Zookeeper

# 1 安装

## 1.1下载

可以从 <https://zookeeper.apache.org/releases.html> 下载ZooKeeper，目前最新的稳定版本为 3.4.8 版本，用户可以自行选择一个速度较快的镜像来下载即可。

或到zookeeper官网： zookeeper     <http://mirror.bit.edu.cn/apache/zookeeper/zookeeper-3.4.8/>

## 1.2 目录结构

### 1.2.1 解压

上传zookeeper-3.4.8.tar.gz到任意一台服务器的根目录，并解压：zookeeper：

# tar -zxvf zookeeper-3.4.8.tar.gz

* bin目录   
  zk的可执行脚本目录，包括zk服务进程，zk客户端，等脚本。其中，.sh是Linux环境下的脚本，.cmd是Windows环境下的脚本。
* conf目录   
  配置文件目录。zoo\_sample.cfg为样例配置文件，需要修改为自己的名称，一般为zoo.cfg。log4j.properties为日志配置文件。
* lib   
  zk依赖的包。
* contrib目录   
  一些用于操作zk的工具包。
* recipes目录   
  zk某些用法的代码示例

## 1.3 单机版

### 1.3.1 运行配置

上面提到，conf目录下提供了配置的样例zoo\_sample.cfg，要将zk运行起来，需要将其名称修改为zoo.cfg。   
打开zoo.cfg，可以看到默认的一些配置。

* tickTime   
  时长单位为毫秒，为zk使用的基本时间度量单位。例如，1 \* tickTime是客户端与zk服务端的心跳时间，2 \* tickTime是客户端会话的超时时间。   
  tickTime的默认值为2000毫秒，更低的tickTime值可以更快地发现超时问题，但也会导致更高的网络流量（心跳消息）和更高的CPU使用率（会话的跟踪处理）。
* clientPort   
  zk服务进程监听的TCP端口，默认情况下，服务端会监听2181端口。
* dataDir   
  无默认配置，必须配置，用于配置存储快照文件的目录。如果没有配置dataLogDir，那么事务日志也会存储在此目录。

### 1.3.2 启动

在Windows环境下，直接双击zkServer.cmd即可。在**[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "http://blog.csdn.net/lihao21/article/details/_blank)**环境下，进入bin目录，执行命令

./zkServer.sh start

这个命令使得zk服务进程在后台进行。如果想在前台中运行以便查看服务器进程的输出日志，可以通过以下命令运行：

./zkServer.sh start-foreground

执行此命令，可以看到大量详细信息的输出，以便允许查看服务器发生了什么。

使用文本编辑器打开zkServer.cmd或者zkServer.sh文件，可以看到其会调用zkEnv.cmd或者zkEnv.sh脚本。zkEnv脚本的作用是设置zk运行的一些环境变量，例如配置文件的位置和名称等

### 1.3.3 链接

如果是连接同一台主机上的zk进程，那么直接运行bin/目录下的zkCli.cmd（Windows环境下）或者zkCli.sh（Linux环境下），即可连接上zk。   
 直接执行zkCli.cmd或者zkCli.sh命令默认以主机号 127.0.0.1，端口号 2181 来连接zk，如果要连接不同机器上的zk，可以使用 -server 参数，例如：

bin/zkCli.sh -server 192.168.0.1:2181

## 1.4 集群模式

在集群模式下，建议至少部署3个zk进程，或者部署奇数个zk进程。如果只部署2个zk进程，当其中一个zk进程挂掉后，剩下的一个进程并不能构成一个[quorum](http://baike.baidu.com/link?url=pqWrzgH-_VhMLnscR1iRTpPjovfyhxG-8Qs9HxGutiGi5bhnA_lX_pmabLQ-3MiDeigcHRFMYSbFg90RAYVAta)的大多数。因此，部署2个进程甚至比单机模式更不可靠，因为2个进程其中一个不可用的可能性比一个进程不可用的可能性还大。

### 1.4.1 环境变量

# vi /etc/profile

export ZOOKEEPER\_HOME=/opt/zookeeper-3.4.8

export PATH=$ZOOKEEPER\_HOME/bin:$PATH或者export PATH=$PATH:$ZOOKEEPER\_HOME/bin:$ZOOKEEPER\_HOME/conf

# source /etc/profile

### 1.4.2运行配置

在集群模式下，所有的zk进程可以使用相同的配置文件（是指各个zk进程部署在不同的机器上面），例如如下配置：

|  |
| --- |
| tickTime=2000  dataDir=/home/myname/zookeeper  clientPort=2181  initLimit=5syncLimit=2  server.1=192.168.229.160:2888:3888  server.2=192.168.229.161:2888:3888  server.3=192.168.229.162:2888:3888 |

