# Redission：分布式场景中使用Redis的大杀器

## 手动实现基于Redis的分布式锁的问题

前面我们手动实现基于Redis的分布式锁，并针对锁误删做了一系列关进，基本可以认为最后的版本是生产可用的分布式锁了。

但是在极端情况下还是有问题，比如，线程一判断锁标识后，在执行释放锁的时候发生了阻塞，而阻塞的时候如果足够长，很有可能会触发锁的超时释放，锁一旦超时释放，那么此时别的线程又可以乘虚而入成功获取锁，而当线程二获取锁成功后，线程一阻塞醒来继续去下单业务，但是当下单操作还未执行完时线程一又阻塞了，于是就把线程二仍然能够满足条件去下单，那么再一次发生线程安全问题！！！

除此之外，该版本的锁仍然还有以下问题：



### 1：不可重入

可重入的意思是用一个线程可以多次获取同一把锁，就是说当前线程持有锁了，当前线程没释放锁时，就不能重新获取该把锁。

### 2：不可重试

此版本去获取锁是非阻塞式的，尝试获取锁如果失败了会立即返回false，没有重试机制，即获取锁不管成功与否，只会尝试一次。而在很多业务下不能立即失败，而是希望如果获取锁失败，要等一等再去尝试获取锁。

### 3：超时释放

锁超时释放虽然可以避免死锁，但如果业务执行耗时较长，也会导致锁释放，存在安全隐患，这就是我之前一直认为的还是存在线程安全的原因。

### 4：主从一致性：

如果redis提供了主从集群，所谓主从模式你也可以简单理解成读写分离模式，也就是说Redis集群会有一个主节点，多个从节点，主机负责写，从机负责读。主从同步存在延迟，我们线程在主节点获取了锁，因为获取锁是set操作，即Redis读操作，写操作在主节点完成了，但尚未同步到从节点，因为主从同步存在延迟，突然主宕机了，因为从机并未同步到主机中的锁数据，则会出现锁失效。

but，以上情况只会在极度极端的情况下出现，我们仍然可以认为我们手动实现的基于Redis的分布式锁已经是在生产可用的了。

## 你想要完美的分布式锁吗？Redission！

既然我编码能力有限，那有没有成熟的分布式锁的解决方案？可以使用Redission框架！

Redission是在Redis基础上实现的分布式工具的集合，即在分布式系统下用到的各种各样的Redis的东西Redission都有，自然也包括分布式锁！即分布式锁只是Redission的子集！

