> 点赞再看，养成习惯，微信搜索【**三太子敖丙**】关注这个互联网苟且偷生的工具人。

> 本文 **GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

前言

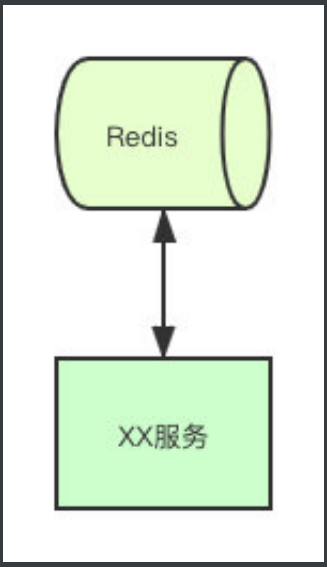
锁我想不需要我过多的去说，大家都知道是怎么一回事了吧？

在多线程环境下，由于上下文的切换，数据可能出现不一致的情况或者数据被污染，我们需要保证数据安全，所以想到了加锁。

所谓的加锁机制呢，就是当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问，直到该线程读取完，其他线程才可使用。

还记得我之前说过Redis在分布式的情况下，需要对存在并发竞争的数据进行加锁，老公们十分费解，Redis是单线程的嘛？为啥还要加锁呢？

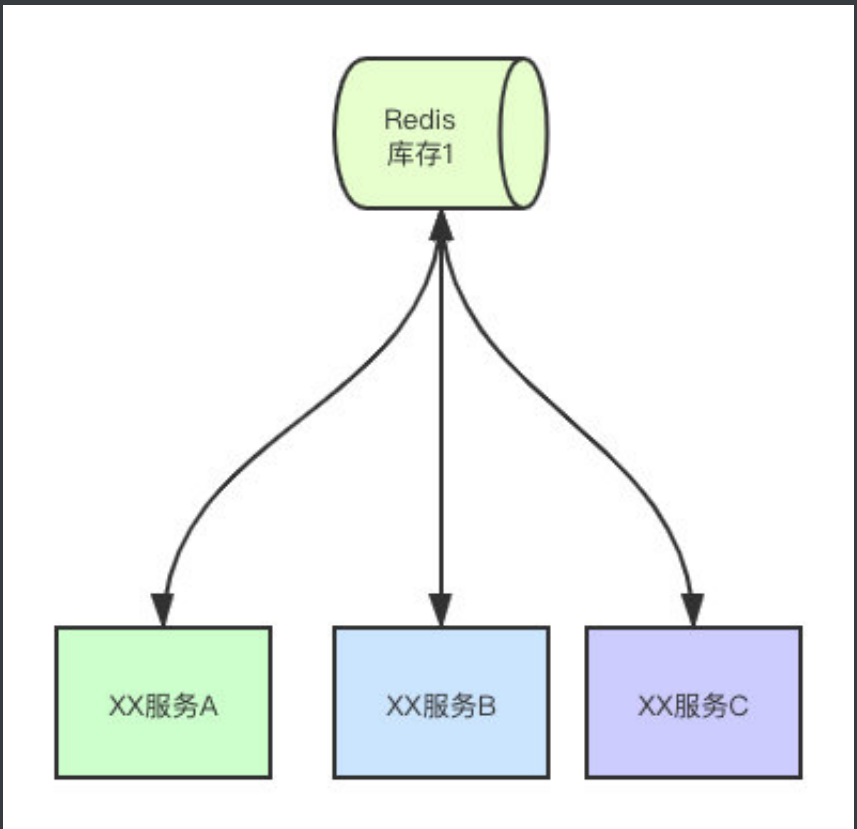
看来老公们还是年轻啊，你说的不需要加锁的情况是这样的：



单个服务去访问Redis的时候，确实因为Redis本身单线程的原因是不用考虑线程安全的，但是，现在有哪个公司还是单机的呀？肯定都是分布式集群了嘛。

老公们你看下这样的场景是不是就有问题了：

你们经常不是说秒杀嘛，拿到库存判断，那老婆告诉你分布式情况就是会出问题的。



我们为了减少DB的压力，把库存预热到了KV，现在KV的库存是1。

1. 服务A去Redis查询到库存发现是1，那说明我能抢到这个商品对不对，那我就准备减一了，但是还没减。
2. 同时服务B也去拿发现也是1，那我也抢到了呀，那我也减。
3. C同理。
4. 等所有的服务都判断完了，你发现诶，怎么变成-2了，超卖了呀，这下完了。

