redis实现分布式锁

分布式（互斥）锁介绍

在很多互联网产品应用中，有些场景需要加锁处理，比如：秒杀、全局递增ID、楼层生成，还有一些分布式任务调度等等。大部分的解决方案是基于DB实现的，Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系。其次，Redis提供一些命令SETNX、GETSET，可以方便实现分布式锁机制。

完美的解决方案（并非真的完美）

基于set nx ex实现

jedisCluster语法：

jedisCluster.set(key, “本机ID”， “nx”, “ex”, 3);

方法声明：

set(final String key, final String value, final String nxxx, final String expx, final long time)

这是redis官网提出的解决方案：Distributed locks with Redis

现在就来看看redis官方推荐的方案，官网将这种算法称为Redlock

安全和可靠性保证

在描述完美的设计之前，我们想先提出三个属性，这三个属性在我们看来，是实现高效分布式锁的基础。

* 安全属性：互斥，不管任何时候，只有一个客户端能持有同一个锁。
* 效率属性A：不会死锁，最终一定会得到锁，就算一个持有锁的客户端宕掉或者发生网络分区。
* 效率属性B：容错，只要大多数Redis节点正常工作，客户端应该都能获取和释放锁。

为什么基于故障切换的方案不够好

为了理解我们想要优化的到底是什么，我们先看下当前大多数基于Redis的分布式锁三方库的现状。

用Redis来实现分布式锁最简单的方式就是在实例里创建一个键值，创建出来的键值一般都是有一个客户端想要释放锁时，它只需要删除这个键值即可。表面来看，这个方法似乎很管用，但是这里存在一个问题：在我们的系统架构里存在一个单点故障，如果Redis的master节点宕机了怎么办呢？有人可能会说：加一个slave节点，在master宕机时用slave就行了！但是其实这个方案明显是不可行的，因为这种方案无法保证第1个安全互斥属性，因为Redis的复制是异步的。总的来说，这个方案里有一个明显的竞争条件（race condition），举例来说：

1. 客户端A在master节点拿到了锁。
2. master节点在把A创建的key写入slave之前宕机了。
3. slave变成了master节点
4. B也得到了和A还持有的相同的锁（因为原来的slave里还没有A持有锁的信息）

当然，在某些特殊场景下，前面提到的这个方案则完全没有问题，比如在宕机期间，多个客户端允许同时都持有锁，如果你可以容忍这个问题的话，那用这个基于复制的方案就完全没有问题，否则的话我们还是建议你采用这篇文章里接下来要描述的方案。

采用单实例的正确实现

在讲述如何用其它方案突破单实例方案的限制之前，让我们先看下是否有什么办法可以修复这个简单场景的问题，因为这个方案其实如果可以忍受竞争条件的话是有望可行的，而且但实例来实现分布式锁是我们后面要讲的算法的基础。要获得锁，要用下面这个命令：

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

这个命令的作用是在只有这个key不存在的时候才会设置这个key的值（NX选项的作用），超时时间设为30000毫秒（PX选项的作用），这个key的值设为”my\_random\_value”。这个值必须在所有获取锁请求的客户端里保持唯一。基本上这个随机值就是用来保证能安全地释放锁，我们可以用下面这个Lua脚本来告诉Redis: 删除这个key当且仅当这个key存在而且值是我期望的那个值。

if redis.call(“get”, KEYS[1]) == ARGV[1] then

return redis.call(“del”, KEYS[1])

else

return 0

end

[ 使用lua脚本可以使得原本非原子操作变成原子的 ]

这个很重要，因为这可以避免误删其他客户端得到的锁，举个例子，一个客户端拿到了锁，被某个操作阻塞了很长时间，过了超时时间后自动释放了这个锁，然后这个客户端之后又尝试删除这个其实已经被其他客户端拿到的锁。所以单纯的用DEL指令有可能造成一个客户端删除了其它客户端的锁，用上面这个脚本可以保证每个客户端都用一个随机字符串“签名”了，这样每个锁就只能被获得锁的客户端删除了。