32 | Raft（一）：不会背叛的信使

2021-12-24 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 16:56 大小 15.52M

你好，我是徐文浩。

在前面课程中，我们了解过的这些大数据处理系统，其实都属于分布式系统。所以，它们也都需要解决分布式一致性，或者说分布式共识的问题。

我们之前已经介绍过 Chubby，这个 Google 开发的分布式锁。正是通过 Chubby 这个系统，使得我们可以确保系统里始终只有一个 Master，以及所有的数据分区在一个时间点只有一个所有者。而 Chubby 的底层，就是 **Paxos** 这样的分布式共识（Distributed Consensus）算法。另外在后面的 Spanner 这样的分布式数据库里，我们也看到了 Raft，也可以直接拿来作为分布式数据库的解决方案。

在之前的这些论文里，我们对 Paxos 的算法做了一些简单的介绍。不过，我们并没有对 Paxos 的算法做非常深入地剖析。一方面，Paxos 算法其实并不容易理解，这也是为什么 Paxos 的论文第一次发表的时候，并没有得到足够的关注。另一方面，目前市场上实际的开源项目，大部分也并不是采用了 Paxos 或者 Multi-Paxos 算法，而是往往采取了简化和变形。Google 的 Chubby 并没有开源，而开源的 ZooKeeper 实现的是自家的 ZAB 算法，对 Paxos 做了改造。

事实上，在 2021 年的今天，最常被使用的分布式共识算法，已经从 Paxos 变成了 **Raft**。这要归功于来自斯坦福大学，在 2013 年发表的一篇论文《In Search of an Understandable Consensus Algorithm》。

