# ACL权限补充

很多同学练习setAcl权限的时候由于失误，导致节点无法删除

create /enjoy1/node1 enjoy

setAcl /enjoy1 world:anyone:r

这个时候无论是delete 还是rmr都没有权限删除

**解决方式:启用super权限**

使用DigestAuthenticationProvider.generateDigest("super:admin"); 获得密码

1. **修改zkServer启动脚本增加**

"-Dzookeeper.DigestAuthenticationProvider.superDigest=super:xQJmxLMiHGwaqBvst5y6rkB6HQs="

1. **启动客户端用管理员登陆**

addauth digest super:admin

# Zookeeper高级

## 一致性协议概述

前面已经讨论过，在分布式环境下，有很多不确定性因素，故障随时都回发生，也讲了CAP理论，BASE理论

我们希望达到，在分布式环境下能搭建一个高可用的，且数据高一致性的服务，目标是这样，但CAP理论告诉我们要达到这样的理想环境是不可能的。这三者最多完全满足2个。

在这个前提下，P（分区容错性）是必然要满足的，因为毕竟是分布式，不能把所有的应用全放到一个服务器里面，这样服务器是吃不消的，而且也存在单点故障问题。

所以，只能从一致性和可用性中找平衡。

怎么个平衡法？在这种环境下出现了BASE理论：

即使无法做到强一致性，但分布式系统可以根据自己的业务特点，采用适当的方式来使系统达到最终的一致性；

BASE由Basically Avaliable 基本可用、Soft state 软状态、Eventually consistent 最终一致性组成，一句话概括就是：平时系统要求是基本可用，除开成功失败，运行有可容忍的延迟状态，但是，无论如何经过一段时间的延迟后系统最终必须达成数据是一致的。

其实可能发现不管是CAP理论，还是BASE理论，他们都是理论，这些理论是需要算法来实现的，今天讲的2PC、3PC、Paxos算法，ZAB算法就是干这事情。

所以今天要讲的这些的前提一定是分布式，解决的问题全部都是在分布式环境下，怎么让系统尽可能的高可用，而且数据能最终能达到一致。

### 两阶段提交 two-phase commit (2PC)

首先来看下2PC，翻译过来叫两阶段提交算法，它本身是一致强一致性算法，所以很适合用作数据库的分布式事务。其实数据库的经常用到的TCC本身就是一种2PC.

回想下数据库的事务，数据库不管是MySQL还是MSSql，本身都提供的很完善的事务支持。

MySQL后面学分表分库的时候会讲到在innodb存储引擎，对数据库的修改都会写到undo和redo中，不只是数据库，很多需要事务支持的都会用到这个思路。

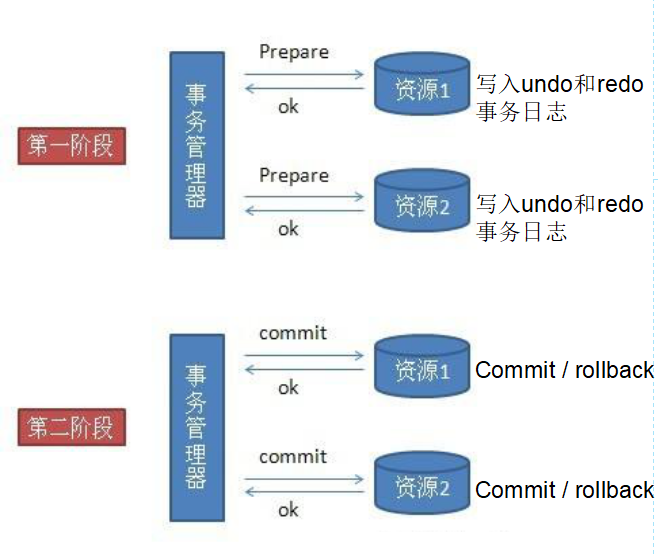
对一条数据的修改操作首先写undo日志，记录的数据原来的样子，接下来执行事务修改操作，把数据写到redo日志里面，万一捅娄子，事务失败了，可从undo里面回复数据。

不只是数据库，在很多企业里面，比如华为等提交数据库修改都回要求这样，你要新增一个字段，首先要把修改数据库的字段SQL提交给DBA（redo），这不够，还需要把删除你提交字段，把数据还原成你修改之前的语句也一并提交者叫（undo）

数据库通过undo与redo能保证数据的强一致性，要解决分布式事务的前提就是当个节点是支持事务的。

这在个前提下，2pc借鉴这失效，首先把整个分布式事务分两节点，首先第一阶段叫准备节点，事务的请求都发送给一个个的资源，这里的资源可以是数据库，也可以是其他支持事务的框架，他们会分别执行自己的事务，写日志到undo与redo，但是不提交事务。

当事务管理器收到了所以资源的反馈，事务都执行没报错后，事务管理器再发送commit指令让资源把事务提交，一旦发现任何一个资源在准备阶段没有执行成功，事务管理器会发送rollback，让所有的资源都回滚。这就是2pc，非常非常简单。



说他是强一致性的是他需要保证任何一个资源都成功，整个分布式事务才成功。

#### 优点：

优点：原理简单，实现方便

#### 缺点：

缺点：同步阻塞，单点问题，数据不一致，容错性不好

同步阻塞

在二阶段提交的过程中，所有的节点都在等待其他节点的响应，无法进行其他操作。这种同步阻塞极大的限制了分布式系统的性能。

单点问题

协调者在整个二阶段提交过程中很重要，如果协调者在提交阶段出现问题，那么整个流程将无法运转。更重要的是，其他参与者将会处于一直锁定事务资源的状态中，而无法继续完成事务操作。

