目录

Geth

以太坊的 Geth github 仓库链接：https://github.com/ethereum/go-ethereum

要 go 语言基础，不会，暑假得学了

以太坊账户

从 UTXO 谈起

- 比特币是基于 UTXO 的结构中存储有关用户余额的数据：系统的整个状态就是一组 UTXO 的集合，每个 UTXO 都有一个所有者和一个面值（就像不同的硬币），而交易会花费若干个输入的 UTXO，并根据规则创建若干个新的 UTXO
- 每个引用的输入必须有效并且尚未花费：对于一个交易，必须包含有与每个输入的所有匹配的签名。总输入必须大于等于总输出值
- 所以，系统中用户的余额是用户具有私钥的 UTXO 的总值。
- 以太坊的总账本就是所有 UTXO 的集合
- UTXO 是什么？_LEVI_104 的博客-CSDN 博客

以太坊的做法

- 以太坊的“状态”，就是系统中所有账户的列表
- 每个账户都包含了一个余额，和以太坊特殊定义的数据（代码和内部存储）
- 如果发送账户有足够的余额来支付，则交易有效；在这种情况下发送账户先扣款，而收款账户将记入这笔收入
- 如果接收账户有相关代码，则代码会自动运行，并且它的内部存储也可能被更改，或者代码还可能向其他账户发送额外的信息，这就会导致进一步的借贷资金关系
- 以太坊的总账本就是所有账户的链表

比特币和以太坊账户的设计的优缺点比较

比特币 UTXO 模式的优点：

- 更高的隐私性：如果用户为他们收到的每笔交易使用新地址，那么通常很难将用户相互链接。这很大程度上实用于货币，但不设用于任意 DApps，因为 DApps 通常涉及跟踪和用户绑定的复杂状态，可能不存在像货币那样蒋丹的用户状态划分方案

- 潜在的可扩展性：UTXO 在理论上更符合可扩展性要求。因为我们只需要以来拥有 UTXO 的那些人去维护基于 Merkle 树的所有权证明就够了，即使包括所有者在内的每个人都决定忘记该数据，那么也只有所有者收到对应的 UTXO 的损失，不影响接下来的交易。而在账户模式中，如果每个人丢失了于账户对应的 Merkle 树的部分，那将会使得和该账户有关的信息完全无法处理，包括发币给它

以太坊账户模式的优点：

- 可以节省大量空间：不将 UTXOs 分开存储，而是为一个账户：每个交易只需要一个输入、一个签名并产生一个输出
- 更好的可替代性：货币本质上都是同质化、可替代的；UTXO 的涉及使得货币从来源分成了“可花费”和“不可花费”两类，这在实际应用很难有对应的模型
- 更加简单：更容易编码和理解，特别是复杂脚本的时候。UTXO 在脚本逻辑复杂时更令人费解
- 便于维护持久轻节点：只要沿着特定方向扫描状态树，轻节点可以很容易地随时访问账户相关的所有数据。而 UTXO 的每个交易都会使得状态引用发生改变，这对轻节点来说长时间运行 DApp 会有很大压力

比特币和以太坊的对比

	BitCoin	Ethereum
设计定位	现金系统	去中心化应用平台
数据组成	交易列表 （账本）	交易和账户状态
交易对象	UTXO	Accounts
代码控制	脚本	智能合约

以太坊账户类型

外部账户（用户账户/普通账户）（Externally owned account, EOA)。

合约账户（内部账户）（Contract accounts)

EOA

- 有对应的以太币余额
- 可发送交易（转币或者出发合约代码）
- 由用户私钥控制
- 没有关联代码
- 以太坊所有的交易的发起者都是 EOA

合约账户

- 有对应的以太币余额
- 有关联代码
- 可由代码控制
- 可通过交易或来自其他合约的调用信息来触发代码执行
- 执行代码时可以操作自己的存储空间，也可以调用其他合约

