# Redisson的锁重试和WatchDog机制

朋友们，我们已经分析了自定义分布式锁不可重入的原因，并且Redisson是如何实现可重入的，不过解决了可重入问题后，我们自定义的分布式锁依然存在几个问题：

首先就是不可重试的问题，也就是我们现在是获取锁一次，如果获取失败就直接结束，没有重试机制。

第二个就是锁超时释放问题，我们的锁设置了超时释放的时间，如果说因为业务执行时间较长，或者是被阻塞了，导致锁释放，此时业务尚未完成，其他线程却可以拿到锁，就会有线程安全的隐患。

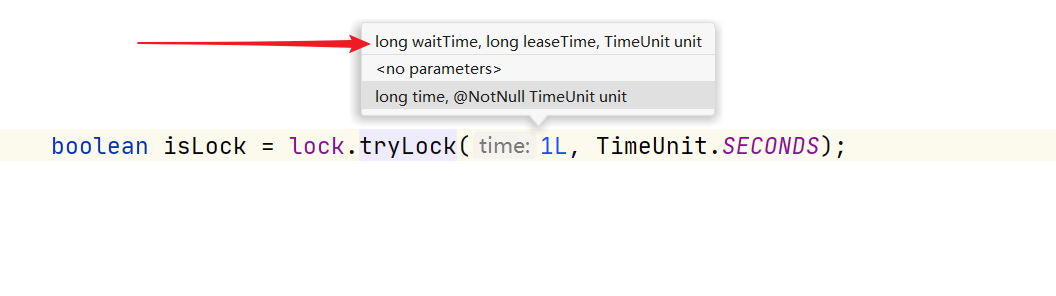
最后就是主从的一致性问题。

这三个问题，Redissson如何解决的呢？现在我们来分析一下下。还是通过跟踪Redisson的源码来弄明白。

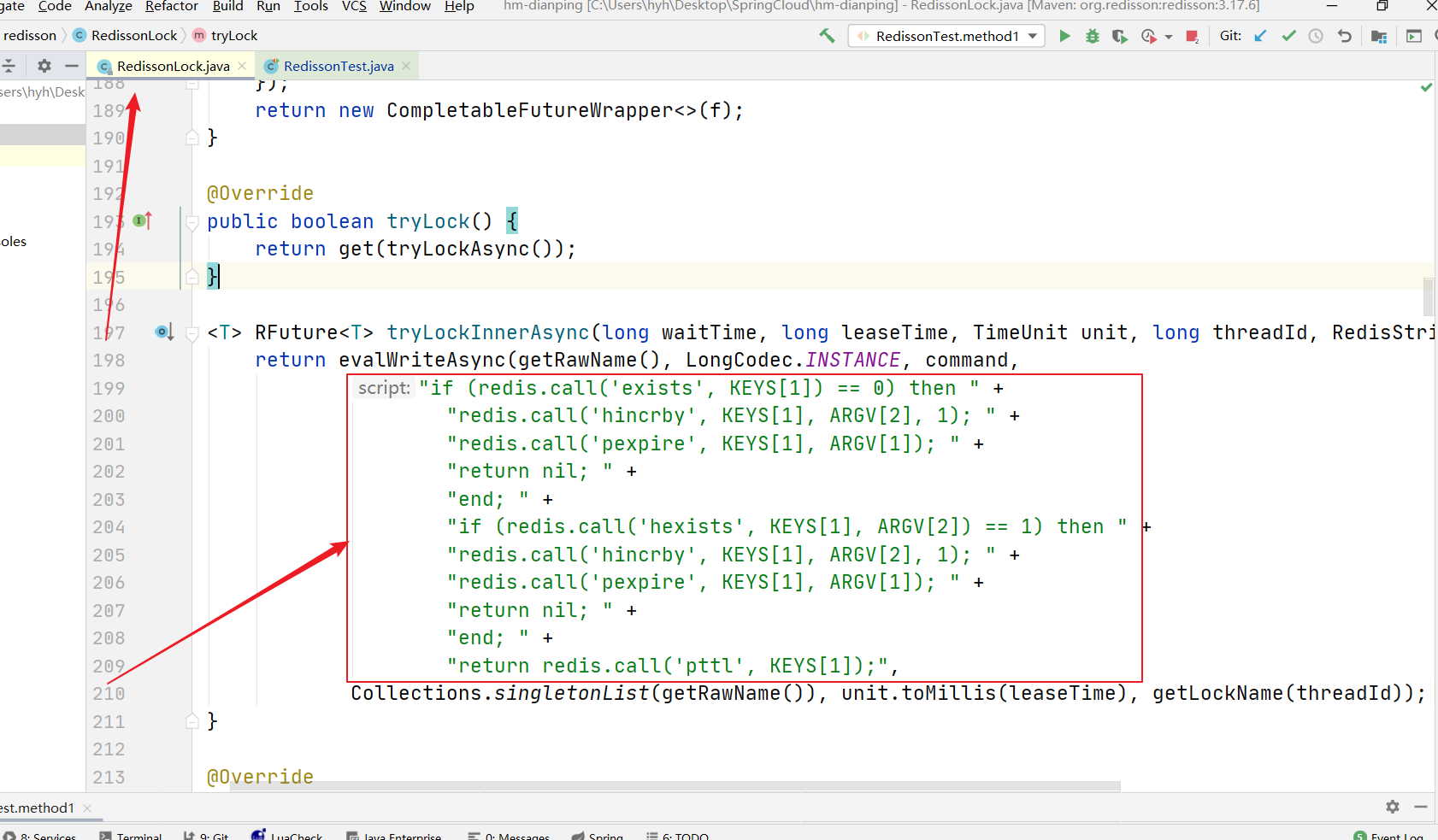
