33 | Raft（二）：服务器增减的“自举”实现

2021-12-27 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 13:21 大小 12.23M

你好，我是徐文浩。

在上节课里，我们了解了 Raft 算法，知道了它是怎么把“状态机复制”这样一个问题，拆解成了 Leader 选举、日志同步以及安全性三个子问题。那么，今天这节课，我们会进一步深入来了解 Raft 算法的另外几个问题。

这些问题，虽然在实践中我们必然会遇到，但是之前讲解各类分布式系统的时候，我们很少提到。正好，趁此机会，我们可以对在 Raft 集群中如何动态增减服务器，以及如何对 Raft 的状态机创建快照加以学习，并予以掌握。

成员变更（Membership Change）

无论是上节课我们讲解 Raft 的算法，还是之前我们在 Chubby 的论文里介绍的 Paxos 算法，我们都没有讲解往集群里增减服务器会发生什么情况。但是，这种情况其实是运维这些系统时，必然会遇到的情况。

比如，我们遇到某台服务器硬件故障了，需要往集群里增加一台服务器，并把坏了的服务器去掉。或者，我们为了系统的可用性考虑，往原先 3 台服务器组成的 Raft 集群里，加两台变成 5 台服务器，这样我们可以容忍集群里挂掉两台服务器，而不是原先的一台。

