# Redisson解决锁超时释放-WatchDog机制

我们知道了Redison的锁为什么可重入和可重试的原理之后，还有一个问题要解决，就是锁超时释放问题。

我现在获取锁是成功了，但是我们的锁有个超时释放时间，如果获取锁执行业务时，业务阻塞了，当业务没执行完，锁就因为超时时间TTL而杯Redis释放了，业务还没执行完呢，这时后其他线程就能去获取锁，这个时候就会出现线程安全问题。那Redisson怎么解决业务没执行完锁超时释放问题？就是我们必须确保所是因为我们业务执行完而释放的！不能是因为阻塞了而释放！我们继续需要通过跟踪Redisson源码才能清除！

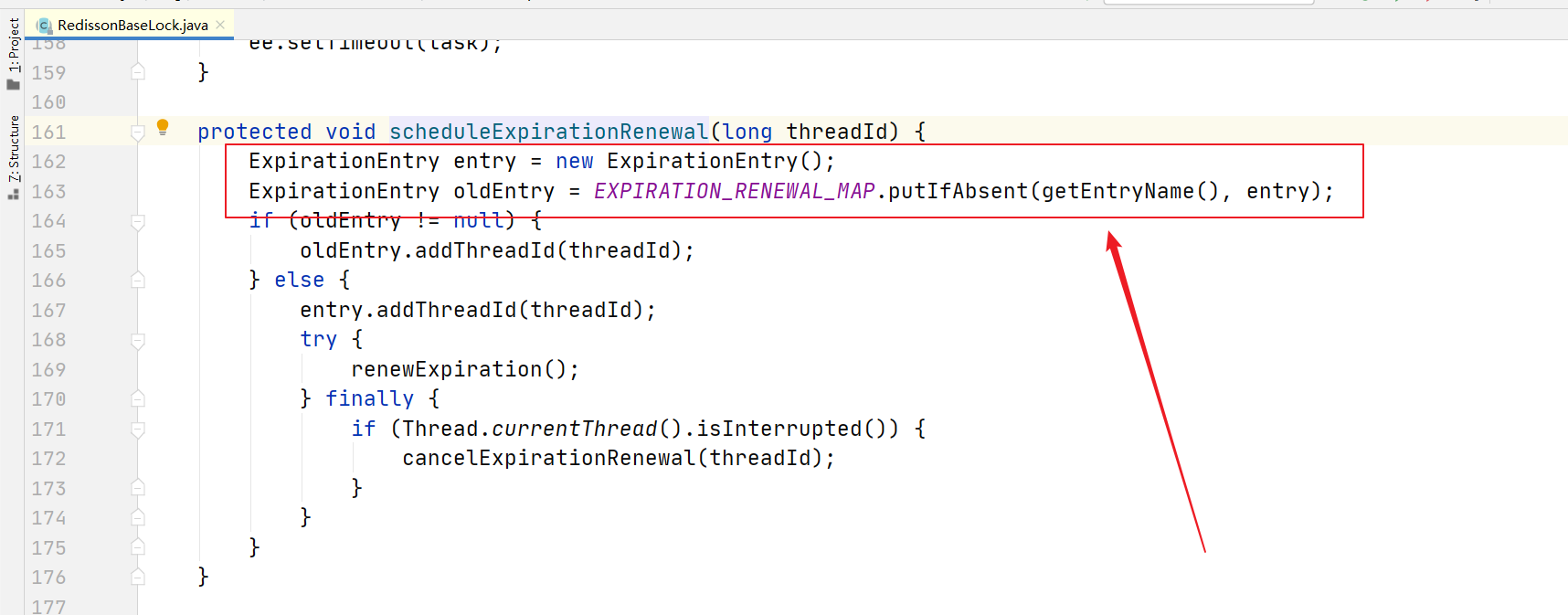
请看在tryAcquireAsync()方法中，我们通过tryLockInnerAsync()方法获取锁，通过Future对象ttlRemainingFuture接收获取锁的结果！之前可重试原理时下面红框部分没看 ，红框部分就是Redisson解决锁超时释放的原理！

