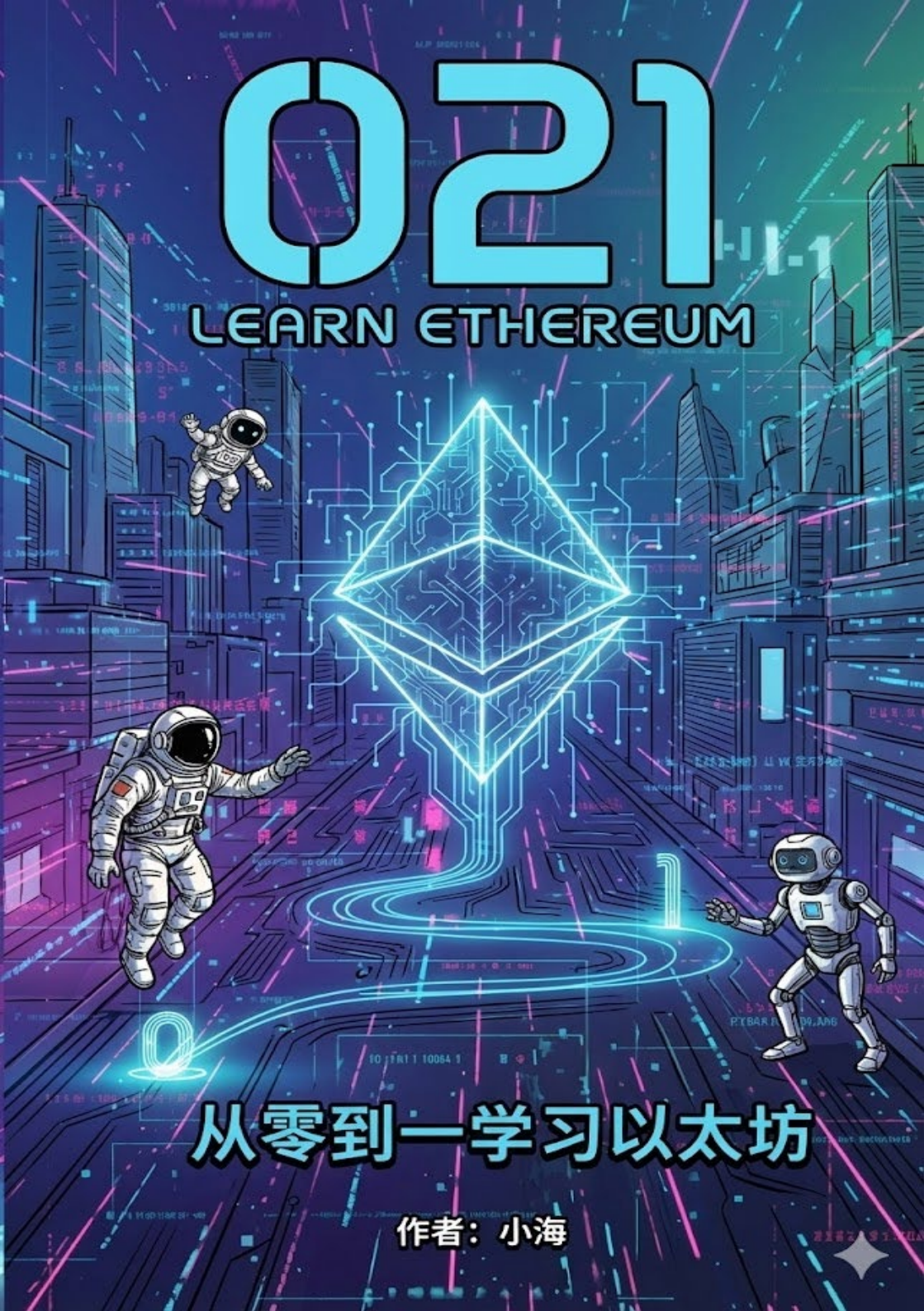

021
LEARN ETHEREUM
从零到一学习以太坊
作者：小海

从零到一学习以太坊

021 LEARN ETHREUM

作者：小海

由 LXDAO、ETHPanda 出版

2025 年 12 月

LXDAO

LXDAO 是一个专注研发的 DAO 组织，致力于探索构建支持公共物品和开源项目的无限循环。

Twitter / X

官网

ETHPanda

ETHPanda 是一个由华语建设者组成的以太坊社区，致力于通过教育、公共服务、活动和技术创新，连接华语建设者与国际以太坊生态，共同推动以太坊的持续发展与创新。

Twitter / X

官网

感谢 LXDAO 、ETHPanda 社区及以下伙伴对本书的贡献

（排名不分先后）

编排：kelvin-秦默、Leo-乐乐、Stariv-星河

目录 Contents

Contents

Contents

应用二进制接口（ABI）

描述合约对外可调用的函数、事件和参数格式的 JSON 文档，前端和其它合约用它来正确编码/解码调用数据与返回值。

自动化做市商（AMM）

去中心化交易所里的合约类型，通过预设数学公式和池子（Liquidity Pool）自动定价、撮合交易，不依赖传统订单簿。

账户抽象（Account Abstraction）

让账户行为更灵活的概念，把钱包能力（例如批量签名、Gas 代付、合约控制策略）从 EOA 转向合约或标准化接口，以优化用户体验和安全性。

借贷协议（例如 Aave）

在链上实现的合约集合，允许用户存入资产获取利息或借入资产并抵押，通常由智能合约自动管理利率和清算逻辑。

信标链（Beacon Chain）

以太坊 PoS 架构中的共识层，负责管理验证者、最终性（Finality）与分片/共识相关的投票与协调，不直接执行用户交易的 EVM 逻辑。

Berlin 升级

以太坊历史上的一次网络升级（若干 EIP 的集合），包含对 Gas 定价和安全性的若干调整，旨在提升效率与兼容性。

Besu 客户端

一种以太坊执行客户端实现（由 Hyperledger 社区维护），可用于运行节点、企业级部署与 RPC 服务。

Byzantium 升级

以太坊早期的重要硬分叉之一，包含多项 EIP 改进（如隐私、效率与安全性相关改动）。

常量变量（constant / immutable）

在合约中声明后不可变或仅在构造时设定的变量，读取成本低（不占 storage），节省 Gas。

Constantinople 升级

另一轮历史性升级，调整 Gas 成本并引入多项执行层改进以提升性能。

CREATE2 操作码

EVM 的一种创建合约的方式，允许根据部署者地址、salt 等预先计算目标合约地址，从而支持可预见的部署与某些“可替换逻辑”模式。

去中心化自治组织（DAO）

由智能合约规则驱动、通过代币持有者投票治理的组织结构，用于社区决策、资金管理与自动化治理。

去中心化应用（dApp）

运行在区块链（或利用区块链服务）的应用，前端可能是网页/移动端，后端逻辑由智能合约实现，具有去中心化与公开可验证的特点。

数据可用性分片（Danksharding / EIP-4844 等）

为降低 Layer-2（rollup）上链成本而设计的方案：在区块中增加“blob”或专用数据承载方式，提升数据可用性但不把所有数据永久写入状态树。

去中心化金融（DeFi）

使用智能合约提供传统金融功能（借贷、交易、衍生品、保险等）的生态体系，强调无需许可、可组合与透明性。

Dencun （坎昆）升级

以太坊的一次复合升级（包含 Deneb + Cancun 等改动），引入/完善了与数据可用性、预编译、执行层优化相关的特性（不同阶段的集合名）。

以太坊改进提案（EIP）

描述网络建议、标准或协议变更的文档化流程；重要 EIP 经社区讨论后可能成为网络升级的一部分。

以太坊请求评论（ERC — Ethereum Request for Comments）

ERC 是 EIP 的一种子类，专注于以太坊网络应用层的标准提案，主要用于定义智能合约接口和代币标准。

同质化代币标准（ERC-20）

用于发行可互换代币的合约接口标准，定义了 balanceOf、transfer、approve 等通用方法，方便钱包与交易所互操作。

非同质化代币标准（ERC-721）

NFT 的常用标准，定义了独一无二代币的铸造、转移与查询接口（如 tokenId 与 ownerOf）。

混合型代币标准（ERC-1155）

支持在同一合约内同时管理可替换（fungible）和不可替换（non-fungible）代币的标准，节省 Gas 并提高灵活性。

Erigon 客户端

一个以太坊执行客户端实现，强调高性能、存储与查询效率，经常用于数据密集型节点或索引服务。

以太坊虚拟机（EVM）

在每个节点上运行合约字节码的沙箱式虚拟机，确保所有节点以确定性方式执行相同逻辑并更新状态。

全同步模式（Full Sync）

节点从创世块开始逐区块执行所有交易并重建状态的方法，最安全但耗时耗资源。

Gas 费

EVM 中衡量和付费计算/存储资源的单位。每条 opcode 有对应的 Gas 消耗，交易总费 = GasUsed × GasPrice（或在 EIP-1559 之后按 BaseFee+Tip 计）。

Gwei（10^9 Wei）

常用的 Gas 单位，1 Gwei = 10^9 Wei，1 ETH = 10^{18} Wei，因此 Gwei 便于表示小额 Gas 价格。

Hardhat 开发框架

主流的 JavaScript/TypeScript 智能合约开发与测试框架，支持脚本化部署、单元测试、Gas 报告与插件体系。

Istanbul 升级

以太坊一次网络升级，包含多项 EIP，旨在改进兼容性、隐私与 Gas 定价。

基于 JSON 的 RPC 接口（JSON-RPC）

节点对外提供的标准远程调用接口，钱包和前端通过它发送查询或构建/广播交易。

Kademlia 分布式哈希表（DHT）

一种 P2P 节点发现与路由算法，用于高效定位网络中其他节点与资源，常用于以太坊的节点发现机制。

第一层协议（主链 / L1）

区块链的基础层，负责最终结算与安全（例如以太坊主网）；Layer-2 建设依赖其安全保证。

第二层扩展协议（L2）

构建在主链之上的扩容方案（如 Rollups、State Channels），把大量计算/数据异步或压缩后提交至 L1，从而降低成本并提高吞吐。

轻节点同步模式（Light Sync / Light Client）

只下载并验证区块头、不保存全部状态的节点模式，依赖完整节点提供证明以节省资源，适合移动或受限设备。

London 升级

含 EIP-1559 的重要升级，改变了 Gas 市场模型并引入销毁机制，改善交易费用体验与货币经济学。

最大可提取价值（MEV）

区块构建者通过重新排序、包含或排除交易所能提取的额外价值（例如三明治攻击或套利），是矿工/验证者与前端竞价中的重要考量。

合并（The Merge，PoW→PoS）

以太坊将共识机制从工作量证明切换到权益证明的重大升级，显著减少能耗并为后续扩容奠定基础。

Metropolis 阶段

以太坊历史上的一个升级阶段名（包含 Byzantium、Constantinople 等在内的中间演进阶段），标识协议演进里程碑。

nonReentrant（防重入）

防止合约在外部调用返回前被递归重复调用的一种模式或修饰器（如 OpenZeppelin 的 ReentrancyGuard），用于防范重入攻击。

点对点网络（P2P）

节点之间直接连接与交换数据的网络架构，没有中心化服务器，网络通过邻居发现和 gossip 协议传播消息。

区块提议者与构建者分离（PBS）

一种提议将“谁构建区块”（builder）与“谁提出/签署区块”（proposer）分离的设计，旨在降低 MEV 负面影响并改进经济激励。

默克尔-帕特里夏树（Merkle-Patricia Trie）

以太坊用于存储账户与合约状态的树形数据结构，支持高效的状态证明与快速验证单条存储项。

权益证明（PoS）

共识机制类型，节点通过质押代币获得验证资格并按比例获得奖励，同时可因恶意行为被惩罚（slashing）。

工作量证明（PoW）

早期的共识机制，节点（矿工）通过计算哈希难题竞争出块，消耗大量电力与算力资源。

拉取付款模式（Pull Payment Pattern）

把付款从“推送”改为“记录可提取余额，由用户主动提取”的设计，避免在循环或大量受益者场景下的 Gas/DoS 风险。

原型数据可用性分片（Proto-Danksharding / EIP-4844）

Danksharding 的先行实现，通过引入可携带大量临时 blob 的交易类型，大幅降低 rollup 提交数据的成本并改善扩容路径。

Prysm 客户端

一种流行的以太坊共识客户端实现（多用于信标链与验证者运行），负责 PoS 层的投票与提议逻辑。

RLPx 加密通信协议

以太坊底层的 P2P 加密通信协议，负责节点之间建立加密连接并在上面传输子协议消息（如区块/交易数据）。

远程过程调用（RPC）

一类让客户端远程调用节点方法的接口机制，JSON-RPC 是其常见实现，前端与脚本通过它访问区块链功能。

浏览器 IDE（Remix）

在浏览器中即可使用的智能合约开发环境，支持编写、编译、部署与调试合约，适合入门与快速原型。

重入攻击（Reentrancy）

当合约在外部调用返回前没有先更新内部状态时，攻击者可反复触发调用并多次提取资金的一类漏洞。

分片（Sharding）

把链的状态与交易分割到不同分片并行处理的扩容思路，用于横向扩展吞吐能力。

验证者惩罚机制（Slashing）

PoS 中对恶意或严重失职验证者的经济惩罚（扣除质押并强制退出），用于维持网络安全与激励诚实行为。

Solidity 语言

以太坊上最常用的智能合约编程语言，语法类似 JavaScript/C++，用于编写合约逻辑并编译为 EVM 字节码。

快照同步模式（Snap Sync / State Sync）

节点通过并行下载最新状态快照与区块头而快速完成同步的方式，比全同步快很多，常为全节点默认选择。

Truffle 开发框架

早期且成熟的智能合约开发工具链，包含迁移（部署）、测试和本地链（Ganache）支持。

总锁仓价值（TVL）

衡量 DeFi 生态规模的指标，表示被协议锁定的资产总价值（通常以美元计）。

未花费交易输出（UTXO）

比特币式账户模型下的交易单元概念（每笔输出可被后续交易作为输入消费），与以太坊的账户模型不同但常在比较时被提及。

多叉向量承诺树（Verkle Tree）

一种替代 Patricia Trie 的承诺数据结构，能显著缩短证明大小并使轻客户端验证更高效。

可验证随机函数（VRF）

提供可验证且不可篡改的随机数源（链下或链上结合预言机），常用于抽签、分配或随机化逻辑。

Wei（10^{-18} ETH）

以太坊的最小计价单位，1 ETH = 10^{18} Wei，交易/余额在链上以 Wei 存储以避免小数误差。

白皮书（Whitepaper）

项目早期的技术与愿景说明文档，阐述设计理念、经济模型与协议细节，通常作为项目前期文献。

Zeppelin / OpenZeppelin（安全库）

广泛使用的开源合约库，提供经审计的可复用组件（如 ERC 标准实现、权限与安全守卫）。

UDP（User Datagram Protocol)-用户数据报协议

是一种无连接、面向数据报的传输层协议，广泛应用于对实时性要求高的场景，如视频通话、在线游戏和 DNS 查询等。

TCP（Transmission Control Protocol）-传输控制协议

是互联网协议族中的核心协议之一，位于传输层，负责在网络中提供可靠的、面向连接的字节流服务。它与 IP 协议共同构成了 TCP/IP 协议栈，是现代网络通信的基础。

MCP（Model Context Protocol）-模型上下文协议

是由 Anthropic 于 2024 年 11 月推出的开放标准，旨在为大型语言模型（LLM）提供一种标准化的方式，以便与外部数据源、工具和服务进行安全、灵活的交互。

Web3 世界日新月异，新名词和新概念层出不穷。对于初学者而言，建议利用在线文档库（例如飞书、语雀、Notion 等）构建个人专属的词库。将新词汇及其解析逐一整理归档，闲暇时刻时常翻阅，这将有助于快速填补知识空白，提升对 Web3 领域的理解。

——@kelvin-秦默

第一章

认识以太坊（What is Ethereum）

本章目标：理解以太坊的定义、历史和核心用途。

一、以太坊的起源与发展

以太坊最初由 Vitalik Buterin（也就是 V 神）在 2013 年提出，并在 2015 年 7 月 30 日随“Frontier（边境）”阶段正式上线主网。

Vitalik Buterin 本是一名参与比特币社群的程序员兼《Bitcoin Magazine》联合创始人，他曾向比特币核心开发人员主张，比特币平台应该要有更完善的编程语言，方便开发者在其上编写各类程序，但未得到他们的同意，因此决定开发一个新的平台来实现这一目标。

Vitalik Buterin 认为，很多程序都可以用类似比特币的原理来实现更进一步的发展，于是在 2013 年写下了《以太坊白皮书》，说明了构建去中心化应用平台的目标。随后，以太坊的开发工作在 2014 年初正式启动，并在 2014 年 7–8 月透过网络公开募资（众筹）获得开发资金，投资人用比特币向项目方购买以太币（Ether）。

2014 年 1 月，以太坊在迈阿密举行的北美比特币大会（North American Bitcoin Conference）上正式对外发布。会议期间，Gavin Wood、Charles Hoskinson 和 Anthony Di Iorio（项目融资人）与 Vitalik Buterin 在迈阿密租了一套房子，共同探讨以太坊的发展及未来。

据 Anthony Di Iorio 介绍，以太坊于 2013 年 12 月由 Vitalik Buterin、Anthony Di Iorio、Charles Hoskinson、Mihai Alisie 和 Amir Chetrit（最初的 5 人）创立。2014 年初，Joseph Lubin、Gavin Wood 和 Jeffrey Wilcke 加入了创始人行列（当然后面因为一些观念上的不合，有的人最终选择了离开）。V 神在浏览维基百科上的科幻元素列表后选择了 “Ethereum（以太坊）” 这个名字。他说：“我立刻意识到，‘以太’这个词是我迄今为止看到的最佳选择，它指的是一种假设的、

遍布宇宙并允许光传播的不可见介质。”这与 V 神希望以太坊平台成为在其上运行的应用程序的底层和不可察觉的“基础介质”的初衷类似。（在漫威宇宙里，“以太”也是一颗无限原石——现实宝石。）

最初，以太坊程序是由一间位于瑞士的公司 Ethereum Switzerland GmbH（简称 EthSuisse）开发，之后逐步转移至一个瑞士的非营利机构——“以太坊基金会”（Ethereum Foundation）。在平台开始发展的最初阶段，就已经有人称赞以太坊在智能合约和去中心化应用方面的技术创新；当然事物都有两面性，也有人质疑其安全性和可扩展性。

以太坊早期的官方路线图，主要将网络开发划分为四个阶段：边境（Frontier，也有“前锋”的意思）、家园（Homestead）、都会（Metropolis）和宁静（Serenity）。其中，2015 年 7 月 30 日主网上线属于 Frontier 阶段，2016 年的 Homestead 升级标志着网络趋于稳定，Metropolis 又细分为 Byzantium 和 Constantinople 等多次升级；宁静（Serenity）阶段则通过信标链与 2022 年的“合并”（The Merge）等升级，实现了以太坊从工作量证明（PoW）向权益证明（PoS）的核心过渡，之后又通过 Dencun（2024）、Pectra（2025）等升级继续演进，以接近其更高扩展性和可持续性的长期愿景。

二、以太坊平台定位与核心特性

以太坊（Ethereum）是一个去中心化、开源并且具备智能合约功能的公共区块链平台。其核心组件是以太坊虚拟机（Ethereum Virtual Machine，EVM），它是一个图灵完备的虚拟机，能够执行智能合约代码，让区块链不再只是“记账”，而是可以运行各类程序和应用。

以太坊使用名为“以太”（Ether，缩写为 ETH）的加密货币作为其内部交易的“燃料”（Gas），用于支付交易费用和计算服务；在以太坊的 Rollup-为主路线图下，ETH 既是资产，又是整个 L2 生态的结算与数据可用性“底层油料”。

从市值和使用度来看，截至 2025 年，以太币长期稳居加密市场市值第二，仅次于比特币，市值在数千亿美元量级；以太坊也是当前 DeFi、NFT 等应用最活跃的公链之一。以太坊网络的日活跃地址常年维持在几十万到上百万之间，并在 2025 年 10 月创下单日约 198.5 万活跃地址的新高，若按“月度累计活跃地址”口径统计，则可达到数千万级规模。

以太坊还允许用户创建和交换不可替代代币（NFT，Non-Fungible Token）。这些代币通常遵循 ERC-721 或 ERC-1155 等标准，可以与独特的数字资产（例如图像、音乐、游戏道具或虚拟土地）相关联。

此外，大量其他加密资产在以太坊区块链之上，利用 ERC-20 代币标准发行和流通，早期 ICO 乃至现在的稳定币、治理代币等，都大量采用 ERC-20。

在很多用户眼中，以太坊被称为“第二代区块链平台”——在比特币之后，把区块链从“记账系统”拓展成“可编程的全球结算与应用平台”。

特点

相较于大多数其他加密货币或区块链技术，以太坊的特点包括但不限于以下几点：

1. 智能合约（Smart Contracts）

智能合约是存储在区块链上的程序，由网络节点执行。现在以太坊已从早期的“矿工（PoW）”时代完全过渡到“验证者（PoS）”时代，这些验证者负责打包并执行合约。任何想要执行合约的人，都需要支付 Gas 作为手续费。

比喻：

可以把智能合约想象成自动贩卖机：

你投入硬币（支付 Gas），选择饮料（调用合约函数），机器就自动出货（执行转账或业务逻辑），无需店员干预。

每一笔操作都由分布式节点“自动执行”，并为此收取手续费，而且整个过程是公开可验证、不可随意篡改的。

2. 分布式应用程序（dApps）

以太坊上的分布式应用程序（dApp）部署在区块链上，不依赖单一服务器，也没有传统意义上的“后台可关机”。只要以太坊网络和客户端软件还活着，dApp 就很难被单一主体关停。

比喻：

分布式应用就像一间“永不打烊的 24 小时自动商店”。

它运行在无数台电脑上，就算某些电脑坏了，其它电脑依然可以继续“营业”；没有哪一个人能按下总电源开关让整个应用消失。

3. 代币（Tokens）

智能合约可以创造代币供分布式应用程序使用，这些代币可以是：

- 可替代的（ERC-20）：例如游戏里的“金币”、治理代币、稳定币等；
- 不可替代的（ERC-721 / ERC-1155）：对应独特资产，如艺术品、门票或游戏角色。

比喻：

可以把代币想成游戏中的“金币”或“点券”，智能合约可以发行这些“金币”，供 dApp 使用。

通过代币，玩家（用户）、投资者、管理者的利益被绑定在一起：大家都希望游戏（应用）做得越好、代币越有价值。

而在融资场景里，代币还可以像电影上线前的“预售券”，在项目正式推出之前就向早期支持者出售——这就是人们常说的 ICO 或代币发行，只是现在监管和合规要求比早期要严格得多。

4. 权益证明（PoS）与早期的工作量证明（PoW）

以太坊最初采用的是工作量证明（Proof-of-Work, PoW）机制，和比特币类似，由“矿工”通过算力竞争出块；自 2022 年 9 月 15 日的 The Merge 升级起，以太坊**完全改用权益证明（Proof-of-Stake, PoS）**，淘汰了挖矿机制，能耗降低约 99.95%。

PoW（工作量证明，历史上的以太坊共识）

- **机制原理：** 节点（矿工）投入算力“挖矿”，竞争解出复杂数学难题，以获得打包新区块的权利和区块奖励。
- **安全性来源：** 依赖高昂的硬件与电力成本——攻击者要想“51% 攻击”，就必须付出极其巨大的算力和电费。

今天，以太坊主网已经不再使用 PoW，但理解 PoW 有助于对比共识机制的差异（例如你也可以顺带对比以太坊与比特币、以太坊与以太坊经典）。

PoS（权益证明——以太坊当前使用的共识）

- **机制原理：** 节点（验证者）需要锁定（质押）一定数量的 ETH（标准门槛为 32 ETH）成为验证者；每个时隙由协议随机选出提议区块的验

证者，并由其他验证者投票确认。质押越多、表现越稳定的验证者，长期来看被选中参与记账的频率更高。

- **安全机制：** 恶意行为（例如双签、试图重组链）会触发“削减（slashing）”，验证者的部分或全部质押资产会被销毁或罚没；长时间离线也会逐步被罚没，形成强约束的经济激励，使验证者必须诚实在线地维护网络安全。

简单总结就是：**PoW 用电力和硬件“押注诚实”，PoS 用锁定的 ETH 和削减机制“押注诚实”**，而以太坊已经完成从前者向后者的迁移。

5. 燃料费（Gas）

“燃料”（Gas）是对计算与存储资源的计量单位，发送 ETH、转账代币、调用合约函数、部署合约，统统会消耗 Gas。EIP-1559 之后，Gas 费由“基础费（base fee）+小费（priority fee）”组成，其中基础费会被销毁，小费奖励给出块验证者。

比喻：

执行合约或转账就像开车：

- 每跑一步、每做一次计算操作都要“烧油”；
- ETH 就是你加在油箱里的燃料；
- 车子开的越复杂、越远（例如复杂 DeFi 操作），耗油就越多。

6. Proto-Danksharding（EIP-4844）与 Dencun 升级

Proto-Danksharding（原型丹克分片，EIP-4844）已在 2024 年 3 月 13 日的 **Dencun 升级** 中正式上线主网。它是以太坊迈向完整 Danksharding 的中间升级：

- 引入了新的“携带 blob 的交易类型”（blob-carrying transactions），为 Rollup 提供**临时数据空间**；

- 这些 blob 数据只在共识层保存一段时间后被丢弃，不进入永久状态，从而极大减轻 L1 的存储压力；
- 实际效果是：L2 发布数据的成本大幅下降，很多 Rollup 的交易费用下降了 90% 甚至接近 99%。

比喻：

想象你开着车（普通以太坊交易），旁边挂着一个“小挂车”（blob 数据）：

- 你把大批货物（Rollup 的批量交易数据）先装到挂车上；
- 车跑完该跑的路，过一段时间挂车会自动脱落回收，不再长期占用空间；
- 主车（L1 状态）保持轻量，而 L2 的数据传输成本大幅降低。

当前网络对 blob 数量采用保守限制：

- Dencun 上线初期每个区块最多约 6 个 blob，目标平均 3 个；随着 PeerDAS 等技术部署，预计可逐步提升到每块十几个乃至几十个 blob。

7. 分片与 Danksharding

基础概念：

“分片”本质上是把一个大系统拆成多个“分区”并行处理，从而提升吞吐量。以太坊早期路线图曾设想通过“64 条分片链”来扩容。

在以太坊中的最新设计：

- 新路线不再强调“多条执行分片链”，而是转向 **Danksharding**：它更像一种“数据分片（data-sharding）”，核心是引入更多 blob 数据空间，加上数据可用性采样（DAS），让节点只需要随机抽样检查一小部分数据，就能对整个大数据块“有把握”。
- Proto-Danksharding 是迈向完整 Danksharding 的第一步；完整 Danksharding 的目标，是在保持去中心化前提下，将每个区块可用的数

据空间扩展到几十甚至上百个 blob，具体数值（如 64、128、256 等）仍在研究和参数调优中。

可以粗略理解为：**Proto-Danksharding 先给以太坊挂上“小挂车”，而完整 Danksharding 则是把挂车扩展成“高速货运列车”，专门为 L2 输送数据。**

8. 叔块（Uncle / Ommer Block）

叔块在转为权益证明后已停用。补充一点背景，方便读者理解这一概念的历史位置：

- 在 PoW 时期，以太坊的出块是概率性的，有时会出现几乎同时挖出两个有效区块的情况。主链只会选择其中一个，另一个则成为“叔块”（或称 *ommer*），为减少因网络延迟对中小矿工的不公平，以太坊会对被包含的叔块给予部分奖励。
- The Merge 之后，以太坊转为 PoS，共识流程完全不同，不再产生新的叔块；相关字段在区块头中基本被置零，仅为兼容历史数据和工具而保留。

比喻：

想象两个演员同时冲向领奖台，但只有一个能上台领奖；另一个虽然表现不差，只是运气稍差没被选中。主办方为了鼓励大家，给没上台的那位发了个“安慰奖”以示认可——叔块在 PoW 时代扮演的就是这种“安慰奖”角色。

在今天的以太坊 PoS 里，这个奖项已经不再颁发，但它仍然是理解以太坊早期设计的重要一环。

下面是一张“开发者友好”总览图：

三、Ether（ETH）的定义与系统职能

以太币（Ether，ETH）是以太坊区块链上的**原生数字资产**，和比特币（Bitcoin）一样，都是可以点对点发送、无需银行等中介机构的数字货币，你可以在各大加密货币交易所买卖 ETH，它有真实且浮动的市场价格。

但与主要被定位为“数字黄金”的比特币不同，**ETH 的设计初衷更偏向“实用 +资产”**：

既是支付 Gas 的燃料，又是质押安全的保证金，同时还是 DeFi / NFT / Web3 生态中最重要的抵押资产之一。

ETH 三大核心作用

1. 网络燃料（Gas）：支付交易和计算费用

这是 ETH 在协议层面**最根本、最核心的作用**。在以太坊网络上，任何操作：

- 转账一笔 ETH；
- 执行一个智能合约；
- 铸造一个 NFT；
- 与 DeFi 应用交互（Swap、借贷、质押等）；

都会消耗链上计算与存储资源。为了防止垃圾交易或恶意程序无限消耗资源，这些操作**都必须付费**，这个费用就是我们常说的 **Gas 费**，并且**必须用 ETH 支付**。

在 PoS 时代，用户支付的 Gas 费主要流向：

- **基础费（Base Fee）**：由协议自动计算，随网络拥堵程度上下浮动；这部分 **被直接销毁（burn）**，不进入任何人的口袋；
- **优先费 / 小费（Priority Fee / Tip）**：由用户设置，作为“加急费”支付给出块验证者，用来激励他们优先打包你的交易。

你在 L2（比如 Optimism）铸 NFT 时支付的那一点 op_ETH，本质上就是该 L2 上的 Gas 费；底层仍然会通过结算，把部分成本最终反映到以太坊 L1 的 Gas 上。

1.1 Gas 费的两个公式（旧机制 vs 现机制）

①EIP-1559 之前（旧版）

```
Gas Fee = Gas Used × Gas Price
```

- *Gas Used*：交易实际消耗的 Gas 单位数；
- *Gas Price*：用户愿意为每单位 Gas 支付的价格（如 20 Gwei）。

②EIP-1559 之后（当前主网机制）

```
实际支付费用 ≈ (Base Fee + Priority Fee) × Gas Used
```

- *Base Fee*：协议根据区块拥堵情况自动调整，并在执行后被销毁；
- *Priority Fee*：用户自愿支付的小费，用于激励验证者优先打包；
- 钱包里看到的 `maxFeePerGas` / `maxPriorityFeePerGas` 是**你愿意支付的上限**，实际扣费不会超过这两个上限组合对应的值。

1.2 为什么每天 Gas 费忽高忽低？

可以归纳成三类因素：

①网络拥堵（供需决定价格）

- DeFi 高峰、NFT 热点铸造、牛市情绪爆发时，大家都争着抢区块空间，Base Fee 和 Tip 都会上升；
- 反之，在交易较少的时段（如部分时区的深夜、周末），Gas 会明显下降。

②全球用户活跃时间差

- 以太坊是 24 小时全球网络，美洲和欧洲工作时间段（例如美国东部时间早 8 点到下午 1 点），dApp / DeFi 交互往往更集中，Gas 通常偏高；
- 在这些区域的夜间或假期，活动量下降，Gas 相对便宜。

③交互复杂程度

- 简单的 **ETH 转账** 只需要基础 Gas（典型值 21,000）；
- 与复杂合约交互（例如多步的 Swap、Mint NFT、批量操作）会消耗更多 Gas 单位，费用自然更高。

可以把它理解为：

> 你想让这台“世界计算机”为你工作，就必须用 ETH 给它“加油”；
> 油价受行情、路况（拥堵）、行程复杂度影响，和现实世界非常像。

2. 参与网络共识与安全（Staking）

自 2022 年 9 月 15 日 “合并”（The Merge）以来，以太坊已经从工作量证明（PoW）彻底转向更节能的权益证明（Proof-of-Stake, PoS）共识。

在 PoS 机制下，ETH 在共识层扮演了三个关键角色：

2.1 质押（Staking）资产

- 用户可以将持有的 ETH 锁定（质押）在网络中，运行验证者节点；
- 验证者负责：存储数据、验证交易、提议新区块和对其投票（attestation）。

2.2 获得奖励（Rewards）

验证者的收入主要来自两部分：

- 共识层奖励（Consensus Layer Rewards）
  来自 PoS 协议本身的 ETH 增发，对应你参与出块与投票的行为。

- 执行层奖励（Execution Layer Rewards）

  来自用户支付的 优先费（tip） 和可能存在的 MEV（最大可提取价值），由打包区块的验证者获得，波动更大但有放大利润的空间。

有些协议还会在你通过其平台质押时**额外抽取服务费**或提供二次激励（例如发放额外代币、返佣等），这属于协议层的设计。

2.3 提供经济安全（Security /“保证金”）

- 质押的 ETH 就像验证者的保证金：
  - 行为诚实、在线率高 → 按规则拿奖励；
  - 作恶 → 面临 **Slashing（削减）** 和强制退出。
- 质押在系统中的 ETH 越多、越分散，攻击者要控制足够多权益发动攻击的成本就越高。

3. 奖惩机制：为什么验证者会被奖励或被 Slashing？

我们为什么要质押 ETH？

直觉上当然是因为有奖励——你在帮网络“干活”，帮大家记账、维护安全，自然应该拿报酬。

3.1 奖励：验证者的收益构成

在 PoS 机制中，质押 ETH 并运行验证者节点，可以获得：

- **共识层奖励：**
  - 为链投票（attestation）、正确提议区块、参与同步委员会（sync committee）等都会获得奖励；
  - 奖励以 ETH 形式在共识层记账。
- **执行层奖励：**
  - 包括用户支付的优先费（tip）、可能存在的 MEV 收入等；

- 直接打到你设置的执行层收益地址（fee recipient）。

- **额外激励（由协议或第三方提供）：**
  - 某些质押协议、LSD/LRT（流动性质押）协议会额外发代币或奖励；
  - 某些场景下，对“举报 slashable 行为”的举报者（whistleblower）还会给额外 ETH 奖励。

简单理解：

> PoS 验证者的收入 = **协议给的工资（共识奖励）** + **用户给的小费 / MEV（执行奖励）** + **某些协议给的补贴**。

3.2 惩罚：什么是 Slashing？为什么要这么狠？

Slashing 是 PoS 系统里最严厉的惩罚，用来针对那些危害网络安全的行为，而不是单纯“偶尔掉线”：

典型的可 Slashing 行为包括：

- 对同一个 slot **双重提议区块**（proposer double block）；
- 对同一高度同时给两个不同区块投票（double vote）；
- 提交 “环绕投票”（surround vote），试图篡改链的最终性。

这类行为一旦被发现，会触发一整套 Slashing 流程：

①立即罚没一部分质押

- 协议立刻烧掉验证者有效余额的一小部分（典型约等于 32 ETH 的 1/32，即约 1 ETH，随实际余额等比例缩放）。

②长达约 36 天的移除期（Removal Period）

- 被 Slashing 的验证者会被从活跃集合中移除，并进入一个大约 36 天的退出期；

- 在这段时间里，验证者还会持续受到额外罚没（类似“你已经被边缘化了，但还得为之前的错误慢慢付出代价”）。

③相关性惩罚（Correlated Penalties）

- 如果在同一个约 36 天窗口中，大量验证者一起被 Slashing（比如一个大质押池集体作恶或被攻击），每个被 Slash 的验证者会被**额外加重罚没比例**，以防止“串谋型攻击”。

④举报奖励（Whistleblower Rewards）

- 发现别人作恶并提交证据的“举报者”，可获得一小部分被 Slash 的 ETH 作为奖励，进一步激励大家监督彼此。

> 补充：普通的“短时间掉线”不会触发 Slashing，但会因为错过投票而被轻微罚没；只有真正危害共识安全的行为，才会触发上面的严厉 Slashing 流程。

4. 价值储存和交换媒介

除了“燃料”和“保证金”这两个功能性角色外，ETH 也在更广泛的加密经济中扮演着**价值载体**的角色：

4.1 数字货币与投资资产

- ETH 长期稳居全球市值第二大的加密货币，仅次于比特币；
- 很多投资者把 ETH 当作“押注 Web3 / DeFi / 智能合约平台”核心资产，长期持有，期待网络使用度与协议收入的增长能够支撑其长期价值。

4.2 DeFi 生态的抵押品与流动性基石

- 在各类借贷协议（如抵押 ETH 借稳定币）、衍生品协议、流动性池（AMM/LP）里，ETH 都是使用最广泛、信用最好、流动性最强的底层资产之一；

- 大量 LSD/LRT 协议（stETH、rETH 等）本质上都是基于质押 ETH 的衍生资产，以 ETH 的安全性为背书。

4.3 NFT 与数字经济的基石货币

- 绝大多数 NFT（艺术、收藏品、游戏道具、虚拟土地等）的铸造与交易，仍主要以 ETH 计价与结算；
- 对很多创作者与玩家来说，“ETH = 进入 NFT / 元宇宙 / 游戏经济的门票与通用货币”。

5. NFT 与同质化 / 非同质化代币

5.1 什么是 NFT（非同质化代币）？

NFT（Non-Fungible Token）是在区块链上具有**唯一标识**的加密代币，每一枚 NFT 都代表某个特定、不可互换的资产：

- 数字艺术（画作、插画、摄影）；
- 收藏品（卡牌、周边）；
- 音乐、视频、门票；
- 游戏角色、装备、虚拟土地……

它更像是数字世界里的“**所有权凭证**”或“**收藏证书**”：

链上的记录清楚写明**谁在什么时候铸造、谁买走了它、现在谁拥有**，从而减少造假和重复发行的空间。

5.2 常见 NFT 标准：

- **ERC-721：**
  - 以太坊上最经典的 NFT 标准；
  - 每个 tokenId 对应一件独立资产，适合 1/1 艺术品或稀有收藏品。

- **ERC-1155：**
  - 支持在同一个合约里同时管理**同质 + 非同质 + 半同质**代币；
  - 支持批量转账，适合游戏内大量道具、成套收藏。

5.3 那什么是“同质化代币”（Fungible Token）？

“同质化”指的是：

- 每一个单位完全等价、可互换；
- 可以拆分成更小单位；
- 不关心“具体这一枚是谁”，只关心数量。

典型例子：

- 每一枚 1 美元纸币；
- 每一个 BTC、每一个 ETH；
- 绝大多数基于以太坊发行的 **ERC-20 代币**（稳定币、治理代币等）。

它们更适用于**货币、计价单位、储值和高流动性资产**的角色，而 NFT 更适合表达**唯一性与所有权**。

四、以太坊为何被誉“全球可编程区块链”

我们可以把这个称号拆解成三个部分来理解：**全球性（Global）**、**可编程（Programmable）** 和 **区块链（Blockchain）**。

1. 区块链（Blockchain）——信任的基石

首先，以太坊是一个区块链。这意味着它继承了区块链技术的所有基本属性：

- **去中心化（Decentralized）：**

网络由分布在全球的成千上万台计算机（节点 / 验证者）共同维护，没有单一中心控制点，不会被某个公司或政府一键关停。

- **不可篡改（Immutable）：**

一旦交易被打包进区块并通过共识确认，就极难被回滚或删除，历史记录具有极强的抗篡改性。

- **安全透明（Secure & Transparent）：**

所有交易和智能合约执行记录都是公开的，任何人都可以下载区块数据并验证，这让整个系统具备“可验证透明”的属性，而不是“黑箱信任”。

这些是它的**地基**。

比特币同样具备这些属性——它也是一个全球性的区块链，只是功能更聚焦在“价值转移和储存”上。

2. 可编程（Programmable）——让以太坊脱颖而出的关键

真正让以太坊从一堆公链里“跳出来”的，是它的**可编程性**。

在比特币网络上，脚本语言是刻意被设计得非常有限的：功能以转账和简单条件控制为主，更像是一台只能做加减法的“**金融计算器**”。

以太坊则引入了更强大的抽象：**智能合约（Smart Contracts） + 图灵完备虚拟机（EVM）**。

3. 什么是智能合约？

智能合约是一段**部署在区块链上的代码**。

当预设条件被满足时，它会自动执行“if…then…”逻辑，不需要人工审批，也不依赖某家公司“说了算”。

可以这样类比：

> 智能合约就像一台“数字世界的自动售货机”：
> 投入正确的“钱”（ETH 或其他代币 / 数据），
> 按下按钮（调用函数），
> 它就自动完成对应动作（转账、铸造、清算、投票……），
> 全程不需要店员，看得见规则、看不见人情。

4. 图灵完备（Turing-complete）

“图灵完备”这个术语本质上就是：

> 只要给足时间和资源，就能表达几乎任何可计算的逻辑。

以太坊的智能合约语言（Solidity、Vyper 等）在 EVM 上运行，使得以太坊不再只是一个“转账系统”，而是：

> 一台分布式的通用计算机，可以跑各种程序，只是这些程序的执行结果会被写进一个全网共享的状态机里。

基于这一点，开发者可以在以太坊上构建几乎无限种类的去中心化应用（dApps），比如：

- **去中心化金融（DeFi）：**
  借贷协议、去中心化交易所、衍生品、稳定币、收益聚合器……
- **非同质化代币（NFT）：**
  艺术品、收藏品、游戏道具、虚拟土地等的数字所有权。
- **去中心化自治组织（DAO）：**
  规则写进合约，由代币持有者投票治理，无“董事会拍脑袋”。
- **以及供应链、数字身份、链上游戏、去中心化社交媒体等等。**
  这些都是“**可编程**”这三个字带来的扩展空间。

5. 全球性（Global）——“世界计算机”的愿景

“全球性”不只是“世界各地都有节点”这么简单，更深一层的含义是：

> 整个以太坊网络像一台单一的、全球共享的“世界计算机（World Computer）”。

- 这台“计算机”不属于任何人，但任何人都可以使用；

- 它由遍布全球的成千上万节点组成，在 PoS 时代由验证者来打包和确认区块；
- 它**强抗审查、强抗宕机**，任何单一节点离线，都不会让整个网络停摆；
- 任何人，无论身处哪个国家，只要能连上互联网，就可以：
  - 部署自己的智能合约；
  - 调用别人部署的合约；
  - 把自己的资产、身份、应用逻辑绑在这个全球共享的状态机上。

从技术视角看，以太坊维护的是一个全球共享的**状态机（State Machine）**：

- 里面记录了**所有账户的余额**；
- 所有智能合约的存储变量、内部状态；
- 所有已发生的交易与事件（event logs）。

每一次交易或合约调用，都是在**修改这台全球状态机的状态**。只要区块被最终确认，全网就对“最新状态”达成共识。

所以，以太坊不是一个只用来记账的“去中心化账本”（那是比特币的典型定位），而是：

> 建立在全球、去中心化、抗审查基础设施上的一台通用“世界计算机”，任何人都可以在上面写程序、跑逻辑、发资产。

五、以太坊与比特币的异同对比

1. 相似点

①区块链基础

两者都基于区块链和分布式账本，保证交易的公开透明与难以篡改。

②去中心化与共识机制

都不依赖中央机构，而是通过共识协议维持网络安全和状态一致：

- 比特币使用 PoW（工作量证明）；
- 以太坊自 2022 年 9 月 15 日 “The Merge” 之后完全切换为 PoS（权益证明）。

③数字资产属性

BTC 与 ETH 都是可以跨境转账、可自由交易的数字资产；价格由市场供需决定，波动很大，也带有投机属性。

④活跃社区

两者都有庞大的开发者与用户社区：

- 比特币社区聚焦“货币与价值储存”；
- 以太坊社区则在 DeFi、NFT、L2、DAO 等方向快速演化。

2. 不同点

特 征	比特币（Bitcoin）	以太坊（Ethereum）
目标定位	“数字黄金”：去中心化货币与长期价值储存	“全球可编程计算机”：可运行智能合约和 dApps 的通用平台
共识机制	一直使用 **PoW**，依赖算力挖矿，能耗较高	自 2022-09-15 完成 **The Merge**，全面采用 **PoS**，能耗下降约 99.95%
发行机制	供应 **上限 2100 万枚 BTC**，通过约每 4 年一次的减半（Halving）逐步释放，预计 2140 年左右挖完	没有硬性供应上限，采用低通胀发行 +EIP-1559 手续费销毁机制，在高链上活动时期整体可呈现**净通缩**（“Ultra-sound money” 叙事）
出块 / 记账节奏	均值约 **10 分钟 / 块**，确认速度相对较慢	PoS 下以 **12 秒 Slot** 为节拍，大多数 Slot 会产出一个区块，一般几十秒到几分钟内就能得到较高置信度的确认
可扩展性与应用	脚本能力有限，链上逻辑以支付为主；更像“防篡改资产账本”	原生支持智能合约与 EVM，已经发展出 DeFi、NFT、DAO、游戏、公链 L2 等完整应用生态

代码 / 协议复杂度	协议相对精简，优先考虑简单与安全，可升级节奏慢	协议和执行环境更复杂，升级频繁（London、Merge、Dencun 等），既带来功能，也带来实现和安全上的挑战
交易费用与能耗	区块空间固定 + PoW 能源成本高，链上扩容有限，拥堵时手续费容易飙升	PoS 大幅降低能耗；EIP-1559 调整费用结构（Base Fee + 小费 + 手续费销毁），整体费用仍由供需决定，但体验更可预期
代币标准	只有原生 BTC，没有“官方标准化的智能合约代币协议”（资产扩展主要靠其它链或侧链）	在主网原生支持 ERC-20、ERC-721、ERC-1155 等标准，形成了可组合的代币与 NFT 生态系统
扩容方案	重点发展 Lightning Network 等支付通道型二层，适合小额高频支付；也有部分侧链方案	走 Rollup-**为中心的路线图**：L2 Rollups 负责执行，L1 负责结算与数据可用性；2024 年的 **Dencun 升级（EIP-4844）** 引入 Blob，显著降低 L2 数据成本，为未来 **Danksharding（数据分片）** 打基础
长期目的	成为全球性的价值储存与自由货币，“链上数字黄金”	成为 Web3 的基础设施：一个全球可编程结算层 + 数据可用性层，为各种 dApps、Rollups 和数字经济提供底层支持

3. 为什么比特币被称为“数字黄金”？

①供应有限，具备稀缺性

比特币总量上限为 **2100 万枚**，通过每约四年一次的减半逐步释放。这种绝对稀缺的发行机制，类似黄金这种自然稀缺资源。

②抗通胀与长期储值叙事

黄金长期被视为对冲通胀的避险资产；比特币因为供应上限固定，也被不少投资者当作对冲货币超发的工具——“央妈印多少，我就拿着不稀释的 2100 万份之一”。

③可验证且去中心化

黄金有实物价值，却难以在全球范围内验证真伪和来源；比特币依托区块链，所有交易公开可查，没有中心化发行人或清算机构，验证成本低很多。

④便利性与流动性更高

黄金的运输、托管、清算成本很高，而比特币是纯数字资产，可以在全球范围内几分钟到账、24 小时交易，更适应互联网时代的流动性需求。

4. 小结：比特币 vs 以太坊

- **比特币：**
  - 类比为“**数字黄金**”，追求的是**简单、稳健、安全**与长期价值储存；
  - 协议层故意做得很克制，不主动承担太多应用层功能。
- **以太坊：**
  - 更像是“**全球可编程的结算与计算平台**”，专注于为金融、艺术、游戏、治理等各种应用提供基础设施；
  - 协议本身在不断迭代（PoS、EIP-1559、Dencun 等），在扩展性和复杂性之间找平衡。

来句粗暴的话：

> 比特币更像是“价值锚”，
>
> 以太坊更像是“**应用发动机**”。

对于投资者来说，一个偏**稳定价值储存**，一个偏**成长与应用绑定**——

就此罢，这里我们先不展开理财话题。

从开发者视角看：**比特币是极简而可靠的“硬钱系统”，以太坊则是你可以在上面搭建整套 Web3 应用的“全球可编程区块链”**。

六、以太坊 dApps 的概念与应用

去中心化应用程序（dApp） 是构建在去中心化网络上的应用，它把两块东西拼在一起：

- **链上的智能合约（后端逻辑）**
- **链下的前端页面 / 客户端（用户界面）**

在以太坊上，智能合约就像一组**开放、透明、永远在线的 API**：任何人都可以调用，你的 dApp 甚至可以直接复用别人已经部署的合约逻辑（例如 Uniswap、Aave、各种 NFT 市场）。

- dApp 的**后端代码**（智能合约）运行在以太坊主网或其 Rollup（二层网络）之上的 EVM 中，由全网节点共同执行和记录；
- dApp 的**前端**可以用任意 Web/App 技术编写，既可以放在传统服务器上，也可以托管在 IPFS、Arweave 等去中心化存储上。

这和传统 App“后端跑在一台（几台）中心化服务器上”形成了鲜明对比。

1. dApp 的几个核心技术特征

这几条其实就是“为什么要用以太坊跑应用”的底层逻辑。

- **去中心化**

  dApps 在以太坊这种开放、公共的去中心化平台上运行：

  - 没有任何单一公司 / 机构可以控制整个网络；
  - 不存在一个“总机房”可以被直接关停或删库跑路。

- **确定性**

  同一份合约在任何节点上执行，给定同样的输入，**永远得到同样的输出**。

  这保证了：世界各地的人调用你的合约，看到的结果都是一致的。

- **图灵完备**

  以太坊虚拟机（EVM）支持图灵完备的智能合约语言（Solidity / Vyper 等），在资源足够（Gas 足够）的情况下，可以实现几乎任何可计算的逻辑，而不仅仅是“转账 + 简单脚本”。

- **隔离性（沙盒）**

  dApp 的执行被沙箱在 EVM 里：

  - 单个合约逻辑写挂了，只会让它自己的调用失败（回退），不会把整个以太坊网络“搞宕机”；
  - 全网节点只是共同重放并验证同一套字节码。

2. dApp 开发的好处

这些基本就是以太坊官方文档里列的优势，和你原文的一一对应。

①零停机时间

- 智能合约一旦部署，全网节点都会保存并执行它；
- 只要以太坊网络还在，你的后端逻辑就不会“被关机”或被运营商下架。

②隐私 / 匿名性（相对传统 Web）

- 与 dApp 交互，只需要一个地址和私钥，不需要实名注册、提交身份证明；
- 当然，链上是**透明伪匿名**而不是“完全隐身”，这点在隐私设计上需要自己权衡。

③抗审查

- 没有单一实体可以阻止你发送交易、部署 dApp 或读取链上的公开数据；
- 要“封杀”一个 dApp，必须同时影响大量节点和入口，现实难度极高。

④数据完整性

- 依托密码学和共识机制，链上数据一旦写入就几乎不可篡改；
- 恶意行为者无法事后伪造交易或随意修改历史状态。

⑤无需信任 / 可验证行为

- 智能合约代码公开可审计，执行路径可重放；
- 用户不需要“相信运营方不会作恶”，而是直接“相信代码 + 共识规则”。
- 这和传统网上银行那种“先相信银行，再查账”的模式本质不同。

3. dApp 能做些什么？

以太坊上的 dApps 已经覆盖非常多场景：

- **金融（DeFi）：**

  去中心化交易所（DEX）、借贷协议、稳定币、衍生品、收益聚合、保险等；

- **资产与内容（NFT & 游戏 / 内容）：**

  NFT 市场、链游、虚拟土地、音乐 / 视频版税分配；

- **组织与治理（DAO）：**

  链上投票、国库管理、协议治理、公共物品资助（例如 Gitcoin）；

- **基础设施类 dApp：**

  钱包、域名系统（ENS）、身份、预言机、中继器、跨链桥、L2 Rollups 本身等；

- **其他实验性应用：**

  去中心化社交、链上存证、供应链追踪、链上订阅与会员系统等等。

  你可以把 dApp 理解为：**“用智能合约做后端，用以太坊做数据库和结算层”的应用程序**。

4. dApp 开发的缺点 / 代价

这些问题截至 2025 年依然存在，只是有部分已经通过 Rollup 等方式缓解了——我在原有缺点的基础上补了一点现状。

4.1 维护困难

- 合约一旦部署，**链上代码默认不可修改**；
- 即使发现漏洞或需求变化，也只能通过：
  - 预先设计好的升级代理模式（proxy / UUPS）；
  - 或部署新合约 + 迁移数据 / 资产；
- 这就要求你在上线前就有更严谨的设计与审计。

4.2 性能开销与扩展难度

- L1 上，每个全节点都要执行所有交易，带来天然的性能上限；
- 目前以太坊 L1 **实际吞吐大约在 15–30 TPS 左右**，不会无限提升，否则会伤害去中心化；
- 扩容主要依靠 **Rollup-为中心的路线图 +Dencun（EIP-4844）等升级：**把绝大部分用户交易搬到 L2（Arbitrum、Optimism、Base 等），L2 TPS 已经可以达到每秒数千甚至上万，而 L1 更像结算与数据可用性层。

换句话说：

> dApp 想跑得快、跑得便宜，优先考虑部署在 L2，但最终安全性仍然锚定在以太坊 L1。

4.3 网络拥堵与 Gas 成本

- 当大量 dApp 在 L1 上抢区块空间时，Gas 价格会飙升，用户体验很差；

- 即便有 L2，部分高价值操作（大额清算、跨链桥结算等）仍会回到 L1，造成“高峰期很贵”；
- 对开发者而言，需要认真做两件事：
  - 合约设计 **gas 友好**（减少无谓存储 / 循环）；
  - 前端提示用户 Gas 情况，必要时支持在多条 L2 之间切换。

