**[百度社招面试题——如何用Redis实现分布式锁](https://segmentfault.com/a/1190000018826044)**

**1.单机Redis实现分布式锁**

**1.1获取锁**

获取锁的过程很简单，客户端向Redis发送命令：

**SET** resource\_name my\_random\_value NX PX 30000

my\_random\_value是由客户端生成的一个随机字符串，它要保证在足够长的一段时间内在所有客户端的所有获取锁的请求中都是唯一的。  
NX表示只有当resource\_name对应的key值不存在的时候才能SET成功。这保证了只有第一个请求的客户端才能获得锁，而其它客户端在锁被释放之前都无法获得锁。  
PX 30000表示这个锁有一个30秒的自动过期时间。

**1.2 释放锁**

**if** redis.**call**("get",KEYS[1]) == ARGV[1] **then**

**return** redis.**call**("del",KEYS[1])

**else**

**return** 0

**end**

之前获取锁的时候生成的my\_random\_value 作为参数传到Lua脚本里面，作为：ARGV[1],而 resource\_name作为KEYS[1]。Lua脚本可以保证操作的原子性。

**1.3 关于单点Redis实现分布式锁的讨论**

网络上有文章说用如下命令获取锁:

SETNX resource\_name my\_random\_**value**

EXPIRE resource\_name 30

由于这两个命令不是原子的。如果客户端在执行完SETNX后crash了，那么就没有机会执行EXPIRE了，导致它一直持有这个锁，其他的客户端就永远获取不到这个锁了。

* 为什么my\_random\_value 要设置成随机值?

保证了一个客户端释放的锁是自己持有的那个锁。如若不然，可能出现锁不安全的情况。

客户端1获取锁成功。

客户端1在某个操作上阻塞了很长时间。

过期时间到了，锁自动释放了。

客户端2获取到了对应同一个资源的锁。

客户端1从阻塞中恢复过来，释放掉了客户端2持有的锁。

* 用 SETNX获取锁

网上大量文章说用如下命令获取锁：

SETNX lock.foo <**current** Unix **time** + **lock** **timeout** + 1>

原文在Redis对SETNX的官网说明，Redis官网文档建议用Set命令来代替，主要原因是SETNX不支持超时时间的设置。

<https://redis.io/commands/setnx>

**2.Redis集群实现分布式锁**

上面的讨论中我们有一个非常重要的假设：Redis是单点的。如果Redis是集群模式，我们考虑如下场景:

客户端1从Master获取了锁。

Master宕机了，存储锁的key还没有来得及同步到Slave上。

Slave升级为Master。

客户端2从新的Master获取到了对应同一个资源的锁。

客户端1和客户端2同时持有了同一个资源的锁，锁不再具有安全性。

就此问题，Redis作者antirez写了RedLock算法来解决这种问题。

**2.1 RedLock获取锁**

* 获取当前时间。
* 按顺序依次向N个Redis节点执行获取锁的操作。这个获取操作跟前面基于单Redis节点的获取锁的过程相同，包含随机字符串my\_random\_value，也包含过期时间(比如PX 30000，即锁的有效时间)。为了保证在某个Redis节点不可用的时候算法能够继续运行，这个获取锁的操作还有一个超时时间(time out)，它要远小于锁的有效时间（几十毫秒量级）。客户端在向某个Redis节点获取锁失败以后，应该立即尝试下一个Redis节点。
* 计算整个获取锁的过程总共消耗了多长时间，计算方法是用当前时间减去第1步记录的时间。如果客户端从大多数Redis节点（>= N/2+1）成功获取到了锁，并且获取锁总共消耗的时间没有超过锁的有效时间(lock validity time)，那么这时客户端才认为最终获取锁成功；否则，认为最终获取锁失败。
* 如果最终获取锁成功了，那么这个锁的有效时间应该重新计算，它等于最初的锁的有效时间减去第3步计算出来的获取锁消耗的时间。
* 如果最终获取锁失败了（可能由于获取到锁的Redis节点个数少于N/2+1，或者整个获取锁的过程消耗的时间超过了锁的最初有效时间），那么客户端应该立即向所有Redis节点发起释放锁的操作（即前面介绍的单机Redis Lua脚本释放锁的方法）。

