
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 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
2019-10-04
Redis

精尽 Redisson 源码分析 —— 可重入分布式锁 ReentrantLock

1. 概述

在 Redisson 中，提供了 8 种分布锁的实现，具体我们可以在 《Redisson 文档 —— 分布式锁和同步器》 中看到。绝大数情况下，我们使用可重入锁（Reentrant Lock）就够了，对应到就是 org.redisson.RedissonLock 类，具体的使用示例可以看看 《芋道 Spring Boot Redis 入门》 的「6.2 Redis 分布式锁」小节 。

在 《精尽 Redis 面试题》 的问题中，我们在聊到“如何使用 Redis 实现分布式锁？”这个题目中，提到了需要考虑的 7 个方面，这里我们再来重复看下：

1、正确的获得锁

set 指令附带 nx 参数，保证有且只有一个进程获得到。

2、正确的释放锁

使用 Lua 脚本，比对锁持有的是不是自己。如果是，则进行删除来释放。

3、超时的自动释放锁

set 指令附带 expire 参数，通过过期机制来实现超时释放。

4、未获得到锁的等待机制

sleep 或者基于 Redis 的订阅 Pub/Sub 机制。

一些业务场景，可能需要支持获得不到锁，直接返回 false ，不等待。

5、【可选】锁的重入性

通过 ThreadLocal 记录是第几次获得相同的锁。

1）有且第一次计数为 1 && 获得锁时，才向 Redis 发起获得锁的操作。
2）有且计数为 0 && 释放锁时，才向 Redis 发起释放锁的操作。

6、锁超时的处理

一般情况下，可以考虑告警 + 后台线程自动续锁的超时时间。通过这样的机制，保证有且仅有一个线程，正在持有锁。

7、Redis 分布式锁丢失问题

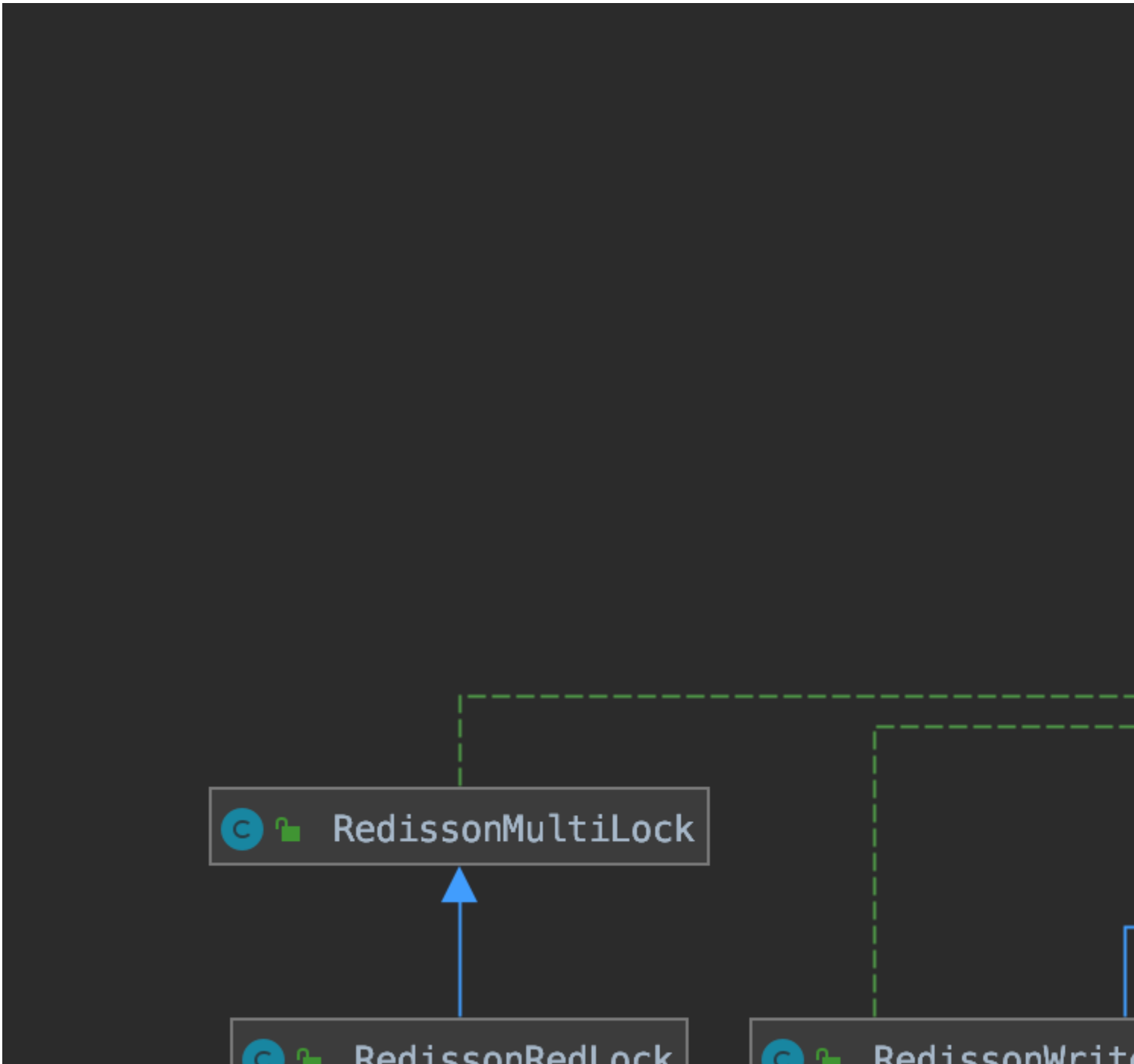
具体看「方案二：Redlock」。

RedissonLock 实现了前 6 点，而第 7 点需要通过 org.redisson.RedissonRedLock 来实现，这个话题，我们后面在聊。

在开始阅读源码之前，胖友可以先看看芮芮画的 Redisson 实现分布式锁的整体流程图，以便更好的阅读源码。猛击传送门

2. 整体一览

我们来看看 Redisson 锁相关的整体类图，如下：



org.redisson.api.RLockAsync ，定义了异步操作的接口。
org.redisson.api.RLock ，继承 RLockAsync 的基础上，定义了同步操作的接口。比较有意思的是，RLock 同时实现继承 JDK 的 java.util.concurrent.locks.Lock 接口，从而符合 Java 的 Lock 的标准。
本文的主角 RedissonLock ，实现 RLock 接口，提供可重入的分布式锁实现。
其它的 RLock 实现的关系，胖友自己看图哈。

RLockAsync 和 RLock 定义的接口，差别就在于同步和异步，所以我们就只看看 RLock 接口。代码如下：

```
String getName();

// 锁定相关的接口方法，还有部分在 Lock 接口上
void lockInterruptibly(long leaseTime, TimeUnit unit) throws InterruptedException;
void lock(long leaseTime, TimeUnit unit);
boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws InterruptedException;

// 解锁相关的接口方法，还有部分在 Lock 接口上
boolean forceUnlock();

// 其它非关键方法
boolean isLocked();
boolean isHeldByThread(long threadId);
boolean isHeldByCurrentThread();
int getHoldCount();
long remainTimeToLive();
```

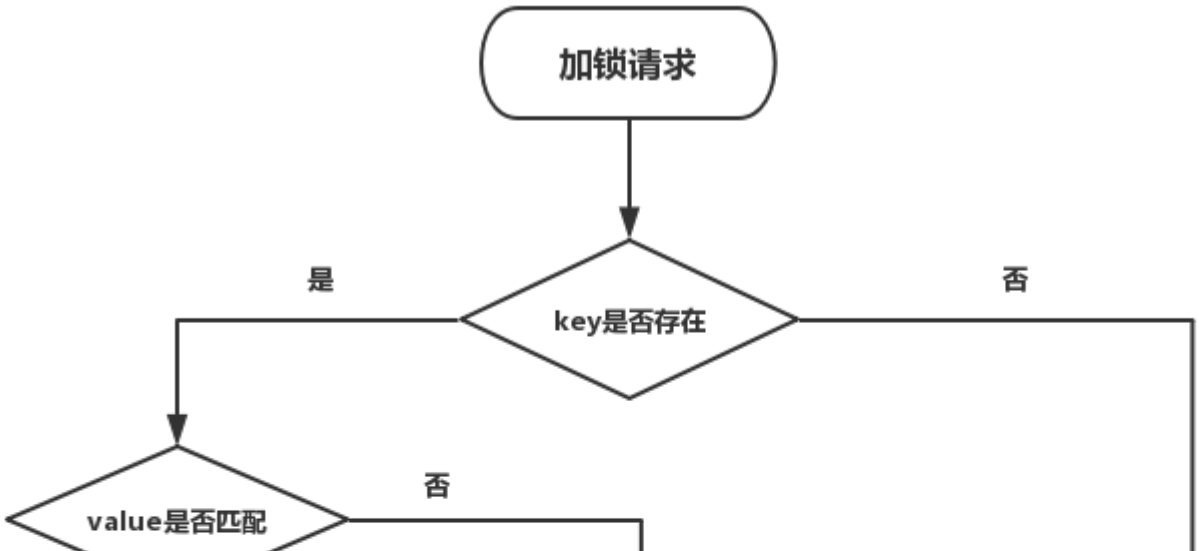
3. Lua 脚本

在我们看具体的代码实现，我们先来看核心的使用到的 Lua 脚本，方便我们后续更好的理解 RedissonLock 的实现。

3.1 tryLockInnerAsync

#tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> command) 方法，实现加锁逻辑，并且支持可重入性。代码如下：

