Zookeeper

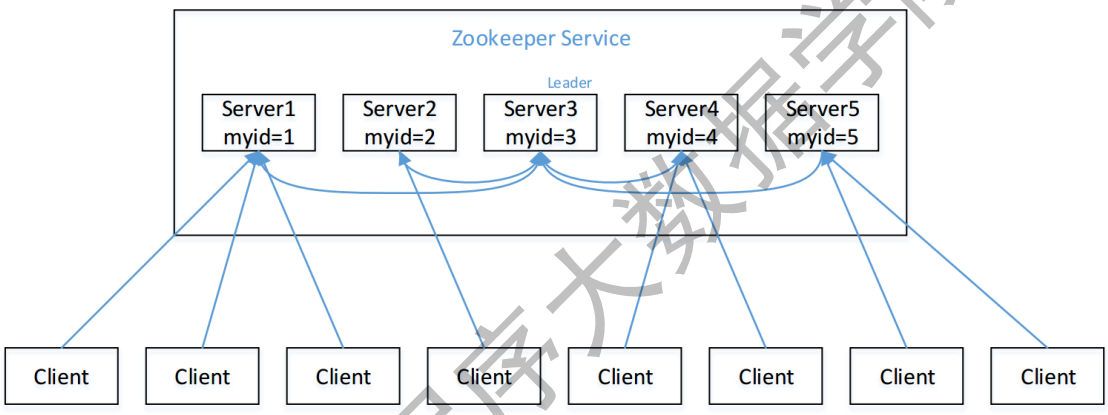
一、Zookeeper概述

1.1 概述

Zookeeper 是一个开源的分布式的，为分布式应用提供协调服务的 Apache 项目

zookeeper从设计模式角度来理解，是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的注册，一旦这些数据的状态发生变化，Zookeeper就将负责通知已经 在Zookeeper上注册的那些观察者做出相应的反应，从而实现集群中类似Master/Slave管理模式

1.2 zookeeper的特点



1）Zookeeper：一个领导者（leader），多个跟随者（follower）组成的集群

2）Leader 负责进行投票的发起和决议，更新系统状态。

3）Follower 用于接收客户请求并向客户端返回结果，在选举 Leader 过程中参与投票。

4）集群中只要有半数以上节点存活， Zookeeper 集群就能正常服务。

5）全局数据一致：每个 server 保存一份相同的数据副本， client 无论连接到哪个 server，数据都是一致的。

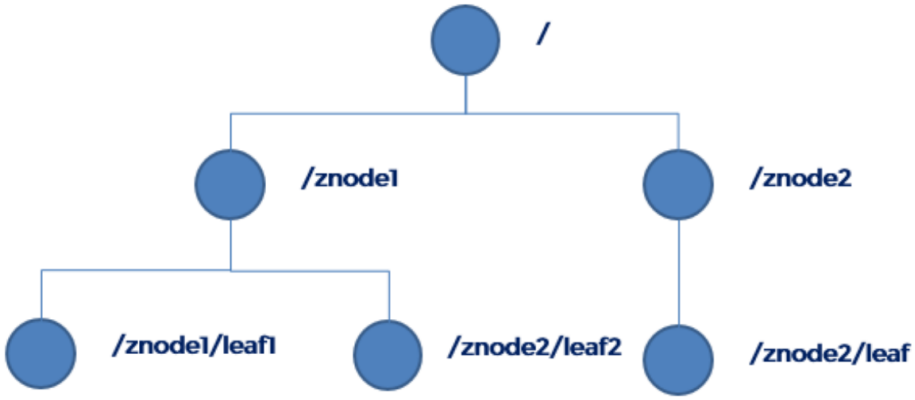
6）更新请求顺序进行，来自同一个 client 的更新请求按其发送顺序依次执行。

7）数据更新原子性，一次数据更新要么成功，要么失败。

8）实时性，在一定时间范围内， client 能读到最新数据

1.3 Zookeeper数据结构

ZooKeeper 数据模型的结构与 Unix 文件系统很类似，整体上可以看作是一棵树，每个节点称做一个 ZNode。每一个 ZNode 默认能够存储 1MB 的数据，每个 ZNode 都可以通过其路径唯一标识。



