

52 | 管理设计篇之“分布式锁”

2018-03-29 陈皓

左耳听风

[进入课程 >](#)



讲述：柴巍

时长 12:07 大小 5.55M



我们知道，在多线程情况下访问一些共享资源需要加锁，不然就会出现数据被写乱的问题。在分布式系统下，这样的问题也是一样的。只不过，我们需要一个分布式的锁服务。对于分布式的锁服务，一般可以用数据库 DB、Redis 和 ZooKeeper 等实现。不管怎么样，分布式的锁服务需要有以下几个特点。

安全性（Safety）：在任意时刻，只有一个客户端可以获得锁（**排他性**）。


避免死锁：客户端最终一定可以获得锁，即使锁住某个资源的客户端在释放锁之前崩溃或者网络不可达。

容错性：只要锁服务集群中的大部分节点存活，Client 就可以进行加锁解锁操作。

Redis 的分布式锁服务

这里提一下，避免死锁的问题。下面以 Redis 的锁服务为例（参考 [Redis 的官方文档](#)）。

我们通过以下命令对资源加锁。

 复制代码

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

解释一下：

SET NX 命令只会在 key 不存在的时候给 key 赋值，PX 命令通知 Redis 保存这个 key 30000ms。

my_random_value 必须是全局唯一的值。这个随机数在释放锁时保证释放锁操作的安全性。


PX 操作后面的参数代表的是这个 key 的存活时间，称作锁过期时间。

当资源被锁定超过这个时间时，锁将自动释放。

获得锁的客户端如果没有在这个时间窗口内完成操作，就可能会有其他客户端获得锁，引起争用问题。

这里的原理是，只有在某个 key 不存在的情况下才能设置（set）成功该 key。于是，这就可以让多个进程并发去设置同一个 key，只有一个进程能设置成功。而其它的进程因为之前有人把 key 设置成功了，而导致失败（也就是获得锁失败）。

我们通过下面的脚本为申请成功的锁解锁：

 复制代码

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

如果 key 对应的 value 一致，则删除这个 key。

通过这个方式释放锁是为了避免 Client 释放了其他 Client 申请的锁。

例如，下面的例子演示了不区分 Client 会出现的一种问题。

1. Client A 获得了一个锁。
2. 当尝试释放锁的请求发送给 Redis 时被阻塞，没有及时到达 Redis。
3. 锁定时间超时，Redis 认为锁的租约到期，释放了这个锁。
4. Client B 重新申请到了这个锁。
5. Client A 的解锁请求到达，将 Client B 锁定的 key 解锁。
6. Client C 也获得了锁。
7. Client B 和 Client C 同时持有锁。

通过执行上面脚本的方式释放锁，Client 的解锁操作只会解锁自己曾经加锁的资源，所以是安全的。

关于 value 的生成，官方推荐从 /dev/urandom 中取 20 个 byte 作为随机数。或者采用更加简单的方式，例如使用 RC4 加密算法在 /dev/urandom 中得到一个种子（Seed），然后生成一个伪随机流。

也可以采用更简单的方法，使用时间戳 + 客户端编号的方式生成随机数。Redis 的官方文档说：“这种方式的安全性较差一些，但对于绝大多数的场景来说已经足够安全了”。

分布式锁服务的一个问题

注意，虽然 Redis 文档里说他们的分布式锁是没有问题的，但其实还是很有问题的。尤其是上面那个为了避免 Client 端把锁占住不释放，然后，Redis 在超时后把其释放掉。不知道你怎么想，但我觉得这事儿听起来就有点不靠谱。

我们来脑补一下，不难发现下面这个案例。

如果 Client A 先取得了锁。

其它 Client（比如说 Client B）在等待 Client A 的工作完成。

这个时候，如果 Client A 被挂在了某些事上，比如一个外部的阻塞调用，或是 CPU 被别的进程吃满，或是不巧碰上了 Full GC，导致 Client A 花了超过平时几倍的时间。

