## **公平锁和可重入锁的原理**

最经典的分布式锁是可重入的公平锁。什么是可重入的公平锁呢？直接讲解的概念和原理，会比较抽象难懂，还是从具体的实例入手吧！这里用一个简单的故事来类比，估计就简单多了。

故事发生在一个没有自来水的古代，在一个村子有一口井，水质非常的好，村民们都抢着取井里的水。井就那么一口，村里的人很多，村民为争抢取水打架斗殴，甚至头破血流。

问题总是要解决，于是村长绞尽脑汁，最终想出了一个凭号取水的方案。井边安排一个看井人，维护取水的秩序。取水秩序很简单：

（1）取水之前，先取号；

（2）号排在前面的，就可以先取水；

（3）先到的排在前面，那些后到的，一个一个挨着，在井边排成一队。

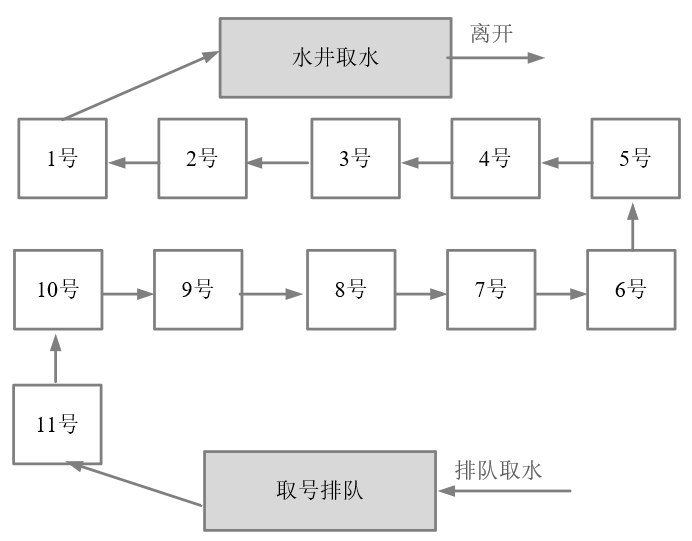
取水示意图，如图10-3所示。  


图10-3 排队取水示意图

这种排队取水模型，就是一种锁的模型。排在最前面的号，拥有取水权，就是一种典型的独占锁。另外，先到先得，号排在前面的人先取到水，取水之后就轮到下一个号取水，挺公平的，说明它是一种公平锁。

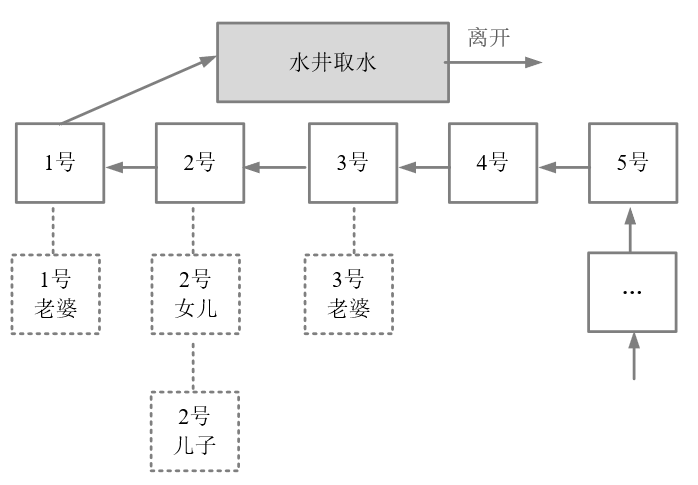
什么是可重入锁呢？  
假定，取水时以家庭为单位，家庭的某人拿到号，其他的家庭成员过来打水，这时候不用再取号，如图10-4所示。  


图10-4 同一家庭的人不需要重复排队

图10-4中，排在1号的家庭，老公取号，假设其老婆来了，直接排第一个，正所谓妻凭夫贵。再看上图的2号，父亲正在打水，假设其儿子和女儿也到井边了，直接排第二个，所谓子凭父贵。总之，如果取水时以家庭为单位，则同一个家庭，可以直接复用排号，不用从后面排起重新取号。

以上这个故事模型中，取号一次，可以用来多次取水，其原理为可重入锁的模型。在重入锁模型中，一把独占锁，可以被多次锁定，这就叫做可重入锁。

## **ZooKeeper分布式锁的原理**

理解了经典的公平可重入锁的原理后，再来看在分布式场景下的公平可重入锁的原理。通过前面的分析，基本可以判定：ZooKeeper  
的临时顺序节点，天生就有一副实现分布式锁的胚子。为什么呢？

（一） ZooKeeper的每一个节点，都是一个天然的顺序发号器。

在每一个节点下面创建临时顺序节点（EPHEMERAL\_SEQUENTIAL）类型，新的子节点后面，会加上一个次序编号，而这个生成的次序编号，是上一个生成的次序编号加一。

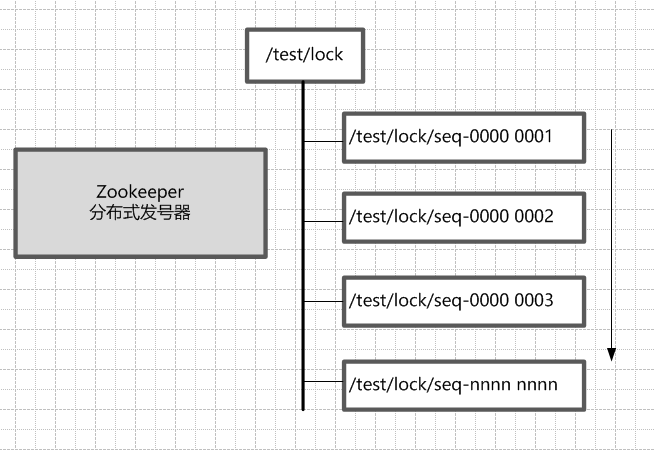
例如，有一个用于发号的节点“/test/lock”为父亲节点，可以在这个父节点下面创建相同前缀的临时顺序子节点，假定相同的前缀为“/test/lock/seq-”。第一个创建的子节点基本上应该为/test/lock/seq-0000000000，下一个节点则为/test/lock/seq-0000000001，依次类推，如果10-5所示。  