老公们是不是发现问题了，这就需要分布式锁的介入了，我会分三个章节去分别介绍分布式锁的三种实现方式（Zookeeper，Redis，MySQL），说出他们的优缺点，以及一般大厂的实践场景。

正文

> 一个骚里骚气的面试官啥也没拿的就走了进来，你一看，这不是你老婆嘛，你正准备叫他的时候，发现他一脸严肃，死鬼还装严肃，肯定会给我放水的吧。

> 咳咳，我们啥也不说了，开始今天的面试吧。

> 正常线程进程同步的机制有哪些?

- 互斥：互斥的机制，保证同一时间只有一个线程可以操作共享资源 synchronized，Lock等。
- 临界值：让多线程串行话去访问资源
- 事件通知：通过事件的通知去保证大家都有序访问共享资源
- 信号量：多个任务同时访问，同时限制数量，比如发令枪CDL，Semaphore等

> 那分布式锁你了解过有哪些么?

分布式锁实现主要以Zookeeper（以下简称zk）、Redis、MySQL这三种为主。

> 那先跟我聊一下zk吧，你能说一下他常见的使用场景么?

他主要的应用场景有以下几个：

- 服务注册与订阅（共用节点）
- 分布式通知（监听znode）
- 服务命名（znode特性）
- 数据订阅、发布（watcher）
- 分布式锁（临时节点）

> zk是啥?

他是个数据库，文件存储系统，并且有监听通知机制（观察者模式）

> 存文件系统，他存了什么？

节点

zk的节点类型有4大类

- 持久化节点（zk断开节点还在）
- 持久化顺序编号目录节点
- 临时目录节点（客户端断开后节点就删除了）
- 临时目录编号目录节点

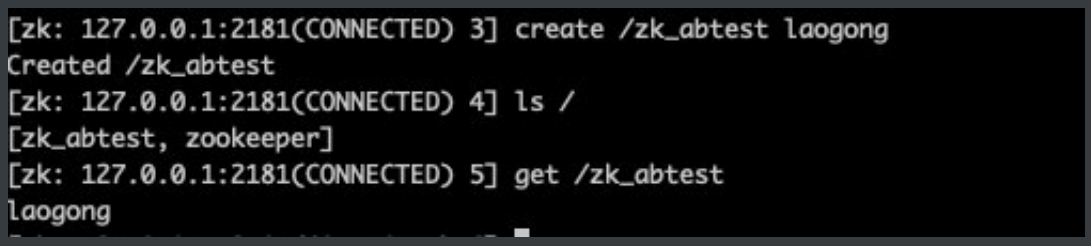
节点名称都是唯一的。

> 节点怎么创建？

我特么，这样问的么？可是我面试只看了分布式锁，我得好好想想！！！

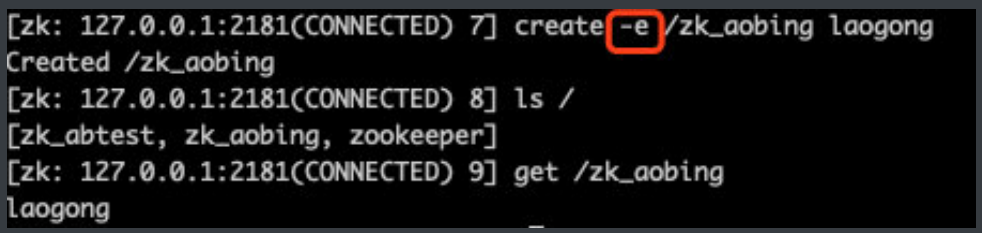
还好我之前在自己的服务器搭建了一个zk的集群，我刚好跟大家回忆一波。

```
create /test laogong // 创建永久节点
```

