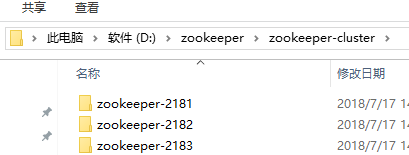
**Zookeeper**

#### Zookeeper集群搭建

1. Windows下
2. 下载zookeeper压缩包

链接：https://pan.baidu.com/s/1r-YGyfbah2orurWIfj-Xyw 密码：hxus

1. 解压到一个目录下，复制三份



1. 打开根目录下conf文件中的zoo.\_sample.cfg文件，修改内容如下，并且修改名字为zoo.cfg，注意clientPort为端口号，不能相同，修改三次。

tickTime=2000

initLimit=10

syncLimit=5

dataDir=D:/zookeeper/zookeeper-cluster/zookeeper-2183/data

dataLogDir=D:/zookeeper/zookeeper-cluster/zookeeper-2183/log

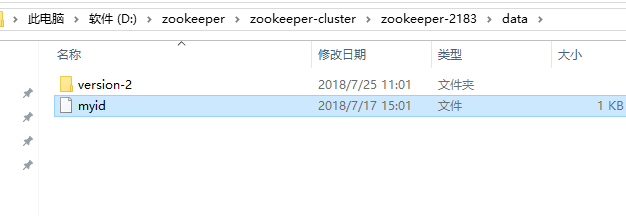
clientPort=2183

server.1=localhost:2887:3887

server.2=localhost:2888:3888

server.3=localhost:2889:3889

1. 在配置文件中的dataDir目录下创建一个myid文件，无后缀，里面编辑一个正整数，与server.id=localhost:2887:3887中的id相同



1. 编辑一个批处理文件start-zkServer.bat启动zookeeper服务

start cmd /c zookeeper-2181\bin\zkServer.cmd

start cmd /c zookeeper-2182\bin\zkServer.cmd

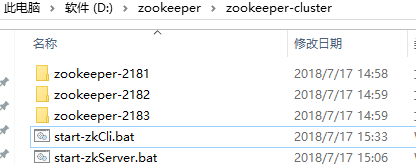
start cmd /c zookeeper-2183\bin\zkServer.cmd