数据结构图

1.4 Zookeeper的应用场景

提供的服务包括： 统一命名服务、统一配置管理、统一集群管理、服务器节点动态上下线、 软负载均衡等。

二、Zookeeper安装

1 ）在 hadoop01、 hadoop02 和 hadoop03 三个节点上部署 Zookeeper。

2) 解压安装

1 解压Zookeeper安装包到/opt/app目录下

tar -zxvf zookeeper-3.4.10.tar.gz -C /opt/app/

2 在/opt/app/zookeeper-3.4.10/这个目录下创建 zkData

mkdir -p zkData

3 重命名/opt/app/zookeeper-3.4.10/conf 这个目录下的 zoo_sample.cfg 为 zoo.cfg

mv zoo_sample.cfg zoo.cfg

3) 配置 zoo.cfg 文件

1 具体配置

dataDir=/opt/module/zookeeper-3.4.10/zkData

增加如下配置

server.1=hadoop01:2888:3888

server.2=hadoop02:2888:3888
server.3=hadoop03:2888:3888

2 配置参数解读

Server.A=B:C:D

A 是一个数字，表示这个是第几号服务器；
B 是这个服务器的 ip 地址；
C 是这个服务器与集群中的 Leader 服务器交换信息的端口；
D 是万一集群中的 Leader 服务器挂了，需要一个端口来重新进行选举，选出一个新的Leader，而这个端口就是用来执行选举时服务器相互通信的端口。
集群模式下配置一个文件 myid，这个文件在 dataDir目录下，这个文件里面有一个数据就是 A 的值，Zookeeper 启动时读取此文件，拿到里面的数据与 zoo.cfg 里面的配置信息比 较从而判断到底是哪个 server

3 集群操作

1)在/opt/app/zookeeper-3.4.10/zkData 目录下创建一个 myid 的文件
touch myid

2) 编辑myid

在文件中添加与 server 对应的编号：如 1

3)拷贝配置好的 zookeeper 到其他机器上

scp -r zookeeper-3.4.10/ root@hadoop01:/opt/app/
scp -r zookeeper-3.4.10/ root@hadoop02:/opt/app/

3) 分别启动zookeeper

bin/zkServer.sh start

4)查看状态

bin/zkServer.sh status

三、客户端命令行操作

命令行工具的一些常用操作命令如下：
1.ls -- 查看某个目录包含的所有文件，例如：
[zk: 127.0.0.1:2181(CONNECTED) 1] ls /

2.ls2 -- 查看某个目录包含的所有文件，与ls不同的是它查看到time、version等信息，例如：
[zk: 127.0.0.1:2181(CONNECTED) 1] ls2 /

3.create -- 创建znode，并设置初始内容，例如：
[zk: 127.0.0.1:2181(CONNECTED) 1] create /test "test"
Created /test

创建一个新的 znode节点“ test ”以及与它关联的字符串

4.get -- 获取znode的数据，如下：
[zk: 127.0.0.1:2181(CONNECTED) 1] get /test

5.set -- 修改znode内容，例如：
[zk: 127.0.0.1:2181(CONNECTED) 1] set /test "ricky"