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



```
// RedissonLock.java

 1: <T> RFuture<T> tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> command) {
 2:     internalLockLeaseTime = unit.toMillis(leaseTime);
 3:
 4:     return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,
 5:               "if (redis.call('exists', KEYS[1]) == 0) then " + // 情况一，当前分布式锁被未被获得
 6:                   "redis.call('hset', KEYS[1], ARGV[2], 1); " + // 写入分布锁被 ARGV[2] 获取到了，设置数量为 1 。
 7:                   "redis.call('pexpire', KEYS[1], ARGV[1]); " + // 设置分布式的过期时间为 ARGV[1]
 8:                   "return nil; " + // 返回 null ，表示成功
 9:               "end; " +
10:               "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " + // 情况二，如果当前分布锁已经被 ARGV[2
11:                   "redis.call('hincrby', KEYS[1], ARGV[2], 1); " + // 写入持有计数字 + 1 。
12:                   "redis.call('pexpire', KEYS[1], ARGV[1]); " + //  设置分布式的过期时间为 ARGV[1]
13:                   "return nil; " + // 返回 null ，表示成功
14:               "end; " +
15:               "return redis.call('pttl', KEYS[1]);", // 情况三，获取不到分布式锁，则返回锁的过期时间。
16:                 Collections.<Object>singletonList(getName()), // KEYS[分布式锁名]
17:             internalLockLeaseTime, getLockName(threadId)); // ARGV[锁超时时间，获得的锁名]
18: }
```

<2> 处，根据传入的 `leaseTime` + `unit` 参数，设置到 `internalLockLeaseTime` 属性上，表示锁的时长。代码如下：

```
// RedissonLock.java

/**
 * 锁的时长
 */
protected long internalLockLeaseTime;

public RedissonLock(CommandAsyncExecutor commandExecutor, String name) {
    // ... 省略其它代码
    this.internalLockLeaseTime = commandExecutor.getConnectionManager().getCfg().getLockWatchdogTimeout();
}
```

- 默认情况下，`internalLockLeaseTime` 属性，使用 Lock 的 WatchDog 的超时时长 `30 * 1000` 毫秒。默认的值，当且仅当我们未显示传入锁的时长时，才有用。例如说，稍后我们会看到的 `#lock()` 等等方法中。
- 有一点，我们要特别注意，`internalLockLeaseTime` 是 RedissonLock 的成员变量，并且也未声明 `volatile` 修饰，所以跨线程使用同一个 RedissonLock 对象，可能会存在 `internalLockLeaseTime` 读取不到最新值的情况。

还是熟悉的配方，通过 Lua 脚本实现。具体传入的参数，朋友看下第 16 和 17 行的代码，对应的 `KEYS` 和 `ARGV` 。可能有几个值胖友会有点懵逼，我们先来看看。

- `KEYS[1]` ：调用 `#getName()` 方法获得分布式锁的名字。稍后，我们会看到分布式锁在 Redis 使用是以 `KEYS[1]` 分布式锁为 KEY ，VALUE 为 HASH 类型。
- `ARGV[1]` ：锁的时长。
- `ARGV[2]` ：调用 `#getLockName(threadId)` 方法，获得的锁名。该名字，用于表示该分布式锁正在被哪个进程的线程所持有。代码如下：

```
// RedissonLock.java
```

```
/**
 * ID ，就是 {@link ConnectionManager#getId()}
 */
final String id;

protected String getLockName(long threadId) {
    return id + ":" + threadId;
}
```

- 可能描述看起来不是很好理解，我们来看一个获取到分布式锁的示例：

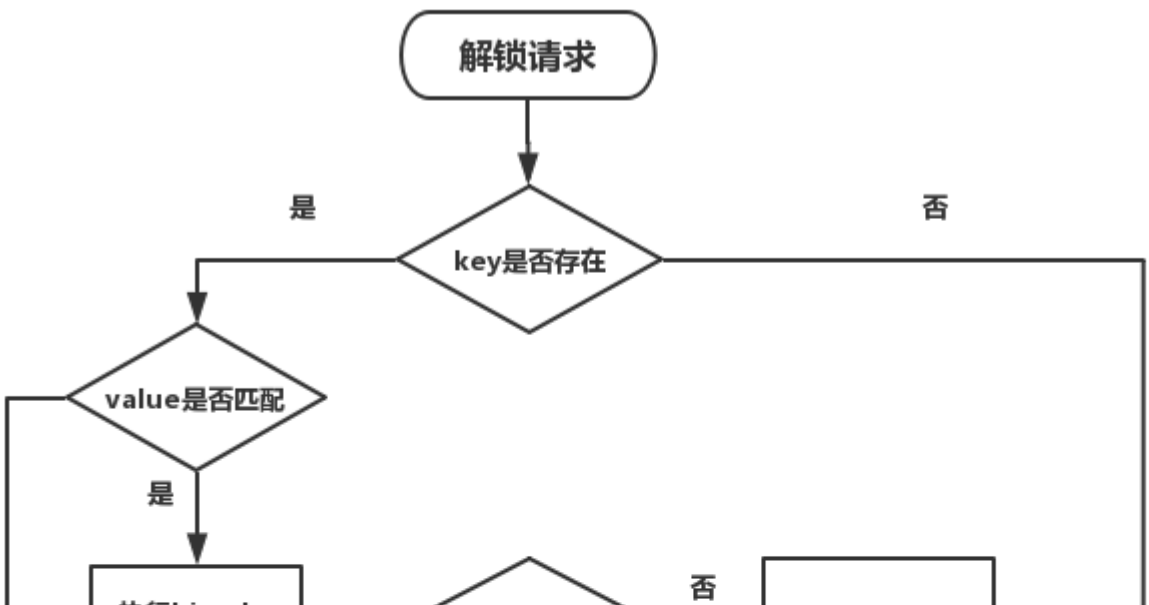
第 4 至 15 行：Lua 脚本，一共分成 3 种情况，胖友认真仔细看看，艿艿已经添加了完整的注释。

不同于我们在市面上看到的 Redis 通过 SET 命令带上 NX 和 EXPIRE 的方式实现获得分布式锁，RedissonLock 提供重入性，所以需要 Lua 脚本来实现。当然，实际上，也可以通过 ThreadLocal 来实现重入性的技术，胖友可以思考一波，不懂的话星球来给艿艿留言。

3.2 unlockInnerAsync

`#unlockInnerAsync(long threadId)` 方法，实现解锁逻辑，并且支持可重入性。代码如下：

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



```
// RedissonLock.java

 1: protected RFuture<Boolean> unlockInnerAsync(long threadId) {
 2:     return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVAL_BOOLEAN,
 3:             "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " + // 情况一，分布式锁未被 ARGV[3] 持有，则
 4:                 "return nil;" +
 5:             "end; " +
 6:             "local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1); " + // 持有锁的数量减 1 。
 7:             "if (counter > 0) then " + // 情况二，如果后还有剩余的持有锁数量，则返回 0 ，表示解锁未完成
 8:                 "redis.call('pexpire', KEYS[1], ARGV[2]); " + // 重新设置过期时间为 ARGV[2]
 9:                 "return 0; " +
10:             "else " + // 情况三，不存在剩余的锁数量，则返回 1 ，表示解锁成功
11:                 "redis.call('del', KEYS[1]); " + // 删除对应的分布式锁对应的 KEYS[1]
12:                 "redis.call('publish', KEYS[2], ARGV[1]); " + // 发布解锁事件到 KEYS[2] ，通知气他可能要获取锁的
13:                 "return 1; "+
14:             "end; " +
15:             "return nil;", // 不存在这个情况。
16:             Arrays.<Object>asList(getName(), getChannelName()), // KEYS[分布式锁名，该分布式锁对应的 Channel 名]
17:             LockPubSub.UNLOCK_MESSAGE, internalLockLeaseTime, getLockName(threadId)); // ARGV[解锁消息，锁超时时
18: }
```

具体传入的参数，朋友看下第 16 和 17 行的代码，对应的 KEYS 和 ARGV 。

- KEYS[1] ：调用 #getName() 方法获得分布式锁的名字。
- KEYS[2] ：调用 #getChannelName() 方法，该分布式锁对应的 Channel 名。因为 RedissonLock 释放锁时，会通过该 Channel 来 Publish 一条消息，通知其它可能在阻塞等待这条消息的客户端。代码如下：

```
// RedissonLock.java

String getChannelName() {
    return prefixName("redisson_lock__channel", getName());
}
```

  - 通过 Redis Pub/Sub 机制，实现未获得到锁的等待机制。
  - 每个分布式锁，对应一个其独有的 Channel 。
- ARGV[1] ：解锁消息 LockPubSub.UNLOCK_MESSAGE 。通过收到这条消息，其它等待锁的客户端，会重新发起获得锁的请求。具体的，我们在下文来一起瞅瞅。
- ARGV[2] ：锁的时长。
- ARGV[3] ：调用 #getLockName(threadId) 方法，获得的锁名。