4.4 用户体验门槛高

- 普通用户需要：钱包、助记词 / 私钥管理、Gas 费用预估、L1/L2 概念……对多数人都不友好；
- 一旦操作失误（私钥泄露 / 转错地址），几乎无法追回；
- 于是很多项目不得不“在外面包一层 Web2 UX”，这又引出下一点。

4.5 “再中心化”倾向

- 为了降低门槛，有些“看起来是 dApp” 的产品，会：
  - 在服务端代管用户私钥（E-mail 一键登录那种）；
  - 把前端放在传统 CDN 上；
  - 在链下先算一堆关键业务逻辑，最后只把结果写到链上。
- 这样做可以大幅提升易用性，但也会重新引入**单点信任和审查风险**，消耗掉本来由区块链带来的很多优势。

七、以太坊的去中心化实现机制

1. 共识与出块（技术 / 经济层）

以太坊自 2022-09-15「The Merge」起全面采用 PoS 共识。任何人只要抵押 ≥32 ETH 就能成为验证者（validator），参与区块提议和投票。区块在每个 slot 中由随机选出的提议者（proposer）产出，同时有数百个验证者在同一时隙对链头

进行投票（attestation），通过 **Gasper** 共识（Casper-FFG 提供终局性 + LMD-GHOST 提供链头选择）完成安全性与活性。

如果验证者作恶（例如双签、提交无效区块），会被**削减（slashing）**，直接损失部分质押；若大规模长期离线，还会触发 **inactivity leak**，使这些离线验证者的有效质押被慢慢稀释，确保链最终仍能恢复终局性。

验证者数量一度在 2024 年初接近 100 万个，2025 年中虽然出现了约 10% 的主动退出、回落到 2024 年春天的水平，但整体仍然是“百万级”的大规模验证者集合，这在经济上进一步摊薄了单一参与者的影响力。你可以在 beaconcha.in 等区块浏览器的公开图表中看到长期走势与活跃验证者数据。

Pectra 升级之后要特别说明的一点是：

EIP-7251 并不是把「**最低质押门槛**」从 32 ETH 提高到 2048 ETH，而是把每个验证者的**有效余额上限**从 32 ETH 提升到了 2048 ETH——也就是说，**你依然可以用 32 ETH 开一个验证者，只是大户可以选择把更多质押合并到同一个验证者里赚取利息，不必拆成很多个 32 ETH 的小验证者。**

2. 节点与网络（技术层）

任何人都可以在家里、办公室或云端运行全节点或轻节点。以太坊执行层使用 DevP2P 协议维护点对点网络：

- **发现层（Discovery Stack）** 基于 UDP，用来发现和加入新节点；
- **DevP2P 通信层** 基于 TCP，负责在已连接节点间传递区块、交易等实际数据。

共识层方面，引入了面向轻客户端的 **同步委员会（Sync Committee）**：每 ~27 小时会随机挑选 512 个验证者组成一组，他们连续为区块头签名，让轻客户端

只通过验证少量签名就能跟随链头，而不需要下载和执行完整区块，从而让手机、浏览器插件甚至 IoT 设备都能以低成本参与。

这一整套设计，让「运行一个以太坊节点」不再是大机构的特权，而是任何技术爱好者都能参与的活动，从网络连接层面放大了去中心化。

3. 客户端多样性（实现层）

以太坊的一个很有意思的文化是 **“多客户端（multi-client）”**：

- **执行层客户端（EL）**：Geth、Nethermind、Besu、Erigon、Reth 等；
- **共识层客户端（CL）**：Lighthouse、Prysm、Teku、Nimbus、Lodestar 等。

不同团队在不同语言和代码库下实现同一份协议规格。社区还专门维护了 clientdiversity.org 这类站点，定期提示哪些客户端占比过高，提醒质押者「换个少数派客户端」。目标是避免「一个客户端一旦出 bug，全网一起躺平」的单点风险。

4. MEV 与 PBS（市场结构层）

最大可提取价值（MEV）天然会推高专业化、集中化的矿工 / 验证者，这与以太坊追求的去中心化存在张力。为缓解这一问题，以太坊引入了 **Proposer-Builder Separation（PBS，提议者-建构者分离）** 的思路。

目前主流的实现是协议外的 **MEV-Boost** 中继系统：

验证者运行 mev-boost，将区块排序与打包的工作交给一批竞争性的「区块构建者」市场，构建者相互竞价，把打包好的区块连同出价提交给验证者；验证者只需在 slot 内选择收益最高、有效的区块。这样一来：

- 「谁来排序交易」的权力被市场化、分散到多个构建者手里；
- 验证者保留了最终选择权；

- 至少在短期内，减轻了 MEV 直接把验证者推向高度专业化寡头的压力。

不过，MEV-Boost 依赖中继（relay），中继本身又带来了「单点故障、信任、审查和集中化」的额外讨论，围绕 **ePBS（Enshrined PBS，内生 PBS）** 的研究正是为此而生，希望在未来把 PBS 直接写进协议，弱化对外部中继的依赖。

5.「Rollup-中心」路线与数据去中心化（扩容 / 职责分离）

Vitalik 在 2020 年提出了「**Rollup-centric Ethereum Roadmap**」：L1 主要做高度去中心化的 **结算层 + 数据可用性层**，把大规模执行吞吐交给各种 L2 Rollup。

2024-03-13 的 **Dencun 升级** 通过 EIP-4844 引入了 blobs（临时数据块），为 Rollup 提供了专用的、廉价的数据存储通道，大幅降低 L2 发布数据的成本，让更多团队可以以较低门槛运营自己的 L2。

在这条路线下：

- L1 追求的是「更小、更简单、更去中心化」；
- L2 追求的是执行扩容和多样化的应用场景；
- 未来像 Danksharding、PeerDAS 等方案则继续提升数据可用性，而不强迫每个节点处理所有执行细节。

这种「职责分离」本身，也是另一种维度上的去中心化：执行负载与创新被推向众多 L2，而安全和数据共识由一个更小、更稳的 L1 来保证。

6. 治理（社会层）

以太坊**没有「1 票 1 COIN」式的链上代币治理**。核心协议演进主要通过：

- 开放的 EIP 流程（任何人都能提交改动提案）；
- 定期的 AllCoreDevs / ACDE 等开发者会议；

- 开发者、节点运营者、应用和用户之间的公开讨论（Ethereum Magicians 论坛、Github、Discord、社区电话会议等）。

最终是否升级，由 **客户端开发团队实现 + 节点运营者选择是否升级软件** 共同决定。这是一套彻底的 **“粗共识 + 退出权”** 模式：如果某个提案在社区争议巨大，往往很难进入主网；即便上了主网，节点也保留不升级、或者分叉出去的权利。

7. 多个机制叠加出来的「去中心化」

综合来看，以太坊的去中心化并不是单一机制的结果，而是多层叠加的产物：

- **共识层**：PoS + Gasper 把出块与投票权分散给百万级别的验证者，slashing 和 inactivity leak 提供了强烈的经济约束。
- **网络层**：全球成千上万的 P2P 节点彼此直连，没有中心服务器；全节点、轻节点和质押节点共同维持网络。
- **实现层**：多客户端、多语言、多团队实现同一协议，主动追求客户端占比的健康分布，降低实现层单点风险。
- **扩容路径**：通过 Rollup-中心路线，把执行和创新推向大量 L2，L1 聚焦于数据和共识的高去中心化、安全性。
- **治理层**：没有形式化的链上投票，靠公开讨论和社会共识推动协议演进，避免简单的「代币持有量 = 治理权」。

任何持有至少 32 ETH 的人，都可以通过质押成为验证者，直接参与到网络共识的一部分；**Pectra 之后，只是把单个验证者的「收益上限」提升到了 2048 ETH，方便大额质押合并，并没有抬高普通人的进入门槛。**

此外，以太坊核心协议完全开源，任何人都可提交 EIP，通过社区共识实现升级；节点与验证者分布全球，交易和智能合约的数据公开透明，消除了对中心

化权限的依赖，从而构建出一个 **无需许可、强抗审查、高透明度的分布式计算平台**。

8. 做个补充：什么是 P2P 网络？

上面多次提到「P2P」，可以简单把它想象成一群“好友”节点互相连线聊天、转发消息，而不是所有人都连到一个中心服务器：

- 当你启动一个以太坊节点时，它会先通过预设的**引导节点**认识一小撮「老朋友」，然后再通过它们慢慢认识全网更多节点；
- **发现协议**跑在 UDP 上，用来互相「打招呼 / 探测」（ping/pong），确认对方在线；
- 一旦建立连接，就会在 TCP 上开启加密的 DevP2P 会话，用来同步区块、交易等真实数据。

在这种网络里，每个节点既是「读者」又是「广播者」：它既可以向别人转发收到的交易 / 区块，也可以把自己看到的新交易广播出去。就像一场没有主持人的群聊，只要还有足够多的人在线、互相转发，这个网络就很难被关闭或控制——这正是以太坊去中心化的基础之一。

八、网络结构的开放性与参与机制

从设计理念上说，以太坊的网络结构是 **无限制参与** 的——

任何人都可以运行节点、发送交易、部署合约，网络本身不设“准入门槛”或白名单，这就是所谓的 **permissionless（无需许可）**。

但大家也都知道：**有“理论”就会有“现实”**，现实世界里总会有物理和经济约束。

1. 节点越来越多，会带来什么问题？

以太坊的去中心化，依赖大量节点一起维护区块链的完整性和安全性。

典型全节点需要：

- 存储完整链状态（几百 GB 到数 TB 级别，取决于客户端和是否裁剪）
- 持续下载新区块、交易，参与验证和广播
- 维持一个**持续在线、带宽稳定**的网络连接

随着网络规模和历史数据增长，节点的**存储、带宽和计算压力**都会上升，这就形成了现实中的门槛：不是任何一台旧笔记本都能轻松跑满功能全节点。

1.1 存储和带宽压力

现在（截止到 2025 年），常见推荐配置大概是：

- 全节点：4 核 CPU、16 GB 内存、1 TB NVMe SSD、至少 20–25 Mbps 稳定带宽，且尽量不要有流量封顶；
- 归档节点：动辄 4–10 TB SSD、64 GB 内存以上，只适合专业服务商或基础设施项目来跑。

> 换句话说：理论上谁都能参加，实际上还是要有一定的硬件和网络条件。

1.2 补充说明：什么是“带宽”？

带宽：单位时间内网络能传多少数据，一般用 bps（bit per second）表示。带宽越大，说明“管子越粗”，同一时间能跑更多数据。

我们用水管来做个类比：

- **水管的宽度 = 带宽**：决定单位时间内最多能通过多少“水”（数据）；
- **水流量 = 实际传输的数据量**；
- **水流速度 = 延迟**：水从这头到那头需要多久。

以太坊节点要不断收发区块、交易，如果带宽太小，就会**同步变慢、落后链头，甚至掉线**。这也是为什么官方和各客户端会给出一个推荐的最低带宽（比如20–50 Mbps）——这不是“必须”，但越低、体验越差。

2. 节点多了，共识会不会变慢？——共识效率

节点数量变多后，要达成“哪条链是真的”“哪个区块被接受”，理论上需要在更多参与者之间传播信息，这会影响**共识效率**。

> 共识效率：衡量区块链网络在“大家对同一状态达成一致”时的时间和资源成本。

常见几个指标可以记住：

2.1 吞吐量（Throughput）

- 单位时间内能处理多少交易（TPS）。
- 对以太坊来说，现在 L1 的 TPS 仍然在几十级别，但加上 L2 后，整个生态的 TPS 已经能跑到上万甚至更多。

2.2 延迟（Latency）

- 从你发起一笔交易，到它被打包并获得较高确认的时间。
- PoS + 现代客户端优化后，一般几十秒内就能达到比较可靠的确认，再加上后续几个 epoch 的“终局性（finality）”。

2.3 资源消耗

- 节点为了参与共识要付出的计算、存储、网络消耗。
- 这决定了“普通人跑节点的门槛高不高”，也直接关系到去中心化程度。

2.4 容错能力

- 系统可以容忍多少节点故障或作恶还不崩盘。

- 以太坊 PoS 在设计时同时考虑了拜占庭容错、安全系数和惩罚机制（slashing/inactivity leak），用经济激励约束验证者行为。

所以，“节点越多越好”不完全对。

太少会中心化，太多又可能拖慢同步和共识，需要架构上的平衡和优化。

3. 那以太坊是怎么“又要人多、又要跑得快”的？

一句话：**把不同的工作分层做**。

这就是后面会详细提到的 **分片（sharding）和 Layer 2（L2）扩容方案** 背后的思路——只是现在的官方路线已经演变成更明确的 **“Rollup 为中心 + 数据分片（Danksharding/PeerDAS）”**。

3.1 Layer 1 是什么？——主链 / 基础层

定义：

Layer 1 就是**区块链本身**，也就是以太坊主网这一层：

- 负责共识、安全、最终结算和数据存储；
- 交易和智能合约在这里获得“最终记账权”；
- 典型如：比特币、以太坊、Solana 都是各自生态的 L1。

职责与特点：

- 有自己的共识机制（以太坊现在是 PoS），保证安全和去中心化；
- 协议升级难、节奏慢，但一旦改动就是“全局生效”；
- **追求的是：安全 + 去中心化优先，哪怕牺牲一部分吞吐量**。

3.2 Layer 2 是什么？——盖在上面的“加速层”

定义：

Layer 2 是构建在以太坊这种 Layer 1 之上的**扩展层 / 第二层网络**，目标是在不牺牲 L1 安全的前提下，让交易**更快、更便宜**。

可以理解为：

> L1 是“结算所 + 大法官”，L2 是高铁 / 支线，把大量小额高频的交互搬到上面处理，最后再把结果汇总回 L1。

常见 L2 类型：

- **Rollups（汇总链）**
  - 把大量交易打包，然后把压缩后的数据和证明提交到 L1；
  - 分为：
    - Optimistic Rollup（乐观式）：先假定交易有效，有争议再提交欺诈证明；
    - ZK Rollup（零知识）：用零知识证明直接向 L1 证明一大批交易是正确的。
- **状态通道（State Channels）**
  - 双方把一部分状态锁进合约里，然后**在链下反复互相更新**，最后只把最终结果结算回链上；适合高频、双边交互。
- **侧链（Sidechains）**
  - 独立链，有自己的共识和安全假设，通过跨链桥与以太坊互通，例如早期的 Polygon PoS 等；
  - 更像是“兼容以太坊的另一条链”，而不是严格意义上的 L2。

优势与权衡：

- 优点：
  - 大幅提升 TPS 和用户体验（更快、更便宜）；
  - 让更多应用可以迁移到 L2，而 L1 负责最终安全和结算。
- 权衡：
  - 不同 L2 的安全模型各不相同（尤其是侧链）；
  - 跨链桥、运营方、排序器等可能引入新的信任假设和攻击面。

4. 总结一句话：以太坊到底能不能“无限制参与”？

- **理念上**：任何人都可以运行节点、发送交易、部署合约，协议层不会问你“谁批准的”；
- **现实中**：
  - 跑全节点需要一定硬件和带宽；
  - 共识协议要在“节点越多越难协调”和“尽量开放参与”之间找平衡；
  - 因此出现了 Rollup、Proto-Danksharding / Danksharding、PeerDAS 等一系列扩容方案，专门用来**在不牺牲去中心化的前提下，把性能拉上去**。

给一个可以背的句子（面试的时候可以直接使用）：

> 以太坊的网络结构 = 理论上对任何人开放（permissionless），现实中受限于硬件和带宽，但通过 Layer 2 + 数据分片等方案，把“更多人参与”和“系统还能跑得动”两件事尽量同时做到。

九、行业应用案例：金融、游戏与社交

以一句话做开端：

> 以太坊比较成熟的赛道：金融（DeFi / 稳定币 / RWA）、NFT & 游戏、社交 & 内容 & DAO。 很多应用已经迁到各类 L2，但结算和安全大多仍依托以太坊生态。

1. 去中心化金融（DeFi、稳定币、RWA）

这是以太坊最成熟、体量最大的应用方向之一。

代表性协议：

- **Uniswap（DEX / 自动做市 AMM）**

最大的去中心化交易所之一，用户用钱包就能直接互换代币，无需开户、无需托管资产。Uniswap 通过 AMM 池子撮合交易，目前已经支持以太坊及多条 L2 / 侧链。

- **Aave（借贷协议）**

去中心化、非托管的借贷市场，用户存入资产赚利息，或超额抵押后借出其他资产。利率由市场供需自动调节，是以太坊上 TVL 常年排前列的借贷协议之一。

- **MakerDAO / Sky + DAI 稳定币**

MakerDAO 通过抵押 ETH 等资产发行去中心化稳定币 DAI，DAI 广泛用作 DeFi 里的计价单位和抵押物；协议治理逐步升级为 Sky 生态后，依然是以太坊上历史最久也最重要的 DeFi 组件之一。

- **Lido、EigenLayer 等质押 / 再质押协议**

Lido 提供 ETH 流动性质押（stETH），让质押者在参与 PoS 安全的同时保留流动性；EigenLayer 在此基础上引入“再质押（restaking）”，允许把 stETH 等 LST 再用于其他协议的安全，为以太坊衍生出一个新的收益与安全层。

- **RWA（Real-World Assets，现实世界资产上链）**

2025 年，国债、债基等现实资产的代币化规模已达到数十亿美元级，其中相当一部分发行在以太坊或其 L2 上；像 BlackRock 这类传统金融机构也在发行链上美元货币基金、国债产品。

从整体数据看，**以太坊仍然是 DeFi 生态的主阵地**：有报告估算 2025 年 Q4 以太坊上的 DeFi TVL 约在数千亿美元量级，在所有公链中保持领先，增长动力主要来自稳定币、LST/LRT（质押衍生品）和借贷协议。

2. 游戏、虚拟世界与 NFT

NFT 赛道最早就是在以太坊上被玩“爆”的。

早期标志性项目：

- **CryptoKitties（加密猫）**

2017 年的虚拟宠物养成游戏，每只猫都是一个 NFT，可以买卖、繁殖。当年热度太高，一度占据全网 20% 左右的交易量，把以太坊主网堵成“猫猫链”，也顺便暴露了早期扩容问题。

- **Decentraland（虚拟土地 / 元宇宙）**

用户可以买 LAND 土地、建房子、办展会、开演唱会，是典型的“链上虚拟世界”。地块、道具、票券等都通过以太坊智能合约来确权和交易。

更“现代”的游戏与虚拟世界：

- **The Sandbox（开放世界 / 创作者经济）**

以太坊上的 UGC 元宇宙平台，可以在 LAND 上用工具做关卡、做 IP 联动；目前正准备上线自己的以太坊 L2——**SANDChain**，专门服务创作者经济，让游戏内大部分交互迁到低费、高 TPS 的二层上执行。

- **Axie Infinity、Illuvium、The Sandbox 等链游生态**

很多链游虽然现在跑在独立侧链或专用 L2（如 Ronin、即将上线的 SANDChain），但最初的资产标准、玩家钱包和一部分基础设施依旧依托以太坊标准和工具。

NFT & 市场：

- **OpenSea、Blur、Magic Eden 等 NFT 市场**

这些市场大多以以太坊为主阵地，支持 ERC-721 / ERC-1155 标准 NFT 的铸造与交易，覆盖艺术作品、PFP 头像、游戏道具、门票等各类资产，其中 OpenSea 长期是以太坊 NFT 交易量的重要一极。

3. 社交、内容创作与 DAO / 公共物品

这一块这两年发展得非常快，基本都是“**协议 + L2**”的形态，但底层还是与以太坊账户体系和资产标准深度绑定。

社交协议与应用：

- **Lens Protocol / Lens Chain**

去中心化社交图谱协议，最初部署在 Polygon 上，2025 年上线了自己的 L2——Lens Chain（基于 zkSync + Avail DA），专门为社交场景做低费率、高可组合性的链上社交组件（关系、feed、groups 等）。

- **Farcaster**

一个被 Vitalik 点名多次关注的去中心化社交协议，目前运行在以太坊 L2 Base 上。2024 年完成 1.5 亿美元融资，引入 Frames 等创新交互后，日活在短时间内从几千飙升到数万，并计划进一步推出自己的高 TPS L2 承载社交流量。

- **friend.tech 等 SocialFi 应用**

运行在以太坊 L2（Base）上的“键盘 / key 社交”应用，把创作者的社交关系代币化，通过买卖 key 进入私密群聊等，探索“注意力可交易”的新模式。

内容创作与写作平台：

- **Mirror.xyz**

典型的 Web3 写作与发布平台：用以太坊账号登录，文章数据存 Arweave，NFT 收藏、众筹等逻辑通过智能合约实现。它既是“链上博客”，又是创作者发行 NFT、进行众筹和会员制的工具箱。

- 此外，早期还有一些基于以太坊或其侧链的去中心化写作 / 讨论平台，虽然未必像 Lens、Farcaster 那样声量大，但为“链上内容生态”做了不少探索。

DAO 与公共物品资助：

- **协议治理 DAO：**

Uniswap DAO、Aave DAO、MakerDAO 等都属于“协议 DAO”，负责参数调整、金库管理和协议升级投票。

- **Gitcoin Grants 等公共物品资助平台**

Gitcoin 通过以太坊上的二次方资助（Quadratic Funding），动员小额捐赠 + 配捐资金，长期为开源软件、公共基础设施和社区项目提供资金支持。

十、生态系统创新：DeFi、NFT 与 DAO

可以把以太坊生态想成三块：**“金融操作系统（DeFi） + 数字资产与身份层（NFT） + 组织与治理层（DAO）**。这三层都不是简单“搬到链上”，而是玩出了全新的 **机制设计**。

1. 去中心化金融（DeFi）：重塑全球金融“操作系统”

以太坊最成熟、体量最大的一块就是 DeFi。这里的创新不只是“把银行放到链上”，而是重新设计了**做市、借贷、稳定币和扩展能力**。

1.1 自动做市商（AMM）& 去中心化交易所（DEX）

- 代表协议：**Uniswap、SushiSwap** 等。
- 创新点：用**自动做市商 AMM** 取代订单簿，任何人都能往池子里存两种资产，按公式（如 $x \cdot y = k$）自动报价和撮合，无需做市商、无托管。
- 效果：

- **24/7 无许可交易**：只要有钱包和网络就能直接 swap。
- **流动性提供者（LP）变成“赚手续费的角色”**，不是传统意义的“做市机构”。

近几年 Uniswap 迭代到 v3 / v4，引入**集中流动性、钩子 hooks 等机制**，被不少研究直接视作 DeFi 中“最核心的 AMM 创新样板”。

1.2 去中心化借贷：链上抵押 + 程序化利率

- 代表协议：**Aave、Compound**。
- 模式：用户把资产存进资金池，获得利息；需要借钱的人超额抵押借出，利率由池子里资金利用率自动调整。
- 创新：
  - **全程非托管（non-custodial）**：协议不“保管你的钱”，都是放在合约里按规则运行。
  - **超额抵押 + 清算机制**：完全由代码执行，避免了人工黑箱。

Aave 还发了自己的去中心化稳定币 **GHO**，同样走超额抵押路线，让“借贷 + 稳定币”形成一体化金融乐高。

1.3 稳定币与合成资产：链上“计价基准”

- 代表：**DAI**（MakerDAO/Sky）、以及在以太坊上广泛使用的 USDC、USDT 等。
- DAI 的创新：
  - 通过**超额抵押 + 清算机制**，保持与 1 美元挂钩；
  - 完全运行在智能合约里，由 MakerDAO 治理，被视为最早成功落地的去中心化稳定币之一。

稳定币给 DeFi 提供了**相对稳定的计价单位**，是借贷、交易、衍生品协议的基础积木。

1.4 跨链 / 跨 Rollup 与 Layer2 扩展

- Rollup 是现在主流的扩容路线：
  - **Optimistic Rollup**（如 Optimism、Base、Arbitrum）：默认交易都正确，只有在有人质疑时才通过“欺诈证明（fraud proof）”回滚，就像老师默认作业都没抄袭，除非有人举报才会复查。
  - **ZK Rollup**（如 zkSync、Starknet、Scroll）：每批交易都带一份**零知识证明**，主链只需要验证“证明正确不正确”，就像超市收银员只看你给的“付款有效证明”，不去一笔笔重算。

> 类比总结：
>
> - **OP Rollup**：*默认你作业写对了，只有被质疑时才复查*；
> - **ZK Rollup**：*一开始就随作业附上一份“证明我写对了”的数学证书*，老师只验证明。

- 这些 Rollup **把大量交易搬到 L2 处理**，只把压缩后的数据和证明提交到以太坊 L1，从而：
  - 大幅降低 gas 成本、提高 TPS；
  - 仍然复用以太坊的安全性。
- **跨链 / 跨 Rollup 互操作性：**
  - 以太坊上出现了各种跨链桥、消息传递协议，支持在不同 L2、侧链和其他 L1 之间转资产 / 传消息。
  - 这让 DeFi 从“单链应用”走向了**多链 / 多 Rollup 流动性网络**，但也带来桥的安全风险，这是现在重点研究的方向之一。

2. 非同质化代币（NFT）：从“所有权证明”到“链上身份与权限系统”

NFT 不是简单的“链上 JPG”，而是把“**唯一性 + 所有权 + 可编程性**”组合在一起，形成一层新的资产与身份基础设施。

2.1 基础标准：ERC-721 / ERC-1155

- **ERC-721**：最经典的 NFT 标准，每个 token 都有唯一 ID，对应一个独一无二的资产（比如一件画、一只猫、一块地）。
- **ERC-1155**：混合标准，可以在同一合约中同时管理同质化 & 非同质化资产，适合游戏里批量铸造道具、门票等（例如“1000 张同一场演唱会门票 + 若干 VIP 特别票”）。

2.2 数字艺术、收藏品与品牌 NFT

- 艺术家可以直接发行 NFT 作品，全球用户用钱包就能购买、收藏、二级市场转卖。
- 奢侈品牌、时尚品牌也在用 NFT 做“**数字藏品 + 实物联动 + 会员体系**”：
  - 例如 **Maison Margiela** 的 Numbers 数字藏品和 MetaTABI 系列，会给持有者额外权益（如提前购入机会、限定商品、Web3 游戏互动等），并在以太坊主网和专业平台上发行。

这类“**Phygital（物理 + 数字）**”尝试，把 NFT 变成了实物商品的数字通行证与会员卡。

2.3 NFT 的“钱包化”：ERC-6551 Token-Bound Accounts

传统 NFT 只是静态资产，现在一个重要创新是：**给 NFT 一个自己的账户**。

- **ERC-6551（Token-Bound Accounts）**
  - 为每个 NFT 绑定一个智能合约账户，让 NFT 本身可以：
    - 持有其他代币或 NFT；
    - 主动与合约交互（比如签到、挂单、装备道具）；
    - 记录它自己的行为与“人生轨迹”。

想象游戏里的角色 NFT：

> 以前它只是“图片 + 属性”，现在它变成一个能自己**装装备、带道具、累积资产**的小钱包，整个角色的成长历史都写在链上。

2.4 NFT 角色管理：ERC-7432

ERC-7432，是近两年非常有意思的一个 NFT 扩展标准：

- 它给 NFT 引入了**角色（role）系统**：
  - 持有人可以给某个地址授予与该 NFT 关联的特定角色；
  - 角色会自动在某个时间戳过期，并可以携带自定义数据（如“可以帮我领取空投一次”“在活动期间拥有管理员权限”等）。

这就把 NFT 从“纯所有权”扩展成一套**可编程权限系统**，适合：

- 游戏里的装备授权、租赁；
- 虚拟土地的使用权、管理权分离；
- DAO 成员的细粒度权限管理等。

3. 去中心化自治组织（DAO）：把“组织结构图”写进代码

DAO 是以太坊推动出来的一种新型组织形式：**以智能合约为规则，以代币/NFT 为成员凭证，以链上投票为决策方式**。

3.1 协议 DAO：管理 DeFi / 基础设施协议

- 像 **Aave、Uniswap、MakerDAO/Sky** 等大型 DeFi 协议，都有自己的 DAO 负责：
  - 上线/下线资产、调整风险参数；
  - 管理金库资金、决定激励计划；
  - 协议升级与新功能上线。

这些 DAO 用治理代币（如 AAVE、UNI、MKR）绑定了**协议价值与治理权**，形成了一种新的“**协议即组织**”形态。

3.2 公共物品与资助 DAO：Gitcoin 等

- **Gitcoin Grants** 通过二次方资助（Quadratic Funding）为开源项目、基础设施与公共物品筹资：
  - 小额捐赠被放大权重，鼓励“广泛参与”；
  - DAO 负责管理配捐资金和规则。

这类机制已经成为政治经济学和机制设计里的研究对象：**“如何用链上激励机制资助公共物品”**。

3.3 创意型 DAO：Nouns 等

- **Nouns DAO** 通过每天拍卖一个 NFT，让拍卖所得直接进入 DAO 金库，再由持有者投票决定怎么花钱（赞助活动、做衍生产品、支持公共物品等）。
- 这种模式把“**IP 创作 + 社区治理 + 金库管理**”统一在一起，是 NFT + DAO 的一个代表性创新。

3.4 DAO 的应用领域

除了协议治理和公共物品，DAO 也在被用在：

- 慈善和捐赠（UkraineDAO 等用来筹集人道援助资金）；
- 艺术收藏 DAO（集体买 NFT / 艺术品）；
- 科研资助 DAO、媒体 DAO、城市/社区 DAO 等等。

十一、社区与开发生态扮演的角色

先给出一句话作为总览：

社区=“大脑 + 声音”，负责共识、价值观、方向感；

> **开发生态=“手和工具”**，负责把这些想法变成真正跑在主网和 L2 上的代码与应用。

下面还是分成「社区之中」和「开发生态系统」两部分来看。

1. 社区之中

1.1 去中心化治理与共识形成

以太坊的治理刻意避免“一个项目方说了算”。协议层升级主要通过两类公开渠道推动：

- **核心开发者会议（AllCoreDevs，含 ACD/ACDC 等电话会）**
  - 是目前以太坊协议层治理里**最核心、最公开的技术决策场合**：客户端团队、研究人员等会在这里讨论下一次升级（Merge、Dencun、Pectra 等）的范围、时间表和实现细节。
  - 这些会议面向所有人直播、记录，会后整理成公开笔记，任何人都能跟进讨论进展。

- **以太坊魔术师（Fellowship of Ethereum Magicians，FEM）**
  - 是一个开放论坛，专门用于讨论 EIP、**技术标准和协议细节**；任何人都可以发帖、跟帖，对提案提出质疑或补充。
  - 很多重要 EIP 在进入 AllCoreDevs 之前，都会先在 FEM 上经过长时间的社区讨论与打磨。

配合 GitHub 上的 EIP 流程、客户端仓库 issue、研究员博客等，形成了一个“多中心、强透明”的治理结构：技术方向不是某个公司拍脑袋决定，而是在一系列公开场合里缓慢收敛成为共识。

1.2 多元化：谁都能参与、谁都能发声

以太坊社区的参与者来自全球，身份非常多元：

- 核心和外围开发者、研究员、节点运营者；
- 设计师、艺术家、律师、会计、产品经理、市场运营；
- 普通用户、投资者、DAO 成员、黑客松选手……

大家通过不同方式参与生态建设：

- 撰写和评审 **EIP / ERC 标准**；
- 在 Devcon、Devconnect、各地的 Ethereum 社区会议上分享经验；
- 参加 **ETHGlobal 等黑客松**，在 36 小时里做出一个 MVP，很多知名项目（例如一些 DeFi、NFT 协议）都是从这里起步；
- 在本地社区组织 meetup、读书会、工作坊，帮助新人进入 Web3 世界。

这种“任何角色都能贡献”的多元性，让以太坊在设计协议时，能听到来自钱包、L2、DeFi、NFT、游戏、监管等不同视角的真实需求。

1.3 文化与价值观：开源、透明、“无限花园”

以太坊社区有一套很强的共同价值观：

- **开源与透明：**
  - 客户端、工具、协议实现大多开源，任何人可以 fork、审计、批评。
  - 治理讨论尽量在公开频道进行，而不是“闭门决策再公布结果”。
- **去中心化与“管理而非控制”：**
  - 以太坊基金会（EF）一再强调：自己的角色是“管理生态、填补空白，而不是控制一切”，EF 的责任是在多层生态中寻找缺口、支持他人，而不是把以太坊变成某个公司的产品。
- **协作优先，而非零和竞争：**
  - 像 Raid Guild 这样的开发者 DAO，会组织设计师、开发者、产品等以“公会”形式接项目，互相协作，把工具、教程、最佳实践回馈给整个生态，而不是只为单一公司打工。

这套文化，让以太坊更像一座“**无限花园（Infinite Garden）**”：EF 和早期团队只是园丁，真正让花园长成什么样的，是全球无数贡献者的长期浇水和修剪。

2. 开发生态系统

如果说“社区”负责共识与价值观，那“开发生态”就是把这些东西落到**代码和工具链**上的那只手。

2.1 工具与基础设施：把开发门槛打下来

截止到 2025 年，以太坊已经有一整套非常成熟的开发工具和基础设施：

- **开发框架与 IDE**
  - **Remix IDE**：浏览器里就能写、编译、部署合约，非常适合新手和教学场景；

- **Hardhat、Truffle、Foundry**：用来写测试、做本地链模拟、脚本化部署，已经成为 Solidity 开发的“三件套”。

- **节点与基础设施服务**
  - 像 **Infura、Alchemy、QuickNode** 这样的基础设施提供商，负责维护高可用的节点和 RPC 服务，让 dApp 不必自己运维节点。

- **脚手架与前端集成工具**
  - Scaffold-ETH、Wagmi、第三方 SDK 等，帮开发者快速搭建前端、集成钱包和合约调用；
  - 这些开源工具大多来自社区公司或个人贡献，在 GitHub 上持续演进。

这些工具本身也是社区项目：有的由公司维护，有的由 DAO 或个人维护，形成了一个**高度模块化、可组合的“开发者堆栈”**。

2.2 教育与培训：把新人源源不断“拉进来”

以太坊的发展离不开“持续 onboarding 新人”的能力，这一块几乎完全由社区驱动：

- **官方与社区文档 / 教程**
  - ethereum.org 上的开发者文档、教程、示例工程不断更新，覆盖从 Solidity 入门到 L2 开发、Rollup、MEV 基础等主题。

- **Devcon / Devconnect / 本地黑客松**
  - Devcon 是 EF 组织的旗舰开发者大会，每一届都像是全球以太坊人的“大团建 + 大课堂”；
  - Devconnect、各地社区活动则更偏“工作周”和专题 workshop，深入讨论某条链、某个协议、某类技术。

- **ETHGlobal 等黑客松平台**
  - ETHGlobal 每年组织多场全球黑客松，官方宣传中提到一年可以催生上千个新项目，不少最终变成真正的产品和协议；
  - 对新人来说，这是学习、获得导师指导、接触投资人和伙伴的绝佳入口。

可以说，**开发生态一大部分工作，其实是在“教学 + 搭路 + 给梯子”**。

2.3 生态扩展与协议创新：把想法变成升级和产品

最后，是“实打实推动以太坊前进”的那部分：

- **生态应用层**
  - DeFi、NFT、DAO、Layer2、RWA（现实资产上链）等赛道的大部分主流协议，都是独立团队或 DAO 构建后再与主网深度结合的；
  - 这些一线开发者会把自己的需求和经验反向反馈到 EIP 讨论、客户端开发中。
- **协议层研发和升级**
  - Merge、Dencun (EIP-4844)、Pectra 等升级，都是由多家客户端团队、研究团队在公开论坛、ACD 会议、测试网中反复迭代出来的成果；
  - 像 PBS（Proposer-Builder Separation）、Verkle 树、Danksharding /PeerDAS 等路线，也都是研究员 + 客户端实现者 + 应用开发者长期博弈、折中之后形成的“技术与现实平衡点”。

> 我们可以这样理解：
>
> **社区**提出问题、表达需求、讨论路线；
>
> **开发生态**给出具体实现、工具和应用；
>
> 两者一起，把“论文里的以太坊”一步步变成“链上真实运行的以太坊”。

最后来做一个总结

> 以太坊之所以能在十多年里不断演化，不只是因为有一套聪明的协议设计，更因为它有一个开放、多元、有价值观约束的社区，和一整套 自下而上的开发者生态：社区负责方向和共识，开发者负责把这些共识写成代码，基金会和各类组织则在中间“管理而非控制”，尽量让这座“无限花园”自己长出来。

以太坊如新时代的火种，以代码点亮人类协作的未来。它让价值在网络中自由流动，让信任在无形中生根发芽。在以太坊的世界里，规则由代码书写，创新由无数双手织就。

——@kelvin-秦默

第二章

网络结构与节点类型

本章目标：了解以太坊节点是如何构成网络的。

一、以太坊节点与客户端软件

先来一句官方的定义：

> 节点（node）：任何一台运行以太坊客户端软件并连接到其他节点的电脑，就是一个以太坊节点。
>
> **客户端（client）**：是对以太坊协议的具体实现，会按协议规则验证数据、同步区块和状态，帮助网络保持安全。

自 The Merge 之后，一个“完整的以太坊节点”不再是单一程序，而是由两个核心客户端组成：

- **执行客户端（Execution Client，EL）**
- **共识客户端（Consensus Client，CL）**

如果还想参与出块、赚质押收益，还需要再挂上一个：

- **验证者客户端（Validator）**——作为共识客户端的“插件”。

1. 执行客户端（Execution Client）——“干活的”

执行客户端（也叫执行层客户端、EL 客户端、旧称 Eth1 客户端），主要负责链上的“**业务逻辑和状态**”：

- 监听网络上广播来的**交易与区块**
- 在 **EVM（以太坊虚拟机）** 中执行交易和智能合约
- 维护当前链的**状态数据库**（账户余额、合约存储等）
- 提供 JSON-RPC 接口（`eth_*`），给 dApp / 钱包 / 脚本调用
- 通过自己的 P2P 网络转发交易和区块头 / 区块体

简单类比一下：

> 执行客户端就像法院里的书记员 + 案卷系统，负责接收材料、执行具体流程、更新“案卷状态”。

常见的执行客户端包括：

- **Geth（go-ethereum）**
- **Nethermind**
- **Besu**
- **Erigon**
- **Reth**

2. 共识客户端（Consensus Client）——“法官”

共识客户端（也叫信标链客户端、CL 客户端、旧称 Eth2 客户端），主要负责**权益证明 PoS 共识**：

- 维护信标链（Beacon Chain）视图和 fork choice 规则（LMD-GHOST 等）
- 在每个 slot 里选择谁来提议区块，并对区块进行投票（attestation）
- 跟踪验证者集合、质押余额、惩罚（slashing）等共识状态
- 把来自执行客户端的结果当作“材料”，决定哪一个区块被接受到链上

类比延续刚才的法院场景：

> 共识客户端就像法官，根据书记员提供的案卷（执行结果），做出“这是不是合法区块、最终选哪条链”的裁决。

常见的共识客户端包括：

- **Lighthouse**
- **Prysm**
- **Teku**
- **Nimbus**
- **Lodestar**

3. 验证者客户端（Validator）——“陪审团代表”

在 PoS 模式下，如果你想质押 ETH、参与出块并获取奖励，就需要在共识客户端基础上再运行一个 **验证者客户端**：

- 它负责：
  - 管理你的验证者密钥
  - 在轮到你时**提议新区块**
  - 在每个 slot 为看到的区块做 **attestation（投票）**
- 从实现角度看，很多共识客户端本身就自带 validator 子进程，也有项目把它拆成单独进程。

继续类比：

> 验证者就像陪审团代表，按规则投票并在应该发言时站出来，拿质押的 ETH 当“保证金”。

4. 节点内部是怎么“搭积木”的？

可以想象一个合并后的节点结构示意图：

- **你的应用 / dApp / 脚本**
  - 使用 Web3 库（如 web3.js、ethers.js，或 Rust/Go/Java SDK）
  - 通过 JSON-RPC 调用执行客户端的 `eth_*` 接口（读状态、发交易等）
- **以太坊节点（Ethereum Node）**
  - 里面跑着一套 **执行客户端（EL）**
  - 再配一套 **共识客户端（CL）**
  - 想当验证者的话，再加上 **validator**

- **P2P 网络（p2p）**
  - 执行客户端和共识客户端各自都有 P2P 网络，用来和其他节点同步区块、交易和共识消息。

有些资料会把“对共识层的访问接口”叫成类似 `web3.beacon` 的概念，其实在现实里：

- 执行层仍然主要通过 **JSON-RPC** 提供 `eth_*` 接口；
- 共识层更多使用 **REST / HTTP / gRPC** 接口，规范定义在共识层规范中。

5. Engine API：执行层和共识层之间的“专线电话”

合并之后，执行客户端和共识客户端不再是一个大程序内部的模块，而是**两个独立进程**，它们之间必须通过一个标准化接口通信，这就是：

> Engine API —— 一个专门给 EL ↔ CL 用的 JSON-RPC 接口。

共识客户端会通过 Engine API 做几件关键的事情：

- 把要提议的区块骨架（谁打包、哪些交易）发给执行客户端，请它：
  - **执行交易、计算状态变更**
  - 生成执行负载（execution payload）和新状态根（state root）
- 向执行客户端询问“这个 payload 合法吗？”、“这条链头能不能继续”等信息
- 在新的区块被其他节点广播时，让执行客户端验证其中的执行结果

类比回法院：

- **执行客户端**：我把所有交易都执行过一次，给你一份“算完的结果”。
- **共识客户端**：我根据这些结果决定接纳哪个区块、哪一条链作为“官方版本”。
- **Engine API**：就是法官与书记员之间的**专用内线电话**，上面通的是区块骨架和执行结果，而不是对外公开的 REST / RPC API。

6. 再补一嘴：节点和客户端的关系

很多人会把 node/client 混用，容易搞糊涂，可以用一句话区分：

- **客户端（client）** 是软件实现（比如 Geth、Lighthouse、Teku…）；
- **节点（node）** 是“跑着这些客户端的软件实例的那台电脑”。

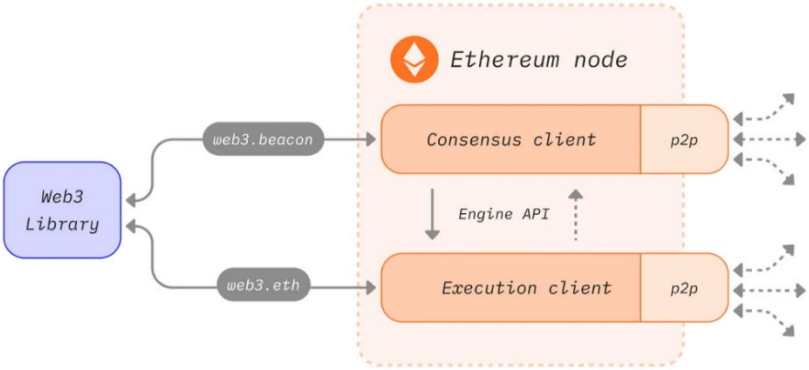
因此：

- 一台电脑上跑着 **1 个执行客户端 + 1 个共识客户端** → 这是一个**全节点**；
- 如果再挂上一个或多个验证者 → 这是一个**验证节点**；
- 轻节点则只保存部分信息，可通过同步委员会等机制验证区块头。

小结（可以直接背下来）

> 以太坊节点 = 一台电脑 + 两个必备客户端 +（可选）一个验证者客户端：
>
> - 执行客户端负责“算交易、管状态、对外提供 RPC”；
> - 共识客户端负责“跑 PoS、选区块、投票达成共识”；
> - 它们通过 **Engine API** 的“专线电话”互相协作，整个节点再通过 P2P 网络和其他节点连接起来，共同维护以太坊这条“世界计算机”。

以下是一个简易示意图：



这张图展示了以太坊（Ethereum）节点在“合并”（The Merge）后的新架构，它将原有的区块链执行层和共识层分离开来。

核心组件解析：

Web3 Library (蓝色方块):

这是与以太坊节点交互的库，比如 web3.js 或 web3.py。

它负责处理与用户应用程序的通信，并将请求发送给以太坊节点。

Ethereum Node (橙色虚线框):

这是以太坊网络的核心节点，现在它由两个主要部分组成：

Consensus client (共识客户端): 负责处理以太坊的共识层。这包括验证交易、打包区块、以及就链的状态达成一致。在合并之前，这部分功能是集成在单个客户端中的。

Execution client (执行客户端): 负责处理以太坊的执行层。这包括执行智能合约、更新账户余额、以及计算交易的结果。

Web3.beacon 和 Web3.eth (灰色箭头):

web3.beacon 连接到共识客户端。

web3.eth 连接到执行客户端。

这些接口是 Web3 Library 与节点分离后的两个客户端进行通信的通道。

Engine API (虚线箭头):

这是共识客户端和执行客户端之间进行通信的接口。

共识客户端通过 Engine API 向执行客户端发送“需要执行的区块头”，执行客户端则计算出执行结果（例如 state root）并返回给共识客户端。

这种分离使得两个客户端可以独立升级和维护，提高了网络的灵活性和可扩展性。

p2p (Peerto-Peer):

每个客户端（共识和执行）都有自己的 p2p 网络连接。

这意味着节点可以通过 P2P 网络与其他节点通信，同步链上的数据，并广播新的交易和区块。

来举个例子方便大家理解，在法院里：

执行客户端（Execution Client）像是负责整理材料的，负责记录证据、整理案卷、执行具体操作：

它接收交易（类似接受证据材料）、

执行交易（就像书记员整理案卷、运算结果）、

并维护当前状态（如记录最新案卷状态）、

共识客户端（Consensus Client） 就像是 法官，负责审判、判决：

它依靠所整理的材料（执行客户端提交的交易结果）来做出判决（决定哪个区块成为链的一部分）；

它确保所有节点就同一个链条达成共识（类似法庭裁决）。

这两者通过一个叫 Engine API 的机制“对话协作”，确保整个“审判流程”顺畅进行。

那么什么是 Engine API 呢？

Engine API 是以太坊合并（Merge）后引入的一组 内部 JSON-RPC 接口，用于连接两类独立运行的客户端：

执行客户端（Execution Client）：负责交易执行、EVM 操作及状态管理。

共识客户端（Consensus Client）：管理 PoS 共识机制、区块提议、验证与最终性等流程。

这两部分通过 Engine API 进行通信，从而协同完成区块构建与验证任务。

二、节点间的连接与通信方式

可以把以太坊节点的“社交流程”拆成三步：

> 先找到人 → 再拉上加密专线 → 最后靠八卦（gossip）把消息传满全网。

1. 节点发现：先“加好友”再扩散（基于 UDP + Kademlia）

当一个以太坊节点刚启动时，它一开始谁也不认识，只知道少数几个引导节点（boot nodes）的地址。

这些引导节点是**长期在线、对外公布地址的全节点**，新节点第一次进网时会先去找它们，就像刚进公司只认识 HR，先加 HR 微信，再通过 TA 认识别人。

1.1 Discovery 协议：Ping / Pong / FindNode / Neighbors

以太坊的节点发现协议（Node Discovery Protocol，discv4/discv5）跑在 UDP 上，专门干两件事：

①确认对方在线

- 给引导节点发 `Ping` → 对方回 `Pong`，证明“我在”。

②要通讯录

- 发送 `FindNode` 请求 → 对方回 `Neighbors`（discv4）或 `Nodes`（discv5），里面是一批“其他活跃节点的联系方式”。

新节点拿到这些“邻居”后，会继续对这些地址发 `Ping/Pong`、`FindNode`，反复迭代，逐步把自己的邻居表填满，直到达到预设上限——这一套就是**Kademlia 风格的 DHT（分布式哈希表）发现机制**。

1.2 Kademlia 的直观类比：快递分拨中心

Kademlia 里，每个节点有一个 256 bit 的 ID，“距离”用 **XOR 差异**来度量，而不是地理距离。

你可以把它类比成**全国快递分拨中心网络**：

- 每个分拨中心有一个“编号”（节点 ID）；
- 每个中心都记着“在不同距离区间内，离我最近的几个中心”；
- 当要把快递寄往一个目标编号时，会一步步转发给“离目标更近”的中心，越转越接近收件人；
- 最终快递就被精准送到目的地。

在以太坊里，**查找节点 / 扩散通讯录**也是这么做的：每一跳都把搜索范围“逼近”目标 ID，大大减少广播范围，提高发现效率。

小结：

> **发现阶段 = 用 UDP + Kademlia 找到一堆靠谱的邻居，把自己的“好友列表”填满。**

2. 建立安全连接：TCP + RLPx / devp2p

发现了别人之后，还需要建立一条真正能稳定传输数据的管道，这就进入第二步：

> 从“打招呼”升级成“拉一个加密的私聊通道”。

2.1 TCP：先搭好稳定管道

节点之间一旦决定互相连接，就会通过 **TCP 建立双向连接**。UDP 用来发现节点，因为它轻量、无连接，速度快；真正传输交易和区块这种重要数据时，还是需要 TCP 这种带确认、可重传的“可靠管道”。

2.2 RLPx：在一个 TCP 里多路复用多种协议

以太坊在 TCP 之上跑的是 **devp2p 协议栈**，其底层加密与多路复用由 **RLPx** 负责：

- 先通过握手交换密钥，建立**加密会话**（防窃听、防篡改）；
- 再在这条加密 TCP 通道上，**复用多种子协议（capabilities）**，比如：
  - `eth/66` / `eth/68`：同步区块、传播交易（执行层）
  - `les`：为轻客户端提供服务
  - 其他自定义子协议

你可以简单理解为：

> RLPx = 在一条 TCP 线上开很多“分机”：同一根网线同时承载区块同步、交易广播、状态请求等多种子协议流量。

这也是为什么会说：“以太坊的 P2P 网络主要由 **UDP 的 Node Discovery 协议 + TCP 上的 RLPx/devp2p 协议** 组成”。

3. 消息是怎么在全网“传八卦”的？—— Gossip + 请求/响应

通道建好了，接下来是最关键的一步：

> 如何在一个拥有成千上万节点的网络中，把新交易和新块尽快传遍？

以太坊这里用的是**两种模式**：

①Gossip（广播式传播）

②点对点请求-响应（拉取指定数据）

3.1 Gossip 协议：像办公室八卦一样扩散

所谓 Gossip 协议，就是模拟人类“传八卦”的传播方式：

- 某节点收到一笔新交易 / 一个新块；
- 它随机挑几个还没听说过这件事的邻居告诉一遍；
- 那些邻居再各自“转发给自己的朋友”；
- 经过若干轮之后，**整个网络几乎所有节点都知道这条消息**。

优点：

- 不需要中心服务器指挥，**完全去中心化、自组织**；
- 传播路径随机，鲁棒性好，局部故障并不会阻止全网同步；
- 实际测量中，以太坊能在几秒钟级别完成一笔交易 / 区块的全网传播。

在以太坊中：

- **执行层**：通过 devp2p 上的 eth/* 子协议，gossip 交易和区块；
- **共识层**：共识客户端有自己的 P2P 网络和 gossip 协议，用来广播 Beacon block、attestation 等共识消息。

3.2 请求-响应：按需拉取历史区块、状态等

Gossip 适合传播“**最新消息**”，但如果你掉线一段时间，需要补历史数据，就不能靠八卦了。

这时节点会通过 devp2p 的子协议，向具体的邻居 **发起精确请求**：

- “给我从高度 N 开始的 128 个区块头”；
- “给我这些区块的区块体 / receipts”；
- “给我某个区块对应的状态部分”。

这种模式就像：

> Gossip = 大家在茶水间互相聊最新八卦；
>
> **请求-响应 = 你专门去档案室翻某一天的旧报纸**。

两者结合在一起，才能既保证**新消息快速散播**，又保证**任何节点都能随时补齐历史、追上最新状态**。

4. 做个汇总收个尾

当一个新的以太坊节点上线时，大致会经历这样一个过程：

①发现阶段（UDP + Kademlia）：

先通过引导节点 +Ping/Pong/FindNode，逐步认识一群“距离 ID 较近”的邻居，把自己的“通讯录”填满。

②连接阶段（TCP + RLPx/devp2p）：

对选中的邻居建立加密的 TCP 连接，在同一条线路上多路复用区块同步、交易传播等子协议。

③传播阶段（Gossip + 请求/响应）：

新交易和区块通过 Gossip 像办公室八卦一样在全网扩散；缺失的历史数据则通过点对点请求-响应按需拉取。

三、全节点、轻节点、归档节点的区别

先来做个一句话总结：

> 全节点 = 当前状态 + 全部区块（但老状态会被“裁剪”）；
>
> 归档节点 = 全节点 + 从创世到现在的“所有历史状态”；
>
> 轻节点 = 只存区块头，需要时向全节点要具体数据。

1. 全节点（Full Node）

1.1 它是什么？

- 运行一对执行客户端 + 共识客户端，**从创世块起验证整条链**（或者从一个可信检查点开始，用 snap/fast sync 补齐后再完全自我验证）。
- 保存 **所有区块的区块头和区块体**（也就是交易、收据等历史记录不会被删）。

- 只保留**最近若干个区块的完整状态**（典型是最近 ~128 个区块的状态数据），更老的状态会被“修剪（pruning）”掉，以节省磁盘。

1.2 它能做什么？

- 验证所有新来的交易和区块，独立判断“这条链是不是合法”；
- 直接为钱包、dApp、脚本提供 JSON-RPC 接口，是一个“**完全自托管的以太坊入口**”；
- 理论上可以**重构任何历史状态**——只是对非常久远的状态，可能需要向归档节点请求一些旧数据，再自己重放计算。

关键点：

> 全节点最重要的特点是“自己验证，不求人”，但不会保存所有历史状态快照，这部分交给归档节点。

2. 归档节点（Archive Node）

2.1 它是什么？

- 本质上就是：**禁用修剪（pruning）的全节点**。
- 除了全节点拥有的所有数据以外，**还保存从创世到现在的每一个区块高度下的完整状态**：
  - 某个老区块高度上每个账户的余额、合约存储、状态树……都能“一查即得”，不需要重算。

