
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芮芮这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-10-05
Redis

精尽 Redisson 源码分析 —— 可靠分布式锁 RedLock

1. 概述

在 《精尽 Redisson 源码分析 —— 可重入分布式锁 ReentrantLock》 中，芮芮臭长臭长的分享了 Redisson 是如何实现可重入的 ReentrantLock 锁，一切看起来很完美，我们能够正确的加锁，也能正确的释放锁。但是，我们来看一个 Redis 主从结构下的示例，Redis 分布式锁是如何失效的：

1、客户端 A 从 Redis Master 获得到锁 anylock 。
2、在 Redis Master 同步 anylock 到 Redis Slave 之前，Master 挂了。
3、Redis Slave 晋升为新的 Redis Master 。
4、客户端 B 从新的 Redis Master 获得到锁 anylock 。

此时，客户端 A 和 B 同时持有 anylock 锁，已经失效。当然，这个情况是极小概率事件。主要看胖友业务对分布式锁可靠性的诉求。

在 Redis 分布式锁存在失效的问题，Redis 的作者 Antirez 大神提出了红锁 RedLock 的想法。我们来看看它的描述。

> FROM 《Redis 文档 —— Redis 分布式锁》
>
> 在 Redis 的分布式环境中，我们假设有 N 个 Redis master 。这些节点完全互相独立，不存在主从复制或者其他集群协调机制。之前我们已经描述了在 Redis 单实例下怎么安全地获取和释放锁。我们确保将在每 N 个实例上使用此方法获取和释放锁。在这个样例中，我们假设有 5 个Redis master 节点，这是一个比较合理的设置，所以我们需要在 5 台机器上面或者 5 台虚拟机上面运行这些实例，这样保证他们不会同时都宕掉。
>
> 为了取到锁，客户端应该执行以下操作:
>
> 1、获取当前 Unix 时间，以毫秒为单位。
> 2、依次尝试从 N 个实例，使用相同的 key 和随机值获取锁。在步骤 2 ，当向 Redis 设置锁时，客户端应该设置一个网络连接和响应超时时间，这个超时时间应该小于锁的失效时间。例如你的锁自动失效时间为 10 秒，则超时时间应该在 5-50 毫秒之间。这样可以避免服务器端 Redis 已经挂掉的情况下，客户端还在死死地等待响应结果。如果服务器端没有在规定时间内响应，客户端应该尽快尝试另外

一个 Redis 实例。
3、客户端使用当前时间减去开始获取锁时间（步骤 1 记录的时间）就得到获取锁使用的时间。当且仅当从大多数（这里是 3 个节点）的 Redis 节点都取到锁，并且使用的时间小于锁失效时间时，锁才算获取成功。
4、如果取到了锁，key 的真正有效时间等于有效时间减去获取锁所使用的时间（步骤 3 计算的结果）。
5、如果因为某些原因，获取锁失败（没有在至少 N/2 + 1 个 Redis 实例取到锁或者取锁时间已经超过了有效时间），客户端应该在所有的 Redis 实例上进行解锁（即便某些Redis实例根本就没有加锁成功）。

释放锁：

1、释放锁比较简单，向所有的 Redis 实例发送释放锁命令即可，不用关心之前有没有从Redis实例成功获取到锁.

可能一看内容这么长，略微有点懵逼。重点理解，需要至少在 N/2 + 1 Redis 节点获得锁成功。这样，即使出现某个 Redis Master 未同步锁信息到 Redis Slave 节点之前，突然挂了，也不容易出现多个客户端获得相同锁，因为需要至少在 N/2 + 1 Redis 节点获得锁成功。。

