基于Redis的分布式锁到底安全吗（下）？

2017-02-24

自从我写完这个话题的上半部分之后，就感觉头脑中出现了许多细小的声音，久久挥之不去。它们就像是在为了一些鸡毛蒜皮的小事而相互争吵个不停。的确，有关分布式的话题就是这样，琐碎异常，而且每个人说的话听起来似乎都有道理。

今天，我们就继续探讨这个话题的后半部分。本文中，我们将从antirez反驳Martin Kleppmann的观点开始讲起，然后会涉及到Hacker News上出现的一些讨论内容，接下来我们还会讨论到基于Zookeeper和Chubby的分布式锁是怎样的，并和Redlock进行一些对比。最后，我们会提到Martin对于这一事件的总结。

还没有看过上半部分的同学，请先阅读：

- 基于Redis的分布式锁到底安全吗（上） (/posts/blog-redlock-reasoning.html)

antirez的反驳

Martin在发表了那篇分析分布式锁的blog (How to do distributed locking (https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html))之后，该文章在Twitter和Hacker News上引发了广泛的讨论。但人们更想听到的是Redlock的作者antirez对此会发表什么样的看法。

Martin的那篇文章是在2016-02-08这一天发表的，但据Martin说，他在公开发表文章的一星期之前就把草稿发给了antirez进行review，而且他们之间通过email进行了讨论。不知道Martin有没有意料到，antirez对于此事的反应很快，就在Martin的文章发表出来的第二天，antirez就在他的博客上贴出了他对于此事的反驳文章，名字叫”Is Redlock safe?”，地址如下:

- http://antirez.com/news/101 (http://antirez.com/news/101)

这是高手之间的过招。antirez这篇文章也条例非常清晰，并且中间涉及到大量的细节。antirez认为，Martin的文章对于Redlock的批评可以概括为两个方面（与Martin文章的前后两部分对应）：

- 带有自动过期功能的分布式锁，必须提供某种fencing机制来保证对共享资源的真正的互斥保护。Redlock提供不了这样一种机制。
- Redlock构建在一个不够安全的系统模型之上。它对于系统的记时假设(timing assumption)有比较强的要求，而这些要求在现实的系统中是无法保证的。

antirez对这两方面分别进行了反驳。

首先，关于fencing机制。antirez对于Martin的这种论证方式提出了质疑：既然在锁失效的情况下已经存在一种fencing机制能继续保持资源的互斥访问了，那为什么还要使用一个分布式锁并且还要求它提供那么强的安全性保证呢？即使退一步讲，Redlock虽然提供不了Martin所讲的递增的fencing token，但利用Redlock产生

的随机字符串(`my_random_value`)可以达到同样的效果。这个随机字符串虽然不是递增的，但却是唯一的，可以称之为unique token。antirez举了个例子，比如，你可以用它来实现“Check and Set”操作，原话是：

> When starting to work with a shared resource, we set its state to “ `<token>` ”, then we operate the read-modify-write only if the token is still the same when we write.
> （译文：当开始和共享资源交互的时候，我们将它的状态设置成“ `<token>` ”，然后仅在token没改变的情况下我们才执行“读取-修改-写回”操作。）

第一遍看到这个描述的时候，我个人是感觉没太看懂的。“Check and Set”应该就是我们平常听到过的CAS操作了，但它如何在这个场景下工作，antirez并没有展开说（在后面讲到Hacker News上的讨论的时候，我们还会提到）。

然后，antirez的反驳就集中在第二个方面上：关于算法在记时(timing)方面的模型假设。在我们前面分析Martin的文章时也提到过，Martin认为Redlock会失效的情况主要有三种：

- 时钟发生跳跃。
- 长时间的GC pause。
- 长时间的网络延迟。

antirez肯定意识到了这三种情况对Redlock最致命的其实是第一点：时钟发生跳跃。这种情况一旦发生，Redlock是没法正常工作的。而对于后两种情况来说，Redlock在当初设计的时候已经考虑到了，对它们引起的后果有一定的免疫力。所以，antirez接下来集中精力来说明通过恰当的运维，完全可以避免时钟发生大的跳动，而Redlock对于时钟的要求在现实系统中是完全可以满足的。

