13 | 智慧之火：详解分布式容错共识算法

2022-11-08 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >

讲述：郑建勋

时长 15:20　大小 14.01M

你好，我是郑建勋。

上节课，我们说分布式系统总是需要在可用性和数据一致性之间找到平衡，也就是既要确保当少部分节点发生故障时，程序仍然能够正确且正常地运行，又要保证分布式节点之间对某一事件达成共识。其实这并不是一件容易的事情，好在许多容错共识算法为我们提供了经过检验的解决方案。

可用性衡量了系统面对网络延迟、网络分区、系统故障时的容错能力。显然，如果我们遇到了极端的事件，例如地球毁灭了，我们是无法保证系统可用的。因此，大多数容错共识算法都会有一个前提，也就是至少需要大部分节点是正常的，这样系统才可以正常运行。

此外，大多数容错共识算法还有第二个前提，就像我在上节课提到的，不考虑节点中可能混入了攻击者，这样才能保证系统不会出现拜占庭问题。

而要保证数据一致性，当前比较有名的算法是：Paxos、Raft 和 Zab。我将其称为分布式容错共识算法，容错代表了在异常情况下仍然具有可用性和正确性，而共识代表的是数据的一致性，它意味着即便是在并发、异常等情况下也能达成共识。

Raft 在现代分布式系统中有着重要的地位，K8s 和课程中用到的 etcd 组件底层都采用了 Raft 算法。这节课，我就以 Raft 算法为例讲解容错共识算法是如何实现数据的一致性与可用性的。

我们先从与 Raft 密切相关的 Paxos 算法开始讲起。

Paxos 算法

Paxos 算法是历史比较悠久的容错共识算法，他由 Lamport 在 20 世纪 80 年代末期提出。

Paxos 算法中的节点分为了 3 个角色。

- 提议者（proposer）：负责提出一个值。
- 接收者（acceptor）：负责选择一个值。
- 学习者（learner）： 负责学习被选中的值。

简单来说，Paxos 算法可分为如下几个过程：

- 提议者选择一个提议编号 n，并把 prepare 请求发送给大多数接收者；
- 接收者回复一个大于等于 n 的提议编号；
- 提议者收到回复，并记录这些回复中最大的提议编号，然后将被选中的值和这个最大的提议编号作为一个 accept 请求，发送给对应的接收者；
- 如果一个接收者收到一个编号为 n 的 accept 请求，那么除非它已经回复了一个编号比 n 大的 prepare 请求，否则它会接受这个提议；
- 当接收者接受一个提议后，它会通知所有的 learner 这个提议，最终所有的节点都会就一个节点的提议达成一致。

Paxos 算法的核心思想是，通过让 proposer 与大多数 acceptor 提前进行一次交流，让 proposer 感知到当前提出的值是否可能被大多数 acceptor 接收。如果不能被接收，proposer