* **initLimit**  
   ZooKeeper集群模式下包含多个zk进程，其中一个进程为leader，余下的进程为follower。   
  当follower最初与leader建立连接时，它们之间会传输相当多的数据，尤其是follower的数据落后leader很多。initLimit配置follower与leader之间建立连接后进行同步的最长时间。
* syncLimit   
  配置follower和leader之间发送消息，请求和应答的最大时间长度，如果超时不能应答，则抛弃此follower
* tickTime   
  tickTime则是上述两个超时配置的基本单位，例如对于initLimit，其配置值为5，说明其超时时间为 2000ms \* 5 = 10秒。
* server.id=host:port1:port2   
  其中id为一个数字，表示zk进程的id，这个id也是dataDir目录下myid文件的内容。   
  host是该zk进程所在的IP地址，port1表示follower和leader交换消息所使用的端口，port2表示选举leader所使用的端口。
* dataDir   
  其配置的含义跟单机模式下的含义类似，不同的是集群模式下还有一个myid文件。在zookeeper的目录下创建新的目录data来作为dataDir的文件位置，并且创建文件myid，myid文件的内容只有一行，且内容只能为1 - 255之间的数字，这个数字亦即上面介绍server.id中的id，表示zk进程的id。
* 2888:3888

Zookeeper 是通过2888端口进行集群间数据一致性通讯

3888位zooleeper的选举端口，当有主leader挂掉后，通过此端口重新选举出新的端口

* minSessiontimeout=tickTime\*2
* maxSessiontimeout=tickTime\*20

zookeeper的规定：如果API指定的sesionTimeout在区间范围内2s~40s，zookeeper取得的就是api的指定时间。比如传的是1s，就用下限2s，如果传的是42s就用40s。

***注意 ：***  
 如果仅为了测试部署集群模式而在同一台机器上部署zk进程，server.id=host:port1:port2配置中的port参数必须不同。但是，为了减少机器宕机的风险，强烈建议在部署集群模式时，将zk进程部署不同的物理机器上面。

### 1.4.3 远程拷贝

scp -r zookeeper-3.4.8/ 192.168.43.150:/usr/local/src/zk

### 1.4.4启动

|  |
| --- |
| CentOS 7.0默认使用的是firewall作为防火墙，使用iptables必须重新设置一下  1、直接关闭防火墙  systemctl stop firewalld.service #停止firewall  systemctl disable firewalld.service #禁止firewall开机启动  2、设置 iptables service  yum -y install iptables-services  如果要修改防火墙配置，如增加防火墙端口3306  vi /etc/sysconfig/iptables  增加规则  -A INPUT -m state --state NEW -m tcp -p tcp --dport 3306 -j ACCEPT  保存退出后  systemctl restart iptables.service #重启防火墙使配置生效  systemctl enable iptables.service #设置防火墙开机启动 |

假如我们打算在三台不同的机器 192.168.229.160，192.168.229.161，192.168.229.162上各部署一个zk进程，以构成一个zk集群。   
三个zk进程均使用相同的 zoo.cfg 配置：

tickTime=2000 dataDir=/home/myname/zookeeper clientPort=2181 initLimit=5 syncLimit=2

server.1=192.168.229.160:2888:3888

server.2=192.168.229.161:2888:3888

server.3=192.168.229.162:2888:3888

在三台机器dataDir目录（ /home/myname/zookeeper 目录）下，分别生成一个myid文件，其内容分别为1，2，3。然后分别在这三台机器上启动zk进程，这样我们便将zk集群启动了起来。

启动：sh zkServer.sh start

状态：./zkServer.sh status

关闭：./zkServer.sh stop

### 1.4.5 链接

bin/zkCli.sh -server 192.168.229.160:2181,192.168.229.161:2181,192.168.229.162:2181

### 1.2.6选举机制

基于Paxos这个算法实现，这个算法的核心新思想是:

要解决在分布式环境下，就某一协议达成一致的算法。

缺点：容易在分布式下产生活锁。

Zk 用的是fast paxos算法，相当于做了改进。

1. 数据恢复阶段：

Zookeeper从本次磁盘里找到一个最大的事物id，也就意味着最新的事务

事物概念：客户端做的新建、更新等操作都是事务，每个事务都会产生事务id，并且这个id是递增的。

cZxid表示创建节点的事务id

mZxid表示修改此节点的事务id

1. 选举阶段：

提交选举议案：

1. 最大事务id
2. 当前机器的领导id（就是myid里的数字）
3. 当前时钟值（记录了当前选举的轮数，作用是确保多个zk处于同一轮选举当中）
4. 当前节点状态：a、Looking表示正处在选举阶段。

b、Leading正在任职阶段

C、Following正在做小弟阶段

选举原则：第一、比谁的事务id大

第二、事务id比不出来，就比领导id，大的为leader

第三、必须满足过半统一机制

### 1.2.7 Leader 工作

做原子广播，目的是要达到zk服务器集群的数据一致性，因为leader肯定是具有最大事务id的，即他具有最新的数据，所以要听过2888原子广播端口与其他follower进行数据一致性。

