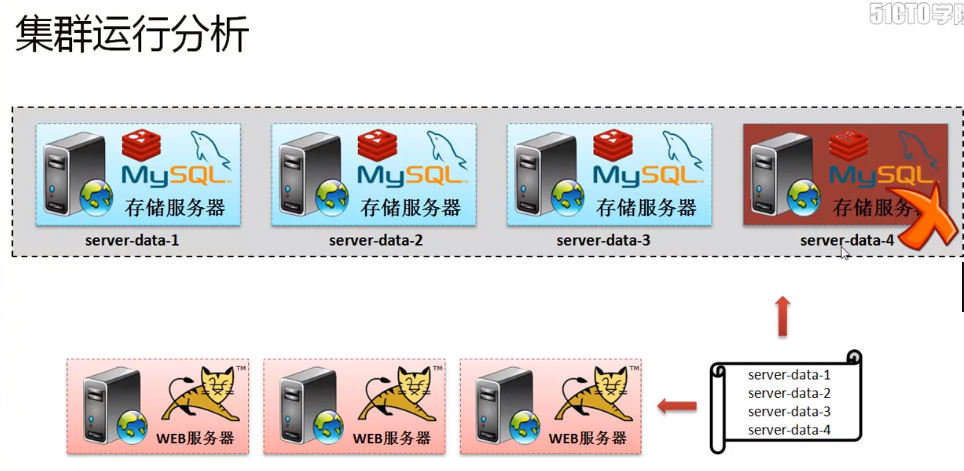
# Zookeeper组件

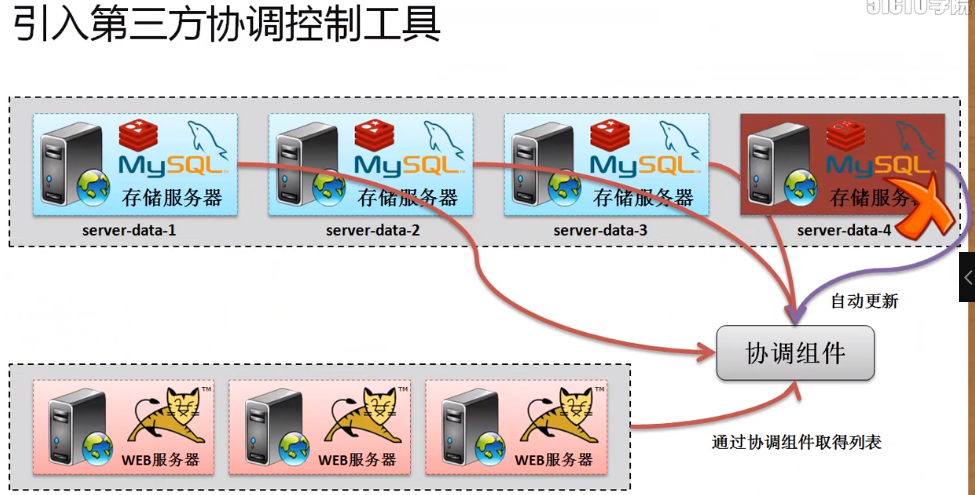
## ZooKeeper组件作用域算法分析

### 服务器集群问题分析



如果有一台mysql服务宕机了，那么web端调用的服务列表信息该如何修改？在实际的项目运行中，web服务器也可能有多台，那么以十台web服务器为例，如果某一台mysql服务器宕机了，那么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，那么此时将会造成一个无法使用的困境。

为解决这样的窘境，往往需要时用一些公共的资源。

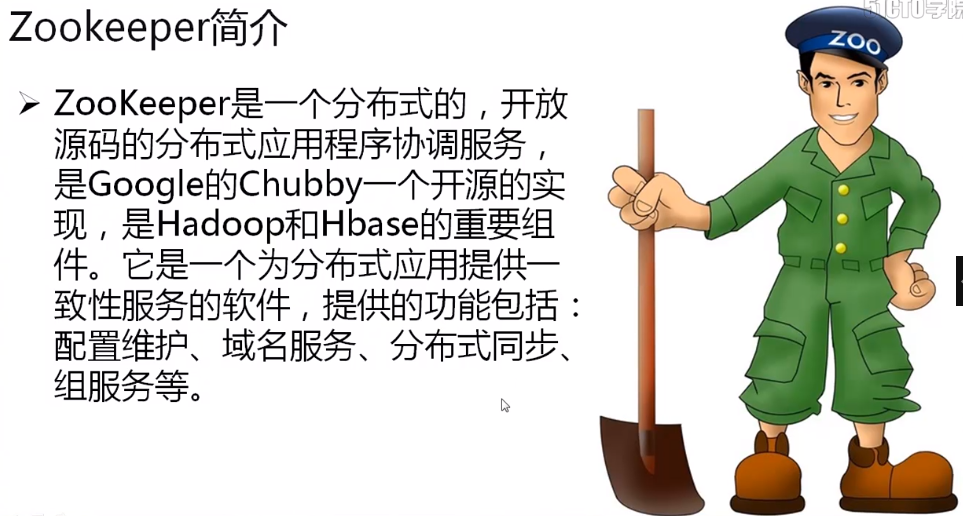


如果突然有台主机宕机了，则在整个协调组件上进行服务器列表的动态更新。这样就可以避免web服务器调用已经宕机的服务。

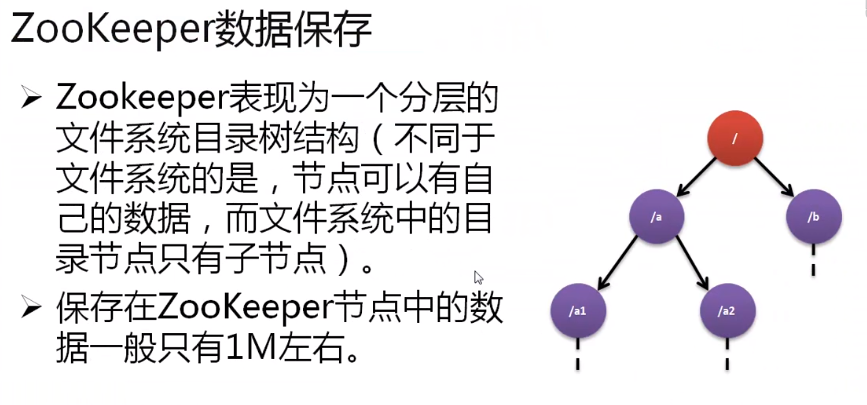
所以这种协调一致性的组件最好的是开源的Apache ZooKeeper组件。

### ZooKeeper简介

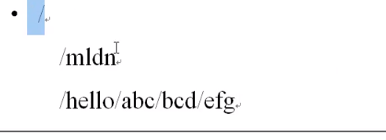
ZooKeeper组件官方下载站点：<https://zookeeper.apache.org/>



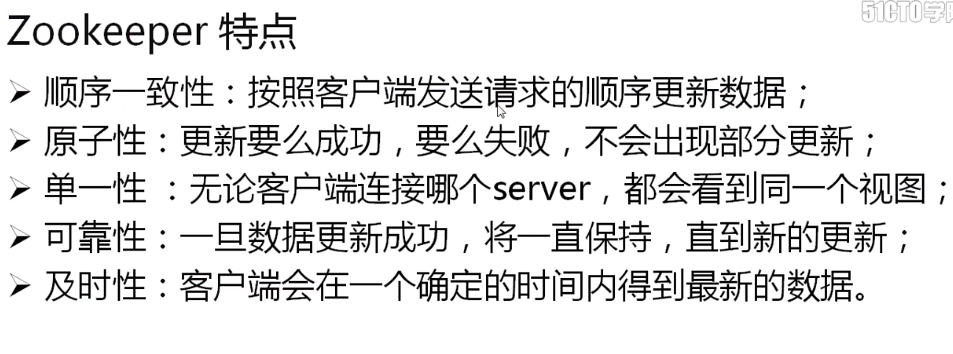
简单解释：所有主机信息或你嗦需要的服务信息都依托ZooKeeper进行保存。



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“/”，根节点下可以有无数个子节点和孙子节点：



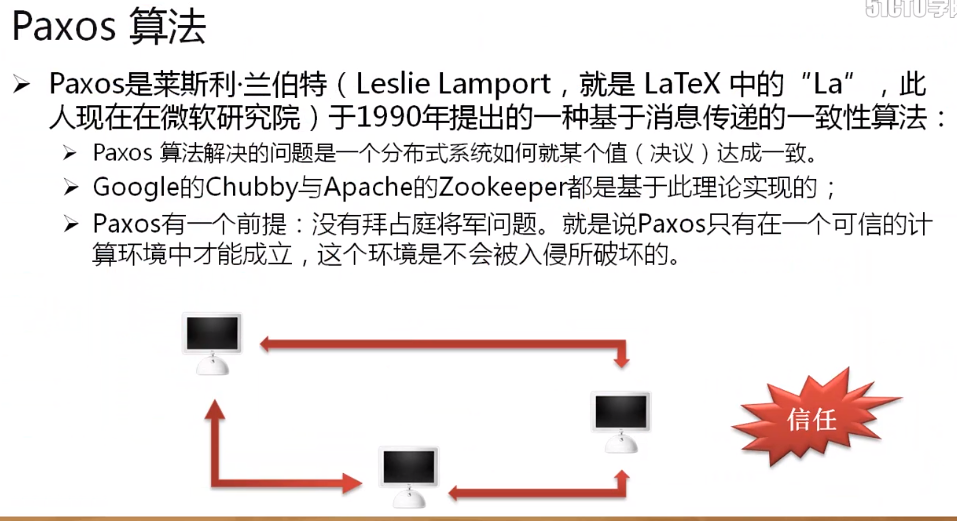
在ZooKeeper进行数据保存的时候保存的都是关键信息，例如：服务器地址、端口号、接口名称。特别说明不要存储中文。



