第 6 章　Raft——共识算法的宠儿

Multi Paxos 理解起来很晦涩，其复杂的架构设计在工程实现中也困难重重。而 ZAB 是为 ZooKeeper 量身定做的，那么解决分布式共识问题又迫在眉睫，难道就没有一个易于理解且门槛较低的共识算法吗？

很明显，计算机领域的工程师们并不会止步于此。为了解决上述问题，必须要有一个新的共识算法，它将作为所有分布式系统核心算法的首选，不仅要求算法架构设计更简单，还要降低使用者的学习成本。Raft 正是在这样的目的指引下诞生了。

Raft 从问世开始就备受关注，被认为是所有共识算法的最优解，无论是工程实现还是协商效率的效果都很好。如果你正在设计一个分布式系统，正在寻找一个简单、有效的共识算法作为核心算法的话，笔者强烈推荐使用 Raft。

6.1　Raft 简介

Raft 与 ZAB 不同，Raft 并不是为了某个系统量身定做的，它与大多数共识算法一样，期望作为一个通用的类库，成为所有共识问题的通用解决方案。

6.1.1　Raft 诞生的背景

很多人对 Raft 的了解来源于斯坦福大学的 Diego Ongaro 和 John Ousterhout 教授联合发表的名为 *In Search of an Understandable Consensus Algorithm* 的论文。通过论文标题（中文为寻找易于理解的共识算法），可见斯坦福大学的研究人员对 Raft 的可理解性的重视。在此之前，斯坦福大学教授学生共识算法相关知识的主要工具是 Paxos，但是 Paxos 太难理解，尽管已有许多工程师为了使其更易懂做出了很多尝试。但是，Paxos 要真正用于实际生产中，还需要改变其复杂的架构。

在工程实现中没有其他共识算法可以代替 Paxos，工程师们陷入了与 Paxos 的斗争之中。Paxos 论文的主要内容是关于 Basic Paxos 的，虽然论文定义了可能实现 Multi Paxos 的方法，但是缺少实现的细节，因此在很长一段时间内没有一个被大众所认同的 Multi Paxos 算法。虽然在 Raft 诞生之初就有工程师尝试具体化 Paxos 的实现，但是这些实现方法各不相同，并且与 Lamport 描述的也不同。例如，Chubby 这样的大型系统已经实现了类似 Paxos 的算法，但是并没有公布细节，我们也就无法得知工程师们实现的 Paxos 是否能被大众接受。

实际上，几乎所有的共识系统都是从 Paxos 开始的，在具体实现过程中发现有很多难

以逾越的难题，于是又开发了一种与 Paxos 完全不一样的架构，这样既费时又容易出错。虽然 Lamport 论证过 Paxos 的正确性，但是实际开发出来的系统与 Paxos 的差别太大，导致这些证明对实际系统没有太大的价值。借用 Chubby 作者的一句话：

Paxos 算法描述和工程实现之间存在巨大的鸿沟，最终的系统往往建立在一个还未被证明的算法之上。

6.1.2　可理解性

考虑到共识问题在大规模系统中的重要性，同时为了在共识问题上提供更好的教学方法，斯坦福大学的学者们决定设计一个可以代替 Paxos 的共识算法，该算法的首要目的就是可理解性，它必须保证能够被大多数人理解。此外，容错与高效也是必要条件。为了降低理解难度，Raft 做出的努力包括问题分解、消除不确定性及详细的细节实现。

- 问题分解：Raft 尽可能将问题拆分成多个独立且易于理解的子问题。例如，Raft 被拆分为 Leader 选举、日志复制、安全性及成员变更 4 个部分。
- 消除不确定性：尽可能消除不确定性，使得学习者尽可能减少在异常情况下遇到的问题。例如，Raft 要求所有日志必须要连续，不允许存在空洞。这样可以快速发现成员之间日志的差异，而不用像 Paxos 一样逐个重确认。另外，一切以 Leader 的数据为基准，这使得数据对齐工作显得格外简单，Follower 只需要被动地复制 Leader 的数据即可。
- 详细的细节实现：这是非常有必要的，这样能大大减少工程师的工作。给出细节实现能避免每位工程师对实现方式产生歧义如对超时的引入。

6.1.3　基本概念

Raft 涉及的关键术语可以分为 3 个方面，即成员状态、运行阶段和 RPC 消息。另外，多数派任期也会在本节中介绍。

1．多数派

Raft 和 Paxos 一脉相承，采用多数派决策的思想。多数派接受某一个日志项，即意味着整个集群接受该日志项。多数派的定义和 Paxos 一样，在一组成员Π中，任意一个多数派 Q 满足以下条件：

$$\forall q \in Q : Q \subseteq \Pi$$
$$\forall Q_1, Q_2 \in Q : Q_1 \cap Q_2 \neq \varnothing$$

2．任期

Raft 将时间分割为不同长度的任期，记作 term。term 可以类比 ZAB 中的 epoch，term 使用连续的整数进行单数递增，并随着 RPC 消息在成员之间交换。Raft 使用 term 作为 Leader 的任期号并遵循以下规则：

- 每个 Raft 成员在本地维护自己的 term。
- 在每轮 Leader 选举之前递增自己本地的 term。
- 每个 RPC 消息都将携带自己本地的 term。
- 每个 Raft 成员拒绝处理小于自己本地 term 的 RPC 消息。

这些规则可以让 Follower 使用 term 发现一些过期的消息和过时的 Leader。如果一个 Candidate 或者 Leader 发现自己的 term 小于接收的消息中的 term，则会立即回到 Follower 状态。

Raft 并没有强制要求在集群中只能存在一个 Leader，而是规范了一个 term 只会存在一个 Leader。这意味着 Raft 允许新旧多个 Leader 同时存在，此时需要通过 term 来拒绝那些过期的 Leader。

3．成员状态

在基础的 Raft 算法中，Raft 为每个成员定义了 3 种状态，分别代表它们当前所扮演的角色：领导者（Leader）、跟随者（Follower）和候选人（Candidate）。任何时候的任意 Raft 成员，都处于 3 种状态之一。正常情况下，在集群中只会存在一个 Leader，其他成员都为 Follower，Candidate 只是 Leader 在选举过程中的中间状态。3 种角色互相转换的逻辑如图 6.1 所示。

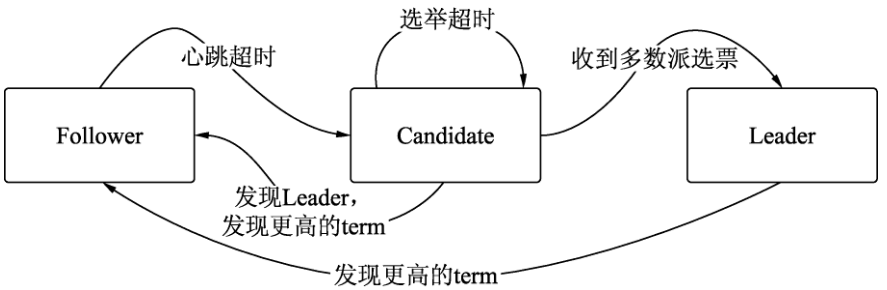

图 6.1　成员角色转换示意

Follower 在等待 Leader 的心跳超时后会进入 Candidate，Leader 只会从 Candidate 成员中诞生。当某个 Candidate 获得多数派选票后将晋升为 Leader，其余成员将变更为 Follower。Leader 在失去多数派的 Follower 或者收到来自更高的 term 的 RPC 消息后会重新回到 Follower 状态。

为了适应复杂的生产场景，Raft 又引入了无投票权成员及 Witness 成员。前者用于新成员上线时尽快赶上 Leader 的数据；后者用于在不降低集群可用性的基础上降低部署成本。具体总结如下：

- Follower：只会处理和响应来自 Leader 和 Candidate 的请求。如果一个客户端和 Follower 通信，Follower 会将请求重定向给 Leader，可以类比为 ZAB 中的 Follower。与其不同的是，在新 Leader 诞生后，ZAB 的 Leader 会参考 Follower 的数据而 Raft 的 Follower 只会以 Leader 的数据为准，被动地接收。
- Candidate：如果 Leader 出现故障或者 Follower 等待心跳超时，则 Follower 会变更为 Candidate，新的 Leader 只能从 Candidate 中产生。

- Leader：当某一个 Candidate 获得多数派的选票时，该成员晋升为 Leader，负责处理接下来的客户端请求、日志复制管理及心跳消息管理。
- 无投票权成员：当新成员上线时，以学习者的身份加入集群，在不影响集群的情况下，快速和 Leader 完成数据对齐。无投票权成员也可用于在不影响协商效率的基础上提供额外的数据备份。
- Witness 成员：一个只投票但不存储数据的成员，它对存储资源的需求较小。通常在一个由 5 个成员组成的集群中，可以选择其中一个成员作为 Witness，在不降低容错性的同时，节省了部分成员的资源。

Raft 是强 Leader 模型的算法，日志项只能由 Leader 复制给其他成员，这意味着日志复制是单向的，Leader 从来不会覆盖本地日志项，即所有日志项以 Leader 为基准。

4．运行阶段

Raft 强化了 Leader 的地位，整个算法基于 Leader 的服务状态可清晰地分成两个阶段。

- Leader 选举：在集群启动之初或者 Leader 出现故障时，需要选出一个新 Leader，接替上一任 Leader 的工作。
- 日志复制：在该阶段中，Leader 可以接收客户端请求并进行正常处理，由 Leader 向 Follower 同步日志项。

5．RPC消息

RPC 消息分为 4 种，其中，前两种是 Raft 基础功能的必要实现，后两种是为了优化 Raft 而引入的。

- RequestVote：请求投票消息，用于选举 Leader。
- AppendEntries：追加条目消息，用于以下场景。
  - Leader 用该消息向 Follower 执行日志复制。
  - Leader 与 Follower 之间的心跳检测。
  - Leader 与 Follower 之间的日志对齐。
- InstallSnapshot：快照传输消息。
- TimeoutNow：快速超时消息，用于 Leader 切换，可立即发起 Leader 选举。

6.2　Raft 算法描述

Raft 作为一个强 Leader 模型的算法（Based-Leader），不需要刻意地拆分，其算法本身就由 Leader 选举和日志复制两部分组成。接下来从这两个方面分别介绍 Raft。

6.2.1　Leader 选举

Raft 算法的日志项只能由 Leader 流向 Follower。为了保证算法的安全性，新晋升的

Leader 一定要满足以下特性才能保证已经达成共识的日志项不会被覆盖。

新晋升的 Leader 不一定拥有最新的日志项，但是一定要拥有已复制到多数派的日志项。

Follower 和 Leader 之间采用心跳机制来检测对方的状态，Leader 会周期性地向所有的 Follower 发送心跳信息（AppendEntries 消息）来维持自己的领导地位，而 Follower 只要能接收到心跳信息将永远停留在 Follower 状态。如果 Follower 在超过约定的时间内未接收到心跳，则会进入 Candidate 状态，从而发起 Leader 选举。

