**为什么要用分布式锁**

Martin Kleppmann是英国剑桥大学的分布式系统的研究员，之前和Redis之父Antirez进行过关于RedLock(红锁，后续有讲到)是否安全的激烈讨论。Martin认为一般我们使用分布式锁有两个场景:

* 效率:使用分布式锁可以避免不同节点重复相同的工作，这些工作会浪费资源。比如用户付了钱之后有可能不同节点会发出多封短信。
* 正确性:加分布式锁同样可以避免破坏正确性的发生，如果两个节点在同一条数据上面操作，比如多个节点机器对同一个订单操作不同的流程有可能会导致该笔订单最后状态出现错误，造成损失。

**分布式锁的一些特点**

当我们确定了在不同节点上需要分布式锁，那么我们需要了解分布式锁到底应该有哪些特点:

* 互斥性:和我们本地锁一样互斥性是最基本，但是分布式锁需要保证在不同节点的不同线程的互斥。
* 可重入性:同一个节点上的同一个线程如果获取了锁之后那么也可以再次获取这个锁。
* 锁超时:和本地锁一样支持锁超时，防止死锁。
* 高效，高可用:加锁和解锁需要高效，同时也需要保证高可用防止分布式锁失效，可以增加降级。
* 支持阻塞和非阻塞:和ReentrantLock一样支持lock和trylock以及tryLock(long timeOut)。
* 支持公平锁和非公平锁(可选):公平锁的意思是按照请求加锁的顺序获得锁，非公平锁就相反是无序的。这个一般来说实现的比较少。

**常见的分布式锁**

我们了解了一些特点之后，我们一般实现分布式锁有以下几个方式:

* MySql
* zookeeper
* Redis
* 自研分布式锁:如谷歌的Chubby。

**为什么zookeeper可以实现分布式锁？**

多个进程内同一时间都有线程在执行方法m，锁就一把，你获得了锁得以执行，我就得被阻塞，那你执行完了谁来唤醒我呢？你并不知道我被阻塞了，你也就不能通知我。你能做的只有用的时候设置锁标志，用完了再取消你设置的标志。我就必须在阻塞的时候隔一段时间主动去看看，但这样总归是有点麻烦的，最好有人来通知我可以执行了。zookeeper对于自身节点的监听者提供事件通知功能，是不是有点雪中送炭的感觉呢。

**节点是什么？**节点是zookeeper中数据存储的基础结构，zk中万物皆节点，就好比java中万物皆对象是一样的。zk的数据模型就是基于好多个节点的树结构，但zk规定每个节点的引用规则是**路径引用**。每个节点中包含子节点引用、存储数据、访问权限以及节点元数据等四部分。

**zk中节点有类型区分吗？**有。zk中提供了四种类型的节点，各种类型节点及其区别如下：

* 持久节点（PERSISTENT）：节点创建后，就一直存在，直到有删除操作来主动清除这个节点
* 持久顺序节点（PERSISTENT\_SEQUENTIAL）：保留持久节点的特性，额外的特性是，每个节点会为其第一层子节点维护一个顺序，记录每个子节点创建的先后顺序，ZK会自动为给定节点名加上一个数字后缀（自增的），作为新的节点名。
* 临时节点（EPHEMERAL）：和持久节点不同的是，临时节点的生命周期和客户端会话绑定，当然也可以主动删除。
* 临时顺序节点（EPHEMERAL\_SEQUENTIAL）：保留临时节点的特性，额外的特性如持久顺序节点的额外特性。