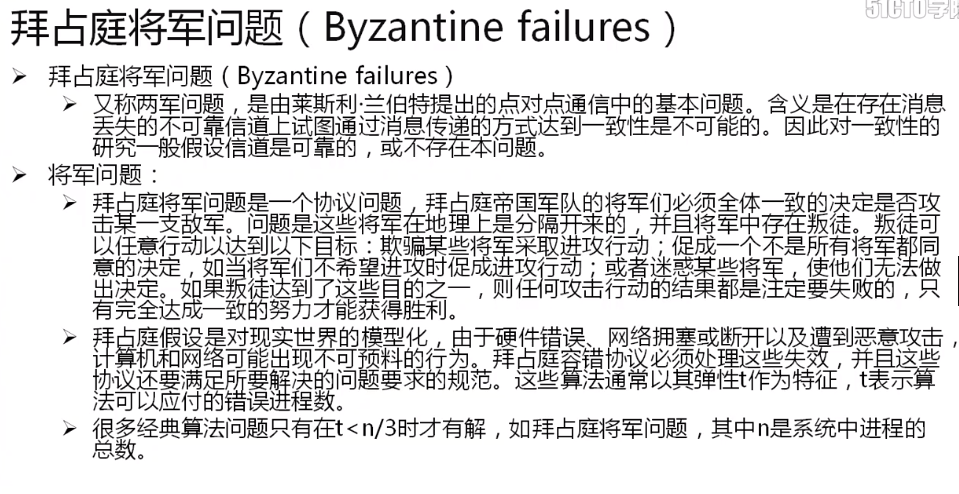
如果每台服务器上都设置了统一的ZooKeeper,那么设置存储的数据所有ZooKeeper节点共享。

### Paxos算法

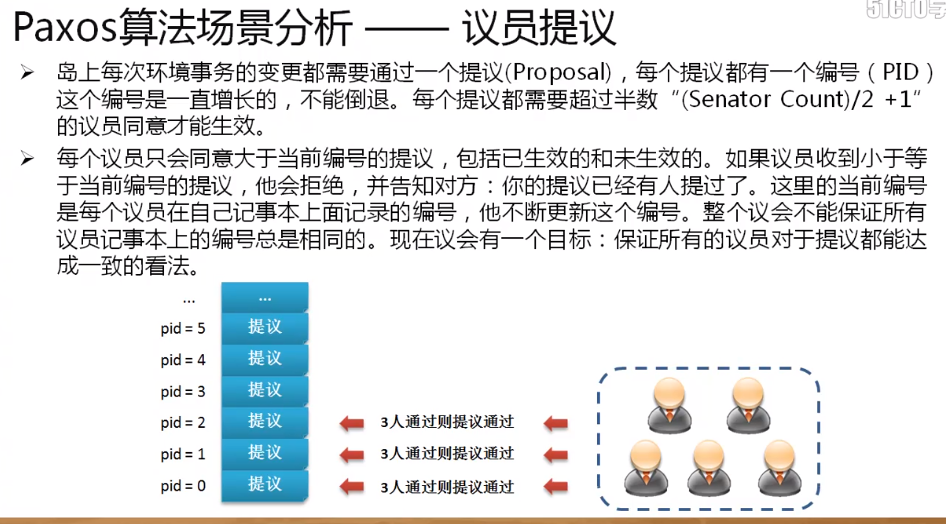
算法本质：在一个信任的环境下，推选领导进行所有ZooKeeper节点操作。

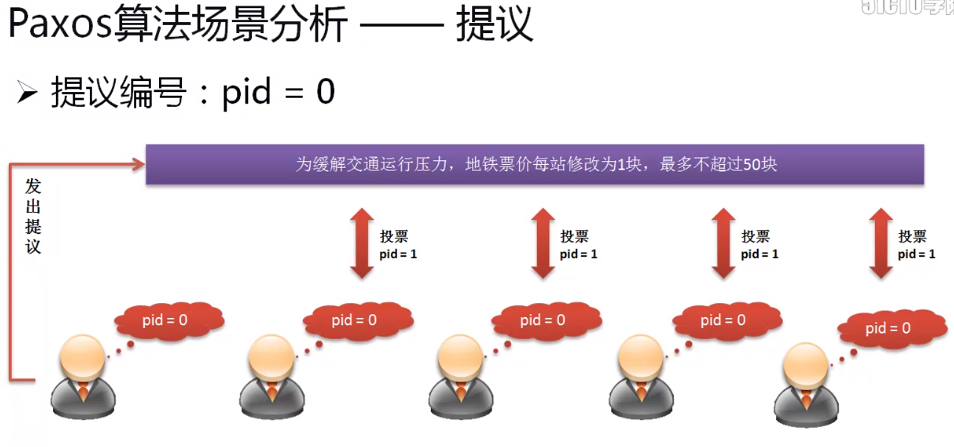


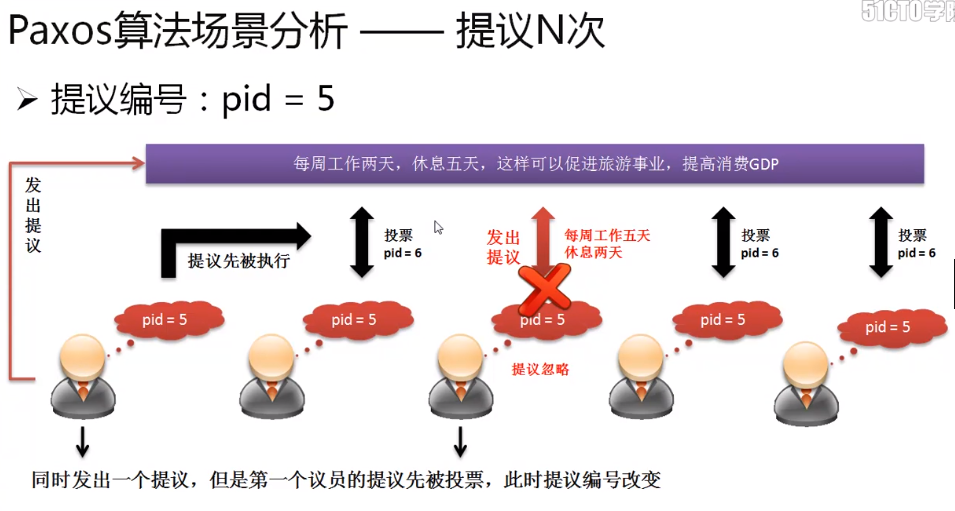
Apache坚持宣称ZooKeeper不是Paxos算法的实现，但是从种种迹象来看ZooKeeper跟Paxos算法描述几乎等同。

