeper成为了解决分布式一致性问题的利器。

### 特性

1. **顺序一致性：**

从一个客户端发起的事物请求，最终将会严格地按照其发起的顺序被应用到Zookeeper中去。

1. **原子性：**

所有事物请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群所有的机器都成功应用了某一事物，要么就没有应用，一定不会出现部分机器应用了该事物，而另一部分没有应用的情况。

1. **单一视图：**

无论客户端连接的是哪一个zookeeper服务器，其看到的服务器端数据模型都是一致的。

1. **可靠性：**

一旦服务器成功地应用了一个事物，并完成了对客户端的响应，那么该事物所引起的服务器端状态将会被一致保留下来。除非有另一个事物对其更改。

1. **实时性：**

通常所说的实时性就是指一旦事物被成功应用，那么客户端就能立刻从服务器上获取变更后的新数据，zookeeper仅仅能保证在一段时间内，客户端最终一定能从服务器端读取最新的数据状态。

|  |
| --- |
| 举例说明上述特性：  3个节点通过paxos算法，ZAB协议选举出1个leader，2个follower，假如leader挂了，只要半数以上节点是好的就能对外提供服务。  IMG_256  顺序一致性、原子性：三个节点构成集群，相当于一个整体，如果有一个client端发起一个请求把某个节点的数据1改为2，zk会把通过ZAB把修改的消息发给其他节点，用的同步算法就是paxos算法。同步数据期间不允许其他client修改数据。  可靠性：事物成功后会给client反馈，一问一答的形式 |

### Zookeeper设计目标

**目标1：简单的数据结构。**

zookeeper就是以简单的树形结构来进行相互协调的（也叫树形命名空间）

**目标2：可以构建集群。**

一般zookeeper集群通常由一组机器构成，一般3~5台机器就可以组成一个zookeeper集群了。只要集群中超过半数以上的机器能够正常工作，那么整个集群就能够正常对外提供服务。

**目标3：顺序访问。**

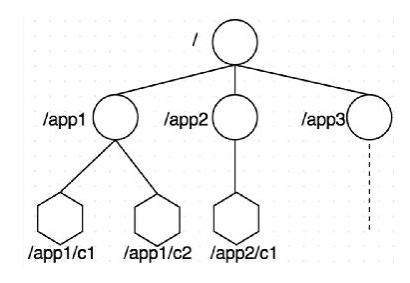
对于来自每一个客户端的每一个请求，zookeeper都会分配一个全局唯一的递增编号，这个编号反应了所有事物操作的先后顺序，应用程序可以使用zookeeper的这个特性来实现更高层次的同步。

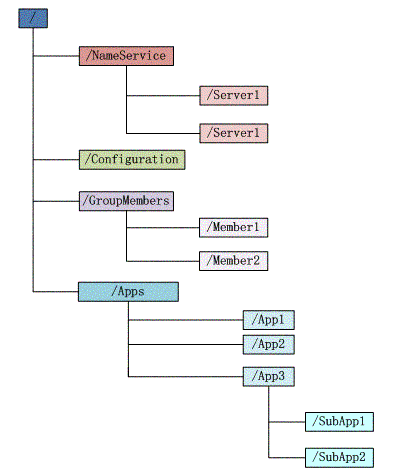
**目标4：高性能。**

由于zookeeper将全量数据存储在**内存中**，并直接服务于所有的非事物请求，因此尤其在读操作为主的场景下性能非常突出。

### Zookeeper数据结构

Zookeeper会维护一个具有层次关系的数据结构，它非常类似于一个标准的文件系统





### Zookeeper数据模型

1，每个子目录项如 NameService 都被称作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

2，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

3，znode是有版本的，每个znode中存储的数据可以有多个版本，也就是一个访问路径可以存储多份数据

4，znode可以是临时节点，一旦创建这个znode的客户端与服务器失去联系，这个znode也将自动删除，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，这个session失效，znode也就删除了。

5，znode 的目录名可以自动编号，如App1 已经存在，再创建的话，将会自动命名为App2

6，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

### Zookeeper组成

ZK Server根据其身份特性分为三种，Leader ，Follower，Observer（就是一个client端），其中Follower和Observer又统称为Learner（学习者）。

　　Leader：负责客户端的writer 类型请求

　　Follower：负责客户端的reader 类型请求，参与leader选举等。

Observer：特殊的“Follower“，其可以接受客户端reader 请求，但不参与选举。（扩容系统支撑能力，提高了读取速度。因为它不接受任何同步的写入请求，只负责与leader同步数据）

### Zookeeper典型应用场景

* 分布式锁
* 服务注册和发现

利用 Znode和 Watcher，可以实现分布式服务的注册和发现。最著名的应用就是阿里的分布式 RPC框架 Dubbo。

* 共享配置和状态信息

Redis的分布式解决方案 Codis（豌豆荚），就利用了 Zookeeper来存放数据路由表和 codis-proxy 节点的元信息。同时 codis-config 发起的命令都会通过 ZooKeeper 同步到各个存活的 codis-proxy。

