基于Redis的分布式锁到底安全吗（上）？

2017-02-11

网上有关Redis分布式锁的文章可谓多如牛毛了，不信的话你可以拿关键词“Redis 分布式锁”随便到哪个搜索引擎上去搜索一下就知道了。这些文章的思路大体相近，给出的实现算法也看似合乎逻辑，但当我们着手去实现它们的时候，却发现如果你越是仔细推敲，疑虑也就越来越多。

实际上，大概在一年以前，关于Redis分布式锁的安全性问题，在分布式系统专家Martin Kleppmann (https://martin.kleppmann.com/)和Redis的作者antirez (http://antirez.com/)之间就发生过一场争论。由于对这个问题一直以来比较关注，所以我前些日子仔细阅读了与这场争论相关的资料。这场争论的大概过程是这样的：为了规范各家对基于Redis的分布式锁的实现，Redis的作者提出了一个更安全的实现，叫做Redlock (https://redis.io/topics/distlock)。有一天，Martin Kleppmann写了一篇blog (https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html)，分析了Redlock在安全性上存在的一些问题。然后Redis的作者立即写了一篇blog (http://antirez.com/news/101)来反驳Martin的分析。但Martin表示仍然坚持原来的观点。随后，这个问题在Twitter和Hacker News上引发了激烈的讨论，很多分布式系统的专家都参与其中。

对于那些对分布式系统感兴趣的人来说，这个事件非常值得关注。不管你是刚接触分布式系统的新手，还是有着多年分布式开发经验的老手，读完这些分析和评论之后，大概都会有所收获。要知道，亲手实现过Redis Cluster这样一个复杂系统的antirez，足以算得上分布式领域的一名专家了。但对于由分布式锁引发的一系列问题的分析中，不同的专家却能得出迥异的结论，从中我们可以窥见分布式系统相关的问题具有何等的复杂性。实际上，在分布式系统的设计中经常发生的事情是：许多想法初看起来毫无破绽，而一旦详加考量，却发现不是那么天衣无缝。

下面，我们就从头至尾把这场争论过程中各方的观点进行一下回顾和分析。在这个过程中，我们把影响分布式锁的安全性的那些技术细节展开进行讨论，这将是一件很有意思的事情。这也是一个比较长的故事。当然，其中也免不了包含一些小“八卦”。

Redlock算法

就像本文开头所讲的，借助Redis来实现一个分布式锁(Distributed Lock)的做法，已经有很多人尝试过。人们构建这样的分布式锁的目的，是为了对一些共享资源进行互斥访问。

但是，这些实现虽然思路大体相近，但实现细节上各不相同，它们能提供的安全性和可用性也不尽相同。所以，Redis的作者antirez给出了一个更好的实现，称为Redlock，算是Redis官方对于实现分布式锁的指导规范。Redlock的算法描述就放在Redis的官网上：

- https://redis.io/topics/distlock (https://redis.io/topics/distlock)

在Redlock之前，很多人对于分布式锁的实现都是基于单个Redis节点的。而Redlock是基于多个Redis节点（都是Master）的一种实现。为了能理解Redlock，我们首先需要把简单的基于单Redis节点的算法描述清楚，因为它是Redlock的基础。

基于单Redis节点的分布式锁

首先，Redis客户端为了**获取锁**，向Redis节点发送如下命令：

```
SET resource_name my_random_value NX PX 30000
```

上面的命令如果执行成功，则客户端成功获取到了锁，接下来就可以**访问共享资源**了；而如果上面的命令执行失败，则说明获取锁失败。

注意，在上面的 SET 命令中：

- `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。当然，这里30秒只是一个例子，客户端可以选择合适的过期时间。

最后，当客户端完成了对共享资源的操作之后，执行下面的Redis Lua脚本来**释放锁**：

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

这段Lua脚本在执行的时候要把前面的 `my_random_value` 作为 `ARGV[1]` 的值传进去，把 `resource_name` 作为 `KEYS[1]` 的值传进去。