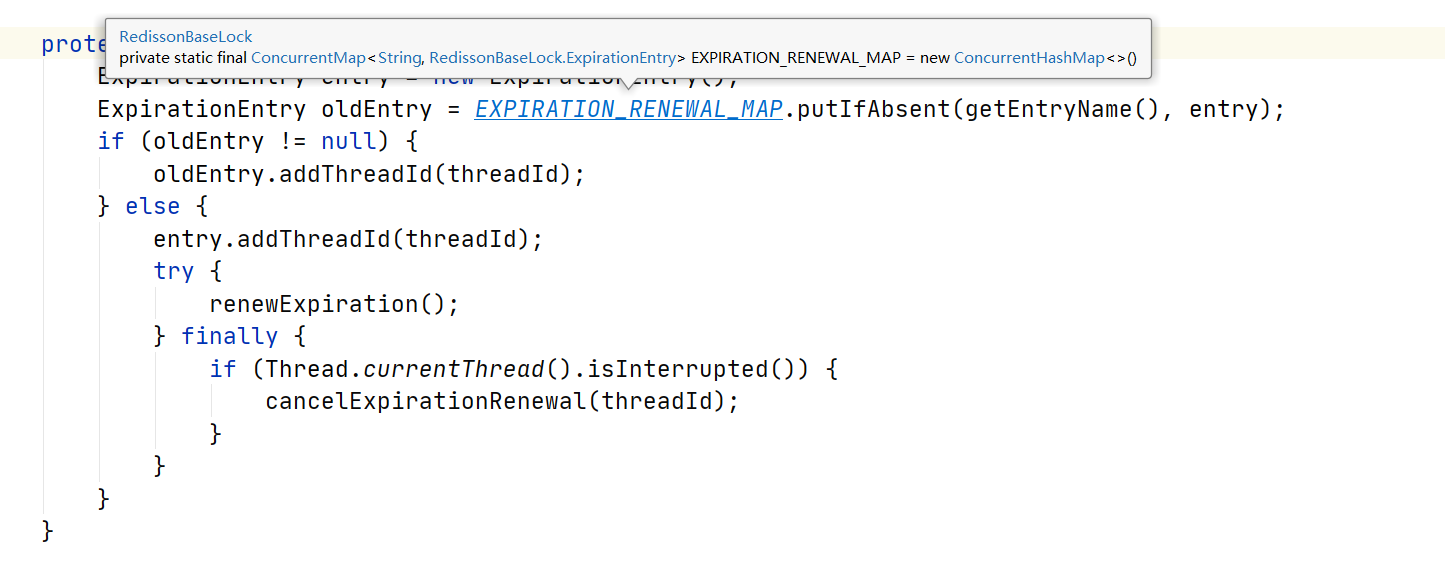


可以看到，在ttlRemainingFuture对象的回调方法thenApply()中，会去判断获取锁tryLockInnerAsync()方法返回的剩余有效期，当剩余有效期ttlRemaining是null时，也就是说获取锁成功。如果获取锁成功了，就要去解决锁有效期的问题，避免业务还么执行完锁就超时释放了，怎么解决的呢？通过调用方法scheduleExpirationRenewal()方法来解决业务还未执行完锁超时释放问题。

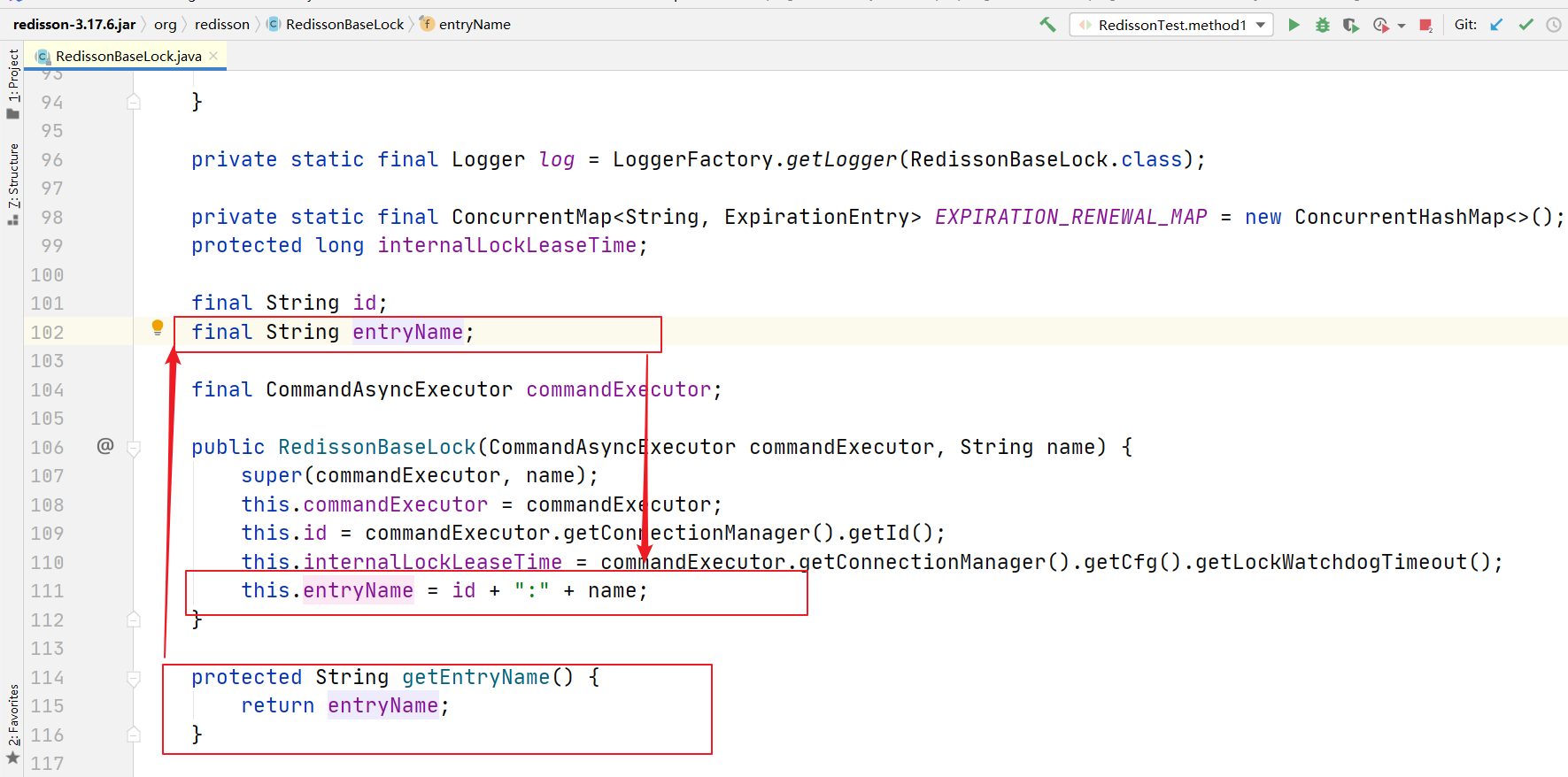
即在获取锁成功后，调用scheduleExpirationRenewal(threadId)自动更新续期的方法，确保线程获取锁成功之后，锁不会因为超时而释放，从而引发线程安全问题。