**应用场景说明**

**一：配置管理**

配置的管理在分布式应用环境中很常见，比如我们在平常的应用系统中，经常会遇到这样的需求，如机器的配置列表、运行时的开关配置、数据库配置信息等。这些全局配置信息通常具有以下3个特性：

1，数据量比较小

2，数据内容在运行时动态发生变化

3，集群中各个节点共享信息，配置一致

**二：集群管理**

Zookeeper 不仅能够帮你维护当前的集群中机器的服务状态，而且能够帮你选出一个“总管” ，让这个总管来管理集群，这就是 Zookeeper 的另一个功能 Leader，并实现集群容错功能。

1，希望知道当前集群中究竟有多少机器工作

2，对集群中每天集群的运行状态进行数据收集

3，对集群中每台集群进行上下线操作

**三：发布与订阅，**

Zookeeper 是一个典型的发布/订阅模式的分布式数控管理与协调框架，开发人员可以使用它来进行分布式数据的发布与订阅

**四：数据库切换，**

比如我们初始化Zookeeper 的时候读取节点上的数据库配置文件，当配置一旦发生变更时，zookeeper 就能帮我们把变更的通知发送到各个客户端，每个客户端在接收到这个变更后，就可以从新进行最新数据的获取。（数据库properties配置在zk，挂了的话切换到另一个）

五：分布式日志收集，我们可以做一个日志系统收集集群中所有的日志信息，进行统一管理。

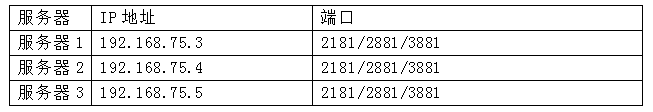
六：Zookeeper 的特性就是在分布式场景下高可用，但是用原生的API 实现的分布式功能非常困难，可以使用第三方客户端的完美实现，比如Curator框架，它是 Apache 的顶级项目

### Zookeeper开源框架的应用

如 Hadoop、Storm、消息中间件、RPC服务框架、数据库增量订阅与消费组件（如mysql binlog）、分布式数据库同步系统，淘宝的Otter。

## ZK集群安装

**1，准备三台机器做集群，都已安装JDK**



1. **下载**

下载zookeeper-3.4.10.tar.gz 到/data/program/software/目录，

wget <http://mirror.bit.edu.cn/apache/zookeeper/zookeeper-3.4.9/zookeeper-3.4.9.tar.gz>