**2.2 RedLock释放锁**

客户端向所有Redis节点发起释放锁的操作，不管这些节点当时在获取锁的时候成功与否。

**2.3 关于RedLock的问题讨论**

* 如果有节点发生崩溃重启

假设一共有5个Redis节点：A, B, C, D, E。设想发生了如下的事件序列：

客户端1成功锁住了A, B, **C**，获取锁成功（但**D**和**E**没有锁住）。

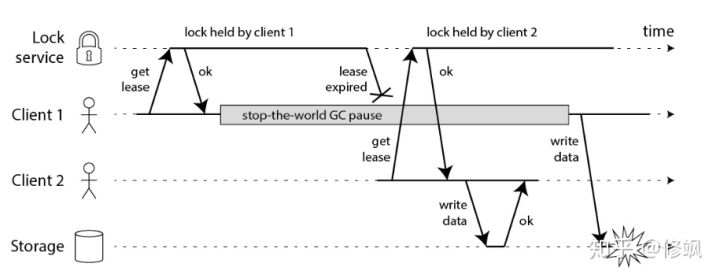
节点**C**崩溃重启了，但客户端1在**C**上加的锁没有持久化下来，丢失了。

节点**C**重启后，客户端2锁住了**C**, **D**, **E**，获取锁成功。

客户端1和客户端2同时获得了锁。

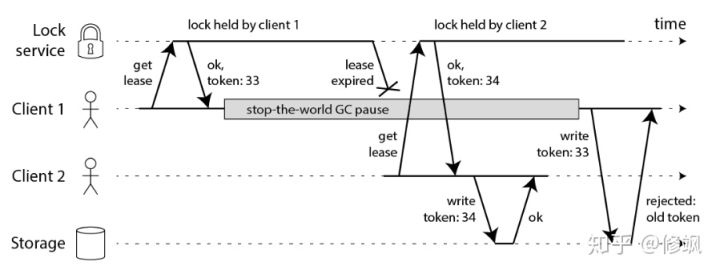
为了应对这一问题，antirez又提出了延迟重启(delayed restarts)的概念。也就是说，一个节点崩溃后，先不立即重启它，而是等待一段时间再重启，这段时间应该大于锁的有效时间(lock validity time)。这样的话，这个节点在重启前所参与的锁都会过期，它在重启后就不会对现有的锁造成影响。

* 如果客户端长期阻塞导致锁过期



解释一下这个时序图，客户端1在获得锁之后发生了很长时间的GC pause，在此期间，它获得的锁过期了，而客户端2获得了锁。当客户端1从GC pause中恢复过来的时候，它不知道自己持有的锁已经过期了，它依然向共享资源（上图中是一个存储服务）发起了写数据请求，而这时锁实际上被客户端2持有，因此两个客户端的写请求就有可能冲突（锁的互斥作用失效了）。

如何解决这个问题呢?引入了fencing token的概念：



客户端1先获取到的锁，因此有一个较小的fencing token，等于33，而客户端2后获取到的锁，有一个较大的fencing token，等于34。客户端1从GC pause中恢复过来之后，依然是向存储服务发送访问请求，但是带了fencing token = 33。存储服务发现它之前已经处理过34的请求，所以会拒绝掉这次33的请求。这样就避免了冲突。

但是其实这已经超出了Redis实现分布式锁的范围，单纯用Redis没有命令来实现生成Token。

* 时钟跳跃问题

假设有5个Redis节点A, B, C, D, E。

客户端1从Redis节点A, B, **C**成功获取了锁（多数节点）。由于网络问题，与**D**和**E**通信失败。

节点**C**上的时钟发生了向前跳跃，导致它上面维护的锁快速过期。

客户端2从Redis节点**C**, **D**, **E**成功获取了同一个资源的锁（多数节点）。

客户端1和客户端2现在都认为自己持有了锁。

这个问题用Redis实现分布式锁暂时无解。而生产环境这种情况是存在的。

结论  
Redis并不能实现严格意义上的分布式锁。但是这并不意味着上面讨论的方案一无是处。如果你的应用场景为了效率(efficiency)，协调各个客户端避免做重复的工作，即使锁失效了，只是可能把某些操作多做一遍而已，不会产生其它的不良后果。但是如果你的应用场景是为了正确性(correctness)，那么用Redis实现分布式锁并不合适，会存在各种各样的问题，且解决起来就很复杂，为了正确性，需要使用zab、raft共识算法，或者使用带有事务的数据库来实现严格意义上的分布式锁。