第5章

共识算法

共识算法是区块链的核心技术之一，直接影响区块链系统的性能、安全和可扩展性。为满足特定的设计目标和应用需求，各种共识算法的设计思想有所不同。

本章首先阐述共识算法的基本概念、基础理论及其分类，进而介绍故障容错类共识算法和拜占庭容错类共识算法两种共识类型及其代表算法的实现原理。其中，故障容错类共识算法研究无恶意节点的网络，包括多阶段提交协议、Paxos和Raft等算法；拜占庭容错类共识算法研究存在恶意节点的网络，包括PBFT、PoW、PoS、DPoS等算法。最后对共识算法的发展趋势进行展望。

5.1 共识算法概述

5.1.1 共识算法的概念

区块链系统通常由分布在互联网上的多个服务节点共同组成，是典型的分布式系统。分布式系统要解决的核心问题是如何保证不同节点在执行一系列操作后得到一致性的处理结果。区块链共识机制是分布式节点间根据协商一致的规则选定分布式账本记账权归属的算法，通过“一人记账大家认同”的方法对“交易”达成共识，从而保障账本数据的真实性和一致性。

单机系统中因为系统数据副本唯一，读写操作被执行或者被放弃，所以不存在一致性问题。但分布式系统中数据通常存在多个副本，且节点和节点间的通信随时可能发生故障，增加了同步数据的难度。一致性问题处理不好，可能造成重大损失。例如，用户在某银行网点取款后，如果数据未能及时同步，又到其他网点取款，可能出现同一笔款项被多次取走的情形。

共识算法是保证分布式系统达成一致的算法，也是区块链系统的核心技术之一，区块链系统基于共识算法，使交易顺序、交易内容、智能合约执行、区块生成等提案达成一致。正确的共识算法需要满足以下3个要求。

(1) 可终止性(Termination)：有限时间内达成共识。

(2) 共同性(Agreement)：所有节点的最终决策值相同。

(3) 有效性(Validity)：决策值由某个合法(有效)节点提出。

5.1.2　共识算法的基础理论

共识算法的基础理论包括拜占庭容错理论、FLP 不可能原理、CAP 定理和 BASE 理论，如表 5-1 所示。

表 5-1　共识算法的基础理论

理　论	发表年份	简 要 概 括
拜占庭容错理论	1982 年	针对恶意伪造消息的容错机制
FLP 不可能原理	1985 年	允许节点失效的异步系统无法达成共识
CAP 定理	2002 年	分布式系统中不同能力属性间的相互限制
BASE 理论	2008 年	大规模分布式系统设计指导

1. FLP 不可能原理

FLP 不可能原理是指在网络可靠但允许至少一个节点失效的异步模型系统中，无法存在一个能够确定性解决一致性问题的共识算法。这一原理由科学家 Fischer、Lynch 和 Patterson 在 1985 年共同发表的论文 *Impossibility of Distributed Consensus with One Faulty Process* 中首次提出。

在分布式系统中，节点间通过通信进行资源共享与协同工作。根据通信模型的不同，系统可以分为同步通信系统和异步通信系统。同步通信系统假设各节点的时钟误差有界，消息传递和处理时间均有限；而异步通信系统则面临更大的挑战，其中节点时钟误差可能很大，导致难以实现时间同步，且消息传输和处理的延迟可能是任意时间。

由于现实世界中分布式系统的复杂性，其通信模型往往更接近异步模型。根据 FLP 不可能原理，在该环境下，如果允许节点失效，就无法保证系统能够达成一致性共识。因此，异步分布式系统在设计一致性共识算法时，必须对某些条件进行限制或妥协。

尽管 FLP 不可能原理揭示了设计共识算法的困难，但实际应用中，人们仍可通过各种优化手段和取舍策略绕过这一限制，实现系统的一致性。

2. CAP 定理

CAP 定理是由加州大学伯克利分校的计算机科学家 Eric Brewer 在 2000 年提出的猜想，随后在 2002 年由麻省理工学院的 Seth Gilbert 和 Nancy Lynch 证明为定理。

CAP 定理指出，在分布式系统中，一致性（Consistency）、可用性（Availability）和分区容错性（Partition Tolerance）三个特性不可能同时完全满足，最多只能满足其中两个，如图 5-1 所示。具体而言：