至此，基于单Redis节点的分布式锁的算法就描述完了。这里面有好几个问题需要重点分析一下。

首先第一个问题，这个锁必须要设置一个过期时间。否则的话，当一个客户端获取锁成功之后，假如它崩溃了，或者由于发生了网络分割（network partition）导致它再也无法和Redis节点通信了，那么它就会一直持有这个锁，而其它客户端永远无法获得锁了。antirez在后面的分析中也特别强调了这一点，而且把这个过期时间称为锁的有效时间(lock validity time)。获得锁的客户端必须在这个时间之内完成对共享资源的访问。

第二个问题，第一步**获取锁**的操作，网上不少文章把它实现成了两个Redis命令：

```
SETNX resource_name my_random_value
EXPIRE resource_name 30
```

虽然这两个命令和前面算法描述中的一个 SET 命令执行效果相同，但却不是原子的。如果客户端在执行完 SETNX 后崩溃了，那么就没有机会执行 EXPIRE 了，导致它一直持有这个锁。

第三个问题，也是antirez指出的，设置一个随机字符串 `my_random_value` 是很有必要的，它保证了一个客户端释放的锁必须是自己持有的那个锁。假如获取锁时 SET 的不是一个随机字符串，而是一个固定值，那么可能会发生下面的执行序列：

1. 客户端1获取锁成功。
2. 客户端1在某个操作上阻塞了很长时间。
3. 过期时间到了，锁自动释放了。
4. 客户端2获取到了对应同一个资源的锁。
5. 客户端1从阻塞中恢复过来，释放掉了客户端2持有的锁。

之后，客户端2在访问共享资源的时候，就没有锁为它提供保护了。

第四个问题，释放锁的操作必须使用Lua脚本来实现。释放锁其实包含三步操作：’GET’、判断和’DEL’，用Lua脚本来实现能保证这三步的原子性。否则，如果把这三步操作放到客户端逻辑中去执行的话，就有可能发生与前面第三个问题类似的执行序列：

1. 客户端1获取锁成功。
2. 客户端1访问共享资源。
3. 客户端1为了释放锁，先执行’GET’操作获取随机字符串的值。
4. 客户端1判断随机字符串的值，与预期的值相等。
5. 客户端1由于某个原因阻塞住了很长时间。
6. 过期时间到了，锁自动释放了。
7. 客户端2获取到了对应同一个资源的锁。
8. 客户端1从阻塞中恢复过来，执行 `DEL` 操纵，释放掉了客户端2持有的锁。

实际上，在上述第三个问题和第四个问题的分析中，如果不是客户端阻塞住了，而是出现了大的网络延迟，也有可能导致类似的执行序列发生。

前面的四个问题，只要实现分布式锁的时候加以注意，就都能够被正确处理。但除此之外，antirez还指出了一个问题，是由failover引起的，却是基于单Redis节点的分布式锁无法解决的。正是这个问题催生了Redlock的出现。

这个问题是这样的。假如Redis节点宕机了，那么所有客户端就都无法获得锁了，服务变得不可用。为了提高可用性，我们可以给这个Redis节点挂一个Slave，当Master节点不可用的时候，系统自动切到Slave上（failover）。但由于Redis的主从复制（replication）是异步的，这可能导致在failover过程中丧失锁的安全性。考虑下面的执行序列：

1. 客户端1从Master获取了锁。
2. Master宕机了，存储锁的key还没有来得及同步到Slave上。
3. Slave升级为Master。
4. 客户端2从新的Master获取到了对应同一个资源的锁。

于是，客户端1和客户端2同时持有了同一个资源的锁。锁的安全性被打破。针对这个问题，antirez设计了Redlock算法，我们接下来会讨论。

【*其它疑问*】

前面这个算法中出现的锁的有效时间(lock validity time)，设置成多少合适呢？如果设置太短的话，锁就有可能在客户端完成对于共享资源的访问之前过期，从而失去保护；如果设置太长的话，一旦某个持有锁的客户端释放锁失败，那么就会导致所有其它客户端都无法获取锁，从而长时间内无法正常工作。看来真是个两难的问题。

而且，在前面对于随机字符串 `my_random_value` 的分析中，antirez也在文章中承认的确应该考虑客户端长期阻塞导致锁过期的情况。如果真的发生了这种情况，那么共享资源是不是已经失去了保护呢？antirez重新设计的Redlock是否能解决这些问题呢？