所以在企业环境下没有必自己去实现锁，用Redission就可以！

## Redisson入门使用

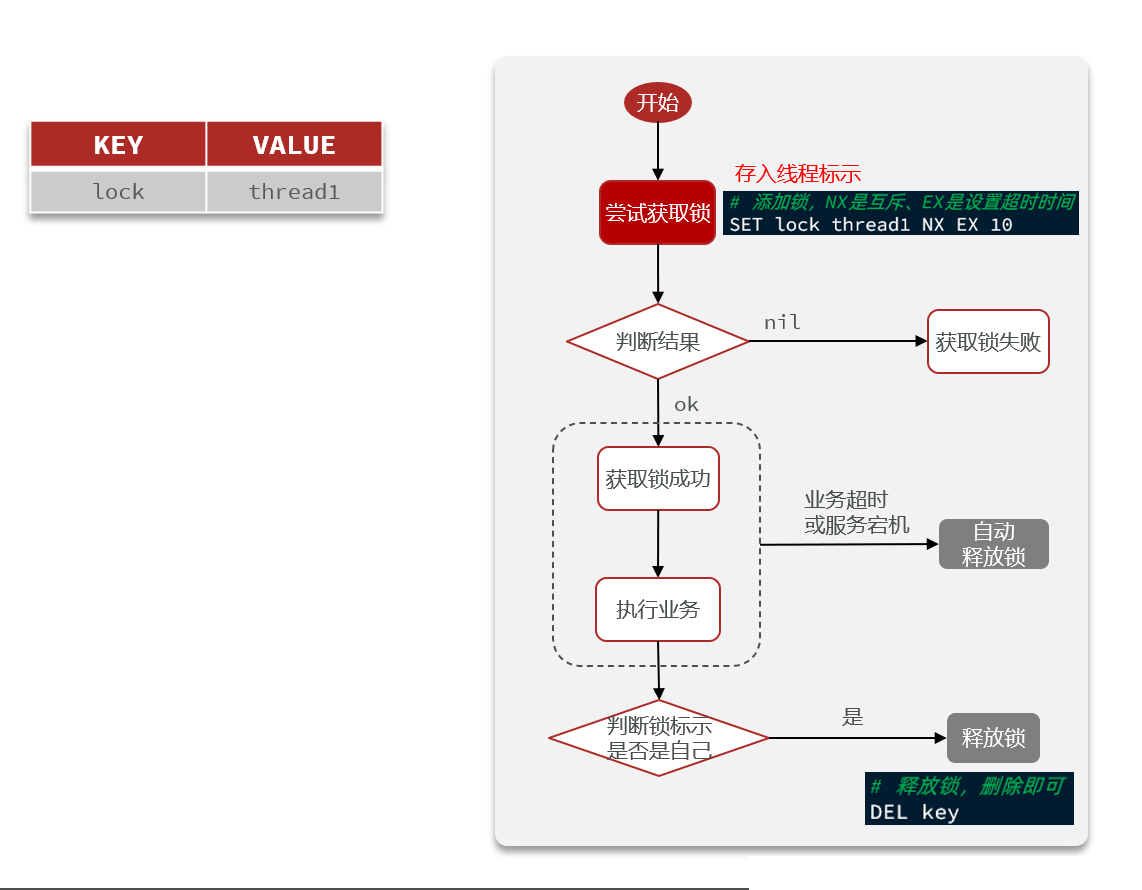




## Reedisson可重入锁原理

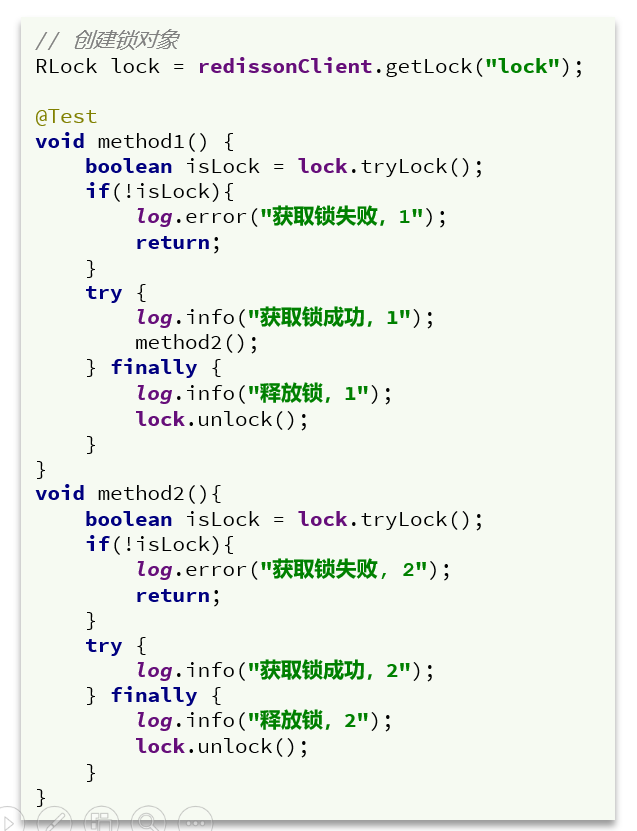
同学们，前面我们已经聊过，我们自定义的分布式锁有一个缺陷，就是无法实现可重入，而Redisson就可以做到可重入！这是怎么做到的呢？接下来我们一起研究探讨一下下。

我们自定的分布式锁采用的是Redis的String数据类型，也就是简单的key-value，整个获取锁的流程是这样子的，再一开始尝试获取锁其实就是执行set命令：**SET lock thread1 NX EX 10**，NX的目的就是为了让实现满足分布式锁的基本要求：互斥，同时获取锁的时候我们还存入了线程标识，用来记录是哪个线程获取的锁，目的就是为了将来在释放锁的时候做判断，避免锁的误删，只有锁是自己的采取做锁的删除。具体流程图如下图，那么这样的一个流程为什么不能做到重入呢？



我们可以通用下面的代码验证我们自定义的分布式锁是不可重入的，可以看到我们再method1中获取锁，然后在方法method2中再次尝试去获取锁。method1调用method2，所以这两个方法是在同一个线程里的。一个线程连续两次去获取同一把锁，这就是锁的重入了。