1．选举进程

当集群刚启动时，所有成员都为 Follower，此时并不存在一个 Leader 向它们发送心跳信息。Follower 在等待一个超时后进入 Candidate 状态，然后发起 Leader 选举并通过 RequestVote 消息广播自己的选票，每张选票包含所支持成员的最新日志索引和所处的任期（term）。直到发生以下情况，当前成员才会退出 Candidate 状态。

- ❑ 自己获得多数派的选票而赢得了本次选举，进入 Leader 状态。
- ❑ 其他成员获得了多数派的选票，进入 Follower 状态。
- ❑ 在一段时间内，没有任何成员获得多数派的选票。

在每次 Leader 选举之前，Candidate 都会递增自己的任期。任意成员投出的每一张选票都是针对一个任期的，在同一个任期内，每个成员都只会投出一张选票，而不会像 ZAB 那样每次都变更选票并重新广播。在投票规则允许的情况下，采用先来先服务的原则。

例如，成员 A 收到来自成员 B 和 C 的同一任期的选票，如果成员 B 和 C 都符合成员 A 投票规则，并且成员 B 的选票比成员 C 的选票先到达成员 A，那么在该任期内，成员 A 会将选票投给成员 B。

当一个 Candidate 获得多数派的选票时，即表示获得本次选举的胜利，进入 Leader 状态。接着，Leader 周期性地向所有 Follower 发送心跳消息（AppendEntries 消息）来维持自己的领导地位，以防止 Follower 发起新的选举而影响集群的稳定性。

当一个 Candidate 在等待选票的过程中收到其他成员发来的 AppendEntries 消息，并且该消息中的任期大于等于自己的任期时，则意味着该成员已经获得了多数派的选票而晋升为 Leader，那么当前成员会退出 Candidate 状态并进入 Follower 状态。在此期间，如果 Candidate 收到 AppendEntries 消息中的任期小于自己的任期，即便这可能是上一任 Leader 迟到的心跳消息并且上一任 Leader 很有可能还可以正常处理请求，则当前成员也会拒绝该消息而继续推进新 Leader 的选举流程。

上面说的第三种情况——没有任何成员获得多数派的选票，这意味着同时存在多个 Candidate 发起 Leader 选举，其选票被瓜分。这种情况在 Raft 中称为 Split Vote，当没有其他方式干预时，选举有可能进入无限循环的状态，永远不会有新的 Leader 诞生。

2．分割选举

分割选举是上面讲到的 Split Vote。至于干预的方式，ZAB 采用的是为每个成员设置一个 myid，并且让 myid 参与投票规则的设定。而 Raft 采用随机超过约定时间的方式来规避多个 Candidate 同时发起 Leader 选举，以降低发生 Split Vote 的概率。

每位 Raft 成员都有一个属于自己的随机等待心跳超时，如果在等待心跳超过约定的时间内没有收到心跳信息，则会进入 Candidate 状态并发起选举。这个设定降低了多个成员同时发起选举导致，选票被瓜分的可能性。

假设在极端情况下，多个成员的随机等待心跳超时一样并且同时发起了选举，导致选票被瓜分，每个 Candidate 为每次选举设置了随机等待选票的超时，因此，即使选票被瓜分，随机等待选票的超时也进一步控制了下一次发起选举的成员数量。

双重随机超时使选举被分割的情况变少。在 Leader 选举过程中，随机超时包含两层含义：

- 随机等待心跳的超时。
- 随机等待选票的超时。

3．投票规则

对于任何基于 Leader 的共识算法，Leader 最终必须拥有所有已提交的日志项。共识算法通常会采用一些额外的机制来收集所有日志项并传送给 Leader，以保证 Leader 拥有最完整的数据，这些机制通常需要大量对比来确认哪些日志项是缺失的，因此会增加算法的复杂度。

增加算法的复杂度是与 Raft 的设计理念相悖的，Raft 期望采用更为简单的方式来保证 Leader 日志项的完整性。与 ZAB 一样，Raft 试图在选举阶段就保证晋升的新 Leader 一定拥有之前所有已提交的日志项，而不用再采用其他机制给 Leader 传送日志项。这种方式的唯一关注点是如何制定一套简单而有效的投票规则。

- 任期长的成员拒绝投票给任期短的 RequestVote 消息。
- 最后一条日志项编号（uncommited）大的成员拒绝投票给最后一条日志项编号（uncommited）小的成员。
- 每个成员在一届任期内只投出一张选票，先来的成员先获得投票。

以上投票规则可以有效阻止那些没有拥有完整数据的成员晋升为 Leader。Candidate 为了获得选举的胜利，必须要与集群内的多数派成员通信。因此在这多数派的成员中一定存在一个成员拥有最完整的数据（即拥有所有已提交的日志项）。

遵循以上投票规则，如果 Candidate 获得了多数派的选票，则意味着该 Candidate 比多数派成员中的任意一个成员的数据都更加完整或者和多数派成员的数据一样完整，这说明该 Candidate 在整个集群中拥有最完整的数据。

4．稳定的Leader

Leader 是 Raft 的核心，一个稳定的 Leader 对 Raft 的可用性至关重要。在 Leader 选举期间，Raft 不能继续向客户端提供服务，因此我们并不希望进行非必要的 Leader 变更。而触发 Leader 选举最重要的条件是心跳间隔时间。

为了让 Raft 的使用者在进行系统设计时有一个参考，Raft 给出了以下不等式，只要系统满足以下要求，Raft 就能选举并维护一个稳定的 Leader。

消息交互时间 << 心跳间隔时间 << 平均故障时间

- 消息交互时间：Leader 并行地向 Follower 发送 RPC 消息并收到响应的平均时间，这里的 RPC 消息通常是 AppendEntries。
- 心跳间隔时间：Follower 等待心跳超过约定的时间，即触发 Leader 选举超过约定的时间。
- 平均故障时间：对于一个成员来说，两次故障间隔时间的平均值即为平均故障时间。

消息交互时间远小于心跳间隔时间是为了避免在消息传输过程中出现延迟、阻塞而引发 Leader 选举。在这种情况下，Leader 通常处于正常运行的状态。

上面的不等式的主要作用是设定心跳间隔时间范围。例如，一轮 AppendEntries 消息需要经过传输及持久化，预估消息交互时间为 5～20ms，这取决于持久化技术，因此心跳间隔时间需要设置为 20～500ms。而平均故障时间通常不由我们控制，大多数服务器的平均故障时间为几个月甚至更长。

6.2.2　日志复制

1．日志项

在 Raft 中，数据都是以日志项（log entry）的形式保存的，客户端每一次的事务请求，都会封装成一个日志项记录在日志中。

日志是一个无限长的列表，列表中的每一个元素都有固定的编号，如表 6.1 所示，每个成员都维护着一个本地日志列表，其中，每个元素就是一个日志项。日志项除了存储事务操作之外，同时还保存着 term 及日志索引。

表 6.1　日志存储

term	1		2	3		
Log Index	1	2	3	4	5	6
Leader A	Add	Mul	Add	Sub	Div	Add
Follower B	Add	Mul	Add	Sub	Div	
Follower C	Add	Mul	Add	Sub	Div	
Follower D	Add	Mul	Add	Sub		
Follower E	Add	Mul				

- 事务操作：用于输入状态机的指令，通常来自客户端事务请求中的操作。
- term：Leader 提出该日志项时的任期号，是判断该日志项是否过期的有效依据。
- 日志索引：全局唯一且连续递增的整数编号，用于标识每个日志项在日志列表中所处的位置。其连续性不限于多个 Leader 周期。

前面提到，Raft 规定日志索引必须是连续的，不允许出现空洞，虽然这是减少不确定因素的有效手段，但是这样设计的好处远不止于此。例如：

- Leader 选举之后，无须考虑在新 Leader 的本地日志中是否存在空洞的日志，从而向其他成员拉取这些空洞的日志。

- Follower 可以用最大的日志索引快速定位自己缺失的数据，无须额外加入遍历日志项或者 checksum[①]机制，以此确保在此之前的日志项不存在空洞。
- 提升了协商效率，将协商过程优化为一阶段。通过下一轮协商的日志索引来确定在该日志索引之前的日志项都能提交（包括上一任 Leader 提出的日志项），因此不需要类似提交阶段的过程。

另外，Raft 只保证已提交的日志项永远不会丢失，因此我们需要准确地定义已提交的日志项和已提出的日志项。

- 已提交的日志项：集群中任意一个成员在本地提交过该日志项，如在表 6.1 中日志索引为 1～5 的日志项。
- 已提出的日志项：该日志项可能被 Leader 提出，也可能被成功复制到集群中[0, N]个成员的日志中，其中，N 为集群成员总数，如在表 6.1 中日志索引为 6 的日志项。

对于已提交的日志项，是一个比较严苛的定义，假设某个日志项被成功复制到多数派成员的日志中，但 Leader 没有来得及在本地日志中提交该日志项，那么它仍然只能被称为已提出的日志项，尽管它已具备了提交的条件。

2．日志复制

处理客户端的事务请求，就是把日志项复制给其他成员并应用到各自状态机的过程，而日志项只能由 Leader 流入 Follower。因此，客户端的事务请求只能由 Leader 处理。Leader 与 Follower 之间的交互步骤如下：

（1）Leader 首先以日志项的形式将该事务请求追加至本地日志中。

（2）Leader 并行地通过 AppendEntries 消息将该日志项广播给 Follower。

（3）Follower 按照 AppendEntries 消息将日志项追加至本地日志中。

（4）Follower 将执行结果发送给 Leader，通常这里可以忽略执行失败的响应，不影响 Raft 的安全性。

（5）Leader 收到多数派 Follower 复制成功的响应后会立即提交日志项，其提交操作包含状态转移。

（6）Leader 执行结果返回给客户端。

日志复制只由一个阶段组成，Leader 只会发送一次 AppendEntries 消息。因此，Raft 本身就是一个优化之后的成果。

Raft 可以保证已提交的日志项永远不会丢失，Leader 已提交的日志项一定已被成功复制到多数派成员的日志中。因此即使 Leader 在此时出现故障，下一任 Leader 也一定拥有该日志项。对于只是被提出的日志项，该日志项能否被提交，取决于下一任 Leader 是否拥有该日志项。

