41 | 如何设计更优的分布式锁？

2019-08-24 刘超

你好，我是刘超。

从这一讲开始，我们就正式进入最后一个模块的学习了，综合性实战的内容来自我亲身经历过的一些案例，其中用到的知识点会相对综合，现在是时候跟我一起调动下前面所学了！

去年双十一，我们的游戏商城也搞了一波活动，那时候我就发现在数据库操作日志中，出现最多的一个异常就是Interrupted Exception了，几乎所有的异常都是来自一个校验订单幂等性的SQL。

因为校验订单幂等性是提交订单业务中第一个操作数据库的，所以幂等性校验也就承受了比较大的请求量，再加上我们还是基于一个数据库表来实现幂等性校验的，所以出现了一些请求事务超时，事务被中断的情况。其实基于数据库实现的幂等性校验就是一种分布式锁的实现。

那什么是分布式锁呢，它又是用来解决哪些问题的呢？

在JVM中，在多线程并发的情况下，我们可以使用同步锁或Lock锁，保证在同一时间内，只能有一个线程修改共享变量或执行代码块。但现在我们的服务基本都是基于分布式集群来实现部署的，对于一些共享资源，例如我们之前讨论过的库存，在分布式环境下使用Java锁的方式就失去作用了。

这时，我们就需要实现分布式锁来保证共享资源的原子性。除此之外，分布式锁也经常用来避免分布式中的不同节点执行重复性的工作，例如一个定时发短信的任务，在分布式集群中，我们只

需要保证一个服务节点发送短信即可，一定要避免多个节点重复发送短信给同一个用户。

因为数据库实现一个分布式锁比较简单易懂，直接基于数据库实现就行了，不需要再引入第三方中间件，所以这是很多分布式业务实现分布式锁的首选。但是数据库实现的分布式锁在一定程度上，存在性能瓶颈。

接下来我们一起了解下如何使用数据库实现分布式锁，其性能瓶颈到底在哪，有没有其它实现方式可以优化分布式锁。

数据库实现分布式锁

首先，我们应该创建一个锁表，通过创建和查询数据来保证一个数据的原子性：

```
CREATE TABLE `order` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `order_no` int(11) DEFAULT NULL,
  `pay_money` decimal(10, 2) DEFAULT NULL,
  `status` int(4) DEFAULT NULL,
  `create_date` datetime(0) DEFAULT NULL,
  `delete_flag` int(4) DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE,
  INDEX`idx_status`(`status`) USING BTREE,
  INDEX`idx_order`(`order_no`) USING BTREE
) ENGINE = InnoDB
```

其次，如果是校验订单的幂等性，就要先查询该记录是否存在数据库中，查询的时候要防止幻读，如果不存在，就插入到数据库，否则，放弃操作。

```
select id from `order` where `order_no`= 'xxxx' for update
```

最后注意下，除了查询时防止幻读，我们还需要保证查询和插入是在同一个事务中，因此我们需要申明事务，具体的实现代码如下：

```
@Transactional
public int addOrderRecord(Order order) {
 if(orderDao.selectOrderRecord(order)==null){
            int result = orderDao.addOrderRecord(order);
           if(result>0){
                 return 1;
           }
      }
 return 0;
}
```

到这，我们订单幂等性校验的分布式锁就实现了。我想你应该能发现为什么这种方式会存在性能瓶颈了。我们在第34讲中讲过，在RR事务级别，select的for update操作是基于间隙锁gap lock实现的，这是一种悲观锁的实现方式，所以存在阻塞问题。

因此在高并发情况下，当有大量的请求进来时，大部分的请求都会进行排队等待。为了保证数据库的稳定性，事务的超时时间往往又设置得很小，所以就会出现大量事务被中断的情况。

除了阻塞等待之外，因为订单没有删除操作，所以这张锁表的数据将会逐渐累积，我们需要设置另外一个线程，隔一段时间就去删除该表中的过期订单，这就增加了业务的复杂度。

除了这种幂等性校验的分布式锁，有一些单纯基于数据库实现的分布式锁代码块或对象，是需要在锁释放时，删除或修改数据的。如果在获取锁之后，锁一直没有获得释放，即数据没有被删除或修改，这将会引发死锁问题。