这个scheduleExpirationRenewal()方法是怎么实现锁有效期自动更新续期的呢？继续进入源码看一下下：

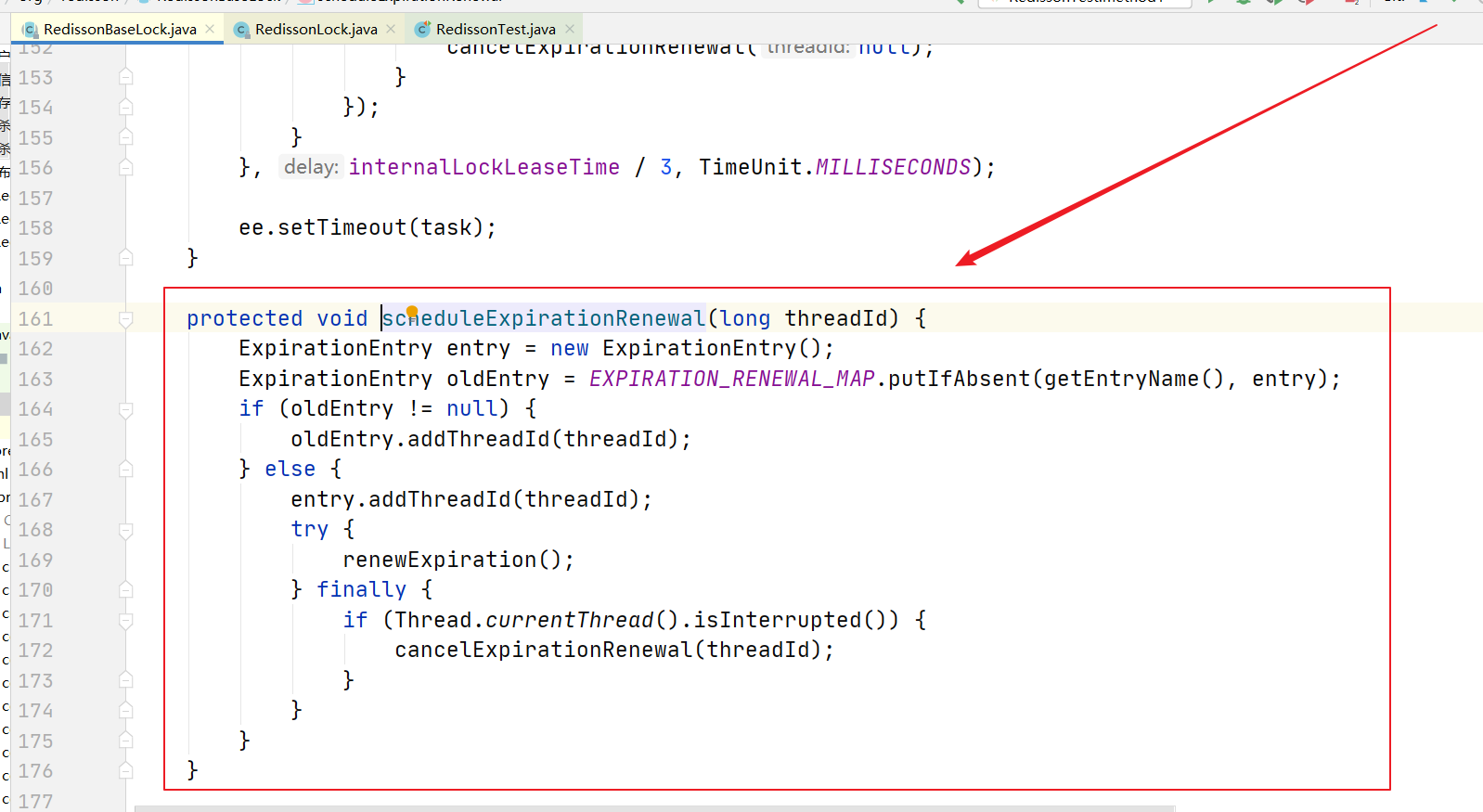


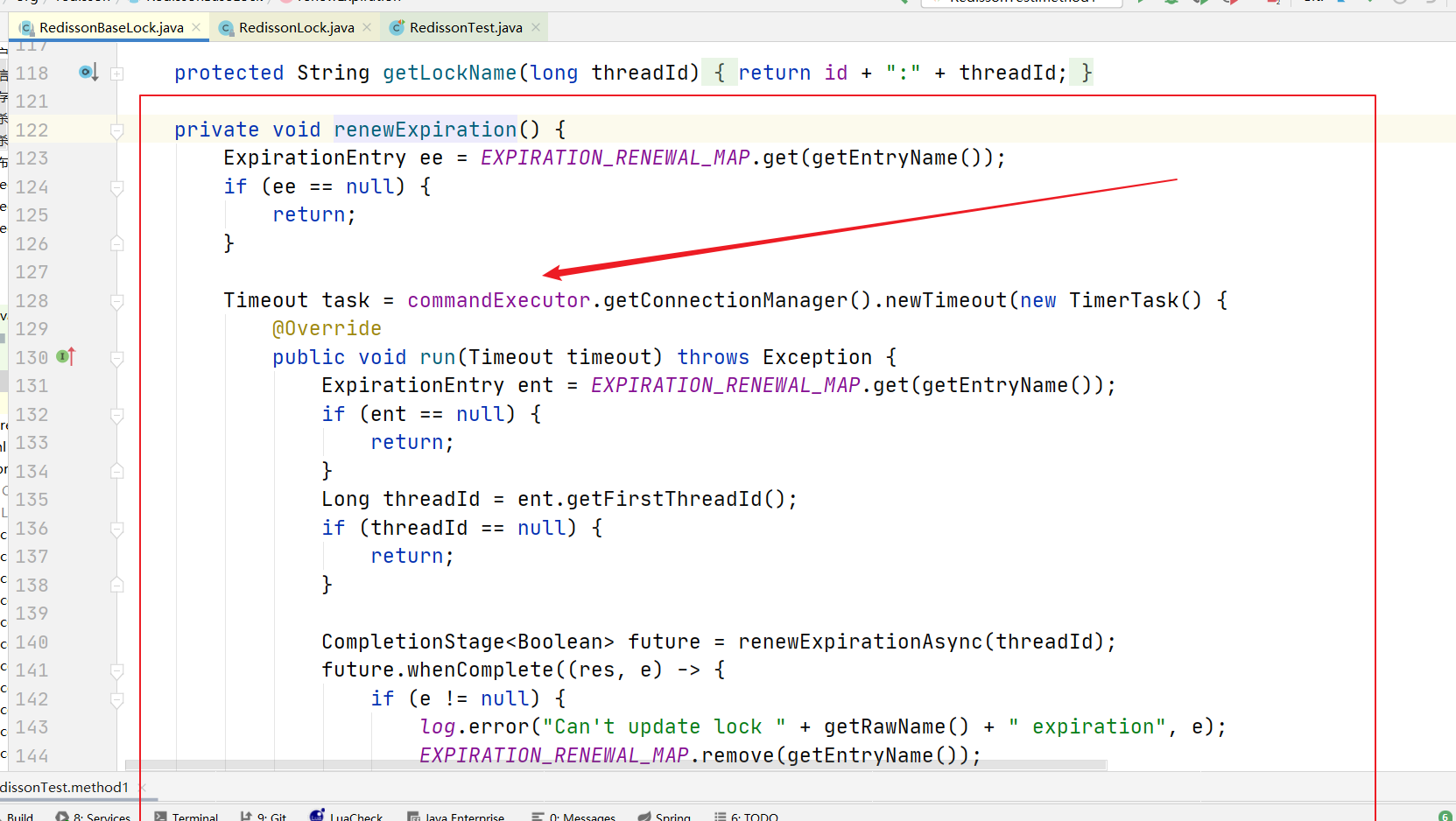


在scheduleExpirationRenewal方法内部，首先可以看到先创建了ExpirationEntry对象entry，然后把entry放到了一个Map里，这个Map是一个ConcurrentHashMap，而且是static final的，这个map的key是字符串，值就是创建的entry，那么key是谁呢？继续跟进源码里去看。可以看到，这个key居然成员变量entryName，这个entryName由两部分组成：id和name，id是当前连接的id，name就是锁的名称，也就是我们在创建Redisscon客户端时传的参数，比如：redissonClient.getLock("order")，那也就是说这个entryName可以理解成当前锁的名称。总而言之，就是一个锁对应一个entry。