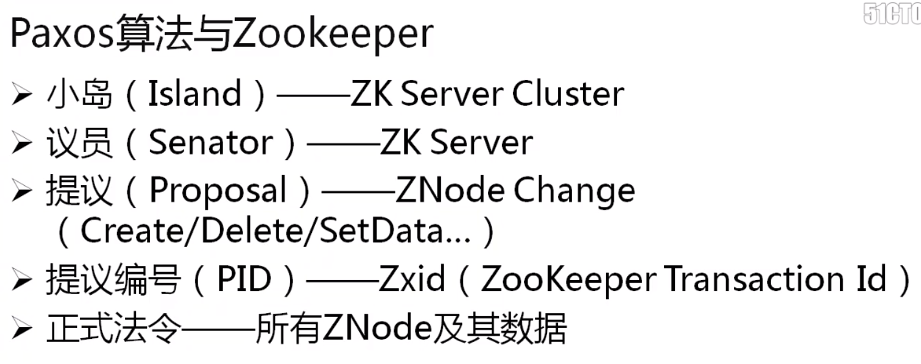


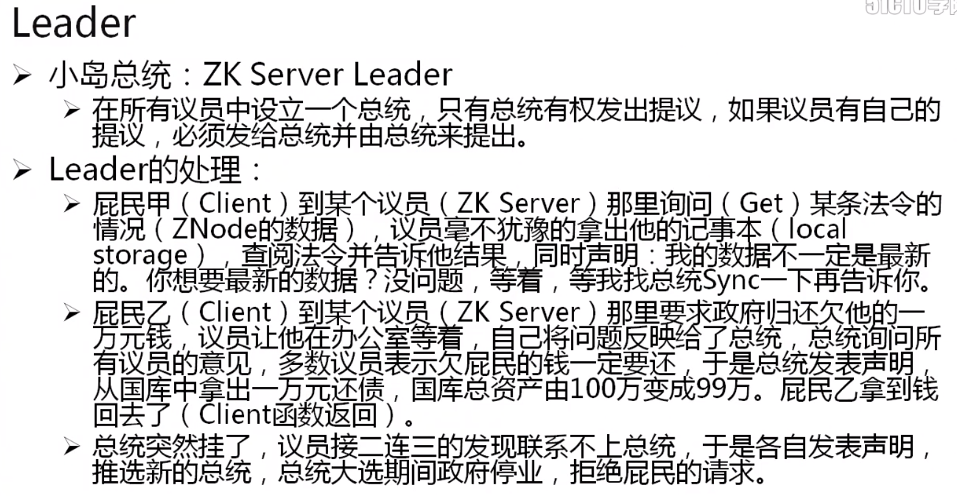












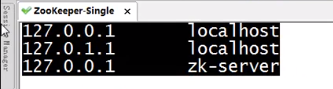
Paxos算法的核心本质：一定要有选举机制（多台服务）,操作的环境一定是信任的环境。

## ZooKeeper安装与配置

ZooKeeper运行环境一定是三台以上的服务器集群环境，但是这种运行的模式是在实际的项目部署环节下才拥有的，可是开发时候往往只需要一台ZooKeeper就够了。

### 安装并配置ZooKeeper

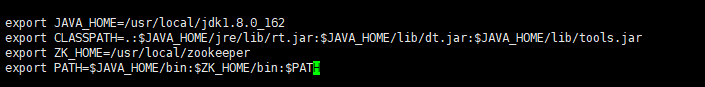
1. 【zk-server】修改主机配置信息
2. 修改主机名称：vi /etc/hostname;
3. 重启主机
4. 修改hosts配置（追加）：vi /etc/hosts



1. 【zk-server】上传ZooKeeper开发包上传到服务器，前提是该系统中已经配置好JDK。
2. 【zk-server】将组件解压缩到“/usr/local”。
3. 【zk-server】为了方便进行ZooKeeper使用，建议将解压缩后的文件夹改个名：



1. 【zk-server】将ZooKeeper的可执行命令配置到系统的环境属性中：
2. 打开环境配置文件：vi /etc/profile
3. 输入如下内容：



1. 是配置文件立即失效：source /etc/profile
2. 【zk-server】进入到ZooKeeper配置文件目录，拷贝配置文件：

cp /usr/local/zookeeper/conf/zoo\_sample.cfg /usr/local/zookeeper/conf/zoo.cfg

ZooKeeper使用的配置文件名称必须是：“zoo.cfg”;

修改数据存储目录：dataDir=/usr/data/zookeeper

创建目录：mkdir -p /usr/data/zookeeper

1. 【zk-server】启动服务进程：

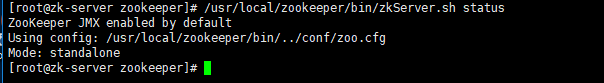
/usr/local/zookeeper/bin/zkServer.sh start

1. 【zk-server】ZooKeeper是一个java进程，所以可以使用“jps”查看本机java进程信息；



1. 【zk-server】查看ZooKeeper运行状态：

/usr/local/zookeeper/bin/zkServer.sh status

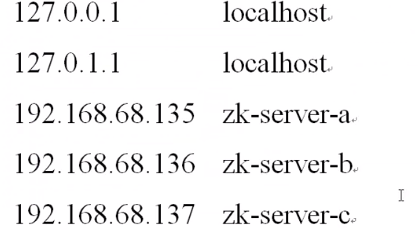


### ZooKeeper集群部署

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 主机名称 | IP地址 | 描述 |
| 1 | Zk-server-a | 192.168.1.223 | ZooKeeper服务 |
| 2 | Zk-server-b | 192.168.1.222 | ZooKeeper服务 |
| 3 | Zk-server-c | 192.168.1.222 | ZooKeeper服务 |

1. 【Zk-server-\*】修改主机名称：hostname、hosts

主要是hosts文件（如果使用IP形式则可以不配置其他主机的IP映射）



1. 【Zk-server-\*】进行ssh的免登陆配置处理（可跳过）
2. 现在为了防止本机已经存在ssh免登陆配置，建议先进行删除处理：



1. 生成新的SSH的key-get:



1. 进行本机授权控制：



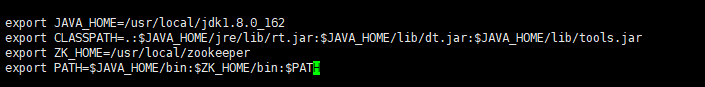
测试免登陆：



1. 进行其他主机的授权连接处理（互相配置，此操作只需在a主机上进行）：

|  |  |
| --- | --- |
| 将本机的ssh公钥拷贝到其他主机： |  |

1. 【zk-server-a】将Zookeeper组件上传服务器，并进行解压缩到“/usr/local/”,将解压缩后的文件夹重命名为zookeeper
2. 【zk-server-a】修改/etc/profile配置文件，并重新加载配置：



1. 【zk-server-a】将此时的profile配置项拷贝给其他主机，并重新加载配置文件

6、【zk-server-a】拷贝zookeeper配置文件：

cp /usr/local/zookeeper/conf/zoo\_sample.cfg /usr/local/zookeeper/conf/zoo.cfg

ZooKeeper使用的配置文件名称必须是：“zoo.cfg”;

1. 【zk-server-a】建立Zookeeper数据保存目录：

mkdir -p /usr/data/zookeeper

1. 【zk-server-a】修改zoo.cfg配置文件（服务器名称可以使用IP代替）：

|  |  |
| --- | --- |
| 修改数据的文件存储目录： | dataDir=/usr/data/zookeeper |
| 新增主机信息列表 | 或者：  server.1=192.168.1.223:2888:3888  server.2=192.168.1.222:2888:3888  server.3=192.168.1.221:2888:3888 |

主机信息格式“server.主机编号=主机名称:程序监听端口：程序选举端口”，其中1,2,3这几个值可随意配置。

1. 【zk-server-a】在ZooKeeper中所有主机编号都在工作目录目录中以“myid”的文件形式出现。

a主机:

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

b主机:

echo 2 >> /usr/data/zookeeper/myid

c主机:

echo 3 >> /usr/data/zookeeper/myid

1. 【zk-server-a】将已经配置好的Zookeeper目录拷贝到其他主机上

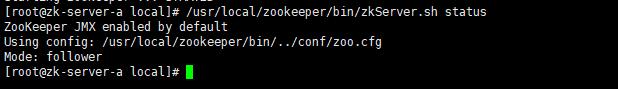
scp -r /usr/local/zookeeper/ 192.168.1.222:/usr/local/

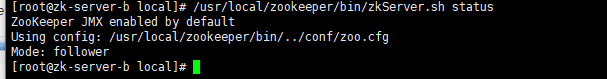
1. 【zk-server-\*】启动Zookeeper服务：

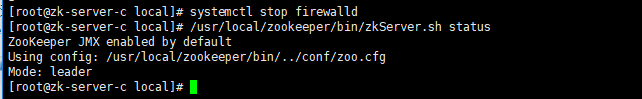
/usr/local/zookeeper/bin/zkServer.sh start

查看Zookeeper状态：

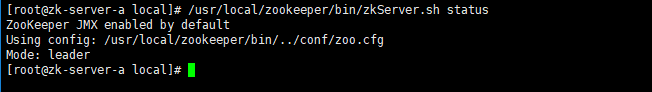
/usr/local/zookeeper/bin/zkServer.sh status

