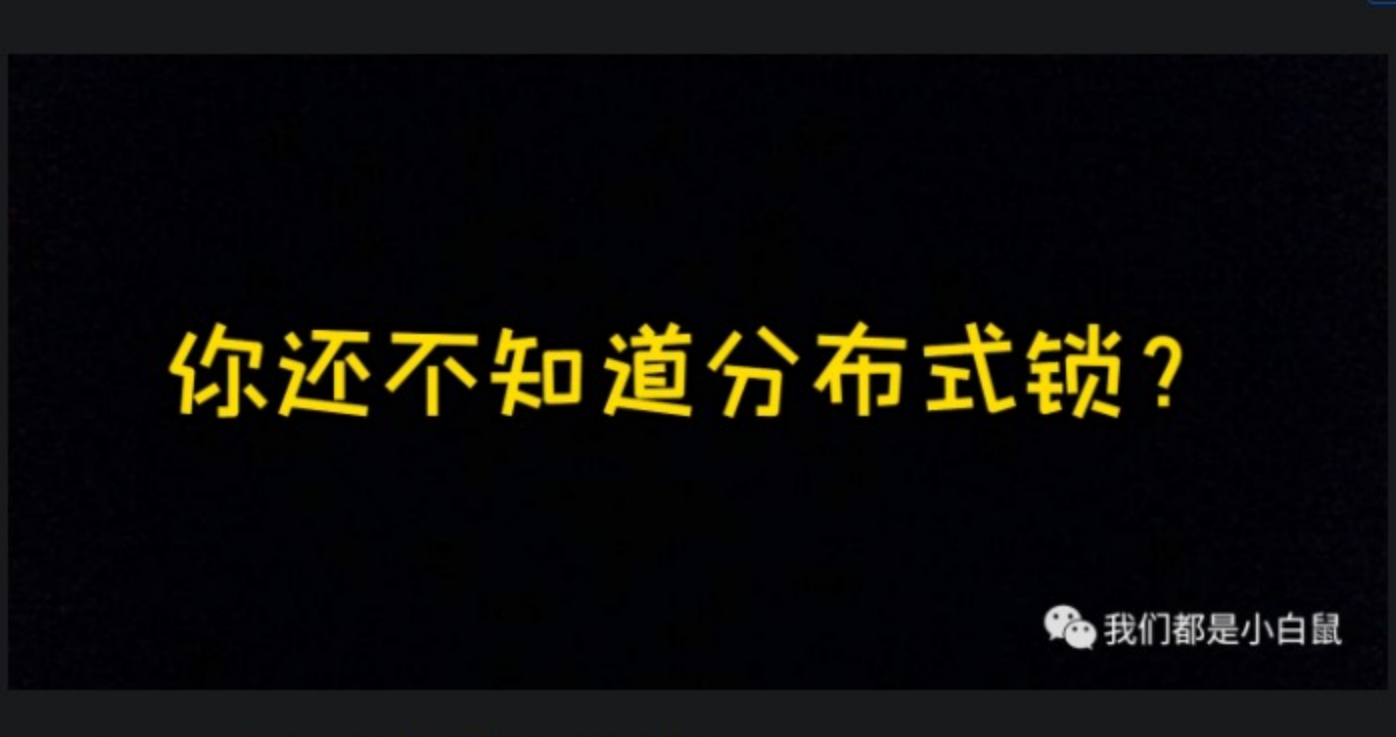


Redisson 实现分布式锁原理分析

我们都是小白鼠
一个普通的程序员

写在前面

在了解分布式锁具体实现方案之前，我们应该先思考一下使用分布式锁必须要考虑的一些问题。

- 互斥性：在任意时刻，只能有一个进程持有锁。
- 防死锁：即使有一个进程在持有锁的期间崩溃而未能主动释放锁，要有其他方式去释放锁从而保证其他进程能获取到锁。
- 加锁和解锁的必须是同一个进程。
- 锁的续期问题。