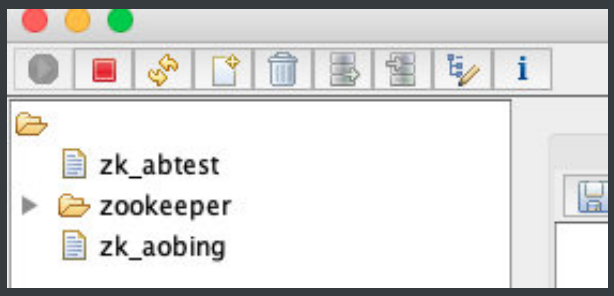


> 那临时节点呢？

```
create -e /test laogong // 创建临时节点
```



临时节点就创建成功了，如果我断开这次链接，这个节点自然就消失了，这是我的一个zk管理工具，目录可能清晰点。



> 如果创建顺序节点呢?

```
create -s /test // 创建顺序节点
```

```
[zk: 127.0.0.1:2181(CONNECTED) 13] create -s /k_test
Created /k_test0000000002
[zk: 127.0.0.1:2181(CONNECTED) 14] ls /
[k_test0000000002, zk_abtest, zk_aobing, zookeeper]
[zk: 127.0.0.1:2181(CONNECTED) 15]
```

> 临时顺序节点呢?

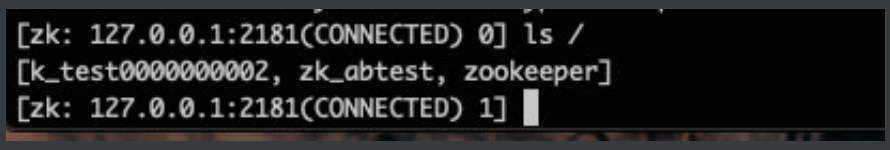
我想聪明的老公都会抢答了

```
create -e -s /test  // 创建临时顺序节点
```

```
[zk: 127.0.0.1:2181(CONNECTED) 15] create -e -s /zk_test
Created /zk_test0000000003
[zk: 127.0.0.1:2181(CONNECTED) 16] ls /
[k_test0000000002, zk_abtest, zk_aobing, zk_test0000000003, zookeeper]
[zk: 127.0.0.1:2181(CONNECTED) 17]
```

我退出后，重新连接，发现刚才创建的所有临时节点都没了。

```
[zk: 127.0.0.1:2181(CONNECTED) 19] quit

WATCHER::

WatchedEvent state:Closed type:None path:null
2020-04-06 13:46:38,406 [myid:] - INFO  [main-EventThread:ClientCnxn$EventThread@566] - EventTh
2020-04-06 13:46:38,406 [myid:] - INFO  [main:ZooKeeper@1618] - Session: 0x100006c839b0001 clos
2020-04-06 13:46:38,411 [myid:] - ERROR [main:ServiceUtils@42] - Exiting JVM with code 0
MacBook-Pro-3:bin aobing$ ./zkCli.sh -server 127.0.0.1:2181
```