## Redisson实现可重试分布式锁

之前我们我们通过Redisson获取锁的空参函数tryLock()获取到的锁，这个锁是可重入的，如果获取锁是，给方法tryLock()加了等待时间参数：waitTime，一旦加了这个参数，获取到的锁就变成了可重试的锁，就是说再等待时间内，如果没获取到锁，会不断地重试。如果说再等待时间内还是没有获取成功，才会返回失败。

至于第二个参数锁释放时间leaseTime，你不给也没关系，Redisson会给默认的释放时间：30秒。



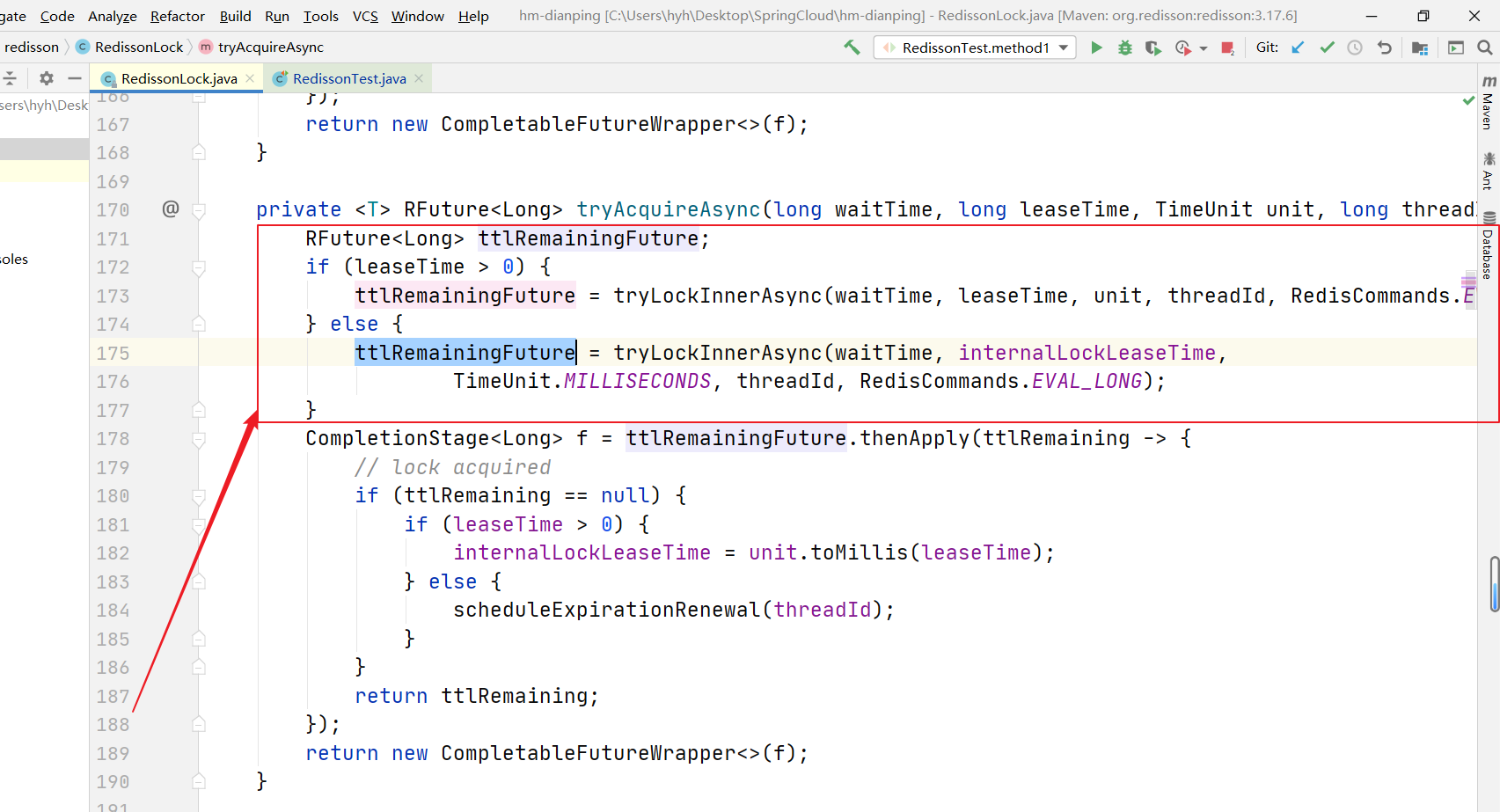
跟踪源码，可以发现获取锁的方法如下图，其实这个方法和之前看的Redisson实现可重入分布式锁是同一个方法，一毛一样！既然一毛一样，那是怎么实现可重试的呢？请看以下解读：