2.2 它的代价：

- 体量非常大：从 TB 甚至多 TB 起步；
- 同步时间长、维护成本高，不适合普通用户在家里跑。

2.3 谁需要归档节点？

- 区块浏览器（Etherscan 类）、链上分析公司；
- 高级调试 / 回测工具：

- 比如你想直接问：“在区块 #4,000,000 时，某合约的某个 storage slot 是多少？”

- 一些需要频繁查 **任意历史状态快照** 的服务。

关键点：

> 归档节点 = “国家档案馆”：
>
> 保存了**所有历史状态**，对研究、分析、服务商非常有用，但对普通用户来说完全是“性能过剩”。

3. 轻节点 / 轻客户端（Light Node / Light Client）

3.1 它是什么？

- 只保存 **区块头（block headers）**，不保存完整区块体和全量状态；
- 需要时，向全节点/归档节点请求具体数据（交易、账户状态、合约存储等），并用区块头里的 **state root / Merkle 根** 对收到的数据进行验证。

3.2 优点：

- 极大降低存储与带宽需求，可以在：
  - 手机、浏览器插件、硬件设备等资源受限环境中运行；
- 依然保留“**可验证性**”：
  - 它不需要完全信任提供数据的全节点，可以用 Merkle 证明 + 状态根独立校验。

3.3 限制：

- 不参与共识：
  - 不能做验证者（不签区块、不投票）；
- 对数据获取有一定依赖，需要有可访问的全节点 / 网关。

以太坊在 PoS 后专门为轻客户端设计了 **同步委员会（sync committee）** 协议，让轻客户端只需验证 512 个验证者对区块头的签名，就能安全地追踪链头，进一步降低资源消耗。

关键点：

> 轻节点 = “只看目录 + 零信任校验”：
>
> 不存全书，只存区块头和必要证明，但依然可以做到“不完全信任别人、自己核对关键部分”。

4. 来做一个图书馆的类比

4.1 全节点（Full Node）

像一个**大型公共图书馆**：

- 有全部书目（所有区块和交易）；
- 有一份“最新版本”的书（当前状态）；
- 旧版本虽然不都放在书架上，但可以通过查记录 + 向档案馆请求，再把旧版本复原。

4.2 归档节点（Archive Node）

像**国家档案馆**：

- 保存了每一本书的**所有历史版本**；
- 想查“2014 年版这本书第 23 页是什么内容”，档案馆秒给你答案；
- 维护成本巨大，平时普通人不会自己建一个。

4.3 轻节点（Light Node）

像一个**移动书柜 / 电子目录机**：

- 自己只存“书目和索引”（区块头），
- 真正要看书时，会去图书馆（全节点）借阅，并对照目录核对“我拿到的是不是对的”。

来做一个表格方便对比：

类型	存储什么数据	资源消耗	典型用途	信任 / 安全属性
全节点	全部区块 + 最近若干块的完整状态（老状态会被修剪）	中等（数百 GB 级磁盘 + 稳定带宽）	自托管 RPC 节点、参与网络、做验证者的基础	自己验证所有交易和区块，最信任最抗审查
归档节点	全节点数据 + **从创世到现在的所有历史状态快照**	极高（TB 级磁盘）	区块浏览器、链上分析、历史调试与回测、节点服务商	同全节点，但还能随时回答“任意高度的任意状态”
轻节点	区块头 + 必要的轻客户端数据，不存完整区块和状态	很低（适合手机 / 嵌入式）	轻钱包、浏览器插件、跨链桥验证、IoT 设备	通过状态根 + Merkle 证明验证数据，安全性接近全节点，但数据依赖全节点提供

四、开发者或机构运行全节点的必要性

正文之前，先来给大家介绍 RPC：

什么是 RPC（Remote Procedure Call）

英文全称：RPC = *Remote Procedure Call*，即“远程过程调用”。

简单来说，就是一个程序（或客户端）可以像调用本地函数/方法一样，向远端服务器节点（或服务）发出请求，让对方执行某个过程（procedure / 方法 / 函数），并返回结果。客户端自己并不关心这个过程实际运行在哪台机器或在什么网络之中。

在区块链里的含义与作用：

在区块链生态中，RPC 通常指 *区块链节点提供的一种接口*，供应用程序（比如钱包、dApp、区块浏览器等）远程调用。通过这种接口，应用可以：

- 查询区块、交易、账户余额、智能合约状态等数据；
- 提交新的事务到网络；
- 调用智能合约的方法；
- 监测交易确认状态等。

节点（RPC 节点）或 RPC 服务商就提供这些接口和 endpoint。应用端只需要知道这个 endpoint 地址，以及支持的 RPC 方法，就能与区块链交互，而不需要每个应用都自己运行完整节点。

如果也理解不了的话，我来做个类比吧。

想象一个公司有一个客户服务中心（客服），你通过电话打过去：

你（客户端）拨通电话，告诉客服你想查询某个事情（比如 “我余额还剩多少？” 或者 “我刚才交易的状态是怎样？”）。

客服中心就像 RPC 节点，它内部连接数据库或后台系统，查出余额 / 交易状态后告诉你答案。

你不需要知道客服后台系统内部结构，也不需要自己去部署整个数据库与服务器；只要有客服电话号码（RPC endpoint）和合适的询问方式（RPC 方法名 + 参数），就能得到你想要的信息。

回归正文，这个问题可以直接记成一句话：

> 谁在给你的 dApp 喂数据，谁就在“半控”你的安全和用户。 自己跑全节点 = 不求人 + 更安全 + 更隐私 + 更去中心化。

1. 不泄露用户隐私：不用把地址 & IP 交给第三方

大多数钱包 / dApp 默认连的是公共 RPC（Infura、Alchemy、QuickNode 等）。

这意味着：

- 你每次查余额、读合约、发交易，**请求都会先到这些 RPC 提供商那里**；
- 在实践里，很多提供商的隐私条款都明确说明会收集 **IP + 钱包地址** 等元数据。

研究和社区讨论也反复指出：

默认使用公共 RPC 的钱包，会让提供商“有能力把 IP 和链上地址关联起来”。

而官方 / 社区文档里对“自己跑节点”的第一条好处，就是：

- **不用把你的地址、调用、交易习惯暴露给别人的节点**；
- 你和以太坊的所有交互都只经过自己的机器。

对开发者/机构来说，这点非常关键：

> 你可以给钱包和内部系统暴露 **自己的 RPC endpoint**，请求只在你的网络 / 合规边界内流转，不被外部基础设施“顺带看光”。

2. 不被卡脖子：抵御 RPC 层面的“软审查”

如果你完全依赖公共 RPC，会遇到几个风险：

- 服务商可以基于 IP、地区、请求特征做 **限流或封锁**；
- 某些交易或合约交互，可能因为内部政策被拒绝广播；
- 服务商宕机或出现 bug 时，你的 dApp 直接一起“挂掉”。

而以太坊官方和不少基础设施文档会强调：

> 运行自己的节点是获得 抗审查性、可用性和主权（sovereignty） 重要一步。

对机构来说，自己跑全节点意味着：

- 任何时候你都能**自主广播交易**，而不是看某个云服务心情；
- 即使有第三方服务“降级/风控”，你的业务仍然能通过自有节点维持基本功能。

3. “不要信任，要验证”：自己验证每一笔交易和区块

从安全模型来看，以太坊真正的哲学只有八个字：

> Don’t trust, verify. 不要信任，要验证。

当你自己运行全节点时：

- 所有区块和交易都会在本地被完整验证（共识 + 执行）；
- 你**不必相信任何公共 RPC 提供的结果是对的**，可以自己重新算一遍；

- dApp、风控系统、清结算逻辑都可以直接基于“自己验证的状态”来做判断。

这在几个场景里特别重要：

- 金融机构 / 交易所做**入账确认**；
- 做清结算、风控时，需要确信数据没有被中间人篡改；
- 做合规审计，需要有可以追溯和复现的“本地真相来源”。

4. 帮以太坊“撑腰”：去中心化和客户端多样性

从网络角度看，你的每一个全节点其实都是：

- 又一个**完全独立的共识和执行视角**；
- 又一个**不会被某家云服务“一键关停”的副本**。

同时，以太坊还有一个非常强调的维度：**客户端多样性（client diversity）**：

- 执行层有 Geth / Nethermind / Besu / Erigon / Reth 等多个实现；
- 共识层有 Lighthouse / Prysm / Teku / Nimbus / Lodestar 等。

社区和研究明确指出：

如果某个客户端占比太高，一旦它出 bug，可能直接拖垮整条链的安全性或终局性，因此大家一直在鼓励把每个客户端的占比压到 1/3 以下。

当开发者或机构选择：

- 跑一个全节点，
- 并有意识地选择一个“不是头部垄断”的客户端实现，

你实际上就是在非常直接地**增强以太坊的弹性和去中心化**。

5. 为 dApp 和内部系统提供稳定的“数据底座”

从工程实用角度说，自己有全节点最大的爽点之一是：

> 你可以把它当成自家的以太坊后端。

典型收益：

- **自定义 RPC 接口**：根据业务需要配置限流、日志、鉴权、白名单等；
- **更低延迟**：节点就在本地机房 / 内网，读写延迟和稳定性比跨地区公共 RPC 要好；
- 可以在节点之上挂：
  - 索引服务（Graph、定制 indexer）；
  - 内部风控 / 数据仓库流水；
  - 各部门使用的分析面板。

很多教程明确写着：跑自己节点的好处之一，就是可以**自托管工具和服务（run your own tools/services）**，而不是完全依赖外部基础设施。

6. 高级玩法：归档节点支撑分析、审计和合规

严格来说，“归档节点（Archive Node）”是全节点的升级版，但在机构语境里**基本是绑在一起谈的**，特别是金融和合规场景。

归档节点在全节点基础上，多做了一件事：

> 保存从创世到现在的全部历史状态快照，可以瞬间回答：
>
> “在区块 #N 的时候，这个地址/合约的状态是什么？”

所以它非常适合：

- 做链上行为的 **合规审计与报表**；
- 做 **量化研究、策略回测、风控模型训练**；
- 为外部团队提供“历史视图丰富的链上数据服务”（区块浏览器、分析平台、RWA 平台等）。

普通开发者只需要全节点就够日常开发；

需要大量历史查询 / 调试 / 审计的机构，就几乎总会考虑自己维护一套归档节点。

7. 升级与分叉选择权：用脚“投票”的能力

在以太坊这种“社会共识链”里：

- 协议升级（比如从 PoW → PoS、Dencun、Pectra 等）
- 或者偶发的分叉

最终都是通过 **“谁跑哪个客户端版本、谁跟随哪条链”** 来定胜负的。

如果你完全依赖第三方节点：

- 他们升级到哪条链，你就“被动跟随”哪条链；
- 你没有真正意义上的 **“协议主权”**。

而当你自己运行全节点时：

- 你可以决定什么时候升级、是否接受某个 EIP；
- 必要时可以观察两条链的情况，**主动决定自己认哪条为“正统”**。

这对大型机构和协议方而言，其实是一种非常重要的“治理话语权”。

8. 配置自由：客户端选型、参数调优、合规要求

最后一条是很工程的现实问题：

> 只有自己的节点，才能做到**“按你的业务来调，以你的合规来配”**。

包括但不限于：

- 选用不同客户端（Geth、Erigon、Nethermind、Besu…）满足：
  - 性能需求（快速同步、大量并发读写）；
  - 语言栈 / 运维团队熟悉度；
  - 企业特性（例如 Besu 对企业网络友好）。

- 自己控制日志留存、监控指标、访问控制策略，满足合规或内部审计要求；
- 为共识层 / 执行层做特殊配置（比如接入本地 MEV 流，或对接专用 L2、跨链组件）。

可以送给读者的一句“背诵版总结”

> 开发者和机构运行全节点，是为了拿回三件东西：
>
> ① 自己验证、自主管理的安全与隐私；
>
> ② 抗审查、抗单点故障的基础设施韧性；
>
> ③ 在以太坊协议和应用层上的话语权与自主权。 这既是为自己负责，也是为整个以太坊网络“增肌”。

五、归档节点在数据查询中的优势

这节理解成一句话：

> 归档节点 = “国家档案馆 + 时间机器”：
>
> 全节点只知道“现在是什么样”，归档节点还能瞬间回答“历史上任意一个时刻是什么样”。

在以太坊里，**归档节点（Archive Node）= 全节点 + 所有历史状态快照**：

- 和全节点一样，它保存**所有区块和交易数据**；
- 额外再保存**每一个区块高度的完整状态**（账户余额、合约存储、状态树等），从创世块一直到当前最新块。

而普通全节点一般只保留**最近约 128 个区块的状态**，更老的状态会被裁剪，只能通过重新回放交易来“重算”出来，既慢又容易 OOM。

1. 即时查询任意历史状态（无需重放）

最大优势就是“**快**”：

- 查询“这个地址在区块 #4,000,000 时的余额是多少？”
- 查询“某合约在两年前的某块高度，某个 storage slot 里是什么值？”
- 查询“在某个老块时，整条链状态根 / 代码哈希是什么？”

对归档节点来说，这些都是**本地直接读盘**的操作，不需要重放几百万个区块；而对普通全节点，这种查询要么很慢，要么直接报 *missing trie node* 错误。

Cloudflare、Alchemy 等文档也明确写了：

> 虽然全节点理论上可以通过重放交易重建历史状态，但在实践中开销巨大；
> 归档节点因为提前把所有状态都存下来，在历史查询上性能优势非常明显。

2. 支持按区块高度查询的高级 RPC

有些 JSON-RPC 方法支持 “`blockNumber` 参数”，比如：

- `eth_getBalance`
- `eth_getCode`
- `eth_getStorageAt`
- `eth_call`（对历史区块执行只读调用）

当你请求的是 128 个区块之前的状态时，就需要归档数据——也就是需要连到归档节点，否则很多客户端会直接报错或超时。

这对下面这些场景非常关键：

- 做“**在某个历史时间点重放调用**”的调试工具；
- 做合约升级前后行为对比；
- 回溯某次攻击发生前后的状态。

3. 历史分析、审计与治理的基础设施

归档节点的“时间机器”特性，让它在很多业务里几乎是必需品：

- **区块浏览器 / 数据服务商：**
  - Etherscan、各种 analytics 平台，需要随时展示“某地址在多年前某块的余额变化曲线”，离不开归档数据。
- **市场与链上行为分析：**
  - 量化研究、MEV 分析、RWA 资产监控，都需要长时间窗口的细粒度链上历史。
- **合规与审计：**
  - 金融机构、审计公司要检查“某账户在某一段时间内的交易与余额”，归档节点提供的是**可以复现的链上证据源**。
- **开发者调试与回归测试：**
  - 想在真实历史数据上重放某合约、检测升级影响、验证修复效果——有归档节点就能“一键在某块高度跑一遍”。

简单说：

> 只要你需要“看过去链上任何时刻的真实状态”，归档节点就是最直接的工具。

4. 代价：空间和同步时间非常“奢侈”

你原来举的数字基本是对的，最新数据也差不多这一量级：

- 运行以太坊归档节点的存储需求，
  - **Geth**：约 **12 TB+**，
  - **Erigon**：约 **2–3 TB**（专门针对归档做过优化），
  - 其他客户端如 Nethermind、Besu 也在十几 TB 级别。

再加上：

- 首次同步需要从创世开始重放全部区块，往往是**数周到数月**的工程；
- 硬件要用 NVMe SSD、较大内存和稳定高带宽，否则中途很容易卡死或超时。

因此，以太坊官方文档会强调：归档节点**不是参与网络共识的必要条件**，更多是给区块浏览器、数据服务商、研究机构和少数开发者用的“重型工具”。

一句形象的小结

> 全节点知道“现在发生了什么”，归档节点记得“从第一天起发生过的一切”。 对所有需要频繁查历史状态的应用来说，归档节点就是一台链上的时间机器——代价是，你得在机房里给它备一整个机柜的硬盘。

六、合并后执行与共识客户端的区别

The Merge 之后，以太坊正式从 PoW 切换到 PoS，同时把原本“一个大客户端”拆成了两块相互独立又强耦合的软件：

- **执行客户端（Execution Client，EL，原 Eth1）**
- **共识客户端（Consensus Client，CL，原 Eth2 / Beacon）**

一个“完整以太坊节点”必须同时运行这两种客户端，并通过 **Engine API** 互相通信。

可以直接记一句：

> 执行客户端管“算账 + 状态”，共识客户端管“选块 + 投票”。

1. 执行客户端（Execution Client）：EVM + 状态 + RPC

执行客户端负责以太坊的**执行层（Execution Layer）**，也就是你理解中的“业务逻辑”和“链上状态机”：

1.1 处理交易

- 接收转账、部署合约、调用合约函数等交易；
- 对交易做基本校验（签名、nonce、余额、gas 等）。

1.2 执行 EVM 操作

- 在以太坊虚拟机里逐条执行交易；
- 计算 gas 消耗、抛出异常、更新 storage 等。

1.3 维护状态与数据库

- 维护当前链的**世界状态**：账户余额、合约代码、storage、状态树等；
- 负责把这些状态写入本地数据库（如 LevelDB、MDBX 等）。

1.4 提供 JSON-RPC API

- 提供 `eth_*`、`net_*`、`web3_*` 等标准 RPC 方法，是 dApp、钱包、脚本接入以太坊的主要入口。

1.5 执行层 P2P 网络

- 通过 devp2p / RLPx 网络协议与其他执行客户端交换区块、交易等数据（执行层的 gossip）。

1.6 配合集成 Engine API

- 接收共识客户端通过 Engine API 发来的“待执行区块骨架”，执行后返回 execution payload（执行结果 + 新状态根）。

常见执行客户端：Geth、Nethermind、Besu、Erigon、Reth 等。

> 可以把执行客户端想成：
>
> **“世界计算机”的 CPU + 内存 + 本地数据库 + RPC 网关。**

2. 共识客户端（Consensus Client）：PoS + Beacon + Finality

共识客户端负责以太坊的**共识层（Consensus Layer / Beacon Chain）**，专门

处理 PoS 逻辑与链头选择：

2.1 运行 PoS 共识，提议 / 验证新区块

- 维护 Beacon 链视图、slot/epoch 节奏；
- 随机选出提议者（block proposer）；
- 收集和处理来自验证者的投票（attestations）。

2.2 追踪链头与最终性（finality）

- 使用 LMD-GHOST + Casper FFG 等机制选择“当前链头”；
- 在足够多的投票支持下，给出 epoch 级别的“最终确定性”（finalized checkpoint）。

2.3 管理验证者集与奖励 / 惩罚

- 负责记录每个验证者的质押余额、在线情况；
- 处理奖励、罚没（slashing）、inactivity leak 等经济机制。

2.4 共识层 P2P 网络

- 通过自己的 P2P 网络 gossip Beacon blocks、attestations、slashings 等；
- 和执行层的 P2P 网络是两个“平行宇宙”。

2.5 Beacon API / 共识 RPC

- 共识客户端也有自己的 HTTP / REST / gRPC 接口（Beacon API），可以查询 Beacon 状态、验证者信息、slashings 等，但**不提供 `eth_*` 这类执行层 RPC**。

常见共识客户端：Prysm、Lighthouse、Teku、Nimbus、Lodestar 等。

> 可以把共识客户端想成：
>
> **一群“裁判 + 选举委员会”**：负责安排谁出块、谁投票、票数到哪条链头上，什么时候给“最终盖章”。

3. 它们之间如何配合？——Engine API 的“内线电话”

后 Merge 架构的精髓在于：

> 以太坊节点 = 执行客户端 + 共识客户端 + 它们之间的 Engine API。

一个典型“出块 / 同步”流程大致是这样：

3.1 共识客户端收到新区块 / 或被选中提议新区块

- 作为 proposer，它会向执行客户端发出类似 `engine_forkchoiceUpdated` / `engine_getPayload` 等调用，让执行客户端根据当前 mempool 组装交易、执行并生成 execution payload。

3.2 执行客户端执行交易，返回执行结果

- 在本地 EVM 里跑完所有交易，更新状态，返回：
  - 交易列表、收据、状态根、logs bloom 等完整 payload。

3.3 共识客户端把执行结果“装进” Beacon 区块

- 将 execution payload 嵌入到 Beacon block 中，作为该 slot 的提议区块；
- 通过共识层 P2P 网络广播出去，供其他验证者投票。

3.4 其他节点的共识客户端接收区块 → 交给本地执行客户端复验

- 验证 PoS 签名、slot/epoch、历史链等共识规则；
- 再让执行客户端验证执行 payload 是否合法（重算一次）；
- 全部通过后，这个区块才被接受到本地链中。

小结：

- **执行客户端**负责“这块里的交易是否合法？最终状态是什么？”
- **共识客户端**负责“我们全网要不要接受这块当下一步？哪条链是正统？谁作恶要被 slash？”

4. 对开发者来说，记住这几个差异就够了

差异：

- **调用 `eth_*` / 部署合约 / 查状态？**

  → 找的是 **执行客户端**（EVM + JSON-RPC）。

- **质押 32 ETH 做验证者 / 关心 attestation、finality、slashing？**

  → 这是 **共识客户端 + 验证者客户端** 的工作。

- **为什么要拆两块？**

  → 为了模块化、易维护、多客户端实现、独立升级，以及更清晰的安全边界：

  执行层可以专心把 EVM 和状态做到极致性能，共识层可以独立演进 PoS、PBS 等协议设计。

一句话做结尾：

The Merge 之后：执行客户端是“世界计算机的大脑皮层（算逻辑）”，共识客户端是“集体神经系统（达成一致）”，两者通过 Engine API 绑在一起，才构成一台真正的以太坊节点。

七、执行与共识客户端的协同配合

1. 一个节点 = 执行客户端 + 共识客户端 + Engine API

The Merge 之后，一个完整以太坊节点必须同时跑：

- 1 个 **执行客户端（EL）**：Geth、Nethermind、Besu、Erigon、Reth 等；
- 1 个 **共识客户端（CL）**：Lighthouse、Prysm、Teku、Nimbus、Lodestar 等；
- 两者通过本地的 **Engine API（JSON-RPC 接口族）** 通信。

执行层和共识层各自有独立的 P2P 网络，但**它们之间的桥**只剩下这一条 Engine API，本质上就是几组方法：

- `engine_newPayload`
- `engine_forkchoiceUpdated`
- `engine_getPayload` 等。

2. 当本地是出块者：CL 找 EL“要一块菜”

场景：你的节点里挂着验证者，轮到你在某个 slot 当 proposer 了。

①共识客户端选中你当 proposer

- 它负责按照 PoS 规则决定“这一 slot 由谁提块”。

②CL 通过 `engine_forkchoiceUpdated` 告诉 EL：“当前链头是这个，高度是那个，我要准备出块了。”

- 这个调用会带上 forkchoice 状态和一份 `payloadAttributes`（时间戳、随机数、feeRecipient、withdrawals 等）。
- EL 收到后返回一个 `payloadId`——理解为“正在烹饪中这道菜的订单号”。

③CL 用 `engine_getPayload(payloadId)` 向 EL 要最终的执行负载

- 执行客户端从 mempool 里挑交易，跑一遍 EVM，生成 `ExecutionPayload`（交易列表 + gas 消耗 + 新状态根等）。

④ CL 把 ExecutionPayload 塞进信标区块，广播给全网

- 这时一个完整的 Beacon block = 共识层头信息 + 执行层 payload。

类比一下：

- 共识客户端（经理）打给执行客户端（厨师）："我们打算把这桌菜当今晚主打，当前菜单和时间给你，你帮我出一整道菜（payload）。"
- 执行客户端负责真正翻锅、算账，最后把做好的菜端给经理，经理再决定是否端上全场（广播）。

3. 当别人出块：CL 收块 → 让 EL 复算 → 再决定投不投票

场景：别的验证者提了一个区块，你的节点需要验证并（可能）投票。

①共识客户端从 P2P 网络收到了新的 Beacon 块

- 它先检查签名、slot、父块是否合理等 PoS 规则。

②CL 提取出里面的 ExecutionPayload，用 `engine_newPayload` 丢给 EL

- 这相当于对执行客户端说："你帮我看看，这块里的交易和状态变更在 EVM 里是不是合法的？"

③执行客户端本地重算这一块

- 重新执行所有交易，检查 gas、余额、状态根是否一致；
- 如果通过，就更新本地执行层链头，并向 CL 回"VALID"；
- 如果不通过，就回"INVALID"，CL 会拒绝把这块接到链上。

④CL 再用 `engine_forkchoiceUpdated` 把"新链头"告诉 EL

- 确保两层都认为现在的 head / finalized / justified 是一致的。

⑤共识客户端据此决定是否为该块投 attestation

- 通过则向全网广播投票；
- 失败则视为对方作恶或错误实现，有可能配合 slashing 机制。

法院类比：

> - 别的法院送来一份“已经判好的判决书”（Beacon block）；
> - 你的法官（CL）把案卷丢给书记员（EL），让他 **按流程再跑一遍所有证据和计算**；
> - 书记员说“这套判决在程序和账目上没问题”，法官才会在自己这边盖章、认可这条“判例”，并投票支持。

4. 同步阶段：谁听谁的？

在同步链的时候，两者的合作大致是这样：

- 共识客户端从共识层 P2P 网里**先把 Beacon 链同步好**，知道“理论上的链头”。
- 再用 `engine_forkchoiceUpdated` 把预期的链头告诉执行客户端，让 EL 按这个方向去同步执行层区块。

这就是 ethereum.org 文档里提到的 **optimistic sync** 思路：EL 可以先“乐观地导入” Beacon block，对执行 payload 延后验证，在追上链头后再逐步完成全部执行验证。

> 简单说：共识层负责“告诉你应该追哪条链”，执行层负责“把这条链上的具体账本和状态补全并自验一遍”。

留一句话总结（可以直接背诵）：

执行客户端负责“算结果、管状态”，共识客户端负责“选哪条结果被全网认可”，两者通过 Engine API 三大动作 `newPayload / forkchoiceUpdated / getPayload` 互相配合 —— 这就是 Merge 之后以太坊节点内部的日常节奏。

八、节点同步的目标与意义

节点要同步的内容主要包括两部分：

①区块链数据（每个区块及其头信息）

②状态数据（每个账户余额、合约存储、代码等）

同步模式，以太坊支持三种主要的同步方式：

1. FullSync（全同步）

节点从创世区块开始，下载每个区块并重新执行其中的所有交易，验证其状态变化。

状态数据是逐块执行产生的，而不是直接从其他节点获取。

优点：最安全、无需信任他人。

缺点：极慢，通常需要数天甚至更久。

把区块链同步比作 “复刻一本百科全书＋图书馆藏书历史档案”。我们来看 Full Sync 是什么样的操作：

想象你要拥有一个新的图书馆，该图书馆要收藏从第一版百科到最新每一版：包括每一版里的所有章节、所有的插图、所有修订的内容。

而且不仅要把每一版书都放进书架，还要自己逐个对比每一版里改动的地方，确认是不是正版，没有伪造或错误。你不仅要下载每一版书，还要校对每一页的修改、每次改版是否按规则执行。

这个过程耗时很久：要把几十年、数百本书、每本书每个版本都处理完；也要有足够大的货架空间来存所有版本的实体书或者纸本（等于区块 + 状态 +历史数据）；还要有精力与工具去校对。

2. SnapSync（快照同步）

节点首先下载最新区块头链（header chain），然后请求一个可信状态快照（snapshot）——主要是 Merkletrie（状态树-默克尔树）的根节点和部分分支节点。

随后通过并发下载账户、存储、代码等状态数据，填充本地数据库。

并发性强、速度快，通常几小时到一天内完成。

完成后节点会继续下载并执行最新的交易数据。

Geth、Nethermind 等客户端都默认采用 SnapSync 模式。

Snap Sync 就像你要搬进一个新房子：

Snap Sync：有人帮你把旧房子里最重要、最近在用的家具、电器等（state snapshot）直接从旧房子里搬过来，还附上重要的房屋布局图（区块头 + 状态树根 +部分校验结构）。然后你自己把房子里最近改变的布置（最新的区块）一个个补上 → 速度快很多。

3. LightSync（轻节点同步）

不保存完整状态，也不执行所有交易，只下载区块头和部分区块体，用于轻量级验证。

适合钱包或浏览器插件，不推荐用于验证或运行 RPC 服务。

基于 LES（Light Ethereum Subprotocol）协议。

状态同步的过程简化为：

①连接到邻居节点，获取区块头（header chain）

②验证区块头哈希链的连续性

③请求最新状态根的快照，并下载账户状态数据

④填充本地状态 trie 数据结构

⑤获取剩余未同步的区块体并执行交易

⑥进入完全同步状态，开始参与网络共识与广播

想象一部电影的复制过程：

LightSync / Light Client 模式 就像你只下载电影的“预览片段”（trailer）＋电影封面＋关键镜头的一些截图。当你真的想看某个整场景或某个时间点细节，你就请求别人发一个对应时间点的关键帧给你。你能知道剧情大致走向、主要角色是谁，但如果想看所有细节（例如某一句台词或某个背景细节）就得等别人提供。

以下是一个表格方便大家理解：

步骤	操作内容
1	下载区块头并验证其正确性
2	下载区块体（交易和收据）
3	Snap 模式下获取状态叶子节点并本地构建状态树
4	Full 模式下从创世块逐块执行状态变化
5	完成状态同步后，节点进入正常运行状态，并继续同步新块

Snap 模式需本地重构状态树并“恢复”同步状态

Full 模式需逐块执行每笔交易并验证状态根

Ligh 节点同步无需构建状态，仅验证区块头与 Merkle 证明

补充一下安全说明：

状态同步时，节点会对所有区块头和状态数据进行哈希验证。由于以太坊的状态根是默克尔树根，客户端能通过哈希路径验证每一个账户或存储项的真实性，避免中间人伪造状态数据。

这三种同步模式什么时候用呢？

如果你是个人 / 小型项目，只需要钱包 / dApp 后端做基本查询 → 用 **SnapSync**（如果客户端支持）或者 **Light Client**（如果资源非常有限）。

如果你需要高安全性 /不信任第三方 /做审核 /历史数据相关功能 → 用 **Full Sync** 或归档节点。

如果你只是偶尔需要历史状态，而且你能接受从外部服务查询 → 轻节点＋外部 provider 是可行。但要注意信任 与可用性。

出现了区块头和区块体，也简单介绍一下。

在区块链中，一个 区块（Block） 通常由两大部分组成：

区块头（Block Header）：包含区块的元数据，用于识别、验证与链接区块。

区块体（Block Body）：包含实际交易数据和其他可选信息，是区块中“做事情”的部分。

区块头里通常包含什么：

字段名称

父区块哈希（Previous Block Hash / Parent Hash）

时间戳（Timestamp）

Merkle root（Merkle 树根）

与共识机制相关的字段

区块体里通常包含什么：

区块体包含真正“干活”的内容，主要是所有被包含在该区块中的交易（Transactions）。除此之外，还可能包含：

所有交易的详细信息（发送者 / 接收者地址、金额、手续费、签名等）；

（在某些协议里）交易收据（receipts）、日志（logs）等与交易执行相关的附加数据；

可能还有合约代码、状态变更、存储数据等（视链协议不同）；

区块头就像章节开头的目录 +简介：告诉你这一章是第几章（父区块哈希就像前一章的章节编号），这一章的主题是什么（Merkle root 就像章节内容摘要），什么时候写的（时间戳），这个版本的书是哪年印刷的（版本号 / 难度等类似）。

区块体就像章节的正文内容：包含所有故事情节（交易详情），对话、事件、细节等。

九、节点间交换的数据类型

交易信息：节点负责存储、验证和转发交易信息。每笔交易都被记录为一个数据“区块”，其中包含资产从一方转移到另一方的详细信息，例如参与方、交易内容、发生时间、地点、原因、交易金额以及满足的任何预设条件。

区块数据：数据以区块的形式存储并链接在一起，形成一个链。节点验证并传播新的数据区块，以更新整个网络。每个区块还包含元数据，如版本号、前一个区块的哈希值（用于链接）、所有交易的加密表示、时间戳、难度目标和随机数（nonce）。

加密哈希：区块通过加密哈希值相互链接，这些哈希值是每个区块的唯一标识符，并确保交易的顺序和时间。区块链上的所有内容，包括交易标识符、钱包和区块标识符，都以哈希值的形式存在。

账户信息：这包括账户余额、随机数、合约代码和合约存储等。账户可以是外部拥有账户（EOA- **Externally Owned Account**），由私钥控制并用于发送和接收交易，也可以是合约账户，由智能合约拥有并在被触发时执行操作或交易。

公钥和私钥：公钥密码学用于唯一识别区块链网络中的参与者。每个成员都有一对密钥：公钥是公开的，作为接收加密货币或数据的地址；私钥是保密的，用于授权交易并控制相关数字资产。

智能合约代码和执行数据：以太坊节点处理智能合约和状态变化。智能合约通常用高级语言编写，然后编译成以太坊虚拟机（EVM）可理解的字节码进行执行。交易数据还可以包含函数参数和外部调用数据（calldata）。

网络状态：EVM 会更新以太坊上所有账户和智能合约的状态，包括账户余额、随机数、合约代码和合约存储。全节点维护区块链的完整副本，并作为区块链状态的“事实来源”。

日志/事件：智能合约在执行过程中可以发出日志或事件，外部系统可以利用这些信息进行监控。

这里补充介绍一下 EOA：

把区块链账户比作“银行账户”：

EOA 就像你在银行开的那种标准账户，你自己持有这张银行卡和密码（私钥）。你可以存钱、取钱、转账，但银行系统里没有自动执行你没告诉它做的任何事。

我不知道大家是否了解私钥公钥和助记词之间的关系。

刚好我之前写过这样的一篇文章，给大家简单介绍一下。

从种子（Seed）生成私钥

助记词经一定流程（例如先变为一个种子数 seed，然后使用 HD 钱包标准如 BIP-32，BIP-44 等），从这个种子派生（derive）出一个或多个私钥（Private Key）。

HD 钱包（Hierarchical Deterministic Wallet）允许你用同一个助记词生成多个私钥，每个私钥控制一个或多个区块链地址 /账户。

私钥生成公钥

一旦有了私钥，就可以通过椭圆曲线加密（Elliptic Curve Cryptography, ECC）等算法生成对应的公钥。私钥是一个大整数；公钥通常是 ECC 点（x, y 坐标）或其压缩形式。

从公钥生成地址

公钥再经哈希（hash）等操作（取公钥哈希的一部分，或加上网络前缀等）产生一个地址（Address）。这是一个更短、更方便用来接收资产的标识符。

区块头（Block Headers）：包含区块编号、前一个区块的哈希、时间戳、状态

根、难度/PoS 信息等。用于快速同步最新区块链进度，验证链的连续性和一致性。

区块体（Block Bodies）：包含完整的交易列表等详细信息。节点下载它们以获取实际交易数据和执行操作。

交易（Transactions）与交易收据（Receipts）：节点会广播新的交易，当收到时会执行并在处理后生成收据，包括 Gas 用量、事件日志等。这些收据会用来验证交易状态和事件触发。

状态 trie 数据（StateTrie/DAG 片段）：通过 SnapSync 或其他同步策略，节点会交换账户/合约代码、存储槽和部分状态树数据（默克尔-Patriciatrie 节点）。用于构建和验证当前链上状态（例如账户余额和合约变量）。

这里又出现了很多名词，来逐个介绍一下。

State Trie / World State Trie（状态树 /全局状态树 /Merkle-Patricia Trie）

以太坊将所有账户的状态（余额、nonce、智能合约代码地址、合约存储根等）组织在一个 Modified Merkle-Patricia Trie（MPT）中。这个状态树的根哈希（stateRoot）被记在区块头里，以表示一块执行完所有交易后的“当前全网状态”

有向无环图（DAG）

DAG 在区块链里可能有多个用法。例如以太坊 PoW 算法里有一个与挖矿相关的 DAG（Ethash 的 dataset），但这与 “状态树 /状态 DAG 片段” 通常无关。这里“状态树/DAG 片段”通常指状态树里的路径 / 子树 /部分节点，不是挖矿算法的 DAG。

这样吧，我再来一个类比……

想象一个大型图书馆拥有很多大书。每本书是一个账户 / 合约，有很多章节和页数。

“状态树（State Trie）”就像整座图书馆里所有书的目录 +内容索引 +书籍内容。你可以通过目录定位书，也可以看到具体页内容。

“状态树片段”像是你借阅时仅索要某本书的某一章 +目录树中的路径：比如你只想读某本书的第 5 章第 2 节，那你可能需要：这本书目录里指向第 5 章的路径 + 第 5 章的章标题 + 第 2 节内容 +必要的章节上下文。你不需要把整本书都带走 (所有章节、所有书)，只拿你要读的那个片段，并且目录里有索引保证你拿的是正确的那一章节，而且与你查询的目录是一致的。

如果你后来又要读别的章节，你再请求别的片段。

这个 trie 呢也比较有趣，他被称为前缀树，或数字查找树，他是来自 re trie val（检索）这个单词，因为它最初就是拿来做检索的。

而以太坊就是采用的 Merkle-PatriciaTrie（默克尔-帕特里夏树）的变体。Merkle Trie 后面会介绍，这节课就暂时不做过多介绍了。

P2P 网络管理数据：节点通过 UDP 交换 Ping/Pong（在线状态确认），FINDNODE/NEIGHBORS（查询新节点）等控制包，构建和维护连接列表。

加密握手和协议协商：建立 TCP+RLPx 通道时，节点会协商加密协议版本、支持的消息类型（如 Ethereum 子协议）。确保所有数据交换安全且兼容。

十、网络节点数量与去中心化保障机制

先把结论说在前面：

现在公开可见的以太坊节点数量大概在 1～2 万这个量级，活跃验证者接近 100 万个； 节点和验证者在客户端实现、地理位置、网络类型和经济主体上都高度分散，这就是它实际“去中心化”的底气。

1. 以太坊网络有多少节点？

这里必须强调两点：

①没有一个“绝对精确”的官方数字，因为很多节点在 NAT / 防火墙后面，不对外暴露端口，扫描工具看不到；

②不同站点的统计口径也略有差异（只看主网？只算已同步节点？要不要算共识层？）。

以几个主流公开数据源为例（时间点：2025 年 11 月 20 日）：

- **Etherscan Node Tracker**：显示当前检测到的以太坊主网节点总数约 **1.17 万**（“Total 11,7xx nodes found”）。
- **Ethernodes（执行层）**：
  - 执行层客户端（Execution Layer Clients）总数约 **1.7 万个客户端实例**，覆盖 Geth、Nethermind、Besu、Erigon、Reth 等多个实现。
- **Ethernodes（共识层）**：
  - 共识层客户端（Consensus Layer Clients）总数约 **9,500 个实例**，包括 Lighthouse、Prysm、Teku、Nimbus、Lodestar 等。
- **按“只统计已同步节点、排除未同步”的口径**，Bitfly / Ethernodes 数据在 2025-10 的分析文章中给出的数字是：
  - 约 **10,475 个执行层节点**、约 **8,597 个共识层节点**。

另外，**验证者数量**跟“节点数”是两件事（很多验证者是多签/多验证者跑在同一台机器上）：

- 2025 年中曾一度超过 **120 万**活跃验证者；
- 最新新闻显示（ 2025-11 ），由于一波退出，目前活跃验证者略降到**约 99.9 万**，仍然是一个极其庞大的去中心化验证者集合。

所以写书 / 写文章时，可以这样表述：

- **“从公开探测数据看，以太坊主网大约有 1～2 万个可见节点，接近 100 万个活跃验证者，具体数字会随时间波动。”**

2. 这些节点如何保障去中心化？

你原来列的几点方向是对的，我帮你用最新数据强化一下：

2.1 多客户端、多团队实现，降低软件层单点风险

- 执行层：Ethernodes 统计显示，Geth 虽然仍是最大客户端，但份额已经降到 ~37%，Nethermind、Besu、Erigon、Reth 等占据了其余大头，没有单一客户端超过半数。
- 共识层：Lighthouse、Prysm、Teku、Nimbus、Lodestar 等共同分担负载，头部四个客户端合计占比 ~90%，但集中度相比几年前明显下降。

这意味着：**即便某个客户端实现出严重 Bug，也不会“一锅端掉整个网络”。**

2.2 全球范围的地理分布，避免“一个国家说了算”

- Ethernodes 和 Bitfly 的分析显示：执行层与共识层节点在 **美国、德国**占比最高，但在欧洲其他国家、亚洲（包括中国大陆 / 香港 / 台湾）、南美、非洲等地也有广泛分布，字面意义上“全球开花”。

这让单一国家或地区的监管 / 断网事件，很难让整个网络停摆。

2.3 自建节点 + 云托管并存，减少基础设施中心化

最新的 Ethernodes 统计与 2025-10 的深度分析里有两个关键信息：

- 执行层节点中，大约 **45% 左右是自建 / 家宽节点**，约 **49–50% 托管在数据中心 / 云服务商**；
- 共识层因为更强调稳定性，托管比例高一些（大约 58–61%），但自建节点也占到三分之一以上。
- 对云服务集中度的最新估计：
- 在“托管节点”之中，约 **35% 左右托管在 AWS 上**；换算成全部节点，大约 **20% 上下**的执行层节点在 AWS，远低于几年前“半个以太坊都在 AWS 上”的老梗。

这说明：

- 以太坊的确**存在云集中化风险**（最近那次 AWS 故障就当了一次“压力测试”）；
- 但趋势是在向 **“云 + 自建混合、多云部署”** 方向演进，而不是完全依赖某一家云厂商。

2.4 大量分散的验证者，提供经济层面的去中心化

- 近 **100 万个验证者** 分布在全球，使用不同的客户端和运营方案（自托管、质押服务、池子等），任何单一机构很难控制超过 1/3 的有效质押。
- PoS 的 **slashing / inactivity leak** 机制，会惩罚作恶或长时间离线的验证者，从经济上鼓励多方、独立、稳定地参与共识。

换句话说，**共识权力是按质押 ETH 和节点数量自然分散出去的**，而不是集中在某几家矿池。

2.5 任何人都可以运行节点，完整验证无需许可

- 协议是开源的，客户端实现也是开源的；
- 不需要许可就能运行全节点或轻节点，自己从零同步区块、独立验证每一笔交易。

这保证了一个重要性质：**哪怕所有“官方 RPC / 大公司节点”都宕机，普通人也能自己拉起节点接入网络，继续验证和广播交易。**

给读者提供一个模版：

> 目前公开探测到的以太坊节点，大致在 1–2 万个量级（取决于统计口径），活跃验证者接近 100 万个。这些节点和验证者分布在全球多个国家，运行着由不同团队、不同语言实现的多种客户端，既有家庭宽带的自建节点，也有云服务托管节点。即便某个国家、某家云厂商、某个客户端实现出现问题，其他地区、其他云、其他客户端仍然可以继续出块和验证。
>
> 正是这种在 **软件实现、地理位置、基础设施和经济主体上的高度分散**，让以太坊在现实世界中尽可能接近“无需许可、抗审查、可自我验证”的去中心化网络。

十一、Gossip 协议在节点传输中的作用

Gossip（这个单词翻译过来也叫流言蜚语）协议是一种点对点（P2P）的通信机制，模仿信息在社交网络中传播的方式。在以太坊中，Gossip 协议用于在节点之间高效地传播信息，如交易、区块和状态更新。

Gossip 协议就是以太坊的“八卦广播系统”：

它负责把新的交易、区块和 PoS 投票，以去中心化、容错的方式尽快传遍全网，让所有节点尽量在同一时间看到同一批消息。

Gossip 协议是一种模仿人类“传八卦”的 P2P 通信机制：

每个节点只把新消息随机转发给少数邻居，邻居再继续转发，如此层层扩散，最终在全网“发酵开花”。

在以太坊中，Gossip 是**节点之间传播关键信息的主力通道**，主要承担：

- 执行层：传播**交易、区块**等数据；
- 共识层：传播 **Beacon 区块、attestations（投票）、slashings** 等 PoS 共识消息。

1. 它是如何“传八卦”的？

核心机制可以概括为两步：

1.1 局部随机转发

- 某个节点一旦收到新交易或新区块，
  不会广播给“它认识的所有人”，
  而是只随机选出一小部分邻居转发。

1.2 邻居再继续扩散

- 这些邻居把消息加入“已见缓存”，
- 再用同样的方式继续转发给它们的邻居（排除已经见过这条消息的节点），
- 经过若干轮，消息以近似指数级扩散，覆盖整个网络。

这种设计的结果是：

- 每个节点的负载是**局部且可控**的；
- 但全网仍能在**几秒钟级别完成一次交易 / 区块的扩散**。

你之前的比喻可以保留：

> A 同事听到一个八卦，只告诉 2 个没听过的同事；这两个人再各自告诉另外 2 个……几轮之后，整个办公室都知道了这个消息——
>
> 这就是典型的 Gossip 传播模式。

2. 它在以太坊中具体解决了什么问题？

2.1 高可用 + 容错：节点挂了也不怕

- Gossip 是**去中心化多路径传播**：
  - 即使某些节点宕机、掉线或作恶不转发，
  - 消息仍然可以通过其他邻居路径扩散出去。
- 没有“中央广播站”，自然也就**没有单点故障**。

这让以太坊在面对网络分区、部分地区断网、部分客户端出错时，仍然能维持较高的可用性。

2.2 可扩展到上万节点：每个节点只管“身边几个人”

- 每个节点只需要维护有限数量的邻居连接（几十个级别），
- 每条新消息只需要发送给少数邻居再由他们继续扩散，
- 即使整个网络规模达到**上万节点**，每个节点的带宽和计算压力仍然是可控的。

这就是 Gossip 非常适合大规模区块链网络的原因之一。

2.3 尽量“快而不炸”：缓存 +TTL 过滤重复消息

Gossip 的副作用是：**一定会有重复消息**。

所以实现里会做几件事来防止“被八卦淹没”：

- **缓存最近见过的消息 ID（如 hash）**：
  - 收到已经见过的交易 / 区块，直接丢弃。

- **设置传播 TTL / 跳数限制：**
  - 超过一定转发层数就不再继续扩散，防止在网络里无限循环。

这样就能在“传播足够快”与“不过载”之间做平衡。

2.4 支持共识与同步：不仅是交易，还有投票

在 PoS 架构下，Gossip 不仅负责：

- 新交易（tx）、
- 新区块（execution payload / Beacon blocks），

还负责传播：

- **attestations（验证者的投票）；**
- **aggregated attestations（聚合签名后的投票）；**
- *slashings（举报作恶的证据）**等共识消息。

这些消息也都是依靠 Gossip 在共识层的 P2P 网络中快速扩散，保证：

- 验证者能在 slot 限时内看到绝大部分有效投票；
- 链头选择（fork choice）能够在全网快速收敛；
- 对作恶者的举报可以尽快被全网知晓并执行惩罚。

3. 留一句话（可以直接背诵）

> Gossip 协议相当于以太坊的“去中心化广播系统”： 它让每个节点只需和少数邻居聊天，就能在几秒内把新交易、新区块及 PoS 投票散播到整个网络，从而在没有中心服务器的前提下，实现高可用、高容错、可扩展的全网信息同步。

十二、个人节点的搭建与开发环境配置

简单来说就是：

先选节点类型 → 选硬件&部署环境 → 装好执行/共识客户端 → 同步 → 开 RPC/ETL，用起来。

1. 先选节点类型 & 运行环境

1.1 节点类型怎么选？

节点类型	适合用途	典型磁盘需求（主网，截至 2025）	适合谁
全节点（Full）	dApp 开发、调试、自己用的 RPC	~500–800 GB（Geth/Nethermind snap）	绝大多数开发者
归档节点（Archive）	深度历史查询、回测、审计、浏览器服务	Geth ≈12 TB；Nethermind/Besu ≈12–14 TB；Erigon ≈2.5–3 TB	数据分析团队 / 基础设施服务商
轻节点（Light/轻客户端）	资源受限设备、只验证区块头	极小（几十 MB～数百 MB）	想自己验证但没资源跑全节点的用户

> 轻节点是 更省存储 但功能有限的节点（只看区块头，很多复杂查询要问全节点），不是“更占存储”。

1.2 本地 vs 云

- **本地机 / NUC / 家用小主机**
  - 优点：隐私好、成本长期可控。
  - 缺点：需要自己维护网络、电源、硬件。
- **云服务器 / VPS（AWS、GCP、Hetzner 等）**
  - 优点：带宽和电力稳定，适合 7×24 在线。
  - 缺点：长期租用成本、以及基础设施偏中心化的问题（如果你是“去中心化洁癖”，这一点要考虑）。

硬件下限可以记个小数：

- 官方推荐跑一个主网全节点：
  - 4 核 CPU、16 GB RAM、**2 TB NVMe SSD**、带宽 ≥25 Mbit/s。
- 归档节点：
  - 至少 8 核+、64 GB RAM、**10–12 TB+ 企业级 SSD**（或 2–3 TB NVMe 对 Erigon 这类紧凑客户端）。

2. 安装执行客户端 & 共识客户端

合并（The Merge）之后，一个“完整节点”**必须**同时跑：

- **执行客户端（Execution Client, EL）：**
  - 负责交易处理、EVM 执行、状态数据库、`eth_*` JSON-RPC。
  - 常见：**Geth、Nethermind、Besu、Erigon、Reth**。
- **共识客户端（Consensus Client, CL）：**
  - 负责 PoS 共识、提议/验证区块、finality（终局性）、validator 管理等。
  - 常见：**Prysm、Lighthouse、Teku、Nimbus、Lodestar**。

两者通过 **Engine API**（一组本地 JSON-RPC 接口，如 `engine_newPayload`、`engine_forkchoiceUpdated` 等）对话工作。

> 实战里常见组合：
>
> - Geth + Lighthouse / Prysm
> - Nethermind + Teku
> - Erigon + Lighthouse / Nimbus
>
> 只要一套 EL + 一套 CL 搭配正确，功能上没有本质差别，更多是**性能、资源占用和个人习惯**的问题。

3. 同步区块链：选择合适的 Sync 模式

3.1 执行层（EL）同步模式

以 Geth 为例，有三种模式（其它客户端概念类似）：

①Snap Sync（推荐）

- 默认模式（不加 `-syncmode` 就是它）。
- 流程：先把历史区块/头下载下来，再下载“一个比较新的状态快照”，最后做 state heal。
- 优点：**数小时到一两天**就能把全节点同步到最新（视硬件/带宽而定），安全性足够高，适合绝大多数开发者。

②Full Sync

- `-syncmode full`
- 从创世区块开始把所有交易完整执行一遍，重建当前状态；只保留最近 ~128 个块的状态，其余状态按需重算。
- 优点：验证最严格；缺点：耗时长很多，通常只在特定研究 / 审计场景才需要。

③Archive Sync

- 在 Full 的基础上**不裁剪历史状态**，保留所有高度的完整状态树。
- 用于构建归档节点（Archive Node），提供“任何区块高度立即可查”的历史状态查询能力。

3.2 共识层（CL）同步模式

大部分共识客户端都支持某种形式的 **checkpoint sync / weak subjectivity sync**：

- 从一个可信的“近期 checkpoint”开始同步 Beacon 链，而不是从创世开始；
- 这样共识层通常 **数小时级别** 就能追上头部，而不是几天到几周。

实战建议：

- **开发用全节点**：执行层用 Snap Sync，共识层开 checkpoint sync，就够了；
- **做数据分析 / 审计**且要频繁查历史状态：执行层跑 Archive，同步周期按 TB 级存储和网络情况预留几天甚至更久。

4. 开启 JSON-RPC，变成“自己的以太坊后端”

当节点追上链头后，就可以把它当成你 dApp 的“自托管后端”来用。

4.1 在执行客户端上启用 HTTP / WebSocket RPC

以 Geth 示例（生产环境请加上鉴权和 IP 限制）：

```
geth \
  --http \
  --http.addr 127.0.0.1 \
  --http.port 8545 \
  --http.api eth,net,web3 \
  --ws \
  --ws.addr 127.0.0.1 \
  --ws.port 8546 \
  --ws.api eth,net,web3
```

然后前端 / 脚本就可以用：

- ethers.js / web3.js / viem
- `ETH_RPC_URL=http://127.0.0.1:8545`

来直接连你自己的节点。

> 安全提醒：不要在公网裸暴露未鉴权的 RPC，尤其是有 personal, admin 等危险 API 的配置；最安全做法是只在内网/localhost 暴露，或者放在反向代理后面做认证和限流。

4.2 开发常见用法

- 部署 / 调试智能合约（Hardhat/Foundry 指向本地 RPC）；
- 做 fork mainnet 测试（mainnet forking 背后就是用一个全节点当数据源）；
- 查询日志、事件、trace 做简单分析。

5. 做数据分析：从节点到数据库

如果你是为了**链上数据分析 / 报表 / 量化研究**来搭节点，建议这样玩：

5.1 用 Ethereum ETL 做 ETL 管线

- Ethereum ETL 是最常用的开源 ETL 项目，可以：
  - 从接入的节点导出 blocks、transactions、logs、token_transfers、traces 等；
  - 输出为 CSV / Parquet 或直接写入 Postgres / BigQuery。

典型方案：

①全/归档节点提供可靠 RPC；

②`ethereum-etl` 把历史数据一次性导出，

③后续用 `streaming` 或 Airflow DAG 做“实时增量同步”；

④数据落到 Postgres / BigQuery，给 BI、Notebook、分析脚本使用。

5.2 利用客户端自带导出能力

像 **Erigon** 这类客户端本身对查询和索引做了大量优化：

- 更紧凑的数据库布局，归档节点体积只有 ~1.6–3 TB；
- 内置一些 debug / trace / dump 功能，适合高频查询。

如果你的主要目标是**做分析而不是跑 dApp**，可以优先考虑用 Erigon 当执行客户端。

5.3 偷懒路线：直接用公共数据集

如果暂时只想分析，不一定非得自己跑节点：

- 可以直接用 Google BigQuery 上的 Ethereum Public Dataset；
- 或者 Flipside、Dune 等数据平台。

6. 运维与优化：别忘了这几件事

6.1 防火墙

- 只开放必要端口：P2P 默认 30303（TCP/UDP）、HTTP/WS RPC 端口只在内网可见。

6.2 监控与日志

- Prometheus + Grafana 监控 CPU、磁盘、peer 数量、同步高度；
- 关注客户端 release note，按时升级应对硬分叉 / 漏洞修复。