Zookeeper实现分布式锁

除了数据库实现分布式锁的方式以外，我们还可以基于Zookeeper实现。Zookeeper是一种提供“分布式服务协调“的中心化服务，正是Zookeeper的以下两个特性，分布式应用程序才可以基于它实现分布式锁功能。

顺序临时节点：Zookeeper提供一个多层级的节点命名空间（节点称为Znode），每个节点都用一个以斜杠（/）分隔的路径来表示，而且每个节点都有父节点（根节点除外），非常类似于文件系统。

节点类型可以分为持久节点（PERSISTENT）、临时节点（EPHEMERAL），每个节点还能被标记为有序性（SEQUENTIAL），一旦节点被标记为有序性，那么整个节点就具有顺序自增的特点。一般我们可以组合这几类节点来创建我们所需要的节点，例如，创建一个持久节点作为父节点，在父节点下面创建临时节点，并标记该临时节点为有序性。

Watch机制：Zookeeper还提供了另外一个重要的特性，Watcher（事件监听器）。ZooKeeper允许用户在指定节点上注册一些Watcher，并且在一些特定事件触发的时候，ZooKeeper服务端会将事件通知给用户。

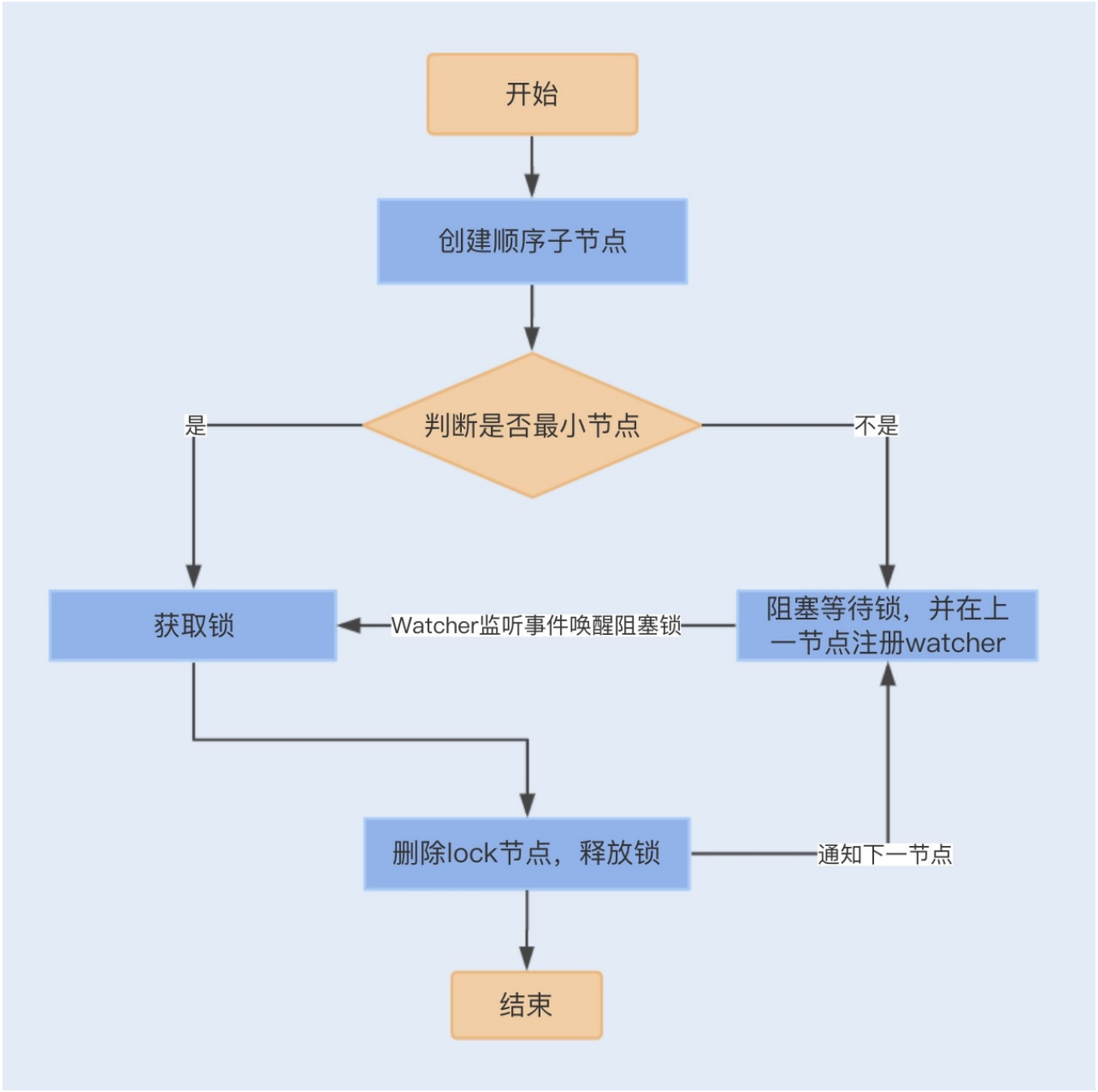
我们熟悉了Zookeeper的这两个特性之后，就可以看看Zookeeper是如何实现分布式锁的了。

