区块链系统入门

区块链极简概述

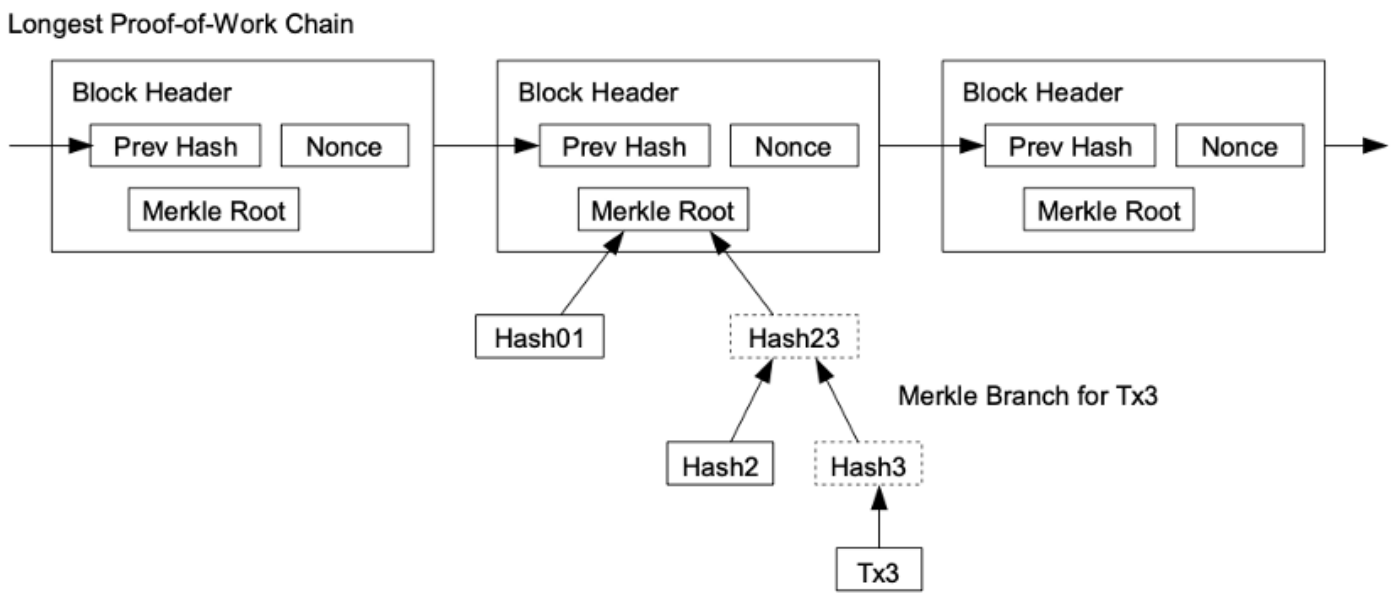
区块链是一种分布式账本技术。在2008年，匿名者中本聪发布白皮书《Bitcoin: A Peer-to-Peer Electronic Cash System》，并在2009年开始运行比特币网络。之后随着比特币的成功，人们将其应用的技术，概括成为区块链技术。

比特币的根本目标就是要建立一套去中心化的，在零信任的环境下运行的电子货币系统。从这个角度出发，可以认为区块链的**本质就是要解决去中心化、去信任中介的问题**。

要实现去中心化这个目标，首先就要求整个系统是开放的，整个系统的运行只依赖于参与者（网络维护者、矿工）形成多数共识，即大家都按既定的、公开明确的规则行事，而不依赖于任何少数个体/机构，这也就是所谓的 **Code is law**，而这也意味着区块链系统必须是**开源**的；在此基础上任何人（包括潜在的网络维护者即矿工）都可以在一个无许可的环境下自由参与或退出，而不受任何中心化机构的审批/审查。

在这种环境下，要让所有参与者在互不信任的基础上，实现对网络运行状态持续达成共识，关键是要解决记账权以及记账出现冲突（分叉）时的选择问题，这就是区块链系统运行的核心机制“共识机制” 要研究的课题。

其次，在去中心化、去信任中介的前提下，为了解决身份识别、账本查询与追溯、防止事后篡改或抵赖等问题，这就涉及到使用非对称加密、数字签名、单向信息摘要等密码学技术来构建账号体系，鉴证对账本变更的请求（即交易）。同时，比特币将一段时间内的交易使用摘要树（Merkle tree）的形式组织起来，连同其他一些信息组成一个区块（Block），每个区块都基于前一个区块的状态 加上当前区块基于交易的状态转移 而形成，因此每个区块都引用前一个区块的摘要，从而形成了对整个状态变更历史的链式追溯结构。这也就是“区块链”这个名称的由来。



图：区块结构图，来自比特币白皮书

最后，既然整个网络是在没有中心化可信机构的环境下运行的，那就还要解决参与者相互之间通信的问题，这就是区块链系统中的一个重要组成部分，p2p网络，网络中的节点以一种对等的方式接收、独立验证、提供数据。当然，存在一个新参与者（新节点）加入的问题，新节点需要一个可信的“网络引介者” 即 Bootnode 来做初始连接进而通过p2p节点发现协议来发现并连接网络中现有的其他节点。这看似引入了中心化信任问题，但实际上并不要紧，因为任何节点都可以公开提供bootnode服务，并且恶意的bootnode最多只会让连接它的新节点无法成功加入网络而已，并不会造成其他安全问题。总体而言，p2p网络是比较成熟的技术，一般受到的关注比较少，除非涉及区块链系统节点通信方面的优化研发。

综上所述，笔者认为**区块链系统就是一堆独立参与者在互不信任的前提下，通过p2p网络，依据某种共识机制实现对一个全局账本进行一致性记账，并以链式结构对账本历史进行记录和追溯的，去中心化、开放开源、抗审查的系统**。

需要明确，上面所讨论的，都属于“公共区块链”的范畴。受“公共区块链”相关技术和思想的启发，人们也在尝试使用相关的密码学技术以及对状态变更的组织方式，对历史记录的链式组织及追溯的方式，基于降低信任成本、降低审计成本、实现可追溯不可篡改等目的，研究相关技术在有准入、受控的环境下的应用，形成了“联盟链”的概念。这属于原始“公共区块链”的一个衍生分支，但主要只有技术方面的相通性，其本质已经有很大的不同。

本文将只讨论“公共区块链”的内容。

部分知名公链简介

作为区块链系统的入门，我们有必要简要了解一下几个知名公链。

BTC

比特币由匿名者中本聪创建，是区块链的鼻祖。

比特币是加密货币系统，具有极强的金融属性，而这也是它的唯一目的。

比特币使用工作量证明机制（Proof of work，PoW）作为其共识机制，实现了真正的去中心化、去信任中介的目标。

简而言之，参与竞争记账权（“挖”区块）的参与者即“矿工”，使用暴力搜索的方式基于“确定的信息+自己选择的随机数” 计算一个密码学Hash值（SHA-256），如果hash值的二进制位前导 0 的个数符合目标要求，则矿工就获得当前记账权，可以打包交易形成区块向全网广播。矿工成功打包区块将获得交易手续费奖励以及额外的区块奖励，从而激励矿工去积极维护网络的运行。当不同的矿工在相近的时间内都获得了记账权时，网络将会出现不同的“分支”，这称为“软分叉”或直接叫“分叉”，出现分叉时比特币采用“最长链”原则进行选择，即最长（拥有最多区块）的分支，代表了拥有最多的算力支持，因而诚实的节点就应该选择这条最长的链。

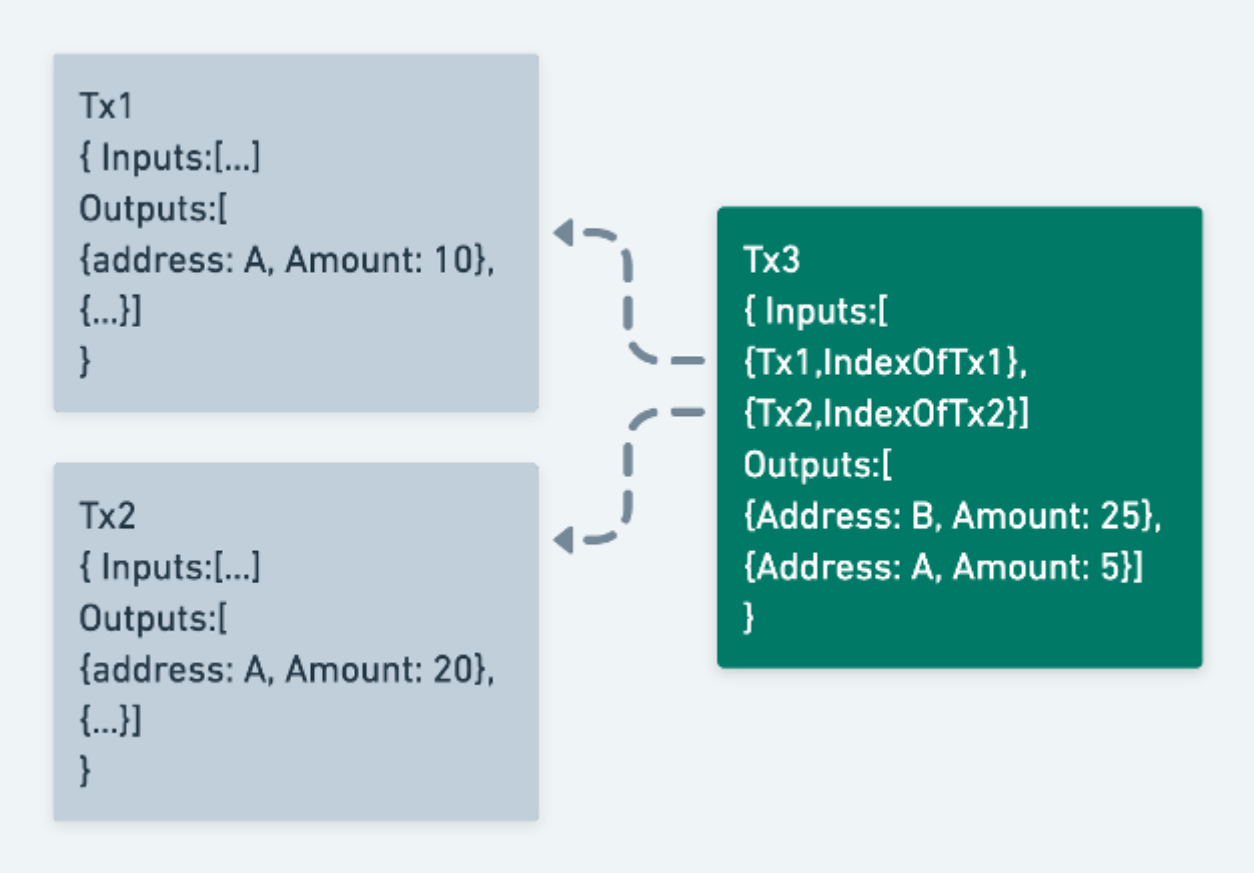
比特币基于PoW+最长链原则，解决了记账权以及双花问题，但存在“区块确认”问题，也就是一笔交易被矿工打包到一个区块（执行生效）之后，不能立即认为它就是最终状态不可回滚了，实际上由于最长链原则，新区块都有可能被回滚，从而交易由成为未被网络接纳的状态。关于这个问题，唯一的解决方案是等待交易所在区块之后链接上一定数量的区块，从而可以认为网络中再也没有足够的算力可以回滚该区块了，我们才能认为该交易的结果可以被安全接受，这就是“安全确认数”的概念。在比特币中，一般安全确认数可以设定为 6 。

比特币的PoW机制，是基于“one-cpu-one-vote”的思想，完全由算力来竞争出块权，经实践证明是一种真正无准入、去中心化的共识机制。但是这种机制也导致了大量的能源消耗，这也是比特币被诟病的一个主要方面。另外，PoW机制，存在“51%攻击”问题，即控制了超过一半的算力之后，将可以控制整个网络，可以通过制造区块回滚而发起“双花”攻击。