当zk服务器集群收到事务请求时，当zk服务器内部过半对此应用表决成功，则此事务才真正的提交成功。

# 2 介绍

## 2.1 名字空间

ZooKeeper的最重要核心就是一个精简文件系统，提供一些简单的操作以及附加的抽象（例如排序和通知）。

ZooKeeper的整个名字空间的结构是层次化的，和一般的**[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "http://blog.csdn.net/rengq126/article/details/_blank)**文件系统结构非常相似，一颗很大的树。这也就是ZooKeeper的数据结构情况。名字空间的层次由斜杠/来进行分割，在名称空间里面的每一个结点的名字空间唯一由这个结点的路径来确定。

每一个节点拥有自身的一些信息，包括：数据、数据长度、创建时间、修改时间等等。从这样一类既含有数据，又作为路径表标示的节点的特点中，可以看出，ZooKeeper的节点既可以被看做是一个文件，又可以被看做是一个目录，它同时具有二者的特点。为了便于表达，今后我们将使用Znode来表示所讨论的ZooKeeper节点。

## 2.2 节点zNode

* Regular ZNode: 常规型ZNode, 用户需要显式的创建、删除
* Ephemeral ZNode: 临时型ZNode, 用户创建它之后，可以显式的删除，也可以在创建它的Session结束后，由ZooKeeper Server自动删除

ZNode还有一个Sequential的特性，如果创建的时候指定的话，该ZNode的名字后面会自动Append一个不断增加的SequenceNo。

Znode维护着数据、ACL（access control list，访问控制列表）、时间戳等交换版本号等数据结构，它通过对这些数据的管理来让缓存生效并且令协调更新。每当Znode中的数据更新后它所维护的版本号将增加，这非常类似于**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/rengq126/article/details/_blank)**中计数器时间戳的操作方式。

另外Znode还具有原子性操作的特点：命名空间中，每一个Znode的数据将被原子地读写。读操作将读取与Znode相关的所有数据，写操作将替换掉所有的数据。除此之外，每一个节点都有一个访问控制列表，这个访问控制列表规定了用户操作的权限。

    ZooKeeper中同样存在临时节点。这些节点与session同时存在，当session生命周期结束，这些临时节点也将被删除。临时节点在某些场合也发挥着非常重要的作用。

## 2.3 watch机制

Watch机制就和单词本身的意思一样，看。看什么？具体来讲就是某一个或者一些Znode的变化。官方给出的定义：一个Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端，以便通知它们。

Watch机制主要有以下三个特点：

### 2.3.1 一次性的触发器（one-time trigger）

当数据改变的时候，那么一个Watch事件会产生并且被发送到客户端中。但是客户端只会收到一次这样的通知，如果以后这个数据再次发生改变的时候，之前设置Watch的客户端将不会再次收到改变的通知，因为Watch机制规定了它是一个一次性的触发器。

### 2.3.2 发送给客户端

这个表明了Watch的通知事件是从服务器发送给客户端的，是异步的，这就表明不同的客户端收到的Watch的时间可能不同，但是ZooKeeper有保证：当一个客户端在看到Watch事件之前是不会看到结点数据的变化的。例如：A=3，此时在上面设置了一次Watch，如果A突然变成4了，那么客户端会先收到Watch事件的通知，然后才会看到A=4。

### 2.3.3 被设置Watch的数据

这表明了一个结点可以变换的不同方式。一个Znode变化方式有两种，结点本身数据的变化以及结点孩子的变化。因此Watch也可以设置为这个Znode的结点数据，当然也可以设置为Znode结点孩子。

## 2.4 ACL访问控制列表

ZooKeeper使用ACL来控制对旗下Znode结点们的访问，在ZooKeeper中，ACL通过设置ID以及与其关联的权限来完成访问控制的。ACL的权限组成语法是：

(scheme:expression, perms)