当然，极端情况下也有。我们以 3 个 Redis Master 节点举例子：

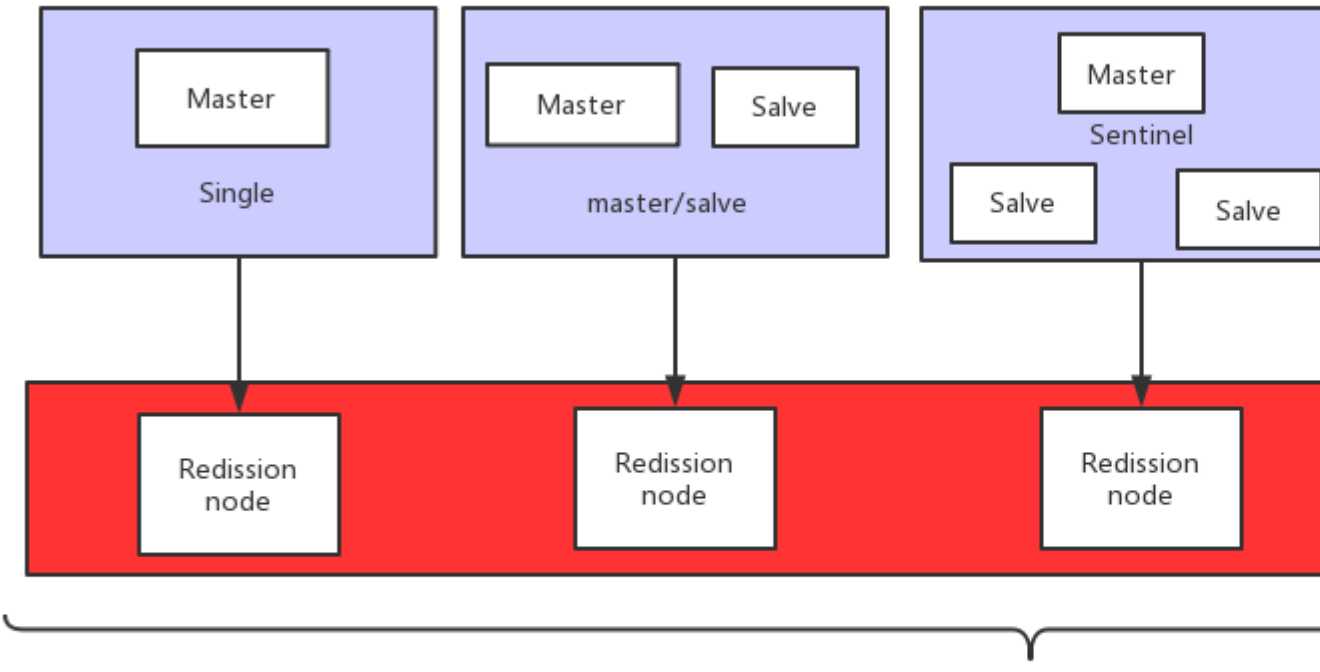
1、客户端 A 从 3 个 Redis Master 获得到锁 anylock 。
2、2 个 Redis Master 同步 anylock 到 Redis Slave 之前，Master 都挂了。
3、2 个 Redis Slave 晋升为新的 Redis Master 。
4、客户端 B 从新的 Redis Master 获得到锁 anylock 。

理论来说，出现 2 个 Redis Master 都挂了，并且数据都未同步到 Redis Slave 的情况，已经是小概率的事件。当然，哈哈哈哈，我们就是可爱的“杠精”，就是要扣一扣这个边界情况。

同时，我们也可以发现，在时候用 RedLock 的时候，Redis Master 越多，集群的可靠性就越高，性能也会越低。　架构设计中，从来没有银弹。我们想要得到更高的可靠性，往往需要失去一定的性能。

对了，有一点要注意，N 个 Redis Master 要毫无关联的。例如说，任一一个 Redis Master 都不能在同一个 Redis Cluster 中。再如下图，就是一个符合条件的：

FROM 《慢谈 Redis 实现分布式锁 以及 Redisson 源码解析》



当然，推荐 N 是奇数个，因为 N / 2 + 1 嘛，哈哈。

推荐胖友阅读如下三篇文章，更进一步了解 RedLock ：

《Redis 文档 —— Redis 分布式锁》
《How to do distributed locking》
《Is Redlock safe?》 简直神仙打架，强烈推荐。

2. RedissonRedLock

org.redisson.RedissonRedLock ，继承自联锁 RedissonMultiLock ，Redisson 对 RedLock 的实现类。代码如下：

```
// RedissonRedLock.java

public class RedissonRedLock extends RedissonMultiLock {

    /**
     * Creates instance with multiple {@link RLock} objects.
     * Each RLock object could be created by own Redisson instance.
     *
     * @param locks - array of locks
     */
    public RedissonRedLock(RLock... locks) {
        super(locks);
    }

    @Override
    protected int failedLocksLimit() {
        return locks.size() - minLocksAmount(locks);
    }

    protected int minLocksAmount(final List<RLock> locks) {
        return locks.size()/2 + 1;
    }

    @Override
    protected long calcLockWaitTime(long remainTime) {
        return Math.max(remainTime / locks.size(), 1);
    }

    @Override
    public void unlock() {
        unlockInner(locks);
    }

}
```

RedissonMultiLock ，联锁，正如其名字“联”，可以将多个 RLock 锁关联成一个联锁。使用示例如下：

```
RedissonMultiLock lock = new RedissonMultiLock(lock1, lock2, lock3);
// 给lock1，lock2，lock3加锁，如果没有手动解开的话，10秒钟后将会自动解开
lock.lock(10, TimeUnit.SECONDS);
```

```
// 为加锁等待100秒时间，并在加锁成功10秒钟后自动解开
boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
...
lock.unlock();
```

RedissonRedLock ，是一个特殊的联锁，加锁时无需所有的 RLock 都成功，只需要满足 N / 2 + 1 个 RLock 即可。使用示例如下：

```
RLock lock1 = redissonInstance1.getLock("lock1");
RLock lock2 = redissonInstance2.getLock("lock2");
RLock lock3 = redissonInstance3.getLock("lock3");

RedissonRedLock lock = 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。
lock.lock();
...
lock.unlock();
```

RedissonRedLock 构造方法，创建时，传入多个 RLock 对象。一般来说，每个 RLock 对应到一个 Redis Master 节点上。

#failedLocksLimit() 方法，允许失败加锁的数量。从 #minLocksAmount(final List<RLock> locks) 方法上，我们已经看到 N / 2 + 1 的要求。
#calcLockWaitTime(long remainTime) 方法，计算每次获得 RLock 的锁的等待时长。目前的计算规则是，总的等待时间 remainTime 平均分配到每个 RLock 上。
#unlock() 方法，解锁时，需要所有 RLock 都进行解锁。