6.3 共识层 checkpoint sync 配置

- 选一个可信 checkpoint URL；
- 可以显著缩短 CL 同步时间（从几天 → 几小时）。

6.4 磁盘与备份

- 全节点：至少 1–2 TB NVMe，预留增长空间；

- 归档：建议 RAID0＋ 定期快照 / 备份，否则硬盘一挂，重同步是地狱级任务。

7. 给大家一份“快速选型小抄”

你的需求	建议配置
只想本地开发 / 调试 dApp	全节点（Full Node），EL 用 Geth/Nethermind Snap Sync＋ 任一 CL，2 TB NVMe，16 GB RAM；开 HTTP RPC 给 Hardhat/Foundry 用。
要做指标、行为分析、回测策略	归档节点（Archive Node），优先 Erigon（≈3 TB），32–64 GB RAM，配合 Ethereum ETL＋Postgres/BigQuery；写 ETL/Notebook 做分析。
想先玩玩，不想一开始砸太多硬件	先用托管节点服务（Infura、Alchemy、QuickNode、GetBlock 等）的免费/开发套餐，RPC 指过去，等需求稳定再迁移到自建节点。

十三、拓展：不跑节点的数据分析方法

> 不跑节点 ≠ 做不了链上分析。
>
> 你可以“借别人的节点 ＋ 别人的 ETL”，站在公共数据平台和 API 的肩膀上搞分析。

我按“从零门槛 → 轻量 API → 深度分析”给你分几类方法，你写文章的时候可以配一张对比表。

1. 用公共分析平台：Dune / Flipside 等（零节点、零运维）

1.1 Dune：写 SQL 就能查全链历史

Dune 是目前最主流的链上数据分析平台之一：

- 已经把 Ethereum 等 100+ 链的原始数据 **预解析进关系表**，你直接用 SQL 查。
- 支持把查询结果做成图表、仪表盘，公开或私有分享。

典型用法：

- 查某个协议的 TVL、活跃地址、手续费收入；
- 按天/周做 Cohort、留存、鲸鱼分析；
- 做 NFT 铸造、二级交易、MEV 模式分析等。

优点：

- 完全 **不需要节点、不需要 ETL、不需要搞存储**；
- 有大量社区现成 Dashboard 可以 Fork 改。

缺点：

- 免费额度有并发和资源限制；
- 表结构由平台定义，某些非常底层的 trace/状态细节可能没有直接暴露。

写作建议：这一类可以叫 “SQL 即服务的链上分析平台”，代表：Dune。

1.2 Flipside：多链免费 SQL + Studio 环境

Flipside 也是一个典型的“预处理好链上数据再开放 SQL 查询”的平台：

- 覆盖 20+ 条链（包含 Ethereum），提供标准化数据模型。
- 自带一个类似 Jupyter Notebook 的 **Studio**，可以边写 SQL 边看结果、画图。

典型用法：

- 直接在浏览器里查询地址余额变化、合约交互记录；
- 分析 DeFi 协议、NFT 项目、桥接行为等。

适合人群：**偏数据科学 / 分析师**，喜欢在网页里直接写 SQL+看图，不想自己运维数据库。

2. 用公共 BigQuery 数据集：当作“已经建好的数据仓库”

如果你习惯用 SQL + 自己的 BI 工具，**Google BigQuery 公共区块链数据集**是一条很稳的路子：

- Google 官方把 Ethereum 整条链的数据 **每天同步进 BigQuery 公共数据集**，包括区块、交易、日志等表结构。
- 你在 BigQuery 里直接用 SQL 查，甚至可以联表其它链或自有业务表。

典型参考：

- 官方博客：介绍 Ethereum BigQuery public dataset 的结构和用法。
- 社区文章：指出“如果想在不跑归档节点的情况下分析 Ethereum 交易和 traces，可以直接用 BigQuery 公共数据集”。
- 甚至有文章专门教你：用 **BigQuery Sandbox** + 公共数据集，免费做钱包画像、DEX 体量分析、NFT mint 追踪等。

优点：

- 不跑节点，不写 ETL，**直接就有“企业级链上数据仓库”**；
- 能和你自己的业务数据（用户、订单、日志）做 Join；
- 可以接 Looker Studio / Power BI / Data Studio 做可视化。

适合人群：

- 数据工程 / 数分团队，已经在公司内用 GCP / SQL 做数仓；
- 想把“链上行为”直接塞进自己的 BI 体系里。

3. 用区块浏览器 API：Etherscan 等（脚本级、轻量查询）

如果你只是想：

- 拉某个地址的交易历史；
- 查某个合约的事件日志、ABI；
- 获取基础的区块、gas、token 数据；

那 **Etherscan API** 这一类浏览器 API 就够了：

- Etherscan 本身是以太坊区块浏览器，也提供标准化 API：账户、合约、交易、日志、gas、token 等各类 endpoint。
- 官方文档和社区教程很多，甚至有“如何用 Etherscan API + Python 做入门链上数据分析”的文章。

优点：

- 门槛低：**注册个 API Key 就能用**；
- 很适合写小脚本 / Notebook 做特定任务：监控某地址、同步指定事件、做简单统计。

缺点：

- 免费版有 **QPS / 日调用次数限制**，深度分析得付费或自己缓存；
- 数据结构是“按接口给”的，不像 Dune/BigQuery 那样已整理成完整模型。

这条路线更适合作为：

> “我还不想上大平台，只想先拉些链上数据，用 Python / Pandas 先玩一玩”的入门路径。

4. 用数据提供商 / Indexing API：Covalent、Bitquery 等

如果你需要更结构化、跨链的数据，又不想自己维护 ETL 和数仓，可以考虑：

- **Covalent、Bitquery、Alchemy Transfers / Subgraphs、Moralis、QuickNode Streams** 这类 **Web3 数据 API 服务**（这里用 Covalent/Bitquery 举例）。
- 它们会在自己那边跑全节点 + 做索引，然后提供：

- 地址投资组合、Token 持仓、NFT 列表；
- DEX 交易、历史 K 线、链上资金流；
- 原始 logs / traces 的高阶封装。

比如：

- 有公司实践方案：用 BigQuery 拉全历史数据，再用 Infura 做实时增量，从而在“不跑节点”的前提下构建完整分析平台。

这类服务的共同特点：

- 按调用量 / 功能收费，适合**小团队快速起步**；
- 某种意义上，你是在“按次租用别人已经搭好的归档节点 + ETL + 索引”。

5. 混合打法：历史数据用 BigQuery / Dune，实时数据用 RPC 服务

很多团队最终会走到一种“混搭”模式：

①历史数据（过去几年的全链数据）

- 用 **BigQuery 公共数据集 / Dune / Flipside** 做一次性冷数据拉取和分析；

②实时数据（最新的几分钟～几小时）

- 使用 **托管 RPC 服务**（Infura、Alchemy、QuickNode 等），监听新块事件、订阅日志，做实时事件驱动；

③在自家数据库里做拼接与落地

- 历史大表来自 BigQuery / Dune 导出；
- 最新增量来自 RPC 或数据 API；
- 最终统一存入自家 Postgres / warehouse 里，做业务 BI。

这样，你既：

- 避免了自己同步和维护 TB 级归档节点；

- 又能在自己的数据库里拥有“几乎全历史 + 小时级延迟”的完整链上视图。

速记表：

不跑节点搞数据分析的几种路线：

①SQL 平台：Dune / Flipside

- 优点：零运维，直接写 SQL + 画图
- 适合：分析师、研究员、想快速产出 Dashboard

②BigQuery 公共数据集

- 优点：企业级数仓，能跟自家业务数据 Join
- 适合：已经有 GCP / BI 体系的团队

②区块浏览器 API（Etherscan 等）

- 优点：脚本接入简单，免费额度够学习与小项目
- 适合：Python / JS 轻量分析、监控脚本

③Web3 数据 API / Indexing 服务

- 优点：拿到的是“已经整理好的业务视图”
- 适合：小团队做产品、需要多链、但不想运维

④混合方案：历史用 BigQuery/Dune，实时用托管 RPC

- 优点：接近“自己有归档节点”的能力，但不自己跑机器
- 适合：认真做数据产品，却又不想从硬件和节点运维起家的团队

以太坊的网络不是冰冷的结构，而是无数节点共同跳动的“分布式心脏”。它们同步、传播、验证、协作，让整个系统得以全天候运转。愿你在深入理解这些基础后，能够更大胆地探索、构建与实践，为这个开放网络写下属于自己的代码与贡献。

——@kelvin-秦默

第三章

账户类型与结构

本章目标：搞懂 EOA 与合约账户差异与互动方式。

正式开始之前先做一些补充知识：

CREATE（传统创建）：合约地址由 *创建者地址 + 创建者的 nonce* 决定，地址不可在部署前精确预测（除非知道 nonce）。公式上是 keccak256(RLP([sender, nonce])) 的后 20 字节。

CREATE2（EIP-1014 引入）：合约地址由 *创建者地址 + salt + init_code*（init_code 的哈希）决定，地址在部署前可以被**精确预计算**：keccak256(0xff ++ deployer ++ salt ++ keccak256(init_code)) 的后 20 字节（0xff 是定界前缀）。CREATE2 于 Constantinople（EIP-1014）引入。

作用差异要点：CREATE2 的最大价值是**可预测地址（deterministic deployment）与跨链/跨环境一致性**（相同输入在任意 EVM 链上会生成相同地址），常用于工厂合约、代理/升级模式与“counterfactual contracts（反事实合约）”。

一、EOA 的定义与控制方式

大家可以把这一节记成一句话：

> EOA = 由“私钥”直接控制的钱包账户，能发交易、收资金、调合约，但自己不带代码逻辑。控制私钥的人 = 控制这个账户的一切。

1. EOA 是什么？和合约账户有什么区别？

1.1 EOA（Externally Owned Account，外部拥有账户）

- 由**公私钥对**控制，用户（人 / 机构）持有私钥；
- 可以发起交易（转账、调用合约）、持有 ETH 和各种代币；
- 账号本身**没有代码**，不会自动执行任何逻辑。

以太坊里只有两类账户：

①EOA：

- 有私钥
- 能“主动发起交易”
- 没有合约代码

②合约账户（Contract Account）：

- 无私钥，由部署时的代码“锁死行为”
- 不能主动发起交易，只能“被调用”后按合约逻辑执行
- 有代码和存储（storage）

用一句对比记忆：

> EOA 像“人 + 银行卡”，合约账户像“写死规则的自动售货机”。

2. 一个 EOA 在状态里到底长什么样？

在以太坊世界状态里，每个账户（不论是 EOA 还是合约账户）都有固定字段：

- **地址（address）**
  - 从公钥派生，20 字节，以 0x 开头的 40 个十六进制字符。
- **nonce（交易计数器）**
  - 对于 EOA：表示**这个账户已经发送过几笔交易**。
  - 一笔交易的 nonce 必须等于当前账户 nonce，打包成功后 nonce+1。
  - 作用：**防止重放攻击 & 保证交易有序** —— 同一个 nonce 只允许执行一次。

- **balance（余额）**
  - 账户持有的 ETH 数量，以 wei 计（1 ETH = 10^{18} wei）。
- **code / storage**
  - 对于 EOA：code 为空，没有合约代码；storage 也为空。
  - 对于合约账户：code 保存 EVM 字节码，storage 保存合约状态。

所以：**EOA 只是一对“钥匙 + 几个数字字段（nonce / balance）”，本身不带逻辑。**

3. 我是怎么“控制”一个 EOA 的？

3.1 私钥签名：真正的控制权

从本质上讲：

持有私钥的人，才是这个 EOA 的真正“主人”。

- 每个 EOA 对应一个**私钥 / 公钥对**；
- 发起交易时，用**私钥**对交易数据签名（包含 from / to / value / nonce / gas …）；
- 网络节点通过公钥验证签名，确认：
  - 这个交易确实由这个地址发出；
  - 交易内容未被篡改。

一旦私钥泄露：

- 对方可以任意发起转账、调用合约、清空你的余额；
- 区块链无法“冻结”或“找回”，因为系统只认“谁能签名”。

3.2 助记词（BIP-39）和派生路径（BIP-44）

实际钱包里，你很少直接看到“私钥字符串”，而是看到一串**助记词**（12/24 个英文单词）：

- **BIP-39** 定义了“助记词 → 种子 → 一棵 HD 钱包密钥树”的标准：
  - 人类容易记的词 ⇋ 大随机数 ⇋ 一堆私钥 / 地址。
- **BIP-44** 在此基础上定义了**多币种、多账户、多地址的分层结构**，约定了常见派生路径（如 `m/44'/60'/0'/0/0` 对应以太坊第一个地址）。

所以：

> 那 12/24 个单词 = 一把“万能总钥匙”，能派生出你钱包里所有 EOA 地址。这也是为什么安全提示永远在强调：

- 不要截图 / 云盘存助记词；
- 不要在不可信网页 / App 里输入助记词。

3.3 Nonce：阻止“把你签过的交易拿出来复读”

Ethereum 官方文档直接说明：

> nonce 是针对每个 EOA 的递增计数器，只允许每个 nonce 对应一笔交易，防止攻击者把已经签过的交易反复广播执行。

也就是说：

- 即使别人截获了你**已成功上链**的交易数据；
- 在你账户 nonce 已经前进的情况下，再次广播这笔交易会被直接拒绝；
- 所以不能简单地“复读你签过的数据”来做链上重放。

（跨链 / 分叉场景会有专门的 replay 问题，那属于更进阶的话题，可以在别处展开。）

3.4 Gas：想让 EOA 动起来，必须给它加油

EOA 想发交易或调合约，必须支付 **gas 费**，而 gas 费用的是 ETH（或 L2 的原生代币）：

- 账户余额不足以覆盖 `gas_limit × gas_price` 的话，交易会被节点拒绝；

- 这保证任何 EOA 想对链发动“垃圾攻击”，都要付真金白银的成本。

4. 实战视角：开发者 & 普通用户要注意什么？

4.1 对开发者

- 你在前端看到的钱包地址（MetaMask 等），本质就是一个 EOA；
- dApp 与 EOA 的交互，就是通过 `eth_sendTransaction` / `eth_signTypedData` 让用户用私钥签名；
- 合约里调用 `msg.sender` 时，如果调用方是 EOA，那就是“真实用户”；如果是合约，则是“合约地址”。

4.2 对普通用户

- **控制私钥 / 助记词的人，就是账户真正的主人；**
- 最安全的做法是使用 **硬件钱包** 管理私钥，把助记词线下保存；
- 任何要求你“输入助记词到网页”的项目，99% 当诈骗处理。

4.3 类比：银行账户

- **EOA 地址**：像**银行卡号**，别人可以转钱给你；
- **私钥**：像**银行卡 + 密码 +U 盾**合体，只要别人拿到了，就能把钱全部转走；
- **助记词**：像一张能复刻所有银行卡的“主密码纸”，丢了就等于把所有卡的控制权送人；
- **nonce**：像银行给每笔转账排的流水号，防止同一笔指令被执行两次。

可以背诵的一句话：

> EOA＝“由私钥直接控制的以太坊账户”。 区块链不认识你是谁，只认你能不能拿出对应的私钥签名。

下面是一个表格，方便大家查看与理解：

属　性	内　容
私钥与公钥	EOA 是由一个密码学 key pair（私钥 + 公钥）控制的。私钥用于签名，公钥经过哈希等计算生成地址。
地址	公钥派生出地址。以太坊地址通常是 20 字节 /40 个十六进制字符 +"0x" 前缀。
Nonce（交易计数器）	跟踪这个 EOA 曾经发送过多少交易，用来防止重放攻击。
Balance（余额）	EOA 可以持有 ETH 或代币余额。区块链状态里记录这个账户在世界状态树（World State）中的余额。
Code / Storage	对于 EOA，code 部分是空的（没有智能合约代码）。也没有“storage root”（存储状态），这些是合约账户（Contract Account）才有的。

二、合约账户的概念与创建流程

一句话把这节定调：

> 合约账户 =“带代码的账户”，不靠私钥，而是靠部署在链上的 EVM 字节码决定它能做什么；
>
> 它是通过 **“创建合约交易 + 部署字节码 + CREATE / CREATE2 地址规则”** 被创建出来的。

1. 合约账户是什么？

在以太坊里有两种账户：

1.1 EOA（Externally Owned Account，外部拥有账户）

- 有私钥，用户通过钱包控制；
- 不能存代码，只能签名、发交易。

1.2 合约账户（Contract Account / Smart Contract Account）

- **没有私钥**，它的行为完全由**账户里的代码（EVM bytecode）**决定；
- 有自己的**地址和 ETH 余额**，也可以持有 ERC-20 / NFT；

- 有 **code**（合约逻辑）和 **storage**（状态存储），可以读写状态、转账、调用其他合约；
- **不能主动发起“外部交易”**，只能在被 EOA 或其他合约调用时执行（内部消息调用）。

从世界状态的角度，合约账户和 EOA 一样，状态里都有 4 个字段：

- nonce：
  - 对 EOA：已发送交易数量；
  - 对合约账户：**曾经用 CREATE / CREATE2 创建过多少个合约**。
- balance：账户持有的 ETH 数量。
- storageRoot：该合约存储 trie 的根哈希（合约状态的入口）。
- codeHash：EVM 字节码的哈希，用于在底层存储中定位合约代码。

2. 合约账户是如何被创建的？

大多数情况下，一个合约账户的诞生会经历下面几步：

1. 编写并编译合约

- 开发者使用 **Solidity / Vyper 等高级语言**编写智能合约；
- 编译器把源码编译成 **EVM 字节码**：
  - **init code（初始化/构造代码）**：部署时执行，类似“构造函数”；
  - **runtime code（运行时代码）**：部署完成后真正留在链上、对外执行的部分。

EOA 或合约发起“创建合约交易”

- 一笔**创建合约的交易**特点是：
  - to 字段为空（null 地址）；
  - data 字段写入 **init code（+ 构造函数参数编码）**；

- 这笔交易一定需要支付 gas：代码存链 + 执行构造逻辑都要花费 gas。

这个“发起人”可以是：

- 一个 **EOA**（你在 Remix/Hardhat 里部署合约时就是这样）；
- 一个已经在链上的**合约（工厂合约 factory）**，通过 EVM 的 `CREATE / CREATE2` 指令再部署一个新合约。

3. EVM 执行 init code，写入 code & 初始 storage

当矿工/验证者在 EVM 中执行这笔“创建合约交易”时，会做几件事：

①看到 `to == null`，识别这是**创建合约交易**；

②运行 `data` 里的 **init code**；

③init code 的最后会 `RETURN` 一段字节码——这就是 **runtime code**；

④EVM 把这段返回值作为**新合约账户的 code**，写入世界状态；

⑤同时，根据构造函数执行结果初始化 storage（状态变量）；

⑥若 gas 不足，或 init code 执行失败，整个创建会回退，**不会生成合约账户**。

最终，新合约账户出现在状态树中，拥有：

- 一个新地址；
- 一段固定的 runtime bytecode；
- 初始存储状态（所有状态变量的初始值）。

3. 合约地址是如何计算出来的？

合约地址**不是随机的**，而是**确定性可计算**的，这一点对“预先计算地址、钱包工厂、Create2 部署”等很重要。

3.1 普通 CREATE：sender + nonce 决定地址

当使用 **CREATE**（最常见方式）部署合约时，新合约地址由：

> address = last_20_bytes(keccak256(rlp([sender, nonce])))

其中：

- sender：发起合约创建的账户地址（EOA 或合约）；
- nonce：这个账户**在创建时的交易/合约创建计数**。

换句话说：**同一个 deployer 地址 + 同一个 nonce → 合约地址唯一确定**。

3.2 CREATE2：0xff + deployer + salt + keccak(init_code)

EIP-1014 引入的 **CREATE2** 允许“提前预测合约地址”，计算公式为：

```
address = last_20_bytes(
   keccak256( 0xff ++ deployer ++ salt ++ keccak256(init_code) )
)
```

- deployer：执行 CREATE2 的合约地址；
- salt：32 字节的盐，由开发者自定义；
- init_code：完整的部署字节码（包含构造逻辑）。

特点：

- **同样的 deployer + salt + init_code → 永远得到同一个地址**；
- 和 nonce 无关，因此可以跨链用相同的组合在多条 EVM 链上得到同一地址（只要规则一致）。

这让很多“钱包工厂”、“合约钱包账户抽象”、“预留地址”玩法成为可能。

4. 合约账户的能力与限制

4.1 能力

- 持有 ETH / Token；
- 内部存 code & storage，可以实现任意复杂逻辑：DEX、NFT、DAO 等；
- 可以在执行中给别的地址转账、调用其他合约、甚至使用 CREATE / CREATE2 再创建新合约。

4.2 限制

- **没有私钥**，因此不能像 EOA 一样“签名并主动发起外部交易”；
  - 所有行为都必须由外部交易触发（EOA 调用它，或另一个合约在执行中调用它）。
- 部署合约必须付较高 gas：
  - 因为需要把代码**永久存储在链上**，这在 EVM 里是昂贵操作。
- 部署后的代码一般不可变（除非自己写了代理 / 升级模式），只有 storage 可以随调用改变。

4.3 类比：自动售货机 vs 银行卡

类比把 EOA 和合约账户串起来：

- **EOA：银行卡账号**
  - 你（持私钥的人）决定什么时候转账，
  - 银行系统只是帮你记账。
- **合约账户：上链的自动售货机 / 自动柜台**
  - 你事先写好规则，把机器放在那里（部署合约）；
  - 任何人往里面投钱/按按钮（发交易调用合约），
  - 售货机会按照写死的程序自动执行：
    - 吐货、退钱、转账、更新库存（storage）。

售货机**没有“密码”可以让你随意改规则**，你只能在设计规则的时候把逻辑想好；这就是智慧合约的特性：**一旦部署，就由代码“接管账户”，而不是由人手动控制。**

三、以太坊地址“0x”开头的由来

先来一句话先掐死这题：

以太坊地址本质是一串 20 字节（160 bit）的二进制数据，写成字符串时通常用十六进制展示，于是按照编程界传统加了个前缀 0x，用来标明：“注意，这是 hex 不是普通数字。”

1. 地址本体是 160 bit，只是“长得像 0x……”

在协议层，以太坊地址**只是一个 160 bit 的值**：

- Yellow Paper 和官方教程都把地址定义为 160 位（2^{160}）空间中的一个值。
- 展示给人看的时候，一般用十六进制字符串表示：
  - 20 字节 = 160 bit；
  - 每个 16 进制字符表示 4 bit；
  - 所以刚好是 **40 个 hex 字符**。

于是就有了我们熟悉的格式：

0x + 40 个十六进制字符（0–9, a–f） → 总长度 42。

2. 为什么偏偏是 0x？

这是**编程界的老传统**：

- 在 C / C++ / JavaScript 这类语言里，0x 一直被用来表示“**后面这串是十六进制**”：如 0xff、0xdeadbeef。
- 以太坊沿用了这一习惯：
  - Yellow Paper 里直接写“地址用十六进制表示，**通常会加上 0x 前缀来显式标记**”。
- 钱包、区块浏览器、文档统一写成 0x...，久而久之变成**行业约定俗成的格式**。

这有几个好处：

①一眼能看出是地址 / hex，而不是某个普通 ID 或十进制数字；

②在文档、日志、代码里查找、正则匹配都更方便；

③避免和其他格式（比如 Base58、十进制）混淆。

3. 0x 算地址的一部分吗？必须写吗？

严格讲：

- **链上的“地址值”本身不包含 0x**，它只是一串 160 bit 的二进制；
- 0x 是**文本表达**时的前缀 —— 类似“标点/标签”，不是数据的一部分。

在很多库/工具里你会看到类似的要求：

- 字符串地址格式：(0x)?[0-9a-fA-F]{40} —— 也就是说，**带不带 0x 都能识别**，只是推荐带上。

所以：

- 钱包、浏览器 UI：几乎都会显示成 0x...；
- 某些 API / 数据库字段：允许不带 0x 的纯 40 位 hex（自己再加也行）。

4. 和 checksum / 大小写有什么关系？

顺带解释 EIP-55：

- EIP-55 定义了“**混合大小写的 checksum 地址**”，用大小写模式来做校验，减少手动输入地址时的错误。
- checksum 地址仍然是：0x + 40 个 hex 字符，只是其中部分字母会变成大写。
- 区块链在执行层面对大小写**不敏感**；checksum 只是“客户端 / UI”层面的安全增强。

5. 类比方便理解

一个类比：

> 可以把 0x 想成“货币符号”：
>
> - $100 里的 $ 告诉你：这是美元金额；
> - 0x1234... 里的 0x 告诉你：这是一个**十六进制地址**。
>
> 真正的钱是数字本身，$ 只是前面的标记；
>
> 真正地址是那 20 字节/40 个 hex 字符，0x 只是告诉你“这是一串 hex 地址”。

一句话（可以直接背诵）：

> 以太坊地址加 0x，不是因为协议强制，而是为了“让人和程序一眼识别：这是 16 进制地址”。

四、EOA 与合约账户的控制方式对比

可以把它理解成：

> EOA =“由私钥直接控制的人类钱包”；
>
> 合约账户 =“由代码控制的自动售货机”。

区别维度	EOA (Externally Owned Account，外部拥有账户)	合约账户 （Contract Account / Smart Contract Account）
控制主体	**私钥 / 助记词** 持有者控制账户；谁拿到私钥，谁就能发交易。	由**部署在账户里的代码**控制行为；没有私钥，规则写死在智能合约逻辑中。
行为触发方式	可以**主动发起交易**：转账、调用合约，只要签名 + 有足够 gas。	**不能主动发起外部交易**：只能在被 EOA 或其他合约调用时执行（内部 message call）。
是否存在私钥	有私钥 + 公钥，对应地址；“控制权 = 私钥所有权”。	**没有私钥**；任何“权限控制”都由合约内部的 require(msg.sender == …)、角色管理、多签等代码实现。
功能灵活性	自身功能很简单：签名 → 发交易；要实现复杂逻辑一般需要配合合约。	逻辑高度可编程：可实现多签、时间锁、访问控制、代币发行、治理、金库、机器人逻辑等。
风险与信任点	主要风险是**私钥管理**：丢失=永久失去控制；泄露=资产被直接转走。	主要风险在于**代码漏洞 / 逻辑错误 / 权限设计不当**；一旦部署，代码通常不可改，只能通过预先设计的升级机制（proxy 等）调整。
创建成本与方式	生成 EOA 只是本地生成一对 key pair，**不需要上链、不花 gas**。	部署合约账户必须发一笔“创建合约交易”，把 bytecode 写入链上，**需要支付不少 gas**。
行为/权限的可修改性	EOA 自身没有代码，行为模式基本固定（签名+发交易）；想要更复杂控制（多签等）通常借助合约钱包。	合约代码一旦部署一般**不可变**；只能通过合约内预留的升级逻辑、代理模式或 selfdestruct 等机制间接“修改”行为；storage 状态可以通过函数调用改变。

这里提到了多签和时间锁，来做个补充介绍。

1. 多签（Multisig）详解

定义（英文全称）

Multisignature (Multisig Wallet)：一种需要多个签名/批准才能执行交易的账户或智能合约钱包。

工作原理（核心）

常见模型为 **M-of-N**：比如 2-of-3 表示 3 个 owner 中至少 2 个签名同意才执行。

技术实现通常是部署一个智能合约（wallet contract）保存资产，并通过合约内函数管理提案、签名集合与执行（Gnosis Safe 是主流实现）。用户发起“交易提案”，其他 owners 在界面或离线签名后，达到阈值合约执行交易。

现实例子

Gnosis Safe（前称 Gnosis Multisig）：被广泛用于 DAO、多方托管、项目国库管理。界面友好、可安装模块。

优点

把单点密钥泄露风险降到最低（需要同时攻破多把钥匙才能拿走资产）。

适合团队决策流程（例如多人审批支出）。

2. 时间锁（Timelock）详解

定义（英文全称）

Timelock / Time lock (Timelock Controller / Timelock Contract)：在智能合约或治理流程中，对某些敏感操作设置 **预定延时（delay）**，在延时到期后操作才可被执行。常见实现：OpenZeppelin 的 TimelockController。

工作原理（核心）

管理角色（proposer、executor、admin）提交一个“计划好的”交易到 timelock 合约。提交后必须经过设定的 delay 时间窗口（例如 2 天、7 天、或 48 小时）才能由 executor 发起实际执行；在这段时间，社区/用户可以审查、挑战或撤出资金。

现实例子与用途

治理（DAO/Protocol upgrades）：在链上治理通过后，升级合约或变更参数先放入 timelock，给用户时间进行反应（例如退出、投票）——这是多数 DeFi 项目的做法（Compound、Aave 等均采用 timelock 或类似流程）。（相关文章与教程说明 timelock 的治理作用）。

资金解锁 / 归属期（vesting）：用于代币团队/投资人的归属锁定期（vesting）与分期释放。

优点

透明：任何人可看到将会发生的操作与执行时间。

缓冲/缓解风险：给用户/审计方时间发现恶意升级或漏洞并采取行动（撤资、告警、软分叉/其他对策）。

3. 限制与风险

延迟影响响应速度：在紧急修复漏洞时，延迟会成为负担——通常会配合“guardian / emergency pause” 机制处理。

依赖 timelock 本身的安全性：timelock 合约要托管重要权限（ownership / admin），必须高度审计。若 timelock 被攻破，攻击者能在延时后执行恶意操作。

控制方式的技术机制差异

私钥签名 VS 合约逻辑判断 EOA 控制事务的权限是通过数字签名（在交易

里签名字段）来实现的。合约账户的“权限”则体现在合约内部函数中，包括 msg.sender, require(...) 等条件判断。合约可以检查是谁在调用、value 是否足够、状态是否允许等。

4. 代码存在性（Code）

可以通过 RPC 方法 eth_getCode(address) 判断某地址是否为合约账户：如果返回的代码非空，就说明有智能合约存在，是合约账户；否则为 EOA。

Nonce 行为 EOA 的每次交易发送会使其 nonce 增加，用来防止重放攻击；合约账户不是通过 nonce 来发起交易，对其行为的控制在于它被谁以何种“transaction / message call”触发。

5. 类比帮助理解

把 EOA 和合约账户之间控制方式的区别，比作现实中“人 vs 机器自动柜员机 /自动售货机”的区别：

EOA（人）：你有钥匙（私钥）。你想转钱、打电话、去买东西，你可以自主行动。你能判断什么时候行动，怎样行动。

合约账户（机器 /自动售货机 /柜员机）：机器本身有程序规则（代码），当你把钱投入 /按下按钮（外部触发）后，它按照预设规则（程序）做动作。你没有钥匙控制机器本身后台逻辑，只能通过前面接口触发它。

五、可主动发起交易的账户类型

能够主动发起交易的账户只有 EOA（外部拥有账户，Externally Owned Account），这是因为：

EOA 由私钥控制，用户通过签名发送交易请求并广播至网络，因此它们能够主动发起交易——包括转账、调用智能合约等行为。根据以太坊当前协议设计，

每一笔链上 transaction 的起点，依然必须是一个由私钥签名的账户，也就是 EOA。

合约账户（Contract Account）由智能合约代码控制，没有私钥，无法主动创建交易。它们只能在收到 EOA 发出的调用时被动执行逻辑，因此不能主动发起交易——这一点在今天依然成立：**合约可以执行 call / delegatecall / create 等内部消息，但不能凭空生成新的顶层交易（top-level transaction）。**

1. 图表：EOA vs 合约账户的“主动发起交易”能力

机　制	为什么 EOA 能发起交易	为什么合约账户不能主动发起（在标准场景下）
私钥控制	EOA 拥有私钥，可以签名交易。签名是发起交易不可或缺的一步。	合约账户本身没有私钥，因此不能进行签名，也就无法“发起”交易。
发起者角色 (transaction origin)	在以太坊中，transaction 一般由 EOA 发起，并提交给网络（mem-pool），然后被矿工 / Validator 包含进区块。	合约账户只能在收到一个交易或者被调用时响应；它不会自己发起一个新的 transaction。
gas /手续费付费机制	发起交易（EOA 发起）时该 EOA 要付 gas（以 ETH 支付）。所以 EOA 必须持有 ETH。	合约账户没有私钥，也不能自主提供 gas；通常是别的账户或调用方付 gas。当合约代码运行时，gas 是由触发它的那笔交易中提供。
以太坊设计 &协议规则	Ethereum 的交易模型设计中要求交易签名必须来自一个私钥，这是 EOA 的功能。协议里没有“合约账户自行生成交易”的机制（除非在某些新的抽象化提案 /标准中有所拓展）。	合约账户只能响应 transaction 或调用；为了安全、可预测性、不可滥用，协议不允许合约自己主动在没有外部触发的情况下发起交易。

2. 也有一些特别的情况：

ERC-4337 账户抽象 和 EIP-7702 / Pectra 升级。

2.1 ERC-4337：合约钱包“像在发交易”，但真正广播的还是 EOA

正如下面的话：

Account Abstraction（账户抽象，如 ERC-4337）：让智能合约类的账户（Smart Contract Accounts）具备类似 EOA 的一些操作能力，比如可以让用户通过合约钱包 / 捆绑操作提交一个 UserOperation，而这个操作由捆绑器（bundler）提交到网络。虽然合约账户在形式上可能“发起”操作，但底层仍有 EOA 或者特殊机制参与签名与 gas 支付。

补充更新：

- **ERC-4337 已在 2023 年 3 月部署到以太坊主网**，无需修改共识层，通过 EntryPoint 合约 + UserOperation 的方式在“应用层”实现账户抽象。
- 从 2023–2024 年，ERC-4337 智能账户在以太坊和各大 L2 上快速增长，统计显示到 2024 年底累计已部署**上千万级别的 smart accounts**，2024 年单年新增接近 2000 万个 ERC-4337 智能账户。
- **关键点没变**：UserOperation 不是链上的“原生交易类型”，而是提交给 **Bundler** 的“意图对象”；Bundler 最终还是用自己的 **EOA 签名并发送一笔普通交易** 调用 EntryPoint，把多条 UserOperation 打包执行。

所以：从协议视角看，真正写进区块的仍然是一笔由 EOA 发起和签名的交易；合约钱包只是让“谁来验证签名、谁替你付 gas、如何组织多步调用”变得可编程。

2.2 “自触发”机制：本质还是外部服务在帮合约按按钮

有些合约设计了“自触发”机制（例如定时器 or oracle trigger），但这些机制仍须由外部资源或者 EOA 或链外服务触发。合约本身不会凭空启动新交易。现在常见的是：

- 各种 **automation / keeper 服务**（如 Gelato、Chainlink Automation 等）在链下定时检查条件，一旦满足，就用自己的 EOA 发起交易调用合约。
- 从链上角度看，依然是“某个地址发起了一笔交易”，合约只是被动地在交易执行过程中跑自己的逻辑。

2.3 UserOperation + Bundler 的“捆绑操作”

做一个最新的补充：

捆绑操作模型更像你写好很多办事单子（UserOperations），把它们交给一个代办服务（Bundler），代办服务帮你拿到银行审批，通过 EntryPoint 办理这些操作。

截至 2025 年：

- ERC-4337 形成了**独立的“替代 mempool”（alt mempool）**，UserOperation 在这里被收集，再由 Bundler 打包为普通 L1 交易。
- 越来越多钱包（如 Safe＋4337 模块、Rhinestone 等）直接提供 AA 钱包体验，但它们依然依赖某个 Bundler/节点在底层用 EOA 把操作送上链。

2.4 账户抽象的现状：ERC-4337 + 即将到来的 EIP-7702

近期更新：

【从 ERC-4337 到 EIP-7702】

- **ERC-4337**：已经在主网运行两年多，被视为“**不改共识层实现账户抽象的第一阶段**”，生态里已经有几十个钱包和项目基于它构建智能账户系统。
- **EIP-7702 + Pectra 升级**：被设计为在协议层引入一种新交易类型，让 EOA 在单笔交易期间可以**临时挂载一段合约代码**，或者持续委托给一个合约地址，从而“像智能钱包一样”执行批量操作、代付 gas 等高级逻辑，而不用迁移到新地址。

但要特别强调的一点：

即便有 ERC-4337 和即将到来的 EIP-7702，从共识 / 协议层看，“只有具备签名能力的账户（EOA 及其扩展形式）才能发起 L1 交易”这一根本规则依然没变；合约账户本身仍然不能凭空在链上创建新的顶层交易。

给出一句话（可以直接背诵）

小结：

今天为止：链上的每一笔交易，起点仍然是某个 EOA 的签名； 合约账户再“聪明”，也只是被调用时执行，而不是自己起头发交易。 账户抽象和 EIP-7702 做的事，是让“谁来签、怎么付费、怎么组合操作”更灵活，而不是改变这一条物理定律。

六、合约账户的余额与状态存储

以太坊中的每个账户（无论是 EOA 还是合约账户）在“世界状态”（world state）里，都被建模为一个**四元组**：

$$\sigma[a] = (nonce, balance, storageRoot, codeHash)$$

ethereum.github.io+2arXiv+2

四个字段的**官方定义**：

- **nonce**
  - 对 EOA：记录该地址已经发送过多少笔交易；
  - 对 **合约账户**：记录该合约地址“创建了多少个合约”（通过 `CREATE` / `CREATE2`）。
- **balance**
  - 账户持有的 ETH 数量（单位为 wei，1 ETH = 10^{18} wei）。
  - **EOA 和合约账户都可以持有余额，并且都能收款 / 付款（只不过合约“付款”的逻辑由代码控制）。**
- **storageRoot**
  - 这是一个 Merkle-Patricia Trie（或将来 Verkle Trie）**根哈希**，指向该账户的“存储树”（account storage trie）。
  - **只有合约账户才真正使用这个字段**：用来保存合约的状态变量（`mapping`、`uint`、`address` 等）的键值对；
  - 对 EOA，规范上这个字段也存在，但值是“空存储根”（默认值）。
- **codeHash**
  - 合约账户：EVM 字节码的哈希 —— 用来标识和校验合约代码；
  - EOA：没有代码，**规范上 codeHash 是空字符串的哈希**（相当于“没有代码”的占位标记）。

也就是说：从协议内部看，EOA 和合约账户在结构上是同一种账户对象，只是对 EOA 来说 codeHash / storageRoot 这两个字段是“空的 / 没意义”；你用“只有合约账户才有存储、才真正使用 storageRoot”来讲是完全可以的，只要在脚注提醒一下“规范里字段也挂在 EOA 上，但不真正使用”就够了。

1. 存储余额与状态

1.1 余额（balance）

- 合约账户同样拥有 ETH 余额字段，可以像 EOA 一样接收 ETH：
  - 收款：直接向合约地址转账；
  - 付款 / 用钱：在合约代码中通过 `transfer` / `call{value: ...}` / `selfdestruct` 等方式转出。
- 文档和教程也都明确写着：**合约账户可以像 EOA 一样持有 ETH 和代币**。

唯一的区别是：

- EOA 用“私钥签名 + 交易”来花钱；
- 合约账户靠“代码逻辑”来决定何时、向谁、在什么条件下花钱。

1.2 存储变量（state）

- 合约账户拥有独立的 **storage trie**，保存所有在合约中定义的状态变量：
  - 例如 `mapping(address => uint256) balances;`、`uint256 totalSupply;` 等，都会映射到这棵存储树中的一个或多个 `slot`；
  - 这棵树的根哈希就是账户状态中的 `storageRoot` 字段。
- EOA 没有代码，也不能修改自己的 storage，因此它的 `storageRoot` 在实现里是固定的空值。

对比表：

属 性	EOA	合约账户
私钥	有私钥，可签名交易	无私钥，不能主动签名
balance	有余额字段	同样有余额字段（可以像 EOA 一样持有 ETH/代币）
codeHash	**逻辑上“无代码”；规范中存的是空代码哈希**	有字节码哈希，用于合约执行与识别
storageRoot	**逻辑上“无存储”；规范中是空存储根**	有 trie 根，用于存储合约状态

从“写合约 / 用合约”的角度——

“只有合约账户才有状态存储、才真正使用 storageRoot / codeHash” ——对读者是很友好的抽象；只是从协议实现看，这两个字段在 EOA 上也“占了一个坑”，但不承载实际业务含义。

1.3 补充解释 codeHash

在以太坊中，codeHash 是合约账户（Contract Account）状态的一部分，表示该合约账户的字节码（bytecode）的哈希值。

对于 EOA，由于没有合约代码，codeHash 通常为空字符串的哈希…

这和现在的规范以及实现完全一致，再加两点“面向开发”的 info：

①为什么要存哈希而不是直接把 code 塞进账户对象？

- account 四元组会被编码进状态 trie，如果每次余额或 nonce 变化都重新 RLP 整段 code，会非常浪费；
- 代码实际单独存储，**账户里只存 codeHash 作为“指针 + 防篡改承诺”**。

②在工具里怎么看到 code？

- 通过 **eth_getCode(address)** 可以根据 codeHash 拿回真实字节码；

- 区块浏览器会基于 codeHash 做合约识别与“已验证源码”匹配。

补充一点 2024–2025 的“未来视角”：Verkle / Stateless 不会改变“账户能否存余额和状态”这件事

最近几年大家讨论很多的 Verkle Trees、Stateless Clients、本地状态压缩等研究，主要是想解决 **“节点存太多状态太重”** 的问题：

- 黄皮书 + 最新研究仍然以“账号状态四元组（nonce, balance, storageRoot, codeHash）”作为逻辑抽象；
- Verkle / Binary Merkle / VOPS 等提案，改变的是 **state 在底层树结构中的存储方式和证明方式**，而不是“合约能不能存余额和状态”；

> 换句话说：
>
> **就算以后底层从 Merkle-Patricia Trie 换成 Verkle 或统一二叉树，对开发者而言：“合约账户能收 ETH、能持久化状态变量” 这件事实不会改变，只是 storageRoot 的内部含义会变得更“高科技”一点。**

1.4 类比帮助大家理解：

> 来做个收尾：
>
> 把“合约账户能存余额和状态”比作现实中的一个“银行账户 + 自动柜台”：
>
> - **余额（balance）** 就是这家账户里的钱，EOA 和合约账户都有；
> - **storage（state）** 像这家账户背后的一整柜档案：白名单、黑名单、股东名册、积分、配置表……
> - **codeHash + storageRoot** 就像“这家银行柜台的工作手册版本号 + 档案柜目录编号”，保证你每次查到的都是一致的版本。
>
> 合约账户之所以特别，是因为：
>
> **它既有钱（balance），又有“记账本 / 档案柜”（storage），还内置了一套“如何动这些钱、如何改这本账”的自动规则（code）。**

这就是为什么 DeFi 协议、NFT 合约、DAO 国库 都爱用“合约账户”：既能收钱，又能记账，还能自动执行规则。

七、EOA 与合约账户的互相调用机制

1. 从 EOA 发起调用到合约账户

EOA 用户通过私钥签名发起**外部交易（external transaction）**，交易中会包含：

- to：目标合约地址（或 EOA 地址）
- data：编码后的调用数据（函数选择器 + 参数）
- value：附带发送的 ETH（可选）
- gas 与 gasPrice / maxFee 等字段

这笔交易被广播到以太坊网络，进入内存池（mempool），最终被验证者打包进区块并执行。

当交易被执行时：

①EVM 会根据 to 找到目标合约账户，加载其 codeHash 对应的字节码；

②用 data 作为输入执行合约函数逻辑；

③过程中可以修改合约的存储状态（storage）、向其他地址转账 ETH，或触发事件（logs）。

> 术语对应：
>
> - 从 EOA 发起的“带 to/data 的交易”是 **交易（transaction）**；
> - 在 EVM 执行过程中合约之间互相调用，是 **消息调用（message call / internal transaction）**。

2. 合约账户之间的调用（合约互调用链路）

在合约执行过程中，当前合约可以通过 EVM 的若干 opcode（在 Solidity 中表现为不同的低级调用）与其他合约账户交互，例如：

- call：普通调用，对方在自己的存储中执行逻辑，msg.sender 变为当前合约；
- delegatecall：**在当前合约的存储上下文中执行对方代码**——存储读写仍落在发起方（调用者）合约上，msg.sender/msg.value 保持外层值，常用于代理 / 升级模式；
- staticcall：只读调用，不允许修改状态，只能 view/pure。

这些调用：

- 在链上表现为**内部交易（internal transactions/message calls）**，不会额外生成一条新的顶层 transaction 记录；
- **仍然消耗 gas**，但消耗的是当前这笔外部交易预付的 gas；合约本身不能“再多发一笔交易来买 gas”，只能在既有 gas 限额里完成所有子调用；
- Solidity 里通常可以拿到调用结果：(bool success, bytes memory ret) = other.call(...);，根据返回值决定后续逻辑。

“链上内部消息（内部交易）”在学术和工具文献里是这样定义的：

> 外部交易由 EOA 发起，合约间调用被称为 internal transactions / message calls，由 EVM 执行引擎在同一条交易上下文中完成。

3. 合约账户无法主动发起 EOA 级交易

这一点截至 2025 年依然是**铁律**：

- 合约账户没有私钥，无法对“新交易”进行签名，因此不能“凭空”构

造一笔新的 **外部交易**；

- 合约账户可以在执行过程中通过 `call{value: ...}` 等方式给 EOA 或其他合约地址转账、调用对方函数，但这都是**当前交易内部的 EVM 步骤**，而不是额外生成一条新的顶层 transaction。

> 换句话说：一条链上交易的“起点”永远是某个具备签名能力的账户（传统 EOA 或经过 AA 扩展的 EOA），合约只是这条执行路径里的“中间站”。

4. 补充解释一些新东西：AA 钱包、ERC-4337 与 EIP-7702

账户抽象、捆绑操作，在“谁起头发交易”，只是让调用链更绕了一层，现在额外补充两个结论：

4.1 ERC-4337（EntryPoint + Bundler）

- 用户（通常是智能合约钱包）构造一个 `UserOperation`，交给 Bundler；
- Bundler 用自己的 EOA 发起一笔普通交易，调用 EntryPoint 合约的 `handleOps`；
- EntryPoint 再在内部依次 `call` 各个智能钱包和目标合约

从底层看：**真正写进区块的，仍然是一笔由 Bundler 的 EOA 发起的交易**，钱包/合约之间的交互全是内部调用。

4.2 EIP-7702（Pectra 升级已在 2025 启用）

- 为 EOA 引入新的 **Type-4 “set code” 交易类型**，允许 EOA 在一个交易生命周期内临时挂载一段合约代码，拥有“智能钱包式”的批量执行、Gas 赞助等能力；
- 但这依然是“EOA 发出一笔交易，然后在执行期间委托某段代码处理逻辑”，并没改变“合约不能凭空发起顶层交易”的基本规则。

对照表：

发起方	调用对象	方式	描述
EOA	合约账户	**外部交易（transaction）**	使用私钥签名发起，广播到网络，被打包后触发合约执行，是整个调用链的起点。
合约账户	合约账户	`call` / `delegatecall` / `staticcall` 等	在同一笔交易执行上下文中的**内部消息调用**，消耗当前交易的 gas，不会生成新的顶层交易记录。
合约账户	EOA	不能主动“发交易”	可以在执行过程中通过 `call{value: ...}` 给 EOA 地址转 ETH（内部调用），但不能像 EOA 那样签名并广播一条独立交易。
EOA / Bundler	钱包合约+目标合约	交易 + 多层内部 call	在 ERC-4337 / EIP-7702 模式下，由 Bundler 或挂载代码后的 EOA 发起一笔交易，再由 EntryPoint / 钱包合约在内部批量调用多个合约，本质仍是一条 EOA 起头的交易。

一句话（可以直接背诵）

> 外部交易永远是 EOA → 合约（或 EOA），之后在同一笔交易里才会出现合约 → 合约 / 合约 → EOA 的内部调用链。

八、MetaMask 钱包对 EOA 的管理

1. 私钥与助记词

当你初次使用 MetaMask 创建钱包时，它会在本地生成一个 **12 个英文单词** 的 Secret Recovery Phrase（SRP，助记词），遵循 BIP-39 标准，从这个种子可以派生出多个私钥与 EOA 地址。

- **SRP / 私钥全部保存在本地设备：**

MetaMask 会用你设置的钱包密码对 SRP 与私钥进行加密，存放在浏览器扩展 / 手机本地存储中，不会上传到 MetaMask 服务器。丢了 SRP 官方也无法帮你找回。

- **拿到 SRP = 控制所有该钱包下的账户：**

SRP 是“主钥匙”，所有基于它派生出来的 EOA 都可以被恢复，所以任何获得你 SRP 的人都可以完全控制你的资金与操作。

> 官方现在明确写在帮助中心里：MetaMask 不会保存你的 SRP，任何“客服”向你要助记词，都是诈骗。

2. EOA 的生成与切换

钱包基于 SRP，按照 BIP-32/BIP-44 的分层确定性规则（HD wallet），在本地连续派生出一串 EOA 地址（比如常见的 `m/44'/60'/0'/0/0`、`0/1`… 这些路径）。

- 在界面里，你点击“创建账户”，其实就是让钱包**用同一个 SRP 派生下一个地址**，而不是再生成一套新的种子。
- 你也可以在 MetaMask 中**导入单独私钥**或另一套助记词，从而把其他钱包的 EOA 一并管理。
- 同一个 EOA 地址可以同时连接多个 EVM 兼容链（以太坊主网、OP、Arbitrum、BSC 等），只是每条链上余额与状态相互独立。

EOA 的“选择 / 切换”本质上就是：告诉 dApp“当前用哪个派生地址签名交易”。

3. 交易签名流程（MetaMask 如何用 EOA 发交易）

当你在 dApp 里点“Swap”“Mint”“Stake”等按钮时，大致会经历这些步骤：

①dApp 通过 `window.ethereum` / EIP-1193 请求你的钱包发起交易；

②MetaMask 弹窗展示：

- 目标地址 `to`
- 调用数据 `data`（对应合约函数 + 参数）

- 发送的 ETH 数量 value
- 预计 gas 费用等

③你确认后，钱包在**本地**用当前 EOA 的私钥对这笔交易做 ECDSA 签名；

④钱包再通过你配置的 RPC 节点（官方默认或自定义，如 Infura、自建节点等）把签名后的交易广播到链上。

关键点：

- 节点只看到“已经签名好的交易”和你的地址，看不到你的私钥；
- 是否广播、是否签名，完全由你本地的钱包控制。

4. 支持智能账户拓展（Account Abstraction）

“智能账户（Smart Account）”，在实际实现上主要是基于 **ERC-4337 账户抽象** 标准，再配合钱包自己的 UI 与后端服务。

4.1 MetaMask Smart Accounts（基于 ERC-4337）

截至 2025 年，MetaMask 在桌面扩展 v12.17.0+ 已经正式支持所谓的 **Smart Account / Account Abstraction**：

- 底层依托 ERC-4337 引入的 EntryPoint、UserOperation、Bundler、Paymaster 等组件，让“合约账户”可以作为用户主账户；
- 提供 **批量操作、Gas 赞助 / 使用代币付 gas、更灵活的恢复策略（如社交恢复、多签逻辑）** 等高级功能；
- 用户可以在 MetaMask 界面里在“普通账户（EOA）”与“智能账户（合约钱包）”之间切换；但底层仍然是你掌握 SRP/ 私钥，MetaMask 官方不会托管密钥。

ERC-4337 生态在 2024–2025 年发展非常快，很多项目和钱包都在用它实现智能账户和 gasless 体验。

> 「通过 EIP-7702/4337」可以理解为：
>
> - **当前主流钱包（包括 MetaMask）真正落地的是基于 ERC-4337 的 Smart Account；**
> - **EIP-7702 则是在协议层为这种“智能账户行为”补了一块更底层的基石。**

4.2 EIP-7702：协议层让 EOA “临时变聪明”

现状与定位：

- 状态：EIP-7702 已随 2025 年的 Pectra 升级在以太坊主网激活（包含在 Pectra 硬分叉中），并在 Sepolia 等测试网可用。
- **它引入新的 0x04 交易类型（SetCode / Authorization 类型）**，允许 EOA 在一笔交易的执行生命周期内，临时把自己“挂接”到某个合约代码上，让这笔交易期间对该地址的操作都委托给那段合约代码执行。
- 设计目标是：
  - 让 EOA 也能拥有类似智能钱包的能力（批量操作、session key、社交恢复、更多签名机制…）；
  - 与现有 ERC-4337 AA 生态兼容，而不是互斥。Alchemy+1

概括：

> EIP-7702 把“EOA 能不能执行合约逻辑”这件事，放进了协议层，用一种可授权、可撤销、带 nonce 防重放的新交易类型来做；钱包（包括 MetaMask）可以在此基础上继续构建更上层的智能账户体验。

风险点（授权给恶意合约、存储布局冲突、nonce 管理复杂度、用户误操作等），在目前社区讨论中依然被视作需要注意的安全重点，尤其是开发者在实现时必须充分审计授权合约与授权流程。

5. 备份与安全防诈骗（与现在官方建议对齐）

备份方式 & 钓鱼手法很实用，额外加一句“和官方安全指引对齐”的补充：

- MetaMask 官方现在在安全文档和 2025 年的安全报告中，反复强调几件事：
  - **永远不要把 SRP/ 私钥输入到网页表单、发给“客服”、截图发给任何人；**
  - 官方支持不会以任何理由向你索要助记词；
  - 备份尽量使用离线、物理介质（纸张 / 金属板 / 保险箱等），避免明文存云盘或聊天软件；
  - 警惕钓鱼站、假钱包扩展与伪造签名弹窗。