数据不一致

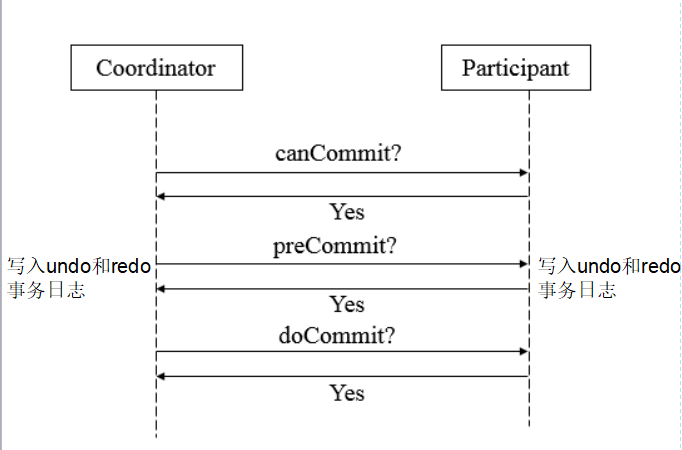
假设当协调者向所有的参与者发送commit请求之后，发生了局部网络异常，或者是协调者在尚未发送完所有 commit请求之前自身发生了崩溃，导致最终只有部分参与者收到了commit请求。这将导致严重的数据不一致问题。

容错性不好

二阶段提交协议没有设计较为完善的容错机制，任意一个节点是失败都会导致整个事务的失败。

### 三阶段提交 three-phase commit (3PC)

由于二阶段提交存在着诸如同步阻塞、单点问题，所以，研究者们在二阶段提交的基础上做了改进，提出了三阶段提交。



#### 第一阶段canCommit

确认所有的资源是否都是健康、在线的，以约女孩举例，你会打个电话问下她是不是在家，而且可以约个会。

如果女孩有空，你在去约她。

就因为有了这一阶段，大大的减少了2段提交的阻塞时间，在2段提交，如果有3个数据库,恰恰第三个数据库出现问题，其他两个都会执行耗费时间的事务操作，到第三个却发现连接不上。3段优化了这种情况

#### 第二阶段PreCommit

如果所有服务都ok，可以接收事务请求，这一阶段就可以执行事务了，这时候也是每个资源都回写redo与undo日志，事务执行成功，返回ack（yes）,否则返回no

#### 第三阶段doCommit

这阶段和前面说的2阶段提交大同小异，这个时候协调者发现所有提交者事务提交者事务都正常执行后，给所有资源发送commit指令。

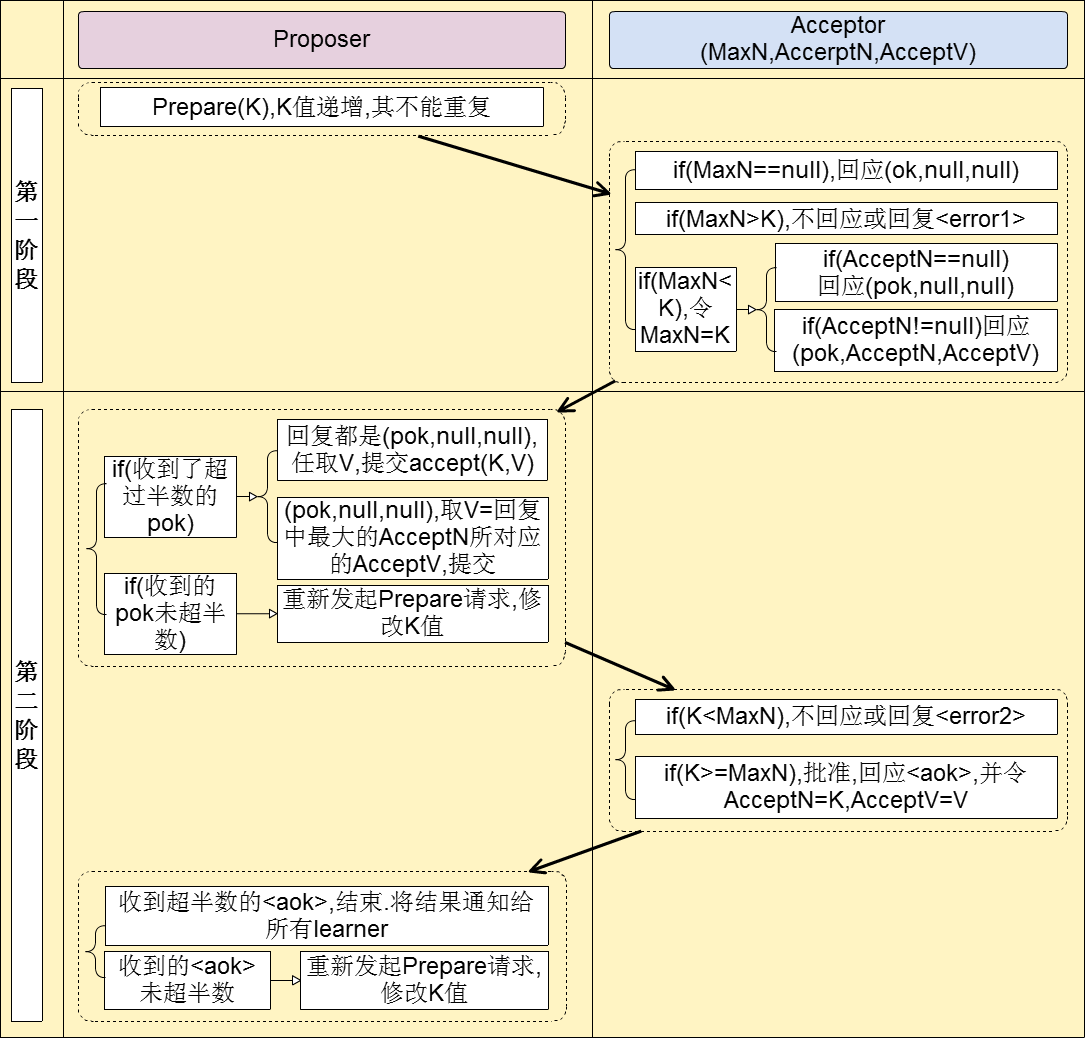
和二阶段提交有所不同的是，他要求所有事务在协调者出现问题，没给资源发送commit指令的时候，三阶段提交算法要求资源在一段时间超时后回默认提交做commit操作。

