# 应用程序的高可用

“高可用性”（High Availability简称HA）通常来描述一个系统经过专门的设计，从而减少停工时间，而保持其服务的高度可用性。

举例来说：为了保证学校门口的安保问题，需要许多保安轮流值岗，这样万一有哪个保安因为身体原因无法值岗的话还可以有其他人顶上，这样就可以保证整个学校门口的安保的高度可用。

## 应用程序高可用的类型

### 主从方式（也称之为主从冷备）

还是学校门口的安保问题，可以让一个保安长期看门，另一个保安做一些其他的事情（比如一些学校的其他杂事儿）或者闲着（但是要跟长期看门的那个保安保持沟通，以了解门口安保的基本情况），等长期看门的保安由于种种原因无法看门时（可能是长期也可能是短期），就让另一个顶上就好了。

对应到应用程序中，就是如下的方式了：

准备有两个相同的应用程序，一个对外提供服务，称之为主程序，另一个平时不运行（主要负责跟对外提供服务的机器进行数据同步等）称之为从程序或备程序，即从程序是主程序的一个备份，等出问题的时候再顶上。常见的例子，mysql的主从复制。

### 双主互备（也称之为双主热备或互为热备）

还是学校门口的安保问题，可以让两个保安来同时看门，例如签到进门等工作一人负责一半，就算一个保安临时出现点问题还有另一个保安可以支撑校门口的安保工作（会造成的问题就是剩下的这个保安的工作量会翻倍）。

对应到应用程序中，就是如下的方式了：

准备两个相同的应用程序，同时对外提供服务（这时两个主程序相互作为对方的备份，故称之为双主热备），当其中一个出现问题时，另一个也可以照常对外提供服务（会造成的问题是单个应用程序的效率肯定不如两个程序一起来的效率高）。常见的例子：双tomcat或双nginx。

### 集群多互备

跟双主互备类似，只不过对外提供服务的程序的数量较多而已。

# Hadoop的master角色的单点故障问题

Hadood中的NameNode和ResourcManager是集群中的重要角色，如果这两个角色出现问题将导致整个集群无法使用。所以保证这两个角色的高可用是保证整个hadoop分布式系统高可用的关键。

为了保证其高可用，可以想到的一个办法是使用主从冷备或双主热备。但是为了在这两个角色出问题时尽快知晓并解决，还需要使用一个额外的应用程序监控这个两个角色的健康状况，当这两个角色出问题时，自动使用相应的解决方案，以减少系统停用时间，保证hadoop 的高可用。

正如上面所说的，这种协助分布式应用程序更好的提供服务的程序，我们称之为分布式协调程序。

因为分布式协调程序的重要性，所以其自身必须要保证高可用，才能保证被其协调的分布式应用程序的高可用。

目前市面上最流行的（也是唯一的企业级的）分布式协调应用程序就是zookeeper。需要注意的是：上述只是zookeeper能解决的问题之一，zookeeper还能解决很多其他的应用场景。

# Zookeeper的官方定义

Welcome to Apache ZooKeeper™

欢迎来到Apache Zookeeper的世界。

Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination.

Apache ZooKeeper是一个努力开发和维护的，可以保证可靠的分布式的开源式服务。

高可用的概念：

服务型服务器是否在24\*365 每天都能够提供服务。

What is ZooKeeper?

Zookeeper是什么？

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them ,which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

Zookeeper 分布式一致性协调服务（工具）

zookeeper是一个集中的服务，用以维护配置信息,命名,提供分布式同步,提供集团服务。所有这些类型的服务都使用某种形式的分布式应用程序。每次实现它们有很多工作,进入修复bug和竞争条件,是不可避免的。因为难以实现这些服务的应用程序最初通常通过他们,使他们脆弱的变化和难以管理。即使做得对,不同的实现这些服务的应用程序部署时导致管理的复杂性。

通俗的来说，ZooKeeper 顾名思义 动物园管理员，他是拿来管大象(Hadoop) 、 蜜蜂(Hive) 、 小猪(Pig) 的管理员， Apache Hbase和 Apache Solr 以及LinkedIn sensei 等项目中都采用到了 Zookeeper。ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，ZooKeeper是以Fast Paxos协议为基础，实现同步服务，配置维护和命名服务等分布式应用。 ZooKeeper 意欲设计一个易于编程的环境，它的文件系统使用我们所熟悉的目录树结构。 ZooKeeper 使用 Java 所编写，但是支持 Java 和 C 两种编程语言。

众所周知，协调服务非常容易出错，但是却很难恢复正常，例如，协调服务很容易处于竞态以至于出现死锁。设计 ZooKeeper 的目的是为了减轻分布式应用程序所承担的协调任务。

