共识机制原理讲义

1. 绪论

1.1 关于共识

要理解共识机制，首先需要明白什么是共识。我们生活在一个多元化的世界，不存在一模一样的两片树叶，也没有完全相同的两个人。所谓共识，就是一群具有各方面差异性的人在某方面达成了一致意见，并将其上升成为共同遵守的规则。人类文明的发展与繁荣离不开社会群体内个体间的分工协作，而如何分工，如何决策，如何分配成果并奖惩，则全都仰赖一个公认的的则，或者叫共识。所谓共识机制指的是一个群体用以达成并维护共识的方式，共识机制往往决定着群体的组织形式。人类文明的早期阶段，武力是各群体建立和维护共识规则的主要方式，人类依托武力建立了部落、国家等中心化管理权威，制定了中心化的管理规则并惩罚不接受规则的个体。

当然，随着文明的发展与演进，我们开始制定法律、设计选举、引入协商与仲裁机制，这成了公司、协会、政府、学校 等社会团体达成与维系共识的方式，但这一切文明做法的背后依旧仰赖庞大的国家暴力机器支撑。人类文明选择通过武力建立中心化权威以构建与维护群体共识，不仅因为在文明早期这一方式最为快捷，更由于依靠中心化权威的管理方式更加高效和稳固。然而，中心化并非唯一可行的方案，只要我们能够提出一种同样安全稳固便捷而并无需依托中心化权威的共识机制，同样可以建立一套与之相对应的全新组织形式。

图1-1 共识机制场景-举手投票

1.2 区块链中的共识

区块链技术的出现恰恰提供了这样的选择，只不过这套去中心化逻辑所依托的并非现实世界而是相对轻量化的网络环境。共识机制作用的对象也不是个人而是一台台所属权各异的计算机，通过区块链共识机制的应用，可将分散在世界各处的计算机集合至一个统一的网络实现大规模的机器协作，下面我们将从技术的角度对区块链的共识机制进行详细拆解。

区块链作为一种点对点的协作网络，其原理是各节点均掌握一个独立的账本，通过保持账本同步来实现共同记账，而在没有一个中心进行指挥和协调同时网络状态复杂多变的情况下，如何保证空间上完全分散的各节点账本能够完全相同，这就要用到区块链的共识机制了。以比特币为例，我们尝试来讨论一个典型的区块链系统的账本记账过程，其公共账本上记录着比特币网络中每时每刻都在发生的转账交易，但这些转账信息并非即刻被写入账本，而是每隔一段时间进行一次统一记账。每次记账会在账本上新生成一个区块，区块中不仅记录了若干条转账记录，还包含了诞生时间、上一个区块摘要等内容，通过这种方式，账本上各个区块间可依照时间顺序形成链状结构。之所以选择这种区块式的记账方式而非所有节点即时记录所有信息，是由于在整个去中心化的系统中，考虑到不同节点 间网络状态以及空间位置的差异，无法做到数据完全同步，通过结构化数据包的方式可以保证各节点记录结果的正确性。而对于每一个新生的区块，基于几乎同样的原因，各节点分别记录的方法并不足以保障账本的唯一性。针对这一问题，区块链网络中所选择的应对方法是采取某种可自动执行的竞争手段选中一个唯一节点首先打包数据生成区块并记录到自身账本，然后由这个被选中的节点将新生产的区块数据通过互联网通知其他节点，其他节点在收到信息并验证区块无误后，同步记录在自己的账本上，以上就是一次完整的去中心化的分布式记账过程。

共识机制在这个过程中重点解决两个问题：一是完全对等的节点之间如何竞争记账权，区块链技术发展至今，已经有多种竞争记账权的办法，我们稍后详细介绍。第二个问题是由于网络延时等原因，偶尔会出现两个节点均认为自己获得了记账权而去打包区块并通知其他节点，其他节点会将首先收到的通知记录下来而忽略另一个，考虑到两次通知时间间隔较小，必然会造成所有节点账本的写入不一致。对于这种情况，区块链网络的解决办法是暂时性接受这种链条分叉的状态，根据后续若干区块打包节点的记录为准。这种保留最长链的方式能够保证在复杂网络环境下多节点依然能够记录一个权威的账本。

现存独立的区块链网络有上百条，所用的共识机制大同小异，主要区别体现在上文的所述竞争记账方法，因而我们常常也用竞争记账的方式来命名不同的区块链共识机制。最为常见的有PoW、PoS、DPoS等。 PoW（工作量证明）被比特币采用，其竞争记帐逻辑是各个节点竞争计算一个随机数，要得到这个随机数，必须经过一定的工作量。这一方法简单粗暴却极为有效，但常常因为耗费能源、易节点集中而被诟病。 PoS（权益证明）则是通过节点持有代币的数量乘以持有时间分配记账权益，记账权益越高获得记账权概率越大，这种竞争方法的实现对于技术要求较高，且容易成为富翁节点的游戏。DPoS（股份授权证明）共识机制的竞争记账逻辑类似于人民代表大会制度，代币持有者拥有相应额度的投票权，投票选出的节点拥有轮流记账权。采用DPoS的区块链项目最为著名的是EOS，这一共识机制最显著优势在于效率极高同时 更加符合目前人们的接受习惯，但劣势同样明显，主要体现在中心化程度最高，很多人认为其违背了区块链的初衷。

1.3 共识机制和分布式网络

分布式计算机网络由多个相互连接在一起的客户机和服务器组成，并且其中的任一系统都可能与另一个系统进行通信。在这种网络中，不存在一个处理和控制中心，网络中任一结点都至少和另外两个结点相连接，信息从一个结点到达另一结点时，可能有多条路径。同时，网络中各个结点均以平等地位相互协调工作和交换信息，并可共同完成一个大型任务。分组交换网、网状形网属于分布式网络。这种网具有信息处理的分布性、可靠性、可扩充性及灵活性等一系列优点。

共识机制是指以去中心化的方式就网络的状态达成统一协议的过程。也被称为共识算法，有助于验证信息被添加到分类账簿，确保只有真实的事务记录在区块链上。因此，共识机制负责安全地更新分布式网络中的数据状态。已经硬编码到协议中的规则确保在全球计算机网络中总是能找到唯一的数据来源并达成一致。这些规则保护整个网络，实现无需信任的网络，而无需中央数据或中介。

共识机制是决定按照哪一个参与节点记账和确保交易安全完成的重要手段。共识机制需要平衡效率和安全的关系，安全措施越复杂，相应的处理时间越慢。而想要提高处理速度，简化安全措施的复杂度是非常重要的一步。

共识机制同时满足一致性和有效性。一致性是指所有诚实节点保存的区块链前缀部分完全相同，而有效性是指由某诚实节点发布的信息终将被其他所有诚实节点记录在自己的区块中。共识机制确保区块链是容错的，因此是可靠和一致的。与中心化系统不同，用户

不必信任系统中的任何人，区块链共识机制中嵌入的协议规则可以确保只有有效和真实的交易才可以被记在公共透明的账簿中，嵌入网络的协议规则保证了公共分类帐的状态总是随着大众的共识变换而更新。

区块链的去中心化的一个重要优势是分配授权，任何人都能在同一个基础上参与进来。而共识机制可以确保区块链不存在区别对待，从而达到公平分配。由于公共区块链具有开源这一特性，使任何人都可以监督并验证底层源代码对网络中的所有参与者是否公平。

共识机制具有激励的能力，可以通过激励好的行为，或在某些情况下惩罚坏的行为者来实现这一点。例如在工作量证明这一机制中，通过奖励比特币给矿工这一方式，奖励他们每一笔交易的担保和验证。任何运算和安全维护都需要大量的算力和钱财，而共识机制可以使这些资源将更好地用于为系统工作。

习题一

1、共识机制主要解决区块链中的哪两个问题？

2、简述共识机制和分布式网络的关系。

2. 分布式系统的基本概念

区块链本身是一种新型的分布式系统，并且极大的促进了分布式计算领域的发展。了解分布式系统的工作原理，了解共识机制如何使人们在分散的网络上达成共识，我们才能真正了解区块链技术的创新和未来发展的方向。

分布式系统是一种通过网络进行通信使用多台服务器来协同完成计算任务的系统，是一种提高业务承载量的基本手段，通过利用更多的服务器，来解决单个服务器无法同时承载大量的用户使用的问题。

2.1 模型

为了控制规模，在开始讨论分布式系统的协议、原理与设计之前，首先给出在本书中研究的分布式系统在分布式层面的基本问题模型。

2.1.1 节点

节点是指一个可以独立按照分布式协议完成一组逻辑的程序个体。在具体的工程项目中，一个节点往往是一个操作系统上的进程。在本书的模型中，认为节点是一个完整的、不可分的整体，如果某个程序进程实际上由若干相对独立部分构成，则在模型中可以将一个进程划分为多个节点。

2.1.2 通信

节点与节点之间是完全独立、相互隔离的，节点之间传递信息的唯一方式是通过不可靠的网络进行通信。即一个节点可以向其他节点通过网络发送消息，但发送消息的节点无法确认消息是否被接收节点完整正确收到。

2.1.3 存储

节点可以通过将数据写入与节点在同一台机器的本地存储设备保存数据。通常的存储设备有磁盘、SSD 等。存储、读取数据的节点称为有状态的节点，反之称为无状态的节点。如果某个节点A存储数据的方式是将数据通过网络发送到另一个节点B，由节点B负责将数据存储到节点B的本地存储设备，那么不能认为节点A是有状态的节点，而只有节点B是有状态的节点。

2.1.4 异常

分布式系统核心问题之一就是处理各种异常(failure)情况。本节着重讨论在本书范围内系统会遇到的各种异常。

2.1.4.1 机器宕机

机器宕机是最常见的异常之一。在大型集群中每日宕机发生的概率为千分之一左右。在实践中，一台宕机的机器恢复的时间通常认为是24小时，一般需要人工介入重启机器。宕机造成的后果通常为该机器上节点不能正常工作。假设节点的工作流程是三个独立原子的步骤，宕机造成的后果是节点可能在处理完某个步骤后不再继续处理后续步骤。由于本书不考虑拜占庭问题，认为不会出现执行完第一个步骤后跳过第二个步骤而执行第三个步骤的情况。宕机是一个完全随机的事件，在本书中，认为在任何时刻都可能发生宕机，从而造成某些协议流程无法完成。

当发生宕机时，节点无法进入正常工作的状态，称之为“不可用”(unavailable)状态。机器重启后，机器上的节点可以重新启动，但节点将失去所有的内存信息。在某些分布式系统中，节点可以通过读取本地存储设备中的信息或通过读取其他节点数据的方式恢复内存信息，从而恢复到某一宕机前的状态，进而重新进入正常工作状态、即“可用”(available)状态。而另一些分布式系统中的某些无状态节点则无需读取读取任何信息就可以立刻重新“可用”。使得节点在宕机后从“不可用”到“可用”的过程称为宕机恢复。

2.1.4.2 网络异常

网络异常是另一类常见的异常形式。在 2.1.2 中已经定义，节点间通过不可靠的网络进行通信。

2.1.4.3 消息丢失

消息丢失是最常见的网络异常。对于常见的 IP 网络来说，网络层不保证数据报文(IP fragment) 的可靠传递，在发生网络拥塞、路由变动、设备异常等情况时，都可能发生发送的数据丢失。由于网络数据丢失的异常存在，直接决定了分布式系统的协议必须能处理网络数据丢失的情况。

依据网络质量的不同，网络消息丢失的概率也不同，甚至可能出现在一段时间内某些节点之间的网络消息完全丢失的情况。如果某些节点的直接的网络通信正常或丢包率在合理范围内，而某些节点之间始终无法正常通信，则称这种特殊的网络异常为“网络分化/分区”(network partition)。网络分化是一类常见的网络异常，尤其当分布式系统部署在多个机

房之间时。

2.1.4.4 消息乱序

消息乱序是指节点发送的网络消息有一定的概率不是按照发送时的顺序依次到达目的节点。通常由于 IP 网络的存储转发机制、路由不确定性等问题，网络报文乱序也是一种常见的网络异常。这就要求设计分布式协议时，考虑使用序列号等机制处理网络消息的乱序问题，使得无效的、过期的网络消息不影响系统的正确性。

2.1.4.5 数据错误

网络上传输的数据有可能发生比特错误，从而造成数据错误。通常使用一定的校验码机制可以较为简单的检查出网络数据的错误，从而丢弃错误的数据。

2.1.4.6 不可靠的 TCP

TCP 协议为应用层提供了可靠的、面向连接的传输服务。TCP 协议是最优秀的传输层协议之一， 其设计初衷就是在不可靠的网络之上建立可靠的传输服务。TCP 协议通过为传输的每一个字节设置顺序递增的序列号，由接收方在收到数据后按序列号重组数据并发送确认信息，当发现数据包丢失时，TCP 协议重传丢失的数据包，从而 TCP 协议解决了网络数据包丢失的问题和数据包乱序问题。TCP 协议为每个TCP 数据段（以太网上通常最大为1460字节）使用 32 位的校验和从而检查数据错误问题。TCP 协议通过设置接收和发送窗口的机制极大的提高了传输性能，解决了网络传输的时延与吞吐问题。TCP 协议最为复杂而巧妙的是其几十年来不断改进的拥塞控制算法，使得 TCP 可以动态感知底层链路的带宽加以合理使用并与其他 TCP 链接分享带宽（TCP friendly）。

上述种种使得TCP 协议成为一个在通常情况下非常可靠的协议，然而在分布式系统的协议设计中不能认为所有网络通信都基于 TCP 协议则通信就是可靠的。一方面，TCP 协议保证了 TCP 协议栈之间的可靠的传输，但无法保证两个上层应用之间的可靠通信。通常的，当某个应用层程序通过TCP 的系统调用发送一个网络消息时，即使 TCP 系统调用返回成功，也仅仅只能意味着该消息被本机的 TCP 协议栈接受，一般这个消息是被放入了 TCP 协议栈的缓冲区中。再退一步讲，即使目的机器的 TCP 协议栈后续也正常收到了该消息，并发送了确认数据包，也仅仅意味着消息达到了对方机器的协议栈，而不能认为消息被目标应用程序进程接收到并正确处理了。当发送过程中出现宕机等异常时，TCP 协议栈缓冲区中的消息有可能被丢失从而无法被目标节点正确处理。更有甚者，在网络中断前，某数据包已经被目标进程正确处理，之后网络立刻中断，由于接收

方的 TCP 协议栈发送的确认数据包始终被丢失，发送方的 TCP 协议栈也有可能告知发送进程发送失败。另一方面，TCP 协议只能保证同一个 TCP 链接内的网络消息不乱序，TCP 链接之间的网络消息顺序则无法保证。但在分布式系统中，一个节点向另一个节点发送数据，有可能是先后使用多个 TCP 链接发送，也有可能是同时并发多个 TCP 链接发送，那么发送进程不能认为先调用 TCP 系统调用发送的消息就一定会先于后发送的消息到达对方节点并被处理。

由上述分析，在设计分布系统的网络协议时即使使用 TCP 协议，也依旧要考虑网络异常，不能简单的认为使用TCP 协议后通信就是可靠的。另一方面，如果完全放弃使用 TCP 协议，使用 UDP 协议加自定义的传输控制机制，则会使得系统设计复杂。尤其是要设计、实现一个像 TCP 那样优秀的拥塞控制机制是非常困难的。

2.1.4.7 分布式系统的三态

由于网络异常的存在，分布式系统中请求结果存在“三态”的概念。在单机系统中，我们调用一个函数实现一个功能，则这个函数要么成功、要么失败，只要不发生宕机其执行的结果是确定的。然而在分布式系统中，如果某个节点向另一个节点发起RPC(Remote procedure call)调用，即某个节点 A 向另一个节点 B 发送一个消息，节点 B 根据收到的消息内容完成某些操作，并将操作的结果通过另一个消息返回给节点 A，那么这个 RPC 执行的结果有三种状态：“成功”、“失败”、“超时（未知）”，称之为分布式系统的三态。

如果请求RPC 的节点A 收到了执行RPC 的节点B 返回的消息，并且消息中说明执行成功，则该 RPC 的结果为“成功”。如果请求 RPC 的节点 A 收到了执行 RPC 的节点 B 返回的消息，并且消息中说明执行失败，则该 RPC 的结果为“失败”。但是，如果请求 RPC 的节点A 在给定的时间内没有收到执行RPC 的节点B 返回的消息，则认为该操作“超时”。对于超时的请求，我们无法获知该请求是否被节点 B 成功执行了。这是因为，如果超时是由于节点 A 发向节点 B 的请求消息丢失造成的，则该操作肯定没有被节点 B 成功执行；但如果节点 A 成功的向节点 B 发送了请求消息，且节点 B 也成功的执行了该请求，但节点 B 发向节点 A 的结果消息被网络丢失了或者节点 B 在执行完该操作后立刻宕机没有能够发出结果消息，从而造成从节点A 看来请求超时。所以一旦发生超时， 请求方是无法获知RPC 的执行结果的。

一个非常易于理解的例子是在网上银行进行转账操作，当系统超时，页面提示：“如果系统长时间未返回，请检查账户余额以确认交易是否成功”。

分布式系统一般需要区别对待RPC 的“成功”、“失败”、“超时”三种状态。当出现“超时”时可以通过发起读取数据的操作以验证 RPC 是否成功（例如银行系统的做法）。另一种简单的做法是， 设计分布式协议时将执行步骤设计为可重试的，即具有所谓的“幂等性”。例如覆盖写就是一种常见的幂等性操作，因为重复的覆盖写最终的结果都相等。如果使用可重试的设计，当出现“失败”和“超时”时，一律重试操作直到“成功”。这样，即使超时的操作实际上已经成功了，重试操作也不会对正确性造成影响，从而简化了设计。