最简单的增减服务器的方式，当然是把整个集群停机，修改配置然后重启。但是，我们使用 Raft 这样的算法就是为了高可用性，所以显然我们不能这么做。

那么，**我们能不能直接增加两台服务器，然后修改各个服务器的配置呢？**答案依然是不行的，因为我们没法做到多台服务器的配置同时生效，我们会遇到这样的情况：

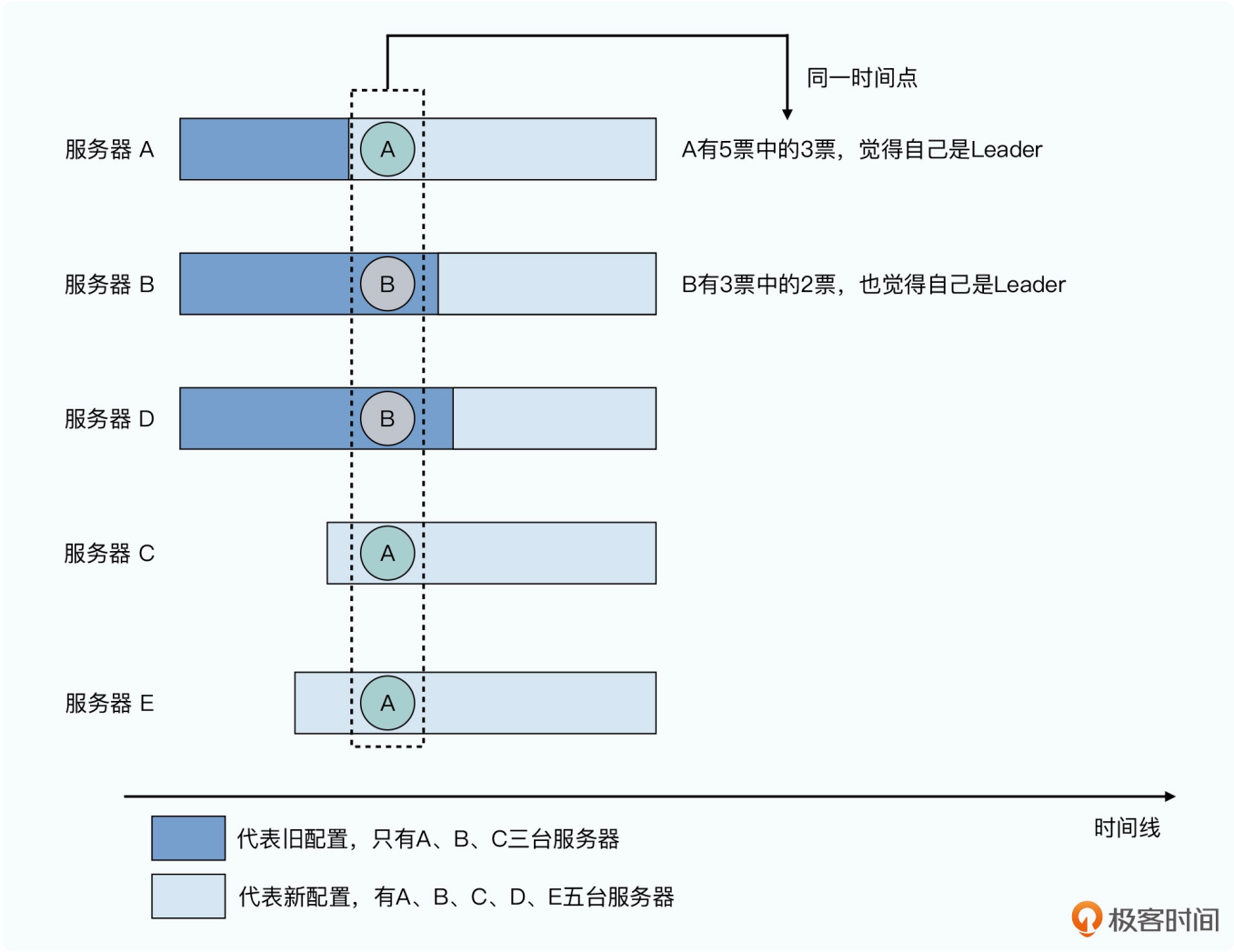
原先有服务器 A，B，C 三台；

新增了服务器 D，E 两台；

服务器 A 的配置先更新，这个时候，它觉得需要 3 台服务器的投票就是多数通过了；

服务器 B 的配置后更新，在没有更新的时候，它觉得有 2 服务器的投票就是多数通过了；

这样，我们就会遇到 A 和 B 都各自认为投票获得通过的情况，这个时候，我们在一个集群里就会有两个 Leader，系统的共识就被破坏了。



要解决这个问题，Raft 采用了一个加入“**过渡共识**（Joint Consensus）”的办法。这个办法是这样的，无论是服务器增加还是减少，我们的 Raft 其实有两组配置。第一组配置是变更之前的服务器配置，我们称之为 C_{old}（也就是 Old Configuration 的意思），第二组配置是我们变更之后的服务器配置，我们称之为 C_{new}（也就是 New Configuration 的意思）。

要变更整个 Raft 集群的配置，我们会进行两阶段的变更，第一阶段是把我们的服务器配置，从 C_{old} 变更成 C_{old}, C_{new}，然后在第二阶段，再从 C_{old}, C_{new} 变更成 C_{new}。在整个集群处于 C_{old}, C_{new} 这个过渡共识的过程中，整个 Raft 集群会做到以下几点：

所有的日志追加写入，都会复制到新老配置里所有的服务器上。

新老配置里的任何一个服务器，都有可能被选举成 Leader 节点。

无论是选举，还是达成共识后提交日志，投票需要同时满足旧配置里半数以上服务器的通过，而且也需要新配置里半数以上服务器的通过。

而两次的配置变更，也是通过 Raft 算法，通过一个日志追加写入更新到整个 Raft 集群里的。那么，整个配置变更的过程就会变成这样：

外部的客户端，先向我们的 Raft 集群，写入一个操作，就是把我们的集群配置，从 C_{old} 变更成 C_{old}, C_{new}。这个写入操作的提交，需要获得半数的 C_{old} 里的服务器通过。

这个写入操作成功之后，集群就进入了“过渡共识”阶段。此时此刻，所有的数据写入，都需要至少获得半数的 C_{old} 里的服务器和半数的 C_{new} 里的服务器通过。