Martin在提到时钟跳跃的时候，举了两个可能造成时钟跳跃的具体例子：

- 系统管理员手动修改了时钟。
- 从NTP服务收到了一个大的时钟更新事件。

antirez反驳说：

- 手动修改时钟这种人为原因，不要那么做就是了。否则的话，如果有人手动修改Raft协议的持久化日志，那么就算是Raft协议它也没法正常工作了。
- 使用一个不会进行“跳跃”式调整系统时钟的ntpd程序（可能是通过恰当的配置），对于时钟的修改通过多次微小的调整来完成。

而Redlock对时钟的要求，并不需要完全精确，它只需要时钟差不多精确就可以了。比如，要记时5秒，但可能实际记了4.5秒，然后又记了5.5秒，有一定的误差。不过只要误差不超过一定范围，这对Redlock不会产生影响。antirez认为呢，像这样对时钟精度并不是很高的要求，在实际环境中是完全合理的。

好了，到此为止，如果你相信antirez这里关于时钟的论断，那么接下来antirez的分析就基本上顺理成章了。

关于Martin提到的能使Redlock失效的后两种情况，Martin在分析的时候恰好犯了一个错误（在本文上半部分(/posts/blog-redlock-reasoning.html)已经提到过）。在Martin给出的那个由客户端GC pause引发Redlock失效的例子中，这个GC pause引发的后果相当于在锁服务器和客户端之间发生了长时间的消息延迟。Redlock对

于这个情况是能处理的。回想一下Redlock算法的具体过程，它使用起来的过程大体可以分成5步：

1. 获取当前时间。
2. 完成**获取锁**的整个过程（与N个Redis节点交互）。
3. 再次获取当前时间。
4. 把两个时间相减，计算**获取锁**的过程是否消耗了太长时间，导致锁已经过期了。如果没过期，
5. 客户端持有锁去访问共享资源。

在Martin举的例子中，GC pause或网络延迟，实际发生在上述第1步和第3步之间。而不管在第1步和第3步之间由于什么原因（进程停顿或网络延迟等）导致了大的延迟出现，在第4步都能被检查出来，不会让客户端拿到一个它认为有效而实际却已经过期的锁。当然，这个检查依赖系统时钟没有大的跳跃。这也就是为什么antirez在前面要对时钟条件进行辩护的原因。

有人会说，在第3步之后，仍然可能会发生延迟啊。没错，antirez承认这一点，他对此有一段很有意思的论证，原话如下：

> The delay can only happen after steps 3, resulting into the lock to be considered ok while actually expired, that is, we are back at the first problem Martin identified of distributed locks where the client fails to stop working to the shared resource before the lock validity expires. Let me tell again how this problem is common with *all the distributed locks implementations*, and how the token as a solution is both unrealistic and can be used with Redlock as well.
> （译文：延迟只能发生在第3步之后，这导致锁被认为是有效的而实际上已经过期了，也就是说，我们回到了Martin指出的第一个问题上，客户端没能够在锁的有效性过期之前完成与共享资源的交互。让我再次申明一下，这个问题对于*所有的分布式锁的实现*是普遍存在的，而且基于token的这种解决方案是不切实际的，但也能和Redlock一起用。）

这里antirez所说的“Martin指出的第一个问题”具体是什么呢？在本文上半部分 (/posts/blog-redlock-reasoning.html)我们提到过，Martin的文章分为两大部分，其中前半部分与Redlock没有直接关系，而是指出了任何一种带自动过期功能的分布式锁在没有提供fencing机制的前提下都有可能失效。这里antirez所说的就是指的Martin的文章的前半部分。换句话说，对于大延迟给Redlock带来的影响，恰好与Martin在文章的前半部分针对所有的分布式锁所做的分析是一致的，而这种影响不单单针对Redlock。Redlock的实现已经保证了它是和其它任何分布式锁的安全性是一样的。当然，与其它“更完美”的分布式锁相比，Redlock似乎提供不了Martin提出的那种递增的token，但antirez在前面已经分析过了，关于token的这种论证方式本身就是“不切实际”的，或者退一步讲，Redlock能提供的unique token也能够提供完全一样的效果。

另外，关于大延迟对Redlock的影响，antirez和Martin在Twitter上有下面的对话：

> **antirez**:
> @martinkl so I wonder if after my reply, we can at least agree about unbound messages delay to don’t cause any harm.
>
> **Martin**:
> @antirez Agree about message delay between app and lock server. Delay between app and resource being accessed is still problematic.
>
> （译文：
> **antirez**问：我想知道，在我发文回复之后，我们能否在一点上达成一致，就是大的消息延迟不会给Redlock的运行造成损害。
> **Martin**答：对于客户端和锁服务器之间的消息延迟，我同意你的观点。但客户端和被访问资源之间的延迟还是有问题的。）