后续本书中，如果说明“不成功”即指“失败”或“超时”两种状态之一。如果说明“失败” 则表示收到了明确的“失败”消息。

2.1.4.8 存储数据丢失

数据丢失指节点存储的数据不可被读取或读取出的数据错误。数据丢失是另一类常见的异常。尤其是使用机械硬盘做存储介质时，硬盘损坏的概率较大。对于有状态节点来说，数据丢失意味着状态丢失，通常只能从其他节点读取、恢复存储的状态。

2.1.4.9 无法归类的异常

在工程实践中，大量异常情况是无法预先可知的，更可恶的是这些异常往往是“半死不活”的状态。例如，磁盘故障会导致IO操作缓慢，从而有可能使得进程运行速度非常慢，“缓慢”是异常吗？如果慢的超过某种程度，则对系统会造成影响。更有甚者，几十秒进程非常慢甚至完全阻塞住，又几十秒后恢复了，如此交替。又例如网络不稳定时也会引起“半死不活”异常，例如网络发生严重拥塞时约等于网络不通，过一会儿又恢复，恢复后又拥塞，如此交替。对于这些极端的无法预先归类的异常，需要在具体的项目中，通过长期的工程实践调整应对。

2.1.4.10异常处理的原则

在设计、推导、验证分布式系统的协议、流程时，最重要的工作之一就是思考在执行流程的每个步骤时一旦发生各种异常的情况下系统的处理方式及造成的影响。

被大量工程实践所检验过的异常处理黄金原则是：任何在设计阶段考虑到的异常

情况一定会在系统实际运行中发生，但在系统实际运行遇到的异常却很有可能在设计时未能考虑，所以，除非需求指标允许，在系统设计时不能放过任何异常情况。

工程中常常容易出问题的一种思路是认为某种异常出现的概率非常小以至于可以忽略不计。这种思路的错误在于只注意到了单次异常事件发生的概率，而忽略了样本的大小。即使单次异常概率非常小，由于系统规模和运行时间的作用，事件样本将是一个非常大的值，从而使得异常事件实际发生的概率变大。

2.2 副本

3.2.1 副本的概念

副本（replica/copy）指在分布式系统中为数据或服务提供的冗余。对于数据副本指在不同的节点上持久化同一份数据，当出现某一个节点的存储的数据丢失时，可以从副本上读到数据。数据副本是分布式系统解决数据丢失异常的唯一手段。另一类副本是服务副本，指数个节点提供某种相同的服务，这种服务一般并不依赖于节点的本地存储，其所需数据一般来自其他节点。

3.2.2 副本一致性

分布式系统通过副本控制协议，使得从系统外部读取系统内部各个副本的数据在一定的约束条件下相同，称之为副本一致性(consistency)。副本一致性是针对分布式系统而言的，不是针对某一个副本而言。

依据一致性的强弱即约束条件的不同苛刻程度，副本一致性分为若干变种或者级别，本书挑选几种常见的一致性级别介绍：

强一致性(strong consistency)：任何时刻任何用户或节点都可以读到最近一次成功更新的副本数据。强一致性是程度最高的一致性要求，也是实践中最难以实现的一致性。

单调一致性(monotonic consistency)：任何时刻，任何用户一旦读到某个数据在某次更新后的值，这个用户不会再读到比这个值更旧的值。单调一致性是弱于强一致性却非常实用的一种一致性级别。因为通常来说，用户只关心从己方视角观察到的一致性，而不会关注其他用户的一致性情况。

会话一致性(session consistency)：任何用户在某一次会话内一旦读到某个数据在某

次更新后的值，这个用户在这次会话过程中不会再读到比这个值更旧的值。会话一致性通过引入会话的概念，在单调一致性的基础上进一步放松约束，会话一致性只保证单个用户单次会话内数据的单调修改，对于不同用户间的一致性和同一用户不同会话间的一致性没有保障。实践中有许多机制正好对应会话的 概念，例如 php 中的 session 概念。可以将数据版本号等信息保存在 session 中，读取数据时验证副本的版本号，只读取版本号大于等于 session 中版本号的副本，从而实现会话一致性。

最终一致性(eventual consistency)：最终一致性要求一旦更新成功，各个副本上的数据最终将达到完全一致的状态，但达到完全一致状态所需要的时间不能保障。对于最终一致性系统而言，一个用户只要始终读取某一个副本的数据，则可以实现类似单调一致性的效果，但一旦用户更换读取的副本，则无法保障任何一致性。

弱一致性(week consistency)：一旦某个更新成功，用户无法在一个确定时间内读到这次更新的值，且即使在某个副本上读到了新的值，也不能保证在其他副本上可以读到新的值。弱一致性系统一般很难在实际中使用，使用弱一致性系统需要应用方做更多的工作从而使得系统可用。

2.3 衡量分布式系统的指标

评价分布式系统有一些常用的指标。依据设计需求的不同，分布式系统对于这些指标也有不同的要求。

3.3.1 性能

无论是分布式系统还是单机系统，都会对性能(performance)有所要求。对于不同的系统，不同的服务，关注的性能不尽相同、甚至相互矛盾。常见的性能指标有：系统的吞吐能力，指系统在某一时间可以处理的数据总量，通常可以用系统每秒处理的总的数据量来衡量；系统的响应延迟，指系统完成某一功能需要使用的时间；系统的并发能力，指系统可以同时完成某一功能的能力，通常也用QPS(query per second)来衡量。上述三个性能指标往往会相互制约，追求高吞吐的系统，往往很难做到低延迟；系统平均响应时间较长时，也很难提高 QPS。

3.3.2 可用性

系统的可用性(availability)指系统在面对各种异常时可以正确提供服务的能力。系统的可用性可以用系统停服务的时间与正常服务的时间的比例来衡量，也可以用某功能的失败次数与成功次数的比例来衡量。可用性是分布式的重要指标，衡量了系统的鲁棒性，是系统容错能力的体现。

3.3.3 可扩展性

系统的可扩展性(scalability)指分布式系统通过扩展集群机器规模提高系统性能（吞吐、延迟、并发）、存储容量、计算能力的特性。可扩展性是分布式系统的特有性质。分布式系统的设计初衷就是利用集群多机的能力处理单机无法解决的问题。然而，完成某一具体任务的所需要的机器数目即集群规模取决于系统的性能和任务的要求。当任务的需求随着具体业务不断提高时，除了升级系统的性能，另一个做法就是通过增加机器的方式扩展系统的规模。好的分布式系统总在追求“线性扩展性”，也就是使得系统的某一指标可以随着集群中的机器数量线性增长。

3.3.4 一致性

分布式系统为了提高可用性，总是不可避免的使用副本的机制，从而引发副本一致性的问题。根据具体的业务需求的不同，分布式系统总是提供某种一致性模型，并基于此模型提供具体的服务。

习题二

1、分布式系统中的异常包括哪些？

2、简述分布式系统中副本的概念，副本的一致性包括哪几种？

3. 分布式系统一致性

区块链是一个典型的分布式系统，在设计上必然也要考虑分布式系统中的典型问题，而一致性一直是分布式系统的核心问题。一致性是指分布式系统中的多个服务节点，给定一列操作，在特定协议的保障下，使这些节点对外呈现的状态是一致的，即保证集群中所有服务节点中的数据完全相同并且能够对某个提案达成一致。

3.1 系统模型

了解一致性先从系统模型开始。系统模型是描述系统特性的一些假设，基于假设可以划分成不同的分布式系统，这些假设包括:

1、每个节点的计算能力以及其失效模式;

2、节点间通信的能力以及是否可能失效;

3、整个系统的属性如时序等。节点即为系统的物理机或者虚拟机，负责系统的存储和计算业务。

1) 网络模型

网络模型根据系统中的时序模式分为同步网络和异步网络。一般来说，时序只有两种模式:同步和异步。同步网络中所有节点的时钟误差存在上限，并且网络的传输时间有上限；异步网络则与同步网络相反：节点的时钟漂移无上限，消息的传输时间是任意长的，节点计算的速度也不可预料。另外早期分布式系统很少会考虑网络分区的问题，但是随着系统规模的快速扩大，网络的失效也变得常见起来。

2) 故障模型

在分布式系统中，故障可能发生在节点或者通信链路上，根据对系统故障行为的最弱限制到最强限制，故障类型可以大致分为以下4类：

(1) 拜占庭(Byzantine)故障：这是最难处理的情况，系统内会发生任意类型的错误，发生错误的节点被称为拜占庭节点，拜占庭节点不仅会出现硬件错误、宕机甚至会向其他节点发送错误消息。

(2) 崩溃-恢复(Crash-Recovery)故障：比拜占庭类故障多了一个限制，节点总是按照

程序逻辑执行，结果是正确的，但是不保证消息返回的时间。

(3)崩溃-遗漏(Crash-Omission)故障：比崩溃-恢复多一个非健忘的限制，即节点崩溃之前能把状态完整地保存在持久存储上，启动之后可以再次按照以前的状态继续执行和通信。

(4)崩溃-停止(Crash-Stop)故障：崩溃-停止是理想化的故障模型，一个节点出现故障后立即停止接收和发送所有消息，或者网络出现故障无法进行任何通信。以上介绍的是使用较为广泛的一种分类方法,他们的关系如图2所示。一致性算法据此也可大致分为拜占庭容错(Byzantine Fault Tolerance, BFT)算法和崩溃容错(Crash Fault Tolerance, CFT)算法。

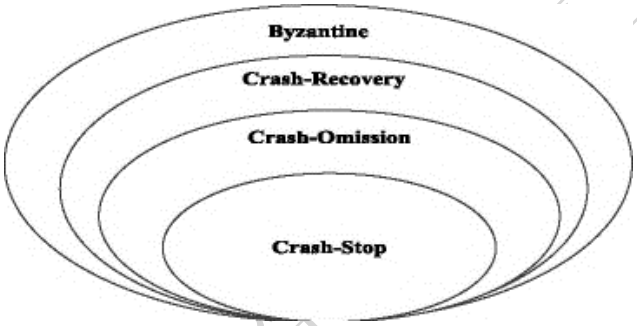

图3-1 故障类型的分类

3.2 拜占庭将军问题

1982年，LAMPORT问等人正式提出拜占庭将军问题。拜占庭是古代东罗马帝国的首都，由于地域宽广，守卫边境的多个将军需要通过信使来传递消息，达成作战与否等决定。但由于将军中可能存在叛徒，这些叛徒将向不同的将军发送不同的消息，试图干扰共识的达成。这种情况与分布式系统中多个节点达成共识的问题相似。拜占庭将军问题是基于现实世界对系统模型的最弱假设，目前普遍采用的假设条件包括：

1. 拜占庭节点的行为可以是任意的，节点之间可以共谋;

2. 节点之间的错误是不相关的且节点可以是异构的；

3. 节点之间通过异步网络连接，网络中的消息可能丢失、乱序到达、延时到达；

4. 节点之间传递的信息，第三方可以知晓但是不能窜改、伪造信息的内容和验证信

息的完整性。

拜占庭将军问题延伸到互联网生活中来，其内涵可概括为：在互联网大背景下，当需要与不熟悉的对方进行价值交换活动时，人们如何才能防止不会被其中的恶意破坏者欺骗、迷惑从而作出错误的决策。进一步将“拜占庭将军问题”延伸到技术领域中来，其内涵可概括为：在缺少可信任的中央节点和可信任的通道的情况下，分布在网络中的各个节点应如何达成共识。

拜占庭将军问题(The Byzantine Generals Problem)提供了对分布式共识问题的一种情景化描述。论文《The Byzantine Generals Problem 》同时提供了两种解决拜占庭将军问题的算法：

1) 口信消息型解决方案(A solution with oral message);

2) 签名消息型解决方案(A solution with signed message).

论文地址：

https://www-inst.eecs.berkeley.edu/~cs162/sp16/static/readings/Original_Byzantine.pdf

本书之后将详细讲述这两种算法。事实上，拜占庭将军问题是分布式系统领域最复杂的容错模型，它描述了如何在存在恶意行为(如消息篡改或伪造)的情况下使分布式系统达成一致。是我们理解分布式一致性协议和算法的重要基础。

3.2.1 问题描述

拜占庭将军问题描述了这样一个场景：

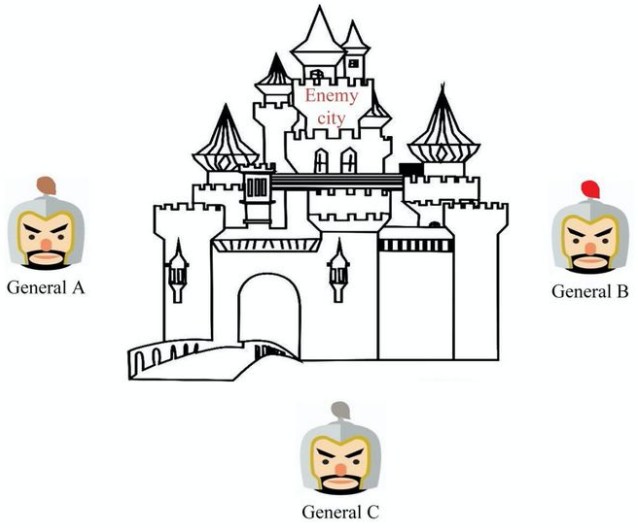

图 3-2 拜占庭将军问题

拜占庭帝国(Byzantine Empire)军队的几个师驻扎在敌城外，每个师都由各自的将军指挥。将军们只能通过信使相互沟通。在观察敌情之后，他们必须制定一个共同的行动计划，如进攻(Attack)或者撤退(Retreat)，且只有当半数以上的将军共同发起进攻时才能取得胜利。然而, 其中一些将军可能是叛徒，试图阻止忠诚的将军达成一致的行动计划。更糟糕的是，负责消息传递的信使也可能是叛徒，他们可能篡改或伪造消息，也可能使得消息丢失。

为了更加深入的理解拜占庭将军问题，我们以三将军问题为例进行说明。当三个将军都忠诚时，可以通过投票确定一致的行动方案，图3-3展示了一种场景， 即General A，B通过观察敌军军情并结合自身情况判断可以发起攻击，而General C通过观察敌军军情并结合自身情况判断应当撤退。最终三个将军经过投票表决得到结果为进攻：撤退=2:1， 所以将一同发起进攻取得胜利。对于三个将军，每个将军都能执行两种决策(进攻或撤退)的情况下, 共存在6中不同的场景，图3-3是其中一种，对于其他5中场景可简单地推得，通过投票三个将军都将达成一致的行动计划。

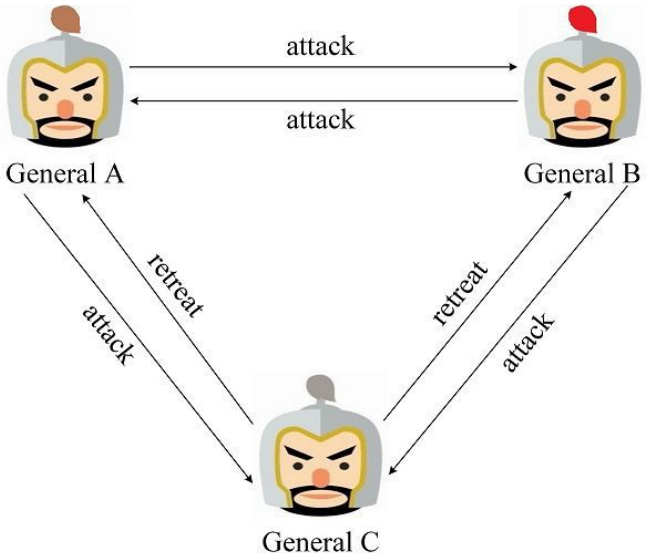

图 3-3 三个将军均为忠诚的场景

当三个将军中存在一个叛徒时，将可能扰乱正常的作战计划。图3-4展示了General C为叛徒的一种场景，他给General A和General B发送了不同的消息，在这种场景下General A通过投票得到进攻：撤退=1:2，最终将作出撤退的行动计划；General B通过投票得到进攻：撤退=2:1，最终将作出进攻的行动计划。结果只有General B发起了进攻并战败。

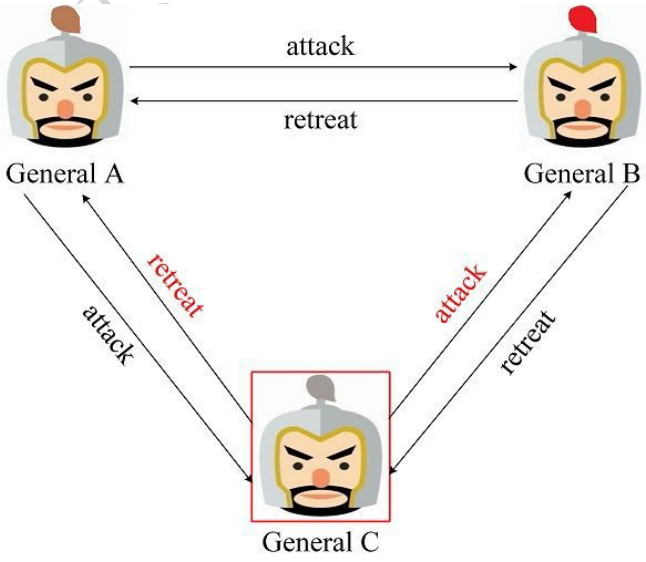

图 3-4 三个将军二忠一叛的场景

事实上，对于三个将军中存在一个叛徒的场景，想要总能达到一致的行动方案是不可能的。详细的证明可参看Leslie Lamport的论文。此外，论文中给出了一个更加普适的结论：如果存在m个叛将，那么至少需要3m+1个将军，才能最终达到一致的行动方案。

