



下载APP

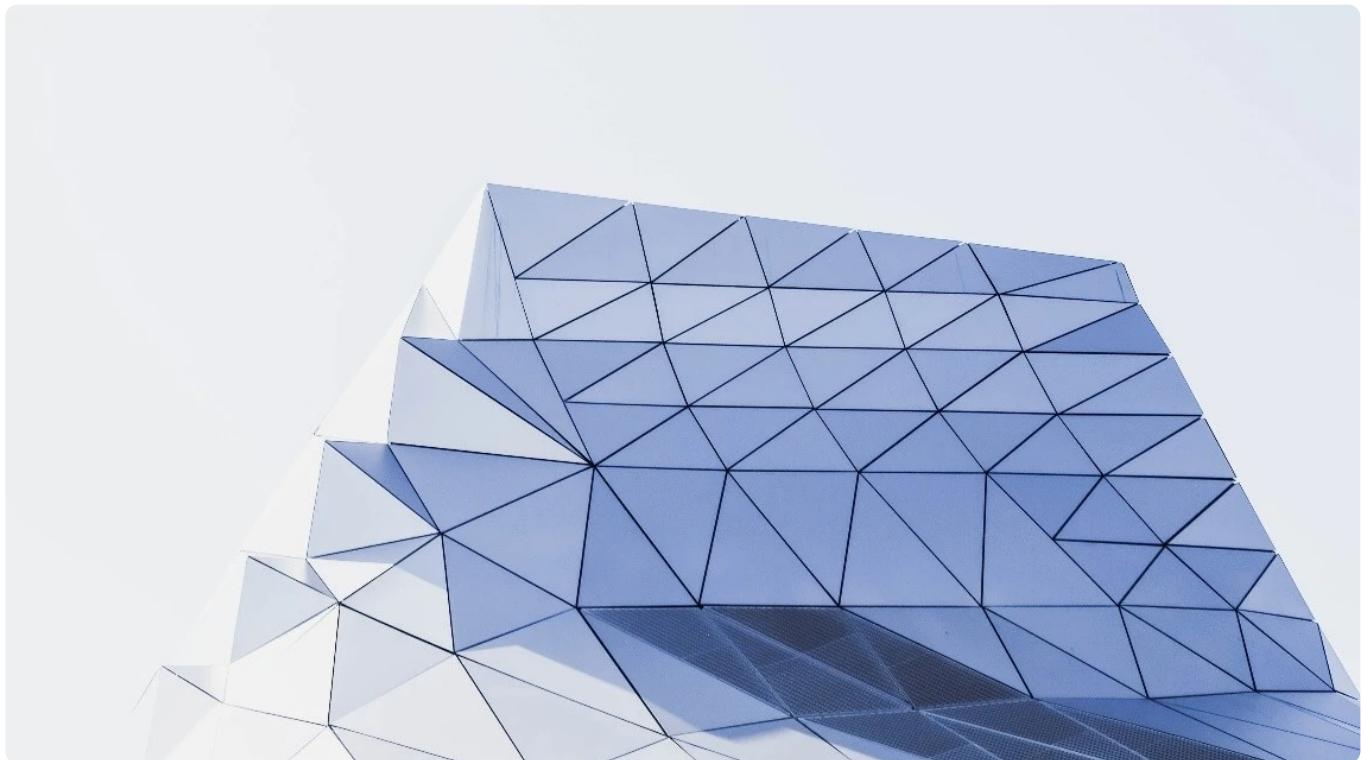


14 | 分布式锁Chubby（三）：移形换影保障高可用

2021-10-22 徐文浩

《大数据经典论文解读》

课程介绍 >



讲述：徐文浩

时长 25:49 大小 23.65M



你好，我是徐文浩。

过去的两讲里，我们都在尝试做一件事情，就是在 Master 和 Backup Master 之间保持数据的同步复制。无论是通过分布式事务的两阶段提交算法，还是通过分布式共识的 Paxos 算法，都是为了做到这一点。

而我们要去保障 Master 和 Backup Master 之间的同步复制，也是为了一个小小的目标，那就是**整个系统的高可用性**。因为系统中只有一个 Master 节点，我们希望能够在 Master 节点挂掉的时候，快速切换到另外一个节点，所以我们需要这两个节点的数据是完全相同的。不然的话，我们就可能会丢失一部分数据。



不过，无论是 GFS 也好，Bigtable 也好，我们能看到它们都是一个单 Master 系统，而不是有多个 Master，能够同时接受外部的请求来保持高可用性。所以，尽管在论文里面，Google 没有说 GFS 在 Master 和 Backup Master 之间数据的同步复制是怎么进行的，但是根据我的推测，采用一个两阶段提交的方式会更简单直接一点。

那么，现在你可能就会觉得有问题了：如果还是使用两阶段提交这样的方式，我们不还是会面临单点故障吗？而且，我们上一讲所说的 Paxos 算法也用不上啊？

要回答这个问题，就请你一起来和我学习今天这一讲，也就是 Chubby 这个系统到底是怎么一回事儿。通过这一讲，我会让你知道：

Chubby 这个分布式锁系统是怎么一回事儿，它和 Paxos 算法的关系是什么；

GFS 和 Bigtable 这样的系统，是如何通过 Chubby 来保障可用性和一致性的；

在系统设计层面，如何尽可能在设计上降低长期协同开发的成本。

好，下面就让我们一起来看看，Chubby 这个系统是怎么一回事吧。

通过 Chubby 转移可用性和“共识”问题

无论是 GFS 还是 Bigtable，其实都是一个单 Master 的系统。而为了让这样的系统保障高可用性，我们通常会采用两个策略。

第一个，是对它进行**同步复制**，数据会同步写入另外一个 Backup Master，这个方法最简单，我们可以用两阶段提交来解决。第二个，是对 **Master 进行监控**，一旦 Master 出现故障，我们就把它切换到 Backup Master 就好了。