分布式锁Redlock

由于前面介绍的基于单Redis节点的分布式锁在failover的时候会产生解决不了的安全性问题，因此antirez提出了新的分布式锁的算法Redlock，它基于N个完全独立的Redis节点（通常情况下N可以设置成5）。

运行Redlock算法的客户端依次执行下面各个步骤，来完成**获取锁**的操作：

1. 获取当前时间（毫秒数）。
2. 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。这里的失败，应该包含任何类型的失败，比如该Redis节点不可用，或者该Redis节点上的锁已经被其它客户端持有（注：Redlock原文中这里只提到了Redis节点不可用的情况，但也应该包含其它的失败情况）。
3. 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。
4. 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。
5. 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Redis Lua脚本）。

当然，上面描述的只是**获取锁**的过程，而**释放锁**的过程比较简单：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。

由于N个Redis节点中的大多数能正常工作就能保证Redlock正常工作，因此理论上它的可用性更高。我们前面讨论的单Redis节点的分布式锁在failover的时候锁失效的问题，在Redlock中不存在了，但如果有节点发生崩溃重启，还是会对锁的安全性有影响的。具体的影响程度跟Redis对数据的持久化程度有关。

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：

1. 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）。
2. 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。
3. 节点C重启后，客户端2锁住了C, D, E，**获取锁**成功。

这样，客户端1和客户端2同时获得了锁（针对同一资源）。

在默认情况下，Redis的AOF持久化方式是每秒写一次磁盘（即执行fsync），因此最坏情况下可能丢失1秒的数据。为了尽可能不丢数据，Redis允许设置成每次修改数据都进行fsync，但这会降低性能。当然，即使执行了fsync也仍然有可能丢失数据（这取决于系统而不是Redis的实现）。所以，上面分析的由于节点重启引发的锁失效问题，总是有可能出现的。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。

关于Redlock还有一点细节值得拿出来分析一下：在最后**释放锁**的时候，antirez在算法描述中特别强调，客户端应该向所有Redis节点发起**释放锁**的操作。也就是说，即使当时向某个节点获取锁没有成功，在释放锁的时候也不应该漏掉这个节点。这是为什么呢？设想这样一种情况，客户端发给某个Redis节点的**获取锁**的请求成功到达了该Redis节点，这个节点也成功执行了 SET 操作，但是它返回给客户端的响应包却丢失了。这在客户端看来，获取锁的请求由于超时而失败了，但在Redis这边看来，加锁已经成功了。因此，释放锁的时候，客户端也应该对当时获取锁失败的那些Redis节点同样发起请求。实际上，这种情况在异步通信模型中是有可能发生的：客户端向服务器通信是正常的，但反方向却是有问题的。

【*其它疑问*】

前面在讨论单Redis节点的分布式锁的时候，最后我们提出了一个疑问，如果客户端长期阻塞导致锁过期，那么它接下来访问共享资源就不安全了（没有了锁的保护）。这个问题在Redlock中是否有所改善呢？显然，这样的问题在Redlock中是依然存在的。

另外，在算法第4步成功获取了锁之后，如果由于获取锁的过程消耗了较长时间，重新计算出来的剩余的锁有效时间很短了，那么我们还来得及去完成共享资源访问吗？如果我们认为太短，是不是应该立即进行锁的释放操作？那到底多短才算呢？又是一个选择难题。

Martin的分析

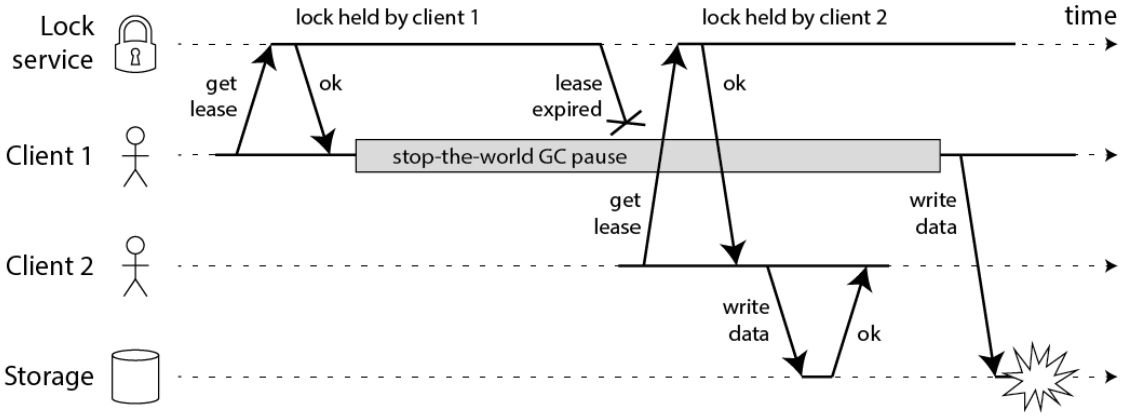
Martin Kleppmann在2016-02-08这一天发表了一篇blog，名字叫”How to do distributed locking “，地址如下：

