分布式共识计算

分布式计算

分布式三大框架

MapReduce

MapReduce 的核心思想可以总结为以下几个关键点：

1. **分而治之（Divide and Conquer）**：
   - MapReduce 采用了“分而治之”的策略来处理大规模数据集。这意味着将一个复杂的、数据量庞大的问题分割成许多较小的、独立的子问题，每个子问题都可以在单独的计算节点上并行处理。这种分割降低了单个任务的复杂度，使得问题更易于管理和解决。
2. **两阶段处理（Two-Phase Processing）**：
   - MapReduce 操作分为两个明确的阶段：Map 阶段和 Reduce 阶段。
     - **Map 阶段**（映射）：在这个阶段，输入数据被分割成多个逻辑数据块（通常与文件系统的块大小相对应），然后由多个 Map 任务并行处理。每个 Map 任务接收一个数据块作为输入，对其中的每一条记录执行用户定义的 Map 函数。Map 函数将每条记录解析，提取出关键信息（键/值对），并生成新的中间键值对。这些中间结果通常是局部的、临时的，并且按照键进行排序和分区，准备传递给 Reduce 阶段。
     - **Reduce 阶段**（归约）：Reduce 任务接收 Map 阶段产生的中间键值对，这些键值对根据键的哈希值已经进行了适当的划分和排序。每个 Reduce 任务负责处理某一范围内的键值对（所有具有相同键的中间结果会被发送到同一个 Reduce 任务）。Reduce 任务通过用户定义的 Reduce 函数，对具有相同键的所有值进行聚合或进一步处理，最终产生一组全局的、经过归约后的输出键值对。
3. **并行与分布式计算**：
   - MapReduce 框架设计为高度并行和分布式，能够在包含大量节点的集群上高效运行。Map 任务和 Reduce 任务可以在不同节点上并发执行，充分利用集群的计算资源。框架自动处理任务调度、数据分片、任务监控、错误恢复等工作，减轻了程序员在编写并行分布式程序时的负担。
4. **容错与可靠性**：
   - MapReduce 框架内置了容错机制，能够容忍硬件故障和网络中断。通过数据复制、任务重新调度以及检查点等手段，确保即使在部分节点失效的情况下，整个计算任务仍能顺利完成。这种对故障的透明处理增强了系统的稳定性和可靠性。
5. **易于编程**：
   - 对于程序员而言，只需关注两个核心函数（Map 函数和 Reduce 函数）的逻辑实现，无需关心底层的分布式系统细节，如数据分布、任务调度、网络通信、容错处理等。这种抽象简化了并行编程模型，使得非分布式系统专家也能利用大规模集群进行大数据处理。

综上所述，MapReduce 的核心思想是通过将大规模数据处理任务分解为可并行执行的 Map 和 Reduce 阶段，利用分布式计算资源实现高效的数据处理，同时提供了一种简洁易用的编程接口和内置的容错机制，以适应大规模、复杂数据集的离线批处理需求。尽管它并不适合所有类型的计算任

务（如实时计算、复杂依赖关系的 DAG 计算等），但对于那些可以自然地表达为“分而治之”模式的批处理作业，MapReduce 仍然是一个强大且实用的工具。

流式计算框架

流式计算框架专为处理数据流而设计，它们允许数据到达后即时进行处理，无需等待数据全部收集完毕再启动计算过程。这种实时或近实时的数据处理方式非常适合于需要快速响应、持续分析或更新结果的应用场景，如实时监控、交易分析、欺诈检测、社交网络分析等。以下是对 Flink、Spark Streaming 和 Storm 这三大代表性开源流式计算框架的简要概述：

Apache Flink

特点：

- **实时处理与事件时间处理**：Flink 提供了强大的实时流处理能力，支持精确的事件时间窗口和 watermark 机制，能够*正确处理乱序事件*，保证结果的准确性。
- **状态管理**：Flink 内建了高效的状态管理机制，支持有状态的流式计算，允许在流处理过程中维护和更新状态信息，这对于复杂的流处理逻辑和长时间窗口计算至关重要。
- **统一的 API**：Flink 提出了 DataStream API 和 Table API/SQL，提供了一种统一的方式来处理流数据和批数据，实现了流批一体的处理模式。
- **容错与高可用**：Flink 具有细粒度的 checkpoint 机制，能够保证在出现故障时恢复到一致状态，确保数据处理的准确性和可靠性。
- **灵活部署**：Flink 可以部署在各种环境中，包括本地、集群（如 YARN、Mesos、Kubernetes）、云环境等，并支持与多种数据源和 sinks 集成。