（1）一致性要求分布式系统中的所有数据备份在同一时间点上是相同的。

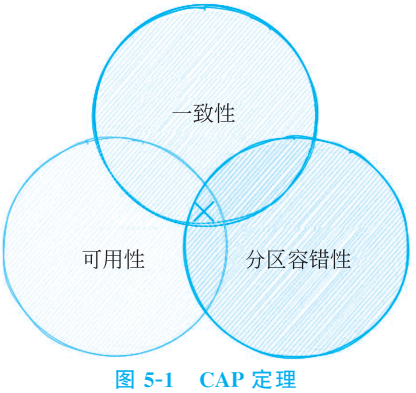

图 5-1　CAP 定理

（2）可用性要求系统在部分节点发生故障时，仍能正常响应客户端的请求。

（3）分区容错性要求系统在网络分区（因通信延迟或中断等原因造成）的情况下，仍能提供服务。

在实际系统设计中，三个特性往往需要进行权衡。由于分区容错性是分布式系统的基本需求，系统通常偏向提供一致性服务（如 BigTable、HBase）或可用性服务（如 CouchDB、Cassandra）。

3. BASE 理论

BASE 理论是 eBay 架构师 Dan Pritchet 在 2008 年提出的关于大规模分布式系统一致性问题的实践总结。BASE 代表基本可用性（Basically Available）、软状态（Soft State）和最终一致性（Eventual Consistency）。

（1）基本可用性表示在出现故障时，系统可以损失部分可用性以保证核心服务的运行。例如，在访问量激增时，可以延长部分用户的请求响应时间或限制某些非核心功能的使用。

（2）软状态允许系统存在短暂的不一致状态，但这种状态不会影响系统的整体可用性。

（3）最终一致性强调经过一段时间后，系统能够达到一致的状态。即系统不必在任何时间点都保持一致性，但最终能够达到一致状态。

BASE 理论是对 CAP 定理的进一步发展和权衡。强调在无法做到 CAP 的强一致性（Strong Consistency）情况下，可以通过适当的策略实现最终一致性（Eventual Consistency）。BASE 理论对设计分布式系统具有重要意义，为人们提供了一种在复杂环境中实现系统一致性的实用方法。

5.1.3 共识算法的分类

1. 故障容错（CFT）类共识算法

CFT 类共识算法设计用于处理如网络中断、存储设备失效或服务器崩溃等常规故障情况，保证系统仍能达到共识。CFT 类共识算法不考虑恶意节点的存在，即假定所有节点都将诚实地执行协议，不会出现故意破坏或欺诈行为。

2. 拜占庭容错（BFT）类共识算法

BFT 类共识算法不仅涵盖了 CFT 类共识算法所能处理的常规故障，还进一步考虑系统中可能存在恶意节点的情况。恶意节点是指不响应其他节点的请求、故意发送错误或伪造的信息、给不同节点发送矛盾的信息或以其他方式攻击系统的节点。在区块链技术背景下，BFT 类共识算法可进一步细分为确定性共识、概率性共识及混合协议等形式（具体细节参见 5.3.1 节）。

在公有区块链中，由于系统的开放性，任何节点都可以在未经许可的情况下加入网络，难免存在恶意节点。因此绝大多数公有区块链系统使用拜占庭容错类共识算法。而在联盟链或私有链的场景中，网络节点的加入需要经过授权，成员间存在一定的信任基础，因此可以选择使用 CFT 类共识算法。