**如何操作节点？**

节点的增删改查分别是creat\delete\setData\getData，exists判断节点是否存在，getChildren获取所有子节点的引用。

上面提到了节点的监听者，我们可以在对zk的节点进行查询操作时，设置当前线程是否监听所查询的节点。getData、getChildren、exists都属于对节点的查询操作，这些方法都有一个boolean类型的watch参数，用来设置是否监听该节点。一旦某个线程监听了某个节点，那么这个节点发生的creat（在该节点下新建子节点）、setData、delete（删除节点本身或是删除其某个子节点）都会触发zk去通知监听该节点的线程。但需要注意的是，线程对节点设置的监听是一次性的，也就是说zk通知监听线程后需要改线程再次设置监听节点，否则该节点再次的修改zk不会再次通知。

zookeeper具备了实现分布式锁的基础条件：多进程共享、可以存储锁信息、有主动通知的机制。

**怎么使用zookeeper实现分布式锁呢？**

分布式锁也是锁，没什么牛的，它也需要一个名字来告诉别人自己管理的是哪块同步资源，也同样需要一个标识告诉别人自己现在是空闲还是被使用。zk中，需要创建一个专门的放锁的节点，然后各种锁节点都作为该节点的子节点方便管理，节点名称用来表明自己管理的同步资源。那么锁标识呢？

方案一：使用节点中的存储数据区域，zk中节点存储数据的大小不能超过1M，但是只是存放一个标识是足够的。线程获得锁时，先检查该标识是否是无锁标识，若是可修改为占用标识，使用完再恢复为无锁标识。

方案二：使用子节点，每当有线程来请求锁的时候，便在锁的节点下创建一个子节点，子节点类型必须维护一个顺序，对子节点的自增序号进行排序，默认总是最小的子节点对应的线程获得锁，释放锁时删除对应子节点便可。

**死锁风险**

两种方案其实都是可行的，但是使用锁的时候一定要去规避死锁。方案一看上去是没问题的，用的时候设置标识，用完清除标识，但是要是持有锁的线程发生了意外，释放锁的代码无法执行，锁就无法释放，其他线程就会一直等待锁，相关同步代码便无法执行。方案二也存在这个问题，但方案二可以利用zk的临时顺序节点来解决这个问题，只要线程发生了异常导致程序中断，就会丢失与zk的连接，zk检测到该链接断开，就会自动删除该链接创建的临时节点，这样就可以达到即使占用锁的线程程序发生意外，也能保证锁正常释放的目的。

那要是zk挂了怎么办？sad，zk要是挂了就没辙了，因为线程都无法链接到zk，更何谈获取锁执行同步代码呢。不过，一般部署的时候，为了保证zk的高可用，都会使用多个zk部署为集群，集群内部一主多从，主zk一旦挂掉，会立刻通过选举机制有新的主zk补上。zk集群挂了怎么办？不好意思，除非所有zk同时挂掉，zk集群才会挂，概率超级小。

要什么东西

1. 需要一个锁对象，每次创建这个锁对象的时候需要连接zk(也可将连接操作放在加锁的时候);
2. 锁对象需要提供一个加锁的方法；
3. 锁对象需要提供一个释放锁的方法；
4. 锁对象需要监听zk节点，提供接收zk通知的回调方法。

实现分析

1. 构造器中，创建zk连接，创建锁的根节点，相关API如下：

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher)  
创建zk连接。该构造器要求传入三个参数分别是：ip:端口(String)、会话超时时间、本次连接的监听器。  
public String create(String path, byte[] data, List<ACL> acl, CreateMode createMode)创建节点。参数：节点路径、节点数据、权限策略、节点类型

1. **加锁**时，首先需要在锁的根节点下**创建一个临时顺序节点**（该节点名称规则统一，由zk拼接自增序号），然后获取根节点下所有子节点，将节点根据自增序号进行排序，判断最小的节点是否为本次加锁创建的节点，若是，加锁成功，若否，阻塞当前线程，等待锁释放（阻塞线程可以使用）。相关API如下：

public List<String> getChildren(String path, boolean watch)  
获取某节点的所有子节点。参数：节点路径、是否监控该节点

1. **释放锁**时，**删除线程创建的子节点**，同时关闭zk连接。相关API如下：

public void delete(String path, int version)  
删除指定节点。参数：节点路径、数据版本号  
public synchronized void close()  
断开zk链接