这一篇论文，也是我们接下来两讲的主题。这两讲的目标，是帮助你掌握这样两点：

理解 Raft 算法的实现，以及它是如何能在不稳定的分布式环境下达成一致的。

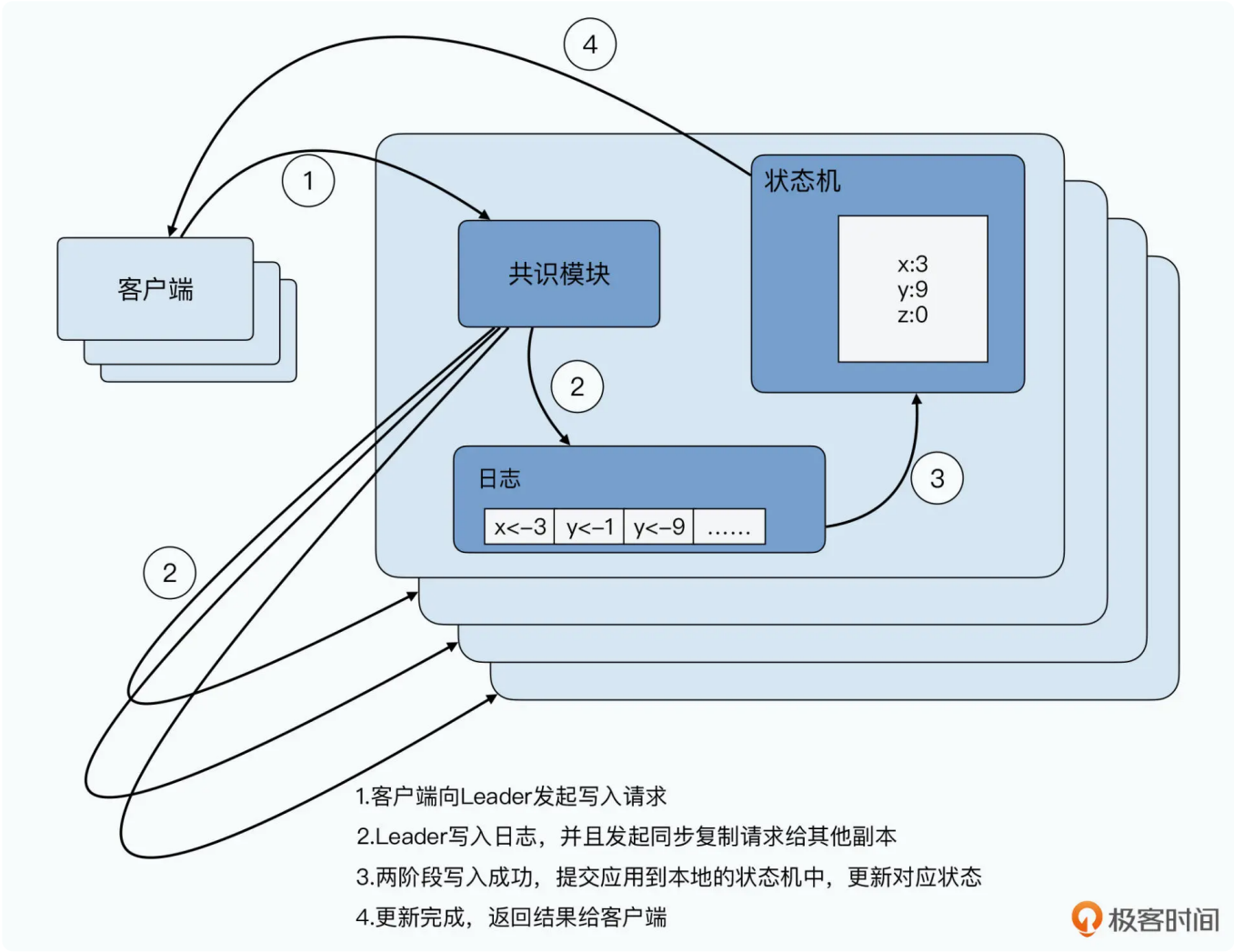
理解 Raft 算法的性能指标，明白它适用的场景，以及它的局限性。

分布式共识问题和复制状态机

在具体讲解 Raft 算法之前，我们先做一个小小的复习，那就是我们之前讲到过的 Paxos 算法，到底是在帮助我们解决一个什么样的问题。

无论是在 Chubby 还是 Spanner 里，我们通过 Paxos 算法想要解决的问题，其实都是一个“**状态机复制**（State Machine Replication）”问题，简称 SMR 问题。

我们可以把一个数据库的写入操作，看成是一系列的顺序操作日志，也就是把对于数据库的写入，变成了一个状态机。而对于这个状态机的更新，我们需要保障高可用性和容错能力。这个需要我们对于某一个服务器上的日志追加写入，能够做到同步复制到多个服务器上。



论文中的图1，理解基于分布式共识算法的“状态机复制”过程是怎么样的

这样，在当前我们写入的节点挂掉的时候，我们的数据并不会丢失。为了避免单点故障，我们希望可以从多台服务器都能写入数据，但是又不能要求所有服务器都同步复制成功。前者，是为了确保某个服务器挂掉的时候，我们可以快速切换到另一台备用服务器；后者，是避免我们写入的时候，因为某一台服务器挂掉了就无法写入成功。

所以，Paxos 这样的分布式共识算法的基本思路，就是这样几点：