图 5-2 对这两类共识算法进行了归纳分类，而表 5-2 则对不同区块链中代表性的共识

算法进行了对比。

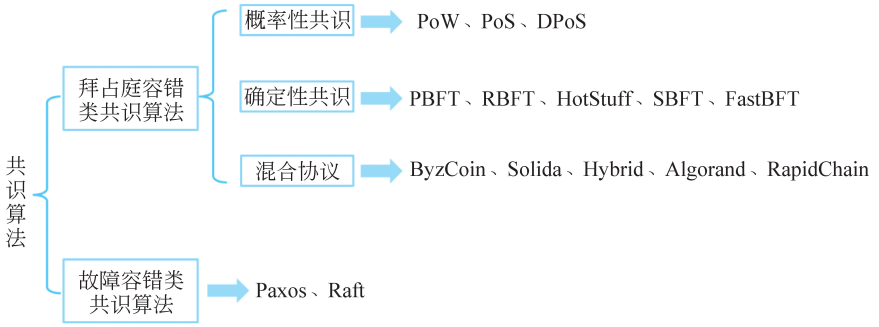

图 5-2　共识算法的分类

表 5-2　代表性共识算法的对比

共识算法	类型	适用类别	选择出块节点方式	优缺点	典型区块链应用
Raft	CFT	私有链、联盟链	广播竞选	优点：易于理解和实现，性能好； 缺点：非拜占庭容错，可扩展性差	Hyperledger Fabric
PBFT	BFT	私有链、联盟链	轮流选择	优点：效率高，性能好； 缺点：通信代价高，可扩展性差	Hyperledger Fabric
PoW	BFT	公有链	资源竞争	优点：分布式，可扩展性较好； 缺点：资源消耗大，性能差	比特币、以太坊 1.0
PoS	BFT	公有链	权益竞争	优点：节约资源，性能较好； 缺点：因权益集中导致中心化倾向	点点币、以太坊 2.0
DPoS	BFT	公有链	权益竞争	优点：进一步提升性能； 缺点：中心化程度较高	比特股、EOS

5.2　故障容错类共识算法

故障容错类共识算法的研究起源于 20 世纪 80 年代，这类共识算法主要应对系统中的常规故障，如网络异常、硬件故障等，而不考虑恶意节点的存在。常见的故障容错类共识算法包括主从同步/异步读写、多数派读写、多阶段提交协议、Paxos 共识算法和 Raft 共识算法。

5.2.1　主从同步/异步读写

在分布式集群中，算法通常配置一个主节点和多个从节点。所有客户端的写操作请求首先发送到主节点，然后由主节点负责将写入操作复制到各个从节点。复制成功后，从节点向主节点发送确认信息。如果主节点等待所有从节点都返回“复制成功”的消息后才向客户端发送响应，称为主从同步读写。如果主节点在接收到客户端的写请求后，立即向其发送成功的响应，然后再异步将数据复制到其他从节点，称为主从异步读写。但无论是同步还是异步方式，都存在主节点单点故障的风险，且缺乏有效的容错机制，因此在可靠性和可用性方面表现欠佳，在实际应用中较少见。

5.2.2 多数派读写

在多数派读写的分布式集群中，所有节点地位平等。每次进行数据读写操作时，都需要获得超过半数节点的同意，这些节点被称为法定节点数(Quorum)。多数派读写系统能够满足最终一致性的要求，并已被广泛应用于如 Dynamo、Cassandra 等分布式数据库系统。然而，算法仍面临非原子更新、脏读以及更新丢失等问题，因此在高可用性方面仍有待提升。

5.2.3 多阶段提交协议

多阶段提交协议主要包括两阶段提交(Two-Phase Commit，2PC)和三阶段提交(Three-Phase Commit，3PC)两种。

1. 两阶段提交协议

2PC 协议是一种保证事务原子性的提交协议。在分布式系统中，数据通常被切分成多个数据块并分散存储在不同的服务器上。当进行数据事务访问时，每个服务器节点都知道自己的操作是否成功，但无法了解其他节点的执行情况。为保证所有服务器上的操作要么全部成功，要么全部失败，2PC 协议引入了协调器(Coordinator)控制所有的节点。该协议的思路可概括为：参与者将操作结果告知协调器，由协调器根据参与者的反馈决定是否继续执行后续操作。2PC 协议主要分为准备和提交两个阶段。

(1) 准备阶段。

事务协调器向所有参与者发送事务内容，并询问是否可以执行提交操作，然后等待参与者的响应。每个参与者执行事务操作，并将操作记录在日志中，但此时并不提交事务。如果参与者成功执行了事务，便向事务协调器反馈 Yes，表示可以提交事务；否则反馈 No，表示无法提交事务。

(2) 提交阶段。

如果所有参与者都反馈 Yes，事务协调器便向所有参与者发送“提交”请求，参与者接收到请求后正式提交事务。如果有参与者反馈 No 或者超时未反馈，事务协调器便向所有参与者发送“回滚”请求，参与者接收到请求后，将准备阶段的日志信息回滚到之前的状态。完成提交或回滚操作后，参与者释放事务处理过程中使用的所有资源。整个过程如图 5-3 所示，其中图 5-3(a)表示执行事务提交，图 5-3(b)表示执行事务回滚。