- https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html (https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html)

Martin在这篇文章中谈及了分布式系统的很多基础性的问题（特别是分布式计算的异步模型），对分布式系统的从业者来说非常值得一读。这篇文章大体可以分为两大部分：

- 前半部分，与Redlock无关。Martin指出，即使我们拥有一个完美实现的分布式锁（带自动过期功能），在没有共享资源参与进来提供某种fencing机制的前提下，我们仍然不可能获得足够的安全性。
- 后半部分，是对Redlock本身的批评。Martin指出，由于Redlock本质上是建立在一个同步模型之上，对系统的记时假设(timing assumption)有很强的要求，因此本身的安全性是不够的。

首先我们讨论一下前半部分的关键点。Martin给出了下面这样一份时序图：



(/assets/photos_redlock/unsafe-lock.png)

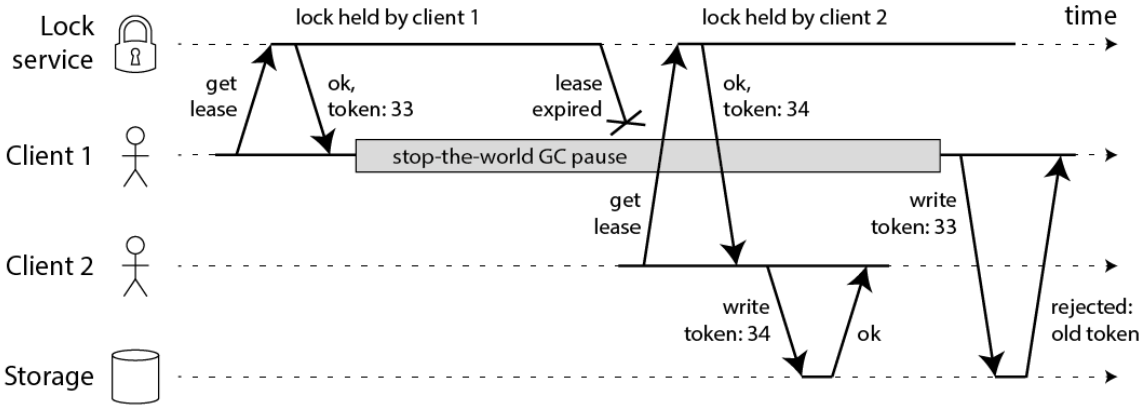
在上面的时序图中，假设锁服务本身是没有问题的，它总是能保证任一时刻最多只有一个客户端获得锁。上图中出现的lease这个词可以暂且认为就等同于一个带有自动过期功能的锁。客户端1在获得锁之后发生了很长时间的GC pause，在此期间，它获得的锁过期了，而客户端2获得了锁。当客户端1从GC pause中恢复过来的时候，它不知道自己持有的锁已经过期了，它依然向共享资源（上图中是一个存储服务）发起了写数据请求，而这时锁实际上被客户端2持有，因此两个客户端的写请求就有可能冲突（锁的互斥作用失效了）。

初看上去，有人可能会说，既然客户端1从GC pause中恢复过来以后不知道自己持有的锁已经过期了，那么它可以在访问共享资源之前先判断一下锁是否过期。但仔细想想，这丝毫也没有帮助。因为GC pause可能发生在任意时刻，也许恰好在判断完之后。

