ZooKeeper协调组件

# 服务器集群问题分析

当一个服务访问的人员逐步增加的时候，那么该服务器的性能一定会有所下降，当访问的并发人员越来越多的时候，那么该服务器的响应速度一定会慢。所以在实际的项目运行过程之中，那么往往会采用服务器集群的设计方案，这样可以提高访问性能，而且在某一台服务器宕机之后，用户可以继续正常访问。

可是这个情况下又会出现一种新的情况，现在假设说某一个WEB服务，需要进行若干个MySQL数据库的访问处理，那么此时对于这台WEB服务而言，最需要注意的就是所有MySQL服务主机的信息。

如果说现在中途有一台MySQL服务宕机了，那么现在就会出现一个问题，WEB端调用的服务列表的信息该如何修改？在实际的项目运行之中，WEB服务器也可能会有多台，那么以十台WEB服务器为例，这个时候如果某一台MySQL宕机了，那么如果将所有的服务器的列表信息以文本文件的方式保存在各个WEB服务器上，那么此时就会造成一个无法正常使用的困境。

为了解决这样的服务器列表更新的问题，那么这个时候往往会使用一些公共的资源—协调组件。

如果此时突然有一台主机宕机了，则在这个协调组件上进行服务器列表的动态更新处理，这样WEB端就可以避免调用已经宕机的数据库。

所以这种协调一致性组件最好的开源的就是Apache ZooKeeper组件。

# ZooKeeper简介

ZooKeeper组件官方下载站点：<http://zookeeper.apache.org>

ZooKeeper是一个开发源码的分布式应用程序协调服务，是Google的Chubby一个开源的实现，是Hadoop和Hbase的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

简单的说，所有主机的信息或你所需要的服务信息都依托ZooKeeper进行保存。

ZooKeeper数据保存：

* Zookeeper表现为一个分层的文件系统目录树结构（不同于文件系统的是，节点可以有自己的数据，而文件系统中的目录节点只有子节点）。
* 保存在ZooKeeper节点中的数据一般只有1M左右。

ZooKeeper严格来讲是根据节点的层级关系来进行数据保存处理的，在ZooKeeper之中最大的节点为根节点“/”，而在根节点下可以有无数个子节点和孙子节点。

在ZooKeeper进行数据保存的时候保存的都是关键信息，例如：服务器地址、端口号、接口名称，而且特别需要说明的是：最好不要使用zookeeper保存中文。

ZooKeeper特点：

* 顺序一致性：按照客户端发送请求的顺序更新数据；
* 原子性：更新要么成功，要么失败，不会出现部分更新；
* 单一性：无论客户端连接哪个server，都会看到同一个视图；
* 可靠性：一旦数据更新成功，将一直保持，直到新的更新；
* 及时性：客户端会在一个确定的时间内得到最新的数据。

在实际的开发过程之中，你可能有N台服务器，那么如果每一台服务器上都设置了统一的ZooKeeper服务，那么这些存储的所有数据都在ZooKeeper节点共享。

# Paxos算法

算法的本质：在一个信任的环境下，推选领导进行所有zookeeper节点操作。

Apache坚持宣称zookeeper不是Paxos算法的实现。很多专家也说zookeeper不是Paxos算法实现，但是从种种迹象来看，zookeeper跟Paxos算法描述几乎等同。

Paxos是莱斯利·兰伯特（Leslie Lanport，就是LaTeX中的“La”，此人现在在微软研究院）于1990年提出的一种基于消息传递的一致性算法：

* Paxos算法解决的问题是一个分布式系统如何就某个值（决议）达成一致。
* Google的Chubby与Apache的zookeeper都是基于此理论实现的；
* Paxos有一个前提：没有拜占庭将军问题。就是说Paxos只有在一个可信的计算环境中才能成立，这个环境是不会被入侵所破坏的。

拜占庭将军问题（Byzantine failures）

* 拜占庭将军问题，又称两军问题，是由莱斯利·兰伯特提出的点对点通信中的基本问题。含义是在存在消息丢失的不可靠信道上试图通过消息传递的方式达到一致性是不可能的。因此对一致性的研究一般假设信道是可靠的，或不存在本问题。
* 将军问题：
  + 拜占庭将军问题是一个协议问题，拜占庭帝国军队的将军们必须全体一致的决定是否攻击某一支敌军。问题是这些将军在地里上是分隔开来的，并且将军中存在叛徒，叛徒可以任意行动以达到以下目标：欺骗某些将军采取进攻行动；促成一个不是所有将军都同意的决定，如当将军们不希望进攻时促成进攻行动；或者魔火某些将军，使他们无法做出决定。如果叛徒达到了这些目的的之一，则任何攻击行动的结果都是注定要失败的，只有完全达成一致的努力才能获得胜利。
  + 拜占庭假设是对现实世界的模型化，由于硬件错误、网络拥塞或断开以及遭到恶意攻击，计算机和网络可能出现不可预料的行为。拜占庭容错协议必须处理这些失效，并且这些协议还要满足所要解决的问题要求的规范。这些算法通常以其弹性t作为特征，t表示算法可以应付的错误进程数。
  + 很多经典算法问题只有在t<n/3时才有解，如拜占庭将军问题，其中n是系统中进程的总数。

Paxos算法场景分析

* 有一个叫做Paxos的小岛（Island）上面住了一批居民，岛上面所有的事情由一些特殊的人决定，它们叫做议员（senator）。议员的总数（senator count）是确定的，不能更改。

Paxos算法场景分析----议员提议

* 岛上每次环境事务的变更都需要通过一个提议（proposal），每个提议都有一个编号（PID），这个编号是一直增长的，不能倒退。每个提议都需要超过半数“(senator count)/2 + 1”的议员同意才能生效。
* 每个议员只会同意大于当前编号的提议，包括已生效的和未生效的。如果议员收到小于等于当前编号的提议，他会拒绝，并告知对方：你的提议已经有人提过了。这里的当前编号是每个议员在自己记事本上面记录的编号，他不断更新这个编号。整个一会不能保证所有议员记事本上的编号总是相同的。现在一会有一个目标：保证所有的议员对于提议都能打成一致的看法。

Paxos算法与zookeeper

* 小岛（Island）—— ZK Server cluster；
* 议员（Senator）—— ZK Server；
* 提议（Proposal）—— ZNode Change（Create/Delete/SetData）；
* 提议编号（PID）—— Zxid（zookeeper transaction id）
* 正式法令 —— 所有Znode及其数据