下面介绍 Follower 接受日志项的条件。在多数派思想的约定下，可以保证单个日志项在同一个日志索引上达成共识，但是无法保证两个成员在两个日志索引上的日志项是完全一致的。因为总存在一些少数派在某个日志索引上没有和多数派达成一致的情况，如果不

① checksum：按照一个特定的算法来比较两份体量较大的数据是否完全一致。

加任何约束，这些少数派可以在下一个日志索引上成为多数派，这种情况造成的结果是执行状态转移的顺序不一样，进而导致状态机输出的结果不一致，这是不能接受的。

因此，我们需要对 Follower 进行一致性检查，这个检查要求 Follower 接受某个日志项的条件是已拥有该日志项的前一个日志项。具体讲，在 Leader 发送的 AppendEntries 消息中，需要携带 Leader 上一个日志索引及其所处的 term，如果 Follower 在本地日志中找不到该日志索引，那么它就应该拒绝 AppendEntries 消息。因此，Leader 收到 AppendEntries 消息接收成功的响应，说明该 Follower 与 Leader 上一个日志项是相同的。

按照归纳法，如果 Follower 给 Leader 回复 AppendEntries 消息接收成功的响应，Leader 就可以推测出该 Follower 从第一个日志索引到 AppendEntries 消息中的日志索引之间所有的日志项与 Leader 完全相同。

之所以只需要比较前一个日志项的索引及其日志项所处的 term，就能断定前一个日志项是否完全相同，是因为日志项只能由 Leader 提出，并且在一个 term 上只会存在一个 Leader，而一个 Leader 不会在同一个日志索引上提出两个不同的日志项。

综上所述，我们总结出日志匹配特性如下：

如果在两个成员中，两个日志项拥有相同的日志索引和 term，那么它们存储着相同的指令。

如果在两个成员中，两个日志项拥有相同的日志索引和 term，那么在这个相同的日志项之前的所有日志项也都相同。

3．日志提交

可能读者会有疑问，Follower 什么时候提交该日志项呢？答案是由 Leader 发送心跳的 AppendEntries 消息或者下一个日志协商的 AppendEntries 消息来通知 Follower 提交日志项。这样的做法，可以使协商优化成一阶段，大幅提高协商效率，并且不会影响算法的安全性，这种优化是高效且有意义的。

为此，Raft 引入了 committedIndex 变量，committedIndex 是已达成共识的日志索引，也是应用到状态机的最大的日志索引。根据日志复制的过程，在第一轮 AppendEntries 消息中，Follower 只会持久化日志项，并不会执行提交操作。只有 Leader 才知道该日志项是否已成功复制到多数派，是否可以执行提交。

当 Leader 收到多数派 Follower 的成功响应后，Leader 将提交该日执项，并更新 committedIndex，同时在下一次心跳的 AppendEntries 消息或者下一个日志协商的 AppendEntries 消息中携带 committedIndex。

Follower 无论收到哪一类 AppendEntries 消息，都会从消息中取得 committedIndex，因此在 Follower 的本地日志中，所有小于或等于 committedIndex 的日志均可以执行提交操作。

4．并行协商

因为Raft规范了日志的顺序，在提交日志项和执行状态转移操作时都涉及日志的顺序。这种约束虽然做不到完全的并行协商，但是仍可以做一些优化来提升效率。

在允许少数 Follower 数据落后的条件下，可以让 Leader 并行给每个 Follower 发送日

志项，但是对于单个 Follower 来说，仍要求 Leader 一个个地发送日志项。

当 Leader 收到事务请求时，不必去理会上一个日志项是否已完成提交，Leader 仍然可以处理本次事务请求。按照处理逻辑，Leader 为新的事务请求单调递增地分配日志索引，并在本地记录该日志项，然后并行地向每个 Follower 按照日志顺序发送日志项。这并不会影响 Raft 的安全性，我们从两方面来讲述：执行状态转移的顺序（提交顺序），以及是否存在空洞日志项。

有两个日志项 γ 和 β，并且 γ 在 β 之前，是否存在 β 在 γ 之前提交的情况？答案是不会的。虽然在 β 持久化时 γ 可能未提交，但是 β 绝不会在 γ 之前提交。因为 Follower 接受 β 的一致性检查约束该 Follower 一定拥有 γ；而对于单个 Follower，Leader 需要该 Follower 已经成功响应 γ 才能发送 β。综上所述，Leader 一定是先获得 γ 的多数派成功响应，然后再获得 β 的多数派成功响应，即 γ 一定在 β 之前执行提交操作。

假设有两个日志项 γ 和 β 且 γ 在 β 之前提出，是否存在一个成员拥有 β 但不拥有 γ 的情况呢？答案同样是不会的。Follower 的一致性检查约束了 Follower 拥有 β，但其一定拥有 γ，因此我们不用担心存在空洞的日志项。

6.2.3　日志对齐

日志对齐通常发生在 Follower 追随 Leader 之初或者 Follower 落后于 Leader 数据时，Raft 约定集群中所有成员的日志项以 Leader 为基准，只能将 Leader 的日志项复制到其他成员中，其他成员不能以任何形式覆盖 Leader 的日志项。这极大简化了日志管理，Raft 易于理解也得益于此。

实际情况是，日志项管理不只是简单地追加。当一些 Follower 新加入集群或者 Leader 刚晋升之时，Leader 并不知道需要发送哪些日志给 Follower，同时，当一些旧 Leader 以 Follower 的身份加入集群时，往往会携带一些上一任 term 仅被提出的日志项，而当前 Leader 不存在这些日志项，此时 Leader 应该命令该 Follower 删除这些冲突的日志项。

因此，在进行日志对齐之前，第一步是要找到每个 Follower 与 Leader 之间的日志差异，Raft 引入了 nextIndex 变量，该变量代表当前 Follower 与 Leader 日志项相同的下一个索引。Leader 为每个 Follower 单独设置了 nextIndex，其初始值为 lastLogIndex+1（lastLogIndex 为本地的最后一个日志项的索引）。

为了探测 Follower 与 Leader 的日志差异，找到对应的 nextIndex，Leader 和 Follower 将进行以下交互。

（1）Leader 会以 AppendEntries 的形式发送探测消息，携带 preLogIndex、preLogTerm，其中，preLogIndex＝nextIndex−1。

（2）Follower 收到探测消息后，将在本地日志中索引为 preLogIndex 和 preLogTerm 的日志项，与消息中的 preLogIndex、preLogTerm 进行对比，然后将结果反馈给 Leader。

只需要比较 index 和 term，这是由日志匹配特性带来的便利。

（3）Leader 收到探测消息的响应后，如果 index 和 term 不一致，则说明 Follower 落后于 Leader，将递减 nextIndex 并发起一轮新的探测消息，直到找到相应的 nextIndex 为止。

找到对应的 nextIndex 后，Leader 从 nextIndex 开始按照日志顺序，以日志复制的形式将日志项发送给 Follower。Follower 收到日志复制的消息，应该以 Leader 的数据为准，覆盖本地的日志项。

通过上述方式，新加入的 Follower 无须 Leader 配合自己来完成日志对齐操作，Leader 通过 AppendEntries 消息就能使 Follower 的数据自动趋于一致。

nextIndex 采用单调递减的方式探测两者之间的差异，并不能达到满意的效率，通常在工程实现中，还会进行以下两点优化。

- Follower 在探测消息的响应中会携带本地最大的日志索引 lastLogIndex，Leader 在收到响应后，无须单调递减 nextIndex，而是将 nextIndex 赋值为响应中的 lastLogIndex+1。因此，通常在下一轮探测消息中就能找到正确的 nextIndex。

如果在下一轮探测消息中仍然未找到正确的 nextIndex，通常是该 Follower 曾经当选过 Leader，并且仅提出了一些提案，这些提案与当前 Leader 冲突了。这种情况下仍需要以当前 Leader 的数据为准，但是当前 Leader 只能单调递减地寻找 nextIndex 了。

- 快照同步。如果 Follower 落后的数据较多，在探测 nextIndex 时，Leader 可能已为该日志项生成了快照并已删除该日志项，此时 Leader 需要发送快照文件给 Follower，并且以快照的 logIndex+1 作为 nextIndex。关于快照技术，将在 6.5 节介绍。

经过以上两点优化，日志对齐通常不会占用太多时间，如果有需要的话，还可以以 term 为单位进行探测。但是 Raft 并不认为这种优化是有必要的，这类情况出现的可能性较小，并且由此引入对齐方式机制会增大实现难度，因此在 Raft 的标准实现中并没有实现这类方案。

6.2.4　幽灵日志

强制要求 Follower 采用 Leader 日志来实现日志对齐可能会出现“幽灵日志”，虽然这是最简单的对齐数据的方案，但是幽灵日志是不可接受的。

如表 6.2 至表 6.5 为不同时间点对于日志索引 1 的存储情况，它们展示了幽灵日志产生的过程。在表 6.2 中，Leader A 提出 Add 日志项，并且只在本地日志中持久化了 Add；在表 6.3 中，成员 A 宕机，成员 E 晋升为 Leader，提出 Mul 日志项，并且只在本地日志中持久化了 Mul；在表 6.4 中，成员 E 宕机，成员 A 重新当选 Leader，并以自己的日志为基准将 Add 复制至多数派成员中；在表 6.5 中，成员 A 宕机，成员 E 重新当选 Leader，并以自己的日志为基准在集群中复制 Mul。

表 6.2　幽灵日志 1

Log Index	Leader A	Follower B	Follower C	Follower D	Follower E
1	Add				

表 6.3　幽灵日志 2

Log Index	~~Leader A~~	Follower B	Follower C	Follower D	Leader E
1	~~Add~~				Mul

表 6.4　幽灵日志 3

Log Index	Leader A	Follower B	Follower C	Follower D	~~Leader E~~
1	Add	Add	Add		~~Mul~~

表 6.5　幽灵日志 4

Log Index	~~Leader A~~	Follower B	Follower C	Follower D	Leader E
1	~~Add~~	Mul	Mul	Mul	Mul

很明显，在处理上一任 Leader 提出的日志项时，多数派的思想已经不适用了。如果继续使用多数派思想处理上一任 Leader 提出的日志项，那么将给客户端造成歧义。以表 6.4 时间点为例，Leader A 在此时应该已提交了 Add，客户端可以查询到 Add。在表 6.5 时间点，根据选举约定，成员 E 在递增自己所处的 term 后，仍然可以晋升为 Leader，而成员 E 在晋升 Leader 后，在集群中复制 Mul，而此时客户端再来查询索引为 1 的日志项时，结果却变成了 Mul。