也有人会说，如果客户端使用没有GC的语言来实现，是不是就没有这个问题呢？Martin指出，系统环境太复杂，仍然有很多原因导致进程的pause，比如虚存造成的缺页故障(page fault)，再比如CPU资源的竞争。即使不考虑进程pause的情况，网络延迟也仍然会造成类似的结果。

总结起来就是说，即使锁服务本身是没有问题的，而仅仅是客户端有长时间的pause或网络延迟，仍然会造成两个客户端同时访问共享资源的冲突情况发生。而这种情况其实就是我们在前面已经提出来的“客户端长期阻塞导致锁过期”的那个疑问。

那怎么解决这个问题呢？Martin给出了一种方法，称为fencing token。fencing token是一个单调递增的数字，当客户端成功获取锁的时候它随同锁一起返回给客户端。而客户端访问共享资源的时候带着这个fencing token，这样提供共享资源的服务就能根据它进行检查，拒绝掉延迟到来的访问请求（避免了冲突）。如下图：



(/assets/photos_redlock/fencing-tokens.png)

在上图中，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。

现在我们再讨论一下Martin的文章的后半部分。

Martin在文中构造了一些事件序列，能够让Redlock失效（两个客户端同时持有锁）。为了说明Redlock对系统记时(timing)的过分依赖，他首先给出了下面的一个例子（还是假设有5个Redis节点A, B, C, D, E）：

1. 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。
2. 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。
3. 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）。
4. 客户端1和客户端2现在都认为自己持有了锁。

上面这种情况之所以有可能发生，本质上是因为Redlock的安全性(safety property)对系统的时钟有比较强的依赖，一旦系统的时钟变得不准确，算法的安全性也就保证不了了。Martin在这里其实是要指出分布式算法研究中的一些基础性问题，或者说一些常识问题，即好的分布式算法应该基于异步模型(asynchronous model)，算法的安全性不应该依赖于任何记时假设(timing assumption)。在异步模型中：进程可能pause任意长的时间，消息可能在网络中延迟任意长的时间，甚至丢失，系统时钟也可能以任意方式出错。一个好的分布式算法，这些因素不应该影响它的安全性(safety property)，只可能影响到它的活性(liveness property)，也就是说，即使在非常极端的情况下（比如系统时钟严重错误），算法顶多是不能在有限的时间内给出结果而已，而不应该给出错误的结果。这样的算法在现实中是存在的，像比较著名的Paxos，或Raft。但显然按这个标准的话，Redlock的安全性级别是达不到的。

随后，Martin觉得前面这个时钟跳跃的例子还不够，又给出了一个由客户端GC pause引发Redlock失效的例子。如下：

1. 客户端1向Redis节点A, B, C, D, E发起锁请求。
2. 各个Redis节点已经把请求结果返回给了客户端1，但客户端1在收到请求结果之前进入了长时间的GC pause。
3. 在所有的Redis节点上，锁过期了。
4. 客户端2在A, B, C, D, E上获取到了锁。

5. 客户端1从GC pause从恢复，收到了前面第2步来自各个Redis节点的请求结果。客户端1认为自己成功获取到了锁。
6. 客户端1和客户端2现在都认为自己持有了锁。

Martin给出的这个例子其实有点小问题。在Redlock算法中，客户端在完成向各个Redis节点的获取锁的请求之后，会计算这个过程消耗的时间，然后检查是不是超过了锁的有效时间(lock validity time)。也就是上面的例子中第5步，客户端1从GC pause中恢复过来以后，它会通过这个检查发现锁已经过期了，不会再认为自己成功获取到锁了。随后antirez在他的反驳文章中就指出来了这个问题，但Martin认为这个细节对Redlock整体的安全性没有本质的影响。

抛开这个细节，我们可以分析一下Martin举这个例子的意图在哪。初看起来，这个例子跟文章前半部分分析通用的分布式锁时给出的GC pause的时序图是基本一样的，只不过那里的GC pause发生在客户端1获得了锁之后，而这里的GC pause发生在客户端1获得锁之前。但两个例子的侧重点不太一样。Martin构造这里的这个例子，是为了强调在一个分布式的异步环境下，长时间的GC pause或消息延迟（上面这个例子中，把GC pause换成Redis节点和客户端1之间的消息延迟，逻辑不变），会让客户端获得一个已经过期的锁。从客户端1的角度看，Redlock的安全性被打破了，因为客户端1收到锁的时候，这个锁已经失效了，而Redlock同时还把这个锁分配给了客户端2。换句话说，Redis服务器在把锁分发给客户端的途中，锁就过期了，但又没有有效的机制让客户端明确知道这个问题。而在之前的那个例子中，客户端1收到锁的时候锁还是有效的，锁服务本身的安全性可以认为没有被打破，后面虽然也出了问题，但问题是出在客户端1和共享资源服务器之间的交互上。

