

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

上一章节我提到了基于zk分布式锁的实现，这章节就来说一下基于Redis的分布式锁实现吧。

- zk实现分布式锁的传送门：[zk分布式锁](#)

在开始提到Redis分布式锁之前，我想跟大家聊点Redis的基础知识。

说一下Redis的两个命令：

```
SETNX key value
```

setnx 是SET if Not eXists(如果不存在，则 SET)的简写。

```
127.0.0.1:6379> setnx ab love
(integer) 1
127.0.0.1:6379> setnx ab kk
(integer) 0
127.0.0.1:6379> get ab
"love"
```

用法如图，如果不存在set成功返回int的1，这个key存在了返回0。

```
SETEX key seconds value
```

将值 value 关联到 key ，并将 key 的生存时间设为 seconds (以秒为单位)。

如果 key 已经存在， setex 命令将覆写旧值。

有小伙伴肯定会疑惑万一set value 成功 set time失败，那不就傻了么，这啊Redis官网想到了。

setex 是一个原子性(atomic)操作，关联值和设置生存时间两个动作会在同一时间内完成。

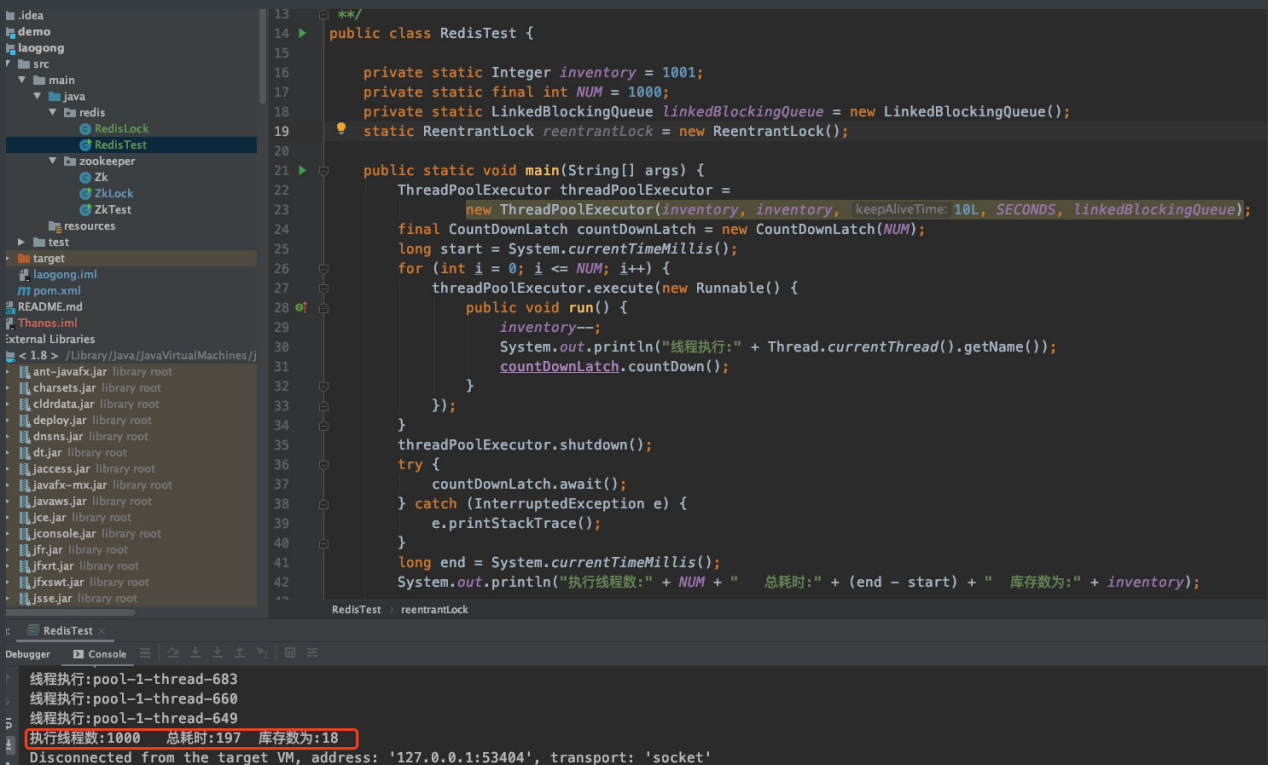
```
127.0.0.1:6379> setex lock 10 test
OK
127.0.0.1:6379> get lock
"test"
127.0.0.1:6379> ttl lock
(integer) -2
127.0.0.1:6379> get lock
(nil)
```