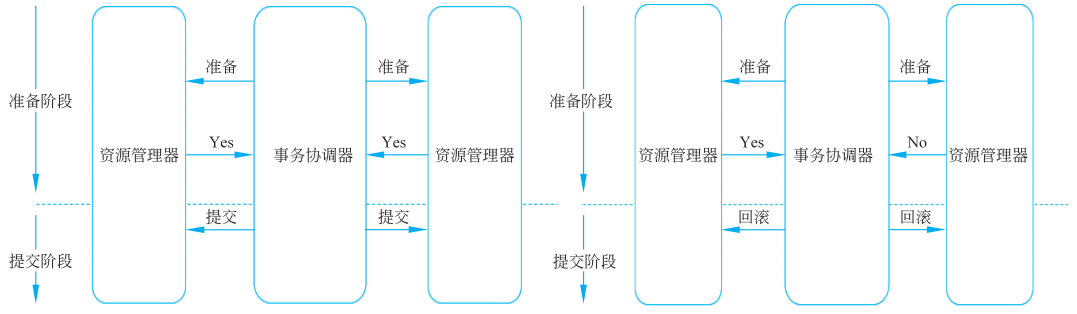

图 5-3 2PC 协议的提交

两阶段提交协议虽然在一定程度上能够保证分布式事务的原子性，但也存在如下缺陷。

(1) 同步阻塞问题。

在 2PC 协议的执行过程中，参与者在等待其他参与者响应时，其事务操作逻辑将处于阻塞状态，无法处理其他事务或请求，导致系统资源的利用率下降。同步阻塞限制了系统的并发性能和吞吐量，特别是在参与者数量众多或网络延迟较大的情况下，问题更为突出。

(2) 单点故障问题。

2PC 协议中的事务协调器负责整个事务的协调和决策。一旦事务协调器发生故障，如宕机或网络中断，参与者将无法接收到进一步的指令，整个 2PC 协议将陷入停滞状态，参与者也无法释放被占有的资源。

(3) 数据不一致问题。

在 2PC 协议的提交阶段，如果由于网络故障等原因，只有部分参与者成功接收到“提交”请求，而其他参与者未能接收到该请求。接收到“提交”请求的参与者将执行事务提交操作，而未接收到请求的参与者则不会进行提交，从而造成不同参与者间数据状态的不一致。

为了解决两阶段提交协议存在的缺陷，研究者提出了诸多改进方案，其中三阶段提交协议是一种重要的改进方法。其通过引入额外的阶段和机制，提高了分布式系统的可靠性、可用性和性能。

2. 三阶段提交协议

三阶段提交协议是对两阶段提交协议的扩展，主要通过增加预提交(preCommit)阶段优化事务的提交过程，有助于减少因协调器故障而导致的事务阻塞问题。3PC 协议由准备提交(canCommit)、预提交(preCommit)和提交(doCommit)三个阶段组成。

1) 准备提交阶段

事务协调器向所有参与者发送 canCommit 请求，询问是否准备好进行事务提交。参与者根据自己的状态和资源情况，判断是否能够成功执行事务。如果确定可以执行，则向协调器反馈 Yes，并进入预备状态；否则，反馈 No。

2) 预提交阶段

当协调器收到所有参与者的 Yes 响应后，进入预提交阶段。协调器向所有参与者发送 preCommit 请求。参与者接收到请求后，执行事务操作并记录相关日志，但此时并不提交事务。如果事务执行成功，参与者向协调器发送 Ack 响应，表示已准备好进行最终提交；如果执行失败或无法执行，则反馈 No。如果协调器在预提交阶段收到任何参与者的 No 响应或超时未收到响应，则向所有参与者发送中断(Abort)请求，参与者回滚事务并释放资源。

3) 提交阶段

如果协调器在预提交阶段收到了所有参与者的 Ack 响应，便进入提交阶段，向所有参与者发送 doCommit 请求。参与者收到请求后，正式提交事务并释放所占用的资源。完成事务提交后，参与者向协调器发送 haveCommited 消息。当协调器收到所有参与者的 haveCommited 消息时，整个事务被视为成功提交。如果在提交阶段出现任何问题，如参与者反馈 No 或超时未反馈，协调器将向所有参与者发送中断请求，参与者回滚事务并释放资源。

