区块链技术与应用

计算机科学与技术学院　李京

04章 区块链共识层

目 录

4.2 分布式一致性算法

- Paxos算法
- Raft算法

1. Paxos算法回顾

- Paxos算法将分布式系统的节点分为三种角色：
  - 提议者（Proposer）负责向acceptor发起提案
  - 接受者（Acceptor）负责响应提案，对提案进行回应以表示自己接受提案
  - 学习者（Learner）不参与前面的决策过程，只从别人那里学习已经确定的、达成一致的提案结果
  - 一个节点可以同时拥有这三种身份，也可以只有部分身份。
- 如果一个提案被半数以上Acceptor接受，它就被选定了（Chosen），并由Learner负责执行选定的提案。

Paxos-问题描述-提案

- 提案Proposal由两部分组成：提案编号+提案值
- 提案编号id是不可重复的递增序列。
- 提案值（Value）是要等待达成共识的数据值。Paxos要保证在众多被提出来的Value中，只有一个会被最终选定。这是Paxos算法的核心。

Paxos-问题描述-约束条件

- 对Value：如果没有Value被提出，就不应该有Value被选定。
  - A：只有被提出的值Value才可以被选定
  - B：只有一个值Value可以被选定
  - C：除非一个值Value被选定，否则它不会被执行

Paxos-问题描述-约束条件

- 对于提案：如果只有一个提案被提出的话,那么这个提案应该被最终选定，Acceptor必须能够接受多个不同的Value。
  - P1：每个 Acceptor必须接受它收到的第一个提案
  - P2：如果一个提案（其值为v）已经被选定，那么对于所有编号更大的被选定的提案，它们所提议的值也必须是v。
    - P2a：如果一个提案（其值为v）已经被选定，那么对于任何Acceptor接受的编号更大的提案，它们的值也是v。
    - P2b：如果一个提案（其值为v）已经被选定，那么对于任何Proposer提出的编号更大的提案，它们的值也是v。
    - P2c：对于任意的v和n，如果提案（编号为n，值为v）被提出，那么存在一个由大多数Acceptors构成的集合S，满足下述两个条件之一：
      - ① 没有成员接受过小于n的提案;
      - ② 成员接受过的提案中，编号最大者的值为v。
  - 上述约束条件的关系是P2<P2a<P2b<P2c，即如果满足P2c就可以确保满足P2，Paxos算法就是建立在P2c上。

Paxos-算法流程

- Paxos算法分为两个阶段：
- Phase 1
  - (1)准备(Prepare)：一个Proposer创建一个提案(编号N)，并向超过半数的Acceptors发送包含提案编号的Prepare(N)消息
  - (2)承诺(Promise)：每个Acceptor收到消息后，检查提案的编号N是否大于它曾接受过的所有提案的编号。如果是，它会回应以Promise(Nx,Vx)消息，承诺不会接受任何编号小于N的提案；否则它将不予回应。其中Nx和Vx是它曾接受过的提案中编号最大的提案的编号与值，如果没有接受过提案，Nx和Vx为NULL。
- Phase 2
  - (1)请求接受(Accept Request)：如果Proposer收到了超过半数Acceptors的Promise消息，它需要先找到这些消息中编号最大的提案的值Vn，然后向这些Acceptors发送Accept(N,Vn)消息；如果所有Promise消息中Nx和Vx都为NULL，则Proposer可以选择任意的值作为V。
  - (2)接受(Accepted)：当Acceptor收到Accept(N,V)消息，它首先检查是否已承诺过编号大于N的提案，如果答案是否定的，它就接受该提案N，并发送Accepted（N,Vn)；否则就拒绝。

Paxos-活锁

- 为了解决这样的问题，在实际使用时可以进行这样的修改：
  - 在集群中选出一个节点作为leader，在对每条数据/每条事务达成一致性的时候，只有leader会作为Proposer而其他节点不会发起提案。对每条事务都运行一次上述的一轮Paxos，由于此时只有一个Proposer，所以就不会出现活锁的情况。
  - Leader可以向其他节点定时发送一个心跳包来声明自己存活。一旦其他节点认为leader消失了，则可以发起竞选来尝试成为leader，选出leader的过程就也能直接使用一轮Paxos算法来选出。

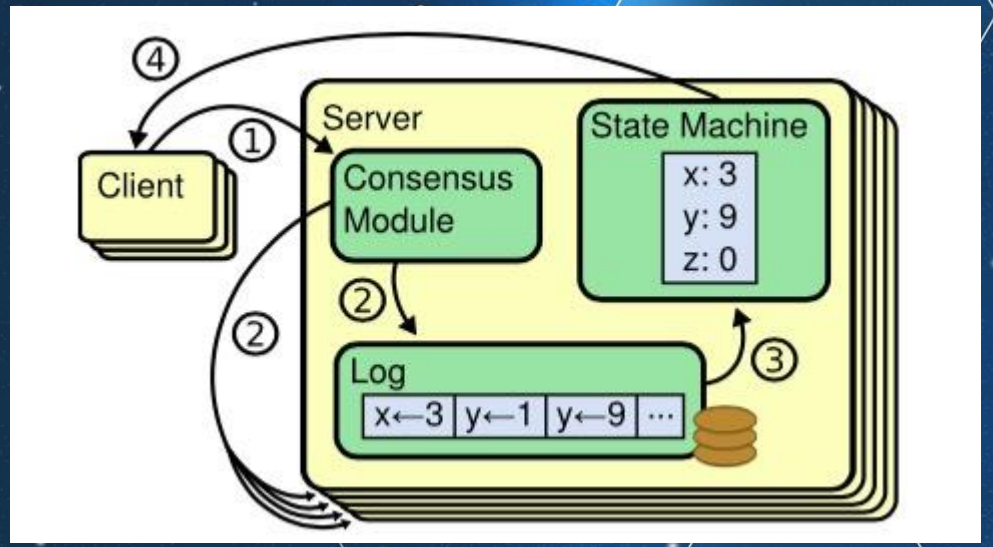
2. Raft算法

- Raft算法是2013年由斯坦福大学的迭戈·翁加罗(Diego Ongaro)和约翰·奥斯特豪特(John Ousterhout)提出的一种适用于非拜占庭容错环境下的分布式一致性算法。“In Search of an Understandable Consensus Algorithm”
- Paxos算法不易理解且不容易实现
- 多值（保持顺序）的Paxos算法将更为复杂
- Raft采用了特定的技术设计来提高算法的可理解性。
  - 模块化（领导选举、日志复制、安全性、成员变更等）
  - 状态空间规约

Raft算法-一些概念

复制状态机RSM(Replicated State Machine)

- 复制状态机是指多台机器具有完全相同的状态，并且运行完全相同的确定性状态机。
- 通过使用这样的状态机，可以解决很多分布式系统中的容错问题。复制状态机通常可以容忍半数节点故障。
- 每个服务器节点都有：
  - 一致性模块
  - 日志
  - 状态机



Raft算法-一些概念

- 基于日志的复制机制
  - 日志记录了导致状态机中状态转换的命令序列。
  - 状态机的状态可以通过执行日志中的命令序列获得，状态机的当前状态是可以重新计算得到。
  - 保证不同节点间的日志一致（即保存有相同顺序的命令序列），即最终可以保证状态机之间的状态一致性。

Raft算法-思路和过程

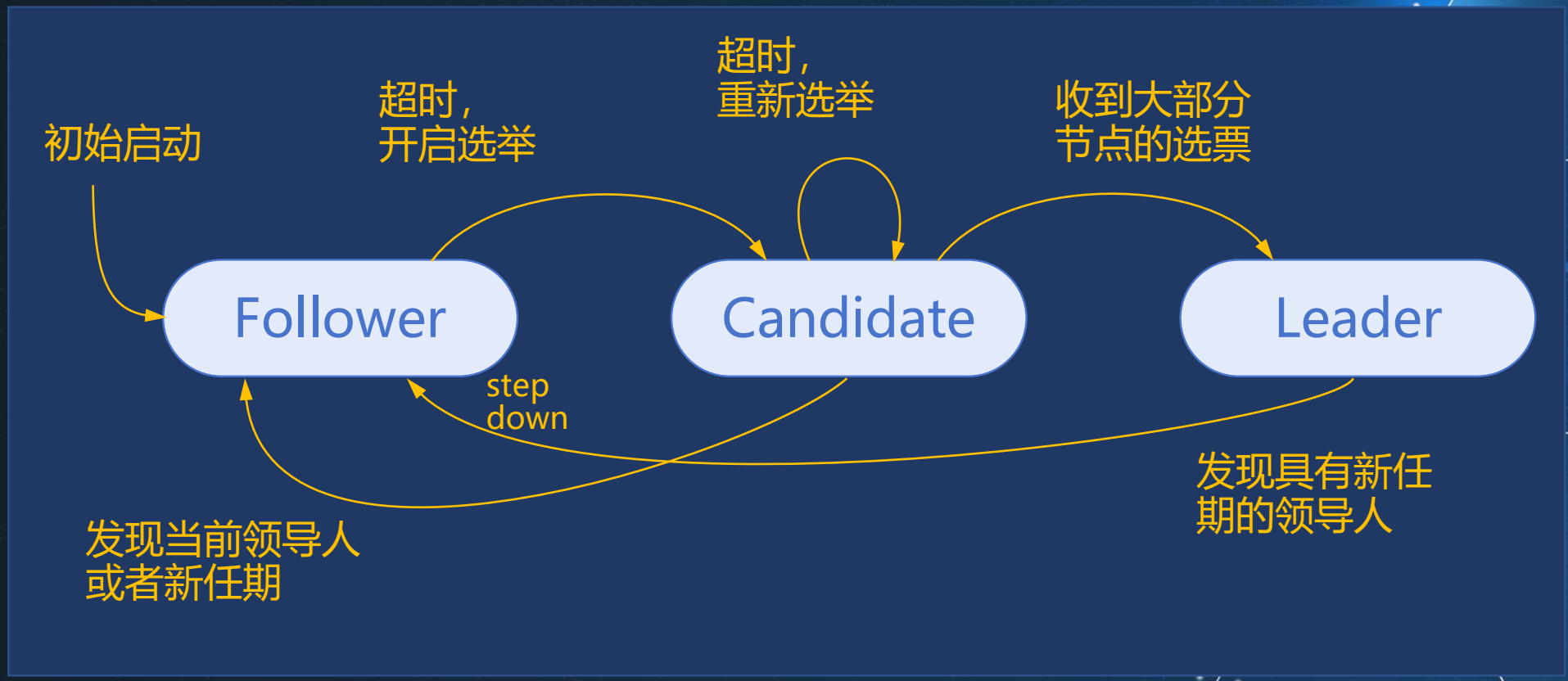
Raft算法采用RSM复制状态机模型，每个节点服务器都是一个状态机，所有的服务器都以同样的顺序响应客户端的请求。如何保证分布式一致性的问题就转换成如何保证所有的状态机的日志一致性的问题。其算法思路：

- 在集群中通过领导选举确定一个领导者，全权负责复制日志的管理。
- 领导者从客户端接收日志条目，将日志条目复制到其他服务器，并且在保证安全性的时候通知其他服务器将日志条目应用到它们的状态机中。
- 基于领导选举的机制大大简化了日志复制的管理，例如领导者可以自主决定新日志条目需要放置在日志的什么位置，而不需要与其他服务器商议，并且数据都是从领导者流向其他服务器。
- 当领导者宕机或者与其他服务器断开连接后，集群其他节点会启动领导选举过程并选出新的领导者。