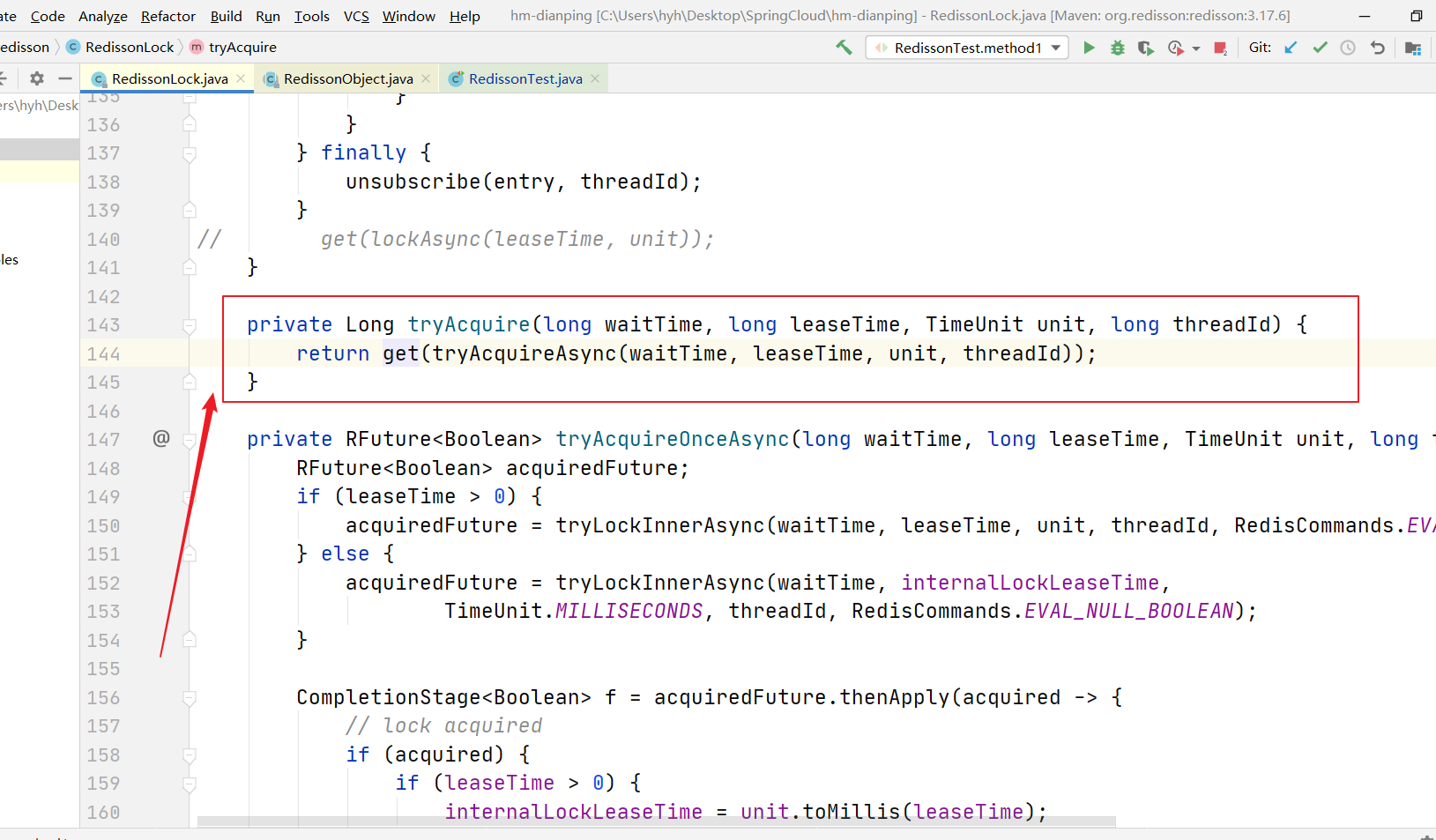
我们可以看到Redisson中获取锁成功返回的是**nil**，这个nil有点像我们Java中的null，也就是空，即Redisson尝试获取锁的lua脚本中，是成功了返回空，失败了却执行命令：**return redis.call('pttl', KEYS[1]);**，返回了结果。

至于失败了返回的是什么结果呢？可以看到是执行了：pttl这个命令，后面跟上的是锁的名称。pttl跟我们所讲的ttl效果是类似的，都是来获取指定key的剩余有效期的，只不过pttl返回的是毫秒为单位的剩余时间，而ttl返回的是以秒为单位的剩余时间。因为我们Redisson代码当中都是以毫秒来做运算的，所以获取锁失败用了pttl来返回的所得剩余时间。