首先，我们需要建立一个父节点，节点类型为持久节点（PERSISTENT），每当需要访问共享资源时，就会在父节点下建立相应的顺序子节点，节点类型为临时节点（EPHEMERAL），且标记为有序性（SEQUENTIAL），并且以临时节点名称+父节点名称+顺序号组成特定的名字。

在建立子节点后，对父节点下面的所有以临时节点名称name开头的子节点进行排序，判断刚刚建立的子节点顺序号是否是最小的节点，如果是最小节点，则获得锁。

如果不是最小节点，则阻塞等待锁，并且获得该节点的上一顺序节点，为其注册监听事件，等待节点对应的操作获得锁。

当调用完共享资源后，删除该节点，关闭zk，进而可以触发监听事件，释放该锁。



以上实现的分布式锁是严格按照顺序访问的并发锁。一般我们还可以直接引用Curator框架来实现Zookeeper分布式锁，代码如下：

```
InterProcessMutex lock = new InterProcessMutex(client, lockPath);
if ( lock.acquire(maxWait, waitUnit) )
{
    try
    {
        // do some work inside of the critical section here
    }
    finally
    {
        lock.release();
    }
}
```

Zookeeper实现的分布式锁，例如相对数据库实现，有很多优点。Zookeeper是集群实现，可以避免单点问题，且能保证每次操作都可以有效地释放锁，这是因为一旦应用服务挂掉了，临时节点会因为session连接断开而自动删除掉。

由于频繁地创建和删除结点，加上大量的Watch事件，对Zookeeper集群来说，压力非常大。且从性能上来说，其与接下来我要讲的Redis实现的分布式锁相比，还是存在一定的差距。

Redis实现分布式锁

相对于前两种实现方式，基于Redis实现的分布式锁是最为复杂的，但性能是最佳的。

大部分开发人员利用Redis实现分布式锁的方式，都是使用SETNX+EXPIRE组合来实现，在Redis 2.6.12版本之前，具体实现代码如下：

```
public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {


    Long result = jedis.setnx(lockKey, requestId);//设置锁
    if (result == 1) {//获取锁成功
        // 若在这里程序突然崩溃，则无法设置过期时间，将发生死锁
        jedis.expire(lockKey, expireTime);//通过过期时间删除锁
        return true;
    }
    return false;
}
```

这种方式实现的分布式锁，是通过setnx()方法设置锁，如果lockKey存在，则返回失败，否则返回成功。设置成功之后，为了能在完成同步代码之后成功释放锁，方法中还需要使用expire()方法给lockKey值设置一个过期时间，确认key值删除，避免出现锁无法释放，导致下一个线程无法获取到锁，即死锁问题。