即使这个时候 Leader 挂掉了，我们去选举一个新的 Leader，一样也需要获得 C_{old} 里和 C_{new} 里半数服务器的通过。

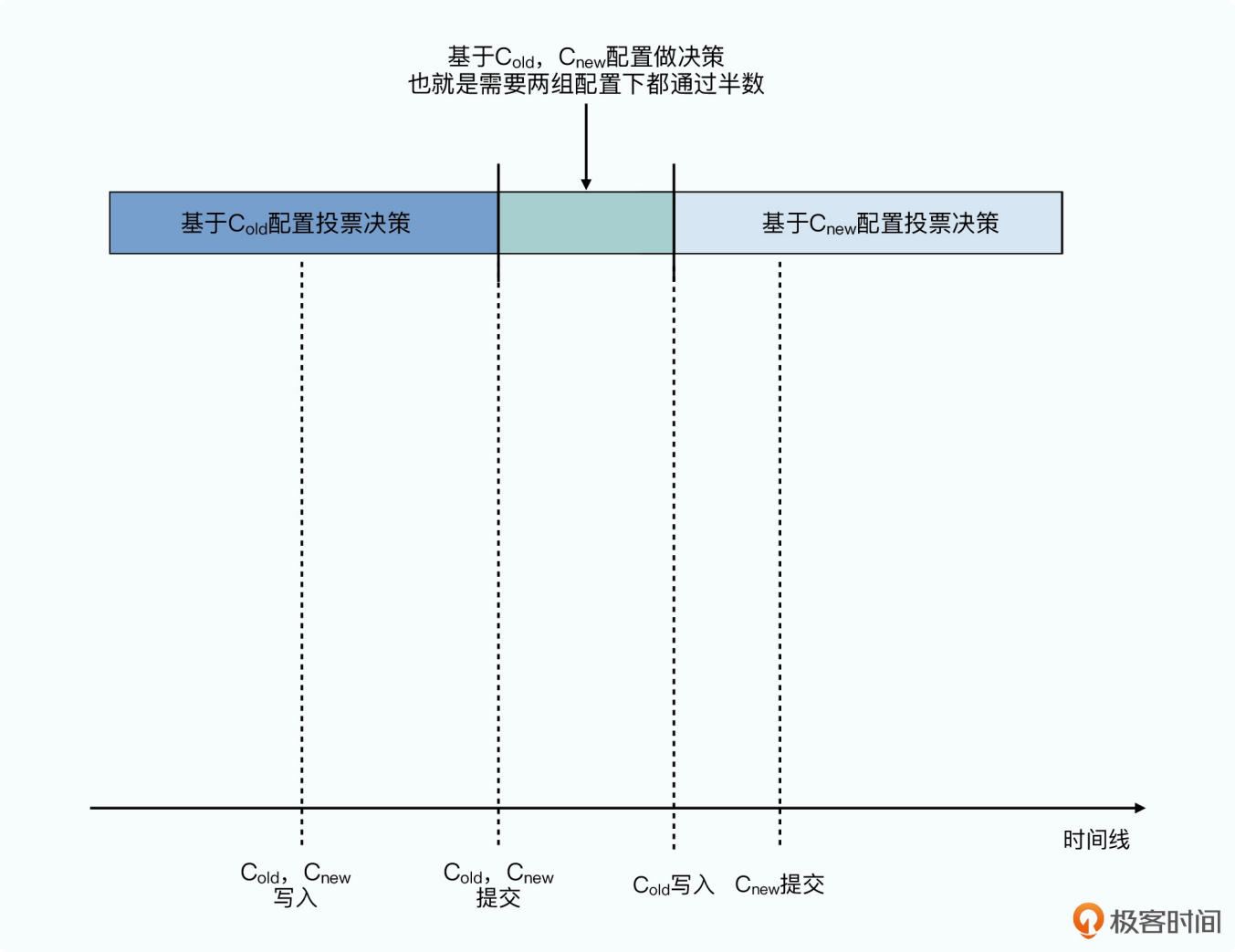
而且，根据我们上节课里要求的安全性，这个过程里新选举出来的 Leader，也一定有最新提交的日志。也就是新的 Leader 的配置，也一定是基于 C_{old}, C_{new} 这个过渡共识的配置的。

所以，在整个过渡阶段，我们可以确保所有的日志写入，无论是在老的配置下，还是新的配置下都是可以达成共识的。那么，接着我们就可以再从这个过渡阶段的配置，切换到新配置下。

这个切换，同样是通过一个日志追加写入来完成的。我们外部的客户端，会向当前的 Leader，写入一个把配置从 C_{old}, C_{new} 变成 C_{new} 的操作。

这个操作，只需要 C_{new} 里的服务器的半数通过，而不再需要 C_{old} 里的半数通过了。因为此时，我们已经不再需要 C_{old} 服务器里的日志信息了。

于是，我们的集群就整个地切换到了 C_{new} 的配置之下了。



通过过渡阶段切换到底由哪些服务器投票的示意图

不过，在这里，我们可能会遇到一种情况，那就是在进行从 C_{old}, C_{new} 变成 C_{new} 的数据写入的时候，当时的 Leader 是 C_{old} 老配置里的服务器，而不是 C_{new} 里的服务器。**一旦提交成功，我们的集群里就已经没有这个服务器的位置了。那该怎么办呢？**

Raft 的解决方案，是 Leader 在完成 C_{new} 的提交后，退回到 Follower 状态。这也意味着，从这个 Leader 写入数据到提交阶段，我们的 Raft 集群，被一个不在集群内的服务器管理。这个时候，我们在进行过半数的投票的时候，只要不要包括这个服务器就好了。这个额外的服务器并不会影响本身 Raft 算法的正确性。