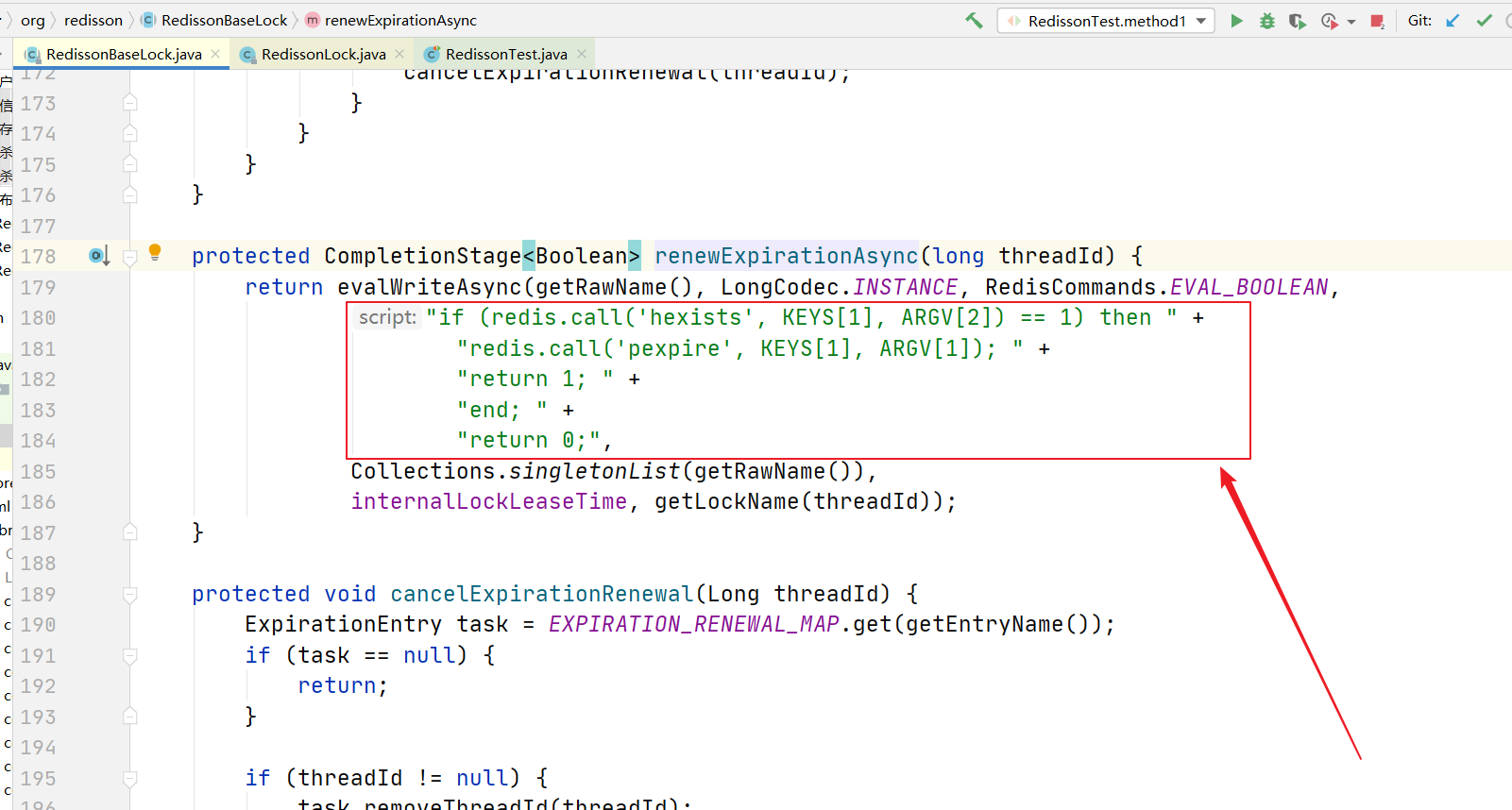
具体看看更新锁释放时间的方法scheduleExpirationRenewal()，最核心的更新锁释放时间的方法是：renewExpiration()，那么我们继续进入这个方法renewExpiration()里面一探究竟，如下图，可以看到这个方法的源码里面有一个延时任务，这个延时任务会在锁释放时间的三分之一后去执行，执行的就是重置锁释放时间的操作，这个定时任务是怎么做的呢？其实重置锁释放时间是通过renewExpirationAsync()方法去完成的。