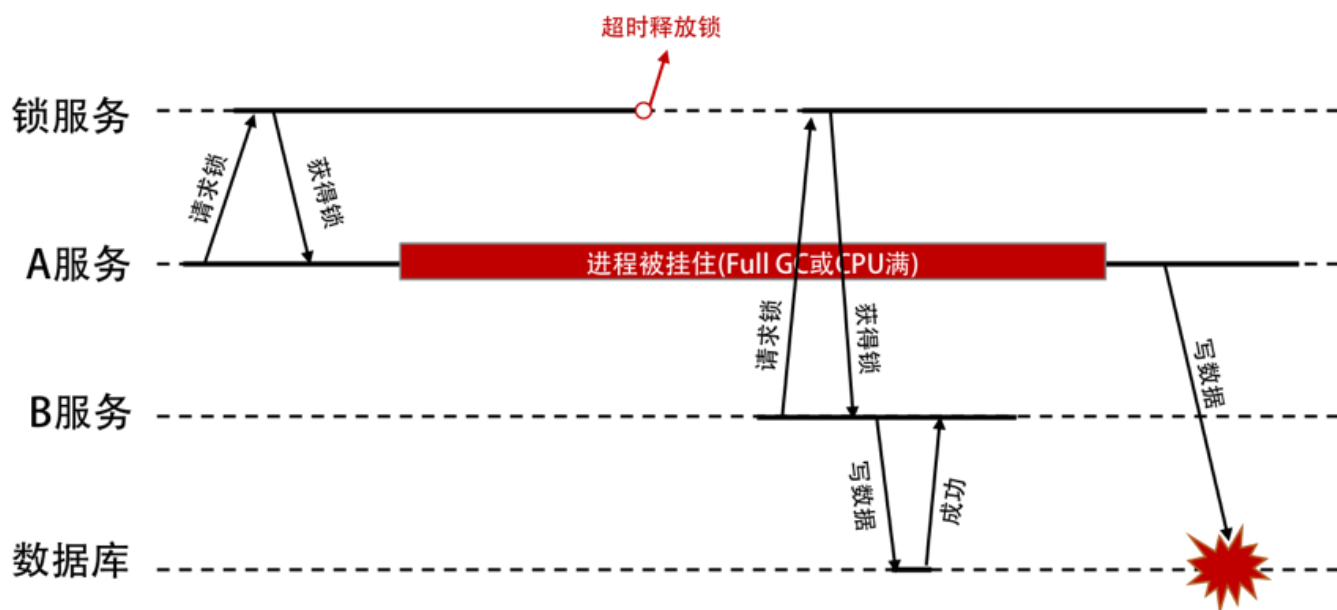
然后，我们的锁服务因为怕死锁，就在一定时间后，把锁给释放掉了。

此时，Client B 获得了锁并更新了资源。

这个时候，Client A 服务缓过来了，然后也去更新了资源。于是乎，把 Client B 的更新给冲掉了。

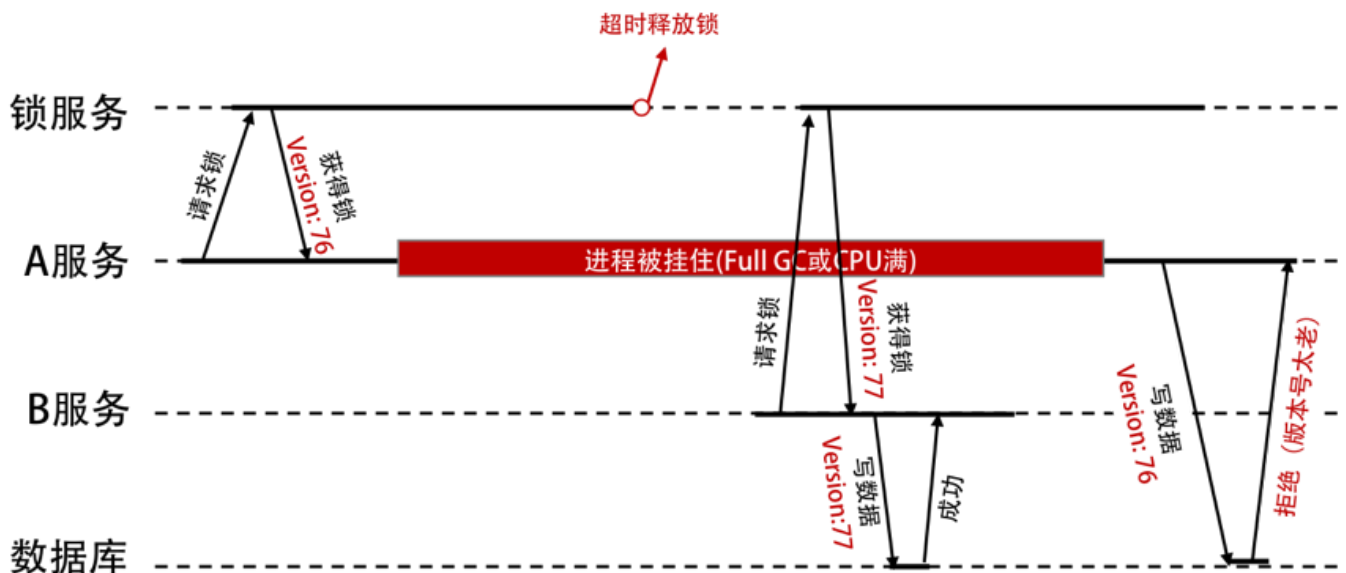
这就造成了数据出错。

这听起来挺严重的吧。我画了个图示例一下。



千万不要以为这是脑补出来的案例。其实，这个是真实案例。HBase 就曾经遇到过这样的问题，你可以在他们的 PPT ([HBase and HDFS: Understanding FileSystem Usage in HBase](#)) 中看到相关的描述。

要解决这个问题，你需要引入 fence (栅栏) 技术。一般来说，这就是乐观锁机制，需要一个版本号排它。我们的流程就变成了下图中的这个样子。