6.delete -- 删除znode，例如：
[zk: 127.0.0.1:2181(CONNECTED) 1] delete /test

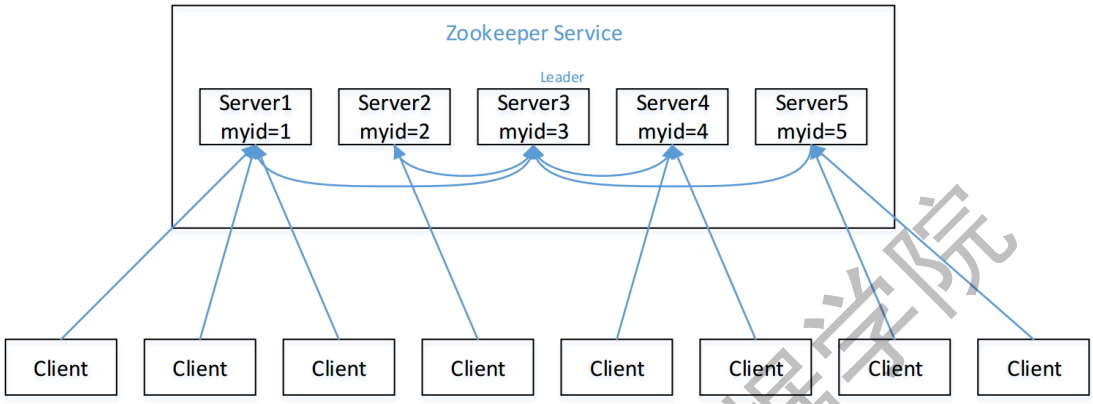
7.quit -- 退出客户端

8.help -- 帮助命令

四、Zookeeper内部原理

4.1 选举机制

1）半数机制（Paxos 协议）：集群中半数以上机器存活，集群可用。所以 zookeeper 适合装在奇数台机器上。
2）Zookeeper 虽然在配置文件中并没有指定 master 和 slave。 但是， zookeeper 工作时，是有一个节点为 leader，其他则为 follower， Leader 是通过内部的选举机制临时产生的。
3）以一个简单的例子来说明整个选举的过程。假设有五台服务器组成的 zookeeper 集群，它们的 id 从 1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来
看看会发生什么。



（1）服务器 1 启动，此时只有它一台服务器启动了，它发出去的报没有任何响应，所以它的选举状态一直是 LOOKING 状态。
（2）服务器 2 启动，它与最开始启动的服务器 1 进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以 id 值较大的服务器 2 胜出，但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是 3)，所以服务器 1、 2 还是继续保持 LOOKING 状态。
（3）服务器 3 启动，根据前面的理论分析，服务器 3 成为服务器 1、 2、 3 中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的 leader。
（4）服务器 4 启动，根据前面的分析，理论上服务器 4 应该是服务器 1、 2、 3、 4 中最大的，但是由于前面已经有半数以上的服务器选举了服务器 3，所以它只能接收当小弟的命了。
（5）服务器 5 启动，同 4 一样当小弟

4.2 节点类型

Znode 有两种类型：

短暂（ephemeral）：客户端和服务器端断开连接后，创建的节点自己删除
持久（persistent）：客户端和服务器端断开连接后，创建的节点不删除

1）Znode 有四种形式的目录节点（默认是 persistent ）

客户端与 zookeeper 断开连接后， 该节点依旧存在。

2）持久化顺序编号目录节点（PERSISTENT_SEQUENTIAL）

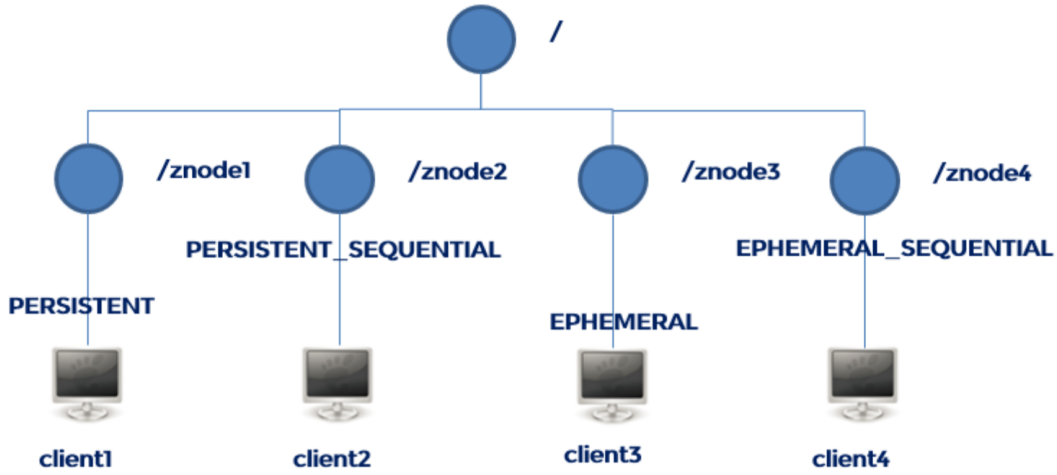
客户端与 zookeeper 断开连接后，该节点依旧存在，只是 Zookeeper 给该节点名称进行顺序编号。