常见的分布式锁实现方案

- 基于 Redis 实现分布式锁
- 基于 Zookeeper 实现分布式锁

本文采用第一种方案，也就是基于 Redis 的分布式锁实现方案。

Redis 实现分布式锁主要步骤

1. 指定一个 key 作为锁标记，存入 Redis 中，指定一个 **唯一的用户标识** 作为 value。
2. 当 key 不存在时才能设置值，确保同一时间只有一个客户端进程获得锁，满足 **互斥性** 特性。
3. 设置一个过期时间，防止因系统异常导致没能删除这个 key，满足 **防死锁** 特性。
4. 当处理完业务之后需要清除这个 key 来释放锁，清除 key 时需要校验 value 值，需要满足 **只有加锁的人才能释放锁** 。

> 特别注意：以上实现步骤考虑到了使用分布式锁需要考虑的互斥性、防死锁、加锁和解锁必须为同一个进程等问题，但是锁的续期无法实现。所以，博主采用 Redisson 实现 Redis 的分布式锁，借助 Redisson 的 WatchDog 机制 能够很好的解决锁续期的问题，同样 Redisson 也是 Redis 官方推荐分布式锁实现方案，实现起来较为简单。

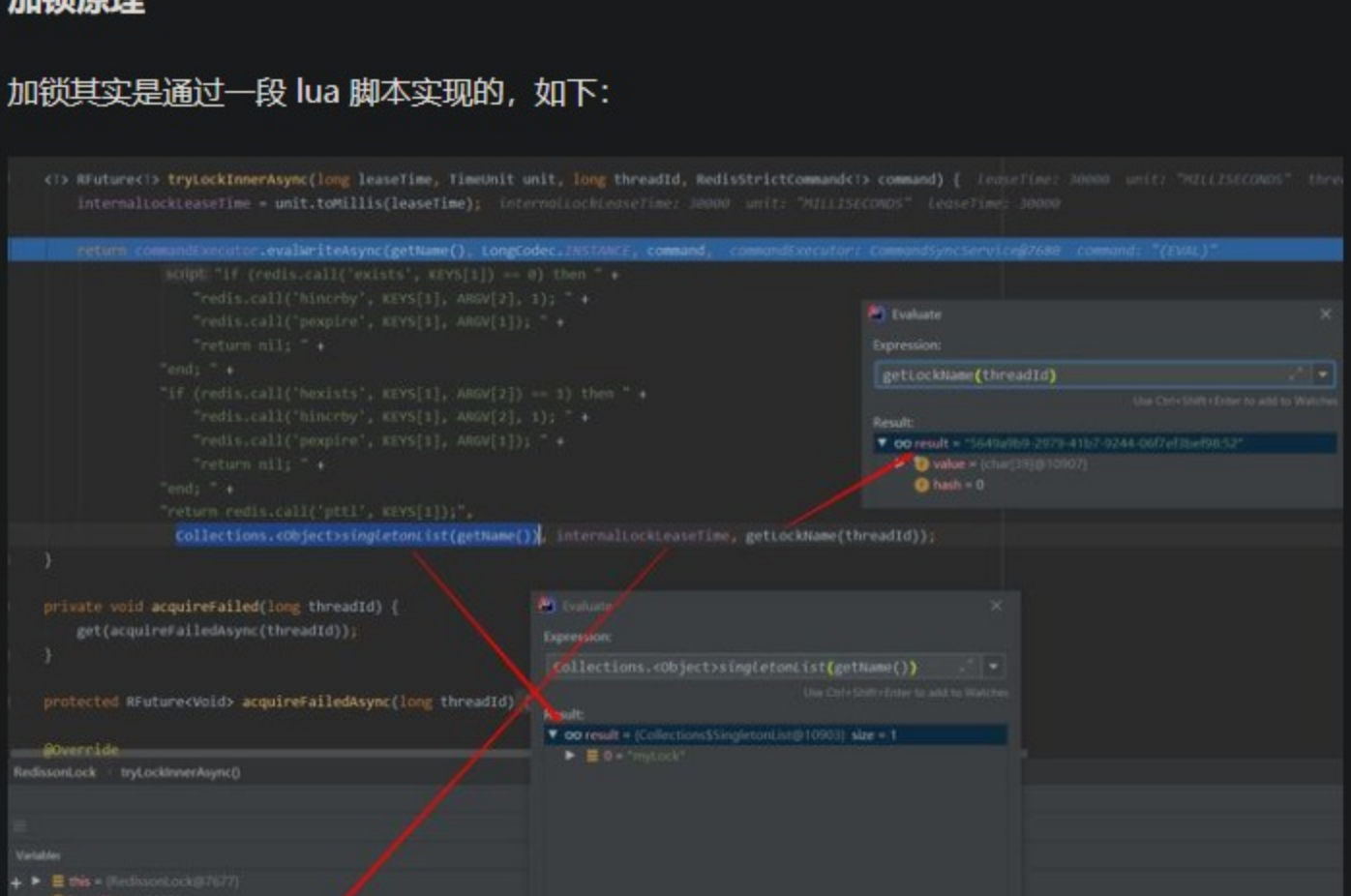
Redisson 实现分布式锁

> 具体实现代码已经上传到博主的仓库，需要的朋友可以在公众号内回复 【分布式锁代码】 获取码云或 GitHub 项目下载地址。

下面从加锁机制、锁互斥机制、Watch dog 机制、可重入加锁机制、锁释放机制，等五个方面对 Redisson 实现分布式锁的底层原理进行分析。

加锁原理

加锁其实是通过一段 lua 脚本实现的，如下：



我们把这一段 lua 脚本抽出来看：

```
if (redis.call('exists', KEYS[1]) == 0) then " +
   "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
   "redis.call('pexpire', KEYS[1], ARGV[1]); " +
   "return nil; " +
   "end; " +
"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
    "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