图 5-4 展示了 3PC 的成功执行流程，从准备提交到最终确认，各阶段均经过细致设计与

紧密协调，以保证分布式事务的原子性和数据一致性。

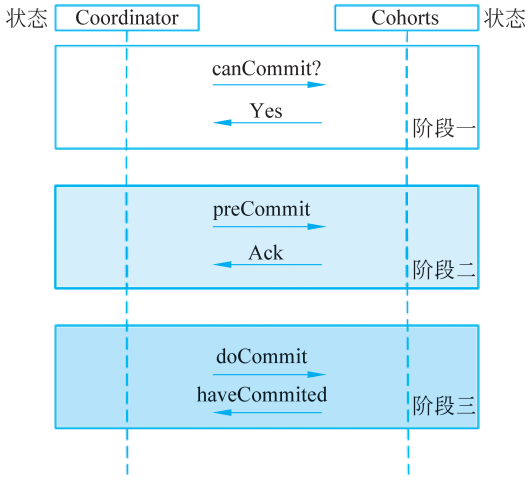

图 5-4　3PC 的成功执行流程

虽然 3PC 在某些方面相比 2PC 有所改进，增强了某些故障场景下的容错性，但其整体容错能力依然受限。特别是在 doCommit 阶段，若协调器与部分参与者间发生网络隔离，参与者可能因无法接收协调器的中断指令，默认执行事务提交操作，而未执行回滚操作，导致数据状态在不同节点间出现不一致。由此可见，虽然 3PC 减轻了 2PC 的阻塞问题，但在解决分布式环境下的数据一致性问题上仍存在局限性。

5.2.4　Paxos 共识算法

Paxos 共识算法由分布式系统领域的杰出人物、图灵奖得主 Leslie Lamport 在 1998 年提出，堪称共识算法中的经典之作。该算法在业界被广泛应用，有效克服了多数派读写、多阶段提交协议等早期算法存在的不足。Google Chubby 的作者 Mike Burrows 曾高度评价 Paxos，认为世界上只有一种一致性算法，就是 Paxos，其他算法都是 Paxos 的不完整版本。著名的 Multi-Paxos、Raft 和 Zab 等算法均是基于 Paxos 算法的优化与改进算法。

Paxos 算法的基础版本，即 Basic Paxos，是整个 Paxos 系列算法的核心，解决了多节点间针对特定提案值的一致性共识问题，并保证每次只能对一个确定的提案达成共识。