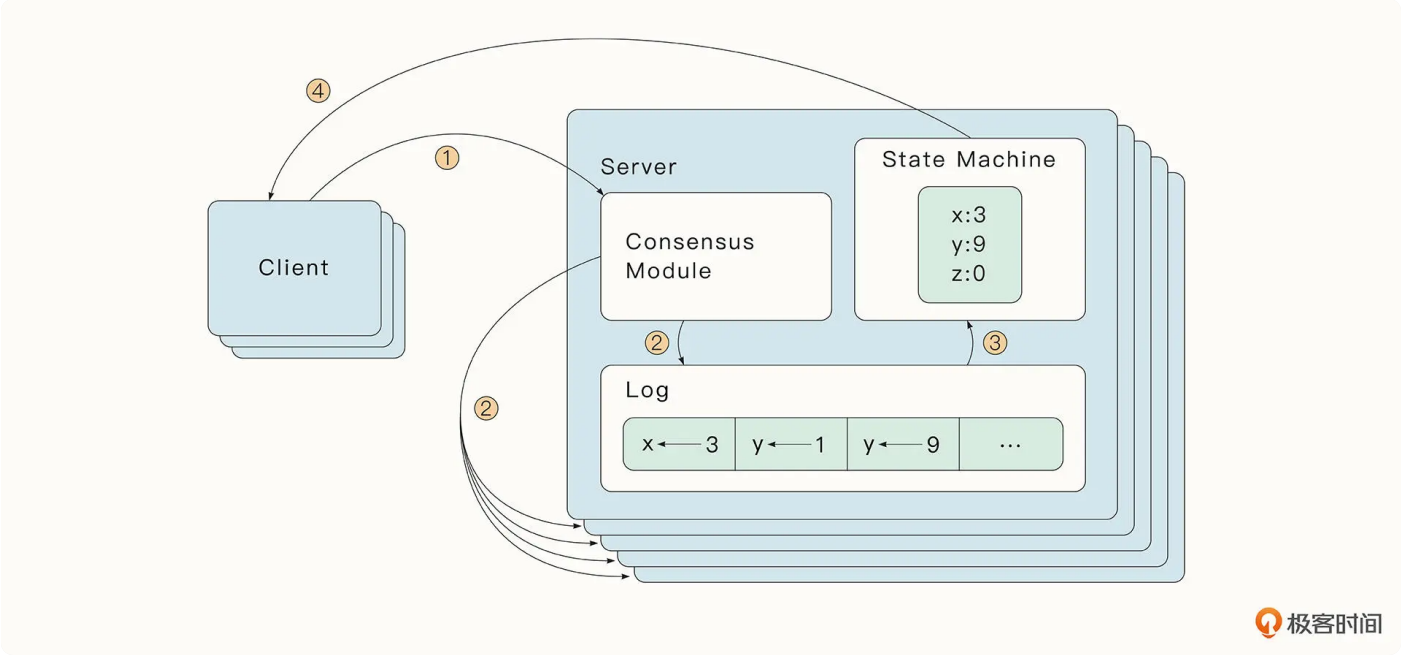
可以改变策略之后（例如增加提议编号，或接收某一个 proposer 已经提出的值）再继续进行协调，最终让大多数接收者就某一个值达成共识。Paxos 通过一个提议编码保证了后面被接收的值一定是编号更大的值，从而实现了写操作的线性一致性。

不过，Paxos 算法虽然描述起来非常简单，但是要完全理解它的原理却比较难。并且，Paxos 算法的官方描述中缺少对实现细节的诸多定义，导致实践中可以有多种灵活的实现方式。如果你对这个复杂的算法感兴趣的话，可以看看《分布式系统与一致性》这本书的第十章。

Raft 算法

总的来说，Paxos 理解起来还是比较困难，在工程上也比较难以实现，所以不太常用，Raft 算法就是在这一背景下诞生的。Raft 算法简单、容易理解、容易实现，已经成为现代多数分布式系统（例如 etcd、TiDB）采用的算法。

Raft 算法实现了一种复制状态机。每个分布式的状态机中存储了一份包含命令序列的日志文件，这些文件通过复制的形式传播到其他节点中。每个日志包含相同的命令，并且顺序也相同。状态机会按顺序执行这些命令并产生相同的状态，最终所有的状态机都将达到一个确定的最终状态。



在 Raft 算法中，每一个节点会维护一份复制日志（Replicated Log），复制日志中存储了按顺序排列的条目（Entry），用户执行的每一个操作都会生成日志中的一个条目，稍后这个条目会通过节点之间的交流复制到所有节点上。

如果一个条目是被大多数节点认可的，那么这种条目被称为 Committed Entry，这也是节点唯一会执行的条目类型。各个节点只要按顺序执行复制日志中的 Committed Entry，最终就会到达相同的状态。这样，即便节点崩溃后苏醒，也可以快速恢复到和其他节点相同的状态。

总结一下 Raft 算法的核心思想就是，保证每个节点具有相同的复制日志，进而保证所有节点的最终状态是一致的。

基本原理

Raft 中的节点有 3 种状态，领导者（Leader），候选人（Candidate）和跟随者（Follower）。

其中，Leader 是大多数的节点选举产生的，并且节点的状态可以随着时间发生变化。某个 Leader 节点在领导的这段时期被称为任期（Term）。新的 Term 是从选举 Leader 时开始增加的，每次 Candidate 节点开始新的选举，Term 都会加 1。

如果 Candidate 选举成为了 Leader，意味着它成为了这个 Term 后续时间的 Leader。每一个节点会存储当前的 Term，如果某一个节点当前的 Term 小于其他节点，那么节点会更新自己的 Term 为已知的最大 Term。如果一个 Candidate 发现自己当前的 Term 过时了，它会立即变为 Follower。