# Zookeeper是怎样保证自身的高可用的？

首先可以想到的是，假如zookeeper只有一个的话，其也无法保证其自身的高可用，所以zookeeper本身也是以集群的形式存在的。对比学校大门的安保问题，我们很容易能都想到的一种方法是可以有多个zookeeper，在某一时刻由一个主要的zookeeper对外提供服务，当其出现问题时，从剩下的多个zookeeper中快速选出来一个继续对外提供服务就可以了。Zookeeper本身就是基于这种思想来设计的。

Zookeeper采用了一个称之为ZAB的协议来保证自身的高可用。其来源于Paxos选举协议，其也是一个用来保证分布式一致的一个经典协议。在历史上，Paxos协议是从二阶段提交协议演变到三阶段提交协议之后再演变成的。

在分布式系统中，每一个机器节点虽然都能够明确的知道自己在进行事务操作过程中的结果是成功或失败，但是无法直接获取到其他分布式节点的操作结果（需要通过网络进行结果传输）。因此，当一个事务操作需要跨越多个分布式节点的时候，为了保持事务处理的ACID（原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability））特性，就需要引入一个称之为“协调者（Coordinator）”的组件来统一调度所有分布式节点的执行逻辑，这些被调度的分布式节点则被称为“参与者”（Participant）。协调者负责调度参与者的行为，并最终决定这些参与者是否要把事务真正提交。基于这个思想，衍生出了二阶段提交和三阶段提交两种协议两种一致性算法。

生活中大部分工作的形式都是分布式的，比如我们将来最常见的工作情景：项目经理将一个大项目分成了前后台（或者更多的部分），指派给不同的员工，每个人只负责自己的事情，所有干活的员工就是参与者的角色，项目经理就是协调者的角色。

## Zookeeper的ZAB协议和Paxos协议

二阶段提交和三阶段提交协议和Paxos协议都是分布式应用程序的通用协议，即在大部分的分布式应用程序中都可以使用。ZAB（zookeeper Atomic Broadcast）zookeeper原子消息广播协议）协议是zookeeper设计之初专门为雅虎内部那些高吞吐量、低延迟、健壮、简单的分布式场景设计的。所以ZAB不是一种通用型算法，而是一种特别为zookeeper设计的崩溃可恢复的原子消息广播算法。ZAB算法可是看成是paxos协议的一种具体实现，理解Paxos协议较为困难，我们不做掌握。

Zookeeper中将自己的角色系统设置主要为Leader、Flower、OBServer。Leader和Flower都是zk的server，只不过由Leader（只会存在一个，类似于Hadoop的NameNode）对外提供服务，众多Flower（类似于secondary namenode，不过Flow会有多个）随时准备等Leader出现问题时选举出一个新leader来继续对外提供以保证zk服务的高可用。

# Zookeeper集群中的机器数量设置

Zookeeper为了防止出现二阶段和三阶段提交协议中出现的数据不一致的情况（例如：网络出现问题，可能会将zk集群分成多个小集群），规定了只有zk集群中半数以上的server存活时才能对外提供服务。所以我们一般安装zk集群时，一般设置为奇数台zk的server。

假如，我们的zk集群有6个server，那么我们最多允许有两个server挂掉zk集群还能继续对外提供服务。但是如果我们的zk集群有5个server的话，我们同样最多允许有两个server挂掉也能继续提供服务。所以一般zk集群安装时server的数量是奇数个。

# Zookeeper的基本概念

## 集群角色

通常在分布式系统中，构成一个集群的每一台机器都有自己的角色，最典型的集群模式是Master/Slave模式（主从模式）。在这种模式中，我们把能够处理所有写操作的机器称为Master，把所有通过异步复制方式获取最新数据，并提供读服务的机器称为Slave机器。

Hadoop就是一个典型的案例。

而在zookeeper中，这些概念被颠覆了。它没有沿用传统的Master/Slave概念，而是引入了Leader、Flower和Observer（3.3版本开始启用）三种角色。Zookeeper集群中的所有机器通过一个Leader选举过程来选定一台称为“Leader”的机器，Leader服务器为客户端系统读和写服务。除Leader外，其他机器包括Flower和Observer。Flower和Observer都能够提供读服务，唯一的区别在于，Observer机器不参与Leader选举过程，也不参与写操作的“过半写成功”策略，因此Obsercer可以在不影响写性能的情况下提升集群的读性能。总结如下图所示：



## 系统模型

如下图所示：



## 会话（Session）

Session是指客户端会话。在zookeeper中，一个客户端连接是指客户端和服务器之间的一个TCP长连接。Zookeeper对外的服务端口默认是2181，客户端启动的时候，首先会与服务器建立一个TCP连接，从第一次连接建立开始，客户端会话的生命周期就开始了，通过这个连接，客户端能够通过心跳检测与服务器保持有效的会话，也能够向zookeeper服务器发送请求并接受响应，同时还能够通过该连接接收来自服务器的watch事件通知，Session的sessionTimeOut值用来设置一个客户端会话的超时间。当由于服务器压力太大、网络故障或是客户端主动断开连接等各种原因导致的客户端连接断开时，只要在sessionTimeOut规定的时间内能够重新链接上集群中任意一台服务器，那么之前创建的会话仍然有效。