图10-5 Zookeeper临时顺序节点的天然的发号器作用

（二） ZooKeeper节点的递增有序性，可以确保锁的公平

一个ZooKeeper分布式锁，首先需要创建一个父节点，尽量是持久节点（PERSISTENT类型），然后每个要获得锁的线程，都在这个节点下创建个临时顺序节点。由于ZK节点，是按照创建的次序，依次递增的。

为了确保公平，可以简单的规定：编号最小的那个节点，表示获得了锁。所以，每个线程在尝试占用锁之前，首先判断自己是排号是不是当前最小，如果是，则获取锁。

（三）ZooKeeper的节点监听机制，可以保障占有锁的传递有序而且高效

每个线程抢占锁之前，先尝试创建自己的ZNode。同样，释放锁的时候，就需要删除创建的Znode。创建成功后，如果不是排号最小的节点，就处于等待通知的状态。等谁的通知呢？不需要其他人，只需要等前一个Znode  
的通知就可以了。前一个Znode删除的时候，会触发Znode事件，当前节点能监听到删除事件，就是轮到了自己占有锁的时候。第一个通知第二个、第二个通知第三个，击鼓传花似的依次向后。

ZooKeeper的节点监听机制，能够非常完美地实现这种击鼓传花似的信息传递。具体的方法是，每一个等通知的Znode节点，只需要监听（linsten）或者监视（watch）排号在自己前面那个，而且紧挨在自己前面的那个节点，就能收到其删除事件了。  
只要上一个节点被删除了，就进行再一次判断，看看自己是不是序号最小的那个节点，如果是，自己就获得锁。

另外，ZooKeeper的内部优越的机制，能保证由于网络异常或者其他原因，集群中占用锁的客户端失联时，锁能够被有效释放。一旦占用Znode锁的客户端与ZooKeeper集群服务器失去联系，这个临时Znode也将自动删除。排在它后面的那个节点，也能收到删除事件，从而获得锁。正是由于这个原因，在创建取号节点的时候，尽量创建临时znode  
节点，

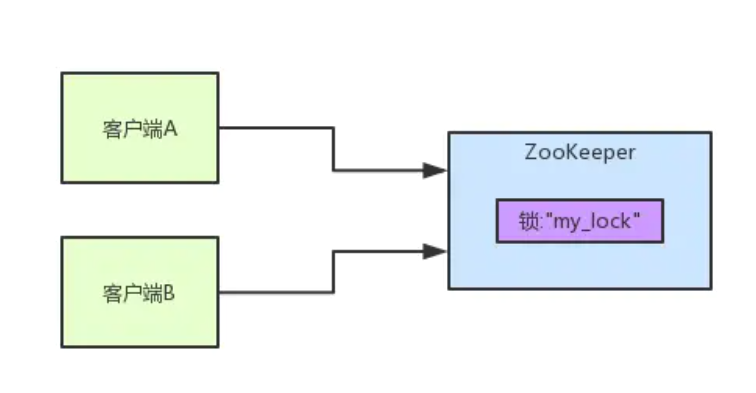
（四）ZooKeeper的节点监听机制，能避免羊群效应

ZooKeeper这种首尾相接，后面监听前面的方式，可以避免羊群效应。所谓羊群效应就是一个节点挂掉，所有节点都去监听，然后做出反应，这样会给服务器带来巨大压力，所以有了临时顺序节点，当一个节点挂掉，只有它后面的那一个节点才做出反应。

## **图解：分布式锁的抢占过程**

接下来我们一起来看看，多客户端获取及释放zk分布式锁的整个流程及背后的原理。

首先大家看看下面的图，如果现在有两个客户端一起要争抢zk上的一把分布式锁，会是个什么场景？



如果大家对zk还不太了解的话，建议先自行百度一下，简单了解点基本概念，比如zk有哪些节点类型等等。

参见上图。zk里有一把锁，这个锁就是zk上的一个节点。然后呢，两个客户端都要来获取这个锁，具体是怎么来获取呢？

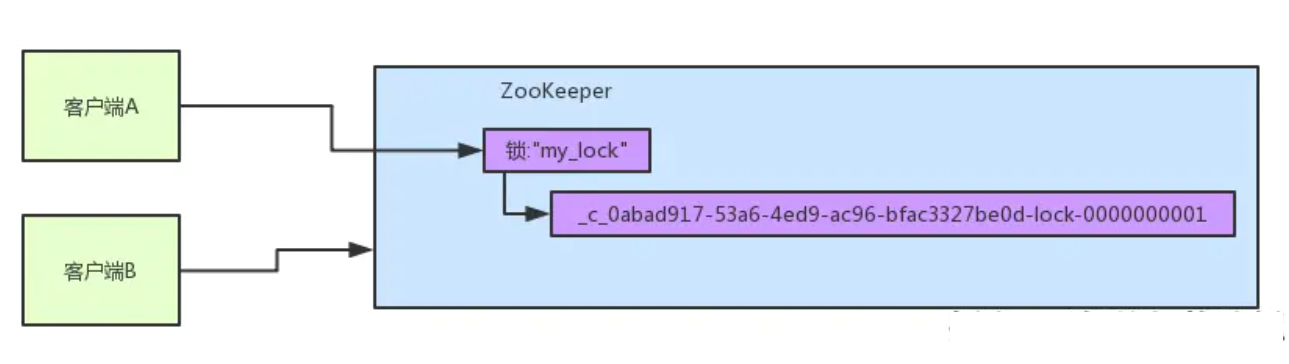
咱们就假设客户端A抢先一步，对zk发起了加分布式锁的请求，这个加锁请求是用到了zk中的一个特殊的概念，叫做“临时顺序节点”。