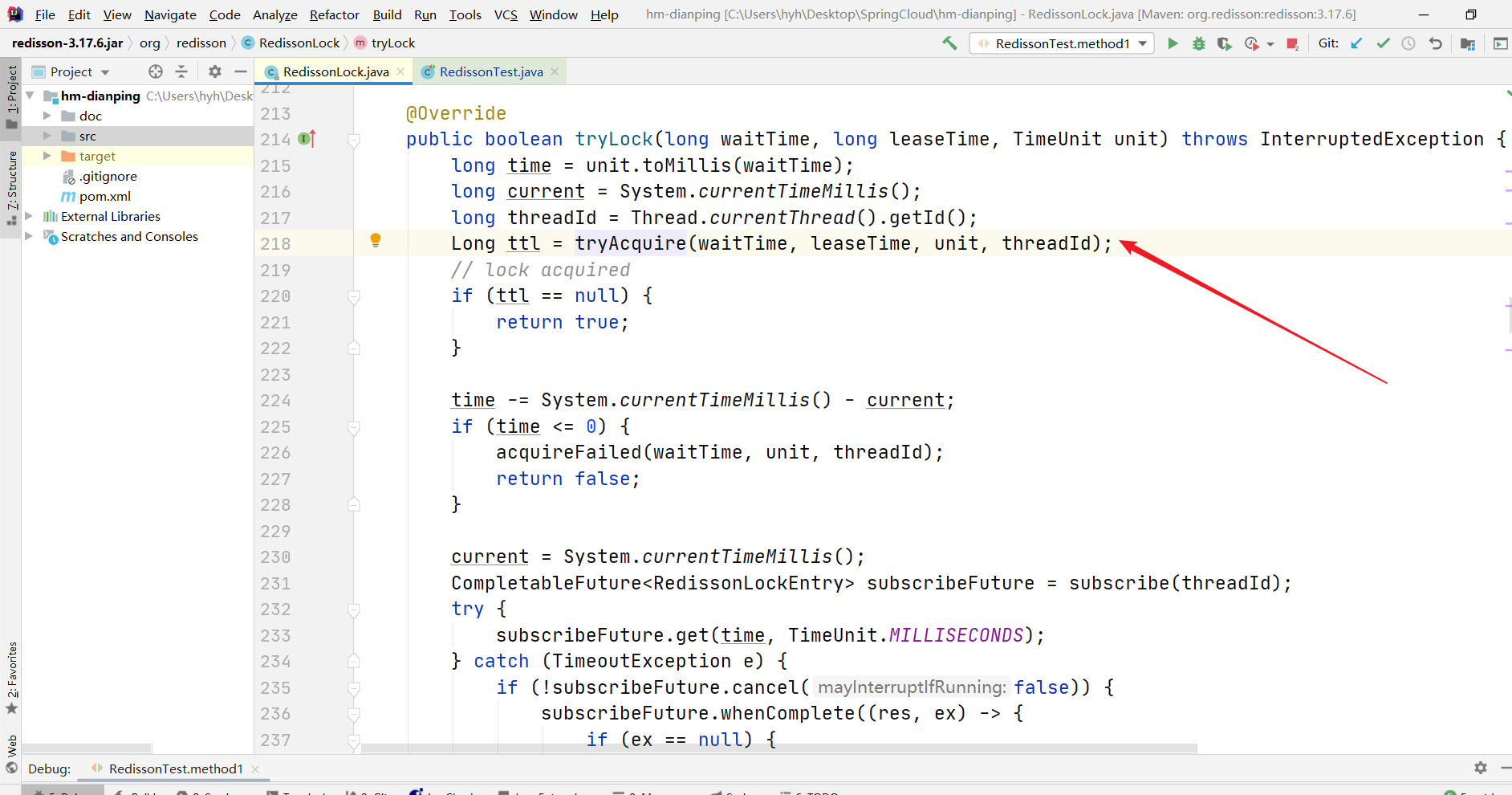
这样我们就明白了，在Redisson整个获取锁的Lua脚本中，获取锁成功返回的是null，获取锁失败返回的是锁的剩余的有效期。那么获取失败拿到锁的剩余有效期有什么用呢？别着急，我们把源码往回倒一下：