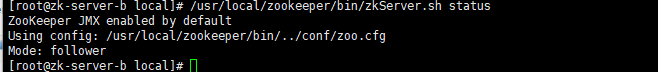






干掉一个领导：在查看。





遇到问题：Error contacting service. It is probably not running，关闭防火墙即可。

参考：<https://blog.csdn.net/lihao21/article/details/51778255>

### ZooKeeper客户端操作

Zookeeper提供的专属操作命令：usr/local/zookeeper/bin/zkCli.sh

1. 【zk-server-a】在zk-server-a上登录zk-server-b主机：



也可以使用ip：

local/zookeeper/bin/zkCli.sh -server 192.168.1.222

1. 【zk-server-a】使用“help”查看命令：
2. 【zk-server-a】列出根目录下的节点信息：

ls /ls

此时会发现存在一个“/zookeeper”的第二层节点。描述的是各个Zookeeper连接信息。

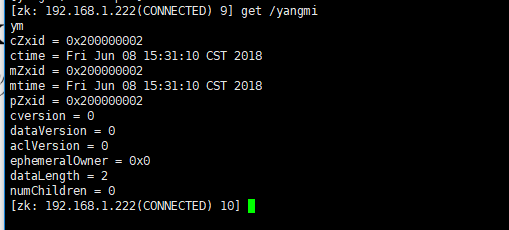
1. 【zk-server-a】创建一个新的节点：
2. 创建节点，不设置数据（很明显是创建失败的）：

create /yangmi

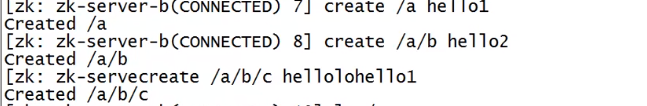
1. 创建节点同时设置节点数据：

create /yangmi ym

1. 【zk-server-c】查看新创建节点的数据信息：get /节点名称



1. 【zk-server-c】创建多层节点：/a/b/c,创建失败，因为Zookeeper不允许多层创建。只能逐一创建：



1. 【zk-server-c】删除节点：delete /节点。不能一次删除多节点，只能从下往上逐一删除。
2. 对于此时的数据实际上还可以进行监听控制。

(1)【zk-server-b】对某一节点监听：

get /yangmi watch

1. 【zk-server-c】随意启动一台主机的Zookeeper客户端连接。

/usr/local/zookeeper/bin/zkCli.sh -server 192.168.1.223

1. 修改被监听节点的数据：

也就是说一旦启用了监听状态，其他节点对此节点的数据修改将会立即发出更新状态。

虽然是多台主机，但是所有的数据更改几乎都是瞬间完成。

set /yangmi ym223

总结：

1. Zookeeper中节点必须一层层创建或者删除；
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，则可以立刻监听到变化，所有的监听只允许监听一次。

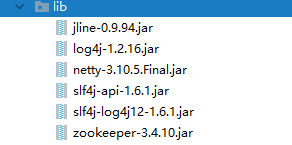
## 使用java操作ZooKeeper

将Zookeeper开发包解压缩，因为此时的Zookeeper开发包中提供lib。

### 连接zookeeper集群

进行Zookeeper连接，那么首先要使用一个核心类：org.apache.zookeeper.ZooKeeper。

1. ZooKeeper(String connectString, int sessionTimeout, Watcher watcher)
   1. connectString:所有要进行连接的Zookeeper的地址信息，默认端口：2181
   2. sessionTimeout：连接超时时间；
   3. watcher：进行监听器的配置
2. 当取得Zookeeper的连接对象后，就可以利用此对象取得所有的节点信息；
   1. 方法：public List<String> getChildren(String path,boolean watch)throws KeeperException,InterruptedException
3. 将开发包加入到项目中。



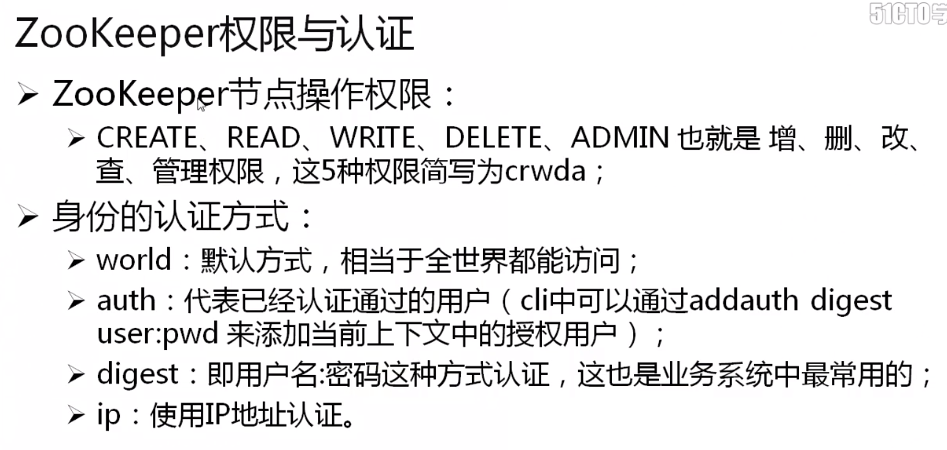
1. 实现Zookeeper连接：

**public class** ConnectionZooKeeperDemo {  
 **public static void** main(String[] args) **throws** Exception {  
 *// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败* ZooKeeper zkClient = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 }  
 });  
 List<String> children = zkClient.getChildren(**"/"**, **false**); *// 得到所有的根节点下的子节点* Iterator<String> iter = children.iterator();  
 **while** (iter.hasNext()) {  
 System.***out***.println(iter.next());  
 }  
 zkClient.close(); *// 释放连接* }  
}

此时连接的Zookeeper集群状态，所以给出的地址“”，正常的使用结构一定是通过第一个地址先读取，而后不行的话在往下读取。

### ACL认证控制

所谓ACL指的是访问控制列表。



1. 【zk-server-a】使用Zookeeper的客户端工具登录Zookeeper：

/usr/local/zookeeper/bin/zkCli.sh -server 192.168.1.222

1. 【zk-server-a】授权控制的前提需要有一个用户：

addauth digest zkuser:yangmi

1. 【zk-server-a】进行授权操作控制：

setAcl / auth:zkuser:yangmi:cdrwa

用另一个客户端访问：

连接的时候不会提示认证，但是一旦执行操作，比如：ls /，则会提示没有认证。



那么此时就需要重新进行认证控制：

addauth digest zkuser:yangmi

1. 【idea开发】如果要进行认证处理，则观察Zookeeper类中提供的方法：
2. 认证设置：void addAuthInfo(String scheme, byte[] auth)

**public class** ConnectionZooKeeperDemo {  
 **public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息* **public static void** main(String[] args) **throws** Exception {  
 *// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败* ZooKeeper zkClient = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 }  
 });  
 zkClient.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes());*// 进行认证授权* List<String> children = zkClient.getChildren(**"/"**, **false**); *// 得到所有的根节点下的子节点* Iterator<String> iter = children.iterator();  
 **while** (iter.hasNext()) {  
 System.***out***.println(iter.next());  
 }  
 zkClient.close(); *// 释放连接* }

### 创建Zookeeper节点

要进行节点的处理基本上=就是create、delete，对于这些命令Zookeeper的类中都有与之对应的处理方法：

1. 节点创建：

void create(String path, byte[] data, List<ACL> acl, createMode

createMode, AsyncCallback.StringCallback cb, Object ctx)

* 1. 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
  2. byte[] data:创建节点的时候可以设置节点对应的内容，对此处的内容使用的字节数组，不要用中文。
  3. List<ACL> acl：进行安全认证，可以直接使用已有的模式完成
     1. 所有节点的认证信息都保存在了“org.apache.zookeeper.ZooDefs.Ids”
        1. static final ArrayList<ACL> CREATOR\_ALL\_ACL:表示所有人可以访问
  4. CreateMode createMode:节点的创建模式，使用“org.apache.zookeeper.CreateMode”枚举类
     1. public static final CreateMode EPHEMERAL：创建一个瞬时节点，连接断开时消失。
     2. public static final CreateMode EPHEMERAL\_SEQUENTIAL：创建一个瞬时的序列节点。
     3. public static final CreateMode PERSISTENT：创建一个持久化节点;
     4. public static final CreateMode PERSISTENT\_SEQUENTIAL:创建一个持久化的序列节点。

此方法返回的是路径。

1. 判断节点是否有存在： Stat exists(String path, boolean watch) ，不存在则返回null

范例：创建一个普通节点：

**public class** CreateZNodeDemoA {  
 **public static final** String ***AUTH\_INFO*** = **"zkuser:mldnjava"**; *// 进行连接的授权信息* **public static final** String ***GROUPNODE*** = **"/mldndata"**; *// 描述的是根节点* **public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/bigdata"**; *// 描述的是子节点* **public static void** main(String[] args) **throws** Exception {  
 *// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.68.135:2181,192.168.68.136,192.168.68.137"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败* ZooKeeper zkClient = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 }  
 });  
 zkClient.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权* **if** (zkClient.exists(***GROUPNODE***, **false**) == **null**) { *// 表示节点不存在  
 // 所有保存在节点中的数据一定要是字节，而且节点的数据千万别设置中文* zkClient.create(***GROUPNODE***, **"HelloData"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  
 }  
 **if** (zkClient.exists(***SUBNODE***, **false**) == **null**) {  
 zkClient.create(***SUBNODE***, **"HelloSub"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  
 }  
 zkClient.close(); *// 释放连接* }  
}