Leader

* 小岛总统：ZK Server Leader，在所有议员中设立一个总统，只有总统有权发出提议，如果议员有自己的提议，必须发给总统并由总统啦提出。
* Leader的处理：
  + 居民甲（Client）到某个议员（ZK Server）那里询问（Get）某条法令的情况（Znode的数据），议员毫不犹豫的拿出他的记事本（local storage），查阅法令并告诉他结果，同时生命：我的数据不一定是最新的。你想要最新的数据？没问题，等着，等我找总统Sync以下再告诉你。
  + 居民乙（Client）到某个议员（ZK Server）那里要求政府归还欠他的一万元钱，议员让他在办公室等着，自己讲问题反映给了总统，总统询问所有议员你的意见，多数议员表示欠居民的钱一定要还，于是总统发表声明从国库中拿出一万元钱来还债，国库总资产由100万变成99万。居民乙拿到钱回去了（Client函数返回）。
  + 总统突然挂了，议员接二连三的发现联系不上总统，于是各自发表声明，推选新的总统，总统大选期间政府停业，拒绝居民的请求。

Paxos算法的核心本质：一定要有选举机制（多台服务），操作的环境一定是信任环境。

# ZooKeeper安装与配置

从正规的开发来讲，zookeeper运行环境一定是三台以上的服务器集群环境，但是这种运行模式是在实际的项目部署环节下才拥有的，可是如果要开发的时候往往只需要一台zookeeper就够了。

本次将按照非标准的配置模式，配置一台单节点的zookeeper主机。

zookeeper本身分为各种操作系统版本，但是考虑到实际的开发环境，所以本次将直接基于Linux版本应用。

1. 修改主机名称：vim /etc/hostname, 输入新的主机名称为“zk-server”，reboot重启；
2. 修改hosts配置文件：vim /etc/hosts,追加挡墙主机名称的配置：127.0.0.1 zk-server；
3. 解压zookeeper压缩包

|  |
| --- |
| tar –xzvf zookeeper-3.4.10.tar.gz –C /usr/local/ |

1. 为了方便进行zookeeper使用，建议将解压缩后的文件夹更名：

|  |
| --- |
| mv /usr/local/zookeeper-3.4.10 /usr/local/zookeeper |

1. 将zookeeper的可执行命令配置到系统环境属性之中：
   1. 打开环境配置文件：vim /etc/profile;
   2. 在配置文件之中输入如下内容；

|  |
| --- |
| export JAVA\_HOME=/usr/local/jdk1.8  export ZK\_HOME=/usr/local/zookeeper  export PATH=$PATH:$JAVA\_HOME/bin:$ZK\_HOME/bin: |

* 1. 保存退出后让配置立即生效：source /etc/profile；

1. 进入到zookeeper配置文件目录：cd /usr/local/zookeeper/conf，可以发现此目录之中存在有一个“zoo\_sample.cfg”配置文件，将此配置文件拷贝一份：

|  |
| --- |
| cp /usr/local/zookeeper/conf/zoo\_simple.cfg /usr/local/zookeeper/conf/zoo.cfg |

zookeeper使用的配置文件名称必须为“zoo.cfg”；

1. 启动zookeeper服务进程：/usr/local/zookeeper/bin/zkServer.sh start

|  |
| --- |
| ZooKeeper JMX enabled by default  Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg  Starting zookeeper ... STARTED |

1. zookeeper是一个java进程，所以可以使用“jps”命令查看所有的本机java的进程信息：QuorumPeerMain。
2. 查看zookeeper运行状态：/usr/local/zookeeper/bin/zkServer.sh status

|  |
| --- |
| ZooKeeper JMX enabled by default  Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg  Mode: standalone |

由于现在没有设置集群关系，所以zookeeper的运行状态为独立运行。

这种配置主要的目的是为了满足开发要求，但是在真实的线上运行环境中，一定是多主机集群处理。

# ZooKeeper集群部署

zookeeper是为了集群而生的，所以在实际的运行环境之中一定要以集群环境为主。本次预计准备使用三台主机，这三台主机的信息如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **主机名称** | **IP地址** | **描述** |
| 1 | zk-server-a | 192.168.127.134 | zookeeper |
| 2 | zk-server-b | 192.168.127.135 | zookeeper |
| 3 | zk-server-c | 192.168.127.136 | zookeeper |

1. 在实际的项目运行过程之中，所有的配置建议都以主机名称为主（IP也一样），所以为了保证三台主机的访问名称没有任何的问题，所以要首先统一hosts配置文件：
   1. 编辑主机的列表文件：vim /etc/hosts

|  |
| --- |
| 127.0.0.1 localhost  127.0.1.1 localhost  127.0.0.1 zk-server  192.168.127.134 zk-server-a  192.168.127.135 zk-server-b  192.168.127.136 zk-server-c |

1. 进行ssh的免登陆的配置处理
   1. 现在为了防止本机可能已经存在有ssh的免登陆配置，建议先进行删除处理：  
      rm –r ~/.ssh/；
   2. 生成一个新的ssh的key-get：ssh-keygen –t rsa;（接下的步骤一直回车）
   3. 进行本机授权控制：cat ~/.ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys;
   4. 进行其他主机的授权连接处理（互相配置）：

|  |  |
| --- | --- |
| 将本机的ssh公钥拷贝其它主机 | ssh-copy-id -i ~/.ssh/id\_rsa.pub zk-server-b |
| 将本机的ssh公钥拷贝其它主机 | ssh-copy-id -i ~/.ssh/id\_rsa.pub zk-server-c |

1. 将zookeeper开发包上传到系统之中，随后将其进行解压缩控制：

|  |
| --- |
| tar –xzvf zookeeper-3.4.10.tar.gz –C /usr/local/ |

1. 为了随后的方便配置，为文件夹更名；

|  |
| --- |
| mv /usr/local/zookeeper-3.4.10 /usr/local/zookeeper |

1. 修改profile配置文件：vim /etc/profile;

|  |
| --- |
| export JAVA\_HOME=/usr/local/jdk1.8  export ZK\_HOME=/usr/local/zookeeper  export PATH=$PATH:$JAVA\_HOME/bin:$ZK\_HOME/bin: |

修改后使配置立即生效：source /etc/profile

1. 拷贝zookeeper配置文件；

|  |
| --- |
| cp /usr/local/zookeeper/conf/zoo\_simple.cfg /usr/local/zookeeper/conf/zoo.cfg |

1. 建立一个zookeeper的数据保存目录：mkdir –p /usr/data/zookeeper
2. 修改zoo.cfg配置文件：vim /usr/local/zookeeper/conf/zoo.cfg

|  |  |
| --- | --- |
| 修改数据的文件目录 | dataDir=/usr/data/zookeeper |
| 增加主机信息列表 | server.1=zk-server-a:2888:3888  server.2=zk-server-b:2888:3888  server.3=zk-server-c:2888:3888 |

1. 在zookeeper之中所有主机编号都爱工作目录中以“myid”的文件形式出现。

|  |
| --- |
| echo 1 >> /usr/data/zookeeper/myid |

如果是“zk-server-b”主机myid就是2，“zk-server-c”主机myid就是3。

1. 将已经配置好的zookeeper目录拷贝到其它主机上：

|  |  |
| --- | --- |
| 拷贝到“zk-server-b”主机 | scp –r /usr/local/zookeeper/ zk-server-b:/usr/local |
| 拷贝到“zk-server-c”主机 | scp –r /usr/local/zookeeper/ zk-server-c:/usr/local |