Raft算法

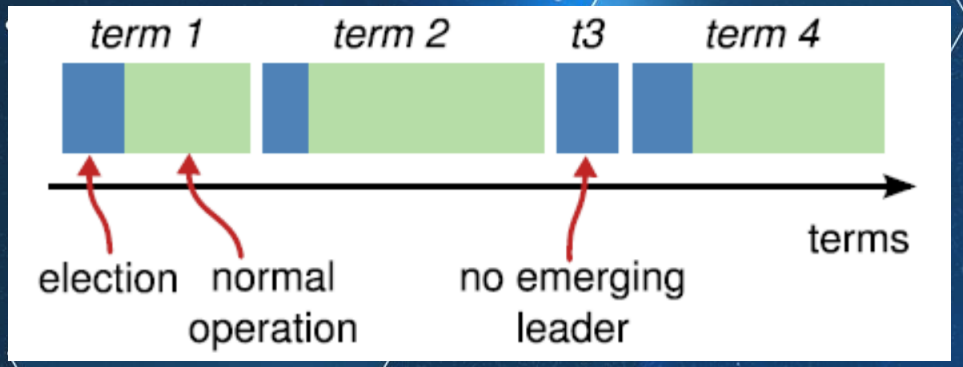
- Raft集群节点在任一时刻都处于三种状态之一：领导者Leader、跟随者Follower、领导候选Candidate。
- 一般情况下，系统中会有一个节点称为Leader，其他节点称为Follower。由Leader向各个Follower同步自己的log，log中每一个entry都代表着系统的一个事务。（日志复制）
- Leader需要定时向Follower发送心跳包来证明自己还存活，一旦Follower等待超过了timeout而还没收到新的心跳包则认为Leader节点已经宕机，此时Follower就会将自己的状态变为Candidate参与下一轮Leader的竞选。（领导选举）
- 集群中的成员可能出现变化，一些新的节点会加入，或者一些性能不好的节点会被移出。Raft实现了联机更新成员的功能。（成员变更）

Raft算法-集群节点的状态转换图



Raft算法-领导选举-任期Term

- Raft引入了“任期”（Term）概念
  - Raft将时间分割成任意长度的任期，任期由任期号标识，单调递增。任期从一次领导选举开始。
  - 一个节点在尝试竞选成为Leader时，会指定自己要成为哪个任期的Leader。
  - 当任期n的Leader节点宕机，等待超时的Follower将自己的状态变为Candidate，申请成为下一任期的Leader



Raft算法-领导选举

- 所有的服务器初始都处于Follower状态，没有Leader，则各个节点也会等待心跳包到超时并参与选举。
- Follower定时会收到Leader节点所发送的心跳包以证明Leader仍然存活，如果它在一个预定的时间周期内没有收到心跳包，表明Leader节点已经宕掉，这个节点就会开启一轮新的选举。并且：
  - 将当前的任期编号+1；
  - 将自己的状态设为Candidate；
  - 投票给自己并向其余的服务器发送 <RequestVote> 消息。

Raft算法-领导选举

领导选举有三种可能：

① 赢得选举

- 在一个给定任期内一个节点只能给一个Candidate投票。
- 如果收到大多数节点的投票后，该节点就成为了该任期的Leader，并发送<AppendEntries>心跳包给其余的节点服务器

② 如果收到一条 <AppendEntries> 消息

- 如果消息中的任期编号大于当前任期，意味着某个节点已当选为Leader，则将自己的状态设为Follower；否则，丢弃该消息。

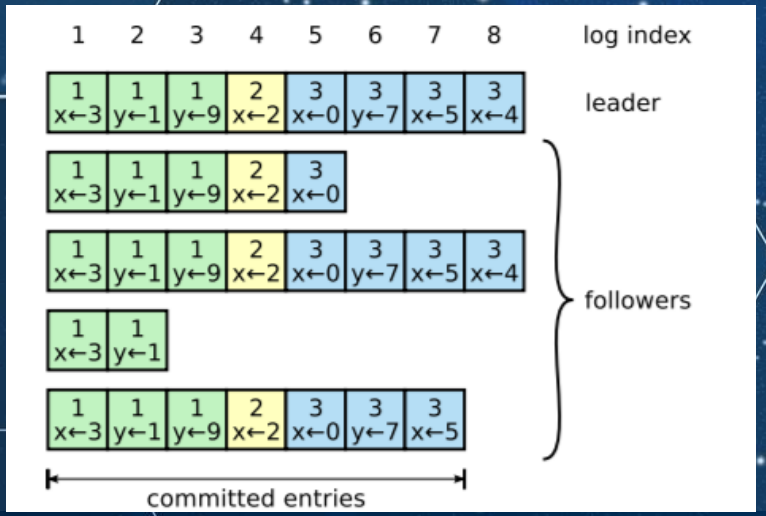
③ 选举失败

- 如果出现多个节点同时竞选Leader，可能导致其他节点投票分散，任一参选节点都无法获得大多数投票，选举失败。
- 每个Candidate超时后重启新的选举，为防止多个Candidate节点恰好同时参与竞选-同时超时-同时进行下一轮竞选这样的死循环出现，每个Candidate的超时时间是在一个范围内取的随机值。

Raft算法-日志复制-日志格式

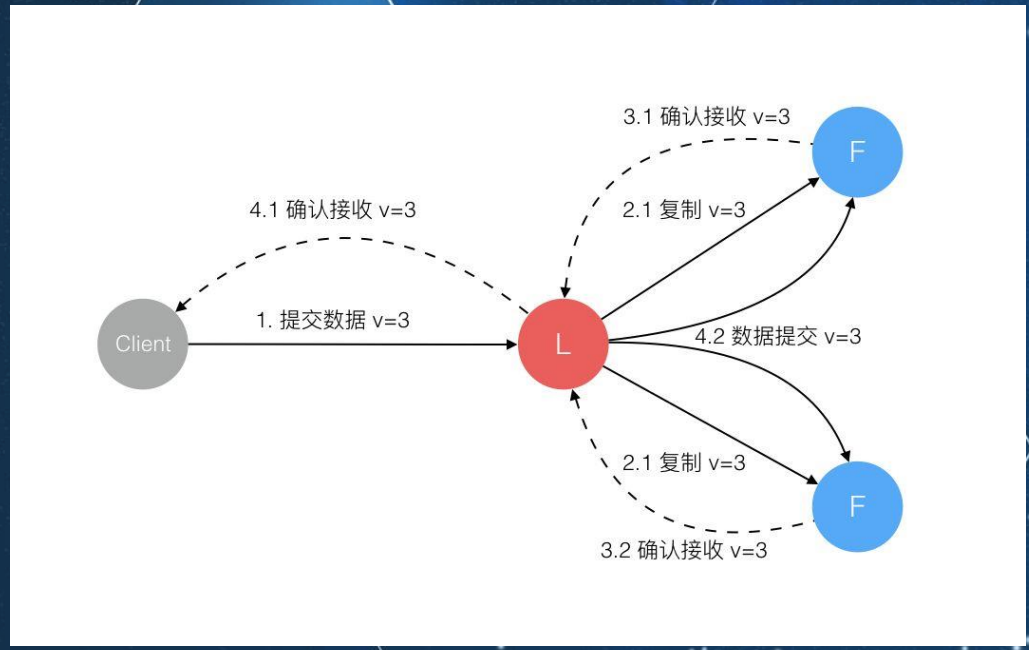
- Raft的日志log的形式上大致如图所示
  - 每一行代表一个节点的log，行内每一个方格代表一个log entry。
  - 每一行从左到右log index递增，每个log entry上都包含着自己所在的任期号（由leader生成日志项时生成。
  - 如果不同节点上的两条log entry有相同的任期号和index，则其内容一定是相同的。
    - 考虑一个任期内只会有一个leader，则在任期号相同的情况下，两条log entry是由相同的leader构造的，如果其index也一样则显然是同一个entry
    - 如何保证一个任期只有一个leader，这由其领导选举的过程来保证
  - 每个节点都会维护一个commitIndex日志条目索引。

Log Entry	
term	term when entry was received by leader
index	position of entry in the log
command	command for state machine



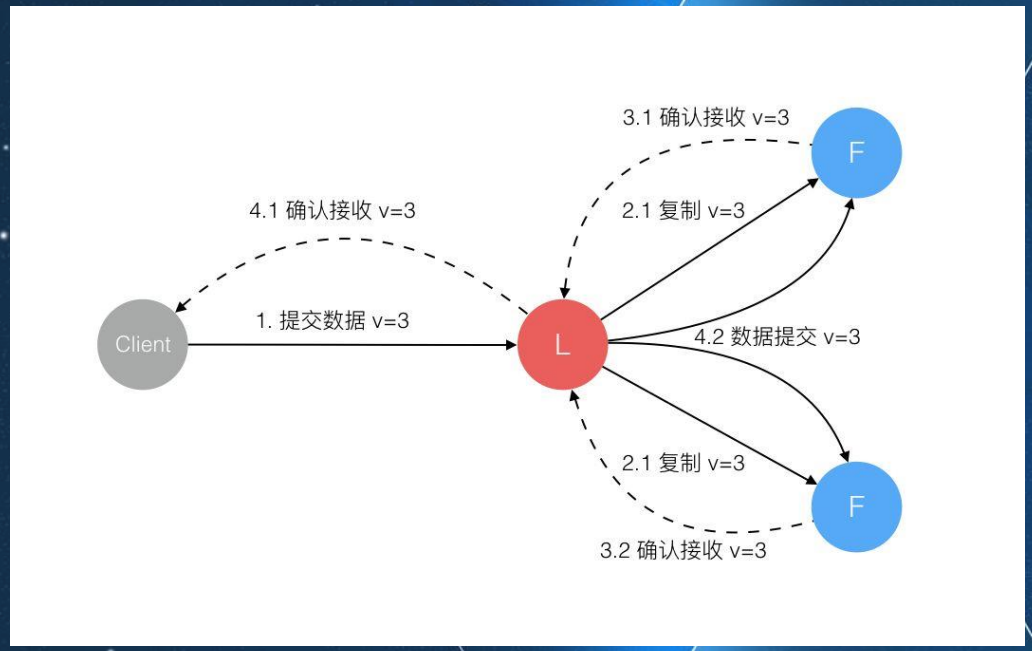
Raft算法-日志复制

- 每当Leader收到了来自client发来的事务请求（1），就将其构造成一条新的log entry加入自己的log中。
- Leader会将新entry用<AppendEntries>消息发送给各个Follower（2），如果发送失败则会不断重试。当收到大部分Follower的响应（表示收到了entry，图中（3））则确认该entry以及其之前的所有entry都可以commit。
- leader在确认一条entry可以被commit之后，会通知各follower节点该entry可以被commit(4.2)，并会向client发送响应通知(4.1)，表示这条事务确实已经被接收。
- Entry被commit，代表其中事务所对应的具体内容可以被执行，比如一条事务是一个数据库操作，则当该entry被commit之后，节点才能确定这个操作确实是要执行的。



Raft算法-日志复制-故障

- 阶段1和阶段2.1之前出故障，不影响一致性
- 阶段2.1失败（部分或全部），有一致性问题
- 阶段3.1失败（部分或全部），有一致性问题
- 阶段4.1失败，客户端重新发送请求后提交即可，无一致性问题
- 某些情况会出现网络分区，而导致双领导，原领导降级为Follower



Raft算法-远程过程调用RPC-节点状态数据

- 每个Raft节点的状态数据

Raft节点状态数据	
所有节点上的持久化状态数据：	
(保存在磁盘上)	
currentTerm	最新任期数，第一次启动时初始化为0，单调递增
votedFor	当前任期内，本节点投票目标节点
log[]	日志项，每项中包含状态机命令，日志项编号（第一个编号为1），以及接受到该日志项时Leader的任期
所有节点上的非持久化状态数据：	
commitIndex	处于提交(committed)状态的最高日志编号（初始化为0，单调递增）
lastApplied	处于应用(applied)状态的最高日志编号（初始化为0，单调递增）
Leader的非持久化状态数据：	
(Leader选举后重置)	
nextIndex[]	对于每个其他节点，Leader下次应该向其发送的日志编号（初始化为Leader最新日志编号+1）
matchIndex[]	对于每个其他节点，已经复制的最高日志编号（初始化为0，单调递增）