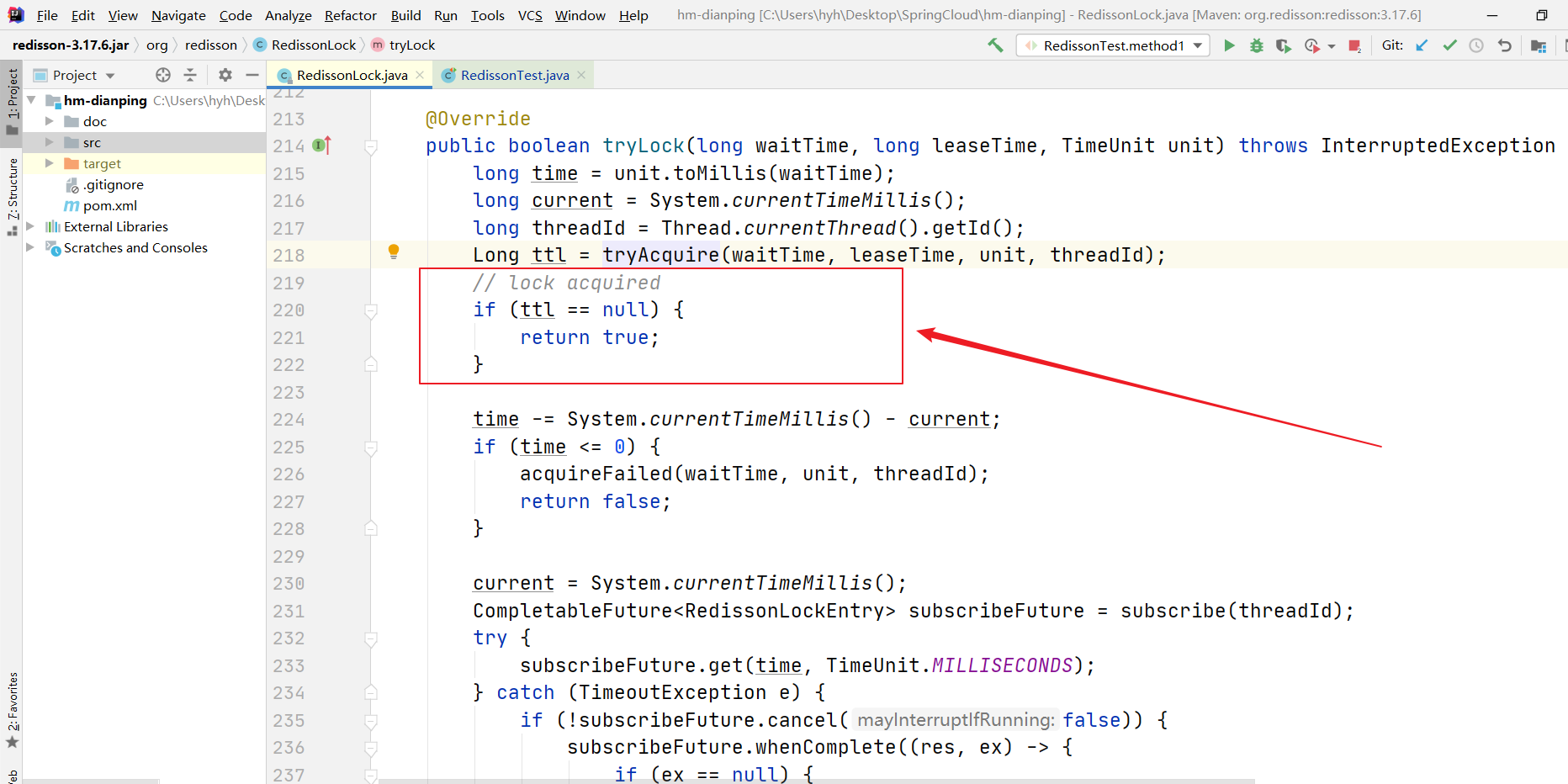
可以看到，Redissoon中我们调用获取锁的方法tryLockInnerAsync()得到的结果，就是锁的剩余有效期，这个结果是封装在Future中，然后在tryAcquireAsync()方法中用Future接收的，为什么要封装在Future中，因为这个tryLockInnerAsync()方法是一个异步方法，也就是说这个tryLockInnerAsync()方法执行完只代表命令发出去了，结果那没拿到还不清楚，所以说结果是一个Future。我们再把源码往上倒一下：



可以看到，tryAcquireAsync()方法是在这个tryAcquire()方法调用的，这个tryAcquire()方法做的就是去获取锁并阻塞等待获取锁的结果并接着返回。继续把源码往上倒，来到我们直接调用的tryLock()的源码：



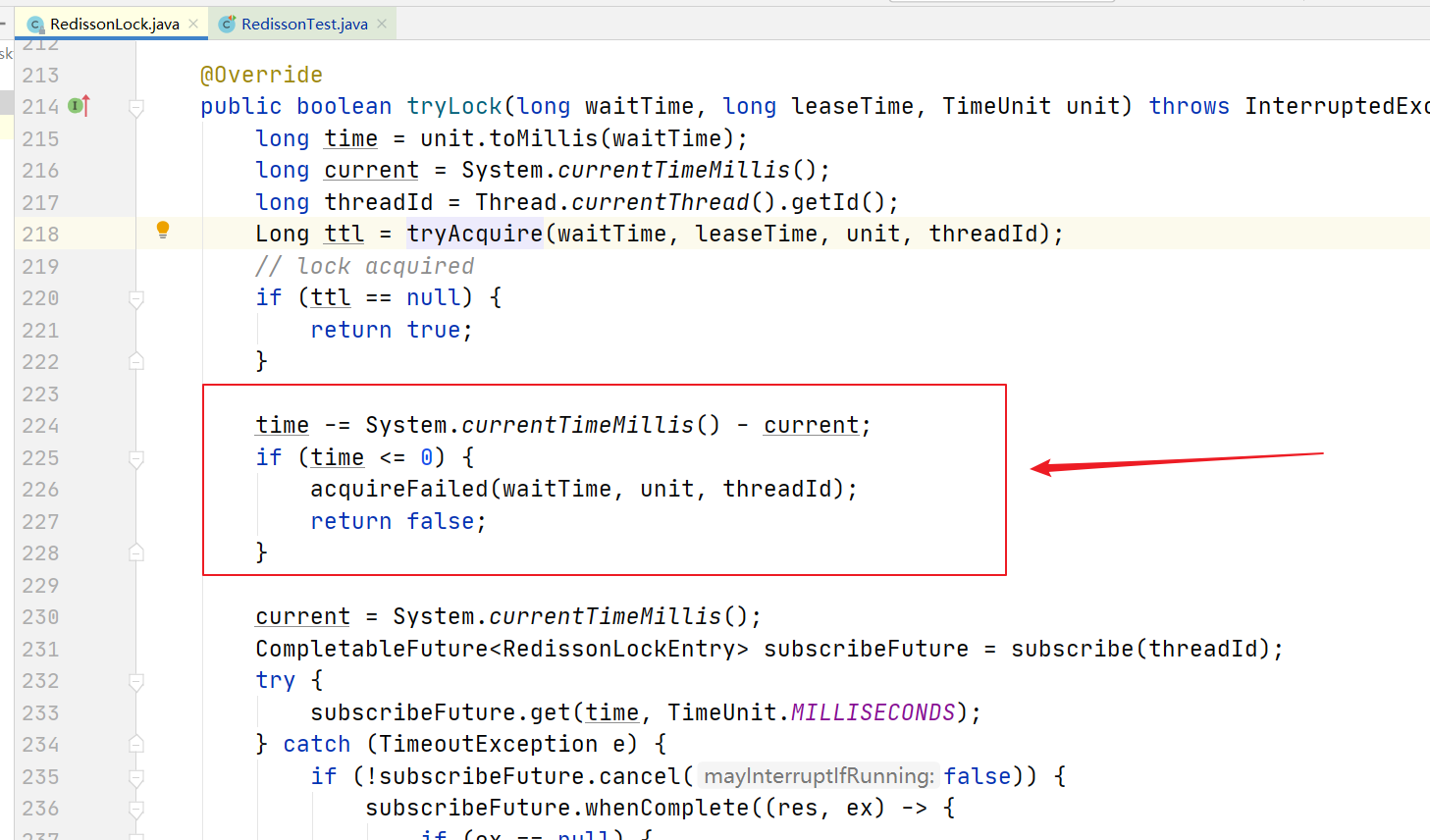
可以看到，回到了最初的起点tryLock()，Redission在tryLock方法内部调用tryAcquire()方法去获取锁时，得到的结果有两种可能性，一是获取锁成功返回的空对象，一种是获取失败返回锁剩余时间。在tryLock()源码中可以看到，当获取到空对象时，返回true，即获取锁成功。