简单来说，就是直接在"my\_lock"这个锁节点下，创建一个顺序节点，这个顺序节点有zk内部自行维护的一个节点序号。

### **客户端A发起一个加锁请求**

比如说，第一个客户端来搞一个顺序节点，zk内部会给起个名字叫做：xxx-000001。然后第二个客户端来搞一个顺序节点，zk可能会起个名字叫做：xxx-000002。大家注意一下，最后一个数字都是依次递增的，从1开始逐次递增。zk会维护这个顺序。

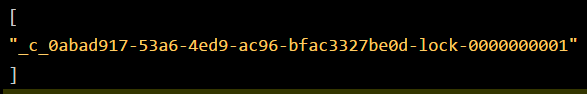
所以这个时候，假如说客户端A先发起请求，就会搞出来一个顺序节点，大家看下面的图，Curator框架大概会弄成如下的样子：



大家看，客户端A发起一个加锁请求，先会在你要加锁的node下搞一个临时顺序节点，这一大坨长长的名字都是Curator框架自己生成出来的。

然后，那个最后一个数字是"1"。大家注意一下，因为客户端A是第一个发起请求的，所以给他搞出来的顺序节点的序号是"1"。

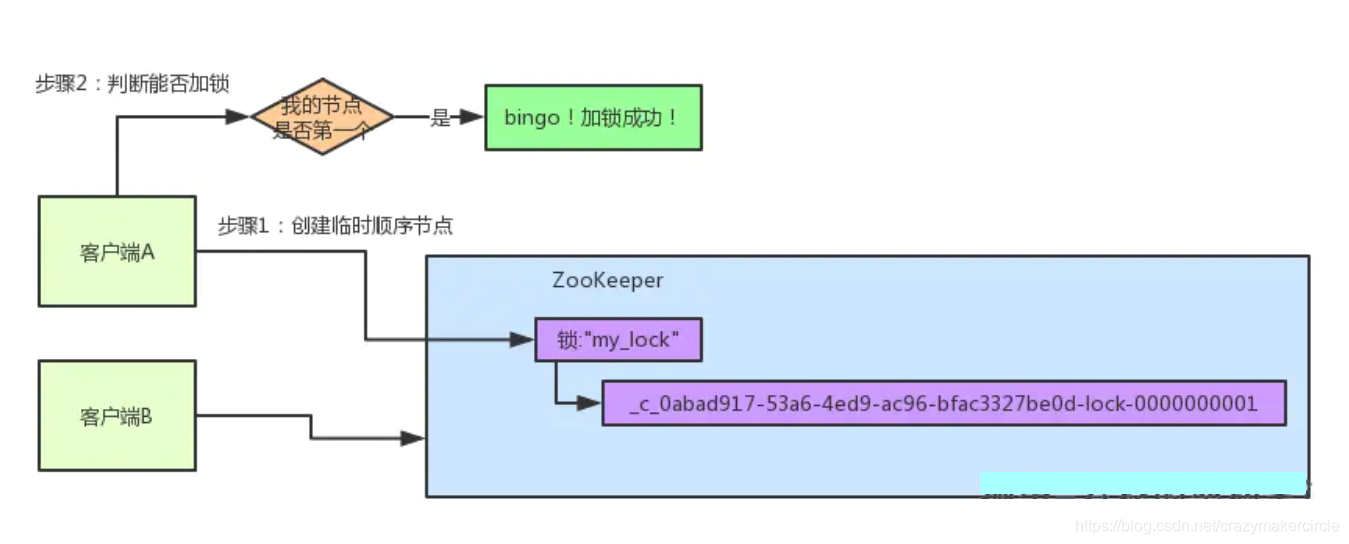
接着客户端A创建完一个顺序节点。还没完，他会查一下"my\_lock"这个锁节点下的所有子节点，并且这些子节点是按照序号排序的，这个时候他大概会拿到这么一个集合：



接着客户端A会走一个关键性的判断，就是说：唉！兄弟，这个集合里，我创建的那个顺序节点，是不是排在第一个啊？

如果是的话，那我就可以加锁了啊！因为明明我就是第一个来创建顺序节点的人，所以我就是第一个尝试加分布式锁的人啊！

bingo！加锁成功！大家看下面的图，再来直观的感受一下整个过程。

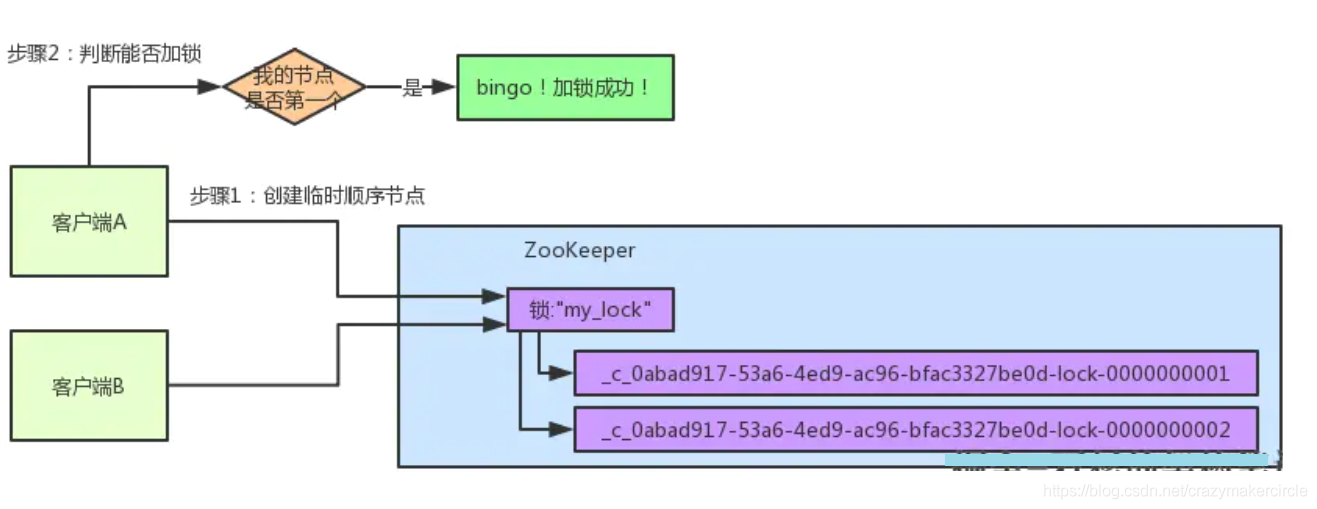


### **客户端B过来排队**

接着假如说，客户端A都加完锁了，客户端B过来想要加锁了，这个时候他会干一样的事儿：先是在"my\_lock"这个锁节点下创建一个临时顺序节点，此时名字会变成类似于：