    "end; " +
"return redis.call('pttl', KEYS[1]);"
```

这里 `KEYS[1]` 代表的是你加锁的 key，比如你自己设置了加锁的那个锁 key 就是 "myLock"。

```
// create a lock
RLock lock = redisson.getLock("myLock");
```

这里 `ARGV[1]` 代表的是锁 key 的默认生存时间，默认 30 秒。 `ARGV[2]` 代表的是加锁的客户端的 ID，类似于下面这样：285475da-9152-4c83-822a-67ee2f116a79:52。至于最后面的一个 1 是为了后面可重入做的计数统计，后面会有讲解到。

我们来看一下在 Redis 中的存储结构：

```
127.0.0.1:6379> HGETALL myLock
1) "285475da-9152-4c83-822a-67ee2f116a79:52"
2) "1"
```

上面这一段加锁的 lua 脚本的作用是：第一段 if 判断语句，就是用 `exists myLock` 命令判断一下，如果你要加锁的那个锁 key 不存在的话，你就进行加锁。如何加锁呢？使用 `hincrby` 命令设置一个 hash 结构，类似于在 Redis 中使用下面的操作：

```
127.0.0.1:6379> HINCRBY myLock 285475da-9152-4c83-822a-67ee2f116a79:52 1
(integer) 1
```

接着会执行 `pexpire myLock 30000` 命令，设置 myLock 这个锁 key 的生存时间是 30 秒。到此为止，加锁完成。

有的小伙伴可能此时就有疑问了，**如果此时有第二个客户端请求加锁呢？** 这就是下面要说的锁互斥机制。

锁互斥机制

此时，如果客户端 2 来尝试加锁，会如何呢？首先，第一个 if 判断会执行 `exists myLock`，发现 myLock 这个锁 key 已经存在。接着第二个 if 判断，判断一下，myLock 锁 key 的 hash 数据结构中，是否包含客户端 2 的 ID，这里明显不是，因为那里包含的是客户端 1 的 ID。所以，客户端 2 会执行：

```
return redis.call('pttl', KEYS[1]);
```