3. RedissonMultiLock

org.redisson.RedissonMultiLock ，实现 RLock 接口，Redisson 对联锁 MultiLock 的实现类。

3.1 构造方法

```
// RedissonMultiLock.java

/**
 * RLock 数组
 */
final List<RLock> locks = new ArrayList<>();

/**
 * Creates instance with multiple {@link RLock} objects.
 * Each RLock object could be created by own Redisson instance.
 *
 * @param locks - array of locks
 */
public RedissonMultiLock(RLock... locks) {
    if (locks.length == 0) {
        throw new IllegalArgumentException("Lock objects are not defined");
    }
```

```
        this.locks.addAll(Arrays.asList(locks));
    }
```

3.2 failedLocksLimit

#failedLocksLimit() 方法，允许失败加锁的数量。代码如下：

```
    // RedissonMultiLock.java

    protected int failedLocksLimit() {
        return 0;
    }
```

默认返回值是 0 ，也就是必须所有 RLock 都加锁成功。

在 RedissonRedLock 中，会重写该方法。

```
    // RedissonRedLock.java

    @Override
    protected int failedLocksLimit() {
        return locks.size() - minLocksAmount(locks);
    }

    protected int minLocksAmount(final List<RLock> locks) {
        return locks.size() / 2 + 1;
    }
```

3.3 calcLockWaitTime

#failedLocksLimit(long remainTime) 方法，计算每次获得 RLock 的锁的等待时长。代码如下：

```
    // RedissonMultiLock.java

    protected long calcLockWaitTime(long remainTime) {
        return remainTime;
    }
```

默认直接返回 remainTime ，也就是说，每次获得 RLock 的锁的等待时长都是 remainTime 。

在 RedissonRedLock 中，会重写该方法。

3.4 tryLock

#tryLock(long waitTime, long leaseTime, TimeUnit unit) 方法，同步加锁，并返回加锁是否成功。代码如下：

```
    // RedissonMultiLock.java
```

```java
@Override
public boolean tryLock() {
    try {
        // 同步获得锁
        return tryLock(-1, -1, null);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return false;
    }
}

@Override
public boolean tryLock(long waitTime, TimeUnit unit) throws InterruptedException {
    return tryLock(waitTime, -1, unit);
}
```

```java
@Override
public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws InterruptedException {
    // 计算新的锁的时长 newLeaseTime
    long newLeaseTime = -1;
    if (leaseTime != -1) {
        if (waitTime == -1) { // 如果无限等待，则直接使用 leaseTime 即可。
            newLeaseTime = unit.toMillis(leaseTime);
        } else { // 如果设置了等待时长，则为等待时间 waitTime * 2 。不知道为什么要 * 2 ？例如说，先获得到了第一个
            newLeaseTime = unit.toMillis(waitTime) * 2;
        }
    }

    long time = System.currentTimeMillis();
    // 计算剩余等待锁的时间 remainTime
    long remainTime = -1;
    if (waitTime != -1) {
        remainTime = unit.toMillis(waitTime);
    }
    // 计算每个锁的等待时间
    long lockWaitTime = calcLockWaitTime(remainTime);

    // 允许获得锁失败的次数
    int failedLocksLimit = failedLocksLimit();
    // 已经获得到锁的数组
    List<RLock> acquiredLocks = new ArrayList<>(locks.size());
    // 遍历 RLock 数组，逐个获得锁
    for (ListIterator<RLock> iterator = locks.listIterator(); iterator.hasNext();) {
        // 当前 RLock
        RLock lock = iterator.next();
        boolean lockAcquired; // 标记是否获得到锁
        try {
            // 如果等待时间 waitTime 为 -1（不限制），并且锁时长为 -1（不限制），则使用 #tryLock() 方法。
            if (waitTime == -1 && leaseTime == -1) {
                lockAcquired = lock.tryLock();
            // 如果任一不为 -1 时，则计算新的等待时间 awaitTime ，然后调用 #tryLock(long waitTime, long leaseTim
            } else {
                long awaitTime = Math.min(lockWaitTime, remainTime);
                lockAcquired = lock.tryLock(awaitTime, newLeaseTime, TimeUnit.MILLISECONDS);
            }
        } catch (RedisResponseTimeoutException e) {
            // 发生响应超时。因为无法确定实际是否获得到锁，所以直接释放当前 RLock
            unlockInner(Collections.singletonList(lock));
            // 标记未获得锁
```

```
44:             lockAcquired = false;
45:         } catch (Exception e) {
46:             // 标记未获得锁
47:             lockAcquired = false;
48:         }
49:
50:         // 如果获得成功，则添加到 acquiredLocks 数组中
51:         if (lockAcquired) {
52:             acquiredLocks.add(lock);
53:         } else {
54:             // 如果已经到达最少需要获得锁的数量，则直接 break 。例如说，RedLock 只需要获得 N / 2 + 1 把。
55:             if (locks.size() - acquiredLocks.size() == failedLocksLimit()) {
56:                 break;
57:             }
58:
59:             // 当已经没有允许失败的数量，则进行相应的处理
60:             if (failedLocksLimit == 0) {
61:                 // 释放所有的锁
62:                 unlockInner(acquiredLocks);
63:                 // 如果未设置阻塞时间，直接返回 false ，表示失败。因为是 tryLock ，只是尝试加锁一次，不会无限重试
64:                 if (waitTime == -1) {
65:                     return false;
66:                 }
67:                 // 重置整个获得锁的过程，在剩余的时间里，重新来一遍
68:                 // 重置 failedLocksLimit 变量
69:                 failedLocksLimit = failedLocksLimit();
70:                 // 重置 acquiredLocks 为空
71:                 acquiredLocks.clear();
72:                 // reset iterator
73:                 // 重置 iterator 设置回迭代器的头
74:                 while (iterator.hasPrevious()) {
75:                     iterator.previous();
76:                 }
77:             // failedLocksLimit 减一
78:             } else {
79:                 failedLocksLimit--;
80:             }
81:         }
82:
83:         // 计算剩余时间 remainTime
84:         if (remainTime != -1) {
85:             remainTime -= System.currentTimeMillis() - time;
86:             // 记录新的当前时间
87:             time = System.currentTimeMillis();
88:             // 如果没有剩余时间，意味着已经超时，释放所有加载成功的锁，并返回 false
89:             if (remainTime <= 0) {
90:                 unlockInner(acquiredLocks);
91:                 return false;
92:             }
93:         }
94:     }
95:
96:     // 如果设置了锁的过期时间 leaseTime ，则重新设置每个锁的过期时间
97:     if (leaseTime != -1) {
98:         // 遍历 acquiredLocks 数组，创建异步设置过期时间的 Future
99:         List<RFuture<Boolean>> futures = new ArrayList<>(acquiredLocks.size());
100:        for (RLock rLock : acquiredLocks) {
101:            RFuture<Boolean> future = ((RedissonLock) rLock).expireAsync(unit.toMillis(leaseTime), TimeUnit.MILL
102:            futures.add(future);
103:        }
```

```
104:
105:         // 阻塞等待所有 futures 完成
106:         for (RFuture<Boolean> rFuture : futures) {
107:             rFuture.syncUninterruptibly();
108:         }
109:     }
110:
111:     // 返回 true ，表示加锁成功
112:     return true;
113: }
```