比特币基于难度值来控制网络中区块产生的速度，目标是10分钟产生一个新区块。每隔2016个区块就调整一次难度值，如果区块间隔小于10分钟，难度值将变大，反之难度值变小，从而使区块间隔向10分钟靠拢。

比特币（以及所有的区块链系统）的账号，都是基于非对称加密技术，具体而言，由于椭圆曲线密码学（相比于RSA非对称加密技术）具有密钥长度短，安全位宽高的特点，因此在区块链中一般都是基于椭圆曲线密码学技术。比特币具体采用的是 secp256k1 曲线。账号由一对公私钥密钥对组成，私钥基于数字签名来参与一切网络活动，**私钥即一切！**公钥可在网络中公开，基于公钥推导出地址，地址是用户身份（私钥）在网络中的唯一代表。

比特币上的账户余额模型，采用的是UTXO模型，即“未花费交易输出”模型。在UTXO模型中，账号在区块链上没有统一的余额记录，交易只是代表了UTXO集合的变更，而账号余额则是在钱包层面更高一层的抽象，也就是钱包会汇总该账号当前所有的“未花费交易输出”从而获得该账号当前的余额。在比特币中，交易（转账）中涉及的余额，由若干“输入”形成，然后在“输出”中对其进行重新分配，从而就完成了转账。交易的抽象示意如下图所示，其中 Tx3 的输入是 Tx1，Tx2（交易哈希表示），给B转账了25btc，给自己找零5btc（比特币钱包一般会使用自己的其他新地址作为找零地址，这里不展开描述）。



比特币交易示意图

Ethereum（及EVM生态）

2013年底，年轻的比特币爱好者Vitalik Buterin（人称 V神）分享了他的《以太坊白皮书》，提出要构建“**一个图灵完备的可编程和通用区块链**”，之后在技术大神 Gavin Wood 的加持下，以太坊于 2015/7/30 正式启动运行，开启了其公链第一大生态的发展之旅。

在共识机制方面，以太坊也沿用了 PoW 共识机制，但加入了对内存的要求，避免对算力的单一竞争。而随着以太坊的发展，以太坊已于 2022/9/15 成功转换共识机制为 PoS（参见 The Merge）。

关于账号余额等信息的记录，以太坊采用了Account模型，在每一个区块高度，都维护有当前所有账号的最新状态，每个账号都记录了其最新的余额等信息，所有账号通过 MPT （Merkle Patricia Trie，一种前缀型默克尔树）结构组织起来，称为状态树，树根记录在区块头中，从而实现不可篡改特性。由于每个时刻都存在这样一个全局一致的账号状态，因此也称为 世界态。

以太坊创建了一种图灵完备的编程语言 Solidity，用户基于该编程语言可编写在以太坊网络上运行的智能合约（Smart Contract），以太坊通过一个状态机来执行所有的状态转换逻辑，包括用户编写的智能合约逻辑，这个状态机称为 以太坊虚拟机 （EVM）。

上述Account模型 以及 Solidity+EVM，形成了以太坊作为“图灵完备的可编程和通用区块链”的基石。如果说单纯作为数字货币系统的比特币是区块链1.0，则可以支持通用应用编程的以太坊无可争议的代表了 区块链2.0 。正是由于以太坊，才有了公链应用生态的繁荣发展。

> 扩展：关于UTXO模型和以太坊的账号模型，各自优劣势深入的比较可以参考：深入解剖比特币和以太坊，对比 UTXO 和 Account 模型优劣

随着以太坊应用生态的发展，以太坊的性能瓶颈愈加明显地成为链上应用发展的最大瓶颈。区块链扩容成为以太坊发展的第一要务。但是由于V神是坚定的去中心化信仰者，其坚持以太坊的扩容不能降低以太坊的去中心化程度，其中保证用户以较普通的硬件设备方便地加入以太坊网络，是保证以太坊去中心化程度所要考虑的一个重要因素，由于这诸多的因素，以太坊对各种扩容方案都极其谨慎，其发展也颇为缓慢。

鉴于以太坊本身在扩容方面的制约及缓慢的进展，行业内众多机构、创业者提出了其他多种称为“链下扩容 Off-chain scaling”的方案，其中一类称为**二层扩容**（layer-2），包括各种rollup方案如 zk-rollup, op-rollup，以及状态通道 等，这些方案将大量的计算工作放到链下（即所谓的二层网络），而只将相关数据以及状态转换的验证工作放到以太坊主网上，从而既能完全继承以太坊主网的安全性，又能实现较大程度的网络扩容。另外一些解决方案，则自己创建一条新的区块链，但会跟以太坊主网通信，在自身的安全机制基础上，额外增加了资产撤退到以太坊主网上的安全机制，这类方案如侧链扩容、Plasma Chain、Validium等，比如当前比较有名的 Polygon PoS链就是融合了侧链和Plasma方案的公链。

另外还存在较多独立的、以性能提升为目标、不惜牺牲去中心化程度的EVM兼容公链，如以 BSC（现称为 BNB Chain）为典型代表的交易所公链。这类公链客观上承接了以太坊应用的溢出需求，扩大了EVM应用生态，在一定程度上也促进了链上应用的繁荣发展。

此外，还有一些本来是要独立发展的，架构、机制与以太坊完全无关的公链，也在发展过程中逐渐增加了对 EVM 的兼容，或者基于原有的架构改造出可兼容EVM的链，如 Avalanche、Tron、基于Cosmos的 EVMOS、NEAR Protocol 上的 Aurora 等等。在笔者看来这些都属于蹭流量，而这也体现了以太坊生态及开发者群体的庞大，凸显出当前以太坊作为应用公链之王的无可撼动的地位。

其他较知名公链

其他比较知名的公链，包括以跨链为目标致力于实现应用链之间的去中心化通信，从而形成像互联网一样的区块链网络 的 Cosmos、Polkadot；主打单链极致扩容的 Solana；提出较具创新的雪崩协议的 Avalanche；具有图灵奖创始人光环，创新引入VRF密码学抽签机制的 Algorand；发展较早已形成一定影响力的 Tron；最早引入DPoS共识机制红极一时但又较快陨落的 EOS 等等。对于这些项目，感兴趣的可以了解一下，此处就不一一介绍了。

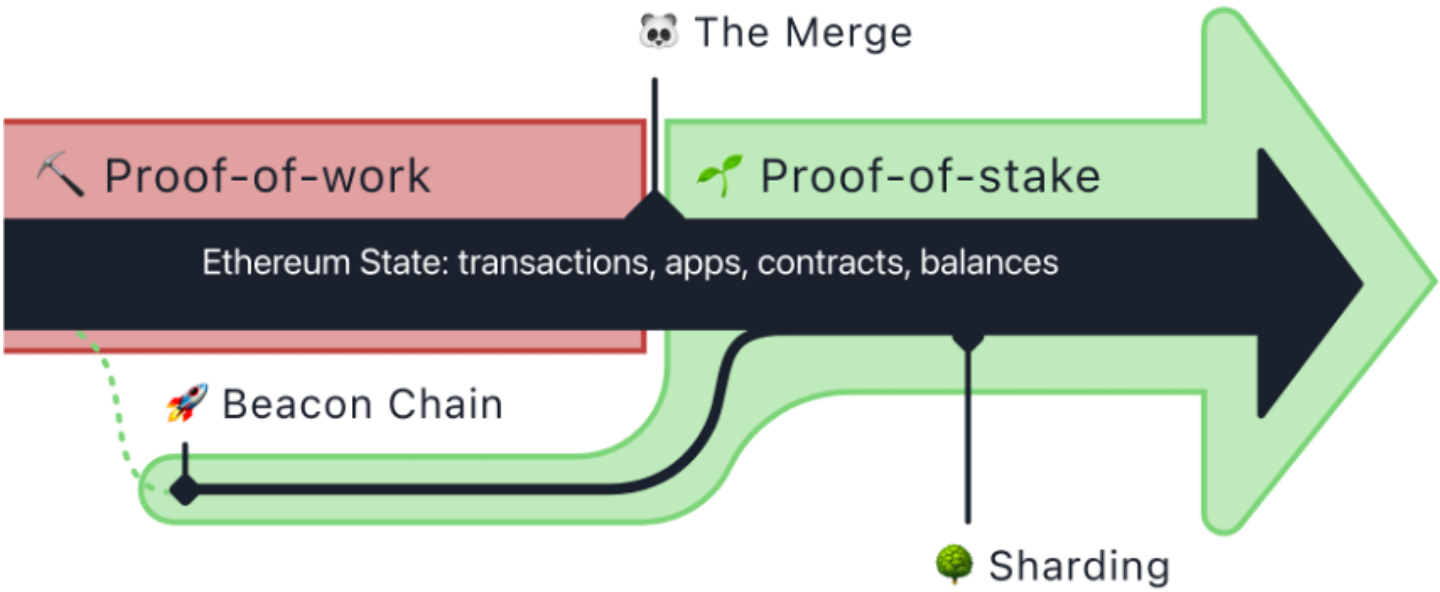
当然，公链项目层出不穷，具有影响力的公链也远不止上述这些，本文就不加赘述了。

以太坊

重大架构变更 The Merge：从PoW 到 PoS

以太坊为了将共识机制从 PoW 变更 为 PoS，同时要尽量保持以太坊的高度去中心化。为了实现这一目标，以太坊社区启动了eth2.0 项目，进行了长时间的研发及测试，并最终在 **2022年9月15日**，成功实现了共识机制从PoW 到 PoS的变更，这个升级的过程，被称为 The Merge，是以太坊发展史上的一个重要里程碑。

在The Merge之后，以太坊架构发生了重大变更，此处将其做一个概要介绍，更多内容详见：ethereum.org: The Merge



概要总结

1、以太坊的PoS架构，由共识层 和 执行层 组成；共识层之前称为 信标链（Beacon chain），而执行层之前也叫 Eth1，也就是原来的PoW链。信标链其实早在 2020年12月已启动，在合并前保持独立运行。The Merge 就是指将信标链和PoW链这两条链进行合并。在合并之后，信标链称为共识层，接替ETH1的PoW机制，负责以太坊网络的共识；而原来的ETH1 变成了只负责状态转移的执行层。

2、信标链即共识层客户端是独立发展的，坚持多客户端实现原则，主要的实现包括Lighthouse、Prysm 等，见 https://ethereum.org/en/developers/docs/nodes-and-clients/#consensus-clients

3、执行层客户端即原来的 eth1客户端，主要是 go-ethereum 等。

4、运行一个节点需要同时运行两个客户端，即共识层客户端和执行层客户端。两者需要一一关联，两者关系为：由共识客户端主动驱动执行客户端执行区块构建、区块验证等业务，即执行客户端不会主动调用共识客户端的任何接口。