我设置了10秒的失效时间，ttl命令可以查看倒计时，负的说明确已经到期了。

跟大家讲这两个命名也是有原因的，因为他们是Redis实现分布式锁的关键。

正文

开始前还是看看场景：



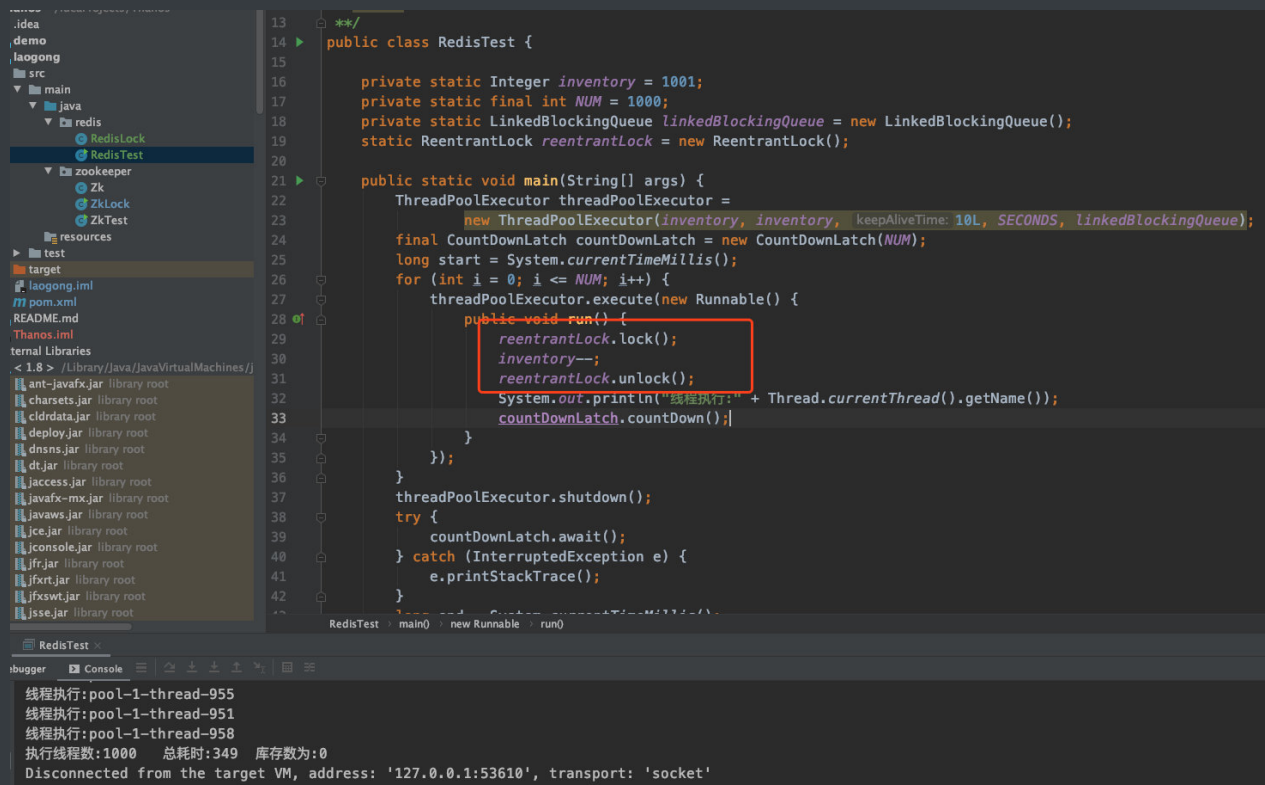
```
13  /**
14  public class RedisTest {
15
16      private static Integer inventory = 1001;
17      private static final int NUM = 1000;
18      private static LinkedBlockingQueue linkedBlockingQueue = new LinkedBlockingQueue();
19      static ReentrantLock reentrantLock = new ReentrantLock();
20
21      public static void main(String[] args) {
22          ThreadPoolExecutor threadPoolExecutor =
23              new ThreadPoolExecutor(inventory, inventory, keepAliveTime: 10L, SECONDS, linkedBlockingQueue);
24          final CountdownLatch countDownLatch = new CountdownLatch(NUM);
25          long start = System.currentTimeMillis();
26          for (int i = 0; i <= NUM; i++) {
27              threadPoolExecutor.execute(new Runnable() {
28                  public void run() {
29                      inventory--;
30                      System.out.println("线程执行:" + Thread.currentThread().getName());
31                      countDownLatch.countDown();
32                  }
33              });
34          }
35          threadPoolExecutor.shutdown();
36          try {
37              countDownLatch.await();
38          } catch (InterruptedException e) {
39              e.printStackTrace();
40          }
41          long end = System.currentTimeMillis();
42          System.out.println("执行线程数:" + NUM + "    总耗时:" + (end - start) + "    库存数为:" + inventory);
43      }
44  }
```