但我们之前也说过，这里面有两个问题，首先是两阶段提交也有单点故障，其次是监控程序怎么去判断 Master 是真的挂掉了，还是只是监控程序和 Master 之间的网络中断了呢？

其实，解决这两个问题的答案，就是 Chubby，也就是 Paxos 算法。Chubby 具体的技术实现并不简单，但是思路却非常简单。那就是，我们的“共识”并不需要在每一个操作、每一条日志写入的时候发生，我们只需要有一个“共识”，确认哪一个是 Master 就好了。

这样，系统的可用性以及容错机制，就从原先的系统里被剥离出来了。我们原先的系统只需要 Master，即使是做同步数据复制，也只需要通过两阶段提交这样的策略就可以了。而一旦出现单点故障，我们只需要做一件事情，就是把故障的节点切换到它同步备份的节点就行。

而我们担心的，系统里会有两个 Master 的问题，通过 Paxos 算法来解决就好了。

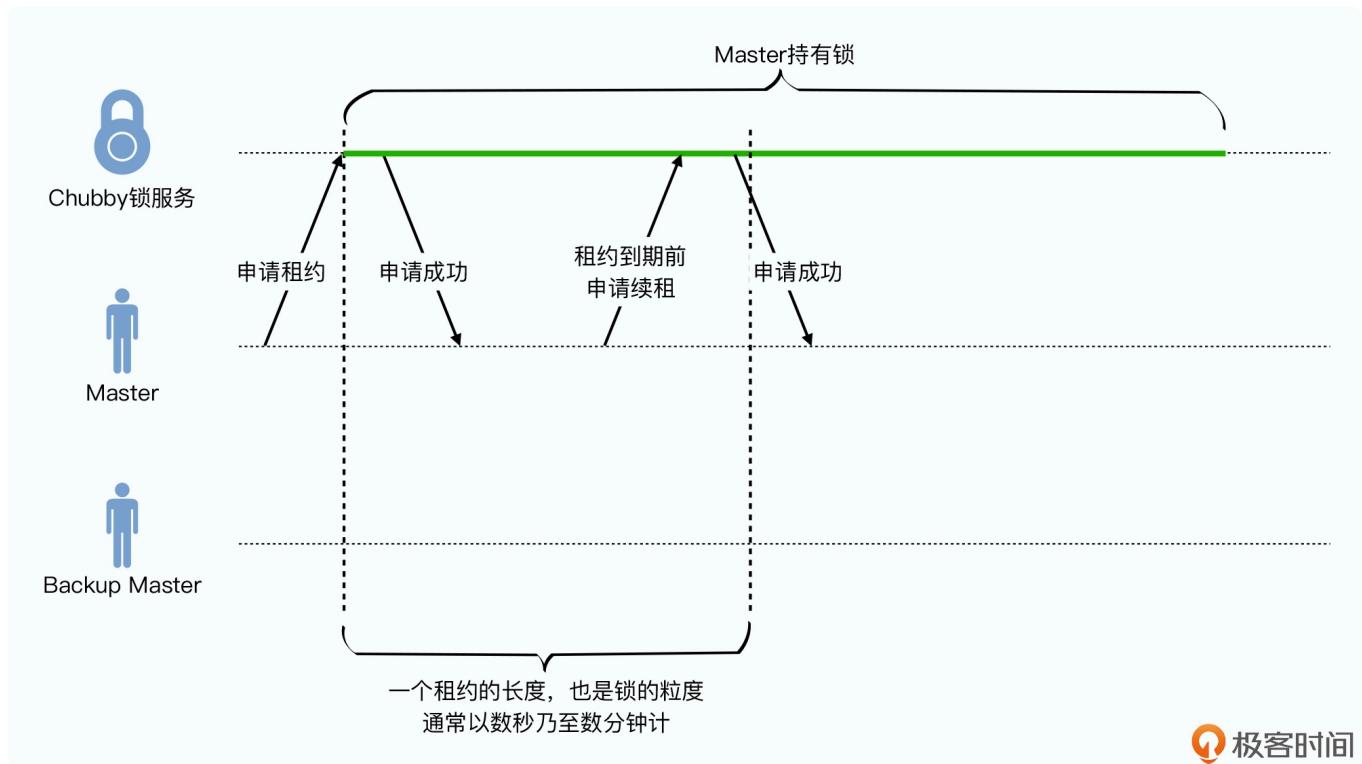
只要通过 Paxos 算法，让一个一致性模块达成共识，当前哪一个是 Master 就好，其他的节点都通过这个一致性模块，获取到谁是最新的 Master 即可。而且，这个一致性模块本身会有多个节点，比如 5 个节点，另外我们在部署的时候，还会把它们放到不同的机架上，也就是不同的交换机下。这样一来，一致性模块里单个节点出现故障，也并不会影响这个一致性模块对外提供服务。

那么，在 Chubby 这个系统里，它其实针对 Paxos 做了封装，把对外提供的接口变成一个锁。这样，Chubby 就变成了一个通用的分布式锁服务，而不是一个 Paxos 的一致性模块。在锁服务下达成的共识，就不是谁是 Master 了，而是哪一台服务器持有了 Master 的锁。对于应用系统来说，谁持有 Master 的锁，我们就认为这台服务器就是 Master。