## 数据节点（Znode）

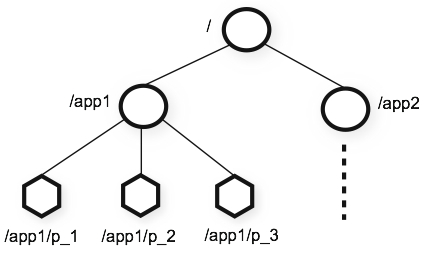
在分布式中我们通常说的“节点”是指组成集群的每一台机器。然而，在zookeeper中，“节点”分为两类，第一类同样是指构成集群的机器，我们称之为机器节点。第二类则是指数据模型中的数据单元，我们称之为数据节点——Znode。Zookeeper将所有数据存储在内存中，数据模型是一棵树（ZNode Tree），由斜杠（/）进行分割的路径，就是一个Znode，例如/foo/path1。每个ZNode上都会保存自己的数据内容，同时还会保存一系属性信息。

在zookeeper中，ZNode可以分为持久节点和临时节点两类。所谓持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则ZNode将一直保存在zookeeper上。而临时节点就不一样了，他的生命周期和客户会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被删除。另外，zookeeper还允许用户为每一个节点添加一个特殊的属性SEQUENTIAL。一旦节点被标记上这个属性，那么在这个节点被创建的时候，zookeeper会自动在其节点名后面追加上一个整形数字，这个整形数字是由父节点维护的自增数字。

Zookeeper在各种分布式场景下发挥作用的主要方式就是数据节点和监控器，我们可以给数据节点赋予任意我们想赋予的含义，来协助我们更好的处理分布式程序中所面临的问题，具体案例详见后文中的zookeeper 的其他典型应用场景。

zookeeper的目录结果： 也是也treenode树形结构

目录也是文件。



## 版本

Zookeeper的每个znode上都会存储数据，对应于每个znode，zookeeper都会为其维护一个叫作stat的数据结构，stat中记录了这个znode的三个数据版本，分别是version（当前znode的版本）、cversion（当前znode子节点的版本）和aversion（当前znode的acl版本）。

Acl: access control list（防火墙）

## Watcher

Watcher（事件监听器），是zookeeper中的一个很重要的特性。Zookeeper允许用户在指定节点上注册一些watcher，并且在一些特定的事件触发的时候，zookeeper服务端会将事件通知到感兴趣的客户端上去。

## ACL

zookeeper采用ACL（Access Control Lists）策略来进行权限控制，类似于UNIX文件系统的权限控制。zookeeper定义了如下5种权限。

CREATE：创建子节点的权限。

READ：获取子节点数据和子节点列表的权限。

WRITE：更新节点数据的权限。

DELETE：删除子节点的权限。

ADMIN：设置节点ACL的权限。

# Zookeeper的设计目的

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m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。

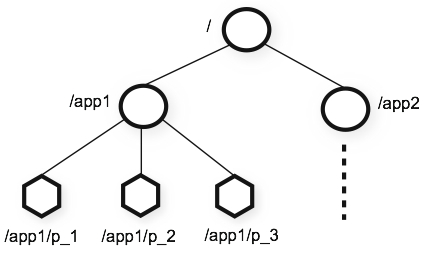
4 .等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

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的数据模型

Zookeeper的结构类似标准的文件系统，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作为保存数据的容器(限制在1mb以内)，也构成了一个层次化的命名空间。



Zookeeper目录中的每一个节点对应着一个znode，每个znode维护着一个属性结构，它包含数据的版本号、时间戳、等信息。Zookeeper就是通过这些属性来实现它特定的功能。每当znode的数据改变时，相应的版本号会增加，每当客户端查询、更新和删除数据时，也必须提供要被操作的znode版本号，如果所提供的数据版本号与实际的不匹配，那么将会操作失败。

节点属性：

|  |  |
| --- | --- |
| 属性 | 描述 |
| czxid | 节点被创建的zxid值 |
| mzxid | 节点被修改时zxid值 |
| ctime | 节点创建的时间 |
| mtime | 节点最后一次的修改时间 |
| vesion | 节点的版本号 |
| cversion | 节点所拥有的子节点被修改的版本号 |
| aversion | 节点的ACL被修改的版本号 |
| dataLength | 节点数据的长度 |
| numChildren | 节点拥有子节点的个数 |
| ephemeralOwner | 如果节点为临时节点，那么它的值为这个节点拥有者的session ID； |