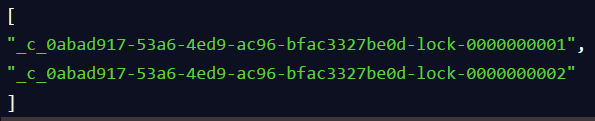
IMG_263

大家看看下面的图：



客户端B因为是第二个来创建顺序节点的，所以zk内部会维护序号为"2"。

接着客户端B会走加锁判断逻辑，查询"my\_lock"锁节点下的所有子节点，按序号顺序排列，此时他看到的类似于：



同时检查自己创建的顺序节点，是不是集合中的第一个？

明显不是啊，此时第一个是客户端A创建的那个顺序节点，序号为"01"的那个。所以加锁失败！

### **客户端B开启监听客户端A**

加锁失败了以后，客户端B就会通过ZK的API对他的顺序节点的上一个顺序节点加一个监听器。zk天然就可以实现对某个节点的监听。

如果大家还不知道zk的基本用法，可以百度查阅，非常的简单。客户端B的顺序节点是：

IMG_266

他的上一个顺序节点，不就是下面这个吗？

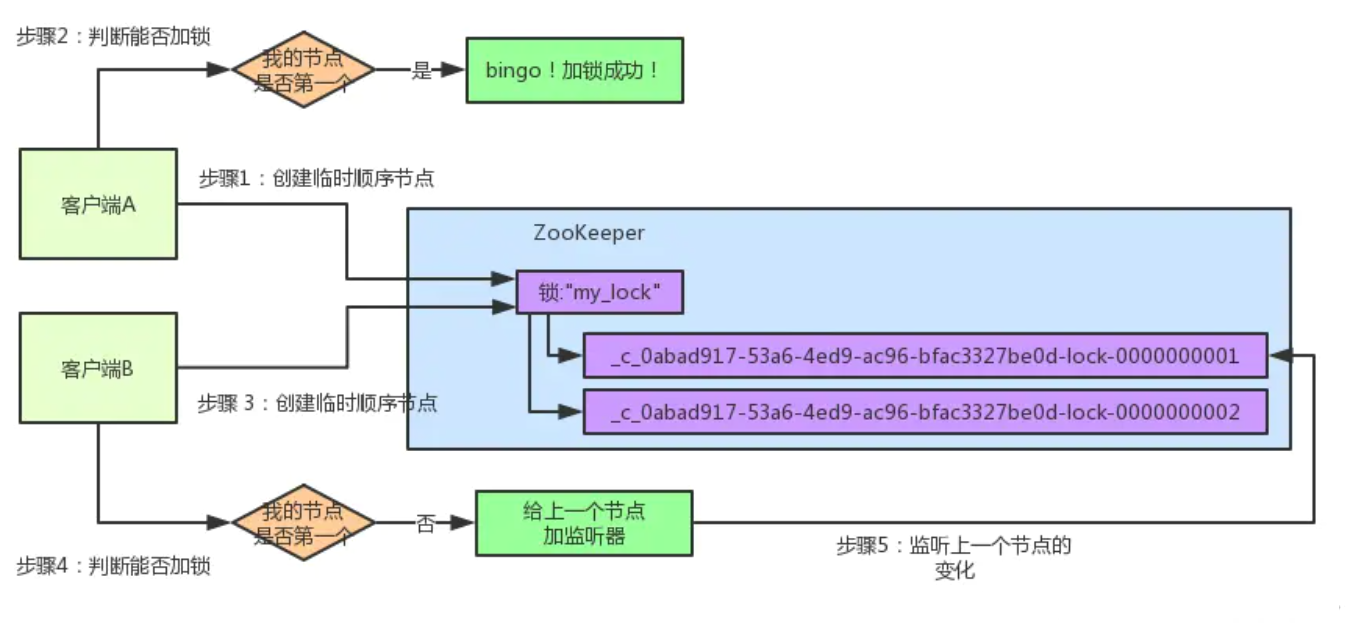
IMG_267

即客户端A创建的那个顺序节点！

所以，客户端B会对：

IMG_268

这个节点加一个监听器，监听这个节点是否被删除等变化！大家看下面的图。



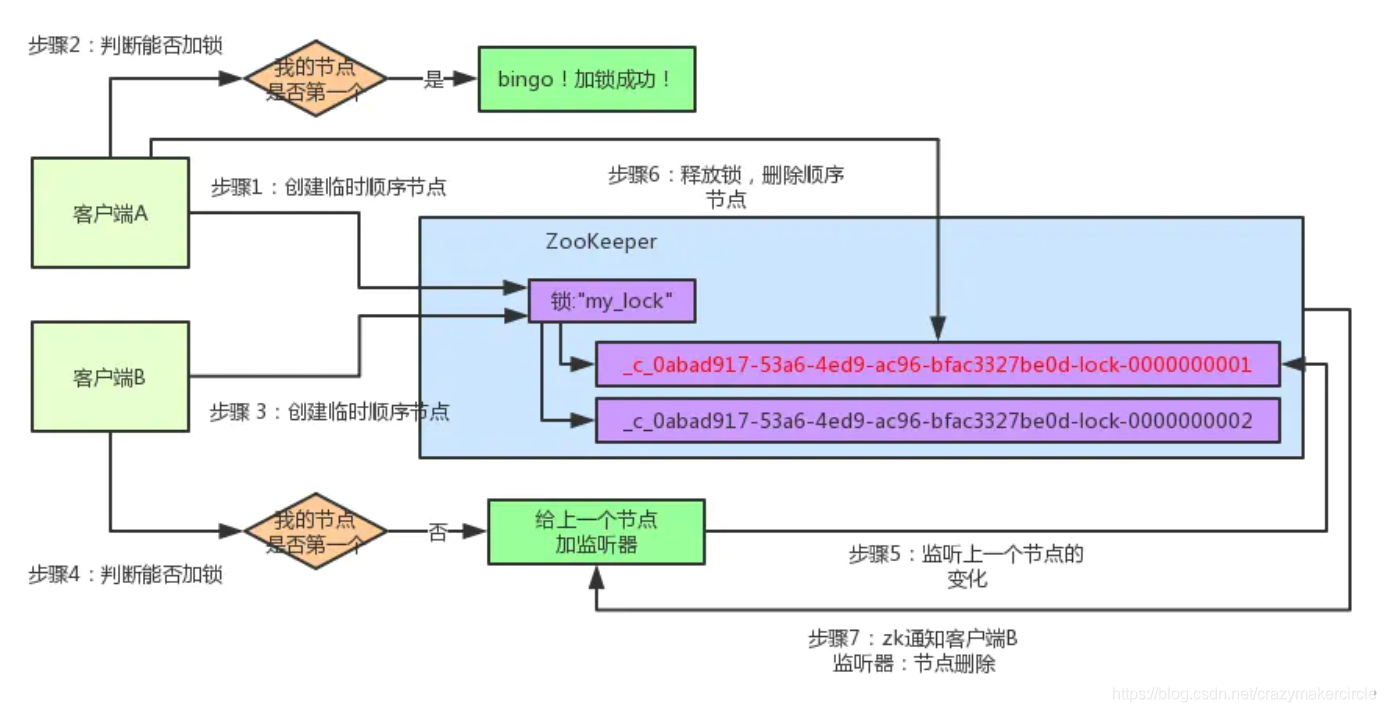
接着，客户端A加锁之后，可能处理了一些代码逻辑，然后就会释放锁。那么，释放锁是个什么过程呢？

其实很简单，就是把自己在zk里创建的那个顺序节点，也就是：

IMG_270

这个节点给删除。

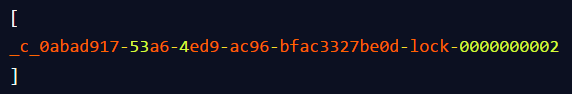
删除了那个节点之后，zk会负责通知监听这个节点的监听器，也就是客户端B之前加的那个监听器，说：兄弟，你监听的那个节点被删除了，有人释放了锁。