这样的要求就减少了前面说的单点故障，万一事务管理器出现问题，事务也回提交。

但回顾整个过程，不管是2pc,还是3pc，同步阻塞，单点故障，容错机制不完善这些问题都没本质上得到解决，尤其是前面说得数据一致性问题，反而更糟糕了。

所有数据库的分布式事务一般都是二阶段提交，而者三阶段的思想更多的被借鉴扩散成其他的算法。

### Paxos算法



这个算法还是有点难度的，本身这算法的提出者莱斯利·兰伯特在前面几篇论文中都不是以严谨的数学公式进行的。

其实这个paxos算法也分成两阶段。首先这个图有2个角色，提议者与接收者

#### 第一阶段

提议者对接收者吼了一嗓子，我有个事情要告诉你们，当然这里接受者不只一个，它也是个分布式集群

相当于星期一开早会，可耻的领导吼了句：“要开会了啊，我要公布一个编号为001的提案，收到请回复”。

这个时候领导就会等着，等员工回复1“好的”，如果回复的数目超过一半，就会进行下一步。

如果由于某些原因（接收者死机，网络问题，本身业务问题），导通过的协议未超过一半，

这个时候的领导又会再吼一嗓子，当然气势没那凶残：“好了，怕了你们了，我要公布一个新的编号未002的提案，收到请回复1”【就其实和老师讲课很像，老师经常问听懂了吗？听懂的回1，没懂的回2，只有回复1的占了大多数，才能讲下个知识点】

#### 第二阶段

接下来到第二阶段，领导苦口婆心的把你们叫来开会了，今天编号002提案的内容是：“由于项目紧张，今天加班到12点，同意的请举手”这个时候如果绝大多少的接收者都同意，那么好，议案就这么决定了，如果员工反对或者直接夺门而去，那么领导又只能从第一个阶段开始：“大哥，大姐们，我有个新的提案003，快回会议室吧。。”

#### 详细说明：

【注意：不懂没事，记住上面那简单情况就好，面试足够】

上面那个故事描绘的是个苦逼的领导和凶神恶煞的员工之间的斗争，通过这个故事你们起码要懂paxos协议的流程是什么样的（paxos的核心就是少数服从多数）。

上面的故事有两个问题：

**苦逼的领导（单点问题）**：有这一帮凶残的下属，这领导要不可能被气死，要不也会辞职，这是单点问题。

**凶神恶煞的下属（一致性问题）**：如果员工一种都拒绝，故意和领导抬杆，最终要产生一个一致性的解决方案是不可能的。

所以paxos协议肯定不会只有一个提议者，作为下属的员工也不会那么强势

协议要求：如果接收者没有收到过提案编号，他必须接受第一个提案编号

如果接收者没有收到过其他协议，他必须接受第一个协议。

举一个例子：

有2个Proposer(老板，老板之间是竞争关系)和3个Acceptor(政府官员)：

阶段一

1.现在需要对一项议题来进行paxos过程，议题是“A项目我要中标！”，这里的“我”指每个带着他的秘书Proposer的Client老板。

2.Proposer当然听老板的话了，赶紧带着议题和现金去找Acceptor政府官员。

3.作为政府官员，当然想谁给的钱多就把项目给谁。

4.Proposer-1小姐带着现金同时找到了Acceptor-1~Acceptor-3官员，1与2号官员分别收取了10比特币，找到第3号官员时，没想到遭到了3号官员的鄙视，3号官员告诉她，Proposer-2给了11比特币。不过没关系，Proposer-1已经得到了1,2两个官员的认可，形成了多数派(如果没有形成多数派，Proposer-1会去银行提款在来找官员们给每人20比特币，这个过程一直重复每次+10比特币，直到多数派的形成)，满意的找老板复命去了，但是此时Proposer-2保镖找到了1,2号官员，分别给了他们11比特币，1,2号官员的态度立刻转变，都说Proposer-2的老板懂事，这下子Proposer-2放心了，搞定了3个官员，找老板复命去了，当然这个过程是第一阶段提交，只是官员们初步接受贿赂而已。故事中的比特币是编号，议题是value。

这个过程保证了在某一时刻，某一个proposer的议题会形成一个多数派进行初步支持

阶段二

5.　现在进入第二阶段提交，现在proposer-1小姐使用分身术(多线程并发)分了3个自己分别去找3位官员，最先找到了1号官员签合同，遭到了1号官员的鄙视，1号官员告诉他proposer-2先生给了他11比特币，因为上一条规则的性质proposer-1小姐知道proposer-2第一阶段在她之后又形成了多数派(至少有2位官员的赃款被更新了);此时她赶紧去提款准备重新贿赂这3个官员(重新进入第一阶段)，每人20比特币。刚给1号官员20比特币， 1号官员很高兴初步接受了议题，还没来得及见到2,3号官员的时候