Znode是客户端访问的Zookeeper的主要实体，它包含了以下主要特征：

(1)  Watch

Znode状态发生改变时(增删改等操作)，watch (监视器)机制可以让客户端得到通知，并且仅仅只会触发一次watch。

在读操作exists、getChildren和getData上可以设置监视器，这些监视器可以被写操作create、delete和setData触发。ACL相关的操作不参与任何监视器。当一监视器触发时，会产生一个事件，这个监视器和触发它的操作共同绝地沟了监视器事件类型。

* 当所监视的znode被创建子节点、删除或其他数据更新时，设置在exists操作上的监视器将会被触发。
* 当所监视的znode被删除或其更新时，设置在getData上的监视器将会被触发，创建znode不会触发getData上的监视器，因为getData操作成功执行的前提是znode必须已经在。
* 当所监视的znode的一个子节点被创建或删除时，或监视的znode自己被删时，设置在getChildren操作上的监视器将会被触发。

设置监视器的操作及对应的触发器

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 设置触发器的操作 | 触发触发器的操作/受影响的节点 | | | | |
| create |  | delete |  | setData |
| znode | 子节点 | znode | 子节点 |  |
| Exists | NodeCreated |  | NodeDeleted |  | NodeDataChanged |
| getData |  |  | NodeDeleted |  | NodeDataChanged |
| getChildren |  |  | NodeDeleted | NodeDeleted |  |

(2)  数据访问控制

每个znode创建时间都会有一个ACL列表，用于决定谁可以执行那些操作。

(3)  临时节点

Zookeeper节点有两种：临时节点和持久节点。节点类型在创建时确定，并且不能修改。临时节点生命周期依赖创建它的会话，一旦会话结束，临时节点将会被删除。临时节点不允许有子节点。

(4)  顺序节点

当创建znode 时设置了顺序节点，那么该znode路径之后便会附加一个递增的计数，这个计数对于此节点的父节点来说是唯一的。

例如：一个客户端请求创建一个名为/a/b-的顺序节点，则所创建znode的名字可能是a/b-5，稍后另外一个名为/a/b-的顺序节点被创建，计数器会给出一个更大的值来保证znode名称的唯一性，例如/a/b-6。

# Zookeeper的安装

Zookeeper使用java编写，运行在jvm上，所以需要提前安装并配置好好java环境，推荐Oracle jdk1.7及以上版本。

## 官网

<http://zookeeper.apache.org/>

## 下载地址

<http://apache.opencas.org/zookeeper/>

从官方网站上下载tar.gz包，我们这里使用的是：zookeeper-3.4.7.tar.gz

## 安装前的准备

本安装示例中的zk集群共有三台机器，机器名分别为master、slave1、slave2

## 安装步骤

将gz文件放置在一个合适的地方，例如：/home/bigdata

**注意：以下举例中zookeeper所在的地址为/home/bigdata**

将gz文件解压缩：

# > tar –xzvf zookeeper-3.4.7.tar.gz

将解压后的文件重命名成zookeeper：

# > mv zookeeper-3.4.7 zookeeper

参数配置

进入到zookeeper的conf目录：

# > cd /home/bigdata/zookeeper/conf

将zoo\_sample.cfg文件删除掉：

# > rm –rf zoo\_sample.cfg

新建zk.cfg配置文件（这是行业惯例，实际上这个配置文件叫什么名字都可以，后缀为cfg）：

# > vi zk.cfg

在zk.cfg中配置服务端和链接等信息

在datadir制定的目录配置myid文件

### 单机模式（单台机器）

直接在zk.cfg中添加如下内容即可，具体每项的含义详见含义文档：

tickTime=2000

initLimit=10

syncLimit=5

dataDir=/home/bigdata/zkmyid

clientPort=2181

server.1= master:2888:3888

在/home/bigdata/ zkmyid目录下创建myid文件，并写入1

### zookeeper的启动

**如下描述法适用于单机方式或集群方式，伪分布方式与此稍有不同**

**需要提前配置zk\_home**

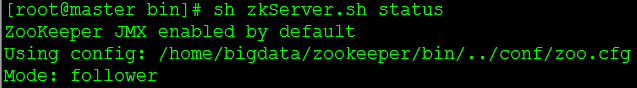
在每台安装zookeeper的机器上都执行如下命令：

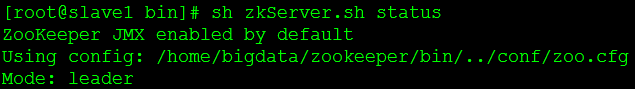
# sh zkServer.sh start

检测zookeeper是否启动成功：

# sh zkServer.sh status

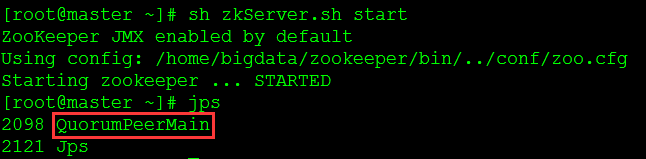
返回如下类似信息即表示启动成功：





注意：zookeeper也是java应用程序，通过jps命令也可查看到zookeeper的进程信息，如下图所示，可以看到zookeeper的进程为QuorumPeerMain。