共识层和执行层各自有一条链。

共识客户端负责：

1）Validator抵押管理、提议及验证共识层区块（共识层区块包含执行层区块完整数据，不包含执行层状态）、投票等共识相关业务；

2）BP调用执行客户端构建执行层区块数据；调用执行客户端验证执行区块、更新执行层链；

3）共识层p2p网络通信

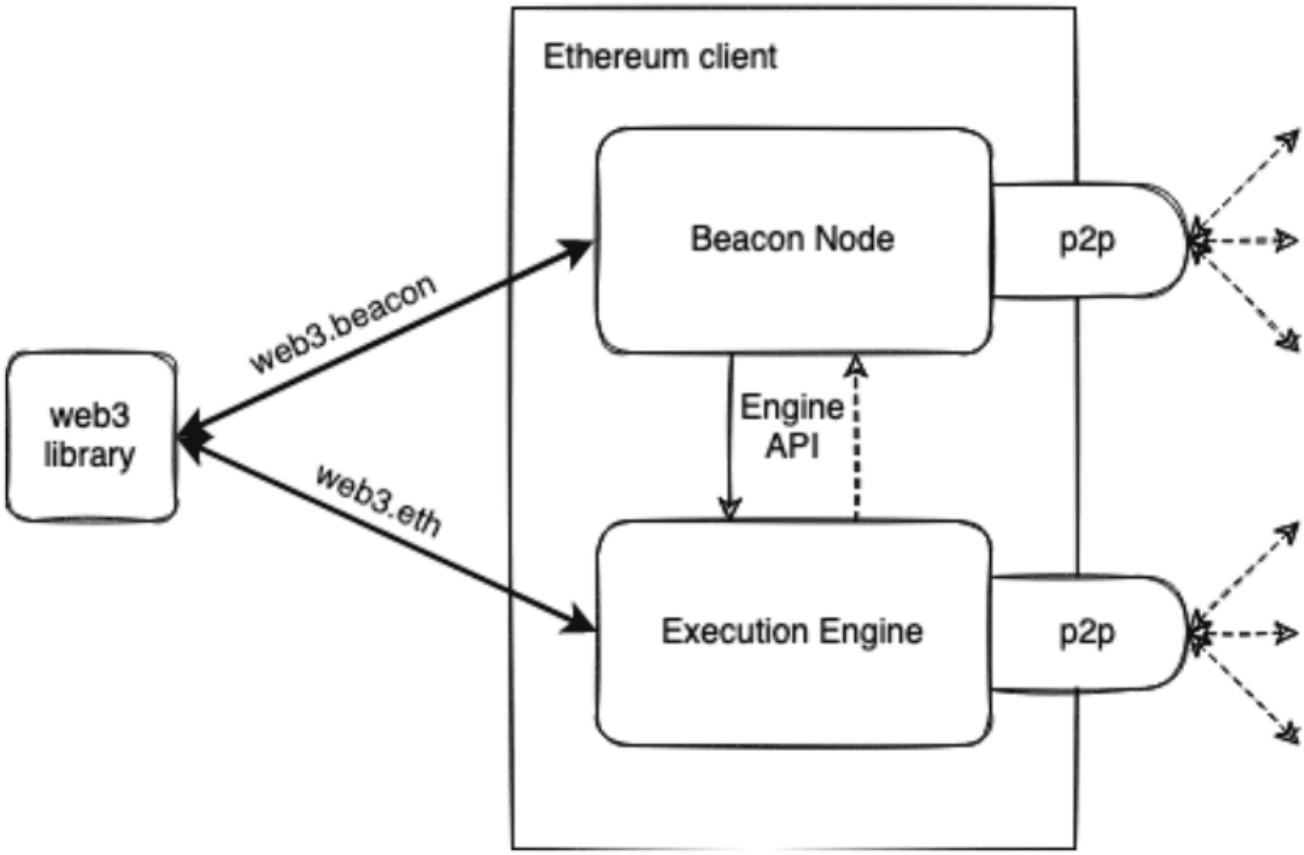
执行层客户端负责：

1）接收共识客户端提供的执行层区块数据，验证执行层区块，维护执行层链（共识层确定哪个分支是权威分支）的状态，即负责EVM虚拟机；

2）执行层txpool

3）执行层p2p网络（涉及交易、状态等的同步，不涉及区块同步，区块数据由共识客户端提供）；

4）供共识客户端调用的共识接口。



核心概念

账号

前面有提到，以太坊采用的是全局统一的账号模型，每个账号都维护有一份最新的状态，里面记录了该账号当前的状态记录。

数据结构

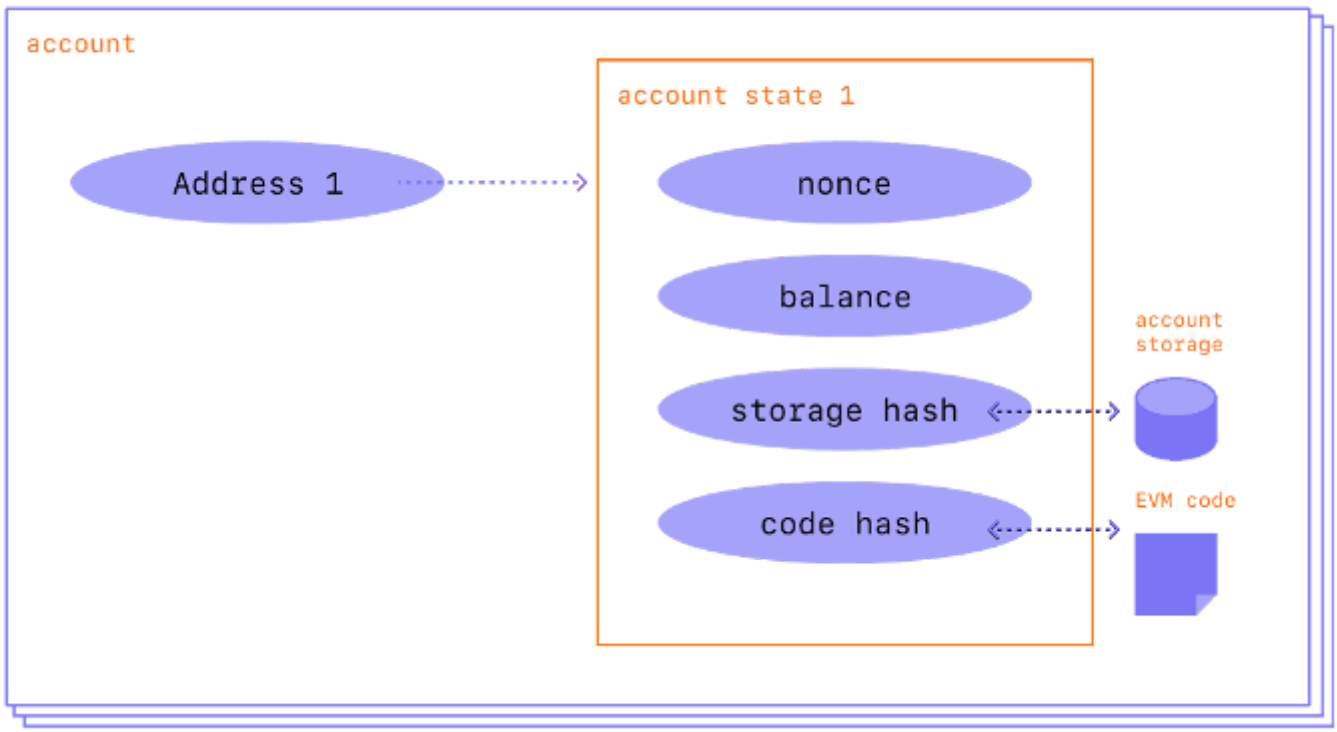
以太坊使用一个逻辑层面的状态数据库 `StateDB` 来管理账户状态及其变更， `StateDB` 管理的账号对象称为 `stateObject` ，其中包含了账号的具体信息以及当前对账号的变更缓存。

具体账号数据结构如下：

```go
// StateAccount is the Ethereum consensus representation of accounts.
// These objects are stored in the main account trie.
type StateAccount struct {
    Nonce    uint64
    Balance  *big.Int
    Root     common.Hash // merkle root of the storage trie
    CodeHash []byte
}
```

字段解释：

- Nonce 可以理解为账号通过发送交易的方式跟链交互的次数，用于保证账号发起的每笔交易都是唯一的，不可重复执行的。每笔交易都将携带Nonce值，而且同一个账号的不同交易的Nonce值必须单调递增，并被区块链依次处理。Nonce值小的交易未被处理时，后续的交易也都不会被处理。一笔交易被打包进区块后（上链后），账号的Nonce值也会加1。
- Balance 即账号的 以太币余额，以 wei 为单位记录。`1 ETH = 10^18 wei`
- Root 和 CodeHash 对“智能合约账户”才有实际值。智能合约代码以 `CodeHash` 加一个固定前缀为 key 存储在数据库中，将CodeHash 存储在账号结构中；合约内部的数据最终也都按 key:value 的形式组织，并构建成一颗独立的树，称为合约存储树，树根即这里的 Root。通过这种方式，构建世界状态树时就只有 StateAccount 这里的4个值真正参与了树节点的组成，从而减少计算节点hash时的数据量。



账户状态示意图（源自ethereum.org)

账号类型

以太坊有两种帐户类型：

- 外部持有账户（EOA） – 私钥的所有者控制
- 合约 （Contract）– 一种由代码控制，部署在网络上的智能合约。

这两种账户类型都能：

- 接收、持有和发送 ETH 和 token
- 与已部署的智能合约进行交互

主要区别

外部持有

- 创建帐户是免费的
- 可以发起交易
- 外部所有的帐户之间只能进行 ETH 和代币交易

合约

- 创建合约存在成本，因为需要使用网络存储空间
- 只能在收到交易时发送（内部）交易
- 从外部帐户向合约帐户发起的交易能触发可执行多种操作的代码，例如转移代币甚至创建新合约
- 合约账户没有私钥，它们被智能合约代码所控制

交易与收据

交易是由帐户发出，带密码学签名的指令。 帐户将发起交易以更新以太坊网络的状态。 最简单的交易是将 ETH 从一个账户转到另一个帐户，而更为普遍且重要的是跟链上合约的交互，这是整个去中心化应用（DApp）生态的基础。

改变 EVM 状态的交易需要广播到整个网络。 任何节点都可以广播在以太坊虚拟机上执行交易的请求；此后，验证者将执行交易并将由此产生的状态变化传播到网络的其他部分。

交易需要付费并且必须包含在一个有效区块中。

交易包括下列信息：

- `to` (有些地方也用 `recipient` 指代） – 接收地址（如果为一个外部持有的帐户，交易将只是普通转账。 如果为合约帐户，交易将执行合约代码）
- `signature` – 发送者的标识符。 当通过发送者的私钥签名交易来确保发送者已授权此交易时，生成此签名。
- `nonce` - 在以太坊的账户模型下，用户发起交易时不再需要像UTXO模型下的交易那样依赖先前的交易哈希，而在这种情况下为了防止交易的重复执行，以太坊的每个账号都记录了一个连续单调递增的nonce值，交易中携带nonce值用于区分同一个账号发起的不同交易。

- `value` – 发送人向接收人转移的以太币金额（以 WEI 为单位）
- `data` – 可包括任意数据的可选字段；对于要执行合约代码的交易，此字段值应该是按ABI编码格式对要调用函数及其入参编码后的数据
- `gasLimit` – 交易可以消耗的最大数量的燃料单位。 燃料单位代表计算步骤
- `maxPriorityFeePerGas` - 作为验证者小费所愿意支付的最大价格
- `maxFeePerGas` - 愿意为交易支付的最大gas价格（包括 `baseFeePerGas` 和 `maxPriorityFeePerGas` ）

Gas是指验证者（PoW共识下称为 矿工）处理交易所需的计算消耗，用户必须为此计算支付费用。 具体在见后面“费用”部分的说明。

一笔普通转账交易看起来像这样：

```
{
  from: "0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8",
  to: "0xac03bb73b6a9e108530aff4df5077c2b3d481e5a",
  gasLimit: "21000",
  maxFeePerGas: "300"
  maxPriorityFeePerGas: "10"
  nonce: "0",
  value: "10000000000",
}
```

但交易对象需要使用发送者的私钥签名。 这证明交易只可能来自发送者，而不是欺诈。

比如，通过以太坊客户端 geth 的 `account_signTransaction` 签名后的返回类似如下：

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "raw": "0xf88380018203339407a565b7ed7d7a678680a4c162885bedbb695fe080a44401a6
    "tx": {
      "nonce": "0x0",
      "maxFeePerGas": "0x1234",
      "maxPriorityFeePerGas": "0x1234",
      "gas": "0x55555",
      "to": "0x07a565b7ed7d7a678680a4c162885bedbb695fe0",
      "value": "0x1234",
      "input": "0xabcd",
      "v": "0x26",
```