1. 编辑一个批处理文件用于启动zookeeper客户端

start cmd /c zookeeper-2181\bin\zkCli.cmd

start cmd /c zookeeper-2182\bin\zkCli.cmd

start cmd /c zookeeper-2183\bin\zkCli.cmd

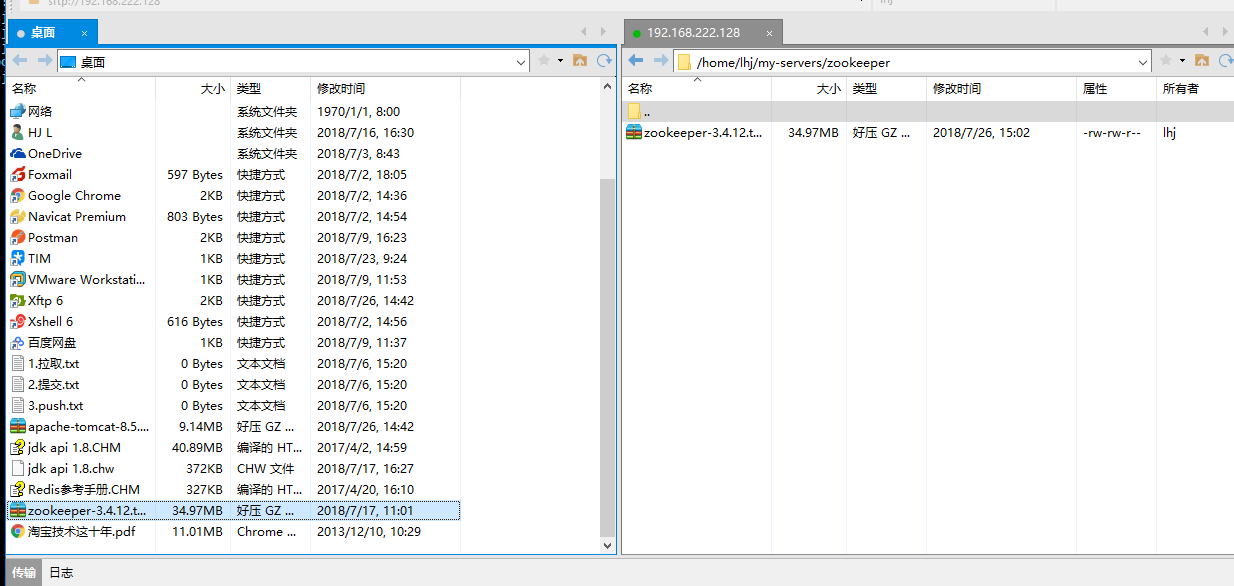


7）双击启动服务->启动客户端

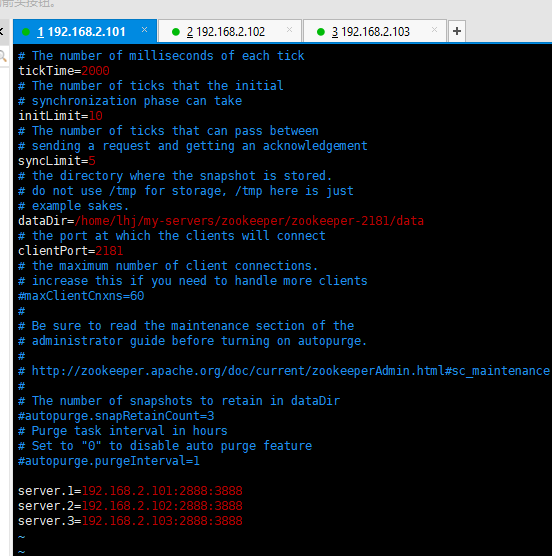
1. Linux下
2. 下载zookeeper压缩文件

链接：https://pan.baidu.com/s/1r-YGyfbah2orurWIfj-Xyw 密码：hxus

1. 传到linux指定目录中



1. 解压并重命名为zookeeper-2181
2. 对zookeeper-2181中conf下的zoo\_sample.cfg文件更名为zoo.cfg,并利用vim命令修改内容如下



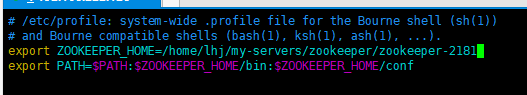
1. 在dataDir的指定目录下创建myid文件，并编辑myid文件内存一个正整数与server.id=ip:port:port中的id对应
2. 配置环境变量