通过这段对话可以看出，对于Redlock在第4步所做的锁有效性的检查，Martin是予以肯定的。但他认为客户端和资源服务器之间的延迟还是会带来问题的。Martin在这里说的有点模糊。就像antirez前面分析的，客户端和资源服务器之间的延迟，对所有的分布式锁的实现都会带来影响，这不单单是Redlock的问题了。

以上就是antirez在blog中所说的主要内容。有一些点值得我们注意一下：

- antirez是同意大的系统时钟跳跃会造成Redlock失效的。在这一点上，他与Martin的观点的不同在于，他认为在实际系统中是可以避免大的时钟跳跃的。当然，这取决于基础设施和运维方式。
- antirez在设计Redlock的时候，是充分考虑了网络延迟和程序停顿所带来的影响的。但是，对于客户端和资源服务器之间的延迟（即发生在算法第3步之后的延迟），antirez是承认所有的分布式锁的实现，包括Redlock，是没有什么好办法来应对的。

讨论进行到这，Martin和antirez之间谁对谁错其实并不是那么重要了。只要我们能够对Redlock（或者其它分布式锁）所能提供的安全性的程度有充分的了解，那么我们就能做出自己的选择了。

Hacker News上的一些讨论

针对Martin和antirez的两篇blog，很多技术人员在Hacker News上展开了激烈的讨论。这些讨论所在地址如下：

- 针对Martin的blog的讨论：https://news.ycombinator.com/item?id=11059738 (https://news.ycombinator.com/item?id=11059738)
- 针对antirez的blog的讨论：https://news.ycombinator.com/item?id=11065933 (https://news.ycombinator.com/item?id=11065933)

在Hacker News上，antirez积极参与了讨论，而Martin则始终置身事外。

下面我把这些讨论中一些有意思的点拿出来与大家一起分享一下（集中在对于fencing token机制的讨论上）。

关于antirez提出的“Check and Set”操作，他在blog里并没有详加说明。果然，在Hacker News上就有人出来问了。antirez给出的答复如下：

> You want to modify locked resource X. You set X.currlock = token. Then you read, do whatever you want, and when you write, you “write-if-currlock == token”. If another client did X.currlock = somethingelse, the transaction fails.

翻译一下可以这样理解：假设你要修改资源X，那么遵循下面的伪码所定义的步骤。

1. 先设置X.currlock = token。
2. 读出资源X（包括它的值和附带的X.currlock）。
3. 按照”write-if-currlock == token”的逻辑，修改资源X的值。意思是说，如果对X进行修改的时候，X.currlock仍然和当初设置进去的token相等，那么才进行修改；如果这时X.currlock已经是其它值了，那么说明有另外一方也在试图进行修改操作，那么放弃当前的修改，从而避免冲突。

随后Hacker News上一位叫viraptor的用户提出了异议，它给出了这样一个执行序列：

- A: X.currlock = Token_ID_A
- A: resource read
- A: is X.currlock still Token_ID_A? yes
- B: X.currlock = Token_ID_B
- B: resource read
- B: is X.currlock still Token_ID_B? yes
- B: resource write
- A: resource write

到了最后两步，两个客户端A和B同时进行写操作，冲突了。不过，这位用户应该是理解错了antirez给出的修改过程了。按照antirez的意思，判断X.currlock是否修改过和对资源的写操作，应该是一个原子操作。只有这样理解才能合乎逻辑，否则的话，这个过程就有严重的破绽。这也是为什么antirez之前会对fencing机制产生质疑：既然资源服务器本身都能提供互斥的原子操作了，为什么还需要一个分布式锁呢？因此，antirez认为这种fencing机制是很累赘的，他之所以还是提出了这种“Check and Set”操作，只是为了证明在提供fencing token这一点上，Redlock也能做到。但是，这里仍然有一些不明确的地方，如果将”write-if-currlock == token”看做是原子操作的话，这个逻辑势必要在资源服务器上执行，那么第二步为什么还要“读出资源X”呢？除非这个“读出资源X”的操作也是在资源服务器上执行，它包含在“判断-写回”这个原子操作里面。而假如不这样理解的话，“读取-判断-写回”这三个操作都放在客户端执行，那么看不出它们如何才能实现原子性操作。在下面的讨论中，我们暂时忽略“读出资源X”这一步。