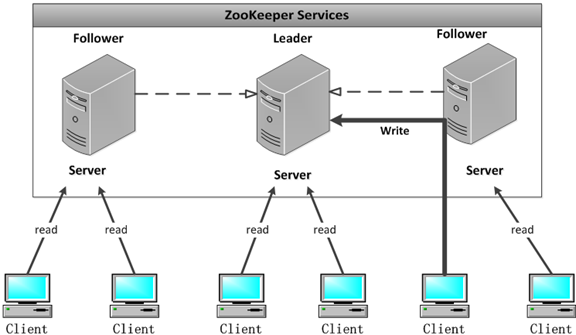
这时proposer-2先生也使用分身术分别找3位官员(注意这里是proposer-2的第二阶段)，被第1号官员拒绝了告诉他收到了20比特币，第2,3号官员顺利签了合同，这时2，3号官员记录client-2老板用了11比特币中标，因为形成了多数派，所以最终接受了Client2老板中标这个议题，对于proposer-2先生已经出色的完成了工作；

这时proposer-1小姐找到了2号官员，官员告诉她合同已经签了，将合同给她看，proposer-1小姐是一个没有什么职业操守的聪明人，觉得跟Client1老板混没什么前途，所以将自己的议题修改为“Client2老板中标”，并且给了2号官员20比特币，这样形成了一个多数派。顺利的再次进入第二阶段。由于此时没有人竞争了，顺利的找3位官员签合同，3位官员看到议题与上次一次的合同是一致的，所以最终接受了，形成了多数派，proposer-1小姐跳槽到Client2老板的公司去了。

总结：Paxos过程结束了，这样，一致性得到了保证，算法运行到最后所有的proposer都投“client2中标”所有的acceptor都接受这个议题，也就是说在最初的第二阶段，议题是先入为主的，谁先占了先机，后面的proposer在第一阶段就会学习到这个议题而修改自己本身的议题，因为这样没职业操守，才能让一致性得到保证，这就是paxos算法的一个过程。原来paxos算法里的角色都是这样的不靠谱，不过没关系，结果靠谱就可以了。该算法就是为了追求结果的一致性。

## ZK集群解析

### Zookeeper集群特点



前面一种研究的单节点，现在来研究下zk集群，首先来看下zk集群的特点。

* 顺序一致性  
  客户端的更新顺序与它们被发送的顺序相一致。
* 原子性  
  更新操作要么成功要么失败，没有第三种结果。
* 单一视图  
  无论客户端连接到哪一个服务器，客户端将看到相同的 ZooKeeper 视图。
* 可靠性  
  一旦一个更新操作被应用，那么在客户端再次更新它之前，它的值将不会改变。
* 实时性  
  连接上一个服务端数据修改，所以其他的服务端都会实时的跟新，不算完全的实时，有一点延时的
* 角色轮换避免单点故障  
  当leader出现问题的时候，会选举从follower中选举一个新的leader

### 集群中的角色

* Leader 集群工作机制中的核心

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

集群内部个服务器的调度者(管理follower,数据同步)

* Follower 集群工作机制中的跟随者

处理非事务请求，转发事务请求给Leader

参与事务请求proposal投票

参与leader选举投票

* Observer 观察者

3.30以上版本提供，和follower功能相同，但不参与任何形式投票

处理非事务请求，转发事务请求给Leader

提高集群非事务处理能力

### Zookeeper集群配置

1.安装jdk运行jdk环境

|  |
| --- |
| 上传jdk1.8安装包 |

2.安装jdk1.8环境变量

|  |
| --- |
| vi /etc/profile  export JAVA\_HOME=/usr/local/jdk1.8.0\_181  export ZOOKEEPER\_HOME=/usr/local/zookeeper  export CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar  export PATH=$JAVA\_HOME/bin:$ZOOKEEPER\_HOME/bin:$PATH    刷新profile文件  source /etc/profile  关闭防火墙 |

3.下载zookeeper安装包

|  |
| --- |
| wget https://mirrors.tuna.tsinghua.edu.cn/apache/zookeeper/zookeeper-3.4.10/zookeeper-3.4.10.tar.gz |

4.解压Zookeeper安装包

|  |
| --- |
| tar -zxvf zookeeper-3.4.10.tar.gz |

5.修改Zookeeper文件夹名称

|  |
| --- |
| 重命名： mv zookeeper-3.4.10 zookeeper |

1. 修改zoo\_sample.cfg文件

|  |
| --- |
| cd /usr/local/zookeeper/conf mv zoo\_sample.cfg zoo.cfg 修改conf: vi zoo.cfg 修改两处 （1） dataDir=/usr/local/zookeeper/data（注意同时在zookeeper创建data目录） （2）最后面添加 server.0=192.168.212.154:2888:3888 server.1=192.168.212.156:2888:3888 server.2=192.168.212.157:2888:3888 |

7.创建服务器标识  
服务器标识配置：  
创建文件夹： mkdir data  
创建文件myid并填写内容为0： vi  
myid (内容为服务器标识 ： 0)

8.复制zookeeper

进行复制zookeeper目录到node1和node2  
还有/etc/profile文件  
把node1、 node2中的myid文件里的值修改为1和2  
路径(vi /usr/local/zookeeper/data/myid)

9启动zookeeper  
启动zookeeper：  
路径： /usr/local/zookeeper/bin  
执行： zkServer.sh start  
(注意这里3台机器都要进行启动)  
状态： zkServer.sh   
status(在三个节点上检验zk的mode,一个leader和俩个follower)

