一个Redis分布式锁的实现引发的思考

最近看了一个老项目（2018年的），发现其中用 Redis 来实现分布式锁。

代码如下

```
// jedis

public String lock(String lockName, long acquireTimeout) {
    return lockWithTimeout(lockName, acquireTimeout,
DEFAULT_EXPIRE);
}

public String lockWithTimeout(String lockName, long acquireTimeout,
long timeout) {

        RedisConnectionFactory connectionFactory =
redisTemplate.getConnectionFactory();
        RedisConnection redisConnection =
connectionFactory.getConnection();
        /** 随机生成一个value */
        String identifier = UUID.randomUUID().toString();
        String lockKey = LOCK_PREFIX + lockName;
        int lockExpire = (int) (timeout / 1000);

        long end = System.currentTimeMillis() + acquireTimeout;    /** 获取
锁的超时时间，超过这个时间则放弃获取锁 */
        while (System.currentTimeMillis() < end) {
            if (redisConnection.setNX(lockKey.getBytes(),
identifier.getBytes())) {
                redisConnection.expire(lockKey.getBytes(), lockExpire);
                /** 获取锁成功，返回标识锁的value值，用于释放锁确认 */
                RedisConnectionUtils.releaseConnection(redisConnection,
connectionFactory);
                return identifier;
            }
            /** 返回-1代表key没有设置超时时间，为key设置一个超时时间 */
            if (redisConnection.ttl(lockKey.getBytes()) == -1) {
                redisConnection.expire(lockKey.getBytes(), lockExpire);
            }
```

```
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                log.warn("获取分布式锁：线程中断！");
                Thread.currentThread().interrupt();
            }
        }
        RedisConnectionUtils.releaseConnection(redisConnection,
connectionFactory);
        return null;
    }

public boolean releaseLock(String lockName, String identifier) {

    if (StringUtils.isEmpty(identifier)) return false;

    RedisConnectionFactory connectionFactory =
redisTemplate.getConnectionFactory();
    RedisConnection redisConnection =
connectionFactory.getConnection();
    String lockKey = LOCK_PREFIX + lockName;
    boolean releaseFlag = false;
    while (true) {
        try {
            byte[] valueBytes = redisConnection.get(lockKey.getBytes());

            /**  value为空表示锁不存在或已经被释放*/
            if (valueBytes == null) {
                releaseFlag = false;
                break;
            }

            /** 通过前面返回的value值判断是不是该锁，若是该锁，则删除，释
放锁 */
            String identifierValue = new String(valueBytes);
            if (identifier.equals(identifierValue)) {
                redisConnection.del(lockKey.getBytes());
                releaseFlag = true;
            }
            break;
        } catch (Exception e) {
            log.warn("释放锁异常", e);
        }
    }
    RedisConnectionUtils.releaseConnection(redisConnection,
connectionFactory);
    return releaseFlag;
}
```

```
public void lockTest(String lockName, Long acquireTimeout,
CouponSummary couponSummary)  {

    String lockIdentify = redisLock.lock(lockName,acquireTimeout);
    if (StringUtils.isNotEmpty(lockIdentify)){
        // 业务代码
        redisLock.releaseLock(lockName, lockIdentify);
    }
    else{
        System.out.println("get lock failed.");
    }

}
```

分析

看完之后，有这几点感悟

1. setNX 和 expire 两个操作是分开的，有一定的风险（忘了释放锁，expire 失败）
2. 加锁时，除了 setNX ，还会去 ttl ，防止死锁的发生。
3. 释放锁时，会通过 UUID 去判断这个锁的值，避免释放其他线程加的锁，但是没有考虑到这个 get 和 del 是两个操作，还是会有意外，比如 releaseLock 时，执行完 get ，判断这个 uuid 是自己的，准备删除，但此时 锁过期 了，其他线程刚好加锁成功，结果又被你删除了。
4. 释放锁时没有在 finally 块中执行
5. 获取不到锁时，尝试自旋等待锁