这个基于random token的“Check and Set”操作，如果与Martin提出的递增的fencing token对比一下的话，至少有两点不同：

- “Check and Set”对于写操作要分成两步来完成（设置token、判断-写回），而递增的fencing token机制只需要一步（带着token向资源服务器发起写请求）。
- 递增的fencing token机制能保证最终操作共享资源的顺序，那些延迟时间太长的操作就无法操作共享资源了。但是基于random token的“Check and Set”操作不会保证这个顺序，那些延迟时间太长的操作如果后到达了，它仍然有可能操作共享资源（当然是以互斥的方式）。

对于前一点不同，我们在后面的分析中会看到，如果资源服务器也是分布式的，那么使用递增的fencing token也要变成两步。

而对于后一点操作顺序上的不同，antirez认为这个顺序没有意义，关键是能互斥访问就行了。他写下了下面的话：

> So the goal is, when race conditions happen, to avoid them in some way.
> ……
> Note also that when it happens that, because of delays, the clients are accessing concurrently, the lock ID has little to do with the order in which the operations were indented to happen.
> （译文： 我们的目标是，当竞争条件出现的时候，能够以**某种方式**避免。
> ……
> 还需要注意的是，当那种竞争条件出现的时候，比如由于延迟，客户端是同时来访问的，锁的ID的大小顺序跟那些操作真正想执行的顺序，是没有什么关系的。）

这里的lock ID，跟Martin说的递增的token是一回事。

随后，antirez举了一个“将名字加入列表”的操作的例子：

- T0: Client A receives new name to add from web.
- T0: Client B is idle
- T1: Client A is experiencing pauses.
- T1: Client B receives new name to add from web.
- T2: Client A is experiencing pauses.
- T2: Client B receives a lock with ID 1
- T3: Client A receives a lock with ID 2

你看，两个客户端（其实是Web服务器）执行“添加名字”的操作，A本来是排在B前面的，但获得锁的顺序却是B排在A前面。因此，antirez说，锁的ID的大小顺序跟那些操作真正想执行的顺序，是没有什么关系的。关键是能排出一个顺序来，能互斥访问就行了。那么，至于锁的ID是递增的，还是一个random token，自然就不那么重要了。

Martin提出的fencing token机制，给人留下了无尽的疑惑。这主要是因为他对于这一机制的描述缺少太多的技术细节。从上面的讨论可以看出，antirez对于这一机制的看法是，它跟一个random token没有什么区别，而且，它需要资源服务器本身提供某种互斥机制，这几乎让分布式锁本身的存在失去了意义。围绕fencing token的问题，还有两点是比较引人注目的，Hacker News上也有人提出了相关的疑问：

- （1）关于资源服务器本身的架构细节。
- （2）资源服务器对于fencing token进行检查的实现细节，比如是否需要提供一种原子操作。

关于上述问题（1），Hacker News上有一位叫dwenzek的用户发表了下面的评论：

> …… the issue around the usage of fencing tokens to reject any late usage of a lock is unclear just because the protected resource and its access are themselves unspecified. Is the resource distributed or not? If distributed, does the resource has a mean to ensure that tokens are increasing over all the nodes? Does the resource have a mean to rollback any effects done by a client which session is interrupted by a timeout?
>
> （译文：…… 关于使用fencing token拒绝掉延迟请求的相关议题，是不够清晰的，因为受保护的资源以及对它的访问方式本身是没有被明确定义过的。资源服务是不是分布式的呢？如果是，资源服务有没有一种方式能确保token在所有节点上递增呢？对于客户端的Session由于过期而被中断的情况，资源服务有办法将它的影响回滚吗？）

这些疑问在Hacker News上并没有人给出解答。而关于分布式的资源服务器架构如何处理fencing token，另外一名分布式系统的专家Flavio Junqueira (https://fpj.me/)在他的一篇blog中有所提及（我们后面会再提到）。

关于上述问题（2），Hacker News上有一位叫reza_n的用户发表了下面的疑问：

> I understand how a fencing token can prevent out of order writes when 2 clients get the same lock. But what happens when those writes happen to arrive in order and you are doing a value modification? Don’t you still need to rely on some kind of value versioning or optimistic locking? Wouldn’t this make the use of a distributed lock unnecessary?
>
> （译文： 我理解当两个客户端同时获得锁的时候fencing token是如何防止乱序的。但是如果两个写操作恰好按序到达了，而且它们在对同一个值进行修改，那会发生什么呢？难道不会仍然是依赖某种数据版本号或者乐观锁的机制？这不会让分布式锁变得没有必要了吗？）