超 100 行代码，是不是有点慌？！核心逻辑是，首先从 locks 数组中逐个获得锁（第 27 至 94 行），然后统一设置每个锁的过期时间（第 96 至 10 9 行）。当然，还是有很多细节，我们一点点来看。

第 3 至 11 行：计算新的锁的时长 newLeaseTime 。注意，newLeaseTime 是用于遍历 locks 数组来获得锁设置的锁时长，最终在第 96 至 109 行的代码中，会设置真正的 leaseTime 锁的时长。整个的计算规则，看下艿艿添加的注释。

第 14 至 18 行：计算剩余等待锁的时间 remainTime 。

第 20 行：调用 #calcLockWaitTime(long remainTime) 方法，计算获得每个锁的等待时间。在 「2. RedissonRedLock」 中，我们已经看到，是 remainTime 进行平均分配。

第 25 行：已经获得到锁的数组 acquiredLocks 。

下面，我们分成阶段一（加锁）和阶段二（设置锁过期时间）来抽丝剥茧。

【阶段一】第 26 至 94 行：遍历 RLock 数组，逐个获得锁。

第 32 至 39 行：根据条件，调用对应的 RLock#tryLock(...) 方法，获得锁。一般情况下，RedissonRedLock 搭配 RedissonLock 使用，这块我们已经在 《精尽 Redisson 源码分析 —— 可重入分布式锁 ReentrantLock》 有个详细的解析了。

第 40 至 44 行：如果发生响应 RedisResponseTimeoutException 超时异常时，因为无法确定实际是否获得到锁，所以直接调用 #unlockInner(Collection<RLock> locks) 方法，释放当前 RLock 。

第 50 至 52 行：如果获得成功，则添加到 acquiredLocks 数组中。

【重要】第 54 至 57 行：如果已经到达最少需要获得锁的数量，则直接 break 。例如说，RedLock 只需要获得 N / 2 + 1 把。

第 77 至 80 行：failedLocksLimit 减一。

第 59 至 76 行：当已经没有允许失败的数量，则进行相应的处理。

- 第 62 行：因为已经失败了，所以调用 #unlockInner(Collection<RLock> locks) 方法，释放所有已经获得到的锁们。
- 第 63 至 66 行：如果未设置阻塞时间，直接返回 false 加锁失败。因为是 tryLock 方法，只是尝试加锁一次，不会无限重试。
- 第 67 至 76 行：重置整个获得锁的过程，在剩余的时间里，重新来一遍。因为已设置了阻塞时间，必须得用完！

第 84 至 93 行：计算剩余时间 remainTime 。如果没有剩余的时间，意味着已经超时，则 调用 #unlockInner(Collection<RLock> locks) 方法，释放所有加载成功的锁，并返回 false 加锁失败。

