区块的物理存储形式可以是文件(如比特币)，也可以是数据库(如以太坊)。相比之下，文件存储更方便日志形式的追加操作，而数据库存储则更便于实现查询操作。区块链系统的交易和区块等基础元素一般都用哈希值加以标识，因此还会选择键值对(Key-Value)数据库作为支撑。比特币系统的每个数据区块主要由区块头(Block Header)和区块体(Block Body)两部分组成，其中区块头记录当前区块的元数据，而区块体则存储封装到该区块的实际交易数据。区块结构：Block Size 区块大小（4B）Block Header 区块头(80B) Transaction Counter 交易数量(1-9B) Transactions 交易列表大小可变；区块头结构：Version 版本号，表示本区块遵守的验证规则,更新于版本升级时(4B)Prev-block(32B) ;Bits 压缩格式的当前目标哈希值(4B); Nonce 32 位随机数(从 0 开始) ; Merkle-root 基于一个区块中所有交易的哈希值(32B); Timestamp 该区块产生的近似时间，精确到秒的 Unix 时间戳(4B) 任何节点都可以通过简单计算获得某个特定区块的哈希值。因此，区块的哈希值可以不必实际存储，而是由区块接收节点计算出来。区块链系统通常被视为一个垂直的栈。创世区块作为栈底的首区块，随后每个区块都被放置在前一区块之上。如果用栈来形象地表示区块依次叠加的过程，就会引申出一些术语,例如通常使用"区块高度"来表示当前区块与创世区块之间的距离，使用"顶部"或"顶端"来表示最新添加到主链的区块。高度为 558859 的区块，记为区块#558859。由此可见，区块一般通过两种方式加以标识，即区块的哈希值或者区块高度。两者的不同之处在于，区块的哈希值可以唯一确定某个特定的区块，而区块高度并不是唯一的标识符：如果区块链发生短暂分叉时，两个或者更多区块可能有相同的高度。比特币网络中的每个完整节点(称为全节点)都会保存一份从创世区块到当前最新区块的本地完整副本。随着新区块的不断产生，完整节点将会逐渐扩展本地的区块链条。为将新区块添加到主链，比特币节点将会检查新区块的区块头并寻找该区块的"前一区块哈希值"，并通过该字段将当前区块连接到父区块，实现现有区块链的扩展。

表 2-3 比特币交易的数据结构

数据项	数据描述	大小
Version No	版本号，目前为 1，表示这笔交易参照的规则	4 字节
In-counter	输入数量，正整数 VI = VarInt	1~9 字节
list of inputs	输入列表，每区块的第一个交易称为"Coinbase"交易	<in-counter>-许多输入
Out-counter	输出数量，正整数 VI = VarInt	1~9 字节
list of outputs	输出列表，每区块第一个交易的输出是给矿工的奖励	<out-counter>-许多输出
lock_time②	锁定时间，如果非 0 并且序列号小于 0xFFFFFFFF，则是指块序号；如果交易已经终结，则是指时间戳	4 字节

交易主要可以分成三部分：元数据、一系列的输入和一系列的输出。除了第一笔 Coinbase 交易是矿工的挖矿收入之外，其他每一笔交易都有一个或多个输入及一个或多个输出。

元数据:主要存放一些内部处理的信息，包含版本号、这笔交易的规模、输入的数量、输出的数量、交易锁定时间，以及作为该交易独一无二的 ID 的哈希值。其他区块可以通过哈希指针指向这个 ID。

3 交易类型：比特币交易通常有三种类型，即生产交易、通用地址交易（可以多对多）和合成地址交易。

合成地址交易是一类特殊交易，其接收地址不是通常意义的地址，而是一个以 3 开头的合成地址。合成地址一般是 M of N 模式的多重签名地址，其中 1WNW3、1WMWN,通常选择 N = 30 合成地址的交易构造、签名和发送过程与普通交易类似，但其地址创建过程需要三对公钥和私钥，其中公钥用于创建地址、私钥用于签名。例如：（1）如果 M = 1 且 N = 3,则 3 个私钥中任意 1 个都可以签名使用该地址上的币，这种私钥冗余可防止私钥丢失，即使其他 2 个私钥丢失也不会造成损失。（2）如果 M = 2 且 N = 3,则 3 个私钥中必须有 2 个同时签名才可使用该地址的币，常见于三方中介交易场景。（3）如果 M = N = 3,则必须 3 个私钥同时签名才可使用该地址的币，常见于多方资产管理场景。

区块链的运行流程：一般说来，从交易的生命周期视角来说，比特币系统的交易流程由如下步骤组成：**(1)** 源节点创建交易，并验证目的节点的地址；**2)** 源节点对交易进行签名加密；**(3)** 源节点将该交易广播至全网其他节点；**(4)** 全网节点接收交易并验证其有效性，直到该交易被全网大多数节点验证和接受；**(5)** 交易暂存于节点内存池，并判断是否孤立交易；**(6)** 交易被打包至节点本地区块中；**(7)** 全网共识结束后，获胜节点将其本地区块追加到主链；**(8)** 交易在主链上被越来越多的后续区块确认。

比特币系统中区块链的运行过程大体上可以分为交易生成、网络传播与验证、共识出块以及激励分配四个主要环节，同时在运行过程中需要处理可能出现的区块链分叉和难度调整

交易被创建后，就由源节点广播发送至比特币网络.由于比特币网络是 P2P 网络，交易的传播采用 Gossip 协议。每个比特币节点在接收到一笔交易之后，都会独立地对交易的有效性进行验证。一个异常交易的传播路径不会超过一个节点，这使得比特币网络能够有效地抵御恶意交易的传播、避免垃圾信息的传播和拒绝服务攻击。比特币交易优先级与难度机制：节点验证交易后，会将有效的交易添加到自己的内存池中。内存池是节点维护的一份未确认交易的临时列表，其主要功能是暂时存放那些已在网络中广播但尚未打包到区块中的待确认有效交易；这些交易将会在内存池中等待矿工将它们打包，封装到下一个区块中。内存池存储在节点本地内存而不是硬盘中，不同节点的内存池可能有很大差异，例如由于节点硬件设施的差异而导致的内存池容量方面的差异以及由于节点启动时间或者重启时间不同而导致的待确认交易集合的差异等。除内存池之外，有些节点还会维护一个独立的孤立交易池。交易并不总是能够按照顺序到达目的节点，有时子交易会在父交易之前到达，从而产生暂时缺失父交易的孤立交易。这些交易将会暂时存放到孤立交易池。孤立交易池默认最多可容纳 100 个孤立交易，这样做的目的是为了防止比特币节点可能发起的拒绝服务攻击。

$$交易优先级=\frac{\sum_{每个输入对应的UTXO}(UTXO\ 交易额 \times UTXO\ 存在时间)}{交易的字节长度}$$

比特币系统采用 0.576 作为交易的基准优先级，如果交易优先级小于该基准优先级则会被收费。这里的基准优先级是由长度为 250 字节的交易，其交易额为 1 个比特币且存在时间为 1 天（即已有 144 个区块的确认）时,按照交易优先级公式所计算得出的"标准"优先级数值。同时，比特币客户端会检查交易的字节长度：如果交易长度超过 10 000 字节但是优先级符合免费标准，则这笔交易可以免费，否则将按照每千字节收费（不足 1000 字节的交易按照 1000 字节计算），单位费用默认为 0.0001 个比特币。交易长度可以根据下式计算：交易字节长度=148x 输入数量+34x 输出数量+10 比特币区块的前 50 个字节将会保留给这些高优先级交易。这种设计同时兼顾了高手续费和高优先级交易，在提高矿工收益的同时，保证了系统不会出现因低手续费而无法进入区块的交易。比特币系统中有一个全局的网络难度。根据比特币协议，从创世区块起，每产生 2016 个区块（大约 14 天），网络中的所有节点都会自动调整难度，以保证比特币网络的出块速度保持在 10 分钟/区块的恒定速度。难度的调整是在每个完整节点中独立自动发生的。比特币的全网难度随区块头目标值（Target）的变化而变化，目标值越小，难度值越大。难度调节公式可以表示为：新难度 = 旧难度 X（过去 2016 个区块的实际时间 / 20160 分钟）另外，为了防止难度变化过快，每个调整周期在调整的时候，如果需要调整的幅度超过 4 倍或者低于 1/4,也只会按照原来难度的 4 倍或 1/4 来调整。比特币区块头中的 Bits 字段标识了当前区块头哈希运算之后要小于或者等于的目标值。目标值是一个 256 位的数值,被压缩成一个 4 字节（32 位）的值（Bits）并存储在区块头中。Bits 可以通过一个计算公式换算成目标值。

以 2019 年生成的第一个区块（高度为 556 459）为例，其 Bits 值为 0x173218a5（十进制为 389159077）这个值存储格式为指数/系数形式，前两位十六进制数字为幂，接下来的六位十六进制数字为系数。在这个区块里,0x17 为幂，（Exponent），而 0x3218a5 为系数（Coefficient），目标值的计算公式为：

Target=Coefficient×256^(Exponet-3)计算出当前区块的目标值之后，就可以利用如下公式进一步计算当前区块的难度 Difficulty 当前块=Target 创世块/Target 当前块 创世区块中，定义 Bits 值为 0x1d00ffff 时，难度值为 1

主链判定与分叉处理：比特币系统采用"最大工作量原则"作为主链判定规则，该原则可体现为多种形式，即当主链出现多个分支区块子链时:•**步骤 1：**如果不同分支的区块高度不同，则选择最长区块高度的分支为主链；•**步骤 2**：如果高度一致，则选择难度系数最大的分支作为主链;•**步骤 3**：如果高度和难度系数均相同，则选择接受时间最早的分支为主链；•**步骤 4：**如果上述所有评判系数均相同，则等待新区块产生并连接到某个或者多个分支、区块高度增加后，重复步骤 1 至步骤 3 直至选出主链。此时，生成新区块的节点即可对当前多个分支子链进行"投票"，并链接至最有可能成为主链的分支子链上。数据层的关键技术：时间戳、哈希函数、默克尔树、非对称加密、数字签名。时间戳(Time Stamp)是区块链不可篡改特性的重要技术支撑，保证数据的可追溯与不可篡改。维基百科将时间戳量化地定义为格林威治时间自 1970 年 1 月 1 日 0 时 0 分 0 秒(北京时间 1970 年 1 月 1 日 8 时 0 分 0 秒)至当前时间的总秒数，其意义在于将用户数据与当前准确时间绑定时间戳可以为区块链数据提供数字公证服务，证明该数据在特定时间点上的存在性；但与传统公证服务不同的是，区块链时间戳不需要可信的第三方。去中心化和共识驱动的区块链系统本身就具有可信第三方的全部特征：例如能够支持安全的在线交易，加盖时间戳的交易具有极难篡改性等。这些特性有助于形成极难篡改和伪造的区块链数据库，从而为区块链应用于公证、知识产权注册等时间敏感的领域奠定了基础。比特币系统采取时间戳服务器的方案。时间戳服务器通过对以区块形式存在的一组比特币交易实施哈希运算并加盖时间戳，并将该哈希值广播到比特币网络中。显然，时间戳能够证实特定区块数据必然在某个特定时间点上是确实存在的，因为只有数据在该时刻存在才能得到相应的哈希值。每个时间戳将前一个时间戳纳入其哈希值，使得每一个随后生成的时间戳都对之前的时间戳进行增强，形成一个完整的、环环相扣的时间戳链条。如果想篡改某一区块的时间戳，必须同时篡改其后生成的所有时间戳，因而链条越长、安全性越好。节点可以随意更改本地时间，因此比特币系统设定两个规则以防止节点的恶意行为。首先，比特币节点会与其连接上的所有其他节点进行时间校正，且要求连接的节点数量至少为 5 个，然后选择这群节点的时间中位数作为时间戳，该中位数时间 (称为网络调整时间)与本地系统时间的差别不超过 70 分钟，否则不会更改并会提醒节点更新本机的时间；由此可见，比特币系统实际上并没有上述理论模型中的全局时钟，而是代之以节点的网络调整时间。其次，合法的时间戳必须大于前 11 个区块的中位数，并且小于比特币节点的网络调整时间+ 2 小时。换言之，比特币节点会拒绝接收时间戳不在此时间范围内的区块。需要注意的是，比特币交易的数据结构里并没有时间戳字段，也就是说没有生成交易的确切时间；当该交易打包封装进区块的时候，也是为交易盖上时间戳的时候，因此区块里交易的时间实际为区块生成的时间。另一方面，区块头的时间戳并非写入之后就固定不变。Nonce 尝试太久无果以后会更新时间戳重新计算。哈希函数的技术特征：•**抗原像性**(Pre-Image Resistance)：也称单向性，对任意预定义的输出数据，无法反推其输入数据。（应用：完整性校验）•**抗第二原像性**(Second Pre-Image Resistance)或称弱抗碰撞性(Weakly Collision-Free) •**强抗碰撞性(**Collision Resistance)：寻找任意两个不同的输入使得它们的哈希值相等在计算上是不可行的。（应用：数据要素管理，完整性校验）•**谜题友好性**(Puzzle Friendly)：对于任意 n 位输出 y 来说，假设 k 是从具有较高不可预测性的高阶最小熵分布中选取的，则无法找到有效方法可在比 2 的 n 次方小很多的时间内找到 x,使得 H(k|x)=y 成立。(应用：共识竞争，pow) •**雪崩效应**：输入数据发生任何细微变化都会导致输出结果发生明显改变。•**定长/定时性**：不同长度输入数据的哈希过程消耗大约相同的时间且产生固定长度的输出。默克尔树：比特币系统采用二叉默克尔树来组织每个区块中的所有交易。这些交易本身并不存储在默克尔树中，而是每个交易作为一个独立的数据块，计算其哈希值并存储在默克尔树的叶子节点。比特币采用 SHA256 算法来对交易数据进行哈希运算，因此默克尔树的每个节点均为 256 位(32 字节)哈希值。一个没有下载完整区块链数据的轻客户端可以通过向其他全节点索要包括从交易 Hk 哈希值沿默克尔树上溯至默克尔根哈希值的哈希序列来快速确认交易的存在性和正确性。通过默克尔路径，基于 spv 技术的轻节点只需很小的开销就可以快速定位一笔交易。简化支付验证(SPV)：用户只需要保存一份最长的工作量证明链条的区块头即可。以比特币为例，无论交易量如何变化，每个区块的区块头始终为 80 字节，目前 60 万个区块的总量约为 45MB；同时，按照每 10 分钟生产一个区块的速度计算，每年新增的存储需求约为 4MB。SPV 技术的设计目标强调的是验证"支付"，而非验证"交易"；前者主要包括验证某一笔交易是否存在于区块链中以及是否获得足够多的网络确认，而后者则需要验证账户余额是否足够支出、是否存在双花交易、交易脚本是否正确等。简单说来,交易验证是要检验这个交易是否合法，而支付验证则仅验证这笔交易是否已经存在。SPV 节点验证支付的具体步骤如下：**步骤 1**：SPV 节点获得待验证交易信息，向区块链网络发起 Merkle Block Message 查询请求。**步骤 2**：其他有完整区块链数据的节点收到请求之后，执行如下步骤。(1) 定位包含该交易的区块。(2) 检查该区块是否属于整个网络中的最长链。(3) 取出所有交易生成默克尔树，利用 getProof 方法获得待验证交易的验证路径。(4) 将验证路径发送回请求源 SPV 节点。**步骤 3**：SPV 节点获得验证路径后，执行如下操作。(1) 同步区块链，确保是整个网络中最长的一条。(2) 先拿默克尔根去区块链中查找，确保该默克尔根哈希是在链条中。(3) 利用获得的验证路径，再进行一次默克尔哈希校验，确保验证路径全部合法，则交易真实存在。(4) 根据该交易所在区块头的位置,确定该交易已经得到多少个确认。非对称加密：非对称加密系统通常使用相互匹配的一对密钥，分别称为公钥和私钥。这对密钥具有如下特点：首先是一个公钥对应一个私钥；其次是用其中一个密钥（公钥或私钥）加密信息后，只有另一个对应的密钥才能解开；最后是公钥可向其他人公开，私钥则保密，其他人无法通过该公钥推算出相应的私钥。例子：RSA，椭圆加密（比特币）