```
      "r": "0x223a7c9bcf5531c99be5ea7082183816eb20cfe0bbc322e97cc5c7f71ab8b20e",
      "s": "0x2aadee6b34b45bb15bc42d9c09de4a6754e7000908da72d48cc7704971491663",
      "hash": "0xeba2df809e7a612a0a0d444ccfa5c839624bdc00dd29e3340d46df3870f8a30
    }
  }
}
```

其中

- `raw` 是已签名交易的 RLP（Recursive Length Prefix）编码形式。
- `tx` 是已签名交易的 JSON 形式。

以太坊有**几种不同类型的交易**：

- 常规交易：从一个EOA帐户到另一个EOA帐户的交易。
- 合约部署交易：没有“to”地址的交易，数据字段用于合约代码。
- 执行合约：与已部署的智能合约进行交互的交易。 在这种情况下， “to”地址是智能合约地址。

交易生命周期

交易提交后，就会发生以下情况：

1. 交易完成签名后，就可以最终确定不可更改，此时将通过密码学算法生成唯一的交易哈希，您需要记住此哈希以便后续查找交易的执行结果，交易哈希如：
   `0x97d99bc7729211111a21b12c933c949d4f31684f1d6954ff477d0477538ff017`
2. 然后将该交易转播到网络，并且与大量其他交易一起包含在节点中的**交易池**中。
3. 验证者按照一定策略选择一批交易并将它们包含在一个区块中，通过EVM执行交易完成状态更改并记录结果。
4. 随着时间的流逝，包含你的交易的区块将升级成“justified”状态，然后变成“finalized”状态。通过这些升级，可以进一步确定 你的交易已经成功并将无法更改。 区块一旦“finalized”，就可以安全地认为它不会被回滚了。

收据receipt

交易被包含在区块中之后（肯定经过了EVM执行），将产生一个对应的收据 receipt，该收据记录了交易执行结果的一些信息，包括交易是否“成功”，消耗了多少gas等，如果是跟智能合约相关的交易，还可能包含合约代码内定义及记录的“事件日志”。

例如，以太坊上一笔针对 ERC20 合约币转账的交易收据，从节点查询的结果如下：

```
{
  "blockHash": "0xc4fad2e9821ec7c7f899316d9b0cce5586792e563d4c0966a0be6b3b3e5299
  "blockNumber": "0xf4bfea",
  "contractAddress": null,
  "cumulativeGasUsed": "0x809ca9",
  "effectiveGasPrice": "0x28aa03fa2",
  "from": "0x0da808d51f07ab111fbbcd62c40b898d68bb4211",
  "gasUsed": "0xb41d",
  "logs": [
    {
      "address": "0xdac17f958d2ee523a2206206994597c13d831ec7",
      "topics": [
        "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
        "0x000000000000000000000000da808d51f07ab111fbbcd62c40b898d68bb4211",
        "0x000000000000000000000000f3e50c433e87f1127d3f3d9bd39ad845c898ee73"
      ],
      "data": "0x0000000000000000000000000000000000000000000000000000000177825f0
      "blockNumber": "0xf4bfea",
      "transactionHash": "0xcce32933c791636c66e5a27c1470323c3ff3315dbd7ea8134cee
      "transactionIndex": "0x6f",
      "blockHash": "0xc4fad2e9821ec7c7f899316d9b0cce5586792e563d4c0966a0be6b3b3e
      "logIndex": "0xb4",
      "removed": false
    }
  ],
  "logsBloom": "0x0000000000000000000000000000000000000000000000000000000001000
  "status": "0x1",
  "to": "0xdac17f958d2ee523a2206206994597c13d831ec7",
  "transactionHash": "0xcce32933c791636c66e5a27c1470323c3ff3315dbd7ea8134cee7907
  "transactionIndex": "0x6f",
  "type": "0x2"
}
```


TYPED TRANSACTION ENVELOPE 交易

以太坊最初有一种交易形式。 每笔交易都包含 Nonce、燃料价格、燃料限制、目的地地址、价值、数据、v、r 和 s。 这些字段采用 RLP 编码，如下所示：

`RLP([nonce, gasPrice, gasLimit, to, value, data, v, r, s])`

以太坊经过演变，已经支持多种类型的交易，从而能够在不影响传统交易形式的情况下实现访问列表和 EIP-1559 等新功能。

EIP-2718：类型化交易封套定义了交易类型，是未来交易类型的“封套”。

EIP-2718 是用于类型化交易的新通用封套。 在新标准中，交易被解释为：

`TransactionType || TransactionPayload`

其中，字段定义如下：

- `TransactionType` - 一个在 0 到 0x7f 之间的数字，总共为 128 种可能的交易类型。
- `TransactionPayload` - 由交易类型定义的任意字节数组。

当前的交易类型定义包括：