如果程序在设置过期时间之前、设置锁之后出现崩溃，此时如果lockKey没有设置过期时间，将会出现死锁问题。

在 Redis 2.6.12版本后SETNX增加了过期时间参数：

```
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";

/**
 * 尝试获取分布式锁
 * @param jedis Redis客户端
 * @param lockKey 锁
 * @param requestId 请求标识
 * @param expireTime 超期时间
 * @return 是否获取成功
 */
public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {

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

    if (LOCK_SUCCESS.equals(result)) {
        return true;
    }
    return false;

}
```

我们也可以通过Lua脚本来实现锁的设置和过期时间的原子性，再通过jedis.eval()方法运行该脚本：

```
// 加锁脚本
private static final String SCRIPT_LOCK = "if redis.call('setnx', KEYS[1], ARGV[1]) == 1 then redis.call('pexpire
// 解锁脚本
private static final String SCRIPT_UNLOCK = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del
```

虽然SETNX方法保证了设置锁和过期时间的原子性，但如果我们设置的过期时间比较短，而执行业务时间比较长，就会存在锁代码块失效的问题。我们需要将过期时间设置得足够长，来保证以上问题不会出现。

这个方案是目前最优的分布式锁方案，但如果是在Redis集群环境下，依然存在问题。由于Redis集群数据同步到各个节点时是异步的，如果在Master节点获取到锁后，在没有同步到其它节点时，Master节点崩溃了，此时新的Master节点依然可以获取锁，所以多个应用服务可以同时获取到锁。

Redlock算法

Redisson由Redis官方推出，它是一个在Redis的基础上实现的Java驻内存数据网格（In-Memory Data Grid）。它不仅提供了一系列的分布式的Java常用对象，还提供了许多分布式服务。Redisson是基于netty通信框架实现的，所以支持非阻塞通信，性能相对于我们熟悉的Jedis会好一些。

Redisson中实现了Redis分布式锁，且支持单点模式和集群模式。在集群模式下，Redisson使用了Redlock算法，避免在Master节点崩溃切换到另外一个Master时，多个应用同时获得锁。我们可以通过一个应用服务获取分布式锁的流程，了解下Redlock算法的实现：

在不同的节点上使用单个实例获取锁的方式去获得锁，且每次获取锁都有超时时间，如果请求超时，则认为该节点不可用。当应用服务成功获取锁的Redis节点超过半数（N/2+1，N为节点数)时，并且获取锁消耗的实际时间不超过锁的过期时间，则获取锁成功。

一旦获取锁成功，就会重新计算释放锁的时间，该时间是由原来释放锁的时间减去获取锁所消耗的时间；而如果获取锁失败，客户端依然会释放获取锁成功的节点。

具体的代码实现如下：

1.首先引入jar包：

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.8.2</version>
</dependency>
```

2.实现Redisson的配置文件：

```
@Bean
public RedissonClient redissonClient() {
    Config config = new Config();
    config.useClusterServers()
            .setScanInterval(2000) // 集群状态扫描间隔时间，单位是毫秒
            .addNodeAddress("redis://127.0.0.1:7000").setPassword("1")
            .addNodeAddress("redis://127.0.0.1:7001").setPassword("1")
            .addNodeAddress("redis://127.0.0.1:7002")
            .setPassword("1");
    return Redisson.create(config);
}
```

3.获取锁操作：

```
long waitTimeout = 10;
long leaseTime = 1;
RLock lock1 = redissonClient1.getLock("lock1");
RLock lock2 = redissonClient2.getLock("lock2");
RLock lock3 = redissonClient3.getLock("lock3");

