**Zookeeper**

**学习手册**

**V1.0**

**张镇**

**目 录**

[1 简介 1](#_Toc490050951)

[2 入门 1](#_Toc490050952)

[2.1 ZooKeeper的基本概念 1](#_Toc490050953)

[2.1.1 角色 1](#_Toc490050954)

[2.1.2 设计目的 2](#_Toc490050955)

[2.2 ZooKeeper的工作原理 2](#_Toc490050956)

[2.2.1 选主流程 3](#_Toc490050957)

[2.2.2 同步流程 5](#_Toc490050958)

[2.2.3 工作流程 6](#_Toc490050959)

[2.2.3.1 Leader工作流程 6](#_Toc490050960)

[2.2.3.2 Follower工作流程 7](#_Toc490050961)

[2.3 安装和配置详解 10](#_Toc490050962)

[2.3.1 单机模式 10](#_Toc490050963)

[2.3.2 集群模式 11](#_Toc490050964)

[2.3.3 数据模型 12](#_Toc490050965)

[2.4 如何使用 14](#_Toc490050966)

[2.4.1 常用接口列表 14](#_Toc490050967)

[2.4.2 基本操作 16](#_Toc490050968)

[2.5 常用概念简介 17](#_Toc490050969)

[2.5.1 Zookeeper 监视（Watches） 简介 17](#_Toc490050970)

[2.5.2 Zookeeper C API 常量与部分结构(struct)介绍 18](#_Toc490050971)

[2.5.2.1 与 ACL 相关的结构与常量： 18](#_Toc490050972)

[2.5.2.2 与 Interest 相关的常量：ZOOKEEPER\_WRITE, ZOOKEEPER\_READ 19](#_Toc490050973)

[2.5.2.3 与节点创建相关的常量：ZOO\_EPHEMERAL, ZOO\_SEQUENCE 19](#_Toc490050974)

[2.5.2.4 与连接状态 Stat 相关的常量 19](#_Toc490050975)

[2.5.2.5 与监视类型(Watch Types)相关的常量 20](#_Toc490050976)

[2.5.3 Zookeeper C API 错误码介绍 ZOO\_ERRORS 20](#_Toc490050977)

[2.5.4 Watch事件类型： 21](#_Toc490050978)

[2.6 ZooKeeper 典型的应用场景 22](#_Toc490050979)

[2.6.1 统一命名服务（Name Service） 22](#_Toc490050980)

[2.6.2 配置管理（Configuration Management） 22](#_Toc490050981)

[2.6.3 集群管理（Group Membership） 23](#_Toc490050982)

[2.6.4 共享锁（Locks） 25](#_Toc490050983)

[2.6.5 队列管理 26](#_Toc490050984)

[2.7 总结 29](#_Toc490050985)

[3 进阶 30](#_Toc490050986)

[3.1 ZooKeeper简单介绍 30](#_Toc490050987)

[3.1.1 分布式协调技术 30](#_Toc490050988)

[3.1.2 分布式锁的实现 31](#_Toc490050989)

[3.1.2.1 面临的问题 31](#_Toc490050990)

[3.1.2.2 分布式锁的实现者 32](#_Toc490050991)

[3.1.3 ZooKeeper概述 32](#_Toc490050992)

[3.1.4 ZooKeeper数据模型 33](#_Toc490050993)

[3.1.4.1 ZooKeeper数据模型Znode 33](#_Toc490050994)

[3.1.4.2 ZooKeeper中的时间 35](#_Toc490050995)

[3.1.4.3 ZooKeeper节点属性 35](#_Toc490050996)

[3.1.5 ZooKeeper服务中操作 36](#_Toc490050997)

[3.1.6 Watch触发器 37](#_Toc490050998)

[3.1.7 ZooKeeper应用举例 38](#_Toc490050999)

[3.1.7.1 分布式锁应用场景 38](#_Toc490051000)

[3.1.7.2 传统解决方案 40](#_Toc490051001)

[3.1.7.3 ZooKeeper解决方案 42](#_Toc490051002)

[3.2 ZooKeeper安装配置 45](#_Toc490051003)

[3.2.1 Zookeeper的搭建方式 45](#_Toc490051004)

[3.2.1.1 Zookeeper的单机模式搭建 45](#_Toc490051005)

[3.2.1.2 Zookeeper的伪集群模式搭建 45](#_Toc490051006)

[3.2.1.2.1 注意事项 46](#_Toc490051007)

[3.2.1.2.2 启动 48](#_Toc490051008)

[3.2.1.3 Zookeeper的集群模式搭建 49](#_Toc490051009)

[3.2.1.3.1 创建myid 49](#_Toc490051010)

[3.2.1.3.2 编写配置文件 50](#_Toc490051011)

[3.2.1.3.3 启动 50](#_Toc490051012)

[3.2.2 Zookeeper的配置 50](#_Toc490051013)

[3.2.2.1 基本配置 51](#_Toc490051014)

[3.2.2.2 高级配置 51](#_Toc490051015)

[3.2.2.3 集群配置 52](#_Toc490051016)

[3.2.3 搭建ZooKeeper服务器集群 52](#_Toc490051017)

[3.2.3.1 安装配置ZK 53](#_Toc490051018)

[3.2.3.2 修改ZK配置文件 53](#_Toc490051019)

[3.2.3.3 配置其他节点 55](#_Toc490051020)

[3.2.4 启动检验 55](#_Toc490051021)

[3.3 ZooKeeper命令操作 56](#_Toc490051022)

[3.3.1 Zookeeper的四字命令 56](#_Toc490051023)

[3.3.2 Zookeeper的简单操作 58](#_Toc490051024)

[3.3.2.1 Zookeeper的shell操作 58](#_Toc490051025)

[3.3.2.1.1 Zookeeper命令工具 58](#_Toc490051026)

[3.3.2.1.2 使用Zookeeper命令的简单操作步骤 60](#_Toc490051027)

[3.3.2.2 Zookeeper的api的简单使用 62](#_Toc490051028)

[3.3.2.2.1 ZookeeperAPI简介 62](#_Toc490051029)

[3.3.2.2.2 Zookeeper API的使用 63](#_Toc490051030)

[3.3.3 ZooKeeper示例 64](#_Toc490051031)

[3.3.3.1 创建组 65](#_Toc490051032)

[3.3.3.1.1 代码示例 65](#_Toc490051033)

[3.3.3.1.2 代码分析 67](#_Toc490051034)

[3.3.3.2 加入组 68](#_Toc490051035)

[3.3.3.3 列出组成员 70](#_Toc490051036)

[3.3.3.4 ZooKeeper命令行工具 73](#_Toc490051037)

[3.3.3.5 删除组 73](#_Toc490051038)

[3.4 构建ZooKeeper应用 74](#_Toc490051039)

[3.4.1 配置服务 74](#_Toc490051040)

[3.4.2 可恢复的ZooKeeper应用 79](#_Toc490051041)

[3.4.2.1 ZooKeeper异常 79](#_Toc490051042)

[3.4.2.2 可靠地服务配置 80](#_Toc490051043)

[3.4.3 锁服务 84](#_Toc490051044)

[3.4.3.1 分布式锁概述 84](#_Toc490051045)

[3.4.3.2 当前问题与方案 85](#_Toc490051046)

[3.4.3.2.1 羊群效应 85](#_Toc490051047)

[3.4.3.2.2 可恢复的异常 85](#_Toc490051048)

[3.4.3.2.3 不可恢复的异常 86](#_Toc490051049)

[3.4.4 ZooKeeper实现共享锁 86](#_Toc490051050)

[3.4.4.1 场景描述 87](#_Toc490051051)

[3.4.4.2 利用节点名称的唯一性来实现共享锁 87](#_Toc490051052)

[3.4.4.3 利用顺序节点实现共享锁 88](#_Toc490051053)

[3.4.4.4 ZooKeeper提供的一个写锁实现 90](#_Toc490051054)

[3.4.4.5 更多分布式数据结构和协议 98](#_Toc490051055)

[3.4.5 BooKeeper 98](#_Toc490051056)

[3.4.5.1 BooKeeper概述 98](#_Toc490051057)

[3.4.5.2 BooKeeper角色 99](#_Toc490051058)

[3.5 ZooKeeper管理分布式环境中的数据 101](#_Toc490051059)

[3.5.1 ZooKeeper产生背景 101](#_Toc490051060)

[3.5.1.1 分布式的发展 101](#_Toc490051061)

[3.5.1.2 ZooKeeper的产生 101](#_Toc490051062)

[3.5.1.3 ZooKeeper的使用 102](#_Toc490051063)

[3.5.2 ZooKeeper应用场景 102](#_Toc490051064)

[3.5.2.1 数据发布与订阅 104](#_Toc490051065)

[3.5.2.2 统一命名服务（Name Service） 105](#_Toc490051066)

[3.5.2.3 分布通知/协调（Distribution of notification/coordination） 106](#_Toc490051067)

[3.5.2.4 分布式锁（Distribute Lock） 107](#_Toc490051068)

[3.5.2.5 集群管理（Cluster Management） 112](#_Toc490051069)

[3.5.2.6 队列管理 117](#_Toc490051070)

[3.5.3 ZooKeeper实际应用 124](#_Toc490051071)

[3.6 ZooKeeper机制架构 125](#_Toc490051072)

[3.6.1 ZooKeeper权限管理机制 125](#_Toc490051073)

[3.6.1.1 权限管理ACL(Access Control List) 125](#_Toc490051074)

[3.6.1.2 ZooKeeper SuperDigest 127](#_Toc490051075)

[3.6.2 Watch机制 128](#_Toc490051076)

[3.6.3 Session机制 129](#_Toc490051077)

[3.6.3.1 会话概述 129](#_Toc490051078)

[3.6.3.2 故障切换 130](#_Toc490051079)

[3.6.4 ZooKeeper实例状态 130](#_Toc490051080)

[3.7 ZooKeeper一致性原理 132](#_Toc490051081)

[3.7.1 ZooKeeper 的实现 132](#_Toc490051082)

[3.7.1.1 ZooKeeper处理单点故障 132](#_Toc490051083)

[3.7.1.2 ZooKeeper运行模式 133](#_Toc490051084)

[3.7.1.3 ZooKeeper的读写机制 134](#_Toc490051085)

[3.7.2 ZooKeeper的保证 136](#_Toc490051086)

[3.7.2.1 CAP理论 136](#_Toc490051087)

[3.7.2.2 ZooKeeper与CAP理论 137](#_Toc490051088)

[3.7.3 ZooKeeper原理 138](#_Toc490051089)

[3.7.3.1 原理概述 138](#_Toc490051090)

[3.7.3.2 Zab协议详解 140](#_Toc490051091)

[3.7.3.2.1 广播模式 140](#_Toc490051092)

[3.7.3.2.2 恢复模式 141](#_Toc490051093)

[3.8 ZooKeeper的伸缩性 147](#_Toc490051094)

[3.8.1 ZooKeeper中Observer 147](#_Toc490051095)

[3.8.1.1 ZooKeeper角色 147](#_Toc490051096)

[3.8.1.2 为什么引入Observer 148](#_Toc490051097)

[3.8.2 Observer应用 150](#_Toc490051098)

[3.8.3 ZooKeeper集群搭建案例 151](#_Toc490051099)

# 简介

Zookeeper 分布式服务框架是 Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，它包含一个简单的原语集，分布式应用程序可以基于它实现同步服务，配置维护和命名服务等。在分布式应用中，由于工程师不能很好地使用锁机制，以及基于消息的协调机制不适合在某些应用中使用，因此需要有一种可靠的、可扩展的、分布式的、可配置的协调机制来统一系统的状态。Zookeeper的目的就在于此。

本文将从使用者角度详细介绍 Zookeeper 的安装和配置文件中各个配置项的意义，以及分析 Zookeeper 的典型的应用场景（配置文件的管理、集群管理、同步锁、Leader 选举、队列管理等），用 Java 实现它们并给出示例代码。

# 入门

## ZooKeeper的基本概念

### 角色

Zookeeper中的角色主要有以下三类，如下表所示：



系统模型如图所示：



### 设计目的

1.最终一致性：client不论连接到哪个Server，展示给它都是同一个视图，这是zookeeper最重要的性能。

2 .可靠性：具有简单、健壮、良好的性能，如果消息被一台服务器接受，那么它将被所有的服务器接受。

3 .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。

4 .等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

5.原子性：更新只能成功或者失败，没有中间状态。

6 .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。

## ZooKeeper的工作原理

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，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，标识当前属于哪个leader的统治时期。低32位用于递增计数。

每个Server在工作过程中有三种状态：

* LOOKING：当前Server不知道leader是谁，正在搜寻
* LEADING：当前Server即为选举出来的leader
* FOLLOWING：leader已经选举出来，当前Server与之同步

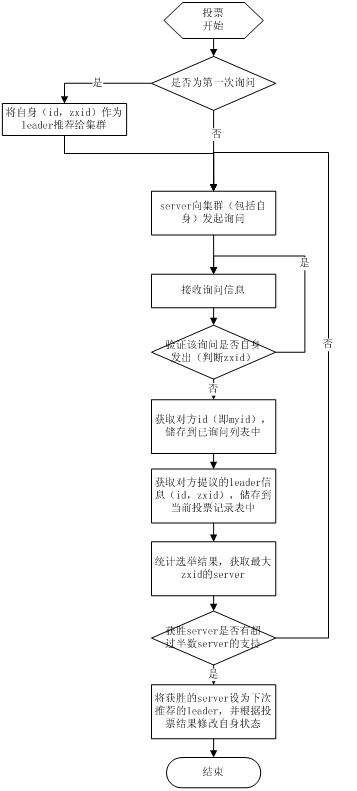
### 选主流程

当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。先介绍basic paxos流程：

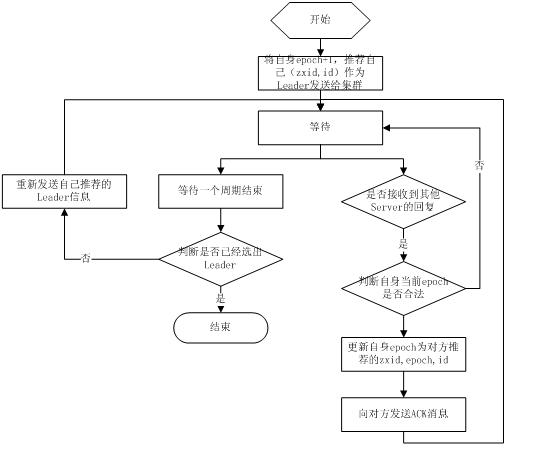
1. 选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；
2. 选举线程首先向所有Server发起一次询问(包括自己)；
3. 选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；
4. 收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；
5. 线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数， 设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。

通过流程分析我们可以得出：要使Leader获得多数Server的支持，则Server总数必须是奇数2n+1，且存活的Server的数目不得少于n+1.

每个Server启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。选主的具体流程图如下所示：



fast paxos流程是在选举过程中，某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。其流程图如下所示：

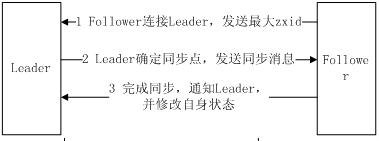


### 同步流程

选完leader以后，zk就进入状态同步过程。

1. leader等待server连接；
2. Follower连接leader，将最大的zxid发送给leader；
3. Leader根据follower的zxid确定同步点；
4. 完成同步后通知follower 已经成为uptodate状态；
5. Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。

流程图如下所示：



### 工作流程

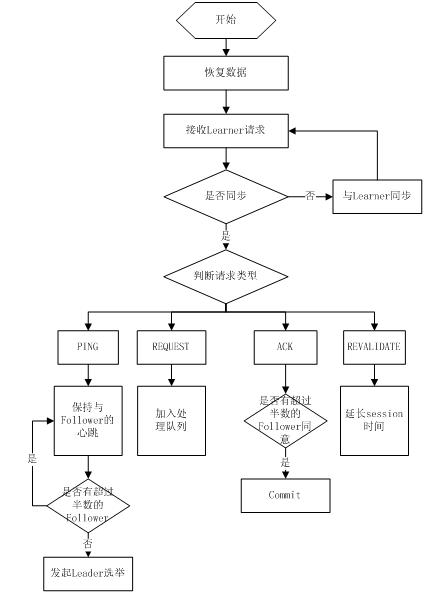
#### ****Leader工作流程****

Leader主要有三个功能：

1. 恢复数据；
2. 维持与Learner的心跳，接收Learner请求并判断Learner的请求消息类型；
3. Learner的消息类型主要有PING消息、REQUEST消息、ACK消息、REVALIDATE消息，根据不同的消息类型，进行不同的处理。

PING消息是指Learner的心跳信息；REQUEST消息是Follower发送的提议信息，包括写请求及同步请求；ACK消息是Follower的对提议的回复，超过半数的Follower通过，则commit该提议；REVALIDATE消息是用来延长SESSION有效时间。

Leader的工作流程简图如下所示，在实际实现中，流程要比下图复杂得多，启动了三个线程来实现功能。



#### ****Follower工作流程****

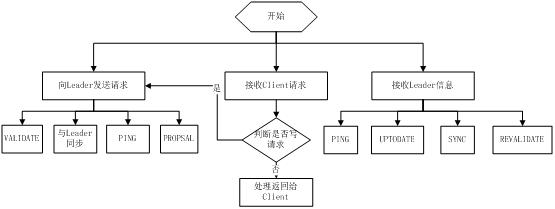
Follower主要有四个功能：

1. 向Leader发送请求（PING消息、REQUEST消息、ACK消息、REVALIDATE消息）；
2. 接收Leader消息并进行处理；
3. 接收Client的请求，如果为写请求，发送给Leader进行投票；
4. 返回Client结果。

Follower的消息循环处理如下几种来自Leader的消息：

1. **PING**消息： 心跳消息；
2. **PROPOSAL**消息：Leader发起的提案，要求Follower投票；
3. **COMMIT**消息：服务器端最新一次提案的信息；
4. **UPTODATE**消息：表明同步完成；
5. **REVALIDATE**消息：根据Leader的REVALIDATE结果，关闭待revalidate的session还是允许其接受消息；
6. **SYNC**消息：返回SYNC结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

Follower的工作流程简图如下所示，在实际实现中，Follower是通过5个线程来实现功能的。



对于observer的流程不再叙述，observer流程和Follower的唯一不同的地方就是observer不会参加leader发起的投票。

主流应用场景：

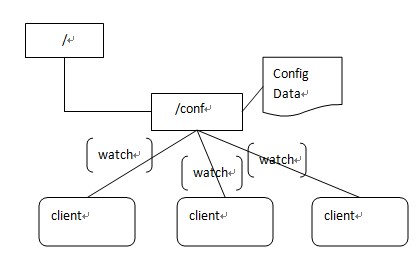
**Zookeeper的主流应用场景实现思路（除去官方示例）**

**(1)配置管理**

集中式的配置管理在应用集群中是非常常见的，一般商业公司内部都会实现一套集中的配置管理中心，应对不同的应用集群对于共享各自配置的需求，并且在配置变更时能够通知到集群中的每一个机器。

Zookeeper很容易实现这种集中式的配置管理，比如将APP1的所有配置配置到/APP1 znode下，APP1所有机器一启动就对/APP1这个节点进行监控(zk.exist("/APP1",true)),并且实现回调方法Watcher，那么在zookeeper上/APP1 znode节点下数据发生变化的时候，每个机器都会收到通知，Watcher方法将会被执行，那么应用再取下数据即可(zk.getData("/APP1",false,null));

以上这个例子只是简单的粗颗粒度配置监控，细颗粒度的数据可以进行分层级监控，这一切都是可以设计和控制的。



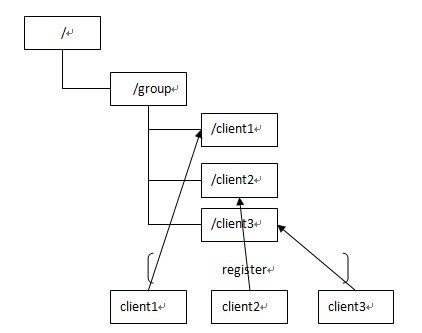
**(2)集群管理**

应用集群中，我们常常需要让每一个机器知道集群中（或依赖的其他某一个集群）哪些机器是活着的，并且在集群机器因为宕机，网络断链等原因能够不在人工介入的情况下迅速通知到每一个机器。

Zookeeper同样很容易实现这个功能，比如我在zookeeper服务器端有一个znode叫/APP1SERVERS,那么集群中每一个机器启动的时候都去这个节点下创建一个EPHEMERAL类型的节点，比如server1创建/APP1SERVERS/SERVER1(可以使用ip,保证不重复)，server2创建/APP1SERVERS/SERVER2，然后SERVER1和SERVER2都watch /APP1SERVERS这个父节点，那么也就是这个父节点下数据或者子节点变化都会通知对该节点进行watch的客户端。因为EPHEMERAL类型节点有一个很重要的特性，就是客户端和服务器端连接断掉或者session过期就会使节点消失，那么在某一个机器挂掉或者断链的时候，其对应的节点就会消失，然后集群中所有对/APP1SERVERS进行watch的客户端都会收到通知，然后取得最新列表即可。

另外有一个应用场景就是集群选master,一旦master挂掉能够马上能从slave中选出一个master,实现步骤和前者一样，只是机器在启动的时候在APP1SERVERS创建的节点类型变为EPHEMERAL\_SEQUENTIAL类型，这样每个节点会自动被编号

我们默认规定编号最小的为master,所以当我们对/APP1SERVERS节点做监控的时候，得到服务器列表，只要所有集群机器逻辑认为最小编号节点为master，那么master就被选出，而这个master宕机的时候，相应的znode会消失，然后新的服务器列表就被推送到客户端，然后每个节点逻辑认为最小编号节点为master，这样就做到动态master选举。



## 安装和配置详解

本文介绍的 Zookeeper 是以 3.2.2 这个稳定版本为基础，最新的版本可以通过官网 <http://hadoop.apache.org/zookeeper/>来获取，Zookeeper 的安装非常简单，下面将从单机模式和集群模式两个方面介绍 Zookeeper 的安装和配置。

### 单机模式

单 机安装非常简单，只要获取到 Zookeeper 的压缩包并解压到某个目录如：/home/zookeeper-3.2.2 下，Zookeeper 的启动脚本在 bin 目录下，Linux 下的启动脚本是 zkServer.sh，在 3.2.2 这个版本 Zookeeper 没有提供 windows 下的启动脚本，所以要想在 windows 下启动 Zookeeper 要自己手工写一个，如清单 1 所示：

**清单 1. Windows 下 Zookeeper 启动脚本**

setlocal

set ZOOCFGDIR=%~dp0%..\conf

set ZOO\_LOG\_DIR=%~dp0%..

set ZOO\_LOG4J\_PROP=INFO,CONSOLE

set CLASSPATH=%ZOOCFGDIR%

set CLASSPATH=%~dp0..\\*;%~dp0..\lib\\*;%CLASSPATH%

set CLASSPATH=%~dp0..\build\classes;%~dp0..\build\lib\\*;%CLASSPATH%

set ZOOCFG=%ZOOCFGDIR%\zoo.cfg

set ZOOMAIN=org.apache.zookeeper.server.ZooKeeperServerMain

java "-Dzookeeper.log.dir=%ZOO\_LOG\_DIR%" "-Dzookeeper.root.logger=%ZOO\_LOG4J\_PROP%"

-cp "%CLASSPATH%" %ZOOMAIN% "%ZOOCFG%" %\*

endlocal

在你执行启动脚本之前，还有几个基本的配置项需要配置一下，Zookeeper 的配置文件在 conf 目录下，这个目录下有 zoo\_sample.cfg 和 log4j.properties，你需要做的就是将 zoo\_sample.cfg 改名为 zoo.cfg，因为 Zookeeper 在启动时会找这个文件作为默认配置文件。下面详细介绍一下，这个配置文件中各个配置项的意义。

tickTime=2000

dataDir=D:/devtools/zookeeper-3.2.2/build

clientPort=2181

* 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。
* 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：这个端口就是客户端连接 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。

当这些配置项配置好后，你现在就可以启动 Zookeeper 了，启动后要检查 Zookeeper 是否已经在服务，可以通过 netstat – ano 命令查看是否有你配置的 clientPort 端口号在监听服务。

### 集群模式

Zookeeper 不仅可以单机提供服务，同时也支持多机组成集群来提供服务。实际上 Zookeeper 还支持另外一种伪集群的方式，也就是可以在一台物理机上运行多个 Zookeeper 实例，下面将介绍集群模式的安装和配置。

Zookeeper 的集群模式的安装和配置也不是很复杂，所要做的就是增加几个配置项。集群模式除了上面的三个配置项还要增加下面几个配置项：

initLimit=5

syncLimit=2

server.1=192.168.211.1:2888:3888

server.2=192.168.211.2:2888:3888

* 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 5\*2000=10 秒
* 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 2\*2000=4 秒
* server.A=B：C：D：其中 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；C 表示的是这个服务器与集群中的 Leader 服务器交换信息的端口；D 表示的是万一集群中的 Leader 服务器挂了，需要一个端口来重新进行选举，选出一个新的 Leader，而这个端口就是用来执行选举时服务器相互通信的端口。如果是伪集群的配置方式，由于 B 都是一样，所以不同的 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。

除了修改 zoo.cfg 配置文件，集群模式下还要配置一个文件 myid，这个文件在 dataDir 目录下，这个文件里面就有一个数据就是 A 的值，Zookeeper 启动时会读取这个文件，拿到里面的数据与 zoo.cfg 里面的配置信息比较从而判断到底是那个 server。

### 数据模型

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，如图 1 所示：

**图 1 Zookeeper 数据结构**



Zookeeper 这种数据结构有如下这些特点：

* 每个子目录项如 NameService 都被称作为 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
* 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
* 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中可以存储多份数据
* 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
* 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
* 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，后面在典型的应用场景中会有实例介绍

[回页首](http://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/#ibm-pcon)

## 如何使用

Zookeeper 作为一个分布式的服务框架，主要用来解决分布式集群中应用系统的一致性问题，它能提供基于类似于文件系统的目录节点树方式的数据存储，但是 Zookeeper 并不是用来专门存储数据的，它的作用主要是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，后面将 会详细介绍 Zookeeper 能够解决的一些典型问题，这里先介绍一下，Zookeeper 的操作接口和简单使用示例。

### 常用接口列表

客户端要连接 Zookeeper 服务器可以通过创建 org.apache.zookeeper. ZooKeeper 的一个实例对象，然后调用这个类提供的接口来和服务器交互。

前 面说了 ZooKeeper 主要是用来维护和监控一个目录节点树中存储的数据的状态，所有我们能够操作 ZooKeeper 的也和操作目录节点树大体一样，如创建一个目录节点，给某个目录节点设置数据，获取某个目录节点的所有子目录节点，给某个目录节点设置权限和监控这个目录 节点的状态变化。

这些接口如下表所示：

**表 1 org.apache.zookeeper. ZooKeeper 方法列表**

| **方法名** | **方法功能描述** |
| --- | --- |
| [String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true)[create](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#create%28java.lang.String,%20byte[],%20java.util.List,%20org.apache.zookeeper.CreateMode%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, byte[] data,[List](http://java.sun.com/javase/6/docs/api/java/util/List.html?is-external=true)<[ACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/ACL.html)> acl,[CreateMode](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/CreateMode.html) createMode) | 创建一个给定的目录节点 path, 并给它设置数据，[CreateMode](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/CreateMode.html) 标 识有四种形式的目录节点，分别是 PERSISTENT：持久化目录节点，这个目录节点存储的数据不会丢失；PERSISTENT\_SEQUENTIAL：顺序自动编号的目录节点，这种目 录节点会根据当前已近存在的节点数自动加 1，然后返回给客户端已经成功创建的目录节点名；EPHEMERAL：临时目录节点，一旦创建这个节点的客户端与服务器端口也就是 session 超时，这种节点会被自动删除；EPHEMERAL\_SEQUENTIAL：临时自动编号节点 |
| [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html)[exists](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#exists%28java.lang.String,%20boolean%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch) | 判断某个 path 是否存在，并设置是否监控这个目录节点，这里的 watcher 是在创建 ZooKeeper 实例时指定的 watcher，[exists](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#exists%28java.lang.String,%20boolean%29)方法还有一个重载方法，可以指定特定的 watcher |
| [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html)[exists](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#exists%28java.lang.String,%20org.apache.zookeeper.Watcher%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path,[Watcher](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/Watcher.html) watcher) | 重载方法，这里给某个目录节点设置特定的 watcher，Watcher 在 ZooKeeper 是一个核心功能，Watcher 可以监控目录节点的数据变化以及子目录的变化，一旦这些状态发生变化，服务器就会通知所有设置在这个目录节点上的 Watcher，从而每个客户端都很快知道它所关注的目录节点的状态发生变化，而做出相应的反应 |
| void[delete](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#delete%28java.lang.String,%20int%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, int version) | 删除 path 对应的目录节点，version 为 -1 可以匹配任何版本，也就删除了这个目录节点所有数据 |
| [List](http://java.sun.com/javase/6/docs/api/java/util/List.html?is-external=true)<[String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true)>[getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch) | 获取指定 path 下的所有子目录节点，同样 [getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)方法也有一个重载方法可以设置特定的 watcher 监控子节点的状态 |
| [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html)[setData](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#setData%28java.lang.String,%20byte[],%20int%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, byte[] data, int version) | 给 path 设置数据，可以指定这个数据的版本号，如果 version 为 -1 怎可以匹配任何版本 |
| byte[][getData](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getData%28java.lang.String,%20boolean,%20org.apache.zookeeper.data.Stat%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch,[Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html) stat) | 获取这个 path 对应的目录节点存储的数据，数据的版本等信息可以通过 stat 来指定，同时还可以设置是否监控这个目录节点数据的状态 |
| void[addAuthInfo](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#addAuthInfo%28java.lang.String,%20byte[]%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) scheme, byte[] auth) | 客户端将自己的授权信息提交给服务器，服务器将根据这个授权信息验证客户端的访问权限。 |
| [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html)[setACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#setACL%28java.lang.String,%20java.util.List,%20int%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path,[List](http://java.sun.com/javase/6/docs/api/java/util/List.html?is-external=true)<[ACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/ACL.html)> acl, int version) | 给某个目录节点重新设置访问权限，需要注意的是 Zookeeper 中的目录节点权限不具有传递性，父目录节点的权限不能传递给子目录节点。目录节点 ACL 由两部分组成：perms 和 id。 Perms 有 ALL、READ、WRITE、CREATE、DELETE、ADMIN 几种  而 id 标识了访问目录节点的身份列表，默认情况下有以下两种： ANYONE\_ID\_UNSAFE = new Id("world", "anyone") 和 AUTH\_IDS = new Id("auth", "") 分别表示任何人都可以访问和创建者拥有访问权限。 |
| [List](http://java.sun.com/javase/6/docs/api/java/util/List.html?is-external=true)<[ACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/ACL.html)>[getACL](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getACL%28java.lang.String,%20org.apache.zookeeper.data.Stat%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, [Stat](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/data/Stat.html) stat) | 获取某个目录节点的访问权限列表 |

除了以上这些上表中列出的方法之外还有一些重载方法，如都提供了一个回调类的重载方法以及可以设置特定 Watcher 的重载方法，具体的方法可以参考 org.apache.zookeeper. ZooKeeper 类的 API 说明。

### 基本操作

下面给出基本的操作 ZooKeeper 的示例代码，这样你就能对 ZooKeeper 有直观的认识了。下面的清单包括了创建与 ZooKeeper 服务器的连接以及最基本的数据操作：

**清单 2. ZooKeeper 基本的操作示例**

// 创建一个与服务器的连接

ZooKeeper zk = new ZooKeeper("localhost:" + CLIENT\_PORT,

ClientBase.CONNECTION\_TIMEOUT, new Watcher() {

// 监控所有被触发的事件

public void process(WatchedEvent event) {

System.out.println("已经触发了" + event.getType() + "事件！");

}

});

// 创建一个目录节点

zk.create("/testRootPath", "testRootData".getBytes(), Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

// 创建一个子目录节点

zk.create("/testRootPath/testChildPathOne", "testChildDataOne".getBytes(),

Ids.OPEN\_ACL\_UNSAFE,CreateMode.PERSISTENT);

System.out.println(new String(zk.getData("/testRootPath",false,null)));

// 取出子目录节点列表

System.out.println(zk.getChildren("/testRootPath",true));

// 修改子目录节点数据

zk.setData("/testRootPath/testChildPathOne","modifyChildDataOne".getBytes(),-1);

System.out.println("目录节点状态：["+zk.exists("/testRootPath",true)+"]");

// 创建另外一个子目录节点

zk.create("/testRootPath/testChildPathTwo", "testChildDataTwo".getBytes(),

Ids.OPEN\_ACL\_UNSAFE,CreateMode.PERSISTENT);

System.out.println(new String(zk.getData("/testRootPath/testChildPathTwo",true,null)));

// 删除子目录节点

zk.delete("/testRootPath/testChildPathTwo",-1);

zk.delete("/testRootPath/testChildPathOne",-1);

// 删除父目录节点

zk.delete("/testRootPath",-1);

// 关闭连接

zk.close();

输出的结果如下：

已经触发了 None 事件！

testRootData

[testChildPathOne]

目录节点状态：[5,5,1281804532336,1281804532336,0,1,0,0,12,1,6]

已经触发了 NodeChildrenChanged 事件！

testChildDataTwo

已经触发了 NodeDeleted 事件！

已经触发了 NodeDeleted 事件！

当对目录节点监控状态打开时，一旦目录节点的状态发生变化，Watcher 对象的 process 方法就会被调用。

[回页首](http://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/#ibm-pcon)

## 常用概念简介

### Zookeeper 监视（Watches） 简介

Zookeeper C API 的声明和描述在 include/zookeeper.h 中可以找到，另外大部分的 Zookeeper C API 常量、结构体声明也在 zookeeper.h 中，如果如果你在使用 C API 是遇到不明白的地方，最好看看 zookeeper.h，或者自己使用 doxygen 生成 Zookeeper C API 的帮助文档。