1. 启动zookeeper服务：/usr/local/zookeeper/bin/zkServer.sh start;

启动之后等一段时间之后可以查看各个zookeeper节点状态：/usr/local/zookeeper/bin/zkServer.sh status;

|  |  |
| --- | --- |
| ZooKeeper JMX enabled by default  Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg  Mode: leader | ZooKeeper JMX enabled by default  Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg  Mode: follower |

这时将会发现，三台主机中，只有一台的状态显示为：“Mode:leader”，其余两台显示为“Mode:follower”。

如果此时有一台主机的节点关闭了，那么剩下的主机会重新进行选举处理，选举出新的“leader”主机。

zookeeper允许单独使用，但是在实际的项目运行环境下一定是集群配置。

# ZooKeeper客户端操作

zookeeper是一个节点树形关系，在zookeeper里面最为核心的节点是“/”根节点，但是如何进行节点的操作呢？在zookeeper之中提供有专属的操作命令：/usr/local/zookeeper/bin/zkCli.sh；

1. 在zk-server-a主机登录zk-server-b主机；

|  |
| --- |
| /usr/local/zookeeper/bin/zkCli.sh –server zk-server-b |

1. 如果要想知道所有的可用命令，可以使用“help”完成：

|  |
| --- |
| [zk: zk-server-b(CONNECTED) 0] help  ZooKeeper -server host:port cmd args  stat path [watch]  set path data [version]  ls path [watch]  delquota [-n|-b] path  ls2 path [watch]  setAcl path acl  setquota -n|-b val path  history  redo cmdno  printwatches on|off  delete path [version]  sync path  listquota path  rmr path  get path [watch]  create [-s] [-e] path data acl  addauth scheme auth  quit  getAcl path  close  connect host:port |

1. 列出根目录下的所有节点信息：“ls /”，此时会发现存在有一个“/zookeeper”的第二层节点。该节点主要描述的是各个zookeeper连接信息。
2. 创建一个新的节点：
   1. 失败：只是创建节点，但是不设置数据：create /mohist;
   2. 成功：创建节点，同时设置节点数据：create /mohist hello。
3. 查看“/mohist”的数据信息：get /mohist

|  |
| --- |
| hello  cZxid = 0x200000004  ctime = Sat Mar 31 23:17:55 PDT 2018  mZxid = 0x200000004  mtime = Sat Mar 31 23:17:55 PDT 2018  pZxid = 0x200000004  cversion = 0  dataVersion = 0  aclVersion = 0  ephemeralOwner = 0x0  dataLength = 5  numChildren = 0 |

1. 创建多层节点：/a/b/c=hello（create /a/b/c hello）

|  |
| --- |
| Node does not exist: /a/b/c |

此时明确的告诉了用户对不起你创建不了，zookeeper有一个设计缺陷：不允许多级创建，那么就意味着，只能一层一层的创建。

1. 删除节点信息：delete /a/b/c，在进行节点删除的时候依然不允许多级删除，只能够一级一级的删除操作。
2. 对于此时的数据实际上还可以进行监听控制；
   1. 对“/mohist”节点进行监听控制：get /mohist watch
   2. 随意启动另外任意一台主机的zookeeper客户端进行登录；
   3. 修改“/mohist”的数据：set /mohist world;

|  |
| --- |
| WatchedEvent state:SyncConnected type:NodeDataChanged path:/mohist |

也就是说一旦启用了监听的状态，其它主机对此节点的数据修改将会立即发出更新状态。

千万要记住一点：虽然是多台主机，但是所有的数据更改几乎瞬间完成的。

总结：

1. zookeeper中的节点必须一层层创建或者一层层删除；
2. zookeeper进行处理的时候可以设置节点的监听，一旦修改了节点数据，则监听将立刻可以监听到变化。所有的监听默认只允许监听一次。

# 使用Java操作ZooKeeper

在之前使用了zookeeper提供的客户端进行了节点以及数据操作，但是实际的开发之中，zookeeper会作为所有服务器的协同组件，所以通过程序访问zookeeper才是关键性的因素。

如果要想通过程序进行zookeeper控制，首先要为zookeeper开发包做一个解压缩。因为此时zookeeper的开发包里面提供有lib。

## 连接zookeeper服务

首先一定要确认好，zookeeper是代表着新一代的组件，新旧组件的最明显区别：就是在于集群支持上。所以如果要想进行zookeeper的连接处理，只需要将所有使用到的zookeeper地址全部填写上即可，而且如果中间有一个弟子服务器无法使用，那么也不影响程序的使用。

如果要想进行zookeeper连接处理，那么首先一定要使用一个核心类：org.apache.zookeeper.ZooKeeper,这个类包含的方法：

* 构造方法：public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher) throws IOException
  + connectString：所有要进行连接的zookeeper的地址信息，默认端口都是2181；
  + sessionTimeout：连接的超时时间（毫秒）；
  + Watcher：进行监听器的配置。

当取得了zookeeper的连接对象之后就可以利用此对象取得所有的节点信息：

* 方法：public List<String> getChildren(String path, boolean watch) throws KeeperException,InterruptedException;

范例：实现zookeeper连接

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo; import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper; import java.util.List; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 16:27  \*/* public class ConnectionZooKeeperDemo {  public static void main(String[] args) throws Exception {  String connectString = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  int sessionTimeout = 2000;  ZooKeeper zkClient = new ZooKeeper(connectString, sessionTimeout, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  }  });  List<String> children = zkClient.getChildren("/", false);  for (String str : children) {  System.*out*.println(str);  }  zkClient.close();  } } |

## ACL授权控制

现在的zookeeper就好比一个赤裸的孩子一样，谁都可以看，谁都可以操作，很明显，这样的处理模式于安全性是不相符的（实际上这个安全性意义不大，因为zookeeper只是保存一些基础信息，而且这些信息有可能都不会以明文的形式出现）。所以可以使用一种ACL（访问控制列表）来进行控制，但是如果要想进行zookeeper控制，首先就必须清楚再zookeeper中提供的认证与授权处理机制。

ZooKeeper权限与认证

* ZooKeeper节点操作权限：
  + CREATE、READ、WRITE、DELETE、ADMIN也就是增、删、改、查、管理权限，这5种权限简写为crwda；
* 身份的认证方式：
  + world：默认方式，相当于全世界都能访问；
  + auth：代表已经认证通过的用户（cli中可以通过addauth digest user:pwd来添加当前上下文中的授权用户）；
  + digest：即用户名:密码这种方式认证，这也是业务系统中最常用的；
  + ip：使用IP地址认证。