3.2.2 解决方案

Leslie Lamport在论文中给出了两种拜占庭将军问题的解决方案，即口信消息型解决方案(A solution with oral message)和签名消息型解决方案(A solution with signed message)。

1、口信消息型解决方案

首先，对于口信消息(Oral message)的定义如下：

A1. 任何已经发送的消息都将被正确传达;

A2. 消息的接收者知道是谁发送了消息;

A3. 消息的缺席可以被检测。

基于口信消息的定义，我们可以知，口信消息不能被篡改但是可以被伪造。基于对图3-3场景的推导，我们知道存在一个叛将时，必须再增加3个忠将才能达到最终的行动一致。为加深理解，我们将利用3个忠将1个叛将的场景对口信消息型解决方案进行推导。在口信消息型解决方案中，首先发送消息的将军称为指挥官，其余将军称为副官。对于3忠1叛的场景需要进行两轮作战信息协商，如果没有收到作战信息那么默认撤退。图3-5是指挥官为忠将的场景，在第一轮作战信息协商中，指挥官向3位副官发送了进攻的消息；在第二轮中，三位副官再次进行作战信息协商，由于General A、B为忠将，因此他们根据指挥官的消息向另外两位副官发送了进攻的消息，而General C为叛将，为了扰乱作战计划，他向另外两位副官发送了撤退的消息。最终Commanding General, General A和B达成了一致的进攻计划，可以取得胜利。

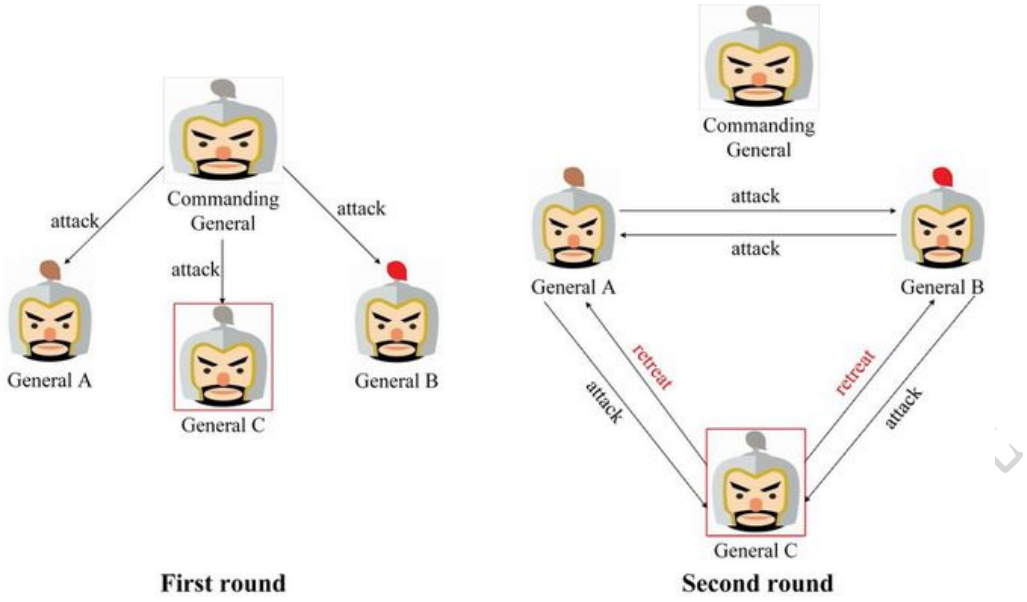

图 3-5 指挥官为忠将的场景

图3-6是指挥官为叛将的场景，在第一轮作战信息协商中，指挥官向General A、B发送了撤退的消息，但是为了扰乱General C的决定向其发送了进攻的消息。在第二轮中，由于所有副官均为忠将，因此都将来自指挥官的消息正确地发送给其余两位副官。最终所有忠将都能达成一致撤退的计划。

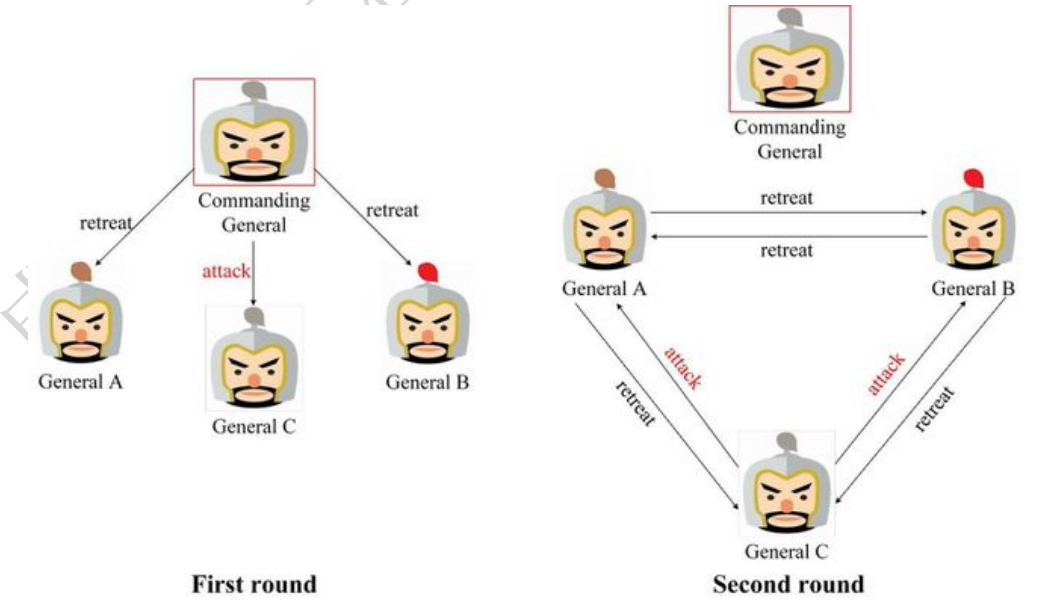

图 3-6 指挥官为叛将的场景

如上所述，对于口信消息型拜占庭将军问题，如果叛将人数为m，将军人数不少于3m+1，那么最终能达成一致的行动计划。值的注意的是，在这个算法中，叛将人数m是已知的，且叛将人数m决定了递归的次数，即叛将数m决定了进行作战信息协商的轮数，如果存在m个叛将，则需要进行m+1轮作战信息协商。这也是上述存在1个叛将时需要进行两轮作战信息协商的原因。

2、签名消息型解决方案

同样，对签名消息的定义是在口信消息定义的基础上增加了如下两条：

A4. 忠诚将军的签名无法伪造，而且对他签名消息的内容进行任何更改都会被发现；

A5. 任何人都能验证将军签名的真伪。

基于签名消息的定义，我们可以知道，签名消息无法被伪造或者篡改。为了深入理解签名消息型解决方案，我们同样以3三将军问题为例进行推导。 图3-7是忠将率先发起作战协商的场景，General A率先向General B、C发送了进攻消息，一旦叛将General C篡改了来自General A的消息，那么General B将将发现作战信息被General C篡改，General B将执行General A发送的消息。

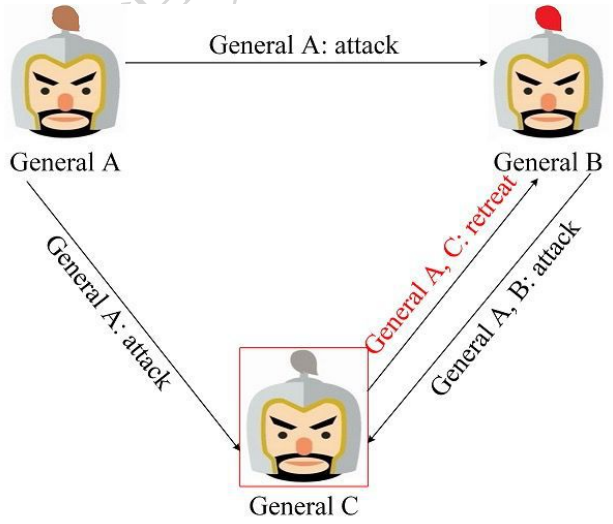

图 3-7 忠将率先发起作战协商

图3-8是叛将率先发起作战协商的场景，叛将General C率先发送了误导的作战信息，那么General A、B将发现General C发送的作战信息不一致，因此判定其为叛将。可对

其进行处理后再进行作战信息协商。签名消息型解决方案可以处理任何数量叛将的场景。

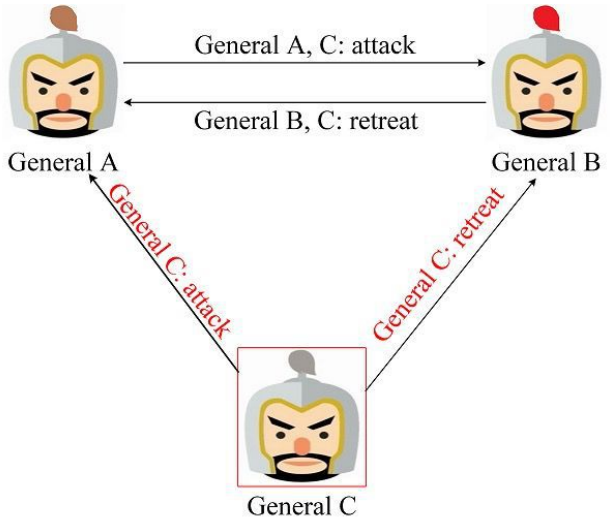

图 3-8 叛将率先发起作战协商

3.2.3 总结

在分布式系统领域, 拜占庭将军问题中的角色与计算机世界的对应关系如下:

1) 将军，对应计算机节点；

2) 忠诚的将军，对应运行良好的计算机节点；

3) 叛变的将军，被非法控制的计算机节点；

4) 信使被杀，通信故障使得消息丢失；

5) 信使被间谍替换，通信被攻击，攻击者篡改或伪造信息。

如上文所述，拜占庭将军问题提供了对分布式共识问题的一种情景化描述，是分布式系统领域最复杂的模型。

3.3 一致性重要定理

3.3.1 FLP定理

1. FLP impossibility背景

FLP Impossibility（FLP不可能性）是分布式领域中一个非常著名的结果，该结果在专业领域被称为“定理”，其地位之高可见一斑。该定理的论文是由Fischer, Lynch and Patterson三位作者于1985年发表,之后该论文毫无疑问得获得了Dijkstra奖。FLP给出了一个的结论：在异步通信场景，即使只有一个进程失败，也没有任何算法能保证非失败进程达到一致性。

因为同步通信中的一致性被证明是可以达到的，因此在之前一直有人尝试各种算法解决以异步环境的一致性问题。

FLP证明最难理解的是没有一个直观的例子，所有提到FLP的资料中也基本都回避了例子的要求。究其原因，例子难以设计，除非你先设计几种一致性算法，并用FLP说明这些算法都是错误的。

2. 系统模型

任何分布式算法或定理，都对系统场景进行了假设，这称为系统模型。FLP基于下面几点假设：

1）异步通信

与同步通信的最大区别是没有时钟、不能时间同步、不能使用超时、不能探测失败、消息可任意延迟、消息可乱序。

2）通信健壮

只要进程非失败，消息虽会被无限延迟，但最终会被送达；并且消息仅会被送达一次（无重复）

3）fail-stop模型

进程失败如同宕机，不再处理任何消息。相对Byzantine模型，不会产生错误消息

4）失败进程数量

最多一个进程失败。

在现实中，我们都使用TCP协议（保证了消息健壮、不重复、不乱序），每个节点

都有NTP时钟同步（可以使用超时），纯粹的异步场景相对比较少。但随着只能终端的发展，每个手机会为省电而关机，也会因为不在服务区而离线，这样的适用场景还是存在。

我们在衡量一个分布式算法是否正确时有三个标准：

1) Termination（终止性）：非失败进程最终可以做出选择。

2) Agreement（一致性）：所有的进程必须做出相同的决议。

3) Validity（合法性）：进程的决议值，必须是其他进程提交的请求值。

终止性，描述了算法必须在有限时间内结束，不能无限循环下去；一致性描述了我们期望的相同决议；合法性是为了排除进程初始值对自身的干扰。

3. 案例说明

假设有A、B、C、D、E五个进程就是否提交事务为例，每个进程都有一个随机的初始值提交（0）或回滚（1）来向其他进程发送请求，进程自己必须接收到其他进程的请求后才能根据请求内容作出本地是提交还是回滚的决定，不能仅根据自己的初始值做出决定。如果所有的进程都做出相同的决定，则认为一致性达成（Validity属性）；根据前面的系统模型，允许最多一个进程失败，因此一致性要求要放松到允许非失败进程达成一致。当然，若有两个不同的值被不同的进程选择，则认为无法达成一致。

现在目标是要设计这样一个算法，保证符合上述三个属性，并允许最多一个进程失败。

假如我们设计一个算法P，每个节点根据多数派表决的方式判断本地是提交还是回滚：

假如C收到了A、B的提交申请，收到了D的回滚申请，而C本身也倾向于回滚。此时，提交、回滚各有两票，E的投票决定着C的最终决议。而此时E失败了，或者E发送给C的消息被无限延迟（无法探测失败），此时C选择一直等待，或者按照某种既定的规则选择提交或失败，后续可能E正常而C失败，总之，导致C没有做出最终决策，或者C做了最终决策失败后无人可知。称所有进程组成的状态为Configuration，如果一系列操作之后，没有进程做出决策称为“不确定的”Configuration；不确定Configuration的

意思是，后续可能做出提交，也可能做出回滚的决议。

相反，如果某个Configuration能准确地说会做出提交/回滚的决议，则称为“确定性”的Configuration。如果某个Configuration是确定的，则认为一致性是可以达成。

对上述算法P，可能存在一种极端场景，每次都构造出一个“不确定”的Configuration，比如每次都是已经做出决议的C失败，而之前失败的E复活（在异步场景中，无法真正区分进程是失败，还是消息延迟），也就是说，因为消息被延迟乱序，导致结果难以预料！

而FLP证明也是遵循这个思路，在任何算法之上，都能构造出这样一些永远都不确定的Configuration，也就没有任何理论上的具体的算法，能避免这种最坏情况。

3.3.2 CAP定理

CAP 理论是由Eric Brewer提出的分布式系统中最为重要的理论之一。CAP 理论的定义很简单，CAP 三个字母分别代表了分布式系统中三个相互矛盾的属性：

Consistency (一致性)：CAP 理论中的副本一致性特指强一致性；

Availability(可用性)：指系统在出现异常时已经可以提供服务；

Tolerance to the partition of network (分区容忍)：指系统可以对网络分区这种异常情况进行容错处理；

CAP理论指出：无法设计一种分布式协议，使得同时完全具备 CAP 三个属性，即1)该种协议下的副本始终是强一致性，2)服务始终是可用的，3)协议可以容忍任何网络分区异常；分布式系统协议只能在CAP这三者间有所折中。

CAP理论的详细证明可以参考相关论文。这里可以简单用一个反例证明不存在CAP兼具的系统。假设系统只有两个副本A和B，Client更新这两个副本，假设在网络分化时，Client与副本A可以正常通信，但副本B与Client、副本B与副本A无法通信，此时，Client对副本A更新的信息永远无法同步到副本B上。如果希望系统依旧具有强一致的属性，则此时需要停止更新服务，即不再修改数据，从而让副本A与副本B保持一致；如果希望系统依旧可以提供更新服务，则只能更新副本A而无法更新副本B，此时无法

保证副本A与副本B一致。

CAP定理表明虽然不能使某个分布式场景同时满足三个性质，但可以使之同时满足三个中的两个。更进一步说，对于包含多个分布式场景的分布式系统，可以在满足三个性质的程度上进行适当的权衡。基于CAP定理，需要根据不同场景的不同业务要求来进行算法上的权衡。对于分布式存储系统来说，网络连接故障是无法避免的，在设计系统时不得不考虑分区容忍性，实际上只能在一致性和可用性之间进行权衡。用一致性模型来描述一致性强弱分为强一致性模型和弱一致性模型，其中弱一致性模型主要考虑最终一致性，最终一致性是系统保证最终所有的访问将返回这个对象的最后更新值。

习题三

1、什么是拜占庭将军问题？问题的本质如何理解？

2、简述FLP和CAP定理，并说明你对他们的认识。

4. 分布式一致性算法

分布式系统是一门理论模型和工程实践并重的学科，当分布式的思想被提出时，研究者们开始根据FLP不可能性定理和CAP定理等设计分布式一致性算法，算法还需满足分布式系统中对安全性（Safety）和活性（Liveness）的要求。

4.1 两阶段提交协议

4.1.1 简介

两阶段提交（two phase commit）协议是一种历史悠久的分布式控制协议。最早用于在分布式数据库中，实现分布式事务。这里有必要首先简单介绍一下两阶段提交的最初问题背景，从而能更好的理解该协议。