Zookeeper 中最有特色且最不容易理解的是监视(Watches)。Zookeeper 所有的读操作——**getData()**, **getChildren()**, 和 **exists()**都 可以设置监视(watch)，监视事件可以理解为一次性的触发器， 官方定义如下： a watch event is one-time trigger, sent to the client that set the watch, which occurs when the data for which the watch was set changes。对此需要作出如下理解：

* **（一次性触发）One-time trigger**  
  当设置监视的数据发生改变时，该监视事件会被发送到客户端，例如，如果客户端调用了 getData("/znode1", true) 并且稍后 /znode1 节点上的数据发生了改变或者被删除了，客户端将会获取到 /znode1 发生变化的监视事件，而如果 /znode1 再一次发生了变化，除非客户端再次对 /znode1 设置监视，否则客户端不会收到事件通知。
* **（发送至客户端）Sent to the client**  
  Zookeeper 客户端和服务端是通过 socket 进行通信的，由于网络存在故障，所以监视事件很有可能不会成功地到达客户端，监视事件是异步发送至监视者的，Zookeeper 本身提供了保序性(ordering guarantee)：即客户端只有首先看到了监视事件后，才会感知到它所设置监视的 znode 发生了变化(a client will never see a change for which it has set a watch until it first sees the watch event). 网络延迟或者其他因素可能导致不同的客户端在不同的时刻感知某一监视事件，但是不同的客户端所看到的一切具有一致的顺序。
* **（被设置 watch 的数据）The data for which the watch was set**  
  这意味着 znode 节点本身具有不同的改变方式。你也可以想象 Zookeeper 维护了两条监视链表：数据监视和子节点监视(data watches and child watches) getData() and exists() 设置数据监视，getChildren() 设置子节点监视。 或者，你也可以想象 Zookeeper 设置的不同监视返回不同的数据，getData() 和 exists() 返回 znode 节点的相关信息，而 getChildren() 返回子节点列表。因此， setData() 会触发设置在某一节点上所设置的数据监视(假定数据设置成功)，而一次成功的 create() 操作则会出发当前节点上所设置的数据监视以及父节点的子节点监视。一次成功的 delete() 操作将会触发当前节点的数据监视和子节点监视事件，同时也会触发该节点父节点的child watch。

Zookeeper 中的监视是轻量级的，因此容易设置、维护和分发。当客户端与 Zookeeper 服务器端失去联系时，客户端并不会收到监视事件的通知，只有当客户端重新连接后，若在必要的情况下，以前注册的监视会重新被注册并触发，对于开发人员来说 这通常是透明的。只有一种情况会导致监视事件的丢失，即：通过 exists() 设置了某个 znode 节点的监视，但是如果某个客户端在此 znode 节点被创建和删除的时间间隔内与 zookeeper 服务器失去了联系，该客户端即使稍后重新连接 zookeeper服务器后也得不到事件通知。

### Zookeeper C API 常量与部分结构(struct)介绍

#### ****与 ACL 相关的结构与常量：****

struct Id 结构为：

struct Id {     char \* scheme;     char \* id; };

struct ACL 结构为：

struct ACL {     int32\_t perms;     struct Id id; };

struct ACL\_vector 结构为：

struct ACL\_vector {     int32\_t count;     struct ACL \*data; };

与 znode 访问权限有关的常量

* const int ZOO\_PERM\_READ; //允许客户端读取 znode 节点的值以及子节点列表。
* const int ZOO\_PERM\_WRITE;// 允许客户端设置 znode 节点的值。
* const int ZOO\_PERM\_CREATE; //允许客户端在该 znode 节点下创建子节点。
* const int ZOO\_PERM\_DELETE;//允许客户端删除子节点。
* const int ZOO\_PERM\_ADMIN; //允许客户端执行 set\_acl()。
* const int ZOO\_PERM\_ALL;//允许客户端执行所有操作，等价与上述所有标志的或(OR) 。

与 ACL IDs 相关的常量

* struct Id ZOO\_ANYONE\_ID\_UNSAFE; //(‘world’,’anyone’)
* struct Id ZOO\_AUTH\_IDS;// (‘auth’,’’)

三种标准的 ACL

* struct ACL\_vector ZOO\_OPEN\_ACL\_UNSAFE; //(ZOO\_PERM\_ALL,ZOO\_ANYONE\_ID\_UNSAFE)
* struct ACL\_vector ZOO\_READ\_ACL\_UNSAFE;// (ZOO\_PERM\_READ, ZOO\_ANYONE\_ID\_UNSAFE)
* struct ACL\_vector ZOO\_CREATOR\_ALL\_ACL; //(ZOO\_PERM\_ALL,ZOO\_AUTH\_IDS)

#### ****与 Interest 相关的常量：ZOOKEEPER\_WRITE, ZOOKEEPER\_READ****

这 两个常量用于标识感兴趣的事件并通知 zookeeper 发生了哪些事件。Interest 常量可以进行组合或（OR）来标识多种兴趣(multiple interests: write, read)，这两个常量一般用于 zookeeper\_interest() 和 zookeeper\_process()两个函数中。

#### ****与节点创建相关的常量：ZOO\_EPHEMERAL, ZOO\_SEQUENCE****

zoo\_create 函数标志，**ZOO\_EPHEMERAL** 用来标识创建临时节点，**ZOO\_SEQUENCE** 用来标识节点命名具有递增的后缀序号(一般是节点名称后填充 10 位字符的序号，如 /xyz0000000000, /xyz0000000001, /xyz0000000002, ...)，同样地，**ZOO\_EPHEMERAL**, **ZOO\_SEQUENCE**可以组合。

#### ****与连接状态 Stat 相关的常量****

以下常量均与 Zookeeper 连接状态有关，他们通常用作监视器回调函数的参数。

|  |  |
| --- | --- |
| ZOOAPI const int | **ZOO\_EXPIRED\_SESSION\_STATE** |
| ZOOAPI const int | **ZOO\_AUTH\_FAILED\_STATE** |
| ZOOAPI const int | **ZOO\_CONNECTING\_STATE** |
| ZOOAPI const int | **ZOO\_ASSOCIATING\_STATE** |
| ZOOAPI const int | **ZOO\_CONNECTED\_STATE** |

#### ****与监视类型(Watch Types)相关的常量****

以下常量标识监视事件的类型，他们通常用作监视器回调函数的第一个参数。

* ZOO\_CREATED\_EVENT; // 节点被创建(此前该节点不存在)，通过 zoo\_exists() 设置监视。
* ZOO\_DELETED\_EVENT; // 节点被删除，通过 zoo\_exists() 和 zoo\_get() 设置监视。
* ZOO\_CHANGED\_EVENT; // 节点发生变化，通过 zoo\_exists() 和 zoo\_get() 设置监视。
* ZOO\_CHILD\_EVENT; // 子节点事件，通过zoo\_get\_children() 和 zoo\_get\_children2()设置监视。
* ZOO\_SESSION\_EVENT; // 会话丢失
* ZOO\_NOTWATCHING\_EVENT; // 监视被移除。

### Zookeeper C API 错误码介绍 ZOO\_ERRORS

|  |  |
| --- | --- |
| ZOK | 正常返回 |
| ZSYSTEMERROR | 系统或服务器端错误(System and server-side errors)，服务器不会抛出该错误，该错误也只是用来标识错误范围的，即大于该错误值，且小于 ZAPIERROR 都是系统错误。 |
| ZRUNTIMEINCONSISTENCY | 运行时非一致性错误。 |
| ZDATAINCONSISTENCY | 数据非一致性错误。 |
| ZCONNECTIONLOSS | Zookeeper 客户端与服务器端失去连接 |
| ZMARSHALLINGERROR | 在 marshalling 和 unmarshalling 数据时出现错误(Error while marshalling or unmarshalling data) |
| ZUNIMPLEMENTED | 该操作未实现(Operation is unimplemented) |
| ZOPERATIONTIMEOUT | 该操作超时(Operation timeout) |
| ZBADARGUMENTS | 非法参数错误(Invalid arguments) |
| ZINVALIDSTATE | 非法句柄状态(Invliad zhandle state) |
| ZAPIERROR | API 错误(API errors)，服务器不会抛出该错误，该错误也只是用来标识错误范围的，错误值大于该值的标识 API 错误，而小于该值的标识 ZSYSTEMERROR。 |
| ZNONODE | 节点不存在(Node does not exist) |
| ZNOAUTH | 没有经过授权(Not authenticated) |
| ZBADVERSION | 版本冲突(Version conflict) |
| ZNOCHILDRENFOREPHEMERALS | 临时节点不能拥有子节点(Ephemeral nodes may not have children) |
| ZNODEEXISTS | 节点已经存在(The node already exists) |
| ZNOTEMPTY | 该节点具有自身的子节点(The node has children) |
| ZSESSIONEXPIRED | 会话过期(The session has been expired by the server) |
| ZINVALIDCALLBACK | 非法的回调函数(Invalid callback specified) |
| ZINVALIDACL | 非法的ACL(Invalid ACL specified) |
| ZAUTHFAILED | 客户端授权失败(Client authentication failed) |
| ZCLOSING | Zookeeper 连接关闭(ZooKeeper is closing) |
| ZNOTHING | 并非错误，客户端不需要处理服务器的响应(not error, no server responses to process) |
| ZSESSIONMOVED | 会话转移至其他服务器，所以操作被忽略(session moved to another server, so operation is ignored) |

### Watch事件类型：

* ZOO\_CREATED\_EVENT：节点创建事件，需要watch一个不存在的节点，当节点被创建时触发，此watch通过zoo\_exists()设置
* ZOO\_DELETED\_EVENT：节点删除事件，此watch通过zoo\_exists()或zoo\_get()设置
* ZOO\_CHANGED\_EVENT：节点数据改变事件，此watch通过zoo\_exists()或zoo\_get()设置
* ZOO\_CHILD\_EVENT：子节点列表改变事件，此watch通过zoo\_get\_children()或zoo\_get\_children2()设置
* ZOO\_SESSION\_EVENT：会话失效事件，客户端与服务端断开或重连时触发
* ZOO\_NOTWATCHING\_EVENT：watch移除事件，服务端出于某些原因不再为客户端watch节点时触发

## ZooKeeper 典型的应用场景

Zookeeper 从设计模式角度来看，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，然后接受观察者的注册，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，从而实现集群中类似 Master/Slave 管理模式，关于 Zookeeper 的详细架构等内部细节可以阅读 Zookeeper 的源码

下面详细介绍这些典型的应用场景，也就是 Zookeeper 到底能帮我们解决那些问题？下面将给出答案。

### 统一命名服务（Name Service）

分 布式应用中，通常需要有一套完整的命名规则，既能够产生唯一的名称又便于人识别和记住，通常情况下用树形的名称结构是一个理想的选择，树形的名称结构是一 个有层次的目录结构，既对人友好又不会重复。说到这里你可能想到了 JNDI，没错 Zookeeper 的 Name Service 与 JNDI 能够完成的功能是差不多的，它们都是将有层次的目录结构关联到一定资源上，但是 Zookeeper 的 Name Service 更加是广泛意义上的关联，也许你并不需要将名称关联到特定资源上，你可能只需要一个不会重复名称，就像数据库中产生一个唯一的数字主键一样。

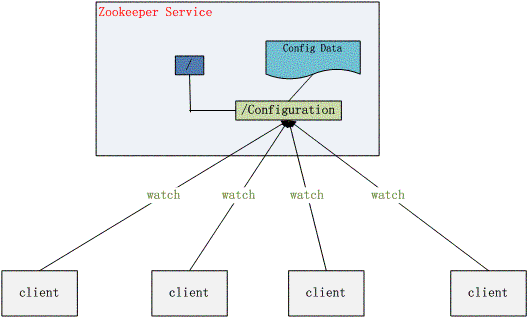
Name Service 已经是 Zookeeper 内置的功能，你只要调用 Zookeeper 的 API 就能实现。如调用 create 接口就可以很容易创建一个目录节点。

### 配置管理（Configuration Management）

配置的管理在分布式应用环境中很常见，例如同一个应用系统需要多台 PC Server 运行，但是它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的配置项，那么就必须同时修改每台运行这个应用系统的 PC Server，这样非常麻烦而且容易出错。

像 这样的配置信息完全可以交给 Zookeeper 来管理，将配置信息保存在 Zookeeper 的某个目录节点中，然后将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。

**图 2. 配置管理结构图**



### 集群管理（Group Membership）

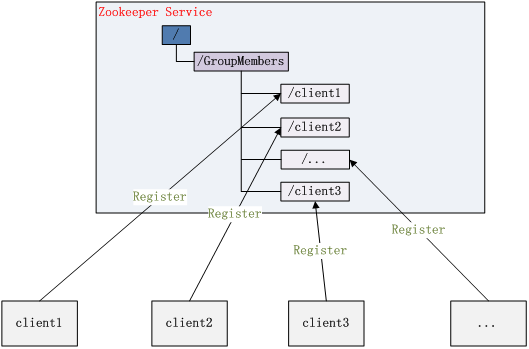
Zookeeper 能够很容易的实现集群管理的功能，如有多台 Server 组成一个服务集群，那么必须要一个“总管”知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，集群中其它集群必须知道，从而做出调整重新分配服 务策略。同样当增加集群的服务能力时，就会增加一台或多台 Server，同样也必须让“总管”知道。

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 Election。

它们的实现方式都是在 Zookeeper 上创建一个 EPHEMERAL 类型的目录节点，然后每个 Server 在它们创建目录节点的父目录节点上调用[getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch) 方法并设置 watch 为 true，由于是 EPHEMERAL 目录节点，当创建它的 Server 死去，这个目录节点也随之被删除，所以 Children 将会变化，这时 [getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)上的 Watch 将会被调用，所以其它 Server 就知道已经有某台 Server 死去了。新增 Server 也是同样的原理。

Zookeeper 如何实现 Leader Election，也就是选出一个 Master Server。和前面的一样每台 Server 创建一个 EPHEMERAL 目录节点，不同的是它还是一个 SEQUENTIAL 目录节点，所以它是个 EPHEMERAL\_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL\_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号，我们可以选择当前是最小编号的 Server 为 Master，假如这个最小编号的 Server 死去，由于是 EPHEMERAL 节点，死去的 Server 对应的节点也被删除，所以当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免了传统意义上单 Master 容易出现单点故障的问题。

**图 3. 集群管理结构图**



这部分的示例代码如下，完整的代码请看附件：

**清单 3. Leader Election 关键代码**

void findLeader() throws InterruptedException {

byte[] leader = null;

try {

leader = zk.getData(root + "/leader", true, null);

} catch (Exception e) {

logger.error(e);

}

if (leader != null) {

following();

} else {

String newLeader = null;

try {

byte[] localhost = InetAddress.getLocalHost().getAddress();

newLeader = zk.create(root + "/leader", localhost,

ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

} catch (Exception e) {

logger.error(e);

}

if (newLeader != null) {

leading();

} else {

mutex.wait();

}

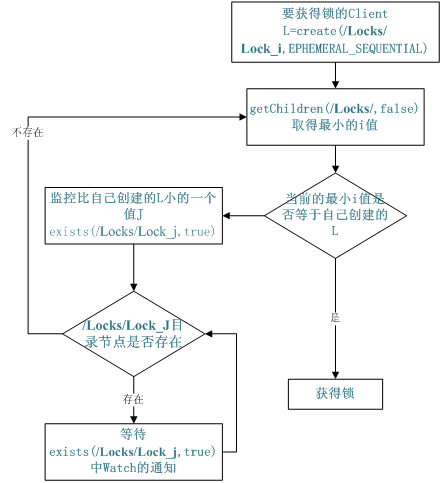
}

}

### 共享锁（Locks）

共享锁在同一个进程中很容易实现，但是在跨进程或者在不同 Server 之间就不好实现了。Zookeeper 却很容易实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL\_SEQUENTIAL 目录节点，然后调用 [getChildren](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#getChildren%28java.lang.String,%20boolean%29)方法获取当前的目录节点列表中最小的目录节点是不是就是自己创建的目录节点，如果正是自己创建的，那么它就获得了这个锁，如果不是那么它就调用 [exists](http://hadoop.apache.org/zookeeper/docs/r3.2.2/api/org/apache/zookeeper/ZooKeeper.html#exists%28java.lang.String,%20boolean%29)([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) path, boolean watch) 方法并监控 Zookeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，从而获得锁，释放锁很简单，只要删除前面它自己所创建的目录节点就行了。

**图 4. Zookeeper 实现 Locks 的流程图**



同步锁的实现代码如下，完整的代码请看附件：

**清单 4. 同步锁的关键代码**

void getLock() throws KeeperException, InterruptedException{

List<String> list = zk.getChildren(root, false);

String[] nodes = list.toArray(new String[list.size()]);

Arrays.sort(nodes);

if(myZnode.equals(root+"/"+nodes[0])){

doAction();

}

else{

waitForLock(nodes[0]);

}

}

void waitForLock(String lower) throws InterruptedException, KeeperException {

Stat stat = zk.exists(root + "/" + lower,true);

if(stat != null){

mutex.wait();

}

else{

getLock();

}

}

### 队列管理

Zookeeper 可以处理两种类型的队列：

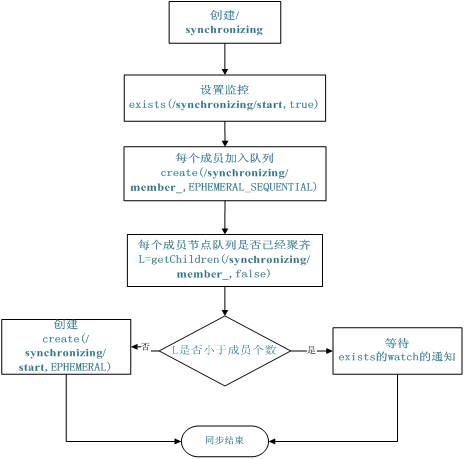
1. 当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种是同步队列。
2. 队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

同步队列用 Zookeeper 实现的实现思路如下：

创 建一个父目录 /synchronizing，每个成员都监控标志（Set Watch）位目录 /synchronizing/start 是否存在，然后每个成员都加入这个队列，加入队列的方式就是创建 /synchronizing/member\_i 的临时目录节点，然后每个成员获取 / synchronizing 目录的所有目录节点，也就是 member\_i。判断 i 的值是否已经是成员的个数，如果小于成员个数等待 /synchronizing/start 的出现，如果已经相等就创建 /synchronizing/start。

用下面的流程图更容易理解：

**图 5. 同步队列流程图**



同步队列的关键代码如下，完整的代码请看附件：

**清单 5. 同步队列**

void addQueue() throws KeeperException, InterruptedException{

zk.exists(root + "/start",true);

zk.create(root + "/" + name, new byte[0], Ids.OPEN\_ACL\_UNSAFE,

CreateMode.EPHEMERAL\_SEQUENTIAL);

synchronized (mutex) {

List<String> list = zk.getChildren(root, false);

if (list.size() < size) {

mutex.wait();

} else {

zk.create(root + "/start", new byte[0], Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

}

}

当队列没满是进入 wait()，然后会一直等待 Watch 的通知，Watch 的代码如下：

public void process(WatchedEvent event) {

if(event.getPath().equals(root + "/start") &&

event.getType() == Event.EventType.NodeCreated){

System.out.println("得到通知");

super.process(event);

doAction();

}

}

FIFO 队列用 Zookeeper 实现思路如下：

实现的思路也非常简单，就是在特定的目录下创建 SEQUENTIAL 类型的子目录 /queue\_i，这样就能保证所有成员加入队列时都是有编号的，出队列时通过 getChildren( ) 方法可以返回当前所有的队列中的元素，然后消费其中最小的一个，这样就能保证 FIFO。

下面是生产者和消费者这种队列形式的示例代码，完整的代码请看附件：

**清单 6. 生产者代码**

boolean produce(int i) throws KeeperException, InterruptedException{

ByteBuffer b = ByteBuffer.allocate(4);

byte[] value;

b.putInt(i);

value = b.array();

zk.create(root + "/element", value, ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT\_SEQUENTIAL);

return true;

}

**清单 7. 消费者代码**

int consume() throws KeeperException, InterruptedException{

int retvalue = -1;

Stat stat = null;

while (true) {

synchronized (mutex) {

List<String> list = zk.getChildren(root, true);

if (list.size() == 0) {

mutex.wait();

} else {

Integer min = new Integer(list.get(0).substring(7));

for(String s : list){

Integer tempValue = new Integer(s.substring(7));

if(tempValue < min) min = tempValue;

}

byte[] b = zk.getData(root + "/element" + min,false, stat);

zk.delete(root + "/element" + min, 0);

ByteBuffer buffer = ByteBuffer.wrap(b);

retvalue = buffer.getInt();

return retvalue;

}

}

}

}

## 总结

Zookeeper 作为 Hadoop 项目中的一个子项目，是 Hadoop 集群管理的一个必不可少的模块，它主要用来控制集群中的数据，如它管理 Hadoop 集群中的 NameNode，还有 Hbase 中 Master Election、Server 之间状态同步等。

本文介绍的 Zookeeper 的基本知识，以及介绍了几个典型的应用场景。这些都是 Zookeeper 的基本功能，最重要的是 Zoopkeeper 提供了一套很好的分布式集群管理的机制，就是它这种基于层次型的目录树的数据结构，并对树中的节点进行有效管理，从而可以设计出多种多样的分布式的数据管 理模型，而不仅仅局限于上面提到的几个常用应用场景。



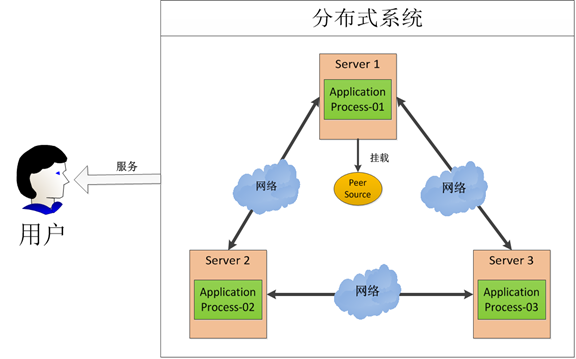
# 进阶

## ZooKeeper简单介绍

### 分布式协调技术

在给大家介绍ZooKeeper之前先来给大家介绍一种技术——分布式协调技术。那么什么是分布式协调技术？那么我来告诉大家，其实分布式协调技术 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，防止造成"脏数据"的后果。这时，有人可能会说这个简单，写一个调 度算法就轻松解决了。说这句话的人，可能对分布式系统不是很了解，所以才会出现这种误解。如果这些进程全部是跑在一台机上的话，相对来说确实就好办了，问 题就在于他是在一个分布式的环境下，这时问题又来了，那什么是分布式呢？这个一两句话我也说不清楚，但我给大家画了一张图希望能帮助大家理解这方面的内 容，如果觉得不对尽可拍砖，来咱们看一下这张图，如图1.1所示。

**图 1.1 分布式系统图**



给大家分析一下这张图，在这图中有三台机器，每台机器各跑一个应用程序。然后我们将这三台机器通过网络将其连接起来，构成一个系统来为用户提供服务，对用户来说这个系统的架构是透明的，他感觉不到我这个系统是一个什么样的架构。那么我们就可以把这种系统称作一个**分布式系统**。

那我们接下来再分析一下，在这个分布式系统中如何对进程进行调度，我假设在第一台机器上挂载了一个资源，然后这三个物理分布的进程都要竞争这个资源，但我们又不希望他们同时进行访问，这时候我们就需要一个**协调器**，来让他们有序的来访问这个资源。这个协调器就是我们经常提到的那个**锁**，比如说"进程-1"在使用该资源的时候，会先去获得锁，"进程1"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1"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。那么我们把这个分布式环境下的这个锁叫作**分布式锁**。这个分布式锁也就是我们**分布式协调技术**实现的核心内容，那么如何实现这个分布式呢，那就是我们后面要讲的内容。

### 分布式锁的实现

好我们知道，为了防止分布式系统中的多个进程之间相互干扰，我们需要一种分布式协调技术来对这些进程进行调度。而这个分布式协调技术的核心就是来实现这个分**布式锁**。那么这个锁怎么实现呢？这实现起来确实相对来说比较困难的。

#### ****面临的问题****

在看了图1.1所示的分布式环境之后，有人可能会感觉这不是很难。无非是将原来在同一台机器上对进程调度的原语，通过网络实现在分布式环境中。是的，表面上是可以这么说。但是问题就在网络这，在分布式系统中，所有在同一台机器上的假设都不存在：因为网络是不可靠的。

比如，在同一台机器上，你对一个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还先调用一些，B后调用，那么最后的结果是不是一定A的请求就先于B到达呢？ 这些在同一台机器上的种种假设，我们都要重新思考，我们还要思考这些问题给我们的设计和编码带来了哪些影响。还有，在分布式环境中为了提升可靠性，我们往 往会部署多套服务，但是如何在多套服务中达到一致性，这在同一台机器上多个进程之间的同步相对来说比较容易办到，但在分布式环境中确实一个大难题。

所以分布式协调远比在同一台机器上对多个进程的调度要难得多，而且如果为每一个分布式应用都开发一个独立的协调程序。一方面，协调程序的反复编写浪 费，且难以形成通用、伸缩性好的协调器。另一方面，协调程序开销比较大，会影响系统原有的性能。所以，急需一种高可靠、高可用的通用协调机制来用以协调分 布式应用。

#### ****分布式锁的实现者****

目前，在分布式协调技术方面做得比较好的就是Google的Chubby还有Apache的ZooKeeper他们都是分布式锁的实现者。有人会问 既然有了Chubby为什么还要弄一个ZooKeeper，难道Chubby做得不够好吗？不是这样的，主要是Chbby是非开源的，Google自家 用。后来雅虎模仿Chubby开发出了ZooKeeper，也实现了类似的分布式锁的功能，并且将ZooKeeper作为一种开源的程序捐献给了 Apache，那么这样就可以使用ZooKeeper所提供锁服务。而且在分布式领域久经考验，它的可靠性，可用性都是经过理论和实践的验证的。所以我们 在构建一些分布式系统的时候，就可以以这类系统为起点来构建我们的系统，这将节省不少成本，而且bug也 将更少。





### ZooKeeper概述

ZooKeeper是一种为分布式应用所设计的高可用、高性能且一致的开源协调服务，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。

***注意：ZooKeeper性能上的特点决定了它能够用在大型的、分布式的系统当中。从可靠性方面来说，它并不会因为一个节点的错误而崩溃。除此之外，它严格的序列访问控制意味着复杂的控制原语可以应用在客户端上。ZooKeeper在一致性、可用性、容错性的保证，也是ZooKeeper的成功之处，它获得的一切成功都与它采用的协议——Zab协议是密不可分的，这些内容将会在后面介绍。***

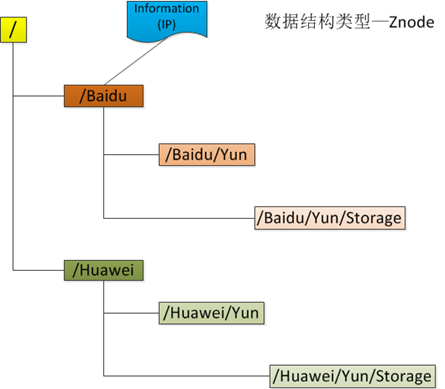
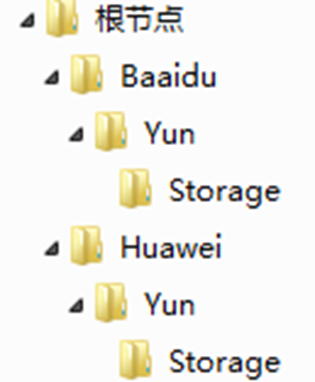
前面提到了那么多的服务，比如分布式锁、配置维护、组服务等，那它们是如何实现的呢，我相信这才是大家关心的东西。ZooKeeper在实现这些服务时，首先它设计一种新的数据结构——Znode，然后在该数据结构的基础上定义了一些原语，也就是一些关于该数据结构的一些操作。有了这些数据结构和原语还不够，因为我们的ZooKeeper是工作在一个分布式的环境下，我们的服务是通过消息以网络的形式发送给我们的分布式应用程序，所以还需要一个通知机制——Watcher机制。那么总结一下，ZooKeeper所提供的服务主要是通过：数据结构+原语+watcher机制，三个部分来实现的。那么我就从这三个方面，给大家介绍一下ZooKeeper。

### ZooKeeper数据模型

#### ****ZooKeeper数据模型Znode****

ZooKeeper拥有一个层次的命名空间，这个和标准的文件系统非常相似，如下图3.1 所示。

**图4.1 ZooKeeper数据模型与文件系统目录树**

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：

**(1) 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。

**(2)** **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。

**(3) 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。

**(4) 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。

**(5)** **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。

**(6) 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。

#### ****ZooKeeper中的时间****

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。

**①** **cZxid**： 是节点的创建时间所对应的Zxid格式时间戳。

**② 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：

**① version**：节点数据版本号

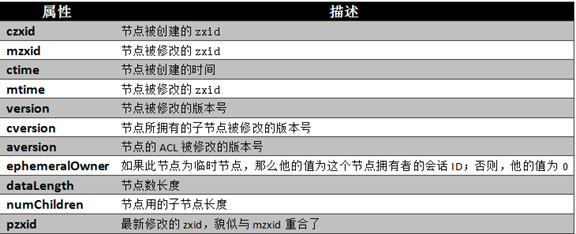
**② cversion**：子节点版本号

**③ aversion**：节点所拥有的ACL版本号

#### ****ZooKeeper节点属性****

通过前面的介绍，我们可以了解到，一个节点自身拥有表示其状态的许多重要属性，如下图所示。

**图 4.2 Znode节点属性结构**



### ZooKeeper服务中操作

在ZooKeeper中有9个基本操作，如下图所示：

**图 5.1 ZooKeeper类方法描述**



更新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。

更新ZooKeeper操作是非阻塞式的。因此客户端如果失去了一个更新(由于另一个进程在同时更新这个Znode)，他可以在不阻塞其他进程执行的情况下，选择重新尝试或进行其他操作。

尽管ZooKeeper可以被看做是一个文件系统，但是处于便利，摒弃了一些文件系统地操作原语。因为文件非常的小并且使整体读写的，所以不需要打开、关闭或是寻地的操作。

### Watch触发器

**(1) watch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。

**(2) watch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

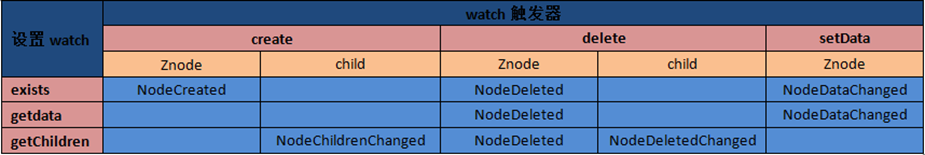
**①** 一个成功的**setData操作**将触发Znode的数据watch

**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

**(3) watch注册与处触发**

**图 6.1 watch设置操作及相应的触发器如图下图所示：**



**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。

**(4) 需要注意的几点**

**Zookeeper的watch实际上要处理两类事件：**

**① 连接状态事件**(type=None, path=null)

这类事件不需要注册，也不需要我们连续触发，我们只要处理就行了。

**② 节点事件**

节点的建立，删除，数据的修改。它是one time trigger，我们需要不停的注册触发，还可能发生事件丢失的情况。

上面2类事件都在Watch中处理，也就是重载的**process(Event event)**

**节点事件的触发，通过函数exists，getData或getChildren来处理这类函数，有双重作用：**

**① 注册触发事件**

**② 函数本身的功能**

函数的本身的功能又可以用异步的回调函数来实现,重载processResult()过程中处理函数本身的的功能。

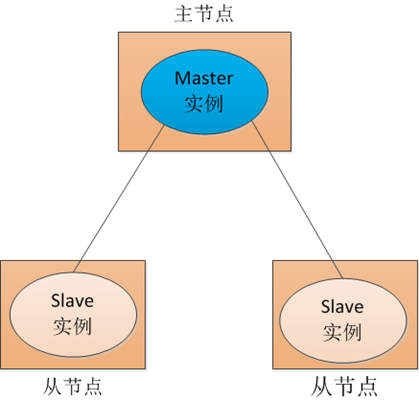
### ZooKeeper应用举例

为了方便大家理解ZooKeeper，在此就给大家举个例子，看看ZooKeeper是如何实现的他的服务的，我以ZooKeeper提供的基本服务**分布式锁**为例。

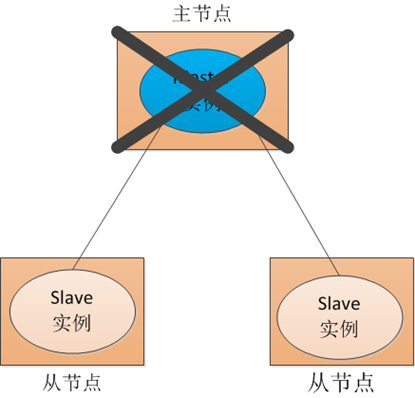
#### ****分布式锁应用场景****

在分布式锁服务中，有一种最典型应用场景，就是通过对集群进行**Master选举**，来解决分布式系统中的**单点故障**。什么是分布式系统中的单点故障：通常分布式系统采用主从模式，就是一个主控机连接多个处理节点。主节点负责分发任务，从节点负责处理任务，当我们的主节点发生故障时，那么整个系统就都瘫痪了，那么我们把这种故障叫作单点故障。如下图7.1和7.2所示：