RedisTest -> reentrantLock

```
线程执行:pool-1-thread-683
线程执行:pool-1-thread-660
线程执行:pool-1-thread-649
执行线程数:1000    总耗时:197    库存数为:18
Disconnected from the target VM, address: '127.0.0.1:53404', transport: 'socket'
```

我依然是创建了很多个线程去扣减库存inventory，不出意外的库存扣减顺序变了，最终的结果也是不对的。

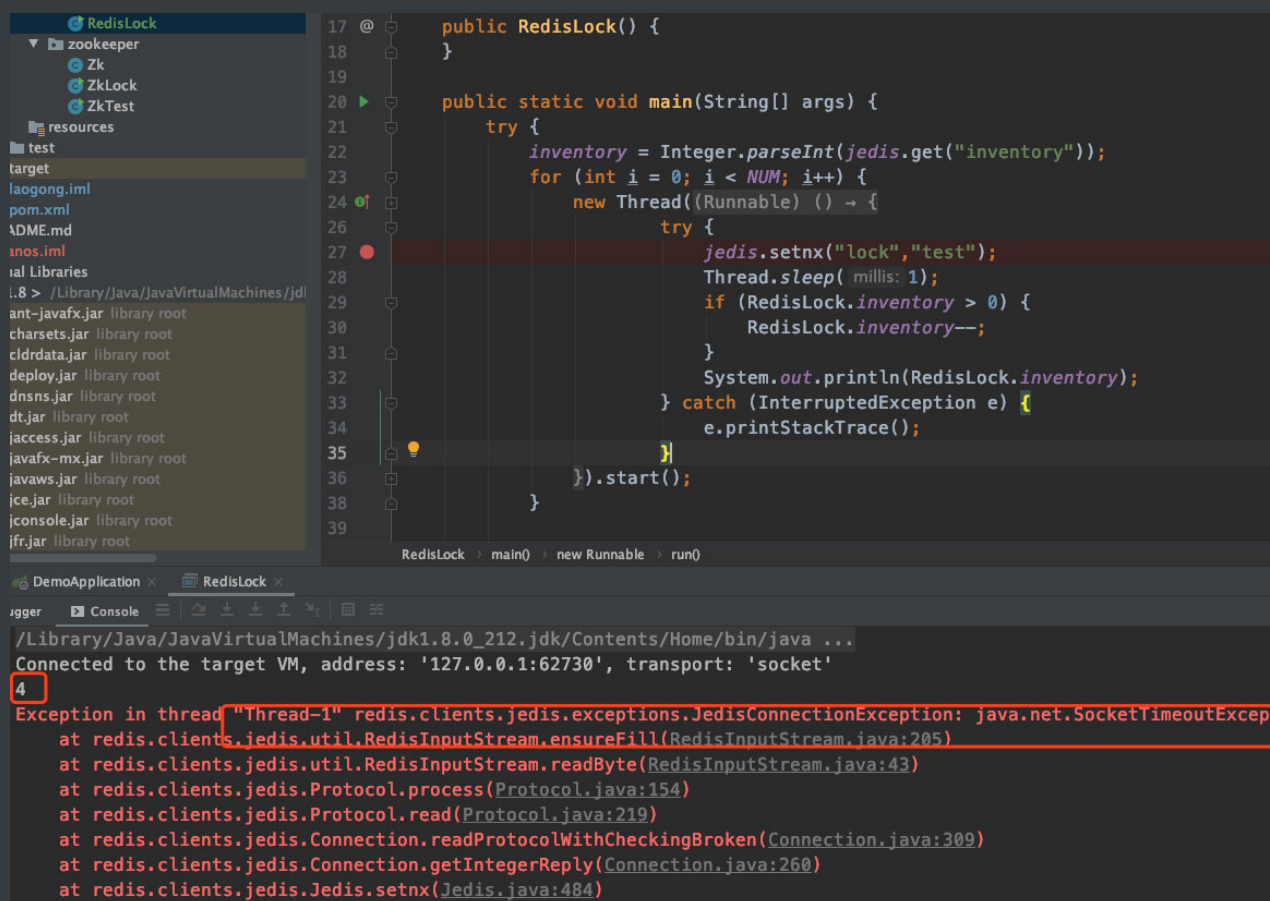
单机加synchronized或者Lock这些常规操作我就不说了好吧，结果肯定是对的。



我先实现一个简单的Redis锁，然后我们再实现分布式锁，可能更方便大家的理解。

还记得上面我说过的命令么，实现一个单机的其实比较简单，你们先思考一下，别往下看。

setnx



可以看到，第一个成功了，没释放锁，后面的都失败了，至少顺序问题问题是解决了，只要加锁，缩放后面的拿到，释放如此循环，就能保证按照顺序执行。

但是你们也发现问题了，还是一样的，第一个set成功了，但是突然挂了，那锁就一直在那无法得到释放，后面的线程也永远得不到锁，又死锁了。

所以....

setex

知道我之前说这个命令的原因了吧，设置一个过期时间，就算线程1挂了，也会在失效时间到了，自动释放。

我这里就用到了nx和px的结合参数，就是set值并且加了过期时间，这里我还设置了一个过期时间，就是这时间内如果第二个没拿到第一个的锁，就退出阻塞了，因为可能是客户端断连了。

```
private String LOCK_KEY = "redis_lock"; // key
protected long INTERNAL_LOCK_LEASE_TIME = 3; // 自动失效时间
private long timeout = 1000; // 超时时间还没拿到 自动退出
private SetParams params = SetParams.setParams().nx().px(INTERNAL_LOCK_LEASE_TIME); // nx px 命令的合集
private static Jedis jedis = new JedisPool( host: "127.0.0.1", port: 6379).getResource(); // redis 客户端
```