vi etc/profile 在最后添加如下两个。

export ZOOKEEPER\_HOME=/opt/zookeeper-3.4.9

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

保存后退出，输入source /etc/profile命令使修改生效。

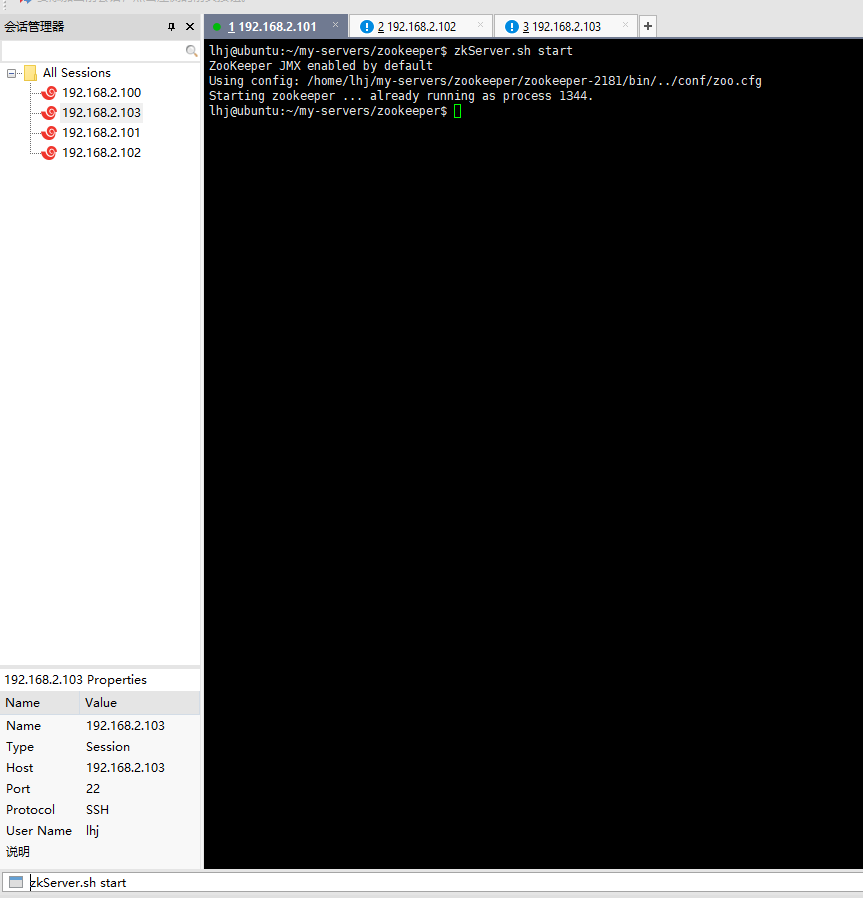


注：若权限不够，可以在root用户下执行

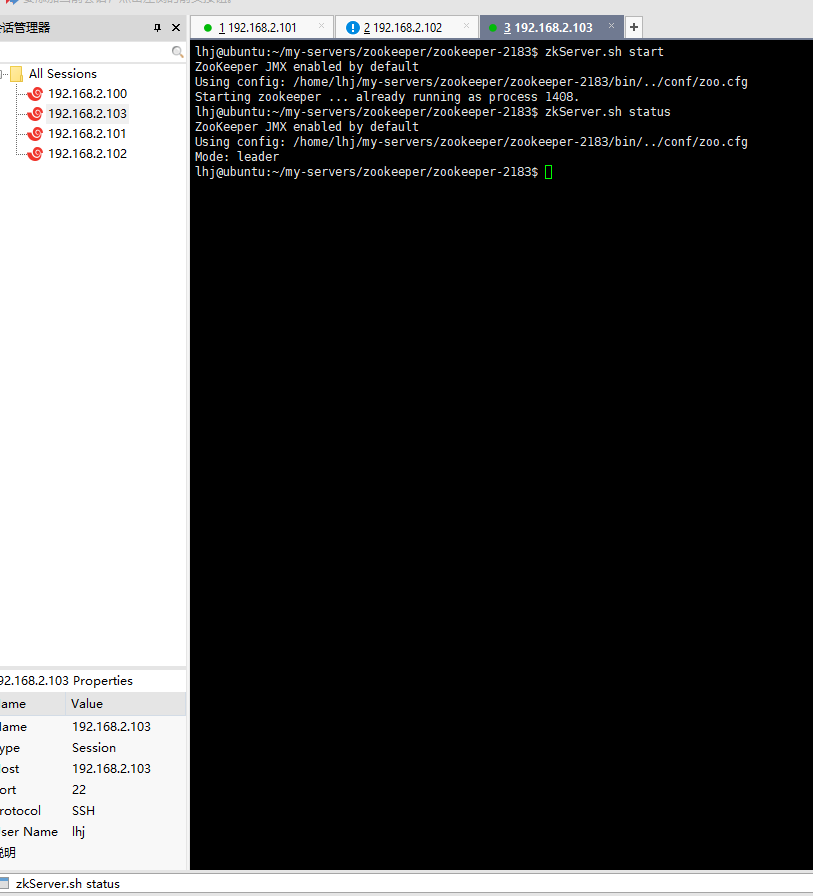
chmod 777 /etc/profile

即可

1. 将上述操作在三个虚拟机中执行三次，注意端口号
2. 启动集群



查看集群状态



1. 注：配置详解

initLimit=10 /\*Follower连接loader初始化时间，超时则退出同步数据过程,重新选举\*/

syncLimit=5 /\* Follower连接loader心跳时间，超时则认为Follower不在线\*/

tickTime=2000 /\*执行时间间隔（不需要修改）\*/

dataDir=/zk/zookeeper/data /\*数据存储目录（根据当前路径进行修改）\*/

clientPort=2181 /\*默认client为2181端口，如果是伪集群，不同zk请改不同端口，这里修改为4181，4182，4183三个端口，在其它应用中都需要使用此端口\*/

server.1=192.168.206.150:2881:3881/\*编号1的zookeeper\*/

server.2=192.168.206.150:2882:3882/\*编号2的zookeeper\*/

server.3=192.168.206.150:2883:3883/\*编号3的zookeeper\*/

最后三个server端口解析：

第一个端口2888是指follower服务器和leader服务器的通信端口，第二个端口3888指的是选举leader服务器时互相通信的端口。

#### Zookeeper原生API

1. 原生API的增删该查

**public class** ZkApiCurd {  
 */\*\*连接地址\*/* **private static final** String ***CONNECT\_ADDR*** = **"127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183"**;  
 */\*\*连接超时时间\*/* **private static final int *CONNECT\_TIME*** = 20000;  
 */\*\*信号量，用于阻塞直到连接成功\*/* **private static final** CountDownLatch ***connectSemaphore*** = **new** CountDownLatch(1);  
 **public static void** main(String[] args) **throws** Exception{  
 ZooKeeper zk = **new** ZooKeeper(***CONNECT\_ADDR***, ***CONNECT\_TIME***, **new** Watcher() {  
 @Override  
 **public void** process(WatchedEvent watchedEvent) {  
 *//获取时间的状态* Event.KeeperState state = watchedEvent.getState();  
 Event.EventType type = watchedEvent.getType();  
 *//如果建立连接* **if**(Event.KeeperState.***SyncConnected***==state){  
 **if**(Event.EventType.***None***==type){  
 ***connectSemaphore***.countDown();  
 System.***out***.println(**"zk连接成功......"**);  
 }  
 }  
 }  
 });  
 ***connectSemaphore***.await();  
 *//创建父节点* String firstNode = zk.create(  
 **"/zkApi"**,  
 **"first persistenet node"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,  
 CreateMode.***PERSISTENT\_SEQUENTIAL*** );  
 System.***out***.println(**new** String(firstNode+**"创建成功!!!"**));  
 *//创建子节点* String children = zk.create(  
 firstNode+**"/children"**,  
 **"this is a children"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,  
 CreateMode.***EPHEMERAL*** );  
 System.***out***.println(**new** String(children+**"创建成功!!!"**));  
 String children1 = zk.create(  
 firstNode+**"/children1"**,  
 **"this is a children1"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,  
 CreateMode.***EPHEMERAL*** );  
 System.***out***.println(**new** String(children1+**"创建成功!!!"**));  
 *//获取父节点* **byte**[] bytes = zk.getData(firstNode ,**false**,**null**);  
 System.***out***.println(**"获得新建父节点内容:"**+**new** String(bytes));  
 *//获取子节点* **byte**[] childrenBytes = zk.getData(children,**false**,**null**);  
 System.***out***.println(**"获得新建子节点内容:"**+**new** String(childrenBytes));  
 List<String> list = zk.getChildren(firstNode,**false**);  
 **for**(String str : list){  
 System.***out***.println(**"---"**+str+**"---"**);  
 }  
 *//修改节点的值，-1表示跳过版本检查，其他正数表示如果传入的版本号与当前版本号不一致，则修改不成功，删除是同样的道理。* zk.setData(children,**"this is a children ---second"**.getBytes(),-1);  
 **byte**[] childrenBytes2 = zk.getData(children,**false**,**null**);  
 System.***out***.println(**"获得修改子点内容:"**+**new** String(childrenBytes2));  
 *//删除节点* zk.delete(children1,-1);  
 System.***out***.println(zk.exists(children1,**false**));  
 zk.delete(children,-1);  
 System.***out***.println(zk.exists(children,**false**));  
 zk.delete(firstNode,-1);  
 System.***out***.println(zk.exists(firstNode,**false**));  
 zk.close();  
 System.***out***.println(zk.getState().isConnected());  
  