事实上，把 Chubby 对外暴露的接口，变成一个分布式锁服务，Google 是经过深思熟虑的。对于锁服务来说，大部分工程师都是非常熟悉的，无论是 MySQL 这样的关系数据库，还是 Redis 这样的 KV 数据库，或者是 Java 标准库里的 Lock 类，都是我们经常使用的开发工具。

但是 Paxos 算法，大部分工程师可能都没有听说过。我们之前在 [MapReduce 的论文](#) 里就说过，**一个好的分布式系统的设计，就是要让使用系统的开发人员意识不到分布式的存在。** 那么，通过把 Paxos 协议封装成一个分布式锁，就可以让所有开发人员立刻上手使用，而不需要让每个人都学习 Paxos 和共识问题，从而能够更容易地在整个组织的层面快速开发出好用的系统。

而且，Chubby 这个锁服务，是一个**粗粒度的锁服务**。所谓粗粒度，指的是外部客户端占用锁的时间是比较长的。比如说，我们的 Master 只要不出现故障，就可以一直占用这把锁。但是，我们并不会用这个锁做很多细粒度的动作，不会通过这个分布式的锁，在 Bigtable 上去实现一个多行数据写入的数据库事务。



Chubby的锁，是一个粗粒度锁，持有锁的时间，可以以秒乃至分钟来计

这是因为，像 Master 的切换这样的操作，发生的频率其实很低。这就意味着，Chubby 负载也很低。而像 Bigtable 里面的数据库事务操作，每秒可以有百万次，如果通过 Chubby 来实现，那 Chubby 的负载肯定是承受不了的。要知道，Chubby 的底层算法，也是 Paxos。我们上一讲刚刚一起来了解过这个算法，它的每一个共识的达成，都是需要通过至少两轮的 RPC 协商来完成的，性能肯定跟不上。

事实上，在 Bigtable 里，Chubby 也主要是被用来做四件事情，第一个是 Master 的高可用性切换；第二个是存储引导位置（Bootstrap Location），让客户端能够找到 METADATA 数据的存储位置；第三个是 Tablet 和 Tablet Server 之间的分配关系；最后一个 Bigtable 里表的 Schema。可以看到，Chubby 存储的这些数据都是很少变化，但是一旦丢失就会导致数据不一致的元数据。

那么相信到这里，你对 Chubby 在整个分布式系统中的作用应该就弄明白了。Chubby 并不是提供一个底层的 Paxos 算法库，然后让所有的 GFS、Bigtable 等等，基于 Paxos 协议来实现数据库事务。而是把自己变成了一个分布式锁服务，主要解决 GFS、Bigtable 这些系统的元数据的一致性问题，以及容错场景下的灾难恢复问题。

GFS 和 Bigtable 这些系统，仍然会采用单个 Master，然后对数据进行分区的方式，来提升整个系统的性能容量。但是在关键的元数据管理，以及 Master 节点挂掉切换的时候，

会利用 Chubby 这个分布式锁服务，来确保整个分布式系统是有“共识”的，避免出现多个人都说自己是 Master 这样“真假美猴王”的情况出现。

Chubby 的系统架构

理解完了 Chubby 在整个分布式系统中的作用，我们下面就来深入看一下，整个 Chubby 的系统是怎么样的。我们会一起来了解下 Chubby 的系统架构、Chubby 对外提供的接口，以及看看具体这个分布式锁是如何使用的。

Chubby 的系统架构

首先，Chubby 这个系统也是有 Master 的。

在 Chubby 里，它自己的多个节点，会先通过“共识”算法，确认一个 Master 节点。这个 Master 节点，会作为系统中唯一的一个提案者（Proposer），所有对于 Chubby 的写入数据的请求，比如获取某个锁，都会发送到这个 Master 节点，由它作为提案者发起提案，然后所有节点都会作为接受者来接受提案达成共识。