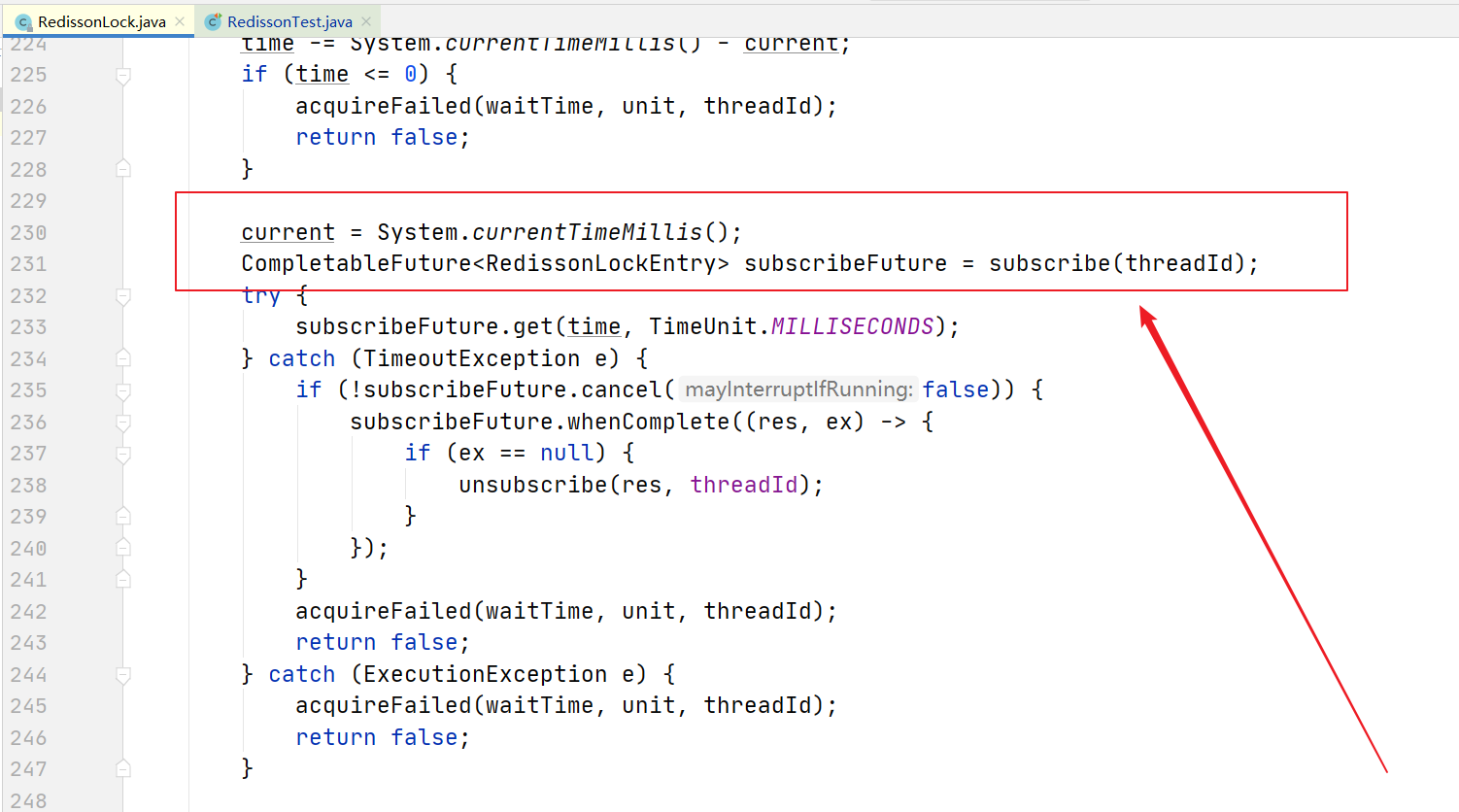
而获取到的时锁剩余时间，即获取失败时，失败就要尝试去再次获取锁了，因为调用tryLock()方法时我们给了waitTime参数嘛！我们要获取的是可重试的分布式锁！怎么去再次获取锁的？