网络层：中心化、多中心化和去中心化网络

以比特币和以太坊为代表的非授权区块链(Permissionless Blockchain)大多采用去中心化网络，其网络节点一般具有海量、分布式、自治、开放可自由进出等特性，因而大多采用对等网络(Peer-to-Peer Net work, P2P 网络)来组织散布全球的参与数据验证和记账的节点。不存在任何中心化的特殊节点和层级结构，每个节点均会承担网络路由、验证区块数据、传播区块数据、发现新节点等功能；P2P 网络的特点：•可扩展性强：节点可以自由加入和退出，P2P 网络的自组织、自配置和自动负载均衡特性，破解了 C/S 模式下中心服务器的性能瓶颈问题。•健壮性好：服务与资源分散于各个节点，不经由第三方，无明显弱节点。•高性价比：有效利用分散于网络中大量节点 的空闲资源

•私密性：信息传输无需通过集中节点，所有节点都具备中继转发能力，大幅提高了通信的匿名性，用户隐私得到保护。•均衡性：资源和处理能力分布于多个节点，避免网络流量过于集中。

P2P 网络分类(根据网络体系结构)•**混合式对等网络**它是 C/S 和 P2P 两种模式的混合，反映了早期网络从 C/S 到 P2P 的过渡。混合式对等网络最具影响力的代表是 Napster,它是一个为音乐迷们提供交流 MP3 文件的平台，最典型的特点就是存在维护共享文件索引与提供查询的服务端（C/S 模式），但具体内容存储在用户硬盘中，内容的传送只在用户节点间进行（文件交换是 P2P 的）。由于服务端的存在，从这个角度来说,早期的 P2P 网络不是完全去中心化的。•**无结构对等网络：**这种网络的特点是无固定网络结构图，无中心节点；每个节点既是客户端又是服务端,节点地址没有统一标准，内容存放位置与网络拓扑无关,对等节点间通过客户端软件搜索网络中存在的对等节点，并直接交换信息。典型的无结构 P2P 网络协议如 Gnutella,它是纯粹意义上的 P2P 网络。•**结构化对等网络：**它以准确、严格的结构来组织网络，一般采用哈希函数将节点地址规范为标准的标识，内容的存储位置与节点标识之间存在映射关系，可以实现有效的节点地址管理，精确定位节点信息。因为所有节点按照某种结构进行有序组织，所以整个网络呈环状或者树状。其最具代表性的经典模型和应用体系如 Chord、Pastry 等。节点类型比特币网络由多种类型的节点组成，其功能集合一般包括网络路由（Network Route,简写为 N)、完整区块链（Full Blockchain,简写为 B)、矿工（Miner,简写为 M)、钱包（Wallet,简写为 W)。每个区块链节点都参与全网路由，同时也可能包含其他功能。根据节点提供的功能不同，主要分为如下几种。拥有全部功能集的称为核心客户端（Bitcoin Core)；不参与挖矿，仅提供完整区块链数据与参与全网路由的节点称为完整区块链节点；拥有完整区块链数据，并参与挖矿与路由的节点称为独立矿工；仅提供钱包功能与参与全网路由的节点称为轻量 (SPV) 钱包。除了这些主要节点类型外，还有一些节点运行其他协议，如挖矿协议，因而网络中还有矿池协议服务器、矿池挖矿节点.Stratum 钱包节点等。拥有完整的、最新区块链数据的节点也称为"全节点"，这样的节点能够独立自主地校验所有交易；只保留区块头数据，通过"简易支付验证"方式完成交易验证的节点为"SPV 节点"/"轻量级节点"，它们没有区块链的完整拷贝。随着比特币生态的发展，比特币 P2P 协议.Stratum 协议、矿池协议以及其他连接比特币系统组件相关协议综合构成了现在的比特币网络，我们称之为"扩展比特币网络"(Extended Bitcoin Network),扩展比特币网络包含了多种类型的节点(如核心客户端、完整区块链节点等)、网关服务器、边缘路由器、钱包客户端以及它们互相连接所需要的 P2P 协议、矿池挖矿协议.Stratum 协议等各类协议。新区块链节点启动后，如何发现网络中其他节点并获知其地址是区块链组网的重要环节，一般通过如下五种方式实现 **1. 地址数据库：**网络节点的地址信息会存储在地址数据库 peers, dat 中。节点启动时，由 addressmanager 载入。节点第一次启动时,无法使用这种方式。**2. 通过命令行指定：**用户可以通过命令行方式将指定节点的地址传递给新节点，命令行传递参数格式形如-addnode Vip〉或者-connect Vip〉。**3. DNS 种子：**当 peers, dat 数据库为空，且用户没有使用命令行指定节点的情况下，新节点可以启用 DNS 种子，默认 DNS 种子有 seed, bitcoin. sipa. be,dnsseed. bluematt. me,dnsseed. bitcoin,dashjr. org,seed, bitcoinstats, com,seed, bitcoin, jonasschnelli. ch,seed. btc. petertodd. Orgo **4. 硬编码地址：**如果 DNS 种子方式失败，还有最后的手段，即硬编码地址。需要注意的是，需要避免 DNS 种子和硬编码种子节点的过载。因此，通过他们获得其他节点地址后，应该断开与这些种子节点的连接。**5. 通过其他节点获得：**节点间通过 getaddr 消息和 addr 消息交换 IP 地址信息数据传播协议一般新节点由初始种子启动，再与相邻节点通信获得更多连接。节点通常采用 TCP 协议，使用 8333 端口与其他对等节点交互。一个通用的区块链网络一般包括如下核心场景. (1)节点入网建立初始连接 (2)节点地址传播发现 (3)矿工、全节点同步区块数据(4)客户端创建一笔交易(5)矿工、全节点接受交易(6)矿工、全节点挖出新区块，并广播到网络中 (7)矿工、全节点接收广播的区块；version 消息和 verack 消息用于建立连接；addr 和 getaddr 消息用于地址传播；getblocks、inv 和 getdata 消息用于同步区块链数据，tx 消息用于发送交易。**1.建立初始连接：**建立连接始于"握手"通信。比特币节点之间的握手过程类似 TCP 三次握手，节点 A 向节点 B 发送包含基本认证内容的 version 消息，节点 B 收到后，检查是否与自己兼容，兼容则确定连接，返回 verack 消息，同时向节点 A 发送自己的 version 内容，如果节点 A 也兼容，则返回 verack,至此连接成功建立。**2.地址广播及发现：**一旦建立连接，新节点将向其相邻节点发送包含自身 IP 地址的 addr 消息。相邻节点则将此 addr 消息再度转发给各自相邻节点，进而保证新节点被更多节点获知。此外，新接入节点还向其相邻节点发送 getaddr 消息，获取邻居节点可以连接的节点列表。**3.同步区块数据**新入网节点只知道内置的创世区块，因此需要同步最新区块。同步过程始于发送 version 消息，该消息含有节点当前区块高度 (BestHeight 标识)。具体而言，连接建立后，双方会互相发送同步消息 getblocks,其包含各自本地区块链的顶端区块哈希值。通过比较，区块数较多的一方向区块较少的一方发送 mv 消息。需要注意的是，mv 消息只是一个清单，并不包括实际的数据。落后方收到 mv 消息后，开始发送 getdata 消息请求数据。需要补充一点的是，SPV 节点同步的不是区块数据，而是区块头，使用 getheaders 消息。**4.交易传播：**交易数据的广播是更为常见的场景：假定有节点要发送交易，那么首先会发送一条包含该交易的 inv 信息给其邻居节点。邻居节点则通过发送 getdata 消息请求 mv 中所有交易的完整信息。如果发送方接收到 getdata 响应信息，则使用 tx 消息发送交易。接收到交易的节点也将以同样的方式转发交易（假定它是一个有效的交易)。比特币系统的交易数据传播协议包括如下步骤：（1）比特币交易节点将新生成的交易数据向全网所有节点进行广播；（2）每个节点都将收集到的交易数据存储到一个区块中；（3）每个节点基于自身算力在区块中找到一个具有足够难度的工作量证明；（4）当节点找到区块的工作量证明后，就向全网所有节点广播此区块（block 消息)；（5）仅当包含在区块中的所有交易都是有效的且之前未存在过的，其他节点才认同该区块的有效性；（6）其他节点接受该数据区块，并在该区块的末尾制造新的区块以延长该链条，而将被接受区块的随机哈希值视为先于新区块的随机哈希值。**5. 检测节点存活** ping 消息有助于确认接收方是否仍处于连接状态。通过发送 ping 消息，可以检测节点是否存活。接收方通过回复 pong 消息，告诉发送节点自己仍然存在。默认情况下，任何超过 20 分钟未响应 ping 消息的节点会被认为已经从网络中断开。**6. Gossip 传播协议** Gossip 协议的特点：在一个有界网络中，每个节点随机地与邻居节点通信，每个节点都遵循这样的操作，经过一番交错杂乱的通信后，最终所有节点状态达成一致。这种最终一致性是指保证在最终某个时刻所有节点一致对某个时间点前的所有历史达成一致。虽然冗余通信，但具有天然的分布式容错优点。Facebook 开发的 Cassandra,就是通过 Gossip 协议维护集群状态。一般而言,Gossip 协议可以分为 Push-based 和 Pull-based 两种。前者工作流程如下。(1) 网络中某个节点 v 随机选择其他 n 个节点作为传输对象。(2) 节点 v 向其选中的 n 个节点传递消息。(3) 接收到信息的节点重复进行相同的操作。Pull-based Gossip 协议则相反：(1) 网络中某个节点 v 随机选择其他 n 个节点询问有没有新消息。(2) 收到询问的节点回复节点 v 其最近收到的消息。为了提高性能，也有结合 Push-Pull 的混合协议。Gossip 协议一般是基于 UDP 协议实现。在区块链领域，Gossip 协议也有广泛应用。面向企业联盟链的超级账本 Fabric 就采用 Gossip 作为其 P2P 网络的传播协议。