**图 7.1 主从模式分布式系统**

****

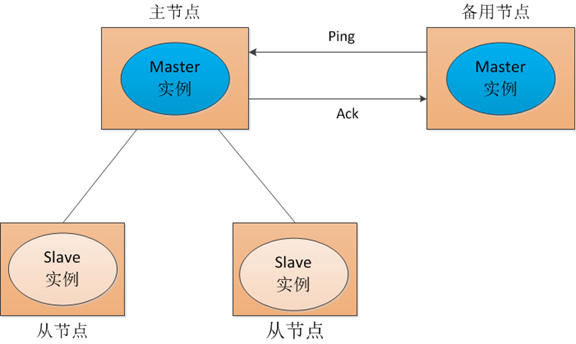
**图7.2 单点故障**



#### ****传统解决方案****

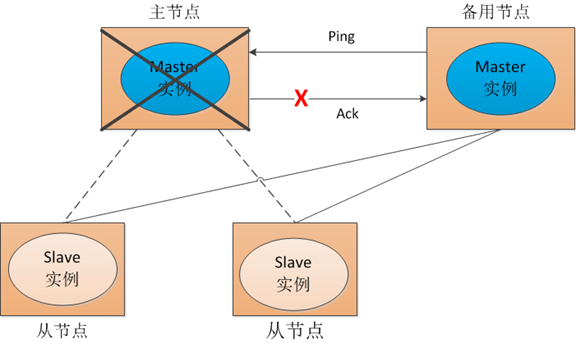
传统方式是采用一个备用节点，这个备用节点定期给当前主节点发送ping包，主节点收到ping包以后向备用节点发送回复Ack，当备用节点收到回复的时候就会认为当前主节点还活着，让他继续提供服务。如图7.3所示：

**图 7.3 传统解决方案**



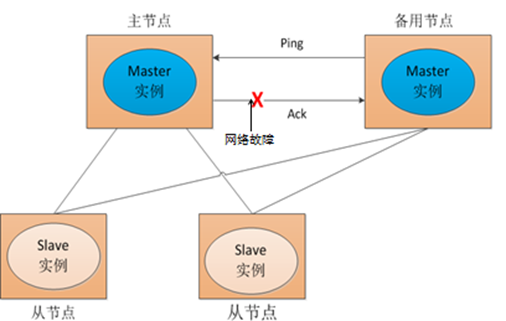
当主节点挂了，这时候备用节点收不到回复了，然后他就认为主节点挂了接替他成为主节点如下图7.4所示：

**图 7.4传统解决方案**



但是这种方式就是有一个隐患，就是网络问题，来看一网络问题会造成什么后果，如下图7.5所示：

**图 7.5 网络故障**



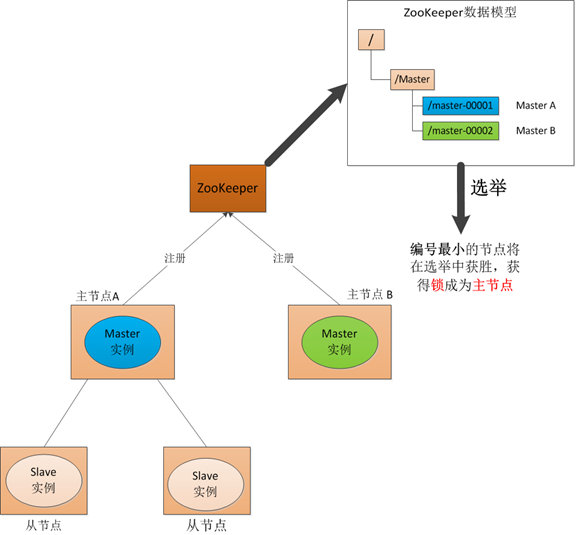
也就是说我们的主节点的并没有挂，只是在回复的时候网络发生故障，这样我们的备用节点同样收不到回复，就会认为主节点挂了，然后备用节点将他的Master实例启动起来，这样我们的分布式系统当中就有了两个主节点也就是---**双Master**， 出现Master以后我们的从节点就会将它所做的事一部分汇报给了主节点，一部分汇报给了从节点，这样服务就全乱了。为了防止出现这种情况，我们引入了 ZooKeeper，它虽然不能避免网络故障，但它能够保证每时每刻只有一个Master。我么来看一下ZooKeeper是如何实现的。

#### ****ZooKeeper解决方案****

**(1) Master启动**

在引入了Zookeeper以后我们启动了两个主节点，"主节点-A"和"主节点-B"他们启动以后，都向ZooKeeper去注册一个节点。我们 假设"主节点-A"锁注册地节点是"master-00001"，"主节点-B"注册的节点是"master-00002"，注册完以后进行选举，编号最 小的节点将在选举中获胜获得锁成为主节点，也就是我们的"主节点-A"将会获得锁成为主节点，然后"主节点-B"将被阻塞成为一个备用节点。那么，通过这 种方式就完成了对两个Master进程的调度。

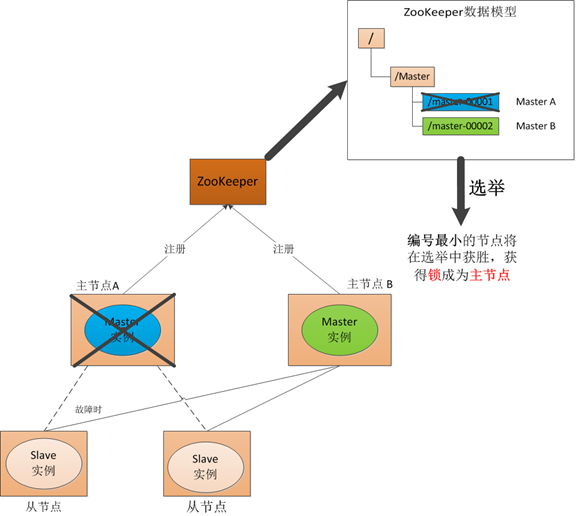
**图7.6 ZooKeeper Master选举**



**(2) Master故障**

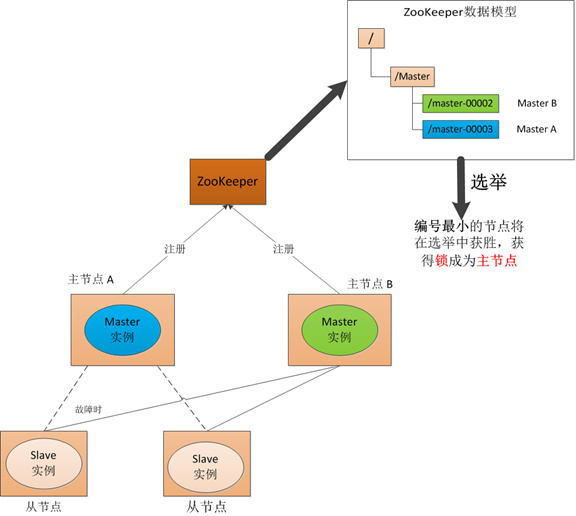
如果"主节点-A"挂了，这时候他所注册的节点将被自动删除，ZooKeeper会自动感知节点的变化，然后再次发出选举，这时候"主节点-B"将在选举中获胜，替代"主节点-A"成为主节点。

**图7.7 ZooKeeper Master选举**



**(3) Master 恢复**

**图7.8 ZooKeeper Master选举**



如果主节点恢复了，他会再次向ZooKeeper注册一个节点，这时候他注册的节点将会是"master-00003"，ZooKeeper会感知节点的变化再次发动选举，这时候"主节点-B"在选举中会再次获胜继续担任"主节点"，"主节点-A"会担任备用节点。

## ZooKeeper安装配置

### Zookeeper的搭建方式

Zookeeper安装方式有三种，**单机模式**和**集群模式**以及**伪集群模式**。

■ 单机模式：Zookeeper只运行在一台服务器上，适合测试环境；

■ 伪集群模式：就是在一台物理机上运行多个Zookeeper 实例；

■ 集群模式：Zookeeper运行于一个集群上，适合生产环境，这个计算机集群被称为一个“集合体”（ensemble）

Zookeeper通过复制来实现高可用性，只要集合体中半数以上的机器处于可用状态，它就能够保证服务继续。**为什么一定要超过半数呢**？这跟Zookeeper的复制策略有关：zookeeper确保对znode 树的每一个修改都会被复制到集合体中超过半数的机器上。

#### ****Zookeeper的单机模式搭建****

**下载**ZooKeeper：<http://pan.baidu.com/s/1pJlwbR9>

**解压**：tar -zxvf zookeeper-3.4.5.tar.gz **重命名**：mv zookeeper-3.4.5 zk

**配置文件**：在conf目录下删除zoo\_sample.cfg文件，创建一个配置文件zoo.cfg。

tickTime=2000

dataDir=/usr/local/zk/data

dataLogDir=/usr/local/zk/dataLog

clientPort=2181

**配置环境变量**：为了今后操作方便，我们需要对Zookeeper的环境变量进行配置，方法如下在/etc/profile文件中加入如下内容：

export ZOOKEEPER\_HOME=/usr/local/zk

export PATH=.:$HADOOP\_HOME/bin:$ZOOKEEPER\_HOME/bin:$JAVA\_HOME/bin:$PATH

**启动**ZooKeeper的Server：zkServer.sh start；**关闭**ZooKeeper的Server：zkServer.sh stop

#### ****Zookeeper的伪集群模式搭建****

Zookeeper不但可以在单机上运行单机模式Zookeeper，而且可以在单机模拟集群模式 Zookeeper的运行，也就是将不同节点运行在同一台机器。我们知道伪分布模式下Hadoop的操作和分布式模式下有着很大的不同，但是在集群为分布 式模式下对Zookeeper的操作却和集群模式下没有本质的区别。显然，集群伪分布式模式为我们体验Zookeeper和做一些尝试性的实验提供了很大 的便利。比如，我们在实验的时候，可以先使用少量数据在集群伪分布模式下进行测试。当测试可行的时候，再将数据移植到集群模式进行真实的数据实验。这样不 但保证了它的可行性，同时大大提高了实验的效率。这种搭建方式，比较简便，成本比较低，适合测试和学习，如果你的手头机器不足，就可以在一台机器上部署了 3个server。

##### ****注意事项****

在一台机器上部署了3个server，需要注意的是在集群为分布式模式下我们使用的每个配置文档模拟一台机器，也就是说单台机器及上运行多个Zookeeper实例。但是，必须保证每个配置文档的各个端口号不能冲突，除了clientPort不同之外，dataDir也不同。另外，还要在dataDir所对应的目录中创建myid文件来指定对应的Zookeeper服务器实例。

■ clientPort端口：如果在1台机器上部署多个server，那么每台机器都要不同的 clientPort，比如 server1是2181,server2是2182，server3是2183

■ dataDir和dataLogDir：dataDir和dataLogDir也需要区分下，将数据文件和日志文件分开存放，同时每个server的这两变量所对应的路径都是不同的

■ server.X和myid： server.X 这个数字就是对应，data/myid中的数字。在3个server的myid文件中分别写入了0，1，2，那么每个server中的zoo.cfg都配 server.0 server.2,server.3就行了。因为在同一台机器上，后面连着的2个端口，3个server都不要一样，否则端口冲突

下面是我所配置的集群伪分布模式，分别通过zoo1.cfg、zoo2.cfg、zoo3.cfg来模拟由三台机器的Zookeeper集群,代码清单 zoo1.cfg如下:

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can pass between

# sending a request and getting an acknowledgement

syncLimit=5

# the directory where the snapshot is stored.

dataDir=/usr/local/zk/data\_1

# the port at which the clients will connect

clientPort=2181

#the location of the log file

dataLogDir=/usr/local/zk/logs\_1

server.0=localhost:2287:3387

server.1=localhost:2288:3388

server.2=localhost:2289:3389

代码清单  zoo2.cfg如下:

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can pass between

# sending a request and getting an acknowledgement

syncLimit=5

# the directory where the snapshot is stored.

dataDir=/usr/local/zk/data\_2

# the port at which the clients will connect

clientPort=2182

#the location of the log file

dataLogDir=/usr/local/zk/logs\_2

server.0=localhost:2287:3387

server.1=localhost:2288:3388

server.2=localhost:2289:3389

代码清单 zoo3.cfg如下:

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can pass between

# sending a request and getting an acknowledgement

syncLimit=5

# the directory where the snapshot is stored.

dataDir=/usr/local/zk/data\_3

# the port at which the clients will connect

clientPort=2183

#the location of the log file

dataLogDir=/usr/local/zk/logs\_3

server.0=localhost:2287:3387

server.1=localhost:2288:3388

server.2=localhost:2289:3389

##### ****启动****

在集群为分布式下，我们只有一台机器，按时要运行三个Zookeeper实例。此时，如果在使用单机模式的启动命令是行不通的。此时，只要通过下面三条命令就能运行前面所配置的Zookeeper服务。如下所示：

zkServer.sh start zoo1.sh

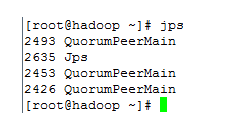
zkServer.sh start zoo2.sh

zkServer.sh start zoo3.sh

启动过程，如下图所示：



启动结果，如下图所示：



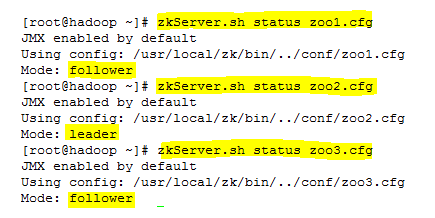
在运行完第一条指令之后，会出现一些错误异常，产生异常信息的原因是由于Zookeeper 服务的每个实例都拥有全局配置信息，他们在启动的时候会随时随地的进行Leader选举操作。此时，第一个启动的Zookeeper需要和另外两个 Zookeeper实例进行通信。但是，另外两个Zookeeper实例还没有启动起来，因此就产生了这的异样信息。我们直接将其忽略即可，待把图中“2 号”和“3号”Zookeeper实例启动起来之后，相应的异常信息自然会消失。此时，可以通过下面三条命令，来查询。

zkServer.sh status zoo1.cfg

zkServer.sh status zoo2.cfg

zkServer.sh status zoo3.cfg

Zookeeper服务的运行状态，如下图所示:



#### ****Zookeeper的集群模式搭建****

为了获得可靠地Zookeeper服务，用户应该在一个机群上部署Zookeeper。只要机群上大多数的Zookeeper服务启动了，那么总的 Zookeeper服务将是可用的。集群的配置方式，和前两种类似，同样需要进行环境变量的配置。在每台机器上conf/zoo.cf配置文件的参数设置 相同

##### ****创建myid****

在dataDir(/usr/local/zk/data)目录创建myid文件

Server0机器的内容为：0

Server1机器的内容为：1

Server2机器的内容为：2

##### ****编写配置文件****

在conf目录下删除zoo\_sample.cfg文件，创建一个配置文件zoo.cfg，如下所示,代码清单  zoo.cfg中的参数设置

# The number of milliseconds of each tick

tickTime=2000

# The number of ticks that the initial

# synchronization phase can take

initLimit=10

# The number of ticks that can pass between

# sending a request and getting an acknowledgement

syncLimit=5

# the directory where the snapshot is stored.

dataDir=/usr/local/zk/data

# the port at which the clients will connect

clientPort=2183

#the location of the log file

dataLogDir=/usr/local/zk/log

server.0=hadoop:2288:3388

server.1=hadoop0:2288:3388

server.2=hadoop1:2288:3388

##### ****启动****

分别在3台机器上启动ZooKeeper的Server：zkServer.sh start；

### Zookeeper的配置

Zookeeper的功能特性是通过Zookeeper配置文件来进行控制管理的(zoo.cfg).这样的设计其实有其自身的原因，通过前面对Zookeeper的配置可以看出，在对Zookeeper集群进行配置的时候，它的配置文档是完全相同的。集群伪分布模式中，有少部分是不同的。这样的配置方式使得在部署Zookeeper服务的时候非常方便。如果服务器使用不同的配置文件，必须确保不同配置文件中的服务器列表相匹配。

在设置Zookeeper配置文档时候，某些参数是可选的，某些是必须的。这些必须参数就构成了Zookeeper配置文档的最低配置要求。另外，若要对Zookeeper进行更详细的配置，可以参考下面的内容。

#### ****基本配置****

下面是在最低配置要求中必须配置的参数：

**(1) client**：监听客户端连接的端口。

**(2) tickTime**：基本事件单元，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，每隔tickTime时间就会发送一个心跳；最小 的session过期时间为2倍tickTime

**dataDir**：存储内存中数据库快照的位置，如果不设置参数，更新食物的日志将被存储到默认位置。

应该谨慎的选择日志存放的位置，使用专用的日志存储设备能够大大提高系统的性能，如果将日志存储在比较繁忙的存储设备上，那么将会很大程度上影像系统性能。

#### ****高级配置****

下面是高级配置参数中可选配置参数，用户可以使用下面的参数来更好的规定Zookeeper的行为：

**(1) dataLogdDir**

这个操作让管理机器把事务日志写入“dataLogDir”所指定的目录中，而不是“dataDir”所指定的目录。这将允许使用一个专用的日志设备，帮助我们避免日志和快照的竞争。配置如下：

# the directory where the snapshot is stored

   dataDir=/usr/local/zk/data

**(2)** **maxClientCnxns**

这个操作将限制连接到Zookeeper的客户端数量，并限制并发连接的数量，通过IP来区分不同的客户端。此配置选项可以阻止某些类别的Dos攻击。将他设置为零或忽略不进行设置将会取消对并发连接的限制。

例如，此时我们将maxClientCnxns的值设为1，如下所示：

# set maxClientCnxns

   maxClientCnxns=1

启动Zookeeper之后，首先用一个客户端连接到Zookeeper服务器上。之后如果有第二个客户端尝试对Zookeeper进行连接，或者有某些隐式的对客户端的连接操作，将会触发Zookeeper的上述配置。

**(3)** **minSessionTimeout**和**maxSessionTimeout**

即最小的会话超时和最大的会话超时时间。在默认情况下，minSession=2\*tickTime；maxSession=20\*tickTime。

#### ****集群配置****

**(1)** **initLimit**

此配置表示，允许follower(相对于Leaderer言的“客户端”)连接并同步到Leader的初始化连接时间，以tickTime为单位。当初始化连接时间超过该值，则表示连接失败。

**(2)** **syncLimit**

此配置项表示Leader与Follower之间发送消息时，请求和应答时间长度。如果follower在设置时间内不能与leader通信，那么此follower将会被丢弃。

**(3)** **server.A=B：C：D**

A：其中 A 是一个数字，表示这个是服务器的编号；

B：是这个服务器的 ip 地址；

C：Leader选举的端口；

D：Zookeeper服务器之间的通信端口。

**(4) myid**和**zoo.cfg**

除了修改 zoo.cfg 配置文件，集群模式下还要配置一个文件 myid，这个文件在 dataDir 目录下，这个文件里面就有一个数据就是 A 的值，Zookeeper 启动时会读取这个文件，拿到里面的数据与 zoo.cfg 里面的配置信息比较从而判断到底是那个 server。

### 搭建ZooKeeper服务器集群

**搭建要求：**

(1) zk服务器集群规模不小于3个节点

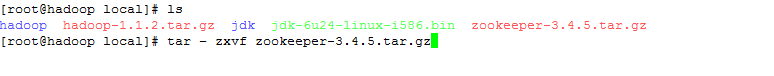
(2) 要求各服务器之间系统时间要保持一致。

#### ****安装配置ZK****

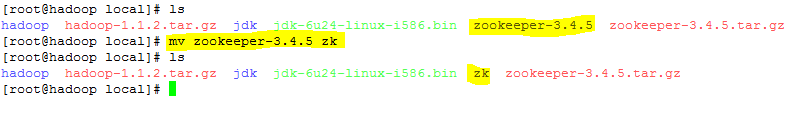
**(1)** 使用WinScp将Zk传输到Hadoop主机上的/usr/local，我用的版本是zookeeper-3.4.5.tar.gz。

**(2)** 在hadoop的/usr/local目录下，解压缩zk....tar.gz，设置环境变量

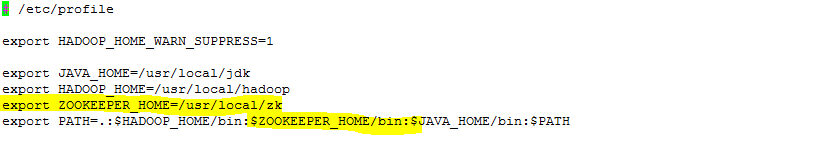
解压缩：在/usr/local目录下，执行命令：tar -zxvf zookeeper-3.4.5.tar.gz，如下图所示:



重命名：解压后将文件夹，重命名为zk，执行命令： mv zookeeper-3.4.5 zk，如下图所示:



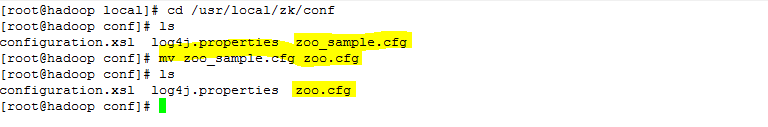
设置环境变量：执行命令： vi /etc/profile ，添加 ：export ZOOKEEPER\_HOME=/usr/local/zk，如图2.3所示的内容。执行命令：source /etc/profile 如下图所示:



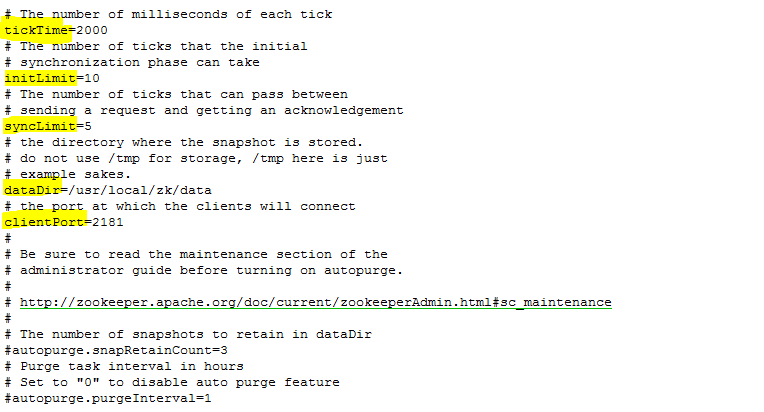
C:\Users\张镇\AppData\Local\Temp\Image.png

#### ****修改ZK配置文件****

**(1) 重命名**：将/usr/local/zk/conf目录下zoo\_sample.cfg，重命名为zoo.cfg，执行命令：mv zoo\_sample.cfg zoo.cfg。如如下图所示:



**(2) 查看**：在/usr/local/zk/conf目录下，修改文件 vi zoo.cfg，文件内容如下图所示。在该文件中dataDir表示文件存放目录，它的默认设置为/tmp/zookeeper这是一个临时存放目录，每次重启后会丢失，在这我们自己设一个目录，/usr/local/zk/data。



**(3) 创建文件夹**：mkdir /usr/local/zk/data

**(4) 创建myid**：在data目录下，创建文件myid，值为0；vi myid ;内容为0。

**(5)** **编辑**：编辑该文件，执行vi zoo.cfg ，修改dataDir=/usr/local/zk/data。

**新增**：

server.0=hadoop:2888:3888

server.1=hadoop0:2888:3888

server.2=hadoop1:2888:3888

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；

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：这个端口就是客户端连接 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。

当这些配置项配置好后，就可以启动 Zookeeper 了，启动后使用命令echo ruok | nc localhost 2181检查 Zookeeper 是否已经在服务。

#### ****配置其他节点****

**(1)** 把haooop主机的zk目录和/etc/profile目录，复制到hadoop0和hadoop1中。执行命令：

scp -r /usr/local/zk/ hadoop0:/usr/local/

　　　　　　scp -r /usr/local/zk/ hadoop1:/usr/local/

　　　　　　scp /etc/profile hadoop0:/etc/

　　　　　　scp /etc/profile hadoop1:/etc/

　　　　　　ssh hadoop0

　　　　　　suorce /etc/profile

　　　　　　vi /usr/local/zk/data/myid

　　　　　　exit

　　　　　　ssh hadoop1

　　　　　　suorce /etc/profile

　　　　　　vi /usr/local/zk/data/myid

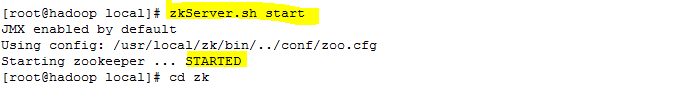
　　　　　　exit

**(2)** 把hadoop1中相应的myid的值改为1，把hadoop2中相应的myid的值改为2。

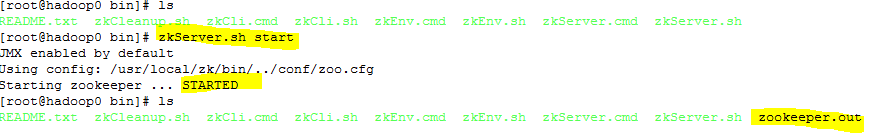
### 启动检验

**(1)** 启动，在三个节点上分别执行命令zkServer.sh start

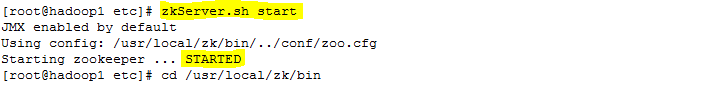
**hadoop节点**：



**hadoop0节点**：

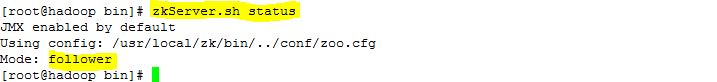


**hadoop1节点**：

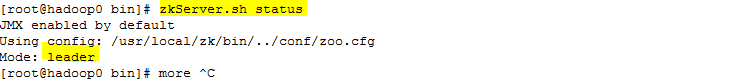


**(2)** 检验，在三个节点上分别执行命令zkServer.sh status，从下面的图中我们会发现hadoop和hadoop1为Follower，hadoop0为Leader。

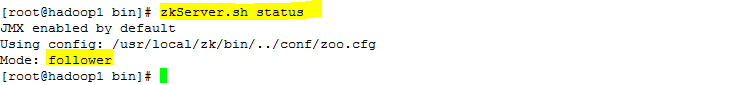
**hadoop节点**：



**hadoop0节点**：



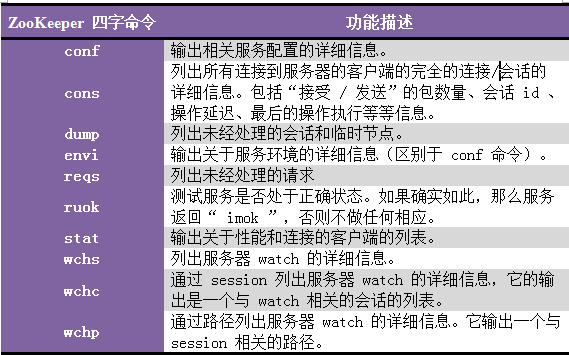
**hadoop1节点**：



## ZooKeeper命令操作

### Zookeeper的四字命令

Zookeeper支持某些特定的四字命令字母与其的交互。他们大多数是查询命令，用来获取Zookeeper服务的当前状态及相关信息。用户在客户端可以通过telnet或nc向Zookeeper提交相应的命令。Zookeeper常用的四字命令见下图所示。



上图，是Zookeeper四字命令的一个简单用例。

[root@hadoop ~]# echo ruok|nc localhost 2181

[root@hadoop ~]# zkServer.sh start zoo1.cfg

JMX enabled by default

Using config: /usr/local/zk/bin/../conf/zoo1.cfg

Starting zookeeper ... STARTED

[root@hadoop ~]# zkServer.sh start zoo2.cfg

JMX enabled by default

Using config: /usr/local/zk/bin/../conf/zoo2.cfg

Starting zookeeper ... STARTED

[root@hadoop ~]# zkServer.sh start zoo3.cfg

JMX enabled by default

Using config: /usr/local/zk/bin/../conf/zoo3.cfg

Starting zookeeper ... STARTED

[root@hadoop ~]# echo ruok|nc localhost 2181

imok[root@hadoop ~]# echo ruok|nc localhost 2182

imok[root@hadoop ~]# echo ruok|nc localhost 2183

imok[root@hadoop ~]# echo conf|nc localhost 2181

clientPort=2181

dataDir=/usr/local/zk/data\_1/version-2

dataLogDir=/usr/local/zk/logs\_1/version-2

tickTime=2000

maxClientCnxns=60

minSessionTimeout=4000

maxSessionTimeout=40000

serverId=0

initLimit=10

syncLimit=5

electionAlg=3

electionPort=3387

quorumPort=2287

peerType=0

[root@hadoop ~]#

### Zookeeper的简单操作

#### ****Zookeeper的shell操作****

##### ****Zookeeper命令工具****

再启动Zookeeper服务之后，输入以下命令，连接到Zookeeper服务：

　　　　zkCli.sh -server localhost:2181

执行结果如下所示：

[root@hadoop ~]# zkCli.sh -server localhost:2181

Connecting to localhost:2181

2014-10-17 03:35:51,051 [myid:] - INFO [main:Environment@100] - Client environment:zookeeper.version=3.4.5-1392090, built on 09/30/2012 17:52 GMT

2014-10-17 03:35:51,055 [myid:] - INFO [main:Environment@100] - Client environment:host.name=hadoop

2014-10-17 03:35:51,057 [myid:] - INFO [main:Environment@100] - Client environment:java.version=1.6.0\_24

2014-10-17 03:35:51,057 [myid:] - INFO [main:Environment@100] - Client environment:java.vendor=Sun Microsystems Inc.

2014-10-17 03:35:51,066 [myid:] - INFO [main:Environment@100] - Client environment:java.home=/usr/local/jdk/jre

2014-10-17 03:35:51,079 [myid:] - INFO [main:Environment@100] - Client environment:java.class.path=/usr/local/zk/bin/../build/classes:/usr/local/zk/bin/../build/lib/\*.jar:/usr/local/zk/bin/../lib/slf4j-log4j12-1.6.1.jar:/usr/local/zk/bin/../lib/slf4j-api-1.6.1.jar:/usr/local/zk/bin/../lib/netty-3.2.2.Final.jar:/usr/local/zk/bin/../lib/log4j-1.2.15.jar:/usr/local/zk/bin/../lib/jline-0.9.94.jar:/usr/local/zk/bin/../zookeeper-3.4.5.jar:/usr/local/zk/bin/../src/java/lib/\*.jar:/usr/local/zk/bin/../conf:

2014-10-17 03:35:51,083 [myid:] - INFO [main:Environment@100] - Client environment:java.library.path=/usr/local/jdk/jre/lib/i386/client:/usr/local/jdk/jre/lib/i386:/usr/local/jdk/jre/../lib/i386:/usr/java/packages/lib/i386:/lib:/usr/lib

2014-10-17 03:35:51,084 [myid:] - INFO [main:Environment@100] - Client environment:java.io.tmpdir=/tmp

2014-10-17 03:35:51,086 [myid:] - INFO [main:Environment@100] - Client environment:java.compiler=<NA>

2014-10-17 03:35:51,099 [myid:] - INFO [main:Environment@100] - Client environment:os.name=Linux

2014-10-17 03:35:51,100 [myid:] - INFO [main:Environment@100] - Client environment:os.arch=i386

2014-10-17 03:35:51,101 [myid:] - INFO [main:Environment@100] - Client environment:os.version=2.6.32-358.el6.i686

2014-10-17 03:35:51,101 [myid:] - INFO [main:Environment@100] - Client environment:user.name=root

2014-10-17 03:35:51,102 [myid:] - INFO [main:Environment@100] - Client environment:user.home=/root

2014-10-17 03:35:51,106 [myid:] - INFO [main:Environment@100] - Client environment:user.dir=/root

2014-10-17 03:35:51,120 [myid:] - INFO [main:ZooKeeper@438] - Initiating client connection, connectString=localhost:2181 sessionTimeout=30000 watcher=org.apache.zookeeper.ZooKeeperMain$MyWatcher@b02e7a

Welcome to ZooKeeper!