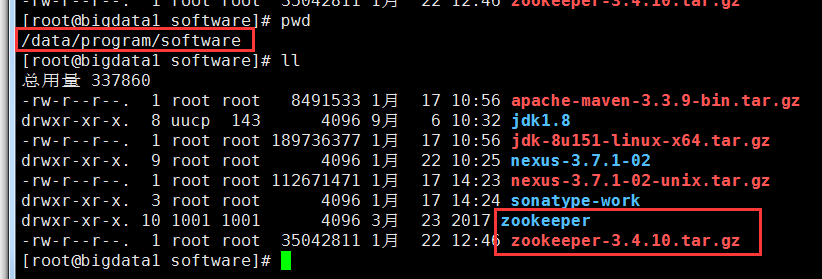
**3，解压zookeeper安装包，并对节点重命名**

#tar -zxvf zookeeper-3.4.10.tar.gz

服务器1：#mv zookeeper-3.4.10 zookeeper

服务器2：#mv zookeeper-3.4.10 zookeeper

服务器3：#mv zookeeper-3.4.10 zookeeper

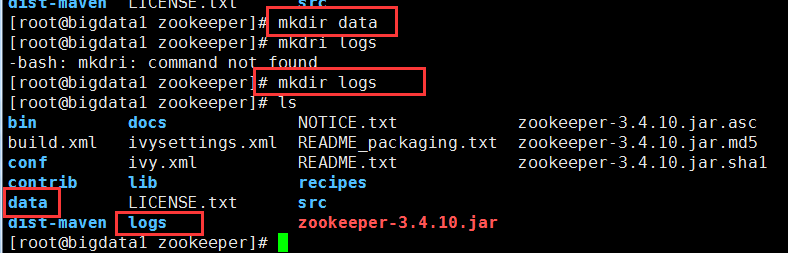


1. **在zookeeper的各个节点下 创建数据和日志目录**

#cd zookeeper

#mkdir data

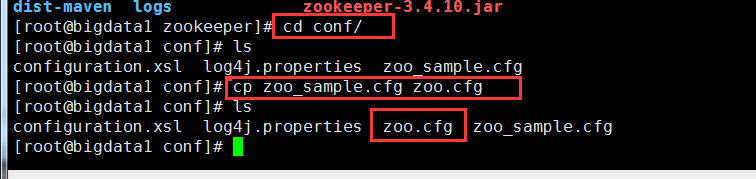
#mkdir logs



1. **配置zoo.cfg**

将zookeeper/conf目录下的zoo\_sample.cfg文件拷贝一份，命名为zoo.cfg

#cp zoo\_sample.cfg   zoo.cfg



vim 修改zoo.cfg 配置文件，修改如下几处，保存

　　　　　　clientPort=2181

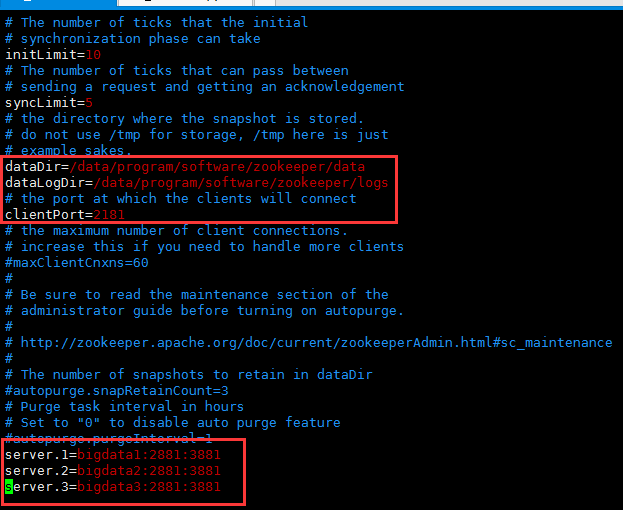
　　　　　　dataDir=/data/program/software/zookeeper/data

　　　　　　dataLogDir=/data/program/software/zookeeper/logs

　　　　　　server.1=bigdata1:2881:3881

　　　　　　server.2=bigdata2:2881:3881

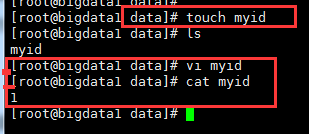
　　　　　　server.3=bigdata3:2881:3881



|  |
| --- |
| zoo.cfg说明：  [root@bigdata1 conf]# cat zoo.cfg  tickTime=2000  initLimit=10  syncLimit=5  dataDir=/data/program/software/zookeeper/data  dataLogDir=/data/program/software/zookeeper/logs  clientPort=2181  server.1=bigdata1:2881:3881  server.2=bigdata2:2881:3881  server.3=bigdata3:2881:3881  参数说明：  **tickTime=2000** 　　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=10** 　　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=5** 　　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=/data/program/software/zookeeper/data** 　　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=2181** 　　clientPort这个端口就是客户端（应用程序）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** 　　server.1=bigdata1:2881:3881 　　server.2=bigdata2:2881:3881 　　server.3=bigdata3:2881:3881  　　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（或者是与IP地址做了映射的主机名）； 　　C此端口表示，用来集群成员的信息交换，表示这个服务器与集群中的Leader服务器交换信息的端口； 　**D是在leader挂掉时专门用来进行选举leader所用的端口。** 　　注意：如果是伪集群的配置方式，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。 |

1. **创建myid文件**

在 dataDir=/data/program/software/zookeeper/data 下创建myid文件



编辑myid文件，并在对应的IP的机器上输入对应的编号。如在1上，myid文件内容就是1, 2上就是2， 3上就是3：   
　　bigdata1    #vi /myid## 值为1  
　　bigdata2    #vi /myid## 值为2  
　　bigdata3    #vi /myid## 值为3

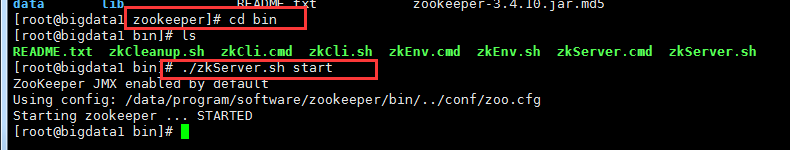
bigdata1 192.168.75.3 的配置就配置完了，bigdata2 、bigdata3也照着配置。

7**启动测试zookeeper**

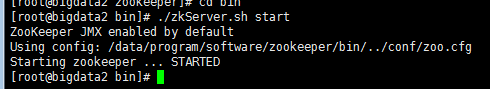
(1)进入/bin目录下执行：

　　　# ./zkServer.sh  start

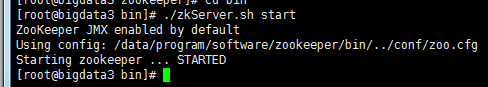
　　　bigdata1:



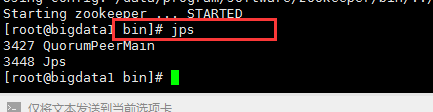
　bggdata2:



bigdata3:



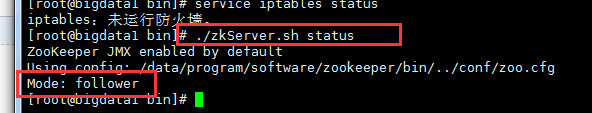
输入jps命令查看进程：



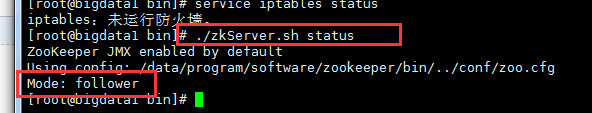
其中，QuorumPeerMain是zookeeper进程，说明启动正常

查看状态：（需关闭防火墙）

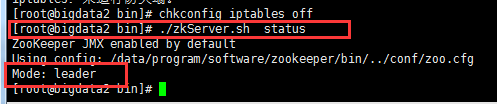
# ./zkServer.sh status



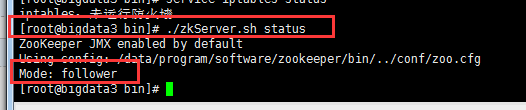
　bigdata1: 是一个follower



bigdata2:

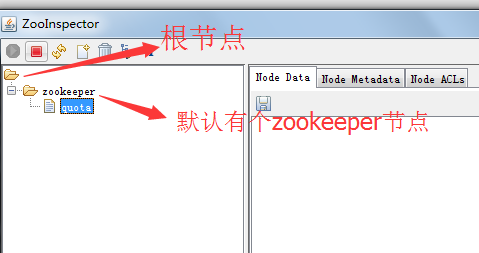


bigdata3:

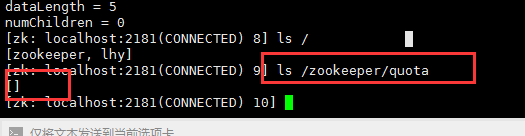


## ZK常用命令

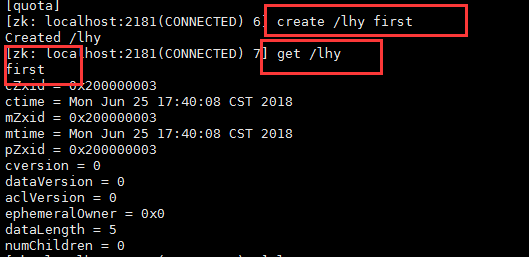
ls get set create delete



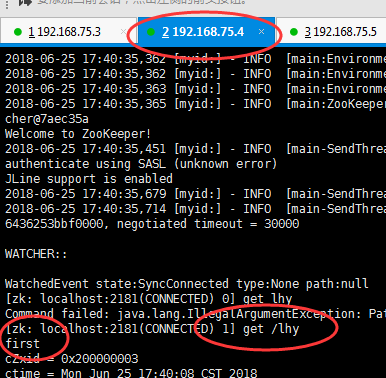
**查找 ： ls  /      ls /zookeeper**



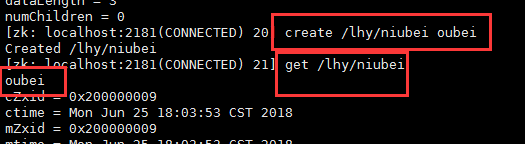
**创建并赋值：create  /lhy first**

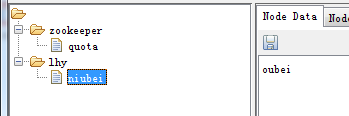


**创建是在192.168.75.3节点上创建的，其他节点也会有数据**

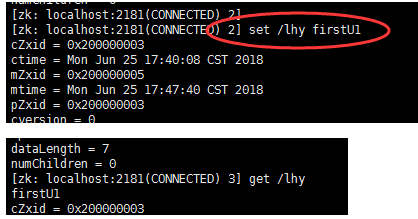


创建子节点



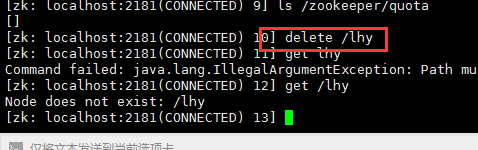


**设值新值覆盖旧值：set  /lhy second**



**递归删除 rmr  /path**

**删除指定的某个节点 delete  /path/child**



## Zookeeper的java客户端都有哪些？

java客户端：zk自带的zkclient及Apache开源的Curator。

## 集群特点

一个领导者(Leader)，多个跟随者(Follower)组成的集群

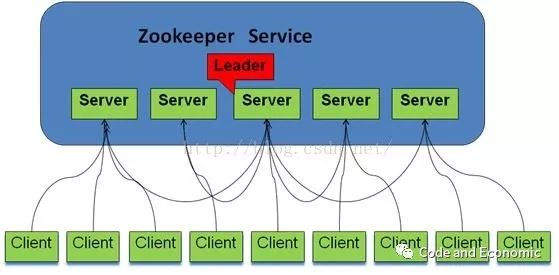
集群中只要有半数以上节点存活，Zookeeper集群就能正常服务

全局数据一致性，每个server保存一份相同的副本，client无论链接哪个server，得到的数据都是一致的

更新请求顺序执行，来自同一个client的请求按顺序执行

数据更新**原子性**，一次数据更新要么成功要么失败

实时性，在一定时间范围内，client能读到最新数据



## 谈谈什么是Zab协议

Zookeeper Atomic Broadcast（Zookeeper 原子广播协议）。

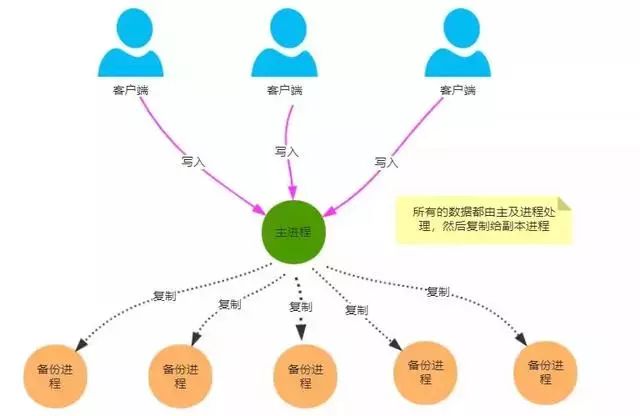
### 定义：

ZAB 协议是为 Zookeeper 专门设计的一种支持 **崩溃恢复** 和 **原子广播** 的协议。

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 数据一致性，

### Zookeeper 如何处理集群中的数据：

所有客户端写入数据都是写入到 主进程（称为 Leader）中，然后，由 Leader 复制到备份进程（称为 Follower）中。从而保证数据一致性



**复制过程：**

ZAB 只需要 Follower 有一半以上返回 Ack 信息就可以执行提交，大大减小了同步阻塞。也提高了可用性。

**消息广播 和 崩溃恢复**

整个 Zookeeper 就是在这两个模式之间切换。

当 **Leader 服务正常使用**，就**进入消息广播**模式，

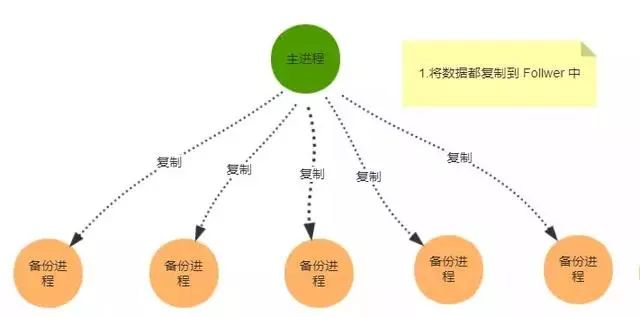
当 **Leader 不可用时**，则进入**崩溃恢复**模式

### 消息广播

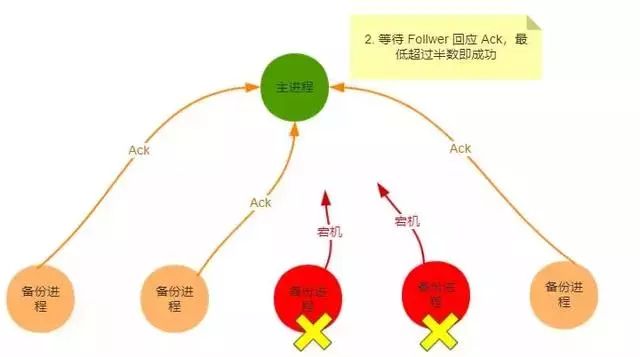
使用的是一个原子广播协议，对于客户端发送的**写请求**，全部由 **Leader 接收**，Leader 将请求**封装成**一个**事务** Proposal，将其**发送给**所有 **Follwer** ，然后，**根据**所有 Follwer 的**反馈**，如果**超过半数成功响应**，则**执行 commit** 操作（先提交自己，再发送 commit 给所有 Follwer）。

****整个广播流程分为 3 步****

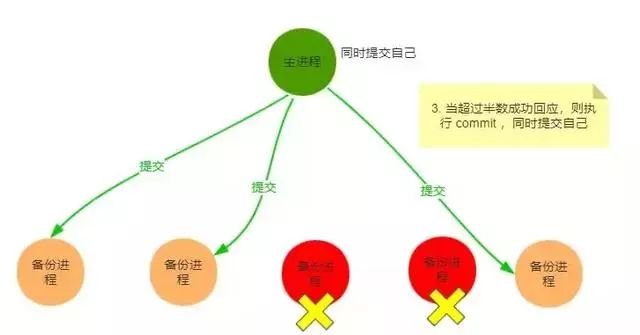
1 将数据都复制到 Follwer 中



2 等待 Follwer 回应 Ack，最低超过半数即成功



3 当超过半数成功回应，则执行 commit ，同时提交自己



这三个步骤，就能够**保持集群之间数据的一致性**。实际上，在 Leader 和 Follwer 之间还有一个消息队列，用来解耦他们之间的耦合，避免同步，实现异步解耦。

#### 一些细节

1、Leader 在收到客户端请求之后，会将这个请求封装成一个事务，并给这个事务分配一个**全局递增的唯一 ID**，称为事务ID（ZXID），ZAB 协议需要保证事务的顺序，因此必须将每一个事务按照 ZXID 进行先后排序然后处理。

2、在 Leader 和 Follwer 之间还有一个消息队列，用来解耦他们之间的耦合，解除同步阻塞。

3、zookeeper集群中为保证任何所有进程能够有序的顺序执行，只能是 Leader 服务器接受写请求，即使是 Follower 服务器接受到客户端的请求，也会转发到 Leader 服务器进行处理。