1. **监听节点**。首先需要明确监听哪个节点，我们可以监听锁的根节点，这样每当有线程释放锁删除对应子节点时，zk就会通知监听线程，有锁被释放了，这个时候只需要获取根节点的所有子节点，根据自增序号判断自己对应的节点是否为最小，便可知道自己能否获取锁。但是上述做法很明显有一点不太好，只要有子节点被移除，zk就会重新通知所有等待锁的线程。
2. 获得不到锁的线程接收到通知后发现自己还需等待，又得重新设置监听再次等待。由于我们要采用临时**有序**节点，该类型节点的特性就是有序，那么就可以只监听上一个节点，也就是等待被移除的节点，这样可以保证接到通知时，就是对应子节点时最小，可以获得锁的时候
3. 。在实现分布式锁的时候，线程加锁时如果不能立马获得锁，便会被通过特定方式阻塞，那么既然**接到通知**时便是**可以获得锁**的时候，那么对应的操作就应该是**恢复线程**的执行，**取消阻塞**。

zk提供了Watcher接口,锁对象需要监听zk中上一个节点，便需要实现该接口。Watcher接口内部包含封装了事件类型和连接类型的Event接口，还提供了唯一一个需要实现的方法。  
**void process(WatchedEvent var1)**  
该方法便是用来接收zk通知的回调方法。参数为监听节点发生的事件。当监听器监听的节点发生变化时，zk会通知监听者，同时该方法被执行，参数便是zk通知的信息。  


虽然是一个简单的分布式锁的实现，代码也有点略长。代码中判断加锁的方法中，使用分隔符字符串是为了区分各个资源的锁。项目中有临界资源A和B，那么管理A的锁释放与否，跟线程要持有管理B的锁是没有关系的。当然，也可以每一类锁单独建立独立的根节点。

public class ZooKeeperLock implementsWatcher {

private ZooKeeper zk = null;

private String rootLockNode; // 锁的根节点

private String lockName; // 竞争资源，用来生成子节点名称

private String currentLock; // 当前锁

private String waitLock;// 等待的锁（前一个锁）

private CountDownLatch countDownLatch; //计数器（用来在加锁失败时阻塞加锁线程）

private int sessionTimeout = 30000;// 超时时间

// 1. 构造器中创建ZK链接，创建锁的根节点 public ZooKeeperLock(String zkAddress, String rootLockNode, StringlockName) {

this.rootLockNode = rootLockNode;

this.lockName = lockName;

try {

// 创建连接，zkAddress格式为：IP:PORT

zk= new ZooKeeper(zkAddress,this.sessionTimeout,this);

// 检测锁的根节点是否存在，不存在则创建

Stat stat = zk.exists(rootLockNode,false);

if (null == stat) {

zk.create(rootLockNode, new byte[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.PERSISTENT);}

} catch (IOException e) {

e.printStackTrace();

} catch (InterruptedExceptione) {

e.printStackTrace();

} catch (KeeperException e) {

 e.printStackTrace(); }}

// 2. 加锁方法，先尝试加锁，不能加锁则等待上一个锁的释放

 public boolean lock(){

if (this.tryLock()) {

System.out.println("线程【"+ Thread.currentThread().getName() + "】加锁（" +this.currentLock + "）成功！");

return true; } else{ return waitOtherLock(this.waitLock, this.sessionTimeout); } }

public boolean tryLock(){ // 分隔符 String split = "\_lock\_";

if (this.lockName.contains("\_lock\_")){

throw new RuntimeException("lockName can't contains '\_lock\_' "); }

try {

// 创建锁节点（临时有序节点）

this.currentLock =zk.create(this.rootLockNode + "/" + this.lockName + split, newbyte[0],

ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);

System.out.println("线程【" +Thread.currentThread().getName() + "】创建锁节点（"+ this.currentLock + "）成功，开始竞争...");

// 取所有子节点 List<String> nodes = zk.getChildren(this.rootLockNode, false);