本次就使用digest：即用户名:密码这种方式认证，这也是业务系统中最常用的来进行认证处理。

1. 使用zookeeper的客户端工具登录zookeeper；

|  |
| --- |
| /usr/local/zookeeper/bin/zkCli.sh -server zk-server-c |

1. 进行授权控制的前提是需要有一个用户：zkuser / 650901;

|  |
| --- |
| addauth digest zkuser:650901 |

1. 进行授权的操作控制；

|  |
| --- |
| setAcl / auth:zkuser:650901:cdrwa |

为了检测当前的操作是否正确，建议使用另外的一个客户端进行zookeeper连接，连接的时候是无法进行授权配置的，但是一旦执行了某些操作，例如：列出根节点下的所有子节点（ls /）会出现如下错误提示：

|  |
| --- |
| Authentication is not valid : / |

那么此时就需要进行重新的认证控制，输入：addauth digest zkuser:650901

1. Java程序上如果需要进行认证处理则观察ZooKeeper类中提供的方法；
   1. 认证设置：public void addAuthInfo(String scheme, byte[] auth);

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo; import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper; import java.util.List; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 16:27  \*/* public class ConnectionZooKeeperDemo {  public static void main(String[] args) throws Exception {  String connectString = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  int sessionTimeout = 2000;  ZooKeeper zkClient = new ZooKeeper(connectString, sessionTimeout, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  }  });  zkClient.addAuthInfo("digest", "zkuser:650901".getBytes());  List<String> children = zkClient.getChildren("/", false);  for (String str : children) {  System.*out*.println(str);  }  zkClient.close();  } } |

这种授权的操作意义一般不会用在根节点，如果要使用，也是一些子节点上进行ACL控制。例如：A服务需要通过zookeeper配置集群信息，这个时候A服务怕别人进行破坏可能会使用ACL，但是这种情况下并不多。

## 创建Znode

当已经成功的链接了zookeeper服务之后，那么下面就需要进行节点的操作。在给定的命令之中，如果要进行节点的处理基本上就是CREATE、DELETE。那么对于这些命令zookeeper的类中也都有与之对应的处理方法：

* 节点创建：public String create(String path, byte[] data, List<ACL> acl, CreateMode createMode) throws KeeperException, InterruptedException
  + “String path”：进行创建的节点名称，但是zookeeper中的节点不允许多层创建；
  + “byte[] data”：创建节点的时候可以设置节点对应的内容，对于此处的内容使用字节数组，别搞中文；
  + “List<ACL> acl”：进行安全的认证，所以可以直接使用字节已有的模式完成；
    - 所有的节点的认证的控制信息都保存在了“org.apache.zookeeper.ZooDefs.lds”，所有人都可以访问：static final ArrayList<ACL> OPEN\_ACL\_UNSAFE；
  + “CreateMode createMode”：节点的创建模式，使用“org.apache.zookeeper.CreateMode”枚举类；
    - public static final CreateMode EPHEMERAL：创建一个瞬时节点；
    - public static final CreateMode EPHEMERAL\_SEQUENTIAL：创建一个瞬时的序列节点；
    - public static final CreateModel PERSISTENT：创建一个持久化节点；
    - public static final CreateModel PERSISTENT\_SEQUENTIAL：创建一个持久化的序列节点。
* 判断节点是否存在：public Stat exists(String path, boolean watch) throws KeeperException, InterrutedException；

范例：创建一个普通持久化节点

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo; import org.apache.zookeeper.\*; import java.util.List; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 16:27  \*/* public class ConnectionZooKeeperDemo2 {  public static final String *CONNECTS* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  public static final int *SESSION\_TIMEOUT* = 2000;  public static final String *SCHEME* = "digest";  public static final String *AUTH\_INFO* = "zkuser:650901";  public static final String *GROUP\_NODE* = "/mohist";  public static final String *CHILDREN\_NODE* = *GROUP\_NODE* + "/bigdata";  public static void main(String[] args) throws Exception {  ZooKeeper zkClient = new ZooKeeper(*CONNECTS*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  }  });  zkClient.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  if (zkClient.exists(*GROUP\_NODE*, false) == null) {  zkClient.create(*GROUP\_NODE*, "HelloData".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  }  if (zkClient.exists(*CHILDREN\_NODE*, false) == null) {  zkClient.create(*CHILDREN\_NODE*, "HelloChildren".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  }  zkClient.close();  } } |

持久化节点指的是在ZooKeeper客户端断开连接之后依然可以保存下来的信息，那么下面再来观察临时节点。

范例：创建临时节点

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo; import org.apache.zookeeper.\*; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 16:27  \*/* public class ConnectionZooKeeperDemo3 {  public static final String *CONNECTS* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  public static final int *SESSION\_TIMEOUT* = 2000;  public static final String *SCHEME* = "digest";  public static final String *AUTH\_INFO* = "zkuser:650901";  public static final String *GROUP\_NODE* = "/mohist";  public static final String *CHILDREN\_NODE* = *GROUP\_NODE* + "/cluster";  public static void main(String[] args) throws Exception {  ZooKeeper zkClient = new ZooKeeper(*CONNECTS*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  }  });  zkClient.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  if (zkClient.exists(*GROUP\_NODE*, false) == null) {  zkClient.create(*GROUP\_NODE*, "HelloData".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  }  if (zkClient.exists(*CHILDREN\_NODE*, false) == null) {  String createPath = zkClient.create(*CHILDREN\_NODE*, "HelloChildren".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*EPHEMERAL*);  System.*out*.println("新的节点路径" + createPath);  }  System.*out*.print("按回车结束本次运行：");  System.*in*.read();  zkClient.close();  } } |

通过观察，当程序还未结束执行前，创建的临时节点还存在。当程序运行结束后，所创建的临时节点将消失。

范例：序列化临时节点

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo; import org.apache.zookeeper.\*; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 16:27  \*/* public class ConnectionZooKeeperDemo4 {  public static final String *CONNECTS* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  public static final int *SESSION\_TIMEOUT* = 2000;  public static final String *SCHEME* = "digest";  public static final String *AUTH\_INFO* = "zkuser:650901";  public static final String *GROUP\_NODE* = "/mohist";  public static final String *CHILDREN\_NODE* = *GROUP\_NODE* + "/cluster-";  public static void main(String[] args) throws Exception {  ZooKeeper zkClient = new ZooKeeper(*CONNECTS*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  }  });  zkClient.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  if (zkClient.exists(*GROUP\_NODE*, false) == null) {  zkClient.create(*GROUP\_NODE*, "HelloData".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  }  for (int i = 0; i < 10; i ++) {  if (zkClient.exists(*CHILDREN\_NODE*, false) == null) {  String createPath = zkClient.create(*CHILDREN\_NODE*, "HelloChildren".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*EPHEMERAL\_SEQUENTIAL*);  System.*out*.println("新的节点路径" + createPath);  }  }  System.*out*.print("按回车结束本次运行：");  System.*in*.read();  zkClient.close();  } } |