RedissonRedLock redLock = new RedissonRedLock(lock1, lock2, lock3);
// 同时加锁：lock1 lock2 lock3
// 红锁在大部分节点上加锁成功就算成功，且设置总超时时间以及单个节点超时时间
redLock.trylock(waitTimeout,leaseTime,TimeUnit.SECONDS);
...
redLock.unlock();
```

总结

实现分布式锁的方式有很多，有最简单的数据库实现，还有Zookeeper多节点实现和缓存实现。我们可以分别对这三种实现方式进行性能压测，可以发现在同样的服务器配置下，Redis的性能是最好的，Zookeeper次之，数据库最差。

从实现方式和可靠性来说，Zookeeper的实现方式简单，且基于分布式集群，可以避免单点问题，具有比较高的可靠性。因此，在对业务性能要求不是特别高的场景中，我建议使用Zookeeper实现的分布式锁。

思考题

我们知道Redis分布式锁在集群环境下会出现不同应用服务同时获得锁的可能，而Redisson中的Redlock算法很好地解决了这个问题。那Redisson实现的分布式锁是不是就一定不会出现同时获得锁的可能呢？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

精选留言

-W.LI-

4

老师好!基于数据库的实现，我现在项目中直接不开事务，select后插入(oeder_no做唯一约束)。try_catch 异常，重试3次。如果查到了返回成功保证密等。这么做会有问题么?

课后题:万一收到的N/2+1节点全部挂了肯定会有问题。不知道，从新选为master节点的算法不知，如果会选择没有收到的节点做master也会有问题。

2019-08-24

作者回复

没有问题。

问题的答案：redis实现的分布式锁，都是有一个过期时间，如果一旦服务A出现stop the world的情况，有可能锁过期了，而此时服务A中仍然存在持有锁，此时另外一个服务B又获取了锁，这个时候存在两个服务同时获取锁的可能。

2019-08-26

a、 4

不一定，因为如果集群中有5个redis，abcde，如果发生网络分区，abc在一个分区，de在一个分区，客户端A向abc申请锁成功，在c节点master异步同步slave的时候，master宕机了，slave接替，然后c的slave又和de在一个分区里，这时候如果客户端B来申请锁，也就可以成功了。zk锁也会出现问题，如果客户端A申请zk锁成功，这时候客户端A和zk不在一个分区里，zk就会把临时节点删除，然后如果客户端B再去申请，也就可以申请成功

2019-08-24

作者回复

对的，这种情况也是可能发生的，前提是c节点在宕机之前没有持久化锁。

第二zk锁的问题，如果连接session已经断开，客户端的锁是会释放的，不会存在同时获取锁的情况。

2019-08-27

我已经设置了昵称 1

不太懂redission机制，每个节点各自去获取锁。超过一半以上获取成功就算成功。那是不是还有这么一步：这些一半以上的机器获取了以后，是否还要决定谁真正拿到锁，才能真正执行这个任务

2019-08-25

作者回复

都会设置锁对象

2019-08-26

风轻扬 0

老师，redisson实现的分布式锁。您写的例子
.setScanInterval(2000) //集群状态的扫描时间，单位是毫秒
这个设置有什么用啊？

2019-10-14

作者回复

这是官方给出的一种连接redis集群的参考方式，具体作用已经写出了，类似一个心跳机制：
https://github.com/redisson/redisson/wiki/2.-%E9%85%8D%E7%BD%AE%E6%96%B9%E6%B

3%95#24-%E9%9B%86%E7%BE%A4%E6%A8%A1%E5%BC%8F

2019-10-19

风轻扬

0

老师，我试了一下redisson实现的分布式锁。有两个问题请教您。

1 .redis的集群模式，我在一台机器上建了一个伪集群。创建集群时，一共6个节点。3主3从。从节点是自动分配的。从节点的只读模式需要改成no吗?(不改成no，往从节点写锁，就会转移到主节点上去)

2 .字数限制，没办法贴代码，我在您例子的基础上增加了3个节点，也就是6个节点。核心代码如下：