```
// Transaction types.
const (
    LegacyTxType = iota
    AccessListTxType
    DynamicFeeTxType
)
```

交易费用

交易需要消耗 gas。

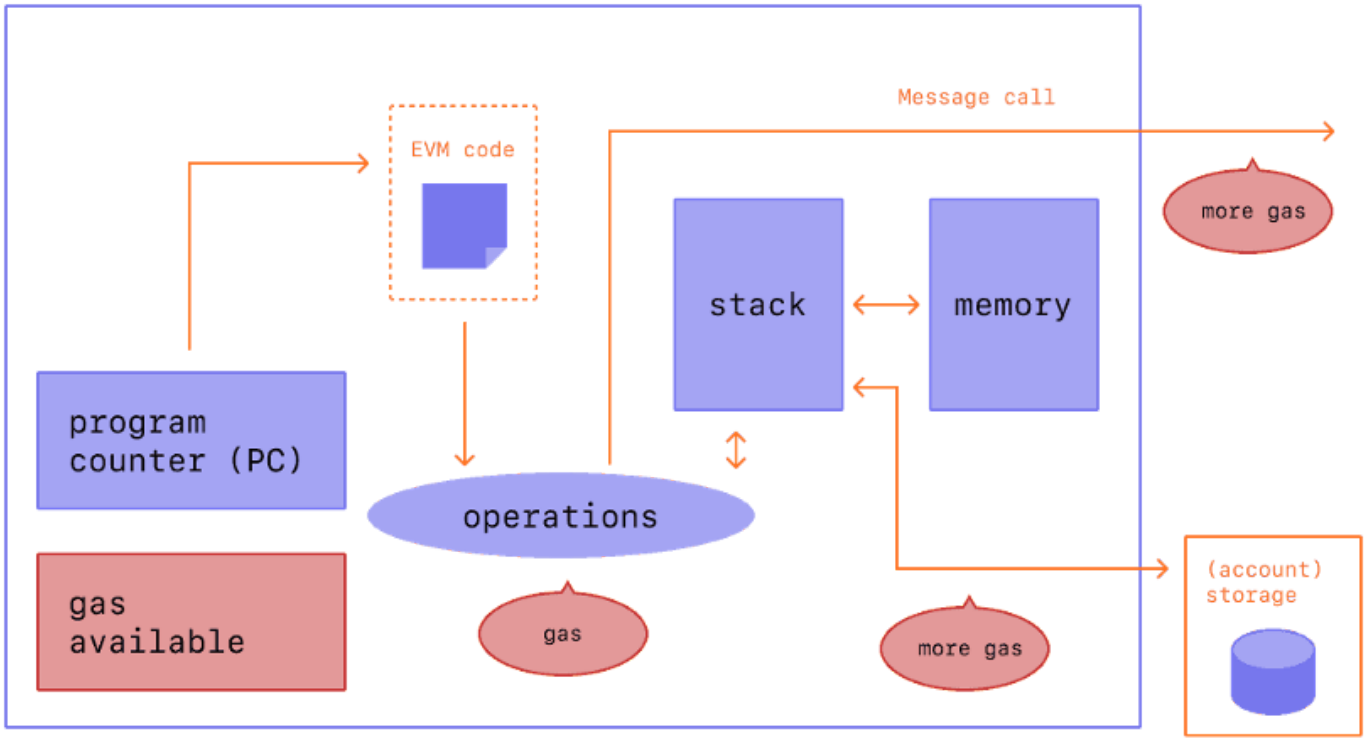
Gas 是指在以太坊网络上执行特定操作所需的计算工作量。

由于每笔以太坊交易都需要计算资源才能执行，每笔交易都需要付费。 在这个方面上，Gas 是指在以太坊成功执行交易所需的费用。

以太坊定义了复杂的gas计量机制，使得EVM中执行的每一个操作，都尽量对应其资源消耗成本，从而最大限度预防各种DoS攻击。

交易中的 `gasLimit` ，设置了用户愿意为交易所消耗的最大gas数量。若是跟智能合约交互的交易并且 `gasLimit` 小于交易实际所需要消耗的gas数量，则交易会以“失败”的状态被打包进区块中。

> 在以太坊中，EOA账户之间的普通转账交易，消耗 21000 gas，常见的ERC20 合约币转账，需消耗 5万gas 左右；而类似Uniswap这样的DEX应用，做一次交易可能要消耗 10万~30万 gas。



交易费用 = 交易消耗的gas * gas价格

由于gas只与交易的执行操作有关，因此无法通过gas来做交易优先级的竞争。以太坊通过给gas设定价格，从而使交易费用可以动态调节，而这个调节是一个完全的市场机制。

交易费用将从用户账户中扣除，在EIP-1559之前，交易费用奖励给生产区块的矿工。在EIP-1559之后，交易费用分成了两部分，一部分基础费用将被销毁，另一部分（可以没有）给矿工的小费才是奖励给矿工。

gasPrice 以 wei 为单位。在最原始的交易类型（LegacyTx）中，gasPrice 是完全由用户构造交易时指定的并且指定后就不再变化的。而在EIP-1559（伦敦升级中包含）之后， 实现了价格依据网络拥堵情况自动动态调节的机制，通过 DynamicFeeTxType 类型的交易就可以利用到这种动态调节机制。

在伦敦升级之后，存在一个基础费用 `baseFeePerGas` ，这是在每个区块上根据之前区块的交易拥堵情况而确定的一个值，每笔交易都需要消耗这个基础费用，这部分费用将被销毁。

在伦敦升级之后，针对LegacyTxType交易：

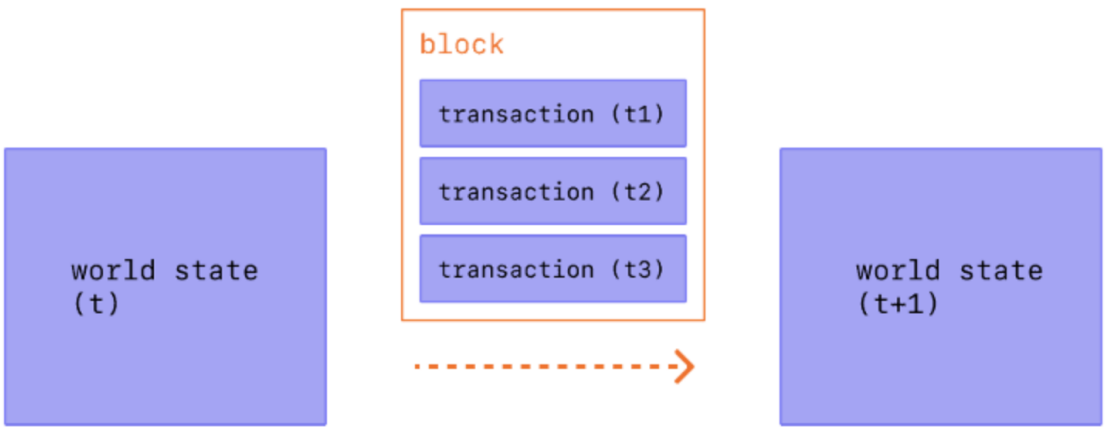
- 只有 gasPrice 一个字段指定用户愿意提供的价格，如果 gasPrice < baseFeePerGas ，交易将不会被处理。反之，用户消耗的费用 = gasUsed*gasPrice；矿工获得的小费 = gasUsed * (gasPrice-baseFeePerGas)；
- 可见，对于LegacyTxType交易，只要交易上链了，对用户而言交易费用就是固定的，无法享受到动态调节机制给自己带来的潜在好处。

而针对DynamicFeeTxType 交易：

- 可以通过 `maxFeePerGas` 设置用户愿意提供的最大价格，并通过 `maxPriorityFeePerGas` 设置愿意支付给矿工小费的最大价格。
- 在 `maxFeePerGas >= baseFeePerGas` ，交易被执行的情况下，用户支付给矿工的小费将为 `tip = gasUsed * MIN(maxFeePerGas-baseFeePerGas, maxPriorityFeePerGas)` ；而用户支付的交易费用 = tip + gasUsed * baseFeePerGas。
- 这种情况下，交易收据中的 effectiveGasPrice 就是交易费用的实际价格，`effectiveGasPrice = baseFeePerGas + MIN(maxFeePerGas - baseFeePerGas, maxPriorityFeePerGas)`
- 举个例子，假设用户在准备交易时，baseFeePerGas 是 50GWei，然后交易的 `maxFeePerGas` 设置为 60 GWei， `maxPriorityFeePerGas` 为 1 Gwei，则后续在 baseFeePerGas <= 60Gwei 时，用户的交易都有可能被矿工打包，并且在 baseFeePerGas <= 59 Gwei 时，用户实际支付的价格 `effectiveGasPrice` 都是 baseFeePerGas + 1 Gwei，从而在baseFeePerGas降低的情况下，用户实际支付的价格也会下降。

区块

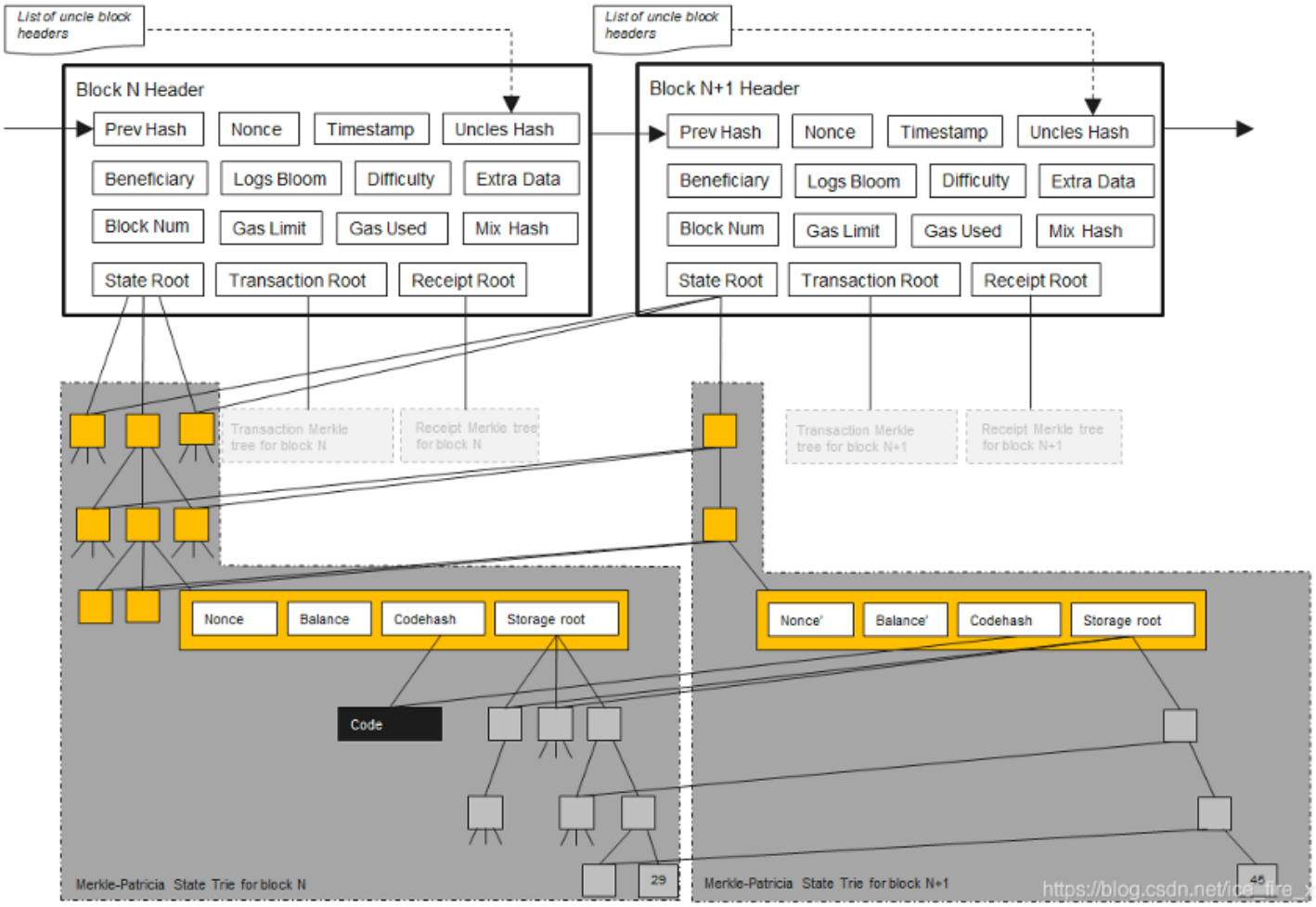
以太坊具有典型的区块链结构，以区块 `Block` 为单位组织状态转移记录也就是交易记录，并在前一个区块形成的状态的基础上，通过应用当前区块的交易完成状态转移，形成当前区块的状态。因此，每一个区块都对应着一个世界状态。



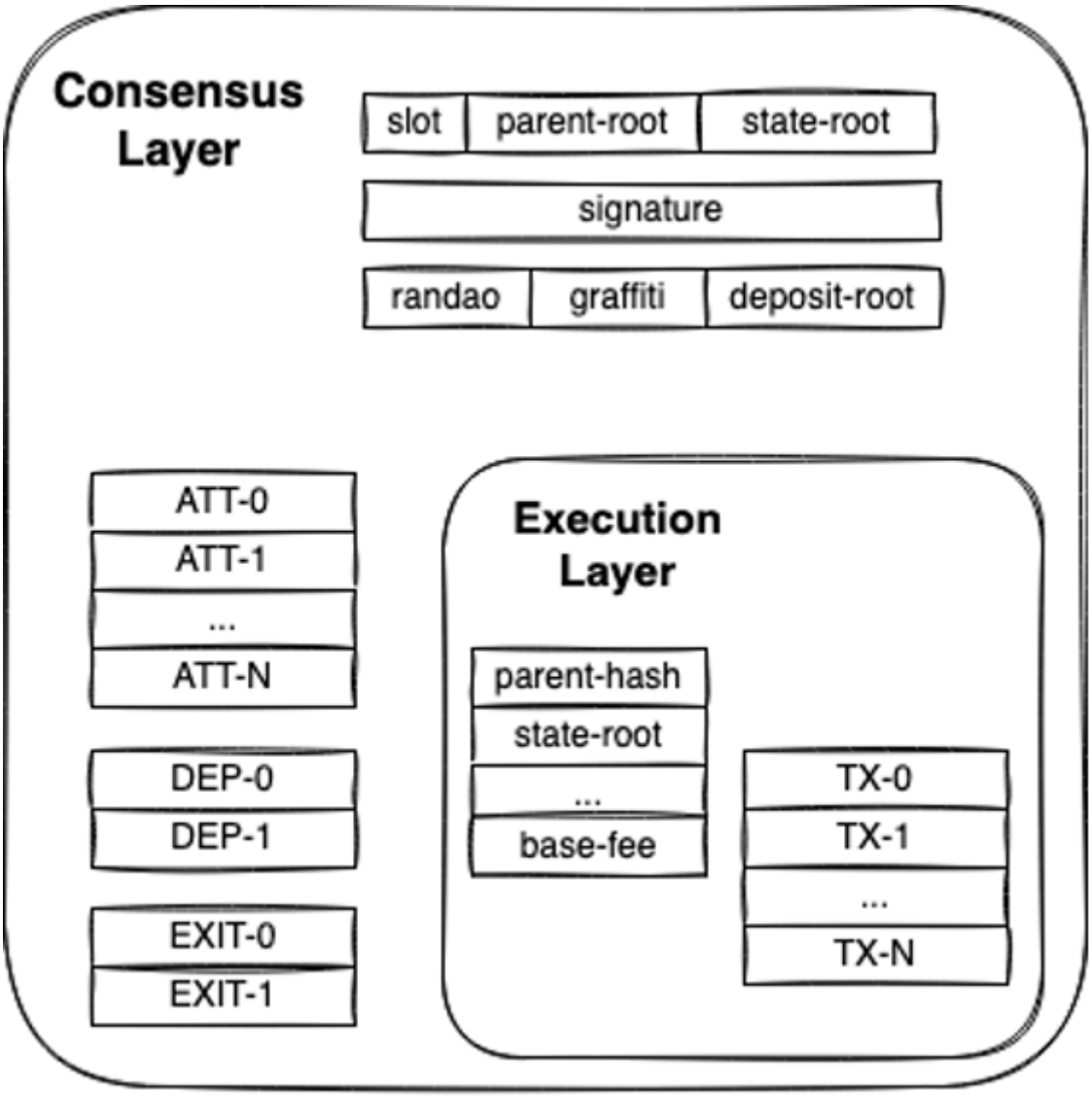
基于Block的状态转移示意图（源自 ethereum.org）

在以太坊转到PoS共识之前，还是比较典型的单链式结构。 `Block` 由区块头 `Header` 和 区块体 `BlockBody` 组成， `Header` 中包含了参与共识所需要的一切信息，而 Body 主要就是交易集合以及交易收据集合（见后文描述），而这些集合的默克尔树根 都记录在 Header中。

区块结构示意图如下：



The Merge之后，执行层链的区块结构有少量变化，但对用户影响不大。而共识层区块将包含执行层区块的完整数据，其结构示意如下：



以太坊PoS基本概念

> 参考：https://ethos.dev/beacon-chain
>
> 投票算法论文：Combining GHOST and Casper

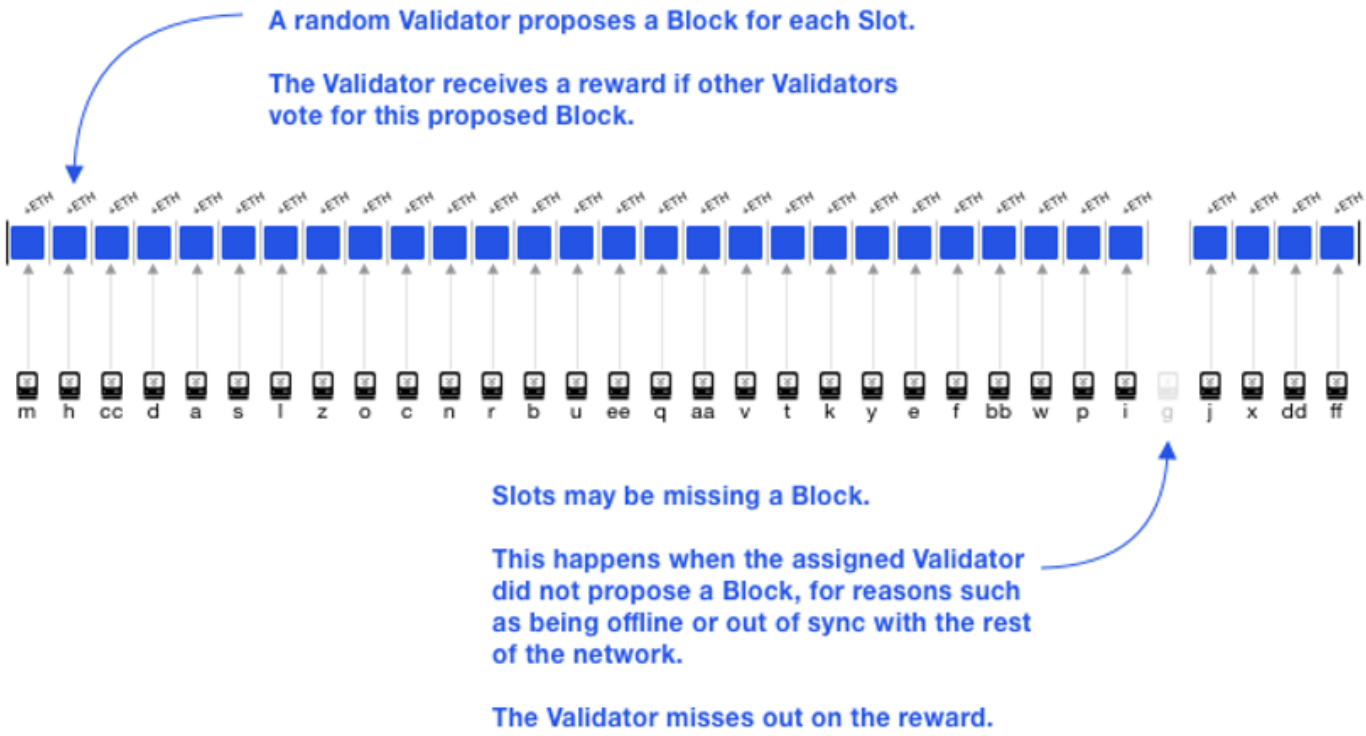
在PoS机制下，网络由抵押了 ETH 的validators 维护。

Slots and Epochs

以太坊共识层将时间分成一个个的 slot ，12秒为一个 slot，每32个slot 组成一个 epoch：

Validators and Attestations