可以看到，获取锁失败后，我们会先会计算出获取锁的耗时，然后再用最大等待时间awaitTime减去获取锁的耗时，得到剩余等待时间time，事实上也是重置等待时间awaitTime。

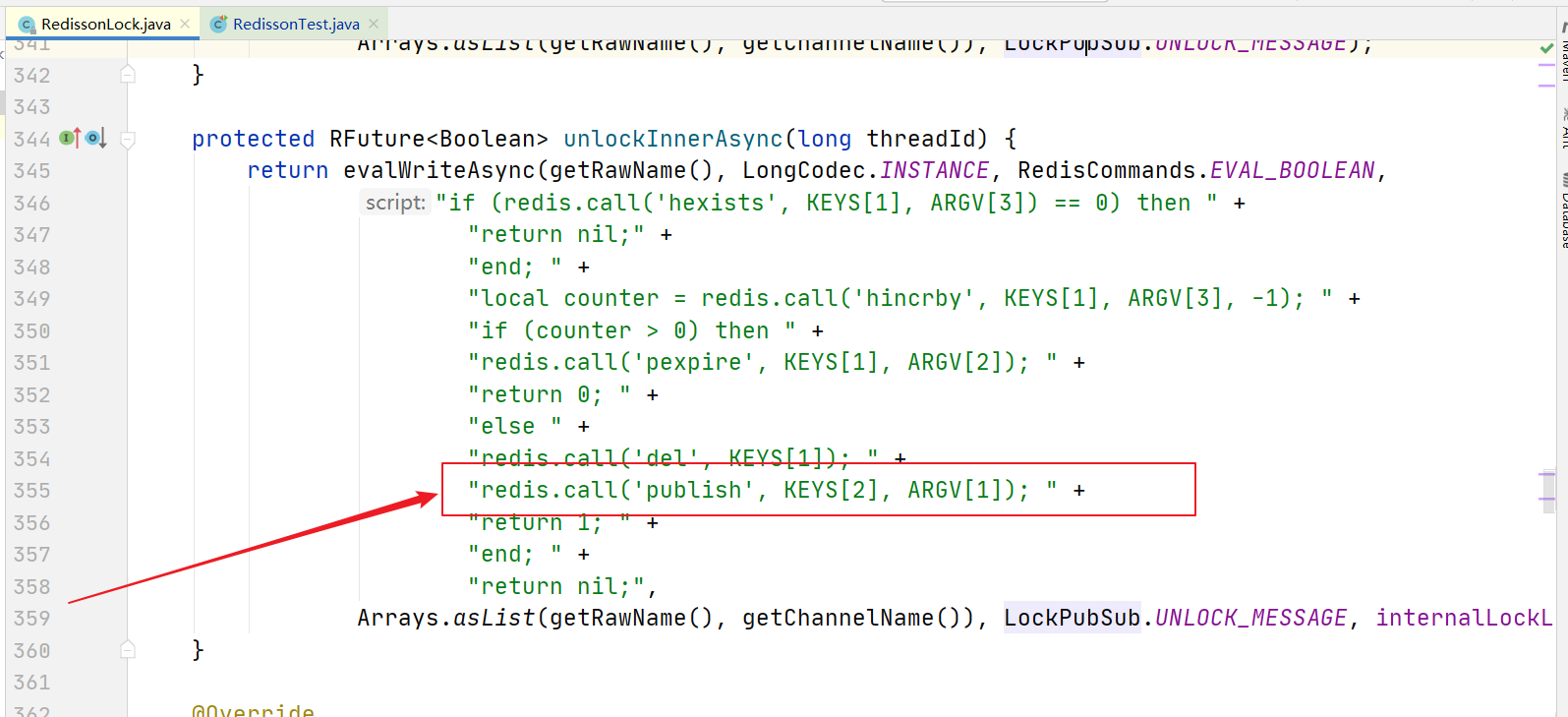
接下来就是剩余等待时间time是否小于0，就是说如果获取锁消耗的时间太长，以至于超过我们设置的获取锁的等待时间，那我们就返回false，代表获取锁失败了，即在等待时间内都没有成功获取锁！