此时客户端B的监听器感知到了上一个顺序节点被删除，也就是排在他之前的某个客户端释放了锁。

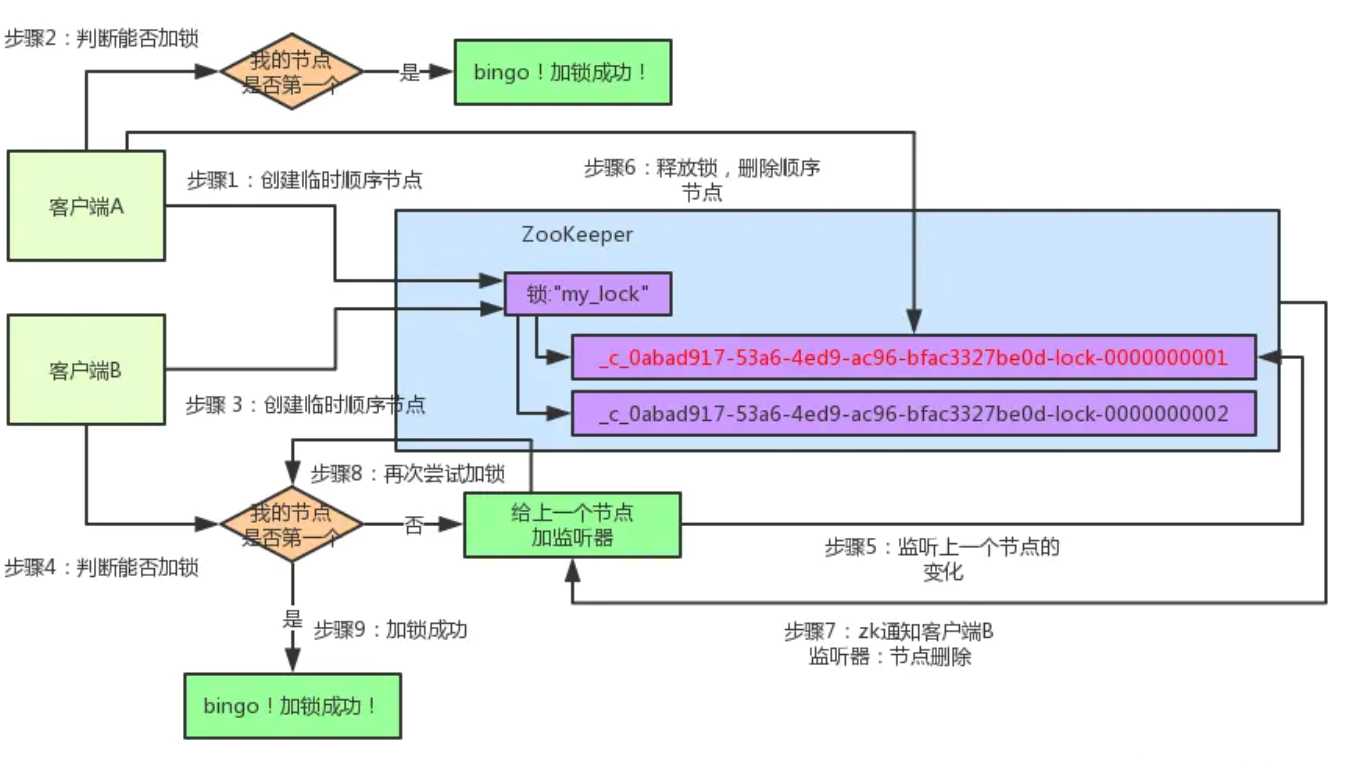
### **客户端B抢锁成功**

此时，就会通知客户端B重新尝试去获取锁，也就是获取"my\_lock"节点下的子节点集合，此时为：



集合里此时只有客户端B创建的唯一的一个顺序节点了！

然后呢，客户端B判断自己居然是集合中的第一个顺序节点，bingo！可以加锁了！直接完成加锁，运行后续的业务代码即可，运行完了之后再次释放锁。



## **分布式锁的基本实现**

接下来就是基于ZooKeeper，实现一下分布式锁。首先，定义了一个锁的接口Lock，很简单，仅仅两个抽象方法：一个加锁方法，一个解锁方法。Lock接口的代码如下：

package com.crazymakercircle.zk.distributedLock;

/\*\*

\* create by 尼恩 @ 疯狂创客圈

\*\*/public interface Lock {

/\*\*

\* 加锁方法

\*

\* @return 是否成功加锁

\*/

boolean lock() throws Exception;

/\*\*

\* 解锁方法

\*

\* @return 是否成功解锁

\*/

boolean unlock();

}

使用ZooKeeper实现分布式锁的算法，有以下几个要点：

（1）一把分布式锁通常使用一个Znode节点表示；如果锁对应的Znode节点不存在，首先创建Znode节点。这里假设为“/test/lock”，代表了一把需要创建的分布式锁。

（2）抢占锁的所有客户端，使用锁的Znode节点的子节点列表来表示；如果某个客户端需要占用锁，则在“/test/lock”下创建一个临时有序的子节点。

这里，所有临时有序子节点，尽量共用一个有意义的子节点前缀。

比如，如果子节点的前缀为“/test/lock/seq-”，则第一次抢锁对应的子节点为“/test/lock/seq-000000000”，第二次抢锁对应的子节点为“/test/lock/seq-000000001”，以此类推。

再比如，如果子节点前缀为“/test/lock/”，则第一次抢锁对应的子节点为“/test/lock/000000000”，第二次抢锁对应的子节点为“/test/lock/000000001”，以此类推，也非常直观。

（3）如果判定客户端是否占有锁呢？  
很简单，客户端创建子节点后，需要进行判断：自己创建的子节点，是否为当前子节点列表中序号最小的子节点。如果是，则认为加锁成功；如果不是，则监听前一个Znode子节点变更消息，等待前一个节点释放锁。

（4）一旦队列中的后面的节点，获得前一个子节点变更通知，则开始进行判断，判断自己是否为当前子节点列表中序号最小的子节点，如果是，则认为加锁成功；如果不是，则持续监听，一直到获得锁。

（5）获取锁后，开始处理业务流程。完成业务流程后，删除自己的对应的子节点，完成释放锁的工作，以方面后继节点能捕获到节点变更通知，获得分布式锁。

## **实战：加锁的实现**

Lock接口中加锁的方法是lock（）。lock（）方法的大致流程是：首先尝试着去加锁，如果加锁失败就去等待，然后再重复。

### **1．lock（）方法的实现代码**

lock（）方法加锁的实现代码，大致如下：

package com.crazymakercircle.zk.distributedLock;

import com.crazymakercircle.zk.ZKclient;import lombok.extern.slf4j.Slf4j;import org.apache.curator.framework.CuratorFramework;import org.apache.zookeeper.WatchedEvent;import org.apache.zookeeper.Watcher;

import java.util.Collections;import java.util.List;import java.util.concurrent.CountDownLatch;import java.util.concurrent.TimeUnit;import java.util.concurrent.atomic.AtomicInteger;