quorum peer 对应的意思是 法定人数 对等 意即：zookeeper的server管理main方法。



小知识：jps所反映的信息即为一个java应用程序中真正运行的main方法所在类的类名。

# > sh zkCli.sh –server 192.168.221.200:2181

### Zookeeper的关闭

**如下描述法适用于单机方式或集群方式，伪分布方式与此稍有不同**

与启动类似，关闭方式如下：

需要在每台安装zookeeper的机器上都执行如下命令

# sh zkServer.sh stop

### 伪分布模式

单机中配置多个端口，产生多个进程，用多个进程模拟多台机器，需要在conf目录下配置多个配置文件，并且需要配置多个myid文件。

* **创建多个配置文件**

# vi /home/test/zk/conf/zk1.cfg

ckTime=2000

initLimit=10

syncLimit=5

dataDir=/home/test/zkmyid1

clientPort=2181

server.1=master:2888:3888

server.2=master:2889:3889

server.3=master:2890:3890

# vi /home/test/zk/conf/zk2.cfg

ckTime=2000

initLimit=10

syncLimit=5

dataDir=/home/test/zkmyid2

clientPort=2182

server.1=master:2888:3888

server.2=master:2889:3889

server.3=master:2890:3890

# vi /home/test/zk/conf/zk3.cfg

tickTime=2000

initLimit=10

syncLimit=5

dataDir=/home/test/zkmyid3

clientPort=2183

server.1=master:2888:3888

server.2=master:2889:3889

server.3=master:2890:3890

* **创建环境目录（为了存放多个myid文件）**

# mkdir /home/test/zkmyid1

# mkdir /home/test/zkmyid2

# mkdir /home/test/zkmyid3

* **在各个目录下均创建myid文件并填入相应的值**

操作方式：略

* **启动伪分布式集群**

# /home/test/zk/bin/zkServer.sh start zk1.cfg

# /home/test/zk/bin/zkServer.sh start zk2.cfg

# /home/test/zk/bin/zkServer.sh start zk3.cfg

相比较单机方式启动而言，此种方式指定了启动时所需要的配置文件，所以略有不同。类似的，查看状态或关闭时也需要指定配置文件。

# jps

5422 QuorumPeerMain

5395 QuorumPeerMain

5463 QuorumPeerMain

5494 Jps

* **查看伪分布式集群状态**

# /home/test/zk/bin/zkServer.sh status zk1.cfg

# /home/test/zk/bin/zkServer.sh status zk2.cfg

# /home/test/zk/bin/zkServer.sh status zk3.cfg

* **关闭伪分布式集群**

# /home/test/zk/bin/zkServer.sh stop zk1.cfg

# /home/test/zk/bin/zkServer.sh stop zk2.cfg

# /home/test/zk/bin/zkServer.sh stop zk3.cfg

### 集群模式

使用多台机器，确保每台机器的配置文件中都写上下列配置

创建新目录

#mkdir /home/bigdata/zookeeper/zkmyid

创建新文件

#vi /home/bigdata/zookeeper/zkmyid/myid

修改myid文件内容

# vi /home/bigdata/zookeeper/zkmyid/myid

1

创建新文件

Vi /home/bigdata/zookeeper/conf/zoo.cfg

多台机器中每台机器的配置文件内容都相同，内容如下：

tickTime=2000

initLimit=10

syncLimit=5

dataDir=/home/bigdata/zkmyid

clientPort=2181

server.1=master:2888:3888

server.2=slave1:2888:3888

server.3=slave2:2888:3888

配置每台机器上的myid文件（保证每台服务器的唯一性）

**每一台机器都需要做如下操作：**

在上面的配置项**dataDir**所指定的地址下新建一个名字为myid的文件，文件中的内容对应为server.后面的数字

注意以下所有演示中所涉及到的前提条件：

默认已经配置好ZOOKEEPER\_HOME了

export ZK\_HOME=/bigdata/zookeeper

exprt PATH=$PATH:$ZK\_HOME/bin:$ZK\_HOME/conf

zookeeper使用集群安装的方式

# Zookeeper shell的使用

## 一：zk的数据节点的特性之临时节点和永久节点

**注意：我们的每一步操作都是有记录号的，在下述操作时可进行观察**

### **开启zk客户端连接zkserver1**

# sh zkCli.sh -server master:2181

Zookeeper功能 用途：服务高可用 、 保证数据一致性

当有2个或以上的同一种角色 脑裂

Active standby zkfc set

### 创建一个临时节点和一个永久节点（并同时给节点赋值）

zookeeper > create -e /tempnode 111

zookeeper > create /normalnode 222

### 查询所有的节点

zookeeper > ls /



**注意：这时候两个节点都是存在的**

### 退出本次会话

zookeeper > quit

* **重新创建会话**

# > sh zkCli.sh –server 192.168.221.200:2181

* **查看所有的节点**



**注意：这时只剩下normal节点，表示临时节点在会话结束后会丢失**

利用临时性节点的特性，我们可以利用我们自己的集群中的每台机器都作为zk的客户端，每个客户端都像server端注册一个临时节点，在任一时刻我们通过ls命令即可查看zk的数据节点中的临时节点的情况，即可映射出集群中的机器的状况，后期可以通过使用java api 形式的客户端，结合web应用程序进行展示，就可以更加形象的展示出集群中的机器的存活状况。