各种钓鱼场景（假客服、假官网、恶意 App、剪贴板劫持等），这些不是理论，已经在真实攻击中多次出现”即可。

小结

- **MetaMask 这类钱包管理 EOA 的核心依然是：本地生成 / 加密保存 SRP → 本地派生私钥 / 地址 → 本地签名 → 通过 RPC 广播。**
- **账户抽象（ERC-4337）+ EIP-7702 只是把“这个 EOA 能做什么”变得更灵活，而不是改变“私钥归谁所有”这件底层事实。**
- 对用户来说，重点没变：

> 记好助记词、管好私钥、看清每一次签名弹窗，
> 其余的“智能账户”“批量交易”“gas 赞助”“7702 高速通道”，都只是你之上的“外挂”和“外骨骼”。

九、ERC20 / ERC721 代币与合约账户的关系

1. 代币就是智能合约存储的“记账系统”

ERC-20 合约（用于同质代币）内部会维护类似下面这些状态变量，用来记录余额和授权额度：

```
mapping(address => uint256) private _balances;
mapping(address => mapping(address => uint256)) private _allowances;
```

也就是说，当你“拥有” ERC-20 代币时，**真正的资产记录是这份合约的 storage 中那几行映射数据**，而不是“某个地址里多了一袋硬币”。不同实现的变量名可能略有差异（`_balances`、`balances` 等），但逻辑都是：**合约用映射来记账**。

ERC-721 合约（用于 NFT）则保存的是**每个唯一 `tokenId` 的持有者地址**，以及余额映射和可选的枚举 / 元数据等结构，例如：

```
mapping(uint256 => address) private _owners;          // tokenId -> owner
mapping(address => uint256) private _balances;       // owner     -> count
```

加上一些扩展（如 `tokenURI`、Enumerable），就能把“这是谁的第几号 NFT、对应什么元数据”都记录在合约存储中。

> 总结一下：ERC-20 / ERC-721 代币本质上都是某个合约账户里的“记账表 + 规则”。

2. 代币操作就变成了合约调用

如果你想转移代币，并不是“从地址 A 打一笔‘代币转账交易’给地址 B”，

而是：

①你的 EOA 向**代币合约地址**发起一笔交易；

②交易的 data 字段里编码了要调用的函数，例如：

- ERC-20：transfer(to, amount)、approve(spender, amount)、transferFrom(from, to, amount)；
- ERC-721：safeTransferFrom(from, to, tokenId)、approve(to, tokenId) 等。

③代币合约在 EVM 中执行：

- 检查调用者权限、余额是否足够；
- 更新 _balances / _owners / _allowances 等存储；
- 发出 Transfer、Approval 等事件；
- 可能还会回调其他合约（比如 onERC721Received 钩子，在 2025 年的安全审计文献里甚至被点名是 NFT 领域重入攻击的重要入口）。

所以：**“转代币” 本质上是 “调用代币合约的一段代码 + 改一行存储”**。

3. 合约账户是逻辑执行与数据存储中心

一个 ERC-20 或 ERC-721 合约账户，本质上就是一个“代币系统的逻辑 + 数据”打包体：

- 它有自己的地址（例如 0xTokenContract...）；
- 它的 codeHash 标识这段代币逻辑（转账、授权、mint、burn 等）；
- 它的 storageRoot 对应完整的存储树：
  - 谁有多少余额；
  - 谁被授权可以代为花费多少（ERC-20 allowance）；
  - 每个 tokenId 属于谁、元数据在哪里（ERC-721）。

在 2024–2025 年，大多数主流代币合约都会在这个基础上叠加更多“合约

逻辑”：比如可升级代理、访问控制、铸造 / 销毁权限、手续费机制等，但**本质仍然是“一个合约账户 + 一棵状态树”**。

4. 用户持有代币，是合约里那份“账”

EOA 地址本身**并不存放 ERC-20 / ERC-721 代币对象**，它只在代币合约的映射里作为一行 key 出现。

- 当钱包（MetaMask、Rainbow、钱包 App 等）展示你的代币余额时，本质是：
  - 先知道“这个代币的合约地址”；
  - 调用代币合约的 `balanceOf(yourAddress)` / `ownerOf(tokenId)` 等接口；
  - 把返回的数字（或 NFT 列表）渲染在界面上。
- 区块浏览器（如 Etherscan）也是同理：只要知道“代币合约地址 + 用户地址”，就能实时从合约 storage 中读出来。

所以更精确地说：

> 你“拥有”某个 ERC-20 / ERC-721 代币，并不是因为你的地址里有一包代币，而是：**在那个代币合约的 storage 里，有一条记录写着：这个地址的余额是多少 / 这个 tokenId 的 owner 是谁。**

十、合约部署后的不可篡改性与销毁

整体结论：**合约代码默认仍然是“不可修改的”，但“删除”这件事在 Dencun 升级（2024-03-13）之后已经基本被废掉，只在极少数同交易场景下还算真正删除。日常开发应视为：合约一旦部署，就几乎不能改、也不能删，只能通过代理等模式“曲线升级”。**

1. 代码不可更改

> 代码不可更改：合约部署后，字节码存储在指定地址，默认为不可变。

而且在 Dencun/EIP-6780 生效后更“硬核”：

- 合约账户在状态树里包含 codeHash 字段，用来引用部署好的 EVM 字节码；一旦写入，这段代码就不会被就地修改。
- 升级协议时，核心开发者专门引入了新的规则，让任何依赖“修改 / 替换代码”的用法都变得不可靠或直接失效，目的是为后续 Verkle tree、无状态客户端铺路。

不要指望“换代码”，只能换合约地址或用代理模式。

2. SELFDESTRUCT：现在基本“不再真的删合约”了

> 可选的 SELFDESTRUCT：如果合约中实现了 SELFDESTRUCT（或 DELEGATECALL+SELFDESTRUCT 等组合），它可以在执行时销毁自身代码和状态，也可能将 ETH 转移到指定地址。

这段在**历史上**是对的，但在 2024 年 Dencun 升级以后，已经不再准确，需要加上“新语义”的注脚：

- EIP-6049 先把 SELFDESTRUCT 标记为**待废弃（deprecated）**，提示未来语义会改变；Solidity 文档也明确不推荐在新合约中继续使用。
- EIP-6780（随 Dencun 上线）真正改变了语义：
  - **只有在“合约创建的同一笔交易里调用 SELFDESTRUCT”** 才会像以前一样，彻底删除该合约的代码和存储；
  - **如果合约是早在过去就部署好的，然后在后续交易中调用 SELFDESTRUCT，现在只会转走余额，不再删除代码和 nonce**——合约的代码实际上还留在链上，只是状态被特殊处理。

学术和社区文章已经明确指出：

> 在 EIP-6780 后，SELFDESTRUCT 不再能用来“清空地址并为重新部署腾位置”，除非在同一 transaction 内创建并销毁，这对依赖它实现“代码变形 / 重新部署”的模式等于判死刑。

更新理解：

- “可以销毁自身代码和状态”这句话，现在需要限定为：**仅在“创建同交易内调用”的极端场景才成立**。
- 在实际业务合约里，**不应该再设计依赖 SELFDESTRUCT 的逻辑**（包括清库、升级、锁仓等），因为行为已变且未来还有被进一步收紧甚至删 opcode 的可能。

3. 地址可重部署：Metamorphic 合约模式几乎被 EIP-6780 “封杀”

> 地址可重部署：自销毁后，通过 CREATE2 机制，有可能在相同地址重新部署新合约（metamorphic contract），但必须在初始部署时设计并允许这种行为。

这说的是经典的 **Metamorphic Contract Pattern**：

- 利用 CREATE2 的可预言地址 + SELFDESTRUCT 重置 nonce，在**旧语义**下，可以在同一地址反复部署不同字节码，实现“同地址不同逻辑”的“合约变形”。

但是在 EIP-6780 之后：

- 一旦合约是在**过去的某笔交易**中部署的，之后的 SELFDESTRUCT 不会再真正清掉代码；
- 这意味着：**大多数依赖 SELFDESTRUCT + CREATE2 的 metamor-**

phic 合约模式已经失效，不能再假设“自毁后地址干净，可以重新部署全新的代码”。官方 EIP 明确把这种用法列为“将被破坏且不再安全”的模式之一。

参考历史：

> 地址可重部署（历史上的模式）：
>
> 过去开发者会用 SELFDESTRUCT + CREATE2 做所谓 metamorphic contracts，在同一地址多次部署不同字节码，实现“同地址升级”。但 **EIP-6780 实施后，这种模式在绝大多数场景下已经失效或被认为不安全**，不应再作为升级方案使用，只能当成历史课题或 CTF 题目来了解。

4. 代理合约模式（Proxy Pattern）：现在几乎是唯一主流升级方案

> 代理合约模式（Proxy Pattern）：大多数“可升级合约”采用代理模式——主合约（proxy）永不变，只是可指向不同的实现合约地址，通过 delegatecall 调用新的逻辑，实现功能升级，同时保留原始地址和状态。

在 SELFDESTRUCT 被弱化之后，**更成为主流推荐方案**：

- 安全审计公司和社区普遍建议：**如果要做可升级合约，请使用代理架构（如 Transparent Proxy、UUPS、ERC-2535 Diamond 等），不要再依赖 SELFDESTRUCT + CREATE2 这类黑魔法**。
- OpenZeppelin 等主流库已经围绕代理模式形成一整套工具链（合约基类 + 插件 + 脚本），新项目基本都会走这一套。

一句话（可以直接背诵）：

> 现在的“正确姿势”是：地址不变 → 代理合约；逻辑变更 → 换实现合约地址。不要再用 SELFDESTRUCT 做升级。

5. 补充一点用户如何与合约账户交互实现功能？

补充：

> 以太坊合约账户交互的核心在于：用户（EOA）通过签名交易或“读”请求，将调用数据发送到合约地址，EVM 按合约字节码执行逻辑并更新存储或返回数据。整个过程可在不同层面完成，包括区块浏览器的“写合约”界面、钱包插件（如 MetaMask）的内置调用、使用前端库（Web3.js/Ethers.js）在网页或脚本中直接调用，以及借助 Truffle/Hardhat 等框架在控制台或脚本中管理合约。无论哪种方式，都需要合约地址与 ABI（或合约接口），并通过 JSON-RPC 与节点通信。

2024–2025 的工具生态现状：

- JS 侧现在更常用 **ethers.js、viem** 等库来发起 JSON-RPC 调用；
- 合约开发测试更多转向 **Hardhat / Foundry**；
- 日常用户大多通过钱包（MetaMask、Rabby 等）签名，前端则只是把 ABI + 参数编码成 `data` 发送给节点。

第四章

智能合约理论基础

本章目标：深入理解智能合约的工作原理与价值

一、智能合约的概念与功能

智能合约是以太坊应用程序层的基石。更严格一点地说，它就是部署在区块链上的一段程序代码 + 一份持久状态：一份“代码（functions）+ 数据（state）的组合，驻留在以太坊上的某个地址上，由交易触发后执行。

可以把它理解为：

> 运行在以太坊上的一段自动执行的“程序合同”，只要输入条件满足，就会按事先写好的规则自动执行。

这些智能合约部署在以太坊主网，或者与主网紧密相连的扩展网络（如各类 Rollup / L2），统一由以太坊的共识和数据可用性来保障安全与可验证性——你之前说的“子区块链”，可以理解为这些挂靠在 L1 之上的扩展链或子系统，而智能合约就是它们上的基本执行单元。

它们遵循“如果...那么...” (IFTTT) 的逻辑，并且保证按代码执行：

- 从协议层面看，合约一旦部署，字节码本身是不可直接修改的（immutable by default）；
- 现实开发中，如果需要“升级合约”，通常会使用代理（Proxy）模式、UUPS 等可升级标准：通过一个固定地址的代理合约持有状态，把逻辑合约替换掉，从而在不改变地址和存储的前提下升级功能，这已经是当前以太坊生态中广泛使用的模式。

此外，自 2024 年 Dencun 升级引入 EIP-6780 后，`SELFDESTRUCT` 的行为被大幅“阉割”：只有在“创建合约的同一笔交易中”调用时才会真正移除代码，否则只会清空余额而不会删掉合约代码和历史，从而为未来的 Verkle 树等结构做准备，也意味着再依赖 `SELFDESTRUCT` 做“升级/回滚”的老套路已经不再推荐。

Nick Szabo 创造了“智能合约”这一术语。1994 年，他撰写了智能合约简介；1996 年，他撰写了对智能合约潜在功能的进一步探索。

Szabo 构想了一个数字市场：在这个市场中，交易和商业功能可以在没有受信任中介的情况下，通过加密学保障安全、自动化执行。以太坊上的智能合约基

本上把这一设想真正落在了公链上——成为 DeFi、NFT、DAO 等应用的底层“执行引擎”。

1. 传统合约中的信任问题

传统合约的最大问题之一是：你不仅要相信“合同写得对”，还要相信“有人会照着合同执行结果”。

举个你原来的例子：

> A 和 B 要进行一场比赛，A 和 B 打赌 10U 他会赢。B 也相信自己能赢，于是同意下注。最后 A 大幅领先赢了比赛，无疑是赢家。但 B 拒绝支付赌注，声称 A 作弊。

这个看起来有点“无赖”的例子，暴露了所有非智能合约世界的痛点：

- 即使协议条件已经满足（A 确实赢了），
- 你仍然得信任对方会履约（B 会乖乖付钱）。

于是现实世界里，我们就引入了各种中心化中介/信任人：

如果下注时有一位共同朋友 C，A 和 B 可以将赌注提前交给 C，由 C 来观察比赛并把赌注发给赢家。

但这种模式显然又把风险交给了 C：

- C 可能失职、舞弊、被收买；
- 或者 C 只是跑路了。

这就是传统合约 + 中介模型脱不开的信任问题。

2. 智能合约能做什么？

3. 2.1 数字自动售货机

一个经典类比就是自动售货机——这个类比最早就是 Szabo 用来说明“智能合约”的：只要你投币达到价格，机器就必须吐出货物，整个过程不依赖“老板讲不讲信用”。

自动售货机的流程：

1. 你选择一种产品；
2. 机器显示价格；
3. 你付款；
4. 机器验证你是否付够钱；
5. 验证通过，机器吐出产品。

如果你没付够钱、或者没有选商品，机器根本不会出货。

智能合约也是类似逻辑：

条件满足 → 自动执行合约条款；

条件不满足 → 直接 `revert`，不发生状态变更。

在以太坊上，这种“自动售货机逻辑”被用在代币发行、借贷、清算、NFT 铸造/转移、DAO 投票等各类场景里。

4. 2.2 自动执行：无需中介、无需等待

智能合约的核心价值之一，是在条件满足时自动、确定地执行，不需要人来解释或操作。

例如：

- 你可以写一个合约，给小孩做**时间锁账户**：资金托管在合约里，只允许在他满 18 岁之后提取。
  - 若在 18 岁之前调用提取函数，合约会直接拒绝执行。
- 或者一个合约：当你向经销商付完款，就自动把“车辆所有权 NFT/登记信息”转到你的地址上，整个过程不需要车管所的人工审核。

现代企业和 DeFi 协议中，这类“条件触发自动执行”的模式已经非常普遍；IBM 等机构也正是以“自动触发、减少中介、减少误差”来描述智能合约的主要优势。

5. 2.3 可预测的结果：相同输入 → 相同输出

传统合约常常依赖人类来解释条款：

- 不同法官可能对同一条合同的理解不同；
- 不同仲裁员可能给出截然相反的结果。

智能合约则遵循这样一种哲学：

> 代码即合约（Code is law）。
> 在同样的链上环境、同样的输入下：
>
> - 智能合约以确定性方式运行，
> - 所有节点必须算出相同的结果，
> - 否则根本无法达成共识、打包进区块。

这让结果具备了高度可预测性：同样条件 → 绝对一致的执行行为，从而规避了一大批“人为解读偏差”的风险（当然，这也要求你在写合约时就把业务边界和异常情况想清楚）。

6. 2.4 公开记录与可审计性

所有智能合约的执行过程和状态变更，都记录在区块链上：

- 合约部署交易、
- 每一次调用、
- 每一次事件（Event）日志、
- 每一次资产转移……

这些都可以通过区块浏览器或者本地节点查询和验证。

这意味着：

- 你可以随时核查某笔资金是否真的打到了指定合约；
- 你可以追踪某个 NFT 从铸造到现在经历了多少次转移、每次价格是多少；
- 合规机构或审计方，也可以基于这些公开数据做审计、分析和取证。

这也是为什么 DeFi、NFT 市场分析、合规审计等领域，往往会构建在合约公开数据之上做指数和风控模型。

7. 2.5 “隐私”与假名制：不是完全匿名

由于以太坊是匿名网络，交易绑定的是地址而非真实身份，因此可以保护隐私。

这里需要稍微更新一下现实情况：

从协议设计上看，以太坊地址确实只是一串公钥哈希，没有强制绑定身份证/手机号等；

但今天的研究和实践已经充分表明：

- 以太坊的隐私其实是“假名制”（pseudonymous）而非真正匿名；
- 交易记录是完全公开、可追踪的；
- 通过链上行为模式 + 交易图分析 + 链下数据（交易所 KYC 等），往往可以把地址还原成现实身份。

因此，智能合约一方面让所有行为都可验证、可审计；

另一方面，如果想要真正的隐私保护，还需要：

- 使用专门的隐私协议（如混币、隐私池、零知识证明方案等），
- 或采用更强的隐私设计，而这些也是以太坊研究社区最近两年重点推进的方向之一。

所以更准确的说法是：

以太坊让你不必暴露实名就能参与网络，但在链上留下的一切行为都是公开可追踪的，隐私需要额外设计。

8.2.6 条款可见与可审查

和传统合同一样，你可以在“签署”（调用）之前审查智能合约内容：

- 源码开源的合约（尤其是经过 Etherscan / Sourcify 验证的）可以直接阅读、审计；
- 工具如 Slither、Mythril、Foundry + formal verification 框架可以自动化发现一些安全问题；
- 大型协议通常会公布多轮审计报告。
- 这带来一个很重要的现实：
- 用户有能力在交互前知道“我在跟什么代码打交道”；
- 但前提是：你要么自己看得懂，要么相信专业审计团队给出的报告和社区共识。

9.2.7 基本上你想到的逻辑，都能写成合约

总结：

其他计算机程序可以做的事情，智能合约在“链上逻辑 + 状态机 + Gas 限制”的约束下基本上都可以做。

典型用例包括但不限于：

铸造与管理货币/代币：

- ERC-20、ERC-721、ERC-1155 及各种自定义标准；

DeFi 协议：

- 借贷、DEX、衍生品、稳定币、收益聚合等；

NFT & 游戏道具：

- 铸造、交易、盲盒、链游内经济系统；

DAO & 治理：

- 投票合约、国库管理、多签和 Timelock；

数据存储与访问控制：

- 白名单、权限控制、访问日志、License 管理；

链上自动结算与清算机器人：

- 定时任务、自动再平衡、预言机驱动逻辑；

结合零知识证明的隐私/验证逻辑：

- 不泄露具体数据，只证明“满足某个条件”。

从 Nick Szabo 在 90 年代纸面上的设想，到今天以太坊主网 + L2 上数十万合约实例运行，智能合约已经从一个理念，变成了构建去中心化应用、重新组织经济关系的底层“乐高积木”。

二、Solidity 语言特性与优势

1. 原生契合 EVM 架构

Solidity 是为 Ethereum Virtual Machine（EVM）专门设计的合约导向语言，它能直接编译为 EVM 字节码，天然适配以太坊架构，能够高效操作存储、Gas 计算、事件等链上资源。官方也明确把它定义为“为 EVM 开发智能合约的高阶、静态类型、面向对象语言”。

截至 2025 年，Solidity 编译器已经发展到 0.8.30，持续跟进上海 / Dencun（EIP-4844）、Pectra 等升级，引入了诸如算术溢出自动回滚、transient storage（EIP-1153）、EOF 实验支持等一系列改进，让合约在安全性和 gas 成本上都有更好表现。

更关键的是：绝大多数 EVM 兼容链（如 Polygon、BSC、Avalanche C-Chain、Arbitrum、Optimism、Base 等），都沿用 EVM，因此学会 Solidity，基本就能“通吃”整个 EVM 链生态。

2. 易学而强大

它的语法融合了 JavaScript、C++、Python 元素，拥有静态类型、继承、多重继承、库、事件、修饰符等面向对象特性，对这些语言有经验的开发者易于上手。

Solidity 支持结构体、映射（mapping）、枚举、接口、抽象合约等高级特性，同时又保持与链上资源（storage/memory、calldata、gas、事件日志）的紧密结合，这让它既像高级语言，又能直接摸到“硬件”（EVM）。

3. 丰富生态与工具支持

Solidity 是以太坊上最普遍使用的合约语言，拥有完整生态系统，包括 Truffle、Hardhat、OpenZeppelin 等框架，以及大量文档、库与教程支持。

近两年需要补充一个“时代变迁”：

- 过去：Truffle 曾经是事实上的标准框架；
- 现在：Hardhat + Foundry 基本成为主流组合——根据 2024 年官方开发者调查，Hardhat、Foundry 在日常开发、测试、fork 主网模拟方面使用率非常高，Foundry 在单元测试和模糊测试场景中尤其受欢迎。

配套生态包括：

- Remix：浏览器 IDE，适合入门和小 demo；
- Hardhat：脚本化部署、网络 forking、本地链；
- Foundry：极速测试（Rust 写的 CLI 工具链）、fuzzing、主网 fork；
- OpenZeppelin Contracts：ERC-20/721/1155、AccessControl、Governor 等标准合约库，审计充分、社区采用率极高。

这**意味着**：写 Solidity = 直接站在整个以太坊 / EVM 生态工具链的风口上。

4. 这里给大家推荐一条学习的路线：

入门准备：

- 先了解区块链基础概念：什么是区块链、以太坊、节点、交易、智能合约。
- 如果你会一些编程（JavaScript / Python / C++ / Java 等），会比较快上手。

读官方文档 + 做例子：

- 从 Solidity 官方文档的 “Introduction to Smart Contracts” 和 “Solidity by Example” 开始。
- 用 Remix 这种网页 IDE 写第一个合约（HelloWorld，存 / 读变量，转账，事件触发）。

做交互式教程：

- 用 CryptoZombies、Solidity by Example 的小项目练手，感受 storage / memory、gas、事件、modifier 等细节。

学习工具 / 测试 / 部署：

- 在本地装 Hardhat 或 Foundry：
  - Hardhat：适合脚本部署、多网络管理；
  - Foundry：适合写大量单测、fuzzing、主网 fork 研究协议。
- 学会写单元测试（require 失败、revert、边界条件），这一步对以后搞 DeFi / NFT 协议非常关键。

安全 / 最佳实践：

- 学习常见漏洞：重入、整数溢出 / 下溢、访问控制错误、price oracle 操纵、flash-loan 攻击等。
- 用 Slither、Mythril 之类的工具做静态分析，配合 Foundry/Hardhat 测试，把显而易见的坑先扫一遍。

做一个小项目：

- 写一个简单 ERC-20 代币、投票合约，或者小型 DeFi 玩具协议，上 testnet 或某条 L2（如 Sepolia / Arbitrum Sepolia / Base 测试网）部署一遍，完整走完：编译 → 部署 → 前端调用 → 调试的闭环。

5. 安全性与成熟度

自 2014 年首次发布以来不断迭代，增加了针对合约安全的机制，例如异常处理、静态类型等，虽有历史漏洞事件，但社区工具（Mythril、Slither 等）可部分规避已知风险。

一个重要里程碑是 Solidity 0.8.0：

- 默认对整数运算做溢出检查（checked arithmetic），一旦溢出会自动 revert，不再像 0.7 之前那样静默 wrap-around，这直接消灭了一大类经典的整数溢出漏洞。

同时，围绕 Solidity 已经形成较完整的安全生态：

- 静态分析：Slither、Mythril、ConsenSys Diligence 等；
- 符号执行 / Fuzzing：Foundry 的 fuzz 测试、Echidna 等；

- 大量 DeFi / L2 项目已经用 Solidity 跑过实战考验（Uniswap、Aave、OpenSea 等），真实 TVL、攻击与修复迭代出来的 best practice 也在反哺语言与工具设计。

6. 图灵完备与功能丰富：

Solidity 是图灵完备语言，支持复杂数据结构（结构体、映射）、继承、接口等特性，能实现 DeFi、DAO、NFT、游戏、身份认证等复杂应用。

这一点在 2020–2025 年的实践里已经被反复证明：

- DeFi：Uniswap、Aave、MakerDAO 等主流协议都用 Solidity 编写核心合约；
- NFT / 游戏：OpenSea 的主流合约标准（ERC-721/1155），以及 CryptoKitties 等早期链游；
- DAO 与治理：Compound / Aave / Optimism 等治理合约与投票逻辑，多是 Solidity + OpenZeppelin Governor 体系。

7. 社区共识与兼容性

作为以太坊默认合约语言，Solidity 拥有最强的社区支持与开发者基础。在大多数协议、桥接、链间系统中享有一致支持，学习和使用大多数往往从 Solidity 入手。

从生态现状看：

- 在 DeFi TVL 维度，Solidity 合约占比接近 95%+，Vyper 等语言只占很小一部分；
- 绝大多数顶级项目（Uniswap、Aave、OpenSea 等）都采用 Solidity，这也意味着审计公司、工具、SDK、教程都围绕它优先构建。
- 各种 EVM L2 / 侧链（Arbitrum、Optimism、Base、Polygon、BSC…）都以“Solidity 兼容”为卖点，方便项目“一键多链”。

这就形成了一个很现实的结论：想写 EVM 世界 90% 的主流应用，Solidity 是默认答案。

8. 标准与库（少造轮子）

代币与 NFT 等有成熟的 ERC 标准与实现，可直接用 OpenZeppelin Contracts 这样的审计过库来继承扩展（ERC-20 / ERC-721 / 权限控制等），显著降低实现和审计成本。

ERC-20、ERC-721、ERC-1155、ERC-4626（金库）、ERC-20 Permit 等标准，已经在 Solidity 生态里形成了“模板 + 工具 + 实战经验”的组合，你只需要继承基础实现，然后在安全前提下做少量业务扩展即可。

9. 安全与审计支持

围绕 Solidity 的安全工具最丰富，常见如 Slither 静态分析等；这些工具能自动扫出常见漏洞并辅助人工审计，降低上线风险。

加上：

- 开源合约库（OpenZeppelin 等）；
- 成熟的审计公司生态（Trail of Bits、OpenZeppelin、Certora、Runtime Verification 等）；
- 以及持续更新的最佳实践（如避免 tx.origin、使用 ReentrancyGuard、采用 Pull Payment 等模式），

让“用 Solidity 写合约 + 用成熟工具和模式做安全防护”成为行业默认路径。

一句（可以直接背诵）：

人们用 Solidity，不只是因为“它是以太坊的语言”，而是因为——EVM 原生 + 生态最强 + 工具 / 标准最成熟 + 多链兼容 + 安全经验最多。如果你打算长期深耕以太坊 / EVM，Solidity 仍然是性价比最高的第一语言。

三、合约编译时产生的内容

1. 智能合约编译产物

2. 字节码（Bytecode）

定义：这是智能合约的核心。它是以太坊虚拟机（EVM）可以理解和执行的低级机器码。你可以把它想象成智能合约的“可执行文件”。

补充 1：创建字节码 vs 运行时字节码

Solidity 编译器实际会生成两段主要字节码：

- Creation Bytecode（创建字节码）：包含构造函数逻辑，用于部署时执行，最终负责把运行时代码写入链上。
- Runtime Bytecode（运行时字节码）：部署完成后真正存储在合约地址上的那段代码，之后每次调用合约都会执行它。

补充 2：元数据哈希附加在字节码尾部

从 Solidity 0.4 之后，编译器会自动在运行时代码的结尾附上一段经 CBOR 编码的“元数据哈希”（通常是 IPFS CID），里面包含 metadata 文件的哈希值和编译器版本等信息，方便在链上验证与检索合约源代码。可以通过 --no-cbor-metadata 关闭。

作用：当你部署智能合约时，实际上就是将这个字节码上传到区块链上。EVM 会读取并执行这些字节码来运行你的合约逻辑。

形式：通常以十六进制字符串的形式呈现，例如 0x6080604052…。

类比：你在工厂里把零件 + 指令拼起来，生产出一个机器（部署）——创建字节码就是整个工厂里的拼装图纸 + 材料；部署完机器装好后，不再包含那些“拼装指令”（构造阶段逻辑），只留下成品逻辑（runtime）。

3. 应用二进制接口（Application Binary Interface，ABI）

定义：ABI 是一个 JSON（JavaScript Object Notation）格式的文件，它描述了智能合约的公共接口。它包含了合约中所有公共函数和事件的详细信息，包括它们的名称、参数类型、返回值类型以及它们的可见性（public、external）。

作用：

- 外部交互：其他外部应用程序（如你的前端 DApp、钱包或另一个智能合约）需要 ABI 来知道如何与你的合约进行交互。它就像一个“说明书”，告诉外部程序如何正确地调用合约的函数和解析合约发出的事件。
- 数据编码 / 解码：ABI 定义了如何将 JavaScript 对象或其他高级语言的数据编码成 EVM 可以理解的字节格式（用于函数调用），以及如何将 EVM 返回的字节数据解码成可读的格式。

类比：像给你的合约写一个“菜单”，菜单上写了你能点什么菜（函数），要什么材料（参数），会有什么反馈（返回值 / 事件）。

补充：ABI 一般直接包含在工具产物里

使用 Hardhat、Foundry、Remix 等开发工具时，ABI 通常会和字节码一起被写进 artifacts 或 out 目录中的 JSON 文件里，前端和脚本直接读取这些 JSON 即可。

4. 合约元数据（Contract Metadata）

定义：这是一个 JSON 文件，包含了关于合约的额外信息，例如：

- 编译器版本和设置
- 源代码路径或其 IPFS / Swarm 等去中心化存储的引用
- ABI
- NatSpec 文档（如果在代码里写了）

Solidity 编译器会默认生成这一文件，并且（默认）把它的 IPFS 哈希通

过 CBOR 编码附加在运行时代码的尾部。

作用：它有助于验证已部署合约的来源和编译过程，增加透明度；区块浏览器（如 Etherscan、Sourcify）等也依赖 metadata 来做源码验证和自动生成交互界面。

类比：像产品说明书 + 质检报告，写明这个机器什么厂制造、什么批次、零件清单、配置如何，这样别人可以确认是不是正品或有没有被恶意篡改。

当前工具链的小更新：

- Hardhat 会在 artifacts/build-info/*.json 中保存一份完整的 Standard JSON Input/Output，其中包含 metadata 字段。
- Foundry 则会在 out/ContractName.sol/ContractName.json 中保存字节码、ABI 和原始 metadata 信息。

5. 源映射 / Source Map / Storage Layout / AST（抽象语法树）

抽象语法树（AST，Abstract Syntax Tree）

定义：AST 是源代码的结构化表示。编译器在编译过程中会生成它，用于分析和优化代码。

作用：主要用于编译器内部工作，或者在开发工具中进行静态分析、代码审计等高级用途。Solidity 编译器支持通过 Standard JSON 接口输出 AST，静态分析工具（如 Slither 等）会利用它来定位漏洞。

类比：机器的内部结构图（电路图 / 内部机械结构图），你要维护 / 升级 / 修理时，这图很重要。

Source Map（源映射）：

定义：编译器可以生成两类映射：

- AST 节点对应的源码区间
- 字节码指令 ↔ 源码区间 的映射

作用：用于调试与审计：例如在 Remix、Hardhat Debugger 中单步调试

时，高亮当前执行的源码行，或者在安全工具中精确指出“某条 opcode 对应哪行源代码”。

6. Assembly（EVM Assembly）与 Yul IR

EVM Assembly：

定义：这是比字节码更高级一些的中间表示，但仍是低级的。它包含了 EVM 操作码（opcode）和它们的操作数，更具可读性。

作用：对于深入理解合约执行逻辑、优化 Gas 消耗或进行安全审计的开发者来说，分析 Assembly 代码会很有帮助。Solidity 编译器可以通过 --asm 等选项输出这部分内容。

Yul / IR：

定义：Yul 是 Solidity 官方提供的中间语言，作为“IR-based codegen”的核心，用于在多后端（EVM、eWASM 等）之间共享优化逻辑。

作用：

- 编译器在内部会先把 Solidity 源代码翻译成 Yul，再从 Yul 生成最终字节码；
- 对 Yul 做优化可以同时优化所有后端目标；
- 你也可以手动写 Yul（或通过 inline assembly）来做极端优化场景。

7. 小结

如今（Solidity 0.8.x 版本及周边工具链）编译一个合约，典型会产生这些核心产物：

- 部署与运行时字节码：真正上链执行的代码（附带 metadata 哈希）。
- ABI：前端、脚本和其他合约用来“看懂你合约接口”的说明书。
- Metadata JSON：记录编译器版本、源码、ABI、NatSpec 等，用于验证与交互。

- AST / Source Map / IR / Assembly：供调试、审计、Gas 优化使用，多数由工具在后台消费。

四、部署合约：地址与 ABI 获取

部署智能合约后，你会获得一个合约地址（Contract Address）。

这个合约地址是智能合约在区块链上的唯一标识符，类似于你在现实世界中的房屋地址。一旦合约部署成功，它就会被分配一个独一无二的地址，所有对该合约的交互（例如调用它的函数、发送以太币给它）都需要通过这个地址进行。

在现代工具链（Hardhat、Foundry 等）中，一般会在部署脚本的输出里同时拿到：

- 交易哈希（deployment tx hash）
- 部署合约地址（contract address）
- 本地 artifacts 中的 ABI + bytecode（保存在 artifacts/ 或 out/ 目录里的 JSON）

至于 ABI，它是在编译阶段生成的，而不是部署后才获得的。在部署合约时，你需要将编译好的字节码（bytecode）上传到区块链，而 ABI 则用于你的应用程序（如 DApp 的前端）在链下与已部署的合约进行交互。Solidity 编译器还会生成一份 metadata JSON，里面同样包含 ABI、编译器版本、源代码哈希等信息，区块浏览器和验证服务（比如 Etherscan、Sourcify）会利用这些元数据来自动展示 ABI 和源码验证状态。

现在主流做法是：

1. 本地编译 → 得到 bytecode + ABI + metadata
2. 部署合约 → 得到链上的合约地址

3. 在 Etherscan 或 Sourcify 上做源码验证（source verification） → 区块浏览器会根据 metadata 解析出 ABI，并在网页上暴露“Read / Write Contract”面板，供任何人交互。

总结一下：

- 部署后获得：
  - 合约地址（用于在区块链上找到并交互合约）
  - 部署交易哈希（方便在区块浏览器追踪这次部署）
- 编译时生成：
  - ABI（用于链下应用程序理解和调用合约功能）
  - bytecode（真正部署到链上的代码）
  - metadata（记录编译器版本、源码哈希、ABI 等，用于验证和工具集成）

1. 智能合约常见组件有哪些？（变量、函数、事件等）

Solidity 最新版本里已经“升格为一等公民”的组件，比如 custom errors、struct/enum、特殊函数 receive/fallback 等。

2. 状态变量（State Variables）

定义：状态变量是存储在区块链上的数据，它们构成了合约的永久状态。一旦部署，这些变量的值就会被记录在区块链上，并且在合约的生命周期内持续存在（除非被合约的函数修改）。

特点：

- 持久性：它们的值不会在函数调用结束后消失，而是永久存储在区块链上。
- 可见性：可以有 public、internal 和 private 等不同的可见性修饰符，决定了谁可以访问这些变量。
  - public：会自动生成一个同名的 getter 函数，允许外部读取。
  - internal：只能在合约内部和继承合约中访问。
  - private：只能在当前合约内部访问。

在 Solidity 0.8.x 之后，还常见使用 constant / immutable 修饰状态变量，用于定义编译期或部署期确定、之后不可修改的常量，有利于节省 gas。

3. 函数（Functions）

定义：函数是智能合约中执行操作的代码块。它们可以修改合约的状态变量，也可以执行计算并返回结果。

特点：

可见性：

- public：可以从合约内部、继承合约或外部调用。
- external：只能从外部调用，不能从合约内部直接调用（但可以通过 this.functionName() 间接调用）。通常用于对外接口函数。
- internal：只能从合约内部或继承合约中调用。
- private：只能从当前合约内部调用。

状态可变性：

- pure：不读取也不修改合约状态的函数。
- view：读取合约状态但不修改状态的函数。
- payable：可以接收以太币的函数。
- 默认（无修饰符）：可以修改合约状态的函数。

从 0.6.0 起，Solidity 对 receive() 和 fallback() 这两个特殊函数做了更清晰的语义区分：

- receive()：专门处理“只带 ETH、不带 calldata”的转账；
- fallback()：处理“调用不存在的函数”或合约没有 receive() 时的 ETH 转账。

4. 事件（Events）

定义：事件是智能合约向区块链日志中“广播”消息的方式。它们是轻量级的、不可修改的，并且是智能合约与外部应用程序（如 DApp 前端、区块链浏览器等）进行通信的主要机制。

特点：

- 日志记录：事件数据存储在区块链的日志中，而不是合约的状态存储中。这使得它们比状态变量的存储成本更低。
- 可监听性：外部应用程序可以“监听”特定的事件，并在事件被触发时执行相应的操作。这对于实时更新 DApp 界面、通知用户或触发其他链下服务非常有用。
- 历史记录：事件提供了合约操作的历史记录，方便审计和追踪。

实践中常用 indexed 关键字对事件参数建立索引（最多 3 个），方便前端和区块浏览器按地址、tokenId 等字段筛选事件日志。

5. 构造函数（Constructor）

定义：构造函数是一种特殊类型的函数，只在合约部署时执行一次。它通常用于初始化合约的状态变量，例如设置合约的所有者、初始值或部署时的配置参数。

特点：

- 一次性执行：部署后不能再次调用。
- 可选参数：可以在部署时接受参数。
- 无名称：在 Solidity 中，构造函数与合约同名（旧版本）或使用 constructor 关键字（新版本）。现在推荐统一使用 constructor 写法。

6. 修饰符（Modifiers）

定义：修饰符是用于修改函数行为的可重用代码块。它们可以在函数执行前或执行后插入逻辑，常用于实现访问控制、条件检查等。

特点：

- 代码复用：避免在多个函数中重复相同的检查逻辑。
- 安全性：有助于强制执行安全策略，如只有合约所有者才能执行特定操作（典型如 onlyOwner）。

7. 结构体 / 枚举（Struct / Enum）与自定义错误（Custom Errors）

这些在 Solidity 最新版本中已经被写进“合约结构”的官方列表里，与状态变量、函数、事件、修饰符等并列。

Struct / Enum:

- struct：定义复合数据结构，如订单、用户信息等；
- enum：定义有限集合的枚举值，比如状态机中的 Pending/Active/Cancelled。

它们都可以作为状态变量、函数参数和返回值使用，帮助你构建更清晰的业务模型。

自定义错误（Custom Errors）：

- 自 Solidity 0.8.4 起，可以定义 error 类型来自定义错误信息，与 revert 配合使用。
- 相比传统的 require(..., "string")，自定义错误在 gas 上更省，而且可以携带结构化参数。

示例（官方文档也采用类似写法）：

```
error InsufficientBalance(uint requested, uint available);

function withdraw(uint amount) public {
    if (amount > balances[msg.sender]) {
        revert InsufficientBalance({
            requested: amount,
            available: balances[msg.sender]
        });
    }
    // ...
}
```

8. 小结

在智能合约开发中，字节码和 ABI 仍然是最重要的两个编译产物：

- 字节码是你部署到区块链上的“程序本体”；
- ABI 是告诉其他程序如何与你的合约“对话”的“说明书”。

五、合约部署的成本核算

部署智能合约，本质就是在链上“上传一段程序 + 初始化状态”，整件事消耗的是 Gas，最后换算成 ETH 成本。

可以拆成两个维度看：

1. 消耗了多少 Gas（Gas Used）？
2. 每单位 Gas 要付多少价钱（Gas 价格 / Fee）？

1. Gas 消耗量（Gas Used）——到底在为哪些操作付费？

Gas Used 反映的是部署过程中，EVM 实际执行了多少“指令”和“写入存储”。

对于一次“部署合约”的交易，Gas 消耗大致由这些部分组成：

- 基础交易成本（Base transaction cost）
  - 任意一笔交易都有的固定成本：21,000 Gas。
- 合约创建开销（CREATE 成本）
  - 使用 CREATE 指令创建新合约本身有一笔额外费用，约 32,000 Gas。
- 字节码存储成本（Code deposit）
  - 部署时要把合约的 runtime bytecode 写入链上代码存储。
  - 当前规则：约 200 Gas / 每 1 字节 runtime 代码。合约越大，这块越贵。
- 构造函数执行（Constructor 执行逻辑）

- constructor(...) 里做的所有运算、条件判断、事件等，都会按实际执行的指令消耗 Gas。
- 如果在构造函数里做了复杂计算、for 循环、创建数组等，Gas 会直线上升。

- 初始存储写入（Storage 初始化）
  - 部署时第一次把某个存储槽从 0 → 非 0，每个槽大约 20,000 Gas，属于非常贵的操作。
  - 典型的如：在构造函数里把很多 mapping、数组、配置参数一次性写入 storage。

一个很粗略的量级：

一个“中等复杂度”的合约（带若干状态变量 + 一些初始化逻辑），部署总 Gas 常在 数十万到数百万 Gas 之间。

一些大型 DeFi 协议 / Proxy 体系的实现合约，部署 Gas 能轻松破 1,000,000+。

小结：

- 代码越长、构造逻辑越复杂、初始化写入的状态越多，Gas Used 越高。
- 想省钱，第一位是：减少字节码体积 + 减少部署时的 storage 写入 + 精简 constructor。

2. Gas 价格（现在不再是单一 GasPrice，而是 Base Fee + Priority Fee）

“GasPrice”那一套，这是 EIP-1559 之前 的模型。现在主网已经长期运行在 EIP-1559 之后的机制：

总费用公式：

Total Fee = Gas Used × (Base Fee + Priority Fee)

- Base Fee（基础费）

- 每个区块由协议自动给出，是一个“当前网络拥堵程度”的函数。
- 这部分会被 直接销毁（burn），不支付给验证者。

- Priority Fee（小费 / Tip）
  - 你愿意额外给验证者的小费，用来提高自己交易的优先级。
  - 真正进验证者口袋的是这一部分。
- 你在交易里实际设置的是：
  - maxFeePerGas：你愿意为 (Base + Priority) 支付的最高单价
  - maxPriorityFeePerGas：你愿意支付的最高手续费小费
  - 节点会保证：

实际支付单价 ≤ maxFeePerGas，多出来的会退还（BaseFee 由协议决定）。

当前主网的 Gas 价格波动非常大：

- 平时可以在 不到 1 Gwei（极度清净时甚至 0.1 Gwei 左右）
- 宏观统计上，2025 年主网平均每笔普通交易的费用约在 几美元量级（比如 ~$3–4 美金），在 NFT/牛市活动高峰则可能冲到 $5–50 一笔。

3. 部署成本怎么计算？（公式 + 例子）

总成本（ETH） = Gas Used × (Base Fee + Priority Fee)

假设：

- 你的合约部署总共用掉 1,000,000 Gas；
- 当前区块：
  - Base Fee = 5 Gwei
  - Priority Fee = 1 Gwei（给验证者的小费）
- 那么：

  总单价 = 5 + 1 = 6 Gwei

  总费用 = 1,000,000 × 6 Gwei = 6,000,000 Gwei = 0.006 ETH

再乘以当时的 ETH 价格，就能得到法币成本（人民币/美元）。

注意：

- 合约稍微复杂一点，部署 Gas 很容易 从几十万变成几百万；
- 网络一拥堵，Base Fee / Priority Fee 也会瞬间抬高好几倍；
- 所以部署成本可能从 几美元 → 几十美元 → 上百美元 都是常见区间。

4. 影响部署成本的关键因素（更新版）

1. 合约复杂度 & 字节码体积
    - 代码越多、逻辑越复杂、构造期写入的 storage 越多，GasUsed 越高（直接乘以单价 => 更贵）。
2. 网络拥堵程度（Base Fee）
    - DeFi 热点、NFT 铸造高峰时，Base Fee 会暴涨；
    - 反之在闲时（例如欧美夜间），Base Fee 会很低，此时部署最划算。
3. 你设置的小费（Priority Fee / maxPriorityFeePerGas）
    - 想快速确认就多给点；不急可以适当压低。
4. 在哪条链部署：L1 与 L2 差异
    - 主网 L1（Ethereum Mainnet）
        - 安全性最高、但费用也最高。
    - Rollup / L2（如 Arbitrum、Optimism、Base 等）
        - 费用一般比主网低一个数量级以上，尤其在 2024 年 Dencun 升级（EIP-4844）之后，Rollup 把数据打包到 blob，L1 数据成本下降了 50–90%，部署合约和交互的费用都明显变便宜。

在 L2 部署的成本结构：

- 总费用 ≈ L2 执行费（执行字节码） + L1 数据费（把批次数据提交回以太坊）
- 现在由于 blob 数据的引入，L1 数据费占比大幅下降，L2 上部署合约的成本对普通开发者更友好。

5. 实战中如何“预估 & 控制”部署成本？

实践 checklist：

1. 在本地 / 测试网先部署一遍
    - 用 Hardhat / Foundry / Truffle 在本地链或测试网部署，查看 gasUsed ；
    - 工具一般会在日志里打印类似 Contract deployment: 1,123,456 gas。
2. 上线前看下实时 Gas
    - 上 mainnet / L2 之前，用 Etherscan / gas tracker 看当前 Base Fee 和建议 Priority Fee。
3. 优化代码减少字节码和 storage 写入
    - 把大数组、固定配置从构造函数写死改为后续按需写入 / 配置；
    - 用 library / 复用逻辑减少重复代码；
    - 开启编译优化（Solc optimizer），在字节码大小和执行 Gas 之间找平衡。
4. 优先考虑在 L2 首发 / 做测试
    - 如果合约逻辑可以在 L2 跑，先在 Rollup 上部署，节省大量费用。

六、合约常见安全漏洞与防范措施

重入攻击（Reentrancy）：

合约在更新状态之前进行外部调用时，攻击者可通过回调反复进入合约，反复提走资产（典型如 TheDAO、各种“取款函数先 transfer 再扣余额”的模式）。

防范：

- CEI 模式（Checks → Effects → Interactions）：先做参数检查与权限校验（Checks），再更新内部状态（Effects），最后才进行外部调用（Interactions）。
- 使用 ReentrancyGuard 的 nonReentrant 修饰符（如 OpenZeppelin 提供的实现），阻止同一函数在执行过程中被递归重入。
- 采用 Pull-Payment 提现模式：不在函数里“主动打钱”给用户，而是记录可提取余额，让用户单独调用提现函数领取 ETH/代币，从架构上隔离外部交互。

1. 发送 ETH / 外部调用的安全性

在 Istanbul（EIP-1884）等升级后，一些 opcode 的 gas 成本被上调，transfer() / send() 固定 2300 gas 津贴变得不再可靠：一旦接收方 fallback 逻辑略微复杂，就可能因为 gas 不够而失败，甚至形成 DoS 风险（资金被永久卡在合约里）。

因此社区已经普遍不再推荐依赖 transfer() / send() 作为“安全发送 ETH”的手段。

防范：

使用低级调用模式：

```
(bool ok, ) = payable(to).call{value: amount}("");
require(ok, "ETH send failed");
```

或使用 OpenZeppelin 的 Address.sendValue 辅助函数，务必检查返回值。对批量打款、退款等高风险场景优先使用 Pull-Payment（提现）模式，把“我给你打钱”改为“你来领钱”。

2. 整数溢出 / 下溢（Integer Overflow / Underflow）

在早期 Solidity 版本（<0.8.0）中，整数溢出会“绕回”（wrap），攻击者可

以利用这一点伪造余额或绕过检查。

自 Solidity ≥ 0.8.0 起，整数运算默认带有溢出检查，发生溢出/下溢会自动 revert，大幅降低此类漏洞风险。

防范：

- 使用 Solidity 0.8.x 及以上版本，默认就有溢出检查，一般无需再引入 SafeMath。
- 仅在已经严格证明安全的场景下使用 unchecked { ... } 代码块做 gas 微优化，并做好测试与代码注释。

3. 访问控制错误 & 滥用 tx.origin

缺乏权限管理或使用错误的鉴权方式（例如 tx.origin）会导致任意地址都能调用敏感函数、转走资产或篡改关键参数。

防范：

- 使用 Ownable / AccessControl（OpenZeppelin）来管理角色：如 onlyOwner、onlyRole(ADMIN_ROLE) 等，关键操作建议叠加多签、时间锁（Timelock）。
- 禁止使用 tx.origin 做权限判断，只用 msg.sender：tx.origin 在跨合约调用场景中容易被钓鱼合约利用。

4. 不可信外部调用（Unchecked External Calls）

低级调用（call / delegatecall / staticcall）若不检查返回值，或者假定对方一定按标准返回，容易造成状态与预期不一致，甚至“以为转账成功其实失败”。

防范：

- 所有低级调用都应显式检查 success 与返回数据，在失败时合理回退或处理。
- 与 ERC-20 交互时，使用 OpenZeppelin 的 SafeERC20，兼容那些“不按标准返回 bool”的代币（比如老 USDT），避免因返回值不规范导致的漏洞。

5. 预言机操控（Oracle Manipulation） / 闪电贷联动

如果合约直接使用某个 DEX 的瞬时价格、或单一预言机作为核心价格输入，攻击者可以通过大额交易 / 闪电贷操纵价格，进而在清算、抵押、铸造等逻辑中获利。

防范：

- 使用 Chainlink 这样的多节点、去中心化价格预言机，避免单点喂价。
- 采用 时间加权平均价格（TWAP） 或滑动平均，避免使用单个区块内的瞬时价格。
- 对关键交易加入 价差限制 / 最大滑点、最小回报（如 minAmountOut）等保护。
- 高风险场景中考虑增加 延迟执行窗口 或多阶段确认，降低单块操纵的攻击面。

6. DoS 与 Gas 相关攻击

如果合约在循环中对大量地址转账、或在关键路径上对外部合约进行调用，一旦某个地址/合约刻意 revert，就可能导致整个函数执行失败，从而形成拒绝服务（DoS）。

同时，合约若依赖“固定 gas 津贴”或没有考虑未来的 gas repricing（类似 EIP-1884），可能在协议升级后突然变得不可用。

防范：

- 避免无上限的 for / while 循环，特别是其中包含外部调用的情况；需要批处理时尽量拆分为多笔交易，或引入“分页处理”机制。
- 不要在循环中同时对多个用户“主动打钱”，改为 记录余额 + Pull-Payment 提现 模式。
- 对潜在失败的外部调用，明确设计失败策略（忽略/跳过/记录）而不是盲目 require(success)。

7. 业务逻辑缺陷（Logic Errors）

即便语法和安全模式都正确，如果价格公式、清算规则、利率模型、手续费分配等业务逻辑设计有漏洞，同样可能被攻击者套利或完全摧毁协议。

防范：

- 编写覆盖边界条件的 单元测试，包括极端输入、边界值、错误路径。
- 使用 模糊测试（Fuzzing） 和 不变式测试（Invariant Testing） 来验证“协议永远应成立的条件”（比如不会出现负余额、总供应不会凭空增加等）。基于 Foundry / Echidna 的不变式测试已经逐渐成为主流做法。
- 对高价值协议，引入第三方审计与（必要时）形式化验证。

8. 随机数不安全（Insecure Randomness）

直接使用 block.timestamp、block.number、blockhash 或现在的 block.prevrandao（合并后）当随机数，会受到矿工 / 验证者一定程度的操纵，在抽奖、NFT 抽卡等场景中过于脆弱。官方文档也明确指出这些值不能单独用作安全随机源。

防范：

- 通过 Chainlink VRF 等可验证随机数服务获取随机数，确保随机源不可被单方操控。
- 对“开奖类”活动考虑多轮承诺与揭示（commit–reveal）方案，避免单区块内就能决定结果。

9. 可升级 / 代理合约风险（Upgradeability）

可升级合约通常通过代理（Proxy）+实现（Implementation）的模式构建：用户与 Proxy 交互，逻辑通过 delegatecall 委托到实现合约。若存储布局出错、实现合约未初始化或升级权限滥用，都容易产生不可逆灾难。

另外，SELFDESTRUCT 的行为在最近的升级中发生了重要变化：

- 在 Dencun 升级引入的 EIP-6780 中，SELFDESTRUCT 被限制为只有在“合约创建的同一交易中”才会真正删除账户；在其他情况下，仅清空余额但不再删除代码与存储。
- 这意味着过去依赖 selfdestruct 来“实现可升级 / 清除状态”的玩法已经基本失效，新设计应避免依赖它。

防范：

- 使用成熟的 OpenZeppelin Upgradeable 模板，严格遵守存储布局规则；
- 在实现合约中使用 initializer / reinitializer 模式，并在实现合约构造函数中调用 _disableInitializers()，避免被“重复初始化”接管所有权。
- 升级入口函数（如 upgradeTo）必须只对多签 + 时间锁控制，且要有完整的审核流程（包括新实现合约的安全审计）。

10. delegatecall 到不受信目标

delegatecall 在调用者的存储上下文中执行被调合约代码，也就意味着对方可以直接修改当前合约的存储槽，是最危险的原语之一。错误使用往往直接导致资金被转光、权限被篡改。