加锁

整体加锁的逻辑比较简单，大家基本上都能看懂，不过我拿到当前时间去减开始时间的操作感觉有点笨，System.currentTimeMillis()消耗很大的。

```
/**
 * 加锁
 *
 * @param id
 * @return
 */
public boolean lock(String id) {
    Long start = System.currentTimeMillis();
    try {
        for (; ; ) {
            //SET命令返回OK，则证明获取锁成功
            String lock = jedis.set(LOCK_KEY, id, params);
            if ("OK".equals(lock)) {
                return true;
            }
            //否则循环等待，在timeout时间内仍未获取到锁，则获取失败
            long l = System.currentTimeMillis() - start;
```

```

        if (l >= timeout) {
            return false;
        }
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
} finally {
    jedis.close();
}
}

```

System.currentTimeMillis消耗大，每个线程进来都这样，我之前写代码，就会在服务器启动的时候，开一个线程不断去拿，调用方直接获取值就好了，不过也不是最优解，日期类还是有很多好方法的。

```

@Service
public class TimeServcie {
    private static long time;
    static {
        new Thread(new Runnable(){
            @Override
            public void run() {
                while (true){
                    try {
                        Thread.sleep(5);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    long cur = System.currentTimeMillis();
                    setTime(cur);
                }
            }
        }).start();
    }

    public static long getTime() {
        return time;
    }

    public static void setTime(long time) {
        TimeServcie.time = time;
    }
}

```

```
}
```

解锁

解锁的逻辑更加简单，就是一段Lua的拼装，把Key做了删除。

你们发现没，我上面加锁解锁都用了UUID，这就是为了保证，谁加锁了谁解锁，要是你删掉了我的锁，那不乱套了嘛。

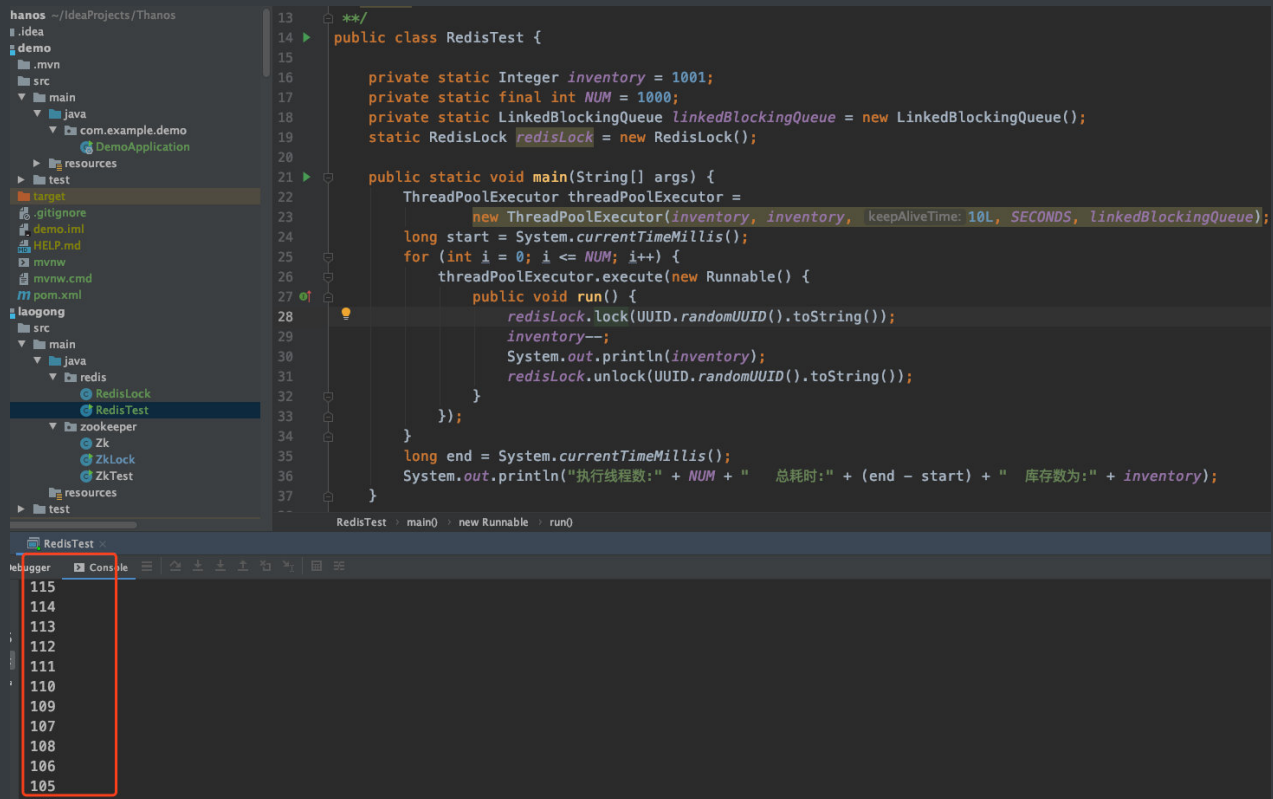
LUA是原子性的，也比较简单，就是判断一下Key和我们参数是否相等，是的话就删除，返回成功1，0就是失败。

```
/**
 * 解锁
 *
 * @param id
 * @return
 */
public boolean unlock(String id) {
    String script =
        "if redis.call('get',KEYS[1]) == ARGV[1] then" +
        "    return redis.call('del',KEYS[1]) " +
        "else" +
        "    return 0 " +
        "end";

    try {
        String result = jedis.eval(script,
            Collections.singletonList(LOCK_KEY),
            Collections.singletonList(id)).toString();
        return "1".equals(result) ? true : false;
    } finally {
        jedis.close();
    }
}
```