一位叫Terr_的Hacker News用户答：

> I believe the “first” write fails, because the token being passed in is no longer “the lastest”, which indicates their lock was already released or expired.
>
> （译文： 我认为“第一个”写请求会失败，因为它传入的token不再是“最新的”了，这意味着锁已经释放或者过期了。）

Terr_的回答到底对不对呢？这不好说，取决于资源服务器对于fencing token进行检查的实现细节。让我们来简单分析一下。

为了简单起见，我们假设有一台（先不考虑分布式的情况）通过RPC进行远程访问文件服务器，它无法提供对于文件的互斥访问（否则我们就不需要分布式锁了）。现在我们按照Martin给出的说法，加入fencing token的检查逻辑。由于Martin没有描述具体细节，我们猜测至少有两种可能。

第一种可能，我们修改了文件服务器的代码，让它能多接受一个fencing token的参数，并在进行所有处理之前加入了一个简单的判断逻辑，保证只有当前接收到的fencing token大于之前的值才允许进行后边的访问。而一旦通过了这个判断，后面的处理不变。

现在想象reza_n描述的场景，客户端1和客户端2都发生了GC pause，两个fencing token都延迟了，它们几乎同时到达了文件服务器，而且保持了顺序。那么，我们新加入的判断逻辑，应该对两个请求都会放过，而放过之后它们几乎同时在操作文件，还是冲突了。既然Martin宣称fencing token能保证分布式锁的正确性，那么上面这种可能的猜测也许是我们理解错了。

当然，还有第二种可能，就是我们对文件服务器确实做了比较大的改动，让这里判断token的逻辑和随后对文件的处理放在一个原子操作里了。这可能更接近antirez的理解。这样的话，前面reza_n描述的场景中，两个写操作都应该成功。

基于ZooKeeper的分布式锁更安全吗?

很多人（也包括Martin在内）都认为，如果你想构建一个更安全的分布式锁，那么应该使用ZooKeeper，而不是Redis。那么，为了对比的目的，让我们先暂时脱离开本文的题目，讨论一下基于ZooKeeper的分布式锁能提供绝对的安全吗？它需要fencing token机制的保护吗?

我们不得不提一下分布式专家Flavio Junqueira (https://fpj.me/)所写的一篇blog，题目叫“Note on fencing and distributed locks”，地址如下：

- https://fpj.me/2016/02/10/note-on-fencing-and-distributed-locks/ (https://fpj.me/2016/02/10/note-on-fencing-and-distributed-locks/)

Flavio Junqueira是ZooKeeper的作者之一，他的这篇blog就写在Martin和antirez发生争论的那几天。他在文中给出了一个基于ZooKeeper构建分布式锁的描述（当然这不是唯一的方式）：

- 客户端尝试创建一个znode节点，比如 `/lock` 。那么第一个客户端就创建成功了，相当于拿到了锁；而其它的客户端会创建失败（znode已存在），获取锁失败。
- 持有锁的客户端访问共享资源完成后，将znode删掉，这样其它客户端接下来就能来获取锁了。
- znode应该被创建成ephemeral的。这是znode的一个特性，它保证如果创建znode的那个客户端崩溃了，那么相应的znode会被自动删除。这保证了锁一定会被释放。

看起来这个锁相当完美，没有Redlock过期时间的问题，而且能在需要的时候让锁自动释放。但仔细考察的话，并不尽然。

ZooKeeper是怎么检测出某个客户端已经崩溃了呢？实际上，每个客户端都与ZooKeeper的某台服务器维护着一个Session，这个Session依赖定期的心跳(heartbeat)来维持。如果ZooKeeper长时间收不到客户端的心跳（这个时间称为Sesion的过期时间），那么它就认为Session过期了，通过这个Session所创建的所有的ephemeral类型的znode节点都会被自动删除。

设想如下的执行序列：

1. 客户端1创建了znode节点 `/lock` ，获得了锁。
2. 客户端1进入了长时间的GC pause。
3. 客户端1连接到ZooKeeper的Session过期了。znode节点 `/lock` 被自动删除。

