本文是对 Martin Kleppmann 的文章 How to do distributed locking 部分内容的翻译和总结，上次写 Redlock 的原因就是看到了 Martin 的这篇文章，写得很好，特此翻译和总结。感兴趣的同学可以翻看原文，相信会收获良多。

开篇作者认为现在 Redis 逐渐被使用到数据管理领域，这个领域需要更强的数据一致性和耐久性，这使得他感到担心，因为这不是 Redis 最初设计的初衷（事实上这也是很多业界程序员的误区，越来越把 Redis 当成数据库在使用），其中基于 Redis 的分布式锁就是令人担心的其一。

Martin 指出首先你要明确你为什么使用分布式锁，为了性能还是正确性？为了帮你区分这二者，在这把锁 fail 了的时候你可以询问自己以下问题：

1. **要性能的：** 拥有这把锁使得你不会重复劳动（例如一个 job 做了两次），如果这把锁 fail 了，两个节点同时做了这个 Job，那么这个 Job 增加了你的成本。
2. **要正确性的：** 拥有锁可以防止并发操作污染你的系统或者数据，如果这把锁 fail 了两个节点同时操作了一份数据，结果可能是数据不一致、数据丢失、file 冲突等，会导致严重的后果。

上述二者都是需求锁的正确场景，但是你必须清楚自己是因为什么原因需要分布式锁。

如果你只是为了性能，那没必要用 Redlock，它成本高且复杂，你只用一个 Redis 实例也够了，最多加个从防止主挂了。当然，你使用单节点的 Redis 那么断电或者一些情况下，你会丢失锁，但是你的目的只是加速性能且断电这种事情不会经常发生，这并不是什么大问题。并且如果你使用了单节点 Redis，那么很显然你这个应用需要的锁粒度是很模糊粗糙的，也不会是什么重要的服务。