Apache Spark Streaming

特点：

- **微批处理**：Spark Streaming 采用了微批处理（micro-batching）的方式，将实时数据流划分为一系列小的批次进行处理，虽然不是严格意义上的实时，但可以达到*近实时*的效果。
- **与 Spark 生态集成**：作为 Spark 生态系统的一部分，Spark Streaming 可以无缝利用 Spark Core、Spark SQL、MLlib 等组件，实现复杂的流数据分析和机器学习应用。
- **容错与恢复**：基于 Spark 的 RDD（Resilient Distributed Datasets）模型，Spark Streaming 具备容错机制，能够自动从失败的任务或节点中恢复。
- **可伸缩性**：得益于 Spark 的并行计算能力，Spark Streaming 可以轻松扩展以处理大规模数据流。

Apache Storm

特点：

- **低延迟处理**：Storm 最初设计目标就是实现*超低延迟*的实时流处理，能够保证数据在进入系统后立即得到处理，具有毫秒级的处理延迟。
- **简单易用**：Storm 采用 Spout（数据源）和 Bolt（处理单元）的概念构建拓扑结构，编程模型直观，易于理解和开发。

- **可靠处理**：Storm 支持事务性的消息处理，确保每个消息至少被完整处理一次（at-least-once 语义），也可通过配置实现 exactly-once 语义。
- **灵活部署**：Storm 支持本地、集群（如 Apache Mesos、Hadoop YARN）以及云环境部署，且能够与多种消息队列（如 Kafka、RabbitMQ）集成。

应用场景与比较

- **Flink**：适用于对实时性要求较高、需要复杂事件处理和窗口计算、状态管理复杂或者希望统一处理流批数据的场景。其事件时间处理能力和流批一体的架构使其在现代数据处理栈中占据重要地位。
- **Spark Streaming**：适合对实时性要求稍低、强调与 Spark 生态系统的紧密集成、需要进行大规模批处理和流处理协同工作的场景。其微批处理模型在某些场景下可能牺牲一定的实时性，但换来了与 Spark 其他组件的良好兼容性。
- **Storm**：在需要极低延迟处理、简单快速开发流处理应用、或者对实时性要求严苛但对状态管理和窗口计算要求不高的场景下表现优秀。Storm 的简单性使得它成为快速搭建实时处理管道的理想选择。

综上所述，Flink、Spark Streaming 和 Storm 各自具有独特的优点和适用场景，选择哪个框架取决于具体的业务需求、现有技术栈、团队熟悉程度以及对性能、实时性、易用性、生态系统整合等因素的权衡。随着技术的发展，Flink 由于其强大的流处理特性、流批一体的设计以及良好的社区支持，近年来在流处理领域的应用越来越广泛。

流水线模式

典型的框架就是Tensorflow

特点：

- **任务分解**：流水线模式将一个复杂的计算任务拆解为一系列*相互依赖*的子任务（或称为阶段、步骤），形成一个线性的或**有向无环图（DAG）** 的工作流程。
- **并行与串行混合**：各子任务之间可能存在串行依赖（前一个子任务完成才能开始下一个），也可能存在并行执行的部分（多个子任务可以同时进行），具体取决于任务间的依赖关系。
- **数据流驱动**：在流水线中，数据以流的形式从一个子任务传递到下一个子任务，形成数据处理的连续链条。每个子任务仅处理其上游任务传递过来的数据，并将其处理结果传递给下游任务。
- **高效资源利用**：通过合理安排任务间的并行与串行执行，流水线模式可以减少数据在计算节点之间的传输成本，提高资源利用率，并通过尽早开始后续任务来缩短整体处理时间。

“流批一体”技术

特点：

- **统一处理模型**：流批一体技术旨在提供一种统一的编程模型和执行引擎，既能处理实时流数据，又能处理批量历史数据，无需为两种场景分别开发和维护不同的代码或系统。
- **灵活窗口定义**：支持对流数据进行滑动窗口、滚动窗口、会话窗口等多种窗口操作，使得流处理具备批处理类似的窗口聚合能力，同时也允许批处理使用流处理的窗口逻辑来处理时间序列数据。