开篇演示这么多呢，我就是想给大家看到的zk大概的一个操作流程和数据结构，中间涉及的搭建以及其他的技能我就不说了，我们重点聊一下他在分布式锁中的实现。

zk就是基于节点去实现各种分布式锁的。

> 就拿开头的场景来说，zk应该怎么去保证分布式情况下的线程安全呢？并发竞争他是怎么控制的呢？

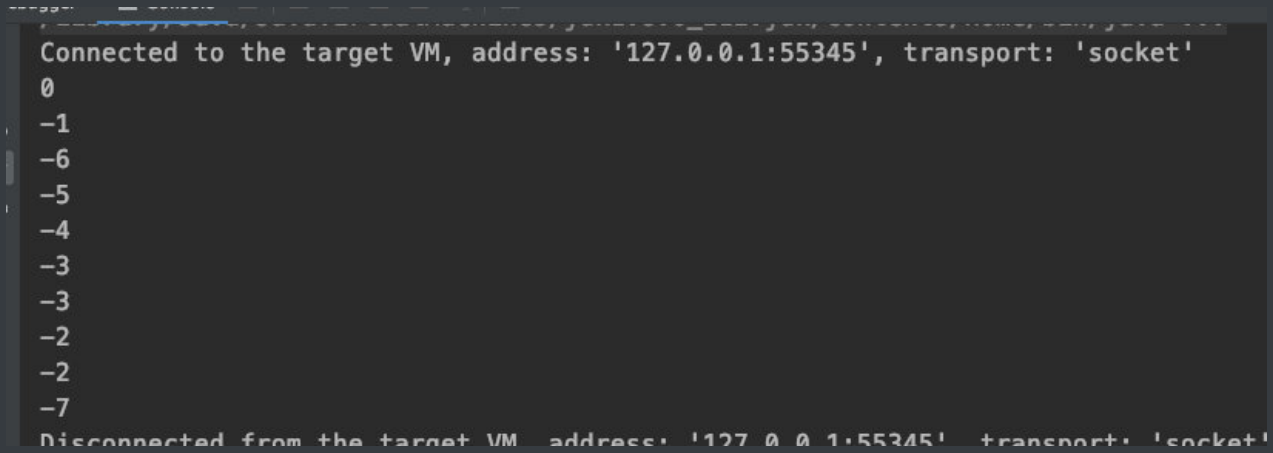
为了模拟并发竞争这样一个情况，我写了点伪代码，大家可以先看看

```java
/**
 * @Description: zkTest
 * @Author: 敖丙
 * @date: 2020-04-06
 **/
public class zkTest implements Runnable {
    static int inventory = 1;
    private static final int NUM = 10;
    private static CountDownLatch cdl = new CountDownLatch(NUM);

    public static void main(String[] args) {
        for (int i = 1; i <= NUM; i++) {
            new Thread(new zkTest()).start();
            cdl.countDown();
        }
    }

    @Override
    public void run() {
        try {
            cdl.await();
            if (inventory > 0) {