数据验证机制：当新区块在区块链网络传播时，每个接收到区块的节点都将对区块进行独立验证，验证通过的区块才会进行转发，以此尽早杜绝无效或者恶意数据在网间传播，预防小部分节点串通作恶导致无效区块被网络接受，尽最大可能保证网络中传播区块的正确性。**(1)** 验证区块大小在有效范畴。**(2)** 确认区块数据结构(语法)的有效性(参见区块结构)。**(3)** 验证区块至少含有一条交易。**(4)** 验证第一个交易是 coinbase 交易(Previous Transaction hash 为 0 且 PreviousTxout-index 为-1)，有且仅有一个。**(5)** 验证区块头部有效性① 确认区块版本号是本节点可兼容的。② 区块引用的前一区块是有效的。③ 区块包含的所有交易构建的默克尔树是正确的。④ 时间戳合理。⑤ 区块难度与本节点计算的相符。⑥ 区块哈希值满足难度要求 **(6)** 验证区块内的交易有效性，具体检查列表如下。① 检查交易语法正确性。② 确保输入与输出列表都不能为空。③ lock_time 小于或等于 INT_MAX,或者 nLockTime 和 nSequence 的值满足 MedianTimePast(当前区块之前的 11 个区块时间的中位数)。④ 交易的字节大小大于等于 100. ⑤交易中签名数量小于签名操作数量上限(MAX_BLOCK_SIGOPS) ⑥ 解锁脚本(scriptSig)只能够将数字压入栈中，并且锁定脚本(scriptPubkey)必须要符合 isStandard 的格式(拒绝非标准交易)。⑦ 对于 coinbase 交易，验证签名长度为 2 至 100 字节。⑧ 每一个输出值，以及总量，必须在规定值的范围内(不超过全网总币量，大于 0)。⑨ 对于每一个输入,如果引用的输出存在内存池中任何的交易，该交易将被拒绝。⑩ 验证孤立交易：对于每一个输入，在主交易和内存池中寻找引用的输出交易，如果输出交易缺少任何一个输入，该交易将被认为是孤立交易。如果与其匹配的交易还没有出现在内存池中，那么将被加入到孤立交易池中。⑪ 如果交易费用太低(低于 minRelayTxFee 设定值)以至于无法进入一个空的区块，则交易将被拒绝。⑫每一个输入的解锁脚本必须依据相应输出的锁定脚本来验证。⑬如果不是 coinbase 交易，则确认交易输入有效，对于每一个输入：•验证引用的交易存于主链；•验证引用的输出存于交易;•如果引用的是 coinbase 交易，确认至少获得 COINBASE_MATURITY(100)个确认；•确认引用的输出没有被花费；•验证交易签名有效；•验证引用的输出金额有效；•确认输出金额小于等于输入金额（差额即为手续费)。⑭如果是 coinbase 交易，确认金额小于等于交易手续费与新区块奖励之和。如果数据有效，则按照接收顺序存储到交易内存池以暂存尚未记入区块的有效交易数据，同时继续向邻近节点转发。如果数据无效，则立即废弃该数据，从而保证无效数据不会在区块链网络中继续传播。

矿池网络协议：Getwork 协议可以认为是最早的挖矿协议——实现区块链数据与挖矿逻辑剥离，拥有完整数据的节点构造区块头（参考区块头)，即 Version. Prev-block. Bits 和 Merkle-root 这 4 个字段必须由节点客户端提供。挖矿程序主要是递增遍历 Nonce,必要时候可以微调 Timestamp 字段。对于 Getwork 而言,矿工对区块一无所知，只知道修改 Nonce 这 4 个字节，共计 232 大小的搜索空间，显然不符合迅猛发展的比特币矿机算力。如果继续使用 Getwork 协议，矿机需要频繁调用 RPC 接口，显然不合时宜。如今比特币和莱特币节点都已经禁用 Getwork 协议，转向更高效的 Getblocktemplate 协议。该协议诞生于 2012 年，其最大的不同点是：Getblocktemplate 协议让矿工自行构造区块。因为由矿工构建 coinbase 交易（参考时间戳最后一段基于 coinbase 字段的随机数扩展)，这种方式所带来的搜索空间巨大。Getblocktemplate 协议虽然扩大了搜索空间，但正常的一次 Getblocktemplate 调用节点都会返回 1.5M 左右的数据，因为要把区块包含的所有交易都交给矿工，数据负载过大。Stratum 协议巧妙解决了这个问题。Stratum 协议是为了扩展支持矿池挖矿而编写的挖矿协议，于 2012 年底出现，是目前最常用的矿机和矿池之间的 TCP 通信协议之一，数据采用 JSON 格式封装。Stratum 协议利用 Merkle 树结构特性，从 coinbase 构造 hashMerkleroot,无须全部交易，只要把与 coinbase 涉及的默克尔路径上的 hash 值返回即可。假如区块包含 N 笔交易，这种方式数据规模将压缩至 log2(N),大大降低了矿池和矿工交互的数据量。Stratum 协议不但保证矿工增加足够的搜索空间，而且仅需很少的数据交互。流程：矿机登记→任务下发→账号登录→结果提交→难度调整

区块链分叉：自然分叉：机器共识过程产生的临时分叉。当两个不同节点近乎同时挖掘出新区块 A、B 并进行广播的时候，就会造成后继区块分叉的发生。区块 A、B 都是有效的，有些节点先接收到 A,其他节点则先接收到 B。节点会把先接收到的区块加入自己的主链，把晚收到的区块加到由主链分叉出来的支链上，临时的分叉由此产生。当后续再发现新区块 C,而 C 所指向的父区块是 A 的时候，网络中的节点有两种选择：原本主链最后一个区块是 A 的节点，自然正常将区块 C 加入主链，而主链最后是区块 B 的节点，则先在支链上将区块 C 加至区块 A 后面，并且将支链切换为主链。随着区块 C 在全网的转播，最终比特币全网节点的区块链又趋于一致，始终保持算力最大的为主链。在比特币网络中发生一次临时分叉很常见，连续 6 次产生分叉的情况，从概率上来说近乎为 0,这也是为什么区块确认是一般是 6 块之后人为分叉：区块链的每一次升级都将导致分布式网络中的节点根据其是否接受升级而运行不同规则，于是人为的分叉就会产生。以比特币为例，比特币系统的改进都是通过 BIP 方式进行，现在比特币社区已经累计有数百条 BIP,这是"人的共识"达成的过程，具体又可以分为"软分叉"和"硬分叉，**软分叉：**"软分叉"是向前兼容的分叉。新规则下产生的区块可被未升级的旧节点所接受，旧节点只是无法识别、解析新规则。新、旧版本互相兼容，软分叉对整个系统的影响较小。新的交易类型一般以软分叉的方式升级,例如 P2SH(Pay-to-Script-Hash)以及隔离见证。另一方面，根据由谁主导"激活"新的提议过程，又可分为矿工激活软分叉(Miner-Activated SoftFork,MASF)与用户激活软分叉(User-Activated Soft Fork, UASF)这里挖矿的节点被称为"矿工"，其他不挖矿的普通节点被称为"用户"。软分叉在比特币发展过程中多次发生，是一种逐步升级的过程。软分叉是在现有结构基础上进行改进，不增加新字段，分叉过程不影响系统的稳妥运行，但是存在升级空间受限的缺点，硬分叉则相反。**硬分叉：**"硬分叉"是不向前兼容的，旧版本节点不会接受新版本节点创建的合法区块，于是新旧版本节点开始在不同的区块链上运行，由于新旧节点可能长期并存，不像软分叉是临时的，硬分叉是有可能会长期存在的，分叉链的存活在于其算力的大小。如果原区块链称为 A 版本，硬分叉产生的同源分叉链称为 B 版本，则具体可以分为如下几种情况。(1) A 版本仍然被广泛支持,B 版本因算力不足而消亡，即还是保留原链。(2) B 版本获得广泛支持,A 版本因算力不足而消亡，即保留新链。(3) A、B 版本都有相当比例的矿工的支持，同时并存，这种情况是最为符合严格意义上的硬分叉，例如 ETH 与 ETC,两者都有其代币，这种分叉存在一定的门槛。(4) A 版本仍然被广泛支持,B 版本通过代码调整难度，小部分节点也能够让它存活。这种分叉币几乎没有门槛，人人可以分叉。(5) B 版本获得支持，A 版本调整代码,小算力也可存活。硬分叉的过程一般经历如下几个阶段。(1) 软件分叉：新的客户端发布，新版本改变规则且不被旧客户端兼容,首先客户端出现了分叉。(2) 网络分叉：接受新版本的节点在网络运行，其发现的区块将被旧版节点拒绝，旧版节点断开与这些新版节点的连接，因此网络进一步出现了分叉。(3) 算力分叉：运行不同客户端版本的矿工的算力将逐渐出现分叉。(4) 链分叉：升级的矿工基于新规则挖矿，而拒绝升级的矿工仍基于旧规则，导致整个区块链出现了分叉。社区分叉：每一种区块链的背后都有其对应的社区、开发者、矿工等利益、信仰共同体，链的硬分叉同时也会带来对应社区的分裂，受关注度最高当属 2016 年因"The DAO 事件"出现的以太坊经典,2017 年催生出的比特币现金以及 2018 年比特币现金的进一步分叉。例如，2017 年针对扩容问题的意见分歧导致 8 月 1 日比特币的首次硬分叉，比特币社区的分裂催生了比特币现金；2018 年 11 月 15 日比特币现金社区再现分裂，分叉为 Bitcoin ABC 和 BitcoinSatoshi、Vision(BSV) 由此可见，社区的分裂才是主导区块链分叉的因素。值得注意的是,2017 年 8 月 1 日，比特币的硬分叉让大量的比特币持有者凭空地增加了一种新的数字货币，即比特币现金(BCH)。这种硬分叉，没有减少资产，反而让人手里多了一种资产，于是区块链分叉就成了一种资产凭空增加的方式。

共识层：分布式系统及其特点：分布式系统时组件分布在网络计算机上且通过消息传递进行通信和行为协调的系统，对外呈现为一个完美的、可扩展的"虚拟节点"，相对单物理节点具备更优越的性能和稳定性。分布式系统具有并发性、缺乏全局时钟（没有一个全局正确的时间来协调各组件的行为)、组件故障的独立性。

