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。

这个随机字符串应该用什么生成呢？我假设这是从/dev/urandom生成的20字节大小的字符串，但是其实你可以有效率更高的方案来保证这个字符串足够唯一。比如你可以用RC4加密算法来从/dev/urandom生成一个伪随机流。还有更简单的方案，比如用毫秒的unix时间戳加上客户端id, 这个也许不够安全，但是也许在大多数环境下已经够用了。

【锁持有者在处理完工作以后要主动释放锁；另外锁超时时间保证了在锁未被正常释放的时候不会产生死锁】

key值的超时时间，也叫做“锁有效时间”。这个是锁的自动释放时间，也是一个客户端在其他客户端能抢占锁之前可以执行任务的时间，这个时间从获取锁的时间点开始计算。所以现在我们有很好的获取和释放锁的方式，在一个非分布式的、单点的、保证永不宕机的环境下这个方式没有任何问题，接下来我们看看无法保证这些条件的分布式环境下我们该怎么做。

Redlock算法

在分布式版本的算法里我们假设有N个Redis master节点，这些节点都是完全独立的，我们不用任何复制或者其他隐含的分布式协调算法。我们已经描述了如何在单节点环境下安全地获取和释放锁。因此我们理所当然地应当用这个方法在每个单节点里来获取和释放锁。在我们的例子里面我们把N设成5，这个数字是一个相对比较合理的数值，因此我们需要在不同的计算机或者虚拟机上运行5个master节点来保证他们大多数情况下都不会同时宕机。一个客户端需要做如下操作来获取锁：

1. 获取当前时间（单位是毫秒）。
2. 轮流用相同的key和随机值在N个节点上请求锁，在这一步里，客户端在每个master上请求锁时，要求redis超时时间比锁的释放时间小的多。比如如果锁自动释放时间是10秒钟，那每个节点锁请求的超时时间可能是5-50毫秒的范围，这个可以防止一个客户端在某个宕掉的master节点上阻塞过长时间，如果一个master节点不可用了，我们应该尽快尝试下一个master节点。
3. 客户端计算第二步中获取锁所花的时间，只有当客户端在大多数master节点上成功获取了锁（在这里是3个），而且总共消耗的时间不超过锁释放时间，这个锁就认为是获取成功了。
4. 如果锁获取成功了，那现在锁自动释放时间就是最初的锁释放时间减去之前获取锁所消耗的时间。
5. 如果锁获取失败了，不管是因为获取成功的锁不超过一半（N/2+1）还是因为总消耗时间超过了锁释放时间，客户端都会到每个master节点上释放锁，即便是那些他认为没有获取成功的锁。

这个算法是否是异步的？

这个算法是基于一个假设：虽然不存在可以跨进程的同步时钟，但是不同进程时间都是以差不多相同的速度前进，这个假设不一定完全准确，但是和自动释放锁的时间长度相比不同进程时间前进速度差异基本是可以忽略不计的。这个假设就好比真实世界里的计算机：每个计算机都有本地时钟，但是我们可以说大部分情况下不同计算机之间的时间差是很小的。现在我们需要更细化我们的锁互斥规则，只有当客户端能在T时间内完成所做的工作才能保证锁是有效的（详见算法的第3步），T的计算规则是锁失效时间T1减去一个用来补偿不同进程间时钟差异的delta值（一般只有几毫秒而已）。

失败的重试

当一个客户端获取锁失败时，这个客户端应该在一个随机延时后进行重试，之所以采用随机延时是为了避免不同客户端同时重试导致谁都无法拿到锁的情况出现。同样的道理，客户端越快尝试在大多数Redis节点获取锁，出现多个客户端同时竞争锁和重试的时间窗口越小，可能性就越低，所以最完美的情况下，客户端应该用多路传输的方式同时向所有Redis节点发送SET命令。这里非常有必要强调一下，客户端如果没有在多数节点获取到锁，一定要尽快在获取锁成功的节点上释放锁，这样就没必要等到key超时后才能重新获取这个锁（但是如果网络分区的情况发生而且客户端无法连接到Redis节点时，会损失等待key超时这段时间的系统可用性）。

释放锁

释放锁比较简单，因为只需要在所有节点都释放锁就行，不管之前有没有在该节点获取锁成功。

安全性的论证

这个算法到底是不是安全的呢？我们可以观察不同场景下的情况来理解这个算法为什么是安全的。开始之前，让我们假设客户端可以在大多数节点都获取到锁，这样所有的节点都会包含一个有相同存活时间的key。但是需要注意的是，这个key是在不同时间点设置的，所以这些key也会在不同的时间超时，但是我们假设最坏情况下第一个key是在T1时间设置的（客户端连接到第一个服务器时的时间），最后一个key是在T2时间设置的（客户端收到最后一个服务器返回结果的时间），从T2时间开始，我们可以确认最早超时的key至少也会存在的时间为MIN\_VALIDITY=TTL-(T2-T1) – CLOCK\_DRIFT, TTL是锁超时时间、(T2 – T1)是最晚获取到的锁的耗时，CLOCK\_DRIFT是不同进程间时钟差异，这个是用来补偿前面的( T2-T1 )。其他的key都会在这个时间点之后才会超时，所以我们可以确定这些key在这个时间点之前至少都是同时存在的。