验证

我们可以用我们写的Redis锁试试效果，可以看到都按照顺序去执行了



思考

大家是不是觉得完美了，但是上面的锁，有不少瑕疵的，我没思考很多点，你或许可以思考一下，源码我都开源到我的GitHub了。

而且，锁一般都是需要可重入行的，上面的线程都是执行完了就释放了，无法再次进入了，进去也是重新加锁了，对于一个锁的设计来说肯定不是很合理的。

我不打算手写，因为都有现成的，别人帮我们写好了。

redisson

redisson的锁，就实现了可重入了，但是他的源码比较晦涩难懂。

使用起来很简单，因为他们底层都封装好了，你连接上你的Redis客户端，他帮你做了我上面写的一切，然后更完美。

简单看看他的使用吧，跟正常使用Lock没啥区别。

```
ThreadPoolExecutor threadPoolExecutor =
    new ThreadPoolExecutor(inventory, inventory, 10L, SECONDS,
        linkedBlockingQueue);
long start = System.currentTimeMillis();
Config config = new Config();
config.useSingleServer().setAddress("redis://127.0.0.1:6379");
final RedissonClient client = Redisson.create(config);
```



```

final RLock lock = client.getLock("lock1");

for (int i = 0; i <= NUM; i++) {
    threadPoolExecutor.execute(new Runnable() {
        public void run() {
            lock.lock();
            inventory--;
            System.out.println(inventory);
            lock.unlock();
        }
    });
}
long end = System.currentTimeMillis();
System.out.println("执行线程数:" + NUM + "    总耗时:" + (end - start) + "    库存数为:" + inventory);

```

上面可以看到我用到了getLock，其实就是获取一个锁的实例。

RedissonLock 也没做啥，就是熟悉的初始化。

```

public RLock getLock(String name) {
    return new RedissonLock(connectionManager.getCommandExecutor(), name);
}

public RedissonLock(CommandAsyncExecutor commandExecutor, String name) {
    super(commandExecutor, name);
    //命令执行器
    this.commandExecutor = commandExecutor;
    //UUID字符串
    this.id = commandExecutor.getConnectionManager().getId();
    //内部锁过期时间
    this.internalLockLeaseTime = commandExecutor.
        getConnectionManager().getCfg().getLockWatchdogTimeout();
    this.entryName = id + ":" + name;
}