之所以zookeeper很重要的主要因素就是它对于临时节点的处理支持。

## 节点数据操作

节点的数据处理本质上就是取得、设置（修改）、删除等基本的处理操作，那么这些操作的方法都在zookeeper程序类中定义；

1. 取得节点中的数据内容：
   1. 取得方法：public byte[] getData(String path, boolean watch, Stat stat) throws KeeperException, InterruptedException;
      1. “String path”：要读取的路径；
      2. “boolean watch”：是否需要进行监听；
      3. “Stat stat”：取得你的统计消息；

范例：取得节点数据

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo; import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper; import org.apache.zookeeper.data.Stat; import java.util.List; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 16:27  \*/* public class ConnectionZooKeeperDemo {  public static void main(String[] args) throws Exception {  String connectString = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  int sessionTimeout = 2000;  ZooKeeper zkClient = new ZooKeeper(connectString, sessionTimeout, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  }  });  zkClient.addAuthInfo("digest", "zkuser:650901".getBytes());  if (zkClient.exists("/mohist", false) != null) {  Stat stat = new Stat();  String data = new String(zkClient.getData("/mohist", false, stat));  System.*out*.println(data);  }  zkClient.close();  } } |

1. 修改（设置）路径数据：
   1. 方法：public Stat setData(String path, byte[] data, int version) throws KeeperException, InterruptedException;

范例：设置数据

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo; import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper; import org.apache.zookeeper.data.Stat; import java.util.List; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 16:27  \*/* public class ConnectionZooKeeperDemo {  public static void main(String[] args) throws Exception {  String connectString = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  int sessionTimeout = 2000;  ZooKeeper zkClient = new ZooKeeper(connectString, sessionTimeout, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  }  });  zkClient.addAuthInfo("digest", "zkuser:650901".getBytes());  if (zkClient.exists("/mohist", false) != null) {  zkClient.setData("/mohist", "HELLO WORLD".getBytes(), -1);  }  zkClient.close();  } } |

1. 删除节点数据：
   1. 方法：public void delete(String path, int version) throws KeeperException, InterruptedException;

范例：删除节点数据

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo;  import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper; import org.apache.zookeeper.data.Stat;  import java.util.List;  */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 16:27  \*/* public class ConnectionZooKeeperDemo {  public static void main(String[] args) throws Exception {  String connectString = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  int sessionTimeout = 2000;  ZooKeeper zkClient = new ZooKeeper(connectString, sessionTimeout, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  }  });  zkClient.addAuthInfo("digest", "zkuser:650901".getBytes());  if (zkClient.exists("/bigdata", false) != null) {  zkClient.delete("/bigdata", -1);  }  zkClient.close();  } } |

在zookeeper里面对于删除是删除路径信息，而不是简单的删除路径中的数据。

## 数据监听

数据监听是整个zookeeper之中最为重要的核心组件，因为每一次节点数据的改变都会引起监听器的触发，那么用户也需要根据这种触发的形式来进行及时的响应处理。

范例：单个节点的监听

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo;  import org.apache.zookeeper.\*;  */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 17:51  \*/* public class WatcherZNodeDemo1 {  public static final String *CONNECTS* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  public static final int *SESSION\_TIMEOUT* = 2000;  public static final String *SCHEME* = "digest";  public static final String *AUTH\_INFO* = "zkuser:650901";  public static final String *GROUP\_NODE* = "/mohist";  private static ZooKeeper *zkClient* = null;  public static void main(String[] args) throws Exception {  *zkClient* = new ZooKeeper(*CONNECTS*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  try {  *zkClient*.exists(*GROUP\_NODE*, true);  } catch (Exception e) {  e.printStackTrace();  }  }  });  *zkClient*.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  *zkClient*.exists(*GROUP\_NODE*, true);  System.*in*.read();  *zkClient*.close();  } } |
| [监听事件处理] path = null、type = None、state = SyncConnected  [监听事件处理] path = /mohist、type = NodeDataChanged、state = SyncConnected  [监听事件处理] path = /mohist、type = NodeDataChanged、state = SyncConnected  [监听事件处理] path = /mohist、type = NodeDeleted、state = SyncConnected  [监听事件处理] path = /mohist、type = NodeCreated、state = SyncConnected |

此时应该发现在节点监听处理时间类“WatchedEvent”类中可以取得操作的事件类型，该类型是一个枚举类（org.apache.zookeeper.Watcher.Event.EventType），该枚举类中存在有如下状态：

* 子节点改变：NodeChildrenChanged；
* 节点创建：NodeCreated；
* 节点数据变更：NodeDataChanged；
* 节点删除：NodeDeleted；
* 没有操作：None（刚刚打开zookeeper连接，所有的节点不可能有任何操作的）。

但是非常遗憾的是，以上的监听程序只能够对一个节点监听，无法对子节点进行监听，因为使用的方法有错误。应该使用getChidren()方法才可以实现子节点的监听；

范例：对所有子节点监听

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.demo; import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 17:51  \*/* public class WatcherZNodeDemo2 {  public static final String *CONNECTS* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  public static final int *SESSION\_TIMEOUT* = 2000;  public static final String *SCHEME* = "digest";  public static final String *AUTH\_INFO* = "zkuser:650901";  public static final String *GROUP\_NODE* = "/mohist";  private static ZooKeeper *zkClient* = null;  public static void main(String[] args) throws Exception {  *zkClient* = new ZooKeeper(*CONNECTS*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent event) {  System.*out*.println("[监听事件处理] path = " + event.getPath() + "、type = " + event.getType() + "、state = " + event.getState());  try {  *zkClient*.getChildren(*GROUP\_NODE*, true);  } catch (Exception e) {  e.printStackTrace();  }  }  });  *zkClient*.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  *zkClient*.getChildren(*GROUP\_NODE*, true);  System.*in*.read();  *zkClient*.close();  } } |
| [监听事件处理] path = null、type = None、state = SyncConnected  [监听事件处理] path = /mohist、type = NodeChildrenChanged、state = SyncConnected  [监听事件处理] path = /mohist、type = NodeChildrenChanged、state = SyncConnected |

对于子节点的监听无法进行数据的监听操作，只能对于子节点的创建或删除进行监听。

# 综合案例：服务器动态列表

在项目集群环境开发之中一定会有多台服务器出现，那么这每一台的服务器的信息我们都希望可以进行动态的监控，也就是说如果现在假设有三台服务器：server-a、server-b、server-c，希望知道每台主机是否存在，所以这种情况下就可以使用zookeeper来进行本功能的实现。

## 服务器动态列表节本实现思路

如果要想进行多台服务器的状态的监听（服务器本身，而不是对服务器中的进程）。

经过ZooKeeper使用分析你可以发现zookeeper特点：在进行zookeeper操作的时候是不允许进行子节点数据的监听的，但是可以监听子节点变化，所以建议将所有服务器的节点信息都统一保存在“/mohist-server”节点之中，而后所有服务器创建的节点都属于瞬时节点，按照如下的方式排列：