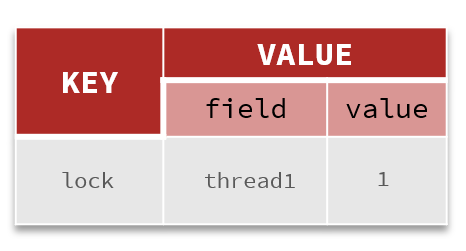
我们可以看看如果按照自定义分布式锁的流程，能不能去实现锁的重入！很明显是不能购重入的，因为只有method1能获取成功，method2不能获取锁成功，因为是NX互斥的。



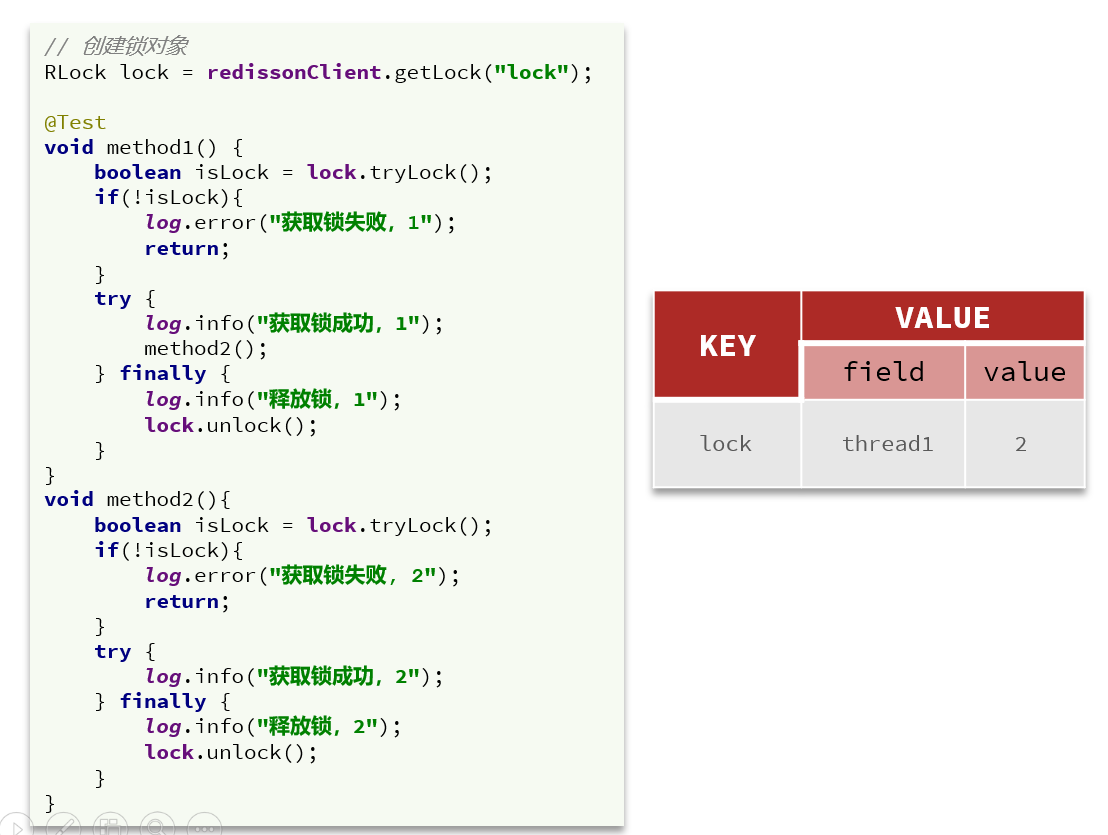
那怎么实现可重入的分布式锁呢？其实可以参考JDK里提供的可重入锁ReedtrantLock的原理。简单说一下ReedtrantLock为什么可以实现可重入：

线程在获取锁的时候，如果判断要获取的锁已经有人的情况下，就去看一下获取锁的是不是自己，也就是说是不是用一个线程，如果是同一个线程，那么也会让线程再次获取锁，但是要多做一件事，就是有计数器会记录线程重入的次数，即该线程总共获取锁几次，也就是说同一个线程在获取锁的时候，计数器会累加，在释放锁的时候，计数器会累减。这就是可重入锁的基本原理。