第 3 至 15 行：Lua 脚本，还是分成 3 种情况，胖友认真仔细看看，艿艿已经添加了完整的注释。

不同于我们在市面上看到的 Redis 通过 Lua 脚本的方式实现释放分布式锁，一共有 2 点：

1、要实现可重入性，所以只有在计数为 0 时，才会真正释放锁。
2、要实现客户端的等待通知，所以在释放锁时，Publish 一条释放锁的消息。

3.3 forceUnlockAsync

#forceUnlockAsync() 方法，实现强制解锁逻辑。代码如下：

```
// RedissonLock.java

@Override
public RFuture<Boolean> forceUnlockAsync() {
    cancelExpirationRenewal(null);
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVAL_BOOLEAN,
            "if (redis.call('del', KEYS[1]) == 1) then " // 情况一，释放锁成功，则通过 Publish 发布释放锁的消息，并返
            + "redis.call('publish', KEYS[2], ARGV[1]); "
            + "return 1 "
            + "else " // 情况二，释放锁失败，因为不存在这个 KEY ，所以返回 0
            + "return 0 "
            + "end",
            Arrays.<Object>asList(getName(), getChannelName()),
            LockPubSub.UNLOCK_MESSAGE);
}
```

比较简单，分成 2 种情况。胖友认真仔细看看，芋艿已经添加了完整的注释。

代码处理的比较细致，Redis DEL 成功，才 PUBLISH 发布释放锁的消息，避免错误通知客户端。

3.4 renewExpirationAsync

`#renewExpirationAsync(long threadId)` 方法，实现续锁逻辑。可能胖友有点懵逼，至少芋艿看到这段逻辑，完全不知道为何意啊。

我们先来看下 [《Redisson 文档 —— 分布式锁和同步器》](#)，有一段奇怪的说明：

```
RLock lock = redisson.getLock("anyLock");
// 最常见的使用方法
lock.lock();
```

> 大家都知道，如果负责储存这个分布式锁的Redisson节点宕机以后，而且这个锁正好处于锁住的状态时，这个锁会出现锁死的状态。为了避免这种情况的发生，Redisson内部提供了一个监控锁的看门狗，它的作用是在Redisson实例被关闭前，不断的延长锁的有效期。默认情况下，看门狗的检查锁的超时时间是30秒钟，也可以通过修改 [Config.lockWatchdogTimeout](#) 来另行指定。

在使用 `RedissonLock#lock()` 方法，我们要求持续持有锁，直到手动释放。但是实际上，我们有一个隐藏条件，如果 Java 进程挂掉时，需要自动释放。那么，如果实现 `RedissonLock#lock()` 时，设置过期 Redis 为无限大，或者不过期都不合适。那么 RedissonLock 是怎么实现的呢？RedissonLock 先获得一个 `internalLockLeaseTime` 的分布式锁，然后每 `internalLockLeaseTime / 3` 时间，定时调用 `#renewExpirationAsync(long threadId)` 方法，进行续租。这样，在 Java 进程异常 Crash 掉后，能够保证最多 `internalLockLeaseTime` 时间后，分布式锁自动释放。

略骚略巧妙~不过为了实现这样的功能，RedissonLock 的整体逻辑，又复杂了一丢丢。

下面，还是先让我们看下具体的 `#renewExpirationAsync(long threadId)` 方法的代码，如下：

```
// RedissonLock.java

protected RFuture<Boolean> renewExpirationAsync(long threadId) {
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVAL_BOOLEAN,
```

```
                "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " + // 情况一，如果持有锁，则重新设置过期时间为 A
                    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
                    "return 1; " +
                "end; " +
                "return 0;", // 情况二，未吃货有，返回 0 续租失败。
        Collections.<Object>singletonList(getName()),
        internalLockLeaseTime, getLockName(threadId));
}
```

比较简单，分成 2 种情况。胖友认真仔细看看，艿艿已经添加了完整的注释。

至此，我们看完了 Lua 脚本部分，其实基本也大体知道 RedissonLock 是如何实现加锁、接锁的逻辑。但是，复杂的逻辑，还在下面，胖友请保持好耐心，开启我们的高能时刻。

> 艿艿：T T RedissonLock 好多重载的方法，文章小标题，都不造杂取了。最关键的是，可能胖友会被绕进去。

4. LockPubSub

在开始研究真正的加锁和解锁的调用之前，我们先看看和其相关的客户端订阅解锁消息，从而实现在持有锁的客户端释放锁时，等待锁的客户端能够快速的去调用加锁逻辑。

> 艿艿： 整个调用栈太深的，艿艿只好先写它，就当卖了一个关子，哈哈哈。

org.redisson.pubsub.LockPubSub ，继承 PublishSubscribe 抽象类，实现 Lock 相关消息的订阅。代码如下：

```
// LockPubSub.java

public class LockPubSub extends PublishSubscribe<RedissonLockEntry> {

    /**
     * 锁释放的消息
     */
    public static final Long UNLOCK_MESSAGE = 0L;
    /**
     * 读锁释放的消息
     */
    public static final Long READ_UNLOCK_MESSAGE = 1L;

    public LockPubSub(PublishSubscribeService service) {
        super(service);
    }

    @Override
    protected RedissonLockEntry createEntry(RPromise<RedissonLockEntry> newPromise) {
        return new RedissonLockEntry(newPromise);
    }

    @Override
    protected void onMessage(RedissonLockEntry value, Long message) {
        if (message.equals(UNLOCK_MESSAGE)) {
            // 回调监听器
```

```
                Runnable runnableToExecute = value.getListeners().poll();
                if (runnableToExecute != null) {
                    runnableToExecute.run();
                }

                // 通过信号量，通知阻塞等待的线程
                value.getLatch().release();
            } else if (message.equals(READ_UNLOCK_MESSAGE)) {
                while (true) {
                    Runnable runnableToExecute = value.getListeners().poll();
                    if (runnableToExecute == null) {
                        break;
                    }
                    runnableToExecute.run();
                }

                value.getLatch().release(value.getLatch().getQueueLength());
            }
        }

    }
```

在 `#createEntry(RPromise<RedissonLockEntry> newPromise)` 方法，会创建 [org.redisson.RedissonLockEntry](#) 对象。代码如下：

```
        // RedissonLockEntry.java

        public class RedissonLockEntry implements PubSubEntry<RedissonLockEntry> {

            /**
             * 计数器
             *
             * 每次发起订阅，则计数器 + 1
             * 每次取消订阅，则计数器 - 1 。当减少到 0 时，才正常取消订阅。
             */
            private int counter;

            /**
             * 信号量，用于实现 RedissonLock 阻塞等待的通知
             */
            private final Semaphore latch;
            private final RPromise<RedissonLockEntry> promise;
            /**
             * 监听器们
             */
            private final ConcurrentLinkedQueue<Runnable> listeners = new ConcurrentLinkedQueue<Runnable>();

            public RedissonLockEntry(RPromise<RedissonLockEntry> promise) {
                super();
                this.latch = new Semaphore(0);
                this.promise = promise;
            }

            @Override
            public void aquire() {
                counter++;
            }
```

```
        @Override
        public int release() {
            return --counter;
        }

        @Override
        public RPromise<RedissonLockEntry> getPromise() {
            return promise;
        }

        public void addListener(Runnable listener) {
            listeners.add(listener);
        }

        public boolean removeListener(Runnable listener) {
            return listeners.remove(listener);
        }

        public ConcurrentLinkedQueue<Runnable> getListeners() {
            return listeners;
        }

        public Semaphore getLatch() {
            return latch;
        }

    }
```

- 虽然代码比较多，我们重点来看 `latch` 和 `listeners` 属性。
- `latch` 属性：信号量，用于实现 RedissonLock 阻塞等待的通知。在我们下面看到同步加锁的逻辑，会看到通过它来实现阻塞等待。
- `listeners` 属性：监听器，实现订阅到锁的释放消息，从而再次发起获得锁。当然，这里的 Runnable 对象肯定无法体现，具体我们后面看看 `#tryLockAsync(AtomicLong time, long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Boolean> result, long currentThreadId)` 或者 `#lockAsync(long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Void> result, long currentThreadId)` 方法，就可以看到方法内部会创建具体的 Runnable 实现类，实现再次发起获得锁的逻辑。

在 `#onMessage(RedissonLockEntry value, Long message)` 方法中，在接收到释放锁的消息后，会执行 `listeners` 的回调，以及 `latch` 的时候放。

当然，单单看 LockPubSub 类，胖友可能会感到懵逼，保持耐心，继续向下。LockPubSub 更多的是实现了锁释放消息的监听，以及回调监听器，释放信号量。真正的逻辑，还是要看监听器的逻辑，以及 RedissonLock 是怎么实现信号量的。

另外，在 RedissonLock 中，提供如下几个方法，发起和取消订阅。代码如下：

```
// RedissonLock.java

/**
 * Sub Entry 名字
 */
final String entryName;

protected final LockPubSub pubSub;

public RedissonLock(CommandAsyncExecutor commandExecutor, String name) {
```

```
        // ... 省略其他无关
        this.entryName = id + ":" + name;
        this.pubSub = commandExecutor.getConnectionManager().getSubscribeService().getLockPubSub();
    }

    /**
     * 获得线程对应的 RedissonLockEntry 对象
     *
     * @param threadId 线程编号
     * @return RedissonLockEntry 对象
     */
    protected RedissonLockEntry getEntry(long threadId) {
        return pubSub.getEntry(getEntryName());
    }

    /**
     * 异步发起订阅
     *
     * @param threadId 线程编号
     * @return RFuture 对象
     */
    protected RFuture<RedissonLockEntry> subscribe(long threadId) {
        return pubSub.subscribe(getEntryName(), getChannelName());
    }

    /**
     * 异步取消订阅
     *
     * @param future RFuture 对象
     * @param threadId 线程编号
     */
    protected void unsubscribe(RFuture<RedissonLockEntry> future, long threadId) {
        pubSub.unsubscribe(future.getNow(), getEntryName(), getChannelName());
    }
```