JLine support is enabled

2014-10-17 03:35:51,233 [myid:] - INFO [main-SendThread(localhost:2181):ClientCnxn$SendThread@966] - Opening socket connection to server localhost/127.0.0.1:2181. Will not attempt to authenticate using SASL (Unable to locate a login configuration)

2014-10-17 03:35:51,247 [myid:] - INFO [main-SendThread(localhost:2181):ClientCnxn$SendThread@849] - Socket connection established to localhost/127.0.0.1:2181, initiating session

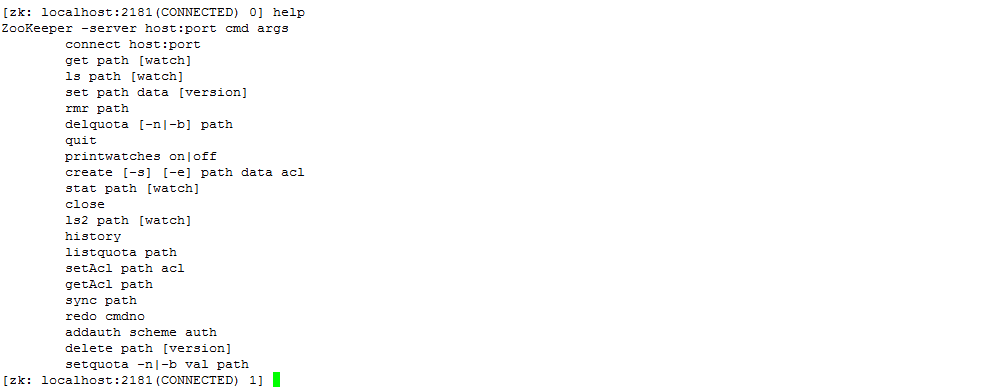
[zk: localhost:2181(CONNECTING) 0] 2014-10-17 03:35:51,290 [myid:] - INFO [main-SendThread(localhost:2181):ClientCnxn$SendThread@1207] - Session establishment complete on server localhost/127.0.0.1:2181, sessionid = 0x491da0e20b0000, negotiated timeout = 30000

WATCHER::

WatchedEvent state:SyncConnected type:None path:null

[zk: localhost:2181(CONNECTED) 0]

连接成功之后，系统会输出Zookeeper的相关环境及配置信息，并在屏幕输出“welcome to Zookeeper！”等信息。输入help之后，屏幕会输出可用的Zookeeper命令，如下图所示



##### ****使用Zookeeper命令的简单操作步骤****

(1) 使用ls命令查看当前Zookeeper中所包含的内容：ls /

[zk: localhost:2181(CONNECTED) 1] ls /

[zookeeper]

[zk: localhost:2181(CONNECTED) 2]

(2) 创建一个新的Znode节点"zk"，以及和它相关字符，执行命令：create /zk myData

[zk: localhost:2181(CONNECTED) 2] create /zk myData

Created /zk

(3) 再次使用ls命令来查看现在Zookeeper的中所包含的内容：ls /

[zk: localhost:2181(CONNECTED) 3] ls /

[zk, zookeeper]

此时看到，zk节点已经被创建。

(4) 使用get命令来确认第二步中所创建的Znode是否包含我们创建的字符串，执行命令：get /zk

[zk: localhost:2181(CONNECTED) 4] get /zk

myData

cZxid = 0x500000006

ctime = Fri Oct 17 03:54:20 PDT 2014

mZxid = 0x500000006

mtime = Fri Oct 17 03:54:20 PDT 2014

pZxid = 0x500000006

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 6

numChildren = 0

(5) 接下来通过set命令来对zk所关联的字符串进行设置，执行命令：set /zk jiang1234

[zk: localhost:2181(CONNECTED) 5] set /zk jiang2014

cZxid = 0x500000006

ctime = Fri Oct 17 03:54:20 PDT 2014

mZxid = 0x500000007

mtime = Fri Oct 17 03:55:50 PDT 2014

pZxid = 0x500000006

cversion = 0

dataVersion = 1

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 9

numChildren = 0

(6) 再次使用get命令来查看，上次修改的内容，执行命令：get /zk

[zk: localhost:2181(CONNECTED) 6] get /zk

jiang2014

cZxid = 0x500000006

ctime = Fri Oct 17 03:54:20 PDT 2014

mZxid = 0x500000007

mtime = Fri Oct 17 03:55:50 PDT 2014

pZxid = 0x500000006

cversion = 0

dataVersion = 1

aclVersion = 0

ephemeralOwner = 0x0

dataLength = 9

numChildren = 0

(7) 下面我们将刚才创建的Znode删除，执行命令：delete /zk

[zk: localhost:2181(CONNECTED) 7] delete /zk

(8) 最后再次使用ls命令查看Zookeeper中的内容，执行命令：ls /

[zk: localhost:2181(CONNECTED) 8] ls /

[zookeeper]

经过验证，zk节点已经删除。

#### ****Zookeeper的api的简单使用****

##### ****ZookeeperAPI简介****

Zookeeper API共包含五个包，分别为：

（1）org.apache.zookeeper

（2）org.apache.zookeeper.data

（3）org.apache.zookeeper.server

（4）org.apache.zookeeper.server.quorum

（5）org.apache.zookeeper.server.upgrade

其中org.apache.zookeeper，包含Zookeeper类，他是我们编程时最常用的类文件。这个类是Zookeeper客户端的主要类文件。如果要使用Zookeeper服务，应用程序首先必须创建一个Zookeeper实例，这时就需要使用此类。一旦客户端和Zookeeper服务建立起了连接，Zookeeper系统将会给次连接会话分配一个ID值，并且客户端将会周期性的向服务器端发送心跳来维持会话连接。只要连接有效，客户端就可以使用Zookeeper API来做相应处理了。

Zookeeper类提供了如下图所示的几类主要方法



##### ****Zookeeper API的使用****

这里通过一个例子来简单介绍如何使用Zookeeper API 编写自己的应用程序，代码如下：

**package** org.zk;

**import** java.io.IOException;

**import** java.util.List;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.ZooKeeper;

**public** **class** ListGroup **extends** ConnectionWatcher {

**public** **void** list(String groupNmae) **throws** KeeperException,

InterruptedException {

String path = "/" + groupNmae;

**try** {

List<String> children = zk.getChildren(path, **false**);

**if** (children.isEmpty()) {

System.*out*.printf("No memebers in group %s\n", groupNmae);

System.*exit*(1);

}

**for** (String child : children) {

System.*out*.println(child);

}

} **catch** (KeeperException.NoNodeException e) {

System.*out*.printf("Group %s does not exist \n", groupNmae);

System.*exit*(1);

}

}

**public** **static** **void** main(String[] args) **throws** IOException,

InterruptedException, KeeperException {

ListGroup listGroup = **new** ListGroup();

listGroup.connect(args[0]);

listGroup.list(args[1]);

listGroup.close();

}

}

此类包含两个主要的 ZooKeeper 函数，分别为 createZKInstance ()和 ZKOperations()。其中：

(1) createZKInstance()函数负责对 ZooKeeper 实例 zk 进行初始化。

ZooKeeper 类有两个构造函数，我们这里使用“ ZooKeeper (String connectString, int sessionTimeout ,Watcher watcher )”对其进行初始化。因此，我们需要提供初始化所需的，连接字符串信息，会话超时时间，以及一个 watcher 实例。 19行到 25行代码，是程序所构造的一个watcher 实例，它能够输出所发生的事件。

(2) ZKOperations ()函数是我们所定义的对节点的一系列操作。

它包括：创建 ZooKeeper 节点（ 35行到 36行代码）、查看节点（ 38 行到 39 行代码）、修改节点数据（ 41 行到 42 行代码）、查看修改后节点数据（ 44 行到 45行代码）、删除节点（ 47行到 48行代码）、查看节点是否存在（ 50 行到 51 行代码）。

代码的运行结果如下：

1. 创建ZooKeeper节点（Znode：/znode;数据：myData2;权限：OPEN\_ACL\_UNSAFE;节点类型：Persistent）

None

2. 查看节点是否创建成功：

/znode myData2

3. 修改节点数据：

4. 查看是否修改成功：

jiang1234

5. 删除节点：

6. 查看/znode节点状态：

节点间状态：[null]

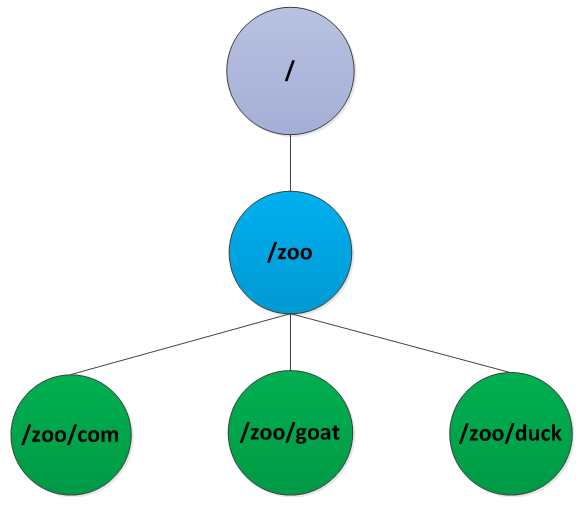
### ZooKeeper示例

假设一组服务器，用于为客户端提供一些服务。我们希望每个客户端都能够能够找到其中一台服务器，使其能够使用这些服务，挑战之一就是维护这组服务器 列表。这组服务器的成员列表明显不能存在网络中的单个节点上，因为如果那个节点发生故障，就意味着是整个系统的故障（我们希望这个列表有很高的可用性）。 假设我们有了一个可靠的方法解决了这个成员列表的存储问题。如果其中一台服务器出现故障，我们仍然需要解决如何从服务器成员列表中将它删除的问题。某个进 程需要负责删除故障服务器，但注意不能由故障服务器自己来完成，因为故障服务器已经不再运行。

我们所描述的不是一个被动的分布式数据结构，而是一个主动的、能够在某个外部事件发生时修改数据项状态的数据结构。ZooKeeper提供这种服务，所以让我们看看如何使用它来实现这种众所周知的组成员管理应用。

**ZooKeeper中的组成员关系**

理解ZooKeeper的一种方法就是将其看作一个具有高可用性的文件系统。但这个文件系统中没有文件和目录，而是统一使用“节点”(node)的概念，称为znode。znode既可以作为保存数据的容器（如同文件），也可以作为保存其他znode的容器（如同目录）。所有的znode构成一个层次化的命名空间。一种自然的建立组成员列表的方式就是利用这种层次结构，创建一个以组名为节点名的znode作为父节点，然后以组成员名（服务器名）为节点名来创建作为子节点的znode。如下图给出了一组具有层次结构的znode。



在这个示例中，我们没有在任何znode中存储数据，但在一个真实的应用中，你可以将“关于成员的数据”存储在它们的znode中，例如主机名。

#### ****创建组****

##### ****代码示例****

让我们通过编写一段程序的方式来再次详细介绍ZooKeeper的Java API，这段示例程序用于为组名为/zoo的组创建一个znode。代码参见如下

**代码 该程序在ZooKeeper中新建表示组的Znode**

**package** org.zk;

**import** java.io.IOException;

**import** java.util.concurrent.CountDownLatch;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.Watcher.Event.KeeperState;

**import** org.apache.zookeeper.ZooDefs.Ids;

**import** org.apache.zookeeper.ZooKeeper;

**public** **class** CreateGroup **implements** Watcher {

**private** **static** **final** **int** *SESSION\_TIMEOUT* = 5000;

**private** ZooKeeper zk;

**private** CountDownLatch connectedSignal = **new** CountDownLatch(1);

@Override

**public** **void** process(WatchedEvent event) {

**if** (event.getState() == KeeperState.SyncConnected) {

connectedSignal.countDown();

}

}

**public** **static** **void** main(String[] args) **throws** IOException,

InterruptedException, KeeperException {

CreateGroup createGroup = **new** CreateGroup();

createGroup.connect(args[0]);

createGroup.create(args[1]);

createGroup.close();

}

**private** **void** close() **throws** InterruptedException {

zk.close();

}

**private** **void** create(String groupName) **throws** KeeperException,

InterruptedException {

String path = "/" + groupName;

**if** (zk.exists(path, **false**) == **null**) {

zk.create(path, **null**/\* data \*/, Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

System.*out*.println("Created:" + path);

}

**private** **void** connect(String hosts) **throws** IOException, InterruptedException {

zk = **new** ZooKeeper(hosts, *SESSION\_TIMEOUT*, **this**);

connectedSignal.await();

}

}

运行该程序需要配置classpath环境变量或者在执行java命令时添加-classpath选项，具体运行方式参见：<http://www.cnblogs.com/sunddenly/p/4050812.html>

运行后的结果为：

[root@hadoop code]# ls

build classes CreateGroup.java HelloWorld.java jar.jar PackageTest.java zookeeper.out

[root@hadoop code]# javac -d ./classes CreateGroup.java

[root@hadoop code]# java org.zk.CreateGroup localhost:2181 zoo

2014-10-28 18:00:26,154 [myid:] - INFO [main:Environment@100] - Client environment:zookeeper.version=3.4.5-1392090, built on

2014-10-28 18:00:26,157 [myid:] - INFO [main:Environment@100] - Client environment:host.name=hadoop

2014-10-28 18:00:26,157 [myid:] - INFO [main:Environment@100] - Client environment:java.version=1.6.0\_24

2014-10-28 18:00:26,157 [myid:] - INFO [main:Environment@100] - Client environment:java.vendor=Sun Microsystems Inc.

2014-10-28 18:00:26,158 [myid:] - INFO [main:Environment@100] - Client environment:java.home=/usr/local/jdk/jre

2014-10-28 18:00:26,158 [myid:] - INFO [main:Environment@100] - Client environment:java.class.path=……

……

Created:/zoo

2014-10-28 18:00:26,236 [myid:] - INFO [main:ZooKeeper@684] - Session: 0x4956f7f1d70005 closed

2014-10-28 18:00:26,237 [myid:] - INFO [main-EventThread:ClientCnxn$EventThread@509] - EventThread shut down

[root@hadoop code]#

##### ****代码分析****

在上面代码中，main()方法执行时，创建了一个CreateGroup的实例并且调用了这个实例的connect()方法。connect方法实例化了一个新的ZooKeeper类的对象，这个类是客户端API中的主要类，并且负责维护客户端和ZooKeeper服务之间的连接。ZooKeeper类的构造函数有三个参数：

* 第一个是：ZooKeeper服务的主机地址，可指定端口，默认端口是2181。
* 第二个是：以毫秒为单位的会话超时参数，这里我们设成5秒。
* 第三个是：参数是一个Watcher对象的实例。

Watcher对象接收来自于ZooKeeper的回调，以获得各种事件的通知。在这个例子中，CreateGroup是一个Watcher对象，因此我们将它传递给ZooKeeper的构造函数。

当一个ZooKeeper的实例被创建时，会启动一个线程连接到ZooKeeper服务。由于对构造函数的调用是立即返回的，因此在使用新建的ZooKeeper对象之前一定要等待其与ZooKeeper服务之间的连接建立成功。我们使用Java的CountDownLatch类来阻止使用新建的ZooKeeper对象，直到这个ZooKeeper对象已经准备就绪。这就是Watcher类的

用途，在它的接口中只有一个方法：

**public void process(WatcherEvent event);**

客户端已经与ZooKeeper建立连接后，Watcher的process()方法会被调用，参数是一个表示该连接的事件。在接收到一个连接事件（由Watcher.Event.KeeperState的枚举型值SyncConnected来表示）时，我们通过调用CountDownLatch的countDown()方法来递减它的计数器。锁存器(latch)被创建时带有一个值为1的计数器，用于表示在它释放所有等待线程之前需要发生的事件数。在调用一欢countDown()方法之后，计数器的值变为0，则await()方法返回。

现在connect()方法已经返回，下一个执行的是CreateGroup的create()方法。在这个方法中，我们使用ZooKeeper实例中的create()方法来创建一个新的ZooKeeper的znode。所需的参数包括：

* 路径：用字符串表示。
* znode的内容：字节数组，本例中使用空值。
* 访问控制列表：简称ACL，本例中使用了完全开放的ACL，允许任何客户端对znode进行读写。
* 创建znode的类型：有两种类型的znode：短暂的和持久的。

创建znode的客户端断开连接时，无论客户端是明确断开还是因为任何原因而终止，短暂znode都会被ZooKeeper服务删除。与之相反，当客户端断开连接时，持久znode不会被删除。我们希望代表一个组的znode存活的时间应当比创建程序的生命周期要长，因此在本例中我们创建了一个持久的znode。

create()方法的返回值是ZooKeeper所创建的路径，我们用这个返回值来打印一条表示路径成功创建的消息。当我们查看“顺序znode”(sequential znode)时．会发现create()方法返回的路径与传递给该方法的路径不同。

#### ****加入组****

下面的这一段程序用于注册组的成员。每个组成员将作为一个程序运行，并且加入到组中。当程序退出时，这个组成员应当从组中被删除。为了实现这一点，我们在ZooKeeper的命名空间中使用短暂znode来代表一个组成员。

在基类ConnectionWatcher中，对创建和连接ZooKeeper实例的程序逻辑进行了重构，参见代码如下

**代码 用于将成员加入组的程序**

**package** org.zk;

**import** java.io.IOException;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.ZooDefs.Ids;

**public** **class** JoinGroup **extends** ConnectionWatcher {

**public** **void** join(String groupName, String memberName)

**throws** KeeperException, InterruptedException {

String path = "/" + groupName + "/" + memberName;

String createdPath = zk.create(path, **null**, Ids.OPEN\_ACL\_UNSAFE,

CreateMode.EPHEMERAL);

System.*out*.println("Created:" + createdPath);

}

**public** **static** **void** main(String[] args) **throws** InterruptedException,

IOException, KeeperException {

JoinGroup joinGroup = **new** JoinGroup();

joinGroup.connect(args[0]);

joinGroup.join(args[1], args[2]);

// stay alive until process is killed or thread is interrupted

Thread.*sleep*(Long.*MAX\_VALUE*);

}

}

**代码 3.3 用于等待建立与ZooKeeper连接的辅助类**

**package** org.zk;

**import** java.io.IOException;

**import** java.util.concurrent.CountDownLatch;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.Watcher.Event.KeeperState;

**import** org.apache.zookeeper.ZooKeeper;

**public** **class** ConnectionWatcher **implements** Watcher {

**private** **static** **final** **int** *SESSION\_TIMEOUT* = 5000;

**protected** ZooKeeper zk;

CountDownLatch connectedSignal = **new** CountDownLatch(1);

**public** **void** connect(String host) **throws** IOException, InterruptedException {

zk = **new** ZooKeeper(host, *SESSION\_TIMEOUT*, **this**);

connectedSignal.await();

}

@Override

**public** **void** process(WatchedEvent event) {

**if** (event.getState() == KeeperState.SyncConnected) {

connectedSignal.countDown();

}

}

**public** **void** close() **throws** InterruptedException {

zk.close();

}

}

JoinGroup的代码与CreateGroup非常相似，在它的join()方法中，创建短暂znode，作为组znode的子节点，然后通过 休眠来模拟正在做某种工作，直到该进程被强行终止。接着，你会看到随着进程终止，这个短暂znode被ZooKeeper删除。

#### ****列出组成员****

现在，我们需要一段程序来查看组成员，参见代码如下：

**代码 用于列出组成员的程序**

**package** org.zk;

**import** java.io.IOException;

**import** java.util.List;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.ZooKeeper;

**public** **class** ListGroup **extends** ConnectionWatcher {

**public** **void** list(String groupNmae) **throws** KeeperException,

InterruptedException {

String path = "/" + groupNmae;

**try** {

List<String> children = zk.getChildren(path, **false**);

**if** (children.isEmpty()) {

System.*out*.printf("No memebers in group %s\n", groupNmae);

System.*exit*(1);

}

**for** (String child : children) {

System.*out*.println(child);

}

} **catch** (KeeperException.NoNodeException e) {

System.*out*.printf("Group %s does not exist \n", groupNmae);

System.*exit*(1);

}

}

**public** **static** **void** main(String[] args) **throws** IOException,

InterruptedException, KeeperException {

ListGroup listGroup = **new** ListGroup();

listGroup.connect(args[0]);

listGroup.list(args[1]);

listGroup.close();

}

}

在list()方法中，我们调用了getChildren()方法来检索并打印输出一个znode的子节点列表，调用参数为：该znode的路径和 设为false的观察标志。如果在一znode上设置了观察标志，那么一旦该znode的状态改变，关联的观察(Watcher)会被触发。虽然在这里我 们可以不使用观察，但在查看一个znode的子节点时，也可以设置观察，让应用程序接收到组成员加入、退出和组被删除的有关通知。

在这段程序中，我们捕捉了KeeperException.NoNodeException异常，代表组的znode不存在时，这个异常就会被抛 出。下面看一下ListGroup程序的工作过程：虽然搭建了分布式的ZooKeeper，但分布式ZooKeeper启动运行比较耗时，我在这采用前面 提到的复制模式下的ZooKeeper来进行测试。

首先我们得启动ZooKeeper，启动以后将上面的源程序放到Linux目录中并进行编译，我将其放到了"/usr/code"目录下，并在该目录下创建一个classes文件夹，用于存放生成字节码文件：

[root@hadoop ~]# cd /usr/code

[root@hadoop code]# ls

ConnectionWatcher.java DeleteGroup.java ListGroup.java

classes CreateGroup.java JoinGroup.java PackageTest.java

[root@hadoop code]# javac -d ./classes ConnectionWatcher.java

[root@hadoop code]# javac -d ./classes \*.java

由于目前我们还没有在组中添加任何成员，因此zoo是空的：

[root@hadoop code]# java org.zk.ListGroup localhost zoo

2014-10-30 01:52:19,703 [myid:] - INFO [main:Environment@100] - Client environment:……

……

No memebers in group zoo

我们可以使用JoinGroup来向组中添加成员。在sleep语句的作用下，这些作为组成员的znode不会自己终止，所以我们可以，以后台进程的方式来启动他们：

[root@hadoop code]# java org.zk.JoinGroup localhost zoo duck &

2014-10-30 02:06:05,018 [myid:] - INFO [main:Environment@100] - Client environment:……

……

Created:/zoo/duck

[root@hadoop code]# java org.zk.JoinGroup localhost zoo cow &

2014-10-30 02:06:05,018 [myid:] - INFO [main:Environment@100] - Client environment:……

……

Created:/zoo/cow

[root@hadoop code]# java org.zk.JoinGroup localhost zoo goat &

2014-10-30 02:06:05,018 [myid:] - INFO [main:Environment@100] - Client environment:……

……

Created:/zoo/goat

最后一行命令保存了将goat添加到组中的java进程的ID。我们需要保存这个进程的ID，以便能够在查看组成员之后杀死进程。

[root@hadoop code]#

2014-10-30 03:15:30,619 [myid:] - INFO [main:Environment@100] - Client environment:……

……

duck

cow

goat

为了从组中删除一个成员，我们杀死了goat所对应的进程：

[root@hadoop code]# kill $goat\_pid

几秒钟之后，由于进程的ZooKeeper会话已经结束（超时为5秒），所以goat会从组成员列表消失，并且对应的短暂znode也已经被删除。

[root@hadoop code]# java org.zk.ListGroup localhost zoo

2014-10-30 03:23:41,120 [myid:] - INFO [main:Environment@100] - Client environment:……

……

duck

cow

对于参与到一个分布式系统中的节点，这样就有了一个建立节点列表的方法。这些节点也许彼此并不了解。例如，一个想使用列表中节点来完成某些工作的客 户端，能够在这些节点不知道客户端的情况下发现它们。最后，注意，组成员关系管理并不能解决与节点通信过程中出现的网络问题。即使一个节点是一个组中的成 员，在与其通信的过程中仍然会出现故障，这种故障必须以一种合适的方式解决（重试、使用组中另外一个成员等）。☆☆☆

#### ****ZooKeeper命令行工具****

ZooKeeper提供了一个命令行工具用于在其命名空间内进行交互。我们可以使用这个命令工具列出/zoo节点之下的znode列表，如下所示

[root@hadoop code]# zkCli.sh -server localhost ls /zoo

Connecting to localhost

……

WATCHER::

WatchedEvent state:SyncConnected type:None path:null

[duck, cow]

[root@hadoop code]#

#### ****删除组****

下面来看如何删除一个组。ZooKeeper类提供了一个delete()方法，该方法有两个参数：

1. 路径

2. 版本号

如果所提供的版本号与znode的版本号一致，ZooKeeper会删除这个znode。这是一种乐观的加锁机制，使客户端能够检测出对znode的修改冲突。通过将版本号设置为-1，可以绕过这个版本检测机制，不管znode的版本号是什么而直接将其删除。ZooKeeper不支持递归的删除操作，因此在删除父节点之前必须先删除子节点。

在代码3.5中，DeleteGroup类用于删除一个组及其所有成员。

**代码3.5用于删除一个组及其所有成员的程序**

**package** org.zk;

**import** java.io.IOException;

**import** java.util.List;

**import** org.apache.zookeeper.KeeperException;

**public** **class** DeleteGroup **extends** ConnectionWatcher {

**public** **void** delete(String groupName) **throws** InterruptedException,

KeeperException {

String path = "/" + groupName;

List<String> children;

**try** {

children = zk.getChildren(path, **false**);

**for** (String child : children) {

zk.delete(path + "/" + child, -1);

}

zk.delete(path, -1);

} **catch** (KeeperException.NoNodeException e) {

System.*out*.printf("Group %s does not exist\n", groupName);

System.*exit*(1);

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException,

IOException, KeeperException {

DeleteGroup deleteGroup = **new** DeleteGroup();

deleteGroup.connect(args[0]);

deleteGroup.delete(args[1]);

deleteGroup.close();

}

}

最后，我们可以删除之前所创建的zoo组：

[root@hadoop code]# java org.zk.DeleteGroup localhost zoo

……

[root@hadoop code]# java org.zk.ListGroup localhost zoo

2014-10-30 05:39:41,974 [myid:] - INFO [main:Environment@100] - Client environment:……

Group zoo does not exist

[root@hadoop code]#

## 构建ZooKeeper应用

### 配置服务

配置服务是分布式应用所需要的基本服务之一，它使集群中的机器可以共享配置信息中那些公共的部分。简单地说，ZooKeeper可以作为一个具有高可用性的配置存储器，允许分布式应用的参与者检索和更新配置文件。使用ZooKeeper中的观察机制，可以建立一个活跃的配置服务，使那些感兴趣的客户端能够获得配置信息修改的通知。

下面来编写一个这样的服务。我们通过两个假设来简化所需实现的服务（稍加修改就可以取消这两个假设）。

第一，我们唯一需要存储的配置数据是字符串，关键字是znode的路径，因此我们在每个znode上存储了一个键／值对。

第二，在任何时候只有一个客户端会执行更新操作。

除此之外，这个模型看起来就像是有一个主人(类似于HDFS中的namenode)在更新信息，而他的工人则需要遵循这些信息。

在名为ActiveKeyValueStore的类中编写了如下代码：

**package** org.zk;

**import** java.nio.charset.Charset;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.ZooDefs.Ids;

**import** org.apache.zookeeper.data.Stat;

**public** **class** ActiveKeyValueStore **extends** ConnectionWatcher {

**private** **static** **final** Charset *CHARSET* = Charset.*forName*("UTF-8");

**public** **void** write(String path, String value) **throws** KeeperException,

InterruptedException {

Stat stat = zk.exists(path, **false**);

**if** (stat == **null**) {

zk.create(path, value.getBytes(*CHARSET*), Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

} **else** {

zk.setData(path, value.getBytes(*CHARSET*), -1);

}

}

**public** String read(String path, Watcher watch) **throws** KeeperException,

InterruptedException {

**byte**[] data = zk.getData(path, watch, **null**);

**return** **new** String(data, *CHARSET*);

}

}

write()方法的任务是将一个关键字及其值写到ZooKeeper。它隐藏了创建一个新的znode和用一个新值更新现有znode之间的区 别，而是使用exists操作来检测znode是否存在，然后再执行相应的操作。其他值得一提的细节是需要将字符串值转换为字节数组，因为我们只用了 UTF-8编码的getBytes()方法。☆☆☆

read()方法的任务是读取一个节点的配置属性。ZooKeeper的getData()方法有三个参数：

（1）路径

（2）一个观察对象

（3）一个Stat对象

Stat对象由getData()方法返回的值填充，用来将信息回传给调用者。通过这个方法，调用者可以获得一个znode的数据和元数据，但在这个例子中，由于我们对元数据不感兴趣，因此将Stat参数设为null。

为了说明ActiveKeyValueStore的用法，我们编写了一个用来更新配置属性值的类ConfigUpdater，如代码1.1所示。

代码1.1 用于随机更新ZooKeeper中的属性

**package** org.zk;

**import** java.io.IOException;

**import** java.util.Random;

**import** java.util.concurrent.TimeUnit;

**import** org.apache.zookeeper.KeeperException;

**public** **class** ConfigUpdater {

**public** **static** **final** String *PATH* = "/config";

**private** ActiveKeyValueStore store;

**private** Random random = **new** Random();

**public** ConfigUpdater(String hosts) **throws** IOException, InterruptedException {

store = **new** ActiveKeyValueStore();

store.connect(hosts);

}

**public** **void** run() **throws** InterruptedException, KeeperException {

**while** (**true**) {

String value = random.nextInt(100) + "";

store.write(*PATH*, value);

System.*out*.printf("Set %s to %s\n", *PATH*, value);

TimeUnit.*SECONDS*.sleep(random.nextInt(100));

}

}

**public** **static** **void** main(String[] args) **throws** IOException,

InterruptedException, KeeperException {

ConfigUpdater configUpdater = **new** ConfigUpdater(args[0]);

configUpdater.run();

}

}

这个程序很简单，ConfigUpdater中定义了一个ActiveKeyValueStore，它在ConfigUpdater的构造函数中连接到ZooKeeper。run()方法永远在循环，在随机时间以随机值更新/config znode。

　　作为配置服务的用户，ConfigWatcher创建了一个ActiveKeyValueStore对象store，并且在启动之后通过 displayConfig()调用了store的read()方法，显示它所读到的配置信息的初始值，并将自身作为观察传递给store。当节点状态发 生变化时，再次通过displayConfig()显示配置信息，并再次将自身作为观察传递给store，参见代码1.2：

例1.2 该用应观察ZooKeeper中属性的更新情况，并将其打印到控制台

**package** org.zk;

**import** java.io.IOException;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.Watcher.Event.EventType;

**public** **class** ConfigWatcher **implements** Watcher {

**private** ActiveKeyValueStore store;

@Override

**public** **void** process(WatchedEvent event) {

**if** (event.getType() == EventType.NodeDataChanged) {

**try** {

dispalyConfig();

} **catch** (InterruptedException e) {

System.*err*.println("Interrupted. exiting. ");

Thread.*currentThread*().interrupt();

} **catch** (KeeperException e) {

System.*out*.printf("KeeperException锛?s. Exiting.\n", e);

}

}

}

**public** ConfigWatcher(String hosts) **throws** IOException, InterruptedException {

store = **new** ActiveKeyValueStore();

store.connect(hosts);

}

**public** **void** dispalyConfig() **throws** KeeperException, InterruptedException {

String value = store.read(ConfigUpdater.PATH, **this**);

System.*out*.printf("Read %s as %s\n", ConfigUpdater.PATH, value);

}

**public** **static** **void** main(String[] args) **throws** IOException,

InterruptedException, KeeperException {

ConfigWatcher configWatcher = **new** ConfigWatcher(args[0]);

configWatcher.dispalyConfig();

// stay alive until process is killed or Thread is interrupted

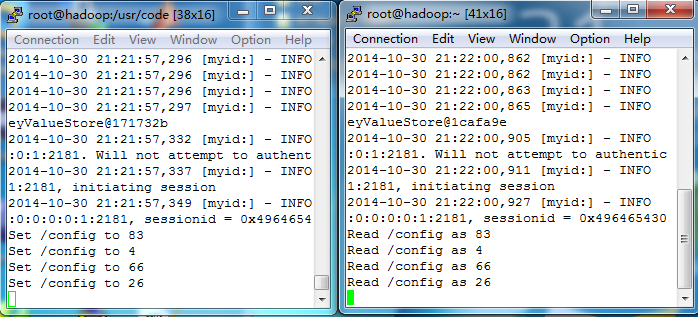
Thread.*sleep*(Long.*MAX\_VALUE*);

}

}

当ConfigUpdater更新znode时，ZooKeeper产生一个类型为EventType.NodeDataChanged的 事件，从而触发观察。ConfigWatcher在它的process()方法中对这个事件做出反应，读取并显示配置的最新版本。由于观察仅发送单次信 号，因此每次我们调用ActiveKeyValueStore的read()方法时，都将一个新的观察告知ZooKeeper来确保我们可以看到将来的更 新。但是，我们还是不能保证接收到每一个更新，因为在收到观察事件通知与下一次读之间，znode可能已经被更新过，而且可能是很多次，由于客户端在这段 时间没有注册任何观察，因此不会收到通知。对于示例中的配置服务，这不是问题，因为客户端只关心属性的最新值，最新值优先于之前的值。但是，一般情况下， 这个潜在的问题是不容忽视的。

让我们看看如何使用这个程序。在一个终端窗口中运行ConfigUpdater，然后在另一个客户端运行ConfigWatcher，我们可以预先 分别在两个客户端输入命令，先不按回车，等两个客户端的命令输入好后，先在运行ConfigUpdater的客户端按回车，再在另一个客户端按回车，运行 结果如下：



### 可恢复的ZooKeeper应用

关于分布式计算的第一个误区是“网络是可靠的”。按照他们的观点，程序总是有一个可靠的网络，因此当程序运行在真正的网络中时，往往会出现各种备样的故障。让我们看看各种可能的故障模式，以及能够解决故障的措施，使我们的程序在面对故障时能够及时复原。

#### ****ZooKeeper异常****

在Java API中的每一个ZooKeeper操作都在其throws子句中声明了两种类型的异常，分别是InterruptedException和KeeperException。

**（一）InterruptedException异常**

如果操作被中断，则会有一个InterruptedException异常被抛出。在Java语言中有一个取消阻塞方法的标准机制，即针对存在阻塞方法的线程调用interrupt()。一个成功的取消操作将产生一个InterruptedException异常。

ZooKeeper也遵循这一机制，因此你可以使用这种方法来取消一个ZooKeeper操作。使用了ZooKeeper的类或库通常会传播 InterruptedException异常，使客户端能够取消它们的操作。InterruptedException异常并不意味着有故障，而是表明相应的操作已经被取消，所以在配置服务的示例中，可以通过传播异常来中止应用程序的运行。

**（二）KeeperException异常**

(1) 如果ZooKeeper服务器发出一个错误信号或与服务器存在通信问题，抛出的则是KeeperException异常。

①针对不同的错误情况，KeeperException异常存在不同的子类。

例如:　KeeperException.NoNodeException是KeeperException的一个子类，如果你试图针对一个不存在的znode执行操作，抛出的则是该异常。

②每一个KeeperException异常的子类都对应一个关于错误类型信息的代码。

例如:　KeeperException.NoNodeException异常的代码是KeeperException.Code.NONODE

(2) 有两种方法被用来处理KeeperException异常：

①捕捉KeeperException异常，并且通过检测它的代码来决定采取何种补救措施；

②另一种是捕捉等价的KeeperException子类，并且在每段捕捉代码中执行相应的操作。

(3) KeeperException异常分为三大类

① 状态异常

当一个操作因不能被应用于znode树而导致失败时，就会出现状态异常。状态异常产生的原因通常是在同一时间有另外一个进程正在修改znode。例如，如果一个znode先被另外一个进程更新了，根据版本号执行setData操作的进程就会失败，并收到一个KeeperException.BadVersionException异常，这是因为版本号不匹配。程序员通常都知道这种冲突总是存在的，也都会编写代码来进行处理。

一些状态异常会指出程序中的错误，例如KeeperException.NoChildrenForEphemeralsException异常，试图在短暂znode下创建子节点时就会抛出该异常。

② 可恢复异常

可恢复的异常是指那些应用程序能够在同一个ZooKeeper会话中恢复的异常。一个可恢复的异常是通过KeeperException.ConnectionLossException来表示的，它意味着已经丢失了与ZooKeeper的连接。ZooKeeper会尝试重新连接，并且在大多数情况下重新连接会成功，并确保会话是完整的。

但是ZooKeeper不能判断与KeeperException.ConnectionLossException异常相关的操作是否成功执行。这种情况就是部分失败的一个例子。这时程序员有责任来解决这种不确定性，并且根据应用的情况来采取适当的操作。在这一点上，就需要对“幂等”(idempotent)操作和“非幂等”(Nonidempotent)操作进行区分。幂等操作是指那些一次或多次执行都会产生相同结果的操作，例如读请求或无条件执行的setData操作。对于幂等操作，只需要简单地进行重试即可。对于非幂等操作，就不能盲目地进行重试，因为它们多次执行的结果与一次执行是完全不同的。程序可以通过在znode的路径和它的数据中编码信息来检测是否非幂等操怍的更新已经完成。

③不可恢复的异常

在某些情况下，ZooKeeper会话会失效——也许因为超时或因为会话被关闭，两种情况下都会收到KeeperException.SessionExpiredException异常，或因为身份验证失败，KeeperException.AuthFailedException异常。无论上述哪种情况，所有与会话相关联的短暂znode都将丢失，因此应用程序需要在重新连接到ZooKeeper之前重建它的状态。

#### ****可靠地服务配置****

首先我们先回顾一下ActivityKeyValueStore的write（）的方法，他由一个exists操作紧跟着一个create操作或setData操作组成：

**public** **class** ActiveKeyValueStore **extends** ConnectionWatcher {

**private** **static** **final** Charset *CHARSET* = Charset.forName("UTF-8");

**public** **void** write(String path, String value) **throws** KeeperException,

InterruptedException {

Stat stat = zk.exists(path, **false**);

**if** (stat == **null**) {

zk.create(path, value.getBytes(*CHARSET*), Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

} **else** {

zk.setData(path, value.getBytes(*CHARSET*), -1);

}

}

**public** String read(String path, Watcher watch) **throws** KeeperException,

InterruptedException {

**byte**[] data = zk.getData(path, watch, **null**);

**return** **new** String(data, *CHARSET*);

}

}

作为一个整体，write()方法是一个“幂等”操作，所以我们可以对他进行无条件重试。我们新建一个类ChangedActiveKeyValueStore，代码如下：

**package** org.zk;

**import** java.nio.charset.Charset;

**import** java.util.concurrent.TimeUnit;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.ZooDefs.Ids;

**import** org.apache.zookeeper.data.Stat;

**public** **class** ChangedActiveKeyValueStore **extends** ConnectionWatcher {

**private** **static** **final** Charset *CHARSET* = Charset.*forName*("UTF-8");

**private** **static** **final** **int** *MAX\_RETRIES* = 5;

**private** **static** **final** **long** *RETRY\_PERIOD\_SECONDS* = 5;

**public** **void** write(String path, String value) **throws** InterruptedException,

KeeperException {

**int** retries = 0;

**while** (**true**) {

**try** {

Stat stat = zk.exists(path, **false**);

**if** (stat == **null**) {

zk.create(path, value.getBytes(*CHARSET*),

Ids.OPEN\_ACL\_UNSAFE, CreateMode.PERSISTENT);

} **else** {

zk.setData(path, value.getBytes(*CHARSET*), stat.getVersion());

}

} **catch** (KeeperException.SessionExpiredException e) {

**throw** e;

} **catch** (KeeperException e) {

**if** (retries++ == *MAX\_RETRIES*) {

**throw** e;

}

// sleep then retry

TimeUnit.*SECONDS*.sleep(*RETRY\_PERIOD\_SECONDS*);

}

}

}

**public** String read(String path, Watcher watch) **throws** KeeperException,

InterruptedException {

**byte**[] data = zk.getData(path, watch, **null**);

**return** **new** String(data, *CHARSET*);

}

}

在该类中，对前面的write()进行了修改,该版本的wirte()能够循环执行重试。其中设置了重试的最大次数MAX\_RETRIES和两次重试之间的间隔RETRY\_PERIOD\_SECONDS.