我们从图中可以看到：

锁服务需要有一个单调递增的版本号。

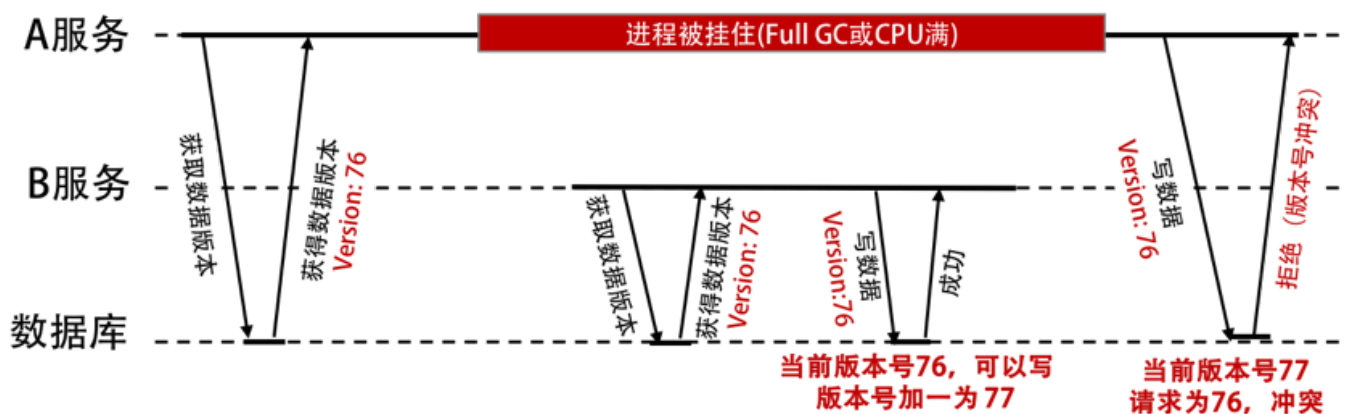
写数据的时候，也需要带上自己的版本号。

数据库服务需要保存数据的版本号，然后对请求做检查。

如果使用 ZooKeeper 做锁服务的话，那么可以使用 `zxid` 或 `znode` 的版本号来做这个 fence 版本号。


从乐观锁到 CAS

但是，我们想想，如果数据库中也保留着版本号，那么完全可以用数据库来做这个锁服务，不就更方便了吗？下面的图展示了这个过程。



使用数据版本（Version）记录机制，即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的“version”字段来实现的。当读取数据时，将 version 字段的值一同读出，数据每更新一次，对此 version 值加一。