                Thread.sleep( millis: 5);
                inventory--;
            }
            System.out.println(inventory);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

我定义了一个库存inventory值为1，还用到了一个CountDownLatch发令枪，等10个线程都就绪了一起去扣减库存。

是不是就像10台机器一起去拿到库存，然后扣减库存了？

所有机器一起去拿，发现都是1，那大家都认为是自己抢到了，都做了减一的操作，但是等所有人都执行完，再去set值的时候，发现其实已经超卖了，我打印出来给大家看看。

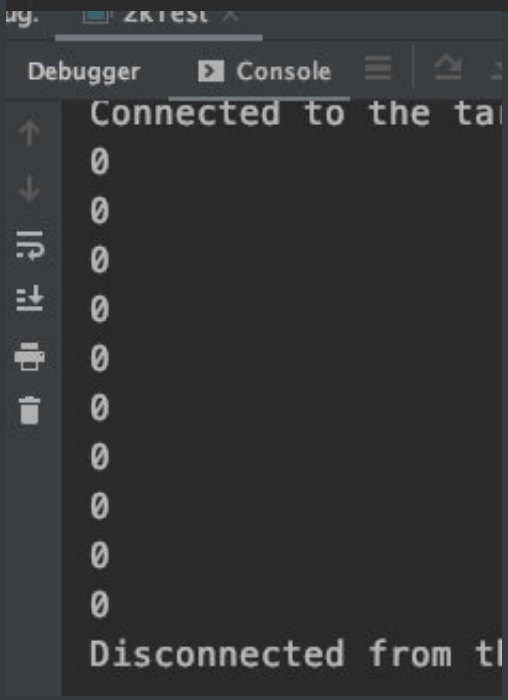


是吧，这还不是超卖一个两个的问题，超卖7个都有，代码里面明明判断了库存大于0才去减的，怎么回事开头我说明了。

> 那怎么解决这个问题?

sync，lock也只能保证你当前机器线程安全，这样分布式访问还是有问题。

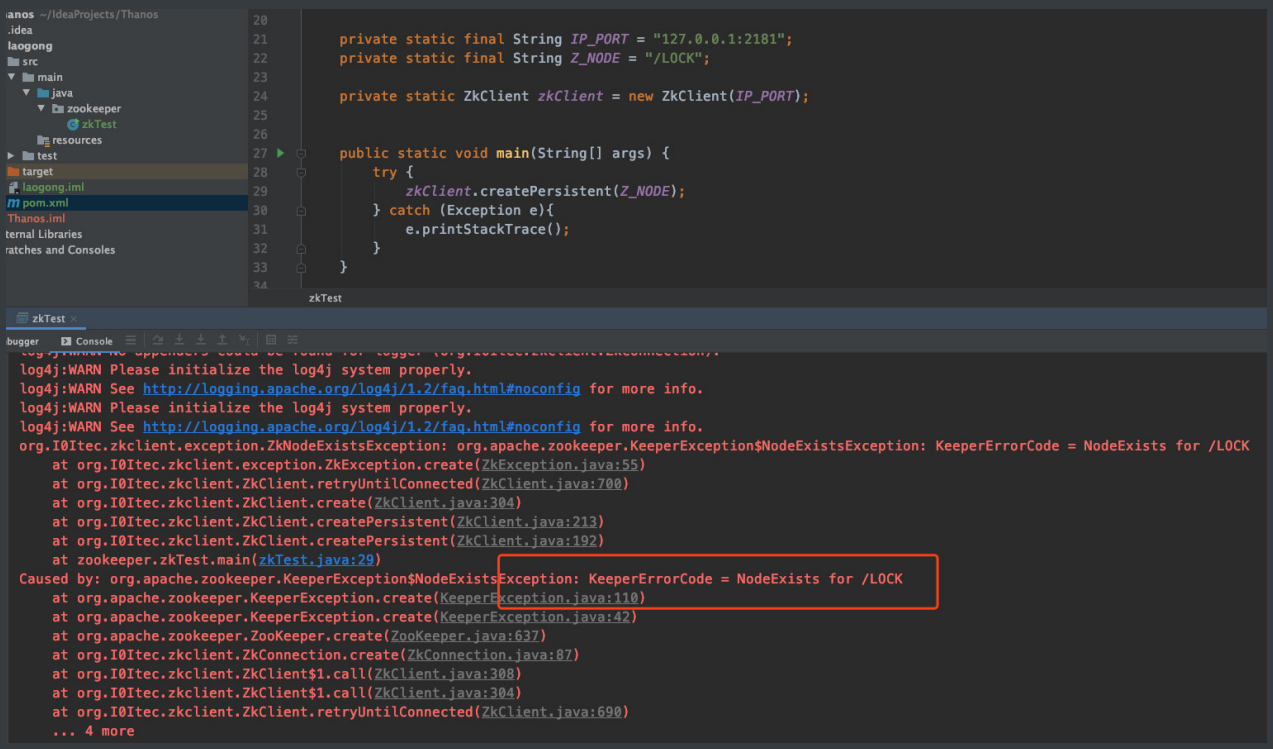
```java
@Override
public void run() {
    lock.lock();
    try {
        cdl.await();
        if (inventory > 0) {
            Thread.sleep( millis: 5);
            inventory--;
        }
        System.out.println(inventory);