3）临时目录节点（EPHEMERAL）

客户端与 zookeeper 断开连接后，该节点被删除

4）临时顺序编号目录节点（EPHEMERAL_SEQUENTIAL）

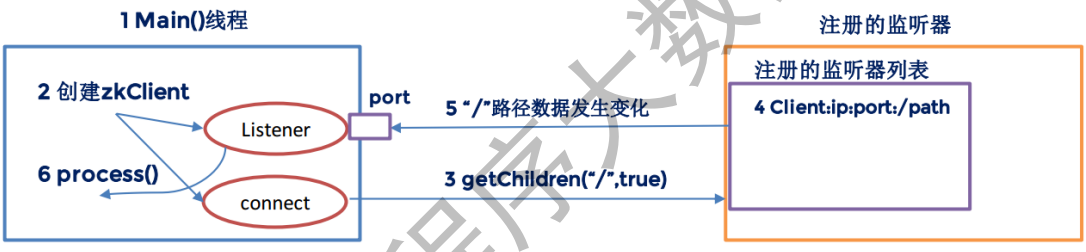
客户端与 zookeeper 断开连接后，该节点被删除，只是 Zookeeper 给该节点名称进行顺序编号。



创建 znode 时设置顺序标识， znode 名称后会附加一个值，顺序号是一个单调递增的计数器，由父节点维护

在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序

4.3 Zookeeper监听原理



监听原理详解

1）首先要有一个main()线程
2）在main线程中创建Zookeeper客户端， 这时就会创建两个线程， 一个负责网络连接通信（connet）， 一个负责监听（listener）。
3）通过connect线程将注册的监听事件发送给Zookeeper。
4）在Zookeeper的注册监听器列表中将注册的监听事件添加到列表中。
5）Zookeeper监听到有数据或路径变化， 就会将这个消息发送给listener线程。
6）listener线程内部调用了process（ ）方法。

4.4 写数据流程

1）Client向Zookeeper的server1上写数据，发送一个写请求

2）如果server1不是Leader，那么server1会把接受到的请求进一步转发给Leader，因为每个Zookeeper的Server里面有一个Leader。这个Leader会将写请求广播给各个Server,比如Server1和Server2，各个Server写成功后就会通知Leader

3) 当Leader收到大数据Server数据写成功了，那么就说明数据写成功了。如果这里三个节点的话，只要有两个节点写成功了，那么久任务数据写成功了，写成功了之后，Leader会告诉Server1数据写的成功了

4）Server1会进一步通知Client数据写成功了，这时就认为整个写操作成功。Zookeeper整个写数据流程就是这样

五、Zookeeper API应用案例(选讲)

Zookeeper提供了Java API方便我们来操作zk服务，可以通过maven引入zk的相关依赖包。通过org.apache.zookeeper.Zookeeper类创建连接zk服务器的示例对象，在创建过程中给定zk服务器地址、会话持续时间以及监视器三个参数，当连接创建成功后，通过Zookeeper实例提供的接口来和服务器进行交互