Raft算法-远程过程调用RPC

RequestVote RPC

由Candidate节点发出，请求其他节点给自己投票。

Arguments:

candidateId	candidate requesting vote
term	candidate's term
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Implementation:

1. If term > currentTerm, currentTerm ← term (step down if leader or candidate)
2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout

AppendEntries RPC

由Leader节点发出，用于复制日志。当增加日志条目数为0是用作心跳heartbeat。

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat)
commitIndex	last entry known to be committed

Results:

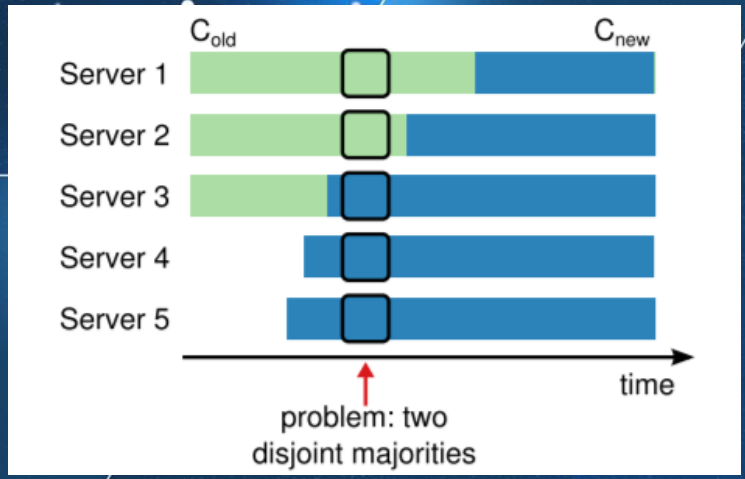
term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Implementation:

1. Return failure if term < currentTerm
2. If term > currentTerm, currentTerm ← term
3. If candidate or leader, step down
4. Reset election timeout
5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
7. Append any new entries not already in the log
8. Advance state machine with newly committed entries

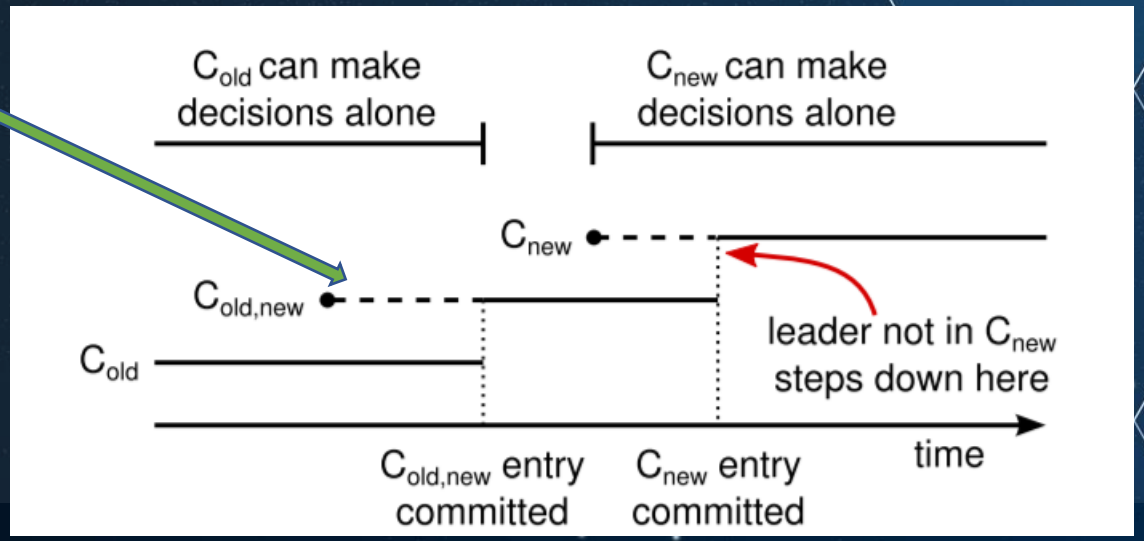
Raft算法 – 集群成员变更

- Raft中leader选举是基于过半成员支持选出的。而这种选举的方式会在集群成员出现变化的时候导致一些问题
  - 如图，集群原来配置有S1S2S3三个节点，新配置中集群里添加了S4S5节点。在新配置还没有完全传到所有旧节点时可能出现一些问题。如图中箭头所指的点。
  - 在这个时间点，S1S2还用着旧配置，于是这两个节点就形成了旧配置中的“大多数节点”，当此时S1请求成为leader而S2同意，则S1会认为自己成为了下一个leader。
  - 而S3已经使用新配置，S3S4S5构成了新配置中的大多数节点，如果此时S3请求成为leader而S4S5同意，则S3也会觉得自己成为了下一个leader。
  - 此时就出现了两个leader。两个相同任期的leader就可能导致“相同index，相同任期而内容不同”的一对log entry出现从而导致系统出现错误。



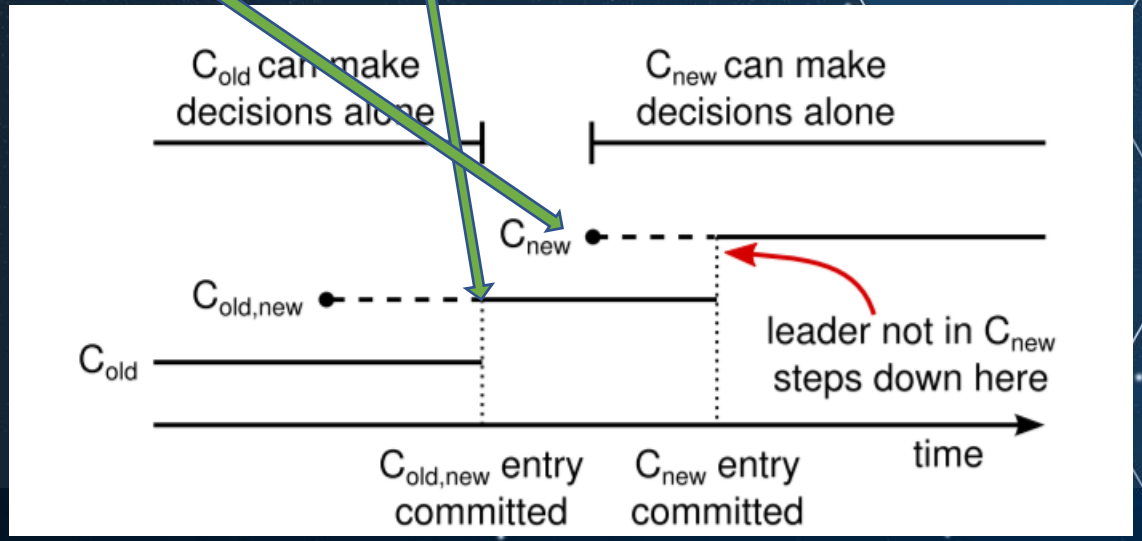
Raft – 集群成员变更

- 为解决此问题，Raft算法使用两个事务来完成一次成员变更。
- 当Leader收到成员变更的请求（从旧配置C-old到新配置C-new），Leader会先构造一个介于二者之间的配置C-old,new，这个配置包含的成员是新旧两个配置的内容。Leader会先将C-old,new作为一个新的entry按照之前的流程同步到各个Follower上。



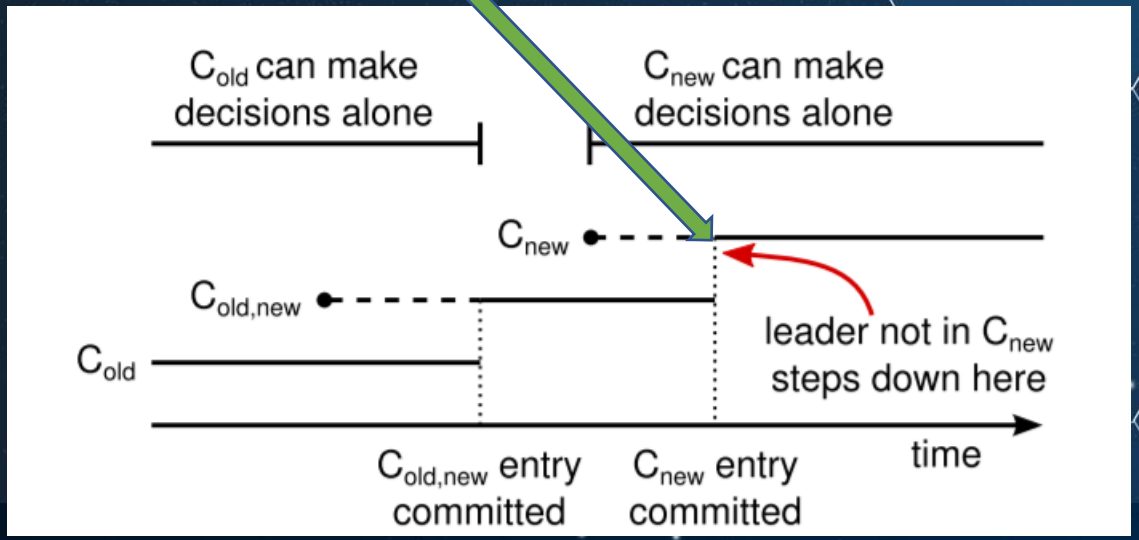
Raft – 集群成员变更

- 为解决此问题，Raft算法使用两个事务来完成一次成员变更。
- 在确认这个中间配置commit了之后，各个节点就按照这个中间配置来行事，即节点需要同时获得新配置和旧配置中大多数人的同意才能成为leader。在这个阶段的leader出现了之后，该leader则会再将新配置C-new作为新entry同步给其他节点。



Raft – 集群成员变更

- 为解决此问题，Raft算法使用两个事务来完成一次成员变更。
- 当C-new配置commit了之后，则各个节点开始按照新配置来运行。旧配置的节点自然离开集群。
- 至此，完成成员变更。



Raft算法-安全性

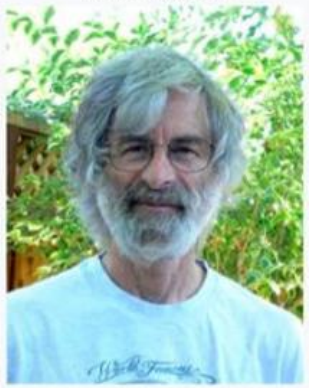
选举安全性	• 在任何时刻只能有一个leader(term).
领导Append-Only	• 日志项不能被覆盖或删除，只能扩展新日志项。
日志一致性	• 不同节点上的两条log entry有相同的任期号和index，则其内容一定是相同的。 • 该index之前的所有日志都是相同的。
领导者完整性	• 对于任意给定的任期号，领导者都包含了此前各个任期所有被提交的日志条目。
状态机安全性	• 如果一个服务器已经在一个状态机上应用了一条log entry，那么所有的服务器都将应用同一个log index的同一entry。

4.3 主流区块链共识算法

- PBFT共识算法
- PoS共识算法
- RPCA共识算法
- PoW共识算法
- DPoS共识算法

PBFT共识算法

Leslie Lamport



Born February 7, 1941 (age 77)
New York City, New York

拜占庭将军问题最早是由Leslie Lamport与另外两人在1982年发表的论文《**The Byzantine Generals Problem**》提出的，他证明了在将军总数大于3f，背叛者为f 或者更少时，忠诚的将军可以达成命令上的一致，即3f+1<=n。算法复杂度为o（n^(f+1)）

1990年发表的论文《**The Part-Time Parliament**》里首次提出Paxos算法。

2013年获得图灵奖。

http://lamport.azurewebsites.net/pubs/pubs.html#byz

Barbara Liskov

Liskov in 2010.