为了不让客户端在表 6.4 时间点的时候读到 Add，在表 6.5 时间点的时候读到 Mul 而产生歧义，最好的解决方式就是在一切完全确定之前，不让客户端读取索引为 1 的日志项。那么在表 6.4 时间点和表 6.5 时间点上，不能因为 Add 或 Mul 已经被复制到多数派成员中而提交。上一任 Leader 提出的日志项，只能在当前 Leader 中提出并且在完成下一个日志项的协商后才能提交。当前的 Leader 通过多数派的思想使某个日志项达成共识后，根据日志匹配特性，意味着存在一个多数派，在该日志项之前的所有日志项都与 Leader 相同且已提交，当然也包含上一任 Leader 提出的日志项。

如果一味地等待客户端发送事务请求，才能提交上一任 Leader 提出的日志项，这就太不及时了。因此在新 Leader 晋升之后，通常会以当前 term 提出一个 Noop 日志项，并在集群中进行协商，当 Noop 日志项获得多数派的写入后，Leader 就可以大胆提交上一任 Leader 提出的日志项了。

6.2.5　安全性

我们已对 Raft 整体流程进行了一些描述，在这些流程中，Raft 必须时刻保证以下安全要求：

- 选举安全性：一个任期（term）内只能选出一个 Leader。
- Leader 只追加日志项：不能以任何形式删除或者覆盖 Leader 的日志项，Leader 只会追加新日志项。
- 日志匹配：对于两个索引相同，term 相同的日志项，它们拥有相同的内容，且在该日志项之前所有的日志项也完全一致。
- Leader 完整性：如果一个日志项被提交了，那么该日志项必然存在于后续任期较高的 Leader 中。
- 状态机安全性：如果一个成员已经应用了一条日志项在状态机中，那么其他成员不会向状态机中应用相同索引下的不同日志项。

接下来我们主要论证 Leader 的完整性，选举安全性和日志匹配都可以基于多数派的思

想很快地被证明，在前面的章节中已有类似的例子。我们假设 Leader 的完整性是不满足的，然后尝试推导出矛盾，即可证明 Leader 的完整性。

在图 6.2 中有两个任期 T1 和 T2，且 T1 < T2。在任期 T1 中，成员 A 当选为 Leader 且提交了日志项 Add，在任期 T2 中，成员 E 当选为 Leader。假设成员 E 不存在日志项 Add，我们尝试推导有矛盾的地方。

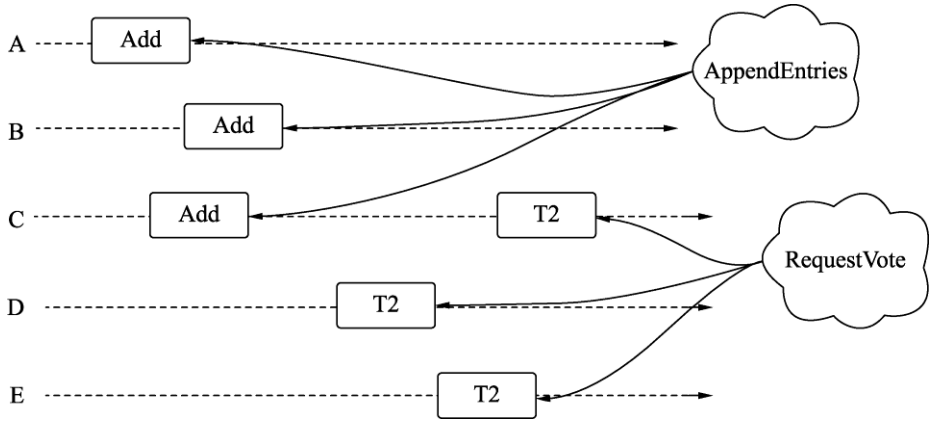

图 6.2　安全性论证

如果成员 E 不存在日志项 Add，那么一定是成员 E 在晋升为 Leader 之前就不存在该日志项，因为 Leader 不会覆盖自己的日志项。

在任期 T1 中，成员 A 一定是将 Add 复制给多数派成员了，而成员 E 在任期 T2 中晋升为 Leader，也一定获得了多数派成员的支持，那么两者之间一定存在一个重合的成员，即成员 C，成员 C 既存在日志项 Add，又把选票投给了成员 E。

成员 C 一定是先接受了成员 A 的 AppendEntries 消息，再接受成员 E 的 RequestVote 消息；否则成员 C 将拒绝成员 A 的 AppendEntries 消息，因为成员 E 的 RequestVote 消息会变更成员 C 的任期为 T2。

接着，成员 C 如果给成员 E 投票了，那么证明成员 E 的日志要么和成员 C 一样完整，要么比成员 C 更完整。两种情况都表明，成员 C 包含日志项 Add，那么成员 E 也一定包含日志项 Add，这便是与假设产生的矛盾。

因此，存在两个任期 T1 和 T2，且 T1＜T2，那么在 T1 中提交的日志一定存在于 T2 中。

6.2.6　Raft 小结

Raft 虽然通过强 Leader 模型、心跳机制和日志连续性，将协商过程优化为一个阶段，但是并不代表算法不包含其他阶段，只不过 Raft 利用一些手段将这些额外阶段变得没那么重要了。

一个日志项从开始协商到提交的过程，仍然可以分为三个阶段，这在 Multi Paxos 中可以找到类似的阶段，如图 6.3 所示。

- Leader 选举：类似 Multi Paxos 中的 Prepare 阶段，同样为一个 term（Multi Paxos

的提案编号）授予发起提案的权利。

- Raft 通过租约机制保证唯一稳定的 Leader，它一直“霸占”发起提案的权利。
- Multi Paxos 允许其他 Proposer 抢占发起提案的权利，由于提案冲突，Prepare 阶段的另一个作用是收集上一个 Proposer 可能发起的提案值。

- 日志复制：类似 Multi Paxos 中的 Accept 阶段。二者的唯一区别是提案值的选择不同。
  - Raft 不存在提案冲突，可以由 Leader 任意指定提案值。
  - Multi Paxos 需要根据 Prepare 阶段的结果选取提案值。
- 日志提交：类似 Multi Paxos 的 Confirm 阶段。
  - Raft 通过下一次协商消息或者心跳消息来完成这个过程。
  - Multi Paxos 需要引入一轮 Confirm 消息。

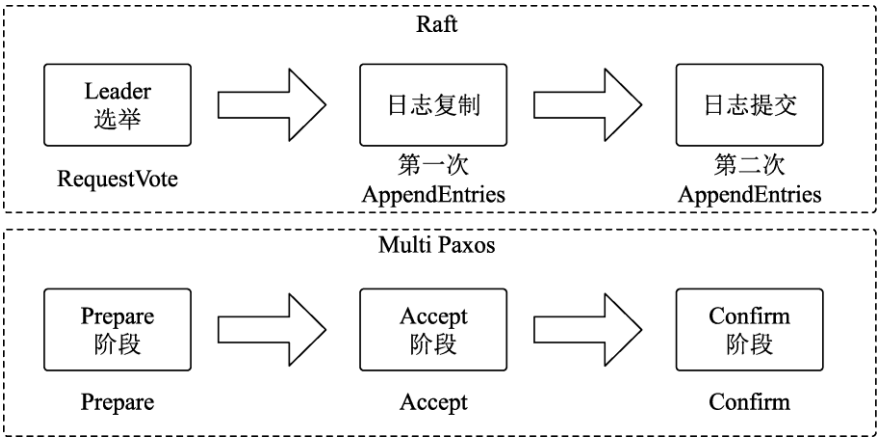

图 6.3　Raft 与 Paxos 对比

6.3　算 法 模 拟

6.3.1　Leader 选举

存在 A、B、C 三个成员组成的 Raft 集群，刚启动时，每个成员都处于 Follower 状态，其中，成员 A 心跳超时为 110ms，成员 B 心跳超时为 150ms，成员 C 心跳超时为 130ms，其他相关信息如图 6.4 所示。

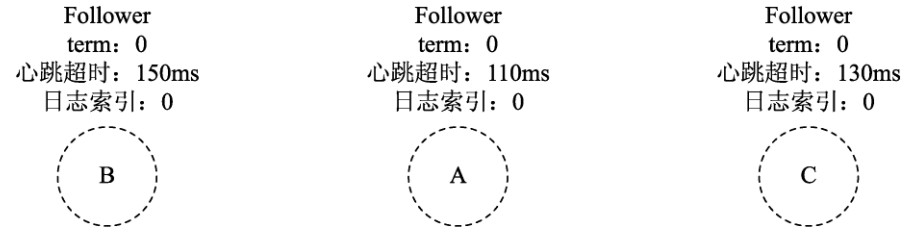

图 6.4　Raft 模拟初始状态

由于集群中不存在 Leader，A、B、C 三个成员都不会收到来自 Leader 的心跳信息。其中，成员 A 的超时最短，最先进入选举状态，修改自己的状态为 Candidate，并增加自己的任期编号为 1，发起请求投票消息，如图 6.5 所示。

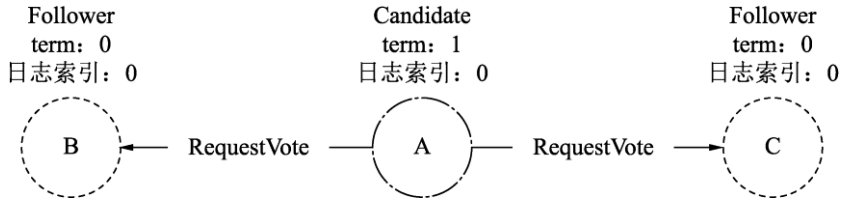

图 6.5　请求投票

成员 A 通过 RequestVote 广播自己的选票给成员 B、C，选票描述了成员 A 所拥有的数据，其包含成员 A 所处的 term 及最新的日志索引。成员 B、C 根据投票规则处理 RequestVote 消息。

- term 大的成员拒绝投票给 term 小的成员。
- 日志索引大的成员拒绝投票给日志索引小的成员。
- 一个 term 内只投出一张选票，采用先来先获得投票的原则。

很明显，成员 B、C 的 term 小于成员 A 的 term，也不存在比成员 A 日志索引更大的日志索引，并且 term 为 1 的选票还没有投给其他成员，因此成员 B、C 将 term 为 1 的选票投给成员 A 并更新自己的 term 为 1。

成员 A 获得包括自己在内的 3 张选票，赢得大多数选票，成员 A 晋升为 Leader，并向其他成员发送心跳信息，维护自己的领导地位，如图 6.6 所示。

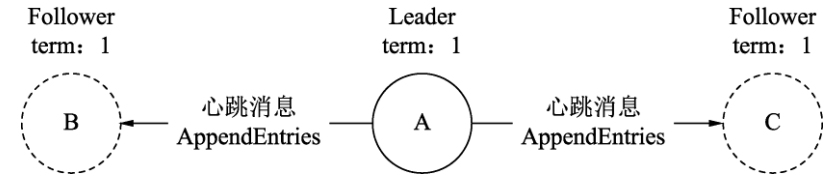

图 6.6　Leader 晋升示意

如果成员 A 在等待投票超过约定的时间内没有收到多数派的选票，则会重置自己的超时，并结束本次选举进程。接着会有其他成员在等待心跳超时后发起 Leader 选举，在当前案例中，发起 Leader 选举的顺序为 A→C→B。