5.1 连接创建

使用Zookeeper类来表示连接，创建的该实例对象有四个构造方法来调用，不过一般最常用的是下面两个构造方法的调用：ZooKeeper(connectString,session-Timeout,watcher)和ZooKeeper(connectString,sessionTimeout,watcher,canBeRead-Only)；其中第一个构造方式底层调用第二个构造方法，只是canBeReadOnly参数设置为false。connectString参数为zk集群服务器的连接url，当给定路径的时候，表示所有的操作都是基于该路径进行操作的(路径只可以添加到最后)。例如:“hh:2181,hh:2182,hh:2183/app”；sessionTimeout为会话过期时间，一般设置为tickTime的3-4倍；watcher是监视器，用于触发相应事件，可以为空；canBeReadOnly是给定是否是只读连接，默认为false。

```
package com.beifeng.zookeeper;

import java.io.IOException;

import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.ZooDefs.Ids;

/**
 * 演示创建节点
 *
 * @author wubo
 *
 */
public class DemoCreate {
    public static void main(String[] args) throws IOException, KeeperException,
InterruptedException {
        test1();
//        test2();
    }

    /**
     * 测试创建连接的时候给定路径
     * @throws IOException
     * @throws InterruptedException
     * @throws KeeperException
     */
    static void test2() throws IOException, KeeperException,
InterruptedException {
        ZooKeeper client = new ZooKeeper("hh:2181,hh:2182,hh:2183/app", 2000,
new Watcher() {
            @Override
            public void process(WatchedEvent arg0) {
```

```
            }
        });

        String result = null;
        // 创建节点
//        result = client.create("/root", null, Ids.OPEN_ACL_UNSAFE,
CreateMode.PERSISTENT); // 创建永久节点
//        System.out.println("创建/root结果:" + result);

        // 添加子节点
        result = client.create("/root/child", "child".getBytes(),
Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT_SEQUENTIAL); // 添加一个永久顺序节点
        System.out.println("添加节点/root/child结果:" + result);

        // 创建临时节点
        result = client.create("/tmp", "".getBytes(), Ids.OPEN_ACL_UNSAFE,
CreateMode.EPHEMERAL);
        System.out.println("创建临时节点/tmp结果为:" + result);
        // 创建临时顺序节点
        result = client.create("/tmp", "".getBytes(), Ids.OPEN_ACL_UNSAFE,
CreateMode.EPHEMERAL_SEQUENTIAL);
        System.out.println("创建顺序临时节点/tmp结果为:" + result);

        Thread.sleep(10000);
        client.close(); // 关闭
    }

    static void test1() throws IOException, KeeperException,
InterruptedException {
        ZooKeeper client = new ZooKeeper("hh:2181,hh:2182,hh:2183", 2000, new
Watcher() {
            @Override
            public void process(WatchedEvent arg0) {
            }
        });

        // 创建节点
        String result = client.create("/root", null, Ids.OPEN_ACL_UNSAFE,
CreateMode.PERSISTENT); // 创建永久节点
        System.out.println("创建/root结果:" + result);

        // 添加子节点
        result = client.create("/root/child", "child".getBytes(),
Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT_SEQUENTIAL); // 添加一个永久顺序节点
        System.out.println("添加节点/root/child结果:" + result);

        result = client.create("/root/child", "child".getBytes(),
Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT); // 添加一个永久节点
        System.out.println("添加节点/root/child结果:" + result);

        // 创建临时节点
        result = client.create("/tmp", "".getBytes(), Ids.OPEN_ACL_UNSAFE,
CreateMode.EPHEMERAL);
        System.out.println("创建临时节点/tmp结果为:" + result);
        // 创建临时顺序节点
        result = client.create("/tmp", "".getBytes(), Ids.OPEN_ACL_UNSAFE,
CreateMode.EPHEMERAL_SEQUENTIAL);
        System.out.println("创建顺序临时节点/tmp结果为:" + result);
```

```
        Thread.sleep(5000);
        client.close(); // 关闭
    }
}
```

ZK中新增子节点和创建节点其实是同一个含义，创建一个节点其实就相当于在根目录下新增一个子节点，zk不支持为不存在的父节点创建子节点(不支持循环创建)。创建节点的时候要求指明节点被创建的类型(CreateMode)。调用Zookeeper实例的create方法，需要给定的参数有：path(节点路径), data(数据), acl(控制权限列表，不考虑权限的情况下给定为: Ids.OPEN_ACL_UNSAFE), createMode(节点类型)。

CreateMode类型	详解
EPHEMERAL	临时节点，在会话结束后自动被删除。
EPHEMERAL_SEQUENTIAL	临时顺序节点，在会话被结束后自动删除，会在给定的path节点名称后添加一个序列号(单调递增)。
PERSISTENT	永久节点，会话结束后不会被自动删除。
PERSISTENT_SEQUENTIAL	永久顺序节点，在节点名称后添加一个序列号(单调递增)，不会自动删除。

5.2 设置和读取数据

```
package com.beifeng.zookeeper;

import java.io.IOException;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;

/**
 * 演示设置数据和读取数据
 *
 * @author wubo
 *
 */
public class DemoData {
    public static void main(String[] args) throws IOException,
InterruptedException, KeeperException {
        ZooKeeper client = new ZooKeeper("hh", 2000, null);
        String data = "";
        // 获取数据
        // 第一种方式，当第二个参数为true的时候，会使用创建client时候给定的watcher实例进行
监控
        // 第二种方式，采用给定的监视器进行监控
        data = new String(client.getData("/root/child", false, null));
        System.out.println("第一次数据:" + data);

        // 设置数据
```

```
        client.setData("/root/child", (data + "new-data").getBytes(), -1);

        // 再重新获取数据
        data = new String(client.getData("/root/child", new Watcher() {
            @Override
            public void process(WatchedEvent arg0) {
            }
        }, null));
        System.out.println("第二次数据为:" + data);
        client.close();
    }

}
```