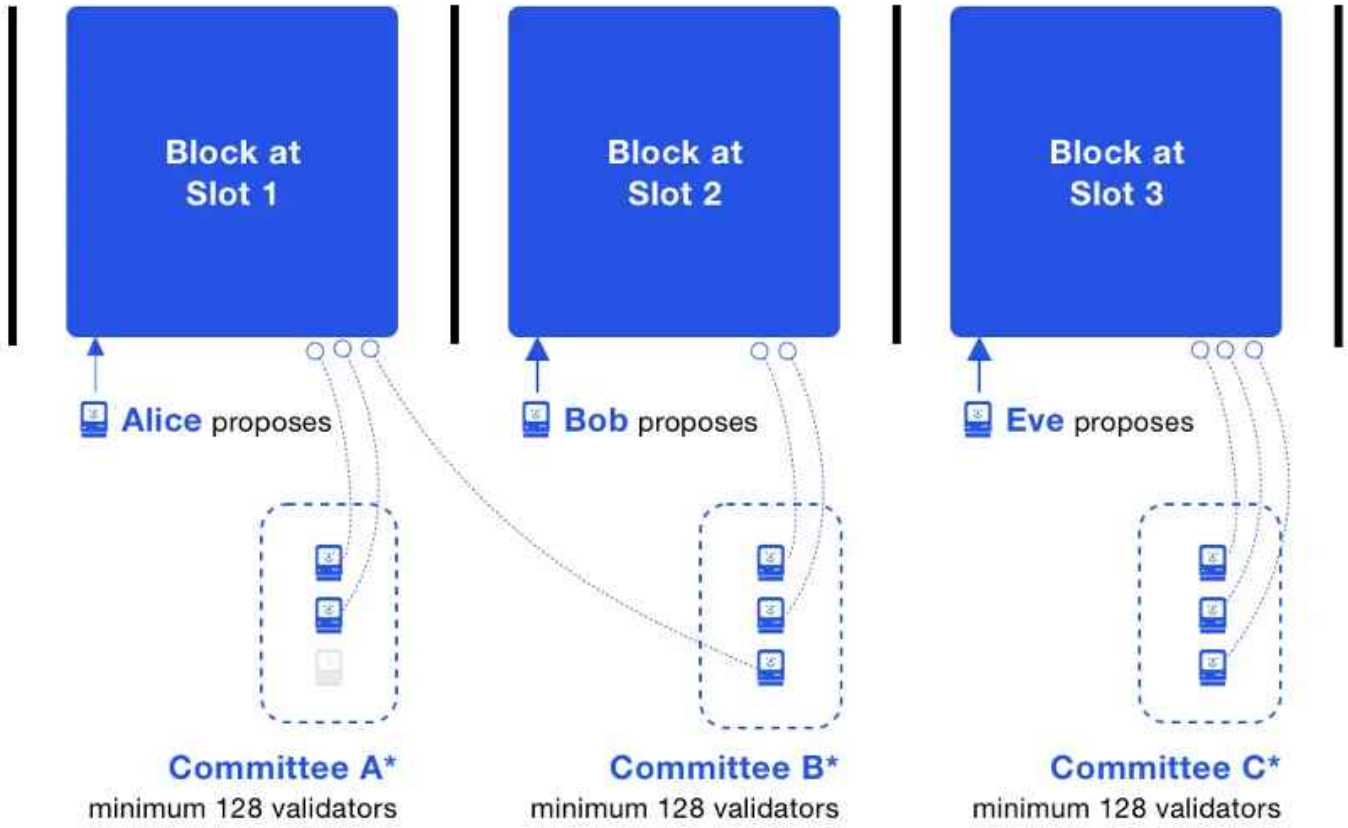
在每个slot，会有一个 block proposer 提议一个区块，然后其他validator会对区块进行（事后）验证及投票。



Committees

以太坊为每个 slot 随机选出至少 128个validator 组成的委员会，一个或多个委员会负责该slot的见证投票。采用 **LMD GHOST** 投票。

LMD GHOST：Last Message Driven Greediest Heaviest Observed SubTree，最新消息驱动的贪婪最重观察子树，是一个分叉选择规则，简而言之就是根据最新投票消息，选择具有最多投票（以投票者抵押的金额计）的分支。

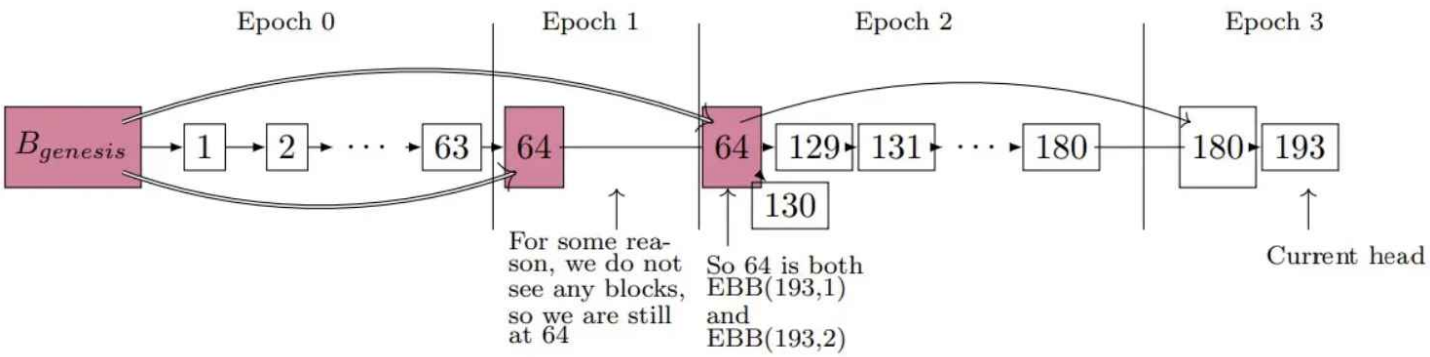


Validators in the committees are supposed to attest to what they believe the head of the blockchain is

*Note there can be more than one committee per slot.

Beacon Chain Checkpoints

每个epoch的第一个 slot，被称为检查点 checkpoint。网络中的所有活跃validator，都会按照一个叫 Casper FFG 的算法额外对检查点进行投票。（下图有一点过时，实际是 32个slot组成一个 epoch）。



Checkpoints for a scenario where epochs contain 64 slots.

Finality

在区块链里面，Finality是指交易上链之后达到一个（很可能是概率性的）最终确定即我们可以安全地认为交易不会被回滚的状态。

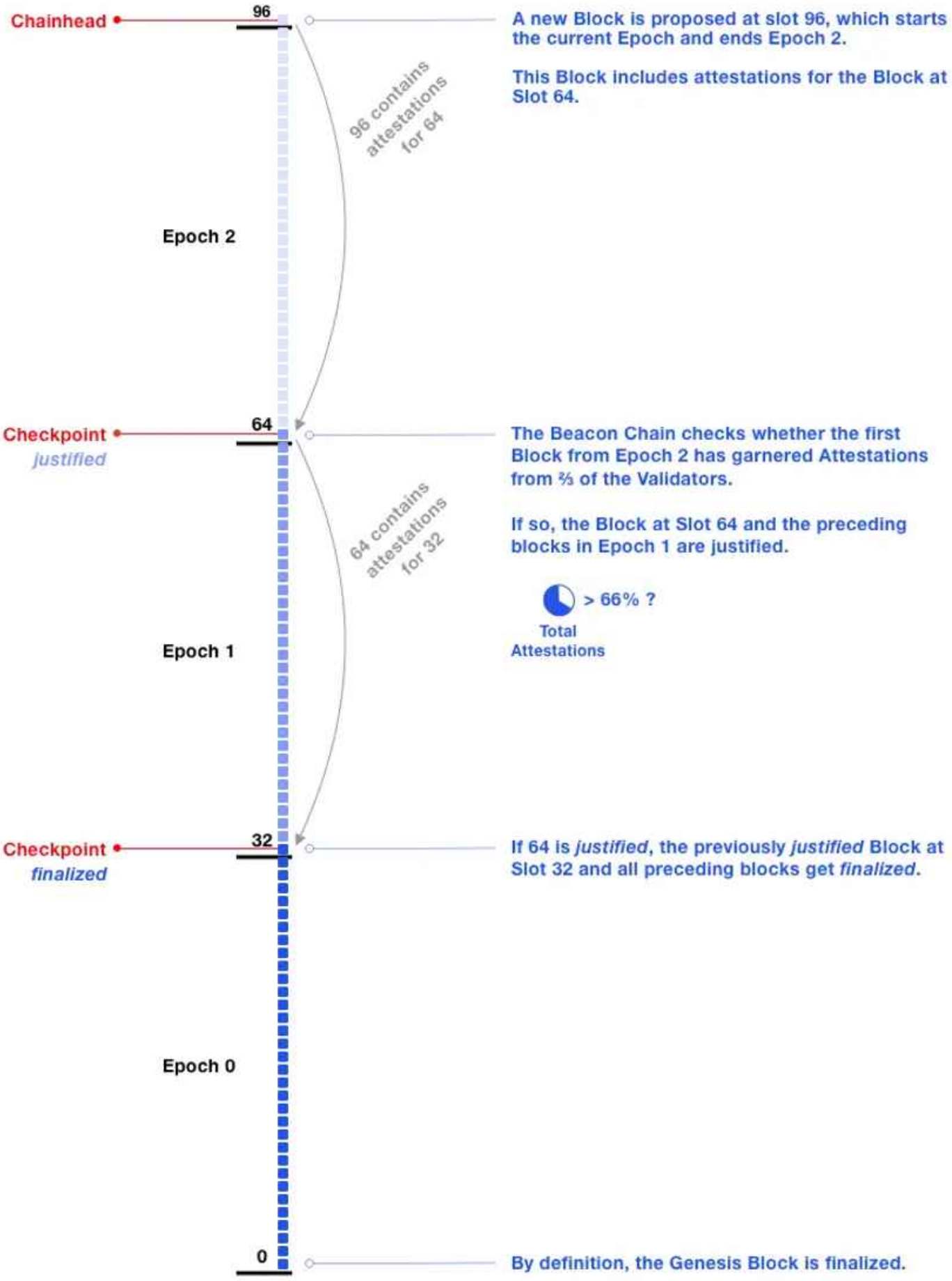
在PoW链中，一般是通过区块确认数来做概率性Finality确认，即交易所在的区块被后续（同一分支）的多个区块作为祖先，区块数量达到一定数量之后，我们可以认为基于该区块链网络算力的限制，交易所在区块不可能被回滚了，我们就认为它被最终确认了。

在eth1 时，区块确认数一般选择 12，约需 160秒。

> 交易回滚给用户带来损失的简单举例：
>
> A 给 B 转 1 ETH 以获取B的某个商品/服务；
>
> B看到A的交易上链了，查询自己的余额确实增加了 1 ETH，然后将商品/服务 提供给A；
>
> 之后区块被回滚，而同时A又用相同nonce发了一笔转账金额为 0 的交易以覆盖之前给B转账的交易，成功之后，A就在只支付了交易手续费的情况下，获得了B的商品/服务；
>
> 而B的余额又回到了之前的数额，白白遭受了损失。

在以太坊当前的PoS共识机制下，区块也不是即时确认的，因此也存在区块回滚的问题。当前的Finality机制是：

- 如果一个检查点获得2/3多数投票，它变成 justified ；
- 如果检查点B是 justified 状态，然后紧随其后的检查点也变成了 justified，此时B升级为 finalized 状态；此时就可以认为B不会被回滚了。
- 正常情况下，检查点会在 2个epoch之后 finalized，大概 12.8 分钟。假设一笔交易被包含在一个 epoch 中间的区块，则它最快需要 2.5 个epoch才能被 finalized，大概 16分钟。



根据上面分析，理论上应该等区块所在的epoch finalized之后，才应该将交易状态最终确认。但实际上，由于不大可能存在足够强大的“作恶”势力，当前很多交易所 都没有按以太坊PoS的Finality规则

进行交易的最终确认，而是几乎还沿用了原来的“区块确认数”概念，如 Binance 还是12区块确认，OKX则是 30区块确认（12区块就到账）。

以太坊客户端与节点

客户端多样化

以太坊是一个去中心化开放网络，生态比较繁荣，开发社区强大。以太坊在链本身的发展上，秉持着“社区共同定协议，鼓励客户端（即对协议的实现软件）多样化”的原则，从而使网络更加安全（一个基本的考虑是，某种客户端出现重大Bug时，也不至于造成整个网络的崩溃）。

执行层客户端

执行层客户端主要有以下几种：

	A	B	C	D	E	F
1	**Client**	**Language**	**Operating systems**	**Networks**	**Sync strategies**	**State pruning**
2	Geth	Go	Linux, Windows, macOS	Mainnet, Sepolia, Görli, Ropsten, Rinkeby	Snap, Full	Archive, Pruned
3	Erigon	Go	Linux, Windows, macOS	Mainnet, Sepolia, Görli, Rinkeby, Ropsten, and more	Full	Archive, Pruned
4	Nethermind	C#, .NET	Linux, Windows, macOS	Mainnet, Sepolia, Görli, Ropsten, Rinkeby, and more	Snap (without serving), Fast, Full	Archive, Pruned
5	Besu	Java	Linux, Windows, macOS	Mainnet, Sepolia, Görli, Ropsten, Rinkeby, and more	Fast, Full	Archive, Pruned
6	Akula	Rust	Linux	Mainnet, Sepolia, Görli, Rinkeby, Ropsten	Full	Archive, Pruned

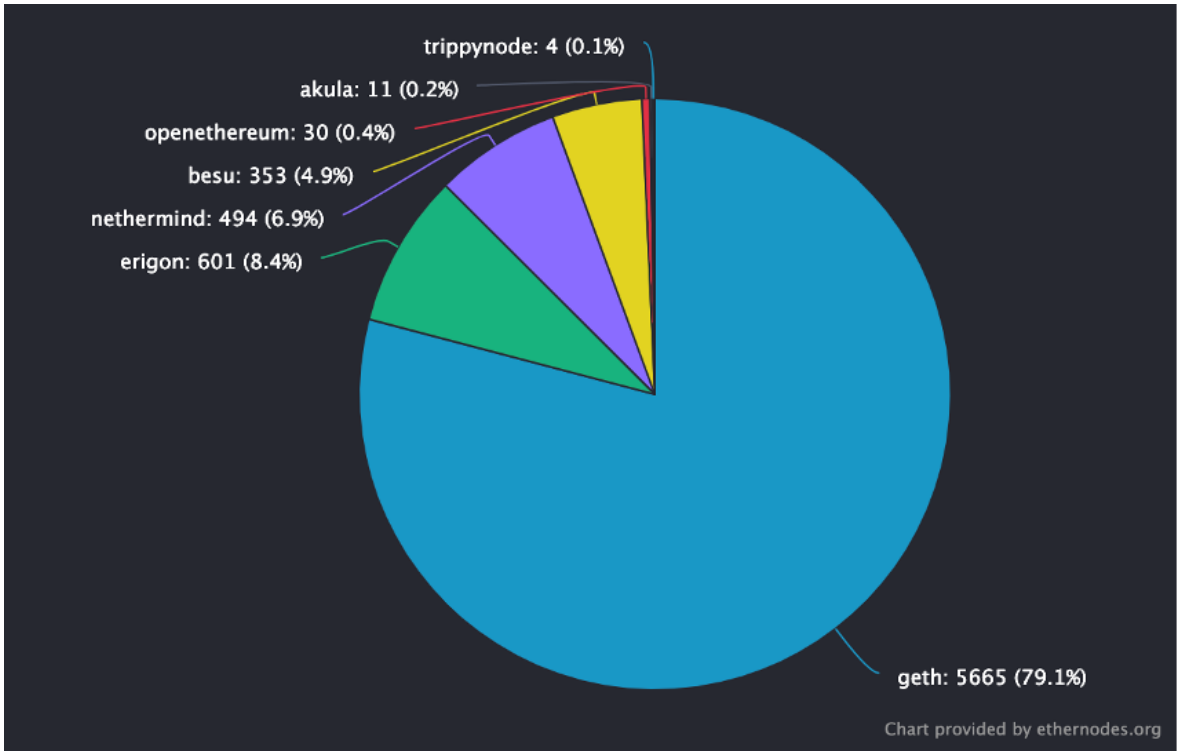
Go Ethereum

Go Ethereum（简称Geth）是Ethereum协议的原始实现之一。目前，它是最广泛的客户端，拥有最大的用户群，为用户和开发者提供各种工具。它是用Go语言编写的，完全开放源代码，并在GNU LGPL v3下许可。

Erigon

Erigon，以前被称为Turbo-Geth，开始是Go Ethereum的一个分叉，面向速度和磁盘空间效率。Erigon是一个完全重新架构的以太坊实现，目前用Go语言编写，但其他语言的实现也在开发中，例如