返回的一个数字，这个数字代表了 myLock 这个锁 key 的剩余生存时间。

接下来我们看一下 Redissson tryLock 的主流程：

```
@Override
    public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws Interr
        long time = unit.toMillis(waitTime);
        long current = System.currentTimeMillis();
        long threadId = Thread.currentThread().getId();
        // 1.尝试获取锁
        Long ttl = tryAcquire(leaseTime, unit, threadId);
        // lock acquired
        if (ttl == null) {
            return true;
        }

        // 申请锁的耗时如果大于等于最大等待时间，则申请锁失败.
        time -= System.currentTimeMillis() - current;
        if (time <= 0) {
            acquireFailed(threadId);
            return false;
        }

        current = System.currentTimeMillis();

        /**
         * 2.订阅锁释放事件，并通过 await 方法阻塞等待锁释放，有效的解决了无效的锁申请浪费资源的
         * 基于信息量，当锁被其它资源占用时，当前线程通过 Redis 的 channel 订阅锁的释放事件，一
         *
         * 当 this.await 返回 false，说明等待时间已经超出获取锁最大等待时间，取消订阅并返回获取
         * 当 this.await 返回 true，进入循环尝试获取锁.
         */
        RFuture<RedissonLockEntry> subscribeFuture = subscribe(threadId);
        // await 方法内部是用 CountDownLatch 来实现阻塞，获取 subscribe 异步执行的结果（应用
        if (!subscribeFuture.await(time, TimeUnit.MILLISECONDS)) {
            if (!subscribeFuture.cancel(false)) {
                subscribeFuture.onComplete((res, e) -> {
                    if (e == null) {
                        unsubscribe(subscribeFuture, threadId);
                    }
                });
            }
            acquireFailed(threadId);
            return false;
        }

        try {
            // 计算获取锁的总耗时，如果大于等于最大等待时间，则获取锁失败.
            time -= System.currentTimeMillis() - current;
            if (time <= 0) {
                acquireFailed(threadId);
                return false;

              }

            /**
             * 3.收到锁释放的信号后，在最大等待时间之内，循环一次接着一次的尝试获取锁
             * 获取锁成功，则立马返回 true，
             * 若在最大等待时间之内还没获取到锁，则认为获取锁失败，返回 false 结束循环
             */
            while (true) {
                long currentTime = System.currentTimeMillis();

                // 再次尝试获取锁
                ttl = tryAcquire(leaseTime, unit, threadId);
                // lock acquired
                if (ttl == null) {
                    return true;
                }
                // 超过最大等待时间则返回 false 结束循环，获取锁失败
                time -= System.currentTimeMillis() - currentTime;
                if (time <= 0) {
                    acquireFailed(threadId);
                    return false;
                }

                /**
                 * 6.阻塞等待锁（通过信号量(共享锁)阻塞,等待解锁消息）:
                 */
                currentTime = System.currentTimeMillis();
                if (ttl >= 0 && ttl < time) {
                    //如果剩余时间(ttl)小于wait time ,就在 ttl 时间内，从Entry的信号量获取一
                    getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
                } else {
                    //则就在wait time 时间范围内等待可以通过信号量
                    getEntry(threadId).getLatch().tryAcquire(time, TimeUnit.MILLISECON
                }

                // 更新剩余的等待时间(最大等待时间-已经消耗的阻塞时间)
                time -= System.currentTimeMillis() - currentTime;
                if (time <= 0) {
                    acquireFailed(threadId);
                    return false;
                }
            }
        } finally {
            // 7.无论是否获得锁,都要取消订阅解锁消息
            unsubscribe(subscribeFuture, threadId);
        }
//        return get(tryLockAsync(waitTime, leaseTime, unit));
    }
```

流程分析：

1. 尝试获取锁，返回 null 则说明加锁成功，返回一个数值，则说明已经存在该锁，ttl 为锁的剩余存活时间。
2. 如果此时客户端 2 进程获取锁失败，那么使用客户端 2 的线程 id（其实本质上就是进程 id）通过 Redis 的 channel 订阅锁释放的事件，如果等待的过程中一直未等到锁的释放事件通知，当超过最大等待时间则获取锁失败，返回 false，也就是第 39 行代码。如果等到了锁的释放事件的通知，则开始进入一个不断重试获取锁的循环。
3. 循环中每次都先试着获取锁，并得到已存在的锁的剩余存活时间。如果在重试中拿到了锁，则直接返回。如果锁当前还是被占用的，那么等待释放锁的消息，具体实现使用了 JDK 的信号量 Semaphore 来阻塞线程，当锁释放并发布释放锁的消息后，信号量的 `release()` 方法会被调用，此时被信号量阻塞的等待队列中的一个线程就可以继续尝试获取锁了。

> 特别注意：以上过程存在一个细节，这里有必要说明一下，也是分布式锁的一个关键点：当锁正在被占用时，等待获取锁的进程并不是通过一个 `while(true)` 死循环去获取锁，而是利用了 Redis 的发布订阅机制,通过 await 方法阻塞等待锁的进程，有效的解决了无效的锁申请浪费资源的问题。