- **状态管理与容错**：对于有状态的流处理，流批一体框架通常提供一致的状态管理机制和容错保障，确保无论是流数据还是批数据都能得到一致且准确的结果。

分布式共识算法-Raft

https://thesecretlivesofdata.com/raft/

CAP理论

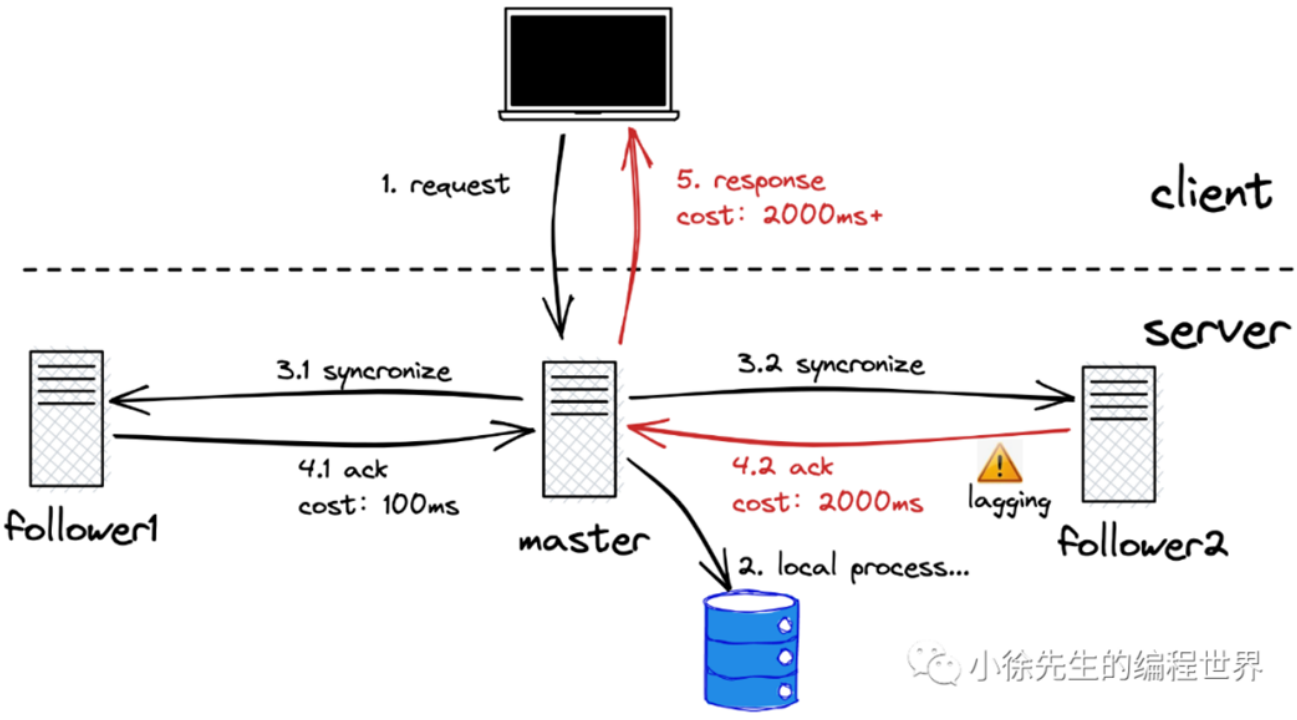
- **数据一致性**：强调数据正确性，每次操作，要么读到最新，要么读失败，整个分布式系统是一个不可拆分的整体，广义“原子性”。
- **可用性**：客户端请求能够得到响应，不发生错误，也不能出现过长的等待时间。
- **分区容错性**：在网络不可靠的情况下，整个系统仍然正常运作的

CAP 理论强调的是，一个系统中，C、A、P三项性质至多只能满足其二，即每个系统依据其架构设计会具有 CP、AP 或者 CA的倾向性.