 }  
}

1. 控制时序的锁

**public class** ZkApiTimeLock {  
 */\*\*连接地址\*/* **private static final** String ***CONNECT\_ADDR*** = **"127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183"**;  
 */\*\*超时时间\*/* **private static final int *CONNECT\_TIME*** = 200000;  
 */\*\*信号量\*/* **private static final** CountDownLatch ***connectSemaphore*** = **new** CountDownLatch(1);  
 **private static final** CountDownLatch ***latch*** = **new** CountDownLatch(1);  
 */\*\*锁根目录\*/* **private static final** String ***root*** = **"/lock"**;  
 */\*\*新建的lock\*/* **private** String **lockid**;  
 */\*\*zookeeper客户端\*/* ZooKeeper **zk** = **null**;  
  
 **public** ZkApiTimeLock(){  
 **try**{  
 **zk** = **new** ZooKeeper(***CONNECT\_ADDR***, ***CONNECT\_TIME***,  
 **new** Watcher() {  
 @Override  
 **public void** process(WatchedEvent event) {  
 Event.KeeperState state = event.getState();  
 Event.EventType type = event.getType();  
 **if**(Event.KeeperState.***SyncConnected***==state){  
 **if**(Event.EventType.***None***==type){  
 System.***out***.println(**"连接成功...."**);  
 ***connectSemaphore***.countDown();  
 }  
 }  
 }  
 });  
 System.***out***.println(**"开始连接服务器...."**);  
 ***connectSemaphore***.await();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
  
 **public boolean** getLock(){  
 **try**{  
 *//创建锁节点* **lockid** = **zk**.create(  
 ***root***+**"/"**,  
 **"lock哒哒哒"**.getBytes(),  
 ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,  
 CreateMode.***EPHEMERAL\_SEQUENTIAL*** );  
 System.***out***.println(Thread.*currentThread*().getName()+**"创建锁节点:"**+**lockid**+**"------------"**);  
 *//获得所有的锁节点并排序* List<String> list = **zk**.getChildren(***root***,**true**);  
 TreeSet<String> set = **new** TreeSet<String>();  
 **for**(String node : list){  
 set.add(***root***+**"/"**+node);  
 }  
 SortedSet<String> lessThanMe = set.headSet(**lockid**);  
 *//判断当前新建节点是否是最小节点* **if**(**lockid**.equals(set.first())){  
 System.***out***.println(Thread.*currentThread*().getName()+**"获得锁:"**+**lockid**+**"!!!!!!!!!!"**);  
 **return true**;  
 }**else if**(!lessThanMe.isEmpty()){  
 String preLockId = lessThanMe.last();  
 **zk**.exists(preLockId, **new** Watcher() {  
 @Override  
 **public void** process(WatchedEvent event) {  
 **if**(Event.EventType.***NodeDeleted***==event.getType()){  
 ***latch***.countDown();  
 }  
 }  
 });  
 ***latch***.await();  
 System.***out***.println(Thread.*currentThread*().getName()+**"已经获得锁:"**+**lockid**);  
 **return true**;  
 }  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 **return false**;  
 }  
  
 **public boolean** unlock(){  
 **try**{  
 System.***out***.println(Thread.*currentThread*().getName()+**"释放锁:"**+**lockid**);  
 **zk**.delete(**lockid**,-1);  
 **return true**;  
 }**catch**(Exception e){  
 e.printStackTrace();  
 }**finally** {  
 **try**{  
 **zk**.close();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
 **return false**;  
 }  
  
 **public static void** main(String[] args){  
  
 *//设置核心线程池大小* **int** corePoolSize = 10;  
 *//设置最大接受数* **int** maximumPoolSiez = 12;  
 *//当前线程数大于corePoolSize、小于maximumPoolSize时，超出corePoolSize的线程数的生命周期* **long** keepActiveTime = 200;  
 *//设置时间单位，秒* TimeUnit timeUnit = TimeUnit.***SECONDS***;  
 *//设置工作队列* BlockingQueue<Runnable> workQueue = **new** ArrayBlockingQueue<Runnable>(5);  
  