在Martin的这篇文章中，还有一个很有见地的观点，就是对锁的用途的区分。他把锁的用途分为两种：

- 为了效率(efficiency)，协调各个客户端避免做重复的工作。即使锁偶尔失效了，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。比如重复发送了一封同样的email。
- 为了正确性(correctness)。在任何情况下都不允许锁失效的情况发生，因为一旦发生，就可能意味着数据不一致(inconsistency)，数据丢失，文件损坏，或者其它严重的问题。

最后，Martin得出了如下的结论：

- 如果是为了效率(efficiency)而使用分布式锁，允许锁的偶尔失效，那么使用单Redis节点的锁方案就足够了，简单而且效率高。Redlock则是个过重的实现(heavyweight)。
- 如果是为了正确性(correctness)在很严肃的场合使用分布式锁，那么不要使用Redlock。它不是建立在异步模型上的一个足够强的算法，它对于系统模型的假设中包含很多危险的成分(对于timing)。而且，它没有一个机制能够提供fencing token。那应该使用什么技术呢？Martin认为，应该考虑类似Zookeeper的方案，或者支持事务的数据库。

Martin对Redlock算法的形容是：

> neither fish nor fowl （非驴非马）

【*其它疑问*】

- Martin提出的fencing token的方案，需要对提供共享资源的服务进行修改，这在现实中可行吗？
- 根据Martin的说法，看起来，如果资源服务器实现了fencing token，它在分布式锁失效的情况下也仍然能保持资源的互斥访问。这是不是意味着分布式锁根本没有存在的意义了？

- 资源服务器需要检查fencing token的大小，如果提供资源访问的服务也是包含多个节点的（分布式的），那么这里怎么检查才能保证fencing token在多个节点上是递增的呢？
- Martin对于fencing token的举例中，两个fencing token到达资源服务器的顺序颠倒了（小的fencing token后到了），这时资源服务器检查出了这一问题。如果客户端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资源服务器，但保持了顺序，那么资源服务器是不是就检查不出问题了？这时对于资源的访问是不是就发生冲突了？
- 分布式锁+fencing的方案是绝对正确的吗？能证明吗？

（未完，故事太长，下半部待续）

其它精选文章：

- 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： 【科幻】光年之外的世界 (/posts/blog-sf-light-years-away.html)

下篇： 基于Redis的分布式锁到底安全吗（下）？ (/posts/blog-redlock-reasoning-part2.html)

栏目分类

分布式 (/posts/distributed_system.html)
散文小说 (/posts/essay.html)
移动开发 (/posts/client_dev.html)
关于我 (/about.html)

机器学习 (/posts/ml.html)
服务端技术 (/posts/server.html)
我的诗词 (/posts/poems.html)

最新文章

分布式领域最重要的一篇论文，到底讲了什么？ (/posts/blog-time-clock-ordering.html)
条分缕析分布式：因果一致性和相对论时空 (/posts/blog-distributed-causal-consistency.html)
条分缕析分布式：浅析强弱一致性 (/posts/blog-distributed-strong-weak-consistency.html)
条分缕析分布式：到底什么是一致性？ (/posts/blog-distributed-consistency.html)
由「精益创业」所想到的 (/posts/blog-lean-startup.html)
看得见的机器学习：零基础看懂神经网络 (/posts/blog-nn-visualization.html)
程序员眼中的「技术-艺术」光谱 (/posts/blog-tech-art-spectrum.html)
在技术和业务中保持平衡 (/posts/blog-tech-and-biz.html)
给普通人看的机器学习(一)：优化理论 (/posts/blog-ml-optimization.html)
卓越的人和普通的人到底区别在哪？ (/posts/blog-imagination.html)