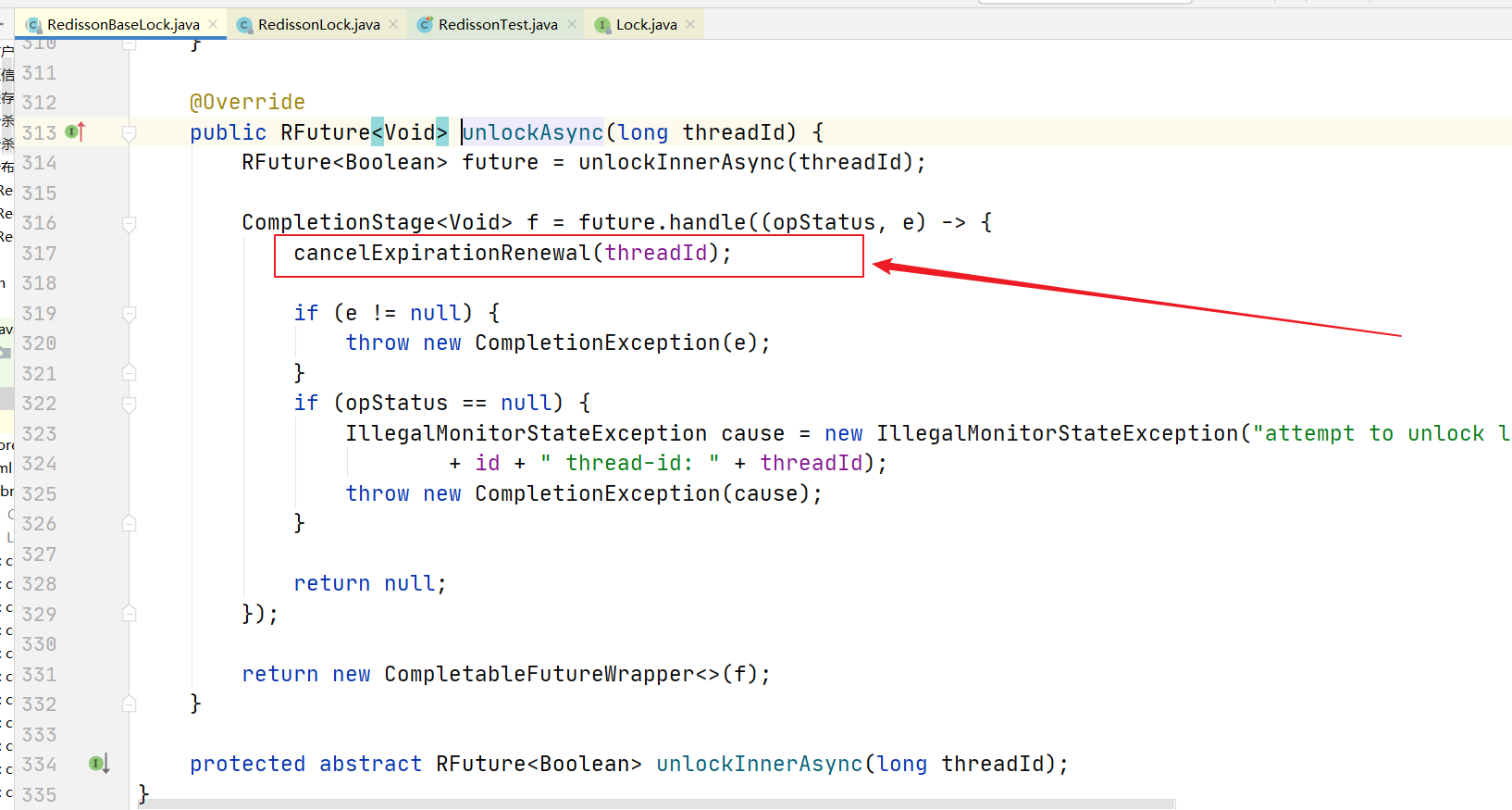




接下来进入重置锁释放时间是通过renewExpirationAsync()方法，看看到底是如何重置锁释放时间的，可以看到，其实是执行了一段Lua脚本，这段脚本去重置了锁的释放时间，于是锁就满血复活了有没有！到此，Redisson实现锁的超时释放的原理就一目了然了，核心步骤就是在获取锁成功之后，会创建一个定时任务，这个定时任务每过三分之一的锁释放时间，就会去重新设置锁的超时释放时间。于是就能确保锁是因为业务执行完（即调用lock.unlock()）才释放，而不会因为阻塞时间超时释放！



那么这个定时任务是么时候会结束呢？这不废话吗，肯定是在释放锁的时候啊，也就是业务执行完成之后呢！见下图，释放锁的方法里面通过方法cancelExpirationRenewal()执行了取消定时任务。



## Redisson解决避免锁超时释放的原理就是：

在线程获取锁成功之后，有一个方法开启一个延时任务，执行延时任务的延时时间是锁释放时间的三分之一，这个延时任务会去重新设置锁的超时释放时间，即线程通过Redisson获取的锁，如果线程在三分之一的超时释放时间内还没有执行完业务，那么Redisson的延时任务就会去重置锁的超时释放时间，确保不会出现业务还没执行完，锁就超时释放了。这个延时任务是循环的，只要业务没执行完，就会每隔锁释放时间的三分之一去执行延时任务重置锁的超时释放时间。