我们再新建一个类ResilientConfigUpdater，该类对前面的ConfigUpdater进行了修改，代码如下：

**package** org.zk;

**import** java.io.IOException;

**import** java.nio.charset.Charset;

**import** java.util.Random;

**import** java.util.concurrent.TimeUnit;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.KeeperException.SessionExpiredException;

**import** org.apache.zookeeper.ZooDefs.Ids;

**import** org.apache.zookeeper.data.Stat;

**public** **class** ResilientConfigUpdater **extends** ConnectionWatcher {

**public** **static** **final** String *PATH* = "/config";

**private** ChangedActiveKeyValueStore store;

**private** Random random = **new** Random();

**public** ResilientConfigUpdater(String hosts) **throws** IOException,

InterruptedException {

store = **new** ChangedActiveKeyValueStore();

store.connect(hosts);

}

**public** **void** run() **throws** InterruptedException, KeeperException {

**while** (**true**) {

String value = random.nextInt(100) + "";

store.write(*PATH*, value);

System.*out*.printf("Set %s to %s\n", *PATH*, value);

TimeUnit.*SECONDS*.sleep(random.nextInt(10));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

**while** (**true**) {

**try** {

ResilientConfigUpdater configUpdater = **new** ResilientConfigUpdater(

args[0]);

configUpdater.run();

} **catch** (KeeperException.SessionExpiredException e) {

// start a new session

} **catch** (KeeperException e) {

// already retried ,so exit

e.printStackTrace();

**break**;

}

}

}

}

在这段代码中没有对KeepException.SeeionExpiredException异常进行重试，因为一个会话过期 时，ZooKeeper对象会进入CLOSED状态，此状态下它不能进行重试连接。我们只能将这个异常简单抛出并让拥有着创建一个新实例，以重试整个 write()方法。一个简单的创建新实例的方法是创建一个新的ResilientConfigUpdater用于恢复过期会话。

处理会话过期的另一种方法是在观察中（在这个例子中应该是ConnectionWatcher）寻找类型为ExpiredKeepState，然后 再找到的时候创建一个新连接。即使我们收到KeeperException.SessionExpiredEception异常，这种方法还是可以让我们 在write（）方法内不断重试，因为连接最终是能够重新建立的。不管我们采用何种机制从过期会话中恢复，重要的是，这种不同于连接丢失的故障类型，需要 进行不同的处理。

注意：实际上,这里忽略了另一种故障模式。当ZooKeeper对象被创建时，他会尝试连接另一个ZooKeeper服务器。如果连接失败或超时， 那么他会尝试连接集合体中的另一台服务器。如果在尝试集合体中的所有服务器之后仍然无法建立连接，它会抛出一个IOException异常。由于所有的 ZooKeeper服务器都不可用的可能性很小，所以某些应用程序选择循环重试操作，直到ZooKeeper服务为止。

这仅仅是一种重试处理策略，还有许多其他处理策略，例如使用“指数返回”，每次将重试的间隔乘以一个常数。Hadoop内核中 org.apache.hadoop.io.retry包是一组工具，用于可以重用的方式将重试逻辑加入代码，因此他对于构建ZooKeeper应用非常 有用。

### 锁服务

#### ****分布式锁概述****

分布式锁在一组进程之间提供了一种互斥机制。在任何时刻，在任何时刻只有一个进程可以持有锁。分布式锁可以在大型分布式系统中实现领导者选举，在任何时间点，持有锁的那个进程就是系统的领导者。

注意：不要将ZooKeeper自己的领导者选举和使用了ZooKeeper基本操作实现的一般领导者选混为一谈。ZooKeeper自己的领导者选举机制是对外不公开的，我们这里所描述的一般领导者选举服务则不同，他是对那些需要与主进程保持一致的分布式系统所设计的。

(1) 为了使用ZooKeeper来实现分布式锁服务，我们使用顺序znode来为那些竞争锁的进程强制排序。

思路很简单：

① 首先指定一个作为锁的znode，通常用它来描述被锁定的实体，称为/leader；

② 然后希望获得锁的客户端创建一些短暂顺序znode，作为锁znode的子节点。

③ 在任何时间点，顺序号最小的客户端将持有锁。

例如，有两个客户端差不多同时创建znode，分别为/leader/lock-1和/leader/lock-2，那么创建/leader/lock-1的客户端将会持有锁，因为它的znode顺序号最小。ZooKeeper服务是顺序的仲裁者，因为它负责分配顺序号。

④ 通过删除znode /leader/lock-l即可简单地将锁释放；

⑤ 另外，如果客户端进程死亡，对应的短暂znode也会被删除。

⑥ 接下来，创建/leader/lock-2的客户端将持有锁，因为它顺序号紧跟前一个。

⑦ 通过创建一个关于znode删除的观察，可以使客户端在获得锁时得到通知。

(2) 如下是申请获取锁的伪代码。

①在锁znode下创建一个名为lock-的短暂顺序znode，并且记住它的实际路径名(create操作的返回值)。

②查询锁znode的子节点并且设置一个观察。

③如果步骤l中所创建的znode在步骤2中所返回的所有子节点中具有最小的顺序号，则获取到锁。退出。

④等待步骤2中所设观察的通知并且转到步骤2。

#### ****当前问题与方案****

##### ****羊群效应****

(1) 问题

虽然这个算法是正确的，但还是存在一些问题。第一个问题是这种实现会受到“羊群效应”(herd effect)的影响。考虑有成百上千客户端的情况，所有的客户端都在尝试获得锁，每个客户端都会在锁znode上设置一个观察，用于捕捉子节点的变化。每次锁被释放或另外一个进程开始申请获取锁的时候，观察都会被触发并且每个客户端都会收到一个通知。  “羊群效应“就是指大量客户端收到同一事件的通知，但实际上只有很少一部分需要处理这一事件。在这种情况下，只有一个客户端会成功地获取锁，但是维护过程及向所有客户端发送观察事件会产生峰值流量，这会对ZooKeeper服务器造成压力。

(2) 方案解决方案

为了避免出现羊群效应，我们需要优化通知的条件。关键在于只有在前一个顺序号的子节点消失时才需要通知下一个客户端，而不是删除（或创建）任何子节点时都需要通知。在我们的例子中，如果客户端创建了znode /leader/lock-1、/leader/lock-2和／leader/lock-3，那么只有当/leader/lock-2消失时才需要通知／leader/lock-3对照的客户端；/leader/lock-1消失或有新的znode /leader/lock-4加入时，不需要通知该客户端。

##### ****可恢复的异常****

(1) 问题

这个申请锁的算法目前还存在另一个问题，就是不能处理因连接丢失而导致的create操作失败。如前所述，在这种情况下，我们不知道操作是成功还是失败。由于创建一个顺序znode是非幂等操作，所以我们不能简单地重试，因为如果第一次创建已经成功，重试会使我们多出一个永远删不掉的孤儿zriode(至少到客户端会话结束前）。不幸的结果是将会出现死锁。

(2) 解决方案

问题在于，在重新连接之后客户端不能够判断它是否已经创建过子节点。解决方案是在znode的名称中嵌入一个ID，如果客户端出现连接丢失的情况，重新连接之后它便可以对锁节点的所有于节点进行检查，看看是否有子节点的名称中包含其ID。如果有一个子节点的名称包含其ID，它便知道创建操作已经成功，不需要再创建子节点。如果没有子节点的名称中包含其ID，则客户端可以安全地创建一个新的顺序子节点。

客户端会话的ID是一个长整数，并且在ZooKeeper服务中是唯一的，因此非常适合在连接丢失后用于识别客户端。可以通过调用Java ZooKeeper类的getSessionld()方法来获得会话的ID。

在创建短暂顺序znode时应当采用lock-<sessionld>-这样的命名方式，ZooKeeper在其尾部添加顺序号之后，znode的名称会形如lock-<sessionld>-<sequenceNumber>。由于顺序号对于父节点来说是唯一的，但对于子节点名并不唯一，因此采用这样的命名方式可以诖子节点在保持创建顺序的同时能够确定自己的创建者。

##### ****不可恢复的异常****

如果一个客户端的ZooKeeper会话过期，那么它所创建的短暂znode将会被删除，已持有的锁会被释放，或是放弃了申请锁的位置。使用锁的应用程序应当意识到它已经不再持有锁，应当清理它的状态，然后通过创建并尝试申请一个新的锁对象来重新启动。注意，这个过程是由应用程序控制的，而不是锁，因为锁是不能预知应用程序需要如何清理自己的状态。

### ZooKeeper实现共享锁

实现正确地实现一个分布式锁是一件棘手的事，因为很难对所有类型的故障都进行正确的解释处理。ZooKeeper带有一个JavaWriteLock，客户端可以很方便地使用它。更多分布式数据结构和协议例如“屏障”(bafrier)、队列和两阶段提交协议。有趣的是它们都是同步协议，即使我们使用异步ZooKeeper基本操作（如通知）来实现它们。使用ZooKeeper可以实现很多不同的分布式数据结构和协议，ZooKeeper网站([http://hadoop.apache.org/zookeeper/)](http://hadoop.apache.org/zookeeper/)提供了一些用于实现分布式数据结构和协议的伪代码。ZooKeeper本身也带有一些棕准方法的实现，放在安装位置下的recipes目录中。

#### ****场景描述****

大家也许都很熟悉了多个线程或者多个进程间的共享锁的实现方式了，但是在分布式场景中我们会面临多个Server之间的锁的问题。

假设有这样一个场景：两台server ：serverA，serverB需要在C机器上的/usr/local/a.txt文件上进行写操作，如果两台机器同时写该文件，那么该文件的最终结果可能会产生乱序等问题。最先能想到的是serverA在写文件前告诉ServerB “我要开始写文件了，你先别写”，等待收到ServerB的确认回复后ServerA开始写文件，写完文件后再通知ServerB“我已经写完了”。假设在我们场景中有100台机器呢，中间任意一台机器通信中断了又该如何处理？容错和性能问题呢？要能健壮，稳定，高可用并保持高性能，系统实现的复杂度比较高，从头开发这样的系统代价也很大。幸运的是，我们有了基于googlechubby原理开发的开源的ZooKeeper系统。接下来本文将介绍两种ZooKeeper实现分布式共享锁的方法。

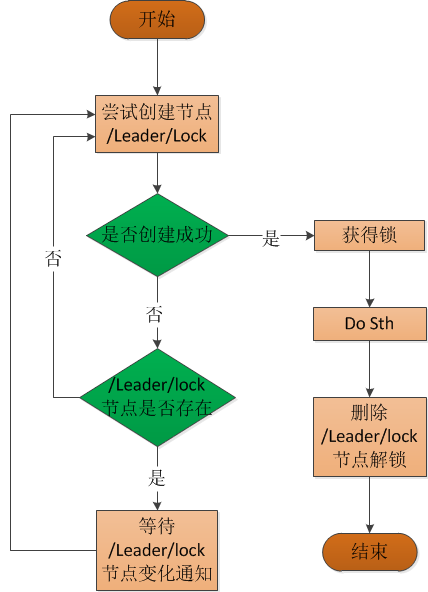
#### ****利用节点名称的唯一性来实现共享锁****

ZooKeeper表面上的节点结构是一个和unix文件系统类似的小型的树状的目录结构，ZooKeeper机制规定：同一个目录下只能有一个唯一的文件名。

例如：我们在Zookeeper目录/test目录下创建，两个客户端创建一个名为lock节点，只有一个能够成功。

(1) 算法思路：利用名称唯一性，加锁操作时，只需要所有客户端一起创建/Leader/lock节点，只有一个创建成功，成功者获得锁。解锁时，只需删除/test/Lock节点，其余客户端再次进入竞争创建节点，直到所有客户端都获得锁。

基于以上机制，利用节点名称唯一性机制的共享锁算法流程如图所示：

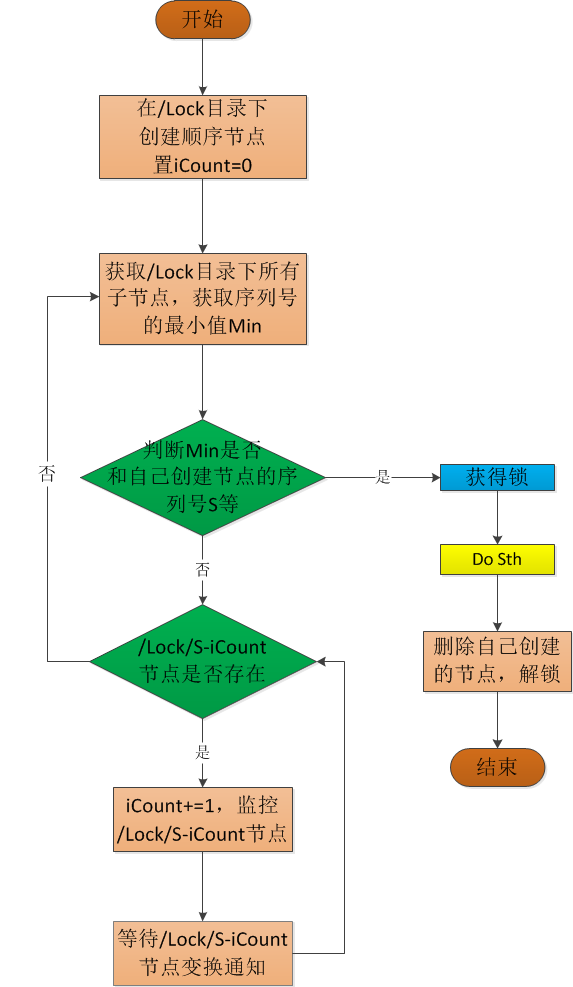


#### ****利用顺序节点实现共享锁****

首先介绍一下，Zookeeper中有一种节点叫做顺序节点，故名思议，假如我们在/lock/目录下创建节3个点，ZooKeeper集群会按照提起创建的顺序来创建节点，节点分别为/lock/0000000001、/lock/0000000002、/lock/0000000003。

ZooKeeper中还有一种名为临时节点的节点，临时节点由某个客户端创建，当客户端与ZooKeeper集群断开连接，。则该节点自动被删除。

算法思路：对于加锁操作，可以让所有客户端都去/lock目录下创建临时、顺序节点，如果创建的客户端发现自身创建节点序列号是/lock/目录下最小的节点，则获得锁。否则，监视比自己创建节点的序列号小的节点（当前序列在自己前面一个的节点），进入等待。解锁操作，只需要将自身创建的节点删除即可。具体算法流程如下图所示:



#### ****ZooKeeper提供的一个写锁实现****

按照ZooKeeper提供的分布式锁的伪代码，实现了一个分布式锁的简单测试代码如下：

**（1）分布式锁，实现了Lock接口 DistributedLock.java**

**package** com.concurrent;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.TimeUnit;

**import** java.util.concurrent.locks.Condition;

**import** java.util.concurrent.locks.Lock;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.ZooDefs;

**import** org.apache.zookeeper.ZooKeeper;

**import** org.apache.zookeeper.data.Stat;

/\*\*

\* DistributedLock lock = null; try { lock = new

\* DistributedLock("127.0.0.1:2182","test"); lock.lock(); //do something... }

\* catch (Exception e) { e.printStackTrace(); } finally { if(lock != null)

\* lock.unlock(); }

\*

\* **@author** xueliang

\*

\*/

**public** **class** DistributedLock **implements** Lock, Watcher {

**private** ZooKeeper zk;

**private** String root = "/locks";// 根

**private** String lockName;// 竞争资源的标志

**private** String waitNode;// 等待前一个锁

**private** String myZnode;// 当前锁

**private** CountDownLatch latch;// 计数器

**private** **int** sessionTimeout = 30000;

**private** List<Exception> exception = **new** ArrayList<Exception>();

/\*\*

\* 创建分布式锁,使用前请确认config配置的zookeeper服务可用

\*

\* **@param** config

\* 127.0.0.1:2181

\* **@param** lockName

\* 竞争资源标志,lockName中不能包含单词lock

\*/

**public** DistributedLock(String config, String lockName) {

**this**.lockName = lockName;

// 创建一个与服务器的连接

**try** {

zk = **new** ZooKeeper(config, sessionTimeout, **this**);

Stat stat = zk.exists(root, **false**);

**if** (stat == **null**) {

// 创建根节点

zk.create(root, **new** **byte**[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

} **catch** (IOException e) {

exception.add(e);

} **catch** (KeeperException e) {

exception.add(e);

} **catch** (InterruptedException e) {

exception.add(e);

}

}

/\*\*

\* zookeeper节点的监视器

\*/

**public** **void** process(WatchedEvent event) {

**if** (**this**.latch != **null**) {

**this**.latch.countDown();

}

}

**public** **void** lock() {

**if** (exception.size() > 0) {

**throw** **new** LockException(exception.get(0));

}

**try** {

**if** (**this**.tryLock()) {

System.*out*.println("Thread " + Thread.*currentThread*().getId()

+ " " + myZnode + " get lock true");

**return**;

} **else** {

waitForLock(waitNode, sessionTimeout);// 等待锁

}

} **catch** (KeeperException e) {

**throw** **new** LockException(e);

} **catch** (InterruptedException e) {

**throw** **new** LockException(e);

}

}

**public** **boolean** tryLock() {

**try** {

String splitStr = "\_lock\_";

**if** (lockName.contains(splitStr))

**throw** **new** LockException("lockName can not contains \\u000B");

// 创建临时子节点

myZnode = zk.create(root + "/" + lockName + splitStr, **new** **byte**[0],

ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.EPHEMERAL\_SEQUENTIAL);

System.*out*.println(myZnode + " is created ");

// 取出所有子节点

List<String> subNodes = zk.getChildren(root, **false**);

// 取出所有lockName的锁

List<String> lockObjNodes = **new** ArrayList<String>();

**for** (String node : subNodes) {

String \_node = node.split(splitStr)[0];

**if** (\_node.equals(lockName)) {

lockObjNodes.add(node);

}

}

Collections.*sort*(lockObjNodes);

System.*out*.println(myZnode + "==" + lockObjNodes.get(0));

**if** (myZnode.equals(root + "/" + lockObjNodes.get(0))) {

// 如果是最小的节点,则表示取得锁

**return** **true**;

}

// 如果不是最小的节点，找到比自己小1的节点

String subMyZnode = myZnode.substring(myZnode.lastIndexOf("/") + 1);

waitNode = lockObjNodes.get(Collections.*binarySearch*(lockObjNodes,

subMyZnode) - 1);

} **catch** (KeeperException e) {

**throw** **new** LockException(e);

} **catch** (InterruptedException e) {

**throw** **new** LockException(e);

}

**return** **false**;

}

**public** **boolean** tryLock(**long** time, TimeUnit unit) {

**try** {

**if** (**this**.tryLock()) {

**return** **true**;

}

**return** waitForLock(waitNode, time);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** **false**;

}

**private** **boolean** waitForLock(String lower, **long** waitTime)

**throws** InterruptedException, KeeperException {

Stat stat = zk.exists(root + "/" + lower, **true**);

// 判断比自己小一个数的节点是否存在,如果不存在则无需等待锁,同时注册监听

**if** (stat != **null**) {

System.*out*.println("Thread " + Thread.*currentThread*().getId()

+ " waiting for " + root + "/" + lower);

**this**.latch = **new** CountDownLatch(1);

**this**.latch.await(waitTime, TimeUnit.*MILLISECONDS*);

**this**.latch = **null**;

}

**return** **true**;

}

**public** **void** unlock() {

**try** {

System.*out*.println("unlock " + myZnode);

zk.delete(myZnode, -1);

myZnode = **null**;

zk.close();

} **catch** (InterruptedException e) {

e.printStackTrace();

} **catch** (KeeperException e) {

e.printStackTrace();

}

}

**public** **void** lockInterruptibly() **throws** InterruptedException {

**this**.lock();

}

**public** Condition newCondition() {

**return** **null**;

}

**public** **class** LockException **extends** RuntimeException {

**private** **static** **final** **long** *serialVersionUID* = 1L;

**public** LockException(String e) {

**super**(e);

}

**public** LockException(Exception e) {

**super**(e);

}

}

}

**（2）并发测试工具 ConcurrentTest.java**

**package** com.concurrent;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** java.util.concurrent.CopyOnWriteArrayList;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.atomic.AtomicInteger;

/\*\*

\* ConcurrentTask[] task = new ConcurrentTask[5]; for(int

\* i=0;i<task.length;i++){ task[i] = new ConcurrentTask(){ public void run() {

\* System.out.println("=============="); }}; } new ConcurrentTest(task);

\*

\* **@author** xueliang

\*

\*/

**public** **class** ConcurrentTest {

**private** CountDownLatch startSignal = **new** CountDownLatch(1);// 开始阀门

**private** CountDownLatch doneSignal = **null**;// 结束阀门

**private** CopyOnWriteArrayList<Long> list = **new** CopyOnWriteArrayList<Long>();

**private** AtomicInteger err = **new** AtomicInteger();// 原子递增

**private** ConcurrentTask[] task = **null**;

**public** ConcurrentTest(ConcurrentTask... task) {

**this**.task = task;

**if** (task == **null**) {

System.*out*.println("task can not null");

System.*exit*(1);

}

doneSignal = **new** CountDownLatch(task.length);

start();

}

/\*\*

\* **@param** args

\* **@throws** ClassNotFoundException

\*/

**private** **void** start() {

// 创建线程，并将所有线程等待在阀门处

createThread();

// 打开阀门

startSignal.countDown();// 递减锁存器的计数，如果计数到达零，则释放所有等待的线程

**try** {

doneSignal.await();// 等待所有线程都执行完毕

} **catch** (InterruptedException e) {

e.printStackTrace();

}

// 计算执行时间

getExeTime();

}

/\*\*

\* 初始化所有线程，并在阀门处等待

\*/

**private** **void** createThread() {

**long** len = doneSignal.getCount();

**for** (**int** i = 0; i < len; i++) {

**final** **int** j = i;

**new** Thread(**new** Runnable() {

**public** **void** run() {

**try** {

startSignal.await();// 使当前线程在锁存器倒计数至零之前一直等待

**long** start = System.*currentTimeMillis*();

task[j].run();

**long** end = (System.*currentTimeMillis*() - start);

list.add(end);

} **catch** (Exception e) {

err.getAndIncrement();// 相当于err++

}

doneSignal.countDown();

}

}).start();

}

}

/\*\*

\* 计算平均响应时间

\*/

**private** **void** getExeTime() {

**int** size = list.size();

List<Long> \_list = **new** ArrayList<Long>(size);

\_list.addAll(list);

Collections.*sort*(\_list);

**long** min = \_list.get(0);

**long** max = \_list.get(size - 1);

**long** sum = 0L;

**for** (Long t : \_list) {

sum += t;

}

**long** avg = sum / size;

System.*out*.println("min: " + min);

System.*out*.println("max: " + max);

System.*out*.println("avg: " + avg);

System.*out*.println("err: " + err.get());

}

**public** **interface** ConcurrentTask {

**void** run();

}

}

**（3）测试  ZkTest.java**

**package** com.concurrent;

**import** com.concurrent.ConcurrentTest.ConcurrentTask;

**public** **class** ZkTest {

**public** **static** **void** main(String[] args) {

Runnable task1 = **new** Runnable() {

**public** **void** run() {

DistributedLock lock = **null**;

**try** {

lock = **new** DistributedLock("127.0.0.1:2182", "test1");

// lock = new DistributedLock("127.0.0.1:2182","test2");

lock.lock();

Thread.*sleep*(3000);

System.*out*.println("===Thread "

+ Thread.*currentThread*().getId() + " running");

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (lock != **null**)

lock.unlock();

}

}

};

**new** Thread(task1).start();

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

ConcurrentTask[] tasks = **new** ConcurrentTask[60];

**for** (**int** i = 0; i < tasks.length; i++) {

ConcurrentTask task3 = **new** ConcurrentTask() {

**public** **void** run() {

DistributedLock lock = **null**;

**try** {

lock = **new** DistributedLock("127.0.0.1:2183", "test2");

lock.lock();

System.*out*.println("Thread "

+ Thread.*currentThread*().getId() + " running");

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

lock.unlock();

}

}

};

tasks[i] = task3;

}

**new** ConcurrentTest(tasks);

}

}

#### ****更多分布式数据结构和协议****

使用ZooKeeper可以实现很多不同的分布式数据结构和协议，例如“屏障”(bafrier)、队列和两阶段提交协议。有趣的是它们都是同步协议，即使我们使用异步ZooKeeper基本操作（如通知）来实现它们。

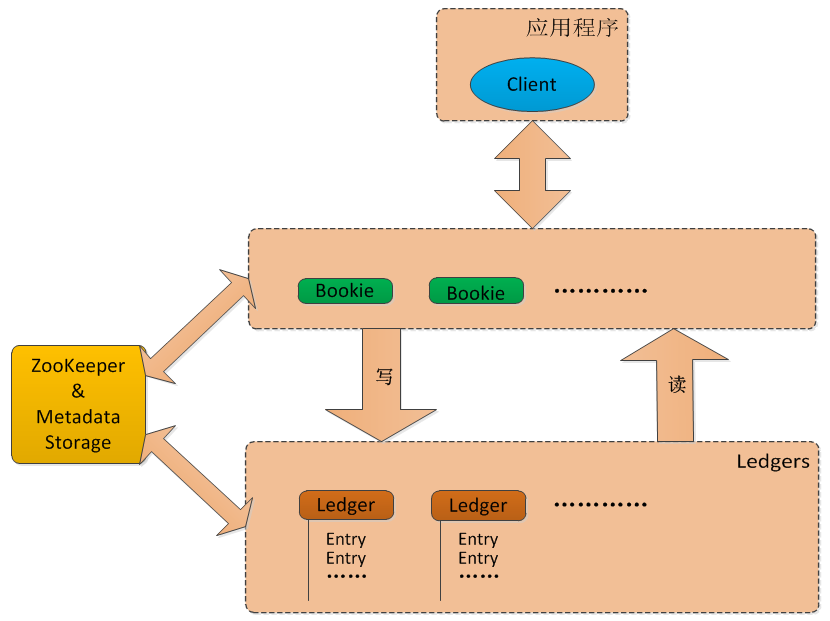
ZooKeeper网站(<http://hadoop.apache.org/zookeeper>)提供了一些用于实现分布式数据结构和协议的伪代码。ZooKeeper本身也带有一些棕准方法的实现，放在安装位置下的recipes目录中。

### BooKeeper

#### ****BooKeeper概述****

BooKeeper具有副本功能，目的是提供可靠的日志记录。在BooKeeper中，服务器被称为账本（Bookies），在账本之中有不同的账户（Ledgers），每一个账户由一条条记录（Entry）组成。如果使用普通的磁盘存储日志数据，那么日志数据可能遭到破坏，当磁盘发生故障的时候，日志也可能被丢失。BooKeeper为每一份日志提供了分布式的存储，并采用了大多数（quorum，相对于全体）的概念。也就是说，只要集群中的大多数机器可用，那么该日志一直有效。

BooKeeper通过客户端进行操作，客户端可以对BooKeeper进行添加账户、打开账户、添加账户记录、读取账户记录等操作。另外，BooKeeper的服务依赖于ZooKeeper，可以说BooKeeper依赖于ZooKeeper的一致性及其分布式特点，在其之上提供另外一种可靠性服务。BooKeeper的架构如下图所示：



#### ****BooKeeper角色****

从上图中可以看出，BooKeeper中总共包含四类角色：

① 账本：Bookies

② 账户：Ledger

③ 客户端：Client

④ 元数据及存储服务：Metadata Storage Service

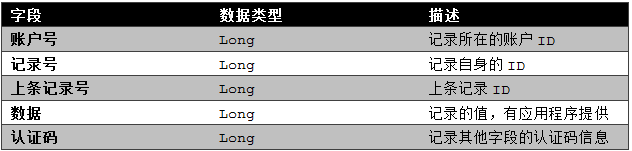
下面简单介绍这四类角色的功能:

(1) 账本 BooKies

账本是BooKeeper的存储服务器，他存储的是一个个的账本，可以将账本理解为一个个节点。在一个BooKeeper系统中存在多个账本（节点），每个账户被不同的账本所存储。若要写一条记录到指定的账户中，该记录将被写到维护该账户所有帐本节点中。为了提高系统的性能，这条记录并不是真正的被写入到所有的节点中，而是选择集群的一个大多数集进行存储。该系统独有的特性，使得BooKeeper系统有良好的扩展性。即，我们可以通过简单的添加机器节点的方法提高系统容量。☆☆

(2) 账户 Ledger

账户中存储的是一系列记录，每一条记录包含一定的字段。记录通过写操作一次性写入，只能进行附加操作不能进行修改。每条记录包含如下字段：



当满足下列两个条件时，某条记录才被认为是存储成功：

① 之前所记录的数据被账本节点的大多数集所存储。

② 该记录被账本节点的大多数集所存储。

(3) 客户端 BooKeeper Client

客户端通常与BooKeeper应用程序进行交互，它允许应用程序在系统上进行操作，包括创建账户，写账户等。

(4) 元数据存储服务 Metadata Storage Service

元数据信息存储在ZooKeeper集群当中，它存储关于账户和账本的信息。例如，账本由集群中的哪些节点进行维护，账户由哪个账本进行维护。应用程序在使用账本的时候，首先要创建一个账户。在创建账户时，系统首先将该账本的Metadata信息写入到ZooKeeper中。每一个账户在某一时刻只能有一个写实例（分布式锁）。在其他实例进行读操作之前首先需要将写实例关闭。如果写操作实例由于故障未能正常关闭，那么下一个尝试打开账户的实例将需要首先对其进行恢复，并正确关闭写操作。在进行写操作的同时需要将最后一次的写记录存储到ZooKeeper中，因此恢复程序仅需要在ZooKeeper中查看该账户所对应的最后一条写记录，然后将其正确的写入到账户中，再在正确关闭写操作。在BooKeeper中该恢复程序有系统自动执行不需要用户参与。

## ZooKeeper管理分布式环境中的数据

### ZooKeeper产生背景

#### ****分布式的发展****

**分布式**这个概念我想大家并不陌生，但真正实战开始还要从google说起，很早以前在实验室中分布式被人提出，可是说是计算机内入行较为复杂学习较为困难的技术，并且市场也并不成熟，因此大规模的商业应用一直未成出现，但从Google 发布了MapReduce 和DFS 以及Bigtable的论文之后，分布式在计算机界的格局就发生了变化，从架构上实现了分布式的难题，并且成熟的应用在了海量数据存储和计算上，其集群的规模也是当前世界上最为庞大的。

