分布式锁原理探究

一、前言

在同一个 JVM 进程中，我们可以使用 JUC 提供的一些锁来解决多个线程竞争同一个共享资源时候的线程安全问题，但是当多个不同 JVM 进程中的线程共同竞争同一个共享资源时候，JUC 包的锁就无能无力了，这时候就需要分布式锁了。

本 Chat 主要讲解几种常见的分布式锁实现方案以及原理，主要内容如下：

- 分布式锁与 JUC 包锁的不同；
- 使用数据库锁来实现分布式锁；
- 使用 Redis 来实现分布式锁；
- 使用 Zookeeper 的序列节点来实现分布式锁；
- 三种方案的简单对比。

二、分布式锁与 JUC 包锁

Java JDK 里面的并发包（JUC）里提供了一些锁，比如 ReentrantLock、ReentrantReadWriteLock、StampedLock 等（这些锁的原理实现可以参考 chat：Java 并发编程之美：并发编程高级篇之三），在同一个 JVM 中多个线程共同竞争同一个资源时，可以使用这些锁来保证访问资源的线程安全性。在同一个 JVM 进程中，你可以创建一个全局的（多个线程都可以访问到的）锁的实例，然后在具体访问资源前调用锁的 Lock 方法获取锁:

```
ReentrantLock lock = new ReentrantLock();//(1)

lock.lock();//(2)获取锁
try {

   //访问共享资源（3）
} catch (Exception e) {
    e.printStackTrace();
} finally {
    lock.unlock();//4释放锁
}
```

如上代码 1 创建了一个锁，然后在多个线程中访问共享资源前都可以调用锁的 lock 方法对资源进行加锁，从而保证资源同时只能被一个线程访问（当然对应读锁来说的话，多

个线程可以同时访问共享资源）。

上面介绍的是在同一个 JVM 中的情况，那么如果在多个 JVM 进程中的多个线程共同竞争同一个共享资源那，更常见的是**在不同主机的 JVM 进程中的多个线程共同访问同一个共享资源时候，这时候 JUC 包的锁能保证对资源访问的安全性?**

答案是不能的，因为 JUC 包锁的作用域是创建该锁的 JVM 进程内的，其它主机的 JVM 进程是没有办法访问到当前主机创建的锁的。也就是 JUC 包的锁在同一个 JVM 中是全局的，所有线程都可以使用方法获取到，然后该锁就可以保证同一个 JVM 进程内多个线程竞争同一个共享资源的安全性。那么对应不同主机的 JVM 进程中的线程，同样可以搞一个对各个主机来说是全局的锁，这个全局的锁就是分布式锁，所谓分布式，说白了，是说存在多个 JVM 进程，而分布式锁，就是能保证多个 JVM 进程中的线程共同访问共享资源时候，资源安全性的锁。

三、使用数据库悲观锁来实现分布式锁

本节我们来使用单实例数据库的悲观锁来实现分布式锁，所谓悲观锁是指对数据记录被外界修改持保守态度。在对数据记录处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的行锁机制，数据库中实现是对数据记录操作前给行记录加排它锁。

多个线程获取同一个行锁时候，如果获取锁失败，则说明数据正在被其它线程修改，则等待或者抛出异常。如果加锁成功，则获取记录，对其修改，然后事务提交后释放锁。

使用悲观锁的一个常用的例子： `select * from 表 where id = #id for update`，当多个线程（无论是同一个 JVM 中的线程还是不同 JVM 中的多个线程）开启事务传递相同的 id 执行该语句时，只有一个线程会获取到该 id 对应的行记录的锁然后返回，其它线程则会阻塞到该语句的执行上，等获取行锁的线程提交事务后就释放了行锁，阻塞的多个线程就会通过竞争使一个线程获取到行锁，其它线程继续阻塞。由于不同 JVM 中线程共同去竞争的同一个行记录，所以这就实现了一个分布式锁。