// 取所有竞争lockName的锁List<String> lockNodes = new ArrayList<String>(); for (StringnodeName : nodes) {

if(nodeName.split(split)[0].equals(this.lockName)) {lockNodes.add(nodeName); } } Collections.sort(lockNodes);

// 取最小节点与当前锁节点比对加锁 String currentLockPath = this.rootLockNode + "/" +lockNodes.get(0);

if (this.currentLock.equals(currentLockPath)) { return true;}

// 加锁失败，设置前一节点为等待锁节点

StringcurrentLockNode=this.currentLock.substring(this.currentLock.lastIndexOf("/") + 1);

int preNodeIndex = Collections.binarySearch(lockNodes, currentLockNode) - 1;

this.waitLock = lockNodes.get(preNodeIndex); }

catch (KeeperException e) {

e.printStackTrace(); }

catch (InterruptedException e) {

e.printStackTrace(); } returnfalse; }

private boolean waitOtherLock(String waitLock, int sessionTimeout) {boolean islock = false;

try {

 // 监听等待锁节点 String waitLockNode= this.rootLockNode + "/" + waitLock;

Stat stat = zk.exists(waitLockNode,true);

if (null != stat) {

System.out.println("线程【"+ Thread.currentThread().getName() + "】锁（" +this.currentLock + "）加锁失败，等待锁（" + waitLockNode+ "）释放...");

// 设置计数器，使用计数器阻塞线程 this.countDownLatch = new CountDownLatch(1);

islock=this.countDownLatch.await(sessionTimeout,TimeUnit.MILLISECONDS);this.countDownLatch = null;

if (islock) {

System.out.println("线程【" + Thread.currentThread().getName() + "】锁（" + this.currentLock + "）加锁成功，锁（"+ waitLockNode + "）已经释放");

 } else {System.out.println("线程【" +Thread.currentThread().getName() + "】锁（" +this.currentLock + "）加锁失败...");

} } else {islock = true; } }

catch (KeeperException e) { e.printStackTrace();

 } catch(InterruptedException e) { e.printStackTrace();

 } return islock;

 }

// 3. 释放锁

public void unlock()

throws InterruptedException {

try { Stat stat= zk.exists(this.currentLock,false);

 if (null != stat) {

System.out.println("线程【" + Thread.currentThread().getName() + "】释放锁 " + this.currentLock);

zk.delete(this.currentLock, -1);

this.currentLock = null; } }

catch (InterruptedException e) {

e.printStackTrace(); }

catch (KeeperException e) {

e.printStackTrace(); }finally {

zk.close(); } } //

 4. 监听器回调

 @Override publicvoid

process(WatchedEvent watchedEvent) {

 if (null != this.countDownLatch&& watchedEvent.getType() == Event.EventType.NodeDeleted) {

 // 计数器减一，恢复线程操作 this.countDownLatch.countDown(); } } }

测试类如下：

public class Test {

    public static void doSomething() {

        System.out.println("线程【"+Thread.currentThread().getName() + "】正在运行...");

    }

    public static void main(String[]args) {

        Runnable runnable = new Runnable() {

            public void run() {

                ZooKeeperLock lock =null;

                lock = new ZooKeeperLock("10.150.27.51:2181","/locks", "test1");

                if (lock.lock()) {

                    doSomething();

                    try {

                       Thread.sleep(1000);

                       lock.unlock();

                    } catch (InterruptedExceptione) {

                       e.printStackTrace();

                    }

                }

            }

        };

        for (int i = 0; i < 5; i++) {

            Thread t = newThread(runnable);

            t.start();

        }

    }

}

这里启动了5个线程来进行验证，输出结果如下。需要注意的是，子节点的创建顺序一定是从小到大的，但是下面输出结果中显示创建顺序的随机是由于创建节点和输出语句不是原子操作导致的。重点是锁的获取和释放，从输出结果中可以看出，每个线程只有在上一个节点被删除后才能执行。ok,一个基于zk的简单的分布式锁就实现了。

线程【Thread-3】创建锁节点（/locks/test1\_lock\_0000000238）成功，开始竞争...