范例：创建（瞬时）临时节点

**public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息* **public static final** String ***GROUPNODE*** = **"/mldndata"**; *// 描述的是根节点* **public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/bigdata"**; *// 描述的是子节点* **public static void** main(String[] args) **throws** Exception {  
*// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败* ZooKeeper zkClient = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 }  
 });  
 zkClient.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权* **if** (zkClient.exists(***GROUPNODE***, **false**) == **null**) { *// 表示节点不存在  
 // 所有保存在节点中的数据一定要是字节，而且节点的数据千万别设置中文* zkClient.create(***GROUPNODE***, **"HelloData"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  
 }  
 **if** (zkClient.exists(***SUBNODE***, **false**) == **null**) {  
 String createPath = zkClient.create(***SUBNODE***, **"HelloSub"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***EPHEMERAL***);  
 System.***out***.println(**"新的节点路径："** + createPath);  
 }  
 Thread.*sleep*(Long.***MAX\_VALUE***);  
 zkClient.close(); *// 释放连接* }

范例：创建序列化临时节点（节点编号顺序增长）

**public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息***public static final** String ***GROUPNODE*** = **"/mldndata"**; *// 描述的是根节点  
// 现在给出的子节点的名称只能够是一个开头的标记***public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/cluster-"**; *// 描述的是子节点***public static void** main(String[] args) **throws** Exception {  
 *// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败* ZooKeeper zkClient = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 }  
 });  
 zkClient.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权* **if** (zkClient.exists(***GROUPNODE***, **false**) == **null**) { *// 表示节点不存在  
 // 所有保存在节点中的数据一定要是字节，而且节点的数据千万别设置中文* zkClient.create(***GROUPNODE***, **"HelloData"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  
 }  
 **for** (**int** x = 0; x < 10; x++) {  
 **if** (zkClient.exists(***SUBNODE***, **false**) == **null**) {  
 String createPath = zkClient.create(***SUBNODE***,  
 **"HelloSub"**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,  
 CreateMode.***EPHEMERAL\_SEQUENTIAL***);  
 System.***out***.println(**"新的节点路径："** + createPath);  
 }  
 }  
 Thread.*sleep*(Long.***MAX\_VALUE***);  
 zkClient.close(); *// 释放连接*}

### Zookeeper数据处理

节点的数据处理本质就是取得、设置（修改）、删除等基本的处理操作，这些做错的方法都在ZooKeeper类中定义。

1. 取得节点中的数据内容：
2. 取得方法： byte[] getData(String path, boolean watch, Stat stat)
3. String path:要读取的路径信息；
4. boolean watch：是否需要监听；
5. Stat stat：取得相关的统计信息；

范例：取得节点数据

**public class** CreateZNodeDemoA {  
 **public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息* **public static final** String ***GROUPNODE*** = **"/mldndata"**; *// 描述的是根节点  
 // 现在给出的子节点的名称只能够是一个开头的标记* **public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/cluster-"**; *// 描述的是子节点* **public static void** main(String[] args) **throws** Exception {  
 *// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败* ZooKeeper zkClient = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 }  
 });  
 zkClient.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权* **if** (zkClient.exists(***GROUPNODE***, **false**) != **null**) { *// 现在节点存在了* Stat stat = **new** Stat();  
 *// 获取指定路径的数据信息，数据信息会以字节数组的形式返回* String data = **new** String(zkClient.getData(***GROUPNODE***, **false**, stat));  
 System.***out***.println(data);  
 }  
 zkClient.close(); *// 释放连接* }  
}

1. 修改（设置）数据
2. 方法： Stat setData(String path, byte[] data, int version) ，version=-1表示不收版本控制

范例：设置节点数据

**public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息***public static final** String ***GROUPNODE*** = **"/mldndata"**; *// 描述的是根节点  
// 现在给出的子节点的名称只能够是一个开头的标记***public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/cluster-"**; *// 描述的是子节点***public static void** main(String[] args) **throws** Exception {  
 *// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败* ZooKeeper zkClient = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 }  
 });  
 zkClient.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权* **if** (zkClient.exists(***GROUPNODE***, **false**) != **null**) { *// 现在节点存在了* zkClient.setData(***GROUPNODE***, **"ymd"**.getBytes(), -1);  
 }  
 zkClient.close(); *// 释放连接*}

1. 删除节点
2. 方法：void delete(String path, int version)

范例：删除节点

zkClient.delete(***GROUPNODE***, -1);

### Zookeeper节点监听

·数据监听是整个Zookeeper中最为重要的核心组成，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进行及时的相应处理。

WatchedEvent类中可以取得操作的事件类型，该类型是一个枚举类型（）:

org.apache.zookeeper.Watcher.Event.EventType,该枚举类中存在有如下状态：

(1)NodeChildrenChanged :子节点改变

(2)NodeCreated :节点创建

(3)NodeDataChanged :节点数据变更

(4)NodeDeleted :节点数据删除

(5)None :没有操作，刚打开连接，所有节点是不可能有任何操作的。

范例：单个节点的监听

**public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息***public static final** String ***GROUPNODE*** = **"/mldndata"**; *// 描述的是根节点***private static** ZooKeeper *zkClient* = **null**;  
*// 现在给出的子节点的名称只能够是一个开头的标记***public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/cluster-"**; *// 描述的是子节点***public static void** main(String[] args) **throws** Exception {  
 *// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败  
 zkClient* = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 **try** {  
 *zkClient*.exists(***GROUPNODE***, **true**);*// 继续监听* } **catch** (KeeperException e) {  
 e.printStackTrace();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
 *zkClient*.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权  
 zkClient*.exists(***GROUPNODE***, **true**); *//表示进行节点的监听控制* Thread.*sleep*(Long.***MAX\_VALUE***);*// ZooKeeper对于监听的控制必须使用此种模式休眠  
 zkClient*.close(); *// 释放连接*}

然后在服务器上登录客户端，进行节点操作，在java端观察监听控制。

通过操作发现，非常遗憾的是，以上的监听程序只能对一个节点监听，无法对子节点监听，因为使用的方法有错误。

getChildred()方法才可以实现子节点的监听：

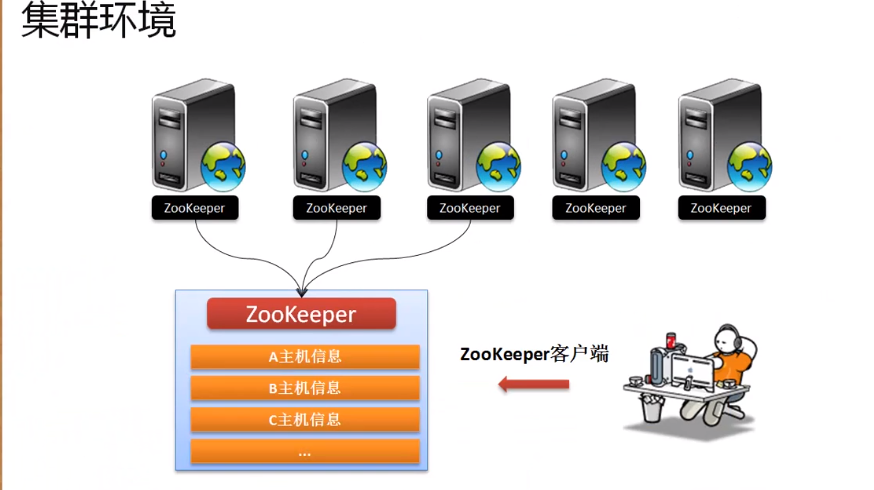
范例：对所有子节点监听

**public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息***public static final** String ***GROUPNODE*** = **"/mldndata"**; *// 描述的是根节点***private static** ZooKeeper *zkClient* = **null**;  
*// 现在给出的子节点的名称只能够是一个开头的标记***public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/cluster-"**; *// 描述的是子节点***public static void** main(String[] args) **throws** Exception {  
 *// 多个连接地址之间使用“,”分割，如果不写端口号就是2181* String connectString = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**; *// 配置所有的连接地址* **int** sessionTimeout = 2000; *// 两秒为超时时间，也就是说如果超过2秒还没有连接成功，则表示连接失败  
 zkClient* = **new** ZooKeeper(connectString, sessionTimeout,  
 **new** Watcher() { *// 对监听进行处理控制* @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"\*\*\* 【监听事件处理】path = "** + event.getPath() + **"、type = "** + event.getType()  
 + **"、状态 = "** + event.getState());  
 **try** {  
 *zkClient*.getChildren(***GROUPNODE***, **true**);*// 继续监听* } **catch** (KeeperException e) {  
 e.printStackTrace();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
 *zkClient*.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权  
 zkClient*.getChildren(***GROUPNODE***, **true**);*// 对所有子节点监听* Thread.*sleep*(Long.***MAX\_VALUE***);*// ZooKeeper对于监听的控制必须使用此种模式休眠  
 zkClient*.close(); *// 释放连接*}

对于子节点的监听无法进行数据的监听操作，但是对于子节点的创建或者是删除是可以监听到的：

## ZooKeeper开发案例：服务列表

### 服务器列表实现思路



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，所以建议将所有的服务器的节点信息都统一保存在“/yangmi-servers”节点之中，所有服务器创建的节点都属于瞬时节点，按如下的方式排列：