/\*\*

\* create by 尼恩 @ 疯狂创客圈

\*\*/@Slf4jpublic class ZkLock implements Lock {

//ZkLock的节点链接

private static final String ZK\_PATH = "/test/lock";

private static final String LOCK\_PREFIX = ZK\_PATH + "/";

private static final long WAIT\_TIME = 1000;

//Zk客户端

CuratorFramework client = null;

private String locked\_short\_path = null;

private String locked\_path = null;

private String prior\_path = null;

final AtomicInteger lockCount = new AtomicInteger(0);

private Thread thread;

public ZkLock() {

ZKclient.instance.init();

synchronized (ZKclient.instance) {

if (!ZKclient.instance.isNodeExist(ZK\_PATH)) {

ZKclient.instance.createNode(ZK\_PATH, null);

}

}

client = ZKclient.instance.getClient();

}

@Override

public boolean lock() {//可重入，确保同一线程，可以重复加锁

synchronized (this) {

if (lockCount.get() == 0) {

thread = Thread.currentThread();

lockCount.incrementAndGet();

} else {

if (!thread.equals(Thread.currentThread())) {

return false;

}

lockCount.incrementAndGet();

return true;

}

}

try {

boolean locked = false;//首先尝试着去加锁

locked = tryLock();

if (locked) {

return true;

}

//如果加锁失败就去等待

while (!locked) {

await();

//获取等待的子节点列表

List<String> waiters = getWaiters();//判断，是否加锁成功

if (checkLocked(waiters)) {

locked = true;

}

}

return true;

} catch (Exception e) {

e.printStackTrace();

unlock();

}

return false;

}

//...省略其他的方法

}

### **2．tryLock（）尝试加锁**

尝试加锁的tryLock方法是关键，做了两件重要的事情：

（1）创建临时顺序节点，并且保存自己的节点路径

（2）判断是否是第一个，如果是第一个，则加锁成功。如果不是，就找到前一个Znode节点，并且保存其路径到prior\_path。

尝试加锁的tryLock方法，其实现代码如下：

/\*\*

\* 尝试加锁

\* @return 是否加锁成功

\* @throws Exception 异常

\*/

private boolean tryLock() throws Exception {

//创建临时Znode

locked\_path = ZKclient.instance

.createEphemeralSeqNode(LOCK\_PREFIX);