 ThreadPoolExecutor executor = **new** ThreadPoolExecutor(corePoolSize,maximumPoolSiez,keepActiveTime,timeUnit,workQueue);  
  
 **for**(**int** i=0;i<10;i++){  
 executor.submit(**new** Runnable() {  
 @Override  
 **public void** run() {  
 ZkApiTimeLock lock = **null**;  
 **try** {  
 lock = **new** ZkApiTimeLock();  
 lock.getLock();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }**finally** {  
 **if**(lock!=**null**){  
 lock.unlock();  
 }  
 }  
 }  
 });  
 }  
 }  
}

1. 原生API的watcher

**public class** ZkApiWatcher **implements** Watcher {  
 */\*\*连接地址\*/* **private static final** String ***CONNECT\_ADDR*** = **"127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183"**;  
 */\*\*超时时间\*/* **private static final int *CONNECT\_TIME*** = 200000;  
 */\*\*信号量\*/* **private static final** CountDownLatch ***connectSemaphore*** = **new** CountDownLatch(1);  
 **private** ZooKeeper **zk** = **null**;  
 */\*\*  
 \* 创建连接  
 \** ***@param addr*** *:连接地址  
 \** ***@param timeOut*** *：超时时间  
 \*/* **public void** createConnection(String addr,**int** timeOut){  
 **this**.releaseConnection();  
 **try**{  
 **zk** = **new** ZooKeeper(  
 addr,  
 timeOut,  
 **this** );  
 System.***out***.println(**"开始连接zk服务器...."**);  
 ***connectSemaphore***.await();  
 }**catch**(Exception e){  
 e.printStackTrace();  
 }  
 }  
 */\*\*  
 \* 释放连接  
 \*/* **public void** releaseConnection(){  
 **if**(**zk**!=**null**){  
 **try**{  
 **this**.**zk**.close();  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 }  
 }  
 */\*\*  
 \* 创建节点  
 \** ***@param path*** *\** ***@param data*** *\** ***@return*** *\*/* **public** String createNode(String path,String data){  
 **try**{  
 **zk**.exists(path,**true**);  
 String node = **zk**.create(path,data.getBytes(), ZooDefs.Ids.***OPEN\_ACL\_UNSAFE***,CreateMode.***PERSISTENT***);  
 System.***out***.println(**"创建节点:"**+node+**"---------------"**);  
 **return** node;  
 }**catch**(Exception e){  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
 */\*\*  
 \* 删除节点  
 \** ***@param path*** *\** ***@return*** *\*/* **public boolean** deleteNode(String path){  
 **try**{  
 **zk**.delete(path,-1);  
 System.***out***.println(**"删除节点:"**+path+**"---------------"**);  
 **return true**;  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 **return false**;  
 }  
 */\*\*  
 \* 获得节点  
 \** ***@param path*** *\** ***@param needWatch*** *\** ***@return*** *\*/* **public** String getNode(String path,**boolean** needWatch){  
 **try**{  
 String data = **new** String(**zk**.getData(path,needWatch,**null**));  
 **return** data;  
 }**catch**(Exception e){  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
 */\*\*  
 \* 修改节点  
 \** ***@param path*** *\** ***@param data*** *\** ***@return*** *\*/* **public** Stat setNode(String path,String data){  
 **try**{  
 Stat stat = **zk**.setData(path,data.getBytes(),-1);  
 System.***out***.println(**"修改节点:"**+path+**"---------------"**);  
 **return** stat;  
 }**catch**(Exception e){  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
 */\*\*  
 \* 获得子节点  
 \** ***@param path*** *\** ***@return*** *\*/* **public** List<String> getChildren(String path,**boolean** needWatch){  
 **try**{  
 **return zk**.getChildren(path,needWatch);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
 */\*\*  
 \* 判断节点是否存在  
 \** ***@param path*** *\** ***@return*** *\*/* **public** Stat existsNode(String path,**boolean** needWatch){  
 **try**{  
 **return zk**.exists(path,needWatch);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
 @Override  
 **public void** process(WatchedEvent event) {  
 System.***out***.println(**"进取process----------event = "**+event);  
 **try**{  
 Thread.*sleep*(200);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }  
 **if**(event==**null**){  
 **return** ;  
 }  
 *//连接状态* Event.KeeperState state = event.getState();  
 *//事件类型* Event.EventType type = event.getType();  
 *//受影响的路径* String path = event.getPath();  
 System.***out***.println(**"收到watch通知"**);  
 System.***out***.println(**"连接状态:"**+state);  
 System.***out***.println(**"事件类型:"**+type);  
 **if**(Event.KeeperState.***SyncConnected***==state){  
 **if**(Event.EventType.***None***==type){  
 System.***out***.println(**"成功连接上服务器...."**);  
 ***connectSemaphore***.countDown();  
 }  
 **else if** (Event.EventType.***NodeDataChanged***==type){  
 System.***out***.println(**"节点数据变更++++++++++"**);  
 **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 **else if**(Event.EventType.***NodeCreated***==type){  
 System.***out***.println(**"创建节点++++++++++"**);  
 **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 **else if**(Event.EventType.***NodeDeleted***==type){  
 System.***out***.println(**"删除节点++++++++++"**);  
 **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 **else if**(Event.EventType.***NodeDataChanged***==type){  
 System.***out***.println(**"子节点数据变更++++++++++"**);  
 **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }**else if** (Event.KeeperState.***Disconnected***==state){  
 System.***out***.println(**"服务器断开连接+++++++++"**);  
 }**else if** (Event.KeeperState.***AuthFailed***==state){  
 System.***out***.println(**"权限检查失败+++++++++"**);  
 }**else if**(Event.KeeperState.***Expired***==state){  
 System.***out***.println(**"会话失效+++++++++++"**);  
 }  
 System.***out***.println(**"++++++++++++++++++++++"**);  
 }  
 **public static void** main(String[] args) **throws** Exception{  
 ZkApiWatcher zk = **new** ZkApiWatcher();  
 zk.createConnection(***CONNECT\_ADDR***,***CONNECT\_TIME***);  
 Thread.*sleep*(1000);  
 zk.createNode(**"/zkApi"**,**"newNode"**);  
 Thread.*sleep*(300);  
 zk.getNode(**"/zkApi"**,**true**);  
 zk.setNode(**"/zkApi"**,**"secondNode"**);  
 Thread.*sleep*(300);  
 String data = zk.getNode(**"/zkApi"**,**true**);  
 System.***out***.println(data);  
 zk.deleteNode(**"/zkApi"**);  
 Thread.*sleep*(300);  
 zk.createNode(**"/lock"**,**"root lock"**);  
 }  
}