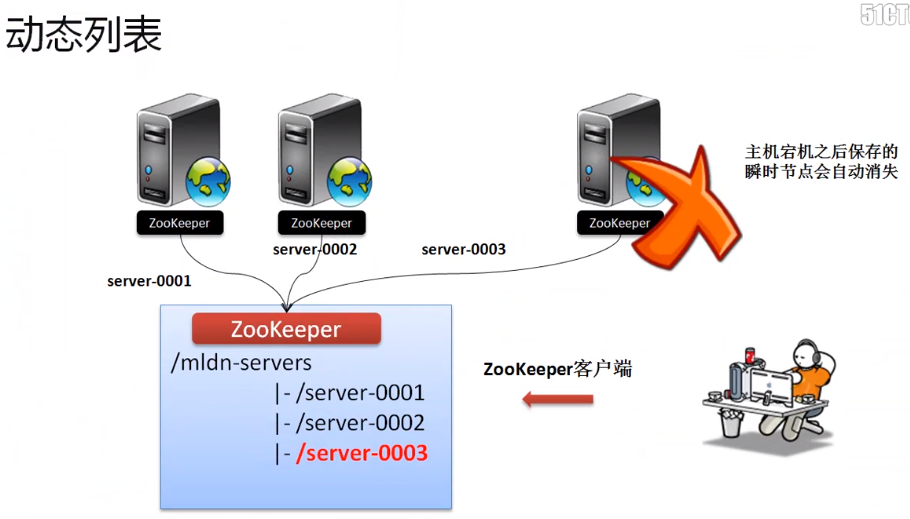
* 【监控节点】/yangmi-server

|- /yangmi-server/server-0000000;

|- /yangmi-server/server-0000001;

|- /yangmi-server/server-0000002;

|- /yangmi-server/server-0000003;



要想实现本类程序，一定要在每一个服务器上都去部署一个Zookeeper（leader、fllower）的运行项目。

### 开发服务器端注册程序

服务器端程序主要指的部署在各个服务器主机上的运行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

**public class** ServerListener {  
 *// 如果要基于ZooKeeper访问，那么肯定需要知道ZooKeeper连接地址信息* **public static final** String ***CONNECTION\_URL*** = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**;  
 **public static final int *SESSION\_TIMEOUT*** = 2000; *// 连接超时时间* **public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息* **public static final** String ***GROUPNODE*** = **"/yangmi-servers"**; *// 根节点* **public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/server-"**; *// 子节点* **private** ZooKeeper **zkClient**; *// 定义一个ZOoKeeper客户端处理对象  
  
 /\*\*  
 \* 在进行服务器信息列表的时候应该明确的知道当前服务器的名称，这个名称可以在程序运行的时候动态设置  
 \*  
 \** ***@param serverName*** *\** ***@throws*** *Exception  
 \*/* **public** ServerListener(String serverName) **throws** Exception {  
 **this**.connectZooKeeperServer(serverName); *// ZooKeeper连接控制，同时进行各个节点的创建* **this**.handle(); *// 连接准备好之后，要自动进行指定任务的调用* }  
  
 */\*\*  
 \* 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有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***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 要执行某些处理的操作 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 Thread.*sleep*(Long.***MAX\_VALUE***); *// 进行休眠* }  
  
 */\*\*  
 \* 取得一个ZooKeeper的连接信息  
 \*  
 \** ***@param serverName*** *\** ***@throws*** *Exception  
 \*/* **public void** connectZooKeeperServer(String serverName) **throws** Exception {  
 **this**.**zkClient** = **new** ZooKeeper(***CONNECTION\_URL***, ***SESSION\_TIMEOUT***, **new** Watcher() {  
 @Override  
 **public void** process(WatchedEvent event) {  
  
 }  
 });  
  
 *// 进行认证授权* **zkClient**.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes());  
 *//// 现在一定要判断父节点是否存在，如果不存在要创建* **if** (**this**.**zkClient**.exists(***GROUPNODE***, **false**) == **null**) {  
 *// 创建的父节点一定是一个持久化节点，而父节点里面的内容才属于瞬时节点* **this**.**zkClient**.create(***GROUPNODE***, **"SERVER-LIST"**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  
 }  
  
 *// 现在子节点不存在* **if** (**this**.**zkClient**.exists(***SUBNODE***, **false**) == **null**) {  
 *// 所有的子节点都应该是一个临时的节点信息，但是该节点信息里面保存有全部的服务器主机名称* **this**.**zkClient**.create(***SUBNODE***, serverName.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,  
 CreateMode.***EPHEMERAL\_SEQUENTIAL***);  
 }  
 }  
}

最终本程序一定是负载服务器上执行，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

**public class** StartServerMain {  
 **public static void** main(String[] args) **throws** Exception {  
 **if** (args.**length** != 1) { *// 现在执行程序的时候没有传递参数* System.***out***.println(**"错误的执行程序，应该在执行的时候输入服务器名称，例如：java 程序类 服务器名称。"**);  
 System.*exit*(1); *// 系统退出* }  
 **new** ServerListener(args[0]); *// 启动ZooKeeper注册* }  
}

登录任意客户端测试：/usr/local/zookeeper/bin/zkCli.sh -server 192.168.1.221

那么此时如果java进程运行着，则表示此时服务器还活着（会发送心跳），则临时节点是不会被删除的，而如果心跳（宕机、网络阻塞、机房故障），则心跳消失，节点也会相应的自动销毁。

### 项目部署与运行

1. 将项目导出为可执行jar；
2. 将jar文件上传到任意一台主机并执行：java -jar xxx.jar;

3、将此jar文件也拷贝到其他主机测试；

### 开发客户端信息取得程序

范例：实现客户端监听

**public class** ServerListClientListener {  
 **public static int** *count* = 0; *// 进行一个计数统计  
 // 如果要基于ZooKeeper访问，那么肯定需要知道ZooKeeper连接地址信息* **public static final** String ***CONNECTION\_URL*** = **"192.168.1.223:2181,192.168.1.222,192.168.1.221"**;  
 **public static final int *SESSION\_TIMEOUT*** = 2000; *// 连接超时时间* **public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息* **public static final** String ***GROUPNODE*** = **"/yangmi-servers"**; *// 根节点* **private** ZooKeeper **zkClient**; *// 定义一个ZOoKeeper客户端处理对象* **public** ServerListClientListener() **throws** Exception {  
 **this**.connectZooKeeperServer(); *// ZooKeeper连接控制，同时进行各个节点的创建* System.***out***.println(**"【第"** + *count*++ + **"次取得服务器列表】"** + **this**.updateServerList());  
 Thread.*sleep*(Long.***MAX\_VALUE***);  
 }  
  
 */\*\*  
 \* 编写一个方法实现所有的服务端信息列表的更新获得  
 \*  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* **public** Set<String> updateServerList() **throws** Exception {  
 *// 只是进行名称的保存* Set<String> allServers = **new** LinkedHashSet<String>();  
 *// 必须监听* List<String> children = **this**.**zkClient**.getChildren(***GROUPNODE***, **true**);  
 Iterator<String> iter = children.iterator();  
 **while** (iter.hasNext()) {  
 String path = ***GROUPNODE*** + **"/"** + iter.next(); *//取得每一个子节点* allServers.add(**new** String(**this**.**zkClient**.getData(path, **false**, **new** Stat())));  
 }  
 **return** allServers;  
 }  
  
 */\*\*  
 \** ***@throws*** *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**) {  
 *// 表示现在有了访问路径* **if** (event.getType() == Event.EventType.***NodeChildrenChanged***) {  
 *// 子节点现在发生了改变* **try** {  
 System.***out***.println(**"【第"** + *count*++ + **"次取得服务器列表】"** + ServerListClientListener.**this**.updateServerList());  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 });  
 *// 进行认证授权* **zkClient**.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes());  
 *// 现在一定要判断父节点是否存在，如果不存在要创建* **if** (**this**.**zkClient**.exists(***GROUPNODE***, **false**) == **null**) {  
 *// 创建的父节点一定是一个持久化节点，而父节点里面的内容才属于瞬时节点* **this**.**zkClient**.create(***GROUPNODE***, **"SERVER-LIST"**.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  
 }  
 }  
}