只有一个提案者带来的好处就是，大部分时间，我们不太会因为两个 Proposer 之间竞争提案，而导致需要很多轮协商才能达成一致的情况。

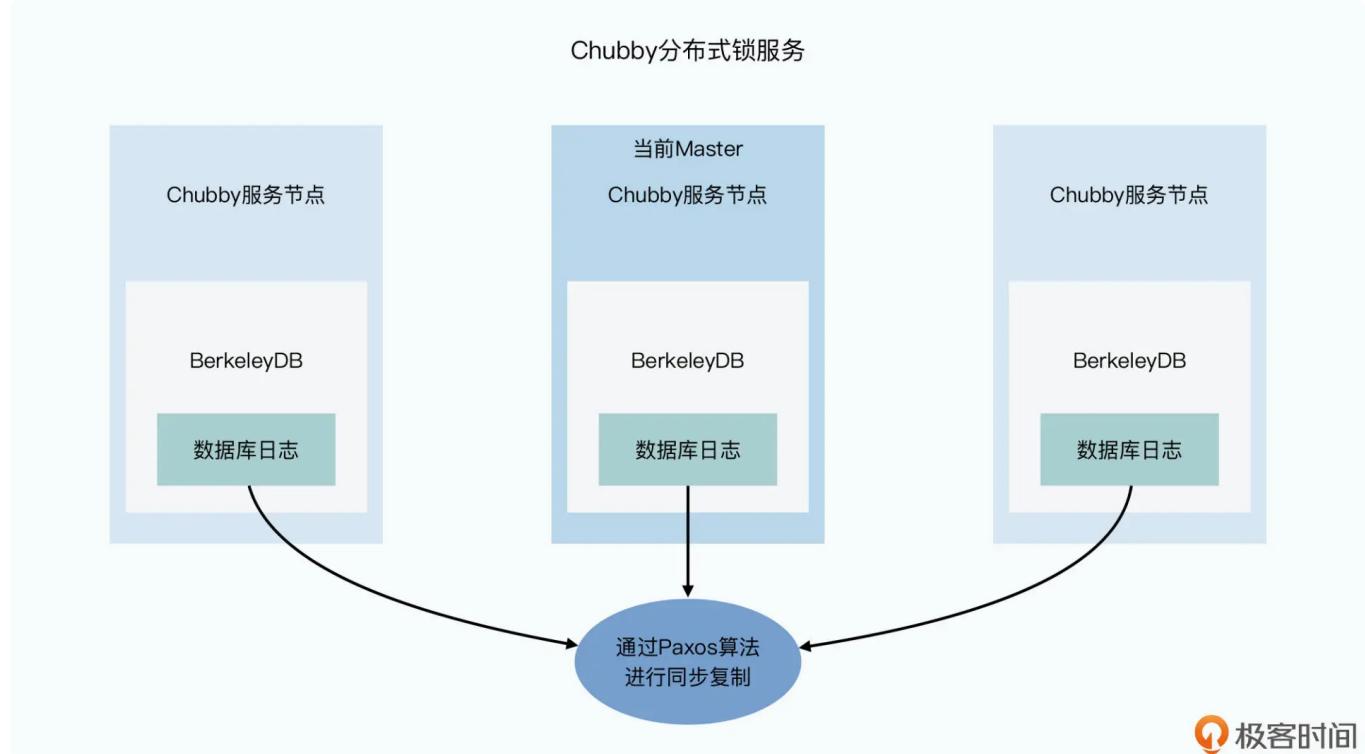
那看到这里你可能会问了，如果 Chubby 的 Master 挂掉了怎么办呢？

不要紧，我们可以通过剩下的节点，**通过共识算法再找一个 Master 出来**。而且如果是因为网络故障，导致有两个 Master 的话，也会很快通过共识算法确定一个 Master 出来。另外，两个 Master 其实只是一致性模块里的两个提案者，即使两边都接受外部请求，也都会通过共识算法，只选择一个值出来。

在论文里面，Master 的生命周期被称之为**租期**（lease）。也就是说，Master 发起的共识算法达成的共识，是在一段时间 T 之内，它是 Master。而只要 Master 不崩溃，一般它都会在这个 T 的时间到期之前，进行续租。而如果 Master 崩溃了，当 T 的时间到了，那么所有节点都可以发起自己是 Master 的一次提案，最终确认一个新的 Master。

可以看到，虽然 Paxos 这样的共识算法其实是不需要单一的 Master 节点的。但是为了实际应用中的效率问题，我们会采用选举出一个 Master 的办法，来让整个系统更加简化。

对于 Chubby 的整个服务器端来说，我们可以把它看成一个三层的系统。最底层，是一个 Paxos 协议实现的同步日志复制的系统，也就是我们上一讲所说的状态机复制的系统。上面一层，就是通过这个状态机实现的数据库了，Google 是直接采用了 BerkeleyDB 作为这个数据库。换句话说，Chubby 是通过 Paxos 在多个 BerkeleyDB 里，实现数据库的同步复制。在 BerkeleyDB 之上，才是由 Chubby 自己实现的锁服务。

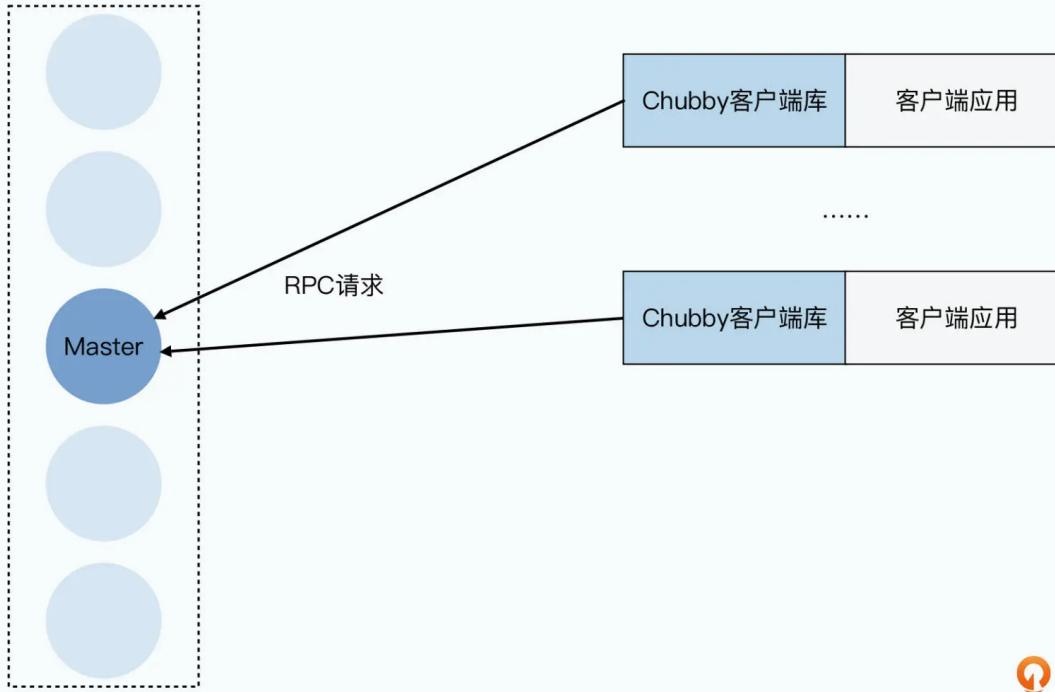


Paxos 算法，就是用来同步复制 BerkeleyDB 里的日志

除了服务器端，Chubby 还给所有想要使用 Chubby 的应用提供了一个客户端。客户端会通过 DNS 拿到所有的 Chubby 的服务端的节点，然后可以去里面任何一个节点询问，哪一个是 Master。无论是读还是写的请求，客户端都是通过访问 Master 来获取的。