前者表明设置的ID，逗号后面表示的是ID相关的权限，例如：

  (ip:172.16.16.1, READ) 只读

以下列举以下ACL所具有的权限

CREATE：表明你可以创建一个Znode的子结点。

READ：你可以得到这个结点的数据以及列举该结点的子结点情况。

WRITE：设置一个结点的数据。

DELETE：可以删除一个结点

ADMIN：对一个结点设置权限。

## 2.5 观察者状态

比如针对事务决议的表决，通过投票表决。过半同意事务就更新

如果从性能的角度来看参与投票的机器越少越好（但至少要满足过半投票人数）

所以zookeeper引入了observer观察者状态

引入观察者，一是通过减少投票机器数量，从而提高性能。

此外，也可以从降低每一台zk服务器的负载压力角度来看，能够降低负载压力。

因为观察者如果宕机，影响也不大，因为不是核心人员。

但是注意，引入观察者，从性能角度来看，是好东西。但是如果从集群高可用的角度来看，要 慎用。

此外再注意，比如3台机器，把两台配置成观察者，这么做是不行的，因为不满足过半机制， 所以选不出leader,导致集群启动不了。

特点：

1不参与投票

2 观察和监听投票结构，然后去更新状态。

设置观察者步骤：

1 在要编程观察者的一起的zk的配置文件

加：peerType=observer

Server.3=ip:2888:3888:observer

2 更改其他机器的配置文件

加 server.3=ip:2888:3888:observer

## 2.6 特性

1.数据一致性

无论从哪一台zk服务器读取数据，数据都是一样的。（也叫单一视图）。是zookeeper最重要 的特性。

2.可靠性

当一个事务被成功更新后，不会因为网络波动或其他原因而回滚

3.原子性

一个事务，要么都更新成功，要么都更新失败。不会有中间状态

4.实时性

能够在很短的时间内，将事件的状态变化反映出来。（前提是网络状态良好，没有网络波动）

5.顺序一致性

比如客户端A，发出三个指令：x=a,x=b,x=c。但是在网络波动的情况下，可能到达顺序是 x=b,x=c,x=a。Zk能够确保发送的顺序和最后执行的顺序是一致的，从而确保最后的结果是 正确的

### 2.5.1 读、写(更新)模式 在ZooKeeper集群中，读可以从任意一个ZooKeeper Server读，这一点是保证ZooKeeper比较好的读性能的关键；写的请求会先Forwarder到Leader，然后由Leader来通过ZooKeeper中的原子广播协议，将请求广播给所有的Follower，Leader收到一半以上的写成功的Ack后，就认为该写成功了，就会将该写进行持久化，并告诉客户端写成功了。

### 2.5.2 WAL和Snapshot 和大多数分布式系统一样，ZooKeeper也有WAL(Write-Ahead-Log)，对于每一个更新操作，ZooKeeper都会先写WAL, 然后再对内存中的数据做更新，然后向Client通知更新结果。另外，ZooKeeper还会定期将内存中的目录树进行Snapshot，落地到磁盘上，这个跟HDFS中的FSImage是比较类似的。这么做的主要目的，一当然是数据的持久化，二是加快重启之后的恢复速度，如果全部通过Replay WAL的形式恢复的话，会比较慢。

### 2.5.3 FIFO 对于每一个ZooKeeper客户端而言，所有的操作都是遵循FIFO顺序的，这一特性是由下面两个基本特性来保证的：一是ZooKeeper Client与Server之间的网络通信是基于TCP，TCP保证了Client/Server之间传输包的顺序；二是ZooKeeper Server执行客户端请求也是严格按照FIFO顺序的。

### 2.5.4 Linearizability 在ZooKeeper中，所有的更新操作都有严格的偏序关系，更新操作都是串行执行的，这一点是保证ZooKeeper功能正确性的关键。

### 2.6.5 数据一致性

# 3 访问zookeeper

## 3.1 通过命令行

  ZooKeeper命令行工具类似于Linux的shell环境，不过功能肯定不及shell啦，但是使用它我们可以简单的对ZooKeeper进行访问，数据创建，数据修改等操作。

当启动 ZooKeeper 服务成功之后，输入下述命令，连接到 ZooKeeper 服务：

zkCli.sh –server 202.115.36.251:2181

连接成功后，系统会输出 ZooKeeper 的相关环境以及配置信息，并在屏幕输出“ Welcome to ZooKeeper ”等信息。

命令行工具的一些简单操作如下：

1 ）使用 ls 命令来查看当前 ZooKeeper 中所包含的内容：

[zk: 202.115.36.251:2181(CONNECTED) 1] ls /

2 ）创建一个新的 znode ，使用 create /zk myData 。这个命令创建了一个新的 znode 节点“ zk ”以及与它关联的字符串：

[zk: 202.115.36.251:2181(CONNECTED) 2] create /zk "myData"

3 ）我们运行 get 命令来确认 znode 是否包含我们所创建的字符串：

[zk: 202.115.36.251:2181(CONNECTED) 3] get /zk

4 ）下面我们通过 set 命令来对 zk 所关联的字符串进行设置：

[zk: 202.115.36.251:2181(CONNECTED) 4] set /zk "zsl"

5 ）下面我们将刚才创建的 znode 删除：

[zk: 202.115.36.251:2181(CONNECTED) 5] delete /zk

## 3.2 JAVA Client API

* create(path, data, flags): 创建一个ZNode, path是其路径，data是要存储在该ZNode上的数据，flags常用的有: PERSISTEN, PERSISTENT\_SEQUENTAIL, EPHEMERAL, EPHEMERAL\_SEQUENTAIL
* delete(path, version): 删除一个ZNode，可以通过version删除指定的版本, 如果version是-1的话，表示删除所有的版本
* exists(path, watch): 判断指定ZNode是否存在，并设置是否Watch这个ZNode。这里如果要设置Watcher的话，Watcher是在创建ZooKeeper实例时指定的，如果要设置特定的Watcher的话，可以调用另一个重载版本的exists(path, watcher)。以下几个带watch参数的API也都类似
* getData(path, watch): 读取指定ZNode上的数据，并设置是否watch这个ZNode
* setData(path, watch): 更新指定ZNode的数据，并设置是否Watch这个ZNode
* getChildren(path, watch): 获取指定ZNode的所有子ZNode的名字，并设置是否Watch这个ZNode
* sync(path): 把所有在sync之前的更新操作都进行同步，达到每个请求都在半数以上的ZooKeeper Server上生效。path参数目前没有用
* setAcl(path, acl): 设置指定ZNode的Acl信息
* getAcl(path): 获取指定ZNode的Acl信息