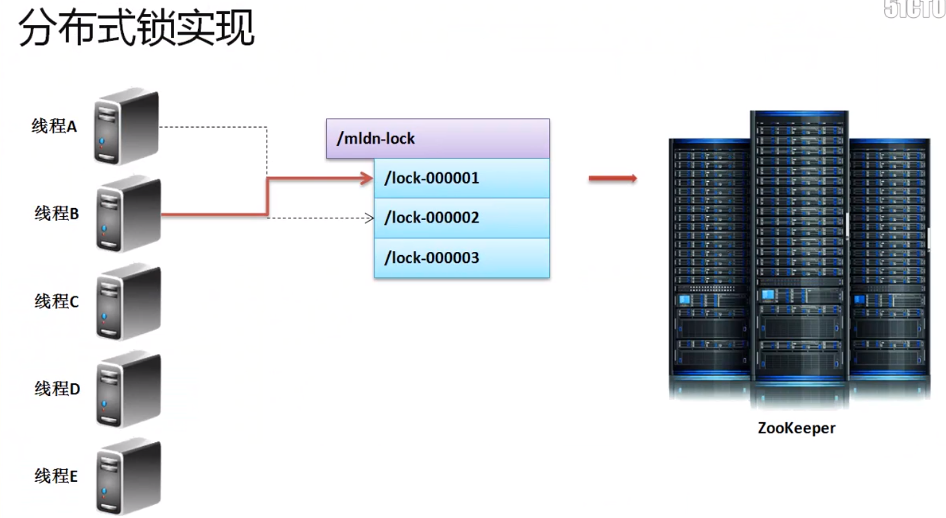
范例：启动客户端

**public static void** main(String[] args) **throws** Exception {  
 ServerListClientListener client = **new** ServerListClientListener() ;  
}

## 综合案例：分布式锁

### 分布式锁实现概述

多台主机（或多个进程）进行资源操作的时候才可能要是用分布式锁。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性组件（zookeeper），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。所有的主机对于Zookeeper来说就是线程，一台主机一个线程。

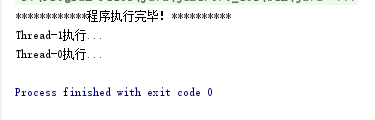


### CountDownLatch

这个类是在jdk1.5之后引入的一个线程同步的处理类，可以轻松的搞定线程的执行问题。

范例：观察以下程序的问题

**class** MyThread **implements** Runnable{  
 @Override  
 **public void** run() {  
 System.***out***.println(Thread.*currentThread*().getName()+**"执行..."**);  
 }  
}  
  
**public class** TestCountDownLatch {  
 **public static void** main(String[] args) **throws** Exception{  
 MyThread mt = **new** MyThread();  
 **new** Thread(mt).start();  
 **new** Thread(mt).start();  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*程序执行完毕！\*\*\*\*\*\*\*\*\*\*"**);  
 }  
}



范例：解决同步的问题：

package cn.mldn.test;

import java.util.concurrent.CountDownLatch;

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。");

latch.countDown(); // 向下减

}

}

public class TestCountDownLatch {

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

CountDownLatch latch = new CountDownLatch(2) ; // 2表示你当前启动的线程数量

MyThread mt = new MyThread(latch) ; // 将CountDownLatch对象传递给MyThread类

new Thread(mt).start();

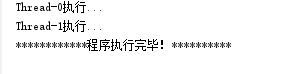
new Thread(mt).start();

latch.await(); // 等待

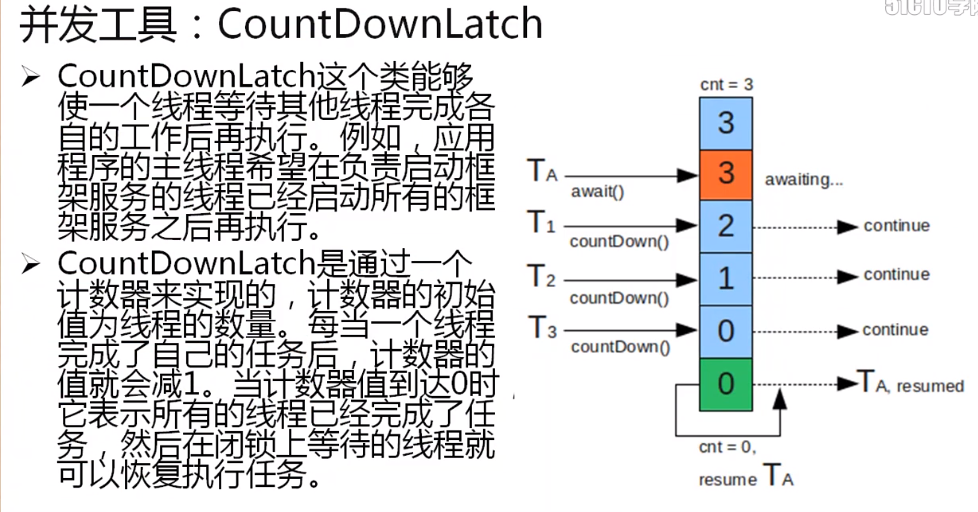
System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 程序执行完毕 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}



CountDownLatch可以理解为一个计数的操作处理，你可以吧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。



在CountDownLatch类中有如下方法：

1. 构造方法：CountDownLatch(int count)
2. 等待其他线程执行完毕后在向下执行：void await()
3. 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。

### 多线程连接模拟

如果想实现分布式锁，那么一定需要将若干个线程的信息以Zookeeper节点的形式存储。

*/\*\*  
 \* Created with IntelliJ IDEA.  
 \* User: Dony  
 \* Date: 2018/6/9  
 \* Time: 20:11  
 \* Description:描述分布式锁的程序处理类  
 \*/***public class** DistributedLock {  
 **public static final** String ***CONNECTION\_URL*** = **"192.168.1.223:2181,192.168.1.222:2181,192.168.1.231:2181"**;  
 **public static final int *SESSION\_TIMEOUT*** = 2000; *// 连接超时时间* **public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息* **public static final** String ***GROUPNODE*** = **"/yangmi-lock"**; *// 根节点* **public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/lockthread-"**; *// 根节点* **private** CountDownLatch **latch** = **null**;  
 *// 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** = **null**; *// 建立ZooKeeper程序控制类* **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;  
  
 */\*\*  
 \* // 进行一些初始化操作使用  
 \*  
 \** ***@param threadId*** *随意给定的一个编号信息  
 \** ***@param latch*** *进行线程同步处理  
 \** ***@throws*** *Exception ZooKeeper异常  
 \*/* **public** DistributedLock(**int** threadId, CountDownLatch latch)  
 **throws** Exception {  
 **this**.**threadId** = threadId; *// 保存每一个线程对象自己的ID信息* **this**.**latch** = latch;  
 **this**.connectionZooKeeper(); *// 进行节点的连接* }  
  
 **public void** handle() **throws** Exception { *// 具体业务处理* **this**.createSubNode(); *// 创建临时节点操作* }  
  
 **public void** handleSuccess() **throws** Exception { *// 表示取得锁之后进行的处理* **if** (**this**.**zkClient**.exists(**this**.**selfPath**, **false**) == **null**) {  
 **return**; *// 如果当前节点不存在* }  
 **this**.handlCallback(); *// 执行具体的业务操作  
 // 如果某一个节点操作完毕了，那么应该立即删除掉该节点，否则获得的最小节点永远都是该节点* **this**.**zkClient**.delete(**this**.**selfPath**, -1); *// 删除节点* **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() **throws** Exception { *// 取得了分布式锁之后的目的是要进行具体的业务操作* Thread.*sleep*(200); *// 实现一个延迟处理* System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Thread-"** + **this**.**threadId** + **"获得操作权，执行具体的业务处理操作 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 }  
  
 **public boolean** checkMinPath() **throws** Exception { *// 进行最小节点的判断* List<String> children = **this**.**zkClient**.getChildren(***GROUPNODE***, **false**); *// 取得所有的节点信息* Collections.*sort*(children); *// 进行所有节点的排序，这样最小的节点就排在最上面* **int** index = children  
 .indexOf(**this**.**selfPath**.substring(***GROUPNODE***.length() + 1));  
 **switch** (index) {  
 **case** 0: {  
 **return true**; *// 已经确定好当前的节点为最小节点* }  
 **case** -1: {  
 **return false**; *// 该节点可能已经消失了* }  
 **default**: { *// 表示现在该节点不属于最小节点，那么就需要继续向后排查* **this**.**waitPath** = ***GROUPNODE*** + **"/"** + children.get(index - 1); *// 获得下一个节点* **try** {  
 **this**.**zkClient**.getData(**waitPath**, **true**, **new** Stat()); *// 取得下一个节点的数据* **return false**; *// 本节点不是当前的操作最小节点* } **catch** (Exception e) { *// 如果出现了异常，则表示该节点不存在* **if** (**this**.**zkClient**.exists(**waitPath**, **false**) == **null**) {  
 **return this**.checkMinPath(); *// 继续向后进行检测* } **else** {  
 **throw** e;  
 }  
 }  
 }  
 }  
 }  
  
 **public void** createSubNode() **throws** Exception { *// 每一个线程对象的启动都要求创建有一个节点信息* **this**.**selfPath** = **this**.**zkClient**.create(***SUBNODE***,  
 (**"Thread-"** + **this**.**threadId**).getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***EPHEMERAL\_SEQUENTIAL***); *// 创建临时节点信息* System.***out***.println(  
 **"【Thread-"** + **this**.**threadId** + **"、创建新的临时节点】"** + **this**.**selfPath**);  
 *// 当节点创建完成之后就需要进行最小节点的检测。* **if** (**this**.checkMinPath()) { *// 如果当前的节点为整个项目中的最小节点* **this**.handleSuccess(); *// 进行获得锁后的处理操作* }  
 }  
  