5.3 删除数据

```
package com.beifeng.zookeeper;

import java.io.IOException;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.ZooKeeper;

/**
 * 删除演示
 *
 * @author wubo
 *
 */
public class DemoDelete {
    public static void main(String[] args) throws IOException,
InterruptedException, KeeperException {
        ZooKeeper client = new ZooKeeper("hh:2182,hh:2181,hh:2183", 2000, null);

        // 删除/root/child
        client.delete("/root/child", -1);
        // 删除root
        client.delete("/root", client.exists("/root", false).getVersion()); //
给定具体的版本号进行删除操作
        // 递归删除会出现异常
        client.delete("/app", -1); // 不支持递归删除

        client.close();
    }
}
```

六、案例 分布式环境中实现共享锁(选讲)

6.1 需求场景

现在有三台机器A、B和C。其中A和B需要在C上对文件/usr/local/c.txt文件进行写操作，如果两台机器同时写该文件，那么该文件的最终结果可能会出现乱序的问题。要想正常的完成写操作，那么第一种方式就是A在写之间先告诉B，“我开始写文件了，你先不要写”，等待收到B的确认回复后，A再开始写文件，写完后再通知B“我已经写完了”。但是这样会存在一个问题，那么就是假设有200台机器，那么中间任意一台机器通信中断怎么办呢？当然了自己从零开始实行这个共享锁，也是可以的，但是采用zk提供的功能来实现更加方便快捷。