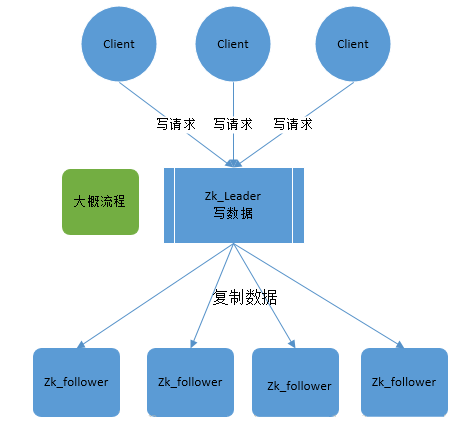
### Zookeeper集群一致性协议ZAB解析

#### 总览

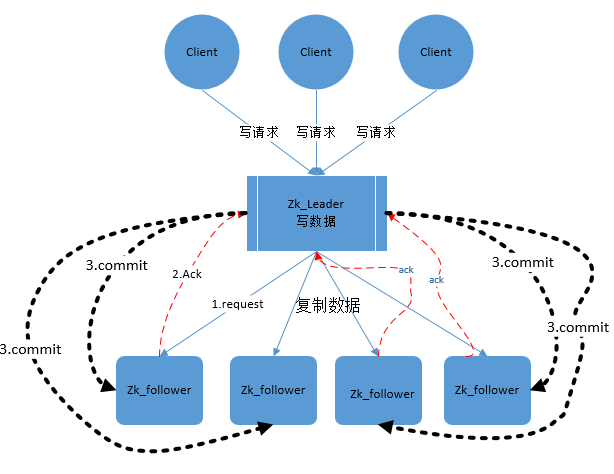
懂了paxos算法，其实zab就很好理解了。很多论文和资料都证明zab其实就是paxos的一种简化实现，但Apache 自己的立场说zab不是paxos算法的实现，这个不需要去计较。

zab协议解决的问题和paxos一样，是解决分布式系统的数据一致性问题

zookeeper就是根据zab协议建立了主备模型完成集群的数据同步（保证数据的一致性），前面介绍了集群的各种角色，这说所说的主备架构模型指的是，在zookeeper集群中，只有一台leader（主节点）负责处理外部客户端的事务请求（写操作），leader节点负责将客户端的写操作数据同步到所有的follower节点中。

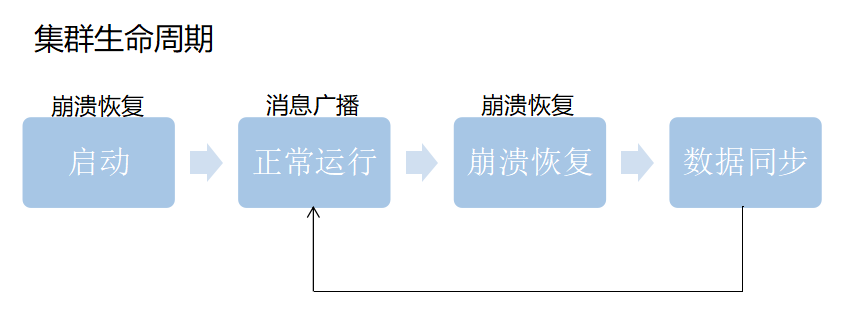


zab协议核心是在整个zookeeper集群中只有一个节点既leader将所有客户端的写操作转化为事务（提议proposal）.leader节点再数据写完之后，将向所有的follower节点发送数据广播请求（数据复制），等所有的follower节点的反馈，在zab协议中，只要超过半数follower节点反馈ok,leader节点会向所有follower服务器发送commit消息，既将leader节点上的数据同步到follower节点之上。



发现，整个流程其实和paxos协议其实大同小异。说zab是paxos的一种实现方式其实并不过分。

Zab再细看可以分成两部分。第一的消息广播模式，第二是崩溃恢复模式。



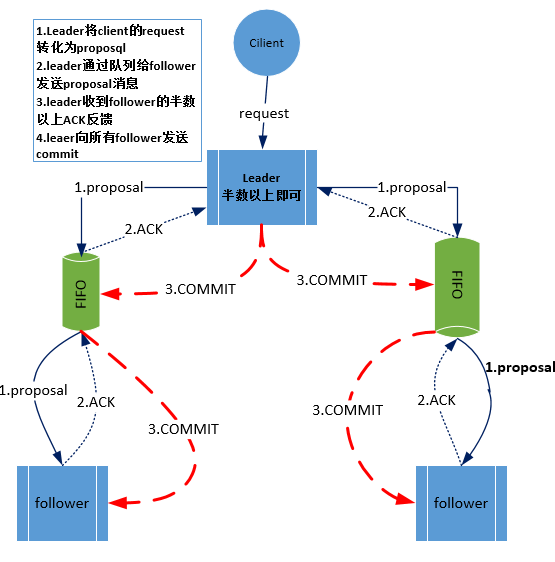
正常情况下当客户端对zk有写的数据请求时，leader节点会把数据同步到follower节点，这个过程其实就是消息的广播模式

在新启动的时候，或者leader节点奔溃的时候会要选举新的leader，选好新的leader之后会进行一次数据同步操作，整个过程就是奔溃恢复。

#### 消息广播模式

为了保证分区容错性，zookeeper是要让每个节点副本必须是一致的

1. 在zookeeper集群中数据副本的传递策略就是采用的广播模式
2. Zab协议中的leader等待follower的ack反馈，只要半数以上的follower成功反馈就好，不需要收到全部的follower反馈。