【阶段二】第 96 至 109 行：如果设置了锁的过期时间 leaseTime ，则重新设置每个锁的过期时间。

第 98 至 103 行：遍历 acquiredLocks 数组，创建异步设置过期时间的 Future 。

第 105 至 108 行：阻塞等待所有 futures 完成。如果任一一个 Future 执行失败，则会抛出异常。

3.5 tryLockAsync

#tryLockAsync(long waitTime, long leaseTime, TimeUnit unit, long threadId) 方法，异步加锁，并返回加锁是否成功。代码如下：

```
// RedissonMultiLock.java

@Override
public RFuture<Boolean> tryLockAsync(long waitTime, long leaseTime, TimeUnit unit) {
    return tryLockAsync(waitTime, leaseTime, unit, Thread.currentThread().getId());
}

@Override
public RFuture<Boolean> tryLockAsync(long waitTime, long leaseTime, TimeUnit unit, long threadId) {
    // 创建 RPromise 对象，用于通知结果
    RPromise<Boolean> result = new RedissonPromise<Boolean>();
    // 创建 LockState 对象
    LockState state = new LockState(waitTime, leaseTime, unit, threadId);
    // <X> 发起异步加锁
    state.tryAcquireLockAsync(locks.listIterator(), result);
    // 返回结果
    return result;
}
```

这个的逻辑在 `<X>` 处，调用 `LockState#tryAcquireLockAsync(ListIterator<RLock> iterator, RPromise<Boolean> result)` 方法，发起异步加锁。

LockState 是 RedissonMultiLock 的内部类，实现异步加锁的逻辑。构造方法如下：

```
// RedissonMultiLock.LockState.java

class LockState {

    private final long newLeaseTime;
    private final long lockWaitTime;
    private final List<RLock> acquiredLocks;
    private final long waitTime;
    private final long threadId;
    private final long leaseTime;
    private final TimeUnit unit;

    private long remainTime;
    private long time = System.currentTimeMillis();
    private int failedLocksLimit;

    LockState(long waitTime, long leaseTime, TimeUnit unit, long threadId) {
        this.waitTime = waitTime;
        this.leaseTime = leaseTime;
        this.unit = unit;
        this.threadId = threadId;

        // 计算新的锁的时长 newLeaseTime
        if (leaseTime != -1) {
            if (waitTime == -1) {
                newLeaseTime = unit.toMillis(leaseTime);
            } else {
                newLeaseTime = unit.toMillis(waitTime) * 2;
            }
        } else {
            newLeaseTime = -1;
        }

        // 计算剩余等待锁的时间 remainTime
```

```
            remainTime = -1;
            if (waitTime != -1) {
                remainTime = unit.toMillis(waitTime);
            }
            // 计算每个锁的等待时间
            lockWaitTime = calcLockWaitTime(remainTime);

            // 允许获得锁失败的次数
            failedLocksLimit = failedLocksLimit();
            // 已经获得到锁的数组
            acquiredLocks = new ArrayList<>(locks.size());
        }

        // ... 省略其它方法
    }
```

构造方法的逻辑，和 「3.4 tryLock(long waitTime, long leaseTime, TimeUnit unit)」 的第 3 至 25 行的代码是一致的。所以，我们可以理解成构造方法，相当于做了一遍变量的初始化。

`LockState#tryAcquireLockAsync(ListIterator<RLock> iterator, RPromise<Boolean> result)` 方法，发起异步加锁。代码如下：