此外，在我们往集群里添加 C_{new} 里新增的服务器的时候，这些服务器里没有任何日志条目。那么，这些服务器一方面需要很长的时间来复制 leader 的日志，而且在这段时间里，它是不能提交新的日志的。这就意味着，如果我们加入多个节点，有可能会影响 Raft 的写入性能。可能有很多写入就是等待某一台新加入的服务器参与投票才能获得半数通过。所以 Raft 的做法，是会先把服务器加入集群参与复制，但是不给投票权，直到复制已经追上 Leader 的进度再把投票权给它。也就是这台服务器本身也有一个“过渡阶段”。

其实，这里的这个“过渡共识”的策略，和我们平时进行应用开发，进行大的数据库表的重构是类似的，那就是采用一个**“双写”的迁移策略**。

在传统的应用开发当中，我们可能要重构数据库，比如，整个电商业务里的订单表的字段结构，都要发生变化。那么，我们通常并不是新建一组数据库表，然后停机进行数据表迁移和版本升级，因为那样停机时间会很长，整个系统的可用时间会大大受到影响。

合理的做法，是我们虽然会新建数据表，但是所有的业务操作，既会在老的数据表上仍然正常写入，同时也会再写入新的数据表。这样，我们前端的业务服务器的代码一样可以一台台滚动更新，从读写老表，变成同时读写新老表，最后迁移到只读写新表。这样，一台服务器上代码的更新，不会影响到另一台还没有更新的老服务器。

在 Raft 里的服务器配置变更也是一样的，我们可以设定一个过渡阶段，在这个阶段里，我们既要保障原有逻辑的正确性，也就是原来老服务器集群的“共识”要求可以满足。这样，我们不用中断对于数据的读写需求。同时，我们也会保障新服务器集群的“共识”要求可以满足。这样，我们在后面切换到新集群的时候也不会缺少数据。

日志压实（Log Compaction）

无论是 Raft 还是 Paxos，分布式共识算法都是基于不断追加写入的日志的。不过，我们之前都回避了这样一个问题，**随着时间的推移，写入的日志越来越多，我们的硬盘空间可能会不够，那我们该怎么办呢？**

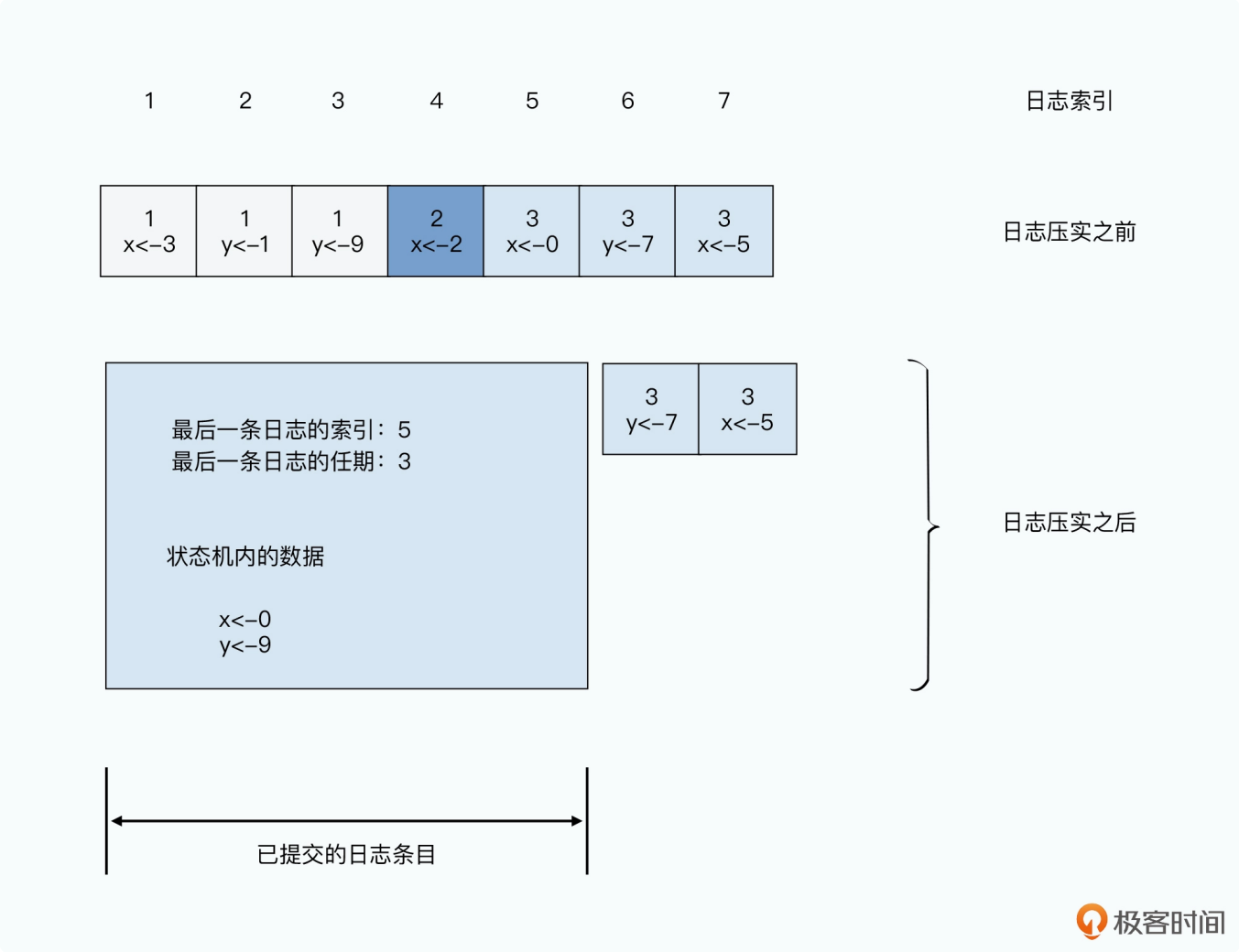
其实，回到我们问题的本源，我们并不需要保留这些写入的日志。我们需要保留的，只是日志应用之后的“状态机”就好了，因为 Raft 要解决的不是日志同步问题，而是状态机复制问题。同步并应用日志，只是这个问题的解决方案。