zookeeper中消息广播的具体步骤如下：

1. 客户端发起一个写操作请求

2. Leader服务器将客户端的request请求转化为事物proposql提案，同时为每个proposal分配一个全局唯一的ID，即ZXID。

3. leader服务器与每个follower之间都有一个队列，leader将消息发送到该队列

4. follower机器从队列中取出消息处理完(写入本地事物日志中)毕后，向leader服务器发送ACK确认。

5. leader服务器收到半数以上的follower的ACK后，即认为可以发送commit

6. leader向所有的follower服务器发送commit消息。

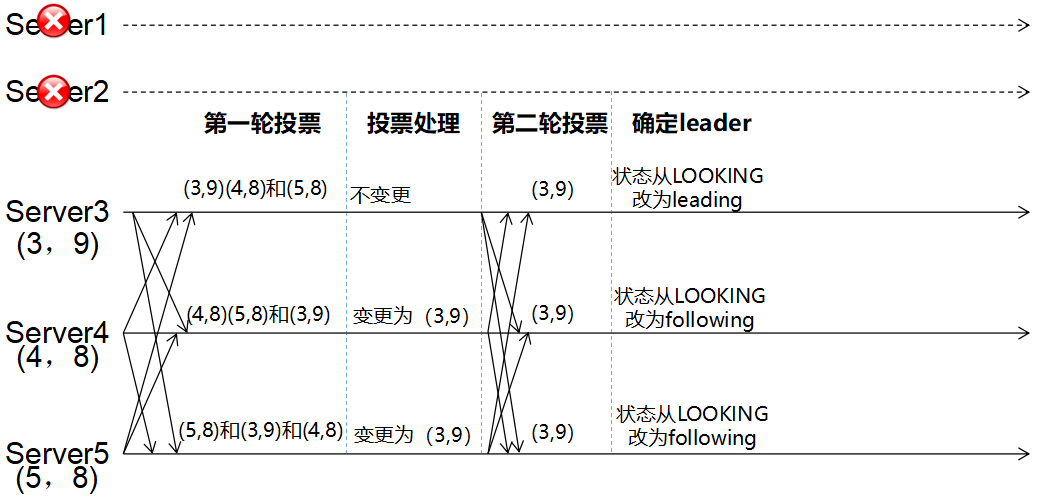
zookeeper采用ZAB协议的核心就是只要有一台服务器提交了proposal，就要确保所有的服务器最终都能正确提交proposal。这也是CAP/BASE最终实现一致性的一个体现。

回顾一下：前面还讲了2pc协议，也就是两阶段提交，发现流程2pc和zab还是挺像的，

zookeeper中数据副本的同步方式与二阶段提交相似但是却又不同。二阶段提交的要求协调者必须等到所有的参与者全部反馈ACK确认消息后，再发送commit消息。要求所有的参与者要么全部成功要么全部失败。二阶段提交会产生严重阻塞问题，但paxos和2pc没有这要求。

为了进一步防止阻塞，leader服务器与每个follower之间都有一个单独的队列进行收发消息，使用队列消息可以做到异步解耦。leader和follower之间只要往队列中发送了消息即可。如果使用同步方式容易引起阻塞。性能上要下降很多

#### 崩溃恢复



背景（什么情况下会崩溃恢复）

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。

如果leader服务器发生崩溃(重启是一种特殊的奔溃，这时候也没leader)，则zab协议要求zookeeper集群进行崩溃恢复和leader服务器选举。

最终目的（恢复成什么样）

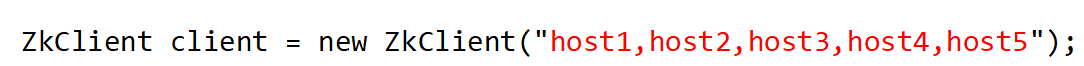
ZAB协议崩溃恢复要求满足如下2个要求：   
确保已经被leader提交的proposal必须最终被所有的follower服务器提交。   
确保丢弃已经被leader出的但是没有被提交的proposal。

新选举出来的leader不能包含未提交的proposal，即新选举的leader必须都是已经提交了的proposal的follower服务器节点。同时，新选举的leader节点中含有最高的ZXID。这样做的好处就是可以避免了leader服务器检查proposal的提交和丢弃工作。

* 每个Server会发出一个投票,第一次都是投自己。投票信息：（myid，ZXID）
* 收集来自各个服务器的投票
* 处理投票并重新投票，处理逻辑：优先比较ZXID,然后比较myid
* 统计投票，只要超过半数的机器接收到同样的投票信息，就可以确定leader
* 改变服务器状态