### 3.2.1 创建可重复利用的zookeeper

|  |
| --- |
| **public class** MyWather **implements** Watcher {  **private static** CountDownLatch *latch*=**new** CountDownLatch(1);   **public static void** main(String[] args) **throws** IOException, InterruptedException {  ZooKeeper zk=**new** ZooKeeper(**"192.168.8.160:2181"**,3000,**new** MyWather());  System.***out***.println(zk.getState());  *latch*.await();  **long** sessionId=zk.getSessionId();  **byte**[] passwd=zk.getSessionPasswd();  zk=**new** ZooKeeper(**"192.168.8.160:2181"**,3000,**new** MyWather(),sessionId,passwd);  Thread.*sleep*(Integer.***MAX\_VALUE***);  }  **public void** process(WatchedEvent watchedEvent) {  System.***out***.println(**"Receive watchd event:"**+watchedEvent);  **if**(watchedEvent.getState()== Event.KeeperState.***SyncConnected***){  *latch*.countDown();  }  } } |

### 3.2.2 创建节点

AsyncCallback包含了StatCallback。DataCallBack、ACLCallBack、ChildrenCallBack、Children2CallBack、StringCallBack、VoidCallBack七中不同的接口。

同步创建接口错误会抛出异常，异步创建接口不会跑异常，会在回调接口中以响应码的形式体现。

同步创建节点名字重复会保存，异步不会报错，返回-110表示创建失败

使用同步节点创建接口

|  |
| --- |
| String path=zk.create(**"/park2"**,**""**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE*** , CreateMode.***EPHEMERAL***); System.***out***.println(**"Success create zNode:"**+path); String path2=zk.create(**"/park2"**,**""**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE*** , CreateMode.***EPHEMERAL\_SEQUENTIAL***); System.***out***.println(**"Success create zNode:"**+path2);  结果：  Success create zNode:/park2  Success create zNode:/park20000000003 |

异步创建节点：

|  |
| --- |
| **class** IStringCallBack **implements** AsyncCallback.StringCallback{  *//创建节点完后，zook客户端会自动调用这个方法，可以处理相关业务  //rc 状态吗，成功返回0，不成功返回-110，-4客户端和服务器端已经断开连接，-112会话已经过期  //path 用户指定的路径  //ctx 构造函数里传递的数据  //name 实际创建的路径节点名称* **public void** processResult(**int** rc, String path, Object ctx, String name) {  System.***out***.println(**"crate path result"**+rc+**"==="**+path+**"==="**+ctx+**"==="**+name);  } }  zk.create(**"/park3"**,**""**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE*** , CreateMode.***EPHEMERAL***,**new** IStringCallBack(),**"i am contect"**); zk.create(**"/park3"**,**""**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE*** , CreateMode.***EPHEMERAL***,**new** IStringCallBack(),**"i am context"**); zk.create(**"/park3"**,**""**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE*** , CreateMode.***EPHEMERAL\_SEQUENTIAL***,**new** IStringCallBack(),**"i am context"**); Thread.*sleep*(Integer.***MAX\_VALUE***);  结果：  crate path result0===/park3===i am contect===/park3  crate path result-110===/park3===i am context===null  crate path result0===/park3===i am context===/park30000000011 |

### 3.2.3获取子节点

List<String> getChildren(final String path,Wathcer wc)

List<String> getChildren(final String path,boolean watch)

Void getChildren(final String path,Watcher watch,ChildrenCallBack cb,Object ctx)

Void getChildren(final String path,boolean watch,ChildrenCallBack cb,Object ctx)

List<String> getChildren(final String path,Wathcer wc,Stat stat)

List<String> getChildren(final String path,boolean wc,Stat stat)

Void getChildren(final String path,Watcher watch,Children2CallBack cb,Object ctx)

Void getChildren(final String path,boolean watch,Children2CallBack cb,Object ctx)