数据一致性问题【即AP下的问题】：

- 即时一致性问题
- 顺序一致性问题

服务高可用性问题【即CP下的问题】：

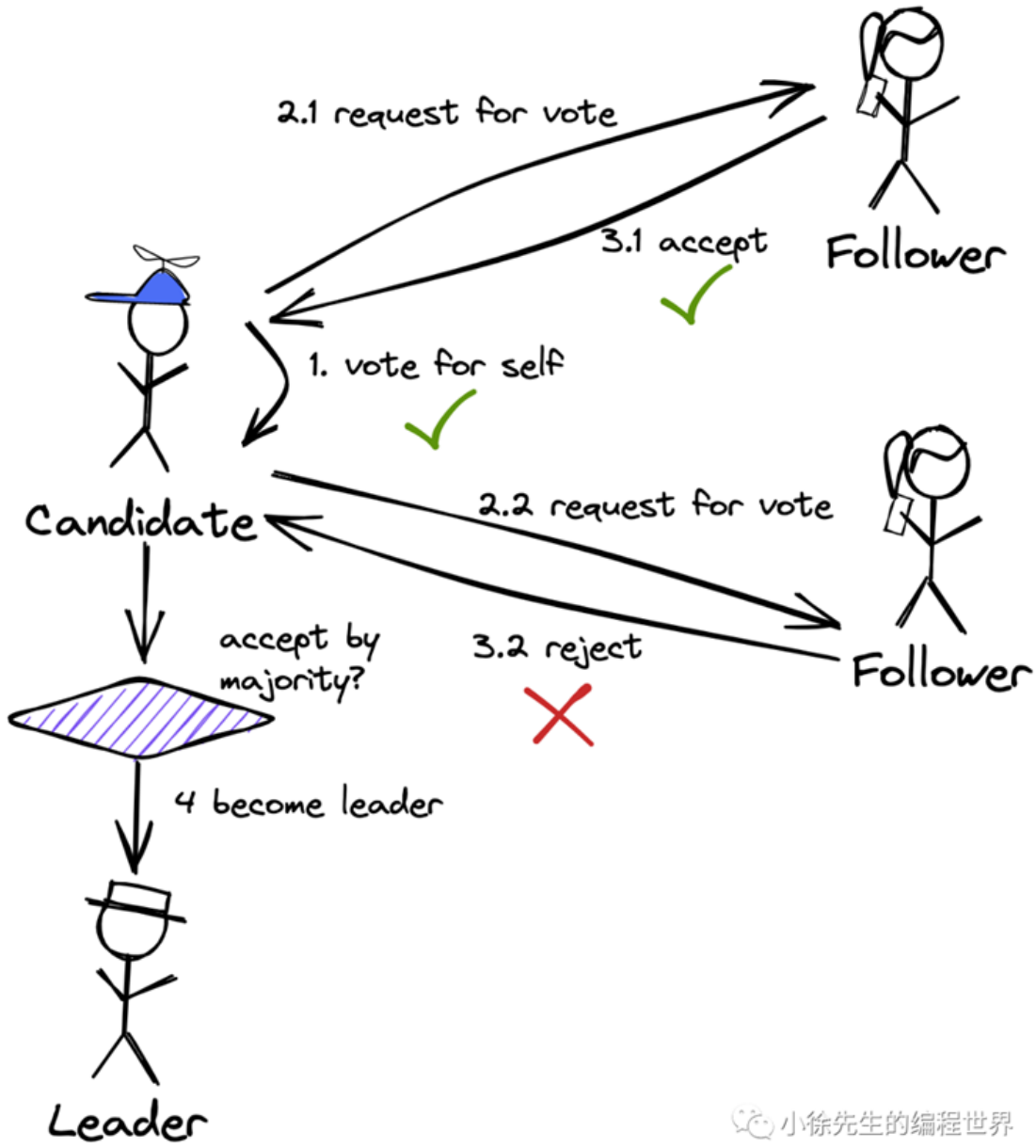


- 倘若集群中某个follower 出现宕机,master 周步数据时会因为未集齐所有 follower 的响应，而无法给客户端 ack，这样一个节点的问题就会被放大到导致整个系统不可用
- 倘若某个follower的网络环境或者本机环境出现问题，它给出同步数据响应的时间出乎意料的长，那么整个系统的响应效率都会被其拖垮，这就是所谓的木桶效应

分布式共识算法： Raft

- 在可用性 A方面，raft算法能保证当分布式系统中存在半数以上节点存活时，系统是稳定可用的，同时请求耗时取决于多数派的下限，而非所有节点的下限;
- 在一致性 C方面，标准的raft算法能够保证数据满足最终一致性，同时在各种工程化的落地实现中，我们可以在raft算法的基础上稍加改进，即可保证数据的即时一致性，进一步做到强C.