分布式系统的系统模型：• 结构模型：构成系统各部分组件的位置、角色和它们之间的关系，定义了系统各组件之间相互交互的方式以及它们映射到下面计算机网络的方式。一般为客户/服务器结构、对等结构（C/S 结构的变种） • 基础模型：对体系结构模型中公共属性的一种更为形式化的描述。包括交互模型，故障模型和安全模型。1.时序模型(Timing Model)根据对时间的假设，一般可以将分布式系统分为同步系统和异步系统两类，前者对于时间有严格的假设，而后者则对时间没有假设，这里时间主要体现在进程执行速度、消息传递延迟和时钟漂移率。同步系统中，进程执行每一步的时间都有明确的上限和下限，每一条消息会在已知的时间范围内确定被接收到，每个进程的本地时钟与实际时间的漂移率也在已知的范围内。同步分布式系统中，由于消息传递时间的上限是已知的，因此可以根据超时(Timeout)来检测进程的(非拜占庭)故障，从而可以大大简化分布式系统设计。然而，实际上分布式情况下很少有真正同步的系统。同步模型的优势在于便于完成理论分析和测试，由于同步模型有时间限制和保证，因此，若在同步模型下都不能解决的问题，意味着没有时间保证的异步系统也不可能解决。异步系统对进程执行速度、消息传递延迟和时钟漂移率都没有限制。换言之，进程每一步执行时间可以是任意时间，消息可能在任意长的时间之后被接收到，时钟漂移率也可能是任意的。实际的分布式系统大多数是异步系统，例如互联网就是典型的异步系统。因此，尽管目前大多数区块链共识算法的理论验证采用同步系统假设，然而其实际运行环境一般都是异步系统。此外，一些学术文献中还提出了部分同步系统(Partially Synchronous Systems)模型，这种模型对系统执行时间有一定的信息，但不一定准确。例如，部分同步系统的进程可以使用近似同步(almost-synchronized)的时钟，或者大概知道其每一步执行时间和消息传输延迟的上限或下限。与完全同步和完全异步系统相比，部分同步系统理论还不完善。2.故障模型(Failure Model)分布式系统中的故障一般发生在计算节点和通信链路上，因而区块链系统大多讨论的是节点故障和信道故障两类。根据故障的性质，可以将其分为崩溃故障(Crash Failure)、遗漏故障(Omission Failure)、时序故障(Timing Failure)和拜占庭故障(Byzantine Failure)四类。崩溃故障是相对简单的一类故障，通常有故障-停止(Fail-Stop).崩溃(Crash)、故障-恢复(Fail-Recovery)等情形。节点正确运行直至崩溃，且一旦崩溃就不再恢复，如果其他节点可以检测到节点的这种崩溃故障，则为"故障-停止"，而如果其他节点可能无法检测到，则称为"崩溃"。在同步系统中，由于消息传输时间有明确上限，因此，通常可以构造基于超时(Timeout)的故障检测器，定期让节点 Ping 访问该故障检测器，如果超过预设的超时时间未收到节点的 Pmg 访问消息，则可认为该节点已经崩溃。如果节点能够在崩溃以后的某个时间点上恢复运行，则称为"故障-恢复"。根据节点是否能恢复到崩溃之前的状态，又可以分为健忘崩溃(Amnesia Crash,不记录崩溃前的状态，重新启动到初始状态)、部分健忘崩溃(Partial Amnesia Crash,部分记录崩溃前的状态)、暂停崩溃(Pause Crash,重新启动到崩溃前的状态)和停机崩溃(Halting Crash,不重新启动)。遗漏故障是指节点或者信道未能执行本来应该执行的动作，部分研究者认为崩溃故障也是遗漏故障的主要形式之一。分布式系统中，节点通信过程一般可分为三个步骤：①发送节点将消息插入到它的外发消息缓冲区并执行发送动作；②通信链路将消息从发送节点的外发消息缓冲区传输到接收节点的接收消息缓冲区；③接收节点执行接收原语并从接收消息缓冲区取走消息。如果在这些步骤中出现缓冲区溢出或者传输错误等问题导致消息丢失，就会引发遗漏故障。Hadzilacos 和 Toueg 将这三个步骤中的消息丢失依次称为发送遗漏故障、信道遗漏故障和接收遗漏故障。时序故障是指节点虽能正确响应,但并未在指定的时间范围内(即过早或过迟的响应)。一般有三种时序故障：①时钟故障，节点的本地时钟超过了实际时钟漂移率的范围；②节点性能故障，节点执行时间超过了两个进程步之间的间隔范围；③信道性能故障，消息传递时间超出了规定的传输延迟范围。时序故障仅适用于同步系统，由于异步系统不存在对时间的显式假设，因此不存在时序故障。拜占庭故障也称为随机故障，是指节点可能任意地、错误地，甚至是恶意地执行某些未经许可的动作。例如，拜占庭故障节点可能会设置或者返回错误的执行结果、执行错误的状态转换规则、篡改消息内容、发送伪造或者重复的消息等。因此，拜占庭故障被认为是分布式系统中最难以解决的故障形式。3.安全模型分布式系统的模块特性以及开放性，使得它们暴露在内部和外部的攻击之下；安全模型的目的是提供依据，以此分析系统可能受到的侵害，并在设计系统时防止这些侵害的发生。分布式一致性：分布式一致性是分布式计算的核心问题。**定义一：**指多个节点对某一变量的取值达成一致，则变量的本次取值即被确定。**定义二：**指分布式系统中的多个服务节点，给定一系列操作，在约定协议的保障下，使它们对外界呈现的状态是一致的。也就是保证集群中所有服务节点中的数据完全相同且能够对某个提案（Proposal）达成一致。

一致性分类：**强一致性：**当分布式系统中更新操作完成后，任何多个进程或线程，访问系统都会获得最新的值。**弱一致性：**指系统并不保证后续进程或线程的访问都会返回最新更新的值。**最终一致性：**最终一致性是弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。也就是说，如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。共识：共识描述了分布式系统中多个节点之间，彼此对某个状态达成一致结果的过程。在实践中，要保障系统满足不同程度的一致性，核心过程往往需要通过共识算法来达成。共识和一致性常常认为是等价的、可互换的。共识侧重的是分布式节点达成一致性的过程和算法，是一种手段。而一致性侧重于节点共识过程最终达成的稳定状态，描述的是结果状态。达成某种共识并不意味着就保障了强一致性。只能说共识机制能够实现某种程度上的一致性。

FLP 定理 FLP 定理的具体表述形式为：在含有多个确定性进程的异步系统中，只要有一个进程可能发生故障，那么就不存在协议能保证有限时间内使所有进程达成一致。值得一提的是，这里并没有考虑拜占庭故障，并且假设消息通信系统是可靠的，即节点可以正确地传输消息。然而，即使是在这样的假设下,单个进程的非拜占庭故障也足以导致任何异步分布式协议无法达成共识。因此，除非我们对计算环境做出进一步假设，或者对容错类型做出更严格的限制，否则此类分布式共识问题不会有鲁棒解。异步系统假设是 FLP 定理的关键。该定理假设进程是完全异步的，没有任何关于进程处理速度或消息传输延迟的先验知识，进程也不能使用同步时钟，因此无法使用基于超时(Timeout)的算法；此外，定理还假设进程无法判断其他进程是处于完全停止状态还是仅运行缓慢。

简单举例说明，考虑如下的稻草人协议(Strawman Protocol):分布式异步共识系统中，每一个决策者(Decider)都会从所有的提议者(Proposer)处收集提案；如果收到多份提案，则选择其中最小的提案达成一致共识。显然，如果没有故障节点，这种共识协议是可行的，尽管速度上会受到最慢节点或网络连接的限制。然而，即使系统中只可能发生简单的崩溃故障，决策者也不会知道到底需要等待多长时间才能收集到"全部"提案，而此时即使使用超时检测方法也无法确知，因此他们除了等待别无他法。如果这样的系统中提议者发生故障，那么决策者们将会无休止地等待而无法达成共识。CAP 定理 CAP 定理是由埃里克-布鲁尔(Eric Brewer)在 2000 年 PODC 会议上提出的猜想，并在 2002 年被证明成立。CAP 定理的具体表述形式为：网络服务不可能同时保证如下三个特点，即一致性、可用性和分区容错性。**1) 一致性(Consistency)**这里指强一致性(Strong),也称原子一致性(Atomic)或线性一致性(Linearizable)，即分布式系统中的所有数据备份在同一时刻必须保持同样的值。如果系统对一个写操作返回成功，那么之后的读请求都必须读到这个新数据；如果返回失败，那么所有读操作都不能读到这个数据。**2) 可用性(Availability)** 集群中部分节点出现故障时，集群整体仍然可以处理客户端的更新请求；所有读写请求在一定时间内得到响应，可终止，不会一直等待。**3) 分区容错性(Partition-tolerance)**在出现网络分区、不同分区的集群节点之间无法互相通信的情况下，被分隔的节点仍能正常对外服务。CAP 定理表明，上述三个性质中，理论上最多只能同时满足其中两个：如果同时满足 C 和 A,即同时满足服务可用且节点间状态一致,则要求不能出现网络分区；如果同时满足 A 和 P,不同分区的节点同时对外服务,则可能导致状态的不一致；如果同时满足 C 和 P,不同分区的节点为实现状态一致，则必须等待而导致不满足 A。CAP 定理在理论上给研究者设计系统提出了一个不可实现的理想结果，但定理中的特性都是非常苛刻的理论性质，因而 CAP 实际上描述的是无法同时构造具有完美一致性、完美可用性和分区容错性的理想系统。工程实践中，通常可以通过适当地放宽对特定性质的假设，来达成对三个性质的部分满足。首先,分区容错性在 CAP 定理的正式表述为"允许丢失任意多的从一个节点发往另一个节点的消息" 对于可能发生网络丢包、节点宕机的分布式系统，特别是可能存在正常故障节点和拜占庭故障节点的区块链系统来说，分区容错性是基本要求和必选项。因此，对于分布式系统实践而言，CAP 定理就转化为：在满足分区容错性基础上，不能同时满足一致性和可用性。其次，在保证分区容错性的前提下，放宽约束后可以兼顾一致性和可用性，两者不是非此即彼。一方面，可以通过牺牲部分一致性来满足可用性：CAP 定理给出的一致性是强一致性，要求无论更新操作是在哪一个副本执行，之后所有的读操作都要能获得最新的数据。这种强一致性在实践中可以放宽为弱一致性(Weak Consistency)，即用户读到某一操作对系统特定数据的更新需要一段时间（称为不一致性窗口)。区块链系统中常见的最终一致性(Eventually Consistency) BP 是弱一致性的表现形式之一。最终一致性允许区块链系统放宽对时间的要求，在数据更新操作完成之后的某个时间点，分布式节点的数据最终达成一致。例如，区块链系统允许在共识过程中，分布式节点的数据可能存在暂时的不一致性,但共识过程后的某个时间点，分布式节点最终将对区块链账本历史数据达成一致。另一方面，可以通过牺牲部分可用性来满足一致性：CAP 定理的可用性要求所有读写操作必须要能终止，实际应用中从主调、被调两个不同的视角，可用性具有不同的含义。当出现网络分区时，主调可以只支持读操作，通过牺牲部分可用性达成数据一致。区块链系统的设计与实现也必须遵从 CAP 定理。目前，以比特币为代表的大多数公有链系统通常以牺牲强一致性来同时满足最终一致性、可用性和分区容错性。此时，区块链节点可能处理不同的交易集合，也可能在分叉情况下暂时维护不同的区块链条；但是随着节点通信和共识过程进行，相互冲突的交易和区块集合中将会最终选出唯一的值并记录到区块链中，并随着区块获得越来越多的确认，达到最终一致性。同时，某些联盟链或私有链可能会以牺牲可用性来满足强一致性和分区容错性。例如，采用 Paxos 的区块链是强一致性共识系统，其在分区容错环境下，就不得不在可用性方面做出让步（例如 Paxos 可能会出现"活锁"问题而无法终止)。

两军问题：（节点可靠而信道不可靠）可以**证明**不存在这样的通信协议：假设确实存在这种协议，那么协议中最后一条信息要么是必要的，要么不是。如果不是必要的，我们可以删除这条消息(以及任何其他不必要的消息)，直到剩余的每条信息都是必要的。此时，如果最后一条消息没有通过山谷中的白军送达对方，由于它是必要消息，则双方将不会进攻。由于最后信息的发送方永远不确定其消息是否安全送达，所以他不会贸然进攻。同理，消息接收方也知道这一点，所以同样不会贸然进攻。由此可见，确实无法设计可使蓝军必胜的通信协议。两军问题也因此成为计算机通信领域首个被证明无解的问题。寻找实际可行的"工程解"就是解决两军问题的必由之路。目前有多种思路可供借鉴。TCP 协议就是两军问题的一种现实解决方案。TCP 协议中，发送方向接收方发出随机数 x,接收方收到 x 之后，返回给发送方另一随机数 y 和 x+ 1,由于破解随机数 x 的可能性较小，所以发送方即可确认接收方已经收到 x;然后发送方再发送 y+1 给接收方，接收方即可确认发送方已经收到 y。这样三次握手后，发送方与接受方就可以"彼此确信"对方已收到消息。然而，事实上发送方仍然并不能确信 y+1 确实送达接收方，因此 TCP 协议仅仅是一种成本可控、相对可行的"工程解"。此外，还可以通过发送大量消息将信道的不可靠性降低到可接受的程度，从而将理论上的不可靠性转化为实际操作时"概率意义上"的可靠性。例如，发送方可以通过发送 100 个信使执行任务，由于所有信使均被俘虏的可能性非常低，所以发送方可以从"概率意义上"认为接收方必然能够接收到消息，从而不必等待接收方确认即可发起攻击。同时，接收方也知道这一点，因此只要收到一个信使的消息，就会发起攻击。考虑计时模型和节点错误模型时的情况：当考虑计时模型时,Pepperland 协定问题要求两支蓝军协商确定由谁先发起对红军的进攻以及具体攻击时间。在异步 Pepperland 系统中，由于不存在信道故障，两支蓝军容易就谁先攻击达成一致，例如双方比较军队的力量，由较强的一方先发起攻击。然而就攻击时间而言,由于信使通过山谷的时间是不确定的，这就存在时间协调问题。例如蓝军#1 发送消息"9 月 1 日凌晨发动攻击"，蓝军#2 可能很快收到消息，也可能攻击时间之后也未能收到。如果是同步 Pepperland 系统，两支蓝军就可知道信使通过山谷的时间存在上限 max 和下限 min,这样先攻击一方在发送"攻击"消息之后，等 min 分钟之后即可发起攻击；而后攻击一方则在收到"攻击"消息后等待 1 分钟，即可发起攻击，并且可以保证在不晚于 max-min+1 分钟后发动攻击。当考虑节点故障时，我们假设白军可以主动攻击并消灭山谷两侧的任意一支蓝军。为此，两支蓝军定时派出信使通报自己的生存状态。显然，异步系统中，每一支蓝军都无法确知对方是已被消灭还是信使穿越山谷的时间太长；同步系统中，每一支蓝军可以通过在给定时间范围内未收到消息来判定对方已被消灭；然而也存在这种情况，即对方在通报自己仍然存活的消息后，就立即被消灭了。