Akula。Erigon的目标是为Ethereum提供一个更快、更模块化、更优化的实现。它可以使用大约2TB的磁盘空间，在3天内完成一个完整的存档节点同步。



From: ethernodes.org

共识层客户端

在以太坊共识层，当前的客户端实现有以下几种：

	A	B	C	D
1	**Client**	**Language**	**Operating systems**	**Networks**
2	Lighthouse	Rust	Linux, Windows, macOS	Beacon Chain, Goerli, Pyrmont, Sepolia, Ropsten, and more
3	Lodestar	TypeScript	Linux, Windows, macOS	Beacon Chain, Goerli, Sepolia, Ropsten, and more
4	Nimbus	Nim	Linux, Windows, macOS	Beacon Chain, Goerli, Sepolia, Ropsten, and more
5	Prysm	Go	Linux, Windows, macOS	Beacon Chain, Gnosis, Goerli, Pyrmont, Sepolia, Ropsten, and more
6	Teku	Java	Linux, Windows, macOS	Beacon Chain, Gnosis, Goerli, Sepolia, Ropsten, and more

Lighthouse

Lighthouse是在Apache-2.0许可下用Rust编写的一个共识客户端实现。它由Sigma Prime维护，自Beacon Chain诞生以来一直很稳定，可以投入生产。它被各种企业、抵押池和个人所依赖。它的目标

是在广泛的环境中，从桌面电脑到复杂的自动化部署，都是安全的、高性能的和可互操作的。

Prysm

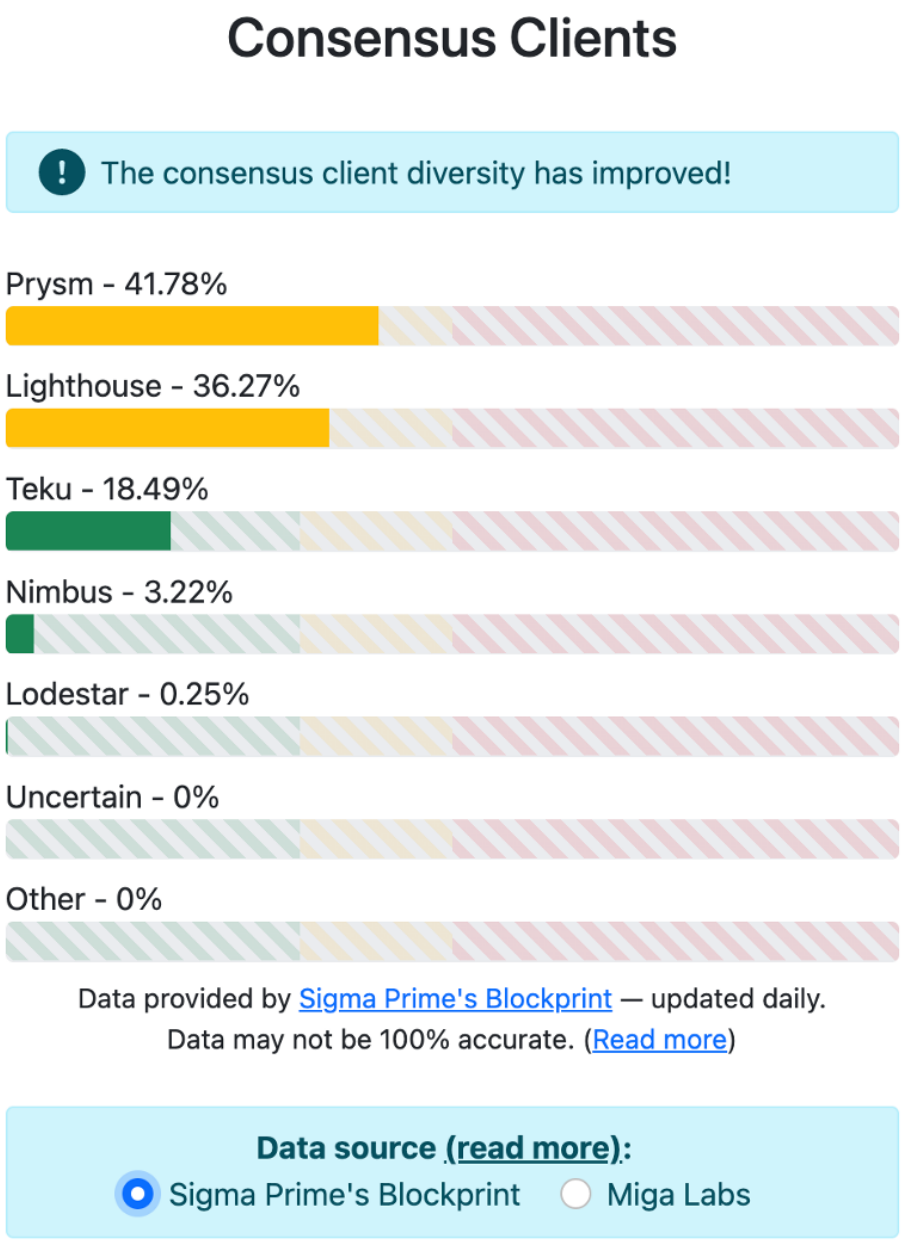
Prysm是一个在GPL-3.0许可下用Go语言编写的全功能、开源的共识客户端。它的特点是有一个可选的webapp用户界面，并优先考虑用户体验、文档和对个人抵押者和机构用户的可配置性。

Teku

Teku是最初的Beacon Chain创世客户端之一。除了通常的目标（安全性、稳健性、稳定性、可用性、性能）外，Teku特别旨在完全遵守所有各种共识的客户端标准。

Teku提供了非常灵活的部署选项。信标节点和验证器客户端可以作为一个单一进程一起运行，这对单独抵押者来说非常方便，或者节点可以单独运行，用于复杂的抵押操作。此外，Teku与Web3Signer完全互通，以实现签署密钥的安全性和惩罚保护。

Teku是用Java编写的，基于Apache 2.0许可。它是由ConsenSys的协议团队开发的，该团队还负责Besu和Web3Signer。



From: clientdiversity.org

节点类型

如果运行自己的以太坊节点，那就涉及到节点类型的考虑。以太坊节点类型（主要指执行层）包括：

- Full node 全节点：保存以太坊从创世区块开始以来所有的区块、交易、收据数据，并存储有最新若干区块（如最近128个区块）的完整世界状态。是最常见的节点类型，可以完成绝大部分场景需求。
- Light node 轻节点：只验证和保存所有的区块头数据。当查询链上状态时，需要从网络中的其他节点查询相应数据以及相应的默克尔证明，然后基于本地区块头上的状态树根 验证数据的正确性。
- Archive node 归档节点：拥有从创世区块以来的所有数据，包括每一个区块高度的世界态，需要大量的磁盘存储。archive节点可以查询历史上任何区块高度上的状态，可以跟踪任何区块高度上的交易（以便分析交易执行过程，debug）。

执行层节点区块同步类型

当一个新节点要加入网络时，涉及到首次数据同步，即将从创世区块以来的数据同步到本地，直至当前最新区块高度。这个同步有几种同步方式，跟节点的类型及完成初始同步的速度息息相关。

- Full sync 全同步：全同步是从网络其他节点同步所有的区块、交易数据，然后本地重放每一个区块并验证每一个区块。这是最安全的同步方式，也是最慢的同步方式。一般只有 archive节点，或对安全要求非常高（只相信本地节点的代码逻辑）的节点才会使用此种同步方式。
- Fast sync 快速同步：同步区块头数据，并依据共识规则验证区块头（会涉及到根据安全概率跳跃式的验证区块头），若区块头验证无误，则认为区块是合法的，然后单独同步区块体（交易、收据），对交易不会重放，而只是根据区块头中的交易哈希 和 收据哈希 进行验证。然后，在接近最新区块的时候，同步世界状态树，世界状态树同步完成后，后面将切换成 full sync模式。这种方式可以大大加速初始同步，但是安全性相对弱一些。在snap sync 实现之前，大部分的 全节点都会选择使用fast sync做初始同步。
- Snap sync 快照同步：这里的snap 指snapshot，而这里的snapshot是指以太坊大概在2021年成熟应用的一种扁平化状态存储结构，以这种结构缓存世界态，可以大大提高对状态的读速度。而snap sync就是基于这种snapshot结构，整体上类似于快速同步，只是在同步世界状态树的时候，基于snapshot结构进行同步，然后本地重建状态树。这种同步方式，要求网络中有大量的节点建好了snapshot结构。当前，snap sync 取代 fast sync称为大部分全节点的选择。
- Light sync 轻同步：用于轻节点，只会同步并验证区块头，不同步区块体（交易、收据），也没有最新的完整的世界状态树。

截至2022/11/28，geth 的archive节点数据存储量已超过 12TB；而快速或snap同步的新全节点数据量大概600+GB。

以太坊客户端实操

参考：https://github.com/0xcoolface/ethernode-operation

共识客户端 prysm 启动参数参考：https://docs.prylabs.network/docs/prysm-usage/parameters

本地可通过以下命令展示：