防范：

- 严格限制 delegatecall 目标为自家合约或白名单地址，绝不允许用户可控地址被直接用于 delegatecall。
- 对所有涉及 delegatecall 的设计做好存储布局审计与测试，避免变量“串位”。

11. ERC-20 交互细节与陷阱

- 兼容“非标准”代币： 有些老代币（如早期 USDT）在 transfer / transferFrom 上不返回 bool，或在失败时 revert 而不返回 false。直接调用容易造成逻辑混乱。

→ 使用 SafeERC20，在内部帮你统一处理这些分支。

- 授权（approve）竞态：简单地从 X 改到 Y 可能被“抢在中途”利用，形成无限花费授权。

→ 优先使用 increaseAllowance / decreaseAllowance，或先 approve(0) 再设新值；支持的话可优先采用 EIP-2612 permit 减少授权交互。

12. 工具与流程（结合当前主流实践）

- 静态分析：
  - Slither：业界广泛使用的静态分析工具，可检测重入、访问控制错误、危险调用等常见模式。
  - 还有 Mythril、Manticore 等符号执行工具。
- 模糊测试 / 不变式测试：
  - Echidna：专注智能合约的 property-based fuzzing。
  - Foundry（forge）：原生支持 fuzz / invariant 测试，已经成为很多 DeFi 项目的默认测试框架之一。
- 审计与竞争审计（Audit / Code Arena）：
  - 除传统审计公司外，Code4rena、Sherlock 等“竞赛式审计”平台也被广泛采用，用于发现更广谱的逻辑问题。

13. 一页式防护清单

- 设计模式：
  - CEI（Checks-Effects-Interactions）；
  - Pull-Payment（提现）；
  - nonReentrant 防重入。
- 语言层：
  - Solidity ≥ 0.8，默认有算术溢出检查；
  - 仅在证明安全场景下使用 unchecked。
- 访问控制：
  - Ownable / AccessControl + 多签 + 时间锁；
  - 禁止使用 tx.origin 鉴权。

- 外部调用：
  - 使用 call + 检查返回值，不再依赖 transfer / send 的 2300 gas；
  - ERC-20 使用 SafeERC20。
- 预言机：
  - 使用多源预言机（如 Chainlink）+ TWAP + 阈值 / 时延保护。
- 随机数：
  - 使用 Chainlink VRF 或 commit–reveal，避免直接用区块属性。
- 可升级合约：
  - 遵循代理 + 实现合约的标准模式；
  - 注意 Dencun/EIP-6780 后 SELFDESTRUCT 行为已改变，不再依赖其“清空合约”。
- 测试与审计：
  - 单元/边界测试 + Fuzz + 不变式测试；
  - 引入专业审计和（需要时）形式化验证。

七、合约部署工具：Remix、Hardhat 等

1. Remix IDE:

类型： 基于 Web 的集成开发环境（IDE）

特点：

- 开箱即用：直接浏览器打开即可使用，无需本地安装，是官方重点维护的在线 IDE 之一。
- 多语言支持：支持 Solidity，也已经支持 Vyper 等 EVM 语言的编译与调试。
- 一体化能力：内置编译器、部署面板、调试器（可回放交易）、静态分析插件等，非常适合学习和快速原型开发。

- 多网络支持：可部署到内置的 Remix VM、本地节点，或者通过 MetaMask 直连测试网 / 主网以及其他 EVM 兼容链。
- 插件生态：通过插件系统集成 Slither 分析、Gas 预估、合约验证等扩展功能。

适用场景：入门学习、课堂演示、小型 PoC、快速验证想法、在浏览器里调试别人公开的合约。

局限：

项目一大（多文件、多库、多网络环境）时，版本管理、测试编排、CI/CD 集成都会比较吃力，更适合作为“实验台”。

2. Hardhat:

类型： 本地开发环境 & 测试框架

特点：

- “专业级”以太坊开发环境：由 Nomic Foundation 维护，定位就是面向专业 Solidity / EVM 开发者。
- 内置 Hardhat Network：提供一个功能丰富的本地链，支持 mainnet forking、opcode 级调试、堆栈跟踪、console.log 等开发体验友好功能。
- 插件体系极其丰富：与 Ethers.js、OpenZeppelin、TypeChain、Tenderly、Fireblocks 等有大量官方或社区插件，可以把部署、验证、脚本、监控串到一个工作流里。
- 脚本化部署：用 JS/TS 写 deploy 脚本，把多合约、多网络、多参数的复杂部署流程自动化；非常适合 CI/CD 和团队协作。
- 调试与测试：集成 Mocha/Chai，方便做单测、集成测试和 mainnet fork 上的“真实环境”测试。

适用场景：中大型项目、团队协作、需要 CI/CD、需要与各种基础设施深度集成（安全模块、多签库、监控平台）的“工程化开发”。

局限：

- 需要有一定的 Node.js / TypeScript 基础；
- 配置和插件很多，刚上手时会感觉“东西有点多”。

3. Foundry:

类型： 基于 Rust 的 Solidity 开发工具链

特点：

- 性能极快，已经从“新贵”变成主流之一：编译、测试、fuzz、部署都非常快，在 DeFi 协议、Rollup 团队中被广泛采用。
- 工具家族：
  - forge：项目初始化、编译、单测、fuzz 测试、部署、合约验证，一条龙搞定。
  - cast：命令行“瑞士军刀”，用于调用合约、发交易、查链上数据（包括主网 & 测试网）。
  - anvil：本地以太坊节点，支持 mainnet fork，非常适合调试和集成测试。
  - chisel：交互式 Solidity REPL，方便做小段代码实验。
- Solidity 原生测试：测试文件直接用 Solidity 写（支持 Foundry 风格的断言 / fuzz / invariant），不用切 JS/TS，逻辑更贴近合约本身。
- Gas 报告 & 优化：天然地支持 gas 报告和 diff，对追求极致 Gas 优化的协议方非常友好。
- 与其他平台联动：已经和 Tenderly、各类节点服务、区块浏览器形成良好集成，适用于专业审计/研究场景。

适用场景：希望测试 / fuzz / invariant 做得很极致、需要高性能本地链 + 主网 fork 的协议开发，或者你本来就更喜欢命令行工作流。

局限：

- 以 CLI 为主，对完全没终端经验的同学稍有门槛；
- 对 Solidity 语法和工具链的理解要求比 Remix 更高。

4. Truffle Suite（+ Ganache）：

类型： 本地开发框架 + 区块链模拟器

特点：

- 老牌框架：Truffle 是最早一代广泛使用的以太坊开发框架之一，生态和教程仍然非常多。
- Ganache 本地链：Ganache 是一个以太坊模拟器，可以在本地快速起链、配置账户余额、调试交易，非常适合作为“可视化测试链”。
- 模块化：过去还配套 Drizzle（前端库），现在更多项目会直接搭配 Ethers.js/viem 等现代库使用 Ganache 或迁移到 Hardhat/Foundry。

优势：

- 历史长、资料多，对很多旧项目来说迁移成本高，所以仍有不少存量项目在用。
- Ganache 作为“轻量级本地链 + 调试器”依然挺好用。

局限 / 现状：

- 相比 Hardhat / Foundry，Truffle 本身的更新节奏和生态活跃度已明显放缓，很多新项目更倾向直接用 Hardhat / Foundry 作为起点。

5. Tenderly:

类型： 智能合约开发 / 调试 / 监控平台（SaaS）

特点：

- 高级调试与监控：支持交易回放、断点调试、堆栈视图、变量跟踪、Gas 分析；还能对合约进行持续监控和告警。
- Fork & Virtual Testnets：可以很方便地 Fork 主网 / 测试网，或者使用它的 Virtual Testnet，在真实链数据基础上模拟部署和调用。
- 一键部署与集成：近几年 Tenderly 已不仅仅是“观察工具”，还支持通过与 Hardhat / Foundry 集成，直接在其虚拟链或测试网部署合约、模拟执行、对比不同版本行为。
- 团队协作：面向团队的仪表盘、共享调试会话、统一监控视图，非常适合多开发者/多审计方协作。

适用场景：

- 需要主网级别的数据 + 可回溯调试；
- 大型 DeFi / NFT / L2 项目在主网前做“灰度模拟”和运维监控；
- 安全团队和审计方进行交易级别的追踪分析。

局限：

- 本质上是云平台，通常作为 Hardhat / Foundry / Remix 的“增强工具”使用，而不是替代本地工具；
- 部分高级功能需要付费，适合团队或生产项目。

6. 小结：

- 刚入门 / 上课 Demo / 小实验

  → Remix 一把梭。

- 正式项目开发（中大型）

  → Hardhat（工程化 + 插件丰富） 或 Foundry（性能 + 测试 / fuzz / 显式 gas 优化），很多团队两者混用。

- 需要命令行 + 高强度测试 / fuzz / invariant

  → 强烈建议了解 Foundry，配合 anvil/mainnet fork 做真实场景测试。

- 维护老项目 / 用到 Ganache / Truffle 老生态

  → 继续用 Truffle + Ganache 也没问题，逐步向 Hardhat / Foundry 迁移。

- 想做高级调试 / 主网级模拟 / 运维监控

  → 在上面任意本地工具之上加一层 Tenderly，体验会显著提升。

八、合约部署后的公开性与可审计性

1. 合约的公开性

字节码公开：

当你部署智能合约时，你实际上是将编译后的字节码上传到区块链。这个字节码是公开可访问的，任何人都可以通过区块浏览器（如 Etherscan、Polygonscan 等）直接查看已部署合约的 bytecode，并通过“已验证合约”功能看到该字节码是否与某份源代码匹配。

交易历史公开：

所有与该合约的交互（例如函数调用、资金转移）都会作为交易记录在区块链上，这些交易记录也是公开可查的。区块浏览器会将这些交互以“交易列表”“内部交易”“事件日志”等视图展示出来，方便分析和审计。

状态变量半公开：

合约的公共状态变量可以通过区块链浏览器或 Web3 客户端直接查询。即使是 private 或 internal 变量，虽然不能通过合约提供的公共接口直接访问，但它们的数据仍然明文存储在链上状态中——如果有人知道其存储槽位（storage slot）的编码规则，理论上也可以通过低层接口读取。

换句话说：以太坊上的“隐私”更多是接口级的，而不是数据级的——“private” 只是对其他合约/调用方不可见，对链上观察者并非真正保密。

2. 合约的可审计性

由于合约的公开性，智能合约天然是可审计的，并且在真实生产环境中强烈建议进行多轮审计与形式化验证。

源码验证（Source Verification）：

开发者通常会将合约的源代码在区块链浏览器（如 Etherscan、Polygonscan 等）上进行验证：即将 Solidity 源码及编译配置与已部署字节码进行匹配，从而证明链上合约确实是由该源码编译而来。一旦源代码被验证，任何人都可以在浏览器中查看、搜索并审计合约的完整逻辑。

除中心化浏览器外，近年来也出现了去中心化合约验证服务，例如 Sourcify，可以将已验证的合约源码去中心化地存储与索引，已经被多种工具和框架集成，被多篇学术与工程实践视为“最佳实践”。

重要性：

审计是识别和修复合约漏洞的关键步骤。由于智能合约一旦部署通常不可随意更改（即使采用可升级代理，也会受到治理与权限约束），任何严重漏洞都可能导致无法挽回的资金损失（典型案例包括早期的 The DAO 事件等）。因此主流 DeFi/NFT 协议在上线前往往会进行多轮审计与公开披露。

3. 常见审计方式

1. 人工审计（Manual Review）

专业的区块链安全公司会雇佣经验丰富的安全研究员，逐行检查合约代码，推演状态机与边界条件，寻找潜在漏洞和业务逻辑错误。这仍然是最核心、最可靠的安全保证手段之一。

2. 自动化工具审计（Static / Symbolic Analysis）

使用静态分析和符号执行工具自动扫描代码，识别已知的漏洞模式与可疑行为。例如：

- Slither：Trail of Bits 维护的静态分析工具，支持数十种漏洞检测规则、可扩展脚本，以及与 Hardhat/Foundry 等现代开发框架集成，截止到 2025 年仍是业内主力工具之一。

- Mythril：由 ConsenSys 维护的字节码安全分析工具，通过符号执行与约束求解检测重入、整数问题等多类漏洞，近年的学术综述仍将其列为主流分析工具之一。

这些工具不能替代人工审计，但可以极大提高问题发现的覆盖面和效率。

3. 社区审计 / Bug Bounty：

许多项目会在源码开源并初步审计后，开启公开的漏洞赏金（Bug Bounty）计划，鼓励白帽黑客和社区研究员审查代码、提交漏洞报告，并根据漏洞严重程度给予奖励，从而形成持续的安全“众测”。

4. 小结

- 是的，合约一经部署即“公开”：
  - 字节码、交易历史、事件日志对所有人可见；
  - 状态变量在链上明文存储，只是访问接口不同。
- 是的，合约是审计且应当被审计：
  - 通过 Etherscan / Polygonscan / Sourcify 等进行源码验证；
  - 结合人工审计 + 静态分析工具（Slither、Mythril 等）提升安全性；
  - 再辅以社区审计与漏洞赏金，形成多层防线。

智能合约部署后是“透明上链”的，因此可审计性本身就是以太坊安全模型的重要组成部分——真正安全的协议，往往都是敢于公开源码、接受社区长期审查的协议。

九、合约逻辑的修改与升级模式

智能合约一旦被部署到区块链上，它的代码就无法直接修改。这是区块链“不可篡改”特性的核心体现：

- 好处：用户可以相信代码不会被开发者暗改；
- 代价：一旦有 bug 或要加新功能，就没法像 Web2 那样“在线热更新”。

也就是说：你永远不能直接“编辑已部署合约”，所有“修改”本质上都是“绕着它修”。

1. 为什么不能直接改？

如果链上代码可以随意改：

- 信任会崩溃：用户不知道你哪天会不会悄悄加后门；
- 审计形同虚设：审计的是旧代码，跑的却是新逻辑；
- 协议安全很难推理：状态 + 逻辑不断漂移，形式化验证会失去意义。

所以协议层就直接把这条路堵死了：部署就是写死。

2. 现实中是怎么“修改”的？

逻辑上只有一条线：部署新的东西 + 通过某种方式把“流量/状态”指过去。实现方式大致分几类：

3. 最直接的：简单替换（重新部署新合约）

适用：无状态 / 状态很少 / 状态可离线迁移的小合约。

做法：

1. 写好新版本合约，部署一个新地址；
2. 前端 / SDK / 其他合约 改成指向新地址；

3. 需要的话，把旧合约里的重要数据（余额、配置等）通过脚本“迁移”到新合约（往往需要用户参与或批处理脚本）。

优点：

- 实现简单、逻辑清晰；
- 没有额外的代理复杂度。

缺点：

- 地址变了：所有依赖该地址的外部系统都要更新；
- 状态迁移成本高、容易出错——尤其是有大批用户余额、投票记录、LP 头寸时。

这个做法现在主要用于工具类合约、小玩具项目、测试版，真正的 DeFi 协议一般不会这么干。

4. 行业主流：代理模式（Proxy Pattern）

这是目前 DeFi / NFT / DAO 项目里最常见的升级方式，也是 OpenZeppelin 官方推荐和支持的一整套方案（Transparent Proxy、UUPS，基于 ERC-1967 定义的标准存储槽）。

核心思想：把“地址 + 存储”锁死，把“逻辑”换着用。

4.1 基本结构

- Proxy（代理合约）
  - 地址固定：用户只和它交互；
  - 存储状态：所有状态变量都写在这里；
  - 内部只做一件事：把调用用 delegatecall 转发到当前“实现合约”。
- Implementation / Logic 合约
  - 里面是真正的业务代码；
  - 不直接被用户调用（正常不会对外暴露接口）；
  - 被认为是“可丢弃”的：升级就是换一个新实现。

Proxy 像是一间办公室，存档案 + 收客人；Logic 像是外包来的业务团

队，来去可以换，但档案永远放在办公室里不搬家。

4.2 两种主流形态

1．Transparent Proxy Pattern

- 最老牌的模式；
- 代理合约负责转发，管理员（admin）有单独的管理入口；
- 用户调用时不会意外触发管理函数；
- OpenZeppelin TransparentUpgradeableProxy 就是这一套。

2. UUPS（Universal Upgradeable Proxy Standard，基于 EIP-1822 + ERC-1967）

- 升级逻辑写在实现合约里（upgradeTo / upgradeToAndCall）；
- Proxy 本身很薄，只保存实现地址 + 必要元数据；
- 更节省 gas，也更灵活，是目前新项目很偏爱的一种。

无论 Transparent 还是 UUPS，它们都会遵循 ERC-1967 里定义的固定存储槽，避免和业务数据冲突。

4.3 升级过程长什么样？

1. 部署 v1 实现合约（LogicV1）；
2. 部署 Proxy，指向 LogicV1，初始化状态；
3. 运行一段时间后，发现需要升级：

- 部署 LogicV2（保证存储布局兼容）；
- 由 Admin（往往是多签 + Timelock）调用升级函数，把实现地址改成 LogicV2；
- 用户继续用原来的 Proxy 地址交互，但底层逻辑已经是 V2 了。

4.4 最新实践小贴士

1. 不要自己手写 proxy：直接用 OpenZeppelin Upgrades + Hardhat / Foundry 插件，它会帮你：

- 检查存储布局是否兼容；
- 限制构造函数用法，改用 initializer；
- 避免一堆微妙的坑。
- 升级权限一定要：

- 挂在 多签（Gnosis Safe 等）；
- 再套一层 时间锁 Timelock（例如 24h / 48h）；
- 大协议通常会配合治理投票一起使用，避免“项目方一时兴起改逻辑”。

5. 数据与逻辑分离 / 模块化（State-Logic Separation）

补充一些“最新标准”的关联。

基本思路依旧是：

- 一个（或少数）合约专门做 数据存储（State / Storage）；
- 逻辑放在独立的多个合约里，通过某种注册表、路由或者 Proxy+delegatecall 来拼装。

这套思想沿着发展，出现了一些更系统的标准，比如：

- EIP-2535 Diamond Standard（钻石标准）：

允许一个“钻石合约”挂接多个逻辑 Facet，通过函数选择器路由到不同模块，做出超大可升级系统，常见于复杂 DeFi / NFT/GameFi 协议。

优点：

- 单个模块可独立升级 / 替换；
- 逻辑拆分清晰，适合超大型项目。

缺点：

- 实现和审计复杂度都比普通 Proxy 高很多；
- 如果路由或存储布局搞错，事故会非常难排查。

总结：模块化 = Proxy 的更重型版本，适合“协议城市级”项目，小项目没必要一上来就玩钻石。

6. “销毁再重建”套路：在上海升级（EIP-6780）后基本退场

以前有一种比较“野”的做法：

- 利用 SELFDESTRUCT 销毁合约，清掉代码和存储；

- 然后用 CREATE2 在同一地址重新部署一个新合约（所谓 metamorphic contracts）。

这类花活本质是把“合约不可变”变成“地址可变逻辑壳”，曾经被一些项目 / 实验性质代码玩过。

但在上海升级（Shanghai）中，以太坊通过 EIP-6780 对 SELFDESTRUCT 做了重大修改：

- 现在 SELFDESTRUCT 只在“同一笔交易中新创建的合约”上才有删除效果；
- 对于已经存在的普通合约，SELFDESTRUCT 不会再像以前那样清空代码和存储，主要只会做一次 ETH 转账。

这意味着：

依赖 SELFDESTRUCT + CREATE2 做“元合约升级 / 原地换壳”这条路，在主网上基本已经被官方路线封死，不再是安全可靠的升级方案。

所以现在的业内共识是：

- 不要再用 selfdestruct 做升级或“反悔”机制；
- 即便合约里还有 selfdestruct，也要小心：它的语义已经和很多老文章里的描述不一样了。

7. 可插拔模块（Pluggable Modules）

主合约保存一个“模块列表”，不同功能拆到不同模块合约里，通过注册 / 反注册方式做“插件化升级”。

补充几个现实注意点：

- 一般会配合 接口白名单 + 角色权限，否则“模块热插拔”也可能成为攻击入口；
- 复杂协议常把“风控逻辑”、“清算逻辑”、“费率模型”做成独立模块，以实现：
  - 在线调参（换一个利率模型模块）；
  - 按市场环境更换风控（比如上线新的抵押品策略）。

8. 实战上应该怎么选？

给读者一个“考试型总结”：

1. 小玩具 / 无资金 / Demo
    - 直接重部署新合约 + 前端指向新地址即可。
2. 有资金、要长期运营的 DeFi / NFT / DAO
    - 默认用 Proxy（Transparent 或 UUPS）模式；
    - 升级权限挂在多签 + 时间锁上。
3. 超大型、模块众多的协议
    - 考虑 Diamond / 模块化架构，但要做好额外审计和工具支持。
4. 永远不要：
    - 指望“以后用 selfdestruct 把合约删了重来”——EIP-6780 已经让这条路基本失效；
    - 自己随手写 proxy、随手改存储布局又不做检查——现在已经有成熟工具帮你做 layout check，没必要拿主网资金练手。

第五章

EVM 与 Gas 机制

本章目标：理解交易执行原理与费用计算机制。

一、EVM：以太坊虚拟机

一句话（可以直接背诵）：

> EVM（Ethereum Virtual Machine）是一台运行在每个以太坊节点上的虚拟计算机，它把“交易+智能合约字节码”变成“新的链上状态”，并由 Gas 机制和确定性的执行规则，保证全网所有节点在同一输入下得到同样的结果。

1. 用一句话理解：EVM = 全网共享的“状态机 + CPU”

从协议角度看，以太坊可以抽象成一个 **状态机**：

给定旧状态 `S`（所有账户余额、合约存储等）和一组交易 `T`，通过一套确定的规则 `f`，算出新状态 `S'`。

这套规则 `f` 的具体执行环境，就是 **EVM**。它负责：

- 解释并执行智能合约编译后的 **字节码（bytecode）**
- 维护执行时的 **栈 / 内存 / 存储**
- 扣除、结算 Gas
- 生成新的状态（余额变动、storage 修改等）

所有全节点都在本地跑一份 EVM，用同样的输入重复执行同样的字节码，才保证“谁来验证区块都能得到同一个结果”。

2. EVM 里都有什么？

组件	作用	小提示
全局状态（World State）	记录所有账户（EOA & 合约）的 nonce、balance、codeHash、storageRoot	你平时看到的余额、mapping、NFT 持有关系，最终都在这里
字节码（Bytecode） & 操作码（Opcodes）	Solidity 等高级语言编译产物，由一条条 EVM 指令组成	例如 `ADD`、`SSTORE`、`CALL`、`LOG` 等，每条都有固定 Gas 消耗
栈（Stack）	计算用的“操作数栈”，最多 1024 深度	所有算术运算、条件判断都在栈上完成
内存（Memory）	交易执行期间的临时字节数组，执行结束即丢弃	适合存放中间数据、编码参数、返回值缓冲区
存储（Storage）	永久存储，每个合约独有的 key-value Trie	即我们写的 `mapping`、`struct`、`uint` 等状态变量，Gas 最贵的地方
执行上下文（msg.sender 等）	一次调用的环境：发送者、value、data、gas 等	决定权限判断、资金流向、函数选择
Gas 计费系统	为每条指令、每次存储变更计费，防止滥用	没 Gas 就 OOG（out-of-gas）并回滚执行结果

简单类比：

- `Stack + Memory` 像 CPU 的寄存器 + 内存
- `Storage` 像链上的“数据库表”
- `Bytecode + Opcodes` 像 CPU 指令集

3. 智能合约是如何在 EVM 上跑起来的？

你之前写的那条流程可以稍微抽象成 4 步：

3.1 写代码 & 编译

- 用 Solidity/Vyper 等写合约
- 编译得到：`bytecode + ABI + metadata` 等产物

3.2 部署（把程序上传到“世界计算机”）

- 一个 EOA 发起“创建合约”的交易，`data` 字段里携带部署字节码
- 节点上的 EVM 执行这段初始化代码，写入合约代码，初始化 storage
- 区块打包确认后，链上出现一个新的“合约账户地址”，里面挂着这段 bytecode

3.3 调用（执行合约函数）

- 用户或另一合约向该地址发送交易 / `CALL`，附带 `calldata`（函数选择器 + 参数）
- EVM 根据函数选择器找到对应入口，按字节码逐条执行，读写 storage、修改余额、发事件

3.4 状态更新 & 共识

- 执行结束得出新的状态 root
- 各节点都用同样规则复算一遍，结果一致则接受该区块

这一整套就是“EVM = 以太坊的执行引擎”的含义。

4. 为什么一定要用 Gas？

每一条 EVM 指令都有一个固定的 Gas 价格，比如算术运算、读写 storage、log 事件等，复杂操作如存储写入、创建合约会特别贵。

Gas 机制带来的好处：

- 防止无限循环、恶意脚本把节点“算死”
- 让复杂运算、写入操作“谁用谁付钱”
- 给打包者（验证者）定价空间（优先打包高 Gas Price 交易）

最近几年，协议层对 **Gas 计价和指令集** 也做了不少微调，例如：

- **Istanbul / Berlin / London 等升级** 重定价了某些操作（如存储访问），让 DoS 成本更高
- **2023 上海升级的 EIP-6780** 大幅限制 `SELFDESTRUCT` 的效果：大部分场景下不再真正“删除”合约，只是清理同一交易中刚创建的合约状态，避免依赖“自毁删库”的设计。

- **2024 Dencun 升级** 引入如 **EIP-5656（MCOPY 指令）** 等优化内存拷贝效率的 EVM 指令，同时通过 **EIP-4844（blob 交易）** 为 L2 降低数据发布成本，从整体上减轻执行层压力。

对你作为 Solidity 开发者来说，感知通常体现在：

- 某些操作的 Gas 变贵/变便宜
- 某些 opcode 行为发生细微变化（典型就是 `SELFDESTRUCT` 不再能随意“删合约”）

5. 确定性 & 沙盒：为什么合约不能“访问外网”？

为了让所有节点在任意地方、任意时间执行合约都得到同样结果，EVM 有两个硬约束：

5.1 确定性（Deterministic）

- 不能依赖“外部的、不固定的”信息（比如当前真实世界时间、HTTP 接口返回值）
- 能用的“随机性”只来自区块头、链上状态等有限字段（这些也并不是真随机）

5.2 沙盒隔离（Sandboxed Runtime）

- 合约代码不能访问文件系统、不能发 HTTP 请求、不能直接访问节点的磁盘
- 只能通过 EVM 暴露的少数上下文（`block.*`、`msg.*` 等）和对其它合约的 `CALL` 来互动

这就是为什么：

- 想要真正随机数，需要外部预言机（如 Chainlink VRF）；
- 想让合约“读外部世界的数据”，要靠 oracle 把数据先写进链上。

6. EVM 不只在以太坊主网：EVM 兼容链 & L2

因为 EVM 已经成为“智能合约平台的事实标准之一”，**很多公链和 L2 都内嵌或兼容 EVM**，比如：

- 以太坊主网 + 各种 Rollup（Optimism、Arbitrum、Base、Scroll、Linea、zkSync 等）
- 其它 L1 的 EVM 链，如 Avalanche 的 C-Chain 就是运行 EVM 智能合约的一条子链
- 研究界甚至在做 FPGA / 专用芯片来跑 EVM（例如 2025 年提出的 EVMx，用硬件加速 opcodes 执行）。

很多项目会强调自己是“**EVM-compatible / EVM-equivalent**”，核心含义就是：

> 你用 Solidity 写的合约、用 Hardhat/Foundry 的工具链、用 MetaMask 的钱包，几乎不改动就能部署到这些链或 L2 上。

这也是现在整个以太坊生态能“放射”到多条链、多种扩容方案上的关键原因之一。

二、EVM 为何如此重要

智能合约的基石：

EVM 是智能合约得以在区块链上运行的基础。没有它，以太坊就不可能成为一个可编程的区块链平台。官方文档也明确把以太坊描述成“在 EVM 上跑状态转移的系统”，所有节点都必须用同一套 EVM 规则来执行合约代码。

1. 统一的执行环境

所有节点（网络里的参与者）都运行 EVM 分布式地执行合约。这样，不管某个合约部署在哪个节点 / 哪个国家 / 哪个公司运行，大家都在一样的规则下执行，保证共识一致性。

这类似于“所有人都用同一个操作系统版本 / 同一台机器的指令集”来跑你的程序。如果不同机器行为不一，会造成混乱。EVM 提供了这个标准。

> 截至 2025 年，不仅以太坊主网在用同一套 EVM 语义，绝大部分以太坊客户端（Geth、Nethermind、Besu 等）都实现了同一份 EVM 规范，EVM 本身也在通过 EOF（EVM Object Format）等提案，往“更易版本化、更好管理字节码”的方向演进。

2. 支持智能合约与去中心化应用（DApps）

EVM 是支持“自动执行合约逻辑”的引擎：合约部署后，按照代码规则执行，无需人工干预。定义好条件，就能自动在某些触发条件下执行。

有了这个能力，就能做 DeFi、NFT、去中心化交易所、借贷协议、DAO 等。没有这样的通用虚拟机，很多创新很难统一落地。

> 现实里你提到的这些应用几乎都已经跑在 EVM 或 EVM 兼容链上：主流的 DeFi 协议、DEX、Lending、NFT 市场，大多在以太坊、Polygon、BNB Chain、Avalanche、Arbitrum、Optimism、Fantom 等 EVM 生态链之间多重部署，一份 Solidity/字节码可以“多链复用”。

3. 隔离性 + 安全性

EVM 为每个智能合约提供“沙箱”（sandbox）环境，这意味着每个合约的代码执行被隔离开来，一个合约被写错或被攻击，不易影响到整个网络。

使用 gas 模型防止无限循环 / 资源滥用攻击，因为每一步运算都要付 gas，交易发起者承担运算资源成本。可以防止恶意攻击或不必要浪费。

4. 去中心化共识 + 状态一致性

网络中每个节点都维护一个“状态机”（state machine），这个状态机使用 EVM 定义的规则从一个区块状态转到下一个区块状态。每个交易 / 合约调用都会使状态改变（比如账户余额、合约存储等）一致地在所有节点 / 副本上更新。

这种确定性（deterministic execution）非常关键：相同输入 / 相同合约代码 / 相同状态下，任何节点执行都会得出完全相同的输出。任何不确定性都会导致节点不同步，破坏共识。

5. 生态系统效应 + 兼容性

由于 EVM 成为了以太坊的事实标准，很多区块链都兼容 EVM /“EVM chain”出现，这样开发者可以写合约一套代码，在多个兼容 EVM 的链上部署，这极大降低了重复开发 / 迁移成本。

大量工具 / 库 / 框架都围绕 EVM 构建（Solidity、Truffle / Hardhat / OpenZeppelin、Foundry 等），使得入门门槛降低。

> 根据近期的梳理，除了以太坊本身，BNB Chain、Polygon PoS、Avalanche C-Chain、Fantom、Gnosis Chain、Tron、zkSync 等十多个 L1/L2 都选择了 EVM 兼容路线，甚至有基于比特币的侧链（如 Rootstock）直接提供 EVM 支持，让 Solidity 合约在比特币安全性之上运行。

> 与此同时，Polygon zkEVM、zkSync、Scroll 等 ZK-Rollup 项目也在追求“EVM 等价 / 等效（EVM-equivalent / EVM-compatible）”，目标是让原有 EVM 字节码几乎不用改，就能在零知识证明环境里执行，并把简短证明提交回以太坊。

6. 创新与扩展能力

基于 EVM 可以做大量创新，比如 Layer-2 Rollups、ZK EVM、EVM 对象格式（EOF）升级、Gas 模型改进等。EVM 不是静止的，而是随着升级（EIP、fork）持续发展。

这些升级既保持兼容性，又能改善性能 / 效率 / 安全。

> 目前社区规划的 Pectra 升级中，就包含 EOF（重新设计合约字节码格式，支持版本化和更安全的跳转）以及 EIP-7702 这类与账户抽象相关的改动，进一步增强合约账户能力、改善用户体验，同时仍然保留 EVM 的整体兼容性和确定性执行模型。

7. 类比

EVM 像一个全球统一的厨房里的烹饪机器 + 食谱标准：

所有人（开发者）编写“菜谱”（Solidity 合约 / bytecode） → EVM 像是一台全球各地厨房都一致的机器，根据菜谱来做菜。厨房环境相同（规则、锅具、燃气定价等），这保证每道菜做出来味道都差不多。没有 EVM 的话，每个厨房菜谱要改好多机器适配，成本高、易出错。

EVM 像一个操作系统（OS）：

在电脑里，你写程序只要遵守操作系统 API（接口）就能在不同机器上运行，不用管底层硬件。EVM 是“以太坊操作系统”。开发者写的智能合约（程序）编译成 Bytecode 后，只要链兼容 EVM，就可以运行在不同节点 / 不同以太坊兼容链上。

一句话，EVM 是以太坊“世界计算机”的大脑，也是整个 EVM 生态的通用 CPU。

8. 但它也有现实中的局限

- **Gas 成本在网络拥堵时可能很高，对用户体验不友好。**

  这也是为什么大量用户操作已经被引导到 EVM 兼容的 Rollup（Optimism、Arbitrum、Polygon zkEVM、zkSync 等）上，让 L2 负责大部分合约执行和状态更新，再把压缩后的证明或数据提交回以太坊主网。

- **性能 / 吞吐量（TPS）限制：**

  因为每个节点都要执行所有交易 / 合约调用（尤其是链上逻辑重、状态存储大时），单链 TPS 注定有限，只能把扩容放在 Rollup、分片、DA 层这些“EVM 外”的组件上解决。

- **升级兼容性压力大：**

  如果 EVM 本身或其 bytecode 格式 / opcode 改变，要兼顾已有合约不出问题，社区才会非常慎重地推进，比如 EOF 就被拆成多轮 EIP 逐步合并。

- **存储效率 / 状态膨胀问题：**

  很多合约状态积累起来，存储开销大，长期会给节点硬件门槛和同步时间带来压力，也催生了各种“状态到期（state expiry）”、“状态租金”、“Rollup + DA 链”等研究方向。

三、合约代码在 EVM 中的运行机制

1. 合约代码在 EVM 上运行的 7 个阶段

阶段	发生什么	EVM / 客户端在做什么	关键组件
1. 写合约	开发者用 Solidity / Vyper 等语言写合约：状态变量、函数、事件、构造函数等。	暂时还没出场，只是人类在写源码。	Solidity / Vyper 源代码、函数、状态变量、事件等。
2. 编译	使用 `solc`、Hardhat、Foundry 等把源码编译成 EVM 能执行的字节码：**创建字节码（含构造函数逻辑）+ 运行时代码（runtime bytecode）**。同时生成 ABI、metadata 等。	编译器把高级语言翻译成 EVM 操作码序列（bytecode），并导出 ABI 方便之后打包 calldata。	部署 bytecode、runtime bytecode、ABI、编译器版本、源映射等。
3. 发送“部署交易”	你从一个 EOA 发出一笔没有 `to` 地址的交易，`data` 字段里放的是创建字节码 + 构造函数参数。	客户端打包交易进区块；EVM 在执行这笔交易时跑创建字节码，执行构造函数、初始化存储，最后返回一段 runtime bytecode。	交易（tx）、nonce、gas、创建 bytecode、构造函数、临时 storage/memory。
4. 分配合约地址并写入代码	创建交易成功后，节点根据创建者地址 + nonce（或 `CREATE2` 的 salt）计算出合约地址，把刚刚返回的 runtime bytecode 存到这个地址的 `code` 里，同时初始化账户状态（余额、storageRoot、codeHash 等）。	EVM / 客户端更新“全局状态”：新建一个合约账户，填上 `balance/nonce/codeHash/storageRoot`：状态树被更新。以太坊整体就是一个“交易驱动的状态机”。	合约地址、code（字节码）、codeHash、storageRoot、全球状态（state trie）。
5. 合约被调用（真正“跑起来”）	之后任何 EOA 或其他合约，只要向这个地址发交易 / `call`，并在 `data` 里按 ABI 编好函数选择器 + 参数，就会触发合约执行。	EVM 读取 calldata → 根据前 4 字节找到要调用的函数 → 设置执行上下文（`msg.sender`、`msg.value`、gas 等）→ 按字节码一条条执行	calldata、stack、memory、storage、opcode、事件 logs、返回值。

		opcode：操作 stack/memory/storage，可能再 `call` 其它合约，最后更新存储、发事件、返回数据。	
6. 错误 / 回滚处理	若执行中遇到 `require` 失败、`revert`、非法 opcode、out-of-gas 等，就会回滚本次调用产生的所有状态改动。	EVM 丢弃这次执行的临时状态，把全局状态还原到执行前的样子；但已经消耗掉的 gas 不退还（防止滥用资源）。	`revert`、异常、call stack、gas 消耗、回滚语义。
7. 状态持久化与全网共识	区块被打包并通过 PoS 共识后，所有节点都按同样顺序执行该区块里的交易，得到同一个“新状态”。这个新状态就成为链的最新世界状态。	每个节点的 EVM 都执行相同的状态转移函数：`State_{n+1} = f(State_n, Block_n)`。共识只在“输入顺序和结果哈希”上达成一致；EVM 确保给定输入下结果是确定的。	区块、状态树更新、最终状态哈希、PoS 共识、finality。

2. EVM 在执行时，内部都在用哪些“部件”？

- **Code（代码）：**

部署后存放在合约地址下的 **runtime bytecode**，是 EVM 真正执行的那段程序。

- **Program Counter（PC，程序计数器）：**

指向当前要执行的 opcode，每执行完一条就移动到下一条，或根据 `JUMP / JUMPI` 做跳转。

- **Stack（栈）：**

256-bit 为单位的操作数栈，深度 1024。大部分 opcode 都只跟栈打交道（`ADD` / `MUL` / `SSTORE` …）。

- **Memory（内存）：**

交易执行期间临时用的一块线性字节数组；调用结束后就被丢弃。

比 storage 便宜，但读写仍然要收 gas，扩展内存会有额外成本。

- **Storage（存储）：**

持久化状态（状态变量）所在的地方，是世界状态的一部分，写入非常贵（`SSTORE`）。合约结束后仍然存在，直到被别的交易修改。

- **Calldata：**

外部调用传入的参数数据，EVM 只读这块区域，不会修改它。函数选择器 + ABI 编码参数都在这里。

- **Gas / gasLeft：**

每条 opcode 都有 gas 消耗；执行前交易会给出 `gasLimit`，执行中每步从 `gasLeft` 里扣。

gas 既是**计费单位**，也是防 DoS / 无限循环的硬约束：gas 用完就 `out-of-gas`，整体回滚，但已经烧掉的 gas 由发起者承担。

3. 一个类比

发动

- Solidity 源码 = 菜谱；编译 = 把菜谱翻成“机器步骤卡”（bytecode）。
- 部署 = 把这张步骤卡钉在某个厨房工位（合约地址），并把食材 / 锅具准备好（初始 storage）。
- 每次有人调用函数 = 点菜：EVM 厨师按步骤卡执行，过程中用临时案板（memory）、工具台（stack）和大仓库（storage）。
- 出锅前如果中途炸厨房（revert / out-of-gas），就把这次做菜的改动全部擦掉，但燃料钱（gas）已经烧掉了。

4. 顺带补一嘴：

- **字节码层面的格式升级（EOF）**

新的 EVM Object Format（EOF）正在作为以太坊升级的一部分推进，它把字节码拆成更结构化的段，方便静态检查和优化，比如在 Cancun / Pectra 路线里被频繁讨论。目标是让合约在保持语义不变的前提下更安全、更易分析。

- **执行层优化与硬件加速**

最近有研究把 EVM 搬到 FPGA 等专用硬件上执行（比如 2025 年的 EVMx 方案），在保证相同字节码和状态转移规则的前提下，把 opcode 执行提速数倍，说明“执行模型”本身已经稳定到可以做硬件实现。

- **针对 bytecode 的静态分析与 Gas 优化**

还有工作专门在 EVM 字节码层做静态分析，识别“不必要的写内存 / 写存储”，帮助开发者减少 gas 消耗，这类研究也在反向验证：合约的执行逻辑在 EVM 层是完全形式化和可推理的。

四、操作码（OpCode）的逐条执行

1. 先搞清楚：EVM 里的“逐条执行”到底指什么？

在 EVM 里，合约已经被编译成一串 **bytecode**，里面是一个个 **opcode + 参数** 的序列。

每个节点上的 EVM 都做一件事：

> 取下一条指令 → 解释它 → 更新栈 / 内存 / 存储 / gas / PC → 再取下一条…

也就是典型的 **fetch–decode–execute 循环**。

这一套在 Yellow Paper 里被形式化成：给定旧状态 + 当前指令 → 产生新状态的状态机转换。

所谓“逐条”，就是：

- 默认 **按字节顺序向前走**（PC++），
- 只有当遇到 `JUMP` / `JUMPI` / `STOP` / `RETURN` / `REVERT` 这些控制流指令时，才会改变程序计数器（PC）或结束执行。

2. 为什么要设计成“逐条执行”的模型？

2.1 为了共识的确定性：所有节点必须算出一样的结果

以太坊是共识系统：同一笔交易、同一份合约代码、同一份旧状态，**所有节点必须得到完全相同的新状态**，否则链会直接分叉爆炸。

- EVM 的做法是：

把整个执行过程拆成一小步一小步的 opcode，

每条 opcode 都在规范中写死了「输入 → 输出」的行为（栈变化、内存变化、存储变化、错误条件等）。

- 这样任意语言实现的客户端（Geth, Nethermind, Besu…）只要按这个规范逐条执行，就一定会得出相同结果。

如果不是逐条的精确定义，而是“高层语义自己发挥”，那不同实现就可能产生微妙差异 —— 共识直接寄。

> 逐条执行 = 把“以太坊状态机”离散成一系列可验证的小步，这是共识安全的基础。

2.2 为了精确计费 Gas：每一步都要算钱

每个 opcode 都有一个 **明确定义的 gas 成本**——比如算术运算几 gas，读存储、写存储多少 gas，call 别的合约多少起步费，再叠加动态开销等等。

- 交易发起人预先给一个 gas limit，
- EVM 每执行一条指令就扣对应的 gas，
- 一旦用完 gas，立即 `out of gas` → revert 状态（但已经烧掉的 gas 不退）。

这种「**细粒度计费**」要的就是 opcode 粒度的执行模型：

- 防止恶意无限循环 / 爆炸级资源消耗；
- 让资源消耗 **和实际执行强绑定**，不会出现“这条高层函数有时候很便宜，有时候巨贵”的不确定性；
- 也让各种 L1/L2、EVM 兼容链都可以复用同一套 gas 经济模型。

如果不是逐条执行，而是“按高级语句 / 函数块”计费，很难精确衡量资源，也更难防御 DoS。

2.3 为了实现简单 + 多客户端一致性

EVM 被设计成一个很小的、**栈机（stack machine）风格**的指令集：

- 指令集体积小、语义简单；
- 所有状态变化都通过 `stack` / `memory` / `storage` 这三样东西完成；
- 客户端实现起来相对容易，也方便做形式化验证。

你可以用 Go、Rust、Java、C++ 写 N 种以太坊客户端，只要它们都去“**逐条解释同一套 opcode**”，就能对齐。

这也是为什么 Yellow Paper 能把 EVM 写成一套比较“数学化”的状态转移定义。

> 逐条执行 ≈ 用最小共同子集来锁死语义，减少实现差异和隐藏 bug 的空间。

2.4 为了安全隔离：沙盒里每一步都可控

EVM 是沙盒环境：合约不能访问本地文件系统、网络、随机硬件，只能用 opcode 操作链上状态。

逐条执行 + 限制好的指令集，带来的好处是：

- 合约 **不可能** 调用“系统层超级指令”去绕过安全边界；
- 每一步都可以检查错误、gas、栈深度等，

一旦越界（如非法 `JUMP`、栈溢出、写 storage 超额等）立即中止并回滚；

这套模型被很多 EVM 兼容链原封不动拿去用（Polygon、Avalanche C-Chain、Arbitrum、Optimism 等），说明对安全 / 工程实践而言是“足够好且可复用的抽象”。

2.5 为了可审计、可调试、可回放

逐条 opcode 执行还有一个巨大的工程优势：**可追踪**。

- 调试器（Remix、Hardhat console、Tenderly 等）可以逐条跟踪 opcode，做“链上单步调试”；
- 安全工具（如 Slither 静态分析、各种 EVM tracer）可以还原执行路径，检查每一步状态变化是否符合预期；
- 区块浏览器 / 分析工具能实现 opcode-level trace，把一次复杂交易拆开理解。

如果只有“黑盒高级调用”，而没有 opcode 级 trace，很多安全审计工作会非常痛苦。

3. “逐条执行”≠只有顺序执行：JUMP / JUMPI / JUMPDEST

已经提到 `JUMP` / `JUMPI` / `JUMPDEST` 这些指令，再整理下它们和“逐条执行”的关系：

指令	类型	作用
`JUMP`	无条件跳转	把栈顶的值当作目标 PC，**直接跳过去执行**（前提是目标位置必须是 `JUMPDEST`）。
`JUMPI`	条件跳转	栈顶两个值：`(dest, condition)`，当 `condition != 0` 时跳到 `dest`，否则继续顺序执行下一条。
`JUMPDEST`	跳转目的标记	标记一段代码的合法跳转入口，所有跳转目标必须落在 `JUMPDEST` 上，否则视为非法，执行会失败。

所以更精确地说：

> EVM 的执行模型是：
>
> **默认顺序逐条执行 + 只有少数控制流指令可以改变程序计数器（PC）。**

这既保留了「逐条解释 + 精确计费」的好处，又支持 if/else、循环、状态机这种高层控制结构在字节码层落地。

4. 一个简单的类比

你可以把 EVM 想象成一个 **非常严谨的翻书机器人**：

- 这本书是你的合约 bytecode，每一行是一条 opcode。
- 机器人按顺序一行一行读：
  - 有的行写着「栈顶两个数相加」
  - 有的写着「从第 X 页继续读」（JUMP）
  - 有的写着「如果栈顶为 0，就跳到第 Y 页」（JUMPI）
- 每读一行就从你的“预付费卡”里扣一点 gas，
- 读错地方、跳到不存在的页码 or 卡里钱不够，都立即停机并回滚刚刚这次修改。

为什么要这么严格一行一行读？

- 因为有很多机器人（节点）全世界同步看同一本书，它们必须在每一步都做同样的事；
- 因为每一步动作都要算钱（gas）；
- 因为要防止有人写一个“无限自转”的书让机器人永远停不下来；
- 也为了之后任何人都可以回放整本书的执行过程，看它到底做了什么。

5. 小结

EVM 之所以要逐条执行 opcode，本质上是为了：

1.保证共识确定性：每条 opcode 的语义被数学化写进 Yellow Paper，所有节点逐条解释，才能在全网得到完全相同的状态转移。

2.精确计费与 DoS 防护：每个 opcode 都有固定 gas 成本，逐条执行才能细粒度计费，防止无限循环和资源滥用。

3.实现简单 & 多客户端兼容：小而明确的指令集 + 逐条执行，使不同语言的客户端实现可以保持一致，降低实现差异和安全风险。

4.安全隔离：合约只能通过受控的 opcode 改变栈 / 内存 / 存储，配合 gas 和错误检查，形成一个可控的沙盒环境。

5.调试与审计友好：逐条执行让开发者、安全工具、区块浏览器都可以按 opcode 级追踪和回放交易，利于发现漏洞、分析问题。

一句扩展：

从设计哲学上讲，EVM 就是一个“最低可用复杂度”的世界状态机：把所有行为拆成小而确定的步骤，然后用 gas 把每一步的代价标好价签，交给全网一起来执行。

五、操作码（OpCode）的 Gas 成本机制

从设计角度看，“每条 OpCode 都要付 gas”其实是在同时解决几件事：

1. 防 DoS / 防无限循环
2. 给矿工 / 验证者提供经济激励
3. 精确度量资源消耗，让“谁用资源谁付钱”
4. 给开发者一个可优化、可预期的成本模型

下面按你的原有结构展开，并穿插最新的一些升级背景。

1. 防止拒绝服务（DoS）攻击 & 无限消耗

问题：如果执行是“免费”的会怎样？

- 攻击者可以发起带有无限循环或极度复杂逻辑的交易，让所有节点一直算下去；
- 或者大量调用某些特别“重”的 opcodes（比如访问存储、外部调用），把节点 CPU、内存、磁盘 IO 吃满；
- 这样网络就会被“拖死”，正常用户的交易很难被打包，这就是链上的 DoS。

EVM 的做法：

- 每条 opcode 都有明确的 gas 成本，执行时会一条条累加；
- 每笔交易在发出时必须设置 gas limit，并预先锁定 `gasLimit * gasPrice` 的最大费用；
- 执行过程中如果 gas 用完，就 立即终止并 revert 状态（但已经花掉的 gas 不退）；

→ 这样“无限循环”在经济上变成了“无限烧钱”，不再有攻击价值。

以太坊黄皮书本身就把 EVM 定义成一个“带 gas 预算的状态机”，每个 opcode 的 gas 价格写死在协议里，并且会随着网络攻击和性能情况更新 gas 表。

现实中确实发生过 DoS 攻击：

2016 年以太坊经历过多次利用“低价 opcode”的 DoS 攻击，促成了 **EIP-150**（Tangerine Whistle）对很多 IO/存储相关指令的涨价，防止攻击者用极低成本拖慢节点。

随后又有 **EIP-1884**、**EIP-2929** 等多次“repricing”，统一目标都是：

> 哪些 opcode 真实资源消耗大，就必须涨价，否则迟早被拿来做 DoS 武器。

更前沿的研究里，2024 年 USENIX Security 的论文 **《Speculative Denial-of-Service Attacks in Ethereum》** 展示了“幽灵交易 / Speculative DoS”——攻击者利用 mempool 中大量**最终不会上链的重交易**，让节点白白执行模拟验证，也是一种新的 DoS 方向。

这进一步说明：**精细的 gas 定价 + 节点实现优化**，现在和未来都非常关键。

2. 给矿工 / 验证者提供经济激励

运行一个全节点 / 验证者，是有真实成本的：

- 计算：执行所有合约代码；
- 存储：保存状态树、历史区块；
- 带宽：同步区块、转发交易。

如果执行交易不收费，矿工 / 验证者就成了免费苦工，没人愿意长期干，网络也就失去了安全性。

Gas 的设计解决了这一点：

- 每笔交易付 `gasUsed * gasPrice`（或优先费）；
- 以太坊在 **EIP-1559** 后将手续费拆成 **base fee（随拥堵自动调节并被烧毁）+ priority tip（给出块者）**，保证既有经济激励，又改善费率拍卖体验；
- “谁让网络干活，谁就付钱”，矿工 / 验证者用手续费来覆盖运维成本并获利。

3. 精确计量资源：不同 opcode 不同价

把“**为什么不同 opcode 成本不同**”部分结合几次升级串起来：

根本原因：不同指令动用的资源差别巨大：

- 简单算术（ADD, MUL）只在 栈 / 寄存器级别操作，成本很低；
- 访问临时内存（MLOAD, MSTORE）要扩展 EVM 内存区域，有一次性线性成本；
- 访问持久存储（SLOAD， SSTORE）需要操作底层状态树（Merkle Patricia Trie），涉及磁盘或数据库 IO，成本是量级上的差异；
- 外部调用（CALL, DELEGATECALL, STATICCALL）甚至要切换上下文、创建新调用帧，计算路径更长。

因此 gas 表不是“统一单价”，而是类似：

- 纯计算：几 gas；
- 内存相关：十几到几十 gas，随内存扩展增长；
- 存储写入：上千甚至上万 gas（还有 refund 逻辑）；
- 冷存储、冷地址访问：Berlin 升级中的 EIP-2929 专门提高了“第一次访问”的成本，并区分冷 / 热访问；
- 为了稍微缓解这一点，EIP-2930 引入 Access List，让交易可以事先声明要访问的地址 / storage slot，降低首访成本。

另外一个大家经常忽略的攻击面是 “gas refund”：

历史上有项目用“写入垃圾数据再删除”的方式薅 gas refund，堆出“gas 票据”，在未来交易里一次性使用，间接制造 block gas 极端波动。

为此 **EIP-3529** 大幅削减了 `SSTORE` 和 `SELFDESTRUCT` 的 gas 退款，防止滥用。

> 一句话：gas 价格表是“活”的，它不断根据真实攻击 & 节点成本被再调参。

4. 防止无限循环与“图灵地狱”

智能合约是**图灵完备**的，理论上可以写出各种：

- 无限循环；
- 指数级复杂度的算法；
- 故意用巨量 storage / 内存的操作。

在传统软件里，你的程序死循环，最多是自己电脑卡死；

在区块链里，如果没有 gas 限制，“大家一起死机”。

GasLimit + 按步收费：

- 交易发出时必须设定 gas limit，上链前节点也会预估是否“明显不够 / 明显太大”；
- EVM 在执行过程中一条 opcode 一条 opcode 扣费；
- 发现 gas 不够就立刻终止并 revert 状态（但发送方已经付掉的那部分 gas 不退），攻击者想要无限 loop，就得先准备无限钱包。

5. 为开发者提供可优化的成本模型

有透明的 gas 成本表，开发者才能优化。

- 知道 storage 写操作最贵，就会尽量减少存储写入、设计紧凑的 struct / mapping；
- 知道 external call 成本高，就会减少不必要的合约间调用；
- 知道 memory 按大小扩展收费，就会优化数组长度、避免超大循环。

在工具层面，像 Hardhat、Foundry、Tenderly 等都会提供：

- 每个函数的 gas 报告；
- 单次调用 / 单条路径的 gas profile；
- 帮你找到那种“无意中 $O(n^2)$”的写法。

6. 进一步拆开看：不同类型 DoS 与 gas 的关系（简版）

DoS 类型可以收束成三大类，顺便和 gas 联系起来：

6.1 网络 / 基础设施级 DoS

- 典型是对 RPC、节点接口做大流量攻击，这更多属于传统 DDoS 范畴，gas 帮不了太多；

6.2 链上计算 / 状态层 DoS

- 利用“低价 opcode”、storage 写入和 refund，或极端复杂合约调用拖慢节点；
- 这类是 gas repricing 和 refund 改动的主要打击对象（EIP-150 / 1884 / 2929 / 3529 等）。

