Zookeeper实现分布式锁需要满足一下条件

1、在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；

2、高可用的获取锁与释放锁；

3、高性能的获取锁与释放锁；

4、具备可重入特性；

5、具备锁失效机制，防止死锁；

6、具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

用原生的Zookeeper API接口实现分布式锁不仅仅需要考虑锁机制问题，还需要解决持续监听、session超时、级联创建删除等问题，是不是有点头大，那有没有现成的框架呢。

Apache Curator是一个比较完善的zk客户端框架，封装了一套高级API 简化了zk的操作，并且实现了完善的分布式锁。

1、安装Zookeeper

2、使用分布式锁

3、浅析锁的实现

1、安装Zookeeper

略，没有安装的可以参考这篇文章

2、引入Curator依赖

<dependency>  
 <groupId>org.apache.zookeeper</groupId>  
 <artifactId>zookeeper</artifactId>  
 <version>3.6.1</version>  
</dependency>  
  
<dependency>  
 <groupId>org.apache.curator</groupId>  
 <artifactId>curator-client</artifactId>  
 <version>4.1.0</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.curator</groupId>  
 <artifactId>curator-framework</artifactId>  
 <version>4.2.0</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.curator</groupId>  
 <artifactId>curator-recipes</artifactId>  
 <version>4.2.0</version>  
</dependency>

三、创建锁工厂

@Slf4j  
public class ZkLock {  
 static CuratorFramework *framework*= null;  
 static {  
 *framework* = CuratorFrameworkFactory.*newClient*(  
 "106.55.154.105:2181",  
 20000,  
 20000,  
 new RetryNTimes(3, 5000));  
 *framework*.start();  
 }  
  
  
 */\*\*  
 \* 获取互斥锁  
 \** ***@param*** *name  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* public InterProcessMutex getLock(String name) throws Exception {  
 return new InterProcessMutex(*framework*,buildPath(name));  
 }  
  
 */\*\*  
 \* 获取互斥锁  
 \** ***@param*** *name  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* public InterProcessMutex getMutexLock(String name) throws Exception {  
 return new InterProcessMutex(*framework*,buildPath(name));  
 }  
  
 */\*\*  
 \* 获取读写锁  
 \** ***@param*** *name  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* public InterProcessReadWriteLock getReadWriteLock(String name) throws Exception {  
 return new InterProcessReadWriteLock(*framework*,buildPath(name));  
 }  
  
 */\*\*  
 \* 获取不可重入互斥锁  
 \** ***@param*** *name  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* public InterProcessSemaphoreMutex getSemaphoreLock(String name) throws Exception {  
 return new InterProcessSemaphoreMutex(*framework*,buildPath(name));  
 }  
  
 */\*\*  
 \* 获取多锁 集合锁  
 \** ***@param*** *names  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* public InterProcessMultiLock getMutilLock(List<String> names) throws Exception {  
 return new InterProcessMultiLock(*framework*,mutilPath(names));  
 }  
  
 public List<String> mutilPath(List<String> names){  
 List<String> paths = new ArrayList<>();  
 for(String name :names)  
 {  
 paths.add(buildPath(name));  
 }  
 return paths;  
 }  
  
  
 public String buildPath(String name)  
 {  
 String path = "";  
 String[] roots = new String[]{"mg","mylock"};  
 for(String str : roots)  
 {  
 if(str.startsWith("/")){  
 path +="/";  
 }  
 path +="/" +str;  
 }  
 path +="/" +name;  
 return path;  
 }  
}