> 日志追加写入的时候，只要能够半数的服务器完成同步复制就算写入成功了。
>
> 在有节点挂掉的时候，因为我们有多个服务器，所以系统可以自动切换恢复，而不是必须修复某一台服务器才能继续运行。
>
> 系统的数据需要有“一致性”，也就是不能因为网络延时、硬件故障，导致数据错乱，写入的数据读出来变了或者读不出来。

对于 Paxos 算法的具体实现，我就不在这里重复了。你可以回头看课程里关于 Chubby 论文的部分，或者去对应看 Paxos 算法的原文。

Raft 算法拆解

Paxos 算法本身在理论上并没有什么问题，它的主要问题是，不太容易理解。不知道你在课程之外，有没有去读一下 Paxos 的原始论文，或者其他能够找到的深入解析这个算法的资料。至少对我来说，我做不到能把算法牢牢记在脑子里，而是每次需要的时候，都要再花上几个小时重新看一遍算法。

而 Raft 算法，其实就是针对 Paxos 算法的这个缺点而来的，相信你从论文的名字《In Search of an Understandable Consensus Algorithm》也能看到这一点了。

那么，Raft 算法是为什么可以让我们容易理解的呢？我觉得有三点原因：

Paxos 算法的出发点，是为了达成分布式共识。状态机复制，只是分布式共识的一个特例，所以如果我们看原始的 Paxos 算法，去理解 Chubby 乃至 ZooKeeper 的实现，会有一个转化过程，不容易理解。**而 Raft 算法，则是直接就从状态机复制的问题出发，算法和需要解决的问题直接对应。**