一般情况下（网络分区除外）在一个时刻只会存在一个 Leader，其余的节点都是 Follower。Leader 会处理所有的客户端写请求（如果是客户端写请求到 Follower，也会被转发到 Leader 处理），将操作作为一个 Entry 追加到复制日志中，并把日志复制到所有节点上。而 Candidate 则是节点选举时的过渡状态，用于自身拉票选举 Leader。

Raft 节点之间通过 RPC（Remote Prcedure Cal，远程过程调用）来进行通信。Raft 论文中指定了两种方法用于节点的通信，其中，RequestVote 方法由 Candidate 在选举时使用，AppendEntries 则是 Leader 复制 log 到其他节点时使用，同时也可以用于心跳检测。RPC 方法可以是并发的，且支持失败重试。

Raft 算法可以分为三个部分：选举、日志复制和异常处理。下面我们分阶段介绍一下。

选举与任期

在 Raft 中有一套心跳检测，只要 Follower 收到来自 Leader 或者 Candidate 的信息，它就会保持 Follower 的状态。但是如果 Follower 一段时间内没有收到 RPC 请求（例如可能是 Leader 挂了），新一轮选举的机会就来了。这时 Follower 会将当前 Term 加 1 并过渡到 Candidate 状态。它会给自己投票，并发送 RequestVote RPC 请求给其他的节点进行拉票。

Candidate 的状态会持续，直到下面的三种情况发生。

- 如果这个 Candidate 节点获得了大部分节点的支持，赢得选举变为了 Leader。一旦它变为 Leader，这个新的 Leader 节点就会向其他节点发送 AppendEntries RPC， 确认自己 Leader 的地位，终止选举。
- 如果其他节点成为了 Leader。它会收到其他节点的 AppendEntries RPC。如果发现其他节点的当前 Term 比自己的大，则会变为 Follower 状态。
- 如果有许多节点同时变为了 Candidate，则可能会出现一段时间内没有节点能够选举成功的情况，这会导致选举超时。

为了快速解决并修复这第三种情况，Raft 规定了每一个 Candidate 在选举前会重置一个随机的选举超时（Election Timeout）时间，这个随机时间会在一个区间内（例如 150-300ms）。

随机时间保证了在大部分情况下，有一个唯一的节点首先选举超时，它会在大部分节点选举超时前发送心跳检测，赢得选举。如果一个 Leader 在心跳检测中发现另一个节点有更高的 Term，它会转变为 Follower，否则将一直保持 Leader 状态。