四、使用分布式锁

curator提供了四种锁

可重入互斥锁 InterProcessMutex

不可重入互斥锁 InterProcessSemaphoreMutex

读写锁 InterProcessReadWriteLock

集合锁 InterProcessMultiLock

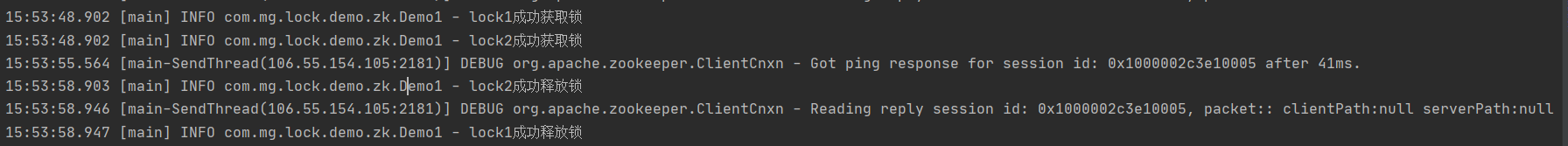
让我们分别看下怎么用吧

1、可重入互斥锁

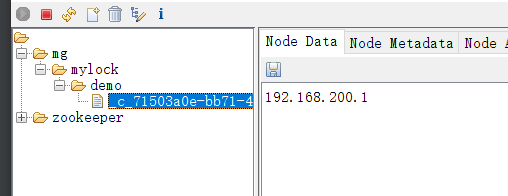
可重入互斥锁具备可重入性，试着执行如下代码

@Slf4j  
public class Demo1 {  
  
 public void lock1(InterProcessMutex lock) throws Exception {  
 lock.acquire();  
 *log*.info("lock1成功获取锁");  
 lock2(lock);  
 lock.release();  
 *log*.info("lock1成功释放锁");  
 }  
  
 public void lock2(InterProcessMutex lock) throws Exception {  
 lock.acquire();  
 *log*.info("lock2成功获取锁");  
 Thread.*sleep*(1000\*10);  
  
 lock.release();  
 *log*.info("lock2成功释放锁");  
 }  
  
 public static void main(String[] args) throws Exception {  
 ZkLock zkLock = new ZkLock();  
 InterProcessMutex lock = zkLock.getLock("demo");  
 Demo1 demo1 = new Demo1();  
 demo1.lock1(lock);  
  
 }  
}

控制台输出



使用ZooInspector查看zk中的锁节点，两次加锁只有一个节点，如下图



2、不可重入互斥锁

不可重入互斥锁不具备锁的可重入性，执行如下代码

@Slf4j  
public class Demo2 {  
  
 public void lock1(InterProcessSemaphoreMutex lock) throws Exception {  
 lock.acquire();  
 *log*.info("lock1成功获取锁");  
 lock2(lock);  
 lock.release();  
 *log*.info("lock1成功释放锁");  
 }  
  
 public void lock2(InterProcessSemaphoreMutex lock) throws Exception {  
 *log*.info("lock2尝试获取锁");  
 boolean result = lock.acquire(1000\*2, TimeUnit.*MILLISECONDS*);  
  
 if(result)  
 {  
 *log*.info("lock2成功获取锁");  
 Thread.*sleep*(1000\*10);  
 lock.release();  
 *log*.info("lock2成功释放锁");  
 }  
 else {  
 *log*.info("lock2获取锁失败");  
 }  
 }  
  
 public static void main(String[] args) throws Exception {  
 ZkLock zkLock = new ZkLock();  
 InterProcessSemaphoreMutex lock = zkLock.getSemaphoreLock("demo");  
 Demo2 demo2 = new Demo2();  
 demo2.lock1(lock);  
 }  
}

控制台输出结果

22:07:00.984 [main] INFO com.mg.lock.demo.zk.Demo2 - lock1成功获取锁

22:07:00.984 [main] INFO com.mg.lock.demo.zk.Demo2 - lock2尝试获取锁

22:07:11.163 [main] INFO com.mg.lock.demo.zk.Demo2 - lock2获取锁失败

22:07:11.220 [main] INFO com.mg.lock.demo.zk.Demo2 - lock1成功释放锁

同一个线程，lock2函数获取锁还是会失败。

五、浅析锁实现

看下InterProcessMutex中加锁和解锁的代码来简单了解下分布式锁的实现

1、创建锁

InterProcessMutex(CuratorFramework client, String path, String lockName, int maxLeases, LockInternalsDriver driver) {  
 this.threadData = Maps.newConcurrentMap();  
 this.basePath = PathUtils.validatePath(path);  
 this.internals = new LockInternals(client, driver, path, lockName, maxLeases);  
}