所以，和一般的数据库系统一样。我们可以定期给状态机**创建快照**（Snapshot），保留下它当前的最新状态，然后把之前的日志都丢弃掉就好了。这个快照里面需要包含两部分信息：

首先自然是状态机本身的最新状态，也就是数据库里面的具体数据；

其次，是对应的日志相关的元数据，也就是当前快照里面，对应最后一条日志的索引和任期信息。

这样，通过这个快照，以及后续我们追加写入的日志，我们同样可以组合获得最新的状态机状态。这里我放了一张图，你可以对照着理解一下这个快照是怎么样的。



Log Compaction的逻辑

这个创建快照、清理日志的方式，一般被称之为**日志压实**（Log Compaction）。有些地方，会把这个 Compaction 翻译成压缩，我觉得是不合理的。因为，这个动作实际上并不是对数据进行压缩（Compression），而是把对于数据库里同一个 Key 的多次更新的日志，压实之后只保留最后一条。

在 Raft 里，日志压实这个动作，是在每个服务器上各自进行，而不是通过 Leader 进行然后复制到各个 Follower。因为后者会占用太多的网络带宽，完全没有必要。

不过，我们还是会遇到需要通过 Leader 进行整个快照同步的情况，那就是 Follower 的日志同步已经落后得太多的情况。比如，我们的某个 Follower 挂掉了比较长的一段时间，而 Leader 在这段时间内已经创建了快照，所以很多没有同步到这个 Follower 的日志，在

Leader 上已经删除了。或者，我们往集群里新加了一个新节点，这个节点没有任何日志，自然也是要去同步 Leader 上的快照，然后再去复制快照之后写入的日志。

客户端交互和可线性化

最后，我们来看看 Raft 的客户端是怎么和整个集群交互的。Raft 的客户端和 Raft 集群的交互，和我们之前在 Chubby 里介绍的 Paxos 是类似的。如果客户端知道哪一个服务器是 Leader，它就会把对应的请求发给 Leader。如果不知道，它就会随机发送给一个服务器节点，如果那个服务器节点并不是 Leader，它就会拒绝请求同时把它所知道的最新的 Leader 信息返回给到客户端。

此外，如果客户端发给 Leader 的请求超时了，客户端会换一个服务器进行重试。因为这种情况下，Leader 很有可能崩溃了。但是为了避免同一个客户端请求，在 Raft 里被执行两次，每一个客户端请求都需要带上一个类似于 UUID 这样的唯一标识。Raft 自己需要在服务端进行去重，如果对应操作已经执行过了，那只需要返回执行结果，而不需要再次执行。