对于数据写入的请求，Master 会作为刚才我们说过的提案者，在所有的 Chubby 服务器节点上通过 Paxos 算法进行同步复制。而对于读请求，Master 直接返回本地数据就好，因为所有服务器节点上的数据是有共识的。

5个节点组成的Chubby服务



极客时间

来自论文中的系统架构图

Chubby 对外提供的接口

既然 Chubby 的底层存储系统，是 BerkeleyDB 这样一个 KV 数据库，那么我们就可以通过它做很多事情了。Chubby 对外封装的访问接口，是一个类似于 Unix 文件系统的接口。使用这个形式，同样也降低了使用 Chubby 的用户的门槛。毕竟每个工程师都熟悉用 ls 命令，去查询目录下的子目录和文件列表。

Chubby 里的每一个目录或者文件，都被称之为一个节点（node）。**外部应用所使用的分布式“锁”，其实就是锁在这个节点上。**哪个客户端获得了锁，就可以向对应的目录或者文件里面写入数据。比如谁是真正的 Master，就是看谁获得了某个特定的文件锁。

举个例子，我们可以定义 /gfs/master 这个命名空间，就用来存放 Master 的相关信息。这样，Master 服务器会通过 RPC 锁住这个文件，然后往里面写下自己的 IP 地址以及其他相关的元数据就好了。而其他客户端在这个时候，就无法获得这个锁，自然也就无法把 Master 改成自己。所有想要知道谁是 Master 的客户端，就只需要去查询 /gfs/master 这个文件就行。

而所有的这些其实是目录和文件的“节点”，在 Chubby 中会分成**永久**（permanent）节点和**临时**（ephemeral）节点两种。

对于永久节点来说，客户端需要显式地调用 API，才能够删除掉。比如 Bigtable 里面，Chubby 存放的引导位置的信息，就肯定应该使用永久节点。而临时节点，则是一旦客户端和服务器的 Session 断开，就会自动消失掉。一个比较典型的使用方式，就是我们在 Bigtable 中所说的 [Tablet Server 的注册](#)。

我们可以用一个 Chubby 里面的目录 /bigtable/tablet_servers，来存放所有上线的 Tablet Server，每个 Tablet Server 都可以在里面创建一个文件，比如 /bigtable/tablet_servers/ts1、/bigtable/tablet_servers/ts2... /bigtable/tablet_servers/ts100 这样排列下去。

Tablet Server 会一直和 Chubby 之间维护着一个会话，一旦这个会话结束了，那么对应的节点会被自动删除掉，也就意味着这个节点下线无法使用了。这样，由于网络故障导致的 Tablet Server 下线，也会表现为会话超时，由此一来，它就很容易在 Chubby 这样的服务里面实现了。

回顾一下我们之前讲过的 Bigtable 的论文，Bigtable 的 Master 一旦发现这种情况，就会尝试去 Chubby 里面获取这个节点对应的锁，如果能够获取到，那么说明 Master 到 Chubby 的网络没有问题，Master 就会认为是 Tablet Server 节点下线了，它就要去调度其他的 Tablet Server，去承接这个下线了的服务器之前服务的那些 tablet。Master 只需要监控 /bigtable/tablet_servers 这个目录，就能够知道线上有哪些 Tablet Server 可供使用了。

而为了减少 Chubby 的负载，我们不希望所有想要知道 Chubby 里面节点变更的客户端，都来不断地轮询查询各个目录和文件的最新变更。因为 Chubby 管理的通常是各种元数据，这些数据的变更并不频繁。所以，Chubby 实现了一个**事件通知的机制**。

这是一个典型的设计模式中的**观察者模型（observer pattern）**。客户端可以注册它自己对哪些事件感兴趣，比如特定目录或者文件的内容变更，或者是某个文件或者目录被删除了，一旦这些事件发生了，Chubby 就会推送对应的事件信息给这些应用客户端，应用客户端就可以去做类似于调度 Tablet Server 这样的操作了。

事件	应用案例
文件内容变更	Bigtable的Master写入到/bigtable/master，所有的Tablet Server都可以注册监听这个事件，一旦Master变更，其他Tablet Server就和新的Master建立TCP连接和网络心跳。
节点删除	各种各样的服务发现机制，都会关心这个事件。我们的服务只要启动了，就会在Chubby里创建一个临时节点，一旦服务进程自己挂掉，那么这个节点会自动删除。监控程序也好，需要访问服务的客户端也好，都会收到对应的通知，进行后续处理。
子节点新增、删除	Bigtable所有的Tablet Server，都注册到/bigtable/tablet_servers目录下，Master会监听这个目录的子节点新增、删除的操作。一旦有新增的节点，或者有节点删除，Master都会收到通知，去进行tablets的调度。
Master节点转移	Chubby自身的Master节点转移，所有和Chubby建立会话的客户端都会收到这个通知，之后的读写请求会访问新的Master。



Chubby支持监听的事件类型

Chubby 作为分布式锁的挑战

现在我们知道，每一个 Chubby 的目录或者文件，就是一把锁。那么是不是我们有了锁之后，分布式共识的问题就被解决了呢？如果你是这么想的，那么你肯定还没有在网络延时上吃到足够的亏。