锁的续期机制

客户端 1 加锁的锁 key 默认生存时间才 30 秒，如果超过了 30 秒，客户端 1 还想一直持有这把锁，怎么办呢？

Redisson 提供了一个续期机制， 只要客户端 1 一旦加锁成功，就会启动一个 Watch Dog。

```
private <T> RFuture<Long> tryAcquireAsync(long leaseTime, TimeUnit unit, long threadId
    if (leaseTime != -1) {
        return tryLockInnerAsync(leaseTime, unit, threadId, RedisCommands.EVAL_LONG);
    }
    RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(commandExecutor.getConnection
    ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
        if (e != null) {
            return;
        }

        // lock acquired
        if (ttlRemaining == null) {
            scheduleExpirationRenewal(threadId);
        }
    });
    return ttlRemainingFuture;
}
```

> 注意：从以上源码我们看到 `leaseTime` 必须是 -1 才会开启 Watch Dog 机制，也就是如果你想开启 Watch Dog 机制必须使用默认的加锁时间为 30s。如果你自己自定义时间，超过这个时间，锁就会自定释放，并不会延长。

```
private void scheduleExpirationRenewal(long threadId) {
    ExpirationEntry entry = new ExpirationEntry();
    ExpirationEntry oldEntry = EXPIRATION_RENEWAL_MAP.putIfAbsent(getEntryName(), entr
    if (oldEntry != null) {
        oldEntry.addThreadId(threadId);
    } else {
        entry.addThreadId(threadId);
        renewExpiration();
    }
}

protected RFuture<Boolean> renewExpirationAsync(long threadId) {
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands
            "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
                "redis.call('pexpire', KEYS[1], ARGV[1]); " +
                "return 1; " +
            "end; " +
            "return 0;",
        Collections.<Object>singletonList(getName()),
        internalLockLeaseTime, getLockName(threadId));
}
```

Watch Dog 机制其实就是一个后台定时任务线程，获取锁成功之后，会将持有锁的线程放入到一个 `RedissonLock.EXPIRATION_RENEWAL_MAP` 里面，然后每隔 10 秒 （`internalLockLeaseTime / 3`） 检查一下，如果客户端 1 还持有锁 key（判断客户端是否还持有 key，其实就是遍历 `EXPIRATION_RENEWAL_MAP` 里面线程 id 然后根据线程 id 去 Redis 中查，如果存在就会延长 key 的时间），那么就会不断的延长锁 key 的生存时间。

> 注意：这里有一个细节问题，如果服务宕机了，Watch Dog 机制线程也就没有了，此时就不会延长 key 的过期时间，到了 30s 之后就会自动过期了，其他线程就可以获取到锁。

可重入加锁机制

Redisson 也是支持可重入锁的，比如下面这种代码：

```
@Override
public void lock() {
    RLock lock = redissonSingle.getLock("myLock");
    try {
        lock.lock();

        // 执行业务
        doBusiness();

        lock.lock();

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        // 释放锁
        lock.unlock();
        lock.unlock();
        logger.info("任务执行完毕，释放锁!");
    }
}
```

我们再分析一下加锁那段 lua 代码：

```
if (redis.call('exists', KEYS[1]) == 0) then " +
   "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
   "redis.call('pexpire', KEYS[1], ARGV[1]); " +
   "return nil; " +
   "end; " +
"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
    "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
    "end; " +
"return redis.call('pttl', KEYS[1]);"
```

第一个 if 判断肯定不成立， `exists myLock` 会显示锁 key 已经存在。第二个 if 判断会成立，因为 myLock 的 hash 数据结构中包含的那个 ID 即客户端 1 的 ID，此时就会执行可重入加锁的逻辑，使用： `hincrby myLock 285475da-9152-4c83-822a-67ee2f116a79:52 1` 对客户端 1 的加锁次数加 1。此时 myLock 数据结构变为下面这样：

```
127.0.0.1:6379> HGETALL myLock
1) "285475da-9152-4c83-822a-67ee2f116a79:52"
2) "2"
```