* 监控节点：/mohist-server
  + /mohist-server/server-0000000;
  + /mohist-server/server-0000001;
  + /mohist-server/server-0000002;

要想实现本类程序，一定要每一个服务器上都去部署一个zookeeper的运行项目（需要有一个zookeeper的进程存在）。

## 开发服务器端程序

服务器端的程序主要是指部署在各个服务主机上的运行Java程序，该程序的主要功能是向zookeeper里面发送临时的节点数据。

范例：建立服务器端程序

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.listener; import org.apache.zookeeper.\*; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 18:25  \*/* public class ServerListener {  */\*\*  \* zookeeper连接地址  \*/* public static final String *CONNECTION\_URL* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";    */\*\*  \* 连接超时时间  \*/* public static final int *SESSION\_TIMEOUT* = 3000;   */\*\*  \* zookeeper授权方式  \*/* public static final String *SCHEME* = "digest";    */\*\*  \* zookeeper授权信息  \*/* public static final String *AUTH\_INFO* = "zkuser:650901";   */\*\*  \* 父节点  \*/* public static final String *GROUP\_NODE* = "/mohist-server";   */\*\*  \* 子节点  \*/* public static final String *CHILDREN\_NODE* = *GROUP\_NODE* + "/server-";   */\*\*  \* zookeeper客户端处理对象  \*/* private ZooKeeper zkClient;    public ServerListener(String serverName) throws Exception {  // zookeeper连接控制，同时进行各个节点的而创建  this.connectZooKeeperServer(serverName);  // 连接准备好之后，要自动进行制定任务的调用  this.handle();  }   */\*\*  \* 定义真正要实现的一些具体的操作业务代码。  \* 例如：在服务器启动的时候又可能要通过某些系统记录出服务器的启动次数  \** ***@throws*** *Exception  \*/* public void handle() throws Exception {  System.*out*.print("[按回车结束]");  System.*in*.read();  this.zkClient.close();  }    */\*\*  \* zookeeper连接处理  \** ***@param*** *serverName 服务器名称  \** ***@throws*** *Exception Exception  \*/* public void connectZooKeeperServer(String serverName) throws Exception {  this.zkClient = new ZooKeeper(*CONNECTION\_URL*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent watchedEvent) {  }  });  this.zkClient.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  if (this.zkClient.exists(*GROUP\_NODE*, false) == null) {  this.zkClient.create(*GROUP\_NODE*, "SERVER-LIST".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  }  if (this.zkClient.exists(*CHILDREN\_NODE*, false) == null) {  this.zkClient.create(*CHILDREN\_NODE*, serverName.getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*EPHEMERAL\_SEQUENTIAL*);  }  } } |

最终本程序一定要放在服务器上执行，所以需要一个启动的处理方法。

范例：建立StartServerMain程序类负责启动程序

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.main; import com.kuhnwei.mohist.examples.zookeeper.listener.ServerListener; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 18:40  \*/* public class StartServerMain {  public static void main(String[] args) throws Exception {  if (args.length != 1) {  // 程序执行时没有传递参数，也就是服务主机名称  System.*err*.println("错误的执行程序，应该在执行的时候输入服务器名称。");  // 系统退出  System.*exit*(1);  }  new ServerListener(args[0]);  } } |

那么此时如果该Java进程运行着，则表示此服务器还活着（会发送心跳），则临时节点是不会被删除的，而如果没有心跳（宕机、网络阻塞、机房故障），那么则心跳消失，消失之后节点也会响应的自动销毁。

## 服务器程序部署

1. 所有的服务器的监听程序最终一定要部署到服务器上，而后才可以进行响应的进程启动，那么在这样的情况下，最简单的做法是将此项目打包为一个可执行的jar文件。
2. 为了方便测试建议先在windows本地完成一次执行：java -jar zkserver.jar win-server;
3. 将此时的“zkserver.jar”分别上传到指定的服务器主机中；
4. 在各服务器主机中直接执行jar文件：java -jar zkserver.jar server-a;

## 开发客户端程序

客户端的主要功能就是通过zookeeper取得所有服务器的信息，也就是说它只需要考虑取得“/mohist-server”节点下所有子节点的内容就可以了，而后所有的节点的变化都应该进行监控。

范例：实现客户端监听

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.client; import org.apache.zookeeper.\*; import org.apache.zookeeper.data.Stat; import java.util.LinkedHashSet; import java.util.List; import java.util.Set; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 19:25  \*/* public class ServerListClientListener {  */\*\*  \* 进行计数统计  \*/* public static int *count* = 0;  */\*\*  \* zookeeper连接地址  \*/* public static final String *CONNECTION\_URL* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  */\*\*  \* 连接超时时间  \*/* public static final int *SESSION\_TIMEOUT* = 3000;   */\*\*  \* zookeeper授权方式  \*/* public static final String *SCHEME* = "digest";   */\*\*  \* zookeeper授权信息  \*/* public static final String *AUTH\_INFO* = "zkuser:650901";   */\*\*  \* 父节点  \*/* public static final String *GROUP\_NODE* = "/mohist-server";   */\*\*  \* zookeeper客户端处理对象  \*/* private ZooKeeper zkClient;   public ServerListClientListener() throws Exception {  this.connectZooKeeperServer();  System.*out*.println("[第 " + (*count* ++) + " 次取得服务列表] " + ServerListClientListener.this.updateServerList());  }   */\*\*  \* 编写一个方法实现所有的服务端信息列表的更新获得  \** ***@return*** *最新的服务列表信息集合  \** ***@throws*** *Exception  \*/* public Set<String> updateServerList() throws Exception {  Set<String> allServers = new LinkedHashSet<>();  List<String> children = this.zkClient.getChildren(*GROUP\_NODE*, true);  for (String c : children) {  String path = *GROUP\_NODE* + "/" + c;  allServers.add(new String(this.zkClient.getData(path, false, new Stat())));  }  return allServers;  }   */\*\*  \* zookeeper连接处理  \** ***@throws*** *Exception Exception  \*/* public void connectZooKeeperServer() throws Exception {  this.zkClient = new ZooKeeper(*CONNECTION\_URL*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent event) {  if (event.getPath() != null) {  if (event.getType() == Event.EventType.*NodeChildrenChanged*) {  try {  System.*out*.println("[第 " + (*count* ++) + " 次取得服务列表] " + ServerListClientListener.this.updateServerList());  } catch (Exception e) {  e.printStackTrace();  }  }  }  }  });  this.zkClient.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  if (this.zkClient.exists(*GROUP\_NODE*, false) == null) {  this.zkClient.create(*GROUP\_NODE*, "SERVER-LIST".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  }  }   public void close() throws InterruptedException {  this.zkClient.close();  } } |