因为是分布式共识问题，所以 Paxos 算法可以从任何一个节点发起数据写入，我们就需要考虑各种数据写入时序情况下的共识问题。**而 Raft 算法则把问题拆分成了一个个独立的子问题**，比如 Leader 选举（Leader Election）、日志复制（Log Replication）、安全（Safety）和成员变化（Membership Changes）等等。这些子问题都解决了，状态机复制的问题自然就解决了。我们只要能够理解和证明每一个子问题就够了，**这个对于问题分而治之的办法，降低了理解问题的复杂度**。

最后，**Raft 算法加强了系统里的限制，使得问题在很多状态下被简化了**。比如，Raft 算法是有一个强 Leader 的，而 Paxos 算法是无 Leader 或者弱 Leader 的。而因为有了强 Leader，Raft 算法就可以进一步地要求日志里面不能有空洞。这使得我们不需要像 Megastore 的论文里那样，面临某一个数据副本有“空洞”的情况，再通过 no-op 这样的操作，再去同步这些空洞的数据。

那么接下来，我们就来深入看看 Raft 算法具体是怎么样实现的。Raft 算法的思路非常简单明确，它做出了这样几个选择：

首先是让系统里始终有一个 **Leader**，所有的数据写入，都是发送到 Leader。一方面，Leader 会在本地写入，另一方面，Leader 需要把对应的数据写入复制到其他的服务器

上。这样，问题就变简单了，我们只需要确保两点，第一个是系统里始终有 Leader 可用；第二个，是基于 Leader 向其他节点复制数据，始终能确保一致性就好了。

因为 Leader 所在的服务器可能会挂掉，那么挂掉之后，我们需要尽快确认一个新 Leader，所以我们就需要解决第一个子问题，就是 **Leader 选举问题**。

我们需要保障分布式共识，所以 Leader 需要把日志复制到其他节点上，并且确保所有节点的日志始终是一致的。这个，就带来了第二个问题，也就是**日志复制问题**。

同时，在 Leader 挂掉，切换新 Leader 之后，我们会遇到一个挑战，新的 Leader 可能没有同步到最新的日志写入。而这可能会导致，新的 Leader 会尝试覆盖之前 Leader 已经写入的数据。这个问题就是我们需要解决的第三个问题，也就是**“安全性”问题**。

基本概念和算法框架

这里，我们先来看看，Raft 系统的基本数据写入流程是怎么样的。首先，Raft 的系统里也会和 Chubby/ZooKeeper 一样，有多台服务器。这些服务器，会分成这样三个角色：

Leader，它会接收外部客户端数据写入，并且向 Follower 请求同步日志的服务器。同一时间，在整个 Raft 集群里，只会有一个 Leader。

Follower，它会接收 Leader 同步的日志，然后把日志在本地复制一份。

Candidate，这是一个临时角色，在 Leader 选举过程中才会出现。

外部的客户端写入数据的时候，都是发送给 Leader。

Leader，本质上是通过一个两阶段提交，来做到同步复制的。一方面，它会先把日志写在本地，同时也发送给 Follower，这个时候，日志还没有提交，也就没有实际生效。Follower 会返回 Leader，是否可以提交日志。当 Leader 接收到超过一半的 Follower 可以提交日志的响应之后，它会再次发送一个提交的请求给到 Follower，完成实际的日志提交，并把写入结果返回给客户端。

Leader 选举

那么，既然是一个两阶段提交，我们就会遇到 Leader 节点挂掉的时候。Raft 不能让整个系统有单点故障，所以节点挂掉的时候，它需要能够协商出来一个新的 Leader，这个协商机制就是 Leader 选举。

Raft 的 Leader 选举是这样的：

在 Raft 里，Leader 会定期向 Follower 发送心跳。Follower 也会设置一个超时时间，如果超过超时时间没有接收到心跳，那么它就会认为 Leader 可能挂掉了，就会发起一轮新的 Leader 选举。

Follower 发起选举，会做两个动作。第一个，是先给自己投一票；第二个，是向所有其他的 Follower 发起一个 **RequestVote 请求**，也就是要求那些 Follower 为自己投票。这个时候，Follower 的角色，就转变成了 Candidate。

在每一个 RequestVote 的请求里，除了有发起投票的服务器信息之外，还有一个**任期**（Term）字段。这个字段，本质上是一个 Leader 的“版本信息”或者说是“逻辑时钟”。