完成一个分布式环境的java.util.concurrent.locks.Lock实现。并进行测试，查看是否能够完成分布式环境的并发要求。

6.2 代码实现

```
package com.beifeng.lock;

import java.io.Closeable;
import java.io.IOException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

import org.apache.log4j.Logger;
import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.EventType;
import org.apache.zookeeper.ZooDefs.Ids;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.data.Stat;

/**
 * 实现自定义的分布式锁机制
 *
 * @author wubo
 *
 */
public class DistributedLock implements Lock, Closeable, Watcher {
    public static final Logger logger = Logger.getLogger(DistributedLock.class);
    private ZooKeeper client = null; // zk的客户端
    private String rootPath = "/distributed_lock"; // 锁机制的根节点
    private String nodePath = null; // 节点路径
    private int sleepTime = 1000;
    private String currentIp = null;

    /**
     * 传入一个lock节点名称
     *
     * @param lockNodeName
     * @throws InterruptedException
     * @throws KeeperException
     * @throws IOException
     */
    public DistributedLock(String lockNodeName) throws IOException,
KeeperException, InterruptedException {
        this("localhost:2181", lockNodeName);
```

```
    }

    /**
     * 根据传入的zk连接字符串和lock的节点名称进行初始化操作
     *
     * @param url
     * @param lockNodeName
     * @throws InterruptedException
     * @throws KeeperException
     * @throws IOException
     */
    public DistributedLock(String url, String lockNodeName) throws IOException,
KeeperException, InterruptedException {
        this(url, 2000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                // nothings
            }
        }, lockNodeName);
    }

    /**
     * 根据传入的参数构建zk客户端和lock实例
     *
     * @param url
     * @param sessiontTimeOut
     * @param watcher
     * @param lockNodeName
     *            要求是一个只包含数字字母和下划线的单词
     * @throws IOException
     * @throws InterruptedException
     * @throws KeeperException
     */
    public DistributedLock(String url, int sessiontTimeOut, Watcher watcher,
String lockNodeName)
            throws IOException, KeeperException, InterruptedException {
        // TODO: 对lockNodeName进行验证
        this.nodePath = this.rootPath + "/" + lockNodeName;
        this.client = new ZooKeeper(url, sessiontTimeOut, watcher);
        try {
            this.currentIp = java.net.InetAddress.getLocalHost().getHostName();
        } catch (Exception e) {
            this.currentIp = "unkown";
        }
        // 在jvm中添加一个client关闭的钩子
        Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    DistributedLock.this.close();
                } catch (IOException e) {
                    // nothings
                }
            }
        }));

        // 进行根节点是否存在的判断操作
        Stat stat = this.client.exists(this.rootPath, false);
```

```
        if (stat == null) {
            // 表示根节点不存在啊，创建
            try {
                this.client.create(this.rootPath, null, Ids.OPEN_ACL_UNSAFE,
CreateMode.PERSISTENT);
            } catch (Exception e) {
                // 1. 其他的进程创建了；2. 确实是创建失败
                stat = this.client.exists(this.rootPath, false);
                if (stat == null) {
                    throw new IOException(e);
                }
            }
        }
    }

    @Override
    public void lock() {
        boolean result = false;
        while (!result) {
            try {
                result = this.internalLock();
                if (result) {
                    break; // 结束循环
                }
            } catch (Exception e) {
                logger.warn("获取锁发生异常", e);
            }

            try {
                Thread.sleep(this.sleepTime);
            } catch (InterruptedException e) {
            }
        }
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
        boolean result = false;
        while (!result) {
            result = this.internalLock();
            if (result) {
                break; // 结束循环
            }

            Thread.sleep(this.sleepTime); // 休眠1秒
        }
    }

    @Override
    public boolean tryLock() {
        boolean result = false;
        try {
            result = this.internalLock();
        } catch (Exception e) {
            logger.warn("获取锁发生异常", e);
        }
        return result;
    }
```

```
    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException
{
        return this.internalLock(time, unit);
    }

    /**
     * 内部的获取锁方法，返回true，表示获取锁成功
     *
     * @return
     * @throws InterruptedException
     */
    private boolean internalLock() throws InterruptedException {
        while (true) {
            try {
                Stat stat = this.client.exists(this.nodePath, false); // 判断是否
存在
                if (stat == null) {
                    // 表示不存在
                    // 创建一个节点，设置为临时节点即可
                    try {
                        this.client.create(this.nodePath,
this.currentIp.getBytes(), Ids.OPEN_ACL_UNSAFE,
                                CreateMode.EPHEMERAL);
                        return true; // 获得锁
                    } catch (KeeperException e) {
                        // 创建失败，添加监视机制
                        this.client.exists(this.nodePath, this);
                    }
                } else {
                    // 存在，所有添加监视机制
                    this.client.exists(this.nodePath, this);
                }

                // 失败
                synchronized (this) {
                    this.wait(); // 等待继续创建
                }
            } catch (KeeperException e) {
                Thread.sleep(this.sleepTime);
            }
        }
    }

    /**
     * 内部的获取锁方法，返回true，表示获取锁成功
     *
     * @return
     * @throws InterruptedException
     */
    private boolean internalLock(long time, TimeUnit unit) throws
InterruptedException {
        if (unit != TimeUnit.MILLISECONDS) {
            throw new IllegalArgumentException("时间单位必须为毫秒");
        }
        long startTime = System.currentTimeMillis();
        long millisTime = time;
```

```
        while (true) {
            try {
                Stat stat = this.client.exists(this.nodePath, false); // 判断是否
存在
                if (stat == null) {
                    // 表示不存在
                    // 创建一个节点，设置为临时节点即可
                    try {
                        this.client.create(this.nodePath,
this.currentIp.getBytes(), Ids.OPEN_ACL_UNSAFE,