//然后获取所有节点

List<String> waiters = getWaiters();

if (null == locked\_path) {

throw new Exception("zk error");

}

//取得加锁的排队编号

locked\_short\_path = getShortPath(locked\_path);

//获取等待的子节点列表，判断自己是否第一个

if (checkLocked(waiters)) {

return true;

}

// 判断自己排第几个

int index = Collections.binarySearch(waiters, locked\_short\_path);

if (index < 0) { // 网络抖动，获取到的子节点列表里可能已经没有自己了

throw new Exception("节点没有找到: " + locked\_short\_path);

}

//如果自己没有获得锁，则要监听前一个节点

prior\_path = ZK\_PATH + "/" + waiters.get(index - 1);

return false;

}

private String getShortPath(String locked\_path) {

int index = locked\_path.lastIndexOf(ZK\_PATH + "/");

if (index >= 0) {

index += ZK\_PATH.length() + 1;

return index <= locked\_path.length() ? locked\_path.substring(index) : "";

}

return null;

}

创建临时顺序节点后，其完整路径存放在locked\_path成员中；另外还截取了一个后缀路径，放在  
locked\_short\_path成员中，后缀路径是一个短路径，只有完整路径的最后一层。为什么要单独保存短路径呢？  
因为，在获取的远程子节点列表中的其他路径返回结果时，返回的都是短路径，都只有最后一层路径。所以为了方便后续进行比较，也把自己的短路径保存下来。

创建了自己的临时节点后，调用checkLocked方法，判断是否是锁定成功。如果锁定成功，则返回true；如果自己没有获得锁，则要监听前一个节点，此时需要找出前一个节点的路径，并保存在  
prior\_path  
成员中，供后面的await（）等待方法去监听使用。在进入await（）等待方法的介绍前，先说下checkLocked  
锁定判断方法。

### **3．checkLocked（）检查是否持有锁**

在checkLocked（）方法中，判断是否可以持有锁。判断规则很简单：当前创建的节点，是否在上一步获取到的子节点列表的第一个位置：

（1）如果是，说明可以持有锁，返回true，表示加锁成功；

（2）如果不是，说明有其他线程早已先持有了锁，返回false。

checkLocked（）方法的代码如下：

private boolean checkLocked(List<String> waiters) {

//节点按照编号，升序排列

Collections.sort(waiters);

// 如果是第一个，代表自己已经获得了锁

if (locked\_short\_path.equals(waiters.get(0))) {

log.info("成功的获取分布式锁,节点为{}", locked\_short\_path);

return true;

}

return false;

}

checkLocked方法比较简单，将参与排队的所有子节点列表，从小到大根据节点名称进行排序。排序主要依靠节点的编号，也就是后Znode路径的10位数字，因为前缀都是一样的。排序之后，做判断，如果自己的locked\_short\_path编号位置排在第一个，如果是，则代表自己已经获得了锁。如果不是，则会返回false。

如果checkLocked（）为false，外层的调用方法，一般来说会执行await（）等待方法，执行夺锁失败以后的等待逻辑。

### **4．await（）监听前一个节点释放锁**

await（）也很简单，就是监听前一个ZNode节点（prior\_path成员）的删除事件，代码如下：

private void await() throws Exception {

if (null == prior\_path) {

throw new Exception("prior\_path error");

}

final CountDownLatch latch = new CountDownLatch(1);

//订阅比自己次小顺序节点的删除事件

Watcher w = new Watcher() {

@Override

public void process(WatchedEvent watchedEvent) {

System.out.println("监听到的变化 watchedEvent = " + watchedEvent);

log.info("[WatchedEvent]节点删除");

latch.countDown();

}

};

client.getData().usingWatcher(w).forPath(prior\_path);/\*

//订阅比自己次小顺序节点的删除事件

TreeCache treeCache = new TreeCache(client, prior\_path);

TreeCacheListener l = new TreeCacheListener() {

@Override

public void childEvent(CuratorFramework client,

TreeCacheEvent event) throws Exception {

ChildData data = event.getData();

if (data != null) {

switch (event.getType()) {

case NODE\_REMOVED:

log.debug("[TreeCache]节点删除, path={}, data={}",

data.getPath(), data.getData());

latch.countDown();

break;

default:

break;

}

}

}

};

treeCache.getListenable().addListener(l);

treeCache.start();\*/

latch.await(WAIT\_TIME, TimeUnit.SECONDS);

}

首先添加一个Watcher监听，而监听的节点，正是前面所保存在prior\_path成员的前一个节点的路径。这里，仅仅去监听自己前一个节点的变动，而不是其他节点的变动，提升效率。完成监听之后，调用latch.await（），线程进入等待状态，一直到线程被监听回调代码中的latch.countDown() 所唤醒，或者等待超时。

说 明

以上代码用到的CountDownLatch的核心原理和实战知识，《Netty Zookeeper Redis 高并发实战》姊妹篇 《Java高并发核心编程（卷2）》。

上面的代码中，监听前一个节点的删除，可以使用两种监听方式：

（1）Watcher 订阅；

（2）TreeCache 订阅。

两种方式的效果，都差不多。但是这里的删除事件，只需要监听一次即可，不需要反复监听，所以使用的是Watcher  
一次性订阅。而TreeCache 订阅的代码在源码工程中已经被注释，仅仅供大家参考。

一旦前一个节点prior\_path节点被删除，那么就将线程从等待状态唤醒，重新一轮的锁的争夺，直到获取锁，并且完成业务处理。

至此，分布式Lock加锁的算法，还差一点就介绍完成。这一点，就是实现锁的可重入。

### **5．可重入的实现代码**

什么是可重入呢？只需要保障同一个线程进入加锁的代码，可以重复加锁成功即可。  
修改前面的lock方法，在前面加上可重入的判断逻辑。代码如下：

@Override

public boolean lock() {

//可重入的判断

synchronized (this) {

if (lockCount.get() == 0) {

thread = Thread.currentThread();

lockCount.incrementAndGet();

} else {

if (!thread.equals(Thread.currentThread())) {

return false;

}

lockCount.incrementAndGet();

return true;

}

}

//....