    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
```



上面跟大家提到的zk的节点就可以解决这个问题。

zk节点有个唯一的特性，就是我们创建过这个节点了，你再创建zk是会报错的，那我们就利用一下他的唯一性去实现一下。

```
[zk: 127.0.0.1:2181(CONNECTED) 3] create /zk_abtest
Node already exists: /zk_abtest
[zk: 127.0.0.1:2181(CONNECTED) 4]
```



> 怎么实现呢?

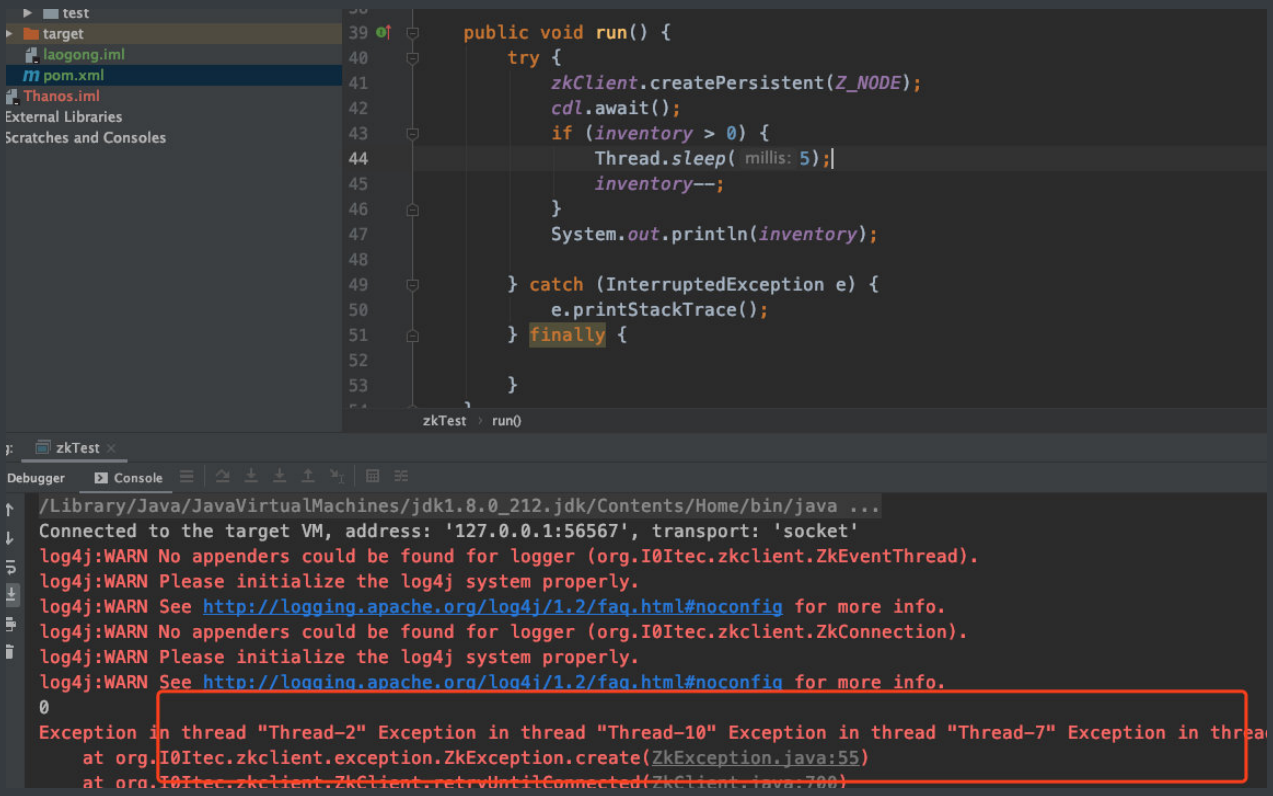
上面不是10个线程嘛?

我们全部去创建，创建成功的第一个返回true他就可以继续下面的扣减库存操作，后续的节点访问就会全部报错，扣减失败，我们把它们丢一个队列去排队。

> 那怎么释放锁呢?

删除节点咯，删了再通知其他的人过来加锁，依次类推。

我们实现一下，zk加锁的场景。



是不是，只有第一个线程能扣减成功，其他的都失败了。

> 但是你发现问题没有，你加了锁了，你得释放啊，你不释放后面的报错了就不重试了。

那简单，删除锁就释放掉了，Lock在finally里面unLock，现在我们在finally删除节点。

> 加锁我们知道创建节点就够了，但是你得实现一个阻塞的效果呀，那咋搞?

死循环，递归不断去尝试，直到成功，一个伪装的阻塞效果。

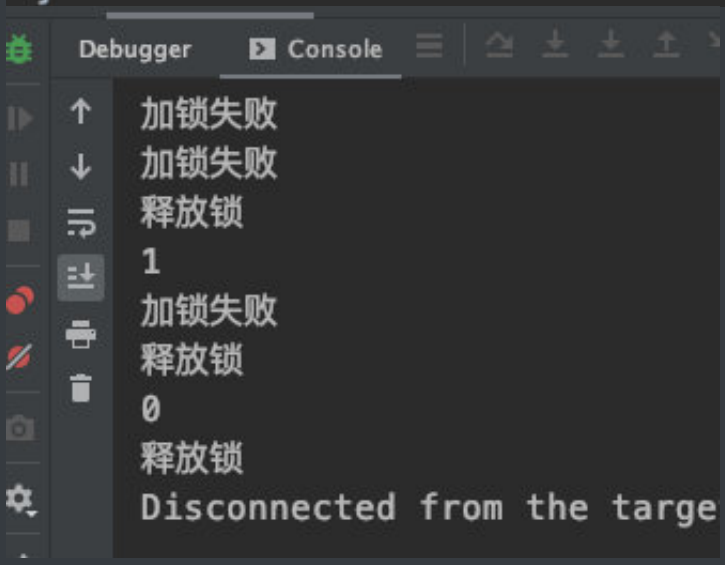
> 怎么知道前面的老哥删除节点了嗯?

监听节点的删除事件