4. 客户端2创建了znode节点 `/lock` ，从而获得了锁。
5. 客户端1从GC pause中恢复过来，它仍然认为自己持有锁。

最后，客户端1和客户端2都认为自己持有了锁，冲突了。这与之前Martin在文章中描述的由于GC pause导致的分布式锁失效的情况类似。

看起来，用ZooKeeper实现的分布式锁也不一定就是安全的。该有的问题它还是有。但是，ZooKeeper作为一个专门为分布式应用提供方案的框架，它提供了一些非常好的特性，是Redis之类的方案所没有的。像前面提到的ephemeral类型的znode自动删除的功能就是一个例子。

还有一个很有用的特性是ZooKeeper的watch机制。这个机制可以这样来使用，比如当客户端试图创建 `/lock` 的时候，发现它已经存在了，这时候创建失败，但客户端不一定就此对外宣告获取锁失败。客户端可以进入一种等待状态，等待当 `/lock` 节点被删除的时候，ZooKeeper通过watch机制通知它，这样它就可以继续完成创建操作（获取锁）。这可以让分布式锁在客户端用起来就像一个本地的锁一样：加锁失败就阻塞住，直到获取到锁为止。这样的特性Redlock就无法实现。

小结一下，基于ZooKeeper的锁和基于Redis的锁相比在实现特性上有两个不同：

- 在正常情况下，客户端可以持有锁任意长的时间，这可以确保它做完所有需要的资源访问操作之后再释放锁。这避免了基于Redis的锁对于有效时间(lock validity time)到底设置多长的两难问题。实际上，基于ZooKeeper的锁是依靠Session（心跳）来维持锁的持有状态的，而Redis不支持Sesion。
- 基于ZooKeeper的锁支持在获取锁失败之后等待锁重新释放的事件。这让客户端对锁的使用更加灵活。

顺便提一下，如上所述的基于ZooKeeper的分布式锁的实现，并不是最优的。它会引发“herd effect”（羊群效应），降低获取锁的性能。一个更好的实现参见下面链接：

- http://zookeeper.apache.org/doc/r3.4.9/recipes.html#sc_recipes_Locks (http://zookeeper.apache.org/doc/r3.4.9/recipes.html#sc_recipes_Locks)

我们重新回到Flavio Junqueira对于fencing token的分析。Flavio Junqueira指出，fencing token机制本质上是要求客户端在每次访问一个共享资源的时候，在执行任何操作之前，先对资源进行某种形式的“标记”(mark)操作，这个“标记”能保证持有旧的锁的客户端请求（如果延迟到达了）无法操作资源。这种标记操作可以是很多形式，fencing token是其中比较典型的一个。

随后Flavio Junqueira提到用递增的epoch number（相当于Martin的fencing token）来保护共享资源。而对于分布式的资源，为了方便讨论，假设分布式资源是一个小型的多备份的数据存储(a small replicated data store)，执行写操作的时候需要向所有节点上写数据。最简单的做标记的方式，就是在对资源进行任何操作之前，先把epoch number标记到各个资源节点上去。这样，各个节点就保证了旧的（也就是小的）epoch number无法操作数据。

当然，这里再展开讨论下去可能就涉及到了这个数据存储服务的实现细节了。比如在实际系统中，可能为了容错，只要上面讲的标记和写入操作在多数节点上完成就算成功完成了（Flavio Junqueira并没有展开去讲）。在这里我们能看到的，最重要的，是这种标记操作如何起作用的方式。这有点类似于Paxos协议

（Paxos协议要求每个proposal对应一个递增的数字，执行accept请求之前先执行prepare请求）。antirez提出的random token的方式显然不符合Flavio Junqueira对于“标记”操作的定义，因为它无法区分新的token和旧的token。只有递增的数字才能确保最终收敛到最新的操作结果上。

在这个分布式数据存储服务（共享资源）的例子中，客户端在标记完成之后执行写入操作的时候，存储服务的节点需要判断epoch number是不是最新，然后确定能不能执行写入操作。如果按照上一节我们的分析思路，这里的epoch判断和接下来的写入操作，是不是在一个原子操作里呢？根据Flavio Junqueira的相关描述，我们相信，应该是原子的。那么既然资源本身可以提供原子互斥操作了，那么分布式锁还有存在的意义吗？应该说有。客户端可以利用分布式锁有效地避免冲突，等待写入机会，这对于包含多个节点的分布式资源尤其有用（当然，是出于效率的原因）。

Chubby的分布式锁是怎样做fencing的?