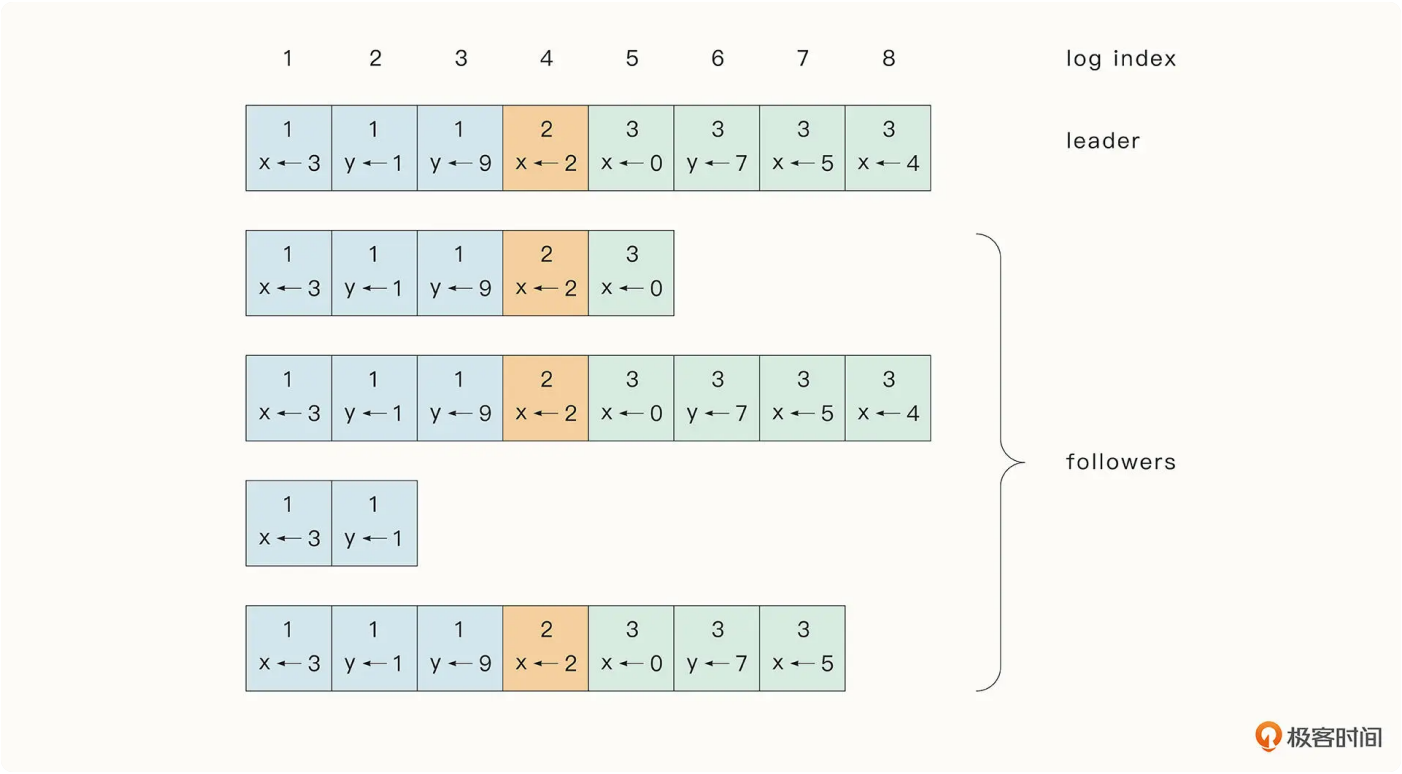
日志复制（Log Replication）

一个节点成为 Leader 之后，会开始接受来自客户端的请求。每一个客户端请求都包含一个节点的状态机将要执行的操作（Command）。Leader 会将这个操作包装为一个 Entry 放入到 log 中，并通过 AppendEntries RPC 发送给其他节点，要求其他节点把这个 Entry 添加到 log 中。

当 Entry 被复制到大多数节点之后，也就是被大部分的节点认可之后，这个 Entry 的状态就变为 Committed。Raft 算法会保证 Committed Entry 一定能够被所有节点的状态机执行。

一旦 Follower 通过 RPC 协议知道某一个 Entry 被 commit 了，Follower 就可以按顺序执行 log 中的 Committed Entry 了。

如图所示，我们可以把 log 理解为 Entry 的集合。Entry 中包含了 Command 命令（例如 x←3），Entry 所在的 Term（方框里面的数字），以及每一个 Entry 的顺序编号（最上面标明的 log index，顺序递增）。