1. 实际上，这是一种简化版本的 2PC，不能解决单点问题。等会我们会讲述 ZAB 如何解决单点问题（即 Leader 崩溃问题）。

### 崩溃恢复

消息广播过程中，Leader 崩溃怎么办？还能保证数据一致吗？如果 Leader 先本地提交了，然后 commit 请求没有发送出去，怎么办？

当 Leader 崩溃，即进入皮【崩溃恢复模式】（崩溃即：Leader 失去与过半 Follwer 的联系）。下面来详细讲述

假设1：Leader 在复制数据给所有 Follwer 之后崩溃，怎么办？

假设2：Leader 在收到 Ack 并提交了自己，同时发送了部分 commit 出去之后崩溃怎么办？

****针对这些问题，ZAB 定义了 2 个原则：****

1、已经在 Leader 提交的事务最终会被所有服务器提交。

2、丢弃那些只在 Leader 提出/复制，但没有提交的事务。

所以，ZAB 设计了下面这样一个选举算法：能够确保提交已经被 Leader 提交的事务，同时丢弃已经被跳过的事务。

针对这个要求，如果让 Leader 选举算法能够保证新选举出来的 Leader 服务器拥有集群总所有机器编号（即 ZXID 最大）的事务，那么就能够保证这个新选举出来的 Leader 一定具有所有已经提交的提案。

而且这么做有一个好处是：可以省去 Leader 服务器检查事务的提交和丢弃工作的这一步操作。



我们刚刚假设的两个问题便能够解决。假设 1 最终会丢弃调用没有提交的数据，假设 2 最终会同步所有服务器的数据。这个时候，就引出了一个问题，如何同步？

### 数据同步

当崩溃恢复之后，需要在正式工作之前（接收客户端请求），Leader 服务器首先确认事务是否都已经被过半的 Follwer 提交了，即是否完成了数据同步。目的是为了保持数据一致。

当所有的 Follwer 服务器都成功同步之后，Leader 会将这些服务器加入到可用服务器列表中。

### 总结

只有一个 Leader，用来保证一致性

有采取过半即成功的机制保证服务可用

ZAB 让整个 Zookeeper 集群在 【消息广播】和【崩溃恢复】之间转换

而支持崩溃恢复后数据准确性的就是数据同步了，数据同步基于事务的 ZXID 的唯一性来保证。通过 + 1 操作可以辨别事务的先后顺序。

## Zookeeper Server三种角色：

Leader，Follower，Observer。

Leader是Zookeeper 集群工作机制的核心，主要工作：

* a.调度者：集群内部各个服务节点的调度者
* b.事务请求：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

Follower主要职责：

* a.非事务请求：Follower 直接处理非事务请求，对于事务请求，转发给 Leader
* b.Proposal 投票：Leader 上执行事务时，需要 Follower 投票，Leader 才真正执行
* c.Leader 选举投票

Observer主要职责：

* a.非事务请求：Follower 直接处理非事务请求，对于事务请求，转发给 Leader

**Observer 跟 Follower的区别：**

* a.Follower 参与投票：Leader 选举、Proposal 提议投票（事务执行确认）
* b.Observer 不参与投票：只用于提供非事务请求的处理

## 集群选举机制

### **Zookeeper Server的状态**

* LOOKING：寻找Leader
* LEADING：Leader状态，对应的节点为Leader。
* FOLLOWING：Follower状态，对应的节点为Follower。
* OBSERVING：Observer状态，对应节点为Observer，该节点不参与Leader选举。

### **其它概念**

* ZXID（zookeeper transaction id）：每个改变Zookeeper状态的操作都会形成一个对应的zxid，并记录到transaction log中。 这个值越大，表示更新越新
* myid：服务器SID，一个数字,通过配置文件配置，唯一
* SID：服务器的唯一标识
* 成为Leader的必要条件： Leader要具有最高的zxid；当集群的规模是n时，集群中大多数的机器（至少n/2+1）得到响应并follow选出的Leader。
* 心跳机制：Leader与Follower利用PING来感知对方的是否存活，当Leader无法相应PING时，将重新发起Leader选举。

选举有两种情况，一是服务器启动的投票，二是运行期间的投票。

### 服务器启动时期的Leader选举

#### 1.每个服务器发送一个投票(SID,ZXID)

其中sid是自己的myid，初始阶段都将自己投为Leader。

#### **2.接收来自其他服务器的投票。**

集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自LOOKING状态的服务器。

#### **3.处理投票**

针对每个投票都按以下规则与自己的投票PK，PK后依据情况是否更新投票，再发送给其他机器。

* a.优先检查ZXID，ZXID较大者优先为Leader
* b.如果ZXID相同，检查SID，SID较大者优先为Leader

#### **5.统计投票**

每次投票后，服务器统计所有投票，判断是否有过半的机器收到相同的投票，如果某个投票达到一半的要求，则认为该投票提出者可以成为Leader。

#### **6.改变服务器状态**

一旦确定了Leader，每个服务器都更新自己的状态，Leader变更为Leading，Follower变更为Following  
正常情况下一旦选出一个Leader则一直会保持，除非Leader服务器宕掉，则再进行重新选举。