```java
public void lock() {
    // 尝试加锁
    if(tryLock()){
        return;
    }
    // 进入等待 监听
    waitForLock();
    // 再次尝试
    lock();
}
```

```java
public void waitForLock(){
    System.out.println("加锁失败");
    IZkDataListener listener = new IZkDataListener() {
        public void handleDataChange(String s, Object o) throws Exception {

        }

        public void handleDataDeleted(String s) throws Exception {
            System.out.println("唤醒");
            cdl.countDown();
        }
    };
    // 监听
    zkClient.subscribeDataChanges(Z_NODE, listener);
    if (zkClient.exists(Z_NODE)) {
        try {
            cdl.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // 释放监听
    zkClient.unsubscribeDataChanges(Z_NODE, listener);
}
```

```
public void run() {
    try {
        new ZkTest().lock();
        if (inventory > 0) {
            inventory--;
        }
        System.out.println(inventory);
        return;
    } finally {
        new ZkTest().unlock();
        System.out.println("释放锁");
    }
}
```



> 但是你发现你这样做的问题没?

是的，会出现死锁。

第一个仔加锁成功了，在执行代码的时候，机器宕机了，那节点是不是就不能删除了?

你要故作沉思，自问自答，时而看看远方，时而看看面试官，假装自己什么都不知道。

哦我想起来了，创建临时节点就好了，客户端连接一断开，别的就可以监听到节点的变化了。

> 嗯还不错，那你发现还有别的问题没?

好像这种监听机制也不好。

> 怎么个不好呢?

你们可以看到，监听，是所有服务都去监听一个节点的，节点的释放也会通知所有的服务器，如果是900个服务器呢?

这对服务器是很大的一个挑战，一个释放的消息，就好像一个牧羊犬进入了羊群，大家都四散而开，随时可能干掉机器，会占用服务资源，网络带宽等等。

这就是羊群效应。

> 那怎么解决这个问题?

继续故作沉思，啊啊啊，好难，我的脑袋。。。。

> 你TM别装了好不好?

好的，临时顺序节点，可以顺利解决这个问题。

怎么实现老公你先别往下看，先自己想想。

之前说了全部监听一个节点问题很大，那我们就监听我们的前一个节点，因为是顺序的，很容易找到自己的前后。

```java
public void lock() {
    if (tryLock()) {
        System.out.println("获得锁");
    } else {
        // 尝试加锁
        // 进入等待 监听
        waitForLock();
        // 再次尝试
        lock();
    }

}

public synchronized boolean tryLock() {
    // 第一次就进来创建自己的临时节点
    if (StringUtils.isBlank(path)) {
        path = zkClient.createEphemeralSequential( path: Z_NODE + "/",  data: "lock");
    }