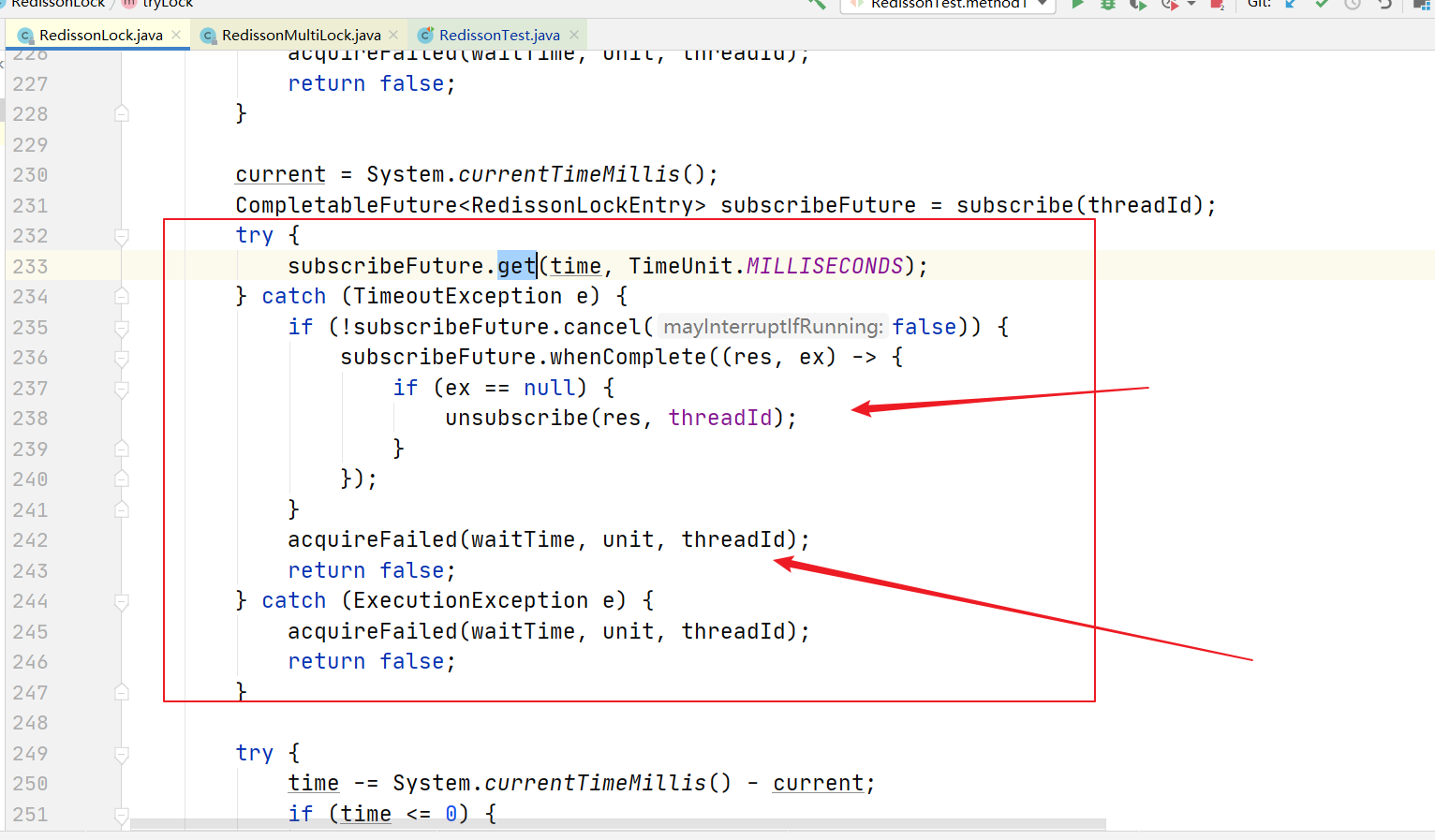
如果剩余等待时间大于0，那么就可以去重新尝试获取锁了。于是就再次拿到当前时间，然后开始去尝试获取锁，但是这个尝试不是说我立即就去尝试获取，你想想看，我上边刚刚获取锁失败，然后我立即尝试，这个时候肯定大概率也是失败的，因为并发场景下哪个获取锁成功的人他肯定还是执行自己的业务还没释放锁呢，你现在立刻就去尝试获取有什么用呢，徒劳无功罢了，除了增加CPU的负担没有任何意义！所以说这里不能立即尝试，而是去订阅别人释放锁的通知信号。也就是说将来如果真的有人释放锁，那么应该发一个通知信息，告诉别人锁已经被释放了，然后别的人收到通知就知道自己可以重新获取这把锁了。当然了，收到通知的具体时间是不确定的哦，所以订阅方法subscribe()的结果也是用Future接收。



为了验证我们说的线程释放锁时会发送通知消息这个结论，可以看一下释放锁的代码，之前我们讲释放锁的Lua脚本的时候，不是看到发送通知的命令吗！请看下图，publish命令就是在发送一条通知消息！欸，你看这不就对应上了吗！



于是，订阅锁释放通知之后，就会去等待锁释放的通知，当然等待不是无限制的等待，最多等剩余等待时间。如果在剩余等待时间内没有收到锁释放的通知。那么就没必要等了，那么就返回获取锁失败的结果，，当然在返回失败结果前，要取消锁释放通知消息的订阅，我获取锁都已经超时了，超过等待时间了，我还订阅释放通知干嘛，没意义了。



相反，如果在剩余等待时间内收到锁释放的通知，那么就去重新尝试获取锁！但还要再次去判断“剩余等待时间”是否大于“从订阅到收到锁释放通知的时间”，当小于时，那么也是向调用者返回获取锁失败结果。当大于时我们终于可以去重新获取锁了，开始while循环，调用tryAcquire方法去重试获取锁。可以看到相当严谨，剩余等待时间要一直精确地去运算。



可以看到，while循环方法里面和上面代码获取锁的流程几乎一样样的，重试获取锁时，通过订阅锁释放的通知和信号量，避免了无效的重试与CPU的占用。