首先，作为分布式锁，客户端去获取的锁都是有时效的，也就是它只能占用这个锁一段时间。这个和我们前面提到的 Chubby 的 Master 的“租约”原理类似，**主要是为了避免某个客户端获取了锁之后，它因为网络或者硬件原因下线了。**

这样乍一听起来，我们只要给锁的时间设置一个时效就好了。不过，一旦涉及到不可靠的网络，事情就没有那么简单了。

Chubby 的论文里给出了这样一种情况，我们可以一起来看一下：

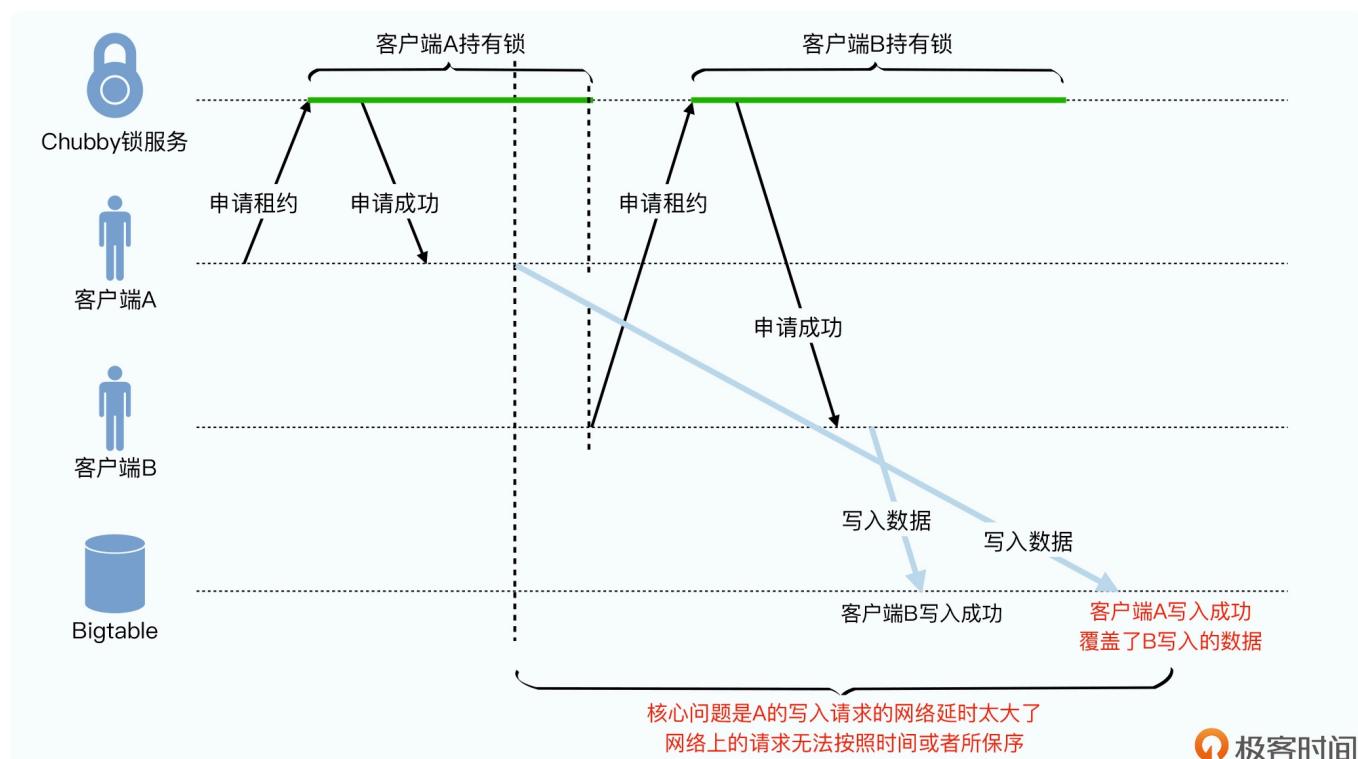
我们有一个应用的客户端 A，获取了某个 Chubby 里面的锁，比方说 /chubby/geektime 这个文件。A 对这个节点的租期呢，是一段时间 T。而这个锁，是告诉我们这个客户端可以往外部的 Bigtable 数据库的一个 geektime 的行，写入数据。在获取到了锁之后，过了一小段时间，A 仍然还持有这个锁，于是 A 就向 Bigtable 发起一个请求 X，想要往 geektime 这个行里面去写入数据。

但是这个时候，可能 A 和 Bigtable 之间的网络非常拥堵，这个请求花了比较长的时间才到达 Bigtable。

而当这个写入请求 X 还在路上的时候，客户端 A 的“租约”到期了。这个时候，另外一个客户端 B 获取到了对应的锁，然后它往这个 Bigtable 的 geektime 的行里，写入了数据 Y。

当 Y 被写入之后，请求 X 才到了 Bigtable。但是 Bigtable 并不知道谁拥有锁，它只会认为应用层面已经通过锁，实现了对于资源的保护。那么，之前客户端 A 的数据会覆盖掉客户端 B 写入的数据。但是这个情况肯定是我们不愿意接受的，因为对于客户端 B 来说，我明明已经持有了锁，为什么我写入的数据会被此时此刻没有锁的人覆盖掉呢？而且，客户端 A 的数据也是更早版本的数据。

这个就好像你租了一个仓库，租期是一年。你呢，在租期快要到期的时候，向仓库里发了一批货。但是因为物流延误，货在路上耽搁了。当你的仓库租期到期的时候，房东把仓库租给了别人，别人也已经往仓库里面放了他自己的货物。而这个时候，你的货到了，但是因为仓库的门卫并不知道房东把仓库租给了谁，它不会检查你是不是还租着仓库，就直接把你的货物也入库了，而把新的租客的货物给“覆盖”掉了。