```
final long waitTimeout = 10;
final long leaseTime = 3;
final RLock lock1 = redissonClient1.getLock("lock1");
final RLock lock2 = redissonClient2.getLock("lock2");
final RLock lock3 = redissonClient3.getLock("lock3");
final RLock lock4 = redissonClient4.getLock("lock4");
final RLock lock5 = redissonClient5.getLock("lock5");
final RLock lock6 = redissonClient6.getLock("lock6");

for (int i = 0; i < 10; i++) {
new Thread(new Runnable() {
@Override
public void run() {
RedissonRedLock redLock = new RedissonRedLock(lock1, lock2, lock3, lock4, lock5, lock6);
try {
if (redLock.tryLock(waitTimeout, leaseTime, TimeUnit.SECONDS)) {
//业务逻辑
}
} catch (InterruptedException e) {
e.printStackTrace();
} finally {
redLock.unlock();
}
}
}).start();
}
```

使用6个节点来实现redisson，但是获取锁一直失败，怎么回事呢？老师

2019-10-14

疯狂咸鱼

0

老师，分布式锁到底锁什么呢，如果说是锁数据库表，分布式应用集群的情况下，如果是单机数据库，数据库自身的锁机制可以保证并发问题吧？难道是分布式锁只是用在数据库分库分表的情况下？

2019-10-09

作者回复

分布式锁是在分布式服务的情况下保证原子性操作，而不是因为数据库产生的分布式锁。

数据库可以实现分布式锁，是一种实现方式。

2019-10-13

风轻扬

0

老师，我试了一下zookeeper的集群分布式锁。测试代码如下：

```
public class TestZookeeperLock {
private static int count = 10;

public static void main(String[] args) {
//重试策略,以下写法为:重试3次，每次间隔时间为3秒
final RetryPolicy retryPolicy = new RetryNTimes(3,2000);
final String connUrl = "192.111.111.111:2181,192.222.222.222:2181";
for (int i = 0; i < 10; i++) {
new Thread(new Runnable() {
@Override
public void run() {
//zookeeper分布式锁
CuratorFramework zk = CuratorFrameworkFactory.newClient(connUrl, retryPolicy);
zk.start();
InterProcessMutex lock = new InterProcessMutex(zk, "/opt/uams/zookeeper-3.4.7/locks");
try {
if (lock.acquire(3, TimeUnit.SECONDS)){
get();
}
} catch (Exception e) {
e.printStackTrace();
} finally {
try {
//释放锁
lock.release();
} catch (Exception e) {
e.printStackTrace();
}
}
}
}).start();
}
}

public static void get (){
```

```
count --;
if (count == 3) {
try {
TimeUnit.SECONDS.sleep(3);//这里设置该线程睡眠2秒,已达到锁住效果
} catch (InterruptedException e) {
e.printStackTrace();
}
}
System.out.println(count);
}
}
```

输出了：

```
9
8
7
6
5
4
java.lang.IllegalMonitorStateException: You do not own the lock: /opt/uams/zookeeper-3.4.7/locks
at org.apache.curator.framework.recipes.locks.InterProcessMutex.release(InterProcessMutex.java:140)
at cn.org.test.TestZookeeperLock$1.run(TestZookeeperLock.java:47)
at java.lang.Thread.run(Thread.java:745)
3
```

多次输出的结果一致，这是怎么回事呢?

2019-09-26

作者回复

将以下代码提出到new Thread之外：