整体逻辑，和 「3.4 tryLock(long waitTime, long leaseTime, TimeUnit unit)」 方法的逻辑基本一致，所以芋艿就不啰嗦详细说，而是告诉它们的对等关系。

```
// RedissonMultiLock.LockState.java

 1: void tryAcquireLockAsync(ListIterator<RLock> iterator, RPromise<Boolean> result) {
 2:     // 如果迭代 iterator 的尾部，则重新设置每个锁的过期时间
 3:     if (!iterator.hasNext()) {
 4:         checkLeaseTimeAsync(result);
 5:         return;
 6:     }
 7:
 8:     // 获得下一个 RLock 对象
 9:     RLock lock = iterator.next();
10:     // 创建 RPromise 对象
11:     RPromise<Boolean> lockAcquiredFuture = new RedissonPromise<Boolean>();
12:     // 如果等待时间 waitTime 为 -1（不限制），并且锁时长为 -1（不限制），则使用 #tryLock() 方法。
13:     if (waitTime == -1 && leaseTime == -1) {
14:         lock.tryLockAsync(threadId)
15:             .onComplete(new TransferListener<Boolean>(lockAcquiredFuture));
16:     // 如果任一不为 -1 时，则计算新的等待时间 awaitTime ，然后调用 #tryLock(long waitTime, long leaseTime, TimeU
17:     } else {
18:         long awaitTime = Math.min(lockWaitTime, remainTime);
19:         lock.tryLockAsync(awaitTime, newLeaseTime, TimeUnit.MILLISECONDS, threadId)
20:             .onComplete(new TransferListener<Boolean>(lockAcquiredFuture));
21:     }
22:
23:     lockAcquiredFuture.onComplete((res, e) -> {
24:         // 如果 res 非空，设置 lockAcquired 是否获得到锁
25:         boolean lockAcquired = false;
26:         if (res != null) {
27:             lockAcquired = res;
28:         }
29:
```

```
30:         // 发生响应超时。因为无法确定实际是否获得到锁，所以直接释放当前 RLock
31:         if (e instanceof RedisResponseTimeoutException) {
32:             unlockInnerAsync(Collections.singletonList(lock), threadId);
33:         }
34:
35:         // 如果获得成功，则添加到 acquiredLocks 数组中
36:         if (lockAcquired) {
37:             acquiredLocks.add(lock);
38:         } else {
39:             // 如果已经到达最少需要获得锁的数量，则直接 break 。例如说，RedLock 只需要获得 N / 2 + 1 把。
40:             if (locks.size() - acquiredLocks.size() == failedLocksLimit()) {
41:                 checkLeaseTimeAsync(result);
42:                 return;
43:             }
44:
45:             // 当已经没有允许失败的数量，则进行相应的处理
46:             if (failedLocksLimit == 0) {
47:                 // 创建释放所有的锁的 Future
48:                 unlockInnerAsync(acquiredLocks, threadId).onComplete((r, ex) -> {
49:                     // 如果发生异常，则通过 result 回调异常
50:                     if (ex != null) {
51:                         result.tryFailure(ex);
52:                         return;
53:                     }
54:
55:                     // 如果未设置阻塞时间，直接通知 result 失败。因为是 tryLock ，只是尝试加锁一次，不会无限重试
56:                     if (waitTime == -1) {
57:                         result.trySuccess(false);
58:                         return;
59:                     }
60:
61:                     // 重置整个获得锁的过程，在剩余的时间里，重新来一遍
62:                     // 重置 failedLocksLimit 变量
63:                     failedLocksLimit = failedLocksLimit();
64:                     // 重置 acquiredLocks 为空
65:                     acquiredLocks.clear();
66:                     // reset iterator
67:                     // 重置 iterator 设置回迭代器的头
68:                     while (iterator.hasPrevious()) {
69:                         iterator.previous();
70:                     }
71:
72:                     // 校验剩余时间是否足够
73:                     checkRemainTimeAsync(iterator, result);
74:                 });
75:                 return;
76:             } else {
77:                 failedLocksLimit--;
78:             }
79:         }
80:
81:         // 校验剩余时间是否足够
82:         checkRemainTimeAsync(iterator, result);
83:     });
84: }
```

第 2 至 6 行：如果迭代 `iterator` 的尾部，则调用 `#checkLeaseTimeAsync(RPromise<Boolean> result)` 方法，重新设置每个锁的过期时间。代码如下：

```
// RedissonMultiLock.LockState.java

private void checkLeaseTimeAsync(RPromise<Boolean> result) {
    // 如果设置了锁的过期时间 leaseTime ，则重新设置每个锁的过期时间
    if (leaseTime != -1) {
        // 创建 AtomicInteger 计数器，用于回调逻辑的计数，从而判断是不是所有回调都执行完了
        AtomicInteger counter = new AtomicInteger(acquiredLocks.size());
        // 遍历 acquiredLocks 数组，逐个设置过期时间
        for (RLock rLock : acquiredLocks) {
            // 创建异步设置过期时间的 RFuture
            RFuture<Boolean> future = ((RedissonLock) rLock).expireAsync(unit.toMillis(leaseTime), TimeUnit.MIL
            future.onComplete((res, e) -> {
                // 如果发生异常，则通过 result 回调异常
                if (e != null) {
                    result.tryFailure(e);
                    return;
                }

                // 如果全部成功，则通过 result 回调加锁成功
                if (counter.decrementAndGet() == 0) {
                    result.trySuccess(true);
                }
            });
        }
        return;
    }

    // 如果未设置了锁的过期时间 leaseTime ，则通过 result 回调加锁成功
    result.trySuccess(true);
}
```

- 对标到 「3.4 tryLock(long waitTime, long leaseTime, TimeUnit unit)」 的第 96 至 109 行。

第 8 至 79 行：对标到 「3.4 tryLock(long waitTime, long leaseTime, TimeUnit unit)」 的第 28 至 81 行。

第 73 和 82 行：调用 #checkRemainTimeAsync(ListIterator<RLock> iterator, RPromise<Boolean> result) 方法，校验剩余时间是否足够。代码如下：

```
// RedissonMultiLock.LockState.java

private void checkRemainTimeAsync(ListIterator<RLock> iterator, RPromise<Boolean> result) {
    // 如果设置了等待超时时间，计算剩余时间 remainTime
    if (remainTime != -1) {
        remainTime += -(System.currentTimeMillis() - time);
        // 记录新的当前时间
        time = System.currentTimeMillis();
        // 如果没有剩余时间，意味着已经超时，释放所有加载成功的锁
        if (remainTime <= 0) {
            // 创建释放所有已获得到锁们的 Future
            unlockInnerAsync(acquiredLocks, threadId).onComplete((res, e) -> {
                // 如果发生异常，则通过 result 回调异常
                if (e != null) {
                    result.tryFailure(e);
                    return;
                }
```

```
                    // 如果全部成功，则通过 result 回调加锁成功
                    result.trySuccess(false);
                });
                // return 返回结束
                return;
            }
        }

        // <Y> 如果未设置等待超时时间，则继续加锁下一个 RLock
        tryAcquireLockAsync(iterator, result);
    }
```