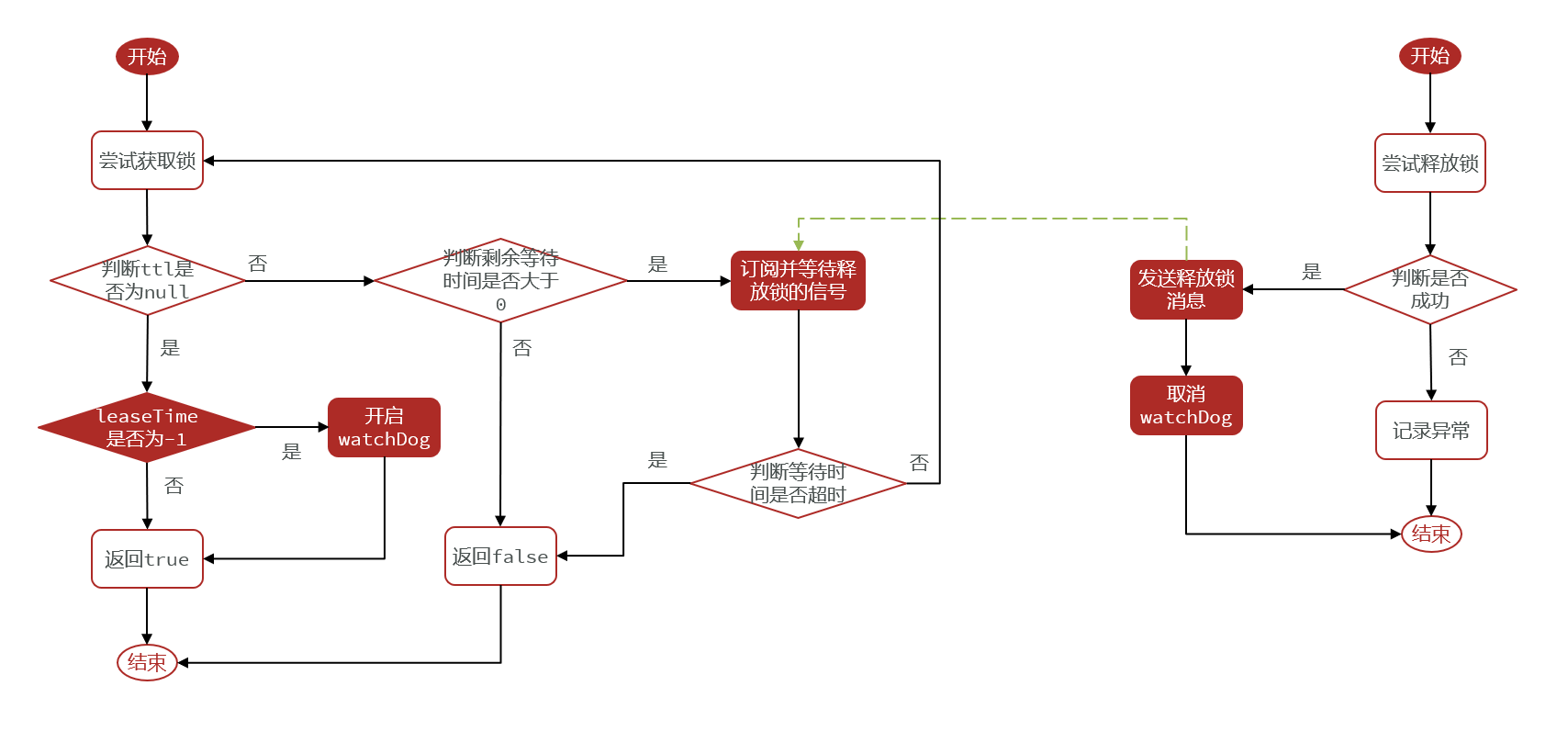
所释放的时候，会发出通知，重新设置锁超时时间的延时任务就会取消。

为什么要延时三分之一再去执行延时任务重置锁释放时间，废话，锁刚刚获取呢，刚刚开始执行业务你就去重置锁释放时间，没有一点点道理，除了增加CPU负担之外没有任务意义！