#### Zookeeper的curator框架

1. Curator框架的增删改查

**public class** ZkCuratorCurd {  
 **private static final** String ***CONNECT\_ADDR*** = **"127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183"**;  
 **private static final int *TIME\_OUT*** = 20000;  
 **public static void** main(String[] args )**throws** Exception{  
 *//重试策略，初始事件1秒，重试10次* RetryPolicy policy = **new** ExponentialBackoffRetry(1000,10);  
 *//通过工厂创建curator客户端* CuratorFramework curator = CuratorFrameworkFactory  
 .*builder*()  
 .connectString(***CONNECT\_ADDR***)  
 .connectionTimeoutMs(***TIME\_OUT***)  
 .retryPolicy(policy)  
 .build();  
 curator.start();  
 *//创建节点，curator使用fluent风格* curator.create()  
 *//创建不存在的父节点* .creatingParentsIfNeeded()  
 .withMode(CreateMode.***PERSISTENT***)  
 *//添加回调* .inBackground((framework,event)->{  
 System.***out***.println(**"Code:"**+event.getResultCode());  
 System.***out***.println(**"Type:"**+event.getType());  
 System.***out***.println(**"Path:"**+event.getPath());  
 })  
 .forPath(**"/curator/parent"**,**"this is parent"**.getBytes());  
 *//为了能够看到回调信息* Thread.*sleep*(5000);  
 *//获得节点数据* String parentData = **new** String(curator.getData().forPath(**"/curator/parent"**));  
 System.***out***.println(**"parent:"**+parentData);  
 *//判断节点是否存在* Stat stat = curator.checkExists().forPath(**"/curator/parent"**);  
 System.***out***.println(stat);  
 *//更新节点* curator.setData().forPath(**"/curator/parent"**,**"this is new parent!!!"**.getBytes());  
 System.***out***.println(**new** String(curator.getData().forPath(**"/curator/parent"**)));  
 *//删除节点* curator.delete()  
 *//安全的删除节点* .guaranteed()  
 *//删除孩子节点* .deletingChildrenIfNeeded()  
 .forPath(**"/curator"**);  
 System.***out***.println(curator.checkExists().forPath(**"/curator/parent"**));  
 System.***out***.println(curator.checkExists().forPath(**"/curator"**));  
 }  
}