- 对标到 「3.4 tryLock(long waitTime, long leaseTime, TimeUnit unit)」 的第 83 至 93 行。
- 注意，<Y> 处，会递归调用 #tryAcquireLockAsync(ListIterator<RLock> iterator, RPromise<Boolean> result) 方法，继续加锁下一个 RLock 。

总的来说，还是蛮简单的不是，哈哈哈哈。

3.6 lock

#lockInterruptibly(long leaseTime, TimeUnit unit) 方法，同步加锁，不返回加锁是否成功。代码如下：

```
// RedissonMultiLock.java

@Override
public void lock() {
    try {
        lockInterruptibly();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

@Override
public void lock(long leaseTime, TimeUnit unit) {
    try {
        lockInterruptibly(leaseTime, unit);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

@Override
public void lockInterruptibly() throws InterruptedException {
    lockInterruptibly(-1, null);
}
```

```
 1: @Override
 2: public void lockInterruptibly(long leaseTime, TimeUnit unit) throws InterruptedException {
 3:     // 计算 waitTime 时间
 4:     long baseWaitTime = locks.size() * 1500;
 5:     long waitTime = -1;
 6:     if (leaseTime == -1) { // 如果未设置超时时间，则直接使用 baseWaitTime
 7:         waitTime = baseWaitTime;
 8:     } else {
```

```
 9:          leaseTime = unit.toMillis(leaseTime);
10:          waitTime = leaseTime;
11:          if (waitTime <= 2000) { // 保证最小 waitTime 时间是 2000
12:              waitTime = 2000;
13:          } else if (waitTime <= baseWaitTime) { // 在 [waitTime / 2, waitTime) 之间随机
14:              waitTime = ThreadLocalRandom.current().nextLong(waitTime / 2, waitTime);
15:          } else { // 在 [baseWaitTime, waitTime) 之间随机
16:              waitTime = ThreadLocalRandom.current().nextLong(baseWaitTime, waitTime);
17:          }
18:      }
19:
20:      // 死循环，直到加锁成功
21:      while (true) {
22:          if (tryLock(waitTime, leaseTime, TimeUnit.MILLISECONDS)) {
23:              return;
24:          }
25:      }
26: }
```

第 2 至 18 行：计算 `waitTime` 时间。 我也没弄懂，为啥是这么设计，难道是因为经验值？后面去细细的翻查下原因。
第 20 至 25 行：死循环，调用 [3.4 #tryLock(long waitTime, long leaseTime, TimeUnit unit)] 方法，直到加锁成功。

3.7 lockAsync

`#lockAsync(long leaseTime, TimeUnit unit, long threadId)` 方法，异步加锁，不返回加锁是否成功。代码如下：

```
// RedissonMultiLock.java

@Override
public RFuture<Void> lockAsync(long leaseTime, TimeUnit unit) {
    return lockAsync(leaseTime, unit, Thread.currentThread().getId());
}
```

```
 1: @Override
 2: public RFuture<Void> lockAsync(long leaseTime, TimeUnit unit, long threadId) {
 3:     // 计算 waitTime 时间
 4:     long baseWaitTime = locks.size() * 1500;
 5:     long waitTime;
 6:     if (leaseTime == -1) { // 如果未设置超时时间，则直接使用 baseWaitTime
 7:         waitTime = baseWaitTime;
 8:     } else {
 9:         leaseTime = unit.toMillis(leaseTime);
10:         waitTime = leaseTime;
11:         if (waitTime <= 2000) { // 保证最小 waitTime 时间是 2000
12:             waitTime = 2000;
13:         } else if (waitTime <= baseWaitTime) { // 在 [waitTime / 2, waitTime) 之间随机
14:             waitTime = ThreadLocalRandom.current().nextLong(waitTime / 2, waitTime);
15:         } else { // 在 [baseWaitTime, waitTime) 之间随机
16:             waitTime = ThreadLocalRandom.current().nextLong(baseWaitTime, waitTime);
17:         }
18:     }
19:
20:     // 创建 RPromise 对象
```

```
21:     RPromise<Void> result = new RedissonPromise<Void>();
22:     // 执行异步加锁
23:     tryLockAsync(threadId, leaseTime, TimeUnit.MILLISECONDS, waitTime, result);
24:     return result;
25: }
```

第 3 至 18 行：计算 `waitTime` 时间。和 `#lockInterruptibly(long leaseTime, TimeUnit unit)` 方法，看到的逻辑是一致的。

第 23 行：调用 `#tryLockAsync(long threadId, long leaseTime, TimeUnit unit, long waitTime, RPromise<Void> result)` 方法，执行异步加锁。代码如下：

```
// RedissonMultiLock.java

protected void tryLockAsync(long threadId, long leaseTime, TimeUnit unit, long waitTime, RPromise<Void> result)
    // <X1> 执行异步加锁锁
    tryLockAsync(waitTime, leaseTime, unit, threadId).onComplete((res, e) -> {
        // 如果发生异常，则通过 result 回调异常
        if (e != null) {
            result.tryFailure(e);
            return;
        }

        // <X2> 如果加锁成功，则通知 result 成功
        if (res) {
            result.trySuccess(null);
        // <X3> 如果加锁失败，则递归调用 tryLockAsync 方法
        } else {
            tryLockAsync(threadId, leaseTime, unit, waitTime, result);
        }
    });
}
```