5. tryLockAsync

芳芳：重点开始了，打起精神。

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

```
    // RedissonLock.java

    @Override
    public RFuture<Boolean> tryLockAsync(long waitTime, TimeUnit unit) {
        return tryLockAsync(waitTime, -1, unit);
    }

    @Override
    public RFuture<Boolean> tryLockAsync(long waitTime, long leaseTime, TimeUnit unit) {
        // 获得线程编号
        long currentThreadId = Thread.currentThread().getId();
        // 执行锁
        return tryLockAsync(waitTime, leaseTime, unit, currentThreadId);
    }
```

最终都调用 #tryLockAsync(long waitTime, long leaseTime, TimeUnit unit, long currentThreadId) 方法，真正实现异步加锁的逻辑。

#tryLockAsync(long waitTime, long leaseTime, TimeUnit unit, long currentThreadId) 方法，代码如下：

```
// RedissonLock.java

@Override
public RFuture<Boolean> tryLockAsync(long waitTime, long leaseTime, TimeUnit unit, long currentThreadId) {
    // 创建 RPromise 对象，用于通知结果
    RPromise<Boolean> result = new RedissonPromise<Boolean>();

    // 表示剩余的等待获得锁的时间
    AtomicLong time = new AtomicLong(unit.toMillis(waitTime));
    // 记录当前时间
    long currentTime = System.currentTimeMillis();
    // 执行异步获得锁
    RFuture<Long> ttlFuture = tryAcquireAsync(leaseTime, unit, currentThreadId);
    ttlFuture.onComplete((ttl, e) -> {
        // 如果发生异常，则通过 result 通知异常
        if (e != null) {
            result.tryFailure(e);
            return;
        }

        // lock acquired
        // 如果获得到锁，则通过 result 通知获得锁成功
        if (ttl == null) {
            if (!result.trySuccess(true)) { // 如果处理 result 通知对结果返回 false ，意味着需要异常释放锁
                unlockAsync(currentThreadId);
            }
            return;
        }

        // 减掉已经等待的时间
        long el = System.currentTimeMillis() - currentTime;
        time.addAndGet(-el);

        // 如果无剩余等待的时间，则通过 result 通知获得锁失败
        if (time.get() <= 0) {
            trySuccessFalse(currentThreadId, result);
            return;
        }

        // 记录新的当前时间
        long current = System.currentTimeMillis();
        // 记录下面的 future 的指向
        AtomicReference<Timeout> futureRef = new AtomicReference<Timeout>();

        // 创建 SUBSCRIBE 订阅的 Future
        RFuture<RedissonLockEntry> subscribeFuture = subscribe(currentThreadId);
        subscribeFuture.onComplete((r, ex) -> {
            // 如果发生异常，则通过 result 通知异常
            if (ex != null) {
                result.tryFailure(ex);
                return;
            }

            // 如果创建定时任务 Future scheduledFuture，则进行取消
            if (futureRef.get() != null) {
```

```
54:                 futureRef.get().cancel();
55:             }
56:
57:             // 减掉已经等待的时间
58:             long elapsed = System.currentTimeMillis() - current;
59:             time.addAndGet(-elapsed);
60:
61:             // 再次执行异步获得锁
62:             tryLockAsync(time, leaseTime, unit, subscribeFuture, result, currentThreadId);
63:         });
64:
65:         // 如果创建 SUBSCRIBE 订阅的 Future 未完成，创建定时任务 Future scheduledFuture 。
66:         if (!subscribeFuture.isDone()) {
67:             Timeout scheduledFuture = commandExecutor.getConnectionManager().newTimeout(new TimerTask() {
68:                 @Override
69:                 public void run(Timeout timeout) throws Exception {
70:                     // 如果创建 SUBSCRIBE 订阅的 Future 未完成
71:                     if (!subscribeFuture.isDone()) {
72:                         // 进行取消 subscribeFuture
73:                         subscribeFuture.cancel(false);
74:                         // 通过 result 通知获得锁失败
75:                         trySuccessFalse(currentThreadId, result);
76:                     }
77:                 }
78:             }, time.get(), TimeUnit.MILLISECONDS); // 延迟 time 秒后执行
79:             // 记录 futureRef 执行 scheduledFuture
80:             futureRef.set(scheduledFuture);
81:         }
82:     });
83:
84:     return result;
85: }
```

整体逻辑是，获得分布锁。如果获取失败，则发起 Redis Pub/Sub 订阅，等待释放锁的消息，从而再次发起获得分布式锁。

第 11 行：调用 `#tryAcquireAsync(long leaseTime, TimeUnit unit, long threadId)` 方法，执行异步获得锁。详细解析，胖友先跳到 「5.1 tryAcquireAsync」 中。

继续开始我们“漫长”的回调之旅。其实也比较容易懂，走起~

第 13 至 17 行：如果发生异常，则通过 `result` 通知异常。

第 19 至 26 行：如果 `ttl` 为空，说明获得到锁了，则通过 `result` 通知获得锁成功。这里，在第 23 至 24 行有个小细节，胖友自己看下注释。

第 41 行：声明 `futureRef` 变量，用于设置第 65 至 81 行创建的定时任务。

第 65 至 82 行：如果创建 SUBSCRIBE 订阅的 Future `subscribeFuture` 未完成，创建定时任务 Future `scheduledFuture` 。因为 `subscribeFuture` 是异步的，而存在一个情况，可能 `subscribeFuture` 未完成时，等待获得锁已经超时，所以通过 `scheduledFuture` 来实现超时通知。

- 第 80 行：记录 `futureRef` 为 `scheduledFuture` 。
- 第 71 行：兜底判断 `subscribeFuture` 未完成。
- 第 73 行：进行取消 `subscribeFuture` 。

- 第 75 行：调用 `#trySuccessFalse(long currentThreadId, RPromise<Boolean> result)` 方法，通知获得锁失败。代码如下：

```
// RedissonLock.java
```

```
    protected RFuture<Void> acquireFailedAsync(long threadId) {
        return RedissonPromise.newSucceededFuture(null);
    }

    private void trySuccessFalse(long currentThreadId, RPromise<Boolean> result) {
        acquireFailedAsync(currentThreadId).onComplete((res, e) -> {
            if (e == null) { // 通知获得锁失败
                result.trySuccess(false);
            } else { // 通知异常
                result.tryFailure(e);
            }
        });
    }
```

- x

第 43 至 63 行：创建 SUBSCRIBE 订阅的 Future `subscribeFuture` 。通过订阅释放锁的消息，从而实现等待锁释放的客户端，快速抢占加锁。

- 第 46 至 50 行：如果发生异常，则通过 `result` 通知异常。
- 第 52 至 55 行：如果创建定时任务 Future `scheduledFuture`，则进行取消。
- 第 57 至 59 行：减掉已经等待的时间。

- 第 62 行：调用 `#tryLockAsync(AtomicLong time, long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Boolean> result, long currentThreadId` 方法，再次执行异步获得锁。详细解析，见 「5.2 更强的 tryLockAsync」 小节。

> 芮芮：特喵的，又是一个 tryLockAsync 重载的方法，我已经瞎取标题了。深呼吸，继续！

感叹，想要写好全异步的代码，实际是非常困难的，所以芮芮的感受，Spring Webflux 反应式框架，想要推广在编写业务逻辑，基本可能性是为零。当然，Webflux 乃至反应式编程，更加适合推广在基础组件中。

5.1 tryAcquireAsync

> 芮芮：看完这个方法，就跳回去哈。MMP 整个调用链，真长，大几百行代码。

`#tryAcquireAsync(long leaseTime, TimeUnit unit, long threadId)` 方法，执行异步获得锁。代码如下：

```
// RedissonLock.java

private <T> RFuture<Long> tryAcquireAsync(long leaseTime, TimeUnit unit, long threadId) {
    // <1> 情况一，如果锁有时长，则直接获得分布式锁
    if (leaseTime != -1) {
        return tryLockInnerAsync(leaseTime, unit, threadId, RedisCommands.EVAL_LONG);
    }

    // <2> 情况二，如果锁无时长，则先获得 Lock WatchDog 的锁超时时长
    RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(commandExecutor.getConnectionManager().getCfg().getLockWatch
    ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
        // 如果发生异常，则直接返回
        if (e != null) {
```

```
            return;
        }

        // lock acquired
        // 如果获得到锁，则创建定时任务，定时续锁
        if (ttlRemaining == null) {
            scheduleExpirationRenewal(threadId);
        }
    });
    return ttlRemainingFuture;
}
```

一共分成两种情况，是否锁有时长。
<1> 处，`leaseTime != -1` ，意味着锁设置了时长，则调用 「3.1 #tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> command)」 方法，直接获得分布式锁。
<2> 处，锁未设置了时长，所以先调用 「3.1 #tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> command)」 方法，获得 Lock WatchDog 的锁超时时长的分布式锁，然后在回调中，再调用 #scheduleExpirationRenewal(long threadId) 方法，创建定时任务，定时调用 「3.4 renewExpirationAsync」 续锁。详细解析，见 TODO 。