关键组件与术语



1. **领导者（Leader）**：在一个Raft集群中，任何时候都存在且只能存在一个领导者节点。领导者负责接收客户端请求、管理日志复制以及发起心跳消息以维持其*领导地位*。
2. **跟随者（Follower）**：除领导者外的其他节点均为跟随者。跟随者*被动响应领导者的消息*，如投票请求、日志复制请求等，并定期发送心跳回应以保持与领导者通信。职责相对简单但同样重要，因为这是一个*基于多数派原则*运作的民众团体，所有角色只要拧成一股绳，聚成了多数派，就能代表整个系统进行决断，甚至包括推翻 leader。
3. **候选人（Candidate）**：当跟随者在一段时间内未收到领导者的心跳时，可以自我提升为候选人，尝试通过选举过程成为新的领导者。
4. **日志条目（Log Entry）**：每个条目包含一个状态机指令和一个任期号。日志是按顺序排列的，每个条目的索引表示其在日志中的位置。

5. **任期（Term）**：Raft使用任期来跟踪逻辑时间，每个任期有一个唯一的编号（递增整数）。任期内可能会有一次或多次领导者选举，但同一任期内只会有一个合法的领导者。

工作原理

领导者选举

- **随机超时**：每个节点都有一个随机的选举超时时间（通常取值在一个范围内），超时后如果没有收到有效领导者的心跳，进入候选人状态并发起新一轮选举。每个候选人都会向其他节点发送 `RequestVote` 请求，试图赢得他们的选票。
- **请求投票（RequestVote RPC）**：候选人向其他节点发送请求投票的消息，包含其当前任期号和最后已知的日志条目信息。
- **投票规则**：节点收到投票请求后，根据以下规则决定是否投票：
  - 如果候选人的任期号小于自身已知的任期号，拒绝投票（遵循任期号更大的原则）。
  - 如果候选人的任期号等于自身已知的任期号，且候选人的日志至少与自己一样新（即最后一条日志的索引和任期号都不小于自己的），则投票给该候选人。
- **赢得选举**：候选人获得集群中大多数节点的投票（即超过半数节点的投票）后，成为新的领导者。

> *多数派原则* 在此处体现为：候选人需要获得集群中大多数节点（即超过半数）的投票才能当选为新的领导者。一旦某候选人获得多数票，它即刻成为合法的领导者，并开始向其他节点发送心跳消息以维持领导地位。剩余的候选人（以及未投票的跟随者）在收到新领导者的心跳后，会承认其领导地位并退回跟随者状态。

读写分离

- 读操作可以由集群的任意节点提供服务
- 写操作统一需要 leader 收口处理，并向 follower 同步，倘若 follower 率先收到了来自客户端的写请求，也需要转发给 leader 进行处理。

状态机与预写日志

状态机 (state machine) 是**节点实际存储数据的容器**, 写请求的最后一步是将结果写入状态机而读请求也需要从状态机中获取数据进行响应.

预写日志 (write ahead log，简称 wal) 是**通过日志的方式记录下每一笔写请求的明细**，使得变更历史有迹可循, 在 raft 算法中，**写请求会先组织成预写日志的形式添加到日志数组中**，当一个日志 (写请求) 达到集群多数派的认可后，才能够被提交将变更应用到状态机当中

日志复制

- **客户端请求**：客户端的所有写请求都直接发送给领导者。领导者将请求转化为一个新的日志条目，附加到自己的日志中，并为该条目分配一个唯一的索引和当前任期号。然后，领导者通过 `AppendEntries` RPC 消息将新日志条目同步给所有跟随者。
- **日志复制（`AppendEntries RPC`）**：领导者定期向跟随者发送日志复制请求，包含连续的一系列日志条目及其索引和任期号，以及跟随者应将这些条目追加到其日志的起始索引。跟随者接收到日志复制请求后，检查日志的一致性：

- 如果领导者发送的日志条目已经在跟随者的日志中，且索引位置和内容完全一致，则确认接收，更新其已提交索引（commitIndex）。
- 如果发现不一致，例如跟随者的日志在相应位置已有更高任期号的条目，领导者将回退并发送缺失的、较低任期号的条目。

> *多数派原则* 在日志复制中表现为：领导者只有在确认一个日志条目已被大多数节点复制（即该条目之前的所有日志条目都已复制到大多数节点）后，才会将该条目标记为已提交（commit）。这意味着即使领导者后续崩溃，新的领导者（通过新一轮选举产生）也将拥有相同的已提交日志，因为它们都是基于大多数节点已有的日志信息。