```

加锁

有没有发现很多跟Lock很多相似的地方呢？

尝试加锁，拿到当前线程，然后我开头说的ttl也看到了，是不是一切都是那么熟悉？


```

public void lockInterruptibly(long leaseTime, TimeUnit unit) throws
InterruptedException {

    //当前线程ID
    long threadId = Thread.currentThread().getId();
    //尝试获取锁
    Long ttl = tryAcquire(leaseTime, unit, threadId);
    // 如果ttl为空, 则证明获取锁成功
    if (ttl == null) {
        return;
    }
    //如果获取锁失败, 则订阅到对应这个锁的channel
    RFuture<RedissonLockEntry> future = subscribe(threadId);
    commandExecutor.syncSubscription(future);

    try {
        while (true) {
            //再次尝试获取锁
            ttl = tryAcquire(leaseTime, unit, threadId);
            //ttl为空, 说明成功获取锁, 返回
            if (ttl == null) {
                break;
            }
            //ttl大于0 则等待ttl时间后继续尝试获取
            if (ttl >= 0) {
                getEntry(threadId).getLatch().tryAcquire(ttl,
TimeUnit.MILLISECONDS);
            } else {
                getEntry(threadId).getLatch().acquire();
            }
        }
    } finally {
        //取消对channel的订阅
        unsubscribe(future, threadId);
    }
    //get(lockAsync(leaseTime, unit));
}

```

获取锁

获取锁的时候, 也比较简单, 你可以看到, 他也是不断刷新过期时间, 跟我上面不断去拿当前时间, 校验过期是一个道理, 只是我比较粗糙。

```

private <T> RFuture<Long> tryAcquireAsync(long leaseTime, TimeUnit unit,
final long threadId) {

    //如果带有过期时间，则按照普通方式获取锁
    if (leaseTime != -1) {
        return tryLockInnerAsync(leaseTime, unit, threadId,
RedisCommands.EVAL_LONG);
    }

    //先按照30秒的过期时间来执行获取锁的方法
    RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(

commandExecutor.getConnectionManager().getCfg().getLockWatchdogTimeout(),
        TimeUnit.MILLISECONDS, threadId, RedisCommands.EVAL_LONG);

    //如果还持有这个锁，则开启定时任务不断刷新该锁的过期时间
    ttlRemainingFuture.addListener(new FutureListener<Long>() {
        @Override
        public void operationComplete(Future<Long> future) throws Exception
    {
        if (!future.isSuccess()) {
            return;
        }

        Long ttlRemaining = future.getNow();
        // lock acquired
        if (ttlRemaining == null) {
            scheduleExpirationRenewal(threadId);
        }
    }
    });
    return ttlRemainingFuture;
}

```

底层加锁逻辑

你可能会想这么多操作，在一起不是原子性不还是有问题么？

大佬们肯定想得到呀，所以还是LUA，他使用了Hash的数据结构。

主要是判断锁是否存在，存在就设置过期时间，如果锁已经存在了，那对比一下线程，线程是一个那就证明可以重入，锁在了，但是不是当前线程，证明别人还没释放，那就把剩余时间返回，加锁失败。

是不是有点绕，多理解一遍。

```

<T> RFuture<T> tryLockInnerAsync(long leaseTime, TimeUnit unit,
                                long threadId, RedisStrictCommand<T> command) {

    //过期时间
    internalLockLeaseTime = unit.toMillis(leaseTime);

    return commandExecutor.evalWriteAsync(getName(),
LongCodec.INSTANCE, command,
        //如果锁不存在，则通过hset设置它的值，并设置过期时间
        "if (redis.call('exists', KEYS[1]) == 0) then " +
            "redis.call('hset', KEYS[1], ARGV[2], 1); " +
            "redis.call('pexpire', KEYS[1], ARGV[1]); " +
            "return nil; " +
        "end; " +
        //如果锁已存在，并且锁的是当前线程，则通过hincrby给数值递增1
        "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then "
+
            "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
            "redis.call('pexpire', KEYS[1], ARGV[1]); " +
            "return nil; " +
        "end; " +
        //如果锁已存在，但并非本线程，则返回过期时间ttl
        "return redis.call('pttl', KEYS[1]);",
        Collections.<Object>singletonList(getName()),
        internalLockLeaseTime, getLockName(threadId));
}