那么是否 Redlock 对于要求正确性的场景就合适呢？Martin 列举了若干场景证明 Redlock 这种算法是不可靠的。

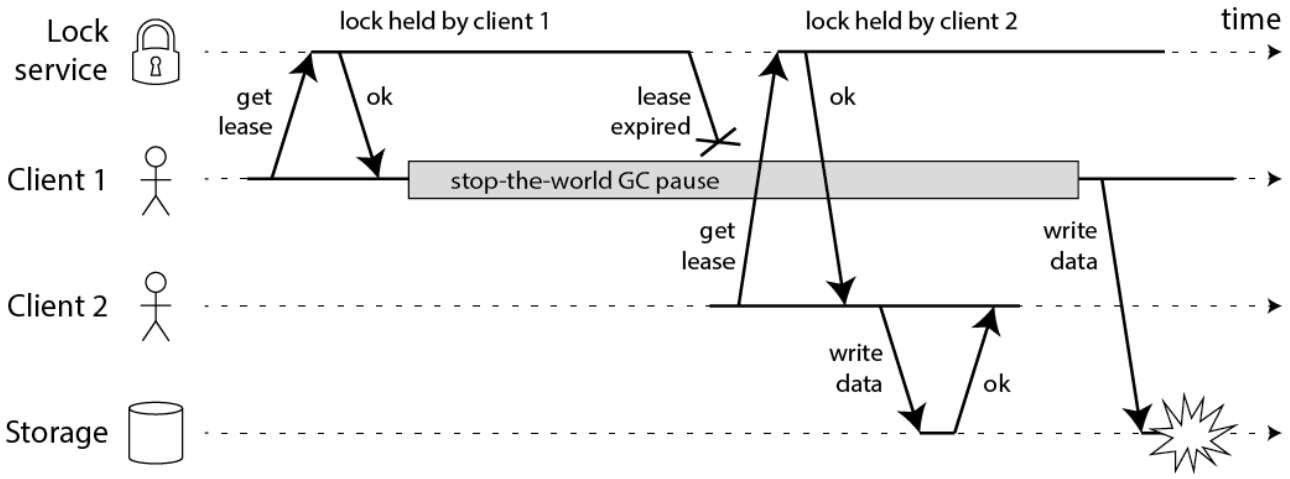
用锁保护资源

这节里 Martin 先将 Redlock 放在了一边而是仅讨论总体上一个分布式锁是怎么工作的。在分布式环境下，锁比 mutex 这类复杂，因为涉及到不同节点、网络通信并且他们随时可能无征兆的 fail 。 Martin 假设了一个场景，一个 client 要修改一个文件，它先申请得到锁，然后修改文件写回，放锁。另一个 client 再申请锁 ... 代码流程如下：

```
// THIS CODE IS BROKEN
function writeData(filename, data) {
    var lock = lockService.acquireLock(filename);
    if (!lock) {
        throw 'Failed to acquire lock';
    }

    try {
        var file = storage.readFile(filename);
        var updated = updateContents(file, data);
        storage.writeFile(filename, updated);
    } finally {
        lock.release();
    }
}
```

可惜即使你的锁服务非常完美，上述代码还是可能跪，下面的流程图会告诉你为什么：

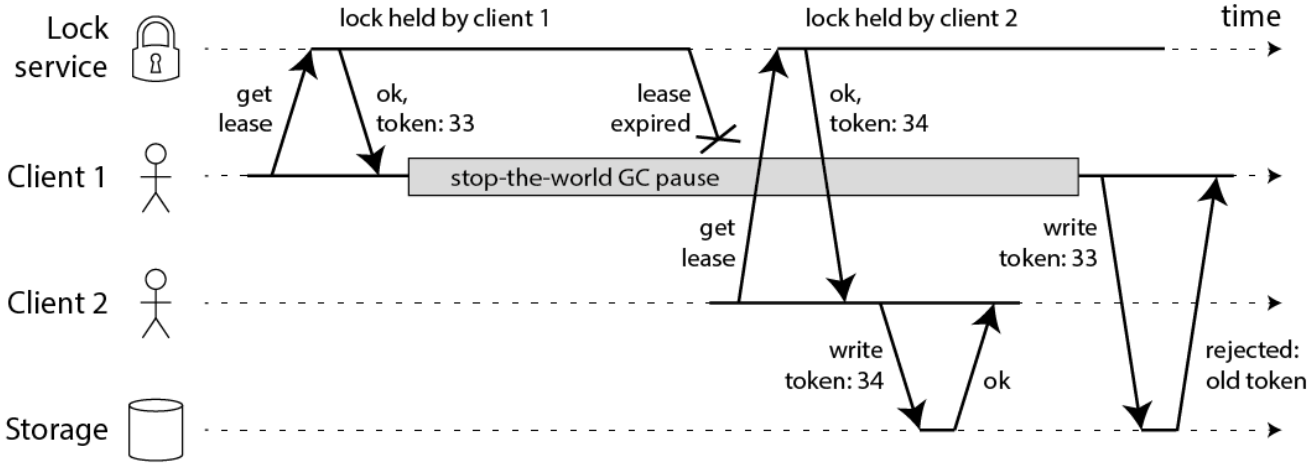


上述图中，得到锁的 client1 在持有锁的期间 pause 了一段时间，例如 GC 停顿。锁有过期时间（一般叫租约，为了防止某个 client 崩溃之后一直占有锁），但是如果 GC 停顿太长超过了锁租约时间，此时锁已经被另一个 client2 所得到，原先的 client1 还没有感知到锁过期，那么奇怪的结果就会发生，曾经 HBase 就发生过这种 Bug。即使你在 client1 写回之前检查一下锁是否过期也无助于解决这个问题，因为 GC 可能在任何时候发生，即使是你非常不便的时候（在最后的检查与写操作期间）。 如果你认为自己的程序不会有长时间的 GC 停顿，还有其他原因会导致你的进程 pause。例如进程可能读取尚未进入内存的数据，所以它得到一个 page fault 并且等待 page 被加载进缓存；还有可能你依赖于网络服务；或者其他进程占用 CPU；或者其他人意外发生 SIGSTOP 等。