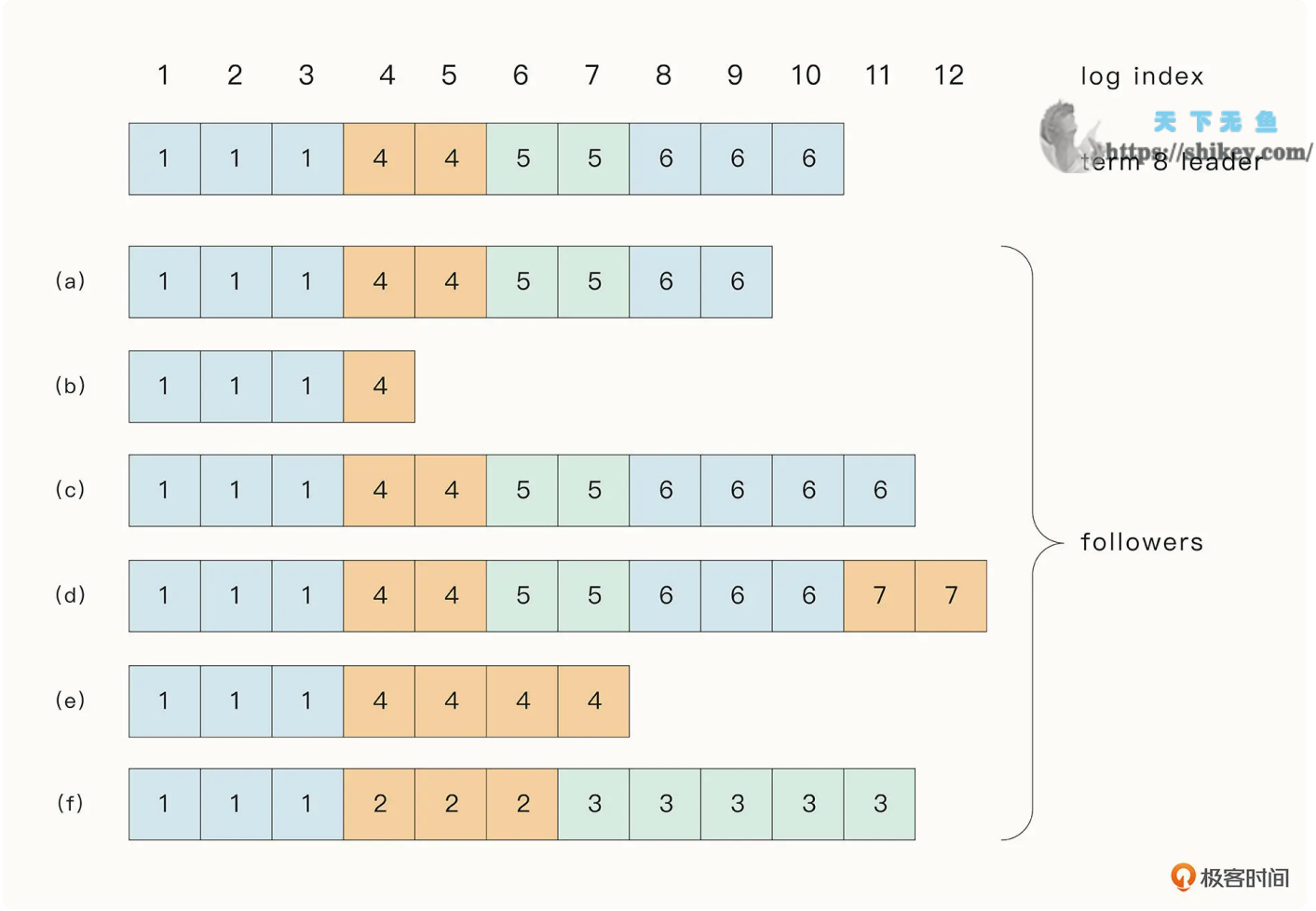
异常处理

但这里还有一个重要的问题，就是 Raft 节点在日志复制的过程中需要保证日志数据的一致性。要实现这一点，需要确认下面几个关键的属性：

- 如果不同节点的 log 中的 Entry 有相同的 index 和 Term, 那么它们存储的一定是相同的 Command；
- 如果不同节点的 log 中的 Entry 有相同的 index 和 Term，那么这个 Entry 之前所有的 Entry 都是相同的。

接下来我们就来看看，Raft 算法是怎么在不可靠的分布式环境中保证数据一致性的。

在实际生产过程中，Raft 算法可能会因为分布式系统中遇到的难题（例如节点崩溃），出现多种数据不一致的情况。如下所示，a → f 分别代表 Follower 的复制日志中可能遇到的情况，方框中的方格表示当前节点复制日志中每一个 Entry 对应的 Term 序号。