因此，我们要想实现可重入的分布式锁，也可以参考JDK的可重入锁ReedtrantLock的做法。也就是说我们不仅仅在锁里边要记录获取锁的线程，还要去记录这个线程的重入次数，该线程总共获取了多少次锁，每拿一次重入次数就加1。现在我们既要记录线程标识，又要记录重入次数，这样一来，我们基于Redis的String类型的结构实现分布式锁显然就不行了。很明显可以使用Hash类型的结构来满足在一个key里边存储两个东西：



那么这个时候我们就可以在key中记录分布式锁的名称，field位置记录线程标识，value位置记录线程获取锁的重入次数。那么这个时候，上面的代码就变得合情合理了，method1中获取锁，会记录线程标识，并且把重入次数重置为1，接着调用method2，method2重新尝试获取锁，首先要做的是这个锁是不是有人了，如果已经有人了，这个时候不代表获取锁失败，还需要判断一下获取锁的线程是不是我自己，因为method1调method2，所以method1和method2是在同一个线程里的，所以线程标识肯定是一样的，于是是自己，那么只需要把重入次数加1就可以了，代表我（线程）是第二次获取锁。以此类推，如果再有重入，那就继续累加就行了。

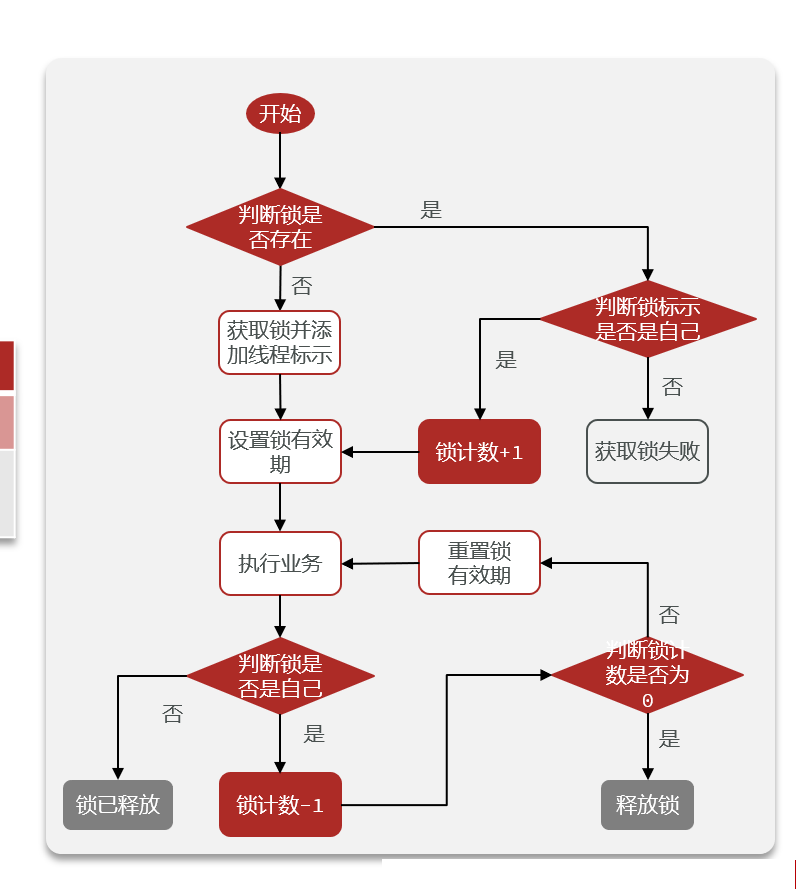


这个时候，method2拿到锁，已经重入了，就可以执行自己的业务了，执行完method2自己的业务后，就应该释放锁，释放锁不能像以前一样：首先判断一下锁的线程标识是不是自己，一判断是自己直接删除。不能直接删除，如果在method2里直接把锁删了，你要知道method2结束并不意味着整个业务结束，因为method2出来以后还要去执行method1中剩余业务，虽然我们的代码没写，但在真实开发中很可能有其他业务的，也就是说在method1业务尚未执行完的时候，method2直接就把锁删了，此时其他线程就能进来获取锁了，于是就会发生并发线程安全问题哦。所以说，对于可重入锁来讲，在内部被调用的方法里边，释放锁的时候，是不能直接删除锁的，二是应该把重入次数减1，也就是每释放一次重入次数减1就行了。那么问题来了，我们释放锁的时候到底什么时候才能真的把这个锁删除呢？肯定是当我们所有的业务都执行完，代码走到最外层的获取锁的方法，方法结束释放锁的时候才能删除锁。