下面我们就是用上面介绍的原理来实现一个分布式锁，首先需要建立一个表 lock，表里面字段有一个 id 就可以了 (保证唯一)，然后插入一行记录。

这个实现原理比较简单，下面我们先来看实现代码：

```
public class DBdistributedLock {

    private DataSource dataSource;

    private static final String cmd = "select * from lock where
id = 1 for update";

    public DBdistributedLock(DataSource ds) {
        this.dataSource = ds;
```

```
    }

    public static interface CallBack{
        public void doAction();
    }
    public void lock(CallBack callBack)  {
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;

        try {
            //3.1try get lock
            System.out.println(Thread.currentThread().getName() +
" begin try lock");
            conn = dataSource.getConnection();
            conn.setAutoCommit(false);
            stmt = conn.prepareStatement(cmd);
            rs = stmt.executeQuery();

            //3.2do business thing
            callBack.doAction();

            //3.3release lock
            conn.commit();
            System.out.println(Thread.currentThread().getName() +
" release lock");

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

        } finally {
            //3.4
            if (null != conn) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }

        }
    }
```


如上代码 DBdistributedLock 类封装了分布式锁实现，其构造函数需要传递一个数据源。其中 lock 方法就是加锁用的方法，其内部代码 3.1 首先从数据源获取一个数据库连接，然后设置事务自动提交为 false（也就是设置为手动提交事务），然后具体执行 CMD 对应的 SQL（也就是使用 for update 锁住记录），多个线程执行，只有一个线程能获取到行锁，其他线程阻塞到 `stmt.executeQuery()` 处。

当线程执行 3.1 获取到记录的行锁后会执行代码 3.2，3.2 执行传递的 callback 的业务逻辑（也就是需要在锁内执行的代码），业务执行完毕后 执行 3.3、commit 提交事务，这意味着当前线程释放了获取的锁，这时候被阻塞的线程会竞争获取该锁。

这里需要注意的是必须设置为事务为手动提交，这保证了：

```
rs = stmt.executeQuery();

//3.2do business thing
callBack.doAction();

//3.3release lock
conn.commit();
```

是在一个事务中执行的，是原子性的。如果不设置为手动提交，则多个线程执行完 `stmt.executeQuery()` 后就释放了行锁，那么多个线程就可以同时执行代码3.2，这显然是错误的。

下面我们来看看具体如何使用：

```
final DBdistributedLock bdistributedLock = new
DBdistributedLock(dataSource);
bdistributedLock.lock(new CallBack() {

                        @Override
                        public void doAction() {

System.out.println(Thread.currentThread().getName() + "beging do
somthing");
                            try {
                            //do business
                            } catch (InterruptedException e) {
                                // TODO Auto-generated catch
block
                                e.printStackTrace();
                            }

System.out.println(Thread.currentThread().getName() + "end do
somthing");

                        }
```


如上代码可知使用时候只需要创建的一个 DBdistributedLock 对象，然后调用其 Lock 方法，并且传递一个 callback 的实现，实现方法里具体做业务，这些业务是受分布式锁保护的，拥有原子性。

在每个 JVM 进程中可以创建一个 DBdistributedLock 的实例，JVM 进程内公用同一个实例，不同 JVM 进程维护自己的 DBdistributedLock 实例，由于加锁时候多个线程是共同竞争同一个行锁，所以实现了分布式锁。

四、使用 Redis 实现分布式锁

在 JUC 包中除了阻塞锁外还有一种叫 CAS 的无阻塞锁（具体可以参考：Java 并发编程之美：并发编程基础晋级篇），CAS 操作本身是原子性的，多个线程操作同一个变量的 CAS 时候只有一个线程能进行 CAS 成功，失败的线程接下来那么使用乐观锁机制直接失败要么使用自旋方式使用 CPU 资源重复进行 CAS 尝试。

那么在分布式锁的实现中我们也可以使用类似的方式，比如 Redis 提供了一个保证原子性的 setnx 函数，多个线程调用该函数操作同一个 key 的时候，只有一个线程会返回 OK，其他线程返回 null，那么多个 JVM 中的线程同时设置同一个 key 时候只有一个 JVM 里面的一个线程可以返回 OK，返回 OK 的线程就相当于获取了全局锁，返回 null 的线程则可以选择自旋重试。获取到锁的线程使用完毕后调用 del 函数删除对应的 key，然后自旋的线程就会有一个返回 OK…