每个 Follower，在本地都会保留当前 Leader 是哪一个任期的。当它要发起投票的时候，会把任期自增 1，和请求一起发出去。

其他 Follower 在接收到 RequestVote 请求的时候，会去对比请求里的任期和本地的任期。如果请求的任期更大，那么它会投票给这个 Candidate。不然，这个请求会被拒绝掉。

除了对比任期之外，**Candidate 还需要满足后面我们讨论的一些“安全性”要求，那就是选举出来的 Leader 上，一定要有最新的已经提交的日志**，这个我们在这里先不聊，后面在讲解安全性的时候会深入讲解。

在一个任期里，一台服务器最多给一个 Candidate 投票，所以投票过程是先到先得，两个服务器都发起了 RequestVote，我们的 Follower 也只能投给一台。

而我们的 Candidate，会遇到三种情况。

第一种，自然是超过半数服务器给它投票。那么，它就会赢得选举，变成 Leader，并且进入一个新的任期。

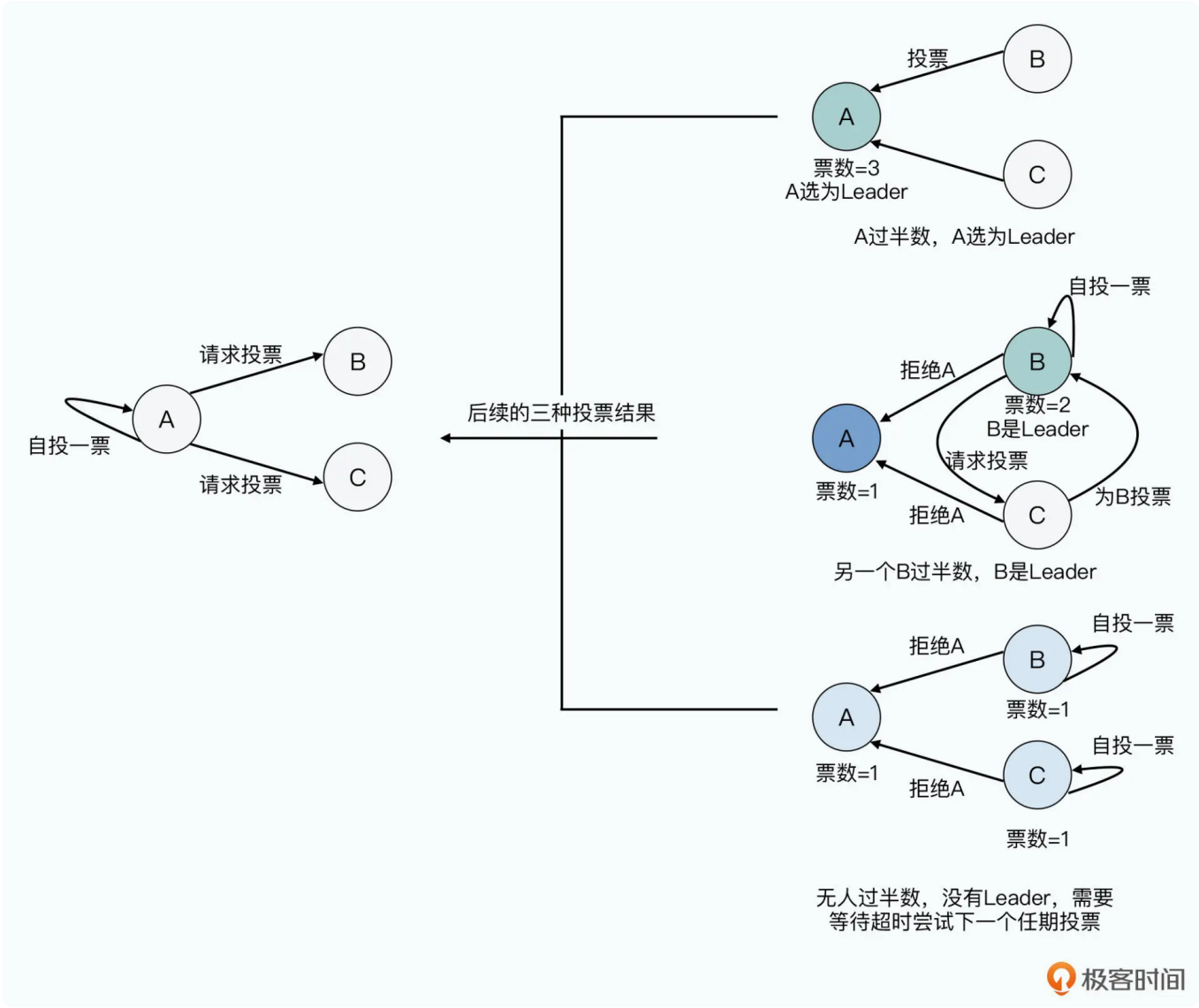
第二种，是有另外一个 Candidate 赢得了选举，变成了下一任的 Leader。这个时候，我们还不知道自己已经输了。过了一会儿，新的 Leader 接收到了外部客户端的写入请求，就会向我们这个 Candidate 发起日志同步的请求。或者，新的 Leader 也会向我们发送一个心跳请求。无论是哪种请求，里面都会包含一个任期的信息，这个任期的信息比我们当前