在经典的分布式数据库模型中，同一个数据库的各个副本运行在不同的节点上，每个副本的数据要求完全一致。数据库中的操作都是事务(transaction)，一个事务是一系列读、写操作，事务满足ACID（原子性：Atomicity、一致性：Consistency、隔离性：Isolation、持久性：Durability）。每个事务的最终状态要么是提交(commit)，要么是失败(abort)。一旦一个事务成功提交，那么这个事务中所有的写操作中成功，否则所有的写操作都失败。在单机上，事务靠日志技术或 MVCC 等技术实现。在分布式数据库中，需要有一种控制协议，使得事务要么在所有的副本上都提交，要么在所有的副本上都失败。对同一个事务而言，虽然在所有副本上执行的事务操作都完全一样，但可能在某些副本上可以提交，在某些副本上不能提交。这是因为，在某些副本上，其他的事务可能与本事务有冲突（例如死锁），从而造成在有些副本上事务可以提交，而有些副本上事务无法提交。本书不再深入讨论事务冲突的问题，只是将问题背景介绍清楚，该类问题可以通过阅读经典的数据库系统相关资料了解。

4.1.2 流程描述

在该协议中，参与的节点分为两类：一个中心化协调者节点（coordinator）和 N 个参与者节点（participant）。每个参与者节点即上文背景介绍中的管理数据库副本的节点。

两阶段提交的思路比较简单，在第一阶段，协调者询问所有的参与者是否可以提交事务（请参与者投票），所有参与者向协调者投票。在第二阶段，协调者根据所有参与者的投票结果做出是否事务可以全局提交的决定，并通知所有的参与者执行该决定。在一个两阶段提交流程中，参与者不能改变自己的投票结果。两阶段提交协议的可以全局提交的前提是所有的参与者都同意提交事务，只要有一个参与者投票选择放弃(abort)事务，则事务必须被放弃。

当分布式事务 T 执行完成，即事务执行的各节点都告知协调者 TC，事务已经执行完成，TC 便开启两阶段提交流程。

Phase 1 Prepare：

1.TC写本地日志，并持久化。TC向所有参与者发送Prepare T消息。

2.各参与者收到Prepare T消息，根据自身情况，决定是否提交事务。

如果决定提交，参与者写日志并持久化，向TC发送Ready T消息。如果决定不提交，参与者写日志并持久化，向TC发送Abort T消息，本地也进入事务abort流程。

Phase 2 Commit :

1.当TC收到所有节点的回应，或者等待超时，决定事务commit或abort。

如果所有参与者回应Ready T，则TC先写日志并持久化，再向所有参与者发送Commit T消息。如果收到至少一个参与者Abort T回应，或者在超时时间内有参与者未回应，则TC先写日志，再向所有参与者发送Abort T消息。

2.参与者收到TC的消息后，写或日志并持久化。

两阶段提交协议可以保证分布式事务执行的一个关键点：参与者在向协调者发生Ready T消息前，随时都可以自己决定是否abort，一旦这个消息发送，那么这个事务就进入ready状态，commit和abort完全由协调者控制。Ready T消息本质上是参与者向协调者发送的一个郑重的、不可逆的承诺。为了保证这一个承诺，参与者需要在发送Ready T消息前将所有必要的信息持久化，否则如果参与者在发送Ready T后异常宕机，重启后可能无法遵守以上承诺。在第二阶段，当协调者写了或日志，整个事务的命运就被决定了，不会再发生变化了。

为了优化2PC性能，减少关键路径的持久化和RPC次数是关键，一种对经典2PC的

优化思路如下：

协调者无状态，不再持久化日志，但是为了方便宕机重启后恢复事务状态，需要向每个参与者发送事务的参与者名单并持久化。这样即使协调者宕机，参与者也可以方便地询问其他参与者事务状态了。该思路相当于参与者在协调者宕机时，自己担当起协调者询问事务状态的任务。

只要所有参与者prepare成功，事务一定会成功提交。因此为了减少提交延时，协调者可以在收到所有参与者prepare成功后就返回客户端成功，但如此，读请求可能会因为提交未完成而等待，从而增大读请求的延时。反过来，如果协调者确认所有参与者都提交成功才返回客户端成功，提交延时比较长，但会减少读请求延时。

4.1.3 简介

1990年，Leslie Lamport 在论文《The Part-time Parliament》中提出了Paxos共识算法，在工程角度实现了一种最大化保障分布式系统一致性(存在极小的概率无法实现一致)的机制。Leslie Lamport作为分布式系统领域的早期研究者，因为相关成果获得了2013年度图灵奖。

Leslie Lamport在论文中将Paxos问题表述如下：

希腊岛屿Paxon上的执法者在议会大厅中表决通过法律（一次Paxos过程），并通过服务员（proposer）传递纸条的方式交流信息，每个执法者会将通过的法律记录在自己的账目上。问题在于执法者和服务员都不可靠，他们随时会因为各种事情离开议会大厅（服务器宕机或网络断开），并随时可能有新的执法者进入议会大厅进行法律表决（新加入机器），使用何种方式能够使得表决过程正常进行，且通过的法律不发生矛盾（对一个值达成一致）。

Paxos过程中不存在拜占庭将军问题（消息不会被篡改）和两将军问题（信道可靠）。Paxos是首个得到证明并被广泛应用的共识算法，其原理类似两阶段提交算法，进行了泛化和扩展，通过消息传递来逐步消除系统中的不确定状态。

作为后续很多共识算法(如 Raft、ZAB等)的基础，Paxos算法基本思想并不复杂，但最初论文中描述比较难懂，甚至连发表也几经波折。

2001年，Leslie Lamport专门发表论文《Paxos Made Simple》进行重新解释，其对

Paxos算法的描述如下：

Phase1:

(a) A proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors.

(b) If an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already

responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with

the highest-numbered pro-posal (if any) that it has accepted.

Phase 2

(a) If the proposer receives a response to its prepare requests (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v , where v is the value of the highest numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

Paxos算法目前在Google的Chubby、MegaStore、Spanner等系统中得到了应用，Hadoop中的ZooKeeper也使用了Paxos算法，但使用的算法是原始Paxos算法的改进算法。通常以Paxos算法为基础，在实现过程中处理实际应用场景中的具体细节，可以得到一个Paxos改进算法。

4.1.4 异常处理

两阶段提交协议的正常流程较为简单，但它还需要考虑分布式系统中各种异常问题（节点失败，网络分区等）。

1. 如果协调者检测到参与者失败。

如果参与者在发送Ready T前失败，则协调者认为该节点事务Abort，并开始abort流程。如果参与者在发送Ready T后失败，证明参与者本地事务已经持久化，协调者忽视参与者失败，继续事务流程。

2. 如果参与者在事务提交过程中失败，其恢复过程，需要根据参与者日志内容，

决定本地事务状态。

如果日志中包含日志，证明事务已经成功提交，则执行REDO(T)。如果日志中包含日志，证明事务已经失败，则执行UNDO(T)。如果日志中包含日志，参与者P需向其它节点咨询当前事务状态。如果协调者正常，则向告知参与者P，事务已经commit或是abort，参与者依此执行REDO(T)或UNDO(T)。如果协调者异常，则向其它参与者询问事务状态。如果其他参与者收到信息，并已知事务是commit还是abort状态，需回复参与者P事务状态。如果所有的参与者现在都不知道该事务的状态（事务上下文销毁了，或者自己也处于未决状态），那么该事务处于暂时既不能commit也不能abort。需要定期向其它节点问询事务状态，直到得到答案。（这是2PC最不想遇到的一个场景）如果日志中不包含上述几种日志，说明该参与者在向协调者发送Ready T消息前就失败了。由于协调者没有收到参与者的回应，会超时Abort，因此该参与者在恢复过程中，遇到这种情况也需要abort。

3. 如果协调者在事务提交过程中失败。参与者需要根据全局事务状态（通过与其它参与者通信）决定本地行为。

事务状态已经形成决议：

如果至少有一个参与者中事务T已经提交（参与者包含日志），说明T必须要提交。如果至少有一个参与者中事务T已经Abort（参与者包含日志），说明T必须要Abort。

事务状态未形成决议：

如果至少有一个参与者没有进入Ready状态（参与者不包含日志）。说明全局还未就提交与否达成协议。有两种选择：（1）等待协调者恢复。（2）参与者自行abort。为了减少资源占用时间，选择后者居多。如果所有参与者都进入了Ready状态，且都没有或日志（事实上，即使有这些日志，查日志也是一种比较费的操作，还需要考虑日志回收的问题），这种情况下，参与者谁都不知道现在事务的状态，只能死等协调者恢复。（又到了这个最不想遇到的场景）当参与者均进入ready状态，等待协调者的下一步指令，协调者在这个时候出现异常，那么参与者将一直持有系统资源，如果基于锁实现的并发控制，还会一直持有锁，导致其他事务等待。这种情况如果持续较旧，会对系统产生巨大的影响。

因此2PC最大的问题就是协调者失败，可能会导致事务阻塞，未决事务的最终状态，只能等待协调者恢复后才确定。同时在这种情况下，参与者宕机重启，回放到这类未决事务，也会因为死等而进入block recovery流程。

4.2 Paxos 协议

4.2.1 算法原理

Paxos算法的基本思路类似两阶段提交：多个提案者先要争取到提案的权利(得到大多数接受者的支持)，成功的提案者发送提案给所有人进行确认，得到大部分人确认的提案成为批准的结案。Paxos协议有三种角色：Proposer（提议者），Acceptor（决策者），Learner（决策学习者）。

Paxos 是一个两阶段的通信协议，Paxos算法的基本流程如下：

第一阶段Prepare：

A、Proposer生成一个全局唯一的提案编号N，然后向所有Acceptor发送编号为N的Prepare请求。

B、如果一个Acceptor收到一个编号为N的Prepare请求，且N大于本Acceptor已经响应过的所有Prepare请求的编号，那么本Acceptor就会将其已经接受过的编号最大的提案（如果有的话）作为响应反馈给Proposer，同时本Acceptor承诺不再接受任何编号小于N的提案。

第二阶段 Accept:

A、如果Proposer收到半数以上Acceptor对其发出的编号为N的Prepare请求的响应，那么Proposer就会发送一个针对[N,V]提案的Accept请求给半数以上的Acceptor。V就是收到的响应中编号最大的提案的value，如果响应中不包含任何提案，那么V由Proposer自己决定。

B、如果Acceptor收到一个针对编号为N的提案的Accept请求，只要该Acceptor没有对编号大于N的Prepare请求做出过响应，Acceptor就接受该提案。

Paxos并不保证系统总处在一致的状态。但由于每次达成共识至少有超过一半的节

点参与，最终整个系统都会获知共识结果。如果提案者在提案过程中出现故障，可以通过超时机制来缓解。Paxos能保证在超过一半的节点正常工作时，系统总能以较大概率达成共识。

4.2.2 协议描述

Paxos 协议中，有三类节点：

Proposer：提案者。Proposer 可以有多个，Proposer 提出议案（Value）。所谓Value在工程中可以是任何操作，例如“修改某个变量的值为某个值”、“设置当前primary 为某个节点”等等。Paxos 协议中统一将这些操作抽象为 Value。不同的Proposer 可以提出不同的甚至矛盾的 Value，例如某个Proposer 提议“将变量 X 设置为1”，另一个 Proposer 提议“将变量 X 设置为 2”，但对同一轮 Paxos过程，最多只有一个Value被批准。

Acceptor：批准者。Acceptor有M个，Proposer提出的Value必须获得超过半数(M/2+1)的 Acceptor批准后才能通过。Acceptor 之间完全对等独立。

Learner：学习者。Learner学习被批准的 value。所谓学习就是通过读取各个Acceptor对 Value的选择结果，如果某个 value 被超过半数Acceptor通过，则 Learner 学习到了这个 value。某个Value需要获得W=M/2 + 1 的Acceptor 批准，从而学习者需要至少读取M/2+1个Accpetor，至多读取M个Acceptor 的结果后，能学习到一个通过的Value。

上述三类角色只是逻辑上的划分，实践中一个节点可以同时充当这三类角色。

流程描述:

Paxos 协议一轮一轮的进行，每轮都有一个编号。每轮 Paxos协议可能会批准一个value，也可能无法批准一个 value。如果某一轮 Paxos 协议批准了某个value，则以后各轮 Paxos 只能批准这个value。上述各轮协议流程组成了一个 Paxos 协议实例，即一次Paxos 协议实例只能批准一个 value， 这也是Paxos 协议强一致性的重要体现。

每轮 Paxos 协议分为阶段：准备阶段和批准阶段，在这两个阶段Proposer 和Acceptor 有各自的处理流程。

Proposer流程-准备阶段：

1.向所有的Acceptor 发送消息“Prepare(b)”。这里b是 Paxos 的轮数，每轮递增。

2.如果收到任何一个 Acceptor 发送的消息“Reject(b)”，则对于这个Proposer而言本轮Paxos失败，将轮数b设置为 b+1 后重新步骤 1；

Proposer流程-批准阶段：

3.如果接收到的 Acceptor 的“Promise(b, v_i)”消息达到 M/2+1 个（M 为 Acceptor 总数，除法取整，下同）。v_i表示Acceptor 最近一次在 i 轮批准过 Value V。

3.1 如果收到的“Promise(b, v)”消息中，V都为空，Proposer 选择一个 value V，向所有 Acceptor广播Accept(b, v)；

3.2 否则在所有收到的“Promise(b, v_i)”消息中，选择 i 最大的 value V，向所有 Acceptor 广播消息 Accept(b, v)；

4.如果收到Nack(B)(否定应答)，将轮数B设置为B+1 后重新步骤 1；

Acceptor流程-准备阶段：

1.接受某个Propeser 的消息 Prepare(b)。

1.1 如果 b>B，回复Promise(b, v_B)，设置 B=b; 表示保证不再接受编号小于b的提案。参数B是该 Acceptor 收到的最大Paxos 轮数编号；V 是Acceptor 批准的 value，可以为空。

1.2 否则，回复Reject(B)

Acceptor流程-批准阶段：

2.接收 Accept(b, v)。

2.1 如果 b < B, 回复Nack(B)，提示proposer有一个更大编号的提案被这个Acceptor接收。

2.2 否则设置 V=v。表示这个Acceptor 批准的 Value 是 v，并广播Accepted 消息。

4.2.3 实例

理解 Paxos 协议的最直观的方法就是考察几个协议运行的实例。本书给出几个典型的场景下协议运行的例子。

1) 基本的例子：

基本例子里有 5 个 Acceptor，1 个 Proposer，不存在任何网络、宕机异常。我们着重考察各个Accpetor 上变量B 和变量V 的变化，及Proposer 上变量 b 的变化。

1. 初始状态

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	0	0	0	0	0
V	NULL	NULL	NULL	NULL	NULL

	Proposer 1
b	1

2. Proposer 向所有Accpetor 发送“Prepare(1)”，所有 Acceptor 正确处理，并回复 Promise(1, NULL)

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	1	1
V	NULL	NULL	NULL	NULL	NULL

	Proposer 1
b	1

3. Proposer 收到 5 个 Promise(1, NULL)，满足多余半数的 Promise 的 value 为空，此时发送Accept(1, v_1)，其中 v_1 是Proposer 选择的 Value。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	1	1
V	v_1	v_1	v_1	v_1	v_1

4. 此时，v_1 被超过半数的Acceptor 批准，v_1 即是本次Paxos 协议实例批准的 Value。如果 Learner学习 value，学到的只能是 v_1 。

2) 批准的 Value 无法改变

在同一个Paxos 实例中，批准的 Value 是无法改变的，即使后续 Proposer 以更高的序号发起 Paxos协议也无法改变Value。

例如，某次 Paxos 协议运行后，Acceptor 的状态是：

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	3	3	3	2	2
V	v_1	v_1	v_1	NULL	NULL

5个Acceptor中，有3个已经在第三轮Paxos 协议批准了v_1作为 value。其他两个Acceptor的V为空，这可能是因为Proposer与这两个Acceptor 的网络中断或者这两个Acceptor 宕机造成的。

此时，即使有新的Proposer发起协议，也无法改变结果。假设 Proposer 发送“prepare(4)消息”，由于 4 大于所有的Accpetor 的B 值，所有收到 prepare 消息的 Acceptor 回复 promise 消息。但前三个 Acceptor 只能回复 promise(4, v_1_3)，后两个 Acceptor 回复 promise(4, NULL)。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	4	4	4	4	4
V	v_1	v_1	v_1	NULL	NULL

此时，Proposer 可能收到若干个 Acceptor 发送的 promise 消息，没有收到的 promise 消息可能是网络异常造成的。无论如何，Proposer 要收到至少 3 个 Acceptor 的 promise 消息后才满足协议中大于半数的约束，才能发送accpet消息。这3 个promise 消息中，至少有1个消息是promise(4, v_1_3)，至多3个消息都是promise(4，v_1_3)。另一方面，Proposer 始终不可能收到 3 个 promise(4, NULL)消息，最多收到 2 个。综上，按协议流程，Proposer 发送的 accept 消息只能是“accept(4, v_1)”，而不能自由选择 value。

无论这个 accept 消息是否被各个 Acceptor 接收到，都无法改变 v_1 是被批准的 value 这一事实。即从全局看，有且只有v_1是满足超过多数Acceptor批准的 value。例如，假设 accept(4, v_1)消息被Acceptor 1、Acceptor2、Acceptor4 收到，那么状态变为：

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	4	4	4	4	4
V	v_1	v_1	v_1	v_1	NULL

从这个例子我们可以看到一旦一个 value 被批准，此后永远只能批准这个 value。

3) 一种不可能出现的状态

Paxos 协议的核心就在与“批准的value无法改变”，这也是整个协议正确性的基础，为了更好的理解后续对 Paxos 协议的证明。这里再看一种看似可能，实际违反协议的

状态，这种状态也是后续反证法证明协议时的一种错误状态。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	2	2
V	v_1	v_1	v_1	v_2	v_2