Barbara Jane Huberman
November 7, 1939 (age 78)

Miguel Castro (卡斯特罗)和Barbara Liskov（利斯科夫）1999年发表的论文《**Practical Byzantine Fault Tolerance**》首次提出pbft算法，该算法容错数量也是3f+1<=n,并解决了原始拜占庭容错算法效率不高的问题，将算法复杂度由指数级降低到多项式级，把算法复杂度从o（n^(f+1)）降到o（n^2），使得拜占庭容错算法在实际系统应用中变得可行。该论文发表在1999年的操作系统设计与实现国际会议上（OSDI99）。Loskov是提出著名的里氏替换原则（LSP）的人，2008年图灵奖得主。

拜占庭将军问题最早是由Leslie Lamport与另外两人在1982年发表的论文《The Byzantine Generals Problem》提出的，他证明了在将军总数大于3f，背叛者为f或者更少时，忠诚的将军可以达成命令上的一致，即 3f+1<=n 。算法复杂度为o(n^(f+1))。而Miguel Castro和Barbara Liskov在1999年发表的论文《Practical Byzantine Fault Tolerance》中首次提出PBFT算法，该算法容错数量也满足3f+1<=n，算法复杂度为o(n^2)。

PBFT共识算法

- PBFT（Practical Byzantine Fault Tolerance，实用拜占庭容错）
  - PBFT可以应用于异步网络，容忍三分之一数量的拜占庭节点，由于相较于之前的拜占庭算法大幅提高了系统的响应效率，因此具有较强的实用性。
  - PBFT算法假设共识过程运行环境是一个异步分布式网络（例如互联网环境）。网络中可能发生消息传输失败、延迟、重新发送等问题，且节点有作恶的可能（故意发送错误消息）。
  - PBFT算法使用数字签名来防止欺骗、重发和检测消息完整性。消息包括公钥签名、消息类型编码、哈希产生的消息摘要等。后续的算法演示设$<m>\sigma_i$表示消息m被节点i签名，D(m)表示消息m的摘要。

PBFT共识算法

- PBFT容错率：设节点总数是n，其中作恶节点有f个，那么剩下的正确节点为n–f个，这意味着只要收到n-f个消息就能做出决定。最坏情况下，这n-f个消息中有f个是由作恶节点冒充的，那么正确的消息就是n-f-f个，根据多数一致的原则，正确消息必须占多数，也就是n-f-f>f，即n>3f。由于节点必须是整数个，所以n最少是3f+1个。采用PBFT算法的系统并非节点数越多越好，要根据f来计算。
- PBFT中的服务被建模为状态机，节点分为主节点和副节点，每个节点上都保存有状态副本。所有副本的状态变迁通过视图（view）的配置更换来进行。在每一个视图中，只存在一个主节点。主节点可以简单地由 视图编号 mod |节点数量| 来决定。每个节点上的副本状态包括服务状态、消息日志（记录副本收到的消息）和当前视图编号。

PBFT共识算法

- 两个限定条件：

  ①节点是确定性的，给定状态和参数相同的情况下，执行结果相同；

  ②所有节点必须从相同状态开始执行。

  即使存在失效副本节点，算法对所有非失效副本节点的请求执行总顺序达成一致。

- 算法的五个阶段：

  ①请求Request

  ②预准备Pre-prepare

  确保同一视图中请求发送的时序性

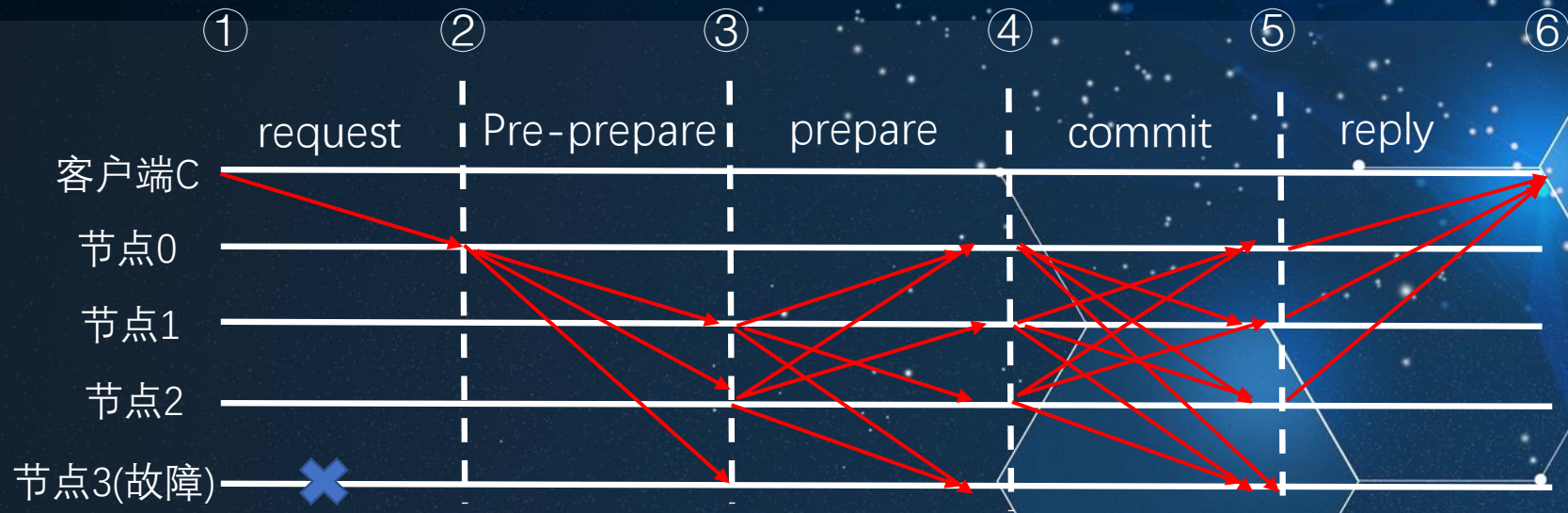
  ③准备Prepare

  ④确认Commit

  确保不同视图之间的确认请求是严格排序的

  ⑤回复Reply

PBFT共识算法



举例：如上图所示，n=4，f=1。

①客户端C发送请求给主节点0，

②主节点0把请求广播给其他副节点1、2、3。

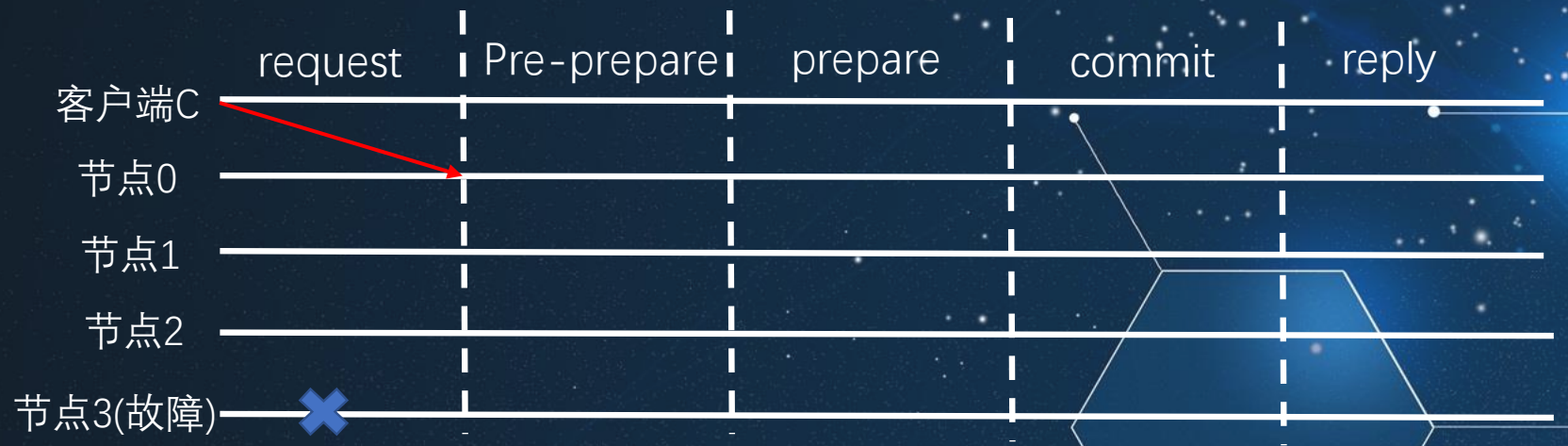
③由于节点3失效（故障或者作恶，图中红线仅代表诚实节点的信道），节点1、2广播消息。