5.2 更强的 tryLockAsync

#tryLockAsync(AtomicLong time, long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Boolean> result, long currentThreadId) 方法，更强的异步加锁。主要强在 2 点：

1、增加监听锁释放的消息的监听器，从而实现等待锁的客户端快速抢占锁的逻辑。
2、增加锁超时自动释放，没有锁释放消息的处理。

整体代码如下：

```
// RedissonLock.java

 1: private void tryLockAsync(AtomicLong time, long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFu
 2:     // 如果 result 已经完成，则直接返回，并取消订阅
 3:     if (result.isDone()) {
 4:         unsubscribe(subscribeFuture, currentThreadId);
 5:         return;
 6:     }
 7:
 8:     // 如果剩余时间 time 小于 0 ，说明等待超时，则取消订阅，并通过 result 通知失败
 9:     if (time.get() <= 0) {
10:         unsubscribe(subscribeFuture, currentThreadId);
11:         trySuccessFalse(currentThreadId, result);
12:         return;
13:     }
14:
15:     // 记录当前时间
16:     long curr = System.currentTimeMillis();
17:     // 获得分布式锁
18:     RFuture<Long> ttlFuture = tryAcquireAsync(leaseTime, unit, currentThreadId);
19:     ttlFuture.onComplete((ttl, e) -> {
20:             // 如果发生异常，则取消订阅，并通过 result 通知异常
21:             if (e != null) {
22:                 unsubscribe(subscribeFuture, currentThreadId);
23:                 result.tryFailure(e);
```

```
24:                 return;
25:             }
26:
27:             // lock acquired
28:             // 如果获得到锁，则取消订阅，并通过 result 通知获得锁成功
29:             if (ttl == null) {
30:                 unsubscribe(subscribeFuture, currentThreadId);
31:                 if (!result.trySuccess(true)) {
32:                     unlockAsync(currentThreadId);
33:                 }
34:                 return;
35:             }
36:
37:             // 减掉已经等待的时间
38:             long el = System.currentTimeMillis() - curr;
39:             time.addAndGet(-el);
40:
41:             // 如果无剩余等待的时间，则取消订阅，并通过 result 通知获得锁失败
42:             if (time.get() <= 0) {
43:                 unsubscribe(subscribeFuture, currentThreadId);
44:                 trySuccessFalse(currentThreadId, result);
45:                 return;
46:             }
47:
48:             // waiting for message
49:             // 记录新的当前时间
50:             long current = System.currentTimeMillis();
51:             // 获得当前线程对应的 RedissonLockEntry 对象
52:             RedissonLockEntry entry = getEntry(currentThreadId);
53:             // 尝试获得 entry 中的信号量，如果获得成功，说明 SUBSCRIBE 已经收到释放锁的消息，则直接立马再次去获得
54:             if (entry.getLatch().tryAcquire()) {
55:                 tryLockAsync(time, leaseTime, unit, subscribeFuture, result, currentThreadId);
56:             } else {
57:                 // 创建 AtomicBoolean 变量 executed ，用于标记下面创建的 listener 是否执行。
58:                 AtomicBoolean executed = new AtomicBoolean();
59:                 // 创建 AtomicReference 对象，用于指向定时任务
60:                 AtomicReference<Timeout> futureRef = new AtomicReference<Timeout>();
61:
62:                 // 创建监听器 listener ，用于在 RedissonLockEntry 的回调，就是我们看到的 PublishSubscribe 监听到
63:                 Runnable listener = () -> {
64:                     // 标记已经执行
65:                     executed.set(true);
66:                     // 如果有定时任务的 Future ，则进行取消
67:                     if (futureRef.get() != null) {
68:                         futureRef.get().cancel();
69:                     }
70:
71:                     // 减掉已经等待的时间
72:                     long elapsed = System.currentTimeMillis() - current;
73:                     time.addAndGet(-elapsed);
74:
75:                     // 再次获得分布式锁
76:                     tryLockAsync(time, leaseTime, unit, subscribeFuture, result, currentThreadId);
77:                 };
78:                 // 添加 listener 到 RedissonLockEntry 中
79:                 entry.addListener(listener);
80:
81:                 // 下面，会创建一个定时任务。因为极端情况下，可能不存在释放锁的消息，例如说锁自动超时释放，所以需
82:                 long t = time.get();
83:                 if (ttl >= 0 && ttl < time.get()) { // 如果剩余时间小于锁的超时时间，则使用剩余时间。
```

```
 84:                     t = ttl;
 85:                 }
 86:                 // 如果 listener 未执行
 87:                 if (!executed.get()) {
 88:                     Timeout scheduledFuture = commandExecutor.getConnectionManager().newTimeout(new TimerTask()
 89:                         @Override
 90:                         public void run(Timeout timeout) throws Exception {
 91:                             // 移除 listener 从 RedissonLockEntry 中
 92:                             if (entry.removeListener(listener)) {
 93:                                 // 减掉已经等待的时间
 94:                                 long elapsed = System.currentTimeMillis() - current;
 95:                                 time.addAndGet(-elapsed);
 96:
 97:                                 // 再次获得分布式锁
 98:                                 tryLockAsync(time, leaseTime, unit, subscribeFuture, result, currentThreadId);
 99:                             }
100:                         }
101:                     }, t, TimeUnit.MILLISECONDS);
102:                     // 记录 futureRef 执行 scheduledFuture
103:                     futureRef.set(scheduledFuture);
104:                 }
105:             }
106:     });
107: }
```

第 2 至 46 行：和 「5. tryLockAsync」 基本一致，就不重复哔哔了。
第 52 行：调用 #getEntry(long threadId) 方法，获得当前线程对应的 RedissonLockEntry 对象。此处有点“失忆”的胖友，看看 「4. LockPubSub」 的结尾。
第 53 至 55 行：尝试获得 entry 中的信号量，如果获得成功，说明 SUBSCRIBE 已经收到释放锁的消息，则调用 「5.2 ##tryLockAsync(AtomicLong time, long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Boolean> result, long currentThreadId)」 方法，直接立马再次去获得锁。
第 58 行：创建 AtomicBoolean 变量 executed ，用于标记下面创建的 listener 是否执行。
第 60 行：声明 futureRef 变量，用于设置第 87 至 104 行创建的定时任务。因为极端情况下，可能不存在释放锁的消息，例如说锁自动超时释放，所以需要改定时任务，在获得到锁的超时后，主动去抢下。

- 第 82 至 85 行：计算定时任务的延迟时间时间。如果剩余时间小于锁的超时时间，则使用剩余时间。
- 第 87 行：通过 executed 变量，判断 listener 未执行。
- 第 103 行： 记录 futureRef 为 scheduledFuture 。
- 第 92 行：移除 listener 从 RedissonLockEntry 中。避免，可能存在的并发执行。
- 第 98 行：调用 「5.2 ##tryLockAsync(AtomicLong time, long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Boolean> result, long currentThreadId)」 方法，再次去获得锁。
- 这个定时任务，真的处理的是细节中的细节。之前思考获得分布式失败客户端的等待通知，只考虑了 Redis Pub/Sub 机制来实现，没有想到如果没有 PUBLISH 消息的场景。这块的逻辑，算是看 RedissonLock 最大的收获吧。

第 62 至 79 行：创建监听器 listener ，用于在 RedissonLockEntry 的回调，就是我们看到的 PublishSubscribe 监听到释放锁的消息，进行回调。

- 第 79 行：添加 listener 到 RedissonLockEntry 中。 如果胖友又“失忆”了，调回到 「4. LockPubSub」 再瞅瞅。
- 第 65 行：通过 executed 标记已经执行。
- 第 66 至 69 行：如果有定时任务的 Future ，则进行取消。
- 第 71 至 74 行：减掉已经等待的时间。
- 第 76 行：调用 「5.2 ##tryLockAsync(AtomicLong time, long leaseTime, TimeUnit unit,

RFuture<RedissonLockEntry> subscribeFuture, RPromise<Boolean> result, long currentThreadId)」 方法，再次去获得锁。

至此，RedissonLock 加锁的逻辑我们已经全部看完。如果觉得略感迷糊的胖友，可以多多调试下。因为芋艿有点偷懒，未画一些图来辅助胖友理解，所以胖友可以自己画一画，嘿嘿。

5.3 遗漏的 tryLockAsync

还有两个重载的 `#tryLockAsync(...)` 方法，它们是未设置锁定时长的两个。代码如下：

```
// RedissonLock.java

@Override
public RFuture<Boolean> tryLockAsync() {
    return tryLockAsync(Thread.currentThread().getId());
}

@Override
public RFuture<Boolean> tryLockAsync(long threadId) {
    return tryAcquireOnceAsync(-1, null, threadId);
}
```

最终都调用 `#tryAcquireOnceAsync(long leaseTime, TimeUnit unit, long threadId)` 方法，真正实现异步加锁的逻辑。