拜占庭将军问题：（信道可靠而节点不可靠）拜占庭将军算法必须保证忠诚将军的通信和决策同时满足：**一致性条件：**所有忠诚的将军必须按照相同的行动计划做出决策。**正确性条件：**少数叛徒不能使得忠诚的将军采纳错误的行动计划。拜占庭将军问题模型——司令-副官模式：司令必须向其 n-1 个副官发送自己的命令,使得：IC1: 所有忠诚的副官遵循相同的命令。IC2：如果司令是忠诚的，则每一个忠诚的副官均遵循他发送的命令。IC1 和 IC2 称为交互一致性条件（Interactive Consistency conditions) ,分别对应拜占庭将军问题的一致性和正确性。显然，如果司令是忠诚的，则 IC1 可由 IC2 导出。至此，拜占庭将军问题完全转化为单个将军视角的形式化模型，任何满足条件 IC1 和 IC2 的算法，均可有效解决拜占庭将军问题。

口头消息算法（OM(m))：解决包含 m 个叛徒的 3m+1 个或更多将军情况下的拜占庭问题。口头消息满足的假设条件：A1:（信道可靠）每个发送的消息都会被正确地传递；A2:（消息来源可判定）消息的接收者知道是谁发送的消息；A3:（叛徒无法通过不发消息阻碍达成共识）消息的缺失可以被检测到算法OM(0) **(1)**司令向每一位副官发送命令**(2)**每一位副官采用司令发来的命令，未收到默认"撤退"

算法OM(m) (1)司令向每一位副官发送命令。(2)对任意 i 来说，令 vi 为副官 i 从司令处收到的命令，或缺失时的默认值"撤退"。副官 i 将作为司令启动算法 $OM(m-1)$，并向其他 $n-2$ 个副官发送命令 vi (3) 对任意 i 和 $j \neq i$ 来说，令 vi 为(2)中 $OM(m-1)$算法中副官 i 从副官 j 处接收到的命令，或者缺失时的默认值"撤退"。副官将采用命令 majority(v1,v2,⋯,v_n-1)OM(m)算法是递归算法，共递归执行 m + 1 轮，每一轮都需要每个副官向其他人发送命令以便相互核实，因而算法复杂度是节点数的平方，即$O(n^2)$，且主要集中于通信开销。

共识过程的主流模型：区块链系统的全体数据节点集记为 P，一般分为生产数据或者交易的普通节点，以及负责对普通节点生成的数据或者交易进行验证、打包、更新上链等挖矿操作的"矿工"节点集合(记为 M)，两类节点可能会有交集；矿工节点通常情况下会全体参与共识竞争过程，在特定算法中也会选举特定的代表节点代替他们参加共识过程并竞争记账权，这些代表节点的集合记为 D；通过共识过程选定的记账节点记为 A。共识过程按照轮次重复执行，每一轮共识过程一般重新选择该轮的记账节点。共识过程的核心是"选主"和"记账"两部分，在具体操作过程中每一轮可以分为选主(Leader Election)、造块(Block Generation) 验证(Data Validation)和上链(ChainUpdation，即记账)四个阶段。共识过程的输入是数据节点生成和验证后的交易或数据，输出则是封装好的数据区块及更新后的区块链。四个阶段循环往复执行，每执行一轮将会生成一个新区块。**第一阶段：选主** 选主是共识过程的核心，即通过选举、证明、联盟、随机或者混合等方式从全体矿工节点集 M 中选出记账节点 A 的过程；我们可以使用公式 $f(M) \rightarrow A$ 来表示选主过程，其中函数 f 代表共识算法的具体实现方式。一般来说，|A|=1，即最终选择唯一的矿工节点来记账。**第二阶段：造块** 第一阶段选出的记账节点根据特定的策略将当前时间段内全体节点 P 生成的交易或者数据打包到一个区块中，并将生成的新区块广播给全体矿工节点 M 或其代表节点 D。这些交易或者数据通常根据区块容量、交易费用、交易优先级、等待时间等多种因素综合排序后，依序打包进新区块。造块策略是区块链系统性能的关键因素，也是贪婪交易打包、自私挖矿等矿工策略性行为的集中体现。**第三阶段：验证** 矿工节点 M 或者代表节点 D 收到广播的新区块后，将各自验证区块内封装的交易或者数据的正确性和合理性。如果新区块获得大多数验证/代表节点的认可，则该区块将作为下一区块更新到区块链。**第四阶段：上链** 记账节点将新区块添加到主链，形成一条从创世区块到最新区块的完整的、更长的链条。如果主链存在多个分叉链，则需根据共识算法规定的主链判别标准，来选择其中一条恰当的分支作为主链。

共识算法的分类：现有研究中的共识问题实际上可以分为算法共识和决策共识两个分支，前者致力于研究在特定的网络模型和故障模型前提下，如何在缺乏中央控制和协调的分布式网络中确保一致性，其实质是一种"机器共识"后者则更为广泛地研究无中心的群体决策中，如何就最优的决策达成一致的问题，例如关于比特币系统扩容问题和分叉问题的社区讨论与路线选择，其实质是"人的共识，本章讨论的区块链共识算法研究应属于算法共识分支的子集，而决策共识则大多见于分布式人工智能、多智能体等研究领域。1. 选主策略 根据选主策略，可以大致将区块链共识算法分为选举类、证明类、随机类、联盟类和混合类共五种。**选举类共识：**矿工节点在每一轮共识过程中通过投票选举的方式选出当前轮次的记账节点，首先获得半数以上选票的矿工节点将会获得记账权；多见于传统分布式一致性算法，例如 Paxos 和 Raft 等。**证明类共识：**也可称为"Proof of X"共识。矿工节点在每一轮必须证明自己具有某种特定的能力，证明方式通常是竞争性地完成某项难以解决但易于验证的任务，在竞争中胜出的矿工节点将获得记账权；例如 PoW 和 PoS 等共识算法是基于矿工的算力或者权益来完成随机数搜索任务，以此竞争记账权。**随机类共识：**矿工节点根据某种随机方式直接确定每一轮的记账节点，例如 Algorand 和 PoET 共识算法。**联盟类共识：**矿工节点基于某种特定方式首先选举出一组代表节点，而后由代表节点以轮流或者选举的方式依次取得记账权，例如 DPoS 共识算法。**混合类共识：**矿工节点采取多种共识算法的混合体来选择记账节点，例如 PoW + PoS 混合共识、DPoS+BFT 共识等。2. 容错类型根据容错类型，可以将区块链共识算法分为拜占庭容错和非拜占庭容错两类。**拜占庭容错共识**指能够容忍拜占庭故障的共识算法，例如PoW和PBFT等。**非拜占庭容错共识**：指不能容忍拜占庭故障的共识算法，通常只能容忍故障-停止或者故障-恢复等普通的崩溃故障，例如Paxos 和 Raft 等 3.部署方式：**公有链共识：**适用于公有链的共识算法，这类共识算法通常去中心化程度较高，可容忍拜占庭故障但技术效率偏低，例如 PoW 和 PoS 等。**私有链共识：**适用于私有链的共识算法；与公有链共识相反，这类共识算法通常是完全中心化的，不能容忍拜占庭故障但技术效率较高，例如 Paxos 和 Raft 等。**联盟链共识：**适用于联盟链的共识算法，在去中心化程度和技术效率等指标上都介于公有链共识和私有链共识之间，例如 PBFT 等。4.一致性程度可以将区块链共识算法分为**强一致性共识**和**弱一致性共识**

通常情况下，公有链基于拜占庭共识实现一致性，联盟链基于非拜占庭共识实现一致性。分布式共识算法分为两类，即拜占庭容错（Byzantime Fault Tolerance,BFT）和非拜占庭容错（Crash Fault Tolerance,CFT）主流区块共识算法：PBFT：PBFT 算法假设共识过程的运行环境是一个异步分布式网络(例如互联网环境)，该网络中可能发生消息传输失败、延迟、重复发送或乱序等，并且可能存在蓄意发送错误消息的恶意节点。PBFT 算法假定这些节点故障都是独立发生的，并采用加密技术来防止欺骗和重播以及检测损坏的消息。消息包括公钥签名、消息认证编码以及由无碰撞散列函数产生的消息摘要等。PBFT 算法将消息表示为表示由节点 i 签名的消息,D(m)表示消息 m 的摘要。按照惯例,PBFT 算法只对消息摘要而非完整消息进行签名，并附在消息文本后。所有节点都知道其他节点的公钥。PBFT 算法模型允许系统中存在一个强大的敌对节点，它可以协调故障节点，延迟通信或延迟正常节点，以造成最大的损害。但是，模型中假设该敌对节点不能无限期延迟正确的节点，并且这个节点(以及它控制的错误节点)在计算上受到限制，因此(非常高概率)它无法破坏上面提到的加密技术。例如，敌对节点不能产生正常节点的有效签名，由摘要反向计算出相应的消息，或找到两个具有相同摘要的消息等。pbft 也是一种基于状态机复制的实用共识算法：服务被建模为状态机，在分布式系统中的不同节点上进行复制；每个状态机副本(Replica)都维护了服务状态，并实施服务的相应操作。以 R 表示状态机副本集合，每个副本以一个 {0,1,⋯, R-1} 之间的整数作为编号。假设|R|=3f+1,其中 r 为可能发生故障的副本的最大数量。尽管 PBFT 系统可以容纳超过 3f + 1 个副本，但这将降低系统整体性能而且没有提供更好的弹性(因为系统必须交换更多的消息)。换言之，PBFT 共识算法可在不高于约 33%的拜占庭错误节点的系统中保证活性和安全性。所有副本的状态变迁通过称为视图(View)的配置更换而进行。在每个视图中，只有一个副本为主节点(Primary),其余副本作为备份(Backups) .主节点可以简单地由 p =p mode |R|产生，其中 p 为整数，是当前视图的编号。若主节点失效，则执行视图更换过程。这一点与 VR 和 Paxos 共识算法相似，区别在于后者不能容忍拜占庭故障。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。与其他状态机复制技术一样，PBFT 对每个副本节点提出了两个限定条件：①所有节点必须是确定性的，也就是说，在给定状态和参数相同的情况下，操作执行的结果必须相同；②所有节点必须从相同的状态开始执行。在这两个限定条件下，即使失效的副本节点存在，PBFT 算法对所有非失效副本节点的请求执行总顺序达成一致，从而保证安全性。一次完整的共识需要经历请求(Request)、预准备(Pre-prepare) 准备(Prepare)、确认(Commit)和回复(Reply)五个阶段。其中，预准备和准备两个阶段用来确保同一个视图中请求发送的时序性(使对请求进行排序的主节点失效)，准备和确认两个阶段用来确保在不同的视图之间的确认请求是严格排序的。PBFT 随节点增多性能下降很快，但是节点较少时性能不错且分叉概率很低。PBFT 主要用于联盟链，但如果能结合 DPoS 这样的节点代表选举规则的话也可应用于公有链，并且可以在不可信网络中解决拜占庭容错问题，TPS 远大于 PoW。

Paxos：Paxos 算法并不考虑拜占庭将军问题，即节点或者进程只可能出现崩溃、遗漏等正常故障，而不会作恶来篡改或者伪造消息。在总共有 2f + 1 个节点或进程的 Paxos 系统中，可以容忍最多不超过 f 个进程同时发生故障。提案是 Paxos 算法的重要概念。每个提案一般由两部分组成，即提案编号和提案值：提案编号通常由 Proposer 自行选择决定，一般是相互独立、不可重复的递增序列。换句话说，每个 Proposer 都不会使用比自己曾经用过的编号更小的编号。Lamport 提出一种简单的编号生成规则：假设总共有 M 个 Proposer，每个 Proposer 被分配一个 1 到 M 之间互不相同的数字；对于分配到数字 i 的 Proposer，其第乃次提案的编号为 M(n-1)+i。提案值(Value)是等待达成共识的数据值，可以是任意二进制数据，例如一条日志或者命令等。假设分布式系统中存在多个 ProposerAcceptor 和 Learner 节点或进程。Paxos 算法需要保证在多个被提出的 Value 中，只有一个 Value 被最终选定。显然，如果没有 Value 被提出，就不应该有 Value 被选定。因此，达到一致性需要满足以下条件。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。P2c：对于任意的 p 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，编号最大的者的值为 v。Paxos 算法流程分为如下两个阶段。**第一阶段 (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，如果答案是肯定的，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编号与值，如果没有接受过提案，Nx 和 Vx 为 NULL。**第二阶段 (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。Paxos 算法将会产生一系列被选定的提案以供 Learner 执行。为了获取被选定的提案值，Learner 必须确定该提案至少已经被半数以上的 Acceptor 接受。一般说来，Learner 可以通过如下三种方式学习(获取)被选定的 value.**方式一：**Acceptor 每接受一个提案，就将该提案发送给所有 Learnero 这种方式可以使得 Learner 快速获取被选定的 value，但是由于每个 Acceptor 都要与每个 Learner 通信，其所需的通信次数等于二者数量的乘积。**方式二：**Acceptor 每接受一个提案，就将该提案发送给主 Learner；当提案值被最终选定后，再由主 Learner 发送给其他 Learner 0 这种方式将通信次数降低为 Acceptor 和 Learner 的数量之和，但是主 Learner 可能会发生单点故障问题，降低了系统可靠性。**方式三：**Acceptor 每接受一个提案，就将该提案发送给一个 Learner 集合，该集合中的每个 Learner 都可以将选定的提案值发送给所有的 Learner0 显然，这是方式一和方式二的折中方案，Learner 集合的数量越多，系统可靠性就越好，但通信复杂度也相应地越高。