在讲解具体实现前，先来讲解下 Redis 的 set,get,del,eval 函数，本文使用的 Redis 版本：


```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```

函数讲解：

1） String set(final String key, final String value, final String nxxx, final String expx,final int time)

- 在 Redis 中支持 kv 存储，这里 key 就是 kv 中的 key,value 就是 kv 中的 value。
- 从 Redis 2.6.12 版本开始， SET 命令的行为可以通过一系列参数来修改；其中 nxxx 的枚举值为 NX 和 XX，模式 NX 意思是说如果 key 不存在则插入该 key 对应的 value 并返回 OK，否者什么都不做返回 null；XX 意思是只在 key 已经存在时，才对 key 进行设置操作, 否者 null, 如果已经存在 key 并且进行了多次设置，则最终 key 对应的值为最后一次设置的值。
- 其中 expx 的枚举值为 EX 和 PX，当为 EX 时候标示设置超时时间为 time 秒，当为 PX 时候标示设置超时时间为 time 毫秒。

为了实现 CAS 的效果，本文选用 nxxx 为 NX 模式，因为这种模式下当多个线程设置同一个 key 时只有一个线程会返回 OK，其他线程则会返回 null，返回 OK 的线程标示获取到了分布式锁，返回 null 的则视为获取锁失败，则通过自旋来不断尝试获取。

然后 value 值使用请求 id 来标示，多个线程设置同一个 key 的时候对应的 value 值要不一样，这是为了保证只有获取到锁的线程才应该释放锁，下面会具体讲解。

2） String get(final String key)

- 获取 key 对应的 value 值，key 不存在则返回 null

3）Long del(String key)

- 删除 key 对应的 value 值, 如果 key 存在则返回 1，否者返回 0

由于需要保证只有获取锁的线程才能释放锁，所以需要在获取锁时候调用 set 方法传递一个唯一的 value 值，上面说了，可以传递请求 id; 然后在释放锁的时候需要调用 get 方法获取 key 对应的 value，如果 value 值等于当前线程的请求 id 则说明是当前线程获取的锁，则调用 del 方法删除该 key 对应的 value，这就相当于当前线程释放了锁；如果 value 不等于当前线程的请求 id 则不做删除操作。

可见释放锁的操作需要调用 get 方法，然后 if 语句进行判断，判断 OK 然后调用 del 删除，而这三步并不是原子性的，如果不是原子性的会存在什么问题那?

假设线程 A 调用 set 方法设置 key 对应的 value 为 AA 成功， 则线程 A 获取到了锁，然后在执行完业务逻辑后，首先通过 get 方法获取 key 对应的 value，然后通过 if 语句判断为 true，假设在执行 del 方法前对应的 key 已经超时了，并且线程 B 调用 set 方法设置 key 对应的 value 为 BB 成功了，也就是线程 B 获取到了锁，但是这时候线程 A 开始执行 del 方法了，则会把线程 B 对应的 key 的值删除了（不同线程调用 set 的时候 key 一样），也就是释放了锁，这时候其他线程就会竞争到该锁。这明显是错误的。

Redis 有一个叫做 eval 的函数，支持 Lua 脚本执行，并且能够保证脚本执行的原子性，也就是在执行脚本期间，其它执行 redis 命令的线程都会被阻塞。这里解锁时候使用下面脚本：

```
if redis.call('get', KEYS[1]) == ARGV[1] then return
redis.call('del', KEYS[1]) else return 0 end
```

其中 `keys[1]` 为 unLock 方法传递的 key， `argv[1]` 为 unLock 方法传递的 requestId; 脚本 `redis.call('get', KEYS[1])` 的作用是获取 key 对应的 value 值，这里会返回通过 Lock 方法传递的 requetId，然后看当前传递的 RequestId 是否等于 key 对应的值，等于则说明当前要释放锁的线程就是获取锁的线程，则继续执行 `redis.call('del', KEYS[1])` 脚本，删除 key 对应的值。

原理讲解完了，下面我们来具体看代码：

```
package com.jiaduo.DistributedLock;

import java.util.Collections;

import redis.clients.jedis.Jedis;
```

```
import redis.clients.jedis.JedisPool;

public class DistributedLock {

    private static final String LOCK_SUCCESS = "OK";
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";
    private static final Long RELEASE_SUCCESS = 1L;

    private static void validParam(JedisPool jedisPool, String
lockKey, String requestId, int expireTime) {
        if (null == jedisPool) {
            throw new IllegalArgumentException("jedisPool obj is
null");
        }

        if (null == lockKey || "".equals(lockKey)) {
            throw new IllegalArgumentException("lock key  is
blank");
        }

        if (null == requestId || "".equals(requestId)) {
            throw new IllegalArgumentException("requestId is
blank");
        }

        if (expireTime < 0) {
            throw new IllegalArgumentException("expireTime is not
allowed less zero");
        }
    }