可能因为网络问题，使集群中的所有成员又发起了一轮选举，但是都没有获得多数派的选票，因此会随机产生新的超时，开始下一个循环的选举。

6.3.2　日志复制

6.2.2 小节提过，日志复制是一个一阶段协商的过程，其中，日志项的提交操作由下一轮协商或者心跳消息来代替完成。因此处理事务请求，Raft 只需要发送一轮 AppendEntries 消息即可。

AppendEntries 消息除了会包含需要复制日志项的相关信息外，通常会携带 Leader 的 committedIndex 参数，标示着最后一个已提交的日志索引。每个 Follower 的本地都维护了 committedIndex，Follower 可以对比 Leader 的 committedIndex 来推进自己的提交操作。

接着图 6.6 所示的示例，一个三个成员组成的集群，成员 A 为 Leader，成员 B 和 C 为 Follower，并且在集群中未提交任何日志项。Leader 收到客户端发送的 Add 请求后，Leader 和 Follower 依次执行以下步骤，如图 6.7 所示。

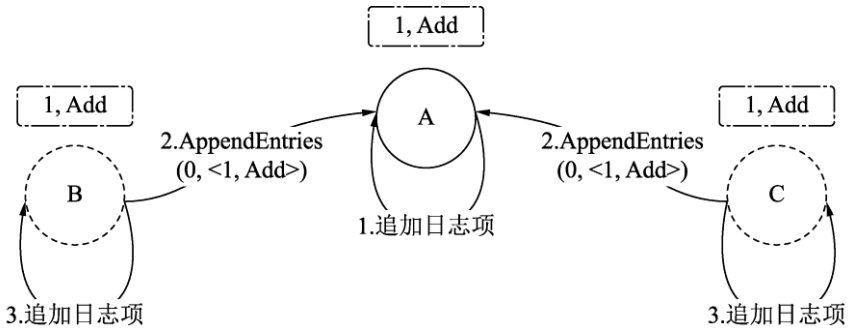

图 6.7　日志复制-复制

（1）Leader 将其封装成日志项追加到本地的日志中，日志索引为 1。

（2）Leader 通过 AppendEntries(0, <1, Add>)消息时将日志项广播给所有的 Follower。其中：

- 第一个参数为 committedIndex，即 Leader 最后提交的日志索引。
- 第二个参数为 Leader 所处的日志索引，即 Add 日志项的索引。
- 第三个参数为事务操作指令，即客户端的指令。

（3）Follower 收到消息，将日志项追加到本地的日志中。

此时，成员 A、B、C 都拥有日志项 Add 且都已在索引为 1 上完成了持久化。Follower 在处理完 AppendEntries 消息后需要回复 ACK 消息给 Leader，代表接受该日志项。Leader 收到多数派的 ACK 消息后，可以在本地提交该日志项并执行状态转移，之后将执行结果返回给客户端，如图 6.8 所示。

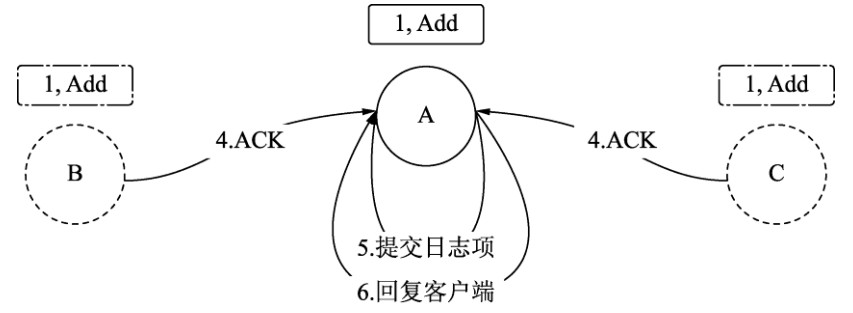

图 6.8　日志复制-回复

在当前场景中，成员 A 提交了索引为 1 的日志项，成员 B、C 仅仅拥有索引为 1 的日志项的所有信息但并未提交。成员 B、C 需要等待下一次 AppendEntries 消息，根据其

committedIndex 推进索引为 1 的日志项的提交操作。以心跳的 AppendEntries 消息为例，该 AppendEntries 消息仅携带了 committedIndex，此时 Leader 已经提交了索引为 1 的日志项，因此 committedIndex 为 1。Follower 则可以提交索引为 1 及其之前的所有日志项，如图 6.9 所示。

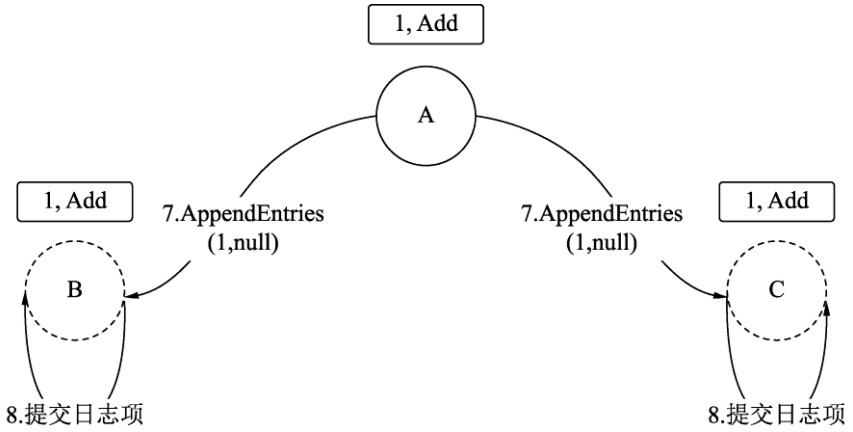

图 6.9　日志复制-心跳

6.3.3　日志对齐

我们使用<term, logIndex>表示一个日志项，如表 6.6 所示为 Follower E 的日志索引 3 和 Follower D 的日志索引 4，与当前 Leader 处理不一致的情况。出现这种情况可能是 Follower E 和 Follower D 曾经当选过 Leader，并且在自己的 term 上提出了日志索引为 3 和 4 的日志项后立即宕机造成的。

表 6.6　日志对齐

Log Index	1	2	3	4	5	6
Leader A	<1, 1>	<1, 2>	<2, 3>	<3, 4>	<3, 5>	<3, 6>
Follower B	<1, 1>	<1, 2>	<2, 3>	<3, 4>	<3, 5>	
Follower C	<1, 1>	<1, 2>	<2, 3>	<3, 4>	<3, 5>	
Follower D	<1, 1>	<1, 2>	<2, 3>	<2, 4>		
Follower E	<1, 1>	<1, 2>	<1, 3>			

要使 Follower E 和 Follower D 与 Leader 数据保持一致，大致步骤分为两步：寻找 nextIndex，复制 nextIndex 及其之后的日志项。在 Raft 中，这个步骤均可由 AppendEntries 消息来完成。这里以 Follower E 成员为例，交互细节如下：

（1）Leader 为 Follower E 初始化 nextIndex，nextIndex=lastLogIndex+1，即 nextIndex=6+1=7。

（2）Leader 通过 AppendEntries 发送探测消息，携带 preLogIndex（nextIndex−1）及 preLogTerm，其中，preLogIndex=6，preLogTerm=3。

（3）Follower 收到探测消息，对比索引为 6 的日志项，返回失败的响应给 Leader 并携

带 lastLogIndex=3。

（4）Leader 收到失败的响应，更新 nextIndex=$lastLogIndex_{msg}$+1，即 nextIndex=4。

（5）Leader 发送下一轮的探测消息，其中，preLogIndex=3，preLogTerm=2。

（6）Follower 收到探测消息，对比索引为 3 的日志项，返回失败的响应给 Leader 并携带 lastLogIndex=3。

（7）Leader 收到失败的响应，此时 $lastLogIndex_{msg}+1 \leqslant nextIndex$，则 nextIndex 单调递减为 3。

（8）Leader 发送下一轮的探测消息，其中，preLogIndex=2，preLogTerm=1。

（9）Follower 收到探测消息，对比索引为 2 的日志项，返回探测成功的响应给 Leader。

（10）Leader 在成功探测到 nextIndex 之后，通过 AppendEntries 消息从 nextIndex 开始发送索引为 3 的日志项给 Follower。

（11）Follower 将以 Leader 的数据为准，覆盖本地的日志项并返回处理成功的响应给 Leader。

（12）Leader 收到成功响应后，单调递增 nextIndex，继续发送下一个日志项。直到 nextIndex 等于 Leader 的 lastLogIndex，意味着该 Follower 拥有 Leader 所有的数据，本次日志对齐即完成。

6.4 成 员 变 更

Raft 之所以受欢迎的一个重要因素是，它是面向生产而设计的，切实地解决了行业内的痛点。Raft 并非只关注其算法的协商过程，对于成员变更也给出了规范的实现方法，而这正是应用于生产所必需的。成员变更这一规范后来也被应用于其他共识算法中。

Raft 的 Leader 选举和事务协商都源于多数派思想，而多数派是相对于一个固定集合来说的，只有在固定集合中，多数派的数量才是恒定的。但是在实际场景中，我们会遇到很多情况需要变更集群成员。例如，替换掉那些发生故障的成员，或者增加成员数量，允许更多的成员发生故障。

集群成员的变更，自然导致多数派的数量也会随之变化。如果处理不当，可能会导致两个“多数派”（变更前的多数派和变更后的多数派）之间不存在相交的成员，这样就会产生两个 Leader 在各自认为的“多数派”中工作的现象。因为这两个“多数派”不存在相交的成员，所以有可能在一个日志索引上会提交两个不同的日志项，从而影响 Raft 的安全性。