 **public void** connectionZooKeeper() **throws** Exception { *// 连接ZooKeeper服务* **this**.**zkClient** = **new** ZooKeeper(***CONNECTION\_URL***, ***SESSION\_TIMEOUT***,  
 **new** Watcher() {  
 @Override  
 **public void** process(WatchedEvent event) {  
 *// 第一次连接Zookeeper的时候会出现none* **if** (event.getType() == Event.EventType.***None***) {  
 DistributedLock.**this**.**connectLatch**.countDown(); *//表示已经连接成功* } **else** {  
 *// 要处理删除节点操作，并且要确定下一个的节点是已经准备出来的节点信息* **if** (event.getType() == Event.EventType.***NodeChildrenChanged*** && event.getPath().equals(DistributedLock.**this**.**waitPath**)) {  
 **try** {  
 **if** (DistributedLock.**this**.checkMinPath()) { *// 如果当前的节点为整个项目中的最小节点* DistributedLock.**this**.handleSuccess();  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 });  
 **this**.**zkClient**.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权* **if** (**this**.**zkClient**.exists(***GROUPNODE***, **false**) == **null**) { *// 此时根节点不存在* **this**.**zkClient**.create(***GROUPNODE***, **"LOCKDEMO"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  
 }  
 **this**.**connectLatch**.await(); *// 等待连接后才执行后续的功能* }  
  
 **public void** releaseZooKeeper() { *// 进行ZooKeeper的连接释放* **if** (**this**.**zkClient** != **null**) { *// 现在是属于有连接* **try** {  
 **this**.**zkClient**.close(); *// 关闭连接* } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

测试：

**public class** TestCountDownLatch {  
 **public static void** main(String[] args) **throws** Exception {  
 *// 2表示你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();  
 latch.await();  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*程序执行完毕！\*\*\*\*\*\*\*\*\*\*"**);  
 }  
}

### 分布锁操作处理

统计最小节点的逻辑思路：

1. 当某个线程连接到Zookeeper服务之后，那么应该创建有相应的子节点（瞬时节点），当有了找个子节点后应该去判断，当前线程的路径编号信息是否为最小的路径值，如果是则直接去的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。
2. 在判断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。
3. 同时在用户操作完毕之后应该将此节点的信息删除，而后在删除后应该再一次判断是否为最小节点。

**public class** DistributedLock {  
 **public static final** String ***CONNECTION\_URL*** = **"192.168.1.223:2181,192.168.1.222:2181,192.168.1.231:2181"**;  
 **public static final int *SESSION\_TIMEOUT*** = 5000; *// 连接超时时间* **public static final** String ***AUTH\_INFO*** = **"zkuser:yangmi"**; *// 进行连接的授权信息* **public static final** String ***GROUPNODE*** = **"/yangmi-lock"**; *// 根节点* **public static final** String ***SUBNODE*** = ***GROUPNODE*** + **"/lockthread-"**; *// 根节点* **private** CountDownLatch **latch** = **null**;  
 *// 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** = **null**; *// 建立ZooKeeper程序控制类* **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;  
  
 */\*\*  
 \* // 进行一些初始化操作使用  
 \*  
 \** ***@param threadId*** *随意给定的一个编号信息  
 \** ***@param latch*** *进行线程同步处理  
 \** ***@throws*** *Exception ZooKeeper异常  
 \*/* **public** DistributedLock(**int** threadId, CountDownLatch latch)  
 **throws** Exception {  
 **this**.**threadId** = threadId; *// 保存每一个线程对象自己的ID信息* **this**.**latch** = latch;  
 **this**.connectionZooKeeper(); *// 进行节点的连接* }  
  
 **public void** handle() **throws** Exception { *// 具体业务处理* **this**.createSubNode(); *// 创建临时节点操作* }  
  
 **public void** handleSuccess() **throws** Exception { *// 表示取得锁之后进行的处理* **if** (**this**.**zkClient**.exists(**this**.**selfPath**, **false**) == **null**) {  
 **return**; *// 如果当前节点不存在* }  
 **this**.handlCallback(); *// 执行具体的业务操作  
 // 如果某一个节点操作完毕了，那么应该立即删除掉该节点，否则获得的最小节点永远都是该节点* **this**.**zkClient**.delete(**this**.**selfPath**, -1); *// 删除节点* **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() **throws** Exception { *// 取得了分布式锁之后的目的是要进行具体的业务操作* Thread.*sleep*(200); *// 实现一个延迟处理* System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Thread-"** + **this**.**threadId** + **"获得操作权，执行具体的业务处理操作 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 }  
  
 **public boolean** checkMinPath() **throws** Exception { *// 进行最小节点的判断* List<String> children = **this**.**zkClient**.getChildren(***GROUPNODE***, **false**); *// 取得所有的节点信息* Collections.*sort*(children); *// 进行所有节点的排序，这样最小的节点就排在最上面* **int** index = children  
 .indexOf(**this**.**selfPath**.substring(***GROUPNODE***.length() + 1));  
 **switch** (index) {  
 **case** 0: {  
 **return true**; *// 已经确定好当前的节点为最小节点* }  
 **case** -1: {  
 **return false**; *// 该节点可能已经消失了* }  
 **default**: { *// 表示现在该节点不属于最小节点，那么就需要继续向后排查* **this**.**waitPath** = ***GROUPNODE*** + **"/"** + children.get(index - 1); *// 获得下一个节点* **try** {  
 **this**.**zkClient**.getData(**waitPath**, **true**, **new** Stat()); *// 取得下一个节点的数据* **return false**; *// 本节点不是当前的操作最小节点* } **catch** (Exception e) { *// 如果出现了异常，则表示该节点不存在* **if** (**this**.**zkClient**.exists(**waitPath**, **false**) == **null**) {  
 **return this**.checkMinPath(); *// 继续向后进行检测* } **else** {  
 **throw** e;  
 }  
 }  
 }  
 }  
 }  
  
 **public void** createSubNode() **throws** Exception { *// 每一个线程对象的启动都要求创建有一个节点信息* **this**.**selfPath** = **this**.**zkClient**.create(***SUBNODE***,  
 (**"Thread-"** + **this**.**threadId**).getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***EPHEMERAL\_SEQUENTIAL***); *// 创建临时节点信息* System.***out***.println(  
 **"【Thread-"** + **this**.**threadId** + **"、创建新的临时节点】"** + **this**.**selfPath**);  
 *// 当节点创建完成之后就需要进行最小节点的检测。* **if** (**this**.checkMinPath()) { *// 如果当前的节点为整个项目中的最小节点* **this**.handleSuccess(); *// 进行获得锁后的处理操作* }  
 }  
  
 **public void** connectionZooKeeper() **throws** Exception { *// 连接ZooKeeper服务* **this**.**zkClient** = **new** ZooKeeper(***CONNECTION\_URL***, ***SESSION\_TIMEOUT***,  
 **new** Watcher() {  
 @Override  
 **public void** process(WatchedEvent event) {  
 **if** (event.getType() == Event.EventType.***None***) { *// 第一次连接Zookeeper的时候会出现none* DistributedLock.**this**.**connectLatch**.countDown(); *// 表示已经连接成功* } **else** { *// 要处理删除节点操作，并且要确定下一个的节点是已经准备出来的节点信息* **if** (event.getType() == EventType.***NodeDeleted*** && event.getPath().equals(  
 DistributedLock.**this**.**waitPath**)) {  
 **try** {  
 **if** (DistributedLock.**this**.checkMinPath()) { *// 如果当前的节点为整个项目中的最小节点* DistributedLock.**this**.handleSuccess(); *// 进行获得锁后的处理操作* }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 });  
 **this**.**zkClient**.addAuthInfo(**"digest"**, ***AUTH\_INFO***.getBytes()); *// 进行认证授权* **if** (**this**.**zkClient**.exists(***GROUPNODE***, **false**) == **null**) { *// 此时根节点不存在* **this**.**zkClient**.create(***GROUPNODE***, **"LOCKDEMO"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  
 }  
 **this**.**connectLatch**.await(); *// 等待连接后才执行后续的功能* }  
  
 **public void** releaseZooKeeper() { *// 进行ZooKeeper的连接释放* **if** (**this**.**zkClient** != **null**) { *// 现在是属于有连接* **try** {  
 **this**.**zkClient**.close(); *// 关闭连接* } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

测试：

**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** { *// 每一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**.handle(); *// 进行具体的操作处理，直接只进行一个处理方法* } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}  
**public class** TestDistributedLock {  
 **public static void** main(String[] args) **throws** Exception {  
 CountDownLatch latch = **new** CountDownLatch(10) ; *// 现在准备出了10个线程对象* **for** (**int** x = 0; x < 10; x++) {  
 **new** Thread(**new** LockThread(x,latch)).start();  
 }  
 latch.await(); *// 等待所有的线程执行完毕* System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\* 所有的线程对象操作完毕 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 }  
}