## 二：zk集群中每台server之间的数据同步（为了保证zk本身的高可用）和leader选举

环境准备：

使用三个server组成zookeeper的集群。

在一台客户端机器上同时连接三个server，即在一台客户端机器上打开三个终端：

Terminal 1中执行：

# > sh zkCli.sh –server 192.168.221.200:2181

Terminal 2中执行：

# > sh zkCli.sh –server 192.168.211.201:2181

Terminal3中执行：

# > sh zkCli.sh –server 192.168.211.202:2181

先看一下三个server上的数据节点的情况：





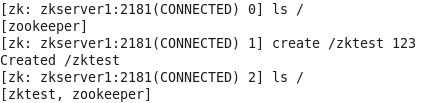


在一zkserver1上创建一个节点

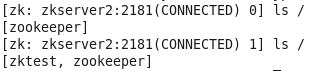
在terminal1 中创建一个节点并查看结果：

create /zktest1 123

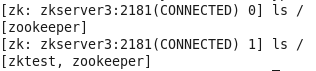
ls /



在Terminal 2 中查看zkserver2上的节点：



在Terminal3中查看zkserver3上的节点



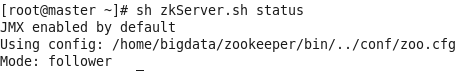
通过实际操作可以看到，另外的server上的数据也同时被更改了，这样的话，使用zookeeper自身集群的方式，多个server上的数据是近实时同步的，这样就可以保证作为leader的那个对外提供服务的节点坏了之后可以使用其他server，而且数据并没有丢失。所以可以提高zk的可用性，防止因为zookeeper自身的单点故障导致我们集群不可用。