我们每次获取锁重入次数加1，每次释放锁重入次数减1，即方法中获取锁和释放锁都是成对出现的，也就是说你获取锁时重入次数加了多少次，将来释放锁的时候重入次数就一定会减多少次，于是当一业务走到最外层的获取锁的方法去释放锁的时候，重入次数的值一定会减成0。所以说我们每次释放锁的时候，除了要把重入次数减1，还要判断这个重入次数的值是不是已经为0了，如果已经是0了，也就是说我们业务已经结束了，没有其他业务需要执行了，此时你就可以放心大胆地把锁删除了。

以上种种，就是我们可重入锁的原理。也就是我们使用Hash结构代替String结构，不仅仅存储线程标识，还要存储重入次数，因此获取锁和释放锁的操作跟以前就会有很大差别，以前我们采取的是String类型的结构，采用的命令是set nx ex，其中set nx是来判断锁是否存在实现互斥，set ex是设置锁的过期时间。但是现在我们用的是hash结构，hash结构里可没有这样的组合命令，即没有set nx ex这样的名令，所以说我们只能是把之前的set nx ex逻辑拆开，先判断锁存不存在，再手动来设置过期时间，exists命令判断锁是否存在，expire命令设置锁的有效期。

于是，利用Hash结构实现基于Redis的可重入分布式锁的流程就变成下面的：

、

注意上面流程图里面有个重置锁有效期的操作，给后续业务执行留够充足的执行时间。

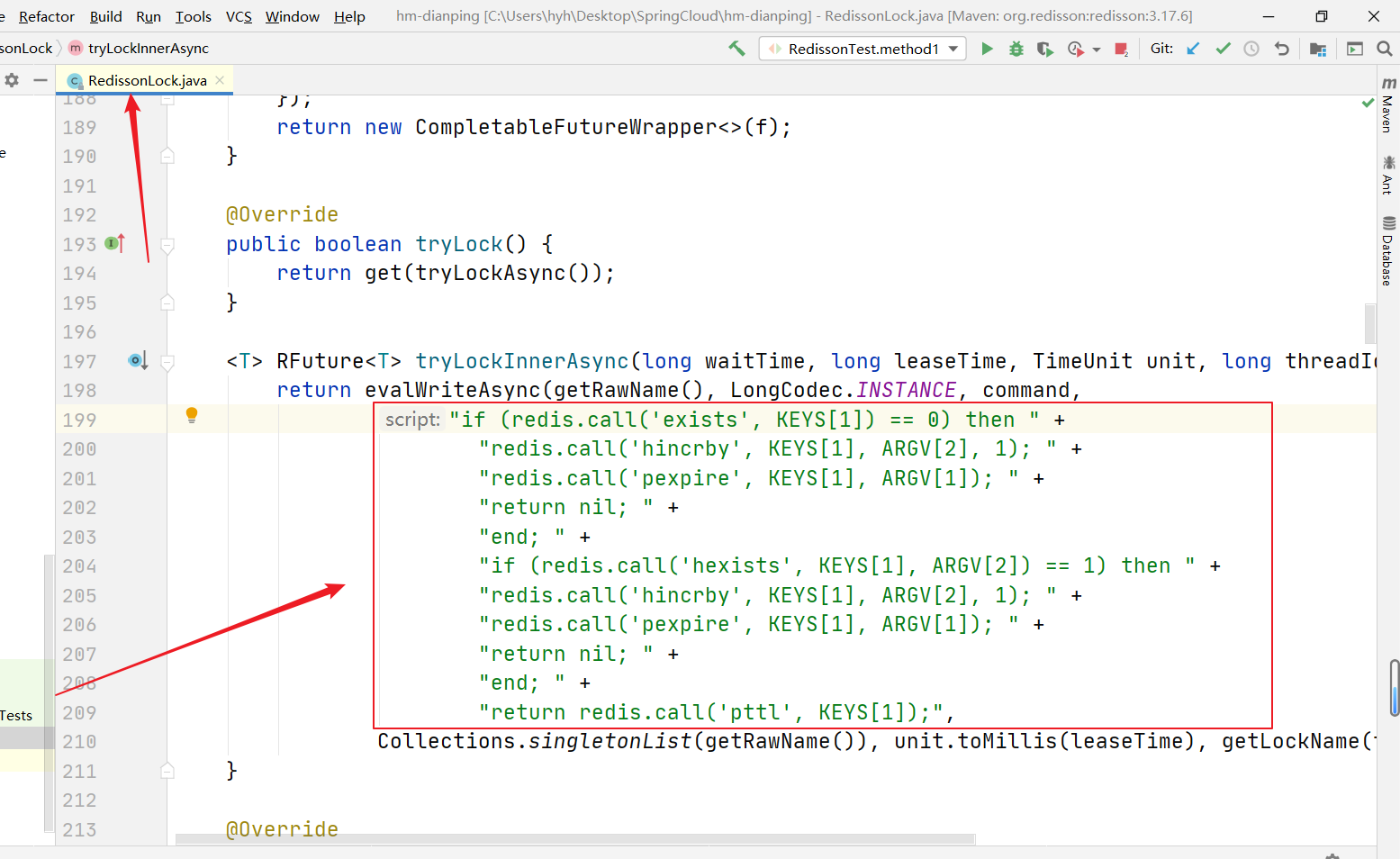
可以看到无论是获取锁还是释放锁，跟以前相比都复杂多了，而且代码有多个步骤，所以说像这样的逻辑我们不可能再用Java代码去实现了。必须用Lua脚本来确保获取锁操作的原子性和释放锁操作的原子性。