当我们提交更新的时候，数据库表对应记录的当前版本信息与第一次取出来的 version 值进行比对。如果数据库表当前版本号与第一次取出来的 version 值相等，则予以更新，否则认为是过期数据。更新语句写成 SQL 大概是下面这个样子：


 复制代码

```
1 UPDATE table_name SET xxx = #{xxx}, version=version+1 where version =#{version};
```

这不就是乐观锁吗？是的，这是乐观锁最常用的一种实现方式。**是的，如果我们使用版本号，或是 fence token 这种方式，就不需要使用分布式锁服务了。**

另外，多说一下。这种 fence token 的玩法，在数据库那边一般会用 timestamp 时间戳来玩。也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则 OK，否则就是版本冲突。

还有，我们有时候都不需要增加额外的版本字段或是 fence token。比如，如果想更新库存，我们可以这样操作：

 复制代码

```
1 SELECT stock FROM tb_product where product_id=#{product_id};  
2 UPDATE tb_product SET stock=stock-#{num} WHERE product_id=#{product_id} AND stock=#{stock};
```

先把库存数量（stock）查出来，然后在更新的时候，检查一下是否是上次读出来的库存。如果不是，说明有别人更新过了，我的 UPDATE 操作就会失败，得重新再来。

细心的你一定发现了，这不就是计算机汇编指令中的原子操作 CAS（Compare And Swap）嘛，大量无锁的数据结构都需要用到这个。（关于 CAS 的话题，你可以看一下我在 CoolShell 上写的[无锁队列的实现](#)）。

我们一步一步地从分布式锁服务到乐观锁，再到 CAS，你看到了什么？你是否得思考一个有趣的问题——我们还需要分布式锁服务吗？

分布式锁设计的重点

最后，我们来谈谈分布式锁设计的重点。

一般情况下，我们可以使用数据库、Redis 或 ZooKeeper 来做分布式锁服务，这几种方式都可以用于实现分布式锁。

分布式锁的特点是，保证在一个集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。这就是所谓的分布式互斥。所以，大家在做某个事的时候，要去一个服务上请求一个标识。如果请求到了，我们就可以操作，操作完后，把这个标识还回去，这样别的进程就可以请求到了。

首先，我们需要明确一下分布式锁服务的初衷和几个概念性的问题。

如果获得锁的进程挂掉了怎么办？锁还不回来了，会导致死锁。一般的处理方法是在锁服务那边加上一个过期时间，如果在这个时间内锁没有被还回来，那么锁服务要自动解锁，以避免全部锁住。

如果锁服务自动解锁了，新的进程就拿到锁了，但之前的进程以为自己还有锁，那么就出现了两个进程拿到了同一个锁的问题，它们在更新数据的时候就会产生问题。对于这个问题，我想说：

像 Redis 那样也可以使用 Check and Set 的方式来保证数据的一致性。这就有点像计算机原子指令 CAS (Compare And Swap) 一样。就是说，我在改变一个值的时候先检查一下是不是我之前读出来的值，这样来保证其间没有人改过。

如果通过像 CAS 这样的操作的话，我们还需要分布式锁服务吗？的确是不需要了，不是吗？

但现实生活中也有不需要更新某个数据的场景，只是为了同步或是互斥一下不同机器上的线程，这时候像 Redis 这样的分布式锁服务就有意义了。

所以，需要分清楚：我是用来修改某个共享源的，还是用来不同进程间的同步或是互斥的。如果使用 CAS 这样的方式（无锁方式）来更新数据，那么我们是不需要使用分布式锁服务