问题：为什么优先选大的zxid

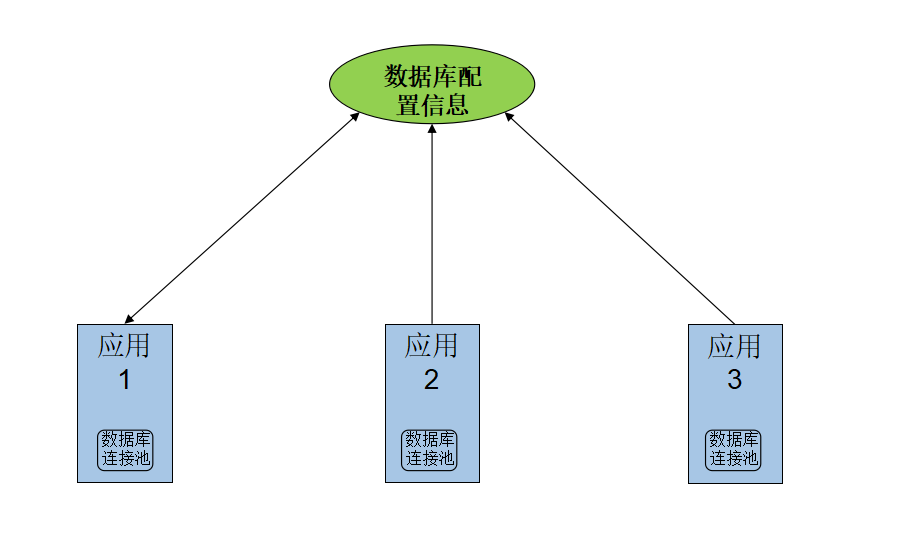
### Java客户端连接集群



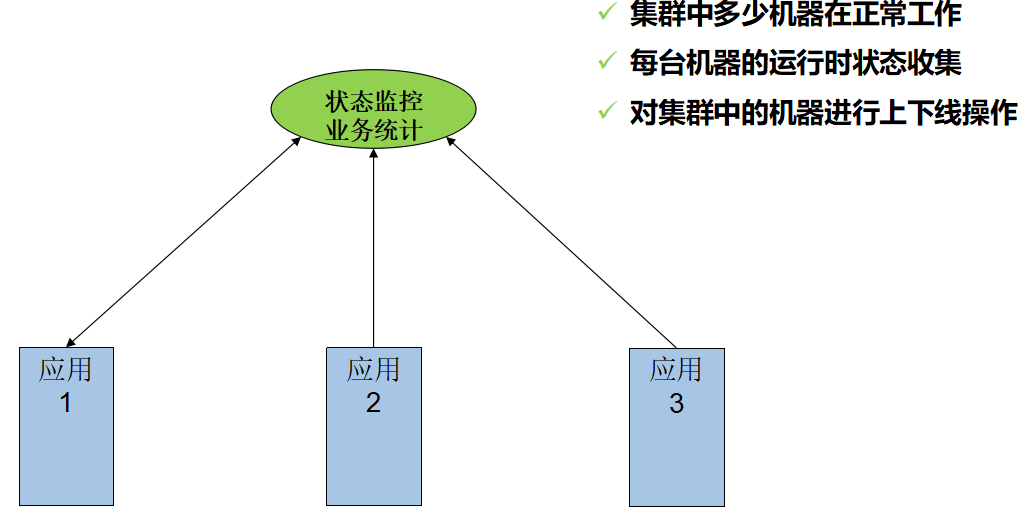
ZK连接集群很简单，只需要把连接地址用逗号分隔就好。

## 典型应用场景

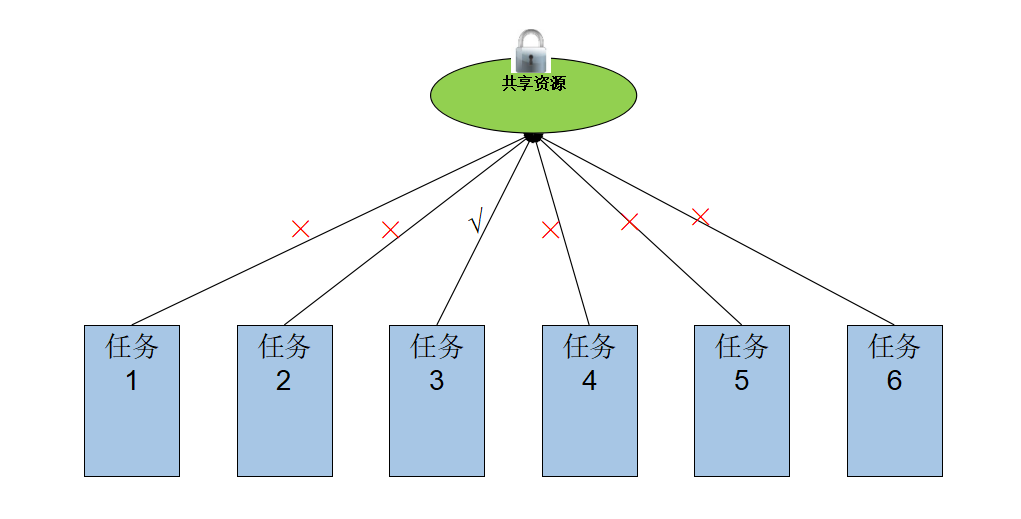
### 数据发布与订阅（配置中心）



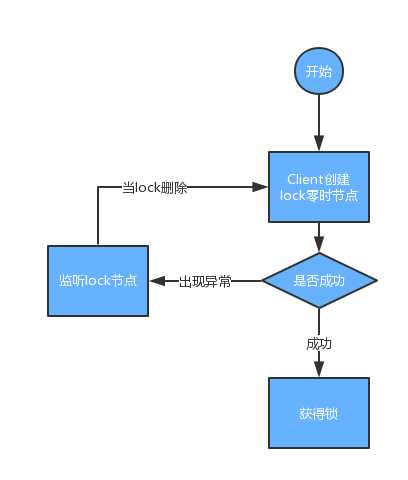
### 集群管理（服务注册于发现）



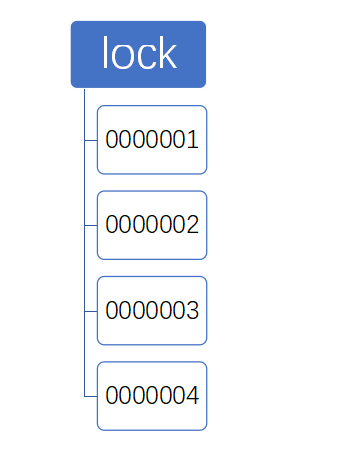
### 分布式锁

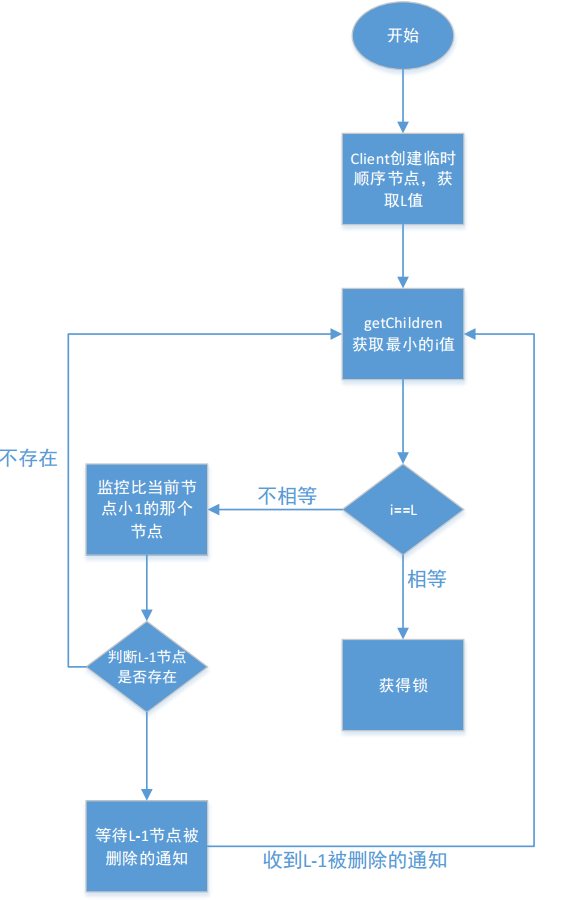


#### 基于同名节点的分布式锁

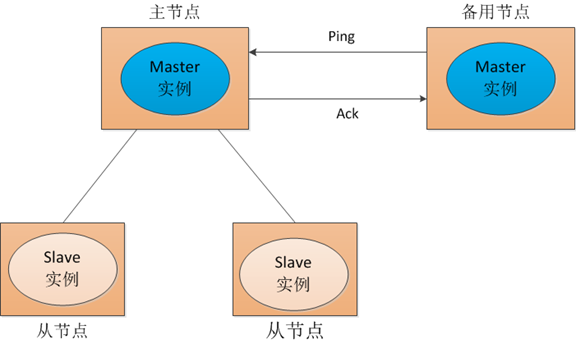


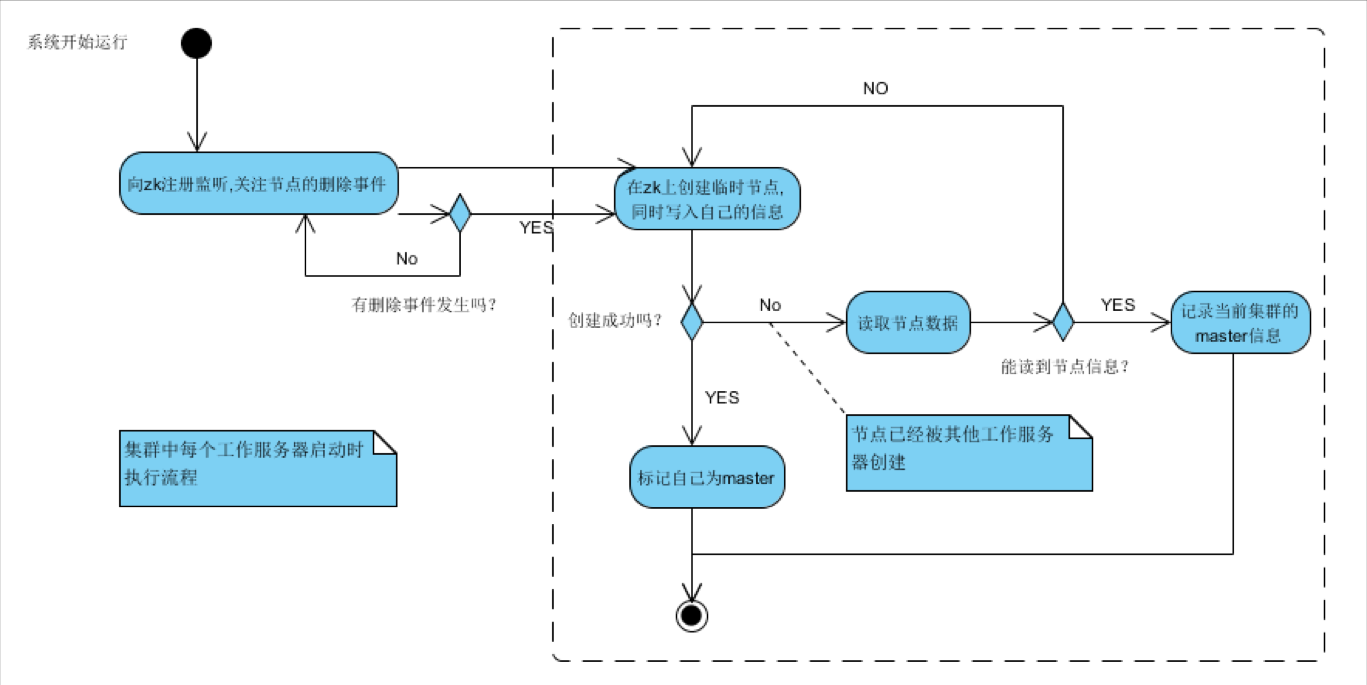
#### 高性能分布式锁





### Master选举





### 命名服务

## ZK使用注意事项

### Zk数据与日志清理

dataDir目录、dataLogDir两个目录会随着时间推移变得庞大，容易造成硬盘满了，清理办法：

自己编写shell脚本，保留最新的n个文件

使用zk自带的zkClient.sh保留最新的n个文件，zkClient.sh –n 15

配置autopurge.snapRetainCount和autopurge.purgeInterval两个参数配合使用；

### Too many connections

配置maxClientCnxns参数，配置单个客户端机器创建的最大连接数；

### 磁盘管理

磁盘的I/O性能直接制约zookeeper更新操作速度，为了提高zk的写性能建议：

使用单独的磁盘

Jvm堆内存设置要小心

### 磁盘管理集群数量

集群中机器的数量并不是越多越好，一个写操作需要半数以上的节点ack，所以集群节点数越多，整个集群可以抗挂点的节点数越多(越可靠)，但是吞吐量越差。集群的数量必须为奇数；

### 磁盘管理集群数量

zk是基于内存进行读写操作的，有时候会进行消息广播，因此不建议在节点存取容量比较大的数据；