以DFS 为基础的分布式计算框架和key、value 数据高效的解决**运算的瓶颈**， 而且开发人员不用再写复杂的分布式程序，只要底层框架完备开发人员只要用较少的代码就可以完成分布式程序的开发，这使得开发人员只需要关注业务逻辑的即 可。Google 在业界技术上的领军地位，让业界望尘莫及的技术实力，IT 因此也是对Google 所退出的技术十分推崇。在最近几年中分布式则是成为了海量数据存储以及计算、高并发、高可靠性、高可用性的解决方案。

#### ****ZooKeeper的产生****

众所周知通常分布式架构都是中心化的设计，就是一个主控机连接多个处理节点。 问题可以从这里考虑，当主控机失效时，整个系统则就无法访问了，所以保证系统的高可用性是非常关键之处，也就是要保证主控机的高可用性。分布式锁就是一个 解决该问题的较好方案，多主控机抢一把锁。在这里我们就涉及到了我们的重点Zookeeper。

ZooKeeper是什么，chubby 我想大家都不会陌生的，chubby 是实现Google 的一个分布式锁的实现，运用到了paxos 算法解决的一个分布式事务管理的系统。Zookeeper 就是雅虎模仿强大的Google chubby 实现的一套分布式锁管理系统。同时，Zookeeper 分布式服务框架是Apache Hadoop的一个子项目，它是一个针对大型分布式系统的可靠协调系统，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，可以高可靠的维护元数据。**提供的功能**包括：配置维护、名字服务、分布式同步、组服务等。ZooKeeper的设计目标就是**封装**好复杂易出错的**关键服务**，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

#### ****ZooKeeper的使用****

Zookeeper 作为一个分布式的服务框架，主要用来解决分布式集群中应用系统的一致性问题，它能提供基于类似于文件系统的目录节点树方式的数据存储，但是 Zookeeper 并不是用来专门存储数据的，它的作用主要是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，后面将 会详细介绍 Zookeeper 能够解决的一些典型问题。

**注意一下**这里的"**数据**"是有限制的：

**(1)** **从数据大小来看**：我们知道ZooKeeper的数据存储在一个叫ReplicatedDataBase 的 数据库中，该数据是一个内存数据库，既然是在内存当中，我就应该知道该数据量就应该不会太大，这一点上就与hadoop的HDFS有了很大的区 别，HDFS的数据主要存储在磁盘上，因此数据存储主要是HDFS的事，而ZooKeeper主要是协调功能，并不是用来存储数据的。

**(2) 从数据类型来看：**正如前面所说的，ZooKeeper的数据在内存中，由于内存空间的限制，那么我们就不能在上面随心所欲的存储数据，所以ZooKeeper存储的数据都是我们所关心的数据而且数据量还不能太大，而且还会根据我们要以实现的功能来选择相应的数据。简单来说，干什么事存什么数据，ZooKeeper所实现的一切功能，都是由ZK节点的性质和该节点所关联的数据实现的，至于关联什么数据那就要看你干什么事了。

例如：

**①** **集群管理**：利用临时节点特性，节点关联的是机器的主机名、IP地址等相关信息，集群单点故障也属于该范畴。

**② 统一命名：**主要利用节点的唯一性和目录节点树结构。

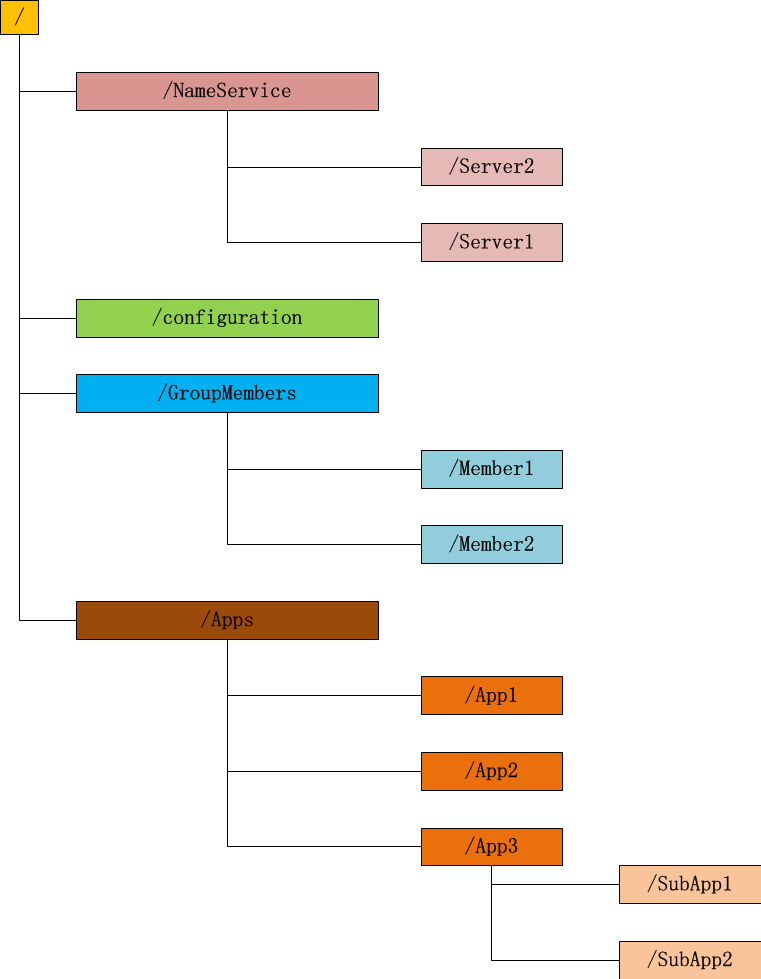
**③ 配置管理：**节点关联的是配置信息。

**④ 分布式锁：**节点关联的是要竞争的资源。

### ZooKeeper应用场景

ZooKeeper是一个高可用的分布式数据管理与系统协调框架。基于对Paxos算法的实现，使该框架保证了分布式环境中数据的强一致性，也正是 基于这样的特性，使得zookeeper能够应用于很多场景。需要注意的是，ZK并不是生来就为这些场景设计，都是后来众多开发者根据框架的特性，摸索出 来的典型使用方法。因此，我们也可以根据自己的需要来设计相应的场景实现。正如前文所提到的，ZooKeeper 实现的任何功能都离不开ZooKeeper的数据结构，任何功能的实现都是利用"Znode结构特性+节点关联的数据"来实现的，好吧那么我们就看一下ZooKeeper数据结构有哪些特性。ZooKeeper数据结构如下图所示：

**图2.1 ZooKeeper数据结构**



Zookeeper 这种数据结构有如下这些特点：

**①** 每个子目录项如 NameService 都被称作为 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中可以存储多份数据；

**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。

#### ****数据发布与订阅****

**(1) 典型场景描述**

发布与订阅即所谓的配置管理，顾名思义就是将数据发布到ZK节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如**全局的配置信息**，**地址列表**等就非常适合使用。集中式的配置管理在应用集群中是非常常见的，一般商业公司内部都会实现一套集中的配置管理中心，应对不同的应用集群对于共享各自配置的需求，并且在配置变更时能够通知到集群中的每一个机器。

**(2) 应用**

**①** 索引信息和集群中机器节点状态存放在ZK的一些指定节点，供各个客户端订阅使用。

**②** 系统日志（经过处理后的）存储，这些日志通常2-3天后被清除。

**③** 应用中用到的一些配置信息集中管理，在应用启动的时候主动来获取一次，并且在节点上注册一个Watcher，以后每次配置有更新，实时通知到应用，获取最新配置信息。

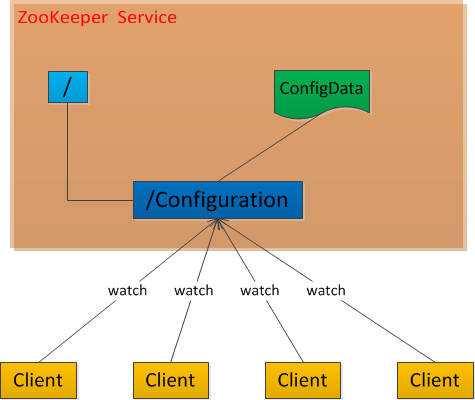
**④** 业务逻辑中需要用到的一些全局变量，比如一些消息中间件的消息队列通常有个offset，这个offset存放在zk上，这样集群中每个发送者都能知道当前的发送进度。

**⑤** 系统中有些信息需要动态获取，并且还会存在人工手动去修改这个信息。以前通常是暴露出接口，例如JMX接口，有了ZK后，只要将这些信息存放到ZK节点上即可。

**(3) 应用举例**

**例如**：同一个应用系统需要多台 PC Server 运行，但是它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的配置项，那么就必须同时修改每台运行这个应用系统的 PC Server，这样非常麻烦而且容易出错。将配置信息保存在 Zookeeper 的某个目录节点中，然后将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。ZooKeeper配置管理服务如下图所示：

**图2.2 配置管理结构图**



Zookeeper很容易实现这种集中式的配置管理，比如将所需要的配置信息放到**/Configuration** 节点上，集群中所有机器一启动就会通过Client对**/Configuration**这个节点进行监控【zk.exist("/Configuration″,true)】，并且实现Watcher回调方法process()，那么在zookeeper上**/Configuration**节点下数据发生变化的时候，每个机器都会收到通知，Watcher回调方法将会被执行，那么应用再取下数据即可【zk.getData("/Configuration″,false,null)】。

#### ****统一命名服务（Name Service）****

**(1) 场景描述**

分布式应用中，通常需要有一套完整的命名规则，既能够产生唯一的名称又便于人识别和记住，通常情况下用树形的名称结构是一个理想的选择，树形的名称 结构是一个有层次的目录结构，既对人友好又不会重复。说到这里你可能想到了 JNDI，没错 Zookeeper 的 Name Service 与 JNDI 能够完成的功能是差不多的，它们都是将有层次的目录结构关联到一定资源上，但是Zookeeper的Name Service 更加是广泛意义上的关联，也许你并不需要将名称关联到特定资源上，你可能只需要一个不会重复名称，就像数据库中产生一个唯一的数字主键一样。

**(2) 应用**

在分布式系统中，通过使用命名服务，客户端应用能够根据指定的名字来获取资源服务的地址，提供者等信息。被命名的实体通常可以是集群中的机器，提供的服务地址，进程对象等等，这些我们都可以统称他们为名字（Name）。其中较为常见的就是一些分布式服务框架中的服务地址列表。 通过调用ZK提供的创建节点的API，能够很容易创建一个全局唯一的path，这个path就可以作为一个名称。Name Service 已经是Zookeeper 内置的功能，你只要调用 Zookeeper 的 API 就能实现。如调用 create 接口就可以很容易创建一个目录节点。

**(3) 应用举例**

阿里开源的分布式服务框架Dubbo中使用ZooKeeper来作为其命名服务，维护全局的服务地址列表。在Dubbo实现中： **服务提供者**在启动的时候，向ZK上的指定节点/dubbo/${serviceName}/providers目录下写入自己的URL地址，这个操作就完成了服务的发布。 **服务消费者**启 动的时候，订阅/dubbo/${serviceName}/providers目录下的提供者URL地址， 并向/dubbo/${serviceName} /consumers目录下写入自己的URL地址。 注意，所有向ZK上注册的地址都是临时节点，这样就能够保证服务提供者和消费者能够自动感应资源的变化。 另外，Dubbo还有针对服务粒度的监控，方法是订阅/dubbo/${serviceName}目录下所有提供者和消费者的信息。

#### ****分布通知/协调（Distribution of notification/coordination）****

**(1) 典型场景描述**

ZooKeeper中特有watcher注册与异步通知机制，能够很好的实现分布式环境下不同系统之间的通知与协调，实现对数据变更的实时处理。使用方法通常是不同系统都对ZK上同一个znode进行注册，监听znode的变化（包括znode本身内容及子节点的），其中一个系统update了znode，那么另一个系统能够收到通知，并作出相应处理。

**(2) 应用**

**①** 另一种**心跳检测机制**：检测系统和被检测系统之间并不直接关联起来，而是通过ZK上某个节点关联，大大减少系统耦合。

**②** 另一种**系统调度模式**：某系统由控制台和推送系统两部分组成，控制台的职责是控制推送系统进行相应的推送工作。管理人员在控制台作的一些操作，实际上是修改了ZK上某些节点的状态，而ZK就把这些变化通知给他们注册Watcher的客户端，即推送系统，于是，作出相应的推送任务。

**③** 另一种**工作汇报模式**：一些类似于任务分发系统，子任务启动后，到ZK来注册一个临时节点，并且定时将自己的进度进行汇报（将进度写回这个临时节点），这样任务管理者就能够实时知道任务进度。

总之，使用zookeeper来进行分布式通知和协调能够大大降低系统之间的耦合。

#### ****分布式锁（Distribute Lock）****

**(1) 场景描述**

**分布式锁**，这个主要得益于ZooKeeper为我们保证了**数据的强一致性**，即用户只要完全相信每时每刻，zk集群中任意节点（一个zk server）上的相同znode的数据是一定是相同的。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

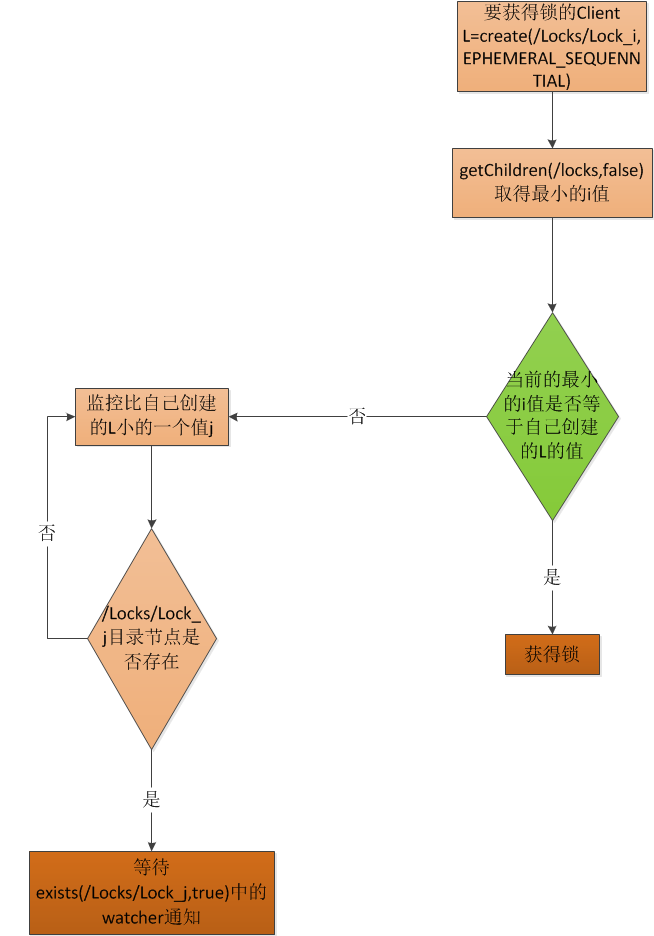
保持独占，就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把 锁。通常的做法是把ZK上的一个znode看作是一把锁，通过create znode的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。

控制时序，就是所有试图来获取这个锁的客户端，最终都是会被安排执行，只是有 个全局时序了。做法和上面基本类似，只是这里 /distribute\_lock 已经预先存在，客户端在它下面创建临时有序节点。Zk的父节点（/distribute\_lock）维持一份sequence,保证子节点创建的时序性， 从而也形成了每个客户端的全局时序。

**(2) 应用**

共享锁在同一个进程中很容易实现，但是在跨进程或者在不同 Server 之间就不好实现了。Zookeeper 却很容易实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL\_SEQUENTIAL 目录节点，然后调用 getChildren方法获取当前的目录节点列表中最小的目录节点是不是就是自己创建的目录节点，如果正是自己创建的，那么它就获得了这个锁，如果不是那么它就调用 exists(String path, boolean watch) 方法并监控 Zookeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，从而获得锁，释放锁很简单，只要删除前面它自己所创建的目录节点就行了。

**图 2.3 ZooKeeper实现Locks的流程图**



**代码清单1 TestMainClient 代码**

**package** org.zk.leader.election;

**import** org.apache.log4j.xml.DOMConfigurator;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.ZooKeeper;

**import** java.io.IOException;

/\*\*

\* TestMainClient

\* <p/>

\* Author By: sunddenly工作室 Created Date: 2014-11-13

\*/

**public** **class** TestMainClient **implements** Watcher {

**protected** **static** ZooKeeper *zk* = **null**;

**protected** **static** Integer *mutex*;

**int** sessionTimeout = 10000;

**protected** String root;

**public** TestMainClient(String connectString) {

**if** (*zk* == **null**) {

**try** {

String configFile = **this**.getClass().getResource("/").getPath()

+ "org/zk/leader/election/log4j.xml";

DOMConfigurator.configure(configFile);

System.*out*.println("创建一个新的连接:");

*zk* = **new** ZooKeeper(connectString, sessionTimeout, **this**);

*mutex* = **new** Integer(-1);

} **catch** (IOException e) {

*zk* = **null**;

}

}

}

**synchronized** **public** **void** process(WatchedEvent event) {

**synchronized** (*mutex*) {

*mutex*.notify();

}

}

}

**清单 2 Locks 代码**

**package** org.zk.locks;

**import** org.apache.log4j.Logger;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.ZooDefs;

**import** org.apache.zookeeper.data.Stat;

**import** org.zk.leader.election.TestMainClient;

**import** java.util.Arrays;

**import** java.util.List;

/\*\*

\* locks

\* <p/>

\* Author By: sunddenly工作室 Created Date: 2014-11-13 16:49:40

\*/

**public** **class** Locks **extends** TestMainClient {

**public** **static** **final** Logger *logger* = Logger.getLogger(Locks.**class**);

String myZnode;

**public** Locks(String connectString, String root) {

**super**(connectString);

**this**.root = root;

**if** (zk != **null**) {

**try** {

Stat s = zk.exists(root, **false**);

**if** (s == **null**) {

zk.create(root, **new** **byte**[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

} **catch** (KeeperException e) {

*logger*.error(e);

} **catch** (InterruptedException e) {

*logger*.error(e);

}

}

}

**void** getLock() **throws** KeeperException, InterruptedException {

List<String> list = zk.getChildren(root, **false**);

String[] nodes = list.toArray(**new** String[list.size()]);

Arrays.*sort*(nodes);

**if** (myZnode.equals(root + "/" + nodes[0])) {

doAction();

} **else** {

waitForLock(nodes[0]);

}

}

**void** check() **throws** InterruptedException, KeeperException {

myZnode = zk.create(root + "/lock\_", **new** **byte**[0],

ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);

getLock();

}

**void** waitForLock(String lower) **throws** InterruptedException, KeeperException {

Stat stat = zk.exists(root + "/" + lower, **true**);

**if** (stat != **null**) {

mutex.wait();

} **else** {

getLock();

}

}

@Override

**public** **void** process(WatchedEvent event) {

**if** (event.getType() == Event.EventType.NodeDeleted) {

System.*out*.println("得到通知");

**super**.process(event);

doAction();

}

}

/\*\*

\* 执行其他任务

\*/

**private** **void** doAction() {

System.*out*.println("同步队列已经得到同步，可以开始执行后面的任务了");

}

**public** **static** **void** main(String[] args) {

String connectString = "localhost:2181";

Locks lk = **new** Locks(connectString, "/locks");

**try** {

lk.check();

} **catch** (InterruptedException e) {

*logger*.error(e);

} **catch** (KeeperException e) {

*logger*.error(e);

}

}

}

#### ****集群管理（Cluster Management）****

**(1) 典型场景描述**

**集群机器监控**：

这通常用于那种对集群中机器状态，机器在线率有较高要求的场景，能够快速对集群中机器变化作出响应。这样的场景中，往往有一个监控系统，实时检测集 群机器是否存活。过去的做法通常是：监控系统通过某种手段（比如ping）定时检测每个机器，或者每个机器自己定时向监控系统汇报"我还活着"。 这种做法可行，但是存在两个比较明显的问题：

**①** 集群中机器有变动的时候，牵连修改的东西比较多。

**②** 有一定的延时。

利用ZooKeeper中两个特性，就可以实施另一种集群机器存活性监控系统：

**①** 客户端在节点 x 上注册一个Watcher，那么如果 x 的子节点变化了，会通知该客户端。

**②** 创建EPHEMERAL类型的节点，一旦客户端和服务器的会话结束或过期，那么该节点就会消失。

**Master选举：**

Master选举则是zookeeper中最为经典的使用场景了，在分布式环境中，相同的业务应用分布在不同的机器上，有些业务逻辑，**例如**一些耗时的计算，网络I/O处，往往只需要让整个集群中的某一台机器进行执行，其余机器可以共享这个结果，这样可以大大减少重复劳动，提高性能，于是这个master选举便是这种场景下的碰到的主要问题。

利用ZooKeeper中两个特性，就可以实施另一种集群中Master选举：

**①** 利用ZooKeeper的强一致性，能够保证在分布式高并发情况下节点创建的全局唯一性，即：同时有多个客户端请求创建 /Master 节点，最终一定只有一个客户端请求能够创建成功。利用这个特性，就能很轻易的在分布式环境中进行集群选举了。

**②**另外，这种场景演化一下，就是动态Master选举。这就要用到 EPHEMERAL\_SEQUENTIAL类型节点的特性了，这样每个节点会自动被编号。允许所有请求都能够创建成功，但是得有个创建顺序，每次选取序列号最小的那个机器作为Master 。

**(2) 应用**

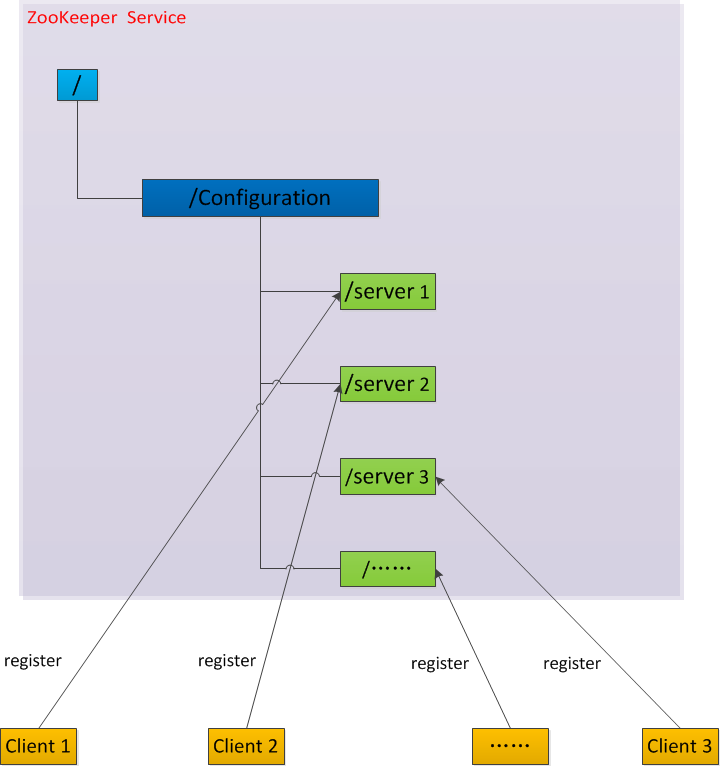
在搜索系统中，如果集群中每个机器都生成一份全量索引，不仅耗时，而且不能保证彼此间索引数据一致。因此让集群中的Master来迚行全量索引的生 成，然后同步到集群中其它机器。另外，Master选丼的容灾措施是，可以随时迚行手动指定master，就是说应用在zk在无法获取master信息 时，可以通过比如http方式，向一个地方获取master。 λ 在Hbase中，也是使用ZooKeeper来实现动态HMaster的选举。在Hbase实现中，会在ZK上存储一些ROOT表的地址和HMaster 的地址，HRegionServer也会把自己以临时节点（Ephemeral）的方式注册到Zookeeper中，使得HMaster可以随时感知到各 个HRegionServer的存活状态，同时，一旦HMaster出现问题，会重新选丼出一个HMaster来运行，从而避免了HMaster的单点问 题的存活状态，同时，一旦HMaster出现问题，会重新选丼出一个HMaster来运行，从而避免了HMaster的单点问题。

**(3) 应用举例**

**集群监控：**

应用集群中，我们常常需要让每一个机器知道集群中或依赖的其他某一个集群中哪些机器是活着的，并且在集群机器因为宕机，网络断链等原因能够不在人工 介入的情况下迅速通知到每一个机器，Zookeeper 能够很容易的实现集群管理的功能，如有多台 Server 组成一个服务集群，那么必须要一个"总管"知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，集群中其它集群必须知道，从而做出调整重新分配服 务策略。同样当增加集群的服务能力时，就会增加一台或多台 Server，同样也必须让"总管"知道，这就是ZooKeeper的集群监控功能。

**图2.4 集群管理结构图**



比如我在zookeeper服务器端有一个znode叫**/Configuration**，那么集群中每一个机器启动的时候都去这个节点下创建一个EPHEMERAL类型的节点，比如server1创建**/Configuration** /Server1，server2创建**/Configuration** /Server1，然后Server1和Server2都watch /Configuration 这个父节点，那么也就是这个父节点下数据或者子节点变化都会通知对该节点进行watch的客户端。因为EPHEMERAL类型节点有一个很重要的特性，就 是客户端和服务器端连接断掉或者session过期就会使节点消失，那么在某一个机器挂掉或者断链的时候，其对应的节点就会消 失，然后集群中所有对**/Configuration**进行watch的客户端都会收到通知，然后取得最新列表即可。

**Master选举：**

Zookeeper 不仅能够维护当前的集群中机器的服务状态，而且能够选出一个"总管"，让这个总管来管理集群，这就是 Zookeeper 的另一个功能 Leader Election。Zookeeper 如何实现 Leader Election，也就是选出一个 Master Server。和前面的一样每台 Server 创建一个 EPHEMERAL 目录节点，不同的是它还是一个 SEQUENTIAL 目录节点，所以它是个 EPHEMERAL\_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL\_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号，我们可以选择当前是最小编号的 Server 为 Master，假如这个最小编号的 Server 死去，由于是 EPHEMERAL 节点，死去的 Server 对应的节点也被删除，所以当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免了传统意义上单 Master 容易出现单点故障的问题**。**

**清单 3 Leader Election代码**

**package** org.zk.leader.election;

**import** org.apache.log4j.Logger;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.ZooDefs;

**import** org.apache.zookeeper.data.Stat;

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

/\*\*

\* LeaderElection

\* <p/>

\* Author By: sunddenly工作室 Created Date: 2014-11-13

\*/

**public** **class** LeaderElection **extends** TestMainClient {

**public** **static** **final** Logger *logger* = Logger.getLogger(LeaderElection.**class**);

**public** LeaderElection(String connectString, String root) {

**super**(connectString);

**this**.root = root;

**if** (zk != **null**) {

**try** {

Stat s = zk.exists(root, **false**);

**if** (s == **null**) {

zk.create(root, **new** **byte**[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

} **catch** (KeeperException e) {

*logger*.error(e);

} **catch** (InterruptedException e) {

*logger*.error(e);

}

}

}

**void** findLeader() **throws** InterruptedException, UnknownHostException,

KeeperException {

**byte**[] leader = **null**;

**try** {

leader = zk.getData(root + "/leader", **true**, **null**);

} **catch** (KeeperException e) {

**if** (e **instanceof** KeeperException.NoNodeException) {

*logger*.error(e);

} **else** {

**throw** e;

}

}

**if** (leader != **null**) {

following();

} **else** {

String newLeader = **null**;

**byte**[] localhost = InetAddress.*getLocalHost*().getAddress();

**try** {

newLeader = zk.create(root + "/leader", localhost,

ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

} **catch** (KeeperException e) {

**if** (e **instanceof** KeeperException.NodeExistsException) {

*logger*.error(e);

} **else** {

**throw** e;

}

}

**if** (newLeader != **null**) {

leading();

} **else** {

mutex.wait();

}

}

}

@Override

**public** **void** process(WatchedEvent event) {

**if** (event.getPath().equals(root + "/leader")

&& event.getType() == Event.EventType.NodeCreated) {

System.*out*.println("得到通知");

**super**.process(event);

following();

}

}

**void** leading() {

System.*out*.println("成为领导者");

}

**void** following() {

System.*out*.println("成为组成员");

}

**public** **static** **void** main(String[] args) {

String connectString = "localhost:2181";

LeaderElection le = **new** LeaderElection(connectString, "/GroupMembers");

**try** {

le.findLeader();

} **catch** (Exception e) {

*logger*.error(e);

}

}

}

#### ****队列管理****

Zookeeper 可以处理两种类型的队列：

**①** 当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种是同步队列。

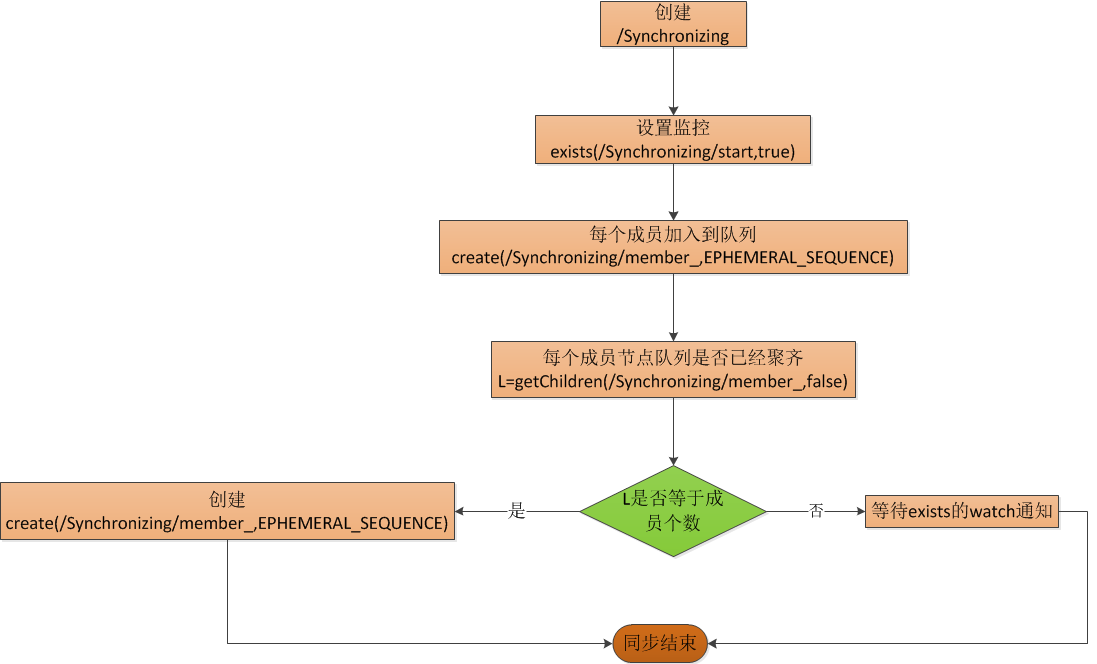
**②** 队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

**(1)** 同步队列用 Zookeeper 实现的实现思路如下：

创建一个父目录 /synchronizing，每个成员都监控标志（Set Watch）位目录 /synchronizing/start 是否存在，然后每个成员都加入这个队列，加入队列的方式就是创建 /synchronizing/member\_i 的临时目录节点，然后每个成员获取 / synchronizing 目录的所有目录节点，也就是 member\_i。判断 i 的值是否已经是成员的个数，如果小于成员个数等待 /synchronizing/start 的出现，如果已经相等就创建 /synchronizing/start。

用下面的流程图更容易理解：

**图 2.5 同步队列流程图**



**清单 4 Synchronizing 代码**

**package** org.zk.queue;

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

**import** java.util.List;

**import** org.apache.log4j.Logger;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.ZooKeeper;

**import** org.apache.zookeeper.ZooDefs.Ids;

**import** org.apache.zookeeper.data.Stat;

**import** org.zk.leader.election.TestMainClient;

/\*\*

\* Synchronizing

\* <p/>

\* Author By: sunddenly工作室 Created Date: 2014-11-13

\*/

**public** **class** Synchronizing **extends** TestMainClient {

**int** size;

String name;

**public** **static** **final** Logger *logger* = Logger.getLogger(Synchronizing.**class**);

/\*\*

\* 构造函数

\*

\* **@param** connectString

\* 服务器连接

\* **@param** root

\* 根目录

\* **@param** size

\* 队列大小

\*/

Synchronizing(String connectString, String root, **int** size) {

**super**(connectString);

**this**.root = root;

**this**.size = size;

**if** (zk != **null**) {

**try** {

Stat s = zk.exists(root, **false**);

**if** (s == **null**) {

zk.create(root, **new** **byte**[0], Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

} **catch** (KeeperException e) {

*logger*.error(e);

} **catch** (InterruptedException e) {

*logger*.error(e);

}

}

**try** {

name = **new** String(InetAddress.*getLocalHost*().getCanonicalHostName()

.toString());

} **catch** (UnknownHostException e) {

*logger*.error(e);

}

}

/\*\*

\* 加入队列

\*

\* **@return**

\* **@throws** KeeperException

\* **@throws** InterruptedException

\*/

**void** addQueue() **throws** KeeperException, InterruptedException {

zk.exists(root + "/start", **true**);

zk.create(root + "/" + name, **new** **byte**[0], Ids.OPEN\_ACL\_UNSAFE,

CreateMode.EPHEMERAL\_SEQUENTIAL);

**synchronized** (mutex) {

List<String> list = zk.getChildren(root, **false**);

**if** (list.size() < size) {

mutex.wait();

} **else** {

zk.create(root + "/start", **new** **byte**[0], Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

}

}

@Override

**public** **void** process(WatchedEvent event) {

**if** (event.getPath().equals(root + "/start")

&& event.getType() == Event.EventType.NodeCreated) {

System.*out*.println("得到通知");

**super**.process(event);

doAction();

}

}

/\*\*

\* 执行其他任务

\*/

**private** **void** doAction() {

System.*out*.println("同步队列已经得到同步，可以开始执行后面的任务了");