    /**
     *
     * @param jedis
     * @param lockKey
     * @param requestId
     * @param expireTime
     * @return
     */
    public static boolean tryLock(JedisPool jedisPool, String
lockKey, String requestId, int expireTime) {

        validParam(jedisPool, lockKey, requestId, expireTime);

        Jedis jedis = null;
        try {

            jedis = jedisPool.getResource();
            String result = jedis.set(lockKey, requestId,
SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, expireTime);
```

```
            if (LOCK_SUCCESS.equals(result)) {
                return true;
            }
        } catch (Exception e) {
            throw e;
        } finally {
            if (null != jedis) {
                jedis.close();
            }
        }

        return false;
    }

    /**
     *
     * @param jedis
     * @param lockKey
     * @param requestId
     * @param expireTime
     */
    public static void lock(JedisPool jedisPool, String lockKey,
String requestId, int expireTime) {

        validParam(jedisPool, lockKey, requestId, expireTime);

        while (true) {
            if (tryLock(jedisPool, lockKey, requestId,
expireTime)) {
                return;
            }
        }
    }

    /**
     *
     * @param jedis
     * @param lockKey
     * @param requestId
     * @return
     */
        public static void unLock(JedisPool jedisPool, String
lockKey, String requestId) {

        validParam(jedisPool, lockKey, requestId, 1);

        String script = "if redis.call('get', KEYS[1]) == ARGV[1]
then return redis.call('del', KEYS[1]) else return 0 end";

        Jedis jedis = null;
        try {
```

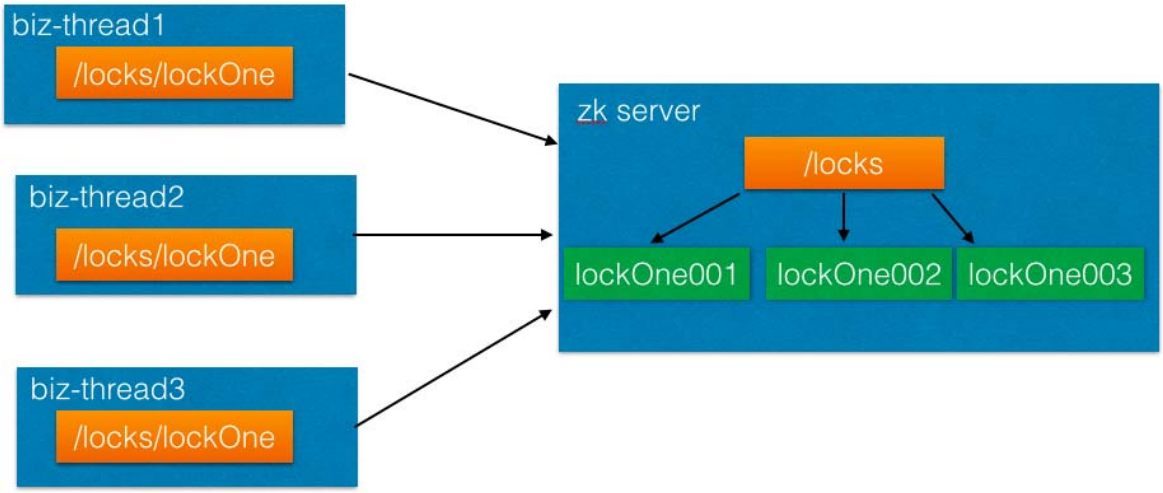
```
            jedis = jedisPool.getResource();
            Object result = jedis.eval(script,
Collections.singletonList(lockKey),
                    Collections.singletonList(requestId));

            if (RELEASE_SUCCESS.equals(result)) {
                System.out.println("relese lock ok ");
            }

        } catch (Exception e) {
            throw e;
        } finally {
            if (null != jedis) {
                jedis.close();
            }
        }