1. Curator的watcher

**public class** ZkCuratorWathcer {  
 */\*\*连接地址\*/* **private static final** String ***CONNECT\_ADDR*** = **"127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183"**;  
 */\*\*超时时间\*/* **private static final int *CONNECT\_TIME*** = 200000;  
 **public static void** main(String[] args) **throws** Exception{  
 RetryPolicy policy = **new** ExponentialBackoffRetry(1000,10);  
 CuratorFramework curator = CuratorFrameworkFactory  
 .*builder*()  
 .connectString(***CONNECT\_ADDR***)  
 .connectionTimeoutMs(***CONNECT\_TIME***)  
 .retryPolicy(policy)  
 .build();  
 curator.start();  
 */\*\*  
 \* NodeCache监听节点的新增与修改操作  
 \* 最后一个参数表示是否进行压缩  
 \*/* NodeCache nodeCache = **new** NodeCache(curator,**"/super"**,**false**);  
 nodeCache.start(**true**);  
 *//添加监听事件* nodeCache.getListenable().addListener(() ->{  
 System.***out***.println(**"Data:"**+**new** String(nodeCache.getCurrentData().getData()));  
 System.***out***.println(**"Path:"**+nodeCache.getCurrentData().getPath());  
 System.***out***.println(**"Stat:"**+nodeCache.getCurrentData().getStat());  
 });  
 curator.create().forPath(**"/super"**,**"this is super"**.getBytes());  
 curator.setData().forPath(**"/super"**,**"this is new super"**.getBytes());  
 curator.delete().guaranteed().forPath(**"/super"**);  
 System.***out***.println(**"------------------------------nodeCache------------------------------"**);  
 */\*\*  
 \* PathChildrenCache：监听子节点的新增、修改、删除操作。  
 \*第三个参数表示是否接收节点数据内容  
 \*/* PathChildrenCache childrenCache = **new** PathChildrenCache(curator,**"/super"**,**true**);  
 */\*\*  
 \* 如果不填写这个参数，则无法监听到子节点的数据更新  
 如果参数为PathChildrenCache.StartMode.BUILD\_INITIAL\_CACHE，则会预先创建之前指定的/super节点  
 如果参数为PathChildrenCache.StartMode.POST\_INITIALIZED\_EVENT，效果与BUILD\_INITIAL\_CACHE相同，只是不会预先创建/super节点  
 如果参数为PathChildrenCache.StartMode.NORMAL时，与不填写参数是同样的效果，不会监听子节点的数据更新操作  
 \*/* childrenCache.start(PathChildrenCache.StartMode.***POST\_INITIALIZED\_EVENT***);  
 childrenCache.getListenable().addListener((framework,event) -> {  
 **switch** (event.getType()){  
 **case *CHILD\_ADDED***:  
 System.***out***.println(**"-----------新增节点---------"**);  
 System.***out***.println(**"节点路径："**+event.getData().getPath());  
 System.***out***.println(**"节点数据："**+**new** String(event.getData().getData()));  
 System.***out***.println(**"节点状态："**+event.getData().getStat());  
 System.***out***.println(**"节点类型："**+event.getType());  
 System.***out***.println(**"----------------------------"**);  
 **break**;  
 **case *CHILD\_UPDATED***:  
 System.***out***.println(**"-----------修改节点---------"**);  
 System.***out***.println(**"节点路径："**+event.getData().getPath());  
 System.***out***.println(**"节点数据："**+**new** String(event.getData().getData()));  
 System.***out***.println(**"节点状态："**+event.getData().getStat());  
 System.***out***.println(**"节点类型："**+event.getType());  
 System.***out***.println(**"----------------------------"**);  
 **break**;  
 **case *CHILD\_REMOVED***:  
 System.***out***.println(**"-----------删除节点---------"**);  
 System.***out***.println(**"节点路径："**+event.getData().getPath());  
 System.***out***.println(**"节点数据："**+**new** String(event.getData().getData()));  
 System.***out***.println(**"节点状态："**+event.getData().getStat());  
 System.***out***.println(**"节点类型："**+event.getType());  
 System.***out***.println(**"----------------------------"**);  
 **break**;  
 **default**:  
 }  
 });  
  
 curator.create().forPath(**"/super"**,**"this is super"**.getBytes());  
 curator.create().creatingParentsIfNeeded().forPath(**"/super/children"**,**"this is children"**.getBytes());  
 curator.setData().forPath(**"/super"**,**"this is new super"**.getBytes());  
 curator.setData().forPath(**"/super/children"**,**"this is new children"**.getBytes());  
 curator.delete().guaranteed().deletingChildrenIfNeeded().forPath(**"/super"**);  
 System.***out***.println(**"------------------------------pathChildrenCache------------------------------"**);  
 Thread.*sleep*(500);  
 */\*\*  
 \* TreeCache：既可以监听节点的状态，又可以监听子节点的状态。类似于上面两种Cache的组合  
 \*/* TreeCache treeCache = **new** TreeCache(curator,**"/super"**);  
 treeCache.start();  
 treeCache.getListenable().addListener((framework,event) ->{  
 **switch** (event.getType()){  
 **case *NODE\_ADDED***:  
 System.***out***.println(**"-----------新增节点---------"**);  
 System.***out***.println(**"节点路径："**+event.getData().getPath());  
 System.***out***.println(**"节点数据："**+**new** String(event.getData().getData()));  
 System.***out***.println(**"节点状态："**+event.getData().getStat());  
 System.***out***.println(**"节点类型："**+event.getType());  
 System.***out***.println(**"----------------------------"**);  
 **break**;  
 **case *NODE\_UPDATED***:  
 System.***out***.println(**"-----------修改节点---------"**);  
 System.***out***.println(**"节点路径："**+event.getData().getPath());  
 System.***out***.println(**"节点数据："**+**new** String(event.getData().getData()));  
 System.***out***.println(**"节点状态："**+event.getData().getStat());  
 System.***out***.println(**"节点类型："**+event.getType());  
 System.***out***.println(**"----------------------------"**);  
 **break**;  
 **case *NODE\_REMOVED***:  
 System.***out***.println(**"-----------删除节点---------"**);  
 System.***out***.println(**"节点路径："**+event.getData().getPath());  
 System.***out***.println(**"节点数据："**+**new** String(event.getData().getData()));  
 System.***out***.println(**"节点状态："**+event.getData().getStat());  
 System.***out***.println(**"节点类型："**+event.getType());  
 System.***out***.println(**"----------------------------"**);  
 **break**;  
 **default**:  
 }  
 });  
 curator.create().forPath(**"/super"**,**"this is super"**.getBytes());  
 curator.create().creatingParentsIfNeeded().forPath(**"/super/children"**,**"this is children"**.getBytes());  
 curator.setData().forPath(**"/super"**,**"this is new super"**.getBytes());  
 curator.setData().forPath(**"/super/children"**,**"this is new children"**.getBytes());  
 curator.delete().guaranteed().deletingChildrenIfNeeded().forPath(**"/super"**);  
 System.***out***.println(**"------------------------------treeCache------------------------------"**);  
 Thread.*sleep*(500);  
  