```
//zookeeper分布式锁
CuratorFramework zk = CuratorFrameworkFactory.newClient(connUrl, retryPolicy);
zk.start();
InterProcessMutex lock = new InterProcessMutex(zk, "/opt/uams/zookeeper-3.4.7/locks");
```

2019-10-06

风轻扬 0

老师，zk实现的锁，不会出现redis锁一样的问题吗？

设想：

应用1和应用2两个服务分别部署到不同的服务器上。是使用zookeeper实现分布式锁。应用1获取到锁，然后开始长时间gc，应用2也开始长时间gc。应用1的zk锁由于心跳超时释放了锁，应用2结束gc获取到锁，应用1结束gc开始执行任务，此时不就有两个任务在同时执行了吗？

2019-09-25

作者回复

是的，这种情况也同样存在同时获取锁的可能

2019-09-25

风轻扬 0

老师，互联网行业，多数都是redis集群啊，如果这样，基于redis实现的分布式锁是不是就不能用了？

2019-09-24

作者回复

可以，使用Redisson就好了

2019-09-25

godtrue 0

我们的导入功能就是用的redis分布式锁，防止多个业务操作人员同时导入，超时时间一般为五分钟。

出现网络分区只能二选一要A或者C，不过互联网企业基本都会选择A。

2019-09-13

木刻 0

老师你好，我尝试了下第一个，模拟并发情况下发现会有概率抛数据库异常： Deadlock found when trying to get lock; try restarting transaction

https://github.com/mygodmele/DbLock.git

2019-09-11

作者回复

运行了代码，并没有出现死锁问题，麻烦贴出数据库脚本

2019-09-15

K 0

老师好，课后问题还是没听懂，首先我理解redis集群可能同时获取锁，是因为锁时间超时了，别的线程也能拿到，是这个原因。Redlock 算法是怎样解决这个问题的呢？

2019-09-08

作者回复

RedLock算法是会去每一个节点获取锁，正常情况下，别的线程无法同时获取锁的。

2019-09-11

知行合一 0

老师，想问个问题，redis集群已经分了槽，客户端写入根据算法应该写入一个节点啊，为啥要多个节点同时枷锁？

2019-09-05

作者回复

写入一个单点只实现了高可用，没有实现集群式分布式锁。单点的问题会存在单个节点挂了的情况下，不同应用服务同时获取锁的可能。

2019-09-07

Jxin 0

1.锁超时，也会出现多个任务同时持有锁进行。
2.解决方式，守护线程续航锁持有时间。
3.弊端，浪费线程，开销太大。
4.根据业务情况设置合理的超时时间是最棒的。

5.集群环境还会导致事务失效（同时提交多个key，多个key在不同节点）挺蛋疼。

2019-09-04

再续啸傲 0

Redisson的“看门狗”watch机制，解决了业务执行时间长于锁过期时间的问题。但是为每一个获取锁的线程设置监听线程，会不会在高并发的场景下耗费过多资源呢？

2019-09-03

作者回复

应该是一个线程监听，具体需要看源码实现。

2019-09-07

zero 0

用etcd实现锁，是不是更好呢

2019-08-28

rong 0

老师，使用select for update防止幻读那里，直接把order_no设置成唯一索引，事务里面只有一条insert语句就可以吧？如果之前有，插入不成功，没有的话，插入成功

2019-08-27

作者回复

是的，唯一索引可以实现该功能。

2019-08-28

-W.LI- 0

谢谢老师!STW问题之前都没想到，不过正常情况STP时间比较短的吧，除非是CMS下的超大老年代，或者代码不合理。G1分segment回收STW应该不会长吧。项目中数据库锁和redis锁用的比较多，不过超时时间都是随意设置10，20S。正常一般几十ms就能就能完成的。请问redis锁超时时间设置多少比较合理呢?项目中大部分情况锁冲突概率比较小。电商项目，商家余额这种冲突概率很大的适合用zk锁是么?

2019-08-26

作者回复

是的，根据自己的需求设定。zk锁则没有超时时间问题。

2019-08-30

我已经设置了昵称 0

数据库实现，select for update是为了放置幻读？是为了同时两个线程走到同一行查询代码，然

后插入两遍的意思吗？那后面的把查询和插入放同一个事务里面的作用是什么？请老师指点下，这边还是不太懂

2019-08-26

作者回复

是的，这是一个间隙锁，可以防止两个事务插入相同订单号的数据。将查询和插入作为一个事务，是保证在查询没有订单时，然后才能插入数据。

2019-08-26

明天更美好

0

我对redisson不是很了解，只是之前看过一些别的帖子，好像底层也是有用lua脚本的。如果对于原生的还好些，但是有些公司自研的分布式缓存是不支持lua的。这时候恐怕就不适用了

2019-08-25