## 基于Redis实现可重入分布式锁获

### 获取锁的Lua脚本：



#### Redisson中获取重入锁的lua脚本（确保了获取锁操作的原子性）：



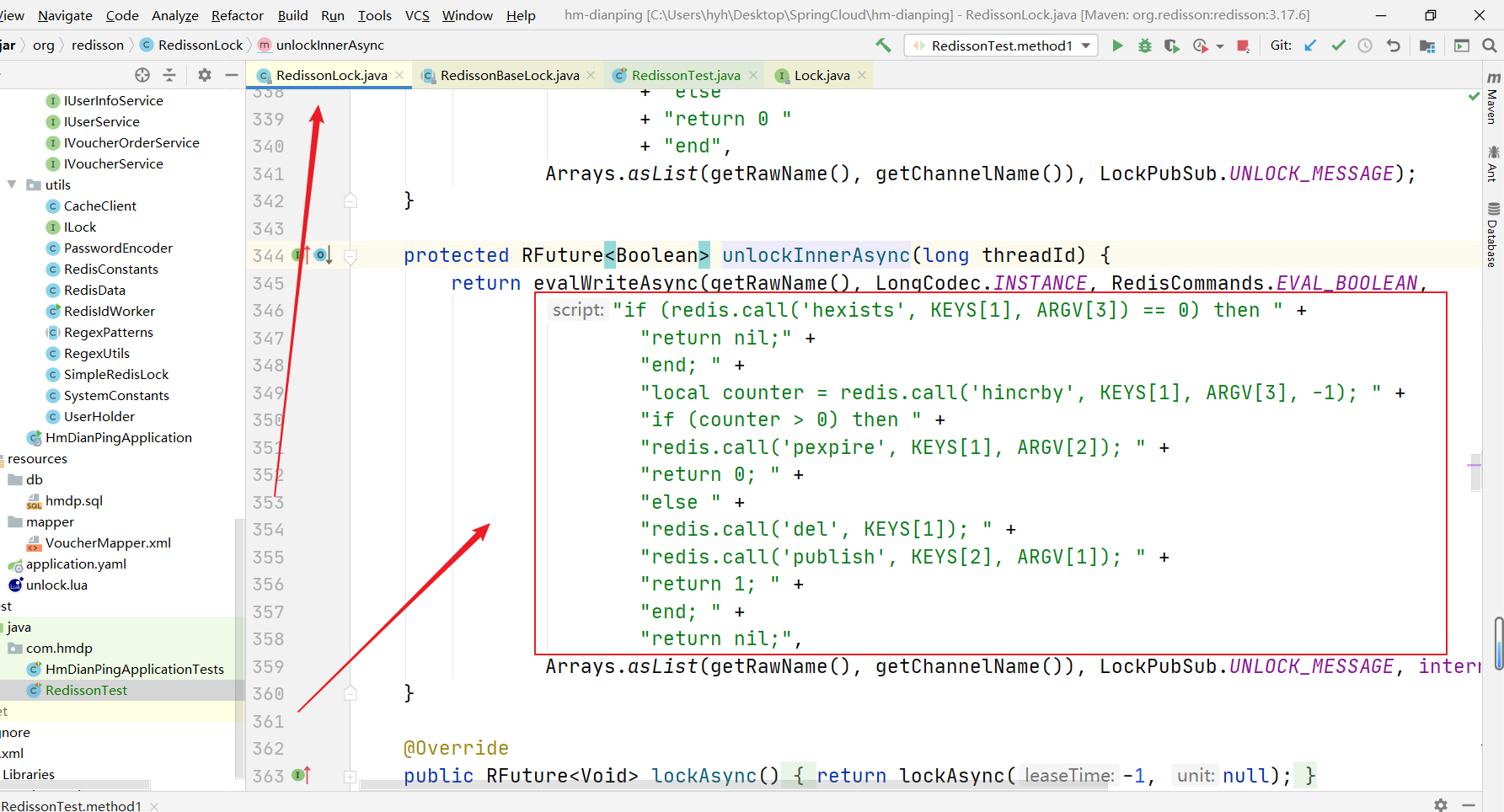
Lua脚本解读：



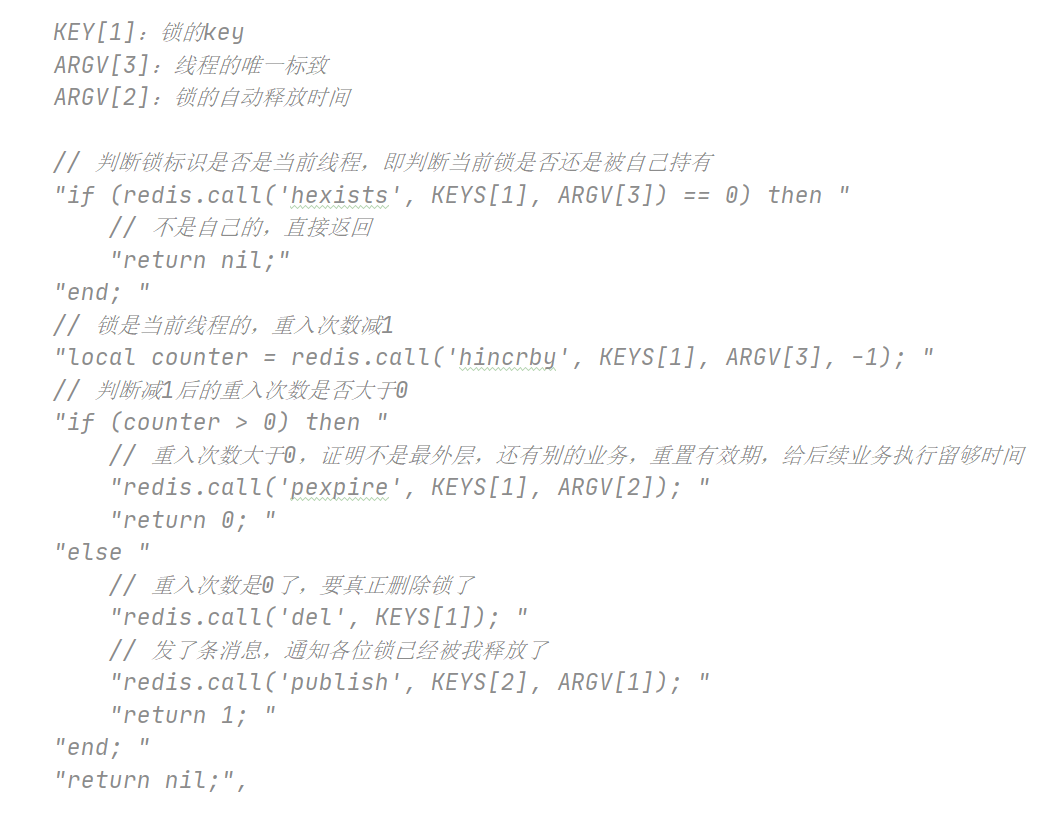
### 释放锁的lua脚本



#### Redisson中释放重入锁的lua脚本（确保了释放锁操作的原子性）：



Lua脚本解读：



### 总结

到这，Redisson可重入锁的原理我们就分析完毕了，获取锁、释放锁整个流程我们也了然了，核心就是利用Hash结构，记录获取锁的线程以及重入次数。与我们JDK提供ReentantLock原理是一致的。