再结合 redisson 框架来看的话，就会发现

1. 少了 **自动续期** 的功能，如果业务执行时间较长，锁过期释放掉了，就可能出现并发问题。
2. 少了 **可重入锁** 的功能，可以预见获取锁的线程，再次去加锁也会失败。
3. 少了 **lua脚本** ，lua 脚本能保证原子性操作，减少这个网络开销。

再把视角移到 Redis 服务器来，就会发现 **单点问题** 的存在，此时分布式锁就无法使用了。

这个问题可以通过 **主从，哨兵，集群** 模式解决，但是又有了一个 **故障转移问题** 。

先简要介绍下这几个模式

1. **Redis 主从复制模式：**
* 一主多从，主节点负责写，并同步到从节点。
* 从节点负责备份数据，处理读操作，提供读负载均衡和故障切换。
2. **Redis 哨兵模式：**
* 主从基础上增加了哨兵节点（Sentinel），一个独立进程，去监控所有节点，当主节点宕机时，会从 slave 中选举出新的主节点，并通知其他从节点更新配置
* 哨兵节点负责执行故障转移、选举新的主节点等操作
3. **Redis 集群模式：**
* 多个主从组成，由 master 去瓜分 16384 个 slot， 将数据分片存储在多个节点上。
* 节点间通过 Gossip 协议进行广播通信，比如 新节点的加入，主从变更等

回到 **分布式锁** 这个话题，通过主从切换，可以实现故障转移。但是当加锁成功时，master 挂了，此时还没同步锁信息到这个 slave 上，那这个分布式锁也是失效了。

网上的方案是通过 **Redlock（红锁）** 来解决。

Redlock 的大致意思就是给多个节点加锁，超过半数成功的话，就认为加锁成功。

redisson 的红锁用法

```
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();

```

我更偏向于解决这个 **主从复制延迟** 的问题，比如

* 升级硬件，更好的 CPU，带宽
* 避免从节点阻塞，比如操作一些 大Key
* 调大 repl\_backlog\_size 参数，避免全量同步

当然，具体问题具体分析，可以根据业务准备补偿措施，但也要避免这个过度设计。

### 红锁争论

在查阅资料时，看到了这么一个事情

《数据密集型应用系统设计》的作者 **Martin** 去反驳这个 **Redlock**，并用一个进程暂停（GC）的例子，指出了 Redlock 安全性问题：

> 1. 客户端 1 请求锁定节点 A、B、C、D、E
> 2. 客户端 1 的拿到锁后，进入 GC（时间比较久）
> 3. 所有 Redis 节点上的锁都过期了
> 4. 客户端 2 获取到了 A、B、C、D、E 上的锁
> 5. 客户端 1 GC 结束，认为成功获取锁
> 6. 客户端 2 也认为获取到了锁，发生「冲突」




还有 时钟 漂移的问题

这里我就不过多 CV 了，可以看看原文
```

相关文章

《一文讲透Redis分布式锁安全问题》：[cloud.tencent.com/developer/a…](http://cxyroad.com/”https://cloud.tencent.com/developer/article/2332108”)

《How to do distributed locking》[martin.kleppmann.com/2016/02/08/…](http://cxyroad.com/”https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html”)

NPC 异常场景

* N：Network Delay，网络延迟
* P：Process Pause，进程暂停（GC）
* C：Clock Drift，时钟漂移

感触

看到这个 18年 的项目，不自觉想起在校的时光，那会还很多不懂，也不会搜索，获取到的信息非常有限。

现在就不一样了，像以前那样搜搜都能看到很多图文并茂的文章，甚至有 AI 出来解答，各种社区文档都非常丰富，但也多了一个麻烦，更需要去验证这个信息的真伪了。

结尾

下文再来贴下这个 Redisson 的代码，看看它的思路。

> 本文就到这里啦，感谢您的阅读，有不对的地方也请您帮忙指正！谢谢~
>
>
> 喜欢的小伙伴们，别忘了点赞呀~ 祝你有个美好的一天！
>
>
> [github.com/Java4ye](http://cxyroad.com/

”https://github.com/Java4ye”)

原文链接: https://juejin.cn/post/7386532770477195273