线程【Thread-2】创建锁节点（/locks/test1\_lock\_0000000237）成功，开始竞争...

线程【Thread-1】创建锁节点（/locks/test1\_lock\_0000000236）成功，开始竞争...

线程【Thread-0】创建锁节点（/locks/test1\_lock\_0000000240）成功，开始竞争...

线程【Thread-4】创建锁节点（/locks/test1\_lock\_0000000239）成功，开始竞争...

线程【Thread-1】加锁（/locks/test1\_lock\_0000000236）成功！

线程【Thread-1】正在运行...

线程【Thread-3】锁（/locks/test1\_lock\_0000000238）加锁失败，等待锁（/locks/test1\_lock\_0000000237）释放...

线程【Thread-2】锁（/locks/test1\_lock\_0000000237）加锁失败，等待锁（/locks/test1\_lock\_0000000236）释放...

线程【Thread-0】锁（/locks/test1\_lock\_0000000240）加锁失败，等待锁（/locks/test1\_lock\_0000000239）释放...

线程【Thread-4】锁（/locks/test1\_lock\_0000000239）加锁失败，等待锁（/locks/test1\_lock\_0000000238）释放...

线程【Thread-1】释放锁 /locks/test1\_lock\_0000000236

线程【Thread-2】锁（/locks/test1\_lock\_0000000237）加锁成功，锁（/locks/test1\_lock\_0000000236）已经释放

线程【Thread-2】正在运行...

线程【Thread-2】释放锁/locks/test1\_lock\_0000000237

线程【Thread-3】锁（/locks/test1\_lock\_0000000238）加锁成功，锁（/locks/test1\_lock\_0000000237）已经释放线程【Thread-3】正在运行...

线程【Thread-3】释放锁 /locks/test1\_lock\_0000000238

线程【Thread-4】锁（/locks/test1\_lock\_0000000239）加锁成功，锁（/locks/test1\_lock\_0000000238）已经释放

线程【Thread-4】正在运行...

线程【Thread-4】释放锁/locks/test1\_lock\_0000000239

线程【Thread-0】锁（/locks/test1\_lock\_0000000240）加锁成功，锁（/locks/test1\_lock\_0000000239）已经释放线程【Thread-0】正在运行...

线程【Thread-0】释放锁 /locks/test1\_lock\_0000000240

三、别人造好的轮子

话说zookeeper红火了这么久，就没有几个牛逼的人物去开源一些好用的工具，还需要自己这么费劲去写分布式锁的实现？是的，有的，上面小白也只是为了加深自己对zk实现分布式锁的理解去尝试做一个简单实现。有个叫Jordan Zimmerman的牛人提供了Curator来更好地操作zookeeper。

curator的分布式锁

curator提供了四种分布式锁，分别是：

* InterProcessMutex：分布式可重入排它锁
* InterProcessSemaphoreMutex：分布式排它锁
* InterProcessReadWriteLock：分布式读写锁
* InterProcessMultiLock：将多个锁作为单个实体管理的容器

pom依赖：

<dependency>  
 <groupId>org.apache.curator</groupId>  
 <artifactId>curator-framework</artifactId>  
 <version>4.0.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.curator</groupId>  
 <artifactId>curator-recipes</artifactId>  
 <version>4.0.0</version>  
 </dependency>

这里使用InterProcessMutex，即分布式可重入排他锁，用法如下：

// 设置重试策略，创建zk客户端

RetryPolicyretryPolicy=newExponentialBackoffRetry(1000,3);

CuratorFrameworkclient=CuratorFrameworkFactory.newClient("10.150.27.51:2181",retryPolicy);

// 启动客户端 client.start();