`#tryAcquireOnceAsync(long leaseTime, TimeUnit unit, long threadId)` 方法，真正实现异步加锁的逻辑。代码如下：

```
// RedissonLock.java

private RFuture<Boolean> tryAcquireOnceAsync(long leaseTime, TimeUnit unit, long threadId) {
    // 情况一，如果锁有时长，则直接获得分布式锁
    if (leaseTime != -1) {
        return tryLockInnerAsync(leaseTime, unit, threadId, RedisCommands.EVAL_NULL_BOOLEAN);
    }

    // 情况二，如果锁无时长，则先获得 Lock WatchDog 的锁超时时长
    RFuture<Boolean> ttlRemainingFuture = tryLockInnerAsync(commandExecutor.getConnectionManager().getCfg().getLockWa
    ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
        // 如果发生异常，则直接返回
        if (e != null) {
            return;
        }

        // lock acquired
        // 如果获得到锁，则创建定时任务，定时续锁
        if (ttlRemaining) {
            scheduleExpirationRenewal(threadId);
        }
    });
    return ttlRemainingFuture;
}
```

看到这个方法，是不是发现很熟悉，和 「5.1 tryAcquireAsync」 基本一模一样。差别在于

它的返回的结果是 RFuture<Boolean> 。
有一点要特别注意，因为本小节我们看到的两个 #tryLockAsync(...) 方法，是尝试去加锁。如果加锁失败，则返回 false 即可，所以不会像我们在 「5.1 tryLockAsync」 方法，无限重试直到等待超时（超过 waitTime）。

6. tryLock

芮芮：本小节和 「5. tryLockAsync」 相对，为同步加锁。扶你起来，胖友还可以继续怼源码。

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

```
// RedissonLock.java

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

```
 1: @Override
 2: public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws InterruptedException {
 3:     long time = unit.toMillis(waitTime);
 4:     long current = System.currentTimeMillis();
 5:     long threadId = Thread.currentThread().getId();
 6:     // 同步获加锁
 7:     Long ttl = tryAcquire(leaseTime, unit, threadId);
 8:     // lock acquired
 9:     // 加锁成功，直接返回 true 加锁成功
10:     if (ttl == null) {
11:         return true;
12:     }
13:
14:     // 减掉已经等待的时间
15:     time -= System.currentTimeMillis() - current;
16:     // 如果无剩余等待的时间，则返回 false 加锁失败
17:     if (time <= 0) {
18:         acquireFailed(threadId);
19:         return false;
20:     }
21:
22:     // 记录新的当前时间
23:     current = System.currentTimeMillis();
24:     // 创建 SUBSCRIBE 订阅的 Future
25:     RFuture<RedissonLockEntry> subscribeFuture = subscribe(threadId);
26:     // 阻塞等待订阅发起成功
27:     if (!await(subscribeFuture, time, TimeUnit.MILLISECONDS)) {
28:         // 进入到此处，说明阻塞等待发起订阅超时
29:         // 取消 SUBSCRIBE 订阅
30:         if (!subscribeFuture.cancel(false)) {
31:             // 进入到此处，说明取消发起订阅失败，则通过设置回调，在启发订阅完成后，回调取消 SUBSCRIBE 订阅
32:             subscribeFuture.onComplete((res, e) -> {
33:                 if (e == null) {
34:                     unsubscribe(subscribeFuture, threadId);
35:                 }
36:             });
37:         }
38:         // 等待超时，则返回 false 加锁失败
```

```
39:         acquireFailed(threadId);
40:         return false;
41:     }
42:
43:     try {
44:         // 减掉已经等待的时间
45:         time -= System.currentTimeMillis() - current;
46:         // 如果无剩余等待的时间，则返回 false 加锁失败
47:         if (time <= 0) {
48:             acquireFailed(threadId);
49:             return false;
50:         }
51:
52:         while (true) {
53:             // 记录新的当前时间
54:             long currentTime = System.currentTimeMillis();
55:             // 同步获加锁
56:             ttl = tryAcquire(leaseTime, unit, threadId);
57:             // lock acquired
58:             // 加锁成功，直接返回 true 加锁成功
59:             if (ttl == null) {
60:                 return true;
61:             }
62:
63:             // 减掉已经等待的时间
64:             time -= System.currentTimeMillis() - currentTime;
65:             // 如果无剩余等待的时间，则返回 false 加锁失败
66:             if (time <= 0) {
67:                 acquireFailed(threadId);
68:                 return false;
69:             }
70:
71:             // waiting for message
72:             // 记录新的当前时间
73:             currentTime = System.currentTimeMillis();
74:
75:             // 通过 RedissonLockEntry 的信号量，阻塞等待锁的释放消息，或者 ttl/time 超时（例如说，锁的自动超时释
76:             if (ttl >= 0 && ttl < time) {
77:                 getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
78:             } else {
79:                 getEntry(threadId).getLatch().tryAcquire(time, TimeUnit.MILLISECONDS);
80:             }
81:
82:             // 减掉已经等待的时间
83:             time -= System.currentTimeMillis() - currentTime;
84:             // 如果无剩余等待的时间，则返回 false 加锁失败
85:             if (time <= 0) {
86:                 acquireFailed(threadId);
87:                 return false;
88:             }
89:         }
90:     } finally {
91:         // 小细节，需要最终取消 SUBSCRIBE 订阅
92:         unsubscribe(subscribeFuture, threadId);
93:     }
94: //    return get(tryLockAsync(waitTime, leaseTime, unit));
95: }
```

第 7 行：调用 `#tryAcquire(long leaseTime, TimeUnit unit, long threadId)` 方法，同步加锁。代码如下：

```
// RedissonLock.java

private Long tryAcquire(long leaseTime, TimeUnit unit, long threadId) {
    return get(tryAcquireAsync(leaseTime, unit, threadId));
}
```

- 该方法内部，调用的就是 「5.1 #tryAcquireAsync(long leaseTime, TimeUnit unit, long threadId)」 方法。

第 8 至 12 行：加锁成功，直接返回 true 加锁成功。

第 15 行：减掉已经等待的时间。
第 17 至 20 行：如果无剩余等待的时间，则返回 false 加锁失败。
第 25 行：调用 #subscribe(long threadId) 方法，创建 SUBSCRIBE 订阅的 Future subscribeFuture 。
【重要差异点】第 27 至 41 行：调用 #await(subscribeFuture, time, TimeUnit.MILLISECONDS) 方法，阻塞等待订阅发起成功，因为 subscribeFuture 是异步的，需要这一步转同步。如果发生超时，则就会进入第 28 至 37 行的取消逻辑，并在第 38 至 40 行返回 false 加锁失败。
第 52 至 89 行：反复重试，直到成功加锁返回 true ，或者超时返回 false 。

- 第 54 至 73 行：重试一波第 6 至 20 行的逻辑。
- 【重要差异点】第 75 至 80 行：通过 RedissonLockEntry 的信号量，阻塞等待锁的释放消息，或者 ttl/time 超时（例如说，锁的自动超时释放）。
  - 相比 「5.2 #tryLockAsync(AtomicLong time, long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Boolean> result, long currentThreadId)」 方法，它把信号量的等待和定时任务的等待融合在一起了。
  - 等待完成后，如果无剩余时间，在第 82 至 88 行的逻辑中，返回 false 加锁失败。
  - 等待完成后，如果有剩余时间，在第 56 行：获得重新同步获得锁。

第 92 行：调用 #unsubscribe(RFuture<RedissonLockEntry> future, long threadId) 方法，小细节，需要最终取消 SUBSCRIBE 订阅。

7. lockAsync

芮芮：本小节和 「5. tryLockAsync」 相似，为异步加锁。继续扶你起来，胖友还可以继续怼源码。

#lockAsync(long leaseTime, TimeUnit unit, long currentThreadId) 方法，异步加锁，无需返回是否成功。代码如下：

```
// RedissonLock.java

    @Override
    public RFuture<Void> lockAsync() {
        return lockAsync(-1, null);
    }

    @Override
    public RFuture<Void> lockAsync(long leaseTime, TimeUnit unit) {
        // 获得线程编号
        long currentThreadId = Thread.currentThread().getId();
        // 异步锁
        return lockAsync(leaseTime, unit, currentThreadId);
```

```
    }

    @Override
    public RFuture<Void> lockAsync(long currentThreadId) {
        return lockAsync(-1, null, currentThreadId);
    }
```

```
 1: @Override
 2: public RFuture<Void> lockAsync(long leaseTime, TimeUnit unit, long currentThreadId) {
 3:     // 创建 RPromise 对象，用于异步回调
 4:     RPromise<Void> result = new RedissonPromise<Void>();
 5:     // 异步加锁
 6:     RFuture<Long> ttlFuture = tryAcquireAsync(leaseTime, unit, currentThreadId);
 7:     ttlFuture.onComplete((ttl, e) -> {
 8:         // 如果发生异常，则通过 result 通知异常
 9:         if (e != null) {
10:             result.tryFailure(e);
11:             return;
12:         }
13:
14:         // lock acquired
15:         // 如果获得到锁，则通过 result 通知获得锁成功
16:         if (ttl == null) {
17:             if (!result.trySuccess(null)) { // 如果处理 result 通知对结果返回 false ，意味着需要异常释放锁
18:                 unlockAsync(currentThreadId);
19:             }
20:             return;
21:         }
22:
23:         // 创建 SUBSCRIBE 订阅的 Future
24:         RFuture<RedissonLockEntry> subscribeFuture = subscribe(currentThreadId);
25:         subscribeFuture.onComplete((res, ex) -> {
26:             // 如果发生异常，则通过 result 通知异常
27:             if (ex != null) {
28:                 result.tryFailure(ex);
29:                 return;
30:             }
31:
32:             // 异步加锁
33:             lockAsync(leaseTime, unit, subscribeFuture, result, currentThreadId);
34:         });
35:     });
36:
37:     return result;
38: }
```

第 6 行：调用 「5.1 #tryAcquireAsync(long leaseTime, TimeUnit unit, long threadId)」 方法，执行异步获得锁。
第 7 至 35 行：又是熟悉的配方，在回调中，处理响应的加锁结果。差异就在第 34 行，见 「7.1 更强的 lockAsync」 的详细解析。

7.1 更强的 lockAsync

实际上，#lockAsync(long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Void> result, long currentThreadId) 方法，和 「5.2 更强的 tryLockAsync」 是基本一致的。那么为什么不直接重用呢？注意，这个方法不需要考虑等待超时，有一种“劳资有钱，必须拿到锁”。

代码如下：