### 服务器运行时期的Leader选举

#### 1.变更状态

当Leader宕机后，余下的所非Observer的服务器都会将自己的状态变更为Looking，然后开启新的Leader选举流程。

#### 2.每个服务器发出一个投票。

生成(SID,ZXID)信息，注意运行期间的ZXID可能是不同的，但是在投票时都会将自己投为Leader，然后发送给其他的服务器。

#### 3.接收来自各个服务器的投票

与启动时过程相同

#### 4.处理投票

与启动时过程相同

#### 5.统计投票

与启动时过程相同

#### 6.改变服务器状态

与启动时过程相同

## ZAB协议简答？

ZAB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。

ZAB协议包括两种基本的模式：崩溃恢复和消息广播。

当整个zookeeper集群刚刚启动或者Leader服务器宕机、重启或者网络故障导致不存在过半的服务器与Leader服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的Leader服务器，然后集群中Follower服务器开始与新的Leader服务器进行数据同步，当集群中超过半数机器与该Leader服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

## 四种类型的数据节点 Znode

* PERSISTENT-持久节点

除非手动删除，否则节点一直存在于Zookeeper上

* EPHEMERAL-临时节点

临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与zookeeper连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。

* PERSISTENT\_SEQUENTIAL-持久顺序节点

基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

* EPHEMERAL\_SEQUENTIAL-临时顺序节点

基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

## Zookeeper 下 Server工作状态

服务器具有四种状态，分别是LOOKING、FOLLOWING、LEADING、OBSERVING。

* LOOKING：寻找Leader状态。当服务器处于该状态时，它会认为当前集群中没有Leader，因此需要进入Leader选举状态。
* FOLLOWING：跟随者状态。表明当前服务器角色是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。

## zookeeper是如何保证事务的顺序一致性的？

zookeeper采用了全局递增的事务Id来标识，所有的proposal（提议）都在被提出的时候加上了zxid，zxid实际上是一个64位的数字，高32位是epoch（时期; 纪元; 世; 新时代）用来标识leader周期，如果有新的leader产生出来，epoch会自增，低32位用来递增计数。当新产生proposal的时候，会依据数据库的两阶段过程，首先会向其他的server发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

## 分布式集群中为什么会有Master？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行leader选举。

## zk节点宕机如何处理？

Zookeeper本身也是集群，推荐配置不少于3个服务器。Zookeeper自身也要保证当一个节点宕机时，其他节点会继续提供服务。  
如果是一个Follower宕机，还有2台服务器提供访问，因为Zookeeper上的数据是有多个副本的，数据并不会丢失；  
如果是一个Leader宕机，Zookeeper会选举出新的Leader。  
ZK集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在ZK节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。  
所以  
3个节点的cluster可以挂掉1个节点(leader可以得到2票>1.5)  
2个节点的cluster就不能挂掉任何1个节点了(leader可以得到1票<=1)

## Zookeeper有哪几种几种部署模式？

部署模式：单机模式、伪集群模式、集群模式。

## 集群最少要几台机器，集群规则是怎样的?

集群规则为2N+1台，N>0，即3台。

## 为什么zookeeper集群的数目，一般为奇数个？

•Leader选举算法采用了Paxos协议；  
　　•Paxos核心思想：当多数Server写成功，则任务数据写成功如果有3个Server，则两个写成功即可；如果有4或5个Server，则三个写成功即可。  
　　•Server数目一般为奇数（3、5、7）如果有3个Server，则最多允许1个Server挂掉；如果有4个Server，则同样最多允许1个Server挂掉由此，