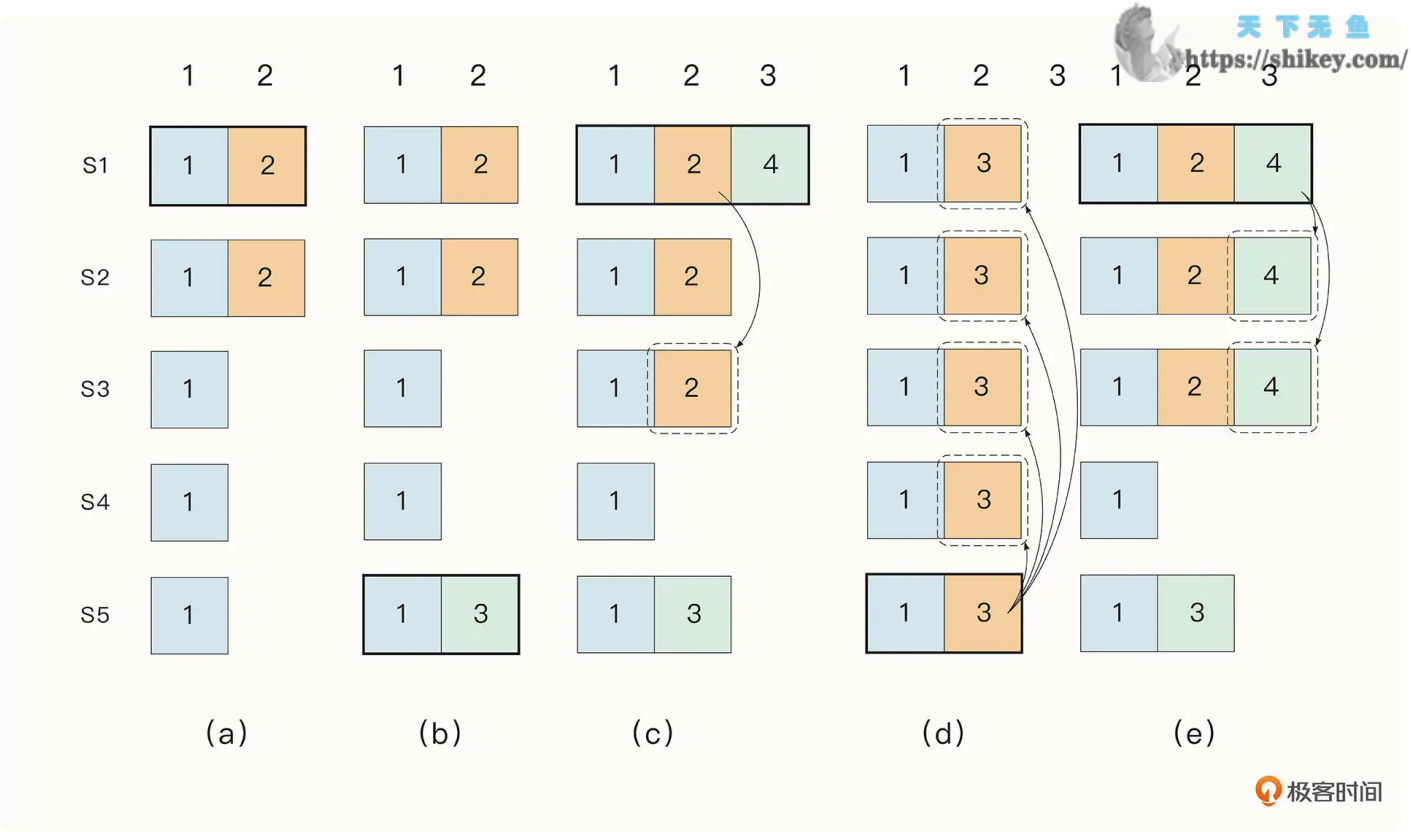
a → e 的情况你可以想一想什么时候会发生，我在这里重点解释一下 f 这种情况，因为 f 看起来是最奇怪的。

f 这种情况可能是这样的：f 是 Term 2 的 Leader, 它添加 Entry 到 log 中之后，Entry 还没有复制到其他节点，也就是说，还没等到 commit 就崩溃了。但是它快速恢复之后又变为了 Term 3 的 Leader， 再次添加 Entry 到 log 之后，没有 commit 又崩溃了。当 f 再次苏醒时，世界已然发生了巨变。

所以我们可以看到，在正常的情况下，Raft 可以满足上面的两个属性，但是异常情况下，这种情况就可能被打破，出现数据不一致的情况。为了让数据保持最终一致，Raft 算法会强制要求 Follower 的复制日志和 Leader 的复制日志一致，这样一来，Leader 就必须要维护一个 Entry index 了。在这个 Entry index 之后的都是和 Follower 不相同的 Entry，在这个 Entry 之前的都是和 Follower 一致的 Entry。

Leader 会为每一个 Follower 维护一份 next index 数组，里面标志了将要发送给 Follower 的下一个 Entry 的序号。最后，Follower 会删除掉所有不同的 Entry，并保留和 Leader 一致的复制日志，这一过程都会通过 AppendEntries RPC 执行完毕。