④节点0、1、2搜集到2f（不包括自己）的消息之后，在本地达成了commit条件，然后将消息广播给全网。

⑤节点0、1、2搜集到2f+1 （包括自己）个commit的消息，执行请求操作，并给客户端回复消息。

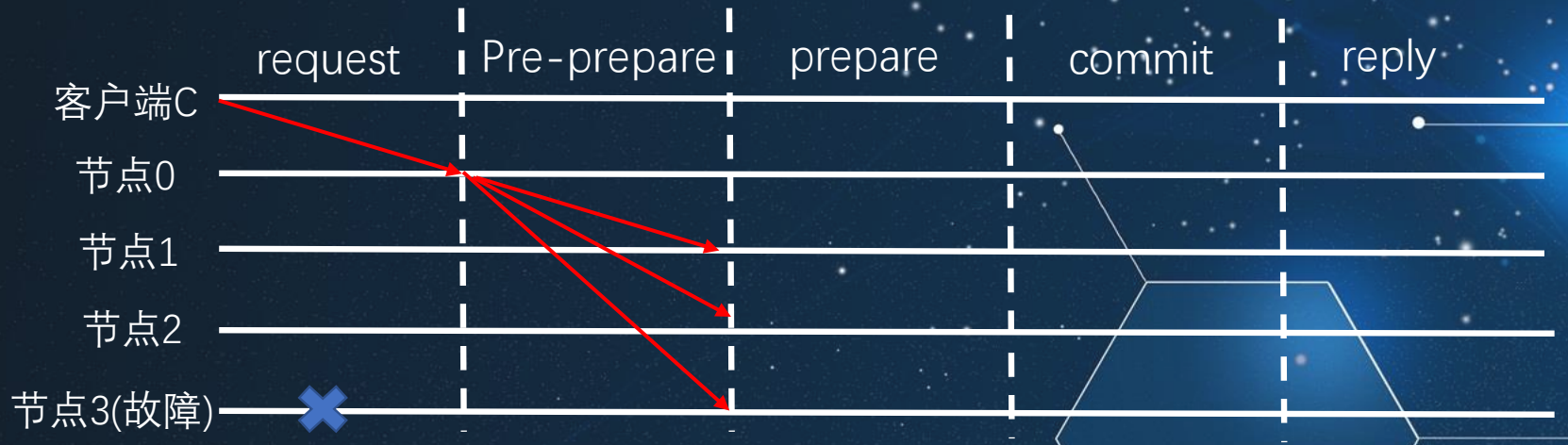
⑥客户端搜集到3个消息，大于f，故请求成功。

PBFT共识算法



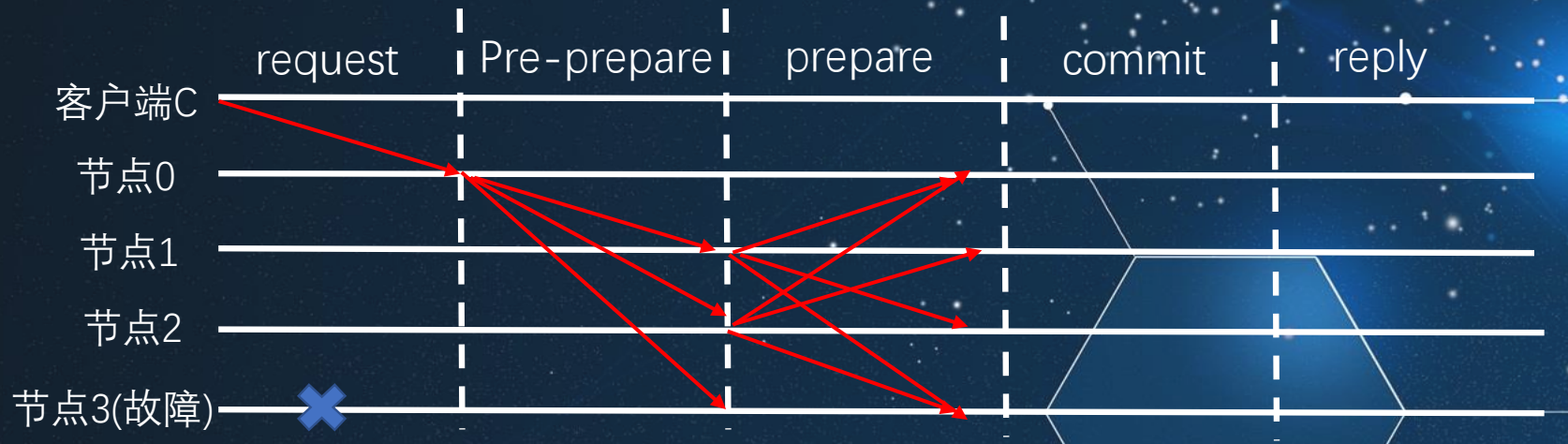
- 在算法开始阶段，主节点由 p = v mod n计算得出，随着v（视图编号）的增长可以看到p不断变化。图中，节点0为第一轮视图的主节点。
- 第一阶段（请求）：客户端签名发送消息<REQUEST, o, t, c>σ_c给主节点0，o为操作；t为时间戳，用于保证请求只被执行一次，也可以用于比较操作执行顺序，例如t可以设置为客户端发送请求时的本地时钟；c为客户端编号。

PBFT共识算法



- 第二阶段（预准备）：主节点构造消息< <PRE-PREPARE, v, n, d>σ_0, m >广播到集群中的其它节点，同时消息追加到消息日志中。
- PRE-PREPARE标识当前消息所处的协议阶段。
- v标识当前视图编号，n为主节点分配给所广播消息的一个唯一递增序号，m为客户端发来的消息，d为m的数字摘要。

PBFT共识算法

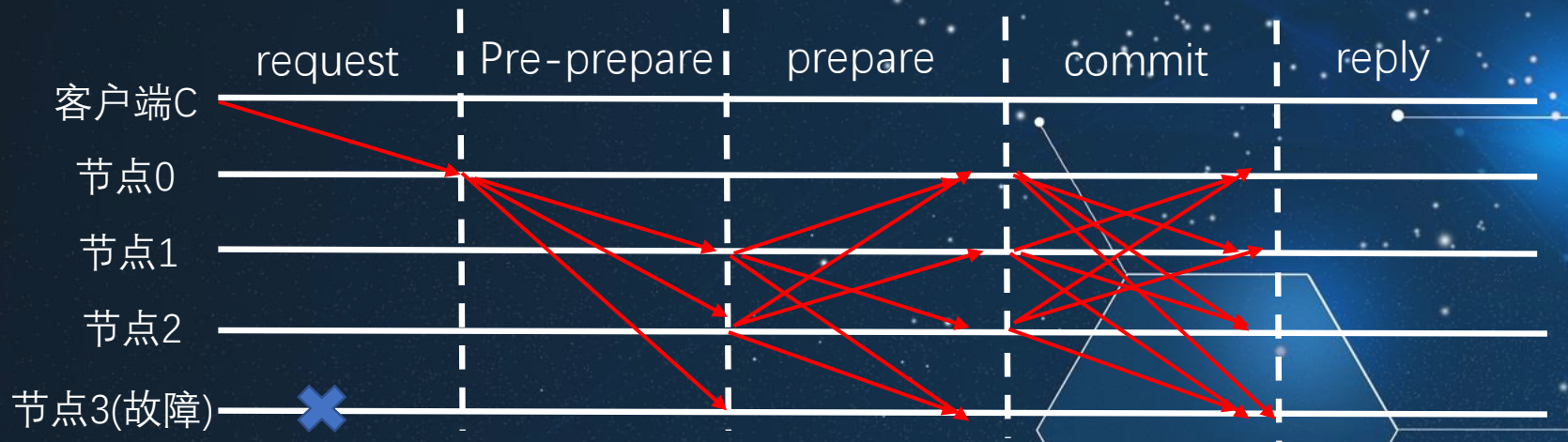


第三阶段（准备）：副本收到主节点请求后，会对消息有效性进行检查，检查通过会追加在消息日志中，并广播消息<PREPARE, v, n, d, i>σ_i，其中i是本节点的编号。

对消息的有效性有如下检查：

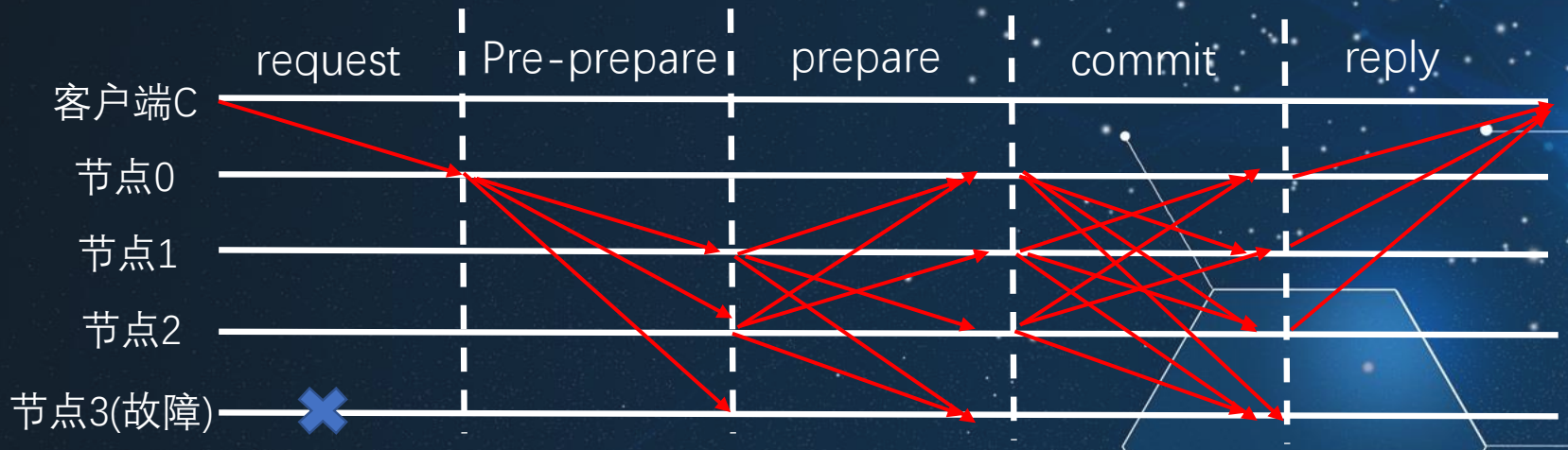
1. 检查收到的消息体中摘要d，是否和自己对m生成的摘要一致，确保消息的完整性。
2. 检查v是否和当前视图v一致。
3. 检查之前是否接收过相同序号n和v，但是不同摘要d的消息。
4. 检查序号n是否在水线h和H之间，避免快速消耗可用序号。（防止作恶节点消耗序号空间）

PBFT共识算法



第四阶段（确认）：副本收到2f（不包括自己）个一致的PREPARE消息后，会进入COMMIT阶段，并且广播消息< COMMIT, v, n, D(m), i >σ_i 给集群中的其它节点。在收到PREPARE消息后，副本同样也会对消息进行有效性检查，包含上一阶段介绍的1、2、4三个检查步骤。

PBFT共识算法

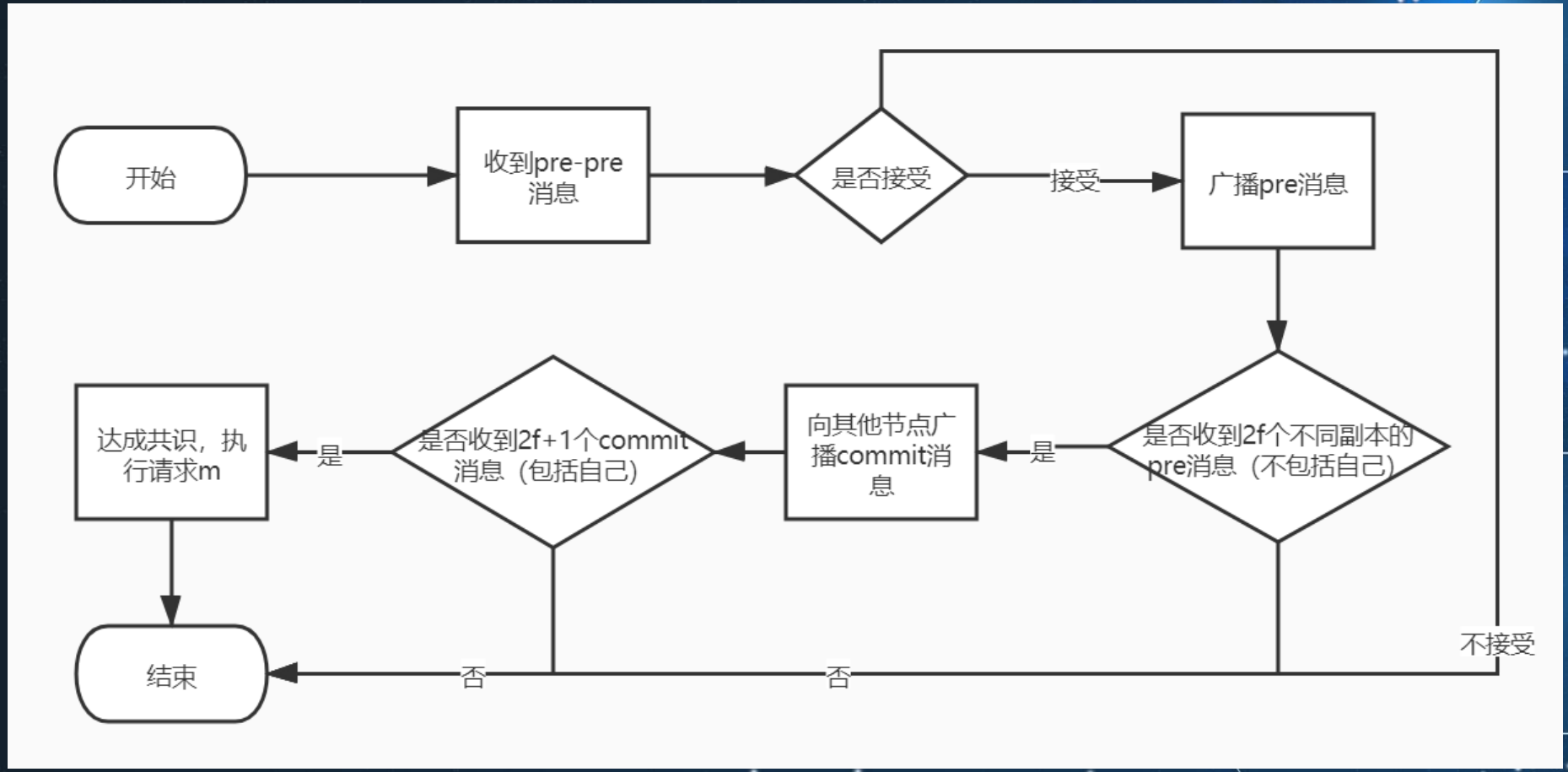


第五阶段（回复）：副本收到2f+1（包括自己）个一致的COMMIT消息后且已经没有序号小于n的请求，则执行m中包含的操作（保证多个m按照序号n从小到大执行），执行完毕后发送消息<REPLY, v, t, c, i, r>σ_i给客户端。v为视图编号，t为时间戳，c为客户端编号, i为节点编号，r为操作结果。