上述状态中，3 个轮次为 1 的 Acceptor 的 value 为 v_1，2 个轮次更高的 Acceptor 的 value 为 v_2。此时被批准的value 是 v_1。

假设此时发生新的一轮 b=3 的 Paxos 过程，Proposer 有可能收到 Acceptor 3、4、5 发出的 3 个promise 消息分别为“promise(1, v_1_1)”,“promise(2, v_2_2)” “promise(2, v_2_2)”。按协议，proposer 选择 value 编号最大的 promise 消息，即 v_2_2 的 promise 消息，发送消息“Accept(3, v_2)”，从而使得最终的批准的value 成为 v_2。就使得批准的 value 从v_1 变成了 v_2。

上述假设看似正确，其实不可能发生。这是因为本节中给出的初始状态就是不可能出现的。这是因为，要到达成上述状态，发起 prepare(2)消息的 proposer 一定成功的向 Acceptor 4、Acceptor 5 发送了 accept(2, v_2)。但发送 accept(2, v_2)的前提只能是 proposer 收到了 3 个“promise(2, NULL)”消息。 然而，从状态我们知道，在 b=1 的那轮 Paxos 协议里，已经有 3 个Acceptor 批准了 v_1,这 3 个Acceptor 在b=2 时发出的消息只能是 promise(2，v_1_1)，从而造成 proposer 不可能收到 3 个“promise(2, NULL)”，至多只能收到 2 个“promise(2, NULL)”。另外，只要 proposer 收到一个“promise(2，v_1_1)”，其发送的 accept 消息只能是 accept(2, v_1)。

从这个例子我们看到 Prepare 流程中的第 3 步是协议中最为关键的一步，它的存在严格约束了“批准的 value 无法改变”这一事实。

4) 节点异常

这里给一个较为复杂的异常状态下Paxos 运行实例。本例子中有 5 个Acceptor 和 2 个Proposer。Proposer 1 发起第一轮 Paxos 协议，然而由于异常，只有 2 个 Acceptor 收到了 prepare(1)消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	0	0	0
V	NULL	NULL	NULL	NULL	NULL

Proposer 1 只收到 2 个 promise 消息，无法发起 accept 消息；此时，Proposer 2 发起第二轮 Paxos协议，由于异常，只有Acceptor 1、3、4 处理了 prepare 消息，并发送

promise(2, NULL)消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	2	1	2	2	0
V	NULL	NULL	NULL	NULL	NULL

Proposer 2 收到了 Acceptor 1、3、4 的 promise(2, NULL) 消息，满足协议超过半数的要求， 选择了 value 为 v1,广播了 accept(2, v_1)的消息。由于异常，只有Accptor 3、4 处理了这个消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	2	1	2	2	0
V	NULL	NULL	v_1	v_1	NULL

Proposer 1 以b=3 发起新一轮的Paxos 协议，由于异常，只有Acceptor 1、2、3、5 处理了prepare(3)消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	3	3	3	2	3
V	NULL	NULL	v_1	v_1	NULL

由于异常，Proposer 1 只收到 Acceptor1、2、5 的 promise(3, NULL)的消息，符合协议要求， Proposer 1 选择 value 为 v_2，广播 accept(3, v_2)消息。由于异常，这个消息只被 Acceptor 1、2 处理。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	3	3	3	2	3
V	v_2	v_2	v_1	v_1	NULL

当目前为止，没有任何value 被超过半数的Acceptor 批准，所以Paxos 协议尚没有批准任何value。然而由于没有 3 个NULL 的 Acceptor，此时能被批准的 value 只能是 v_1 或者 v_2 其中之一。

此时 Proposer 1 以b=4 发起新的一轮Paxos 协议，所有的 Acceptor 都处理了prepare(4)消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	4	4	4	4	4
V	v_2	v_2	v_1	v_1	NULL

由于异常，Proposer 1 只收到了 Acceptor3 的 promise(4, v_1_3)消息、Acceptor4 的 promise(4, v_1_2)、Acceptor5 的 promise(4, NULL)消息，按协议要求，只能广播 accept(4, v_1)消息。假设 Acceptor2、3、4 收到了 accept(4, v_1)消息。由于批准 v_1 的Acceptor 超过

半数，最终批准的value 为 v_1。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	4	4	4	4	4
V	v_2	v_1	v_1	v_1	NULL

4.2.4 竞争及活锁

从前面的例子不难看出，Paxos 协议的过程类似于“占坑”，哪个 value 把超过半数的“坑”（Acceptor）占住了，哪个 value 就得到批准了。

这个过程也类似于单机系统并行系统的加锁过程。假如有这么单机系统：系统内有 5 个锁，有多个线程执行，每个线程需要获得 5 个锁中的任意 3 个才能执行后续操作，操作完成后释放占用的锁。我们知道，上述单机系统中一定会发生“死锁”。例如，3 个线程并发，第一个线程获得 2 个锁， 第二个线程获得 2 个锁，第三个线程获得 1 个锁。此时任何一个线程都无法获得 3 个锁，也不会主动释放自己占用的锁，从而造成系统死锁。

但在 Paxos 协议过程中，虽然也存在着并发竞争，不会出现上述死锁。这是因为，Paxos 协议引入了轮数的概念，高轮数的 paxos 提案可以抢占低轮数的 paxos 提案，从而避免了死锁的发生。然而这种设计却引入了“活锁”的可能，即 Proposer 相互不断以更高的轮数提出议案，使得每轮 Paxos 过程都无法最终完成，从而无法批准任何一个 value。

Proposer 1 以 b=1 提起议案，发送 prepare(1)消息，各 Acceptor 都正确处理,回应 promise(1, NULL)

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	1	1
V	NULL	NULL	NULL	NULL	NULL

Proposer 2 以 b=2 提起议案，发送 prepare(2)消息，各 Acceptor 都正确处理,回应 promise(2, NULL)

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	2	2	2	2	2
V	NULL	NULL	NULL	NULL	NULL

Proposer 1 收到 5 个 promise(1, NULL)消息，选择value 为 v_1 发送 accept(1, v_1)消息，然而这个消息被所有的Acceptor 拒绝，收到 5 个 Nack(2)消息。

Proposer 1 以 b=3 提起议案，发送 prepare(3)消息，各 Acceptor 都正确处理,回应

promise(3, NULL)

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	3	3	3	3	3
V	NULL	NULL	NULL	NULL	NULL

Proposer 2 收到 5 个 promise(2, NULL)消息，选择value 为 v_2 发送 accept(2, v_2)消息，然而这个消息被所有的Acceptor 拒绝，收到 5 个 Nack(3)消息。

上述过程交替进行，则永远无法批准一个 value，从而形成 Paxos 协议活锁。Paxos 协议活锁问题也是这个协议的主要问题。

4.3 Raft算法

4.3.1 简介

1. 背景

在分布式系统中，一致性算法至关重要。在所有一致性算法中，Paxos 最负盛名，它由莱斯利·兰伯特（Leslie Lamport）于 1990 年提出，是一种基于消息传递的一致性算法，被认为是类似算法中最有效的。

Paxos 算法虽然很有效，但复杂的原理使它实现起来非常困难，截止目前，实现 Paxos 算法的开源软件很少，比较出名的有 Chubby、LibPaxos。此外，Zookeeper 采用的 ZAB（Zookeeper Atomic Broadcast）协议也是基于 Paxos 算法实现的，不过 ZAB 对 Paxos 进行了很多改进与优化，两者的设计目标也存在差异——ZAB 协议主要用于构建一个高可用的分布式数据主备系统，而 Paxos 算法则是用于构建一个分布式的一致性状态机系统。

由于 Paxos 算法过于复杂、实现困难，极大地制约了其应用，而分布式系统领域又亟需一种高效而易于实现的分布式一致性算法，在此背景下，Raft 算法应运而生。

Raft 算法在斯坦福 Diego Ongaro 和 John Ousterhout 于 2013 年发表的《In Search of an Understandable Consensus Algorithm》中提出。相较于 Paxos，Raft 通过逻辑分离使其更容易理解和实现，目前，已经有十多种语言的 Raft 算法实现框架，较为出名的有 etcd、Consul。

2. Raft角色

根据官方文档解释，一个 Raft 集群包含若干节点，Raft 把这些节点分为三种状态：

Leader、 Follower、Candidate，每种状态负责的任务也是不一样的。正常情况下，集群中的节点只存在 Leader 与 Follower 两种状态。

• **Leader（领导者）**：负责日志的同步管理，处理来自客户端的请求，与Follower保持HeartBeat的联系；

• **Follower（追随者）**：响应 Leader 的日志同步请求，响应Candidate的邀票请求，以及把客户端请求到Follower的事务转发（重定向）给Leader；

• **Candidate（候选者）**：负责选举投票，集群刚启动或者Leader宕机时，状态为Follower的节点将转为Candidate并发起选举，选举胜出（获得超过半数节点的投票）后，从Candidate转为Leader状态。

3. Raft 三个子问题

通常，Raft 集群中只有一个 Leader，其它节点都是 Follower。Follower 都是被动的，不会发送任何请求，只是简单地响应来自 Leader 或者 Candidate 的请求。Leader 负责处理所有的客户端请求（如果一个客户端和 Follower 联系，那么 Follower 会把请求重定向给 Leader）。

为简化逻辑和实现，Raft 将一致性问题分解成了三个相对独立的子问题。

• **选举（Leader Election）**：当 Leader 宕机或者集群初创时，一个新的 Leader 需要被选举出来；

• **日志复制（Log Replication）**：Leader 接收来自客户端的请求并将其以日志条目的形式复制到集群中的其它节点，并且强制要求其它节点的日志和自己保持一致；

• **安全性（Safety）**：如果有任何的服务器节点已经应用了一个确定的日志条目到它的状态机中，那么其它服务器节点不能在同一个日志索引位置应用一个不同的指令。

4.3.2 选举

根据 Raft 协议，一个应用 Raft 协议的集群在刚启动时，所有节点的状态都是 Follower。由于没有 Leader，Followers 无法与 Leader 保持心跳（HeartBeat，一般为RPC请求），因此，Followers 会认为 Leader 已经下线，进而转为 Candidate 状态。然后，Candidate 将向集群中其它节点请求投票，同意自己升级为 Leader。如果 Candidate 收到超过半数节点的投票（M/2 + 1），它将获胜成为 Leader。

第一阶段：所有节点都是 Follower

上面提到，一个应用 Raft 协议的集群在刚启动（或 Leader 宕机）时，所有节点的状态都是 Follower，初始 Term（任期）为 0。同时启动选举定时器，每个节点的选举定时器超时时间都在 100~500 毫秒之间且并不一致（避免同时发起选举）。

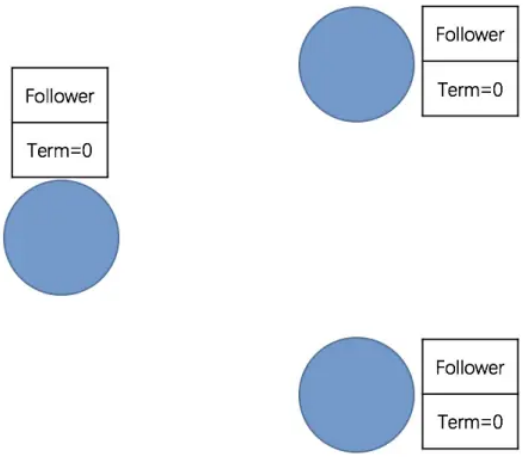

图4-1 所有节点都是 Follower

第二阶段：Follower 转为 Candidate 并发起投票。

没有 Leader，Followers 无法与 Leader 保持心跳（HeartBeat），节点启动后在一个选举定时器周期内未收到心跳和投票请求，则状态转为候选者 Candidate 状态，且 Term 自增，并向集群中所有节点发送投票请求并且重置选举定时器。

注意，由于每个节点的选举定时器超时时间都在 100-500 毫秒之间，且彼此不一样，以避免所有 Follower 同时转为 Candidate 并同时发起投票请求。换言之，最先转为 Candidate 并发起投票请求的节点将具有成为 Leader 的“先发优势”。

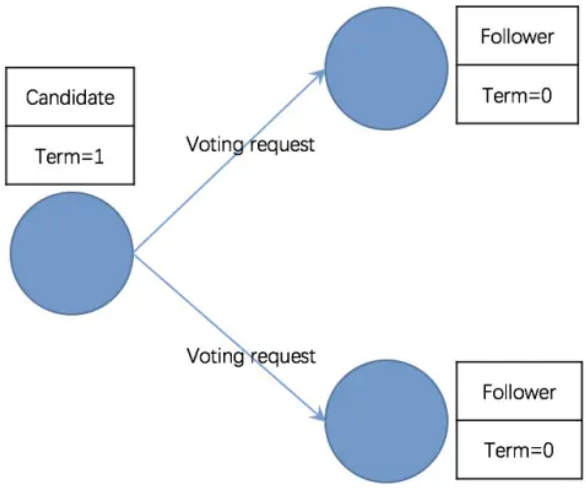

图4-2 Follower 转为 Candidate 并发起投票

第三阶段：投票策略

节点收到投票请求后会根据以下情况决定是否接受投票请求（每个 follower 刚成为 Candidate 的时候会将票投给自己）：

请求节点的 Term 大于自己的 Term，且自己尚未投票给其它节点，则接受请求，把票投给它；

请求节点的 Term 小于自己的 Term，且自己尚未投票，则拒绝请求，将票投给自己。

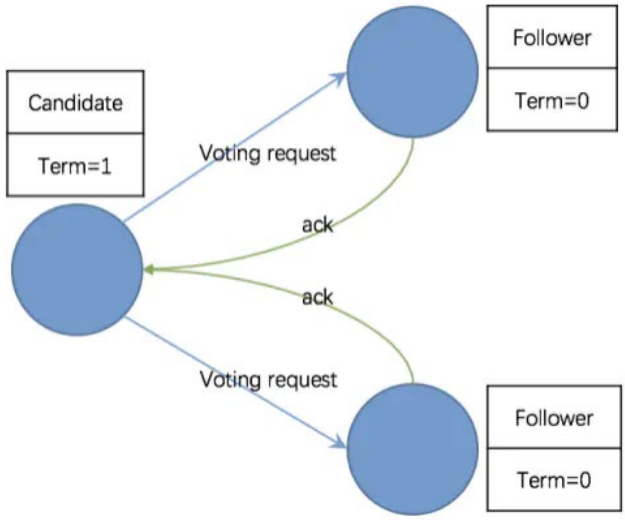

图4-3 投票策略

第四阶段：Candidate 转为 Leader

一轮选举过后，正常情况下，会有一个 Candidate 收到超过半数节点（M/2 + 1）的投票，它将胜出并升级为 Leader。然后定时发送心跳给其它的节点，其它节点会转为 Follower 并与 Leader 保持同步，到此，本轮选举结束。

注意：有可能一轮选举中，没有 Candidate 收到超过半数节点投票，那么将进行下一轮选举。

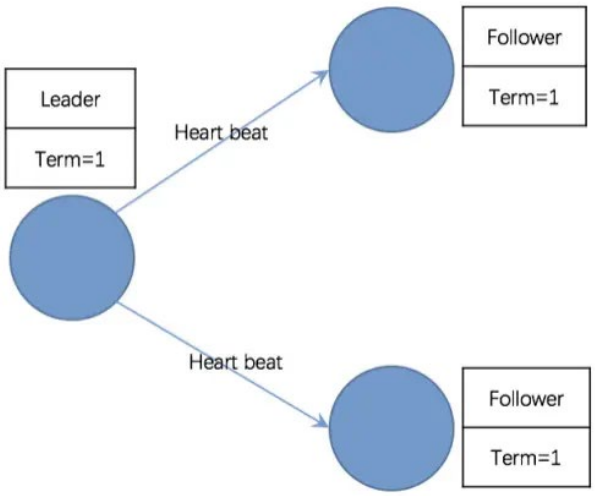

图4-4 Candidate 转为 Leader

4.3.3 日志复制

在一个 Raft 集群中，只有 Leader 节点能够处理客户端的请求（如果客户端的请求发到了 Follower，Follower 将会把请求重定向到 Leader），客户端的每一个请求都包含一条被复制状态机执行的指令。Leader 把这条指令作为一条新的日志条目（Entry）附加到日志中去，然后并行得将附加条目发送给 Followers，让它们复制这条日志条目。

当这条日志条目被 Followers 安全复制，Leader 会将这条日志条目应用到它的状态机中，然后把执行的结果返回给客户端。如果 Follower 崩溃或者运行缓慢，再或者网络丢包，Leader 会不断得重复尝试附加日志条目（尽管已经回复了客户端）直到所有的 Follower 都最终存储了所有的日志条目，确保强一致性。