    }
}
```

- 通过 tryLock 方法尝试获取锁，内部是具体调用 Redis 的 set 方法，多个线程同时调用 tryLock 时候, 会同时调用 set 方法，但是 set 方法本身是保证原子性的，对应同一个 key 来说，多个线程调用 set 方法时候只有一个线程返回 OK，其它线程因为 key 已经存在会返回 null，返回 OK 的线程就相当与获取到了锁，其它返回 null 的线程则相当于获取锁失败。
- 通过 lock 方法让使用 tryLock 获取锁失败的线程本地自旋转重试获取锁，这类似 JUC 里面的 CAS。
- 通过 unLock 方法使用 redis 的 eval 函数传递 lua 脚本来保证操作的原子性。

四、使用 Zookeeper 来实现分布式锁

在 ZK 中是使用文件目录的格式存放节点内容，其中节点类型分为：

- 持久节点（PERSISTENT）：节点创建后，一直存在，直到主动删除了该节点。
- 临时节点（EPHEMERAL）：生命周期和客户端会话绑定，一旦客户端会话失效，这个节点就会自动删除。
- 序列节点（SEQUENTIAL）：多个线程创建同一个顺序节点时候，每个线程会得到一个带有编号的节点，节点编号是递增不重复的，如下图：



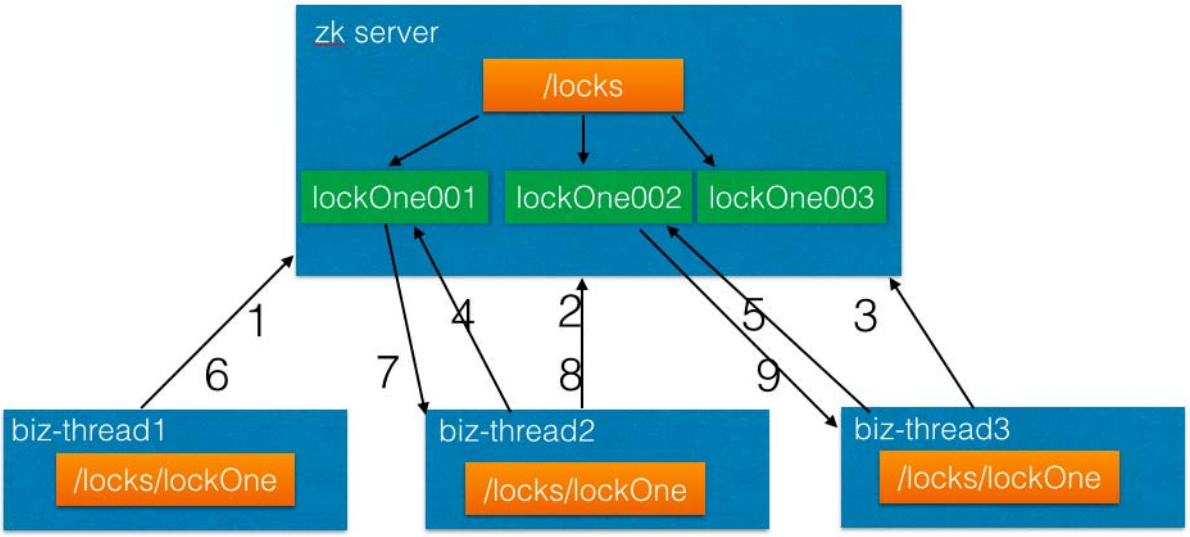
如上图，三个线程分别创建路径为 /locks/lockOne 的节点，可知在 ZK 服务器端会在根路径 locks下创建三个 lockOne 节点，并且器编号是唯一递增的。

具体在节点创建过程中，可以混合使用上面三种模式，比如创建临时顺序节点（EPHEMERAL_SEQUENTIAL），这里我们就使用临时顺序节点来实现分布式锁。

分布式锁实现步骤，每个想要获取锁的线程都要执行下面步骤：

- 创建临时顺序节点，比如 /locks/lockOne，假设返回结果为 /locks/lockOne000000000*。
- 获取 /locks下所有孩子节点，用自己创建的节点 /locks/lockOne000000000* 的序号 lockOne000000000* 与所有子节点比较，看看自己是不是编号最小的。如果是最小的则就相当于获取到了锁;如果自己不是最小的，则从所有子节点里面获取比自己次小的一个节点，然后设置监听该节点的事件，然后挂起当前线程。
- 当最小编号的线程获取锁，处理完业务后删除自己对应的节点，删除后会激活比自己大一号的节点的线程从阻塞变为运行态，被激活的线程应该就是当前 node 序列号最小的了，然后就会获取到锁。

整个过程是一个类似循环监听的模式：



- 如上图当三个线程启动时候分别执行步骤（1）（2）（3），分别在 zk 服务器上创建自己的顺序节点。
- 由于线程1创建的节点的序列最小，所以线程1获取到了锁；线程 2 发现自己不是最小的所以首先注册监听线程1创建的 LockOne001 节点的事件，然后挂起自己；线程 3 发现自己不是最小的所以首先注册监听线程 2 创建的 LockOne002 节点的事件，然后挂起自己。
- 当线程 1 获取锁后，执行完了业务逻辑后，会执行步骤 6 删除创建的 LockOne001 节点，删除后线程 2 由于设置了对 LockOne1 的监听，所以 zk 服务器会给线程 2 所在机器发送事件，接受事件后发现是 LockOne1 的删除事件，则会激活线程 2，这时候线程 2 就获取到了锁。
- 当线程 2 获取锁后，执行完了业务逻辑后，会执行步骤 8 删除创建的 LockOne002 节点，删除后线程3由于设置了对 LockOne2 的监听，所以 zk 服务器会给线程 3 所在机器发送事件，接受事件后发现是 LockOne2 的删除事件，则会激活线程 3，这时候线程 3 就获取到了锁。

下面我们看看代码实现：