**注意：由于leader的选举是随机的，所以每次测试的结果都有可能不同。**

这时候可以通过执行以下命令查看三台机器的角色类型：

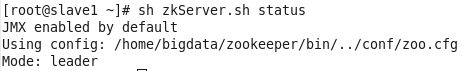
在zkserver1上的terminal中执行：

# > sh zkServer.sh status



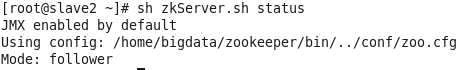
在zkserver2上的terminal中执行：

# > sh zkServer status



在zkserver3上的terminal中执行：

# > sh zkServer status

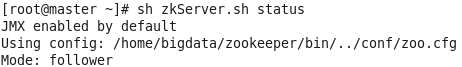


现在，关闭leader所在机器上的zookeeper的服务，查看另外两台的状态，本此测试中leader为zkserver2。

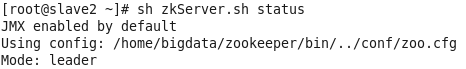
在zkserver2上关闭zk服务：

# > sh zkServer.sh stop

此时，本次测试中，zkserver1上的状态为：



本次测试中，zkserver3上的状态为：



可以明显看出，由zkserver3这台机器担任leader角色了。

# Zookeeper java API的使用

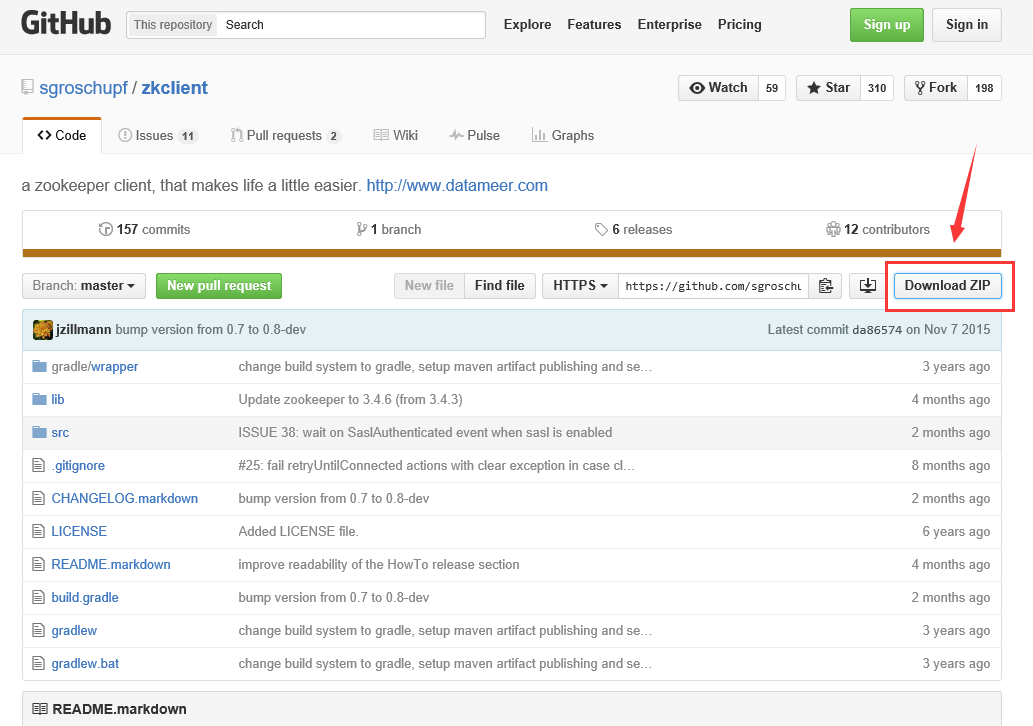
## 前提

因需要与zookeeper程序进行交互，故需要将安装后的zookeeper的lib目录下的相关jar包拷贝出来一份，用作我们写的程序的引用jar。

若要使用zk的常见框架zkClient，还需要添加zkClient相关的jar包，zkClient是Github上的一个开源项目，地址为：

<https://github.com/sgroschupf/zkclient>

下载方式见下图：



关于客户端的使用需要在项目中使用maven添加对zkClient的依赖。对于zkClient，大家了解即可。

## 使用

详见代码

## 使用总结

**org.apache.zookeeper.ZooKeeper类的主要方法列表**

|  |  |
| --- | --- |
| 方法名称 | 描述 |
| String create(final String path, byte data[], List acl, CreateMode createMode) | 创建一个znode节点，  参数： 路径、 znode内容，ACL(访问控制列表)、 znode创建类型 |
| void delete(final String path, int version) | 删除一个znode节点，  参数： 路径、版本号；如果版本号与znode的版本号不一致，将无法删除，是一种乐观加锁机制；如果将版本号设置为-1，不会去检测版本，直接删除； |
| Stat exists(final String path, Watcher watcher) | 判断某个znode节点是否存在  参数： 路径、Watcher(监视器)；当这个znode节点被改变时，将会触发当前Watcher |
| Stat exists(String path, boolean watch) | 判断某个znode节点是否存在  参数： 路径、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 |
| Stat setData(final String path, byte data[], int version) | 设置某个znode上的数据  参数： 路径、数据、版本号；如果为-1，跳过版本检查 |
| byte[] getData(final String path, Watcher watcher, Stat stat) | 获取某个znode上的数据  参数： 路径、监视器、数据版本等信息 |
| List getChildren(final String path, Watcher watcher) | 获取某个节点下的所有子节点  参数： 路径、监视器;该方法有多个重载 |

Znode创建类型(CreateMode)：

|  |  |
| --- | --- |
| PERSISTENT | 持久化节点 |
| PERSISTENT\_SEQUENTIAL | 顺序自动编号持久化节点，这种节点会根据当前已存在的节点数自动加1 |
| EPHEMERAL | 临时节点，客户端session超时这类节点就会被自动删除 |
| EPHEMERAL\_SEQUENTIAL | 临时自动编号节点 |

# Zk常见案例之Hadoop NameNode HA

详见Hadoop NameNode HA文档HDFS HA.PDF

# Zookeeper其他的典型应用场景

## 数据发布与订阅（配置中心）

发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到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，服务式服务框架的服务地址列表等就非常适合使用。

* 应用中用到的一些配置信息放到ZK上进行集中管理。这类场景通常是这样：应用在启动的时候会主动来获取一次配置，同时，在节点上注册一个Watcher，这样一来，以后每次配置有更新的时候，都会实时通知到订阅的客户端，从来达到获取最新配置信息的目的。
* 分布式搜索服务中，索引的元信息和服务器集群机器的节点状态存放在ZK的一些指定节点，供各个客户端订阅使用。
* 分布式日志收集系统。这个系统的核心工作是收集分布在不同机器的日志。收集器通常是按照应用来分配收集任务单元，因此需要在ZK上创建一个以应用名作为path的节点P，并将这个应用的所有机器ip，以子节点的形式注册到节点P上，这样一来就能够实现机器变动的时候，能够实时通知到收集器调整任务分配。
* 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的发问。通常是暴露出接口，例如JMX接口，来获取一些运行时的信息。引入ZK之后，就不用自己实现一套方案了，只要将这些信息存放到指定的ZK节点上即可。