```
// RedissonLock.java

private void lockAsync(long leaseTime, TimeUnit unit, RFuture<RedissonLockEntry> subscribeFuture, RPromise<Void> resu
    // 获得分布式锁
    RFuture<Long> ttlFuture = tryAcquireAsync(leaseTime, unit, currentThreadId);
    ttlFuture.onComplete((ttl, e) -> {
        // 如果发生异常，则取消订阅，并通过 result 通知异常
        if (e != null) {
            unsubscribe(subscribeFuture, currentThreadId);
            result.tryFailure(e);
            return;
        }

        // lock acquired
        // 如果获得到锁，则取消订阅，并通过 result 通知获得锁成功
        if (ttl == null) {
            unsubscribe(subscribeFuture, currentThreadId);
            if (!result.trySuccess(null)) {
                unlockAsync(currentThreadId);
            }
            return;
        }

        // 获得当前线程对应的 RedissonLockEntry 对象
        RedissonLockEntry entry = getEntry(currentThreadId);
        // 尝试获得 entry 中的信号量，如果获得成功，说明 SUBSCRIBE 已经收到释放锁的消息，则直接立马再次去获得锁。
        if (entry.getLatch().tryAcquire()) {
            lockAsync(leaseTime, unit, subscribeFuture, result, currentThreadId);
        } else {
            // waiting for message
            // 创建 AtomicReference 对象，用于指向定时任务
            AtomicReference<Timeout> futureRef = new AtomicReference<Timeout>();
            // 创建监听器 listener ，用于在 RedissonLockEntry 的回调，就是我们看到的 PublishSubscribe 监听到释放锁的消
            Runnable listener = () -> {
                // 如果有定时任务的 Future ，则进行取消
                if (futureRef.get() != null) {
                    futureRef.get().cancel();
                }
                // 再次获得分布式锁
                lockAsync(leaseTime, unit, subscribeFuture, result, currentThreadId);
            };
            // 添加 listener 到 RedissonLockEntry 中
            entry.addListener(listener);

            // 下面，会创建一个定时任务。因为极端情况下，可能不存在释放锁的消息，例如说锁自动超时释放，所以需要改定时
            if (ttl >= 0) {
                Timeout scheduledFuture = commandExecutor.getConnectionManager().newTimeout(new TimerTask() {
                    @Override
                    public void run(Timeout timeout) throws Exception {
                        // 移除 listener 从 RedissonLockEntry 中
                        if (entry.removeListener(listener)) {
                            // 再次获得分布式锁
                            lockAsync(leaseTime, unit, subscribeFuture, result, currentThreadId);
                        }
                    }
                }, ttl, TimeUnit.MILLISECONDS);
                // 记录 futureRef 执行 scheduledFuture
                futureRef.set(scheduledFuture);
```

```
            }
        }
    });
}
```

更加熟悉的配方，全程无需处理等待锁超时的逻辑。胖友自己瞅瞅，哈哈哈。

8. lock

芮芮：本小节和 「6. tryLoc」 相对，为同步加锁。再次扶你起来，胖友还可以继续怼源码。

#tryLock(long waitTime, long leaseTime, TimeUnit unit) 方法，同步加锁，无需返回是否成功。代码如下：

```
// RedissonLock.java

@Override
public void lock() {
    try {
        lock(-1, null, false);
    } catch (InterruptedException e) {
        throw new IllegalStateException();
    }
}

@Override
public void lock(long leaseTime, TimeUnit unit) {
    try {
        lock(leaseTime, unit, false);
    } catch (InterruptedException e) {
        throw new IllegalStateException();
    }
}

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

@Override
public void lockInterruptibly(long leaseTime, TimeUnit unit) throws InterruptedException {
    lock(leaseTime, unit, true);
}
```

```
 1: private void lock(long leaseTime, TimeUnit unit, boolean interruptibly) throws InterruptedException {
 2:     long threadId = Thread.currentThread().getId();
 3:     // 同步获加锁
 4:     Long ttl = tryAcquire(leaseTime, unit, threadId);
 5:     // lock acquired
 6:     // 加锁成功，直接返回
 7:     if (ttl == null) {
 8:         return;
 9:     }
10:
11:     // 创建 SUBSCRIBE 订阅的 Future
12:     RFuture<RedissonLockEntry> future = subscribe(threadId);
```

```
13:     // 阻塞等待订阅发起成功
14:     commandExecutor.syncSubscription(future);
15:
16:     try {
17:         while (true) {
18:             // 同步获加锁
19:             ttl = tryAcquire(leaseTime, unit, threadId);
20:             // lock acquired
21:             // 加锁成功，直接返回
22:             if (ttl == null) {
23:                 break;
24:             }
25:
26:             // waiting for message
27:             // 通过 RedissonLockEntry 的信号量，阻塞等待锁的释放消息，或者 ttl/time 超时（例如说，锁的自动超时释
28:             if (ttl >= 0) {
29:                 try {
30:                     getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
31:                 } catch (InterruptedException e) {
32:                     // 如果允许打断，则抛出 e
33:                     if (interruptibly) {
34:                         throw e;
35:                     }
36:                     // 如果不允许打断，则继续
37:                     getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
38:                 }
39:             } else {
40:                 if (interruptibly) {
41:                     getEntry(threadId).getLatch().acquire();
42:                 } else {
43:                     getEntry(threadId).getLatch().acquireUninterruptibly();
44:                 }
45:             }
46:         }
47:     } finally {
48:         // 小细节，需要最终取消 SUBSCRIBE 订阅
49:         unsubscribe(future, threadId);
50:     }
51: //    get(lockAsync(leaseTime, unit));
52: }
53:
54: private Long tryAcquire(long leaseTime, TimeUnit unit, long threadId) {
55:     return get(tryAcquireAsync(leaseTime, unit, threadId));
56: }
```

太过熟悉，就不哔哔了。

至此，加锁的几种组合排列，我们就已经看完了。 是不是有一种加锁的 Lua 脚本蛮简单的，调用 Lua 脚本实现阻塞等待的逻辑，细节还是蛮多的。如果让艿艿自己来实现这块的逻辑，估计会有一些细节处理不到位。嘿嘿。

9. unlockAsync

#unlockAsync(long threadId) 方法，异步解锁。代码如下：

```
// RedissonLock.java

    @Override
    public RFuture<Void> unlockAsync() {
        // 获得线程编号
        long threadId = Thread.currentThread().getId();
        // 执行解锁
        return unlockAsync(threadId);
    }
```

```
 1: @Override
 2: public RFuture<Void> unlockAsync(long threadId) {
 3:     // 创建 RPromise 对象，用于异步回调
 4:     RPromise<Void> result = new RedissonPromise<Void>();
 5:
 6:     // 解锁逻辑
 7:     RFuture<Boolean> future = unlockInnerAsync(threadId);
 8:
 9:     future.onComplete((opStatus, e) -> {
10:         // 如果发生异常，并通过 result 通知异常
11:         if (e != null) {
12:             cancelExpirationRenewal(threadId);
13:             result.tryFailure(e);
14:             return;
15:         }
16:
17:         // 解锁的线程不对，则创建 IllegalMonitorStateException 异常，并通过 result 通知异常
18:         if (opStatus == null) {
19:             IllegalMonitorStateException cause = new IllegalMonitorStateException("attempt to unlock lock, not l
20:                     + id + " thread-id: " + threadId);
21:             result.tryFailure(cause);
22:             return;
23:         }
24:
25:         // 取消定时过期
26:         cancelExpirationRenewal(threadId);
27:
28:         // 通知 result 解锁成功
29:         result.trySuccess(null);
30:     });
31:
32:     return result;
33: }
```

第 7 行：调用 [3.2 #unlockAsync(long threadId)](#) 方法，执行解锁逻辑。
第 10 至 15 行：如果发生异常，并通过 `result` 通知异常。
第 17 至 23 行：解锁的线程不对，则创建 `IllegalMonitorStateException` 异常，并通过 `result` 通知异常。 这里，仔细回忆下解锁 Lua 脚本的返回值。嘿嘿。
第 26 行：调用 `#cancelExpirationRenewal(long threadId)` 方法，取消定期过期。TODO
第 29 行：通知 `result` 解锁成功。

10. unlock

`#unlock()` 方法，同步解锁。代码如下：