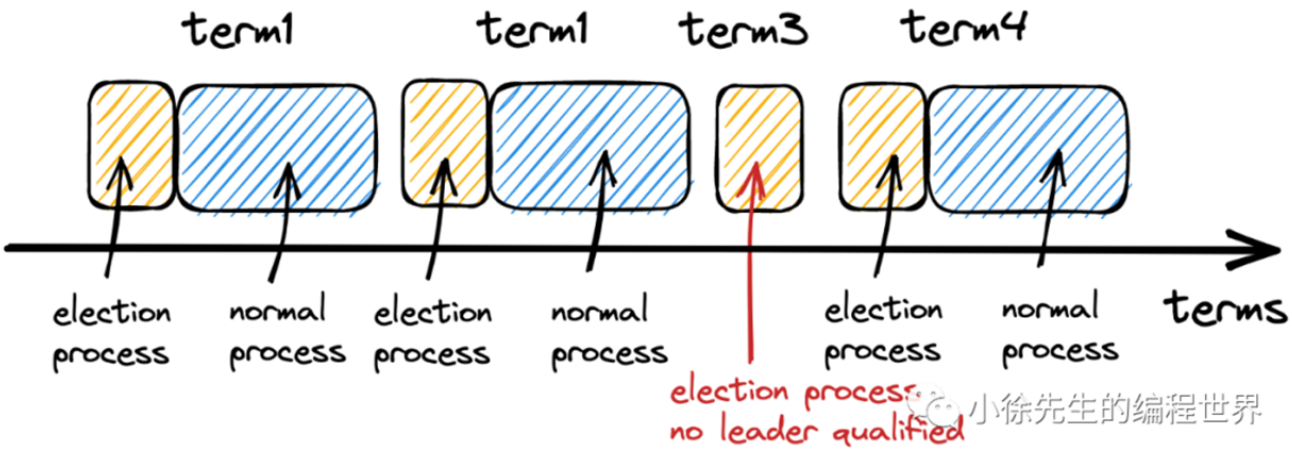
- **日志强制同步**：如果跟随者的日志落后太多，领导者会强制跟随者删除其冲突的日志部分，以确保跟随者的日志与领导者完全一致。

两阶段提交

- Leader 将写请求添加到本地的预写日志中，并向集群中其他节点广播同步这笔写请求. 这个过程可以称之为“提议”（proposal）
- 倘若集群总计*半数以上的节点（包括 leader 自身）都将这笔请求添加预写日志，并给予了 leader 肯定的答复（ack）*，那么 leader 此时会“提交”这个请求，并给予客户端写请求已成功处理的响应；

任期与日志索引

任期 term，集群由一个 leader 切换到另一个 leader ，对应的任期数会进行累加。



- Term：标志了这则日志是哪个任期的 leader 在位时同步写入的
- Index：标志了这则日志在预写日志数组的位置.

通过 {term , index} 二元组可以组成一个全局唯一键，定位到一则日志，并且能够保证位于不同节点中日志，只要其 term 和 index 均相同，其内容一定完全一致.

安全性保证

- **领导权转移**：新领导者上任时，其日志至少与之前任期的领导者一样新，因此不会丢失任何已提交的日志条目。

- **日志复制与提交**：领导者只有在确认某个日志条目已经被大多数节点复制（即该条目之前的日志条目都已复制到大多数节点的日志中）后，才会将该条目标记为已提交（通过更新commitIndex）。这样，即使领导者发生故障，新的领导者也会拥有相同的已提交日志，从而保证了状态机的一致性。

附加特性

- **集群成员变更（Joint Consensus）**：Raft提供了安全的集群成员变更机制，允许在不中断服务的情况下动态添加或移除节点，通过维护一个过渡期的联合共识来保证数据一致性。

应用实例

Raft因其简洁性和易实现性，在实际工程中得到广泛应用，如Etcd（分布式键值存储系统）、Consul（服务发现与配置工具）等均采用Raft作为其底层共识算法。

总结来说，Raft算法通过领导者选举、日志复制和严格的投票规则确保了分布式系统中节点间的日志一致性，进而保证了状态机的一致性执行。其清晰的阶段划分和有限状态机模型使Raft易于理解和实现，成为分布式系统中实现共识的一种主流选择。