到这里，小伙伴本就都明白了 hash 结构的 key 是锁的名称，field 是客户端 ID，value 是该客户端加锁的次数。

这里有一个细节，如果加锁支持可重入锁，那么解锁呢？

释放锁机制

执行

```
lock.unlock()
```

就可以释放分布式锁。我们来看一下释放锁的流程代码：

```
@Override
public RFuture<Void> unlockAsync(long threadId) {
    RPromise<Void> result = new RedissonPromise<Void>();
    // 1. 异步释放锁
    RFuture<Boolean> future = unlockInnerAsync(threadId);
    // 取消 Watch Dog 机制
    future.onComplete((opStatus, e) -> {
        cancelExpirationRenewal(threadId);

        if (e != null) {
            result.tryFailure(e);
            return;
        }

        if (opStatus == null) {
            IllegalMonitorStateException cause = new IllegalMonitorStateException("att
                    + id + " thread-id: " + threadId);
            result.tryFailure(cause);
            return;
        }

        result.trySuccess(null);
    });

    return result;
}

protected RFuture<Boolean> unlockInnerAsync(long threadId) {
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands
            // 判断锁 key 是否存在
            "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " +
                "return nil;" +
            "end; " +
            // 将该客户端对应的锁的 hash 结构的 value 值递减为 0 后再进行删除
            // 然后再向通道名为 redisson_lock__channel publish 一条 UNLOCK_MESSAGE 信息
            "local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1); " +
            "if (counter > 0) then " +
                "redis.call('pexpire', KEYS[1], ARGV[2]); " +
                "return 0; " +
            "else " +
                "redis.call('del', KEYS[1]); " +
                "redis.call('publish', KEYS[2], ARGV[1]); " +
                "return 1; "+
            "end; " +
            "return nil;",
            Arrays.<Object>asList(getName(), getChannelName()), LockPubSub.UNLOCK_MESS
}
```

从以上代码来看，释放锁的步骤主要分三步：

1. 删除锁（这里注意可重入锁，在上面的脚本中有详细分析）。
2. 广播释放锁的消息，通知阻塞等待的进程（向通道名为 `redisson_lock__channel` publish 一条 `UNLOCK_MESSAGE` 信息）。
3. 取消 Watch Dog 机制，即将 `RedissonLock.EXPIRATION_RENEWAL_MAP` 里面的线程 id 删除，并且 cancel 掉 Netty 的那个定时任务线程。

方案优点

1. Redisson 通过 Watch Dog 机制很好的解决了锁的续期问题。
2. 和 Zookeeper 相比较，Redisson 基于 Redis 性能更高，适合对性能要求高的场景。
3. 通过 Redisson 实现分布式可重入锁，比原生的 `SET mylock userId NX PX milliseconds` + lua 实现的效果更好些，虽然基本原理都一样，但是它帮我们屏蔽了内部的执行细节。
4. 在等待申请锁资源的进程等待申请锁的实现上也做了一些优化，减少了无效的锁申请，提升了资源的利用率。

方案缺点

1. 使用 Redisson 实现分布式锁方案最大的问题就是如果你对某个 Redis Master 实例完成了加锁，此时 Master 会异步复制给其对应的 slave 实例。但是这个过程中一旦 Master 宕机，主备切换，slave 变为了 Master。接着就会导致，客户端 2 来尝试加锁的时候，在新的 Master 上完成了加锁，而客户端 1 也以为自己成功加了锁，此时就会导致多个客户端对一个分布式锁完成了加锁，这时系统在业务语义上一定会出现问题，导致各种脏数据的产生。所以这个就是 Redis Cluster 或者说是 Redis Master-Slave 架构的主从异步复制导致的 Redis 分布式锁的最大缺陷（在 Redis Master 实例宕机的时候，可能导致多个客户端同时完成加锁）。
2. 有个别观点说使用 Watch Dog 机制开启一个定时线程去不断延长锁的时间对系统有所损耗（这里只是网络上的一种说法，博主查了很多资料并且结合实际生产并不认为有很大系统损耗，这个仅供大家参考）。

总结

以上就是基于 Redis 使用 Redisson 实现分布式锁的所有原理分析，希望可以帮助小伙伴们对分布式锁的理解有所加深。其实分析完源码后发现基于 Redis 自己手动实现一个简版的分布式锁工具也并不是很难，有兴趣的小伙伴可以试试。

参考

- juejin.im/post/5e828328...
- juejin.im/post/5e1977dd...
- blog.csdn.net/tianyalei...
- cnblogs.com/qdhxhz/p/11...

最后

欢迎大家关注我的关注我的公众号一起探讨研究技术。

编辑于 2020-05-19