6.3 mempool / 交易选择层 DoS（例如 Speculative DoS/Ghost TX）

- 攻击者构造很多**不会最终上链但很重**的交易，让节点在“模拟执行”阶段消耗大量资源；
- 2023–2024 年的研究表明，如果节点实现不当，即使有 gas 模型，也有可能在“模拟阶段”被拖慢。

结论是：**gas 不是银弹，但不计价就根本没法玩**；剩下的要靠协议升级、客户端优化和 mempool 设计一起配合。

7. 厨房类比

- 每条 opcode 就是一道菜的一个“步骤”：切菜、炒、煮、烤；
- 厨房（EVM / 节点）提供燃气、油、电、厨具，这些都是有限资源；
- 每一步都要支付“燃气费”：
  - 简单“翻炒”很便宜（算术、栈操作）；
  - “开烤箱 2 小时”很贵（写 storage、创建合约）；
- 如果有人想一直在厨房里反复做没意义的操作，就得一直付钱，很快就破产，而不是把整个厨房拖垮。

8. 小结

> 为什么每条 OpCode 都有 gas 成本？
>
> 因为以太坊要在一个开放、无需许可、图灵完备的环境里安全地执行任意代码。
>
> 给每个原子指令定价，让：
>
> ●DoS 和无限循环在经济上变得“不划算”；
>
> ●节点执行工作有经济激励；
>
> ●真正昂贵的资源（存储 / 外部调用）被合理计费；
>
> ●协议可以通过调整 gas 表来应对新型攻击与成本变化。

这就是 EVM 能维持“世界计算机”秩序的核心机制之一。

六、Gas 计量单位：gwei 与 ETH 的关系

1. Gas 本身是什么单位？

在以太坊里，**Gas 是“计算工作量”的抽象单位**，专门用来度量一笔交易或一次合约调用消耗了多少计算、存储、带宽等资源。比如：

- 一个简单的加法操作：大约消耗几单位 Gas
- 一次 `SSTORE` 写入存储：可能要上万 Gas

也就是说：

> Gas（单位） = 资源用量；ETH / Gwei = 你为这些资源实际付的钱。

2. Gwei Wei 和 ETH 的关系

- ETH 是以太坊的主币
- wei 是 ETH 的最小单位
  - 关系：`1 ETH = 10^18 wei`
- Gwei 是中间单位（giga-wei）
  - `1 Gwei = 10^9 wei`
  - 所以：`1 ETH = 10^9 Gwei`

这个关系可以这样记：

> ETH ←(×10^9)→ Gwei ←(×10^9)→ wei

表述为：

> Gas 价格通常以 Gwei/Gas 表示，本质上就是“每单位 Gas 要付多少 Gwei（也就是多少 ETH）”。

3. Gas 费用是怎么算出来的？

> 总 Gas 费用（用 ETH 表示） = Gas 消耗量（GasUsed） × Gas 价格（GasPrice，单位 Gwei/Gas）

再加上单位换算：

```
总费用(ETH) = GasUsed × GasPrice(Gwei/Gas) ÷ 10^9
```

> 假设一笔简单转账消耗 21,000 Gas，当前 Gas 价格是 20 Gwei/Gas：
>
> ●先算总 Gwei：`21,000 × 20 = 420,000 Gwei`
>
> ●换算成 ETH：`420,000 ÷ 10^9 = 0.00042 ETH`

完全符合当前以太坊对 Gas 的计费方式。

4. 补充：EIP-1559 之后，Gas 价格不再是单一的 GasPrice

经典公式 `GasUsed × GasPrice`，这个在概念上仍然没错，但 **自 2021 年伦敦升级（EIP-1559）之后**，链上的实际收费结构变成了“三段式”：

4.1 Base Fee（基础费）

- 由协议自动计算、动态调整
- 按区块拥堵程度上下浮动，并且**会被销毁（burn）**，不会付给验证者

4.2 Priority Fee / Tip（优先费 / 小费）

- 由用户出价，作为“加小费让自己更快打包”的激励
- 这部分是 **真正付给出块者（验证者）的收入**

4.3 Max Fee（你愿意承受的最高单价）

- 交易里会带上 `maxFeePerGas` 和 `maxPriorityFeePerGas` 两个参数
- 实际支付的有效单价 ≈ `baseFeePerGas + priorityFeePerGas`，但不会超过你设置的 `maxFeePerGas`

所以在 EIP-1559 之后，更精确的“总费用”可以写成：

```
实际费用（ETH）
  ≈ GasUsed × ( BaseFeePerGas + PriorityFeePerGas )   （再按 Gwei→ETH 换算）
  ≤ GasUsed × MaxFeePerGas
```

但从**教学直觉**上，原本那句：

> “总 Gas 费用 = GasUsed × GasPrice”

依旧可以认为是：

“GasPrice”=“你最终实际付出的每单位 Gas 的有效单价（Base + Tip）”，只是现在这“单价”是按 EIP-1559 机制动态算出来，而不是简单由你一个值说了算。

5. 小结

- Gas：抽象的计算单位，衡量一次交易/合约调用用了多少资源；
- Gwei：Gas 单价最常用的计价单位，`1 Gwei = 10^9 wei = 10^-9 ETH`；
- ETH：最终你实际支付的费用单位，`费用(ETH) = GasUsed × 有效 Gas 单价(Gwei) ÷ 10^9`；
- 伦敦升级之后，**单价由 Base Fee + Priority Fee 组成**，并受 `maxFeePerGas` 上限约束，但对入门理解来说，你可以继续用“GasUsed × GasPrice”来记住费用是怎么来的。

七、London 升级：BaseFee、销毁、Tip 等改变

1. London 升级整体在干什么？

- 发生时间：以太坊主网在 **2021-08-05**（区块高 12965000）激活 London 升级，其中最核心的就是 **EIP-1559：新的手续费市场机制**。
- 同时还包含 EIP-3198（BASEFEE opcode）、EIP-3529（gas 退款调整）、EIP-3541、EIP-3554 等配套改动。

之前是“**第一价格拍卖**”：

- 你自己拍一个 `gasPrice`，矿工挑价高的先打包。
- 结果：**费率极度难以预估、波动巨大**。

London/EIP-1559 改成“**协议自动给出底价 + 用户只调小费**”的模式：

- 协议计算一个全网统一的 **Base Fee（基础费）**。
- 这部分 **必须支付，并且会直接被销毁**。
- 用户只需要根据自己想要的速度调一个 **Tip（priority fee/小费）** 激励打包者。

2. Base Fee：自动调节的“底价”

Base Fee 是什么？

- 每个区块都有一个 `baseFeePerGas`，表示：

> “在这个区块里，每 1 Gas 至少要付多少 Gwei，才能被包含进来。”

- 所有被打包的交易都要按这个 Base Fee 付费；
- 这部分费用 **不会给矿工 / 验证者，而是直接销毁（burn）**。

Base Fee 如何自动调节？

EIP-1559 定义了一套简单的调节规则：

- 每个区块有一个 **目标 gas 使用量 targetGas**（大约等于区块 gas 上限的一半）。
- 如果某个区块：
  - `gasUsed > targetGas` → 表示很拥堵 → **Base Fee 上调**；
  - `gasUsed < targetGas` → 需求偏少 → **Base Fee 下调**。
- **每个区块 Base Fee 的调整幅度最多 ±12.5%**，避免上下波动过猛。

直观理解：

> 区块长期“过满”，Base Fee 会越来越贵，直到用户开始“用不起”为止，需求被压下去；
>
> 区块经常“半空”，Base Fee 会被调得越来越便宜，鼓励更多人打交易。

3. Base Fee 销毁（Burn）：ETH 供应的“刹车”

支付的 Base Fee 去哪了？

- 旧机制：**所有手续费都归矿工 / 出块者**。
- EIP-1559 之后：
  - **Base Fee 部分：直接在协议层销毁（burn）**；
  - 只有 **Tip（优先费） + 区块奖励** 归矿工 / 验证者。

这带来两个重要影响：

3.1 ETH 变成“带使用费的资产”

- 每一笔交易都会直接把一小部分 ETH 永久烧掉。
- 网络越繁忙，烧掉的越多。
- 各家统计显示，自 EIP-1559 上线后，已经累计销毁了 **数百万枚 ETH**（2024 年中就已超过 3.7M ETH，被多篇文章引用自 ultrasound.money）。

3.2 配合合并后的低发行率，有时呈现“净通缩”

- Merge 之后，给验证者的 ETH 发行速度大幅下降；
- 当一段时间内 **burn > 发行** 时，ETH 总供应会短期呈现净通缩。

4. Tip（Priority Fee）：给验证者的“小费”

因为 Base Fee 被烧掉了，出块者（以前是矿工，现在是 PoS 验证者）仍需要激励，所以引入 **Priority Fee（Tip，小费）**：

- 交易结构里新增：
  - `maxFeePerGas`：你愿意为“（BaseFee + Tip）”支付的上限；
  - `maxPriorityFeePerGas`：你愿意给出块者的小费上限。
- 实际支付：
  - 实际支付单价 `= BaseFee + priorityFee`（priorityFee 不超过你设置的 maxPriorityFee）。
  - `BaseFee` 被销毁；
  - `priorityFee` 归出块者（矿工/验证者）。

用户怎么用？

- 大部分钱包会帮你：自动读取当前 Base Fee，然后推荐一个 tip（比如 1–3 Gwei，或者“慢 / 标准 / 快”几档）。
- 你只要接受“估算价格”就行，不再需要自己盲猜 `gasPrice`。

5. 区块“弹性容量”：Gas Target + 2 倍上限

EIP-1559 还顺带改了一个非常关键但常被忽略的点：

- 每个区块不再有一个固定的“硬容量”，而是：
  - 有一个 **目标 gas 使用量 targetGas**；
  - 区块可以临时扩展到 **2 × targetGas**。
- 当短时间内交易突然激增时：
  - 区块可以临时“吃下”更多交易（最多 2 倍目标容量），缓和拥堵；
  - 同时 Base Fee 会显著上调，逐渐把需求压回 target 附近。

类比：

> 以前电梯一次最多站 10 个人，后面排队多少都得等。
>
> 现在电梯目标是 10 人，但在特别拥挤时允许挤到 20 人上去，同时门口票价自动变贵，让“不着急”的人自觉晚点再来。

6. 新交易类型 & 对开发者的改动

London 升级对开发者视角也带来几项重要改变：

6.1 新的交易类型（type-2 / EIP-1559 交易）

- 旧式：只有一个 `gasPrice` 字段；
- EIP-1559 之后：
  - `maxFeePerGas`（交易愿付总上限）
  - `maxPriorityFeePerGas`（小费上限）
- 节点根据当前 Base Fee 自动算出真实支付价格。

6.2 BASEFEE opcode（EIP-3198）

- 合约里可以通过新 opcode 读取当前区块 Base Fee；
- 一些协议用它来做动态收费、预言机安全补充等逻辑。

6.3 Gas Refund 改动（EIP-3529）

- 大幅削弱 / 移除了某些过去可以“薅羊毛”的 gas refund 模式（例如依赖 SELFDESTRUCT / 清空存储的 GasToken 模式）；
- 目的是防止恶意膨胀状态、滥用 refund 造成 DoS 风险。

7. London 升级之后，从不同角色看有什么变化？

对普通用户：

- 不再手动猜 `gasPrice`，钱包基于 Base Fee 给出推荐；
- 费用波动依然有，但 **可预测性显著提升**。

对 dApp / 开发者：

- 需要适配新的交易参数（`maxFeePerGas` / `maxPriorityFeePerGas`）；
- 有了 BASEFEE opcode，可以在合约内感知 fee 水平；
- 设计经济模型、MEV 抽取、清算等逻辑时，要考虑 fee burn 对长期收益的影响。

对 ETH 本身：

- 手续费的一部分被“自动销毁”，配合合并后的低发行率，ETH 从“纯通胀资产”变成“**视网络使用情况而定的通胀/通缩混合资产**”。

八、Gas 机制：防止网络攻击与滥用

一句话（可以直接背诵）：

以太坊把“计算和存储”全部用 **Gas 计价**，再用 **交易 gas limit + 区块 gas limit + 费用市场（BaseFee + Tip）**把攻击者锁在一个非常“贵”的沙盒里——你可以攻击，但每一秒都在真金白银地烧 ETH，而且能造成的伤害是被硬限制的。

1. 每条 OpCode 都要付费：算力直接变成“成本”

> 核心目的：让 DoS 攻击不再是“免费玩网络资源”。

- EVM 把所有操作拆成最小指令（OpCode），比如：
  - `ADD` / `MUL`：算术运算，Gas 便宜；
  - `SLOAD` / `SSTORE`：读写存储，Gas 很贵；
  - `CALL` / `DELEGATECALL`：跨合约调用，也不便宜。
- 每条指令执行时都会消耗预先定义好的 Gas：
  - 简单计算 → 低 Gas；
  - 访问持久存储 / 改状态 → 高 Gas；
  - 创建合约 / 写大量 storage → 更高 Gas。

效果：

- 想滥用“重操作”（写很多存储、部署海量合约）？

👉 你要为每个写操作付钱，而且单价不低。

- 想写一个计算量巨大的函数反复调用？

👉 每次调用都要在链上烧 Gas，本质上是自己给全网“打赏算力”。

> 历史上，以太坊在 2016 年几次遭遇过利用“低估价 OpCode”的 DoS 攻击，后来通过升级提高这些指令的 Gas 成本（比如重定价 EXTCODESIZE、SLOAD 等）来修补漏洞——说明 Gas 定价本身也是一个不断调优的安全参数。

2. 交易有 Gas Limit：禁止单笔交易“无限循环”

核心目的：防止“单笔交易把节点拖死”。

- 发交易时必须指定一个 **gas limit**：
  - 表示“最多愿意为这笔交易消耗多少 Gas”。
- EVM 在执行过程中：
  - 每执行一步就从剩余 Gas 中扣除对应的成本；
  - 一旦 Gas 用完，立刻 `revert`：
    - 所有状态改动全部回滚；
    - 已经消耗掉的 Gas 不退。

效果：

- 想在合约里写 `while (true) { ... }` 死循环？

只要消耗完交易提供的 gas，EVM 直接终止，状态回滚。

攻击者最多烧掉 **自己这笔交易愿意付的那点 gas**，不能“无限白嫖运算力”。

- 状态的“原子性”：“要么全部成功，要么全部回滚”，保证不会只修改半拉状态导致链上数据紊乱。

3. 区块也有 Gas 上限：防止区块级的“塞车攻击”

核心目的：不能靠大量高 gas 的交易“一次性塞满区块”、拖垮整网。

- 除了每笔交易的 gas limit，还有一个 **区块级别的总 gas 上限**（Block Gas Limit）。
- 矿工 / 验证者在打包交易时，所有交易的 `gasUsed` **总和不能超过这个上限**。
- London / EIP-1559 之后，引入了“目标 gas 使用量 + 允许最多 2 倍目标”的机制，但本质上仍然有 **上限**，只是变成一个更弹性的目标/上限体系。

效果：

- 即使攻击者拍下整个区块也只是那一个区块的事：
  - 对其它区块的影响，被限制在“出块速度 ＋gas 上限”范围内；
  - 而攻击者要真这么干，每个区块都要付出极高的费用。
- 对正常用户来说，区块 gas limit＋BaseFee 的调整机制，使网络在拥堵时自动提高费用，抑制无意义交易。

4. Gas 价格（BaseFee＋Tip）让“Spam 变得很贵”

> 核心目的：经济上让垃圾交易不划算。

在 London 升级（EIP-1559）引入的新费率模型中：

- **Base Fee（基础费）**
  - 由协议自动调节，根据最近区块的拥堵程度（gas 使用量是否超过 target）上下浮动；
  - 这部分费用被直接 **销毁（burn）**，不会送给出块者。
- **Tip / Priority Fee（优先费）**
  - 用户给打包者的小费，以获得更高的优先级。
- **Max Fee（`maxFeePerGas`）**
- 用户愿意为单位 gas 支付的上限，`有效费用 = BaseFee + Tip`，且 ≤ `maxFeePerGas`。

对滥用 / 攻击的影响：

- 想塞满区块或持续 spam？
  随着你自己推高区块利用率，**BaseFee 也会越来越贵**，而且这部分是直接烧掉，不给矿工，也不给你自己。
- 攻击者要想维持长时间的大规模垃圾交易：
  - 不仅要持续出高价，
  - 而且还在不断减小 ETH 总供应（通过 burn），给自己“反向抬价”攻击成本。

5. 状态膨胀 & Gas Refund 的“反滥用”升级

> 核心目的：防止“用 Gas 机制本身来薅羊毛 / 攻击状态”。

早期以太坊有 **Gas Refund** 机制：某些操作（比如 `SELFDESTRUCT` 合约或把存储槽设回 0）可以退回部分 Gas，初衷是激励“清理垃圾状态”。

后来发现有人专门铸造 “Gas Token”，用一堆合约 / storage 把状态写满，在高 Gas 时清理来拿 refund，相当于“预存低价 Gas、在高峰期取出来用”，同时也带来状态膨胀和潜在 DoS 面。

为此，以太坊做了几次升级：

- **提高某些存储相关操作的 gas cost**（防止低估、被滥用）；
- **削减 /移除大额 Gas Refund**，尤其是 `SELFDESTRUCT` + 部分 `SSTORE` 的退款；
- 把状态膨胀和退款滥用的空间大幅压缩。

结果：

- 想通过大量创建 / 销毁合约、频繁写 storage 来“薅 gas 退款羊毛”或拖慢节点？

成本显著增加，策略空间变小，变成一个效率极低、极贵的攻击方式。

6. 链上逻辑 DoS：Gas 让很多“逻辑攻击”变得昂贵

攻击 / 滥用类型	典型手法	Gas 视角下的限制
无限循环 /极端复杂计算	在合约里写死循环或超大循环、递归，试图拖死节点	单笔交易 gas limit 限死执行步数；Gas 用尽就 revert，攻击者为这段无效计算买单
交易垃圾 /Spam	发送大量无意义小交易 / 调用	每笔都有 base gas 成本（21,000 gas 起步），大量发送会烧掉非常多 ETH
滥用低价指令	不断调用早期低估 Gas 的 opcode（如早年的 EXTCODESIZE、BALANCE 等）	多次分叉中已经把这些指令重定价；未来发现低估也会再调价
状态膨胀 /Storage Bloat	写入大量存储 / 创建大量合约	`SSTORE`、创建合约的 Gas 极贵，再加上退款削弱，攻击成本成倍上升
合约逻辑层 DoS	利用循环遍历数组、外部调用回退等让某函数永远不能成功	精心设计的合约会避免“依赖外部状态 /不受控数组长度”的模式；一旦函数 gas 消耗过高，交易会自动失败而不是拖垮节点

7. Mempool / 节点策略：Gas 价格决定“谁有资格进入队列”

除了协议层面的 Gas 模型，**节点自己的策略** 也在防止滥用：

- 每个节点的本地交易池（mempool）有容量限制：
  - gas price 太低 / 长期不打包的交易可能会被踢出；
  - 所谓“无限量垃圾低价交易占满 mempool”的攻击效果有限。
- 在 EIP-1559 模型下，钱包往往会给出“推荐 BaseFee + Tip”，低于这个价格的交易很难抢到打包机会。

这个组合的效果：

- 想用极低价格大量塞交易？

大概率进不去区块，只是在少数节点的 mempool 里浪费一下存储，还会被清理掉。

- 想用高价大量塞交易？

你可以抢占区块空间，但那意味着你要**为所有这些垃圾交易付出真金白银**，而且越塞越贵。

8. 小结

可以把 Gas 机制看成一个三层防护网：

8.1 每条指令都要付钱：任何计算或存储都不再免费；

8.2 交易 / 区块都有 gas 上限：攻击者能消耗的资源是被硬性限制的；

8.3 费用市场 + 销毁机制：越想压榨网络，就越要烧掉越来越多的 ETH。

加上不断迭代的 **opcode 重定价、退款削减、状态成本调整**，Ethereum 把很多潜在的 DoS 攻击路径，从“廉价玩具”变成了“高成本赌博”。

九、交易异常与 Gas 耗尽处理

当一笔交易在执行过程中 **Gas 用完（Out of Gas）**，会触发 EVM 的“异常退出”，结果可以概括成三句话：

1. **状态全部回滚（revert）**
2. **已消耗的 Gas 费用不会退还**
3. **交易会被打进区块，但状态是“失败 / Reverted（Out of Gas）”**

下面分点细化一下这些行为。

1. 交易失败并回滚（Revert）

一旦执行过程中 Gas 消耗达到交易设定的 `gasLimit`：

- EVM 会立刻抛出“Out of Gas”异常；
- 当前调用栈（包括内部调用）中的**所有状态修改全部回滚**；
- 这符合以太坊“原子性”的设计：**要么全部成功，要么全部不生效**。

也就是说：

- 转账类交易：ETH 不会真正转出去；
- 修改合约状态：storage 不会被更新；
- 部署合约：最终链上不会出现这个新合约（只是留下了一条失败的创建交易记录）。

但**注意**：虽然状态回滚，**交易本身仍然被写入区块**，只是 `status = 0`（失败），区块浏览器会显示 `Out of gas` 或 `Reverted (out of gas)`。

2. 已消耗的 Gas 不会退还——为什么？

即使交易失败、状态回滚，**已经执行过的指令所消耗的 Gas 也不会退**，对应的 ETH 会支付给出块者（矿工 / 验证者）作为报酬。

这正是安全设计的一部分：

- 节点确实 **付出了真实算力和时间** 去执行这笔交易，直到 Gas 耗尽；
- 如果失败交易也把 Gas 退回，那攻击者就可以**免费狂丢“必然失败”的重计算交易**，消耗网络资源，形成一种 DoS 攻击；
- 让失败交易也付钱，可以把这种攻击变成“非常烧钱”的行为，从经济学上直接抑制。

换句话说：

> Gas 付的是“执行尝试成本”，不是“成功结果费”。

3. 区块浏览器中的表现

在 Etherscan、OKLink 等浏览器里查看一笔 Gas 不足的交易，你通常会看到：

- `Status: Fail` / `Reverted`
- `Error: Out of gas` 或类似错误信息
- `Gas Used = Gas Provided`（几乎把上限全烧光）
- 但：**余额、合约状态都保持在交易前的状态**（因为已经回滚）。

4. 跟 Gas 相关的两个小补充

4.1 `gasLimit` 估算的重要性

- 钱包 / 框架（MetaMask、Hardhat 等）都会在本地模拟执行，帮你估算合适的 `gasLimit`；
- 但如果合约逻辑有分支 / 外部调用，有时仍可能低估，导致 Out of Gas。

4.2 BaseFee + PriorityFee 不影响“是否 Out of Gas”，只影响“花多少钱”

- 是否 Out of Gas 只看 `gasLimit` 与**执行消耗的 Gas**；
- London 升级后的 `baseFee` & `priorityFee` 决定的是单位 Gas 要花多少 ETH。

十、降低 Gas 成本的合约代码设计

> 尽量少写 / 少改 on-chain storage，多用 calldata / memory 做临时计算，按需拆分 + 批量化操作，配合编译器优化和专业工具量化每一次改动的 gas 影响。

再补一条特别重要的：

> 能搬到 L2 的逻辑就尽量放到 Rollup 上，让主网只做“结算层”。

1. 最高优先级的几条设计原则

1.1 把昂贵的持久化写入（`SSTORE`）降到最低

- 合约吞 gas 的最大黑洞就是写 storage。
- 设计上要“读多写少”：
  - 能通过计算现场算出来的，就不要写到 storage；
  - 必须写的时候，合并多次修改为一次写（先在 memory 里算完）。

1.2 优先用 `calldata` / `memory` 做临时计算

- 外部函数参数：能 `calldata` 一律 `calldata`；
- 重复读取的 storage 值先 `cache` 到 `memory` 变量，只在最后写回 storage。

1.3 避免在链上大循环 / 扫描整表

- 尽量把“遍历、统计、排序”这种事情放到链下做，链上只验证结果（Merkle proof / bitmask / 分批执行等）。

1.4 选对数据结构和布局

- 常用的套路：`mapping` + 少量索引数组，而不是在一个大数组里扫描；
- 把多个小整数/标志位打包到同一个 32 字节 slot 里（打包时才有意义）。

1.5 所有“优化”都要被工具数字化验证

- 用 Hardhat Gas Reporter、Foundry 的 gas snapshot、Tenderly 等工具，把每次改动的 gas 增减量量化出来，防止“自我感觉良好型优化”。

2. 存储写入：最贵的一环

2.1 降低 `SSTORE` 次数

- **先算后写**

  典型模式：

```
// 差一点写法：多次 SSTORE
function bad(uint256 x) external {
    total += x;         // SSTORE
    total += 1;         // 再 SSTORE 一次
}

// 推荐：只写一次
function good(uint256 x) external {
    uint256 newTotal = total + x + 1; // 在 memory/stack 里算完
    total = newTotal;                 // 单次 SSTORE
}
```

- **集中结算**

  对一大批变更，可以让用户先离线算好差分，或通过一笔“批量结算交易”一次性写入。

2.2 Slot 打包（storage packing）

- 多个 `uint128` / `uint64` / `bool` 等紧挨着声明，Solidity 会尝试帮你打包到同一个 32 字节 slot：

```
struct Packed {
    uint128 amount;
    uint64  createdAt;
    bool    isActive;
    uint8   flag;
    // 这些加一起 ≤ 32 bytes → 1 个 slot
}
```

- **注意**：只有在“同一个 slot 被多个变量共享”的情况下，小整数才省 gas；

  在 `memory` / `stack` 里用 `uint8` / `uint16` 并不会更省。

2.3 利用事件替代部分 Storage

- 某些只用来“查历史记录、不参与状态计算”的数据可以放到事件里，而不是写入 storage：
  - 如“充值记录、操作日志”等；
  - 比如只需要 off-chain 分析的统计数据，都可以用 `event` 替代状态变量。

3. `constant` / `immutable` 与配置类变量

补充一下语义和取舍：

3.1 `constant`

- 编译期就确定的值，直接写进字节码，不占 storage；
- 典型例子：

```
uint256 public constant FEE_DENOMINATOR = 10_000;
address public constant WETH = 0x...;
```

- 对部署 gas 的节省在实战中通常非常显著，尤其是大数组 / 常量表抽出来做 `constant` 时，部分项目确实能看到 30% 以上的部署 gas 降低，不过比例取决于具体代码结构。

3.2 `immutable`

- 部署时由构造函数设置一次，之后视为只读；
- 编译器会像常量一样处理访问，而不占用常规 storage slot，非常适合：
  - 协议关键地址（Router / Factory / Oracle）；
  - 初始治理角色等。

使用建议：

- “不会变”的参数 → `constant` 或 `immutable`；
- “理论上可以升级 / 改”的关键变量，就放在 storage 并配合治理 / timelock。

4. 数据位置：`calldata` / `memory` / `storage` 策略

4.1 外部函数参数：优先 `calldata`

```
function batchTransfer(address[] calldata recipients,
uint256 amount) external;
```

- `calldata` 不会复制一份到内存，比 `memory` 便宜很多；
- 对只读的数组、字符串、bytes 参数尤其明显。

4.2 重复读取的 storage → 先缓存到 `memory`

```
function foo() external {
    Config memory cfg = config; // 从 storage 拉一份副本
    // 后续逻辑都用 cfg.xxx，最后必要时再写回
}
```

- 避免在循环中重复 SLOAD；
- 对复杂结构体尤为关键。

4.3 只在单次调用生命周期内用到的数据：`memory`

- 局部变量、临时数组、排序缓冲区等，全部用 `memory`；
- 避免把只在本次调用有用的东西写进 storage，后面又永远不用。

5. 循环、批量操作与“把活搬到链下”

5.1 尽量避免“不受控制长度”的链上循环

- 不要在链上“按地址数组 size 遍历所有用户”；
- 改成：
  - 由前端 / off-chain 服务分页调用；
  - 只对单个用户或一个小 batch 处理；
  - 或让用户自助 claim（典型“空投 Merkle Tree 领取”模式）。

5.2 链下计算 + 链上验证

- 大部分“统计 / 排序 / 配置生成”可在链下做完：
  - 用 Merkle Tree / Merkle Sum / Sparse Merkle Tree、或简单签名证明；
  - 链上只验证 proof（通常几次哈希 + 比较），gas 远低于遍历整个列表。

6. 合约级架构与新特性（含 EIP-1153）

2024 的 Dencun 升级已在主网上线，引入了 **EIP-1153 transient storage**，增加了 `TSTORE` / `TLOAD` 这类“短期存储槽”，适合存放只在单笔交易中使用的临时状态，可以显著减少对长期 storage 的依赖和状态膨胀。

对设计有几个启发：

6.1 短期状态 → transient storage

- 例如：
  - 重入锁（reentrancy guard）；
  - 单笔交易内的计数器或状态；

- 这些原本要写入 storage 的小标志位，迁移到 transient storage 后可以节约不少 gas（同时减轻全网状态负担）。

6.2 长周期状态 → 正常 storage

- 用户余额、配置、治理参数等仍然使用 standard storage；
- 但要尽量“少写多用”。

6.3 模块化 + Proxy + 库

- 你前面提过代理合约，这里从 gas 角度再强调一下：
  - 复杂逻辑可以拆成库或多个实现合约，主合约只保留薄逻辑层；
  - 对于多实例合约（例如 N 个几乎相同的 Vault），可以用 **EIP-1167 Minimal Proxy** 这种克隆模式，大幅降低部署 gas。

7. 算术 & 控制流层面的微优化

打包成一小节 checklist：

7.1 `unchecked {}` 块

- 在 Solidity ≥0.8 默认有溢出检查；
- 对能证明不会溢出的简单加减法（如循环计数器），用 `unchecked { ++i; }` 可以略微省 gas；
- 前提是：**真的能证明永不溢出**，否则是引 bug 的温床。

7.2 短路逻辑与条件顺序

- `if (cheapCheck && expensiveCheck) { ... }` → 把便宜的判断放前面，让失败早点发生；
- 减少无用计算。

7.3 位运算

- 某些场景可以用 bitmask 存一组 bool 标志，配合 `&`, `|`, `<<`, `>>` 操作：
  - Example：白名单组、权限位、状态位；
- 通常比多个独立 `bool` slot 更省存储 + gas。

8. 如何“科学地”测试和度量 Gas

8.1 Hardhat + hardhat-gas-reporter

- 在测试时自动输出：
  - 每个测试用例的 gas；
  - 每个方法的平均调用 gas；
  - 部署成本，并且可以换算成法币显示。

8.2 Foundry（forge）gas snapshot

- `forge test --gas-report` 或 `forge snapshot` 输出函数级别的 gas 使用；
- 适合做“前后对比”，一眼看出某次提交让哪几个函数 gas 上升了。

8.3 Tenderly / 区块浏览器 Trace

- Tenderly 提供逐 op-code 的 trace，以及主网 / 测试网 fork 调试；
- 也可以在 Etherscan / Blockscout 上查看已上链交易的 gas 分布，看看哪个内部调用最贵。

8.4 CI 中拉通

- 在 CI 里加一条规则：如果 gas 增长超过某个阈值（比如 +20%），就提示 / 阻断；
- Hardhat gas reporter 有和 Codechecks 集成，可以自动在 PR 上展示 gas diff。

9. 小结：

- **状态层**
  - 能不写 storage 就不写；能合并写就合并；
  - 打包多个小变量到一个 slot；
  - 用事件记录“只读历史”，少占用状态。

- **数据位置 & 结构**
  - external 参数优先 `calldata`；
  - 重复使用的 storage 先缓存到 `memory`；
  - 多用 `mapping` + 索引，而不是大数组暴力遍历。
- **合约结构**
  - 引入代理 / 模块化 / 库，减少代码重复；
  - 利用 `constant` / `immutable` 存放不会改的配置；
  - 使用 transient storage 存短期状态（EIP-1153 之后可选）。
- **控制流**
  - 避免不受控长度循环；
  - 能 off-chain 算的别 on-chain 算；
  - 慎用 `unchecked`，用好短路逻辑和位运算。
- **工具**
  - 开发期：Hardhat / Foundry 开启 gas report；
  - 预上线：在 fork 主网环境、Tenderly 上跑关键路径；
  - 上线后：通过浏览器 trace / 监控观察真实调用的 gas 分布。

EVM 与 Gas 机制的精妙在细节中展现，越学越能感受到它的优雅。

——@leopc999（乐乐）

第六章

共识机制与生态展望

本章目标：了解以太坊共识优势与生态扩展方式。

一、早期共识机制：工作量证明（PoW）

1. 经过验证的安全机制

当以太坊在 2015 年主网启动时，选择的是和比特币类似的 **工作量证明（PoW）** 共识。彼时 PoW 已在比特币上运行多年，被证明在开放网络里能够有效抵御双花和多数攻击，是最成熟、最稳妥的公共链安全方案之一。

对以太坊创世团队来说：

- **要做智能合约 + 公链**，一开始最不能出问题的是“安全性”和“共识是否靠谱”，
- 在 PoS 研究尚不成熟的 2014－2015 年，直接采用已经被比特币实战验证过的 PoW，是一个相对保守但非常务实的选择。

2. 去中心化抵抗攻击

以太坊没有直接沿用比特币的 SHA-256，而是设计了自己的 PoW 算法 **Ethash**，核心目标之一就是 **“memory-hard（内存密集型）+ 尽量弱化 ASIC 优势”**：

- Ethash 需要对一个容量较大的 DAG 数据结构做随机读取，强依赖显存带宽；
- 这让定制 ASIC 难以以极大倍数碾压 GPU，从而鼓励更多普通显卡矿工参与，有利于去中心化。

从今天回看，Ethash 的“显存友好 +ASIC 抑制”设计，确实让早期以太坊算力在全球 GPU 矿工之间分布得相对分散，而不是马上被少数 ASIC 矿场垄断，这对早期网络的抗审查和抗攻击能力帮助巨大。

3. 确保最低门槛、技术成熟

PoW 的另一个现实优势是：**参与门槛低、技术栈成熟。**

- 任何人只要有一台普通电脑或 GPU，就能下载客户端开始挖矿，不需要一开始就持有 ETH；
- 共识算法、网络模型、挖矿工具链，都可以直接复用比特币生态已经打磨过的经验。

相比之下，当年 PoS 还停留在论文与原型阶段，没有像 PoW 那样经过多年主网“生死考验”。为了让以太坊 **先跑起来、先把智能合约生态做起来**，选择技术上更成熟、实施路径更清晰的 PoW，是一个极具工程现实主义的决定。

4. 为未来 PoS 转型铺路

以太坊在设计之初就没打算“永远 PoW”。

- 早期白皮书和官方资料中多次提到：**PoW 只是过渡方案，长期目标是切换到 PoS（Casper）**，以降低能耗、提升经济安全性。
- 选择 PoW 启动网络，可以先让开发者和用户生态成形，同时给研究团队足够时间，在主网之外反复实验和迭代 PoS 协议。

这个路线在之后多年中逐步兑现：从 2020 年信标链（Beacon Chain）上线，到 2022 年 “The Merge（合并）” 把主网正式切换到 PoS，以太坊完成了当初“先 PoW、后 PoS”的长期规划。

5. 易于实现、快速启动

在 2014–2015 年，以太坊面临一个很现实的问题：

> “要在有限时间内，把一个可编程公链真正上线，让人能写合约、发代币、做 DApp。”

当时：

- PoW 共识已经有成套实现经验，客户端（如早期的 Geth）只需在成熟代码基础上扩展 EVM 和合约层逻辑；
- 而 PoS 若要直接用于主网，需要额外解决长程攻击、惰性攻击、经济惩罚（slashing）等一系列全新问题，风险远高于 PoW。

因此，从**“尽快上线、先让世界计算机跑起来”**的角度，PoW 是在安全、开发周期、社区认知三方面都更可控的起点。

6. 从今天回头看：PoW → PoS 的闭环

- 以太坊主网已经在 2022 年 9 月的 **The Merge** 完成从 PoW 向 PoS 的切换；PoW 挖矿在主网正式停止。
- 官方测算显示，相比 PoW 时代，以太坊在合并后能耗下降约 **99.95%**，也印证了当初“PoW 只是启动阶段、长期过渡到 PoS”的设计初衷。

> 当年选 PoW，是利用“已被验证的安全方案”来安全启动一个全新的智能合约平台；而从一开始，把 PoW 当作“起飞用的助推火箭”，在生态成熟后再切换到 PoS，正是以太坊路线图里早就写好的故事线。

二、The Merge 的定义与重要性

“The Merge”（合并）是以太坊历史上最重要的技术升级之一，于 **2022 年 9 月 15 日** 成功实施。它标志着以太坊从工作量证明（Proof-of-Work, PoW）彻底过渡到权益证明（Proof-of-Stake, PoS）共识机制。

1. The Merge 是什么？

在 The Merge 之前，以太坊网络实际上由两条“叠在一起”的链组成：

1.1 以太坊主网（Execution Layer，执行层）

- 也就是我们日常使用的以太坊链，记录所有账户、余额、智能合约和交易历史。
- 早期运行在 PoW 共识下，由矿工通过算力挖矿来打包区块、验证交易。

1.2 信标链（Beacon Chain, Consensus Layer，共识层）

- 自 **2020 年 12 月** 起单独运行的一条 PoS 链，只负责 PoS 共识逻辑：质押、验证者集合、出块顺序、投票与最终性等，不处理用户交易。
- 可以理解为“先把 PoS 引擎搭好，但先不接到主网刹车油门上”。

The Merge 做的事情，就是：

让原本跑在 PoW 上的以太坊主网“接管”信标链这个新的 PoS 引擎——

- **主网继续保留原有历史和状态（账户、合约、余额都不变）；**
- **原来的 PoW 挖矿被关掉，出块和共识改由信标链上的 PoS 验证者负责。**

所以它不是“换了一条新链”，而是 **把主网的执行层和信标链的共识层合并**到一起：矿工退场，验证者接棒。

2. 为什么 The Merge 如此重要？

2.1 能源消耗大幅降低：从“矿场”到“质押”

这是 The Merge 最直观、也最常被媒体提到的改变。

- 在 PoW 时代，以太坊需要大量电力驱动 GPU/矿机进行哈希计算，曾被多篇研究比喻为“接近一个中小国家的能耗水平”。
- **合并后，安全性不再依赖算力竞赛，而改为依赖质押的 ETH 数量和经济惩罚机制。**

以太坊基金会以及多篇独立分析报告的结论是：

> 以太坊在合并后，能源消耗约减少 99.95% 左右。

这意味着：

- 以太坊不再是“挖矿用电大户”，整体能耗现在更接近一个大型在线服务 / 数据中心，而非能源密集型矿业。
- 在 ESG（环境、社会、治理）和合规叙事上，这个转变非常关键——许多之前因为环保舆论犹豫的机构，更有理由在合规框架内考虑使用 / 参与以太坊生态。

> 小结一句：The Merge 把以太坊从“高能耗矿场”变成了“低功耗质押网络”。

2.2 安全性与经济惩罚：从“烧电”到“锁仓”

合并后，以太坊的安全根基从“谁更能烧电”迁移到了“谁愿意锁更多真金白银在链上”。

- **验证者必须质押 ETH** 才能参与出块和投票。
- 如果出现作恶行为（例如双重签名、验证无效区块等），共识层会对其**进行 Slashing（削减质押）**，直接罚没一部分甚至大部分质押资金。

这带来几个安全上的变化：

① 攻击成本高度金融化、可量化

- 想要发动 51% 攻击，不再是买矿机 + 电，而是要买入并愿意暴露大量可被 Slash 的 ETH，成本高且难以套现退出。

② 更易追责与恢复

- 恶意验证者可以在链上被识别，并通过 Slash + 强制退出验证者集合的方式驱逐出网络。

③ 参与门槛的结构变化

- 理论上，只要质押 32 ETH 或通过质押池参与，就可以成为验证者或间接参与验证。
- 从“硬件门槛 + 电力”转为“资金抵押 + 运行节点软硬件”，形态上把安全从物理侧更多迁移到了金融 / 协议侧。

2.3 The Merge 对 TPS 和 Gas 的真实影响

一个很容易被误解的点：

> The Merge 本身并没有显著提升 TPS，也没有直接让 Gas 变得更便宜。

原因是：

- The Merge 主要只替换了 **共识机制（PoW→PoS）**，执行逻辑、区块大小、EVM 规则等大体保持不变；
- TPS 和 Gas 主要受区块大小、执行效率和扩容方案（如 Rollup、未来的分片）影响，而不是 PoS/PoW 本身。

不过：

- PoS 统一了区块时间（slot 时间约 12 秒），使得 **出块节奏更稳定、重组概率更低**；
- 为未来的 **分片（Sharding）和 Rollup-centric roadmap** 清理了技术债，是往后所有扩容路线图的“地基”。

可以理解为：**The Merge 是换了发动机，但车速要明显提升，还得靠后续的 Dencun、分片等升级与 Layer2。**

2.4 为可扩展性（分片、Rollup）铺路

- 以太坊现在走的是 **“Rollup-centric + 未来分片”** 的扩容路线：
  - 短中期依赖 Optimistic Rollup、ZK-Rollup 等 Layer2 吞吐主流交易；
  - 主链负责数据可用性和结算（DA + Settlement）。
- **PoS + 信标链结构，是后续数据分片 / Danksharding 的前置条件**；
- 2024 年的 **Dencun 升级**（包含 Proto-Danksharding / EIP-4844）已经上线，为 Rollup 提供更便宜的数据空间，显著降低了 L2 的成本，对 The Merge 之后的扩容路线是一次关键承接。

所以从时间线看：

> The Merge → Shanghai/Capella（开放取款）→ Dencun（EIP-4844）→ 未来更完整的分片 / Danksharding。

The Merge 是这条路线的“第一颗多米诺骨牌”。

2.5 ETH 经济模型的变化：从“通胀”到“有通缩压力”

The Merge 与更早的 London 升级（EIP-1559）叠加后，ETH 的供给逻辑发生了深刻变化：

① 发行量下降

- PoW 时代需要不断给矿工支付区块奖励，发行量较高；
- PoS 时代对验证者的奖励率远低于 PoW 挖矿补贴，理论发行速度明显降低。

② **BaseFee 销毁（EIP-1559）继续生效**

- 每笔交易的基础费（BaseFee）仍会被直接销毁；
- 当链上使用活跃时，销毁量会变大。

③ **综合效果：有“净通缩”的可能，但取决于链上活跃度**

- 在 Merge 之后的一段时间内，ETH 总供应曾多次出现 **“净通缩”** 阶段，即销毁量 > 发行量；
- 但随着 2024 年 Dencun 让 L2 变便宜、主网 Gas 整体验证负载下降，销毁量有所减少，近一两年 ETH 供给大体在**“低通胀和轻微通缩之间摆动”**的状态，比 PoW 时代高通胀要温和得多。

这意味着：

> ETH 更像是一种“有时通缩的生产性资产”：既是 Gas / 抵押资产，又通过销毁机制对长期供给形成压力。

2.6 对生态与叙事的综合影响

最后，从宏观视角看，The Merge 带来的不仅是技术层面的改变，也深刻影响了以太坊的整体叙事：

- **环保叙事**：从高能耗 PoW 跳到超低能耗 PoS，使以太坊在监管、机构合规、ESG 指标上更“说得过去”；
- **金融安全叙事**：安全性从“电力 + 矿机”转向“经济抵押 +Slashing”，与传统金融的“保证金 / 准备金”逻辑更接近，更容易被机构理解；
- **技术路线稳态**：确立了 Rollup+PoS+ 未来分片的长期技术框架，给开发者和基础设施提供了相对清晰的预期。

一句话（可以直接背诵）：

> The Merge 不是“换壳”，而是给以太坊换了发动机和底盘：把安全从烧电迁移到质押，把能耗砍掉了 99.95%，为后续的扩容、分片和更健康的 ETH 经济模型打好了基础。

三、权益证明（PoS）的本质与变革

1. PoS 的本质

PoS 的核心思想是：谁拥有更多的代币，并愿意将其锁定作为对网络安全的“赌注”（stake），谁就更有可能被选中来验证下一个区块。

其运作方式可以概括为：

质押（Staking）：想要成为验证者的网络参与者需要将一定数量的区块链原生代币（例如以太坊的 ETH）锁定在一个智能合约中。这笔锁定的代币就是他们的“权益”或“质押”。

验证者选择：协议会根据一套算法（通常结合质押数量、质押时长和随机性等因素）伪随机地选择一个验证者来提议（创建）下一个区块。质押的代币越多，被选中的概率就越大。

验证与共识：被选中的验证者会负责收集、验证交易，并创建新的区块。其他验证者会验证这个新区块的有效性。一旦大多数验证者确认区块有效，它就会被添加到区块链上。

奖励与惩罚：

奖励：成功提议和验证区块的验证者会获得区块奖励（新发行的代币）和/或交易费作为报酬。

惩罚(Slashing)：如果验证者行为不端（例如，试图验证无效交易、双重签名等），他们质押的部分或全部代币将被没收（slashed）。这种经济惩罚机制是 PoS 安全模型的核心，它鼓励验证者诚实行为。

2. PoS 带来的主要变化

PoS 机制的引入对区块链生态系统带来了多方面的重大变化，尤其是在以太坊成功过渡到 PoS 之后，这些变化变得更为显著：

极大地提高能源效率

这是 PoS 最受关注和最直接的优势。

告别能源密集型挖矿：PoW 依赖于矿工投入大量计算能力（算力）来解决复杂的数学难题，导致巨大的电力消耗和碳排放。

能耗骤降：PoS 则通过经济质押来确保安全，不需要大规模的计算竞赛。例如，以太坊在转向 PoS 后，其能源消耗骤降了大约 99.95%，使得区块链更加环保和可持续。这有助于吸引更多关注环境、社会和治理（ESG）因素的机构和企业。

增强网络安全性与经济惩罚

PoS 引入了新的安全模型。

经济激励对齐：验证者通过质押代币，与网络的健康和价值深度绑定。如果网络受到攻击，他们质押的代币价值也会受损，从而提供强大的经济动机来维护网络安全。

削减机制(Slashing)：这是 PoS 独有的强大安全特性。恶意或不当行为（如验证无效交易、双重签名区块）会直接导致验证者质押的代币被没收，使其攻击成本高昂且损失惨重。这比 PoW 中仅仅是浪费电力要更直接、更痛。

降低 51%攻击的长期风险：虽然 PoW 中发动 51%攻击需要巨额计算硬件投入，但在 PoS 中，攻击者需要获得网络中 51%或更多的质押代币。这同样非常昂贵，而且一旦攻击发生，社区可以通过协议升级来惩罚恶意质押者，使其攻击成本进一步增加，并可能导致其失去所有质押代币。

为可扩展性奠定基础

虽然 PoS 本身不直接提高交易吞吐量，但它是实现未来可扩展性的关键。

分片(Sharding)的基础：PoW 机制下的分片实现非常复杂，而 PoS 设计与分片技术天然兼容。分片可以将区块链分成多个并行处理的小部分（分片链），每个分片独立处理交易，从而大大提高整个网络的总交易吞吐量。PoS 为以太坊未来的分片升级铺平了道路。

更稳定的出块时间：PoS 通常能实现更稳定的出块时间（例如以太坊 PoS 约 12 秒一个区块），这有助于提高交易的确定性和可预测性，从而更好地支持各种去中心化应用。

改变代币经济模型

PoS 对原生代币的供需关系产生深远影响。

通缩潜力：结合以太坊的 EIP-1559 销毁机制，PoS 中新的代币发行量（支付给验证者的质押奖励）远低于 PoW 中的挖矿奖励。在网络交易活跃时，销毁的代币数量可能超过新发行的数量，使代币供应量减少，从而可能形成通缩效应。

质押收益：用户可以通过质押代币来获得收益，这为代币持有者提供了一种新的被动收入方式，改变了代币的效用。

降低进入门槛（某种程度上）：相对于 PoW 中对专业硬件和大量电力的需求，PoS 允许任何人只要质押最低数量的代币（如以太坊的 32 ETH 或通过流动性质押池）就可以参与网络安全，这可能促进更广泛的参与。

3. 既然讲到这里了，那就顺便介绍一下所有的共识机制（可能不全，有遗漏的可以告诉我哈）

工作量证明（Proof of Work, PoW）

本质：通过计算能力竞争产生新区块。

工作原理：矿工们竞争解决一个复杂的密码学难题（寻找一个哈希值，使其小于某个目标值）。第一个找到答案的矿工可以提议新的区块，并获得区块奖励。解决这个难题需要消耗大量计算资源和电力（即“工作量”），因此被称为“挖矿”。

优点：

安全性高：攻击者需要控制网络超过 51%的计算能力才能发动攻击，成本极高。

去中心化：任何拥有计算设备的人都可以参与，理论上是开放的。

历史最悠久：比特币的成功证明了其韧性。

缺点：

能源消耗巨大：“挖矿”过程消耗大量电力，对环境不友好。

扩展性差：区块生产速度慢，导致交易吞吐量低（例如比特币约 7TPS，以太坊 PoW 约 15TPS）。

中心化风险：随着专业化挖矿设备（ASICs）的出现和大型矿池的形成，算力可能逐渐集中。

代表项目：比特币（Bitcoin）、以太坊（Ethereum）（已于 2022 年 9 月转为 PoS）。

权益证明（Proof of Stake, PoS）

本质：通过质押代币竞争产生新区块。

工作原理：验证者将一定数量的区块链原生代币锁定（“质押”）在一个智能合约中。协议会根据质押的数量、时长和随机性等因素，伪随机地选择验证者来提议或验证新区块。成功的验证者会获得区块奖励和/或交易费。如果验证者行为不端，他们质押的代币会被罚没（“削减”）。

优点：

能源效率高：不再需要大量计算，能耗大幅降低（如以太坊 PoS 降低 99.95%）。

扩展性潜力：更容易实现分片等扩容技术，提高交易吞吐量。

经济安全性：恶意行为会导致质押代币被罚没，攻击成本极高且有直接损失。

缺点：

“富者愈富”争议：拥有更多代币的验证者获得更多奖励的机会更大。

中心化风险：代币可能过度集中在少数实体手中。

“长程攻击”/“Nothing at Stake”问题：早期 PoS 理论上存在的漏洞，但现代 PoS 协议通过惩罚机制和检查点等方案已基本解决。

代表项目：以太坊（Ethereum）（通过 TheMerge 已完成 PoS 转型）、Solana（Tower BFT PoS）、Cardano（Ouroboros PoS）、Avalanche（Snowman PoS）。

委托权益证明（Delegated Proof of Stake, DPoS）

本质：通过投票选举少数代表来生产区块。

工作原理：代币持有者投票选举出一定数量的“代表”（或称“超级节点”、“见证人”），由这些少数代表轮流负责验证交易和创建区块。用户可以通过将自己的代币“委托”给某个代表来增加其当选概率。

优点：

高吞吐量：少数代表之间更容易快速达成共识，交易处理速度快。

低交易费用：效率高，Gas 成本通常较低。

治理灵活：社区可以通过投票更换不称职的代表。

缺点：

中心化风险：区块生产者数量有限，相对 PoW 和纯 PoS 更为中心化。

寡头政治：可能形成少数代表的利益集团，影响网络公平性。

代表项目：EOS、Tron（TRX）、BNBChain（BSC）（部分使用 DPoS 元素）。

权限证明（Proof of Authority, PoA）

本质：少数授权节点（由权威机构或组织指定）负责验证和生成区块。

工作原理：网络中的验证者是一组预先批准的、有明确身份和声誉的实体。这些实体轮流验证交易和创建区块。由于验证者是已知的和可信的，共识速度非常快。

优点：

高吞吐量：交易处理速度非常快。

低交易费用：几乎没有共识成本。

确定性强：几乎立即确认交易。

适合联盟链/私有链：适用于企业和政府等对性能和可控性要求高的场景。

缺点：

中心化程度高：验证者数量非常有限，且身份公开，牺牲了去中心化。

信任依赖：整个系统的安全性依赖于这些授权节点的诚实性。

代表项目：POA Network、Binance Smart Chain（BSC）（部分使用 PoA 元素）、Hyperledger Fabric（可配置不同共识，包括类 PoA）。

拜占庭容错类共识（BFT-based Consensus）

本质：通过多轮消息交换，节点就交易顺序达成一致，即使存在恶意节点。

工作原理：这类共识机制通常在少数预设的验证者集合中运行。节点之间通过发送和接收消息来互相验证交易和区块，即使有部分节点是恶意或离线（少于 1/3 的拜占庭故障节点），系统也能保持正常运行。

优点：

即时最终性：交易一旦被确认，就不会被回滚，提供非常高的确定性。

高吞吐量：效率通常很高，适用于需要快速交易确认的场景。

缺点：

扩展性受限：节点数量增加时，消息传递的复杂性和开销会呈指数级增长，因此通常只适用于节点数量较少的联盟链或高性能公链。

中心化风险：验证者集合是有限的。

代表项目：

实用拜占庭容错(PBFT)：最早的 BFT 算法，用于私有链和联盟链。

Tendermint（Cosmos SDK 核心)：Cosmos 生态系统的核心，结合了 BFT 和 PoS 思想。

HotStuff（Libra/Diem 曾用，现在 Aptos 和 Sui 使用其变体)：高度优化的 BFT 协议，具有高吞吐量和线性扩展性。

其他共识机制（简要提及）

Proof of History（PoH)：Solana 特有的机制，结合 PoS。它通过密码学证明历史事件的顺序，以提高网络效率和速度，但本身不是共识机制，而是共识的辅助。