的，而后者可能是需要的。所以，这是我们在决定使用分布式锁服务前需要考虑的第一个问题——我们是否需要？

如果确定要分布式锁服务，你需要考虑下面几个设计。

需要给一个锁被释放的方式，以避免请求者不把锁还回来，导致死锁的问题。Redis 使用超时时间，ZooKeeper 可以依靠自身的 sessionTimeout 来删除节点。

分布式锁服务应该是高可用的，而且是需要持久化的。对此，你可以看一下 [Redis 的文档 RedLock](#) 看看它是怎么做到高可用的。

要提供非阻塞方式的锁服务。

还要考虑锁的可重入性。

我认为，Redis 也是不错的，ZooKeeper 在使用起来需要有一些变通的方式，好在 Apache 有 [Curator](#) 帮我们封装了各种分布式锁的玩法。

小结

好了，我们来总结一下今天分享的主要内容。首先，我介绍了为什么需要分布式锁。就像单机系统上的多线程程序需要用操作系统锁或数据库锁来互斥对共享资源的访问一样，分布式程序也需要通过分布式锁来互斥对共享资源的访问。

分布式锁服务一般可以通过 Redis 和 ZooKeeper 等实现。接着，以 Redis 为例，我介绍了怎样用它来加锁和解锁，由此引出了锁超时后的潜在风险。我们看到，类似于数据库的乐观并发控制，这种风险可以通过版本号的方式来解决。

进一步，数据库如果本身利用 CAS 等手段支持这种版本控制方式，其实也就没必要用一个独立的分布式锁服务了。最后，我们发现，分布式锁服务还能用来做同步，这是数据库锁做不了的事情。下篇文章中，我们将聊聊配置中心相关的技术，希望对你有帮助。

也欢迎你分享一下你在留言区给我分享下哪些场景下你会用到锁？你都用哪种平台的锁服务？有没有用到数据库锁？是 OCC，还是悲观锁？如果是悲观锁的话，你又是怎样避免死锁的？

我在这里给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。

弹力设计篇

[认识故障和弹力设计](#)

[隔离设计 Bulkheads](#)

[异步通讯设计 Asynchronous](#)

[幂等性设计 Idempotency](#)

[服务的状态 State](#)

[补偿事务 Compensating Transaction](#)

[重试设计 Retry](#)

[熔断设计 Circuit Breaker](#)

[限流设计 Throttle](#)

[降级设计 degradation](#)

[弹力设计总结](#)

管理设计篇

[分布式锁 Distributed Lock](#)

[配置中心 Configuration Management](#)

[边车模式 Sidecar](#)

[服务网格 Service Mesh](#)

[网关模式 Gateway](#)

[部署升级策略](#)

性能设计篇

[缓存 Cache](#)

[异步处理 Asynchronous](#)

[数据库扩展](#)

[秒杀 Flash Sales](#)

[边缘计算 Edge Computing](#)

左耳朵耗子

全年独家专栏《左耳听风》

20000 名程序员的练级攻略

陈皓

资深技术专家
骨灰级程序员



新版升级：点击「👤 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 51 | 弹力设计篇之“弹力设计总结”

下一篇 53 | 管理设计篇之“配置中心”

精选留言 (22)

写留言



Randy

2018-04-19

14

现实生活中也有不需要更新某个数据的场景，只是为了同步或是互斥一下不同机器上的线程，这时候像 Redis 这样的分布式锁服务就有意义了

这句没明白，如果要进行互斥或同步操作，那就是要对同一个资源进行写操作，如果只是读操作那就不需要锁保护了，那分布式锁的意义是什么？

展开



林子

2018-07-09

6

如果用cas方式或者叫乐观锁来修改数据库中表（共享资源），会出现脏读问题，耗子叔，这点没提到。



天天向上

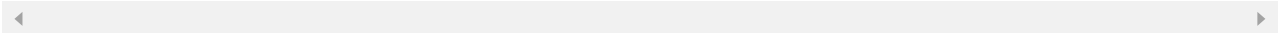
2018-05-08

👍 3

分布式锁 应该是 先获取锁 再进行业务操作 属于悲观锁 而用乐观锁代替 又演变为cas代替 这样合适吗？ 其实悲观和乐观 核心是面对的并发度不一样，如果在大并发下用乐观锁 应该失败的几率会增大，用悲观锁避免大量失败，但是会block！麻烦耗子哥 指导指导