```

解锁

锁的释放主要是publish释放锁的信息，然后做校验，一样会判断是否当前线程，成功就释放锁，还有个**hincrby**递减的操作，锁的值大于0说明是可重入锁，那就刷新过期时间。

如果值小于0了，那删掉Key释放锁。

是不是又和AQS很像了？

AQS就是通过一个volatile修饰status去看锁的状态，也会看数值判断是否是可重入的。

所以我说代码的设计，最后就万剑归一，都是一样的。

```

public RFuture<Void> unlockAsync(final long threadId) {
    final RPromise<Void> result = new RedissonPromise<Void>();

    //解锁方法

```

```

RFuture<Boolean> future = unlockInnerAsync(threadId);

future.addListener(new FutureListener<Boolean>() {
    @Override
    public void operationComplete(Future<Boolean> future) throws
Exception {
        if (!future.isSuccess()) {
            cancelExpirationRenewal(threadId);
            result.tryFailure(future.cause());
            return;
        }
        //获取返回值
        Boolean opStatus = future.getNow();
        //如果返回空, 则证明解锁的线程和当前锁不是同一个线程, 抛出异常
        if (opStatus == null) {
            IllegalMonitorStateException cause =
                new IllegalMonitorStateException("
thread by node id: "
                    + id + " thread-id: " + threadId);
            result.tryFailure(cause);
            return;
        }
        //解锁成功, 取消刷新过期时间的那个定时任务
        if (opStatus) {
            cancelExpirationRenewal(null);
        }
        result.trySuccess(null);
    }
});

return result;
}

protected RFuture<Boolean> unlockInnerAsync(long threadId) {
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE,
EVAL,

        //如果锁已经不存在, 发布锁释放的消息
        "if (redis.call('exists', KEYS[1]) == 0) then " +
            "redis.call('publish', KEYS[2], ARGV[1]); " +
            "return 1; " +
        "end;" +
        //如果释放锁的线程和已存在锁的线程不是同一个线程, 返回null

```

```

        "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " +
            "return nil;" +
        "end; " +
        //通过hincrby递减1的方式，释放一次锁
        //若剩余次数大于0，则刷新过期时间
        "local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1); "
+
        "if (counter > 0) then " +
            "redis.call('pexpire', KEYS[1], ARGV[2]); " +
            "return 0; " +
        //否则证明锁已经释放，删除key并发布锁释放的消息
        "else " +
            "redis.call('del', KEYS[1]); " +
            "redis.call('publish', KEYS[2], ARGV[1]); " +
            "return 1; "+
        "end; " +
        "return nil;";
    Arrays.<Object>asList(getName(), getChannelName()),
        LockPubSub.unlockMessage, internalLockLeaseTime,
        getLockName(threadId));
}

```

总结

这个写了比较久，但是不是因为复杂什么的，是因为个人工作的原因，最近事情很多嘛，还是那句话，程序员才是我的本职写文章只是个爱好，不能本末倒置了。

大家会发现，你学懂一个技术栈之后，学新的会很快，而且也能发现他们的设计思想和技巧真的很巧妙，也总能找到相似点，和让你惊叹的点。

就拿 Doug Lea 写的AbstractQueuedSynchronizer (AQS) 来说，他写了一行代码，你可能看几天才能看懂，大佬们的思想是真的牛。

我看源码有时候也头疼，但是去谷歌一下，自己理解一下，突然恍然大悟的时候觉得一切又很值。

学习就是一条时而郁郁寡欢，时而开环大笑的路，大家加油，我们成长路上一起共勉。

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，大家的「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，欢迎人才们留言，**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