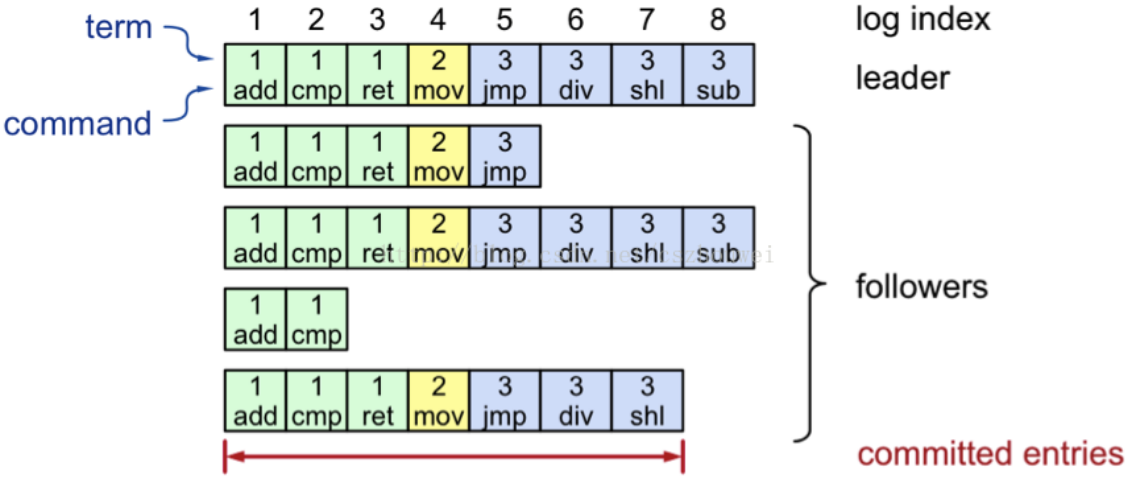

图4-5 Log结构

第一阶段：客户端请求提交到 Leader

如下图所示，Leader 收到客户端的请求，比如存储数据 5。Leader 在收到请求后，会将它作为日志条目（Entry）写入本地日志中。需要注意的是，此时该 Entry 的状态是未提交（Uncommitted），Leader 并不会更新本地数据，因此它是不可读的。

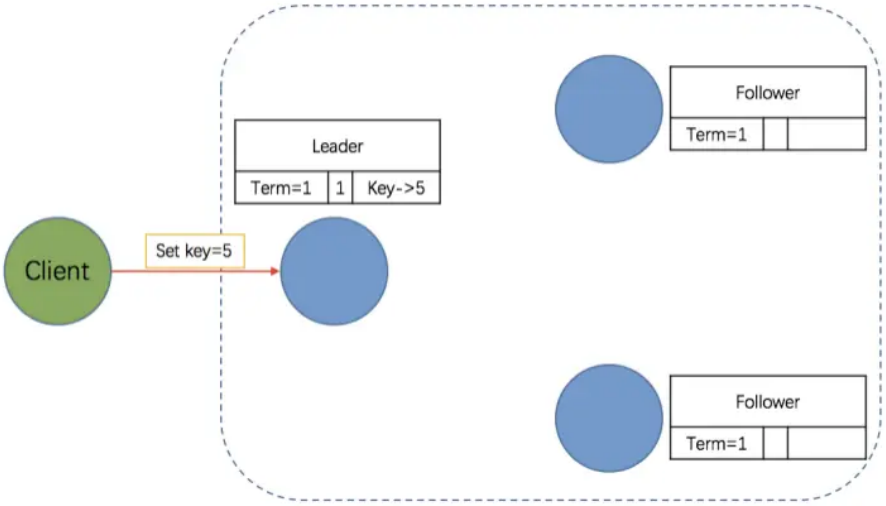

图4-6 客户端请求提交到 Leader

第二阶段：Leader 将 Entry 发送到其它 Follower

Leader 与 Followers 之间保持着心跳联系，随心跳 Leader 将追加的 Entry（AppendEntries）并行地发送给其它的 Follower，并让它们复制这条日志条目，这一过程称为复制（Replicate）。

有几点需要注意：

1. 为什么 Leader 向 Follower 发送的 Entry 是 AppendEntries 呢？

因为 Leader 与 Follower 的心跳是周期性的，而一个周期间 Leader 可能接收到多条客户端的请求，因此，随心跳向 Followers 发送的大概率是多个 Entry，即 AppendEntries。当然，在本例中，我们假设只有一条请求，自然也就是一个Entry了。

2. Leader 向 Followers 发送的不仅仅是追加的 Entry（AppendEntries）。

在发送追加日志条目的时候，Leader 会把新的日志条目紧接着之前条目的索引位置（prevLogIndex）， Leader 任期号（Term）也包含在其中。如果 Follower 在它的日志中找不到包含相同索引位置和任期号的条目，那么它就会拒绝接收新的日志条目，因为出现这种情况说明 Follower 和 Leader 不一致。

3. 如何解决 Leader 与 Follower 不一致的问题？

在正常情况下，Leader 和 Follower 的日志保持一致，所以追加日志的一致性检查从来不会失败。然而，Leader 和 Follower 一系列崩溃的情况会使它们的日志处于不一致状态。Follower可能会丢失一些在新的 Leader 中有的日志条目，它也可能拥有一些 Leader 没有的日志条目，或者两者都发生。丢失或者多出日志条目可能会持续多个任期。

要使 Follower 的日志与 Leader 恢复一致，Leader 必须找到最后两者达成一致的地方（说白了就是回溯，找到两者最近的一致点），然后删除从那个点之后的所有日志条目，发送自己的日志给 Follower。所有的这些操作都在进行附加日志的一致性检查时完成。

Leader 为每一个 Follower 维护一个 nextIndex，它表示下一个需要发送给 Follower 的日志条目的索引地址。当一个 Leader 刚获得权力的时候，它初始化所有的 nextIndex 值，为自己的最后一条日志的 index 加 1。如果一个 Follower 的日志和 Leader 不一致，那么在下一次附加日志时一致性检查就会失败。在被 Follower 拒绝之后，Leader 就会减小该 Follower 对应的 nextIndex 值并进行重试。最终 nextIndex 会在某个位置使得 Leader 和 Follower 的日志达成一致。当这种情况发生，附加日志就会成功，这时就会把 Follower 冲突的日志条目全部删除并且加上 Leader 的日志。一旦附加日志成功，那么 Follower 的日志就会和 Leader 保持一致，并且在接下来的任期继续保持一致。

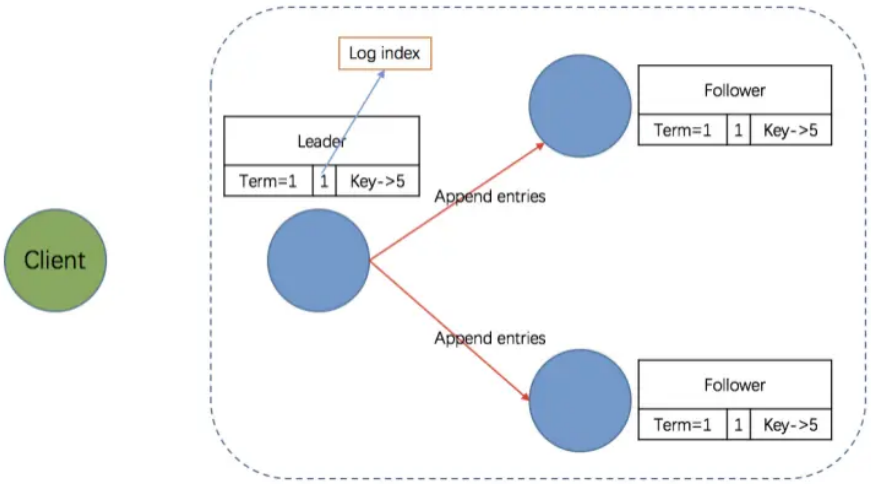

图4-7 Leader 发送Append Entries到 Follower

第三阶段：Leader 等待 Followers 回应。

Followers 接收到 Leader 发来的复制请求后，有两种可能的回应：

写入本地日志中，返回 Success；

一致性检查失败，拒绝写入，返回 False，原因和解决办法上面已做了详细说明。

需要注意的是，此时该 Entry 的状态也是未提交（Uncommitted）。完成上述步骤后，Followers 会向 Leader 发出 Success 的回应，当 Leader 收到大多数 Followers 的回应后，会将第一阶段写入的 Entry 标记为提交状态（Committed），并把这条日志条目应用到它的状态机中。

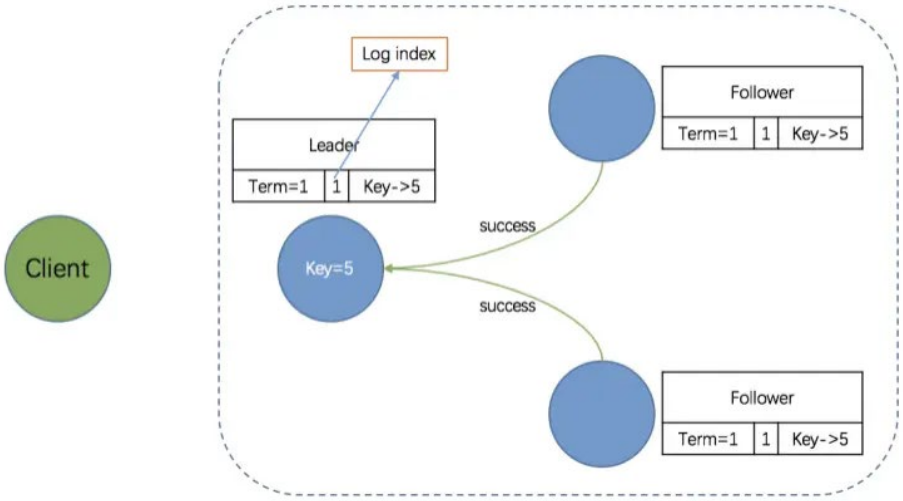

图4-8 Leader 等待 Followers 回应

第四阶段：Leader 回应客户端。

完成前三个阶段后，Leader会向客户端回应 OK，表示写操作成功。

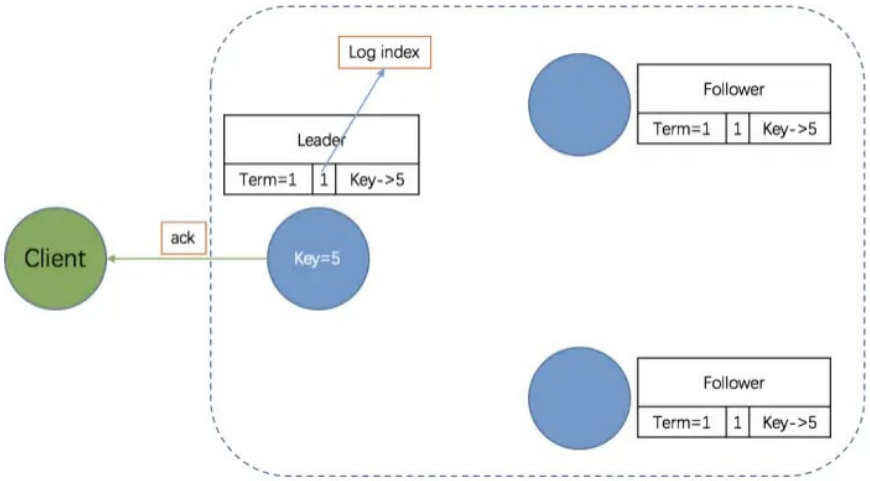

图4-9 Leader 回应客户端

第五阶段，Leader 通知 Followers Entry 已提交

Leader 回应客户端后，将随着下一个心跳通知 Followers，Followers 收到通知后也会将 Entry 标记为提交状态。至此，Raft 集群超过半数节点已经达到一致状态，可以确保强一致性。

需要注意的是，由于网络、性能、故障等各种原因导致“反应慢”、“不一致”等问题的节点，最终也会与 Leader 达成一致。

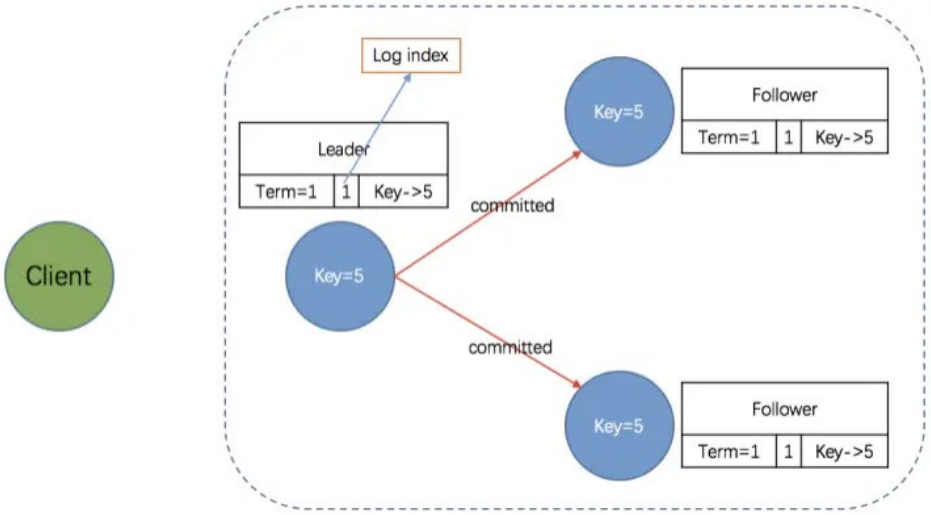

图4-10 Leader 通知 Followers Entry 已提交

4.3.4 算法安全性

前面描述了 Raft 算法是如何选举 Leader 和复制日志的。然而，到目前为止描述的机制并不能充分地保证每一个状态机会按照相同的顺序执行相同的指令。例如，一个 Follower 可能处于不可用状态，同时 Leader 已经提交了若干的日志条目；然后这个 Follower 恢复（尚未与 Leader 达成一致）而 Leader 故障；如果该 Follower 被选举为 Leader 并且覆盖这些日志条目，就会出现问题，即不同的状态机执行不同的指令序列。

为了保证整个过程的正确性，Raft算法保证以下属性时刻为真：

1. Election Safety

在任意指定Term内，最多选举出一个Leader

2. Leader Append-Only

Leader从不“重写”或者“删除”本地Log，仅仅“追加”本地Log

3. Log Matching

如果两个节点上的日志项拥有相同的Index和Term，那么这两个节点[0, Index]范围

内的Log完全一致

4. Leader Completeness

如果某个日志项在某个Term被commit，那么后续任意Term的Leader均拥有该日志项

5. State Machine Safety

一旦某个server将某个日志项应用于本地状态机，以后所有server对于该偏移都将应用相同日志项。

鉴于此，在 Leader 选举的时候需增加一些限制来完善 Raft 算法。这些限制可保证任何的 Leader 对于给定的任期号（Term），都拥有之前任期的所有被提交的日志条目（所谓 Leader 的完整特性）。关于这一选举时的限制，下文将详细说明。

选举限制：

在所有基于 Leader 机制的一致性算法中，Leader 都必须存储所有已经提交的日志条目。为了保障这一点，Raft 使用了一种简单而有效的方法，以保证所有之前的任期号中已经提交的日志条目在选举的时候都会出现在新的 Leader 中。换言之，日志条目的传送是单向的，只从 Leader 传给 Follower，并且 Leader 从不会覆盖自身本地日志中已经存在的条目。

Raft 使用投票的方式来阻止一个 Candidate 赢得选举，除非这个 Candidate 包含了所有已经提交的日志条目。Candidate 为了赢得选举必须联系集群中的大部分节点。这意味着每一个已经提交的日志条目肯定存在于至少一个服务器节点上。如果 Candidate 的日志至少和大多数的服务器节点一样新（这个新的定义会在下面讨论），那么它一定持有了所有已经提交的日志条目（多数派的思想）。投票请求的限制中请求中包含了 Candidate 的日志信息，然后投票人会拒绝那些日志没有自己新的投票请求。

Raft 通过比较两份日志中最后一条日志条目的索引值和任期号，确定谁的日志比较新。如果两份日志最后条目的任期号不同，那么任期号大的日志更加新。如果两份日志最后的条目任期号相同，那么日志比较长的那个就更加新。

提交之前任期内的日志条目：

Leader 知道一条当前任期内的日志记录是被提交了，只要它被复制到了大多数的 Follower 上（多数派的思想）。如果一个 Leader 在提交日志条目之前崩溃了，继任的 Leader 会继续尝试复制这条日志记录。然而，一个 Leader 并不能断定被保存到大多数

Follower 上的一个之前任期里的日志条目就一定已经提交了。这很明显，从日志复制的过程可以看出。

当 Leader 复制之前任期里的日志时，Raft 会为所有日志保留原始的任期号，这在提交规则上产生了额外的复杂性。但是，这种策略更加容易辨别出日志，即使随着时间和日志的变化，日志仍维护着同一个任期编号。此外，该策略使得新 Leader 只需要发送较少日志条目。

习题四

1、简述Paxos协议的原理和流程？

2、简述Raft算法3个子问题的解决过程。

5. 区块链共识机制

5.1 PoW共识机制

5.1.1 简介

PoW工作量证明（英文全称为Proof of Work）早在比特币出现之前就已经有人探索，常见的是利用HASH运算的复杂度进行CPU运算实现工作量确定，当然也可以利用卷积求导、大质数分解这些复杂的运算来达到工作量证明的目的。随着比特币成功后，PoW为人们熟知，基于HASH的PoW算法常被人误解为是PoW的代名词。

PoW最早是一个经济学名词，它是指系统为达到某一目标而设置的度量方法。简单理解就是一份证明，用来确认你做过一定量的工作。监测工作的整个过程通常是极为低效的，而通过对工作的结果进行认证来证明完成了相应的工作量，则是一种非常高效的方式。在1999年，Markus Jakobsson and Ari Juels两人将PoW概念引入计算机体系，设计系统用以抵挡拒绝服务攻击和网络爬虫，后来在反垃圾邮件中被广泛使用。其设计理念是一个正常用户写一封邮件是需要一定的时间，而发送垃圾邮件者是无法接受这个等待的时间，如果PoW系统能够使垃圾邮件发送者需要更多的时间来发送邮件，就可以增大他们的成本，起到抵挡攻击的作用。