知道的最新的 Leader 大，那么我们就知道自己在投票里面输了，会把自己变成一个 Follower，并更新最新的任期和 Leader 信息。

第三种，是过了一段时间，无人获胜。我们会为这种情况设置一个超时时间，如果到了超时时间，我们既没有赢也没有输，那么我们会对任期自增 1，再发起一轮新的投票。

你会发现这个选举过程，会遇到一种尴尬的情况。那就是当 Leader 挂掉的时候，很多个 Follower 都会成为 Candidate。他们都会先给自己投票，然后向其他人发起 RequestVote。这样，就会导致**票被很多人“瓜分”**，没有人能拿到超过半数的票。然后我们就会陷入前面的第三种情况，无限循环出不来了。

Raft 对于这个问题的解决方案，是让选举的超时时间在一个区间之内**随机化**。这样，不同的服务器会在不同的时间点超时，最先超时的那个服务器，大概率会在其他服务器发现超时之前，就赢得投票。



投票选举Leader的三种情况

日志复制

Leader 选出来了，那我们自然就可以向 Leader 发送写入数据的请求。既然是一个“状态机复制”的方案，写入请求其实就是**一条操作日志的追加写**。在 Raft 里，就是通过一个 **AppendEntries 的 RPC 调用实现的**。

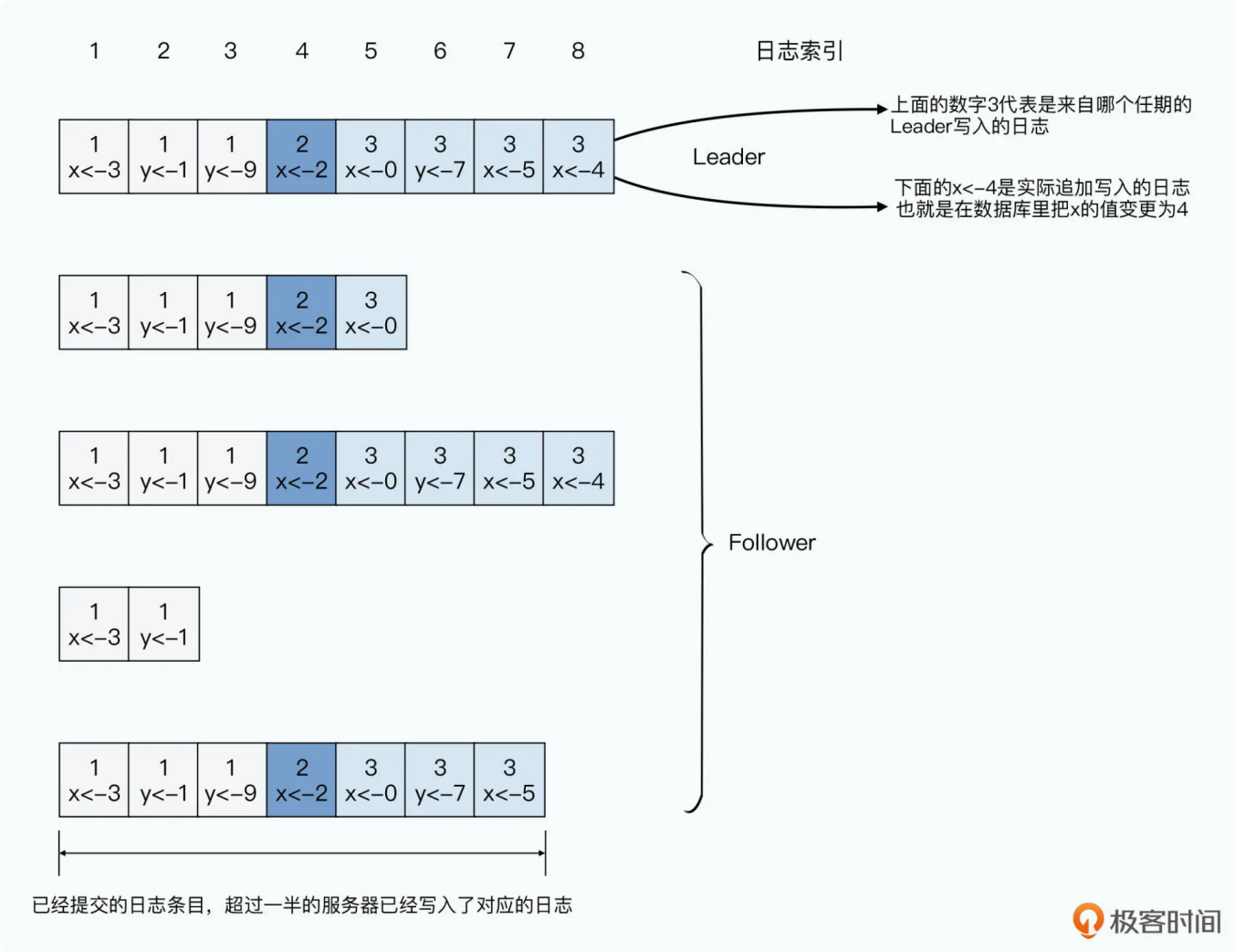
整个数据写入，就是一个前面我们说的两阶段提交的过程，只不过这个两阶段提交只需要“半数”通过，就可以发起第二阶段的提交操作，而不需要等待所有服务器都确认可以提交。当然，我们可能会遇到某些节点挂掉了，和两阶段提交里一样，我们要做的就是无限重试。