1. Paxos 算法的基本假设

Paxos 算法建立在以下 3 个基本假设上。

(1) 数据存储具有高稳定性，不会遭受删除或篡改。

(2) 系统允许节点出现异常，包括服务进程的慢速运行、重启或终止等异常情况。

(3) 系统允许网络出现异常，涵盖消息的延迟、丢失、重复以及乱序等问题，但消息内容不会被篡改。

2. Paxos 算法中的节点角色

Basic Paxos 算法中的所有节点被划分为提案者(Proposer)、接受者(Acceptor)和学习者(Learner)3 种角色，如图 5-5 所示。

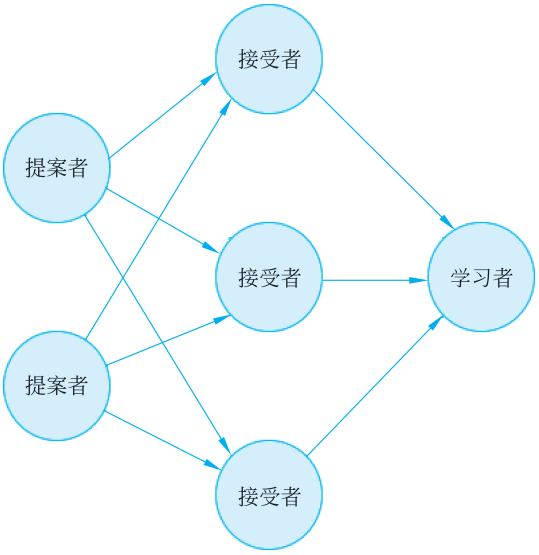

图 5-5　Paxos 算法中的三种角色

Paxos 各角色的主要职责分别如下。

(1) 提案者：负责接收客户端请求，并据此提出包含提案编号和提案值的提案。其中，提案编号为全局唯一且递增。

(2) 接受者：对提案进行共识投票，并存储已达成共识的提案值。

(3) 学习者：不直接参与共识投票过程，而是接收并执行已达成共识的提案值，并根据需要对外进行传播。

3. Basic Paxos 算法的主要流程

Basic Paxos 算法的执行过程主要分为 Prepare、Accept 和 Learn 3 个阶段。

1) Prepare 阶段

提案者向全体接受者广播带有提案编号的 Prepare 请求，试探是否可获得大多数接受者的支持。接受者在收到请求后，将保留当前所见编号最大的提案。若收到的提案编号大于自身当前编号，则更新为最大提案号，并向提案者发送 Yes 响应。

2) Accept 阶段

当提案者收到来自多数接受者的 Yes 反馈后，便向所有接受者发送包含提案编号的 Accept 请求。接受者在收到请求后，针对该提案进行 Accept 处理。

3) Learn 阶段

提案者收到来自多数接受者的 Accept 确认，标志本次 Accept 流程成功，达成共识决议。提案者将这一决议广播给所有学习者。

Paxos 算法作为首个被严格证明有效的共识算法，展现出强大的容错能力，能够应对系统异步通信、消息丢失以及进程失效等复杂场景。然而，Paxos 算法也存在“活锁”问题。活锁是指系统虽未发生阻塞，但由于某些条件未满足而持续处于活跃状态，不断进行“重试-失败”的循环，导致无法取得实际进展。例如，在 Paxos 算法中，若连续出现两个编号递增的提案，便可能触发活锁现象。具体而言，当 Proposer A 提出编号为 M 的提案，并在 Prepare 阶

段获得多数支持后，若在进入 Accept 阶段前，Proposer B 提出了编号为 $M+1$ 的新提案，并同样获得多数支持，则 Proposer A 的提案 M 将无法通过 Accept 阶段。为寻求共识，Proposer A 可能继续提出编号为 $M+2$ 的新提案，进而又使 Proposer B 的提案 $M+1$ 无法通过 Accept 阶段。如此往复，系统便陷入活锁状态，无法满足共识算法的“可终止性”要求，导致提案无法达成一致。但在实际应用中，Paxos 算法遭遇活锁的概率相对较低。后续的优化算法如 Multi-Paxos 等通过引入领导人选举、超时重试等机制，有效避免了活锁情况的发生。

Paxos 算法因“难以理解和实现”而著称，但作为共识算法的重要支柱之一，依然受到谷歌等科技巨头的青睐。例如 Chubby、Bigtable、Spanner 以及 Megastore 等分布式系统，都是基于 Paxos 算法实现了高效的共识机制。

5.2.5 Raft 共识算法

Raft 算法由斯坦福大学的 Diego Ongaro 和 John Ousterhout 共同提出，是在 Paxos 算法基础上设计的更易于理解和实现的分布式系统共识算法。其将分布式系统的一致性问题分解为三个核心子问题：领导人选举(Leader Election)、日志复制(Log Replication)以及安全性(Safety)。

1. 领导人选举

在 Raft 共识算法中，节点被划分为三种状态：领导人(Leader)、候选人(Candidate)和群众(Follower)，状态之间的转换遵循严格的规则，如图 5-6 所示。