异常情况：活锁：基本 Paxos 算法(Basic Paxos)算法可能会出现活锁(Livelock)的异常情况，即两个 Proposer 持续地交替生成编号递增的一系列提案，但是没有提案会被最终选定。例如：Proposer #1 提出编号为 n1 的提案并且完成了 Paxos 算法的第一阶段后，另一个 Proposer #2 也提出编号为 n2(n2>n1)的提案并执行完成第一阶段。此时，Proposer #1 后续第二阶段针对编号为 n1 的提案的所有 Accept 请求将被忽略，因为 Acceptor 已经承诺不再通过任何编号小于 n2 的提案。于是 proposer #1 就会用一个新的编号 n3(n3>n2)重新开始并完成第一阶段，而这又会导致 Proposer #2 在其第二阶段的所有 Accept 请求被忽略，如此循环往复，形成活锁。解决活锁问题的主要途径是使得任意时刻只有唯一的 Proposer 提出提案，然而这会导致单点故障的风险。因此，当唯一的 Proposor 出现故障时，其他节点或进程需要检测到并再启动一个新的 Proposor。显然，由于 Paxos 算法的异步模型假设，其消息传递时间没有限制，其他节点或进程通过有限的时间等待无法判断运行 Proposor 的节点或进程是出现故障，还是消息尚未发送到。如果设置等待一段时间无响应后再启动新的 Proposer，那么当旧的 Proposer 并未真正出现故障时，就会出现两个 Proposer，仍然可能会出现活锁现象。由此可见，基本 Paxos 算法仍然遵循 FLP 不可能定理，即在理论上的"异步系统"中，Paxos 算法可能无限地运行下去永远无法达成一致，虽然这种概率很小。

Raft：Raft 的首要设计原则就是直观、易懂，其技术设计包括模块化拆分和状态空间规约：前者把 Raft 算法过程划分为领导选举(Leader Election)、日志复制(Log Replication)、安全性(Safety)和集群成员变更(Cluster Membership Changes)等相对松散耦合的模块；后者则降低了 Raft 共识过程的不确定性程度和服务器间不一致的方式。

为确保交易有效，使用智能合约创建的交易通常需要由多个组织签名才能提交到通道账本。多个签名是 Fabric 信任模型不可或缺的一部分。一项交易需要多次背书，以防止一个通道上的单一组织使用通道不同意的业务逻辑篡改其对等节点的分类账本。要签署交易，每个组织都需要调用并在其对等节点上执行智能合约，然后签署交易的输出。如果输出是一致的并且已经有足够的组织签名，则可以将交易提交到账本。该政策被称为背书政策，指定需要执行智能交易的通道上的已设置组织合同，针对每个链码设置为链码定义的一部分。Raft 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。

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 日志复制过程：**步骤 1：**假设客户端向领导者发出日志请求，内容为设置，v=3 **步骤 2：**领导者将该请求作为新的日志条目附加到其日志中，然后发送 Append-Entries RPC 给所有跟随者，要求让他们复制这条日志条目 **步骤 3：**跟随者确认接收到日志复制请求,执行后反馈给领导者 **步骤 4：**领导者收到大多数写日志的成功消息后就会应用这条日志条目到它的状态机中，然后把执行结果返回给客户端。此时的日志条目被称为处于已提交状态；在下一个心跳中，领导人通知所有跟随者更新已提交日志，应用到各自的状态机中。Raft 算法中，领导者通过强制跟随者复制它的日志来解决可能产生的不一致问题。领导者如果发现某个跟随者的日志跟自己不一致，那么就通过 Append-Entries RPC 一致性检查过程找到两者达成一致的索引最大的日志条目，并且用自己的数据更新该跟随者日志中从该点之后的所有日志条目。这就保证了日志复制过程中的一致性，实现了分布式系统中的共识

Raft 故障：**(1)** 领导者在接收到客户端请求前发生故障：此时不影响一致性。**(2)** 领导者已经接收到客户端请求，但尚未复制到跟随者节点时发生故障：此时日志条目处于未提交状态。跟随者节点会启动领导选举过程，重新选出新任期的领导者。客户端因收不到确认消息，会认为超时失败而重新提交请求给新的领导者，此时日志复制即可成功。原来的领导者从故障中恢复后作为跟随者重新加入集群，并从当前任期的新领导者处同步数据，强制保持和新领导者的数据一致性。**(3)** 领导者已经接收到客户端请求、并成功复制到"所有"跟随者节点，但尚未收到确认消息前发生故障：此时日志条目处于未提交状态，但在所有跟随者节点中保持一致，重新选举出新的领导者后即可完成提交。此时客户端可能会重新提交日志请求，因而需要 Raft 算法有内部去重机制。**(4)** 领导者已经接收到客户端请求，并成功复制到"部分"跟随者节点，但尚未收到确认消息前发生故障：与第 3 种情形相比，此时日志请求处于未提交状态且在跟随者节点中不一致。Raft 有特定的机制保证只有拥有最新数据的节点才会被选为领导者，因此新领导者强制同步到跟随者节点后，日志请求不会丢失并最终一致。**(5)** 领导者已经接收到客户端请求，并成功复制到"所有或者多数"跟随者节点，接收到跟随者的确认接收消息之后发生故障，此时日志条目在领导者处是已提交状态，而在跟随者处是未提交状态：此时重新选举出新领导者后的流程与第 3 种情形一致。**(6)** 领导者已经接收到客户端请求，并成功复制到"所有或者多数"跟随者节点，日志条目在所有节点处都是已提交状态，但尚未反馈到客户端时发生故障：此时服务器集群内部日志是一致的，客户端重新发送请求后即可成功，对一致性无影响。**(7)** 因网络分区导致出现双领导者：如因网络分区使得领导者无法访问大多数跟随者，那么这些跟随者将会重新选举出一位新领导者，从而形成双领导者的局面。当网络分区故障恢复，原领导者发现集群中有更新任期的领导者后，会自动降级为跟随者并从新领导者处同步数据以保证集群数据一致性。

链设计模式：Order-Execute(Bitcoin, Etherem) vs Execute-Order-Commit(Fabric)Execute-Order-Commit（Fabric）

(1)客户端发起交易，多个节点会先执行交易并返回结果(节点执行交易但是不会更改本地账本) **(2)**客户端检测是否满足共识算法，若满足则发给区块产生的节点产生区**(3)**所有节点接收到区块后提交执行结果到本地　　(Fabric 中对部分名词做了修改)

Fabric 共识算法：主要检查交易执行结果是否一致，是否收到的足够数量的交易结果（背书策略）优点：共识算法非常快；支持 non-deterministic 的交易；交易并发；Execute：1. 客户端发送交易请求到若干个背书节点（背书策略）2. 客户端验证请求响应 3. 若验证通过则发送给 Order 节点 Order 阶段：4. Order 节点验证接收到的交易请求，并打包成区块 5. Order 节点将区块发送给Peer 节点 Validate：Peer 节点对区块中的交易进行验证，并添加到链末尾，同时更改状态数据库 **•联盟：**所有有业务往来的组织组织的集合称为一个联盟 **•通道：**一个具体的应用其对应的组织节点，账本数据等称为一个通道# Fabric 上通道的设立主要是为了数据的隔离，以在隐私上提供保护 **•组织：**逻辑上是一个主体，包含了很多节点，这些节点之间互相信任 **•节点：**区块链网络上的具体节点。由于 Fabric 的交易流程的独特性，节点在功能上分为 orderer 和 peer，分别用来产生区块，和处理交易和区块。**•链码：**链码就是智能合约的实例，一个链码可能包含多个合约。**•账本：**Fabric 账本相比于普通区块链的账本范围更广，除了包含链式数据结构（区块链），还包含了有状态数据库，另外还包含了通道和节点本身的一些状态信息。

以太坊：以太坊在整体上可看作是一个基于交易的状态机：起始于一个创世(Genesis)状态，然后随着交易的执行，状态逐步改变一直到最终状态，这个最终状态就是以太坊世界的权威版本。以太坊中引入了账户的概念以取代比特币 UTXO 模型，账户分为外部账户(Externally Owned Account,EOA)和合约账户(Contract Account, CA)两类，两类账户都具有与之关联的账户状态和账户地址，都可以存储以太坊专用加密货币(以太币)，区别在于外部账户由用户私钥控制，没有代码与之关联；合约账户则由合约代码控制，有代码与之关联。两类账户的账户状态都包含以下四个字段。(1) Nonce：随机数，即账户发出的交易数及创建的合约数量之和。(2) Balance：余额，即账户拥有的以太币数量，单位为 Wei，是以太币的最小单位。一个以太币 Ether 等于 1018 个 Wei (3) CodeHash：代码哈希，与账户关联的 EVM 代码的哈希值，外部账户的 CodeHash 为一个空字符串的哈希，创建后不可更改。状态数据库中包含所有代码片段哈希，以便后续使用。(4)StorageRoot：存储根节点，账户内容的 MerklePatricia 树根节点的哈希编码

BTC 交易生成：假设源节点希望将自己的一个比特币转账给目的节点。源节点的钱包将会创建一个新的交易，该交易中使用的比特币来自于源节点的一个或者多个 UTXO 源节点将目的节点的公钥作为交易的参数，同时使用自己的私钥对生成的新交易进行签名。如图 2-6 所示为比特币白皮书对交易结构的描述：当 Owner 1 希望向 Owner 2 转账时，Owner 1 构造的交易（图 2-6 居中的交易）需要包含接收方 Owner 2 的公钥、上一交易 UTXO（图 2-6 居左的交易）的哈希值以及 Owner 1 的私钥签名。这里私钥签名的目的是向比特币网络的其他节点证明这笔交易确实是由源节点 Owner 1 创建并发送的，而封装目的节点 Owner 2 的公钥则使得比特币网络中只有对应私钥的接收方 Owner 2 才能使用这笔交易中的比特币；其他节点虽然能够接收和验证这笔交易，但是由于没有对应的正确私钥，因而并不能动用这笔比特币。这种设计可以增加比特币系统的安全性。

BTC 的 UTXO：用户每次创建新转账交易时，都会消耗其一定数量的 UTXO,同时生成新的 UTXO,这也使得比特币从旧的 UTXO 流入新的 UTXO。新 UTXO 将会使用接收方的公钥锁定，只有接收方使用其私钥才能花费该 UTXO 的比特币。BTC 钱包：**非确定性钱包：**又称随机钱包，钱包只是随机生成的私钥的集合。每个密钥都是从随机数独立生成的。密钥彼此无关，缺点是，钱包丢失，密钥丢失，则钱包所控制的资金不可避免地丢了。**确定性钱包：**其中所有的密钥都是从一个主密钥派生出来的，这个主密钥即为种子。该类型钱包中所有密钥都相互关联，如果有原始种子，则可以再次生成全部密钥。确定性钱包中，使用了许多不同的密钥派生方法。最常见的派生方法是使用树状结构，称为分层确定性钱包或 HD 钱包。除了简单的测试外，任何场合都不鼓励使用非确定性钱包，备份和使用过于麻烦，通常推荐使用基于行业标准的 HD 钱包，只需要备份种子助记词。公钥和地址生成：1、生成私钥（合适的范围内）；2、用 ECDSA 算法由私钥生成公钥；3、生成钱包地址（由数字 1 开头）。发起交易：1、钱包软件搜集 UTXO；2、提供正确的解锁脚本；3、构造支付给接收者的输出；4、交易构造完毕，发送；5、结点受到交易，独立验证，通过则转发；6、某个挖矿结点验证并加入到区块中。