```
./prysm.sh beacon-chain -h
# 或找到具体的二进制文件执行
./dist/beacon-chain-v3.1.2-darwin-amd64 -h
```

执行客户端geth 启动参数参考：https://geth.ethereum.org/docs/interface/command-line-options

本地可通过以下命令展示：

```
./geth -h
```

Geth 常见子命令解读：

```
COMMANDS:
   account                  管理本地账户。如 `geth account new` 可以创建-
   attach                   连接到一个已运行的节点，进入JavaScript交互终端
   dumpconfig               显示配置参数
   init                     初始化创世区块，一般只在运行私链时有用
   snapshot                 基于snapshot的一些操作，比如对历史状态进行离线
   version                  打印版本信息
   version-check            在线检查当前版本是否涉及安全漏洞
   help, h                  帮助
```

Geth部分常见参数解读：

```
ETHEREUM OPTIONS:
  --config value                TOML 配置文件
  --networkid value             Explicitly set network id (integer)(For te
  --syncmode value              Blockchain sync mode ("snap", "full" or "l
  --gcmode value                Blockchain garbage collection mode ("full"
  --txlookuplimit value         Number of recent blocks to maintain transa
  --ethstats value              Reporting URL of a ethstats service (noden
  --identity value              Custom node name
```

```
  --mainnet                          Ethereum mainnet
  --ropsten                          Ropsten network: pre-configured proof-of-s
  --rinkeby                          Rinkeby network: pre-configured proof-of-a
  --goerli                           Görli network: pre-configured proof-of-aut
  --sepolia                          Sepolia network: pre-configured proof-of-w
  --kiln                             Kiln network: pre-configured proof-of-work
  --datadir value                    Data directory for the databases and keyst
  --datadir.ancient value            Data directory for ancient chain segments

  TRANSACTION POOL OPTIONS:
  --txpool.locals value              Comma separated accounts to treat as local
  --txpool.nolocals                  Disables price exemptions for locally subm
  --txpool.journal value             Disk journal for local transaction to surv
  --txpool.rejournal value           Time interval to regenerate the local tran
  --txpool.pricelimit value          Minimum gas price limit to enforce for acc
  --txpool.pricebump value           Price bump percentage to replace an alread
  --txpool.accountslots value        Minimum number of executable transaction s
  --txpool.globalslots value         Maximum number of executable transaction s
  --txpool.accountqueue value        Maximum number of non-executable transacti
  --txpool.globalqueue value         Maximum number of non-executable transacti
  --txpool.lifetime value            Maximum amount of time non-executable tran

PERFORMANCE TUNING OPTIONS:
  --cache value                      Megabytes of memory allocated to internal
  --cache.database value             Percentage of cache memory allowance to us
  --cache.trie value                 Percentage of cache memory allowance to us
  --cache.trie.journal value         Disk journal directory for trie cache to s
  --cache.trie.rejournal value       Time interval to regenerate the trie cache
  --cache.gc value                   Percentage of cache memory allowance to us
  --cache.snapshot value             Percentage of cache memory allowance to us
  --cache.noprefetch                 Disable heuristic state prefetch during bl
  --cache.preimages                  Enable recording the SHA3/keccak preimages
  --fdlimit value                    Raise the open file descriptor resource li

```

注意:

- geth 的控制台选项优先于配置文件，即如果配置文件中指定 `DataDir = "data1"`，然后命令行参数又指定了 `--datadir data2`，则最终数据目录是 `data2`。
- 而在 prysm 中刚好相反，在 prism中是配置文件中的值优先级更高!

Geth交互接口

接口传输协议

geth支持三种交互接口数据传输协议:

- http服务
- Websocket 服务
- 进程内通信管道 IPC

每一种服务可以单独配置是否启用及其使用的端口（文件）。

接口交互方式

跟geth（执行层客户端）交互，可认为有以下几种方式：

- JSON-RPC
- JavaScript Console 交互终端
- JavaScript API
- GraghQL

JSON-RPC

以太坊定义了 JSON-RPC 规范，通过json-rpc 协议提供用户跟节点的交互接口，所有的执行客户端都会实现这些接口。

geth相关的文档参考：https://geth.ethereum.org/docs/rpc/server

一个示例请求如下：

```
curl --location --request POST 'localhost:8545' \
--header 'Content-Type: application/json' \
--data-raw '{"jsonrpc": "2.0","method": "eth_blockNumber","params": [],"id": 1}'
```

示例返回：

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": "0xf5670f"
}
```

JavaScript console

在交互终端，以太坊默认提供了使用javascript对 json-rpc接口的封装，即 内置的web3.js ，用户可用其实现跟节点的交互。另外，用户也可以自己加载自己编写的javascript文件，从而使用自定义的功能。

参考：https://geth.ethereum.org/docs/interface/javascript-console

```
./geth attach datadir/geth.ipc

./geth attach --preload "demo.js" sepolia/geth.ipc

./geth attach --exec eth.blockNumber sepolia/geth.ipc

```

JavaScript API

做dapp开发，最常用的就是使用那些对 json-rpc接口进行了封装的 sdk。著名的如具有官方血统的 web3.js 以及方便易用的 ethers.js 。这部分主要参考 Solidity开发相关的培训和资料文档。

GraphQL

以太坊还定义了 GraphQL schema，geth 提供了 GraphQL 查询的接口支持。这部分需要时再具体研究了解。

链上交互核心流程

构造交易

估算gas

签名及广播交易

主动查询交易结果

订阅机制，被动实时接受通知 geth pubsub

进阶：交易失败问题排查

> 注意：geth在不断发展，文档不一定实时更新。

https://geth.ethereum.org/docs/rpc/ns-debug#debug_tracetransaction

https://geth.ethereum.org/docs/evm-tracing/builtin-tracers

```
 debug.traceTransaction('', {enableMemory: true,disableStack: false,disableStorage: false,enableReturnData: true});

 >debug.traceTransaction("0x39bb4548ea312465c449dbf12b40da719ff0a627570bdd165b3869101ad70d98", { tracer: "callTracer", tracerConfig: { onlyTopCall: false } })

 debug.traceTransaction("0x39bb4548ea312465c449dbf12b40da719ff0a627570bdd165b3869101ad70d98", { tracer: "callWithReasonTracer" })

curl --location --request POST 'localhost:8545' --header 'Content-Type: application/json' --data-raw '{"jsonrpc": "2.0","method": "debug_traceTransaction","pa
```

```
rams": [ "0x580970f9ba6a79b9255deedd383aaa875aebdf340fd67abcee6abe470d762c85",
{"tracer": "callWithReasonTracer"} ],"id": 83}'

```


```
{
  "jsonrpc": "2.0",
  "id": 83,
  "result": {
    "type": "CALL",
    "from": "0xa27d573683766f78a818f169c20e287149d26b09",
    "to": "0xeec20bf2a833b7c8c7df435e1d17dde87fa1a3fa",
    "value": "0x0",
    "gas": "0x38b2",
    "gasUsed": "0x38b2",
    "input": "0x7a7de4a900000000000000000000000000000000000000000000000000000000000000
    "output": "0x",
    "calls": [
      {
        "type": "CALL",
        "from": "0xeec20bf2a833b7c8c7df435e1d17dde87fa1a3fa",
        "to": "0x1726234cecdf30e83c8e9482400bb1e9e4bdfa41",
        "value": "0x0",
        "gas": "0x22c2",
        "gasUsed": "0x1b45",
        "input": "0x4686edcb00000000000000000000000000000000000000000000000000000000
        "output": "0x00000000000000000000000000000000000000000000000000000000000000
        "calls": [
          {
            "type": "CALL",
            "from": "0x1726234cecdf30e83c8e9482400bb1e9e4bdfa41",
            "to": "0x81f48ca0d4fab4611ae0fc3b850527ace587a3c9",
            "value": "0x0",
            "gas": "0xb65",
            "gasUsed": "0x2b6",
            "input": "0xde4c6b0a00000000000000000000000000000000000000000000000000
            "error": "execution reverted: num must be greater then 0"
          }
        ]
      }
    ]
  }
}
```

```solidity
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

// business
interface IBiz {
    function fnBiz(uint256) external returns (bool);
}

contract WithInnerError {
    address laddr;
    IBiz baddr;
    event Called(address indexed _to, bool _success);

    constructor() {
        laddr = address(new Logic());
        baddr = new Biz(laddr);
    }

    function callEntry(uint256 num) public {
        bool ok = baddr.fnBiz(num);
        emit Called(address(baddr),ok);
    }
}

contract Biz is IBiz {

    address public laddr;

    constructor(address _laddr) {
        laddr = _laddr;
    }

    function fnBiz(uint256 num) public override returns (bool) {
        bytes memory payload = abi.encodeWithSignature("myLogic(uint256)", num);
        (bool success, ) = laddr.call(payload);
        return success;
    }
}

contract Logic {
    event DoLogic(uint256 indexed _num);
    function myLogic(uint256 num) public {
        require(num > 0, "num must be greater then 0");
        emit DoLogic(num);
    }
}
```