因为网络延时，并不是获取到锁，写入数据就能保障我们期望的线性一致的表现的了

当然，Chubby 也解决了这个问题，它主要是通过两种方式来解决的。

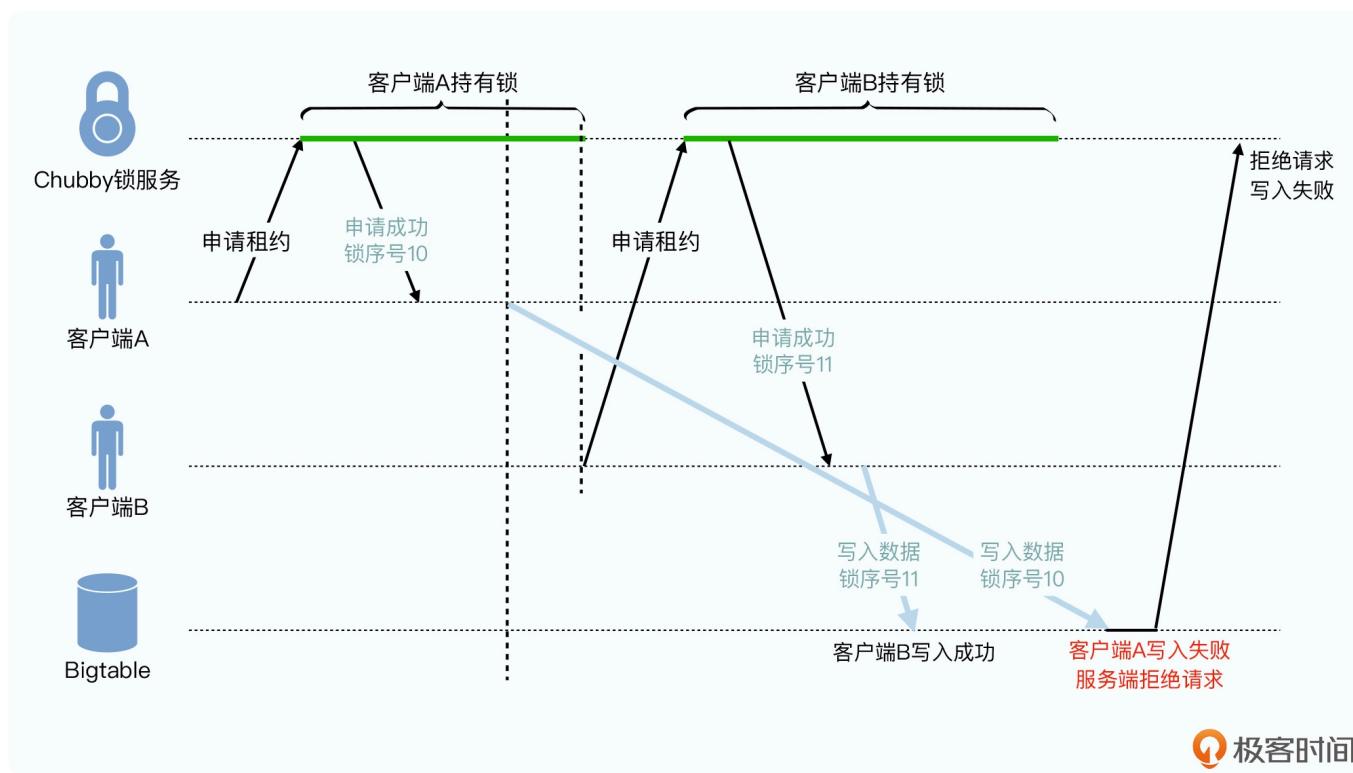
首先是锁延迟 (lock-delay)

也就是当客户端 A 的“租约”不是正常到期由客户端主动释放的话，它会让客户端继续持有这个锁一段时间。这很好理解，如果是客户端主动释放的话，意味着它已经明确告诉 Chubby，我不会再往里面写入数据。而没有主动释放，很有可能是还有请求在网络上传输，我们就再稍微等一会儿。

而如果等一会儿还是没有过来，那么 Chubby 就会再把锁释放掉。这个就好像你在现实生活中租房子，租约要到期了，如果你是主动和房东说你不再续租了，房东自然可以立刻租给别人。但是可能你因为出差或者疫情隔离，没有来得及和房东沟通，房东也会善意地多等你几天，直到一段时间之后你还是失去联系了，才再会去租给别人。

其次是锁序列器 (lock-sequencer)

它本质上是一个 **乐观锁**，或者在很多地方也叫做 Fencing 令牌。这种方式是这样的：客户端在获取 Chubby 的锁的时候，就要拿到对应的锁的序号，比如说 23。在发送请求的时候，客户端会带上这个序号。而当 Chubby 把锁给了别的客户端之后，对应的锁的序号会变大，变成了 24。而我们对应的业务服务，比如 Bigtable 呢，也要记录每次请求的锁序列号，通过对比锁序列号来确定是否会有之前的锁，尝试去覆盖最新的数据。当遇到这种情况的时候，我们姗姗来迟的来自上一个锁的客户端请求，就会被业务服务拒绝掉。