// 创建分布式可重入排他锁，监听客户端为client，锁的根节点为/locks InterProcessMutex mutex= new InterProcessMutex(client,"/locks");

try {

// 加锁 if (mutex.acquire(3,TimeUnit.SECONDS)) {

 // TODO-同步操作 //释放锁 mutex.release();

 } } catch (Exceptione) { e.printStackTrace();

 } finally { client.close(); }

InterProcessMutex源码解读

InterProcessMutex改造器较多，这里就不展示改造器源码了，建议感兴趣的朋友自己看看。InterProcessMutex内部有个ConcurrentMap类型的threadData属性，该属性会以线程对象为键，线程对应的LcokData对象为值，记录每个锁的相关信息。在new一个InterProcessMutex实例时，其构造器主要是为threadData进行Map初始化，校验锁的根节点的合法性并使用basePath属性记录，此外还会实例化一个LockInternals对象由属性internals引用，LockInternals是InterProcessMutex加锁的核心。

**加锁**

    // InterProcessMutex.class

    public void acquire() throwsException {

        if (!this.internalLock(-1L, (TimeUnit)null)) {

            throw new IOException("Lostconnection while trying to acquire lock: " + this.basePath);

        }

    }

    public boolean acquire(long time,TimeUnit unit) throws Exception {

        return this.internalLock(time, unit);

    }

    private boolean internalLock(longtime, TimeUnit unit) throws Exception {

        Thread currentThread =Thread.currentThread();

        InterProcessMutex.LockDatalockData = (InterProcessMutex.LockData)this.threadData.get(currentThread);

        if (lockData != null) {

            // 锁的可重入性

           lockData.lockCount.incrementAndGet();

            returntrue;

        } else {

            // 加锁并返回锁节点

            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);

                returntrue;

            } else {

                returnfalse;

            }

        }

    }

加锁提供了两个接口，分别为不设置超时和设置超时。不设置超时的话，线程等待锁时会一直阻塞，直到获取到锁。不管哪个加锁接口，都调用了internalLock()方法。这个方法里的代码体现了锁的可重入性。InterProcessMutex会直接从threadData中根据当前线程获取其LockData,若LockData不为空，则意味着当前线程拥有此，在锁的次数上加一就直接返回true。若为空，则通过internals属性的attemptLock()方法去竞争锁，该方法返回一个锁对应节点的路径。若该路径不为空，代表当前线程获得到了锁，然后为当前线程创建对应的LcokData并记录进threadData中。

**竞争锁**

    // LockInternals.class

    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 ? newbyte[0] : lockNodeBytes;

        int retryCount = 0;

        String ourPath = null;

        boolean hasTheLock = false;

        boolean isDone = false;

        while(!isDone) {

            isDone = true;

            try {

                // 创建锁节点

                ourPath =this.driver.createsTheLock(this.client, this.path,localLockNodeBytes);

                // 竞争锁

                hasTheLock =this.internalLockLoop(startMillis, millisToWait, ourPath);

            } catch (NoNodeExceptionvar14) {

               if(!this.client.getZookeeperClient().getRetryPolicy().allowRetry(retryCount++,

                       System.currentTimeMillis() - startMillis,RetryLoop.getDefaultRetrySleeper())) {

                    throw var14;

                }

                isDone = false;

            }

        }

        return hasTheLock ? ourPath : null;

    }

一大堆的变量定义，全部先忽略掉。最终的返回值由hasTheLock决定，为true时返回ourPath。ourPath初始化为null,后经this.driver.createsTheLock(this.client, this.path, localLockNodeBytes)赋值，这个方法点击去可看到默认的锁驱动类的创建锁节点方法，可知这里只是创建了锁节点。再看hasTheLock，为internalLockLoop()方法的返回值，只有该方法返回true时，attemptLock()才会返回锁节点路径，才会加锁成功。那OK，锁的竞争实现是由internalLockLoop进行。上面循环中的异常捕捉中是根据客户端的重试策略进行重试。