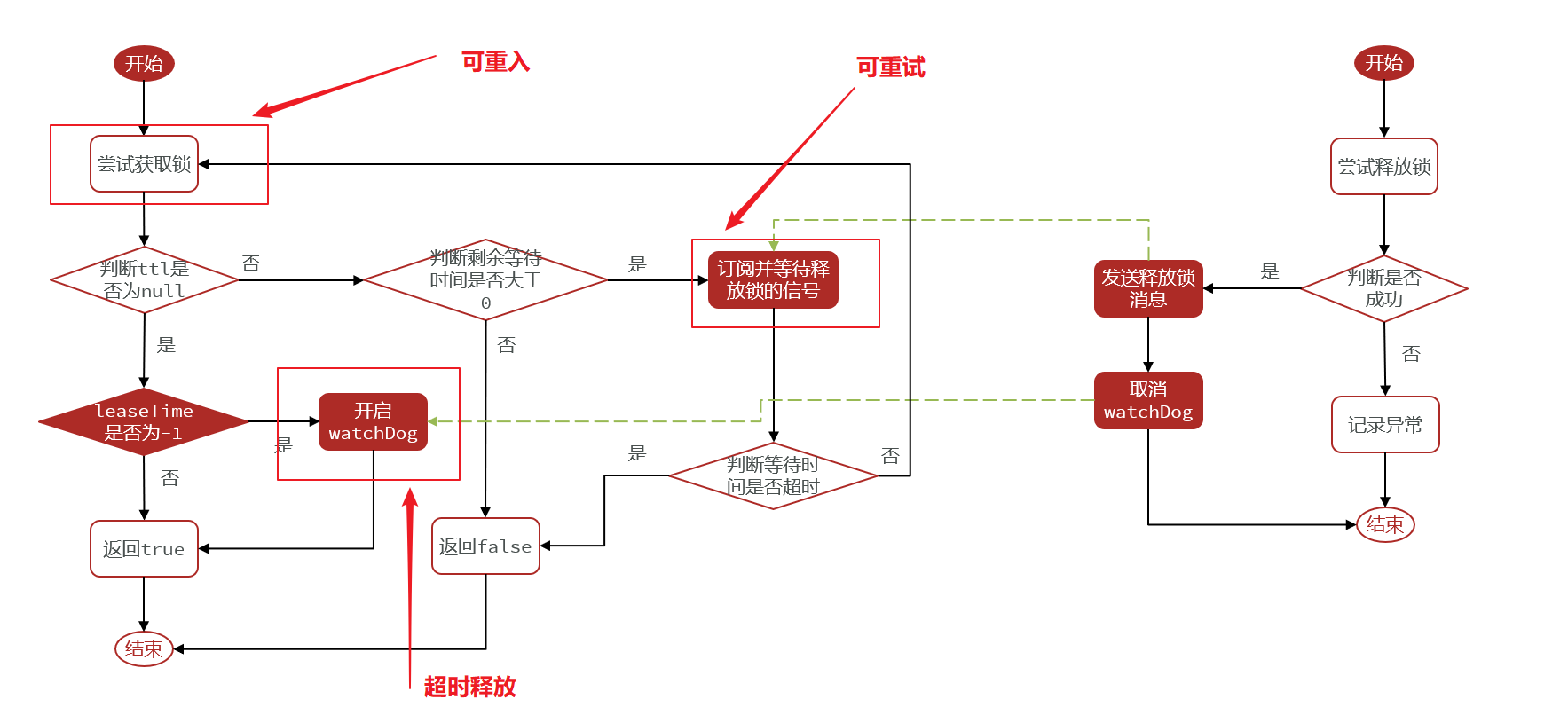
renewExpiration()方法里开启一个延时（锁释放时间  / 3）任务去重置锁的超时释放时间，然后继续递归调用renewExpiration()方法去重置锁的超时释放时间！

释放锁时，会调用 cancelExpirationRenewal(threadId)方法取消自动续期！

## 于是，通过Redisson获取的锁就能同时满足可重入、可重试、避免超时释放这三个特定了。



看下面流程图，实现可重入、可重试、超时释放的部分：



## Redisson分布式原理：

### 可重入：利用Hash结构记录线程id和重入次数

代理原来自定义分布式锁的String结构，Hash能记录两个东西：线程标识和重入次数，每一次获取锁的时候先判断锁是否存在，如果不存在就直接获取，如果已经存在不代表获取失败了，而是判断锁的线程标识是否是当前线程，如果是那么可以再次获取，只需要把重入次数加1就行了，将来释放锁的时候，没释放一次重入次数就减1，知道重入次数减为0，证明业务走到最外层，所有业务都结束了，再真正地释放锁。于是实现了锁的可重入。

### 可重试：利用信号量和Publish功能实现等待、唤醒，获取锁失败的重试机制

也就是说在第一次尝试获取锁失败以后，不是立即失败，而是去做一个等待，等待释放锁的消息，利用了Redis中Pubilsh的机制，而获取锁成功的线程在释放锁的时候就会去发送锁释放消息，从而被我们捕获到，当我们得到锁释放消息后，我们就可以重新获取锁了，如果获取锁再次失败，我们还可以继续等待释放锁的消息，然后再次去重试，当然这个重试不是无限次去重试，会有一个等待时间，如果说超过等待时间就不再重试了。因为重试机制采用了等待唤醒的方案，所以说并不会过多地去占用CPU，整个效率是非常不错的。

### 超时续约：利用WatchDog，每隔一段时间（releaseTime / 3），重置锁超时时间

简单来讲就是在获取锁成功之后，开启一个定时任务，这个任务每隔一段时间就会去重置锁超时时间，也就是重置配一下expire，这样一来，我们的锁的重试时间就会重新计时，满血复活，这就是所谓的看门狗机制。