不过，仅仅通过上面的措施还不足以保证数据的一致性。想想下图这个例子：



从这张图可以看出，一个已经被 Committed 的 Entry 是有可能被覆盖掉的。例如在 a 阶段，节点 s1 成为了 Leader，Entry 2 还没有成为 Committed。在 b 阶段，s1 崩溃，s5 成为了 Leader ，添加 Entry 到自己的 log 中，但是仍然没有 commit。在 c 阶段，s5 崩溃，s1 成为了 Leader，而且在这个过程中 Entry 2 成为了 Committed Entry。接着在 d 阶段 s1 崩溃，s5 成为了 Leader，它会将本已 commit 的 Entry 2 给覆盖掉。但我们真正想期望的是 e 这种情况。

怎么解决这个问题呢？Raft 使用了一种简单的方法。Raft 为 Leader 添加了下面几个限制：

- 要成为 Leader 必须要包含过去所有的 Committed Entry；
- Candidate 要想成为 Leader，必须要经过大部分 Follower 节点的同意。而当 Entry 成为 Committed Entry 时，表明该 Entry 其实已经存在于大部分节点中了，所以这个 Committed Entry 会出现在至少一个 Follower 节点中。因此我们可以证明，当前 Follower 节点中，至少有一个节点是包含了上一个 Leader 节点的所有 Committed Entry 的。Raft 算法规定，只有当一个 Follower 节点的复制日志是最新的（如果复制日志的 Term 最大，则其日志最新，如果 Term 相同，那么越长的复制日志越新），它才可能成为 Leader。

总结

在分布式系统中，让大多数节点就某一事件达成一致并不是一件容易的事情，因为会存在网络延迟，节点崩溃等异常情况，而这就是分布式容错共识算法为我们解决的问题。这节课，我们看到了分布式容错共识算法 Raft 构建复制状态机的过程，看到了它保证数据一致性的方法和在故障情况下的容错能力。

在 Raft 算法中，写请求具有线性一致性，但是读请求由于 Follower 节点数据暂时的不一致，可能会读取到过时的数据。因此，Raft 保证的是读数据的最终一致性，这是为了性能做的一种妥协。但我们可以在此基础上很容易地实现强一致性的读取，例如将读操作转发到 Leader 再读取数据。

在容错方面，Raft 通过合理的 Leader 选择以及 Leader 与 Follower 之间强制的日志同步，在保证数据正确性的基础上，也能保证当前 Leader 崩溃、网络分区、网络延迟之后大部分节点仍然能够正常工作。

Raft 在工程上实现起来是相对比较容易的。在 Raft 的论文中对算法的实现细节有详细的描述，感兴趣的话推荐你看看。如果你想实践一下如何把枯燥的论文变为现实，我推荐给你两份资料，一个是 MIT 的经典分布式课程 6.824，他用实验的形式最后用 Go 语言实现了 Raft 算法，另外你还可以参考一下 etcd 对 Raft 的实现。

思考题

最后，还是给你留两道思考题。

1. 就像前面提到的，如果希望具有一致性读，一种手段是将读操作转发到 Leader。但是当前的 Leader 并不一定是真正的 Leader，例如，刚好网络分区导致 Leader 变更，这样的话，我们还是有可能读取到过时的数据。你知道怎么解决这类问题吗？
2. Raft 节点出现网络分区的情况，由于一部分节点由于无法与 Leader 取得通信，会一直增加 Term 号。在网络恢复后，由于这些节点 Term 号更大，因此会变为 Leader。 然而这种处理看起来并不太合理，你知道有什么办法可以解决这个问题吗？

欢迎你在留言区与我交流讨论，我们下节课见！

上一篇 12丨分布式系统设计：数据一致性与故障容错的纠葛

下一篇 14丨谋定而动：爬虫项目需求分析与架构设计

精选留言 (2)

写留言

Realm 酥

2022-11-09 来自浙江

1 猜测一下，在结果返回给请求方之前，查下自己有没有收到新leader的心跳包，有的话，重新向新leader请求最新数据；没有的话，就把结果返回给请求者，这可能也还是有概率拿不到最新数据；

2 要成为 Leader 必须要包含过去所有的 Committed Entry；

2

一步 酥

2022-11-17 来自广东

我记得 如果读请求到达 follow 节点，follow会请求 leader 节点 来判断 commit index 是否是最新的， 如果不是就会进行同步，在进行响应