```
// RedissonLock.java

@Override
public void unlock() {
    try {
        get(unlockAsync(Thread.currentThread().getId()));
    } catch (RedisException e) {
        if (e.getCause() instanceof IllegalMonitorStateException) {
            throw (IllegalMonitorStateException) e.getCause();
        } else {
            throw e;
        }
    }
}
```

简单，基于 「9. #unlockAsync(long threadId)」 方法实现。

11. forceUnlock

#forceUnlock() 方法，强制解锁。代码如下：

```
// RedissonLock.java

@Override
public boolean forceUnlock() { // 同步
    return get(forceUnlockAsync());
}

@Override
public RFuture<Boolean> deleteAsync() { // 异步
    return forceUnlockAsync();
}
```

无论是同步还是异步的强制解锁，都是基于 「3.3 #forceUnlockAsync()」 方法实现。

12. ExpirationEntry

本小节，我们来看看在 「3.4 renewExpirationAsync」 中，提到的续锁的功能。

首先，我们来看看 ExpirationEntry 类。它是 RedissonLock 的内部类，记录续租任务的信息。代码如下：

```
// RedissonLock.java

public static class ExpirationEntry {

    /**
     * 线程与计数器的映射
     *
     * KEY：线程编号
     * VALUE：计数
```

```
 */
private final Map<Long, Integer> threadIds = new LinkedHashMap<>();
/**
 * 定时任务
 */
private volatile Timeout timeout;

public ExpirationEntry() {
    super();
}

/**
 * 增加线程的计数
 *
 * @param threadId 线程编号
 */
public void addThreadId(long threadId) {
    Integer counter = threadIds.get(threadId);
    if (counter == null) {
        counter = 1;
    } else {
        counter++;
    }
    threadIds.put(threadId, counter);
}

public boolean hasNoThreads() {
    return threadIds.isEmpty();
}

public Long getFirstThreadId() {
    if (threadIds.isEmpty()) {
        return null;
    }
    return threadIds.keySet().iterator().next();
}

/**
 * 减少线程的技术
 *
 * @param threadId 线程编号
 */
public void removeThreadId(long threadId) {
    Integer counter = threadIds.get(threadId);
    if (counter == null) {
        return;
    }
    counter--;
    if (counter == 0) {
        threadIds.remove(threadId);
    } else {
        threadIds.put(threadId, counter);
    }
}

public void setTimeout(Timeout timeout) {
    this.timeout = timeout;
}

public Timeout getTimeout() {
```

```
        return timeout;
    }

}
```

可能粗略这么一看，有种然并卵的感觉，不要着急。我们下面接着看。
在 RedissonLock 的类中，有个 EXPIRATION_RENEWAL_MAP 静态属性，如下：

```
// RedissonLock.java

/**
 * ExpirationEntry 的映射
 *
 * key : {@link #entryName}
 */
private static final ConcurrentMap<String, ExpirationEntry> EXPIRATION_RENEWAL_MAP = new ConcurrentHashMap<>();
```

12.1 scheduleExpirationRenewal

#scheduleExpirationRenewal() 方法，发起续锁的定时任务。代码如下：

```
// RedissonLock.java

 1: private void scheduleExpirationRenewal(long threadId) {
 2:     // 创建 ExpirationEntry 对象
 3:     ExpirationEntry entry = new ExpirationEntry();
 4:     // 添加到 EXPIRATION_RENEWAL_MAP 中
 5:     ExpirationEntry oldEntry = EXPIRATION_RENEWAL_MAP.putIfAbsent(getEntryName(), entry);
 6:     // 添加线程编号到 ExpirationEntry 中
 7:     if (oldEntry != null) {
 8:         oldEntry.addThreadId(threadId);
 9:     } else {
10:         entry.addThreadId(threadId);
11:         // 创建定时任务，用于续锁
12:         renewExpiration();
13:     }
14: }
15:
16: private void renewExpiration() {
17:     // 获得 ExpirationEntry 队形，从 EXPIRATION_RENEWAL_MAP 中
18:     ExpirationEntry ee = EXPIRATION_RENEWAL_MAP.get(getEntryName());
19:     if (ee == null) { // 如果不存在，返回
20:         return;
21:     }
22:
23:     // 创建 Timeout 定时任务，实现定时续锁
24:     Timeout task = commandExecutor.getConnectionManager().newTimeout(new TimerTask() {
25:
26:         @Override
27:         public void run(Timeout timeout) throws Exception {
28:             // 获得 ExpirationEntry 对象
29:             ExpirationEntry ent = EXPIRATION_RENEWAL_MAP.get(getEntryName());
30:             if (ent == null) { // 如果不存在，返回
31:                 return;
32:             }
```

```
33:             // 获得 threadId 编号
34:             Long threadId = ent.getFirstThreadId();
35:             if (threadId == null) { // 如果不存在，则返回
36:                 return;
37:             }
38:
39:             // 执行续锁
40:             RFuture<Boolean> future = renewExpirationAsync(threadId);
41:             future.onComplete((res, e) -> {
42:                 // 如果发生异常，则打印异常日志，并返回。此时，就不会在定时续租了
43:                 if (e != null) {
44:                     log.error("Can't update lock " + getName() + " expiration", e);
45:                     return;
46:                 }
47:
48:                 // 续锁成功，则重新发起定时任务
49:                 if (res) {
50:                     // reschedule itself
51:                     renewExpiration();
52:                 }
53:             });
54:         }
55:
56:     }, internalLockLeaseTime / 3, TimeUnit.MILLISECONDS); // 定时，每 internalLockLeaseTime / 3 秒执行一次。
57:
58:     // 设置定时任务到 ExpirationEntry 中
59:     ee.setTimeout(task);
60: }
```

第 2 至 10 行：创建 ExpirationEntry 对象，并添加到 EXPIRATION_RENEWAL_MAP 中，之后添加线程编号到 ExpirationEntry 中。

第 12 行：当且仅当 entryName 对应的 ExpirationEntry 对象首次创建时，才会调用 #renewExpiration() 方法，创建定时任务，用于续锁。

- 【重要】第 23 至 56 行：创建 Timeout 定时任务，定时每 internalLockLeaseTime / 3 秒执行一次续锁。
- 第 40 行：会调用 「3.4 #renewExpirationAsync(long threadId) 方法」 方法，执行续锁。
- 第 42 至 46 行：如果发生异常，则打印异常日志，并返回。此时，就不会在定时续租了。
- 【重要】第 48 至 52 行：如果续锁成功，则调用 #renewExpiration() 方法，重新发起定时任务。

第 59 行：设置定时任务到 ExpirationEntry 中。

12.2 cancelExpirationRenewal

#cancelExpirationRenewal(Long threadId) 方法，取消定时任务。代码如下：

```
// RedissonLock.java

void cancelExpirationRenewal(Long threadId) {
    // 获得 ExpirationEntry 对象
    ExpirationEntry task = EXPIRATION_RENEWAL_MAP.get(getEntryName());
    if (task == null) { // 如果不存在，返回
        return;
    }
```

```
        // 从 ExpirationEntry 中，移除线程编号
        if (threadId != null) {
            task.removeThreadId(threadId);
        }

        // 如果 ExpirationEntry 的所有线程被清空
        if (threadId == null || task.hasNoThreads()) {
            // 取消定时任务
            task.getTimeout().cancel();
            // 从 EXPIRATION_RENEWAL_MAP 中移除
            EXPIRATION_RENEWAL_MAP.remove(getEntryName());
        }
    }
```

当且仅当 entryName 对应的 EXPIRATION_RENEWAL_MAP 的 ExpirationEntry 对象，所有线程都被移除后，会取消定时任务。
整体逻辑比较简单，胖友自己瞅瞅。

13. 其它方法

其它方法，比较简单，胖友自己瞅瞅即可。代码如下：

```
// RedissonLock.java

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

@Override
public boolean isLocked() {
    return isExists();
}

@Override
public RFuture<Boolean> isLockedAsync() {
    return isExistsAsync();
}

@Override
public RFuture<Boolean> isExistsAsync() {
    return commandExecutor.writeAsync(getName(), codec, RedisCommands.EXISTS, getName());
}

@Override
public boolean isHeldByCurrentThread() {
    return isHeldByThread(Thread.currentThread().getId());
}

@Override
public boolean isHeldByThread(long threadId) {
    RFuture<Boolean> future = commandExecutor.writeAsync(getName(), LongCodec.INSTANCE, RedisCommands.HEXISTS, getNam
    return get(future);
}
```

```
    private static final RedisCommand<Integer> HGET = new RedisCommand<Integer>("HGET", ValueType.MAP_VALUE, new IntegerR

    public RFuture<Integer> getHoldCountAsync() {
        return commandExecutor.writeAsync(getName(), LongCodec.INSTANCE, HGET, getName(), getLockName(Thread.currentThrea
    }

    @Override
    public int getHoldCount() {
        return get(getHoldCountAsync());
    }
```

666. 彩蛋

细节比想象中的多，代码也比想象中的多，整篇博客差不多写了 1.5 天左右。

胖友看完之后，如果还有一些细节不清晰，建议可以多多调试。总的来说，如果项目中，想要使用 Redis 分布式锁，可以考虑直接使用 Redisson 提供的 Redisson 可重入锁。可能有些胖友项目中，已经使用了 Jedis 作为 Redis 的客户端，那么可以单独使用 Redisson 来做分布式锁。

之前也和一些朋友聊过，他们项目也是采用 Jedis + Redisson 的组合，妥妥的，没问题。

满足，在 2019-10-04 的 18:59 写完了这篇博客，美滋滋。

文章目录