PoW系统中一定有两个角色，工作者和验证者，他们需要具有以下特点：

(1) 工作者需要完成的工作必须有一定的量，这个量由工作验证者给出。

(2) 验证者可以迅速的检验工作量是否达标。

(3) 工作者无法自己"创造工作"，必须由验证者发布工作。

(4) 工作者无法找到很快完成工作的办法。

中本聪在其比特币奠基性论文中设计了PoW 共识机制，其核心思想是通过引入分布式节点的算力竞争来保证数据一致性和共识的安全性。 比特币系统中, 各节点 (即矿工) 基于各自的计算机算力相互竞争来共同解决一个求解复杂但验证容易的 SHA256 数学难题 (即挖矿)， 最快解决该难题的节点将获得区块记账权和系统自动生成的比特币奖励。 该数学难题可表述为: 根据当前难度值, 通过搜索求解一个合适的随机数 (Nonce) 使得区块头各元数据的双 SHA256 哈希值小于或等于目标哈希值。在节点成功找到满足的

Hash值之后，会马上对全网进行广播打包区块，网络的节点收到广播打包区块，会立刻对其进行验证。比特币系统通过灵活调整随机数搜索的难度 值来控制区块的平均生成时间为 10 分钟左右。

5.1.2 主要流程

在了解PoW共识机制前，我们先了解下比特币区块的结构，下图是比特币区块的结构图：

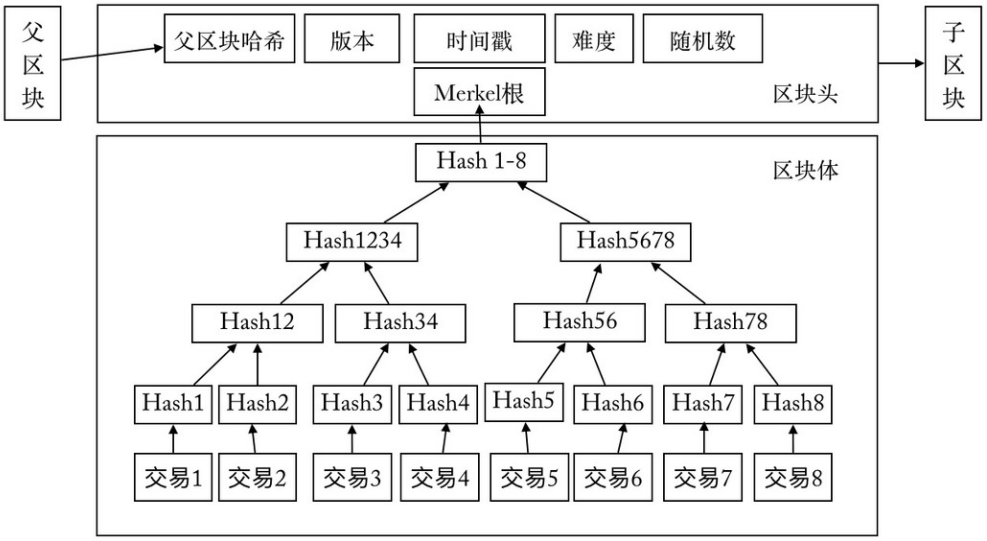

图5-1 比特币区块的结构图

从图上可知，比特币的结构分为区块头和区块体，其中区块头细分为：

- 父区块头哈希值：前一区块的哈希值，使用SHA256(SHA256(父区块头))计算。占32字节
- 版本：区块版本号，表示本区块遵守的验证规则。占4字节
- 时间戳：该区块产生的近似时间，精确到秒的UNIX时间戳，必须严格大于前11个区块时间的中值，同时全节点也会拒绝那些超出自己2个小时时间戳的区块。占4字节
- 难度：该区块工作量证明算法的难度目标，已经使用特定算法编码。占4字节
- 随机数(Nonce)：为了找到满足难度目标所设定的随机数，为了解决32位随机数在算力飞升的情况下不够用的问题，规定时间戳和coinbase交易信息均可更改，以此扩展nonce的位数。占4字节
- Merkle根：该区块中交易的Merkle树根的哈希值，同样采用SHA256(SHA256())计算。占32字节

区块体包含的交易列表则附加在区块头后面，其中的第一笔交易是coinbase交易，这是一笔为了让矿工获得奖励及手续费的特殊交易。

区块头中包含了用于比特币工作量证明的输入字符串。为了使区块头能体现区块所包含的所有交易，在区块的构造过程中，需要将该区块要包含的交易列表，通过Merkle Tree算法生成Merkle Root Hash，作为交易列表的摘要存到区块头中。

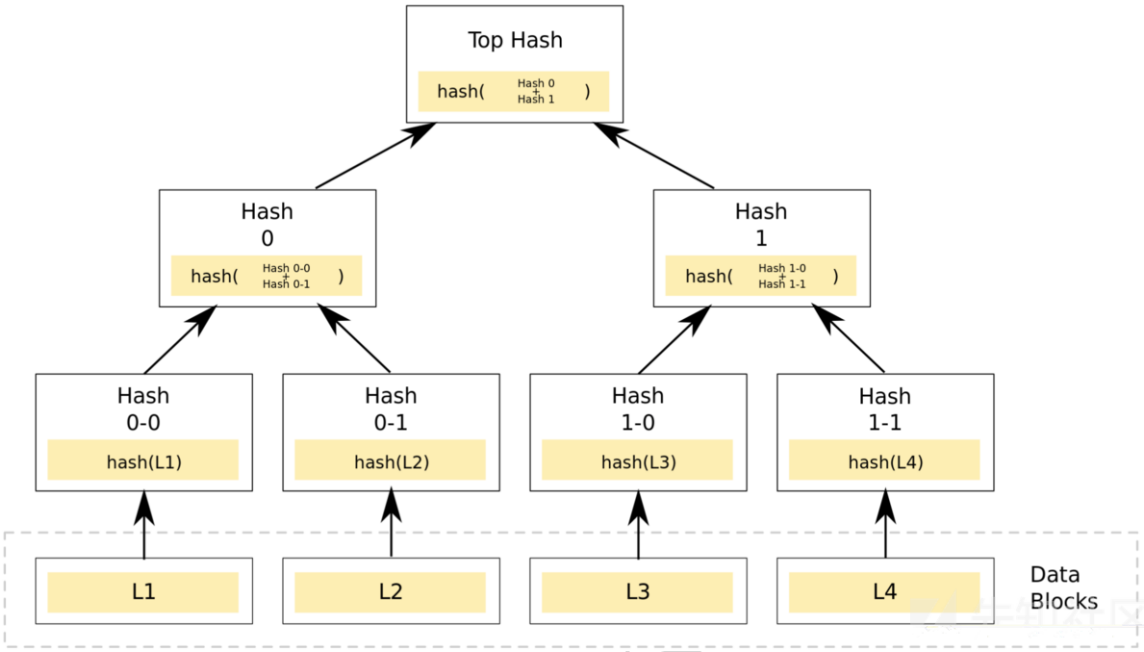

图5-2 生成Merkle Root Hash

作为比特币核心的共识算法PoW，其核心公式如式（1）所示，其中data是将时间戳、版本号、区块高度等信息组合的数据，nonce为一种随机值，$D(d)$为目标值，d为挖矿难度，随着挖矿难度增加，目标值与哈希值也会越难匹配。从式（1）可以看出，PoW是一个不断枚举的过程，算法使用嵌套哈希函数求解，当新的交易数据出现时，各个区块链节点开始全力运算，当某个节点最先满足式（1）则被称为矿工，矿工节点有权利将交易打包成区块，并发给其他节点进行验证交易合法性，如果交易通过验证，则将区块上链存储并在节点间同步。当然为了奖励矿工节点的工作，比特币系统奖励比特币给矿工作为收益。在比特币中规定每产生2016块就会对难度值进行相应的调整，一个区块的验证时间一般为10min左右。

$$SHA256(SHA256(data|nonce)) \leq D(d) \qquad (1)$$

$$D(d) = \frac{2^{224}-1}{d} \qquad (2)$$

在比特币中所谓的“挖矿”就是修改区块中特定位置的值，找满足要求的区块，该区块的哈希输出要小于特定的值，该值称作目标值，在比特币中，规定256位整数：

0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

为最大目标值，区块要被比特币网络接受其哈希结果必须要小于最大目标值，把该目标值对应的难度定义1，比特币网络会自动的调整难度值，随着比特币挖矿的进行，区块的目标值会越来越小，哈希结果满足要求的概率越来越低，即找到满足要求的区块会越来越难。

由于哈希的随机性，进行一次哈希运行256位整数任意位置的值要么为0要么为1，以最高位为例，要使最高位为零（其他位不关心）理论上进行2次运算即可出现一次，进而要使最高位和次高位都为零，则理论上要进行2^2才会出现一次，依次类推，要使最高32位都为零则需要进行2^32次运行才行，即找到难度为1的哈希值理论上需要进行2^32次运行，进而要找到难度为d的哈希值，理论上要进行d*2^32次运行，数字货币挖矿时，矿池在对矿工进行算力统计时就是根据该理论进行的。

PoW 共识的随机数搜索过程如下：

(1) 搜集当前时间段的全网未确认交易, 并增加一个用于发行新比特币奖励的 Coinbase 交易, 形成当前区块体的交易集合;

(2) 计算区块体交易集合的Merkle根记入区块头, 并填写区块头的其他元数据, 其中随机数 Nonce 置零;

(3) 随机数 Nonce 加 1; 计算当前区块头的双 SHA256 哈希值, 如果小于或等于目标哈希值, 则成功搜索到合适的随机数并获得该区块的记账权; 否则继续步骤 3 直到任一节点搜索到合适的随机数为止;

(4) 如果一定时间内未成功, 则更新时间戳和未确认交易集合、重新计算 Merkle 根后继续搜索。

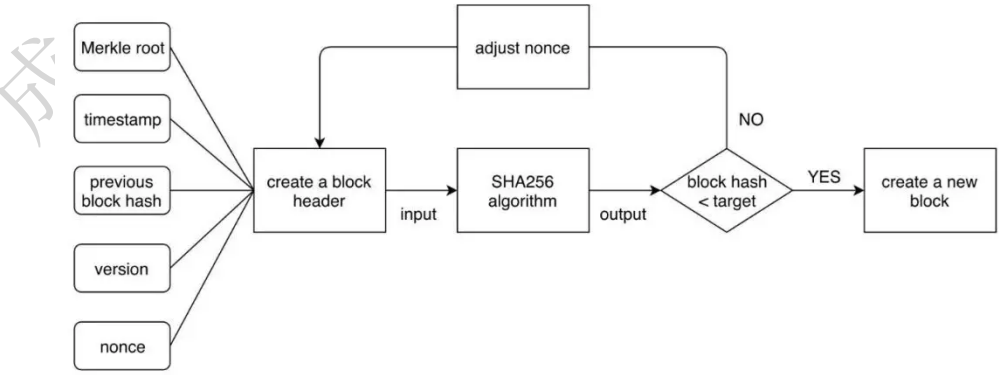

图5-3 PoW计算流程

如在区块277, 316被挖出的时候，区块结构中用来表示版本号的字段值为2，长度为4

字节，以小段格式编码值为0x20000000。接着，挖矿节点需要填充“前区块哈希”，打包节点从网络上接收到的区块277,315的区块头哈希值，它是区块277316候选区块的父区块。区块277,315的区块头哈希值为：

000000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569为了向区块头填充merkle根字段，要将全部的交易组成一个merkle树。创币交易作为区块中的首个交易，后将余下的418笔交易添至其后，这样区块中的交易一共有419笔。Merkle树中必须有偶数个叶子节点，所以需要复制最后一个交易作为第420个节点，每个节点是对应交易的哈希值。这些交易的哈希值逐层地、成对地组合，直到最终组合并成一个根节点。merkle数的根节点将全部交易数据摘要为一个32字节长度的值，如当前区块merkel根的值如下：

c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e

挖矿节点会继续添加一个4字节的时间戳，以Unix纪元时间编码，即自1970年1月1日0点到当下总共流逝的秒数。本例中的1388185914对应的时间是2013年12月27日，星期五，UTC/GMT。接下来，节点需要填充难度目标值，为了使得该区块有效，这个字段定义了所需满足的工作量证明的难度。难度目标值在区块中以“尾数-指数”的格式，编码并存储，这种格式称作“难度位”。这种编码的首字节表示指数，后面的3字节表示尾数(系数)。以区块277316为例，难度位的值为0x1903a30c，0x19是指数的十六进制格式，后半部0x03a30c是系数。

计算难度目标的公式为：

$$target = coefficent * 2^{(8*(exponent-3))} \quad (3)$$

由此公式及难度位的值 0x1903a30c，可得：

target = 0x03a30c * 2^(0x08 * (0x19 - 0x03))

=> target = 0x03a30c * 2^(0x08 * 0x16)

=> target = 0x03a30c * 2^0xB0

按十进制计算为：

=> target = 238,348 * 2^176

=> target =

22,829,202,948,393,929,850,749,706,076,701,368,331,072,452,018,388,575,71

5,328

转化为十六进制后为：

=> target

=0x0000000000000003A30C0000000000000000000000000000000000000000000000

也就是说高度为277,316的有效区块的头信息哈希值是小于这个目标值的。

最后一个字段是nonce，初始值为0。

区块头完成全部的字段填充后，挖矿就可以开始进行了。挖矿的目标是找到一个使区块头哈希值小于难度目标的nonce。挖矿节点通常需要尝试数十亿甚至数万亿个不同的nonce取值，直到找到一个满足条件的nonce值。

5.1.3 难度调整

比特币的区块平均每10分钟生成一个。这就是比特币的心跳，是货币发行速率和交易达成速度的基础。不仅是在短期内，而是在几十年内它都必须要保持恒定。在此期间，计算机性能将飞速提升。此外，参与挖矿的人和计算机也会不断变化。为了能让新区块的保持10分钟一个的产生速率，挖矿的难度必须根据这些变化进行调整。事实上，难度是一个动态的参数，会定期调整以达到每10分钟一个新区块的目标。简单地说，难度被设定在，无论挖矿能力如何，新区块产生速率都保持在10分钟一个。那么，在一个完全去中心化的网络中，这样的调整是如何做到的呢？难度的调整是在每个完整节点中独立自动发生的。每2,016个区块中的所有节点都会调整难度。难度的调整公式是由最新2,016个区块的花费时长与20,160分钟（两周，即这些区块以10分钟一个速率所期望花费的时长）比较得出的。难度是根据实际时长与期望时长的比值进行相应调整的（或变难或变易）。简单来说，如果网络发现区块产生速率比10分钟要快时会增加难度。如果发现笔10分钟慢时则降低难度。

图5-4 比特币挖矿难度变化图

5.1.4 算法特点

PoW 算法主要存在三方面的问题：

(1) 效率低。在比特币系统中的区块容量只有 1MB，其网络中交易量大约为 7 笔 / s，出块时间则为 10min 左右，这种规格的吞吐量非常不适合高并发的商业场景。

(2) 资源消耗大。由于矿工节点在挖矿的过程中需要大量的电力与流处理器等挖矿机器，根据 Digiconomist 比特币能源耗指数和剑桥比特币电力消耗指数等估算，比特币网络 2018 年用电量则相当于菲律宾或芬兰的用电量。

(3) 去中心化程度较低。中本聪设计比特币挖矿的初衷是每个人都用 CPU 来挖矿，而现在算力大部分掌握在算力矿池手中，矿池与矿池也会联盟来竞争记账权利。

5.2 PoS共识机制

5.2.1 简介

PoS机制全称是proof of stake，也就是权益证明，它是除PoW之外的另一类主流共识机制。PoW机制有一些缺点，比如挖矿时会浪费大量的资源，整个网络处理效率较低等问题。为了解决这些问题，就在PoW的基础上重新设定一个机制，既能保留PoW是又能解决问题，于是PoS共识机制就此诞生，PoS机制主要是通过权益记账的方式来解决网络的

效率低下，资源浪费和各节点的一致性问题，简单来说，谁拥有的权益多谁说了算，它的原理是这样的，PoS之所以有种种问题，主要是因为人人都可以自由的成为节点，而每个节点又通过竞争的方式参与数据处理，一笔数据要经过这么多人的处理，肯定会造成资源浪费和效率低下，PoS之所以能解决这个问题，是因为提高了节点处理数据的门槛，它规定每个人可以自由的加入进来成为节点，但只有满足一定条件的节点，比如抵押一定数量的代币才有资格成为验证节点，每隔一段时间，会重新选择，选举的结果不能被操纵，也不能被预测，从而避免网络被某一节点所控制。

2011 年，权益证明共识算法(Proof of Stake，PoS)被提出，在 2012 年发布的点点币(Peercoin，PPC)中首次实现。与 PoW 共识算法的算力比拼不同，权益证明共识算法的记账权决定于节点的币龄大小，其中币龄表示为持币数量与持币时间之积。根据币龄权值的大小关系降低了计算机 Hash 计算的难度，这有效缓解了工作量证明的算力浪费，缩短了共识时间，提高了共识的效率。

5.2.2 主要流程

参照 PoW 共识机制的定义，将 PPC（点点币） 中的 PoS 共识算法称为原生 PoS 共识算法。原生 PoS 共识算 法中引入了“币龄”的概念，相关定义如下：币 龄（coinAge）是 指 持 币 数 量 （coins）与持币时间（holdTime）的乘积： coinAge = coins × holdTime