提到分布式锁，就不能不提Google的Chubby。

Chubby是Google内部使用的分布式锁服务，有点类似于ZooKeeper，但也存在很多差异。Chubby对外公开的资料，主要是一篇论文，叫做“The Chubby lock service for loosely-coupled distributed systems”，下载地址如下：

- https://research.google.com/archive/chubby.html (https://research.google.com/archive/chubby.html)

另外，YouTube上有一个的讲Chubby的talk，也很不错，播放地址：

- https://www.youtube.com/watch?v=PqItueBaiRg&feature=youtu.be&t=487 (https://www.youtube.com/watch?v=PqItueBaiRg&feature=youtu.be&t=487)

Chubby自然也考虑到了延迟造成的锁失效的问题。论文里有一段描述如下：

> a process holding a lock L may issue a request R, but then fail. Another process may ac- quire L and perform some action before R arrives at its destination. If R later arrives, it may be acted on without the protection of L, and potentially on inconsistent data.
>
> （译文： 一个进程持有锁L，发起了请求R，但是请求失败了。另一个进程获得了锁L并在请求R到达目的方之前执行了一些动作。如果后来请求R到达了，它就有可能在没有锁L保护的情况下进行操作，带来数据不一致的潜在风险。）

这跟Martin的分析大同小异。

Chubby给出的用于解决（缓解）这一问题的机制称为sequencer，类似于fencing token机制。锁的持有者可以随时请求一个sequencer，这是一个字节串，它由三部分组成：

- 锁的名字。
- 锁的获取模式（排他锁还是共享锁）。
- lock generation number（一个64bit的单调递增数字）。作用相当于fencing token或epoch number。

客户端拿到sequencer之后，在操作资源的时候把它传给资源服务器。然后，资源服务器负责对sequencer的有效性进行检查。检查可以有两种方式：

- 调用Chubby提供的API，CheckSequencer()，将整个sequencer传进去进行检查。这个检查是为了保证客户端持有的锁在进行资源访问的时候仍然有效。
- 将客户端传来的sequencer与资源服务器当前观察到的最新的sequencer进行对比检查。可以理解为与Martin描述的对于fencing token的检查类似。

当然，如果由于兼容的原因，资源服务本身不容易修改，那么Chubby还提供了一种机制：

- lock-delay。Chubby允许客户端为持有的锁指定一个lock-delay的时间值（默认是1分钟）。当Chubby发现客户端被动失去联系的时候，并不会立即释放锁，而是会在lock-delay指定的时间内阻止其它客户端获得这个锁。这是为了在把锁分配给新的客户端之前，让之前持有锁的客户端有充分的时间把请求队列排空(draining the queue)，尽量防止出现延迟到达的未处理请求。

可见，为了应对锁失效问题，Chubby提供的三种处理方式：CheckSequencer()检查、与上次最新的sequencer对比、lock-delay，它们对于安全性的保证是从强到弱的。而且，这些处理方式本身都没有保证提供绝对的正确性(correctness)。但是，Chubby确实提供了单调递增的lock generation number，这就允许资源服务器在需要的时候，利用它提供更强的安全性保障。

关于时钟

在Martin与antirez的这场争论中，冲突最为严重的就是对于系统时钟的假设是不是合理的问题。Martin认为系统时钟难免会发生跳跃（这与分布式算法的异步模型相符），而antirez认为在实际中系统时钟可以保证不发生大的跳跃。

Martin对于这一分歧发表了如下看法（原话）：

> So, fundamentally, this discussion boils down to whether it is reasonable to make timing assumptions for ensuring safety properties. I say no, Salvatore says yes — but that’s ok. Engineering discussions rarely have one right answer.
>
> （译文： 从根本上来说，这场讨论最后归结到了一个问题上：为了确保安全性而做出的记时假设到底是否合理。我认为不合理，而antirez认为合理 —— 但是这也没关系。工程问题的讨论很少只有一个正确答案。）

那么，在实际系统中，时钟到底是否可信呢？对此，Julia Evans (http://jvns.ca/about/)专门写了一篇文章，“TIL: clock skew exists”，总结了很多跟时钟偏移有关的实际资料，并进行了分析。这篇文章地址：

- http://jvns.ca/blog/2016/02/09/til-clock-skew-exists/ (http://jvns.ca/blog/2016/02/09/til-clock-skew-exists/)