- `<X1>` 处，调用 3.5 `#tryLockAsync(long waitTime, long leaseTime, TimeUnit unit, long threadId)` 方法，执行异步加锁锁。
- `<X2>` 处，如果加锁成功，则通知 `result` 成功。
- `<X3>` 处，如果加锁失败，则递归调用 `#lockAsync(long leaseTime, TimeUnit unit, long threadId)`（自己）方法，继续执行异步加锁。

3.8 unlock

`#unlock()` 方法，同步解锁。代码如下：

```
// RedissonMultiLock.java

@Override
public void unlock() {
    // 创建 RFuture 数组
    List<RFuture<Void>> futures = new ArrayList<>(locks.size());

    // 逐个创建异步解锁 Future，并添加到 futures 数组中
    for (RLock lock : locks) {
        futures.add(lock.unlockAsync());
```

```
        }

        // 同步阻塞 futures ，全部释放完成
        for (RFuture<Void> future : futures) {
            future.syncUninterruptibly();
        }
    }
```

在 RedissonMultiLock 类中，存在一个 #unlockInner(Collection<RLock> locks) 方法，同步解锁指定 RLock 数组。代码如下：

```
    // RedissonMultiLock.java

    protected void unlockInner(Collection<RLock> locks) {
        // 创建 RFuture 数组
        List<RFuture<Void>> futures = new ArrayList<>(locks.size());

        // 逐个创建异步解锁 Future，并添加到 futures 数组中
        for (RLock lock : locks) {
            futures.add(lock.unlockAsync());
        }

        // 同步阻塞 futures ，全部释放完成
        for (RFuture<Void> unlockFuture : futures) {
            unlockFuture.awaitUninterruptibly();
        }
    }
```

在 RedissonMultiLock 类中，存在一个 #unlockInner(Collection<RLock> locks) 方法，异步解锁指定 RLock 数组。代码如下：

```
    // RedissonMultiLock.java

    protected RFuture<Void> unlockInnerAsync(Collection<RLock> locks, long threadId) {
        // 如果 locks 为空，直接返回成功的 RFuture
        if (locks.isEmpty()) {
            return RedissonPromise.newSucceededFuture(null);
        }

        // 创建 RPromise 对象
        RPromise<Void> result = new RedissonPromise<Void>();
        // 创建 AtomicInteger 计数器，用于回调逻辑的计数，从而判断是不是所有回调都执行完了
        AtomicInteger counter = new AtomicInteger(locks.size());
        // 遍历 acquiredLocks 数组，逐个解锁
        for (RLock lock : locks) {
            lock.unlockAsync(threadId).onComplete((res, e) -> {
                // 如果发生异常，则通过 result 回调异常
                if (e != null) {
                    result.tryFailure(e);
                    return;
                }

                // 如果全部成功，则通过 result 回调解锁成功
                if (counter.decrementAndGet() == 0) {
                    result.trySuccess(null);
                }
```

```
        });
    }
    return result;
}
```

3.9 未实现的方法

在 RedissonMultiLock 中，因为一些方法暂时没必要实现，所以就都未提供。如下：

```
// RedissonMultiLock.java

@Override
public Condition newCondition() {
    throw new UnsupportedOperationException();
}

@Override
public RFuture<Boolean> forceUnlockAsync() {
    throw new UnsupportedOperationException();
}

@Override
public RFuture<Integer> getHoldCountAsync() {
    throw new UnsupportedOperationException();
}

@Override
public String getName() {
    throw new UnsupportedOperationException();
}

@Override
public boolean forceUnlock() {
    throw new UnsupportedOperationException();
}

@Override
public boolean isLocked() {
    throw new UnsupportedOperationException();
}

@Override
public RFuture<Boolean> isLockedAsync() {
    throw new UnsupportedOperationException();
}

@Override
public boolean isHeldByThread(long threadId) {
    throw new UnsupportedOperationException();
}

@Override
public boolean isHeldByCurrentThread() {
    throw new UnsupportedOperationException();
}

@Override
```

```
    public int getHoldCount() {
        throw new UnsupportedOperationException();
    }

    @Override
    public RFuture<Long> remainTimeToLiveAsync() {
        throw new UnsupportedOperationException();
    }

    @Override
    public long remainTimeToLive() {
        throw new UnsupportedOperationException();
    }
```

666. 彩蛋

一开始，以为 RedLock 红锁的代码会比较复杂，所以在撸这块的源码时，有点懵逼。一度计划，准备花小 1 天的时间来研究和输出这篇博客。结果发现，竟然是个纸老虎，哈哈哈。

所以把，碰到任何源码，都不要怂。该肝就肝！

爽，在 2019-10-05 的 01:30 写完了这篇博客，美滋滋。

文章目录