}

为了变成可重入，在代码中增加了一个加锁的计数器lockCount  
，计算重复加锁的次数。如果是同一个线程加锁，只需要增加次数，直接返回，表示加锁成功。

至此，lock（）方法已经介绍完成，接下来，就是去释放锁

## **实战：释放锁的实现**

Lock接口中的unLock（）方法，表示释放锁，释放锁主要有两个工作：

（1）减少重入锁的计数，如果最终的值不是0，直接返回，表示成功的释放了一次；

（2）如果计数器为0，移除Watchers监听器，并且删除创建的Znode临时节点。

unLock（）方法的代码如下：

/\*\*

\* 释放锁

\*

\* @return 是否成功释放锁

\*/

@Override

public boolean unlock() {//只有加锁的线程，能够解锁

if (!thread.equals(Thread.currentThread())) {

return false;

}//减少可重入的计数

int newLockCount = lockCount.decrementAndGet();//计数不能小于0

if (newLockCount < 0) {

throw new IllegalMonitorStateException("Lock count has gone negative for lock: " + locked\_path);

}//如果计数不为0，直接返回

if (newLockCount != 0) {

return true;

}

//删除临时节点

try {

if (ZKclient.instance.isNodeExist(locked\_path)) {

client.delete().forPath(locked\_path);

}

} catch (Exception e) {

e.printStackTrace();

return false;

}

return true;

}

这里，为了尽量保证线程安全，可重入计数器的类型，使用的不是int类型，而是Java并发包中的原子类型——AtomicInteger。

## **实战：分布式锁的使用**

写一个用例，测试一下ZLock的使用，代码如下：

@Test

public void testLock() throws InterruptedException {

for (int i = 0; i < 10; i++) {

FutureTaskScheduler.add(() -> {

//创建锁

ZkLock lock = new ZkLock();

lock.lock();//每条线程，执行10次累加

for (int j = 0; j < 10; j++) {//公共的资源变量累加

count++;

}

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

log.info("count = " + count);

//释放锁

lock.unlock();

});

}

Thread.sleep(Integer.MAX\_VALUE);

}

以上代码是10个并发任务，每个任务累加10次，执行以上用例，会发现结果会是预期的和100，如果不使用锁，结果可能就不是100，因为上面的count是一个普通的变量，不是线程安全的。

说 明

有关线程安全的核心原理和实战知识，请参阅本书的下一卷《Java高并发核心编程（卷2）》。

原理上一个Zlock实例代表一把锁，并需要占用一个Znode永久节点，如果需要很多分布式锁，则也需要很多的不同的Znode节点。以上代码，如果要扩展为多个分布式锁的版本，还需要进行简单改造，这种改造留给各位自己去练习和实现吧。

## **实战：curator的InterProcessMutex 可重入锁**

分布式锁Zlock自主实现主要的价值：学习一下分布式锁的原理和基础开发，仅此而已。实际的开发中，如果需要使用到分布式锁，并建议去自己造轮子，建议直接使用Curator客户端中的各种官方实现的分布式锁，比如其中的InterProcessMutex  
可重入锁。

这里提供一个简单的InterProcessMutex 可重入锁的使用实例，代码如下：

@Test

public void testzkMutex() throws InterruptedException {

CuratorFramework client = ZKclient.instance.getClient();

final InterProcessMutex zkMutex =

new InterProcessMutex(client, "/mutex");

;

for (int i = 0; i < 10; i++) {

FutureTaskScheduler.add(() -> {

try {

//获取互斥锁

zkMutex.acquire();

for (int j = 0; j < 10; j++) {//公共的资源变量累加

count++;

}

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

log.info("count = " + count);

//释放互斥锁

zkMutex.release();

} catch (Exception e) {

e.printStackTrace();

}

});

}

Thread.sleep(Integer.MAX\_VALUE);

}

## **ZooKeeper分布式锁的优点和缺点**

总结一下ZooKeeper分布式锁：

（1）优点：ZooKeeper分布式锁（如InterProcessMutex），能有效的解决分布式问题，不可重入问题，使用起来也较为简单。

（2）缺点：ZooKeeper实现的分布式锁，性能并不太高。为啥呢？  
因为每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。大家知道，ZK中创建和删除节点只能通过Leader服务器来执行，然后Leader服务器还需要将数据同不到所有的Follower机器上，这样频繁的网络通信，性能的短板是非常突出的。

总之，在高性能，高并发的场景下，不建议使用ZooKeeper的分布式锁。而由于ZooKeeper的高可用特性，所以在并发量不是太高的场景，推荐使用ZooKeeper的分布式锁。

在目前分布式锁实现方案中，比较成熟、主流的方案有两种：

（1）基于Redis的分布式锁

（2）基于ZooKeeper的分布式锁

两种锁，分别适用的场景为：

（1）基于ZooKeeper的分布式锁，适用于高可靠（高可用）而并发量不是太大的场景；

（2）基于Redis的分布式锁，适用于并发量很大、性能要求很高的、而可靠性问题可以通过其他方案去弥补的场景。

总之，这里没有谁好谁坏的问题，而是谁更合适的问题。

最后对本章的内容做个总结：在分布式系统中，ZooKeeper是一个重要的协调工具。本章介绍了分布式命名服务、分布式锁的原理以及基于ZooKeeper的参考实现。本章的那些实战案例，建议大家自己去动手掌握，无论是应用实际开始、还是大公司面试，都是非常有用的。另外，主流的分布式协调中间件，也不仅仅只有Zookeeper，还有非常著名的Etcd中间件。但是从学习的层面来说，二者之间的功能设计、核心原理都是差不多的，掌握了Zookeeper，Etcd的上手使用也是很容易的。