- 客户端在收到回复后要进行签名验证、时间戳比较和操作结果r比较。当搜集到 f+1 个一致结果的回复之后才能确定执行结果。如果在客户端给主节点发送请求之后的一定时间内没有响应，客户端会广播请求，所有副本会进行响应。如果响应结果发现是主节点失效，则会通过视图更换（ ViewChange ）来切换主节点。

PBFT共识算法

核心三阶段流程图



PBFT共识算法

检查点消息（Checkpoint消息）及其作用

- Checkpoint(检查点)：当前节点处理的最新请求序号。前面已经提到主节点收到请求会给请求消息编号。比如一个节点正在共识的一个请求编号是101，那么对于这个节点，它的 checkpoint 就是101。
- stable checkpoint (稳定检查点)：节点i发送<CheckPoint, n, d, i>σ_i给其他节点，当收到了2f+1个验证过的CheckPoint消息，比如系统有 4 个节点，三个节点都对n是213达成了共识，那么stable checkpoint=213。
- stable checkpoint 作用：最大的目的是减少内存的占用。因为每个节点的消息日志记录下之前共识过什么请求，随着系统运行，日志数据会越来越大，所以应该有一个机制来实现对数据的删除。例如，现在的稳定检查点是 213 ，那么代表 213 号之前的记录已经共识过，所以之前的记录就可以删掉了。

PBFT共识算法

ViewChange（视图更改）事件

当主节点挂了（超时无响应）或者副节点集体认为主节点是问题节点时，就会触发ViewChange 事件，ViewChange 完成后，视图编号将会加 1 。下图展示 ViewChange 的流程图：

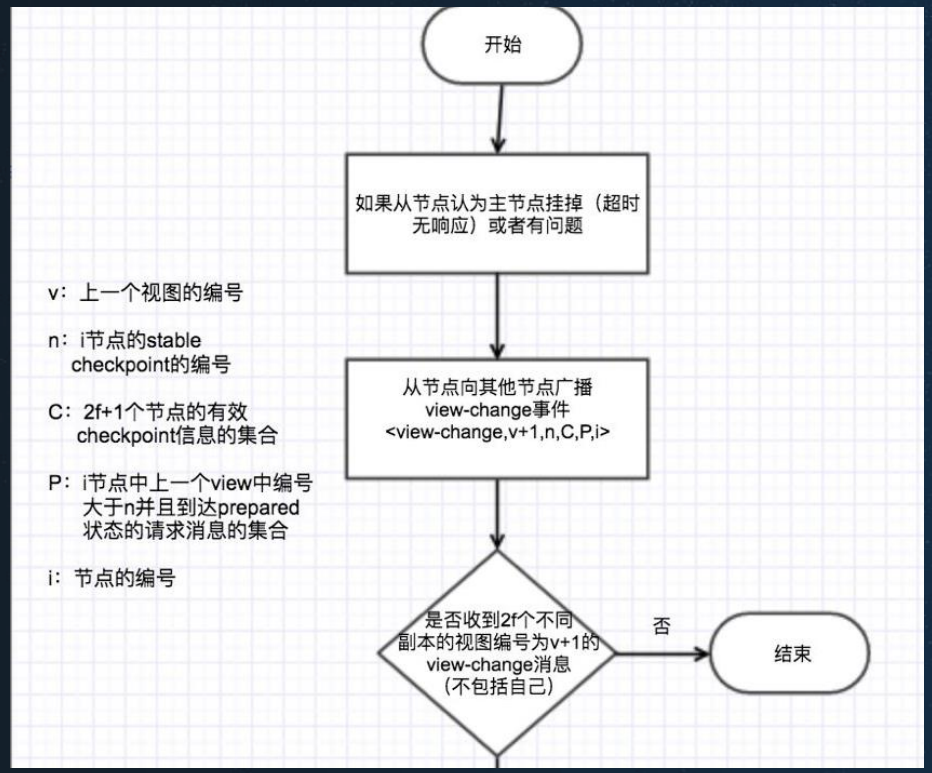


图1

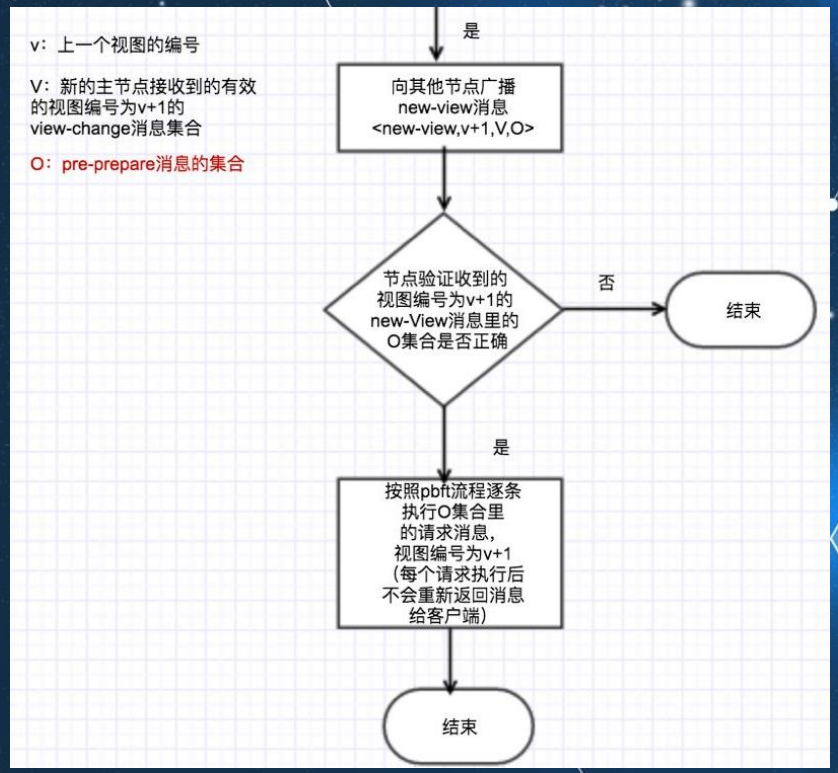


图2

PBFT共识算法

Raft与PBFT对比

对比点	Raft	Pbft
适用环境	私链	联盟链
算法通信复杂度	O（n）	O（n^2）
最大故障和容错节点	故障节点：2f+1<=N	容错节点：3f+1<=N
流程对比	1.初始化leader选举（谁快谁当） 2.共识过程 3.重选leader机制	1.初始化leader选举（按编号依次轮流做主节点） 2.共识过程 3.重选leader机制

PBFT是在联盟链共识节点较少的情况下BFT的一种解决方案。

PBFT共识算法-总结

- 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有链，并且可以在一个不可信的网络里解决拜占庭容错问题，TPS可以远大于PoW。

PoW共识算法

工作量证明最早是为防止服务和资源滥用，或者拒绝服务攻击等场景而提出的一种经济对策。一般要求证明方在使用服务或资源之前，首先完成具有一定难度或者适当工作量的复杂运算，这种工作量对于证明方是“昂贵的”且“没有捷径”的，但对于验证方是快速和简单的；

2008年，中本聪设计了区块链的PoW共识算法，区块链系统中的稀缺资源是“区块记账权”以及随区块发行的比特币奖励。PoW共识机制通过引入分布式节点的算力竞争来作为工作量证明，利用其算力来完成大量的哈希函数计算工作，以便选出每个10分钟时间窗口的唯一“记账人”，从而保证区块链账本数据的一致性和共识的安全性；

工作量证明的核心技术是哈希，哈希函数具有单向性、随机性、定长性和定时性等特点，比特币中选择使用SHA256哈希算法，矿工需要找到区块头中的一个随机数Nonce，使区块头的两次哈希结果满足以n个0开头，这一过程目前没有比穷举法更好的算法，因此从概率上，一般需要16^n次哈希才能找到结果。

PoW共识算法

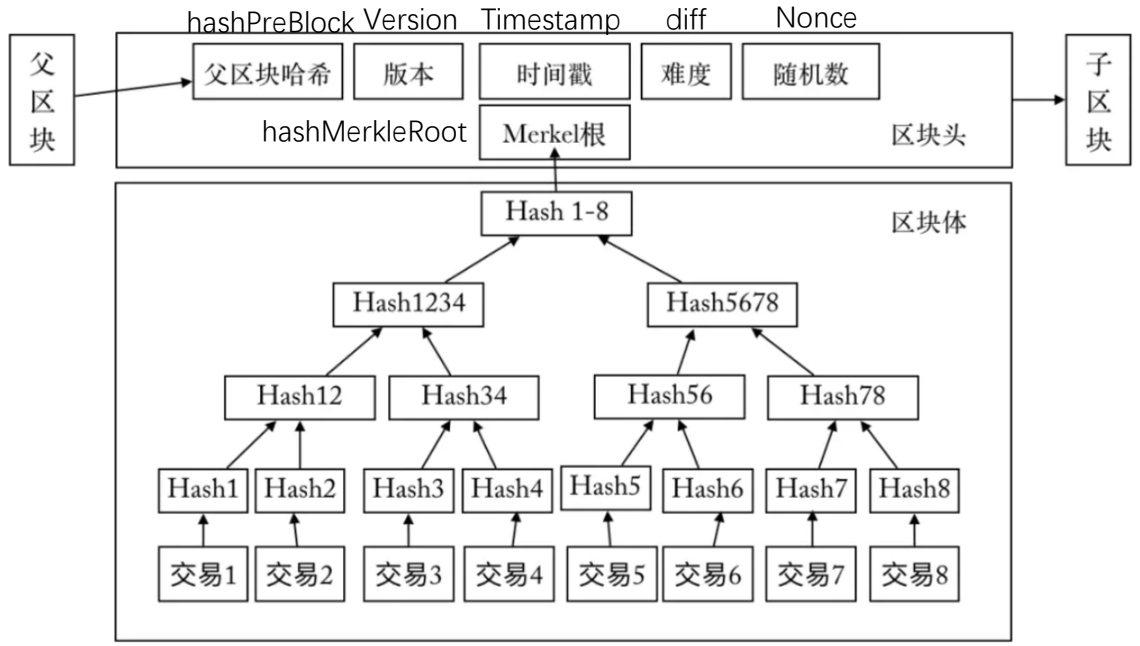
- 比特币中的PoW：

三大要素：PoW函数、区块信息、难度值；

PoW函数定义为：

$$\mathrm{F}_{\mathrm{diff}}(BlockHeader) \rightarrow \mathrm{SHA256}\left(SHA256(BlockHeader)\right) < \frac{MaxTarget}{diff}$$

其中MaxTarget为比特币系统的最大目标值，难度值diff为正实数，diff越大，不等号右边的式子值越小，前导0越多，计算难度越大；diff会被系统动态调整，使比特币的出块速度大致稳定在10分钟；