通过一个锁序列器这样的一个乐观锁的解决方案，我们避免了网络延迟带来的数据不一致性

所以你会看到，Chubby 的每个锁，除了文件、目录本身，以及 ACL 权限这样的元数据之外，还有这样四个编号。

实例编号 (instance number) : 当这个“节点”每次被创建的时候自增。

文件内容编号 (content generation number) : 当文件内容被写入的时候会自增。

锁编号 (lock generation number) : 当锁从“释放”(Free)的状态转变为“持有”(Held)的状态的时候自增。

ACL 编号 (ACL generation number) : 当这个“节点”的权限 ACL 信息更新的时候会自增。

这样，通过锁编号，我们就很容易实现前面所说的锁序列器的功能。其他的编号，也都是实现了数据的“版本”功能。这个也使得我们在不确定的网络情况下，能确保写入的数据是按照我们期望的顺序。如果我们尝试拿过时“版本”的锁来更新最新的数据，那么更新就不会成功。

留给你自己读的 Caching 部分

最后，和其他的分布式系统一样，为了提升性能，Chubby 也会在客户端里维护它拿到的数据缓存。Chubby 也有像代理、分区等等其他一系列的机制，来让整个系统更容易扩展，不过这些，就不是 Chubby 在整个大数据领域的核心和重点功能了，我把这部分内容留给你自己去好好研读啦。

小结

我们漫长的旅程终于告一段落了。在过去三讲里，我们从 GFS 的 Master 的“同步复制”这个需求出发，逐步了解了两阶段提交、三阶段提交，以及 Paxos 算法，并且最终在今天一起学习完了 Chubby 的论文。

我们能看到，Google 并没有非常僵化地在所有的分布式系统里面，都简单通过实现一遍 Paxos 算法，来解决单点故障问题，而是选择通过 Chubby 实现了一个粗粒度的锁。这个锁，只是帮助我们解决大型分布式系统的元数据管理的一致性，以及 Master 节点出现故障后的容错恢复问题。

因为大型分布式系统，并不是时时刻刻都会出现数据不一致的风险的。把“哪一个是 Master”这个问题通过共识算法来解决，我们在系统的容错恢复上，就避免了出现两个 Master 的情况。而 Chubby 也是一个非常适合拿来管理非常重要的元数据的地方，这一点我们在 Bigtable 的论文里，其实已经看到了。

Chubby 的系统本身，其实是通过 Paxos 实现了多个 BerkeleyDB 之间日志的同步复制。Chubby 相当于是在 BerkeleyDB 之上，封装好了一个 Unix 文件系统形式的对外访问接口。并且，为了减轻自己的负载，Chubby 还实现了一个**观察者模式**，外部的客户端可以监听 Chubby 里的某一个“目录”或者“文件”，一旦内容变更，Chubby 会通知这些客户端，而不需要客户端反复过来轮询。这个也是为了提升系统的整体性能。

另外，由于网络延时的存在，即使我们的客户端获取到了锁，当写入请求到达锁对应的业务系统的时候，可能这个锁已经过期了。这个会导致我们错误地用旧数据去覆盖新数据，或者说，在没有获取对应资源的锁的情况下，写入了数据。不过，Chubby 通过**锁延迟和锁序列器**这两种方式解决了这个问题。

可以看到，和所有之前 Google 发布的系统一样，Chubby 并没有发明什么新的理论，而是巧妙地通过系统工程，来保障系统的可用性。并且在这个过程中，Google 选择了尽可能使用各种一般工程师都熟悉的编程模型。Google 没有开发一个 Paxos 库，让所有工程师去学习分布式共识算法，而是提供了一个分布式锁服务 Chubby。而在 Chubby 里，提供的也是我们熟悉的类似 Unix 的文件系统、观察者模式这样，普通工程师耳熟能详的编程模型。

这一点，在大型组织中设计基础设施的时候非常关键。**我们设计系统的时候，不能光考虑对应系统的功能，如何让整个系统对于其他团队的开发者和使用者易用，也非常关键。** 我们不仅在 Chubby 这个系统里看到了这一点，从 GFS、MapReduce、Bigtable 这些系统里面也能看到这个设计思路，其实这是一个一以贯之的设计思想。

到这里，我们对于大数据论文的基础知识部分就已经学习完了。接下来，我们就要迈入对于大数据论文里面关于数据库部分的学习啦。

推荐阅读

市面上，对于 Paxos 算法的教程和分析着实不少，但是对于 Chubby 这篇论文的讲解反倒不太容易找到。如果你希望阅读中文资料，我推荐你去读一读 [《从 Paxos 到 Chubby》](#)

Zookeeper- 分布式一致性原理与实践》这本书的第三章，里面有对 Chubby 系统比较详细的讲解。如果你习惯于看视频的话，那么这个 Youtube 上，Data Council 社区放出来的关于 Chubby 的 [讲解视频](#)，值得去好好看一看。

思考题

最后，给你留一道思考题。

我们之前提到过分布式的 CAP 三者难以同时满足的问题，并且我们也看到，两阶段提交是一个 CP 系统，保障一致性但是会牺牲可用性。而三阶段提交则是一个 AP 系统，提升可用性但是会牺牲一致性。

那么，在使用 Chubby 帮助我们解决了 Master 的容错切换问题之后，我们的系统在 CAP 上是否都满足了呢？有人说，CAP 之间难以满足是一个伪问题，在学完了过去这几讲之后，你对这个问题的看法是什么呢？

欢迎在留言区分享下你的观点，和其他同学共同讨论学习。

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

 生成海报并分享

 赞 1  提建议

© 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 [13 | 分布式锁Chubby（二）：众口铄金的真相](#)

下一篇 [加餐1 | 选择和努力同样重要：聊聊如何读论文和选论文](#)

1024 活动特惠

VIP 年卡直降 ¥2000

新课上线即解锁，享 365 天畅看全场

超值拿下 ¥999 ↗



精选留言 (1)



Cc

2021-10-22

选主的时候没有办法提供服务 应该是牺牲了可用性吧

写留言



1