**注意**：在上面提到的应用场景中，有个默认前提是：数据量很小，但是数据更新可能会比较快的场景。

## 2.2 负载均衡

这里说的负载均衡是指软负载均衡。在分布式环境中，为了保证高可用性，通常同一个应用或同一个服务的提供方都会部署多份，达到对等服务。而消费者就须要在这些对等的服务器中选择一个来执行相关的业务逻辑，其中比较典型的是消息中间件中的生产者，消费者负载均衡。

消息中间件中发布者和订阅者的负载均衡，linkedin开源的KafkaMQ和阿里开源的[metaq](http://metaq.taobao.org/)都是通过zookeeper来做到生产者、消费者的负载均衡。这里以metaq为例如讲下：

**生产者负载均衡：**

metaq发送消息的时候，生产者在发送消息的时候必须选择一台broker上的一个分区来发送消息，因此metaq在运行过程中，会把所有broker和对应的分区信息全部注册到ZK指定节点上，默认的策略是一个依次轮询的过程，生产者在通过ZK获取分区列表之后，会按照brokerId和partition的顺序排列组织成一个有序的分区列表，发送的时候按照从头到尾循环往复的方式选择一个分区来发送消息。

**消费负载均衡：**

在消费过程中，一个消费者会消费一个或多个分区中的消息，但是一个分区只会由一个消费者来消费。MetaQ的消费策略是：

* 每个分区针对同一个group只挂载一个消费者。
* 如果同一个group的消费者数目大于分区数目，则多出来的消费者将不参与消费。
* 如果同一个group的消费者数目小于分区数目，则有部分消费者需要额外承担消费任务。

在某个消费者故障或者重启等情况下，其他消费者会感知到这一变化（通过 zookeeper watch消费者列表），然后重新进行负载均衡，保证所有的分区都有消费者进行消费。

## 2.3 命名服务(Naming Service)

命名服务也是分布式系统中比较常见的一类场景。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名字来获取资源或服务的地址，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，远程对象等等——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。

阿里巴巴集团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，[点击这里](http://code.alibabatech.com/wiki/display/dubbo/Home)查看Dubbo开源项目。在Dubbo实现中：

**服务提供者**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。

## 2.4 分布式通知/协调

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

* 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。
* 另一种系统调度模式：某系统有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。
* 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

## 2.5 集群管理与Master选举

### 2.5.1 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：

* 集群中机器有变动的时候，牵连修改的东西比较多。
* 有一定的延时。

利用ZooKeeper有两个特性，就可以实时另一种集群机器存活性监控系统：

* 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。
* 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。

例如，监控系统在 /clusterServers 节点上注册一个Watcher，以后每动态加机器，那么就往 /clusterServers 下创建一个 EPHEMERAL类型的节点：/clusterServers/{hostname}. 这样，监控系统就能够实时知道机器的增减情况，至于后续处理就是监控系统的业务了。

### 2.5.2 Master选举

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处理），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 /currentMaster 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取了。

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。

上文中提到，所有客户端创建请求，最终只有一个能够创建成功。在这里稍微变化下，就是允许所有请求都能够创建成功，但是得有个创建顺序，于是所有的请求最终在ZK上创建结果的一种可能情况是这样： /currentMaster/{sessionId}-1 ,?/currentMaster/{sessionId}-2 ,?/currentMaster/{sessionId}-3 ….. 每次选取序列号最小的那个机器作为Master，如果这个机器挂了，由于他创建的节点会马上小时，那么之后最小的那个机器就是Master了。

### 2.5.3 搜索系统

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，而且不能保证彼此之间索引数据一致。因此让集群中的Master来进行全量索引的生成，然后同步到集群中其它机器。另外，Master选举的容灾措施是，可以随时进行手动指定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。

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，会重新选举出一个HMaster来运行，从而避免了 HMaster的单点问题

## 2.6 分布式锁

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。锁服务可以分为两类，一个是**保持独占**，另一个是**控制时序**。

所谓保持独占，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，就是所有视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（这个可以通过节点的属性控制：CreateMode.EPHEMERAL\_SEQUENTIAL来指定）。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。

## 2.7 分布式队列

队列方面，简单地讲有两种，一种是常规的先进先出队列，另一种是要等到队列成员聚齐之后的才统一按序执行。对于第一种先进先出队列，和分布式锁服务中的控制时序场景基本原理一致，这里不再赘述。

第二种队列其实是在FIFO队列的基础上作了一个增强。通常可以在 /queue 这个znode下预先建立一个/queue/num 节点，并且赋值为n（或者直接给/queue赋值n），表示队列大小，之后每次有队列成员加入后，就判断下是否已经到达队列大小，决定是否可以开始执行了。这种用法的典型场景是，分布式环境中，一个大任务Task A，需要在很多子任务完成（或条件就绪）情况下才能进行。这个时候，凡是其中一个子任务完成（就绪），那么就去 /taskList 下建立自己的临时时序节点（CreateMode.EPHEMERAL\_SEQUENTIAL），当 /taskList 发现自己下面的子节点满足指定个数，就可以进行下一步按序进行处理了。