比特币区块结构图

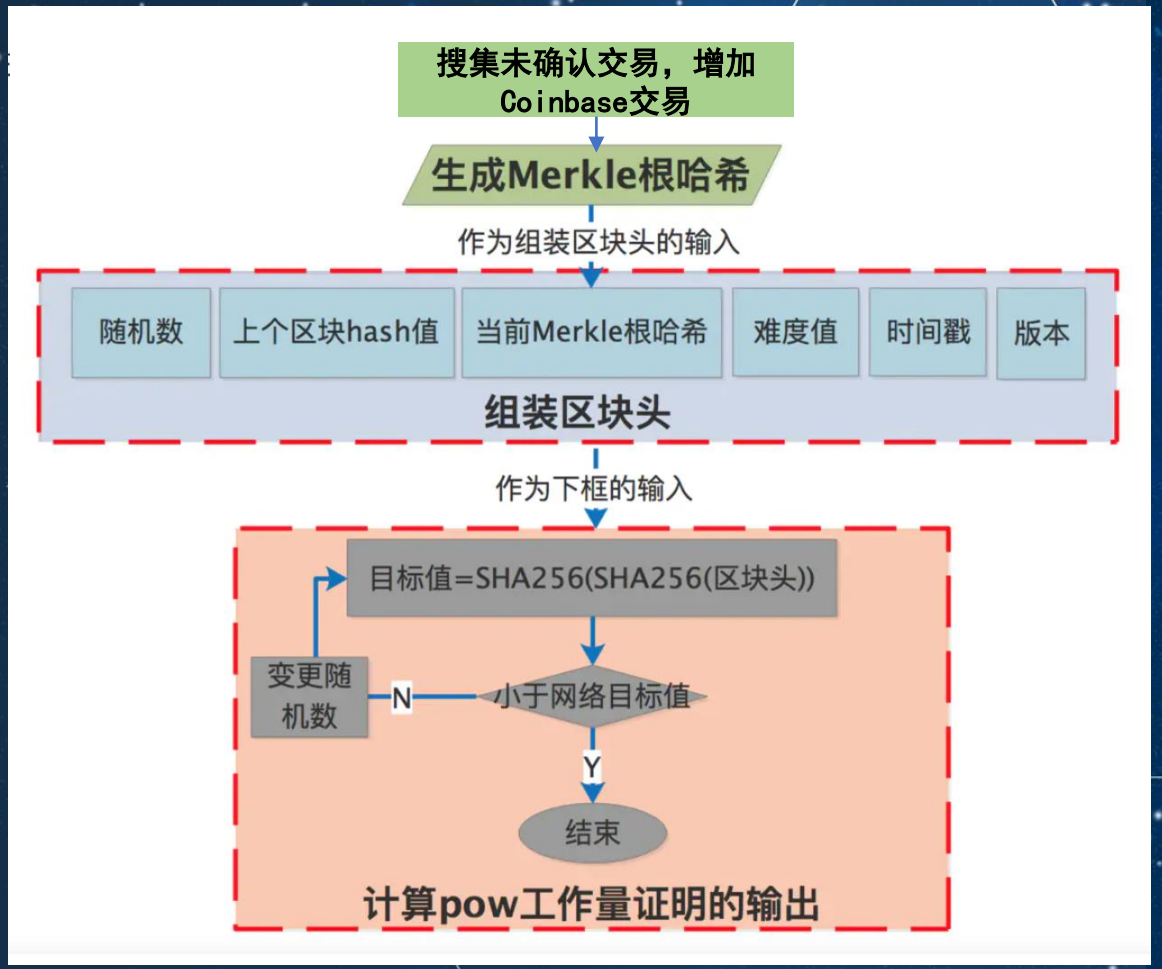
最大目标值：

```
0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

PoW共识算法

- 比特币中的PoW共识过程：

1. 每个比特币节点搜集当前时间段（最近10分钟）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根计入区块头，并填写区块头的其他元数据，其中随机数Nonce置0。这个区块头就是PoW函数的输入数据；
3. Nonce加1；计算当前区块头的两次SHA256哈希值，如果小于目标哈希值，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根后搜索



PoW共识算法

从统计意义上讲，当节点拥有全网n%的算力时，该节点总是有n%的概率首先找到合理的随机数。

比特币区块链系统的安全性和不可篡改性是由PoW共识的强大算力保证的。当区块头目标哈希值越小，成功找到合适随机数并挖出新区块的难度越大，任何对于区块数据的攻击或篡改都必须重新计算该区块以及其后区块的SHA256难题，且计算速度必须使得伪造链长度超过主链，这种攻击难度形成的成本将远超其收益。

PoW共识算法

- PoW的优势
  1. PoW共识的架构简明扼要、有效可靠；
  2. PoW可以实现某种意义上的公平性，即投入算力越多就可以等比例地增加越多的获胜概率；
  3. PoW可以有效抵御51%攻击，攻击者必须拥有超过整个系统51%的算力，才有可能篡改比特币账本，这使得攻击成功的成本变得非常高昂，难以实现。

- PoW的缺陷
  1. 强大的算力造成了极大的资源浪费（电力）；
  2. 长达10分钟的交易确认时间使其相对不适合小额交易的商业应用；
  3. 矿机和矿池的出现，使得算力逐渐集中在大型矿池手上，有违去中心化的初衷。

PoS共识算法

随着比特币系统算力的提升和挖矿设备的专业化，特别是AISC矿机和大型矿池的出现，使得矿工群体逐渐从持币者群体中独立出来，形成了完全不同的两个群体。同时，PoW不仅耗能巨大，而且算力中心化问题日益凸显，整个系统的安全性逐渐不是掌握在使用者手中，而是取决于矿工和矿池。因此诞生了PoS(Proof of Stake,权益证明)共识算法。

PoS共识中，具有最高权益的节点最有可能获得记账权，其权益体现为节点对特定数量货币的所有权。PoS共识更多地是代表一种理念，实际上有多种不同的表现形式，目前主要有以下三种PoS共识算法：

1. PoS+PoW混合共识（PoS 1.0）
2. 纯PoS共识（PoS 2.0）
3. PoS共识的扩展形式（PoS 3.0）

PoS共识算法

- PoS+PoW混合共识（PeerCoin）：

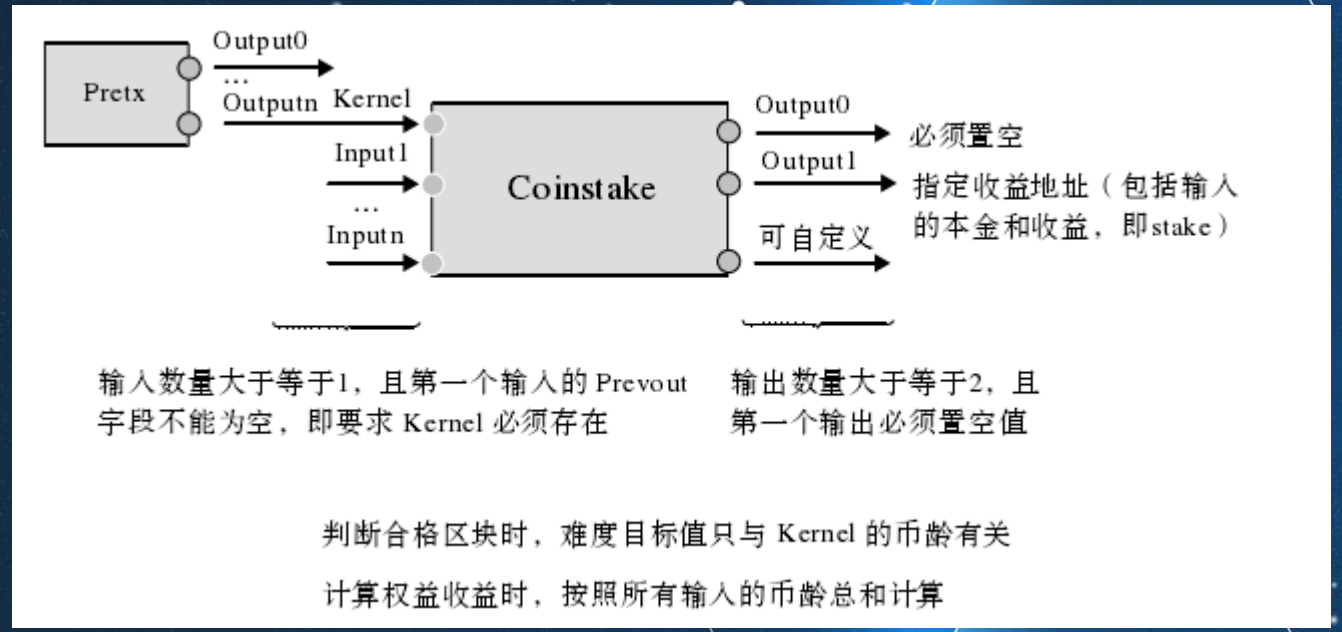
PeerCoin（点点币）中，PoW共识用于早期货币发行，随着系统挖矿难度上升，逐渐过渡到PoS共识；

两类区块：PoW区块和PoS区块；

CoinStake（币权）交易：

为PoS特殊设计的交易：交易输入数量≥1，第一个输入不能为空，称为Kernel；交易输出数量≥2，第一个输出必须置空；PoS区块的标识是第二笔交易为CoinStake(所有区块第一笔交易都是coinbase)；PoS中的挖矿，就是找到合适的Kernel交易；

激励机制：Coinstake全部输入的币龄总和按一定比例转化为矿工收益；



PeerCoin的CoinStake交易结构

PoS共识算法

- PoS+PoW混合共识（PeerCoin）：
  - PoS函数：

$$SHA256\big(SHA256(nStakeModifier + txPrev.block.nTime + txPrev.offset + txPrev.nTime + txPrev.vout.n + nTime)\big) < bnTarget * nCoinDayWeight$$

txPrev: Kernel对应的前一笔交易；
txPrev.block.nTime：txPrev交易所在区块的时间戳；节点有可能通过提前计算预估到未来对自己有利的时间戳，而由于一笔交易被纳入区块的时间不能确定，因此使用此参数防止节点利用这种优势提前生成大批交易；
txPrev.offset、*.nTime、*.vout.n：txPrev交易在区块中的偏移量，txPrev的生成时间，Kernel在txPrev中输出下标，这三个参数用于降低网络节点同时生成Coinstake交易的概率；
bnTarget：全网目前目标难度基准值；
nCoinDayWeight：Kernel的币龄；是特定数量的币与最后一次交易到现在的时间长度的乘积；
nStakeModifier：PoS共识的调节器，没有此参数时，当一笔交易确认之后，接受者就能立即知道自己在未来何时能锻造区块；每个区块对应一个nStakeModifier值，并隔一段时间重新计算一次；

PoS共识算法

- PoS+PoW混合共识（PeerCoin）：

◆ 共识流程：

节点首先从自己所有的UTXO中选定一个作为Kernel，构造Coinstake交易，计算两次SHA256哈希值；如果不满足上式，则重新构造Coinstake交易，重构过程中时间戳nTime会改变，同时也可以改变Kernel，以得到不同的Coinstake交易。如此重复下去，知道获得合格区块为止。

由上可知，由于时间戳以秒为单位，因此共识的搜索空间大大减少，同时Kernel的币龄是影响找到合格区块的最大因素；

区块生成后，每个区块中的交易都会将其消耗的币龄提交给区块，获得最高消耗币龄的区块将会被选中成为主链；

◆ 权益激励：

$$stakeReward=（0.01\times nCoinAge/365）\times COIN$$

其中nCoinAge是Coinstake所有输入的币龄总和，这部分收益奖励给矿工；

PoS共识算法

- 纯PoS共识（Nextcoin未来币）：

NXT是第一个纯粹应用PoS机制运行的数字货币，基于账户结构而非UTXO，NXT中不存在挖矿，10亿NXT一次性创建在第一个区块中，之后网络靠交易费用来维护；NXT采用透明锻造机制，通过指定锻造下个区块的节点，其余节点将交易发到此节点上，来缩短交易时间；

- ◆ PoS函数：