例如，存在一个由三个成员组成的集群，集群成员为 C_{old}={A, B, C}，现在需要上线两个新成员，变更后的集群成员 C_{new}={A, B, C, D, E}。在变更过程中，由于不能保证 C_{new} 在五个成员中同时生效，便会出现如图 6.10 所示的场景。在 T1 时刻，成员 C、D、E 拥有 C_{new} 的配置，成员 A、B 继续使用 C_{old} 的配置，此时可以形成两个多数派 Q_{old} 和 Q_{new} 满足：

$$Q_{old}=\{A,B\}:Q_{old}\in C_{old}$$

$$Q_{new}=\{C,D,E\}:Q_{new}\in C_{new}$$

两个多数派自然也可以选举出两个 Leader，这两个 Leader 都自认为能正常工作且都能

正常完成事务协商。

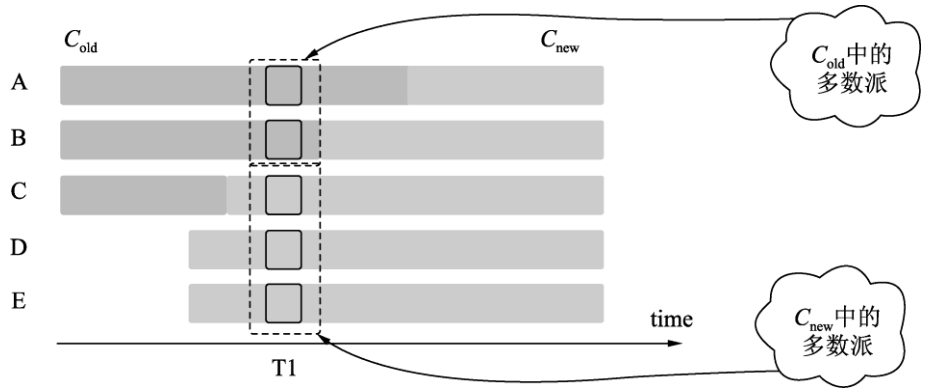

图 6.10　成员变更

通过引入一个物理配置来管理所有的成员信息，这是我们最容易想到的方案。为了让所有成员在同一时刻都能获取更新后的集群配置，可以选择集群停机后再更新配置。不否认这种做法是易于理解且有用的，但有很大的局限性。先不说集群停机时服务将不可用这个问题，将所有成员优雅关机就是一个大工程。

显然，通过关机并更新集群配置来完成成员变更是不能接受的，但问题还得解决。我们知道，在分布式环境中，让所有成员都认可某个配置，其本质就是共识问题。因此，解决该问题的方案只能是共识算法，因此可以考虑在 Raft 的基础上增加额外的机制进行变更成员，但是 Raft 的运行又依赖于固定的集合，而变更成员就是为了获得一个暂时固定的集合，这就变成一个“先有鸡还是先有蛋”的问题。因此，不可能通过一次变更就能以原子方式更新所有成员的配置，在变更过程中，应该需要一个中间集合来接替“多数派”的工作。

6.4.1　联合共识

为了规范 Raft 成员变更的方案，Diego Ongaro 博士在论文中提出了联合共识（Joint-Consensus）的概念。联合共识是指引入一个临时的“多数派”来保证算法的安全性，其被认为是基于多数派思想实现的共识算法的成员变更通用协议，被应用于某些 Paxos 的实现中。

因为“鸡”和“蛋”之间的循环，一次性自动变更所有成员配置是不可能的，需要通过两个阶段来完成。通过两个阶段切换成员配置有很多方法。例如，可以在第一阶段禁用旧的成员配置，这样便不会再接收客户端请求，接着在第二阶段启用新的成员配置，除此之外，还可以使用前面介绍的停机更新成员配置的方法。这些方法简单、有效且都能保证算法的安全性，但 Raft 还是做出了其他选择。对于共识算法，除了在变更过程中保证算法的安全性之外，还希望其能实现以下基本要求：

- 任何时候（包括成员变更时），集群都能提供完整的服务。
- 无论何种情况宕机重启后，集群都能正常选举 Leader 并正常运行。

为了在任何时候都能提供完整的服务，引入了一个中间状态，称为联合共识[①]。处于联合共识的集群允许处理客户端的事务请求，但是需要通过比多数派更加严格的决策。

1．变更过程

为了方便描述，我们使用 C_{old} 表示旧的集群配置（旧的成员集合），Q_{old} 表示旧的成员集合中的多数派，C_{new} 表示新的集群配置（新的成员集合），Q_{new} 表示新的成员集合中的多数派，$C_{old,new}$ 表示联合共识时的集群配置（旧的成员集合+新的成员集合）。

成员变更的过程由 Leader 发起，由 Leader 主导成员变更的过程。当 Leader 收到将 C_{old} 变更至 C_{new} 的请求后，分两阶段完成成员变更。

（1）第一阶段，将 $C_{old,new}$ 的配置发送给 $C_{old} \cup C_{new}$ 的成员，获得多数派 $Q_{old} \cup Q_{new}$ 的支持即执行第二阶段。

（2）第二阶段，将 C_{new} 的配置发送给 $C_{old} \cup C_{new}$ 的所有成员，但只需要获得 Q_{new} 的支持则完成成员变更。

在成员变更过程中，一旦完成第一阶段，即处于联合共识的状态，对于任何操作，C_{old} 和 C_{new} 都不能单方面地做出决定，这是为了保证算法的安全性。例如当提交一个日志项时，需要同时获得 Q_{old} 和 Q_{new} 的支持；在第二阶段的工作完成后，新的成员配置立即生效，后续的提案只需要获得 Q_{new} 的支持即可。

为了保证变更过程中的服务可用，在联合共识期间处理事务请求，需要保证：

- 将日志项分别发送至拥有 C_{old} 和 $C_{old,new}$ 配置的所有成员。
- 当提交一个日志项时，要求得到 Q_{old} 和 Q_{new} 两个多数派的支持。
- 拥有任意一种配置的成员都可以晋升为 Leader。

2．协议模拟

有一个 C_{old}＝{A, B, C}的集群，需要变更为 C_{new}＝{C, D, E}，如图 6.11 所示。

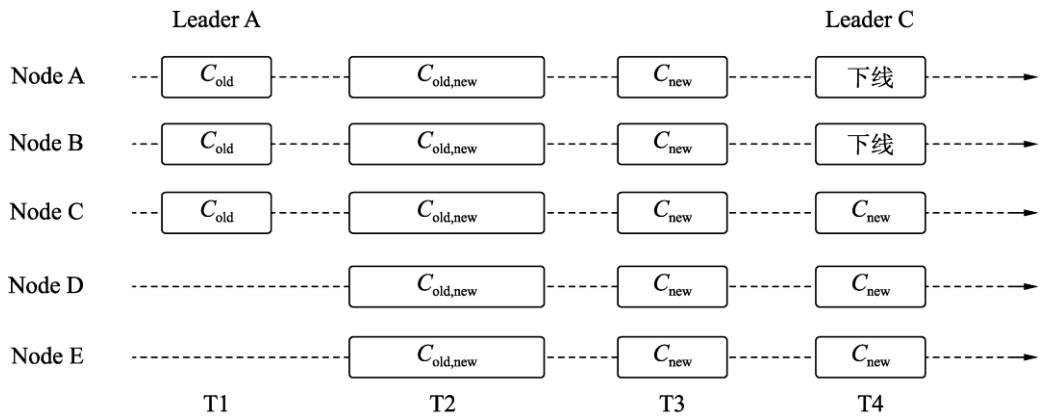

图 6.11　联合共识示意

① 联合共识是指旧的成员集合和新的成员集合联合起来共同批准提案。

（1）在 T1 时刻，Leader 为成员 A，在收到成员变更的请求后，将 $C_{old,new}$ 配置发送给 $C_{old} \cup C_{new}$ 的成员。

（2）在 T2 时刻，Leader A 完成了第一阶段的工作，所有成员都批准了 $C_{old,new}$ 配置，Leader A 开始处理第二阶段的工作。

（3）在 T3 时刻，Leader A 完成了第二阶段的工作，所有成员都批准了 C_{new} 配置。

（4）在 T4 时刻，成员 A、B 发现新配置中不包含自己，因此选择主动下线。在没有 Leader 的情况下会重新选出 Leader。

3．异常恢复

虽然联合共识能在保证安全性的情况下完成成员的变更，但这只满足了第一个基本要求，还有另一个基本要求需要考虑，即无论何种情况，宕机重启后，集群都能正常选举 Leader 并正常运行。

这里只讨论在成员变更过程中出现的故障问题，成员变更前后出现的故障问题前面已经讨论过了，只需要重新进行 Leader 选举即可。如果集群在联合共识的状态下 Leader 崩溃了，成员变更过程应该如何继续呢？

第一步是选举 Leader，因为成员变更需要 Leader 的驱动。Leader 崩溃时，集群中会存在三类配置成员。第一阶段会出现拥有 $C_{old,new}$ 配置的成员；第二阶段会出现拥有 C_{new} 配置的成员，至成员变更结束之前，都可能存在拥有 C_{old} 配置的成员。

- 拥有 C_{old} 配置的成员晋升 Leader 时，需要获得 Q_{old} 的支持。
- 拥有 C_{new} 配置的成员晋升 Leader 时，需要获得 Q_{new} 的支持。
- 拥有 $C_{old,new}$ 配置的成员晋升 Leader 时，需要获得 $Q_{old} \cup Q_{new}$ 的支持。

如果拥有 C_{old} 配置的成员晋升为 Leader 后并不知道当前是否处于成员变更的过程中，那么本次成员变更只能以失败告终，但是这并不影响集群以 C_{old} 的配置方式运行。

如果拥有 C_{new} 配置的成员晋升为 Leader，则意味着成员变更已经完成了第一阶段的工作并获得了 $Q_{old} \cup Q_{new}$ 的支持，且第二阶段的工作已经完成或者正在进行中。无论第二阶段处于哪个状态，第二阶段的工作都是将 C_{new} 复制到所有成员中且争取 Q_{new} 的支持；而 C_{new} 成员晋升成 Leader 也需要 Q_{new} 的支持，并且在晋升后 Leader 会将 C_{new} 复制到所有成员中，即间接地完成了成员变更第二阶段的工作。

如果拥有 $C_{old,new}$ 配置的成员晋升为 Leader，其一定是获得了 $Q_{old} \cup Q_{new}$ 的支持，这代表成员变更已经完成了第一阶段的工作，第二阶段的工作有可能尚未进行，或者正在进行中，也或者已完成。无论第二阶段处于哪个状态，Leader 都可以大胆地再完成一次第二阶段的协商，这并不会影响集群的运行。

4．脑裂

在 Raft 中，我们推荐以日志项的形式存储成员配置，每个成员只需要使用日志最后一个成员配置即可。

对于 $C_{old,new}$ 和 C_{new} 达成共识的方式，可以以日志复制的方式进行协商，不用再引入其他机制。同时，以日志项的形式存储成员配置，可以有效地限制变更过程中切换 Leader 的