 }  
  
}

1. Curator的可重入锁-----InterProcessMutex

**public class** ZkInterProcessMutex {  
 **private static final int *QTY*** = 5;  
 **private static final int *REPETITIONS*** = ***QTY*** \* 10;  
 **private static final** String ***PATH*** = **"/examples/locks"**;  
 **private static final** String ***CONNECT\_ADDR*** = **"127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183"**;  
 **public static void** main(String[] args) **throws** Exception {  
 **final** FakeLimitedResource resource = **new** FakeLimitedResource();  
 ExecutorService executor = Executors.*newFixedThreadPool*(***QTY***);  
 **try** {  
 **for**(**int** i=0; i<***QTY***; i++) {  
 **final int** index = i;  
 Callable<Void> task = () -> {  
 CuratorFramework curator = CuratorFrameworkFactory  
 .*newClient*(***CONNECT\_ADDR***, **new** RetryNTimes(3, 1000));  
 curator.start();  
 **try** {  
 **final** ExampleClientThatLocks example = **new** ExampleClientThatLocks(curator, ***PATH***, resource, **"Client "** + index);  
 **for**(**int** j=0; j<***REPETITIONS***; j++) {  
 example.doWork(10, TimeUnit.***SECONDS***);  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 CloseableUtils.*closeQuietly*(curator);  
 }  
 **return null**;  
 };  
 executor.submit(task);  
 }  
 executor.shutdown();  
 executor.awaitTermination(10, TimeUnit.***MINUTES***);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}  
*/\*\*  
 \* 模拟的公共资源，这个资源期望只能单线程的访问，否则会有并发问题。  
 \*/***class** FakeLimitedResource {  
 **private final** AtomicBoolean **inUse** = **new** AtomicBoolean(**false**);  
 **public void** use() **throws** Exception{  
 **if**(!**inUse**.compareAndSet(**false**,**true**)){  
 **throw new** IllegalStateException(**"一次只能一个客户端访问"**);  
 }  
 **try** {  
 Thread.*sleep*((**long**) (3 \* Math.*random*()));  
 } **finally** {  
 **inUse**.set(**false**);  
 }  
 }  
}  
  
*/\*\*  
 \* 负责请求锁、使用资源、释放锁这样一个完整的访问过程。  
 \*/***class** ExampleClientThatLocks {  
 **private final** InterProcessMutex **lock**;  
 **private final** FakeLimitedResource **resource**;  
 **private final** String **clientName**;  
 **public** ExampleClientThatLocks(CuratorFramework framework,String path, FakeLimitedResource resource, String clientName) {  
 **this**.**lock** = **new** InterProcessMutex(framework,path);  
 **this**.**resource** = resource;  
 **this**.**clientName** = clientName;  
 }  
 **public void** doWork(**long** time, TimeUnit timeUnit) **throws** Exception{  
 **if**(!**lock**.acquire(time,timeUnit)){  
 **throw new** IllegalStateException(**clientName** +**"不能获取锁"**);  
 }  
 System.***out***.println(**clientName**+**"获得了锁----------"**);  
 **try**{  
 **resource**.use();  
 }**finally** {  
 System.***out***.println(**clientName**+**"释放了锁"**);  
 **lock**.release();  
 }  
 }  
}

1. Curator的其他应用

1）分布式锁

可重入锁：InterProcessMutex

不可重入锁：InterProcessSemaphoreMutex

可重入读写锁：InterProcessReadWriteLock

信号量：InterProcessSemaphoreV2

多锁对象：InterProcessMultiLock

2）Watcher实现类

监听之--- NodeCache

监听之--- PathChildrenCache

监听之--- TreeCache

3）分布式计数器

分布式计数器之--- DistributedAtomicInteger

4）Barrier（栅栏）

栅栏：DistributedBarrier

双栅栏：DistributedDoubleBarrier