在大多数节点的key都set了的时间段内，其他客户端无法抢占这个锁，因为在N/2+1个客户端的key已经存在的情况下不可能再在N/2+1个客户端上获取锁成功，所以如果一个锁获取成功了，就不可能同时重新获取这个锁成功（不然就违反了分布式锁互斥原则），然后我们也要确保多个客户端同时尝试获取锁时不会都同时成功。如果一个客户端获取大多数节点锁的耗时接近甚至超过锁的最大有效时间时（就是我们为SET操作设置的TTL值），那么系统会认为这个锁是无效的，同时会释放这些节点上的锁，所以我们仅仅需要考虑获取大多数节点锁的耗时小于有效时间的情况。在这种情况下，根据我们前面的证明，在MIN\_VALIDITY时间内，没有客户端能重新获取锁成功，所以，多个客户端都能同时成功获取锁的结果，只会发生在多数节点获取锁的时间都大大超过TTL时间的情况下，实际上这种情况下这些锁都会失效。我们非常期待和欢迎有人能提供这个算法安全性的公式化证明，或者发现任何bug。

性能论证

这个系统的性能主要基于以下三个主要特征：

1. 锁自动释放的特征，一定时间后某个锁都能被再次获取。
2. 客户端通常会在不再需要锁或者任务执行完成之后主动释放锁，这样我们就不用等到超时时间后再去获取这个锁。
3. 当一个客户端需要重试获取锁时，这个客户端会等待一段时间，等待的时间相对来说会比我们重新获取大多数锁的时间要长一些，这样可以降低不同客户端竞争锁资源时发生死锁的概率。

然而，我们在网络分区时要损失TTL的可用性时间，所以如果网络分区持续发生，这个不可用会一直持续。这种情况在每次一个客户端获取到了锁并在释放锁之前被网络分区了时都会出现。

基本来说，如果持续的网络分区发生的话，系统也会在持续不可用。

性能、故障恢复和fsync

很多使用Redis做锁服务器的用户在获取锁和释放锁时不止要求低延时，同时要求高吞吐量，也即单位时间内可以获取和释放的锁数量。为了达到这个要求，一定会使用多路传输来和N个服务器进行通信以降低延时（或者也可以用假多路传输，也就是把socket设置成非阻塞模式，发送所有命令，然后再去读取返回的命令，假设说客户端和不同Redis服务节点的网络往返延时相差不大的话）。

然后如果我们想让系统可以自动故障恢复的话，我们还需要考虑一下信息持久化的问题。

为了更好地描述问题，我们先假设我们Redis都是配置成非持久化的，某个客户端拿到了总共5个节点中的3个锁，这三个已经获取到锁的节点总随后重启了，这样一来我们又有3个节点可以获取锁了（重启的那个加上另外两个），这样一来其他客户端又可以获得这个锁了，这样就违反了我们之前说的锁互斥原则了。

如果我们启用AOF持久化功能，情况会好很多。举例来说，我们可以发送SHUTDOWN命令来升级一个Redis服务器然后重启之，因为Redis超时时效是语义层面实现的，所以在服务器关掉期间时超时时间还是算在内的，我们所有要求还是满足了的。然后这个是基于我们做的是一次正常的shutdown，但是如果是断电这种意外停机呢？如果Redis是默认地配置成每秒在磁盘上执行一次fsync同步文件到磁盘操作，那就可能在一次重启后我们锁的key就丢失了。理论上如果我们想要在所有服务重启的情况下都确保锁的安全性，我们需要在持久化设置里设置成永远执行fsync操作，但是这个反过来又会造成性能远不如其他同级别的传统用来实现分布式锁的系统。然后问题其实并不像我们第一眼看起来那么糟糕，基本上只要一个服务节点在宕机重启后不去参与现在所有仍在使用的锁，这样正在使用的锁集合在这个服务节点重启时，算法的安全性就可以维持，因为这样就可以保证正在使用的锁都被所有没重启的节点持有。为了满足这个条件，我们只要让一个宕机重启后的实例，至少在我们使用的最大TTL时间内处于不可用状态，超过这个时间之后，所有在这期间活跃的锁都会自动释放掉。使用延时重启的策略基本上可以在不适用任何Redis持久化特性情况下保证安全性，然后要注意这个也必然会影响到系统的可用性。举个例子，如果系统里大多数节点都宕机了，那在TTL时间内整个系统都处于全局不可用状态（全局不可用的意思就是再获取不到任何锁）。

扩展锁来使得算法更可靠

如果客户端做的工作都是由一些小的步骤组成，那么就有可能使用更小的默认锁有效时间，而且扩展这个算法来实现一个锁扩展机制。基本上，客户端如果在执行计算期间发现锁快要超时了，客户端可以给所有服务实例发送一个Lua脚本让服务端延长锁的时间，只要这个锁的key还存在而且值还等于客户端获取时的那个值。客户端应当只有在失效时间内无法延长锁时再去重新获取锁（基本上这个和获取锁的算法是差不多的），然而这个并不会从本质上改变这个算法，所以最大的重新获取锁数量应该被设置成合理的大小，不然性能必然会受到影响。

总结

大多数情况下只要使用”set nx ex”然后主动解锁就可以了。不过这样要考虑到其局限性，上面已经讲到。

任何方案都有其存在的价值以及其适应的场景，几乎没有一劳永逸的完美的解决方法，不要一概而论，选择合适的就好。