成员范围。

对于其他算法使用联合共识的情况，如果不能以日志项/提案的方式来实现，则需要着重关注在变更过程中 Leader 切换的安全性问题，下面介绍这一点的重要性。

在成员变更过程中，如果在第二阶段的工作未完成的情况下 Leader 宕机，则此时进行的 Leader 选举可能会出现两个 Leader 且拥有各自的“多数派”。如表 6.7 所示，一个 C_{old} ={A, B, C}的集群需要变更成 C_{new}={C, D, E}的集群，成员 C 为 C_{old} 的 Leader。在第一阶段的工作完成后，成员 A 拥有的配置为 C_{old}，成员 B、C、D、E 拥有的配置为 $C_{old,new}$，在第二阶段只有成员 C 复制了 C_{new}。此时，如果 Leader 宕机，则成员 A 开始竞选 Leader 并获得 Q_{old}={A, B}的支持，成员 C 开始竞选 Leader 并获得 Q_{new}={C, D, E}的支持。

表 6.7　崩溃重启后的集群配置情况

	Node A	Node B	Node C	Node D	Node E
第一阶段	C_{old}	$C_{old,new}$	$C_{old,new}$	$C_{old,new}$	$C_{old,new}$
第二阶段	C_{old}	$C_{old,new}$	C_{new}	$C_{old,new}$	$C_{old,new}$

要解决拥有 C_{old} 配置的成员和 C_{new} 配置的成员同时晋升为 Leader 的问题，需要限制拥有 $C_{old,new}$ 配置的成员的投票规则。为此，需要为成员配置引入 Version 属性，并要求 Version 值高的成员不能投票给 Version 值低的成员。在这个约束条件下，拥有 C_{old} 配置的成员是获取不到多数派的支持的，只有拥有 $C_{old,new}$ 或者 C_{new} 配置的成员才有可能晋升 Leader，而拥有 $C_{old,new}$ 配置的成员晋升 Leader 也需要获取 Q_{new} 的支持，自然不可能同时出现拥有 $C_{old,new}$ 配置的 Leader 和拥有 C_{new} 配置的 Leader。

6.4.2　工程实践

联合共识通过两个阶段来实现成员变更，理论上并不复杂，但是真正用于生产中的话并没有那么简单。因为成员变更会改变多数派集合，而 Raft 本身就依赖于多数派集合，所以在变更的临界点上会面临一些问题。

1．新成员可能没有存储任何日志项

当一个全新成员加入集群时，它自身没有参与之前日志项的协商，由此会出现两个问题。

- 当全新成员加入集群时，无法决策 Leader 提出新日志项，如果将它作为多数派的关键一员，则会阻断 Raft 的协商过程，直到它赶上 Leader 的日志。
- 在旧成员全部下线后，已提交的提案可能再也获取不到。

第一个问题很好理解，一个数据落后的成员是不能跳过落后的日志项来处理新日志项的 AppendEntries 消息的，因此它可能导致集群在一段时间内的协商获取不到多数派的支持。

第二个问题以一个案例来描述，如表 6.8 所示，索引 7 在成员 A 和 B 上提交了 Add 日志项，接下来需要将成员配置由 C_{old}={A, B, C}变更成 C_{new}={C, D, E}。经过成员变更，

C_{new} 生效后，在集群中就没有成员拥有索引为 7 的日志项了。

表 6.8　旧成员下线

	C_{old}={A, B, C}	$C_{old,new}$	C_{new}={C, D, E}
Log Index	7	8	9
Node A	Add	$C_{old,new}$	C_{new}，下线
Node B	Add	$C_{old,new}$	C_{new}，下线
Node C		$C_{old,new}$	C_{new}
Node D		$C_{old,new}$	C_{new}
Node E		$C_{old,new}$	C_{new}

为了避免这类情况产生的不良影响，在成员变更之前，需要为 Raft 加入一个额外的阶段，让这些新成员以学习的身份加入集群，但是不参与日志项的决策，以便让它们尽快赶上 Leader 中的日志。需要注意的是，Leader 在统计多数派时不应该将它们纳入在内。

2．成员配置在什么时候生效

成员配置日志与普通日志不同，并不能约定要在提交之后再生效。成员变更运行在新旧配置交替的临界点上，因此我们必须明确每个成员对于新旧成员的配置何时生效，这样才能保证算法能正常推进。

一个日志项的协商过程，可以分为三个关键点：收到日志项、持久化日志及提交日志项。成员配置生效也必定在这三个时间点中。

对于 Leader 来说，第一阶段完成的标志是获得 $Q_{old} \cup Q_{new}$ 的支持，这要求 Leader 此时不能使用 C_{old} 了，因为 C_{old} 只需要 Q_{old} 的支持。因此，Leader 一定要在广播 $C_{old,new}$ 之前就应用了 $C_{old,new}$，而持久化日志项与广播日志项通常是并行的，我们约定 Leader 在收到成员变更请求时，就应该应用 $C_{old,new}$。第二阶段类似，Leader 应该在第二阶段开始时就应用 C_{new}。

对于 Follower 来说，它不需要统计多数派，因此，应用成员配置的临界点就没有那么敏感，只需要保证 Follower 能处理 Leader 的消息即可。在变更过程中，Leader 可能存在于 C_{old} 中，也可能存在于 C_{new} 中（旧 Leader 宕机，C_{new} 中的成员晋升 Leader，继续完成成员变更），但是 Follower 都要继续处理成员变更操作。因此约定：第一阶段，Follower 在收到 $C_{old,new}$ 后应用 $C_{old,new}$；第二阶段，Follower 在提交 C_{new} 后应用 C_{new}（保证仅包含 C_{old} 的 Leader 能继续完成它的工作）。

我们约定，任何成员在提交 C_{new} 时，如果发现自己不在成员配置中，就应该主动关闭自己，Leader 也不例外。这意味着在 Leader 提交 C_{new} 之前，它管理着一个不包含自己的集群，因此在协商日志项时，它不应该将自己纳入多数派的统计范围之内。

之所以在提交 C_{new} 之后而不是在应用 C_{new} 时关闭，主要原因有两个：

- 第一个原因是，只有已提交的日志项才不会改变，Leader 需要继续完成在此之前已提出且尚未完成的日志项的协商，而提交 C_{new} 就意味着在此之前所有日志项的协商都已完成。
- 第二个原因是，只有在此刻才能保证拥有 C_{new} 的成员晋升 Leader，在此之前，可

能由拥有 $C_{old,new}$ 配置的 C_{old} 成员晋升为 Leader，新晋升的 Leader 会在应用 C_{new} 时关闭自己，因此这是一个无效的 Leader 选举。

3．Leader如何切换

在实际场景中，我们应该允许 Leader 主动将自己的领导地位转移给其他成员，通常这类需求出现于下面两种情况：

- Leader 必须下台。例如，在成员变更中，Leader 不存在于 C_{new} 的配置中，Leader 需要定期维护或者重启。
- 其他成员可能更适合当选 Leader。例如，与客户端相邻的成员具有较低的延迟，某些成员的处理性能更好等。

虽然上面这些情况使得 Leader 强制关机，Raft 也能安全地选出下一任 Leader，但是在这个过程中，集群将闲置一个选举超时，直到下一任 Leader 晋升。这种短暂的不可用是可以优化的，可以将 Leader 的领导地位转移至其他成员进行优化。

切换 Leader，只需要通知其目标成员在不等待选举超时的情况下进入 Candidate 状态并发起选举。为了使目标成员能成功晋升 Leader，需要保证目标成员拥有当前任期内的所有日志项，而这个条件，当前 Leader 可以通过 AppendEntries 消息很快地确定，所以切换 Leader，仍需要当前 Leader 主动发起，具体步骤如下：

（1）当前 Leader 停止接收新的客户端请求，并通过 AppendEntries 消息将自己的日志项完全复制给目标成员。

（2）当前 Leader 发送 TimeoutNow 消息给目标 Leader，这个消息和目标成员选举超时具有同样的效果，将会引发目标成员发起一个正常的 Leader 选举。

（3）目标成员收到 TimeoutNow 消息后进入 Candidate 状态，递增 term 并发起选举。

（4）目标成员成功晋升 Leader 后，将发送心跳给前任 Leader，前任 Leader 就可以安全下线了，Leader 切换完成。

如果本轮选举没有在一个选举超过约定的时间内完成（可能是延迟或者目标成员未拥有完整的日志等原因），那么当前 Leader 需要尝试终止切换并重新接收客户端请求。如果只是延迟原因，目标 Leader 可能正在正常地运行，那么最坏的情况就是当前 Leader 在等到一个心跳超时后主动下线，安全性也不会受到影响。

4．被移除的成员会干扰集群

在第二阶段的工作完成后，Leader 不会给那些被移除的成员发送 Leader 心跳消息，因此它们会发起新的选举，这样会导致当前 Leader 下台。虽然这些被移除的成员不会晋升为 Leader，但是它们会一直发起选举，反复重试。

为了防止这种情况发生，当成员认为当前的 Leader 存活时，会忽略 RequestVote 消息。具体说，如果成员从当前 Leader 接收的 AppendEntries 消息在最小选举时间内，则不会更新其任期号或者不会为 RequestVote 消息投票，这不会影响正常选举，因为在最小选举时间内，Leader 有很大可能正常工作，拒绝这些 RequestVote 消息。这有助于避免被移除的成员造成的服务中断：如果一个 Leader 能够将心跳发送到集群，那么它就不会被更大的任

期号所取代。

5．只有少数派成员存活时，如何恢复集群

只有少数派成员存活时，Raft 将失去安全性，这是一种 Raft 的异常情况。此时，为了保证集群继续运行，系统管理员就要在容忍数据丢失的前提下容许集群继续运行。这种情况，成员变更的目的是组建可用的多数派集合，但是成员变更本身又依赖于多数派集合。为了打破这个循环，通常需要提供一个强制更新成员配置的接口。

例如，有一个由 3 个成员组成的集群，当其中 2 个成员都出现故障时，那么只有 1 个成员存活的集群则无法进行成员变更。这里可以使用强制更新成员配置的接口将集群成员数量强制修改为 1 个，则唯一存活的这个成员就能获得多数派，然后再进行后续的成员变更即可。

虽然强制修改成员配置能尽快恢复服务的可用性，但是无法保证数据安全性。因为我们无法得知当前存活的成员是否拥有最新且完整的日志，也无法得知那些异常成员的日志执行情况。因此，当集群出现这种状况时，建议优先恢复旧的多数派集合，然后再进行成员变更。

6.4.3　单个成员变更

联合共识依赖两个阶段来实现成员变更算法，在工程实现中却有难度。于是有研究者发现了一个更加简单的方案，即不允许一次变更多个成员，而是一次只能从集群中添加或删除一个成员，称为单个成员变更（Single-Server Changes）。在这个限制条件下，我们可以使用 Raft Log 的形式，在日志协商阶段使新的成员配置在集群中得以应用，无须引入额外的机制。