}

**public** **static** **void** main(String args[]) {

// 启动Server

String connectString = "localhost:2181";

**int** size = 1;

Synchronizing b = **new** Synchronizing(connectString, "/synchronizing",

size);

**try** {

b.addQueue();

} **catch** (KeeperException e) {

*logger*.error(e);

} **catch** (InterruptedException e) {

*logger*.error(e);

}

}

}

**(2)** FIFO 队列用 Zookeeper 实现思路如下：

实现的思路也非常简单，就是在特定的目录下创建 SEQUENTIAL 类型的子目录 /queue\_i，这样就能保证所有成员加入队列时都是有编号的，出队列时通过 getChildren( ) 方法可以返回当前所有的队列中的元素，然后消费其中最小的一个，这样就能保证 FIFO。

下面是生产者和消费者这种队列形式的示例代码

**清单 5 FIFOQueue 代码**

**package** org.zk.queue;

**import** org.apache.log4j.Logger;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.ZooDefs;

**import** org.apache.zookeeper.data.Stat;

**import** java.nio.ByteBuffer;

**import** java.util.List;

/\*\*

\* FIFOQueue

\* <p/>

\* Author By: sunddenly工作室 Created Date: 2014-11-13

\*/

**public** **class** FIFOQueue **extends** TestMainClient {

**public** **static** **final** Logger *logger* = Logger.getLogger(FIFOQueue.**class**);

/\*\*

\* Constructor

\*

\* **@param** connectString

\* **@param** root

\*/

FIFOQueue(String connectString, String root) {

**super**(connectString);

**this**.root = root;

**if** (zk != **null**) {

**try** {

Stat s = zk.exists(root, **false**);

**if** (s == **null**) {

zk.create(root, **new** **byte**[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT);

}

} **catch** (KeeperException e) {

*logger*.error(e);

} **catch** (InterruptedException e) {

*logger*.error(e);

}

}

}

/\*\*

\* 生产者

\*

\* **@param** i

\* **@return**

\*/

**boolean** produce(**int** i) **throws** KeeperException, InterruptedException {

ByteBuffer b = ByteBuffer.*allocate*(4);

**byte**[] value;

b.putInt(i);

value = b.array();

zk.create(root + "/element", value, ZooDefs.Ids.OPEN\_ACL\_UNSAFE,

CreateMode.PERSISTENT\_SEQUENTIAL);

**return** **true**;

}

/\*\*

\* 消费者

\*

\* **@return**

\* **@throws** KeeperException

\* **@throws** InterruptedException

\*/

**int** consume() **throws** KeeperException, InterruptedException {

**int** retvalue = -1;

Stat stat = **null**;

**while** (**true**) {

**synchronized** (mutex) {

List<String> list = zk.getChildren(root, **true**);

**if** (list.size() == 0) {

mutex.wait();

} **else** {

Integer min = **new** Integer(list.get(0).substring(7));

**for** (String s : list) {

Integer tempValue = **new** Integer(s.substring(7));

**if** (tempValue < min)

min = tempValue;

}

**byte**[] b = zk.getData(root + "/element" + min, **false**, stat);

zk.delete(root + "/element" + min, 0);

ByteBuffer buffer = ByteBuffer.wrap(b);

retvalue = buffer.getInt();

**return** retvalue;

}

}

}

}

@Override

**public** **void** process(WatchedEvent event) {

**super**.process(event);

}

**public** **static** **void** main(String args[]) {

// 启动Server

TestMainServer.start();

String connectString = "localhost:" + TestMainServer.CLIENT\_PORT;

FIFOQueue q = **new** FIFOQueue(connectString, "/app1");

**int** i;

Integer max = **new** Integer(5);

System.*out*.println("Producer");

**for** (i = 0; i < max; i++)

**try** {

q.produce(10 + i);

} **catch** (KeeperException e) {

*logger*.error(e);

} **catch** (InterruptedException e) {

*logger*.error(e);

}

**for** (i = 0; i < max; i++) {

**try** {

**int** r = q.consume();

System.*out*.println("Item: " + r);

} **catch** (KeeperException e) {

i--;

*logger*.error(e);

} **catch** (InterruptedException e) {

*logger*.error(e);

}

}

}

}

### ZooKeeper实际应用

假设我们的集群有：

(1) **20个搜索引擎的服务器**：每个负责总索引中的一部分的搜索任务。

**①** 搜索引擎的服务器中的15个服务器现在提供搜索服务。

**②** 5个服务器正在生成索引。

这20个搜索引擎的服务器，经常要让正在提供搜索服务的服务器停止提供服务开始生成索引,或生成索引的服务器已经把索引生成完成可以搜索提供服务了。

(2) **一个总服务器**：负责向这20个搜索引擎的服务器发出搜索请求并合并结果集。

(3) **一个备用的总服务器**：负责当总服务器宕机时替换总服务器。

(4) **一个web的cgi**：向总服务器发出搜索请求。

使用Zookeeper可以保证：

(1) **总服务器：**自动感知有多少提供搜索引擎的服务器，并向这些服务器发出搜索请求。

(2) **备用的总服务器：**宕机时自动启用备用的总服务器。

(3) **web的cgi：**能够自动地获知总服务器的网络地址变化。

(4) 实现如下：

**①** **提供搜索**引擎的服务器都在Zookeeper中创建znode，zk.**create(**"/search/nodes/node1"**,** "hostname".getBytes()**,** Ids.OPEN\_ACL\_UNSAFE**,** CreateFlags.EPHEMERAL**)**;

**② 总服务器**可以从Zookeeper中获取一个znode的子节点的列表，zk.getChildren("/search/nodes", true);

**③** **总服务器**遍历这些子节点，并获取子节点的数据生成提供搜索引擎的服务器列表；

**④** **当总服务器**接收到子节点改变的事件信息,重新返回第二步；

**⑤** **总服务器**在Zookeeper中创建节点，zk.**create**("/search/master", "hostname".getBytes(), Ids.OPEN\_ACL\_UNSAFE, CreateFlags.EPHEMERAL);

**⑥ 备用的总服务器**监控Zookeeper中的"/search/master"节点。当这个znode的节点数据改变时，把自己启动变成总服务器，并把自己的网络地址数据放进这个节点。

**⑦** **web的cgi**从Zookeeper中"/search/master"节点获取总服务器的网络地址数据，并向其发送搜索请求。

**⑧** **web的cgi**监控Zookeeper中的"/search/master"节点，当这个znode的节点数据改变时，从这个节点获取总服务器的网络地址数据,并改变当前的总服务器的网络地址。

## ZooKeeper机制架构

### ZooKeeper权限管理机制

#### ****权限管理ACL(Access Control List)****

ZooKeeper 的权限管理亦即ACL 控制功能，使用ACL来对Znode进行访问控制。ACL的实现和Unix文件访问许可非常相似：它使用许可位来对一个节点的不同操作进行允许或禁止的权 限控制。但是和标准的Unix许可不同的是，Zookeeper对于用户类别的区分，不止局限于所有者(owner)、组 (group)、所有人(world)三个级别。Zookeeper中，数据节点没有"所有者"的概念。访问者利用id标识自己的身份，并获得与之相应的 不同的访问权限。

ZooKeeper 的权限管理通过Server、Client 两端协调完成：

**(1) Server端**

一个ZooKeeper 的节点存储两部分内容：**数据**和**状态**，状态中包含ACL 信息。创建一个znode 会产生一个ACL 列表，列表中每个ACL 包括：

**①** 权限perms

**②** 验证模式scheme

**③** 具体内容expression：Ids

例如，当scheme="digest" 时， Ids 为用户名密码， 即"root ：J0sTy9BCUKubtK1y8pkbL7qoxSw"。ZooKeeper 提供了如下几种验证模式：

**① Digest**： Client 端由用户名和密码验证，譬如user:pwd

**② Host**： Client 端由主机名验证，譬如localhost

**③ Ip**：Client 端由IP 地址验证，譬如172.2.0.0/24

**④ World** ：固定用户为anyone，为所有Client 端开放权限

当会话建立的时候，客户端将会进行自我验证。

权限许可集合如下：

**① Create** 允许对子节点Create 操作

**②** **Read** 允许对本节点GetChildren 和GetData 操作

**③** **Write** 允许对本节点SetData 操作

**④** **Delete** 允许对子节点Delete 操作

**⑤** **Admin** 允许对本节点setAcl 操作

另外，ZooKeeper Java API支持**三种标准的用户权限**，它们分别为：

**①** ZOO\_PEN\_ACL\_UNSAFE：对于所有的ACL来说都是完全开放的，任何应用程序可以在节点上执行任何操作，比如创建、列出并删除子节点。

**②** ZOO\_READ\_ACL\_UNSAFE：对于任意的应用程序来说，仅仅具有读权限。

**③** ZOO\_CREATOR\_ALL\_ACL：授予节点创建者所有权限。需要注意的是，设置此权限之前，创建者必须已经通了服务器的认证。

Znode ACL 权限用一个int 型数字perms 表示，perms 的5 个二进制位分别表示set**a**cl、**d**elete、**c**reate、**w**rite、**r**ead。比如adcwr=0x1f，----r=0x1，a-c-r=0x15。

***注意的是，exists操作和getAcl操作并不受ACL许可控制，因此任何客户端可以查询节点的状态和节点的ACL。***

**(2) 客户端**

Client 通过调用addAuthInfo()函数设置当前会话的Author信息**（**针对Digest 验证模式**）**。Server 收到Client 发送的操作请求**（**除exists、getAcl 之外**）**，需要进行**ACL 验证**：对该请求携带的Author 明文信息加密，并与目标节点的ACL 信息进行比较，如果匹配则具有相应的权限，否则请求被Server 拒绝。

下面演示一个通过digest(用户名：密码的方式)为创建的节点设置ACL的例子，代码如下：

**package** org.zk.queue;

**import** org.apache.Zookeeper.\*;

**import** org.apache.Zookeeper.server.auth.DigestAuthenticationProvider;

**import** org.apache.Zookeeper.data.\*;

**import** java.util.\*;

**public** **class** NewDigest {

**public** **static** **void** main(String[] args) **throws** Exception {

// new一个acl

List<ACL> acls = **new** ArrayList<ACL>();

// 添加第一个id，采用用户名密码形式

Id id1 = **new** Id("digest",

DigestAuthenticationProvider.generateDigest("admin:admin"));

ACL acl1 = **new** ACL(ZooDefs.Perms.ALL, id1);

acls.add(acl1);

// 添加第二个id，所有用户可读权限

Id id2 = **new** Id("world", "anyone");

ACL acl2 = **new** ACL(ZooDefs.Perms.READ, id2);

acls.add(acl2);

// Zk用admin认证，创建/test ZNode。

ZooKeeper Zk = **new** ZooKeeper("host1:2181,host2:2181,host3:2181", 2000,

**null**);

Zk.addAuthInfo("digest", "admin:admin".getBytes());

Zk.create("/test", "data".getBytes(), acls, CreateMode.PERSISTENT);

}

}

#### ****ZooKeeper SuperDigest****

**(1)** 一次Client 对Znode 进行操作的验证ACL 的方式为：

**a) 遍历znode的所有ACL：**

**①** 对于每一个ACL，首先操作类型与权限（perms）匹配

**②** 只有匹配权限成功才进行session 的auth 信息与ACL 的用户名、密码匹配

**b)** 如果两次匹配都成功，则允许操作；否则，返回权限不够error（rc=-102）

**(2)** 如果Znode ACL List 中任何一个ACL 都没有setAcl 权限，那么就算superDigest 也修改不了它的权限；再假如这个Znode 还不开放delete 权限，那么它的所有子节点都将不会被删除。唯一的办法是通过手动删除snapshot 和log 的方法，将ZK 回滚到一个以前的状态，然后重启，当然这会影响到该znode 以外其它节点的正常应用。

**(3)** superDigest 设置的步骤：

**①** 启动ZK 的时候( zkServer.sh ) ， 加入参数：

Java"-Dzookeeper .DigestAuthenticationProvider.superDigest=super:D/InIHSb7yEEbrWz8b9l71RjZJU=" （无空格）。

**②** 在客户端使用的时候， addAuthInfo("digest", "super:test", 10, 0, 0); " super:test" 为"super:D/InIHSb7yEEbrWz8b9l71RjZJU="的明文表示，加密算法同setAcl。

### Watch机制

Zookeeper客户端在数据节点上设置监视，则当数据节点发生变化时，客户端会收到提醒。ZooKeeper中的各种读请求，如getDate()，getChildren()，和exists()，都可以选择加"监视点"(watch)。"监视点"指的是一种**一次性的触发器**(trigger)，当受监视的数据发生变化时，该触发器会通知客户端。

**(1) 监视机制有三个关键点：**

**①** "监视点"是一次性的，当触发过一次之后，除非重新设置，新的数据变化不会提醒客户端。

**②** "监视点"将数据改变的通知客户端。如果数据改变是**客户端A引起**的，不能保证"监视点"通知事件会在引发数据修改的函数返回前**到达客户端A**。

**③** 对于"监视点"，ZooKeeper有如下保证：客户端一定是在接收到"监视"事件（watch event）之后才接收到数据的改变信息。

**(2)** "**监视点**"保留在ZooKeeper服务器上，则当客户端连接到新的ZooKeeper服务器上时，所有需要被触发的相关"监视点"都会被触发。当客户端断线后重连，与它的相关的"监视点"都会自动重新注册，这对客户端来说是透明的。在以下情况，"监视点"会被错过：客户端B设置了关于节点A存在性的"监视点"，但B断线了，在B断线过程中节点A被创建又被删除。此时，B再连线后不知道A节点曾经被创建过。

**(3)** ZooKeeper的"监视"机制保证以下几点：

**①** "监视"事件的触发顺序和事件的分发顺序一致。

**②** 客户端将先接收到"监视"事件，然后才收到新的数据

**③** "监视"事件触发的顺序与ZooKeeper服务器上数据变化的顺序一致

**(4)** 关于ZooKeeper"监视"机制的注意点：

**①** "监视点"是一次性的。

**②** 由于"监视点"是一次性的，而且，从接收到"监视"事件到设置新"监视点"是有延时的，所以客户端可能监控不到数据的所有变化。

**③** 一个监控对象，只会被相关的通知触发一次。如果一个客户端设置了关于某个数据点exists和getData的监控，则当该数据被删除的时候，只会触发"文件被删除"的

通知。

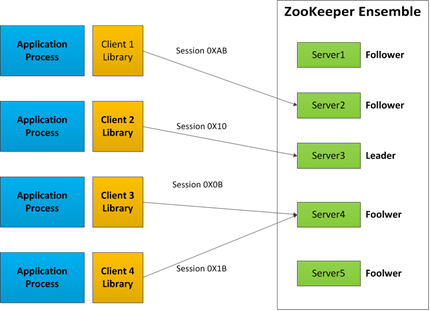
**④** 当客户端断开与服务器的连接时，客户端不再能收到"监视"事件，直到重新获得连接。所以关于Session的信息将被发送给所有ZooKeeper服务器。由于当连接断开时收不到"监视"，所以在这种情况下，模块行为需要容错方面的设计。

### Session机制

#### ****会话概述****

每个ZooKeeper客户端的配置中都包括集合体中服务器的列表。在启动时，客户端会尝试连接到列表中的一台服务器。如果连接失败，它会尝试连接另一台服务器，以此类推，直到成功与一台服务器建立连接或因为所有ZooKeeper服务器都不可用而失败。

**图 3.1 ZooKeeper体系结构**



一旦客户端与一台ZooKeeper服务器建立连接，这台服务器就会为该客户端创建一个新的会话。每个会话都会有一个超时的时间设置，这个设置由创建会话的应用来设定。如果服务器在超时时间段内没有收到任何请求，则相应的会话会过期。一旦一个会话已经过期，就无法重新打开，并且任何与该会话相关联的短暂znode都会丢失。会话通常长期存在，而且会话过期是一种比较罕见的事件，但对应用来说，如何处理会话过期仍是非常重要的。

只要一个会话空闲超过一定时间，都可以通过客户端发送ping请求（也称为心跳）保持会话不过期。ping请求由ZooKeeper的客户端库自动发送，因此在我们的代码中不需要考虑如何维护会话。这个时间长度的设置应当足够低，以便能档检测出服务器故障（由读超时体现），并且能够在会话超时的时间段内重新莲接到另外一台服务器。

#### ****故障切换****

ZooKeeper客户端可以自动地进行故障切换，切换至另一台ZooKeeper服务器。**并且关键的一点是**，在另一台服务器接替故障服务器之后，所有的会话和相关的短暂Znode仍然是有效的。**在故障切换过程中**，应用程序将收到断开连接和连接至服务的通知。当**客户端断开连接时**，观察通知将无法发送；但是当客户端成功恢复连接后，这些延迟的通知会被发送。当然，在客户端重新连接至另一台服务器的过程中，如果应用程序试图执行一个操作，这个操作将会失败。这充分体现了在真实的ZooKeeper应用中处理连接丢失异常的重要性。

### ZooKeeper实例状态

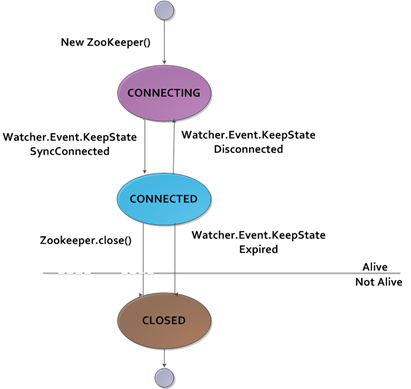
**(1) ZooKeeper状态**

ZooKeeper对象在其生命周期中会经历几种不同的状态。你可以在任何时刻通过getState()方法来查询对象的状态：

public States getState()

States被定义成代表ZooKeeper对象不同状态的枚举类型值（不管是什么枚举值，一个ZooKeeper的实例在一个时刻只能处于一种状态）。在试图与ZooKeeper服务**建立连接的过程中**，一个新建的ZooKeeper实例处于CONNECTING状态。一旦**建立连接**，它就会进入CONNECTED状态。

**图 3.2 ZooKeeper状态转换**



通过注册观察对象，使用了ZooKeeper对象的客户端可以收到状态转换通知。在进入CONNECTED状态时，观察对象会收到一个WatchedEvent通知，其中KeeperState的值是SyncConnected。

**(2) Watch与ZooKeeper状态**

ZooKeeper的观察对象肩负着双重责任：

**①** 可以用来获得ZooKeeper状态变化的相关通知；

**②** 可以用来获得Znode变化的相关通知。

监视**ZooKeeper状态变化**：可以使用ZooKeeper对象默认构造函数的观察。

监视**Znode变化：**可以使用一个专用的观察对象，将其传递给适当的读操作。也可以通过读操作中的布尔标识来设定是否共享使用默认的观察。

ZooKeeper实例可能失去或重新连接ZooKeeper服务，在CONNECTED和CONNECTING状态中切换。如果连接断 开，watcher得到一个Disconnected事件。需要注意的是，这些状态的迁移是由ZooKeeper实例自己发起的，如果连接断开他将自动尝 试自动连接。

如果任何一个close()方法被调用，或是会话由Expired类型的KeepState提示过期时，ZooKeeper可能会转变成第三种状态 CLOSED。一旦处于CLOSED状态，ZooKeeper对象将不再是活动的了(可以使用states的isActive()方法进行测试)，而且不 能被重用。客户端必须建立一个新的ZooKeeper实例才能重新连接到ZooKeeper服务。

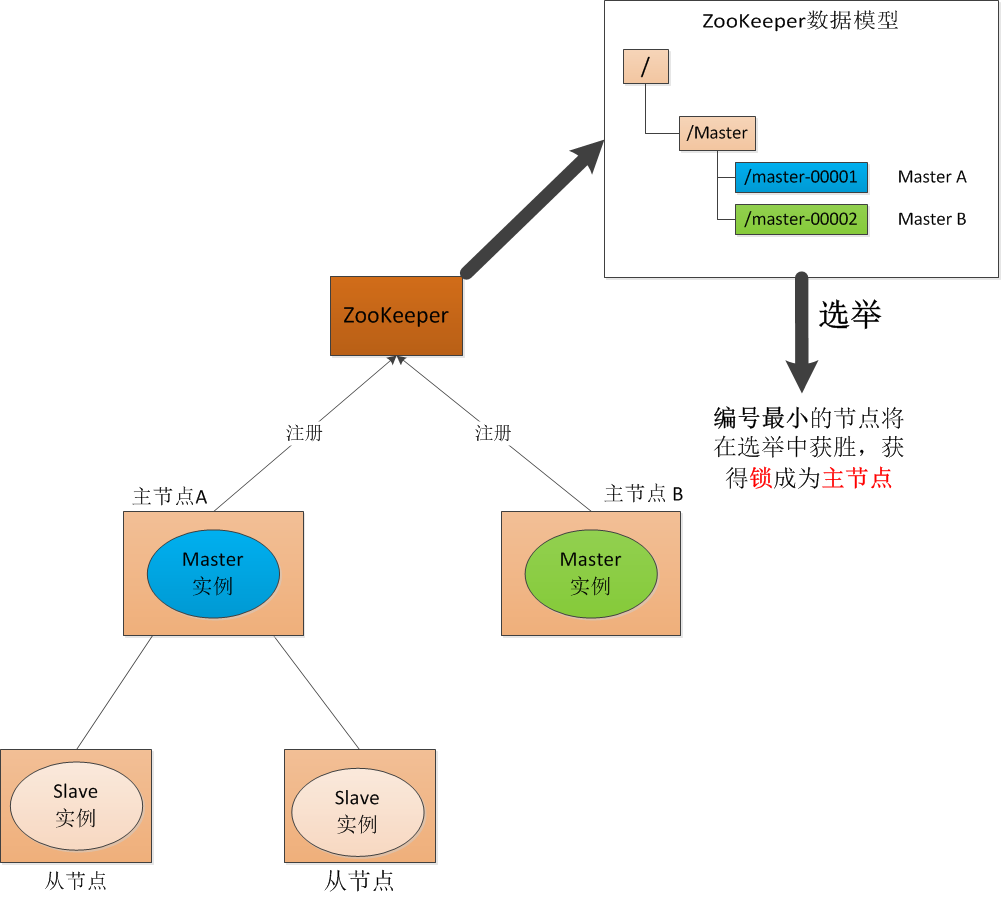
## ZooKeeper一致性原理

### ZooKeeper 的实现

#### ****ZooKeeper处理单点故障****

我们知道可以通过ZooKeeper对分布式系统进行Master选举，来解决分布式系统的**单点故障，**如图所示。

**图 1.1 ZooKeeper解决单点故障**

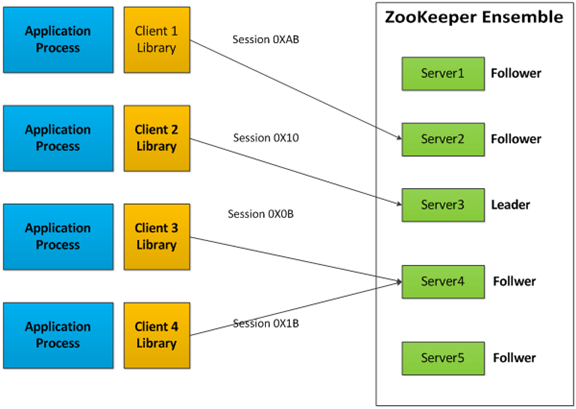


那么我们继续分析一下，ZooKeeper通过Master选举来帮助分布式系统解决**单点故障**， 保证该系统中每时每刻只有一个Master为分布式系统提供服务。也就是说分布式的单点问题交给了ZooKeeper来处理，不知道大家此时有没有发现一 个问题——"故障转移到了ZooKeeper身上"。大家看一下图就会发现，如果我们的ZooKeeper只用一台机器来提供服务，若这台机器挂了，那么 该分布式系统就直接变成双Master模式了，那么我们在分布式系统中引入ZooKeeper也就失去了意义。那么这也就意味着，ZooKeeper在其**实现的过程**中要做一些可用性和恢复性的保证。这样才能让我们放心的以ZooKeeper为起点来构建我们的分布式系统，来达到节省成本和减少bug的目的。

#### ****ZooKeeper运行模式****

ZooKeeper服务有两种不同的运行模式。一种是"**独立模式**"(standalone mode)，即只有一个ZooKeeper服务器。这种模式较为简单，比较适合于测试环境，甚至可以在单元测试中采用，但是不能保证高可用性和恢复性。在生产环境中的ZooKeeper通常以"**复制模式**"(replicated mode)运行于一个计算机集群上，这个计算机集群被称为一个"**集合体**"(ensemble)。

**图 1.2 ZooKeeper集群**



ZooKeeper通过**复制**来实现高可用性，只要集合体中**半数以上**的机器处于可用状态，它就能够提供服务。例如，在一个有5个节点的集合体中，每个Follower节点的数据都是Leader节点数据的副本，也就是说我们的每个节点的数据视图都是一样的，这样就可以有五个节点提供ZooKeeper服务。并且集合体中任意2台机器出现故障，都可以保证服务继续，因为剩下的3台机器超过了半数。

***注意，6个节点的集合体也只能够容忍2台机器出现故障，因为如果3台机器出现故障，剩下的3台机器没有超过集合体的半数。出于这个原因，一个集合体通常包含奇数台机器。***

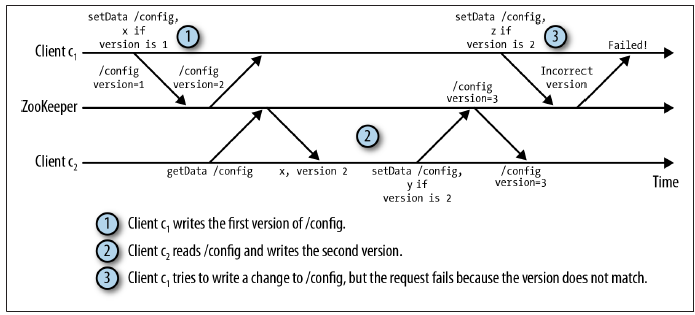
从概念上来说，ZooKeeper它所做的就是确保对Znode树的每一个修改都会被复制到集合体中超过半数的 机器上。如果少于半数的机器出现故障，则最少有一台机器会保存最新的状态，那么这台机器就是我们的Leader。其余的副本最终也会更新到这个状态。如果 Leader挂了，由于其他机器保存了Leader的副本，那就可以从中选出一台机器作为新的Leader继续提供服务。

#### ****ZooKeeper的读写机制****

**(1) 概述**

**ZooKeeper**的**核心思想**是，提供一个非锁机制的Wait Free的用于分布式系统同步的核心服务。提供简单的文件创建、读写操作接口，其系统核心本身对**文件读写**并不提供加锁互斥的服务，但是提供基于版本比对的更新操作，客户端可以基于此自己实现加锁逻辑。如下图1.3所示。

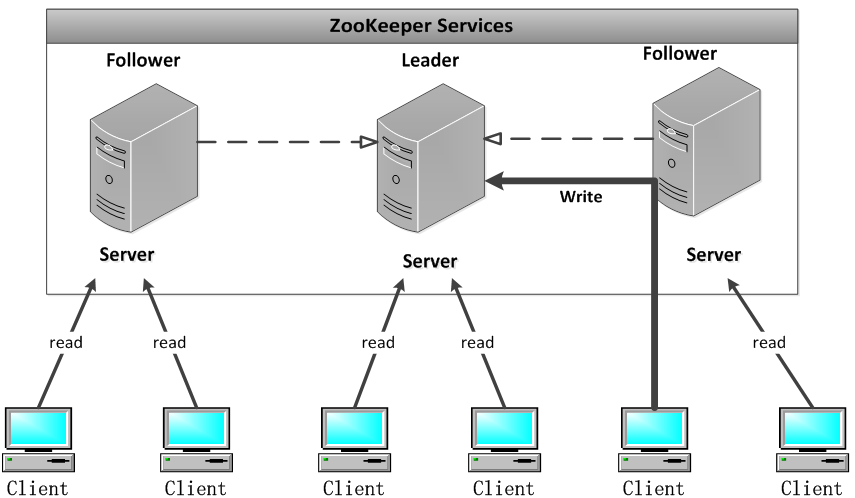
**图 1.3 Using versions to prevent inconsistencies due to concurrent updates**



**(2) ZK集群服务**

 Zookeeper是一个由多个Server组成的集群，该集群有一个Leader，多个Follower。客户端可以连接任意ZooKeeper服务节点来读写数据，如下图1.4所示。

**图 1.4 ZooKeeper集群服务**



ZK集群中每个Server，都保存一份数据副本。Zookeeper使用简单的同步策略，通过以下两条基本保证来实现数据的一致性：

① 全局**串行化**所有的**写操作**

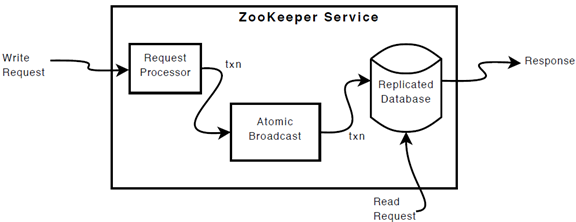
② 保证**同一客户**端的指令被FIFO（队列，先进先出）执行（以及消息通知的FIFO）

所有的读请求由Zk Server 本地响应，所有的更新请求将转发给Leader，由Leader实施。

**(3) ZK组件**

ZK组件，如图1.5所示。ZK组件除了**请求处理器**（Request Processor）以外，组成ZK服务的每一个Server会复制这些组件的副本。

**图 ZooKeeper组件图**



**ReplicatedDatabase**是一个内存数据库，它包含了整个Data Tree。为了恢复，更新会被记录到磁盘，并且写在被应用到内存数据库之前，先被序列化到磁盘。

每一个ZK Server，可服务于多个Client。Client可以连接到一台Server，来提交请求。**读请求**，由每台Server数据库的本地副本来进行服务。改变服务器的状态的写请求，需要通过一致性协议来处理。

作为一致性协议的一部分，来自Client的所有写请求，都要被转发到一个单独的Server，称作Leader。ZK集群中其他Server 称作Follower，负责接收Leader发来的提议消息，并且对消息转发达成一致。消息层处理leader失效，同步Followers和Leader。

ZooKeeper使用自定义的原子性消息协议。由于消息传送层是原子性的，ZooKeeper能够保证本地副本不产生分歧。当leader收到一个写请求，它会计算出当写操作完成后系统将会是什么状态，接着将之转变为一个捕获状态的事务。

**(4) ZK性能**

ZooKeeper被应用程序广泛使用，并有数以千计的客户端同时的访问它，所以我们需要高吞吐量。我们为ZooKeeper 设计的工作负载的读写比例是 2：1以上。然而我们发现，ZooKeeper的高写入吞吐量，也允许它被用于一些写占主导的工作负载。ZooKeeper通过每台Server上的本地ZK的状态副本，来提供高读取吞吐量。因此，**容错性**和**读吞吐量**是以添加到该服务的服务器数量为尺度。**写吞吐量**并不以添加到该服务的机器数量为尺度。

例如，在它的诞生地Yahoo公司，对于写占主导的工作负载来说，ZooKeeper的基准吞吐量已经超过每秒10000个操作；对于常规的以读为主导的工作负载来说，吞吐量更是高出了好几倍。

### ZooKeeper的保证

经过上面的分析，我们知道要保证ZooKeeper服务的高可用性就需要采用**分布式模式**，来冗余数据写多份，写多份带来一致性问题，一致性问题又会带来性能问题，那么就此陷入了无解的死循环。那么在这，就涉及到了我们分布式领域的著名的CAP理论，在这就简单的给大家介绍一下，关于CAP的详细内容大家可以网上查阅。

#### ****CAP理论****

**(1) 理论概述**

分布式领域中存在CAP理论：

**①** **C：Consistency**，一致性，数据一致更新，所有数据变动都是同步的。

**②** **A：Availability**，可用性，系统具有好的响应性能。

**③** **P：Partition tolerance**，分区容错性。以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择，也就是说无论任何消息丢失，系统都可用。

该理论已被**证明**：任何分布式系统只可同时满足两点，无法三者兼顾。 因此，将精力浪费在思考如何设计能满足三者的完美系统上是愚钝的，应该根据应用场景进行适当取舍。

(2) **一致性分类**

一致性是指从系统外部读取系统内部的数据时，在一定约束条件下相同，即数据变动在系统内部各节点应该是同步的。根据一致性的强弱程度不同，可以将一致性级别分为如下几种：

**① 强一致性**（strong consistency）。任何时刻，任何用户都能读取到最近一次成功更新的数据。

**② 单调一致性**（monotonic consistency）。任何时刻，任何用户一旦读到某个数据在某次更新后的值，那么就不会再读到比这个值更旧的值。也就是说，可获取的数据顺序必是单调递增的。

**③ 会话一致性**（session consistency）。任何用户在某次会话中，一旦读到某个数据在某次更新后的值，那么在本次会话中就不会再读到比这个值更旧的值。会话一致性是在单调一致性的基础上进一步放松约束，只保证单个用户单个会话内的单调性，在不同用户或同一用户不同会话间则没有保障。

**④** **最终一致性**（eventual consistency）。用户只能读到某次更新后的值，但系统保证数据将最终达到完全一致的状态，只是所需时间不能保障。

**⑤ 弱一致性**（weak consistency）。用户无法在确定时间内读到最新更新的值。

#### ****ZooKeeper与CAP理论****

我们知道ZooKeeper也是一种分布式系统，它在一致性上有人认为它提供的是一种强一致性的服务（通过sync操作），也有人认为是单调一致性（更新时的大多说概念），还有人为是最终一致性（顺序一致性），反正各有各的道理这里就不在争辩了。然后它在分区容错性和可用性上做了一定折中，这和CAP理论是吻合的。ZooKeeper从以下几点保证了数据的一致性

**① 顺序一致性**

来自任意特定客户端的更新都会按其发送顺序被提交。也就是说，如果一个客户端将Znode z的值更新为a，在之后的操作中，它又将z的值更新为b，则没有客户端能够在看到z的值是b之后再看到值a（如果没有其他对z的更新）。