```
public class ZookeeperDistributedLock {
    public final static Joiner j = Joiner.on("|").useForNull("");

    //zk客户端
    private ZooKeeper zk;
    //zk是一个目录结构，root为最外层目录
    private String root = "/locks";
    //锁的名称
    private String lockName;
    //当前线程创建的序列node
    private ThreadLocal<String> nodeId = new ThreadLocal<>();
    //用来同步等待zkclient链接到了服务端
```

```
    private CountDownLatch connectedSignal = new
CountDownLatch(1);
    private final static int sessionTimeout = 3000;
    private final static byte[] data= new byte[0];

    public ZookeeperDistributedLock(String config, String
lockName) {
        this.lockName = lockName;

        try {
            zk = new ZooKeeper(config, sessionTimeout, new
Watcher() {

                @Override
                public void process(WatchedEvent event) {
                    // 建立连接
                    if (event.getState() ==
KeeperState.SyncConnected) {
                        connectedSignal.countDown();
                    }
                }

            });

            connectedSignal.await();
            Stat stat = zk.exists(root, false);
            if (null == stat) {
                // 创建根节点
                zk.create(root, data,
ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    class LockWatcher implements Watcher {
        private CountDownLatch latch = null;

        public LockWatcher(CountDownLatch latch) {
            this.latch = latch;
        }

        @Override
        public void process(WatchedEvent event) {

            if (event.getType() == Event.EventType.NodeDeleted)
                latch.countDown();
        }
    }

    public void lock() {
```

```
        try {

            // 创建临时子节点
            String myNode = zk.create(root + "/" + lockName ,
data, ZooDefs.Ids.OPEN_ACL_UNSAFE,
                    CreateMode.EPHEMERAL_SEQUENTIAL);


System.out.println(j.join(Thread.currentThread().getName() +
myNode, "created"));

            // 取出所有子节点
            List<String> subNodes = zk.getChildren(root, false);
            TreeSet<String> sortedNodes = new TreeSet<>();
            for(String node :subNodes) {
                sortedNodes.add(root +"/" +node);
            }

            String smallNode = sortedNodes.first();
            String preNode = sortedNodes.lower(myNode);

            if (myNode.equals( smallNode)) {
                // 如果是最小的节点,则表示取得锁

System.out.println(j.join(Thread.currentThread().getName(),
myNode, "get lock"));
                this.nodeId.set(myNode);
                return;
            }

            CountDownLatch latch = new CountDownLatch(1);
            Stat stat = zk.exists(preNode, new
LockWatcher(latch));//
            // 判断比自己小一个数的节点是否存在,存在则注册监听
            if (stat != null) {