范例：启动客户端

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.client; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 19:34  \*/* public class TestClient {  public static void main(String[] args) throws Exception {  ServerListClientListener client = new ServerListClientListener();  System.*in*.read();  client.close();  } } |
| [第 0 次取得服务列表] [zk-server-b, zk-server-c]  [第 1 次取得服务列表] [zk-server-b, zk-server-a, zk-server-c]  [第 2 次取得服务列表] [zk-server-a, zk-server-c]  [第 3 次取得服务列表] [zk-server-a, zk-server-c, zk-server-b] |

这些列表在进行处理的时候都是向同一个存储空间进行信息的保存。

# 综合案例：分布式锁

所谓的分布式指的是需要两台以上的主机（多个线程）进行同一资源操作的时候才有可能需要使用到分布式锁。Java本身也是提供有锁的机制的，但是这种锁指的是针对于某一台单独的主机来实现的，所以它不属于分布式的锁。

为什么在开发之中需要有分布式锁呢？例如：在现实生活之中，机场很忙碌，因为几乎每时每分都会有飞机降落，那么在这样的情况下就会出现一种情况了：现在有十架飞机同时请求降落。

机场现在只有一个，那么所有的飞机如果要想进行降落只能够依次完成降落任务，所以这个时候就需要进行队列的排序。

当多台主机要访问同一资源的时候那么久需要使用分布式锁的概念来完成。这样可以保证所有的主机按照指定的顺序执行操作。很明显，如果以服务器的集群为例，那么可以实现分布式锁的必定就是协调一致组件（ZooKeeper），那么对于这些主机的访问信息可以使用临时的序列化节点来控制完成。

## CountDownLatch同步类

这个类是JDK1.5之后引入的一个线程同步的处理类，利用这个类可以轻松的搞定线程的执行顺序问题。

范例：观察一下程序的问题

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.lock; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 20:46  \*/* public class TestCountDownLatch {  public static void main(String[] args) {  MyThread mt = new MyThread();  new Thread(mt).start();  new Thread(mt).start();  System.*out*.println("[程序执行完毕]");  } } class MyThread implements Runnable {  @Override  public void run() {  System.*out*.println(Thread.*currentThread*().getName() + "执行。");  } } |
| [程序执行完毕]  Thread-1执行。  Thread-0执行。 |

此时的程序发现实际上并没有取得我们所希望的同步处理，因为主方法本身也属于一个线程，而且多个线程严格来讲是属于并行启动的状态，所以此时如果要想控制好当前的问题，就需要通过CountDownLatch类来完成。

范例：解决同步问题

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.lock;  import java.util.concurrent.CountDownLatch; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 20:46  \*/* public class TestCountDownLatch {  public static void main(String[] args) throws InterruptedException {  // 表示当前启动的线程数量  CountDownLatch latch = new CountDownLatch(2);  // 将CountDownLatch对象传递给MyThread类  MyThread mt = new MyThread(latch);  new Thread(mt).start();  new Thread(mt).start();  // 等待  latch.await();  System.*out*.println("[程序执行完毕]");  } } class MyThread implements Runnable {  private CountDownLatch latch;  public MyThread(CountDownLatch latch) {  this.latch = latch;  }  @Override  public void run() {  System.*out*.println(Thread.*currentThread*().getName() + "执行。");  // 向下减  latch.countDown();  } } |
| Thread-1执行。  Thread-0执行。  [程序执行完毕] |

CountDownLatch可以简单理解为是一个计数的操作处理。你可以把CountDownLatch理解为一个点名后依次报完名了，在执行后续安排的处理模式。

并发工具：CountDownLatch

* CountDownLatch这个类能够使一个线程等待其他线程完成各自的工作后再执行。例如，应用程序的主线程希望在负责启动框架服务的线程已经启动所有的框架服务之后再执行。
* CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。每当一个线程完成了字节的任务后，计数器的值就会减1。当计数器值达到0时它表示所有的线程已经完成了任务，然后在闭锁上等待的线程就可以恢复执行任务。

在CountDownLatch类中有如下方法：

* 构造方法：public CountDownLatch(int count);
* 等待其它线程执行完毕后再向下执行：public void await() throws InterruptedException;
* 减少线程执行量：public void countDown()，计数到0之后将恢复等待执行的线程。

## 多线程访问节点创建

如果要想实现分布式锁，那么一定需要将若干个线程的信息以zookeeper节点的形式存储，那么本次假设要保存的根节点名称为“mohist-lock”。

范例：分布式锁的基本模型

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.lock;  import org.apache.zookeeper.\*;  import java.util.concurrent.CountDownLatch;  */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 21:03  \*/* public class DistributedLock {  public static final String *CONNECTION\_URL* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  public static final int *SESSION\_TIMEOUT* = 3000;  public static final String *SCHEME* = "digest";  public static final String *AUTH\_INFO* = "zkuser:650901";  public static final String *GROUP\_NODE* = "/mohist-lock";  public static final String *CHILDREN\_NODE* = *GROUP\_NODE* + "/lockthread-";  private CountDownLatch latch;  private ZooKeeper zkClient;  private String selfPath; // 保存每次创建的临时节点的路径信息  private int threadId = 0;   public DistributedLock(int threadId, CountDownLatch latch) throws Exception {  this.threadId = threadId;  this.latch = latch;  this.connectZooKeeperServer();  }   public void handle() throws Exception {  this.handlCallback(); // 执行具体的业务操作  this.releaseZooKeeper(); // 释放掉当前的zookeeper连接信息  this.latch.countDown(); // 进行减减的操作  }   public void handlCallback() {  // 取得分布式锁之后的目的是要进行具体的业务操作  System.*out*.println("[Thread - " + this.threadId + "] 获得操作权，执行具体的业务处理操作。");  }   public void createChildrenNode() throws Exception {  this.selfPath = this.zkClient.create(*CHILDREN\_NODE*, ("Thread-" + this.threadId).getBytes(),  ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*EPHEMERAL\_SEQUENTIAL*);  System.*out*.println("[Thread-"+this.threadId+"、创建新的临时节点] " + this.selfPath);  }   public void connectZooKeeperServer() throws Exception {  this.zkClient = new ZooKeeper(*CONNECTION\_URL*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent watchedEvent) {  }  });  this.zkClient.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  if (this.zkClient.exists(*GROUP\_NODE*, false) == null) {  this.zkClient.create(*GROUP\_NODE*, "LOCKDEMO".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  }  }   public void releaseZooKeeper() {  if (this.zkClient != null) {  try {  this.zkClient.close();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