Watch如果使用boolean值为true，会调用一个默认的Watcher

Stat用法在接口中传入一个旧的stat变量，该stat变量会方法在执行的过程中。被来自服务端响应的新的stat锁替换。

异步通常用在：应用启动的时候，会获得一些配置信息，例如机器列表，但这些配置信息一般都比较大，并不希望配置的获取影响应用的主流程。

同步获取

|  |
| --- |
| **public void** process(WatchedEvent watchedEvent) {  System.***out***.println(**"Receive watchd event:"**+watchedEvent);  **if**(watchedEvent.getState()== Event.KeeperState.***SyncConnected***){  **if**(Event.EventType.***None***==watchedEvent.getType()&&**null**==watchedEvent.getPath()){  *latch*.countDown();  }**else if**(watchedEvent.getType()== Event.EventType.***NodeChildrenChanged***){  **try** {  System.***out***.println(*zk*.getChildren(watchedEvent.getPath(),**true**));  } **catch** (Exception e) {  e.printStackTrace();  }  }  } }  *zk*.create(**"/park3"**,**""**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,CreateMode.***PERSISTENT***); *zk*.create(**"/park3/c1"**,**""**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,CreateMode.***EPHEMERAL***);  //true表示调用创建zk链接时的watcher List<String> childrenList=*zk*.getChildren(**"/park3"**,**true**); System.***out***.println(childrenList); *zk*.create(**"/park3/c2"**,**""**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,CreateMode.***EPHEMERAL***); Thread.*sleep*(Integer.***MAX\_VALUE***);  结果：  Receive watchd event:WatchedEvent state:SyncConnected type:None path:null  [c1]  Receive watchd event:WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/park3  [c1, c2] |

### 3.2.4 获取数据

Byte[] getData(final String path,Watcher watch,Stat stat)

Byte[] getData(final String path,boolean watch,Stat stat)

Byte[] getData(final String path,Watcher watch,DataCallback cb,Object ctx)

Byte[] getData(final String path,boolean watch,DataCallback cb,Object ctx)

节点的数据变化、版本变化都会触发DataNodeChanged通知。

### 3.2.5 更新节点数据

同步跟新和异步更新

Stat setData(final String path,byte data[],int version)

Stat setData(final String path,byte data[],int version,StatCallBack cb,Obejct ctx)

|  |
| --- |
| String path=**"/park5"**; *zk*.create(path,**"gao"**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,CreateMode.***EPHEMERAL***); *zk*.getData(path,**true**,**null**); Stat stat=*zk*.setData(path,**"jian"**.getBytes(),-1); *zk*.setData(path,**"jianjian"**.getBytes(),stat.getVersion()); **try** {  *zk*.setData(path, **"jianjianjian"**.getBytes(), stat.getVersion()); }**catch** (Exception e){  System.***out***.println(**"duplicated!"**); } |

### 3.2.6 检查节点是否存在

无论节点是否存在，通过调用exists都可以注册时间通知Watcher

对于指定节点的子节点的所有变化都不会通知客户端

Exists注册的watcher能够对节点的创建、节点的删除、节点的数据更新时间进行监听。

Public void exists(final String path,Watcher watch)

Public void exists(String path,boolean watch)

Public void exists(final String path,Watcher watch,StatCallBack cb,Object ctx)

Public void exists(String path,boolean watch,StatCallBack cb,Object ctx)

### 3.2.7 权限控制

Zookeeper提供了 多种权限控制模式(下面的参数schema)分别是world、Auth、digest、ip、super

主要讲解digest模式

权限信息（AuthInfo）设置：

addAuthInfo（String schema，byte[] auth） auth为具体的权限信息

添加权限信息后，凡是通过该会话对zk服务器端进行任何操作都会带上该权限信息

使用包含权限信息的会话zk创建节点：

|  |
| --- |
| ZooKeeper zk=**new** ZooKeeper(**"192.168.8.160:2181"**,3000,**null**); zk.addAuthInfo(**"digest"**,**"foo:true"**.getBytes()); zk.create(**"/zk-test"**,**"123"**.getBytes(), ZooDefs.Ids.***CREATOR\_ALL\_ACL***, CreateMode.***EPHEMERAL***); Thread.*sleep*(Integer.***MAX\_VALUE***); |

当客户端对一个数据节点添加权限信息后，对于删除操作而言，其范围是其子节点。也就是说给一个节点添加权限信息后仍然可以自由的删除该节点，但是删除其子节点需要权限信息。

