这篇文章主要是对 Redis 官方网站刊登的 Distributed locks with Redis 部分内容的总结和翻译。

什么是 RedLock

Redis 官方站这篇文章提出了一种权威的基于 Redis 实现分布式锁的方式名叫 *Redlock*，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
2. 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务

怎么在单节点上实现分布式锁

> SET resource_name my_random_value NX PX 30000

主要依靠上述命令，该命令仅当 Key 不存在时（NX保证）set 值，并且设置过期时间 3000ms （PX保证），值 my_random_value 必须是所有 client 和所有锁请求发生期间唯一的，释放锁的逻辑是：

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

上述实现可以避免释放另一个client创建的锁，如果只有 del 命令的话，那么如果 client1 拿到 lock1 之后因为某些操作阻塞了很长时间，此时 Redis 端 lock1 已经过期了并且已经被重新分配给了 client2，那么 client1 此时再去释放这把锁就会造成 client2 原本获取到的锁被 client1 无故释放了，但现在为每个 client 分配一个 unique 的 string 值可以避免这个问题。至于如何去生成这个 unique string，方法很多随意选择一种就行了。

Redlock 算法

算法很易懂，起 5 个 master 节点，分布在不同的机房尽量保证可用性。为了获得锁，client 会进行如下操作：

1. 得到当前的时间，微秒单位
2. 尝试顺序地在 5 个实例上申请锁，当然需要使用相同的 key 和 random value，这里一个 client 需要合理设置与 master 节点沟通的 timeout 大小，避免长时间和一个 fail 了的节点浪费时间
3. 当 client 在大于等于 3 个 master 上成功申请到锁的时候，且它会计算申请锁消耗了多少时间，这部分消耗的时间采用获得锁的当下时间减去第一步获得的时间戳得到，如果锁的持续时长（lock validity time）比流逝的时间多的话，那么锁就真正获取到了。
4. 如果锁申请到了，那么锁真正的 lock validity time 应该是 origin（lock validity time） - 申请锁期间流逝的时间
5. 如果 client 申请锁失败了，那么它就会在少部分申请成功锁的 master 节点上执行释放锁的操作，重置状态

失败重试

如果一个 client 申请锁失败了，那么它需要稍等一会在重试避免多个 client 同时申请锁的情况，最好的情况是一个 client 需要几乎同时向 5 个 master 发起锁申请。另外就是如果 client 申请锁失败了它需要尽快在它曾经申请到锁的 master 上执行 unlock 操作，便于其他 client 获得这把锁，避免这些锁过期造成的时间浪费，当然如果这时候网络分区使得 client 无法联系上这些 master，那么这种浪费就是不得不付出的代价了。

放锁

放锁操作很简单，就是依次释放所有节点上的锁就行了

性能、崩溃恢复和 fsync

如果我们的节点没有持久化机制，client 从 5 个 master 中的 3 个处获得了锁，然后其中一个重启了，这是注意**整个环境中又出现了 3 个 master 可供另一个 client 申请同一把锁！** 违反了互斥性。如果我们开启了 AOF 持久化那么情况会稍微好转一些，因为 Redis 的过期机制是语义层面实现的，所以在 server 挂了的时候时间依旧在流逝，重启之后锁状态不会受到污染。但是考虑断电之后呢，AOF部分命令没来得及刷回磁盘直接丢失了，除非我们配置刷回策略为 fsnyc = always，但这会损伤性能。解决这个问题的方法是，当一个节点重启之后，我们规定在 max TTL 期间它是不可用的，这样它就不会干扰原本已经申请到的锁，等到它 crash 前的那部分锁都过期了，环境不存在历史锁了，那么再把这个节点加进来正常工作。