智能合约：狭义定义：运行在分布式账本上预置规则、具有状态、条件响应的，可封装、验证、执行分布式节点复杂行为，完成信息交换、价值转移和资产管理的计算机程序。广义定义：无需中介、自我验证、自动执行合约条款的计算机交易协议。按照其设计目的可分为：旨在作为法律的替代和补充的智能法律合约，旨在作为功能型软件的智能软件合约，以及旨在引入新型合约关系的智能替代合约。智能合约的雏形：比特币脚本 •基于非图灵完备字节码语言 OP-RETURN 的比特币脚本是最早应用于区块链的智能合约雏形。•计算能力非常有限，不支持循环语句，只能实现基本的算术、逻辑运算及验证加密功能，因此早期的智能合约通常无法具有复杂逻辑。•可编程货币。以太坊产生背景：•比特币开创了去中心化密码货币的先河。区块链事实上是一套分布式的数据库，如果再在其中加进一个符号（比特币），并规定一套协议使得这个符号可以在数据库上安全地转移，并且无需信任第三方，这些特征的组合完美地构造了一个货币传输体系：比特币网络。•然而比特币并不完美，其中协议的扩展性是一项不足，例如比特币网络里只有一种符号（比特币），用户无法自定义另外的符号，这些符号可以是代表公司的股票，或者是债务凭证等，这就限制了一些功能。另外，比特币协议尽管使用了一套基于堆栈的脚本语言，然而却不足以构建更高级的应用，例如去中心化交易所等。以太坊从设计上就是为了解决比特币扩展性不足的问题。什么是以太坊：以太坊是一个为去中心化应用程序而生的全球开源平台，经常被称为"世界计算机"。从计算机科学的角度来看，以太坊是一个具备确定性但实际上却没有边际的状态机。它有两个特点：具有一个全球范围可访问的单体状态；还有一个执行状态更改的虚拟机。从更加实际的角度看，以太坊是一个开源的、全球去中心化的计算基础架构，可以执行称为智能合约的程序。它使用区块链同步和保存系统状态，借助以太币这种数字货币来计量并控制程序执行的资源开销。

以太坊的基本组件：**点对点(P2P)网络：**以太坊运行在 Ethereum Main Network 上，这是一个通过 TCP 30303 端口寻址的网络，网络层运行的协议名为 DEVp2p **共识规则：**以太坊的共识规则，由以太坊黄皮书中的参考标准进行精确定义。**交易：**以太坊交易是一个网络消息，主要包含交易的发送方、接收方、价值和数据载荷。**状态机：**以太坊的状态转换由以太坊虚拟机(EVM)处理，这是一个基于栈的虚拟机，执行 bytecode(字节码指令)。被称为"智能合约"的 EVM 程序采用高级语言(例如 Solidity)编写，并编译为通过 EVM 执行的字节码。**数据结构：**以太坊的区块链以数据库(通常采用 Google 的 LevelDB)的方式保存在每一个节点之上，区块链内包含了交易和系统的状态，经过哈希处理的数据保存在 Merkle Patricia Tree(MPT)数据结构之内。**共识算法：**以太坊使用比特币的共识模型 Nakamoto Consensus，它使用顺序单一签名块，由 PoW 加权重要性来确定最长链，从而确定当前状态。但是，有计划在不久的将来转向代号为 Casper 的 PoS 加权投票系统。UTXO 模型的好处**可扩展性** - 由于可以同时处理多个 UTXO，因此可以实现并行交易并鼓励可扩展性创新。**隐私性** - 即使比特币不是一个完全匿名的系统，假设用户每笔交易都使用新地址，那么也能够提供更高级别的隐私。如果需要更好的隐私性可以考虑更复杂的方案，例如环形签名。账户/余额模型的好处：**简单性** - 以太坊选择了更直观的模型，以使开发人员在开发那些需要状态信息或涉及多方的智能合约时更简单。例如智能合约，它能够直接跟踪状态并能够执行不同的任务。UTXO 的无状态模型会强制交易包含状态信息，这使得合约的设计复杂化。**效率** - 除简单性外，帐户/余额模型更有效，因为交易时只需要验证发送帐户是否有足够的余额。账户/余额模型的一个缺点是**无法避免双重支付攻击**，但通过递增的随机数可以消除这种类型的攻击。在以太坊中，每个帐户都有一个公共可见的随机数，每次进行交易时，随机数增加一。这可以防止同一个交易被多次提交。

以太坊区块链：以太坊对比特币区块链技术上做了一些调整，以太坊区块主要由区块头、交易列表和叔区块头三部分组成。区块头包含下列信息：父块的散列值(Prev Hash)、叔区块的散列值(Uncles Hash)、状态树根散列值(stateRoot)、交易树根散列值(Transaction Root)、收据树根散列值(Receipt Root)、时间戳(Timestamp)、随机数(Nonce) 等。以太坊区块链上区块数据结构的一个重大改变就是保存了三棵 Merkle 树根，分别是状态树、交易树和收据树。存储三棵树可方便账户做更多查询。交易列表是由矿工从交易池中选择打包进区块中的一系列交易。区块链上的第一个区块称为"创世区块"，区块链上除了创世区块以外每个区块都有它的父区块，这些区块连接起来组成一个区块链，出块时间约为 15 秒。

以太坊 Header 成员：**• ParentHash：**指向父区块(parentBlock)的哈希指针。除了创世块(Genesis Block)外，每个区块有且只有一个父区块。**• Coinbase：**挖掘出这个区块的矿工地址。在每次执行交易时系统会给与一定补偿的 Ether，这笔金额就是发给这个地址的。**• UncleHash：** Block 结构体的成员 uncles 的 RLP 哈希值。uncles 是一个 Header 数组。**• Root：** StateDB 中的"state Trie" 的根节点的 RLP 哈希值。Block 中，每个账户以 stateObject 对象表示，账户以 Address 为唯一标示，其信息在相关交易(Transaction)的执行中被修改。所有账户对象可以逐个插入一个 Merkle-Patrica-Trie(MPT)结构里，形成"state Trie"。**• TxHash:** Block 中 "tx Trie" 的根节点的 RLP 哈希值。Block 的成员变量 transactions 中所有的 tx 对象，被逐个插入一个 MPT 结构，形成"tx Trie"。**• ReceiptHash：** Block 中的 "Receipt Trie" 的根节点的 RLP 哈希值。Block 的所有 Transaction 执行完后会生成一个 Receipt 数组，这个数组中的所有 Receipt 被逐个插入一个 MPT 结构中，形成"Receipt Trie"。**• Bloom：** Bloom 过滤器(Filter)，用来快速判断一个参数 Log 对象是否存在于一组已知的 Log 集合中。**• Difficulty：**区块的难度。Block 的 Difficulty 由共识算法基于 parentBlock 的 Time 和 Difficulty 计算得出，它会应用在区块的'挖掘'阶段。**• Number：**区块的序号。Block 的 Number 等于其父区块 Number +1。**• Time：**区块"应该"被创建的时间。由共识算法确定，一般来说，要么等于 parentBlock.Time + 10s，要么等于当前系统时间。**• GasLimit：**区块内所有 Gas 消耗的上限，打包进该区块的交易的 GasLimit 之和不能大于区块的 GasLimit。该数值在区块创建时设置，与父区块有关。具体来说，根据父区块的 GasUsed 同 GasLimit * 2/3 的大小关系来计算得出。**• GasUsed：**区块内所有 Transaction 执行时所实际消耗的 Gas 总和。**• Nonce：**一个 64bit 的哈希数，它被应用在区块的"挖掘"阶段，并且在使用中会被修改。

以太坊交易与消息：
• 以太坊的"交易"是指一条外部账户发送到区块链上另一账户的消息的签名数据包，包含了发送者的签名、接收者的地址以及发送者转移给接收者的以太币数量等内容。
• 以太坊交易数据包还包括 GasLimit 和 GASPRICE。以太坊的每一笔交易都需要支付一定的费用，用于支付交易执行所需要的计算开销。计算开销的费用并不是以太币直接计算的，而是引入 Gas 作为执行开销的基本单位，通过 Gas Price 与以太币进行换算。
• 引入 Gas，还可以防止代码的指数型爆炸和无限循环，每笔交易需要对执行代码所引发的计算步骤-包括初始消息和所有执行中引发的消息-做出限制。GasLimit 就是限制，GASPRICE 是每一计算步骤需要支付矿工的费用。如果执行交易的过程中，"用完了瓦斯"，所有的状态改变恢复原状态，但是已经支付的交易费用不可收回了。如果执行交易中止时还剩余瓦斯，那么这些瓦斯将退还给发送者。
• 创建合约有单独的交易类型和相应的消息类型；合约的地址是基于账号随机数和交易数据的哈希计算出来的。• 以太坊的消息在某种程度上类似于比特币的交易，一个消息就是一个交易• 消息包括：消息发送者、消息的接收者、可选的数据域、合约实际输入数据、gasLimit，同交易。

交易可以分为三种：**(1)转账交易：**是最简单的一种交易，从一个账户向另一个账户发送以太币。发送转账交易只需要指定交易的发送者、接收者、转币的数量。**(2)创建合约的交易：**将合约部署到区块链上的交易，发送者是合约的创建者，接受者为空，交易数据字段中指定合约的二进制代码。**(3)执行合约的交易：**调用合约中的方法，需要将交易的接受者指定为要调用的合约的地址，通过交易数据字段指定要调用的方法以及向该方法传递的参数。一段智能合约是被唯一的（合约）地址所标识，该地址有自己的资金余额（以太币），并且一旦有一笔交易发送至该（合约）地址，以太坊网络节点就会执行合约逻辑。以太坊使用以太坊虚拟机(EVM) 来执行智能合约。

以太坊中一笔交易的执行流程大致为：客户端构造交易→通过 p2p 网络广播交易→矿工节点收到交易→ 将交易反序列化为 Transaction 结构→ 将交易放到 mempool → 矿工挖矿 → 在 EVM 中执行这笔交易→交易执行结果写入 stateDB

交易执行：以太坊的状态转换函数：APPLY(S,TX) → S'，可以定义如下：• 检查交易的格式是否正确、签名是否有效和随机数是否与发送者账户的随机数匹配。如否，返回错误。• 计算交易费用：fee=GASLimit * GASPRICE，并从签名中确定发送者的地址。从发送者的账户中减去交易费用和增加发送者的随机数。如果账户余额不足，返回错误。• 设定初值 GAS = GASLimit，并根据交易中的字节数减去一定量的瓦斯值。• 从发送者的账户转移价值到接收者账户。如果接收账户还不存在，创建此账户。如果接收账户是一个合约，运行合约的代码，直到代码运行结束或者 gas 用完。• 如果因为发送者账户没有足够的钱或者代码执行耗尽瓦斯导致价值转移失败，恢复原来的状态，但是还需要支付交易费用，交易费用加至矿工账户。• 否则，将所有剩余的瓦斯归还给发送者，消耗掉的瓦斯作为交易费用发送给矿工。

MPT 树结合了 Patricia 树和 Merkle 树的优点，在 Patricia 树中根节点是空的，而 MPT 树可以在根节点保存整棵树的哈希校验和，而校验和的生成则是采用了和 Merkle 树一致的生成方式。以太坊采用 MPT 树来保存交易、交易的收据以及世界状态。以太坊还使用了一种特殊的十六进制前缀(hex-prefix, HP)编码，使得字母表只有 16 个字符（一个字符称为一个 nibble）。MPT 树节点分成了四种类型：• 空节点 (hashNode)• 叶节点 (valueNode)• 分支节点 (fullNode)
• 扩展节点 (shortNode • 状态树中的每个节点最多有 16 个孩子节点，每个叶节点表示一个账户，这些叶节点的父节点由叶节点的数列组成，而这些父节点再组成更高一层的父节点，直至到形成根节点。• 状态树包含一个键值映射，其中键是账户地址，值是账户内容，主要是{nonce, balance, codeHash, storageRoot}。• 状态树代表发布区块后的整个状态。交易树：• 每个区块都有一棵独立的交易树。区块中交易的顺序主要由"矿工"决定，在这个块被挖出前这些数据都是未知的。不过"矿工"一般会根据交易的 GasPrice 和 nonce 对交易进行排序。首先会将交易列表中的交易划分到各个发送账户，每个账户的交易根据这些交易的 nonce 来排序。每个账户的交易排序完成后，再通过比较每个账户的第一条交易，选出最高价格的交易，这些是通过一个堆(heap) 来实现的。每挖出一个新块，生成一个交易树。•在交易树包含的键值对中，其中每个键是交易的编号，值是交易内容。收据树：每个区块都有自己的收据树，收据树不需要更新，收据树代表每笔交易相应的收据。收据树也包含一个键值映射，其中键是索引编号，用来指引这条收据相关交易的位置，值是收据的内容。收据树工作原理：每个收据里都有一个 Bloom Filter，记录了此交易的类型、地址等相关信息。在区块头中有一个总的 Bloom Filter，是区块中所有交易收据的 Bloom Filter 的并集。当我们需要查找过去某段时间某个智能合约相关的所有交易时，第一步，在区块头中查找是否存在相关的交易类型；第二步，若存在，则在区块内部的所有收据里的 Bloom Filter 中查找。若第一步中没有查到，则无需第二步了，直接查找下个区块。综上可得，通过收据中的 Bloom Filter 可以快速排除掉无关的收据，提高了查询效率。

利用存储的三棵 MPT 树，以太坊客户端可以轻松地查询以下内容：
• 某笔交易是否被包含在特定的区块中。• 查询某个地址在过去的 30 天中发出某种类型事件的所有实例（例如，一个众筹合约完成了它的目标）。• 目前某个账户的余额。• 一个账户是否存在。• 假如在某个合约中进行一笔交易，交易的输出是什么。