## 3.3 集群命令

安装nc（NetCat）网络工具，通过TCP或者UDP传输数据

安装nc需要先安装glibc

Yum list glibc\* 通过yum原查看glibc相关资源

### 1 更换yum源：

中科大的yum源：

wget http://centos.ustc.edu.cn/CentOS-Base.repo

sohu的yum源

wget http://mirrors.sohu.com/help/CentOS-Base-sohu.repo

1. cd /etc/yum.repos.d

2. mv CentOS-Base.repo CentOS-Base.repo.backup

3. wget http://mirrors.163.com/.help/CentOS6-Base-163.repo

4. mv CentOS6-Base-163.repo CentOS-Base.repo

5.yum makecache

### 2 安装nc的依赖

yum install glibc.i686

### 3 安装nc

Rpm -qa|grep netcat 查看nc

Rpm -ivh netcat-0.7.1-1.i386.rpm 安装nc

rpm -ev --nodeps mysql-libs-5.1.71-1.el6.x86\_64 卸载应用

### 4 常用命令

echo stat|nc 127.0.0.1 2181 查看节点状态，**可以查看最大事务id**

**echo ruok|nc 127.0.0.1 2181 查看某个服务点是否启动，若回复imok表示已经启动。**

**Echo conf|nc 127.0.0.1 2181 输出相关服务配置信息**

**echo dump| nc 127.0.0.1 2181 ,列出未经处理的会话和临时节点。**

**5. echo cons | nc 127.0.0.1 2181 ,列出所有连接到服务器的客户端的完全的连接 / 会话的详 细信息。**

**6. echo envi |nc 127.0.0.1 2181 ,输出关于服务环境的详细信息（区别于 conf 命令）。**

**echo reqs | nc 127.0.0.1 2181 ,列出未经处理的请求。**

**8. echo wchs | nc 127.0.0.1 2181 ,列出服务器 watch 的详细信息。**

**9. echo wchc | nc 127.0.0.1 2181 ,通过 session 列出服务器 watch 的详细信息，它的输出 是一个与 watch 相关的会话的列表。**

**10. echo wchp | nc 127.0.0.1 2181 ,通过路径列出服务器 watch 的详细信息。它输出一个与 session 相关的路径。**

# 4 应用场景

## 4.1 名字服务****(NameService)****

分布式应用中，通常需要一套完备的命令机制，既能产生唯一的标识，又方便人识别和记忆。 我们知道，每个ZNode都可以由其路径唯一标识，路径本身也比较简洁直观，另外ZNode上还可以存储少量数据，这些都是实现统一的NameService的基础。下面以在HDFS中实现NameService为例，来说明实现NameService的基本布骤:

* 目标：通过简单的名字来访问指定的HDFS机群
* 定义命名规则：这里要做到简洁易记忆。下面是一种可选的方案： [serviceScheme://][zkCluster]-[clusterName]，比如hdfs://lgprc-example/表示基于lgprc ZooKeeper集群的用来做example的HDFS集群
* 配置DNS映射: 将zkCluster的标识lgprc通过DNS解析到对应的ZooKeeper集群的地址
* 创建ZNode: 在对应的ZooKeeper上创建/NameService/hdfs/lgprc-example结点，将HDFS的配置文件存储于该结点下
* 用户程序要访问hdfs://lgprc-example/的HDFS集群，首先通过DNS找到lgprc的ZooKeeper机群的地址，然后在ZooKeeper的/NameService/hdfs/lgprc-example结点中读取到HDFS的配置，进而根据得到的配置，得到HDFS的实际访问入口

## 4.2 配置管理****(Configuration Management)****

在分布式系统中，常会遇到这样的场景: 某个Job的很多个实例在运行，它们在运行时大多数配置项是相同的，如果想要统一改某个配置，一个个实例去改，是比较低效，也是比较容易出错的方式。通过ZooKeeper可以很好的解决这样的问题，下面的基本的步骤：

* 将公共的配置内容放到ZooKeeper中某个ZNode上，比如/service/common-conf
* 所有的实例在启动时都会传入ZooKeeper集群的入口地址，并且在运行过程中Watch /service/common-conf这个ZNode
* 如果集群管理员修改了了common-conf，所有的实例都会被通知到，根据收到的通知更新自己的配置，并继续Watch /service/common-conf

## 4.3组员管理(Group Membership)

在典型的Master-Slave结构的分布式系统中，Master需要作为“总管”来管理所有的Slave, 当有Slave加入，或者有Slave宕机，Master都需要感知到这个事情，然后作出对应的调整，以便不影响整个集群对外提供服务。以**[Hbase](http://lib.csdn.net/base/hbase" \o "Hbase知识库" \t "http://blog.csdn.net/xinguan1267/article/details/_blank)**为例，HMaster管理了所有的RegionServer，当有新的RegionServer加入的时候，HMaster需要分配一些Region到该RegionServer上去，让其提供服务；当有RegionServer宕机时，HMaster需要将该RegionServer之前服务的Region都重新分配到当前正在提供服务的其它RegionServer上，以便不影响客户端的正常访问。下面是这种场景下使用ZooKeeper的基本步骤：

* Master在ZooKeeper上创建/service/slaves结点，并设置对该结点的Watcher
* 每个Slave在启动成功后，创建唯一标识自己的临时性(Ephemeral)结点/service/slaves/${slave\_id}，并将自己地址(ip/port)等相关信息写入该结点
* Master收到有新子结点加入的通知后，做相应的处理
* 如果有Slave宕机，由于它所对应的结点是临时性结点，在它的Session超时后，ZooKeeper会自动删除该结点
* Master收到有子结点消失的通知，做相应的处理

## 4.4 简单互斥锁****(Simple Lock)****

我们知识，在传统的应用程序中，线程、进程的同步，都可以通过**[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "http://blog.csdn.net/xinguan1267/article/details/_blank)**提供的机制来完成。但是在分布式系统中，多个进程之间的同步，操作系统层面就无能为力了。这时候就需要像ZooKeeper这样的分布式的协调(Coordination)服务来协助完成同步，下面是用ZooKeeper实现简单的互斥锁的步骤，这个可以和线程间同步的mutex做类比来理解：