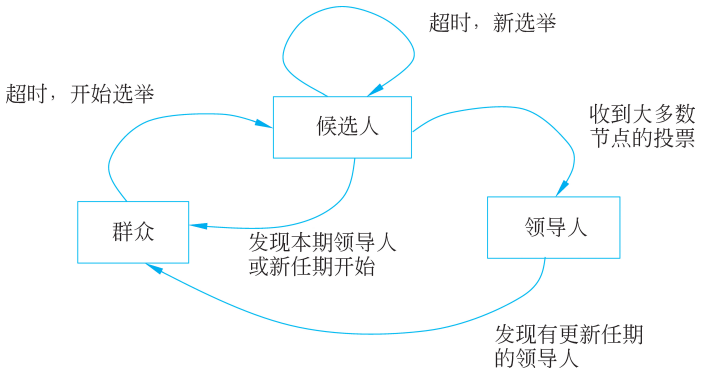

图 5-6 Raft 算法中节点状态转换规则

领导人以固定的时间间隔向系统中的节点广播心跳消息，以宣告自己的活跃状态。若群众节点在预定的时间窗口内未能接收到来自领导人的心跳消息，则判定系统中当前无有效领导人，并自动将自身状态转变为候选人。此时，Raft 系统触发领导人选举流程，候选人向其他节点发起投票邀请(通过 RequestVote RPC)。若某一候选人节点成功获得大多数节点的投票支持，则晋升为新的领导人。

Raft 算法通过引入“任期”(Term)概念管理时间，每个任期从选举开始，并使用连续递增的整数进行标识。每个任期进一步细分为选举期和任职期两个阶段。在选举期内，候选人节点通过竞选方式争取成为领导人，首个获得多数选票的候选人担任新领导人。新领导

人立即向其他节点发送心跳消息以巩固其地位，其他候选人收到心跳消息后，将放弃当前的竞选活动，并转回群众状态。

在系统运行中，可能出现选举失败的情况，例如多个候选人几乎同时发起投票邀请，但均未能获得所需的多数选票。为防止这种选举失败可能导致的活锁现象，Raft 为候选人设置了选举超时机制。当群众转变为候选人后，若在一定时间内未能成功选举出领导人，则候选人将增加其任期编号，并发起新一轮的选举。尽管选举超时机制可在选举失败后触发新的选举，但也可能引发“选举碰撞”问题，即多个候选人在选举失败后几乎同时再次发起选举，导致连续选举失利。为了减少碰撞的发生，Raft 算法为每个候选人的选举超时时间引入了一个随机值，在一定程度上错开了各候选人再次发起选举的时间点。

一旦领导人选举成功，系统便进入任职期。在此期间，领导人是唯一具有决策权的节点，其他节点均作为群众节点存在。领导人负责处理来自客户端的所有请求，并将请求及其处理结果以日志条目的形式复制到所有群众节点上。只要领导人保持健康状态，当前任期就会持续下去。

2. 日志复制

在 Raft 共识算法中，日志复制是保证分布式系统各节点数据一致性的关键环节。领导人节点负责接收客户端的请求，并将请求及其处理结果作为请求处理条目(Entry)追加到自身的日志中。每个条目都被赋予一个唯一的、递增的索引，以便跟踪和排序。

为了将这些条目同步到整个系统，领导人定期(或在特定条件触发下)将一定时间间隔内收集到的所有请求处理条目打包成数据包，通过 AppendEntries RPC 并行发送给所有群众节点。数据包不仅包含条目内容本身，还附加了领导人的任期信息、最新提交的条目索引等关键元数据。

当群众节点接收到 AppendEntries RPC 后，将执行一系列核对工作，验证请求处理条目的正确性和有效性。如果出现领导人发送的条目索引范围与群众节点本地日志中的条目索引不匹配时，例如领导人发送了索引 5～19 的条目，而群众节点日志中仅有 0～3 的条目，缺少索引 4，则核对不能通过。同样，如果领导人的任期低于群众节点日志中的记录任期，核对也将失败。

如果核对成功，群众节点便向领导人发送“成功”反馈，并在本地日志中追加新的请求处理条目。如果核对失败，群众反馈“失败”信息，并可能请求领导人重新发送缺失或冲突的条目。

领导人收到来自大多数群众节点的“成功”反馈后，将本次 AppendEntries RPC 中包含的所有请求处理条目标记为已提交(Commit)状态。表明条目已经被足够多的节点接收并确认，可以安全应用到系统的状态机中。随后，领导人在心跳消息中通知群众节点最新的提交条目范围，群众节点据此更新自己的提交状态。

通过这种方式，Raft 算法确保了分布式系统中的所有节点都维护一份顺序一致、内容相同的日志条目序列。每个节点按照日志的顺序依次执行这些操作，从相同的初始状态出发，应用相同的操作序列，最终达到一致的结束状态，从而保证系统状态的一致性和同步性。

3. 安全性

在 Raft 共识算法中，安全性通过日志机制和选举策略得到保障。日志机制保证所有节