PoS共识算法的流程：

PoS共识算法给定一个全网统一的难度值D，以及新打包进区块的元数据 tradeData，寻找 满足条件的记数器 timeCounter，使得：

$$HA256(SHA256(tradeData|timeCounter)) \leq D \times coinAge \qquad (4)$$

PoS 共识算法的记账规则与 PoW 共识算法基本 相似，但 PoS共识算法不需要矿工枚举所有的随机数 Nonce，而是在1s内只允许一次哈希值，大大减轻了计算工作量，从而减缓了算力竞争带来的资源消耗。

例如当前全网的难度值是4369，A矿工的输入的币龄是15，那么A矿工的目标值为65535，换算成十六进制就是 0xFFFF，完整的哈希长度假设是8个字节，也就是0x0000FFFF。而B矿工比较有钱，他输入的币龄是240，那么B矿工的目标值就是0x000FFFFF,所以B矿工获得记账权的概率肯定要比A高。

5.2.3 算法特点

由于 PoS 共识算法的提出主要是为了解决 PoW 共识算法的缺陷，通过与 PoW 共识算法的比较便可 以体现 PoS共识算法的优势：

(1) PoS 是通过权益大小来寻找哈希值，不存在因算力竞争浪费能源的问题。诞生于 2013 年 11 月 24 日的 Nextcoin（未来币）是一个全新的使用 PoS 共识算法的第二代虚拟货币。

(2) 由于 PoS 共识算法采用类似“币龄”的权益 值，并对成功挖矿者采用了币龄“清零”机制，PoS 共 识算法更加公平。

(3) 为了防止节点离线积累币龄，2014 年 Vasin 提出了 PoS2.0并应用到黑币（BlackCoin）中，从而使黑币从当时的 PoW+PoS 共识模式发展到了纯 PoS 共识模式。PoS2.0 共识算法中的权益大小与用户在线时间成正比，这种激励机制有效增强了区块链 P2P 网络的健壮性。但 PoS 共识算法容易产生分叉，且区块链的去中心化特点被弱化。

5.3 DPoS共识机制

5.3.1 简介

针对PoS共识算法主要存在离线节点积累币龄和某些拥有权益的节点无意挖矿等缺陷，2014年4月，Larimer在PoS共识算法的基础上提出了委托权益证明（delegated proof of stake，DPoS）共识算法。 DPoS共识算法引入了类似企业董事会的管理机制， 权益持有者通过投票方式选择能够代表自己利益的委托人（票数与持有代币的多少成正比），被选出的受信任的委托人需要交纳一定数量的保证金，最后形成一个由101个委托人组成的集合。该集合中的委托人将以平等的权利按规则轮流作为出块者生成区块，并从每笔交易的手续费中获得收益。在此期间，如果某个委托人违反了出块规则，其委托人身份将被新选出的委托人替代，而且其保证金也被系统没收。DPoS共识算法通过选举委托人的方式来确定区块的生成者，缩短了交易的确认时间，提高了系统的效率，但由于被选举出的负责挖矿的委任人数量仅为101个，DPoS共识算法是一个去中心化（针对权益持有者）和中心化（委托人）相结合的共识算法。另外，DPoS 共识算法可能会存在委托人通过联合操纵区块链网络来损害普通节点利益，以及通过选出不符合要求的委托人从而迫使网络的性能下降等现象。

5.3.2 主要流程

在 DPoS 共识制中，引入了见证人(witness)的概念，见证人可以当作是“董事会决策”中的代表。DPoS 通过选举见证人的方式行使权利，具体流程如下：

(1) 选举出块者。见证人即代表，通过持有选票者投票选举产生。见证人在系统中保持中立，每24小时更新一次，且投票情况决定其收益。持有选票的节点根据自己的意愿给被选举节点投票，投票的总数大于全网票数的 1/2 则选举有效，得票最高的101个节点成为见证人。

(2) 提出区块。见证人轮流产生区块，签名、添加时间戳、广播新块。见证人在规定时间内没有产生新块，超时后，将由下一个见证人产生新区块。

(3) 验证区块。“董事会”中的其他见证人将验证新产生区块的合法性，验证结果为 合法新区块才能上链。

(4) 更新区块链。验证结果合法的新区块上链，更新区块链。 相比于PoW和PoS的算力比拼和权益比拼，DPoS 可以称为“民主集中式”的记账方式。DPoS 达到了秒级验证，保证了共识过程大于51%的权益参与，能够抵抗 51%攻击。 因此，DPoS 的速度、效率、安全性和去中心化都具有明显的优势。

5.3.3 算法特点

DPoS算法的优点：

(1) 能耗更低。DPoS在确保网络安全的前提下，能够降低全网功耗，网络运营成本比较低。

(2) 交易更快。减少了记账节点的数量，提升了区块验证和记账速度，能达到秒级共识。

DPoS算法的缺点：

(1) 节点收益不够合理。DPoS 按照得票股份分红，落选的备选见证人节点对于整个系统产生的边际收益小，分红收益低，对于落选的备选见证人节点收益分配不合理。

(2) 投票积极性不高。投票需要花费各种资源，使大多数节点积极性降低，甚至有百分之九十的持股人都从未参与过投票。

(3) 安全隐患。对于出现离线节点、坏节点或恶意节点的情况，DPoS不能及时处理，

因此网络安全存在极大的隐患。

5.4 PBFT

5.4.1 简介

1999 年，Castro M 和 Liskov B 提出了拜占庭容错算法(Practical Byzantine Fault Tolerance，PBFT)，算法提出之初是为了解决拜占庭将军问题。PBFT能提升系统的响应速度，同时有效降低复杂度，之后在其他领域有较为广泛的应用。 PBFT中将节点分为客户端节点(Client)和共识节点，共识节点包含主节点(Primary)和副本节点(Replica)。Client负责发送请求；Primary负责对请求消息进行排序，并将相关消息广播至所有Replica，每轮共识过程有且只有一个Primary；Replica负责区块共识，在验证并通过来自Primary的消息之后执行相关操作并将结果返回给 Client，每轮共识过程有多个Replica且工作内容类似。

5.4.2 主要流程

PBFT的共识属于典型的三阶段，分别为预准备(preprepare)、准备(prepare)和确认(commit)等阶段，主要过程如下：

(1) Client 提出请求，将请求消息发送给Primary；

(2) 预准备阶段：Primary接收请求消息，对请求消息进行处理形成预准备消息，然后将预准备消息广播发送至所有Replica；

(3) 准备阶段：所有Replica节点接收预准备消息，验证预准备消息摘要是否被篡改，若未被篡改，则将预准备消息处理成为准备消息，然后将准备消息广播至所有共识节点；

(4) 确认阶段：共识节点接收并验证准备消息，若准备消息为真则将生成的确认消息 广播至其他的共识节点；

(5) 共识节点在接收和验证确认消息后，生成结果，然后将结果返回给Client。

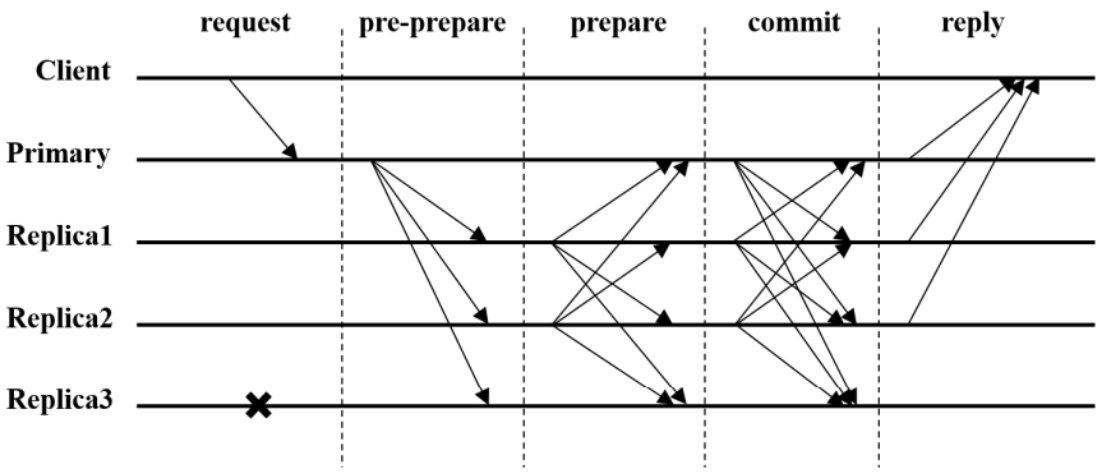

图5-5 PBFT算法流程图

算法详细说明如下：

PBFT算法前提，采用密码学算法保证节点之间的消息传送是不可篡改的。

PBFT容忍无效或者恶意节点数：f，为了保障整个系统可以正常运转，需要有2f+1个正常节点，系统的总节点数为：|R| = 3f + 1。也就是说，PBFT算法可以容忍小于1/3个无效或者恶意节点，该部分的原理证明请参考PBFT论文，论文地址：http://pmg.csail.mit.edu/papers/osdi99.pdf。

PBFT是一种状态机副本复制算法，所有的副本在一个视图（view）轮换的过程中操作，主节点通过视图编号以及节点数集合来确定，即：主节点 p = v mod |R|。v：视图编号，|R|节点个数，p：主节点编号。通过View Changes，可以选举出新的、让请求在有限时间内达成一致的主节点，向客户端响应，从而满足可用性的要求。

1. REQUEST：

客户端c向主节点p发送<REQUEST, o, t, c>请求。o: 请求的具体操作，t: 请求时客户端追加的时间戳，c：客户端标识。REQUEST: 包含消息内容m，以及消息摘要d(m)。客户端对请求进行签名。

2. PRE-PREPARE：

主节点收到客户端的请求，需要进行以下校验：

(1) 客户端请求消息签名是否正确。

非法请求丢弃。正确请求，则分配一个编号n，编号n主要用于对客户端的请求进行

排序。然后广播一条<<PRE-PREPARE, v, n, d>, m>消息给其他副本节点。v：视图编号，d客户端消息摘要，m消息内容。<PRE-PREPARE, v, n, d>进行主节点签名。n是要在某一个范围区间内的[h, H]，具体原因参见垃圾回收章节。

3. PREPARE：

副本节点i收到主节点的PRE-PREPARE消息，需要进行以下校验：

(1) 主节点PRE-PREPARE消息签名是否正确。

(2) 当前副本节点是否已经收到了一条在同一v下并且编号也是n，但是签名不同的PRE-PREPARE信息。

(3) d与m的摘要是否一致。

(4) n是否在区间[h, H]内。

非法请求丢弃。正确请求，则副本节点i向其他节点包括主节点发送一条<PREPARE, v, n, d, i>消息，v, n, d, m与上述PRE-PREPARE消息内容相同，i是当前副本节点编号。<PREPARE, v, n, d, i>进行副本节点i的签名。记录PRE-PREPARE和PREPARE消息到log中，用于View Change过程中恢复未完成的请求操作。

4. COMMIT：

主节点和副本节点收到PREPARE消息，需要进行以下校验：

(1) 副本节点PREPARE消息签名是否正确。

(2) 当前副本节点是否已经收到了同一视图v下的n。

(3) n是否在区间[h, H]内。

(4) d是否和当前已收到PRE-PPREPARE中的d相同

非法请求丢弃。如果副本节点i收到了2f+1个验证通过的PREPARE消息，则向其他节点包括主节点发送一条<COMMIT, v, n, d, i>消息，v, n, d, i与上述PREPARE消息内容相同。<COMMIT, v, n, d, i>进行副本节点i的签名。记录COMMIT消息到日志中，用于View Change过程中恢复未完成的请求操作。记录其他副本节点发送的PREPARE消息到log中。

5. REPLY：

主节点和副本节点收到COMMIT消息，需要进行以下校验：

(1) 副本节点COMMIT消息签名是否正确。

(2) 当前副本节点是否已经收到了同一视图v下的n。

(3) d与m的摘要是否一致。

(4) n是否在区间[h, H]内。

非法请求丢弃。如果副本节点i收到了2f+1个验证通过的COMMIT消息，说明当前网络中的大部分节点已经达成共识，运行客户端的请求操作o，并返回<REPLY, v, t, c, i, r>给客户端，r：是请求操作结果，客户端如果收到f+1个相同的REPLY消息，说明客户端发起的请求已经达成全网共识，否则客户端需要判断是否重新发送请求给主节点。记录其他副本节点发送的COMMIT消息到log中。

在上述算法流程中，为了确保在View Change的过程中，能够恢复先前的请求，每一个副本节点都记录一些消息到本地的log中，当执行请求后副本节点需要把之前该请求的记录消息清除掉。最简单的做法是在Reply消息后，再执行一次当前状态的共识同步，这样做的成本比较高，因此可以在执行完多条请求K（例如：100条）后执行一次状态同步。这个状态同步消息就是CheckPoint消息。

副本节点i发送<CheckPoint, n, d, i>给其他节点，n是当前节点所保留的最后一个视图请求编号，d是对当前状态的一个摘要，该CheckPoint消息记录到log中。如果副本节点i收到了2f+1个验证过的CheckPoint消息，则清除先前日志中的消息，并以n作为当前一个stable checkpoint。

这是理想情况，实际上当副本节点i向其他节点发出CheckPoint消息后，其他节点还没有完成K条请求，所以不会立即对i的请求作出响应，它还会按照自己的节奏，向前行进，但此时发出的CheckPoint并未形成StablePoint，为了防止i的处理请求过快，设置一个上文提到的高低水位区间[h, H]来解决这个问题。低水位h等于上一个stable checkpoint的编号，高水位H = h + L，其中L是我们指定的数值，等于checkpoint周期处理请求数K的整数倍，可以设置为L = 2K。当副本节点i处理请求超过高水位H时，此时就会停止脚步，等待stable checkpoint发生变化，再继续前进。

5.4.3 视图切换

如果主节点作恶，它可能会给不同的请求编上相同的序号，或者不去分配序号，或者让相邻的序号不连续。备份节点应当有职责来主动检查这些序号的合法性。如果主节

点掉线或者作恶不广播客户端的请求，客户端设置超时机制，超时的话，向所有副本节点广播请求消息。副本节点检测出主节点作恶或者下线，发起视图切换（View Change）协议。

副本节点向其他节点广播<VIEW-CHANGE, v+1, n, C, P, i>消息。n是最新的stable checkpoint的编号，C是2f+1验证过的CheckPoint消息集合，P是当前副本节点未完成的请求的PRE-PREPARE和PREPARE消息集合。

当主节点p = v + 1 mod |R|收到2f个有效的VIEW-CHANGE消息后，向其他节点广播<NEW-VIEW, v+1, V, O>消息。V是有效的VIEW-CHANGE消息集合。O是主节点重新发起的未经完成的PRE-PREPARE消息集合。PRE-PREPARE消息集合的选取规则：

1. 选取V中最小的stable checkpoint编号min-s，选取V中prepare消息的最大编号max-s。

2. 在min-s和max-s之间，如果存在P消息集合，则创建<<PRE-PREPARE, v+1, n, d>, m>消息。否则创建一个空的PRE-PREPARE消息，即：<<PRE-PREPARE, v+1, n, d(null)>, m(null)>, m(null)空消息，d(null)空消息摘要。

副本节点收到主节点的NEW-VIEW消息，验证有效性，有效的话，进入v+1状态，并且开始O中的PRE-PREPARE消息处理流程。

5.4.4 算法特点

PBFT算法由于每个副本节点都需要和其他节点进行P2P的共识同步，因此随着节点的增多，性能会下降的很快，但是在较少节点的情况下可以有不错的性能，并且分叉的几率很低。PBFT主要用于联盟链，但是如果能够结合类似DPOS这样的节点代表选举规则的话也可以应用于公联，并且可以在一个不可信的网络里解决拜占庭容错问题，TPS应该是远大于POW的。

5.5 共识算法对比

通过分析区块链共识算法的原理和算法特点，在表5-1中总结了4种区块链共识算法的记账节点选取方式、算法优点及缺点。在表5-2中进行了4种区块链共识算法性能对比。

表5-1 区块链共识算法性能对比

算法	记账节点选取方式	优点	缺点
PoW	拥有最高算力的节点易获得新区块的记账权和区块奖励	算法简单，可操作性强，节点自由加入或者退出，可扩展性强	严重浪费电力等资源，依赖专业挖矿硬件资源，算力集中在几大矿场之间，有中心化风险且效率低下，交易吞吐量小
PoS	拥有最高权益的节点易获得新区块的记账权和区块奖励	降低了PoW机制算力的浪费，节点不再依赖专业挖矿硬件资源，节点自由加入或者退出，可扩展性强	寡头优势，需要完善解决PoS机制存在的“无利害关系”问题，吞吐量小，一些平台手续费高
DPoS	由拥有权益的节点选出前N个超级节点作为新区块的记账节点，这些受理人轮流获得记账权和区块奖励	高吞吐量，更快地确认时间，降低能源损耗，节省了区块确认时间，减少了交易延迟，吞吐量得到极大提高，节点自由加入或者退出，可扩展性强	超级节点投票麻烦，投票期间需要锁定代币，不利于激发普通节点参与，固定数量超级节点的竞选规则令人担忧，去中心化的实现存在争议
PBFT	取模运算$P = V \ mod \ \|R\|$	共识结果的一致性和正确性程度高，确认时间快	算法复杂度较高，通信量大，无法避免恶意节点成为主节点

表5-2 区块链共识算法性能对比

算法	拜占庭容错	出块时间	交易吞吐量(TPS)	可扩展性
PoW	50%	>500s	<10	强
PoS	50%	<100s	<1000	强
DPoS	50%	<100s	>1000	强
PBFT	33%	<10s	<2000	弱