范例：线程启动测试类

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.lock;  import java.util.concurrent.CountDownLatch;  */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 21:15  \*/* public class LockThread implements Runnable {  private DistributedLock lock;   public LockThread(int threadId, CountDownLatch latch) throws Exception {  this.lock = new DistributedLock(threadId, latch);  }   @Override  public void run() {  try {  // 每个线程对象启动之后都应该创建有一个临时的节点数据  this.lock.createChildrenNode();  this.lock.handle();  } catch (Exception e) {  e.printStackTrace();  }  } } class TestDistributedLock {  public static void main(String[] args) throws Exception {  CountDownLatch latch = new CountDownLatch(10);  for (int i = 0; i < 10; i ++) {  new Thread(new LockThread(i, latch)).start();  }  latch.await();  System.*out*.println("[所有的线程对象操作完毕]");  } } |

此时线程启动之后将会创建各自的临时节点，而当操作完毕之后会进行临时节点的释放处理。

## 统计最小节点

统计最小节点的逻辑思路：

* 当某一个线程连接到ZooKeeper服务之后，那么应该创建有响应的子节点（瞬时节点），当有了这个子节点之后，应该去判断，当前线程的路径编号信息是否为最小路径值，如果是，则直接取得当前的线程对象进行操作，如果不是，则应该继续向下取得，此时需要考虑一个问题：有可能某些操作项的数据在你判断之后已经不在了，所以需要考虑移除掉此类情况。
* 在判断你的时候应该取得全部的瞬时节点数据，而后排序取得里面最小的内容；
* 同时在用户操作完毕之后应该讲此节点的信息删除掉，而后再删除后应该再一次判断是否为最小节点。

范例：实现分布式锁

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.lock;  import org.apache.zookeeper.\*; import org.apache.zookeeper.data.Stat;  import java.util.Collections; import java.util.List; import java.util.concurrent.CountDownLatch;  */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 21:03  \*/* public class DistributedLock2 {  public static final String *CONNECTION\_URL* = "192.168.127.134:2181,192.168.127.135:2181,192.168.127.136:2181";  public static final int *SESSION\_TIMEOUT* = 3000;  public static final String *SCHEME* = "digest";  public static final String *AUTH\_INFO* = "zkuser:650901";  public static final String *GROUP\_NODE* = "/mohist-lock";  public static final String *CHILDREN\_NODE* = *GROUP\_NODE* + "/lockthread-";  private CountDownLatch latch;  // 本操作的主要目的是为了在取得zookeeper连接之后才能够进行后续处理  private CountDownLatch connectLatch = new CountDownLatch(1);  private ZooKeeper zkClient;  private String selfPath; // 保存每次创建的临时节点的路径信息  private String waitPath; // 保存下一个要处理的节点  private int threadId = 0;   public DistributedLock2(int threadId, CountDownLatch latch) throws Exception {  this.threadId = threadId;  this.latch = latch;  this.connectZooKeeperServer();  }   public void handle() throws Exception {  this.createChildrenNode();  }   public void handldSuccess() throws Exception {// 表示取得锁之后进行的处理  if (this.zkClient.exists(this.selfPath, false) == null) {  return;  }  this.handlCallback();  this.zkClient.delete(this.selfPath, -1);  this.releaseZooKeeper();  this.latch.countDown();  }   public void handlCallback() throws InterruptedException {  Thread.*sleep*(200);  // 取得分布式锁之后的目的是要进行具体的业务操作  System.*out*.println("[Thread - " + this.threadId + "]获得操作权，执行具体的业务处理操作。");  }   public void createChildrenNode() throws Exception {  this.selfPath = this.zkClient.create(*CHILDREN\_NODE*, ("Thread-" + this.threadId).getBytes(),  ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*EPHEMERAL\_SEQUENTIAL*);  System.*out*.println("[Thread - " + this.threadId + "]创建新的临时节点 " + this.selfPath);  if (this.checkMinPath()) {  this.handldSuccess();  }  }   public boolean checkMinPath() throws Exception { // 进行最小节点的判断  List<String> children = this.zkClient.getChildren(*GROUP\_NODE*, false);  Collections.*sort*(children);  int index = children.indexOf(this.selfPath.substring(*GROUP\_NODE*.length() + 1));  switch (index) {  case 0 : {  // 已经确定好当前的节点为最小节点  return true;  }  case -1 : {  // 该节点可能已经消失了  return false;  }  default:{  // 表示现在该节点不属于最小节点，那么久需要继续向后排查  this.waitPath = *GROUP\_NODE* + "/" + children.get(index - 1);  try {  this.zkClient.getData(waitPath, true, new Stat());  return false;  }catch (Exception e) {  if (this.zkClient.exists(this.waitPath, false) == null) {  return this.checkMinPath();  } else {  throw e;  }  }  }  }  }   public void connectZooKeeperServer() throws Exception {  this.zkClient = new ZooKeeper(*CONNECTION\_URL*, *SESSION\_TIMEOUT*, new Watcher() {  @Override  public void process(WatchedEvent event) {  if (event.getType() == Event.EventType.*None*) {  DistributedLock2.this.connectLatch.countDown();  } else {  if (event.getType() == Event.EventType.*NodeDeleted* && event.getPath().equals(DistributedLock2.this.waitPath)) {  try {  if (DistributedLock2.this.checkMinPath()) {  DistributedLock2.this.handldSuccess();  }  } catch (Exception e) {  e.printStackTrace();  }  }  }  }  });  this.zkClient.addAuthInfo(*SCHEME*, *AUTH\_INFO*.getBytes());  if (this.zkClient.exists(*GROUP\_NODE*, false) == null) {  this.zkClient.create(*GROUP\_NODE*, "LOCKDEMO".getBytes(), ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  }  this.connectLatch.await();  }   public void releaseZooKeeper() {  if (this.zkClient != null) {  try {  this.zkClient.close();  System.*out*.println("[Thread - " + this.threadId + "]销毁临时节点 " + this.selfPath);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

范例：测试代码

|  |
| --- |
| package com.kuhnwei.mohist.examples.zookeeper.lock;  import java.util.concurrent.CountDownLatch;  */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/1 21:15  \*/* public class LockThread2 implements Runnable {  private DistributedLock2 lock2;   public LockThread2(int threadId, CountDownLatch latch) throws Exception {  this.lock2 = new DistributedLock2(threadId, latch);  }   @Override  public void run() {  try {  this.lock2.handle();  } catch (Exception e) {  e.printStackTrace();  }  } }  class TestDistributedLock2 {  public static void main(String[] args) throws Exception {  CountDownLatch latch = new CountDownLatch(10);  for (int i = 0; i < 10; i ++) {  new Thread(new LockThread2(i, latch)).start();  }  latch.await();  System.*out*.println("[所有的线程对象操作完毕]");  } } |

如果此时要是有多台服务器的话，本质上锁的概念是没有区别的，但是一定要注意一个问题，这些判断的处理都是通过zookeeper保存的信息动态决定的。后到的操作对象一定是最后执行。

如果多个服务器的对象操作同一资源该怎么做？

可以使用统一的协调组件进行分布式锁实现。