和 Paxos 一样，我们的 Raft 算法也需要满足“**可线性化**”。也就是，已经写入的数据，必须下次读取的时候一定能被读取到。一般来说，像 Raft 这样的读写都是通过 Leader 进行的算法不会遇到这种问题。但是，在遇到 Leader 切换的时候，就不一定了。可能，我们的 Leader 已经因为网络分区，其他人已经选出了新的 Leader，但是此时我们的旧 Leader 还不知道。

所以，在 Raft 里，Leader 一样需要一个**租约**的机制。我们让 Leader 拥有一个固定时长的租约，并且通过和 Follower 的心跳来不断续期。但是，只要在租约时间范围之内，即使 Leader 服务器挂掉了，其他人仍然不能担任 Leader，要到租约过期了，我们新选出的 Leader 才能生效。

小结

好了，到这里，我们对于整个 Raft 算法的学习也就告一段落了。在上节课讲解的 Leader 选举、日志复制以及安全性三个子问题之外，这节课，我们主要学习和理解了 Raft 是如何进行成员变更、日志压实，以及客户端是如何和整个集群交互的。

Raft 的成员变更，本质上是一个“双写”的策略。通过先将配置修改为在新、老两个版本的配置上都能达成共识，Raft 使得服务器的集群变更可以平滑过渡，不需要中断服务。而整个配置策略的变更本身，也是通过一条 Raft 日志，来实现异步在多个服务器上更新并最终达成共识。

Raft 的日志压实，本质上和数据库的快照 + 增量日志的备份策略没有区别。而让各个服务器各自在本地进行对应的日志压实策略，Raft 最小化了需要占用的网络带宽。

在客户端和整个 Raft 集群的交互上，为了确保整个系统的可线性化，最佳的实现策略还是为 Leader 建立一个固定时长的“租约”。

可以看到，无论是成员变更、日志压实还是和客户端的交互，Raft 的这些设计思路，我们都可以在日常使用的其他系统中找到对应的思路和方法，这也是计算机工程的一大特点，那就是具体实现可能各不相同，但是对应的思路和方法往往是相通的。

推荐阅读

《In Search of an Understandable Consensus Algorithm》这篇论文，和我们看过的其他论文一样，非常简短。所以对于共识问题的很多基础知识都是通过提及的 Paxos、Chubby 和 ZooKeeper 这些论文和系统覆盖到的。其中，对于具体算法也只是做了最精简的介绍。

不过，作者 Diego Ongaro 在 2014 年同一年发表的博士论文《Consensus：Bridging Theory And Practice》里面，对于从算法理论到具体实现有了更详细的剖析，对于 Raft 之外的其他分布式共识算法也有更深入的介绍，你可以去好好读一读。

思考题

我们看到，Raft 算法和我们之前学习过的 Paxos 算法，在实现上有一个重大的区别。那就是，它对于日志的写入不是“Append Only”的，也就是不只有追加写操作。在 Leader 切换之后，对于 Follower 我们会遇到寻找共识点，然后删除旧的日志，重新从 Leader 复制新的日志的情况出现。也就是算法上，需要能够对日志进行修改。这会使得 Raft 不能享受到只有追加写情况下硬盘的高性能。

那么，如果是你，你会采用什么样的方法来应对这个问题呢？欢迎在留言区分享出你的答案，也欢迎你把今天的内容分享给更多的朋友。

上一篇 32 | Raft（一）：不会背叛的信使

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

精选留言 (5)

写留言

dahai

2021-12-31

Raft集群 仍然可能会遇到：即使是压实后的数据仍然接近单机的存储，这时候可能要替换现有的机器换成更高配置的机器，这种情况下怎么处理？


Helios

2021-12-28

思考题：类似lsm tree的方式，通过append only把删除某条记录追加进去。


许灵

2021-12-27

关于思考题，应该可以用标记删除，加append only的方式进行，同时在log compaction的时候再进行物理删除


那时刻

2021-12-27

关于思考题，我的想法是，采用dataflow的高低水位方式来处理，从其它节点同步数据是低水位，自己节点写入数据是高水位。如果需要删除日志，从低水位复写就好了，同时利用了硬盘顺序写


csyangchsh

2021-12-27

关于思考题，当找到共同点后，是否可以直接将共同点之前的日志复制到新文件，在新文件上日志写入就可以是append only方式了。因为有快照机制，共同点之前的日志数量应该不会很多。

共 1 条评论 >