展开 ∨

作者回复: 一切都是trade-off。不过实际情况下，乐观加上重试机制会好一些。



华煜

2018-04-19

👍 3

数据库用timestamp的判断是否冲突是有风险的吧

展开 ∨



董泽润

2018-06-21

👍 2

<https://github.com/dongzerun/dlock>

这是我在用的，redis lua 实现，和耗子叔的相似



王磊

2018-06-08

👍 1

皓哥，有个问题想确认下，redlock是要求各个节点独立部署，都是master,那么这样的部署方式是否就限定这N台Redis不能同时作为缓存服务器了？

展开 ∨



约书亚

2018-04-19

👍 1

皓哥，这篇文章前半段一直到cas之前基本是在说redlock，后面说到fence和cas，恰恰是redlock争论当中反方的观点---如果你想锁的资源，能提供给你cas功能，那还要分布式锁干嘛？这也是我的疑问，我觉得是悖论

在我使用consul时，我发现，如果我要锁住一个资源，理论上100%安全的必要条件是，我

的资源就是那个锁本身，在consul就是那个资源只能是锁住key对应的value。consul本...
展开 ▾

作者回复: 这篇文章其实我是在不断地变换思路解决问题，不能说Redlock没用，其至少可以同步不同的client。而你的理解是对的，如果有共享资源，最好是在那个资源上提供无锁实现。



Geek_9ed47...

2019-04-18



用过redis和etcd的分布式锁，用分布式锁的场景有两个，一个是定时任务，一个是阻止客户端的重复提交。



靠人品去赢

2019-04-15



分布式锁有点像并发编程

展开 ▾



小新是也

2018-11-18



锁还是感觉zk专业一点

展开 ▾



格瑞图

2018-11-16



另外，多说一下。这种 fence token 的玩法，在数据库那边一般会用 timestamp 时间戳来玩。时间戳吧？



周星星

2018-07-15



比方说对余额加减可以使用CAS来实现，对于那种每人只能领取一次的优惠活动就要用锁来对每个用户做同步互斥操作因为这里除了资源的修改还有业务上的判断

展开 ▾



tiger

2018-07-10



MySQL的事务在第一个事务获取共享锁之后，再获取排他锁的过程中，其它的事务也在等待获取排他锁，于是产生了死锁，这个可以通过重入锁解决，为什么MySQL没有这么做呢？



polk

2018-07-04



首先不应该把锁的压力给数据库，数据库的性能相比redis差很多。
然后redis的那个潜在问题，把锁的timeout设置成http的timeout，不就可以解决了。
我的理解是否有问题？

展开 ∨



LI

2018-05-14



分布式锁的概念和单机的比较，需要网络超时

展开 ∨



黑小子在路...

2018-04-27



老师，如果作为服务提供方，某个接口出现异常，或者说这个接口响应时间长，怎么处理



极客er

2018-04-21



锁的可重入，耗子哥讲讲啊

展开 ∨



programath

2018-04-20



，我们发现，分布式锁服务还能用来做同步，这是数据库锁做不了的事情。

这里的同步指的是什么？为什么数据库锁做不了？请问能详细说下吗？谢谢

展开 ∨



prajba

2018-04-19



去年参与的自动化平台项目里用到了ZooKeeper悲观锁，回想起来当时并没有特别关注死锁问题，实现方式牺牲了故障情况下的一致性。

展开 ∨



流迷的咸菜

2018-04-19



文中说的RedLock应该是单节点的lock的实现吧？分布式的应该如下吧？

1. It gets the current time in milliseconds.
2. It tries to acquire the lock in all the N instances sequentially, using the same key name and random value in all the instances. During step 2, when setting the lock in each instance, the client uses a timeout which is small compared to the total...

展开 ∨