Proof of Capacity（PoC)：通过硬盘存储空间证明来挖矿。

Proof of Elapsed Time（PoET)：基于可信硬件（如 IntelSGX）的共识，随机选择领导者。

Directed Acyclic Graph（DAG)共识：一些项目（如 IOTA、Nano）不使用传统的区块链结构，而是使用 DAG，其共识机制也与传统链不同，强调异步和并行处理。

这里来做一些介绍：

IPFS 的基本机制（无区块共识）

IPFS 使用 **内容可寻址 / 内容哈希（CID, Content Identifier）**来标识数据块、文件、对象。每个对象/文件通过其哈希被唯一标识。

节点之间通过 **分布式哈希表（DHT, Distributed Hash Table）**来查找哪个节点持有某个 CID 对应的数据块。

数据存取是 **peer-to-peer（点对点）**的协作模式：节点请求某个 CID，网络上有该块的节点响应。

IPFS 不要求所有节点存储全部数据，也不强求节点持续保持某块；节点可以选择 cache 或 keep（固定 pin）或不保留某些数据。

因为没有全局状态、不需要出块、也不存在交易顺序这种东西，IPFS 本身不具备类似区块链那样的共识层。

因此，当我们谈 IPFS 的“共识 / 协调”时，更准确的指是 IPFS 附加系统（如 IPFS Cluster）在多个节点之间就“哪些数据要被 pin / 保留”、“数据副本数量 / 副本分布”这些状态达成协作 / 一致性的机制。

IPFS Cluster 的“共识组件 / 协调机制”

IPFS Cluster 是 IPFS 的一个协作系统，用于多个节点共同管理一个“pinset”（即哪些 CID 被固定 / 保留）和副本策略。它提供两种“共识组件”实现，以确保所有 cluster 节点对 pinset 的视图最终一致。

这两种机制是：

机制	核心方式	优点	缺点 / 适用条件
CRDT-based（Merkle-CRDT / go-ds-crdt 等）	每个节点维护一个可合并的、不可变（append-only）的 Merkle-CRDT 结构，通过 Gossip / pubsub 传播更新，然后各节点合并更新以达成最终一致性；使用 IPFS 内容寻址 / DAG 结构 + gossip 传播变动	容错性好、容易扩展、支持异步 /无主复制、在节点失败 /加入 /断开时有较强鲁棒性	一致性是 eventually consistency（最终一致），可能短时间内状态不一致；变动量大时 DAG 会变深、合并成本上升；新节点初始化时需遍历 DAG 历史（可能慢）
Raft（基于 HashiCorp Raft）	类似经典 Raft 共识：选一个 leader，所有更新通过 leader 日志（log）广播给其他节点并得到多数确认后提交变动	强一致性（线性化写入顺序）、操作有确定顺序、日志机制成熟	扩展性较差（节点很多时性能下降）、leader 容错有成本、网络分区 /节点延迟影响较大、对大规模副本集群不适合

在 IPFS Cluster 启动时要指定使用哪一种共识组件（--consensus crdt 或 --consensus raft）来管理 pinset。

IPFS Cluster 的共识组件负责的主要职责包括：

管理全局 pinset 的变动（哪些 CID 被 pin / unpin）

维护对等节点之间的一致视图（最终要趋于一致）

管理节点加入 / 退出 / peer 信任 / peerset 管理

将 pin 操作写入本地存储 / 数据库 /DAG，以便节点能恢复状态或重启后同步

简而言之，IPFS 的“共识”更多是协作式状态同步 / 最终一致性，而不是像链上 PoS / BFT 那样的出块共识。

Bearchain / Berachain 的 PoL（Proof-of-Liquidity）

Berachain 是一个区块链 / Layer-1 项目，它提出一种称为 Proof-of-Liquidity（PoL）的共识 / 激励机制。从文档看，PoL 是在传统 PoS 基础上的一种扩展 / 经济模型重塑，旨在把流动性（liquidity）纳入共识层面，从而使网络安全（staking）和 DeFi 活跃度 / 生态价值更紧密地对齐。

下面是 PoL 的关键机制和逻辑。

PoL 的基本结构 / 核心要素

双代币（Two-Token）模型：

$BERA：用作质押 / 安全 / gas 代币，是用来参与 staking、确保网络安全的主要代币。

$BGT：治理 / 奖励代币（Soulbound / 非可转移 / 某种限制性代币），通过共识 / 区块生产给出，和生态奖励 / 治理参与有关。

Active Set / 验证者选择：验证者要用 $BERA 进行质押（stake），达到最低额度、在前 N 名中才能进入验证者集合（Active Set）。验证者 stake 越多，获得出块 / 投票机会越大（类似 PoS）

治理 / 奖励的流动性关联

验证者还要向生态 / 治理 / 奖池（Reward Vaults）分配 / 导向其获得的 $BGT 奖励，或者其奖励与其收到的 BGT 委托量有关。这样做的目的是让验证者和流动性提供者（liquidity providers）在经济上更紧密绑定。

区块奖励 / emissions 分配

验证者产生区块 / 执行共识有奖励部分属于 BGT，而不是全部奖励都归验证者或质押者。验证者需要把 BGT 的大部分投向生态 / 协议 / 应用等（Reward Vaults），只有一小部分保留自己使用。这样链上生态 / 应用方也能从共识层获得激励。

激励对齐 / 资本效率 PoL 的设计目标之一是希望减少验证者 / 质押者把资本锁定在 staking 的动机太强，从而减弱对 DeFi 应用 / 生态活跃度的支持。PoL 使得提供流动性成为参与共识 / 赚奖励的一部分，从而更好对齐区块链安全与 DeFi 生态繁荣。

共识机制 / 网络安全基础

虽然 PoL 引入了流动性维度，但其底层在安全 / 区块链共识上仍类属于一种 PoS / staking 架构（或 PoS 的变种）。换句话说，流动性是额外的 “经济维度 / 约束 / 激励模式”，但参与共识、验证、投票、提案的机制仍然依赖 staking / 验证者 + 权益机制。Berachain 文档里并没有把 PoL 描述为一种完全脱离 staking 的共识，而是加强 staking 机制与生态 / 流动性的融合。

也就是说，PoL ≠ 一种全新的底层共识算法（像 BFT、PoW 那种），而是一种在传统 staking / 验证者基础之上，加上流动性 / 治理 / 奖励分配机制的扩展 / 创新。

总结一下就是：

IPFS 的“共识 / 协调” 更像是节点之间就内容 / pin 计划 / 副本状态做协作与一致性机制，不是区块链意义上的共识。它使用 CRDT 或 Raft 等方式同步 pinset 状态。

Berachain 的 PoL 是一种融合了 PoS 和流动性 / 生态激励的区块链共识 / 经济设计。它把流动性（为生态 / 应用提供资金 / 资金使用效率）纳入共识层的奖励 / 分配机制中，以期在链安全与生态活跃之间取得更好的对齐。

四、验证者惩罚机制 Slashing 解析

大部分质押的 ETH（本金损失）：

这是最直接和最主要的损失。具体罚没金额会有所不同，但其设计目的是为了让作恶者付出沉重代价。

即时罚金：他们的质押 ETH 会立即被扣除一部分，通常对于一个质押 32 ETH 的验证者来说，这笔金额大约在 1 ETH 左右。这部分 ETH 会被销毁（永久移除流通）。

关联性罚金（大规模罚没罚金）：这是以太坊罚没机制的一个独特之处。如果在同一时间段内（例如 36 天内），有大量验证者被罚没，那么每个被罚没验证者的罚金会根据网络中被罚没的 ETH 总量成比例增加。这是一个强大的威慑，旨在阻止协同攻击。在极端情况下，如果大部分质押的 ETH 被罚没（例如，由于一个影响众多验证者的客户端重大 Bug，或一次协同攻击），单个验证者可能会损失全部 32 ETH 质押。这笔罚金通常会在其罚没期（36 天）的中点（约 18 天）应用。

罚没期内错过证明的罚金：一旦被罚没，验证者将被安排强制退出网络。这个退出过程通常需要 36 天。在此期间，该验证者会持续遭受不活跃罚金（就像他们离线一样），直到其提款周期。这种渐进式的“流血”会增加总损失，对于单个孤立事件而言，通常会损失 0.1-0.2 ETH。

强制退出网络：

被罚没的验证者将不可逆转地从活跃验证者集合中移除。他们不能再参与提议或证明区块。

他们将停止获得任何未来的质押奖励。

损失未来的质押奖励：

由于他们被从活跃集合中移除，显然他们将停止获得任何未来验证奖励。

当然也会对名誉造成损害：

被罚没是一个在区块链上公开记录的事件。对于专业的质押服务提供商或个人验证者而言，这会严重损害他们的声誉和信任度，使他们未来难以吸引委托者。

难以重新加入（如果可能的话）：

如果验证者在被罚没后想重新加入网络，他们需要存入全新的 32 ETH 并生成新的验证者密钥，这实际上是从头开始。他们之前被罚没的质押资金将不复存在。

五、PoS 对能耗与发行速度的影响

其实在上面 The Merge 就差不多讲完了，但不影响再来一次。

网络能耗的变化：骤降 99.95%

在 PoS 之前，以太坊采用 PoW 机制，矿工通过消耗大量电力进行计算竞争来验证交易和生成新区块。这种“挖矿”过程需要庞大的计算设备和持续的电力供应，导致以太坊网络的能耗非常高，与一些小型国家的耗电量相当。

转向 PoS 后，网络的安全性不再依赖于物理的计算竞赛，而是依赖于验证者质押的 ETH。验证者通过锁定代币来获得验证和提议区块的权利。这意味着：

告别高能耗挖矿：不再需要能源密集型的矿机和电力消耗。

能耗大幅降低：据估计，以太坊在 PoS 机制下的能源消耗比 PoW 时期降低了约 99.95%。这使得以太坊成为一个对环境更加友好的区块链，极大地提升了其可持续性，也更容易被注重 ESG（环境、社会和治理）因素的机构和企业接受。

ETH 发行速度的变化：显著降低，具备通缩潜力

在 PoW 机制下，新的 ETH 主要通过区块奖励发行给矿工。而在 PoS 机制下，ETH 的发行方式发生了根本性变化：

发行量显著减少：PoS 下的 ETH 发行量远低于 PoW。新的 ETH 主要用于奖励诚实的验证者，激励他们质押 ETH 并维护网络安全。由于不再需要支付巨额的挖矿成本（电力和硬件），验证者所需的 ETH 奖励相对较少即可维持激励。

具体数据：在 PoW 时代，以太坊每年大约发行 430 万枚新的 ETH（包括区块奖励和叔块奖励）。在 PoS 时代，根据网络中总质押量，每年新发行的 ETH 大约在 60 万至 70 万枚之间，大幅减少了约 90%。

结合 EIP-1559 销毁机制：London 升级（EIP-1559）引入了 BaseFee 销毁机制。每笔交易支付的 BaseFee 不再支付给矿工（现在是验证者），而是被永久地销毁，从流通中移除。

通缩潜力：当销毁的 ETH 数量（来自 BaseFee）大于新发行的 ETH 数量（奖励给验证者）时，ETH 的总供应量就会减少，呈现通缩状态。这意味着 ETH 理论上可以变得越来越稀缺，从而可能增加其长期价值。

这种通缩效应在网络交易活跃、BaseFee 较高时会更加明显。这使得 ETH 成为一种“可编程的货币”，其供应量可以根据网络使用情况动态调整。

做个对比表格方便直接对比：

比较项	PoW 前	PoS 后
每日 ETH 发行	~ 13,000 ETH	~ 1,600 – 1,700 ETH
发行量减少	—	≈ 90%降低
年通胀率	~ 4 – 4.6%	~ 0.5%
EIP-1559 销毁机制	有，但不足抵消发行	BaseFee 继续被烧毁，净发行率可能降至 0%或负值(即净销毁)

六、历次升级对系统的改进

下面按时间线看一下以太坊几次主网升级，各自为系统带来了什么？

> 时间线大致为：
>
> **Frontier → Homestead → Byzantium → Constantinople / St.Petersburg → Istanbul → Muir Glacier → Berlin → London → The Merge → Shanghai / Capella（Shapella）→ Dencun → Pectra（Prague+Electra）。**

1. Frontier（前沿）— 2015 年 7 月

带来了什么？这是以太坊的「创世纪版本」—— 一个最低可行产品（MVP）。

- **主网启动**：以太坊主网正式上线，支持账户、交易、挖矿。
- **EVM 诞生**：引入以太坊虚拟机（EVM），让链上可以运行通用的智能合约逻辑。
- **基本功能就绪**：可以部署简单智能合约、执行转账、做最基础的 dApp 实验。

可以理解为：**这一版本证明了“可编程区块链”在现实世界里是可以跑起来的。**

2. Homestead（家园）— 2016 年 3 月

带来了什么？早期的稳定性和安全性改进。

- **协议稳定性增强：**修了一批 Frontier 时期的 bug，让客户端和网络更稳定。
- **Gas 成本微调：**对部分操作的 Gas 定价做了优化。
- **合约创建改进：**增强了一些合约创建流程上的细节，为后续复杂合约打基础。

可以把 Homestead 看成是「从实验室走向稍微能用一点的正式环境」。

3. Byzantium（拜占庭）— 2017 年 10 月

（Metropolis 第一阶段）

带来了什么？性能、安全性和隐私能力的显著增强，并第一次为 PoS 过渡做铺垫。

- **难度炸弹延迟 + 区块奖励下调：**推迟「难度炸弹」，同时降低区块奖励，为未来从 PoW → PoS 腾挪空间。
- **预编译合约（precompile）增强：**加入若干新的密码学预编译，尤其是支持 zk-SNARKs 相关操作，使零知识证明在链上更可行。
- **DoS 抗性更强：**对部分 opcode 的成本做重新定价，降低 DoS 风险。

Byzantium 之后，以太坊第一次正式具备了**隐私增强应用的密码学底座**（比如 zk-SNARKs）。

4. Constantinople & St. Petersburg（君士坦丁堡 & 圣彼得堡）— 2019 年 2 月

带来了什么？进一步优化 EVM、降低 Gas 成本，并继续延迟难度炸弹。

- **Gas 效率提升：**多个 EIP 调低了某些 opcode 的 Gas（例如位运算、部分存储操作），合约执行整体更便宜。

- **合约部署更灵活：**允许在部署阶段执行更多逻辑，提高部署期的可编程性。
- **再次延迟难度炸弹 + 降低区块奖励：**继续把「难度炸弹」往后推，并再次降挖矿奖励，为 PoS 转型争取更多时间。

St. Petersburg 主要任务是“紧急拆雷”——禁用一个在测试中发现存在安全隐患的 EIP（1283），以保证主网安全。

5. Istanbul（伊斯坦布尔）— 2019 年 12 月

带来了什么？Gas 重新定价 + DoS 抵御强化 + 与其他链 / Layer2 的兼容性提升。

- **Gas 成本调整：**对 SLOAD 等操作的 Gas 成本再次重算，减少某些攻击面，并反映真实计算/存储成本。
- **Layer2 友好：**引入新的预编译，使 zk-Rollup / SNARK / STARK 等方案在链上运行成本降低。
- **跨链 / 互操作性增强：**更好地与其他链上的工具和加密原语兼容。

Istanbul 可以视为为后面 **Rollup 为中心（rollup-centric）扩容路线**打前站。

6. Muir Glacier（穆尔冰川）— 2020 年 1 月

带来了什么？单一目的——再次推迟难度炸弹。

- 纯粹是「把难度炸弹再往后延一次」，避免 PoW 链在 PoS 未就绪的情况下“挖不动”。

这次升级本身没有功能性变化，但对网络生存至关重要。

7. Berlin（柏林）— 2021 年 4 月

带来了什么？Gas 结构优化 + 新交易类型，为后面的 London 铺路。

- **Gas 成本优化：**重新调整了多种 state 访问相关 opcode 的成本，以减少 DoS 风险并让费用更贴近真实资源消耗。

- **Typed Transactions（类型化交易）框架**：EIP-2718/EIP-2930 引入了新的交易封装格式（access list tx type 1），形成了“类型化交易”的基础，为后来的 EIP-1559（type 2）打好地基。

可以理解为：Berlin 把「交易格式」先做成一个可扩展的壳子，London 再往里塞新的 fee 机制。

8. London（伦敦）— 2021 年 8 月

带来了什么？最重要的一次经济模型变更之一：EIP-1559 费市场改革 + ETH 销毁机制。

- **EIP-1559：BaseFee + Tip 模型**
  - BaseFee：每个区块由协议自动计算的“基础费率”，随区块拥堵程度上下浮动，且 **被直接销毁（burn）**。
  - Tip（优先费）：用户额外给出的小费，直接给矿工/验证者，作为优先打包的激励。
- **费用可预测性提升**：用户不再需要完全凭经验猜 gasPrice，钱包可以根据 BaseFee 给出更准确的费用建议。
- **ETH 通缩压力**：因为 BaseFee 会被销毁，在网络繁忙时被烧掉的 ETH 数量可以接近甚至超过新发行量，从而给 ETH 带来通缩倾向。

从这次开始，ETH 真正具备了**「使用越多，越稀缺」**的经济特性。

9. The Merge（合并）— 2022 年 9 月 15 日

带来了什么？以太坊共识层的历史性切换：从 PoW → PoS。

- **PoW 矿工下线，PoS 验证者接管共识**：执行层仍然是原来的主网，但底层共识改为由信标链上的 PoS 验证者来出块与投票。
- **能耗骤降 ~99.95%**：不再依赖算力竞赛挖矿，而是依靠质押 ETH 来保证安全，能耗直接腰斩到传统金融系统甚至更低的量级。

- **安全模型转变：**引入 Slashing 等经济惩罚机制，作恶节点会直接损失质押 ETH。
- **为未来扩容打基础：**PoS 是实现分片（sharding）等后续大规模扩容路线图的前置条件。

The Merge 本身 **并没有直接提升 TPS 或降低 Gas**，但它是后面所有扩容设计的前提。

10. Shanghai / Capella（上海 / 卡佩拉）— 2023 年 4 月

（合称 Shapella）

带来了什么？完成 PoS 路线图上最关键的一步——解锁质押 ETH 提款。

- **启用取款（Withdrawals）：**EIP-4895 等提案让信标链上的质押 ETH 和奖励可以部分/全部提取到执行层地址。
- **缓解质押锁仓焦虑：**早期质押者从 2020 年就锁在信标链，Shapella 让他们第一次可以自由流动，极大提升了 PoS 的可接受度和可持续性。
- **部分 EVM 改进：**例如 push0 指令等小优化，略微提升合约代码效率。

从 Shapella 开始，「质押 ETH」从“单向票”正式变成**可进可出**的长期参与机制。

11. Dencun（坎昆 / Deneb + Cancun）— 2024 年 3 月

带来了什么？L2 成本大幅下降的关键升级，引入 blob 交易和 proto-danksharding。

- **EIP-4844：Proto-Danksharding & Blob 交易**
  - 引入一种新的交易类型，携带所谓的 “blob data”（数据块），专门给 Rollup 存放批量压缩后的 L2 数据。
  - 这些 blob 数据不进入传统执行层存储，而是以“临时数据”的形式存储，大约保存 ~18 天后可以被删除，只保留承诺（承诺哈希）。

- **L2 费用显著下降**：Rollup 把证明/批量数据放进 blob，而不是昂贵的 calldata，大幅降低了数据可用性成本，直接带动绝大多数 L2 手续费下调。
- **为真正 sharding 铺路**：proto-danksharding 是「半成品分片」，主要解决数据可用性；未来完整 danksharding 会在此基础上扩展。

一句话：**Dencun 把 Ethereum 彻底推向了“Rollup 中心的扩容时代”，主网更像是高安全的数据可用性层。**

12. Pectra（Prague + Electra）— 2025 年

> Pectra = 执行层的 Prague 升级 + 共识层的 Electra 升级，在 2025 年完成，是紧接 Dencun 的下一次里程碑升级。

总体目标：进一步改善钱包体验、优化质押架构，并继续强化对 L2 / Rollup 的支持和 EVM 的可扩展性。

12.1 执行层（Prague）侧的变化（部分）：

- **账户体系 & 钱包体验提升（账户抽象相关 EIP）**

  围绕 EIP-3074 / EIP-7702 等账户授权提案，增强 EOA 的“智能账户”能力：

  - 支持批量交易、一次签名执行多步操作；
  - 支持更灵活的 Gas 支付（如由第三方赞助）；
  - 为基于 ERC-4337 的账户抽象钱包提供更好的协议级支持。

- **EVM Object Format（EOF）初步落地**
  - 引入新的字节码封装格式 EOF，约束合约字节码结构，使验证更简单、安全边界更清晰。
  - 对已有合约是 **向后兼容、可选 opt-in** 的；主要影响新合约和开发体验，为未来扩展 EVM 提供更标准化的基础。

- **对 Dencun 的延续优化**
  - 在 blob 和 calldata 成本上做进一步微调，把 rollup 数据发布路径完全倾斜到 blobs 上，巩固“L2 使用 blob、L1 尽量少用 calldata”这一模式。

12.2 共识层（Electra）侧的变化（部分）：

- **EIP-7251：提高 validator 最大有效质押上限**
  - 将单个验证者的 **max effective balance 从 32 ETH 提高到最多 2048 ETH**。
  - 质押服务提供者不再需要拆成海量 32 ETH 小验证者，可以用更少的验证者节点管理更多质押，减少网络通信与资源消耗。
- **验证者运维体验优化**
  - 更灵活的退出/再质押逻辑，降低运维复杂度，有利于大型质押池和专业节点优化成本。

总体来看，**Pectra 并不是一个“讲故事的大升级”，而是一次非常“工程化”的迭代：**

- 对开发者：更强的 EOF / 账户抽象能力，合约结构更清晰、长远扩展性更好；
- 对质押者/节点：更高的单节点质押上限、更少验证者数量，通信压力下降；
- 对 L2：在 Dencun 基础上继续优化 blob 路线，L2 费用进一步下探。

13. 总结

升级名称	主要提升方向	带来的关键改变与好处
Frontier / Homestead	启动 & 稳定基础设施	上线 EVM 和智能合约能力，修复早期 bug，网络从实验走向「勉强可用」。
Byzantium / Constantinople	性能 + 隐私 + 减少 DoS 风险	引入 zk-SNARKs 预编译、调整 Gas、延迟难度炸弹，为 PoS 铺路。
Istanbul / Berlin	Gas 重新定价 + L2 兼容 + 新交易类型	让 zk-Rollup 更便宜，定义“类型化交易”框架，为 EIP-1559 做壳子。
London	手续费市场改革 +ETH 销毁	EIP-1559：BaseFee + Tip，BaseFee 被销毁，费用更可预测，ETH 出现通缩压力。
The Merge	共识层从 PoW → PoS	能耗下降约 99.95%，引入质押+Slashing 模型，为分片和后续扩容打基础。
Shanghai / Capella	质押流动性释放	支持质押 ETH 和奖励提款，提高 PoS 的长期可用性与参与意愿。
Dencun	数据可用性 & L2 成本	EIP-4844 blob 交易，Rollup 数据走 blob，L2 手续费大幅下降，主网转型为 DA 层。
Pectra（Prague+Electra）	智能账户能力 + 验证者架构 + EOF/L2 继续演进	EIP-7251 提高单验证者质押上限；推进 EOF 和账户抽象提案；对 blob 及 L2 数据路径做进一步优化。

七、分片 Sharding、Danksharding 与 Verkle 树

一句话（可以直接背诵）

- **分片（Sharding）**：把“要处理 / 存的东西”拆小，**缓解“所有节点都要干所有活”的问题**。
- **Danksharding（数据分片）**：把“给 Rollup 用的数据空间”做大、做便宜，**专注解决数据可用性（Data Availability）和 L2 成本**。
- **Verkle 树**：换一个更高级的状态树结构，**缩小证明体积，为“无状态客户端”铺路**，让跑节点更轻。

一张对比表：

维度	经典分片 Sharding	Danksharding / Proto-dank	Verkle 树
主要解决	TPS & 节点负载	为 Rollup 提供便宜的 DA 空间	状态膨胀 & 节点存储压力
作用层级	“整条链”被分成多片并行处理	只管“数据块（blob）”的发布、采样与定价	替换状态树（MPT → Verkle）
面向对象	所有 L1 交易 & 状态	L2 Rollup 的批量数据	所有账户 & 合约存储
目标效果	单链处理能力大幅提升	L2 手续费显著下降、整体扩容	轻节点/无状态客户端更容易实现

1. 分片（Sharding）：把链拆成很多“小分区”

1.1 传统设想：状态分片（State Sharding）

最早的以太坊分片设计，是想把整个链拆成很多“分片链”：

- 每个分片只处理自己那一部分交易和状态（账户、合约存储）。
- 节点不再需要保存“全网所有状态”，只需要关注若干个分片。
- 多个分片**并行出块和执行**，整体 TPS 理论上会近似线性提升。

解决的问题是很直观的：

- **吞吐量**：从“单车道”变成“多车道高速公路”，多条分片链并行处理交易。
- **节点负载**：每个节点只“管一块地”，运行全节点的门槛降低，有利于去中心化。

不过，状态分片非常复杂：

跨分片调用、跨分片转账的一致性、安全性都很难设计，而且 Rollup 崛起后，以太坊开始转向**“Rollup-centric”路线**，分片的角色被重新定位为“主要给 Rollup 提供数据空间”，而不是直接在 L1 做所有扩容。

2. Danksharding：给 Rollup 提供“大带宽 + 便宜空间”

2.1 Rollup 需要的不是“算力”，而是“便宜可验证的数据空间”

今天的扩容主力是各种 Rollup（Optimistic / ZK）。Rollup 在链下执行大部分计算，只把**压缩后的交易数据**和证明定期发回以太坊 L1，关键是：

> 这部分数据必须被 L1 “可靠公开”，任何人都能拿到，才能在必要时重放或验证 —— 这叫 数据可用性（Data Availability, DA）。

之前 Rollup 把数据写进 L1 的 calldata，非常贵；

所以 Ethereum 选择：**L1 重点扩容“数据可用性层”**，让 Rollup 有更大、更便宜的“写数据空间”。

2.2 Proto-Danksharding（EIP-4844）：已上线的第一阶段

- **EIP-4844（Proto-Danksharding）** 在 2024 年 Dencun 升级中上线，为以太坊引入了新的**“blob-carrying transactions”**：
  - 交易可以附带一批“blob”（大数据块），
  - blob 数据不会进入普通执行状态，只用于 DA，保留一段时间后可被节点丢弃，
  - 成本远低于 calldata，是迈向完整数据分片的过渡方案。
- 这一步已经**大幅降低了 Rollup 的数据发布成本**，L2 手续费随之明显下降（你现在在 Arbitrum/Optimism 上看到的降费，很大程度就是 EIP-4844 的效果）。

可以理解为：

> Proto-dank = 把“给 L2 存批量数据”的仓库先搭出来，但暂时还没有完整的分片 / 数据采样机制，只是先提供 更便宜、暂存的 blob 空间。

2.3 完整的 Danksharding：单 proposer + DA 采样

接下来要走向的，是“完整 Danksharding”，它的关键点有：

- **所有 blob 数据统合进一个区块，由单个区块提议者负责（Single-slot / single-proposer）**，而不是多条分片链独立出块，这是 Dankrad Feist 提出的设计思路；
- 引入 **数据可用性采样（Data Availability Sampling, DAS）**：
  - 验证者只随机抽样验证一小部分 blob 数据；
  - 如果大多数人都能成功 sample，说明整体数据高度可能是可用的；
  - 这使得验证者无需下载所有 blob，也能保证 DA 安全性。

和“传统分片”比起来：

- 以前设想：**很多分片链，各自出自己的块**；
- Danksharding：**只有一个块，但块里挂了大量“分片化的数据（blobs）”**，验证者通过采样保证这些数据真正存在。

结果就是：为 **Rollup 提供了一个巨大但价格便宜的数据高速公路**，而业务逻辑的执行仍然主要在 L2 完成。

3. Verkle 树：把状态树换成更“轻”的结构

3.1 现状：Merkle Patricia Trie（MPT）的瓶颈

目前以太坊的全局状态（账户、合约 storage）都存在 MPT 里。

要证明“某个 key 的值是 x”，需要从根一路下到叶子，给出一条较长的 Merkle 证明（Merkle proof）：

- 状态越大、树越深，证明就越长；
- 轻节点如果想验证状态，需要拉很多数据，网络和存储开销都不小。

随着状态膨胀，这种证明变得越来越重，这限制了轻节点和无状态客户端的可行性。

3.2 Verkle 树：用向量承诺（vector commitments）缩小证明

Verkle 树也是一棵前缀树，但每个内部节点用向量承诺（例如基于 KZG/IPA 的承诺）对一整组子节点进行承诺。这样：

- 要证明某个 key 的值，只需要提供少量承诺和开销，证明大小可以从几 KB 降到百余字节级别；
- 证明的大小与整个状态规模的关系没那么线性膨胀，更适合状态巨大的系统。

3.3 为“无状态客户端（Stateless Clients）”铺路

更小的证明带来的直接好处：

- **无状态客户端**不需要存整棵状态树，只需依赖区块里附带的 Verkle 证明，就能验证交易结果；
- 运行一个“验证节点”的硬件要求会显著降低（主要是带宽 / 内存，而不是 TB 级硬盘），

→ 这会让**更多人有能力跑节点**，整体去中心化水平更高；

→ 同时也方便做更轻量级的移动客户端、浏览器客户端等。

现在 Verkle 相关的 EIP 和客户端实现还在持续推进与测试网试运行阶段，主网激活预计会在 Pectra 之后的后续升级中完成；它依然被视为通往无状态客户端和更强扩展性的关键里程碑。

4. 三者放在一起看：各司其职、互相补位

类比：

- **分片 / Danksharding：**

把图书馆拆成很多“临时档案柜 + 多车道”，让所有新文件（Rollup 的批量交易数据）能便宜、快速地存放进去，而且任何人都可以检查这些档案确实存在（DA 采样）。

- **Verkle 树：**

是图书馆内部重新设计的“目录 / 索引系统”，让你证明“某本书/某条记录确实在某个位置”的时候，只需要给出一小段证明，而不必拿一大堆目录页，大大减轻了跑图书馆的成本。

小结：

1. **Danksharding → 把“带宽 & 数据可用性”的瓶颈打开**，让 Rollup 承接大部分执行工作；
2. **Verkle 树 → 把“状态存储 & 节点负担”的瓶颈压缩**，让更多人能轻松运行节点并验证状态；
3. 经典“状态分片”概念，则被融合进这个新路线：不再追求 L1 上所有执行并行分片，而是 **L1 做 DA + 共识，L2 做执行 + 业务逻辑**。

八、DeFi、NFT 与 Layer-2 生态的亮点

如果把以太坊看成一座“金融 + 内容 + 计算”的城市：

- **DeFi：从“挖矿收益”走向“真实资产 + 质押收益 + 再质押”**
- **NFT：从纯投机走向“游戏资产 + 会员身份 + 凭证化资产”**
- **Layer-2：Rollup 大爆发，手续费被 Dencun / EIP-4844 一脚踢下去**

下面分三个部分：

1. DeFi：流动质押、再质押、RWA 成为主线

1.1 流动质押（LST）已经是 DeFi 最大板块

- 以太坊切到 PoS 后，**质押收益 + 流动性** 这件事成了最刚需的 DeFi 赛道：Lido、Rocket Pool、EtherFi 等协议发行的 stETH、wstETH 之类的 LST（Liquid Staking Token），在 DeFi 协议里被当成“带利息的 ETH 抵押品”到处流通。

- 各家数据统计里，**流动质押赛道长期占 DeFi TVL 的最大份额**，稳居头部板块，说明现在链上“赚利息 + 做抵押”的基底资产，很大一部分已经不是裸 ETH，而是各种 LST。

这对 Solidity 学习有个很现实的启发：

> 新协议几乎不会直接跟 ETH 打交道，而是跟一堆“带收益的代币包装层”打交道。
>
> 你在写合约时，得非常熟悉：LST 的利息如何计入、如何作为抵押、清算时如何折价。

1.2 再质押（Restaking）和 LRT：把“安全性”也 DeFi 化

在 LST 上再叠一层的是 **再质押（Restaking）**：

- 像 EigenLayer 这样的协议允许你把已质押的 ETH/LST“再抵押”给各种 AVS（Active Validation Services），为预言机、排序器、数据可用性层等提供安全性，并获得额外收益。
- 为了提高资金利用率，又出现了一批 **LRT（Liquid Restaking Token）**协议（EigenPie、Kelp、EtherFi、Renzo 等），本质上是：

> ETH → LST（比如 stETH）→ 再质押 → 再产出一个 LRT
>
> 用户拿着 LRT 又可以去别的 DeFi 做抵押 / 借贷 / 做 LP。

这类设计的亮点：

- 经济层面：把“安全性提供者”这件事代币化、可组合化，用收益把更多资金吸进整个以太坊安全系统。
- 技术 / 安全层面：
  - 多层质押叠加，一旦底层有 Bug 或 Slash，**风险会通过整个 DeFi 图谱级联传导**；
  - 对 Solidity / 协议设计者来说，如何在接口层**清晰标记风险来源**，就非常关键。

1.3 真实世界资产（RWA）上链：国债、基金、信用等被代币化

RWA（Real World Assets）这两年是真的“拉盘 narrative”，不再是空喊口号：

- 多家项目把 **美国国债、货币基金、私募债、房地产收益权** 等打包成链上代币，提供 4–5% 左右的“类无风险收益”给 DeFi 用户。
- 传统机构也直接下场：例如 2024 年黑石（BlackRock）在以太坊上发行代币化基金 BUIDL，规模迅速突破数亿美元级，并被多个 DeFi 协议当作底层资产或抵押品使用。

这带来的变化：

- DeFi 不再只是「链上互相借来借去的魔法互联网钱」，而是**开始承接现实世界的收益流**（国债利率、租金、应收账款等）。
- 合约设计上，需要考虑 **KYC / 名单白名单、冻结 / 赎回逻辑、监管合规** 等传统金融约束，这和“纯链上匿名资金池”是两套世界观。

1.4 DeFi 基础设施：跨链消息、意图（Intent）和 DeFi+AI

- **跨链消息与共享流动性**

LayerZero、Wormhole、CCIP 等通用消息协议，让“资产跨链 + 状态同步”逐渐从“桥”进化为“跨链调用函数”，用户在前端看到的可能只是：

> “我从 A 链某代币换成 B 链某代币”
>
> 具体路径（桥 +DEX+ 聚合）由后台协议自动算。

- **Intent / 智能路由**

越来越多前端开始走“Intent-centric”路线：

用户只说“我要这个结果”（比如：花不超过 X 美金，帮我换到 Y 种资产），后端+MEV 竞价 + 路由器的工作就是想办法“在不作恶的前提下，最优地满足这个意图”。

- **DeFi + AI**

现在已经有项目用 AI 做：

  - 头寸自动再平衡、杠杆管理；
  - 风控评分、清算预测；
  - “链上代理（AI Agent）”自动在多链 DeFi 之间做策略。

从 Solidity / EVM 的视角看，这意味着你的合约调用者越来越可能是：

> 机器人 + 意图撮合器 + 其他合约
>
> 而不是一个个手动点击的钱包用户。

2. NFT：从 JPG 投机，到“游戏道具 + 会员卡 + 凭证化资产”

2.1 市场从泡沫后恢复，叙事更多元

- 2021 年 NFT 疯牛之后，2022–2023 进入深熊；
- 到 2024–2025，**整体交易量依然远低于高点，但结构发生了明显变化**——不再是 PFP（头像）一统天下，而是：
  - 游戏资产 + 道具；
  - 品牌会员 / 票务；
  - RWA 凭证；
  - 比特币 Ordinals / Runes 等新玩法。

2.2 游戏 / 元宇宙：NFT = 道具 + 身份 + 经济系统接口

Web3 游戏目前被普遍看作 NFT 复苏的一个主要场景：

- 游戏中的 **装备、皮肤、角色、土地** 都可以是 NFT；
- 玩家能够 **自由在市场交易这些资产**，而不是被某个中心化游戏平台“锁死”；
- 部分项目采用 **独立 App-chain / Rollup**，以保障高 TPS 和低手续费。

这类项目对合约要求：

- 标准 NFT（ERC-721/1155）+ 游戏逻辑合约（升级、合成、分解、租赁、抽奖等）；
- 要非常注意：**随机数安全（VRF）、防脚本滥用、防刷子农**。

2.3 实用型 NFT（Utility NFT）：会员、门票、忠诚度

越来越多品牌把 NFT 当成一种“**可验证的会员卡 / 门票**”：

- **会员 / 忠诚度计划**：

品牌发 NFT 作为层级会员，持有人享受专属折扣、周边、线下活动等；

- **票务与 proof-of-attendance**：

像 POAP 一类的出席凭证，用于记录你“参加过什么活动”；

- **链上身份**：

某些项目将“Contributor / OG / 核心成员”固化为 NFT，实现链上 reputations。

从合约层来看，这类 NFT 通常搭配：

- 白名单 / 仅限特定钱包铸造；
- 到期 / 销毁 / 转让限制；
- 与前端 / 后端系统联动（扫码验证、签到等）。

2.4 NFTFi：NFT 抵押、借贷、分片

NFT 的金融化也越来越丰富：

- 抵押借贷（BendDAO、JPEG’d、Paraspace 等）；
- 点对点贷款、租赁市场；
- 将昂贵 NFT“分片”（fractionalization）成可交易的 ERC-20 份额，降低参与门槛。

这里典型风险包括：

- 底层 NFT 地板价操纵导致清算异常；
- Oracle 精度 / 更新频率问题；
- 清算逻辑设计不当导致协议坏账。

2.5 RWA + NFT、证书与身份

NFT 也被用来表达**现实资产或身份凭证**：

- 艺术品、房产、收藏品的所有权 / 份额；
- 学历证书、职业资格、活动证明；
- 某些项目结合 SBT、ERC-6551（Token Bound Account）等，把“**一个 NFT = 一个钱包 + 资产组合**”。

因此在写这类合约时，往往得思考：

- 是否允许转让？可否丢失 / 找回？
- 如何与 Off-chain 的法律 / 注册系统对齐？
- 数据隐私：哪些信息上链，哪些留在链下？

3. Layer-2：Rollup 大爆发 + Dencun 之后的“低费率时代”

3.1 Rollup 成为主流扩容路径

过去两年，“**以太坊 = L1 定义安全 + L2 承载大部分用户交易**”已经基本成共识：

- **Optimistic Rollup**：Arbitrum、Optimism、Base 等
- **ZK Rollup**：zkSync、Starknet、Scroll、Linea、Polygon zkEVM 等

这些 L2 上：

- DeFi 协议开始“多链部署”：LST / DEX / 借贷 / 衍生品在多个 L2 同时上线；
- NFT 与游戏直接部署在手续费更低的 L2，而不是挤在 L1 上。

3.2 Dencun & EIP-4844：L2 手续费断崖式下降

2024 年 3 月的 Dencun 升级引入 EIP-4844（Proto-Danksharding），核心就是：

- 新增 **Blob 交易**，专门为 Rollup 上传批量数据用；
- Blob 的定价远低于原先的 calldata，**Rollup 的数据成本大幅下降**；

- 实测中，很多主流 L2 的普通转账 / Swap 手续费直接打到几美分甚至更低。

这对生态的意义：

- 以前：L2 只是“比 L1 稍便宜一点”；
- 现在：L2 上的交互成本已经低到可以支撑 **高频游戏、社交、微支付、自动策略** 等场景。

3.3 模块化区块链：DA 层、共享排序器、Restaked Rollups

围绕 Rollup，又冒出一整套“**模块化**”叙事：

- **独立 DA（数据可用性）层：**

Celestia、EigenDA、Avail 等，专门做 DA。Rollup 可以选择把数据发到这些 DA 层，而不只依赖以太坊 calldata。

- **共享排序器（Shared Sequencer）：**

多个 Rollup 共用一个排序器，以降低 MEV 垄断、实现跨 Rollup 的原子性。

- **Restaked Rollups：**

结合 EigenLayer 把 “Rollup 安全性绑定到以太坊再质押集”，形成新的安全模型。

从合约视角来看，意味着：

- L2 上的合约调用路径会越来越多样（主网 ↔ L2 ↔ 其它 L2 ↔ DA 层）；
- 需要更频繁处理跨链消息、异步状态、重试和回滚。

3.4 比特币生态的 Layer-2 与 NFT

除了以太坊，**比特币的扩展层也非常热闹：**

- 支付层：Lightning Network 继续负责高频小额支付；
- 智能合约 / 应用层：Stacks、Rootstock、BOB、Merlin 等尝试给 BTC 带上 EVM 或智能合约能力；

- Ordinals / Runes 把“类 NFT / 铭文资产”引入比特币，上面也开始出现 DeFi + NFT + Game 的组合。

对学习 EVM / Solidity 的人来说，这里有两个有趣方向：

1. 在 EVM 兼容的 BTC L2 上写合约（Merlin、BOB 等）；
2. 做“跨资产 / 跨 L1”的统一前端与协议设计，比如 BTC 作为抵押，ETH 生态作为执行环境。

4. 小结

时代背景：

- **DeFi 的重点：**
  - 从“挖矿农耕”转向“质押收益 + 再质押 + RWA 收益”；
  - 协议之间高度组合，很多资产本身就是“多层包装”（LST、LRT、RWA Token）。
- **NFT 的重点：**
  - 从“JPG 赌行情”转向“游戏道具 + 会员卡 + 各种链上凭证”；
  - 走向和现实身份、真实资产的深度绑定。
- **Layer-2 的重点：**
  - 手续费足够低，足以承载真正的“高频应用”；
  - 模块化 + DA + shared sequencer + restaking 把底层结构玩得越来越复杂。

你后面在讲 Solidity 语法、合约设计模式的时候，其实完全可以穿插这些真实协议做例子：

- 用一个 LST / LRT 协议讲 **ERC-20 + 权限控制 + 协议费**；
- 用一个 NFT 会员项目讲 **ERC-721 + 白名单 + 到期 / 转让限制**；
- 用一个 L2 的桥合约讲 **跨链消息、重入 / 安全、Gas 成本控制**。

你不是在学一门“孤立的语言”，而是在学一整套正在高速演化的金融 / 应用基础设施。

九、企业支持以太坊的原因

智能合约的能力和灵活性：

自动化和无需信任：以太坊是第一个引入图灵完备智能合约的区块链。智能合约是自动执行的数字协议，一旦满足预设条件，就会自动执行。这使得企业能够自动化许多业务流程，减少对中介的依赖，从而降低成本和时间，并提高效率和透明度。

定制化和广泛应用：Solidity 语言的灵活性允许开发者构建高度定制化的 DApp，满足各种行业需求，从金融服务到供应链管理，再到数字身份验证。

安全性与透明度：

去中心化安全：作为最大的去中心化区块链网络之一，以太坊在 PoS 机制下拥有强大的安全保障。海量的验证者和经济惩罚机制使得攻击网络变得极其昂贵和困难，确保了交易和数据的不可篡改性。

公开透明：所有交易和合约执行都在公共账本上记录，提供了前所未有的透明度，这对于审计、合规性和建立信任至关重要。

庞大且活跃的生态系统与开发者社区：

领先的生态系统：以太坊拥有最成熟、最活跃的区块链生态系统，包括 DeFi、NFT、Web3 游戏等各个领域。这意味着企业可以利用现有的基础设施、工具和标准，加速其区块链项目的开发和部署。

开发者人才库：以太坊拥有全球最大的区块链开发者社区，这意味着企业更容易找到具备相关技能的人才来构建和维护其基于以太坊的解决方案。

行业联盟：像企业以太坊联盟（Enterprise Ethereum Alliance, EEA）这样的组织，汇聚了全球众多财富 500 强公司和区块链初创公司，共同制定企业级以太坊标准和最佳实践，促进了以太坊在企业中的应用。

可扩展性与成本效益（Layer2 解决方案）：

解决了早期痛点：传统上，以太坊主网的 Gas 费用高和交易速度慢是企业采用的障碍。然而，Layer 2 解决方案（如 Optimistic Rollups 和 ZK-Rollups）的蓬勃发展，极大地解决了这些问题。

低成本和高吞吐量：Layer 2 协议在链下处理大量交易，然后将汇总数据提交到以太坊主网进行最终结算，显著降低了交易费用并提高了吞吐量。这使得以太坊对于需要处理大量交易的企业用例变得更加经济可行。

数据可用性（Blob 交易）：Dencun 升级引入的 Blob 交易进一步大幅降低了 Layer 2 的数据成本，这直接转化为企业和用户在 Layer 2 上更低的费用。

可持续性和 ESG 因素：

能源效率：以太坊从 PoW 到 PoS 的转型，使其能耗降低了约 99.95%。这一巨大的改变使得以太坊成为一个高度可持续和环保的区块链平台。对于日益关注环境、社会和治理（ESG）的企业来说，这是一个非常重要的吸引力，有助于提升其品牌形象和社会责任感。

互操作性和代币化潜力：

跨链能力：以太坊生态系统在跨链互操作性方面不断进步，使得资产和数据能在不同链或 Layer 2 之间更顺畅地流动。

资产代币化（RWA）：以太坊是代币化现实世界资产（RWA）的理想平台，从法定货币（稳定币）到商品、房地产、证券和碳信用。企业可以利用以太坊将实物资产转化为可编程、可交易的数字代币，从而提高资产流动性、降低交易成本并引入新的商业模式。例如，摩根大通（JP Morgan）和贝莱德（BlackRock）等金融巨头都在积极探索基于以太坊的 RWA 代币化。

成熟生态系统与行业标准：

以太坊是第一个支持 Turing-完备智能合约的区块链，其生态包括数十万开发者、丰富工具（如 Solidity、Hardhat、Infura、Alchemy）、以及海量文档资源。

企业可构建私有或许可链，同时兼容主网（如 JP Morgan 的 Quorum、EY 的 Nightfall），保障安全与合规性。

Enterprise Ethereum Alliance（EEA）汇聚了微软、摩根大通等企业，共同制定企业级标准，推动行业共识。

跨国支付与结算效率提升：

通过智能合约可实现即时结算和全球支付，无需传统中介，节省时间与成本，尤其好用于贸易、保险、跨境支付等领域。

十、以太坊未来发展的潜力

如果用一句话概括，以太坊接下来几年是在做三件事：

> 让更多人用得起（扩容），更多人跑得起（去中心化），更多人用得爽（体验 & 协议简化）。

1. 可扩展性：从 Proto-Danksharding 到「百万 TPS」级生态（The Surge）

现在已经发生的：Dencun + Proto-Danksharding（EIP-4844）

2024 年 3 月的 Dencun 升级已经把 EIP-4844（俗称 proto-danksharding）上线，它给以太坊引入了新的“Blob 交易”，专门给 L2 rollup 存批量数据用，比传统 calldata 便宜得多。

结果就是：主流 L2（Arbitrum / Optimism / Base 等）的交易费出现了数量级下降。

这一点非常关键：**扩容已经不再是“纸上方案”，而是确实在账单上体现出来的成本下降。**

接下来要做的：完整 Danksharding + PeerDAS

接下来的路线是把 proto-danksharding 升级为真正的 **Danksharding**：

- 每个区块可以携带更多、更大、更便宜的 **blob 数据**；
- 配套引入 **数据可用性采样（DAS）/ PeerDAS**，验证者只需要抽样检查数据即可，而不必完整下载所有 blob；
- L2 rollup 因此可以在保持安全前提下，把吞吐继续拉高。

这意味着：

> 主网更像“数据可用性层 + 结算层”，而真正的执行与高 TPS 会发生在一圈丰富的 L2 / app-chain 上。

对开发者而言，未来更像是在「以太坊安全层」之上选择不同的执行环境：通用型 L2、游戏链、DeFi 专用链、企业专用链等等。

2. 去中心化、抗审查和轻节点：Verkle / 无状态客户端（The Verge & The Scourge）

今天跑一个全节点，最大的压力其实不是算力，而是：**状态太大、证明太重**。

2.1 Verkle 树 & 无状态客户端

以太坊今天的状态结构是 Merkle Patricia Trie，证明（Merkle 证明）随状态膨胀越来越大。Verkle 树被设计成下一代数据结构：

- **证明更小**：在状态极其庞大的情况下，证明体积仍能保持在很小的级别；
- **更适合“无状态客户端”**：区块可以携带验证所需的证明，客户端不必本地保存完整状态也能验证交易。

路线图里把这一块归到 **The Verge**：

> 让更多人可以用更低成本跑起一个验证客户端，去中心化不仅是「有很多节点」，而是**“任何人都能轻松跑一个节点”**。

Nimbus 团队和核心开发者最近也在探索 Verkle 之外的统一二叉树等方案，但目标一致：**让状态证明和同步成本大幅下降**。

2.2 抗审查与 MEV：PBS 等方向（The Scourge）

另一方面，**MEV + 中心化打包**是当前以太坊的一个隐忧：不少区块是由少数 builder 构建，再由 proposer 引入链上。

路线图中的 **The Scourge** 方向在做的事包括：

- **提议者-构建者分离（PBS）标准化 / 内嵌化**，降低单一 builder 的垄断和审查能力；
- 探索 MEV 捐赠 / MEV-burn 等机制，让「抽取价值」更难被少数主体独占；
- 结合上面的轻客户端 / Verkle，使更多验证者能独立参与，而不是完全依赖少数中介。

这部分还在快速演进，但方向很明确：

> 以太坊的“扩容”不能牺牲抗审查性和开放参与性。

3. 协议简化与运行成本的大扫除（The Purge）

随着 10 年演进，以太坊协议本身积累了大量「历史包袱」：旧历史数据、已不再需要的特殊规则、难以维护的边角逻辑。

The Purge 这一条线的目标，就是协议层面的“瘦身”：

- **历史数据裁剪**：类似 EIP-4444 提出的做法，让节点可以只保留最近一段必要历史，老数据交由专门的历史节点 / 外部方案保存；
- 移除少数已不再需要的复杂特性，简化共识和执行逻辑；
- 降低运行一个全节点的硬盘、带宽和维护成本。

这和上面的 Verkle / 无状态客户端是互相配合的：

> 通过削减“必须保存的东西 + 必须理解的复杂度”，让以太坊作为协议本身更容易被实现、审计和长期维护。

4. 账户抽象、智能账户与用户体验（The Splurge & Pectra）

用户体验上，以太坊正在往一个目标前进：

> “让钱包像一个可编程的 App，而不是一把只会签名的钥匙。”

这里最关键的一步，就是账户抽象（Account Abstraction）和智能账户的能力提升。

4.1 Pectra 升级：EIP-7702 + EOF

重要升级 Pectra 已明确会纳入：

- **EIP-7702**：允许 EOA 在特定交易中临时挂接一段合约逻辑来执行，使原本“只有私钥”的外部账户，拥有类似智能合约钱包的行为（批量操作、权限控制、代付 Gas 等），但又不强制你迁移地址。
- **EOF（EVM Object Format）**：一种新的 EVM 程序封装格式，重构字节码布局和验证规则，让合约代码更易分析、安全性更高，也为后续优化铺路。

配合 Dencun 之后更便宜的 L2，账户抽象、社交恢复、多签、支付代币化 Gas、批量交易等能力，会逐渐从「高阶玩家的玩具」变成**普通用户看不见但默默享受的基础设施**。

5. 更强的安全性与最终性：单槽最终性等长期研究

在 PoS 成熟后，以太坊还有一些更“学术向”的研究方向：

- **单槽最终性（Single-Slot Finality, SSF）**：希望在一个 slot 内就给出强最终性，而不是当前的多 epoch 终局时间。这会让以太坊更适合做高价值结算层和跨链基石。
- 更强的恢复 / 社区干预工具：在极端情况下如何惩罚大规模作恶质押者并恢复链的健康。

这些大多还在研究和原型阶段，但目标很清晰：

> 把以太坊打造成“高价值结算层 + 强安全 L1”，而不是只拼 TPS 的执行链。

6. 生态层的延展：L2、RWA、DeFi、NFT、AI……

协议层之外，前面我们已经学过一整节 DeFi / NFT / L2 的现状，这里可以把“潜力”高度概括成三条：

6.1 L2 作为“新执行层”全面爆发

- Proto-danksharding 之后，L2 手续费继续下探，体验更接近 Web2；
- 泛 Rollup + app-chain 组合，让每个垂直领域都可以拥有为自己调优的执行环境。

6.2 RWA + 机构化资金的长期进场

- 国债、信用、商品、股权等资产逐渐被代币化，DeFi 不再只围绕“链上原生资产”，而是把**真实收益**拉到链上；
- 监管、审计、合规基础设施随之成熟，形成“合规 DeFi / on-chain finance”。

6.3 身份、AI 与链上自动化

- 基于 NFT / soul-bound token 的身份与凭证系统，让访问控制、声誉、会员体系上链；
- AI 用于做风控、策略优化、自动化交易 / 清算，链上变成一块高透明、高自动化的金融 & 游戏实验场。

7. 小结

① 让更多交易“装得下”——扩容与 Rollup / Danksharding（The Surge）。

② 让更多人“跑得起”——Verkle / 无状态客户端 / 协议瘦身（The Verge & The Purge）。

③ 让更多人“用得爽”——账户抽象、智能账户、L2 低费与更顺滑 UX（The Splurge）。

以太坊未来更像是一个安全结算 + 数据可用性核心层，外面包着一圈高速、便宜、可定制的执行层生态。

理解以太坊的共识，就是看到信任如何被工程化；让我们以好奇和耐心，共同见证以太坊生态的持续演进。

——@leopc999（乐乐）

寄 语 读 者

感谢所有的读者和支持者，以太坊的未来由大家共同书写。——@XiaoHai67890