**② 原子性**

每个更新要么成功，要么失败。这意味着如果一个更新失败，则不会有客户端会看到这个更新的结果。

**③ 单一系统映像**

一个客户端无论连接到哪一台服务器，它看到的都是同样的系统视图。这意味着，如果一个客户端在同一个会话中连接到一台新的服务器，它所看到的系统状态不会比在之前服务器上所看到的更老。当一台服务器出现故障，导致它的一个客户端需要尝试连接集合体中其他的服务器时，所有滞后于故障服务器的服务器都不会接受该连接请求，除非这些服务器赶上故障服务器。

**④ 持久性**

一个更新一旦成功，其结果就会持久存在并且不会被撤销。这表明更新不会受到服务器故障的影响。

### ZooKeeper原理

#### ****原理概述****

Zookeeper的核心是原子广播机制，这个机制保证了各个server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是**恢复模式**和**广播模式**。

**(1) 恢复模式**

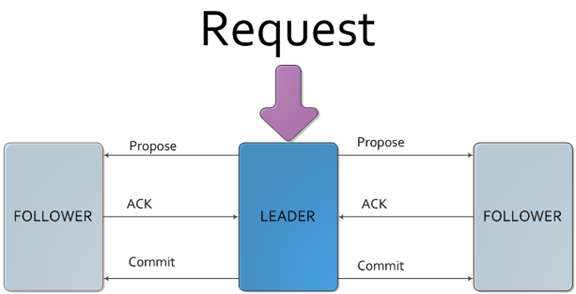
当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和server具有相同的系统状态。

**(2) 广播模式**

一旦Leader已经和多数的Follower进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个Server加入ZooKeeper服务中，它会在恢复模式下启动，发现Leader，并和Leader进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper服务一直维持在Broadcast状态，直到Leader崩溃了或者Leader失去了大部分的Followers支持。

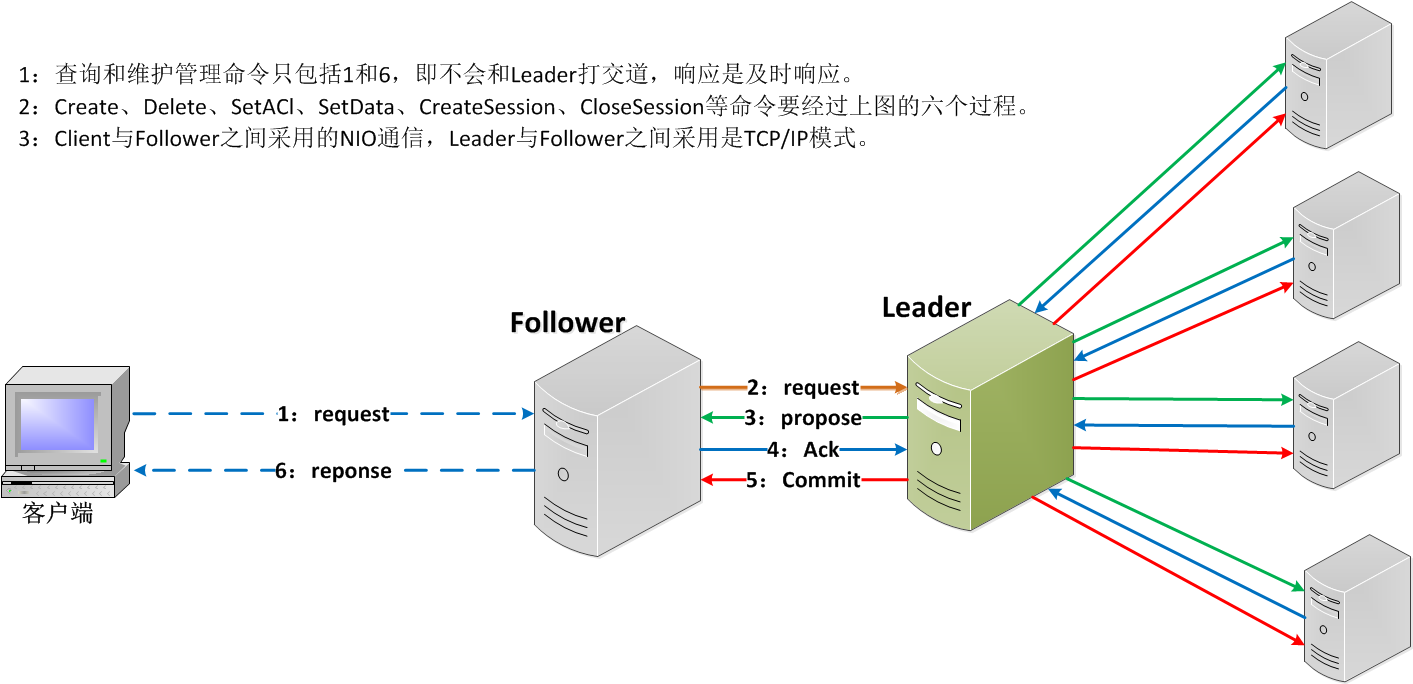
**Broadcast模式**极其类似于分布式事务中的2pc（two-phrase commit **两阶段提交**）：即Leader提起一个决议，由Followers进行投票，Leader对投票结果进行计算决定是否通过该决议，如果通过执行该决议（事务），否则什么也不做。

**图3.1两阶段提交**



在**广播模式**ZooKeeper Server会接受Client请求，所有的写请求都被转发给**领导者**，再由领导者将更新广播给**跟随者**。当半数以上的跟随者已经将修改**持久化**之后，领导者才会提交这个更新，然后客户端才会收到一个更新成功的响应。这个用来达成共识的协议被设计成具有原子性，因此每个修改要么成功要么失败。

**图 3.2 ZooKeeper数据流动图**

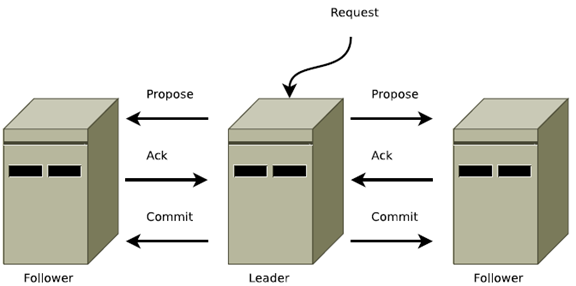


#### ****Zab协议详解****

##### ****广播模式****

**广播模式**类似一个简单的**两阶段提交**：Leader发起一个请求，收集选票，并且最终提交，图3.3演示了我们协议的消息流程。我们可以简化该两阶段提交协议，因为我们并没有"aborts"的情况。followers要么确认Leader的Propose，要么丢弃该Leader的Propose。没有"aborts"意味着，只要有指定数量的机器确认了该Propose，而不是等待所有机器的回应。

**图 3.3 The flow of message with protocol**



广播协议在所有的通讯过程中使用TCP的FIFO信道，通过使用该信道，使保持有序性变得非常的容易。通过FIFO信道，消息被有序的deliver。只要收到的消息一被处理，其顺序就会被保存下来。

Leader会广播已经被deliver的Proposal消息。在发出一个Proposal消息前，Leader会分配给Proposal一个单调递增的唯一id，称之为**zxid**。因为Zab保证了因果有序，所以递交的消息也会按照zxid进行排序。广播是把Proposal封装到消息当中，并添加到指向Follower的输出队列中，通过FIFO信道发送到 Follower。当Follower收到一个Proposal时，会将其写入到磁盘，可以的话进行批量写入。一旦被写入到磁盘媒介当中，Follower就会发送一个ACK给Leader。 当Leader收到了指定数量的ACK时，Leader将广播commit消息并在本地deliver该消息。当收到Leader发来commit消息时，Follower也会递交该消息。

需要注意的是， 该简化的两阶段提交自身并不能解决Leader故障，所以我们 添加恢复模式来解决Leader故障。

##### ****恢复模式****

**(1) 恢复阶段概述**

正常工作时Zab协议会一直处于广播模式，直到Leader故障或失去了指定数量的Followers。为了保证进度，恢复过程中必须选举出一个新Leader，并且最终让所有的Server拥有一个正确的状态。对于Leader选举，需要一个能够成功高几率的保证存活的算法。Leader选举协议，不仅能够让一个Leader得知它是leader，并且有指定数量的Follower同意该决定。如果Leader选举阶段发生错误，那么Servers将不会取得进展。最终会发生超时，重新进行Leader选举。在我们的实现中，**Leader选举**有两种不同的实现方式。如果有指定数量的Server正常运行，快速选举的完成只需要几百毫秒。

**(2)恢复阶段的保证**

该恢复过程的复杂部分是在一个给定的时间内，提议冲突的绝对数量。最大数量冲突提议是一个可配置的选项，但是默认是1000。为了使该协议能够即使在Leader故障的情况下也能正常运作。我们需要做出两条具体的保证：

**①** 我们绝不能遗忘已经被deliver的消息，若一条消息在一台机器上被**deliver**，那么该消息必须将在每台机器上**deliver**。

**②** 我们必须丢弃已经被skip的消息。

**(3) 保证示例**

**第一条：**

**若一条消息在一台机器上被deliver，那么该消息必须将在每台机器上deliver，即使那台机器故障了**。例如，出现了这样**一种情况**：Leader发送了commit消息，但在该commit消息到达其他任何机器之前，Leader发生了故障。也就是说，只有Leader自己收到了**commit**消息。如图3.4中的**C2**。

**图 3.4 The flow of message with protocol**

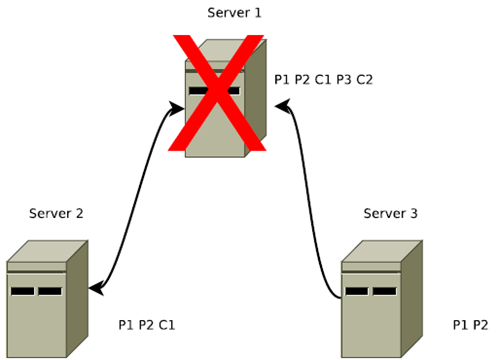


图3.4是"第一条保证"（deliver消息不能忘记）的一个示例。在该图中Server1是一个**Leader，**我们用**L1**表示，Server2和Server3为**Follower**。首先Leader发起了两个Proposal，P1和P2，并将P1、P2发送给了Server1和Server2。然后Leader对P1发起了Commit即C1，之后又发起了一个Proposal即P3，再后来又对P2发起了commit即C2，就在此时我们的Leader挂了。那么这时候，P3和C2这两个消息只有**Leader自己**收到了。

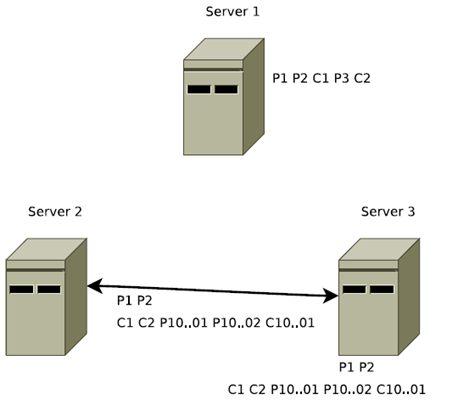
因为Leader已经deliver了该C2消息，client能够在消息中看到该事务的结果。所以该事务必须能够在其他所有的Server中deliver，最终使得client看到了一个一致性的服务视图。

**第二条：**

**一个被skip的消息，必须仍然需要被skip。**例如，发生了这样**一种情况：**Leader发送了propose消息，但在该propose消息到达其他任何机器之前，Leader发生了故障。也就是说，只有Leader自己收到了propose消息。如图3.4中的P3所示。

在图3.4中没有任何一个server能够看到**3号提议**，所以在图3.5中当server 1恢复时他需要在系统恢复时丢弃三号提议P3。

**图3.5**



在图3.5是"第二条保证"（skip消息必须被丢弃）的一个示例。Server1挂掉以后，Server3被选举为Leader，我们用L2表示。L2中还有未被deliver的消息P1、P2，所以，L2在发出新提议P10000001、P10000002之前，L2先将P1、P2两个消息**deliver**。因此，L2先发出了两个**commit**消息C1、C2，之后L2才发出了新的提议P10000001和P10000002。

如果Server1 恢复之后**再次成为了Leader**，此时再次将**P3**在P10000001和P10000002之后**deliver**，那么将违背顺序性的保障。

**(4) 保证的实现**

如果Leader选举协议保证了**新Leader**在**Quorum Server**中具有最高的提议编号，即Zxid最高。那么新选举出来的leader将具有所有已deliver的消息。新选举出来的Leader，在提出一个新消息之前，首先要保证**事务日志**中的所有消息都由**Quorum Follower**已Propose并deliver。需要注意的是，我们可以让新Leader成为一个用最高zxid来处理事务的server，来作为一个优化。这样，作为新被选举出来的Leader，就不必去从一组Followers中找出包含最高zxid的Followers和获取丢失的事务。

**① 第一条**

所有的正确启动的Servers，将会成为Leader或者**跟随**一个**Leader**。Leader能够确保它的Followers看到所有的提议，并deliver所有已经deliver的消息。通过将新连接上的Follower所没有见过的所有PROPOSAL进行**排队，**并之后对该Proposals的COMMIT消息进行**排队**，直到最后一个COMMIT消息。在所有这样的消息已经排好队之后，Leader将会把Follower加入到广播列表，以便今后的提议和确认。**这一条是为了保证一致性**，因为如果一条**消息P**已经在**旧Leader**-Server1中deliver了，即使它刚刚将**消息P** deliver之后就挂了，但是当旧**Leader**-**Server1**重启恢复之后，我们的Client就可以从该Server中看到该**消息P** deliver的事务，所以为了保证每一个client都能看到一个一致性的视图，我们需要将该消息在每个Server上deliver。

**② 第二条**

skip已经Propose，但不能deliver的消息，处理起来也比较简单。在我们的实现中，**Zxid**是由**64位**数字组成的，**低32**位用作简单**计数器**。**高32位**是一个**epoch**。每当**新Leader接管它时**，将获取日志中Zxid最大的epoch，新Leader **Zxid**的**epoch位**设置为epoch+1，**counter位**设置0。用epoch来标记领导关系的改变,并要求**Quorum Servers** 通过epoch来识别该leader，避免了多个Leader用同一个**Zxid**发布不同的提议。

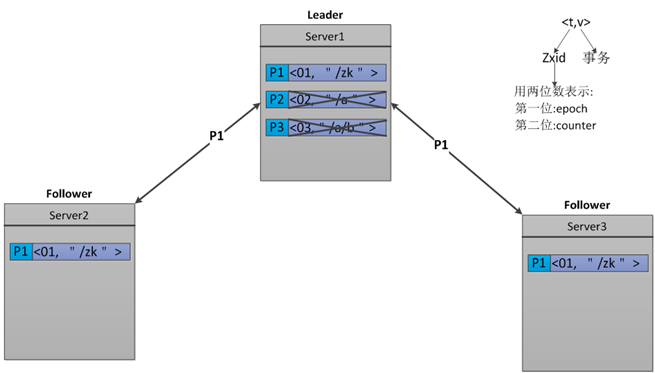
这个方案的一个优点就是，我们可以skip一个失败的领导者的实例，从而加速并简化了恢复过程。如果一台宕机的Server重启，并带有未发布的Proposal，那么先前的未发布的所有提议将永不会被deliver。并且它不能够成为一个新leader，因为任何一种可能的 Quorum Servers ，都会有一个Server其Proposal 来自与一个新epoch因此它具有一个较高的zxid。当Server以Follower的身份连接，领导者检查自身最后提交的提议，该提议的epoch 为Follower的最新提议的epoch（也就是图3.5中新Leader-Server2中deliver的C2提议），并告诉Follower截断事务日志**直到**该epoch在新Leader中deliver的最后的Proposal即C2。在图3.5中，当**旧Leader**-Server1连接到了**新leader**-Server2，leader将告诉他从事务日志中清除3号提议P3，具体点就是清除P2之后的所有提议，因为P2之后的所有提议只有**旧Leader**-Server1知道，其他Server不知道。

**(5) Paxos与Zab**

**① Paxos一致性**

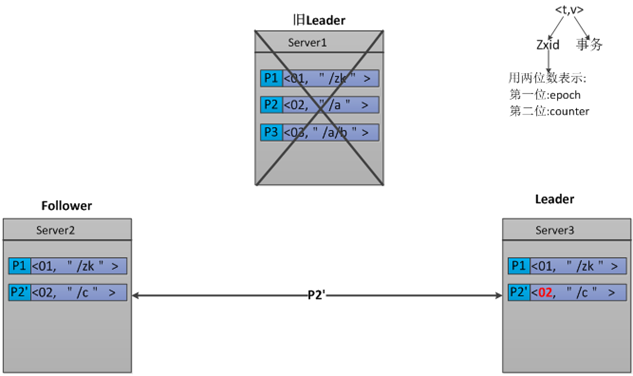
Paxos的一致性不能达到ZooKeeper的要求，我们可以下面一个例子。我们假设ZK集群由三台机器组成，Server1、Server2、Server3。Server1为Leader，他生成了三条Proposal，P1、P2、P3。但是在发送完P1之后，Server1就挂了。如下图3.6所示。

**图 3.6 Server1为Leader**



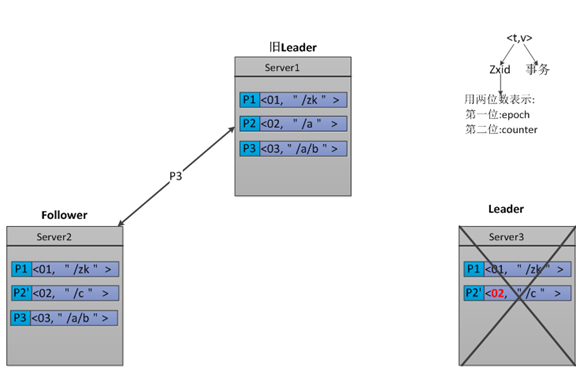
Server1挂掉之后，Server3被选举成为Leader，因为在Server3里只有一条Proposal—P1。所以，Server3在P1的基础之上又发出了一条新Proposal—P2＇，P2＇的Zxid为02。如下图3.7所示。

**图3.7 Server2成为Leader**



Server2发送完P2＇之后，它也挂了。此时Server1已经重启恢复，并再次成为了Leader。那么，Server1将发送还没有被deliver的Proposal—P2和P3。由于Follower-Server2中**P2＇**的Zxid为02和Leader-Server1中**P2**的Zxid相等，所以P2会被拒绝。而**P3**，将会被Server2接受。如图3.8所示。

**图3.8 Server1再次成为Leader**



我们分析一下Follower-Server2中的Proposal，由于P2'将P2的内容覆盖了。所以导致，Server2中的Proposal-P3无法生效，因为他的父节点并不存在。

**② Zab一致性**

首先来分析一下，上面的示例中为什么不满足ZooKeeper需求。ZooKeeper是一个树形结构，很多操作都要先检查才能确定能不能执行，比如，在图3.8中Server2有三条Proposal。**P1**的事务是创建节点"/zk"，**P2'**是创建节点"/c"，而**P3**是创建节点 "/a/b",由于"/a"还没建，创建"a/b"就搞不定了。那么，我们就能从此看出Paxos的一致性达不到ZooKeeper一致性的要求。

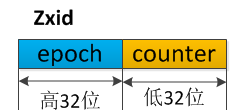
为了达到ZooKeeper所需要的一致性，ZooKeeper采用了Zab协议。Zab做了如下几条保证，来达到ZooKeeper要求的一致性。

(a) Zab要保证同一个leader的发起的事务要按顺序被apply，同时还要保证只有先前的leader的所有事务都被apply之后，新选的leader才能在发起事务。

(b) 一些已经Skip的消息，需要仍然被Skip。

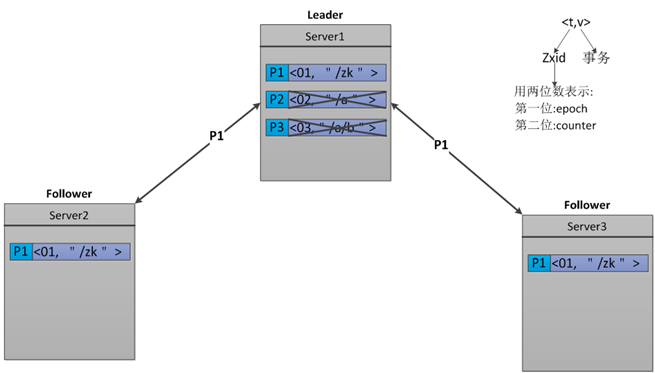
我想对于第一条保证大家都能理解，它主要是为了保证每个Server的数据视图的一致性。我重点解释一下第二条，它是如何实现。为了能够实现，Skip已经被skip的消息。我们在Zxid中引入了epoch，如下图所示。每当Leader发生变换时，epoch位就加1，counter位置0。

**图 3.9 Zxid**



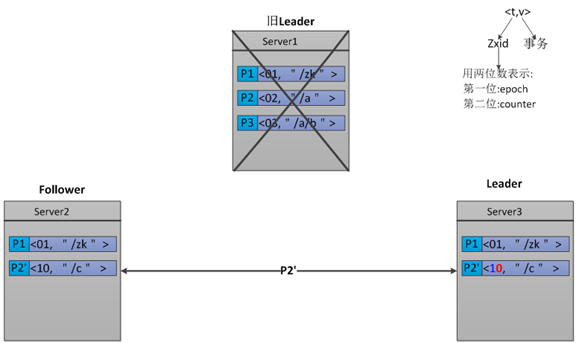
我们继续使用上面的例子，看一下他是如何实现Zab的第二条保证的。我们假设ZK集群由三台机器组成，Server1、Server2、Server3。Server1为Leader，他生成了三条Proposal，P1、P2、P3。但是在发送完P1之后，Server1就挂了。如下图3.10所示。

**图 3.10 Server1为Leader**



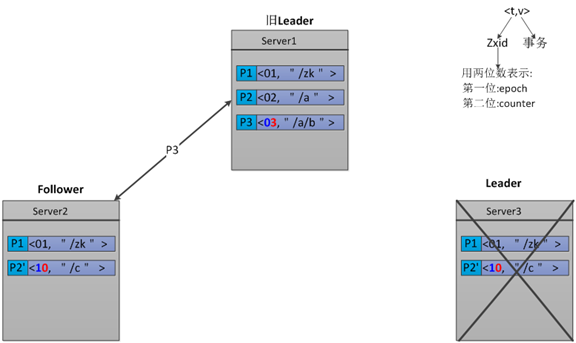
Server1挂掉之后，Server3被选举成为Leader，因为在Server3里只有一条Proposal—P1。所以，Server3在P1的基础之上又发出了一条新Proposal—P2＇，由于Leader发生了变换，epoch要加1，所以epoch由原来的0变成了1，而counter要置0。那么，P2＇的Zxid为**1**0。如下图3.11所示。

**图 3.11 Server3为Leader**



Server2发送完P2＇之后，它也挂了。此时Server1已经重启恢复，并再次成为了Leader。那么，Server1将发送还没有被deliver的Proposal—P2和P3。由于Server2中**P2＇**的Zxid为**10，**而Leader-Server1中**P2**和**P3**的Zxid分别为**02**和**03**，**P2＇**的epoch位高于P2和P3。所以此时Leader-Server1的P2和P3都会被拒绝,那么我们Zab的第二条保证也就实现了。如图3.12所示。

**图 3.12 Server1再次成为Leader**



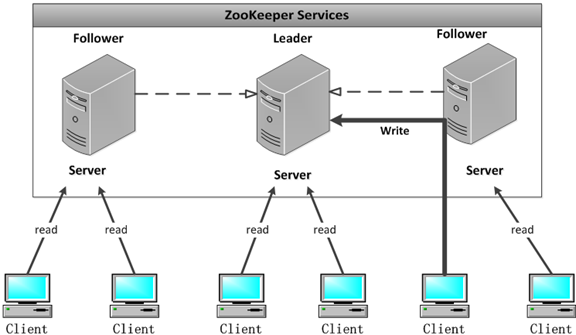
## ZooKeeper的伸缩性

### ZooKeeper中Observer

#### ****ZooKeeper角色****

经过前面的介绍，我想大家都已经知道了在ZooKeeper集群当中有两种角色Leader和Follower。Leader可以接受client 请求，也接收其他Server转发的写请求，负责更新系统状态。 Follower也可以接收client请求，如果是写请求将转发给Leader来更新系统状态，读请求则由Follower的内存数据库直接响应。 ZooKeeper集群如图1.1所示。

**图 1.1 ZooKeeper集群服务**



但在ZooKeeper的3.3.3版本以后，ZooKeeper中又添加了一种新角色Observer。Observer的作用同Follower类似，唯一区别就是它不参与选主过程。那么，我们就可以根据该特性将ZK集群中的Server分为两种：

(1) 投票Server：Leader、Follower

(2) 非投票Server：Observer

#### ****为什么引入Observer****

**(1) ZooKeeper可伸缩性**

那么，ZooKeeper为什么要引入Observer这个角色呢？其实在ZooKeeper中引入Observer，主要是为了使 ZooKeeper具有更好的可伸缩性。那么，何为可伸缩性？关于伸缩性，对于不同的人意味着不同的事情。 而在这里是说，如果我们的工作负载可以通过给系统分配更多的资源来分担，那么这个系统就是可伸缩的；一个不可伸缩的系统却无法通过增加资源来提升性能，甚 至会在工作负载增加时，性能会急剧下降。

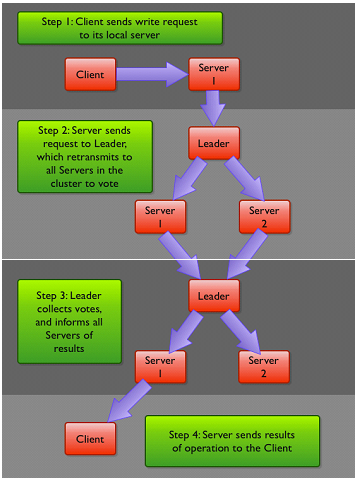
在Observer出现以前，ZooKeeper的伸缩性由Follower来实现，我们可以通过添加Follower节点的数量来保证 ZooKeeper服务的读性能。但是随着Follower节点数量的增加，ZooKeeper服务的写性能受到了影响。为什么会出现这种情况？在此，我 们需要首先了解一下这个"ZK服务"是如何工作的。

**(2) ZK服务过程**

ZooKeeper服务中的每个Server可服务于多个Client，并且Client可连接到ZK服务中的任一台Server来提交请求。若是**读请求**，则由每台Server的本地副本数据库直接响应。若是改变Server状态的**写请求**，需要通过一致性协议来处理，这个协议就是我们前面介绍的Zab协议。

简单来说，Zab协议规定：来自Client的所有写请求，都要转发给ZK服务中唯一的Server**—**Leader， 由Leader根据该请求发起一个Proposal。然后，其他的Server对该Proposal进行Vote。之后，Leader对Vote进行收 集，当Vote数量过半时Leader会向所有的Server发送一个通知消息。最后，当Client所连接的Server收到该消息时，会把该操作更新 到内存中并对Client的写请求做出回应。该工作流程如下图1.2所示。

**图1.2 ZK 写请求工作流程图**



从图中我们可以看出， ZooKeeper 服务器在上述协议中实际扮演了两个职能。它们一方面从客户端接受连接与操作请求，另一方面对操作结果进行投票。这两个职能在 ZooKeeper集群扩展的时候彼此制约。例如，当我们希望增加 ZK服务中Client数量的时候，那么我们就需要增加Server的数量，来支持这么多的客户端。然而，从Zab协议对写请求的处理过程中我们可以发 现，增加服务器的数量，则增加了对协议中投票过程的压力。因为Leader节点必须等待集群中过半Server响应投票，于是节点的增加使得部分计算机运 行较慢，从而拖慢整个投票过程的可能性也随之提高，写操作也会随之下降。这正是我们在实际操作中看到的问题——随着 ZooKeeper 集群变大，写操作的吞吐量会下降。

**(3) ZooKeeper扩展**

所以，我们不得不，在增加Client数量的期望和我们希望保持较好吞吐性能的期望间进行权衡。要打破这一耦合关系，我们引入了不参与投票的服务 器，称为 Observer。 Observer可以接受客户端的连接，并将写请求转发给Leader节点。但是，Leader节点不会要求 Observer参加投票。相反，Observer不参与投票过程，仅仅在上述第3歩那样，和其他服务节点一起得到投票结果。

**图 1.3 Observer 写吞吐量测试**

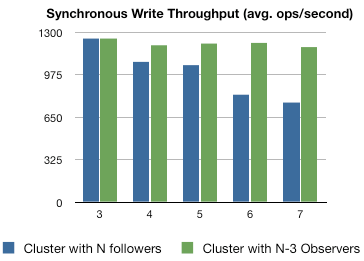


图1.3 显示了一个简单评测的结果。纵轴是，单一客户端能够发出的每秒钟同步写操作的数量。横轴是 ZooKeeper 集群的尺寸。蓝色的是每个服务器都是**投票Server**的情况，而绿色的则只有三个是**投票Server**，其它都是 Observer。从图中我们可以看出，我们在扩充 Observer时写性能几乎可以保持不变。但是，如果扩展**投票Server**的数量，写性能会明显下降，显然 Observers 是有效的。

这个简单的扩展，给 ZooKeeper 的可伸缩性带来了全新的空间。我们现在可以加入很多 Observer 节点，而无须担心严重影响写吞吐量。但他并非是无懈可击的，因为协议中的通知阶段，仍然与服务器的数量呈线性关系。但是，这里的串行开销非常低。因此，我 们可以认为在通知服务器阶段的开销无法成为主要瓶颈。

### Observer应用

**(1) Observer提升读性能的可伸缩性**

应对Client的数量增加，是 Observer的一个重要用例，但是实际上它还给集群带来很多其它的好处。Observer作为ZooKeeper的一个优化，Observer服务器可以直接获取Leader的本地数据存储，而无需经过投票过程。但这也面临一定的"时光旅行"风险，也就是说：可能在读到新值之后又读到老值。但这只在服务器故障时才会发生，在这种情况下，Client可以通过"sync"操作来保证下一个值是最新的。

因此，在大量读操作的工作负载下，Observer会使ZooKeeper的性能得到巨大提升。若要增加投票Server数量来承担读操作，那么就 会影响ZooKeeper服务的写性能。而且Observer允许我们将读性能和写性能分开，这使ZooKeeper更适用于一些以读为主的应用场景。

**(2) Observer提供了广域网能力**

Observer还能做更多。Observer对于跨广域网连接的Client来说是很好的候选方案。Observer可作为候选方案，原因有三：

① 为了获得很好的读性能，有必要让客户端离服务器尽量近，这样往返时延不会太高。然而，将 ZooKeeper 集群分散到两个集群是非常不可取的设计，因为良好配置的 ZooKeeper 应该让投票服务器间用低时延连接互连——否则，我们将会遇到上面提到的低反映速度的问题。

② 而Observer 可以被部署在，需要访问 ZooKeeper 的任意数据中心中。这样，投票协议不会受到数据中心间链路的高时延的影响，性能得到提升。投票过程中 Observer 和领导节点间的消息远少于投票服务器和领导节点间的消息。这有助于在远程数据中心高写负载的情况下降低带宽需求。

③ 由于Observer即使失效也不会影响到投票集群，这样如果数据中心间链路发生故障，不会影响到服务本身的可用性。这种故障的发生概率要远高于一个数据中心中机架间的连接的故障概率，所以不依赖于这种链路是个优点。

### ZooKeeper集群搭建案例

前面介绍了ZooKeeper集群中的几种角色，接下来给大家来介绍一下如何利用这些角色，来搭建一个性能良好的ZooKeeper集群。我以一个项目为例，给大家分析一下该如何规划我们的ZooKeeper集群。

假设我们的项目需要进行跨机房操作，我们的总部机房设在杭州，但他还要同美国，青岛等多个机房之间进行数据交互。但机房之间的网络延迟都比较大，比如中美机房走海底光缆有ping操作200ms的延迟，杭州和青岛机房有70ms的延迟。 为了提升系统的网络性能，我们在部署ZooKeeper网络时会在每个机房部署节点，多个机房之间再组成一个大的网络，来保证整个ZK集群数据一致性。

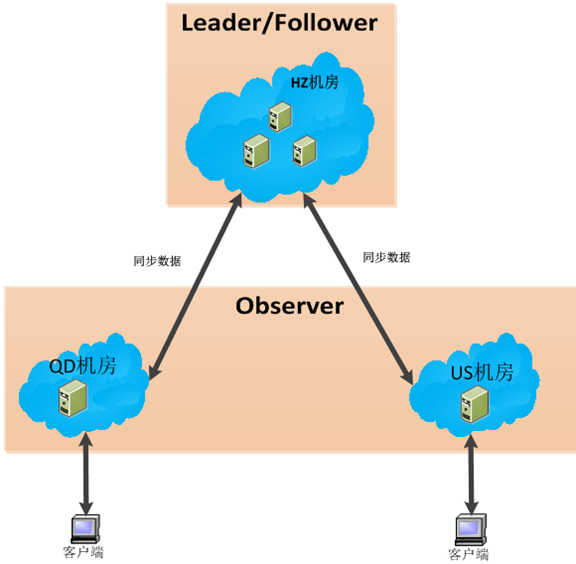
根据前面的介绍，最后的部署结构就会是：

(总部) 杭州机房  >=3台 ：由Leader/Follower构成的投票集群

(分支) 青岛机房  >=1台 ：由Observer构成的ZK集群

(分支) 美国机房  >=1台  : 由Observer构成的ZK集群

**图 3.1 ZooKeeper集群部署图**



从图中我们可以看出，我们在单个机房内组成一个投票集群，外围的机房都会是一个Observer集群和投票集群进行数据交互。 至于这样部署的一些好处，大家自己根据我前面对ZooKeeper角色的介绍，对比着体会一下，我想这样更能帮助大家理解ZooKeeper。而且针对这 样的部署结构，我们会引入一个**优先集群问题**： 比如在美国机房的Client，需要优先去访问本机房的ZK集群，访问不到才去访问HZ(总部)机房。