构造函数中创建了一个InterProcessMutex对象，并保存了锁的path、客户端等信息。

2、加锁

acquire函数如下

public boolean acquire(long time, TimeUnit unit) throws Exception {  
 return this.internalLock(time, unit);  
}

进到internalLock函数中

private boolean internalLock(long time, TimeUnit unit) throws Exception {  
 Thread currentThread = Thread.currentThread();  
 InterProcessMutex.LockData lockData = (InterProcessMutex.LockData)this.threadData.get(currentThread);  
 if (lockData != null) {  
 lockData.lockCount.incrementAndGet();  
 return true;  
 } else {  
 String lockPath = this.internals.attemptLock(time, unit, this.getLockNodeBytes());  
 if (lockPath != null) {  
 InterProcessMutex.LockData newLockData = new InterProcessMutex.LockData(currentThread, lockPath);  
 this.threadData.put(currentThread, newLockData);  
 return true;  
 } else {  
 return false;  
 }  
 }  
}

internalLock中，判断获取锁的线程是否是当前线程，是lockCount（AtomicInteger类型的原子类）加1继续执行，否则尝试获取锁。

再看下attemptLock这个函数

String attemptLock(long time, TimeUnit unit, byte[] lockNodeBytes) throws Exception {  
 long startMillis = System.currentTimeMillis();  
 Long millisToWait = unit != null ? unit.toMillis(time) : null;  
 byte[] localLockNodeBytes = this.revocable.get() != null ? new byte[0] : lockNodeBytes;  
 int retryCount = 0;  
 String ourPath = null;  
 boolean hasTheLock = false;  
 boolean isDone = false;  
  
 while(!isDone) {  
 isDone = true;  
  
 try {  
 ourPath = this.driver.createsTheLock(this.client, this.path, localLockNodeBytes);  
 hasTheLock = this.internalLockLoop(startMillis, millisToWait, ourPath);  
 } catch (NoNodeException var14) {  
 if (!this.client.getZookeeperClient().getRetryPolicy().allowRetry(retryCount++, System.currentTimeMillis() - startMillis, RetryLoop.getDefaultRetrySleeper())) {  
 throw var14;  
 }  
  
 isDone = false;  
 }  
 }  
  
 return hasTheLock ? ourPath : null;  
}

其中最主要的就是这两行代码

ourPath = this.driver.createsTheLock(this.client, this.path, localLockNodeBytes);//创建临时节点  
hasTheLock = this.internalLockLoop(startMillis, millisToWait, ourPath);//判断是否为最小节点  
已经基本和原生ZK API实现了一致，创建临时节点，监听等待当前节点为最小节点，获取到锁执行同步代码块。

3、解锁

Release方法如下

public void release() throws Exception {  
 Thread currentThread = Thread.currentThread();  
 InterProcessMutex.LockData lockData = (InterProcessMutex.LockData)this.threadData.get(currentThread);  
 if (lockData == null) {  
 throw new IllegalMonitorStateException("You do not own the lock: " + this.basePath);  
 } else {  
 int newLockCount = lockData.lockCount.decrementAndGet();  
 if (newLockCount <= 0) {  
 if (newLockCount < 0) {  
 throw new IllegalMonitorStateException("Lock count has gone negative for lock: " + this.basePath);  
 } else {  
 try {  
 this.internals.releaseLock(lockData.lockPath);  
 } finally {  
 this.threadData.remove(currentThread);  
 }  
  
 }  
 }  
 }  
}