Julia Evans在文章最后得出的结论是：

> clock skew is real （时钟偏移在现实中是存在的）

Martin的事后总结

我们前面提到过，当各方的争论在激烈进行的时候，Martin几乎始终置身事外。但是Martin在这件事过去之后，把这个事件的前后经过总结成了一个很长的故事线。如果你想最全面地了解这个事件发生的前后经过，那么建议去读读Martin的这个总结：

- https://storify.com/martinkl/redlock-discussion (https://storify.com/martinkl/redlock-discussion)

在这个故事总结的最后，Martin写下了很多感性的评论：

> For me, this is the most important point: I don't care who is right or wrong in this debate — I care about learning from others' work, so that we can avoid repeating old mistakes, and make things better in future. So much great work has already been done for us: by standing on the shoulders of giants, we can build better software.
> ……
> By all means, test ideas by arguing them and checking whether they stand up to scrutiny by others. That's part of the learning process. But the goal should be to learn, not to convince others that you are right. Sometimes that just means to stop and think for a while.
>
> （译文：
> 对我来说最重要的一点在于：我并不在乎在这场辩论中谁对谁错 —— 我只关心从其他人的工作中学到的东西，以便我们能够避免重蹈覆辙，并让未来更加美好。前人已经为我们创造出了许多伟大的成果：站在巨人的肩膀上，我们得以构建更棒的软件。
> ……
> 对于任何想法，务必要详加检验，通过论证以及检查它们是否经得住别人的详细审查。那是学习过程的一部分。但目标应该是为了获得知识，而不应该是为了说服别人相信你自己是对的。有时候，那只不过意味着停下来，好好地想一想。）

关于分布式锁的这场争论，我们已经完整地做了回顾和分析。

按照锁的两种用途，如果仅是为了效率(efficiency)，那么你可以自己选择你喜欢的一种分布式锁的实现。当然，你需要清楚地知道它在安全性上有哪些不足，以及它会带来什么后果。而如果你是为了正确性(correctness)，那么请慎之又慎。在本文的讨论中，我们在分布式锁的正确性上走得最远的地方，要数对于ZooKeeper分布式锁、单调递增的epoch number以及对分布式资源进行标记的分析了。请仔细审查相关的论证。

Martin为我们留下了不少疑问，尤其是他提出的fencing token机制。他在blog中提到，会在他的新书《Designing Data-Intensive Applications》的第8章和第9章再详加论述。目前，这本书尚在预售当中。我感觉，这会是一本值得一读的书，它不同于为了出名或赚钱而出版的那种短平快的书籍。可以看出作者在这本书上投入了巨大的精力。

最后，我相信，这个讨论还远没有结束。分布式锁(Distributed Locks)和相应的fencing方案，可以作为一个长期的课题，随着我们对分布式系统的认识逐渐增加，可以再来慢慢地思考它。思考它更深层的本质，以及它在理论上的证明。

（完）

感谢：

由衷地感谢几位朋友花了宝贵的时间对本文草稿所做的review：CacheCloud的作者付磊，快手的李伟博，阿里的李波。当然，文中如果还有错漏，由我本人负责^-^。

其它精选文章：

- 基于Redis的分布式锁到底安全吗（上） (/posts/blog-redlock-reasoning.html)
- Redis内部数据结构详解(7)——intset (/posts/blog-redis-intset.html)
- Redis内部数据结构详解(6)——skiplist (/posts/blog-redis-skiplist.html)
- Redis内部数据结构详解(5)——quicklist (/posts/blog-redis-quicklist.html)
- Redis内部数据结构详解(4)——ziplist (/posts/blog-redis-ziplist.html)
- Redis内部数据结构详解(3)——robj (/posts/blog-redis-robj.html)
- Redis内部数据结构详解(2)——sds (/posts/blog-redis-sds.html)
- Redis内部数据结构详解(1)——dict (/posts/blog-redis-dict.html)
- 知识的三个层次 (/posts/blog-knowledge-hierarchy.html)
- 技术的成长曲线 (/posts/blog-growth-curve.html)
- 技术的正宗与野路子 (http://mp.weixin.qq.com/s?__biz=MzA4NTg1MjM0Mg==&mid=2657261357&idx=1&sn=ebb11a1623e00ca8e6ad55c9ad6b2547#rd)

上篇： 基于Redis的分布式锁到底安全吗（上）？ (/posts/blog-redlock-reasoning.html)
下篇： 蓄力十年，做一个成就 (/posts/blog-10-years-planning.html)

栏目分类

最新文章