1．单个成员变更简介

联合共识之所以分为两个阶段，是因为对需要变更的成员没有过多的要求，为了避免在变更过程中拥有 C_{old} 和 C_{new} 配置的成员形成没有交集的多数派，因此引入了中间状态。要求这个中间状态的决策条件是得到 Q_{old} 和 Q_{new} 的支持，这是比“多数派”更强的约束，变相地引入了一个相交的实体（中间状态），其目的就是为了保证两个决策集合一定存在相交的成员，进而保证算法的正确性。

联合共识对需要变更的成员没有过多的要求，甚至在保证集群可用的情况下，允许替换所有成员，但是在实际场景中并不经常有变更大量成员的需求，并且两阶段的逻辑在工程实现上比较困难。后来发现了一个更简单的方法，即不允许因为成员变更而产生不同的多数派。因此，Raft 限制了变更条件：一次只能从集群中添加或删除一个成员。这样便能保证 Q_{old} 和 Q_{new} 之间一定存在相交的成员，如图 6.12 所示。

这种重叠阻止了集群分区，保证一定存在重叠的成员，因此当只添加或删除单个成员时，直接切换到新的配置是安全的。利用这一点，不需要复杂的联合共识即可安全地更改集群成员身份。

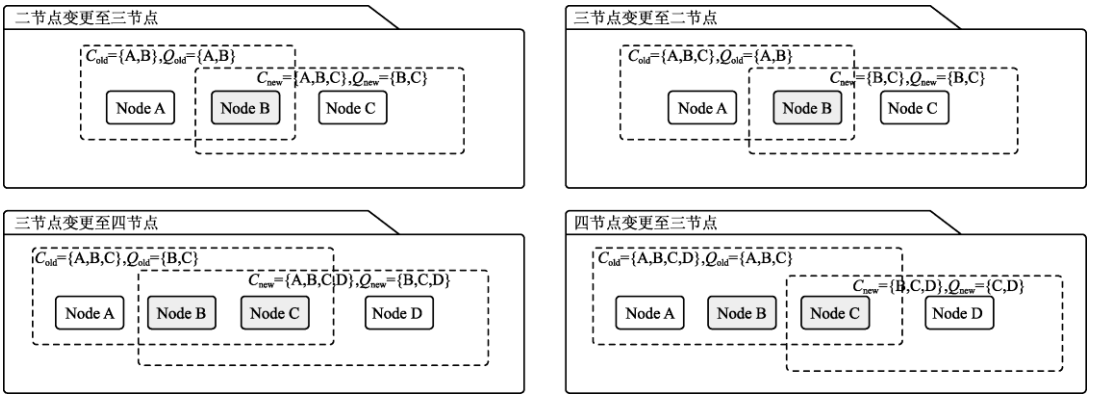

图 6.12　单成员变更

2．安全性

为了证明单个成员变更的安全性，假设当前成员数量为 N，那么多数派 Q_{old} 的数量为 $\lfloor N/2 \rfloor+1$，上线一个成员后，成员数量为 $N+1$，多数派 Q_{new} 的数量为 $\lfloor (N+1)/2 \rfloor+1$。

- 如果 N 为偶数，那么 $\lfloor N/2 \rfloor$ 可以整除，结果就是商；$\lfloor (N+1)/2 \rfloor$ 不能整除，模等于前者的商，余数为 1，得出 $\lfloor N/2 \rfloor+1=\lfloor (N+1)/2 \rfloor+1$。
- 如果 N 为奇数，根据同样的推论，也可以得出 $\lfloor N/2 \rfloor+1+1=\lfloor (N+1)/2 \rfloor+1$。

偶数情况，Q_{old} 即为 Q_{new}，两个 Q_{new} 一定相交，那么 Q_{old} 与 Q_{new} 也一定相交，即 $|Q_{\text{old}}|+|Q_{\text{new}}|>N$。

奇数情况，也可以表示为 $|Q_{\text{old}}|+1=|Q_{\text{new}}|$，两个 Q_{new} 一定相交，那么数量为 $|Q_{\text{old}}|+1$ 的集合与 Q_{new} 也一定相交，于是有：

$$|Q_{\text{old}}|+1+|Q_{\text{new}}|>N+1 \Rightarrow |Q_{\text{old}}|+|Q_{\text{new}}|>N$$

这又回到了偶数情况，因此得出结论：在一次只变更一个成员的前提下，Q_{old} 与 Q_{new} 一定相交。

这里展示了如果变更前后的两种配置只相差一个成员，那么两种配置的多数派至少有一个成员重叠的情况。因此，Leader 可以轻松地以日志复制的方式将新配置同步给其他成员且安全性不受影响。

3．注意要点

Raft 以日志协商的形式实现成员变更，在变更过程中需要明确以下要点：

- 在一个成员变更日志项提交之后，才可以进行下一次成员变更。
- Leader 需要明确感知到成员变更已经完成。
- 本次变更是移除成员，因此在日志项提交之后，该成员应主动关机。

虽然单个成员可以使 Q_{old} 和 Q_{new} 相交，但是两个相互重叠的成员变更操作仍然会导致变更过程中出现两个多数派。例如，集群需要增加两个新成员，需要执行两次成员变更，如果在第一次成员变更未提交之前进行第二次成员变更，那么第二次成员变更即相当于在旧配置的基础上一次变更增加了两个成员，这就破坏了我们最初对单个成员变更的设想，

因此安全性也会失效。

要满足第一个要点，Leader 需要明确感知成员变更是否已经完成了，这要求 Leader 应收集所有 Follower 在第二阶段的执行情况，并确保它们都已完成数据落盘，这会让 Leader 花费较长的时间。因此，我们约定 Raft 总是以最后一条成员变更日志项为当前的成员配置，这意味着成员变更日志项在持久化后即会生效，而不用等到日志项提交的那一刻。不幸的是，这个设计会导致：在切换 Leader 后，成员变更日志项可能会被丢弃，程序需要回滚到前一个成员配置阶段。

4．变更中的日志覆盖

在一些极端情况下，成员变更方案曾引发过比较激烈的关于安全性的讨论。在成员变更过程中，如果频繁切换 Leader，可能会丢失已提交的日志。Diego Ongaro 曾公开讨论过这个 Bug[①]，但这个 Bug 仅限于单个成员变更，联合共识不受影响。

Diego Ongaro 在论坛中提出了 3 个反例来重现这个 Bug，其本质都是新成员配置被提交后，旧成员配置仍有可能被提交，这里以向集群中增加一个成员同时删除另一个成员为例进行说明。如图 6.13 所示，存在一个集群，其初始配置为 C_1={S1, S2, S3, S4}，先后收到两个成员变更请求 C_2 和 C_3，C_2 为集群增加成员 S5，C_3 为集群删除成员 S1。具体配置是 C_2={S1, S2, S3, S4, S5}，C_3={ S2, S3, S4}。

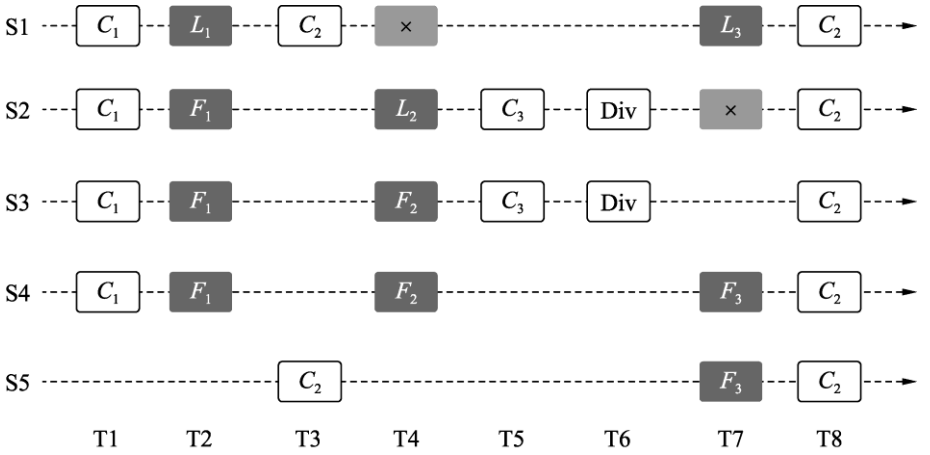

图 6.13　单成员变更-日志丢失

- T2 时刻：S1 在任期 term 为 1 时晋升为 Leader。
- T3 时刻：收到成员变更请求 C_2，并将其复制到 S1 和 S5 中。
- T4 时刻：S1 宕机，S2 在任期 term 为 2 时晋升为 Leader。
- T5 时刻：收到成员变更请求 C_3，并将其复制到 S2 和 S3 中。
- T6 时刻：收到事务请求 Div，并将其复制到 S2 和 S3 中，然后在获得多数派的支持后提交该日志项。

① https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E。

- T7 时刻：S2 宕机，S1 在任期 term 为 3 时晋升为 Leader，其多数派为{S1, S4, S5}。
- T8 时刻：S1 将 C_2 复制到所有成员中，但是 S1 没有拥有 Div 的日志项，因此 Div 最终会被抛弃。

笔者给这种情况取了一个形象的名字为穿越时间的脑裂，即两个不同时间的 Leader 拥有互不重叠的两个多数派。出现这种情况的原因是，Leader 一上任就开始执行成员变更，并使用新的成员配置协商日志项。

解决这个问题的方案是 Leader 上任之后，必须在当前任期中提交一条日志项，然后才能执行成员变更。这一动作的目的是使下一个晋升 Leader 的多数派与该日志项的多数派存在一个重叠的成员，这样就有效限制了后续 Leader 晋升的成员范围。

接着图 6.13 的示例，L_2 在晋升 Leader 之后，协商并提交一条 Noop 的日志项，其{S2, S3, S4}包含该日志项，那么在 term 为 3 的任期中，S1 不能再晋升 Leader，能晋升 Leader 的成员只能是 S3，最终 Div 日志项才得以延续保存。

为什么只能是 S3 晋升 Leader 呢？具体情况如下：

- S1 配置为 C_2＝{S1, S2, S3, S4, S5}，多数派数量为 3，会给它投票的成员为{S1, S5}。
- S2 宕机。
- S3 配置为 C_3＝{ S2, S3, S4}，多数派数量为 2，会给它投票的成员为{S3, S4}。
- S4 配置为 C_1＝{S1, S2, S3, S4}，多数派数量为 3，会给它投票的成员为{S1, S4}。
- S5 配置为 C_2＝{S1, S2, S3, S4, S5}，多数派数量为 3，会给它投票的成员为{S1, S5}。