　　  我们看出3台服务器和4台服务器的的容灾能力是一样的，所以为了节省服务器资源，一般我们采用奇数个数，作为服务器部署个数。

## ZAB和Paxos算法的联系与区别？

相同点：

两者都存在一个类似于Leader进程的角色，由其负责协调多个Follower进程的运行

Leader进程都会等待超过半数的Follower做出正确的反馈后，才会将一个提案进行提交

ZAB协议中，每个Proposal中都包含一个 epoch 值来代表当前的Leader周期，Paxos中名字为Ballot

不同点：

ZAB用来构建高可用的分布式数据主备系统（Zookeeper），Paxos是用来构建分布式一致性状态机系统。

## Paxos 算法

基于消息传递的一致性算法

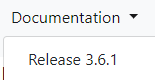
**数据一致性与paxos 算法**

 据说Paxos算法的难理解与算法的知名度一样令人敬仰，所以我们先看如何保持数据的一致性，这里有个原则就是：  
　　•**在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。**  
**• Paxos算法解决的什么问题呢，解决的就是保证每个节点执行相同的操作序列。好吧，这还不简单，master维护一个**  
**全局写队列，所有写操作都必须 放入这个队列编号，那么无论我们写多少个节点，只要写操作是按编号来的，就能保证一**  
**致性。没错，就是这样，可是如果master挂了呢。**  
**• Paxos算法通过投票来对写操作进行全局编号，同一时刻，只有一个写操作被批准，同时并发的写操作要去争取选票，**  
**只有获得过半数选票的写操作才会被 批准（所以永远只会有一个写操作得到批准），其他的写操作竞争失败只好再发起一**  
**轮投票，就这样，在日复一日年复一年的投票中，所有写操作都被严格编号排 序。编号严格递增，当一个节点接受了一个**  
**编号为100的写操作，之后又接受到编号为99的写操作（因为网络延迟等很多不可预见原因），它马上能意识到自己 数据**  
**不一致了，自动停止对外服务并重启同步过程**。任何一个节点挂掉都不会影响整个集群的数据一致性（总2n+1台，除非挂掉大于n台）。

## 集群支持动态添加机器吗？

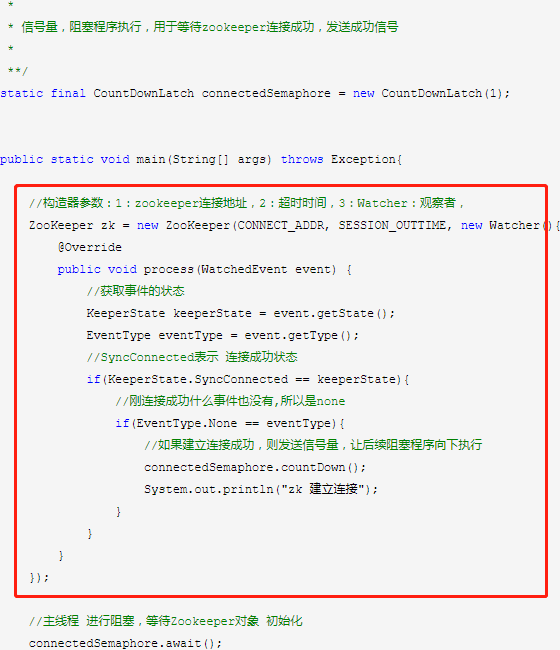
其实就是水平扩容了，Zookeeper在这方面不太好。两种方式：

* 全部重启：关闭所有Zookeeper服务，修改配置之后启动。不影响之前客户端的会话。
* 逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

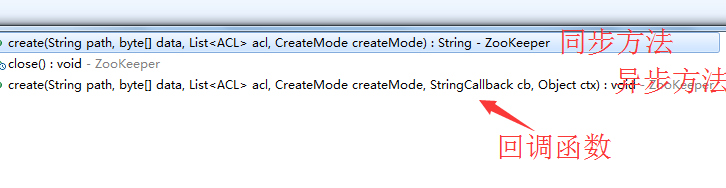
3.5版本开始支持动态扩容。（2020-07-06当前版本）

# ========事件=======

zookeeper客户端和服务器端会话的建立是一个异步的过程



zookeeper增删改查都提供了**同步**和**异步**两套方法，来操作节点，同步的等待操作执行完响应结果，异步的方法另起线程执行。异步的方法提供了回调函数来获取一些参数。



## Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper允许客户端向服务端的某个Znode注册一个Watcher监听，当服务端的一些指定事件触发了这个Watcher，服务端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，然后客户端根据Watcher通知状态和事件类型做出业务上的改变。

工作机制：

客户端注册watcher

服务端处理watcher

客户端回调watcher

**Watcher特性总结：**

1. 一次性

无论是服务端还是客户端，一旦一个Watcher被触发，Zookeeper都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

1. 客户端串行执行

客户端Watcher回调的过程是一个串行同步的过程。

1. 轻量

Watcher通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。

客户端向服务端注册Watcher的时候，并不会把客户端真实的Watcher对象实体传递到服务端，仅仅是在客户端请求中使用boolean类型属性进行了标记。

1. watcher event异步发送watcher的通知事件从server发送到client是异步的，这就存在一个问题，不同的客户端和服务器之间通过socket进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于Zookeeper本身提供了ordering guarantee，即客户端监听事件后，才会感知它所监视znode发生了变化。所以我们使用Zookeeper不能期望能够监控到节点每次的变化。Zookeeper只能保证最终的一致性，而无法保证强一致性。
2. 注册watcher getData、exists、getChildren
3. 触发watcher create、delete、setData
4. 当一个客户端连接到一个新的服务器上时，watch将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到watch的。而当client重新连接时，如果需要的话，所有先前注册过的watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch可能会丢失：对于一个未创建的znode的exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个watch事件可能会被丢失。

## Zookeeper对节点的watch监听通知是永久的吗？为什么不是永久的?

不是。官方声明：一个Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端，以便通知它们。

为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。  
一般是客户端执行getData(“/节点A”,true)，如果节点A发生了变更或删除，客户端会得到它的watch事件，但是在之后节点A又发生了变更，而客户端又没有设置watch事件，就不再给客户端发送。  
在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。