我们追加写入的每一条日志，都包含三部分信息：

第一个，是日志的索引，这是一个随着数据写入不断自增的字段。

第二个，是日志的具体内容，也就是具体的状态机更新的操作是什么。

第三个，就是这一次数据写入，来自 Leader 的哪一个任期。



论文中的图6，可以看到Raft里面，追加写入的数据结构

我们在追加写入日志的时候，不只是单单追加最新的一条日志，还需要做一个**校验**，确保对应的 Follower 的数据和 Leader 是一致的：

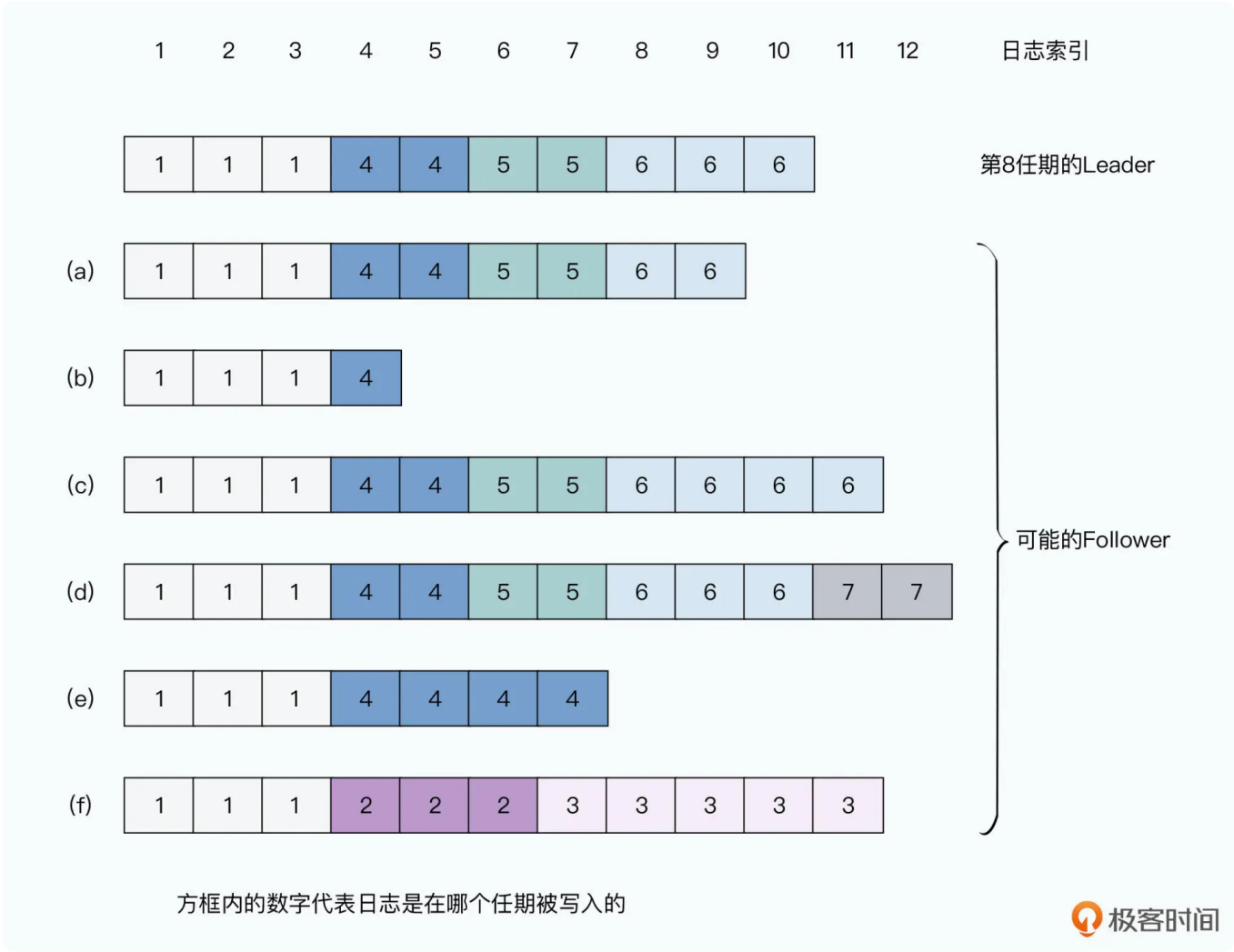
首先，在发起追加写入日志的复制请求的时候，Leader 的 AppendEntries 的 RPC 里，不仅会有最新的一条日志，还会有上一条日志里的日志索引和任期信息。

Follower 会先对比日志索引和任期信息，在自己的日志里寻找相同索引和任期的日志所在的位置。

- 如果找到了，Follower 会把这个位置之后的日志都删除掉。然后把新日志追加上去。
- 如果找不到，Follower 会拒绝新追加的日志。然后，Leader 就知道，这个 Follower 没有同步到最新的日志，那么 Leader 会在自己的日志里，找到前一条日志，再重新发送给 Follower。

Leader 会不断重复这个过程，确保找到和 Follower 的同步点，找到了之后，就会把这个位置之后的日志都删除掉，然后把同步点之后的日志一条条复制过去。

本质上，Raft 的复制操作，是让 Leader 为每一个 Follower 都从 Leader 的尾部往头部循环，找到 Follower 最新同步到哪里的日志。然后从这个位置开始，往后复制 Leader 的日志，直到最新一条的日志。通过这个过程，我们把每一次 Leader 的日志复制，都变成了一次**强制所有 Follower 和当前 Leader 日志同步**的过程。



论文中的图7，可以看到，其他服务器中可能有未提交的脏日志，也会因为没有来得及同步数据而相比于Leader落后较多

之所以需要这个过程，是因为我们不同服务器的日志，可能因为网络延时还没有同步，也可能因为硬件故障还包含未提交的日志。我在这里放上了论文里的图 7，你可以看到：

我们的 Leader，此时此刻已经到了第 10 条日志；

服务器 a 和 b 里，日志还没有及时复制到最新的日志；

服务器 c 和 d 里，包含了没有提交的“脏日志”；

服务器 f，不仅没有复制到最新的日志，也包含很多旧的脏日志。

这里的脏日志，则是来自服务器在担任 Leader 的时候，可能已经在本地写入了日志，然后挂掉了，所以日志没有能够完成提交。

而我们让 Leader 在追加写入的时候，顺便进行强行同步的过程，给我们**带来了一个好处**，那就是我们不需要有什么特殊机制，在硬件故障或者两阶段提交失败的时候去做回滚，而是只需要不断强制 Follower 和 Leader 同步，就能保障数据的一致性。