Ethash 算法：用 Ethash 算法来代替原有的 PoW 算法，是为了解决挖矿中心化问题。Ethash 算法的特点：挖矿的效率基本与 CPU 无关，而与内存大小、带宽正相关，目的是去除专用硬件的优势，抵抗ASIC。该算法的基本流程：① 对于每一个区块，首先通过扫描区块头的方式计算出一个种子，该种子只和当前块的信息有关，然后根据种子生成一个 16M 的伪随机数据集（Cache）。② 基于 Cache 生成一个 1GB 大小的数据集合 DAG(有向无环图)，它是一个完整的搜索空间。③ 挖矿的过程就是从 DAG 中随机选择元素(类似于比特币挖矿中查找合适 Nonce)再进行哈希运算，可以从 Cache 快速计算 DAG 指定位置的元素，进而哈希验证。挖矿者存储数据集，该数据集周期性更新，每 3000 个块更新一次，并且规定 DAG 的大小随着时间推移线性增长，从 1G 开始，每年大约增长 7G 左右。④ 验证者能够基于缓存计算得到 DAG 中自己需要的指定位置的元素，然后验证这些指定元素的散列是不是小于某个散列值，也就是验证"矿工"的工作是否符合要求。

PoS-Casper •以太坊的PoS 共识协议称为 Casper，采用"权益"(Stake，即以太币）为记账权背书。•大致的思路是，将 Casper 的应用逻辑通过智能合约来实现，在这个合约中，记账权归属于"验证者"。任何拥有以太币的账户都可以在 Casper 合约中成为验证者，前提是必须要在 Casper 智能合约中抵押一定的以太币（抵押的以太币越多，被选中作为验证者的概率就越高）。之后 Casper 合约通过一种随机方式，选出一个验证者集合。被选中的验证者集合按照一定顺序依次验证区块（当然也可以选择放弃），如果区块没有问题，就将其添加到区块链中，同时相应的验证者将会获得一笔与他们的抵押成比例的奖励。如果验证者不遵守合约规定的规则，合约就会没收他抵押的以太币作为惩罚。

以太坊 Ghost 协议和叔块（Uncle Block）• GHOST（Greedy Heaviest Observed Subtree）是一种主链选择协议（不是侧链选择协议）。GHOST 协议是以包含子树数目最多为基本原则。• GHOST 协议，认为孤块也是有价值的，会给发现孤块的矿工以回报，也会给引用孤块的矿工以奖励。在以太坊中，孤块被称为"叔块"(uncle block)，它们可以为主链的安全作出贡献。• 叔块上的交易不会得到执行。
• 区块可以引用 0-2 个叔块。叔块必须是区块的前 2 层~前 7 层的祖先的直接的子块，被引用过的叔块不能重复引用。引用叔块的区块，可以获得挖矿报酬的 1/32，也就是 5*1/32=0.15625Ether。最多获得 2*0.15625=0.3125Ether。被引用的叔块，其矿工的报酬和叔块与引用它的区块之间的间隔层数有关系。

叔块的好处：**(1)**以太坊十几秒的出块间隔，大大增加了孤块的产生，并且降低了安全性。通过鼓励引用叔块，使引用主链获得更多的安全保证(因为孤块本身也是合法的) **(2)**比特币中，采矿中心化成为一个问题。给与叔块报酬，可以一定程度上缓解这个问题。

以太坊虚拟机（简称为 EVM）是以太坊协议和具体操作的核心。
• EVM 是一个基于栈的架构，是以太坊中的堆栈指令解释执行器，提供图灵完备的执行环境。其基本数据处理单元是 256 位的"字"。
• 由于智能合约的执行结果需要在以太坊的网络节点之间进行共识，因此 EVM 的所有指令执行必须具有确定性的输出。• EVM 是单线程的。• EVM 为了保证交易的可终止性，引入了执行器燃料 Gas，按照执行指令的复杂度、所需要的内存空间等对每个步骤进行计价，消耗指定数量的 Gas，当交易所赋予的 Gas 消耗为零时则停止指令的执行，退出执行环境。EVM 执行费用**(1)** 为了能够激励节点有足够的动力来维护以太坊网络的稳定运行，对提供节点算力的用户需要支付一定的奖励，这种奖励来源于两份方面：一是单纯的区块打包奖励；一是打包交易的手续费。**(2)**交易手续费在以太坊中通过交易执行的复杂度来定价，这种定价取决于执行交易需要消耗多少算力。以太坊的交易手续费也从一定程度上保证了智能合约不会出现执行后无法终止的情况，通过经济的手段防止了恶意的交易在区块链网络上传播从而引起网络瘫痪。**(3)**以太坊的 VM 除了执行指令消耗 Gas 外，还对存储空间的使用也按 Gas 进行计价。消耗的存储空间越大，那么需要支付的 Gas 费用也会越高。这个存储空间包括内存空间和持久空间，持久空间是指智能合约的 Storage 变量。

预言机：预言机定义• 区块链外信息写入区块链内的机制，一般被称为预言机(oracle mechanism)。•预言机的功能就是将外界信息写入到区块链内，完成区块链与现实世界的数据互通。•它允许确定的智能合约对不确定的外部世界作出反应，是智能合约与外部进行数据交互的唯一途径，也是区块链与现实世界进行数据交互的接口。•简言之，Oracle 可以视作 Smart Contracts 与外部交互的 API。为什么需要预言机？•区块链是确定性的环境，它不允许不确定的事情或因素，智能合约不管何时何地运行都必须是一致的结果，所以虚拟机（VM）不能让智能合约有直接的 network call（网络调用），否则结果就可能是不确定的。•智能合约无法主动去获取链外的数据，只能被动接受数据。预言机的工作流程•用户的智能合约把请求给链上 Oracle 合约，通过链下的 API 接口获得外部数据。外部把数据给链上的 Oracle 合约，然后 Oracle 合约再把数据给用户的智能合约。预言机当前的问题 •中心化预言机是非常易受攻击的。大多数项目选择使用自己搭建的预言机服务，但实际上这会导致单点故障、易受攻击等问题。因此，去中心化预言机在未来将成为更好的方案。•链上是无法生成随机数的，或者说在链上的随机数是可以被预测和破解的。比如说，大多数博彩游戏都是在链上生成随机数，所以很容易被预测和破解，导致资产被盗。这时候就需要预言机从外部给智能合约安全的、不可预测的随机数。Oracle 分类基于 DATA SOURCE • Software oracles。在互联网上处理来自网络的数据，通过浏览这些数据来找到它需要的信息，提取所需的信息，并将其返回到智能合同中。例如，黄金价格、航班延误和货币汇率。
• Hardware oracles。通过扫描仪和传感器等设备直接从物理世界收集数据。例如，用于跟踪供应链货物的 RFID 传感器，以及装运集装箱中的温度和湿度传感器。• Human oracles。依靠人的行为向区块链系统提供外部数据。Humanoracles 为智能合同提供了问题的答案。例如，人们可以对一个事件的真相进行投票。基于 TRUST MODEL • Centralized trust model。集中式信任模型依赖于来自单一来源的数据。Centralized trust model 的效率很高，但其可用性、可访问性和数据有效性仅依赖于一个节点，可能出现单点故障。
• Decentralized trust model。解决了 Centralized trust model 中的单点失效问题。然而与集中式信任模型相比，在数据处理上有更高的延迟，效率更低。基于 DESIGN PATTERN • Request-response。当数据空间太大而无法存储在智能合约中，并且预期用户一次只需要整个数据集的一小部分时，将使用请求-响应设计模式。此设计模式的一个适用用例是数据提供程序。这种设计模式可以实现为链上智能合约系统，其中对 oracle 的请求被发起，而链外基础设施用于监视请求、检索和返回数据。• Publish-subscribe。有效地为预期会发生变化的数据(如价格和天气信息)提供广播服务。此模式类似于 RSS feeds，其中使用新信息更新 oracle，并使用一个标志表示订阅者可以使用新数据。订阅者可以轮询 oracle 以检查最新信息是否已更改，也可以侦听 oracle 契约的更新并在更新发生时采取行动。
•Immediate-read。用于提供即时决策所需的数据的 oracle，如学位认证和拨号代码。这种类型的 oracle 将数据存储在其合约存储中，并可以对其进行更新。oracle 存储中的数据可以让任何其他智能合约使用对 oracle 合约的请求调用进行查询，也可以让支持区块链的应用程序直接查询这些数据。基于 INTERACTION •Inbound oracles。将外部世界的数据插入到区块链中。一个例子就是资产的价格，当它达到期望的价格时，就可以自动购买。•Outbound oracles。允许智能合同向外部世界传送数据。智能锁是一个例子，当在智能锁的区块链地址上收到付款时，实体世界中的智能锁会自动解锁。预言机服务举例：Oraclize，Chainlink，OracleChain，DOS network

智能合约区块链上的智能合约就是存储在区块链上的代码，用以实现执行特定的功能。•与传统合约相比，区块链智能合约有如下优势：**(1)**逻辑的明确性，不容易产生歧义。**(2)**不可能被篡改。合约执行记录，可以作为永久凭证。**(3)**合约执行的强制力可以保证。

智能合约运行机制•智能合约一般具有值和状态两个属性，代码中预置了合约条款的相应触发场景和响应规则。•智能合约经多方共同协定、各自签署后随用户发起的交易提交，经 P2P 网络传播，矿工验证后存储在区块链特定区块中，用户得到返回的合约地址以及合约接口等信息后即可通过发起交易来调用合约。•矿工受系统预设的激励机制激励，将贡献自身算力来验证交易，矿工收到合约创建或调用交易后在本地沙箱执行环境 EVM 中创建合约或执行合约代码 •合约代码根据可信外部数据源（也称为预言机 Oracle）和世界状态的检查信息，自动判断当前所处场景是否满足合约触发条件，以严格执行响应规则并更新世界状态。•交易验证有效后被打包进新的数据区块，新区块经共识算法认证后链接到区块链主链，所有更新生效。智能合约的基础架构模型 **•运维层：**封装了一系列对合约层中静态合约数据的动态操作，包括机制设计、形式化验证、安全性检查、维护更新、自毁等，是保证智能合约能够按照设计者意愿正确、安全、高效运行的关键。**•合约层：**封装静态的合约数据，可看作是智能合约的静态数据库，封装所有智能合约调用、执行、通信规则。包括合约各方达成一致的合约条款、合约条款代码化后的情景-应对型规则（if-then, whatif），以及按照平台特性和立契者意愿补充必要的智能合约与用户之间、合约与合约之间的访问权限与通信方式等。**•基础设施层：**封装了支持智能合约及其衍生应用实现的所有基础设施，包括分布式账本及其关键技术、开发环境和可信数据源等，这些基础设施的选择在一定程度上影响智能合约的设计模式和合约属性。智能合约生命周期：编译、创建、部署、调用、监听或销毁 去中心化应用 DAppDApp—后端（智能合约）智能合约被用来存储应用的业务逻辑（程序代码）和相关的状态信息，可看做一个常规的应用程序的服务端组件。以太坊的智能合约允许构建出复杂的架构，网络中的智能合约可以相互调用并传递数据，以及随时读写自己的状态变量。由于智能合约的执行非常昂贵，例如以太坊智能合约计算需要消耗 gas，因此其复杂性受限于区块 gas 的限制。智能合约架构设计的主要考虑因素：**(1)**一旦合约代码被部署，就无法对其进行修改。**(2)**DApp 的大小，一个体量非常大的合约在部署和使用的时候会花费非常多的 gas。DApp—前端（Web 用户界面）DApp 客户端界面开发只需要基础的 Web 前端技术。与 DApp 的交互，如签名消息、发送交易、密钥管理等，通常通过 MetaMask 等浏览器扩展工具完成，MetaMask 是一个开源的以太坊钱包。前端通常通过 Web3.js JavaScript 库连接到以太坊，该库与前端资源捆绑到一起，并由 Web 服务器提供给浏览器。Dapp—数据存储由于昂贵的 gas 费用以及当前较低的区块 gas 上限，智能合约不太适合存储和处理大量数据，因此大多数 DApp 利用链下数据存储服务。数据存储平台可以是中心化的，或者点对点形式（IPFS、Swarm）的。去中心化的点对点存储系统是存储和分发大型静态资产如图片、视频以及客户端应用程序其余部分的理想选择。星际文件系统 IPFS：是一个去中心化的、内容可寻址的存储系统，可以在 P2P 网络中分配存储的对象。内容的每一小块都被哈希处理，并且这个哈希值可以标识文件，通过哈希值在 IPFS 的任何节点可以将文件取回。Swarm：是以太坊基金会创建，是另一个内容可寻址，类似于 IPFS 的 P2P 存储系统。去中心化消息通信协议：任何应用都会包含的重要组件是进程之间的通信。这意味着不同的应用之间、相同应用的不同实例之间、以及同一应用的不同用户之间可以交换信息。通信可以依赖中心化的服务器进行，但也有许多基于 P2P 网络的替代方案。- Whisper：是一种加密通讯协议，允许节点间安全地直接发送信息，帮助信息发送者和接受者屏蔽掉多余的第三方。