    // 对节点排序
    List<String> children = zkClient.getChildren(Z_NODE);
    Collections.sort(children);

    // 当前的是最小节点就返回加锁成功
    if (path.equals(Z_NODE + "/" + children.get(0))) {
        System.out.println(" i am true");
        return true;
    } else {
        // 不是最小节点 就找到自己的前一个 依次类推 释放也是一样
        int i = Collections.binarySearch(children, path.substring(Z_NODE.length() + 1));
        beforePath = Z_NODE + "/" + children.get(i - 1);
    }
    return false;
}
```

和之前监听一个永久节点的区别就在于，这里每个节点只监听了自己的前一个节点，释放当然也是一个个释放下去，就不会出现羊群效应了。

```java
public void waitForLock() {

    IZkDataListener listener = new IZkDataListener() {
        public void handleDataChange(String s, Object o) throws Exception {
        }

        public void handleDataDeleted(String s) throws Exception {
            System.out.println(Thread.currentThread().getName() + ":监听到节点删除事件! ----------------------------");
            cdl.countDown();
        }
    };
    // 监听
    this.zkClient.subscribeDataChanges(beforePath, listener);
    if (zkClient.exists(beforePath)) {
        try {
            System.out.println("加锁失败 等待");
            cdl.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // 释放监听
    zkClient.unsubscribeDataChanges(beforePath, listener);
```

以上所有代码我都会开源到我的https://github.com/AobingJava/Thanos其实上面的还有瑕疵，大家可以去拉下来改一下提交pr，我会看合适的会通过的。

> 你说了这么多，挺不错的，你能说说ZK在分布式锁中实践的一些缺点么？

Zk性能上可能并没有缓存服务那么高。

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。

ZK中创建和删除节点只能通过Leader服务器来执行，然后将数据同步到所有的Follower机器上。（这里涉及zk集群的知识，我就不展开了，以后zk章节跟老公们细聊）

> 还有么？

使用Zookeeper也有可能带来并发问题，只是并不常见而已。

由于网络抖动，客户端可ZK集群的session连接断了，那么zk以为客户端挂了，就会删除临时节点，这时候其他客户端就可以获取到分布式锁了。

就可能产生并发问题了，这个问题不常见是因为zk有重试机制，一旦zk集群检测不到客户端的心跳，就会重试，Curator客户端支持多种重试策略。

多次重试之后还不行的话才会删除临时节点。

Tip：所以，选择一个合适的重试策略也比较重要，要在锁的粒度和并发之间找一个平衡。

> 有更好的实现么？

基于Redis的分布式锁

> 能跟我聊聊么？

我看看了手上的表，老公，今天天色不早了，你全问完了，我怎么多水几篇文章呢？

> 行确实很晚了，那你回家去把家务干了吧？

我？？？？

总结

zk通过临时节点，解决掉了**死锁**的问题，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉，其他客户端自动获取锁。

zk通过节点排队监听的机制，也实现了**阻塞**的原理，其实就是个递归在那无限等待最小节点释放的过程。

我上面没实现锁的**可重入**，但是也很好实现，可以带上线程信息就可以了，或者机器信息这样的唯一标识，获取的时候判断一下。

zk的集群也是**高可用**的，只要半数以上的或者，就可以对外提供服务了。

这周会写完Redis和数据库的分布式锁的，老公们等好。

我是敖丙，一个在互联网苟且偷生的工具人。

最好的关系是互相成就，老公们的**「三连」**就是丙丙创作的最大动力，我们下期见！

注：如果本篇博客有任何错误和建议，欢迎老公们留言，**老公你快说句话啊**！

> 文章持续更新，可以微信搜索「 **三太子敖丙** 」第一时间阅读，回复【**资料**】【**面试**】【**简历**】有我准备的一线大厂面试资料和简历模板，本文 **GitHub** https://github.com/JavaFamily 已经收录，有大厂面试完整考点，欢迎Star。

你知道的越多，你不知道的越多