$$hit < baseTarget \times effectiveBalance \times elapseTime$$

hit：每个区块有一个生成签名(generationSignature)字段，用户使用自己的私钥对上一区块的generationSignature进行签名，结果进行SHA256哈希，哈希结果的前8字节作为hit变量；

baseTarget：全网难度基准值；

effectiveBalance：账户有效余额，账户中具有铸币权利的货币余额；

elapseTime：当前时间与上一区块的时间间隔；

PoS共识算法

- 纯PoS共识（Nextcoin）：

显然，当一个新区块生成后，每个用户锻造下一区块的hit值已经成为常量，因此用户不需要挖矿，只需等待时间推移至不等式成立即可锻造区块；对于用户，其有效余额越大，同等条件下更有可能获得锻造区块的机会；NXT规定优先选择最早生成的区块；

◆ 区块锻造流程：

用户必须实时在线，当网络上有新区块产生时，每个账户立即计算自己的hit，然后根据公式elapseTime=hit/(baseTarget×effectiveBalance)计算得知自己锻造区块的期望时间值，并将这个时间广播给其他节点。当全网每个节点都知道其他节点的期望时间时，也就知道下个区块优先由哪个节点锻造，并将交易发给该节点。该节点在自己的时间窗口锻造好区块后立即广播全网，其他节点检验新区块是否有效。

PoS共识算法

- PoS的优缺点：

◆ 优点：

1. 有效降低能源消耗；
2. 避免算力集中问题；

◆ 缺陷：

1. 存在“富者更富”的马太效应问题；
2. 存在无利害关系等安全性问题；无利害关系攻击，即由于PoS中挖矿代价较低，当链出现分叉时，对于一个节点来说，其利益最大化的方法是在所有链上挖矿，因此可能会导致链分叉越来越多；

DPoS共识算法

DPoS（Delegated Proof of Stake，委托权益证明）即通过共识节点的权益投票将区块数据的记账权和区块链参数的配置权赋予特定的少数代表节点，从而实现公平和民主的共识过程和区块链治理，并解决PoW的能源消耗和PoS的无利害关系攻击等问题。DPoS通常包含见证人选举和产生区块两个阶段。

- 见证人选举：

DPoS共识过程中，股东节点可以将其持有的股份权益作为选票授予一个代表，称为见证人。股东节点的投票权重与其持币数量成正比。DPoS选择得票数最多且有意愿的N个节点进入董事会，轮流对交易进行打包和生成新区块；作为回报，见证人会获得区块奖励、交易费或系统发行的特定奖励，同时，其需要缴纳一定数量的保证金。见证人对股东负责，若其错过签署对应的区块，则股东会撤回选票将其投出董事会，因此，见证人节点通常需保证99%以上的在线时间以实现盈利目标；

DPoS共识算法

DPoS系统中还会通过选举方式产生一组特定的授权代表(Delegates)，这些授权代表有权配置和调整区块链系统的参数，如交易费用、区块大小等。如果大多数授权代表都同意变更提案，则所有股东节点有一段时间来审查变更提案，在此期限内可以罢免授权代表并废止提案；

- 生产区块：

DPoS共识过程中，见证人节点按照预先定义的顺序轮流产生区块。每个区块被签署生产之前，需要验证前一个区块已经被受信任的见证人签署。每个见证人轮流在一个固定时间内生产一个区块，如果见证人没有在其时间段中生产区块，那么该时间段后这个见证人将会被跳过，由下一个见证人生产下一个区块，如此循环。

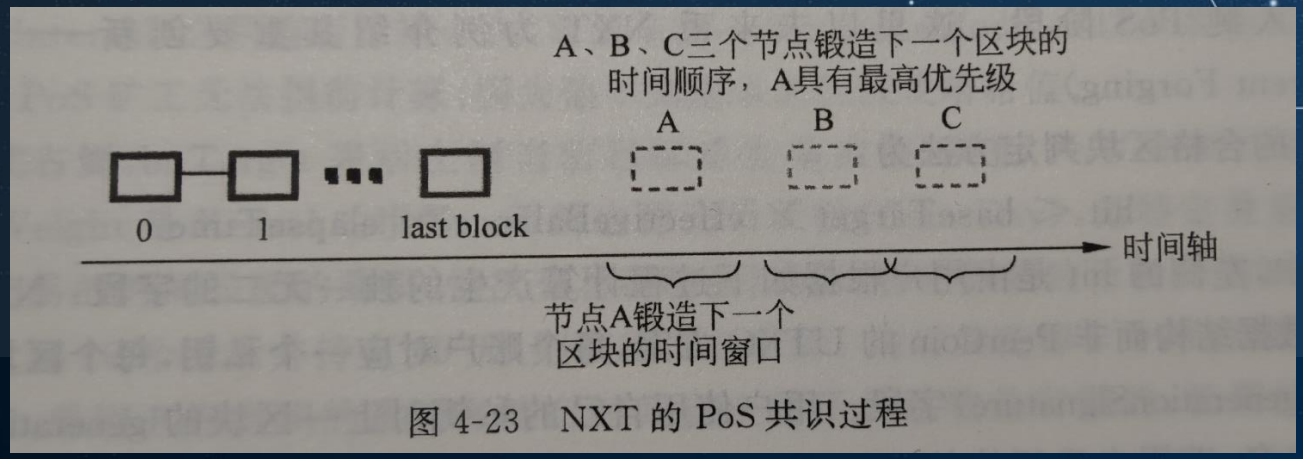


图 4-23　NXT 的 PoS 共识过程

RPCA共识算法

Ripple协议共识算法(Ripple Protocol Consensus Algorithm,RPCA)是Ripple系统及其数字加密货币——瑞波币(XRP)所采用的的共识算法。RPCA通过利用网络中的集体可信的子网络来克服所有节点同步通信的延迟问题，因此具有很低的延迟。

- RPCA基本概念

1. 服务器：是指运行Ripple服务器软件并参与共识过程的验证节点，是被其他服务器加入到信任列表中的节点。非验证节点不参与共识过程；
2. 最新关闭区块(账本)：最近被共识过的区块；
3. 开放区块：正在被共识的区块，开放区块通过共识后，就成了最新关闭区块；
4. 可信任节点列表(UNL)：每个服务器会维护一个可信任节点列表UNL，这里的信任指相信这个列表中的节点不会联合起来作弊。在共识过程中只接受来自信任节点列表中的节点投票；

RPCA共识算法

- 收集交易，形成交易集

1. **收集交易“候选集”**：每个服务器节点从网络中收集客户端发起的新交易以及之前共识过程留下来的旧交易，检查后暂时存放在“交易候选集”中；
2. **做并集**：每个服务器节点对它UNL中服务器的交易候选集做并集，检查所有交易的正确性，并将验证通过的交易打包成提案并广播到网络中；
3. **投票**：当服务器接从网络中接收到一个新的提案后，如果发送提案的服务器在自己的UNL中，则对既在该提案又在自身提案中的交易投一票；验证服务器不断处理和发送提案，但只有达到一定投票比例的交易会进入下一轮，在最终轮中，超该服务器UNL中80%节点投票的交易会被放入交易集中；

- 区块打包

1. **计算哈希**：形成交易集后，服务器对这些交易验证双花，并打包新的区块，对当前区块号、交易集的Merkle根哈希、父区块哈希等内容计算一个哈希；

RPCA共识算法

2. **广播**：每个验证服务器在网络中传播它的区块哈希值；
3. **阈值**：验证服务器收到它UNL广播过来的区块哈希后，只有当其UNL中超过80%的节点对同一区块哈希时，该区块被认为是有效的，区块被关闭；如果某服务器的区块哈希与通过共识的哈希不同，则需要去其他节点拉取正确的区块；如果没有区块哈希超过设定阈值，则重新开始共识过程。

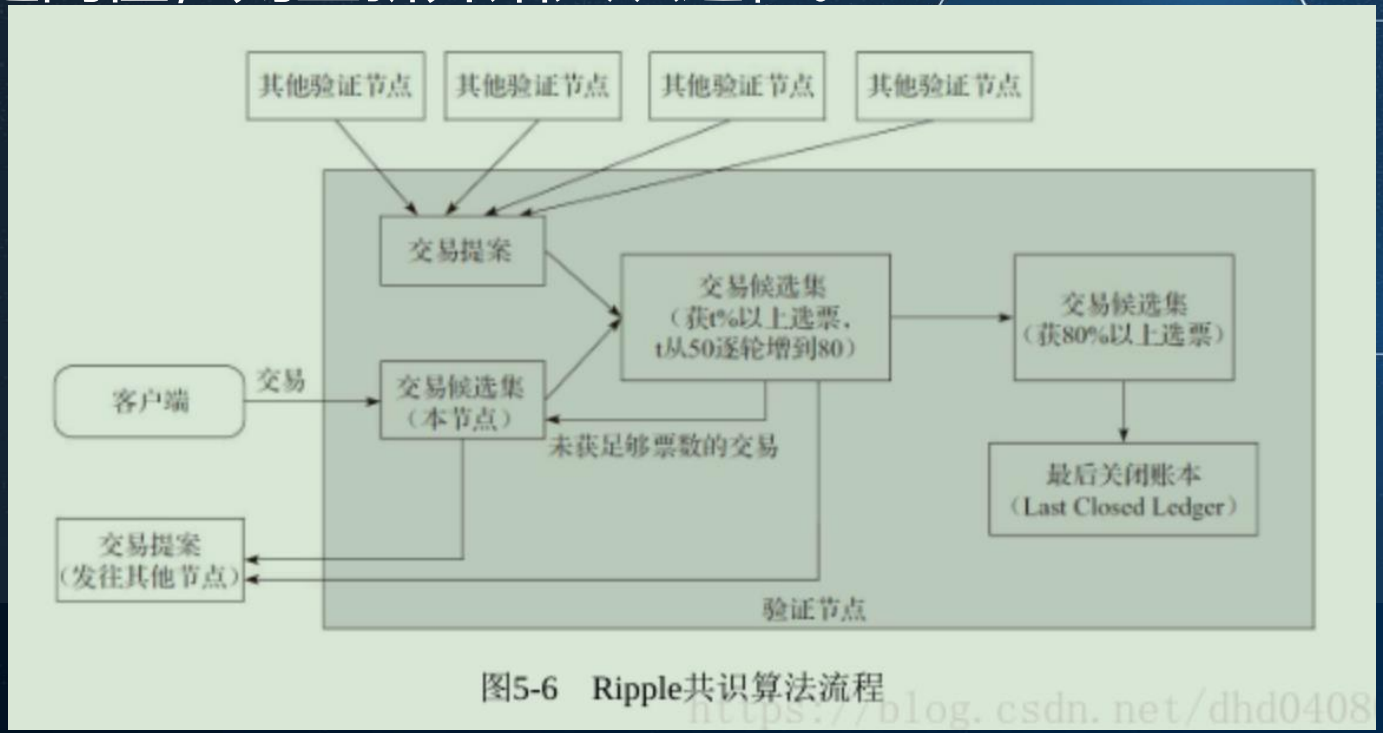


图5-6　Ripple共识算法流程

共识算法的新进展

1. 原生PoW扩展共识算法，为了改进区块链扩展性而对PoW共识做出的改进，有Bitcoin-NG、ByzCoin、Elastico以及ByzCoinX等；
2. 原生PoS扩展共识算法，有Casper投注共识、Ouroboros、PoA等；
3. PoW+PoS混合共识算法，最典型的即2-hop共识算法；
4. 其他共识算法，恒星共识协议、DPoS+BFT扩展共识算法、Algorand等。

谢谢！