不过，这个也给我们**带来了一个挑战**，那就是我们需要确保，我们的 Leader 始终是包含了最新提交的日志。也就是无论我们因为故障也好，其他原因也好，随便怎么切换 Leader，Leader 的日志都是最新并准确的。

而这个，则需要我们下面所说的安全性机制来保障。

安全性

首先要注意，Raft 里，每一个服务器写入的日志，会有两种状态，一种是未提交的，一种是已提交的。我们这里所说的**最新，指的是已提交的日志**。我们想要确保 Leader 的日志是最新的，只需要在 Leader 选举的时候，让只有最新日志的 Leader 才能被选上就好了。

而要做到这一点，也并不困难，Raft 的做法是，**直接在选举的 RPC 里，顺便完成 Leader 是否包含所有最新的日志就好了**。Raft 是这么做的：

> 在 RequestVote 的请求里，除了预期的下一个任期之外，还要带上 Candidate 已提交的日志的最新的索引和任期信息。
>
> 每一个 Follower，也会比较本地已提交的日志的最新的索引和任期信息。
>
> 如果 Follower 本地有更新的数据，那么它会拒绝投票。

在这种情况下，一旦投票通过，就意味着 Candidate 的已提交的日志，至少和一半的 Follower 一样新或者更新。而 Raft 本身写入数据，就需要至少一半成功，才会提交成功。所以，在前面愿意给 Candidate 投票的里面，至少有一个服务器，包含了最新一次的数据提交，而 Candidate 至少和它一样新，自然也就包含了最新一次的数据提交。

就这样，我们简单地通过共识算法需要半数以上通过的原理，很容易就可以只让包含最新提交数据的服务器，才能选上 Leader。

小结

好了，到这里，我们对于 Raft 的基本算法原理已经讲完了。Raft 的主要目标是找到一个容易理解的分布式共识算法。所以，它通过**直接从“状态机复制”这个角度，来直接设计算法**，而不是通过共识问题绕一圈，映射成一个状态机复制问题。

在算法设计层面，Raft 采取了**分而治之**的办法。通过把问题拆分成 Leader 选举、日志复制、安全性等一系列子问题，来使得算法更容易理解。

在 Leader 选举上，Raft 采用的是典型的**心跳检测 Leader 存活**，以及**随机超时时间投票，确保不会死循环**，来确保我们可以快速选出 Leader。

在日志复制层面，Raft 采用的是一个典型的**两阶段提交**。为了让算法简单，它直接在日志复制过程中，强制 Follower 和 Leader 同步，来解决删除未提交的脏日志的办法。而为了做到这一点，Raft 又需要确保无论 Leader 如何通过选举切换，都需要包含最新已提交的日志的办法。而要确保这一点，它也只是在 Leader 选举的时候，带上了日志索引和任期信息就简单地解决了。

那么，在下一讲里，我们会深入 Raft 的安全性，看看能不能通过理论证明，进一步地，我们还会一起来看看 Raft 的成员变化问题，以及 Raft 的最终性能是怎么样的。

推荐阅读

对于 Raft 算法，你不仅可以去读论文原文，也能直接在网上找到论文两位作者对应的讲座。我把链接放在这里，你可以自己去看一下。

思考题

在 Raft 集群刚启动的时候，我们是如何来确定哪一台服务器是 Leader 的呢？欢迎在留言区分享你的答案和思考，也欢迎你把今天的内容分享给更多的朋友。

上一篇 复习课（九）| Megastore

精选留言 (2)

写留言

在路上

2021-12-24

徐老师好，我认为启动时的选举涉及两个问题，第一、什么时候发起投票，第二、如何才能被选中。对于第一点，集群启动是需要一段时间的，每台机器启动完成的时间不同，本身符合随机性，所以可以在启动后立即发起投票，对于第二点，还是需要本地已提交的日志是最新的，并获得大多数的Follower同意。如果选举失败，则随机等待一段时间，进入下一个选举循环。

展开

曾轼麟

2021-12-24

我理解刚启动的时候有点类似于抢占模式，每个刚启动的服务都会发起一次选举请求，最先获得半数的实例会宣布为master

我这边有一个疑问，redis的raft实现好像又是另一种变种，通过一群哨兵去基于raft选举redis master，而哨兵本身是不能成为master，不知道这种方式下能否满足raft的状态机复...

展开