// LockInternals.class

private booleaninternalLockLoop(long startMillis, Long millisToWait, String ourPath) throwsException {

boolean haveTheLock = false;

boolean doDelete = false;

try {

if(this.revocable.get()!=null{

((BackgroundPathable)this.client.getData().usingWatcher(this.revocableWatcher)).forPath(ourPath);}

while(this.client.getState()==CuratorFrameworkState.STARTED &&!haveTheLock) {

// 获取所有子节点 List<String> children =this.getSortedChildren();

// 获取当前锁节点

StringsequenceNodeName = ourPath.substring(this.basePath.length() + 1);

 // 使用锁驱动加锁

PredicateResults predicateResults =this.driver.getsTheLock(this.client, children,

sequenceNodeName,this.maxLeases);

if (predicateResults.getsTheLock()) { haveTheLock = true;

} else{

 // 阻塞等待上一个锁释放 String previousSequencePath =this.basePath + "/" + predicateResults.getPathToWatch();

synchronized(this){

try{

((BackgroundPathable)this.client.getData().usingWatcher(this.watcher)).forPath(previousSequencePath);

if (millisToWait == null) {

// 未设置超时一直阻塞 this.wait();

}else { millisToWait = millisToWait - (System.currentTimeMillis() -startMillis);

startMillis = System.currentTimeMillis();

 // 根据时间设置阻塞时间 if (millisToWait > 0L) { this.wait(millisToWait); } else {

// 已经超时，设置删除节点标识doDelete = true; break;

 } } } catch (NoNodeException var19) {;

 } } } } } catch (Exception var21) {

ThreadUtils.checkInterrupted(var21);

doDelete= true; throw var21;

 } finally { if (doDelete) {

// 删除已超时的锁节点

this.deleteOurPath(ourPath);

 } } return haveTheLock;

}

返回值是haveTheLock,布尔型，看名字就知道这个变量代表竞争锁的成功与否。该变量的赋值发生在循环内，ok,看循环。先是获取所有子节点以及当前节点名称，再由驱动类进行锁竞争，竞争结果封装在PredicateResults类中，该类中包含一个布尔型的结果标识getsTheLock和一个监听节点路径pathToWatch。最后根据所竞争结果决定是否阻塞线程等待监听锁节点的释放。需要注意的是，这里阻塞使用的是对象的wait()机制，同时根据是否设置超时时间，是否已经超时决定线程阻塞时间或是删除超时节点。but,锁竞争的具体实现还是不在这里，这里只是有详细的锁等待实现。Curator默认的锁驱动类是StandardLockInternalsDriver。

//StandardLockInternalsDriver.classpublic

PredicateResults

getsTheLock(CuratorFramework client, List<String>children, String sequenceNodeName, int maxLeases)

throws Exception {

intourIndex = children.indexOf(sequenceNodeName);

validateOurIndex(sequenceNodeName, ourIndex);

boolean getsTheLock = ourIndex< maxLeases;

String pathToWatch = getsTheLock ?

null :(String)children.get(ourIndex - maxLeases);

return newPredicateResults(pathToWatch, getsTheLock); }

首先获取所有子节点中当前节点所在的位置索引，然后校验该索引，内部实现为判断是否小于0，成立则抛出一个NoNodeException。那肯定不是0啦。最终能否获得锁取决于该位置索引是否为0，也就是当前节点是否最小（maxLeases在InterProcessMutex构造器中初始化LockInternals设定的是1）。

总结

本文基于ZK实现分布式锁的思路、实现以及Curator的分布式可重入排他锁的原理剖析，算是小白研究ZK实现分布式锁的所有收获了。个人觉的关键点还是在于以下几点：

* 利用临时节点避免客户端程序异常导致的死锁；
* 利用有序节点设定锁的获取规则；
* 利用进程内的线程同步机制实现跨进程的分布式锁等待。  
  好了今天的分享就到这里了，如果想要了解更多JAVA高级教程，可以免费来Java直播公开课学习，或者进群：901439810，领取架构师全套学资料