* 多个进程尝试去在指定的目录下去创建一个临时性(Ephemeral)结点 /locks/my\_lock
* ZooKeeper能保证，只会有一个进程成功创建该结点，创建结点成功的进程就是抢到锁的进程，假设该进程为A
* 其它进程都对/locks/my\_lock进行Watch
* 当A进程不再需要锁，可以显式删除/locks/my\_lock释放锁；或者是A进程宕机后Session超时，ZooKeeper系统自动删除/locks/my\_lock结点释放锁。此时，其它进程就会收到ZooKeeper的通知，并尝试去创建/locks/my\_lock抢锁，如此循环反复

## 4.5 互斥锁****(Simple Lock without Herd Effect)****

上一节的例子中有一个问题，每次抢锁都会有大量的进程去竞争，会造成羊群效应(Herd Effect)，为了解决这个问题，我们可以通过下面的步骤来改进上述过程：

* 每个进程都在ZooKeeper上创建一个临时的顺序结点(Ephemeral Sequential) /locks/lock\_${seq}
* ${seq}最小的为当前的持锁者(${seq}是ZooKeeper生成的Sequenctial Number)
* 其它进程都对只watch比它次小的进程对应的结点，比如2 watch 1, 3 watch 2, 以此类推
* 当前持锁者释放锁后，比它次大的进程就会收到ZooKeeper的通知，它成为新的持锁者，如此循环反复

这里需要补充一点，通常在分布式系统中用ZooKeeper来做Leader Election(选主)就是通过上面的机制来实现的，这里的持锁者就是当前的“主”。

## 4.6 读写锁(Read/Write Lock)

我们知道，读写锁跟互斥锁相比不同的地方是，它分成了读和写两种模式，多个读可以并发执行，但写和读、写都互斥，不能同时执行行。利用ZooKeeper，在上面的基础上，稍做修改也可以实现传统的读写锁的语义，下面是基本的步骤:

* 每个进程都在ZooKeeper上创建一个临时的顺序结点(Ephemeral Sequential) /locks/lock\_${seq}

${seq}最小的一个或多个结点为当前的持锁者，多个是因为多个读可以并发

* 需要写锁的进程，Watch比它次小的进程对应的结点
* 需要读锁的进程，Watch比它小的最后一个写进程对应的结点
* 当前结点释放锁后，所有Watch该结点的进程都会被通知到，他们成为新的持锁者，如此循环反复

**每个节点只需要关注/shared\_lock节点下序号比自己小的那个节点是否存在就可以：**

1. 客户端调用create（）方法创建一个类似”/shared\_lock/[hostname]-请求序列号”的临时顺序节点。
2. 客户端调用getChildren（）接口来获取所有已经创建的子节点列表，不做任何watch（）。
3. 如果无法获得共享锁，就对比自己序号小的节点用exist（）来注册watcher

读请求：向比自己序号小的最后一个写请求注册watcher

写请求：向比自己序号小的最后一个请求注册watcher

4.等待watcher通知，继续进入步骤2

## 4.7 分布式队列

### 4.7.1 屏障

在分布式系统中，屏障是这样一种语义: 客户端需要等待多个进程完成各自的任务，然后才  
能继续往前进行下一步。下用是用ZooKeeper来实现屏障的基本步骤：

/queue\_barrier节点是一个已经存在的默认子节点，并且将其节点的数据内容设置为一个数字n=10代表默认的barrier的值，当/queue\_barrier下的子节点的数量达到10后，才会开启barrier，之后所有的客户端都会到/queue\_barrier节点下创建一个临时节点

1. 通过getData（）接口获取/queue\_barrier下的数据内容10.
2. 通过getChildren()获取节点下的所有子节点，即获取队列的所有的元素，同时注册对子节点列表变更的watcher监听
3. 统计子节点的数量
4. 如果子节点的个数还不足10个，那么就需要等待
5. 接受到watcher通知后，重复步骤2

### 4.7.2 FIFO先进先出

现实方式类似于共享锁（读写锁），

1. 通过调用getChildren()获得节点下的所有子节点
2. 确定自己在节点中的顺序
3. 如果自己不是顺序最小的节点，则用exist（）在比自己序号小的最后一个子节点上注册watch监听
4. 接受到watcher后，重复步骤一

## 4.8 双屏障

双屏障是这样一种语义: 它可以用来同步一个任务的开始和结束，当有足够多的进程进入屏障后，才开始执行任务；当所有的进程都执行完各自的任务后，屏障才撤销。下面是用ZooKeeper来实现双屏障的基本步骤：

进入屏障：

* Client Watch /barrier/ready结点, 通过判断该结点是否存在来决定是否启动任务
* 每个任务进程进入屏障时创建一个临时结点/barrier/process/${process\_id}，然后检查进入屏障的结点数是否达到指定的值，如果达到了指定的值，就创建一个/barrier/ready结点，否则继续等待
* Client收到/barrier/ready创建的通知，就启动任务执行过程

离开屏障：

* Client Watch /barrier/process，如果其没有子结点，就可以认为任务执行结束，可以离开屏障
* 每个任务进程执行任务结束后，都需要删除自己对应的结点/barrier/process/${process\_id}