System.out.println(j.join(Thread.currentThread().getName(),
myNode,
                        " waiting for " + root + "/" + preNode +
" released lock"));

                latch.await();// 等待，这里应该一直等待其他线程释放锁
                nodeId.set(myNode);
                latch = null;
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

    }

    public void unlock() {
```

```
            try {

    System.out.println(j.join(Thread.currentThread().getName(),
    nodeId.get(), "unlock "));
                if (null != nodeId) {
                    zk.delete(nodeId.get(), -1);
                }
                nodeId.remove();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (KeeperException e) {
                e.printStackTrace();
            }
        }

    }
```

如上代码，ZookeeperDistributedLock 的构造函数里面首先创建 zkclient，并且注册了监听事件，然后调用 connectedSignal.await() 挂起当前线程。

当 zkclient 链接到服务器后，会给监听器发送 SyncConnected 事件，监听器判断当前链接已经建立了，则调用 connectedSignal.countDown(); 激活当前线程，然后创建 locks 根节点。

获取锁的方法 lock，内部首先创建 /locks/lockOne 的顺序临时节点，然后获取 /locks 下所有的孩子节点，并对子节点进行排序，然后判断自己是不是最小的编号，如果是直接返回 true 标示获取锁成功。

否者看比自己小一个号的节点是否存在，存在则注册该节点的事件，然后挂起当前线程，等待比自己小一个数的节点释放锁后发送节点删除事件，事件里面激活当前线程。

释放锁的方法 unlock 比较简单，就是简单的删除获取锁时候创建的节点。

五、三种实现方式对比

使用数据库悲观锁来做分布式锁由于使用数据库自带的行锁机制，优点是实现比较简单；但是缺点也很明显，当高并发情况大量线程同时竞争时候，只有一个线程会获取到行锁，其它线程必须挂起等待，更严重的是每个挂起的线程都持有一个数据库的链接，在高并发下，数据库链接可能会被占用完，从而不能进行正常业务的访问；另外使用这种方式没有手动设置超时自动释放锁的概念，等待的线程要么获取到了锁，那么等待获取行锁超时后返回（这里的超时，是数据库 SQL 执行的超时时间）；所以这种实现在并发量过大的时候不是很适用。

使用 Redis 来实现分布式锁优点是实现简单，并且获取锁的 setnx 方法使用 cas 算法来判断获取锁是否成功，吞吐量不错；另外 setnx 方法自带了超时参数，这可以有效避免当一个线程获取到锁后，在释放锁前机器挂了后，其他线程一直阻塞到获取锁的情况，等

超时时间过了，锁会被自动释放；缺点也很明显，本文例子获取锁时候是类似 CAS 自旋重试的，在高并发情况下会造成大量线程共同竞争锁时候的本地自旋，这很像 JUC 中的 AtomicLong 一样，在高并发下多个线程竞争同一个资源时候造成大量线程占用 cpu 进行重试操作。这时候其实可以随机生成一个等待时间，等时间到后在进行重试，以减少潜在的同时对一个资源进行竞争的并发量；另外本文使用的是最简单的 Redis 单实例实现，如果单实例挂了，也会存在问题，大家可以下去学习下多实例的情况，参考链接。

使用 Zookeeper 实现分布式锁优点是可以对节点进行监听，多个线程获取锁时候没有获取到锁的线程不需要本地自旋重试，而是挂起自己，等待获取锁的线程释放锁后发送事件激活自己；由于线程阻塞自己使用的是 JUC 包的 CountDownLatch，在调用 await 的时候是可以添加超时时间的（本文并没有加这个参数），所以 zk 方式也可以在实现获取锁时候超时候自动返回；缺点是使用 zk 实现比较重，实现起来不是那么简单，其实 Apache Curator 对 zk 进行了封装，大家下去可以研究下：https://curator.apache.org/

六、总结

本文讲解了常用的实现分布式锁的三种方式：数据库、Redis、Zookeeper；使用最简单的方式实现了分布式锁，虽然这些锁的实现要想在生产环境中使用还需要一些改进；但是本文对了解其原理实现，以及应对面试还是绰绰有余的，希望读者能够在结合本文原理的基础上，进入深入思考，另外希望读者能够下去研究下第 5 节推荐的两个开源的分布式锁实现。