                                CreateMode.EPHEMERAL);
                        return true; // 获得锁
                    } catch (KeeperException e) {
                        // 创建失败，添加监视机制
                        this.client.exists(this.nodePath, this);
                    }
                } else {
                    // 节点存在，添加监视机制
                    this.client.exists(this.nodePath, this);
                }

                // 失败
                synchronized (this) {
                    this.wait(millisTime);
                }
                long endTime = System.currentTimeMillis();
                if (endTime - startTime > millisTime) {
                    // 如果过期时间超过time
                    return false;
                }
            } catch (KeeperException e) {
                Thread.sleep(this.sleepTime);
            }
        }
    }

    @Override
    public void unlock() {
        // 解锁，  直接删除节点
        try {
            if (this.client.exists(this.nodePath, false) != null) {
                this.client.delete(this.nodePath, -1);
            }
        } catch (InterruptedException | KeeperException e) {
            throw new RuntimeException("解锁失败", e);
        }
    }

    @Override
    public Condition newCondition() {
        return null;
    }

    @Override
    public void close() throws IOException {
        if (this.client != null) {
            try {
```

```
                this.client.close();
            } catch (InterruptedException e) {
                throw new IOException(e);
            } finally {
                this.client = null;
            }
        }
    }

    @Override
    public void process(WatchedEvent event) {
        if (EventType.NodeDeleted.equals(event.getType()) &&
this.nodePath.equals(event.getPath())) {
            // 表示是节点删除操作,而且是锁节点
            synchronized (this) {
                this.notifyAll(); // 通知全部
            }
        }
    }

}
```

```
package com.beifeng.lock;

import java.io.IOException;
import java.util.Random;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

import org.apache.zookeeper.KeeperException;

/**
 * 测试我们的分布式锁能否使用
 *
 * @author wubo
 *
 */
public class DistributedLockDemo {
    public static void main(String[] args) throws IOException, KeeperException,
InterruptedException {
//        test1();
        test2();
//        test3();
    }

    static void test2() throws IOException, KeeperException,
InterruptedException {
        final DistributedLock lock = new DistributedLock("hh:2181,hh:2182",
"lock");
        int n = 10;
        final CountDownLatch latch = new CountDownLatch(1);
        final CountDownLatch latch2 = new CountDownLatch(n);
        final Random random = new Random(System.currentTimeMillis());

        for (int i = 0; i < n; i++) {
            new Thread(new Runnable() {
```

```
                @Override
                public void run() {
                    int millis = random.nextInt(5000) + 2000;
                    try {
                        latch.await();// 等待
                        System.out.println(Thread.currentThread().getName() +
"线程休眠:" + millis);
                        Thread.sleep(millis);
                    } catch (InterruptedException e) {
                    }

                    // 开始执行
                    System.out.println(Thread.currentThread().getName() + "休眠结
束，开始获取锁");
                    try {
                        lock.tryLock(1000, TimeUnit.MILLISECONDS);
                    } catch (InterruptedException e1) {
                        throw new RuntimeException(e1);
                    }
                    try {
                        System.out.println(Thread.currentThread().getName() +
"获得锁 " + System.currentTimeMillis());
                        try {
                            Thread.sleep(millis);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    } finally {
                        System.out.println(Thread.currentThread().getName() +
"准备释放锁");
                        lock.unlock();
                        latch2.countDown();
                    }
                }
            }, "thread(" + i + ")").start();
        }

        latch.countDown(); // 启动
        latch2.await(); // 等待执行完成
        System.out.println("所有线程执行完");
        lock.close();
    }
    static void test3() throws IOException, KeeperException,
InterruptedException {
        final DistributedLock lock = new DistributedLock("hh:2181,hh:2182",
"lock");
        int n = 10;
        final CountDownLatch latch = new CountDownLatch(1);
        final CountDownLatch latch2 = new CountDownLatch(n);
        final Random random = new Random(System.currentTimeMillis());

        for (int i = 0; i < n; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    int millis = random.nextInt(5000) + 1000;
                    try {
                        latch.await();// 等待
```

```
                        System.out.println(Thread.currentThread().getName() +
"线程休眠:" + millis);
                        Thread.sleep(millis);
                    } catch (InterruptedException e) {
                    }

                    // 开始执行
                    System.out.println(Thread.currentThread().getName() + "休眠结
束，开始获取锁");
                    lock.lock();
                    try {
                        System.out.println(Thread.currentThread().getName() +
"获得锁 " + System.currentTimeMillis());
                        try {
                            Thread.sleep(millis);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    } finally {
                        System.out.println(Thread.currentThread().getName() +
"准备释放锁");
                        lock.unlock();
                        latch2.countDown();
                    }
                }
            }, "2thread(" + i + ")").start();
        }

        latch.countDown(); // 启动
        latch2.await(); // 等待执行完成
        System.out.println("所有线程执行完");
        lock.close();
    }

    static void test1() throws IOException, KeeperException,
InterruptedException {
        DistributedLock lock = new DistributedLock("hh:2181,hh:2182", "lock");
        lock.lock(); // 获取锁
        try {
            System.out.println("获得锁");
            Thread.sleep(10000);
        } finally {
            lock.unlock();
        }
        lock.close();
    }
}
```