... . 这里 Martin 又增加了一节列举各种进程 pause 的例子，为了证明上面的代码是不安全的，无论你的锁服务多完美。

使用 Fencing （栅栏）使得锁变安全

修复问题的方法也很简单：你需要在每次写操作时加入一个 fencing token。这个场景下，fencing token 可以是一个递增的数字（lock service 可以做到），每次有 client 申请锁就递增一次：



client1 申请锁同时拿到 token33，然后它进入长时间的停顿锁也过期了。client2 得到锁和 token34 写入数据，紧接着 client1 活过来之后尝试写入数据，自身 token33 比 34 小因此写入操作被拒绝。注意这需要存储层来检查 token，但这并不难实现。如果你使用 Zookeeper 作为 lock service 的话那么你可以使用 zxid 作为递增数字。 但是对于 Redlock 你要知道，没什么生成 fencing token 的方式，并且怎么修改 Redlock 算法使其能产生

fencing token 呢？好像并不那么显而易见。因为产生 token 需要单调递增，除非在单节点 Redis 上完成但是这又没有高可靠性，你好像需要引进一致性协议来让 Redlock 产生可靠的 fencing token。

使用时间来解决一致性

Redlock 无法产生 fencing token 早该成为在需求正确性的场景下弃用它的理由，但还有一些值得讨论的地方。

学术界有个说法，算法对时间不做假设：因为进程可能pause一段时间、数据包可能因为网络延迟延后到达、时钟可能根本就是错的。而可靠的算法依旧要在上述假设下做正确的事情。

对于 failure detector 来说，timeout 只能作为猜测某个节点 fail 的依据，因为网络延迟、本地时钟不正确等其他原因的限制。考虑到 Redis 使用 gettimeofday，而不是单调的时钟，会受到系统时间的影响，可能会突然前进或者后退一段时间，这会导致一个 key 更快或更慢地过期。

可见，Redlock 依赖于许多时间假设，它假设所有 Redis 节点都能对同一个 Key 在其过期前持有差不多的时间、跟过期时间相比网络延迟很小、跟过期时间相比进程 pause 很短。

用不可靠的时间打破 Redlock

这节 Martin 举了个因为时间问题，Redlock 不可靠的例子。

1. client1 从 ABC 三个节点处申请到锁，DE由于网络原因请求没有到达
2. C节点的时钟往前推了，导致 lock 过期
3. client2 在CDE处获得了锁，AB由于网络原因请求未到达
4. 此时 client1 和 client2 都获得了锁

在 Redlock 官方文档中也提到了这个情况，不过是C崩溃的时候，Redlock 官方本身也是知道 Redlock 算法不是完全可靠的，官方为了解决这种问题建议使用延时启动，相关内容可以看之前的这篇文章。但是 Martin 这里分析得更加全面，指出延时启动不也是依赖于时钟的正确性的么?

接下来 Martin 又列举了进程 Pause 时而不是时钟不可靠时会发生的问题：

1. client1 从 ABCDE 处获得了锁
2. 当获得锁的 response 还没到达 client1 时 client1 进入 GC 停顿
3. 停顿期间锁已经过期了
4. client2 在 ABCDE 处获得了锁
5. client1 GC 完成收到了获得锁的 response，此时两个 client 又拿到了同一把锁

同时长时间的网络延迟也有可能导致同样的问题。

Redlock 的同步性假设

这些例子说明了，仅有在你假设了一个同步性系统模型的基础上，Redlock 才能正常工作，也就是系统能满足以下属性：

1. 网络延时边界，即假设数据包一定能在某个最大延时之内到达
2. 进程停顿边界，即进程停顿一定在某个最大时间之内
3. 时钟错误边界，即不会从一个坏的 NTP 服务器处取得时间

结论

Martin 认为 Redlock 实在不是一个好的选择，对于需求性能的分布式锁应用它太重了且成本高；对于需求正确性的应用来说它不够安全。因为它对高危的时钟或者说其他上述列举的情况进行了不可靠的假设，如果你的应用只需要高性能的分布式锁不要求多高的正确性，那么单节点 Redis 够了；如果你的应用想要保住正确性，那么不建议 Redlock，建议使用一个合适的一致性协调系统，例如 Zookeeper，且保证存在 fencing token。