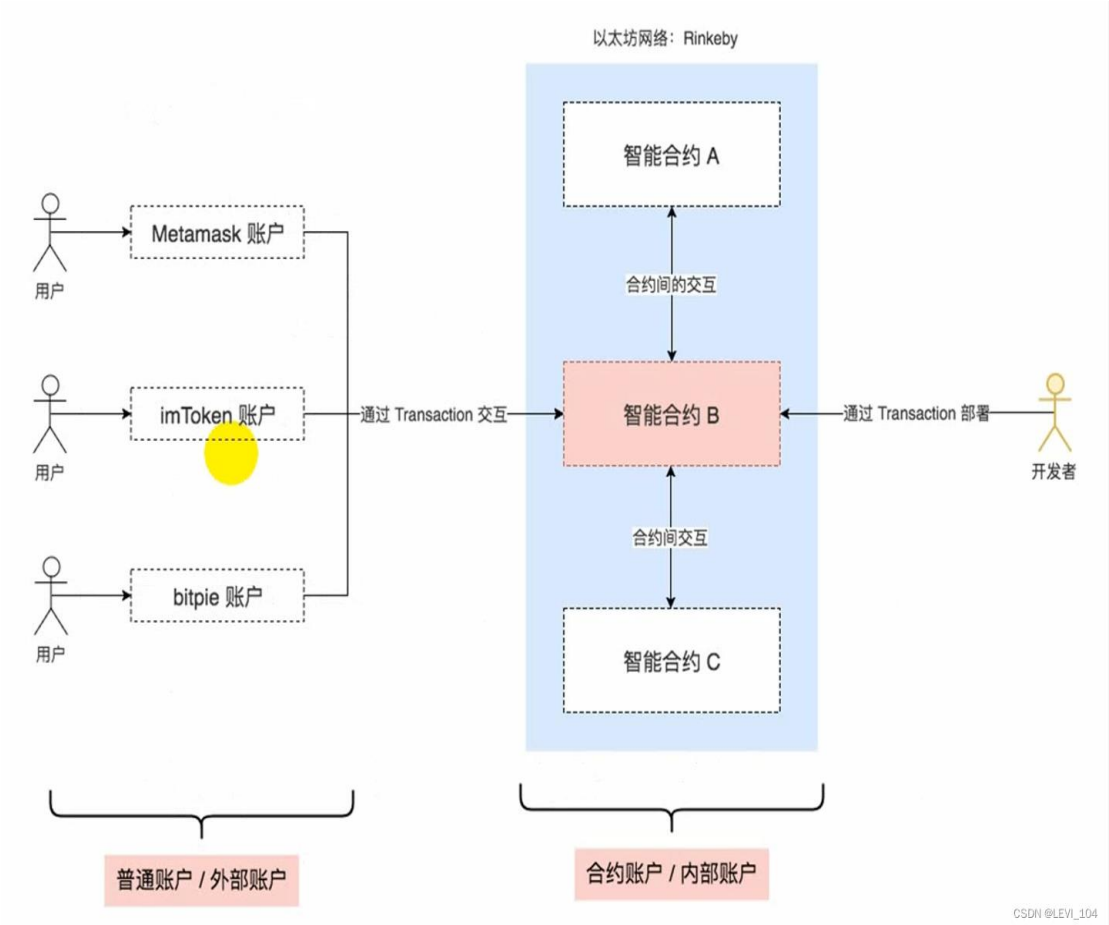

图 1-1：来自尚硅谷区块链全套教程视频

以太坊交易（Transaction）

签名的数据包，由 EOA 发送到另一个账户（EOA 或者合约）

- 信息的接收方地址
- 发送方签名（包含了发送方的地址）
- 金额（VALUE）
- 数据（DATA，可选）

- START GAS（交易给定的上限）
- GAS PRICE（GAS 单价）

消息（Message）

合约可以向其它合约发送“消息”。消息是不会被序列化的虚拟对象，只存在于以太坊的执行环境（EVM）中。可以看作函数调用。

- 消息发送方
- 消息接收方
- 金额（VALUE）
- 数据（DATA，可选）
- START GAS

合约（代码）（Contract）

- 可以读/写自己的内部存储（32 字节 k e y －value 的数据库）
- 可向其他合约发送消息，依次触发执行
- 一旦合约运行结束，并由它发送的消息触发的所有子程序（ s u b － execution）结束，EVM 就会终止运行，直到下一次交易被唤醒

以太坊交易详解（Ｅｔｈｅｒｅｕｍ Ｔｒａｎｓａｃｔｉｏｎｓ）

交易的本质

- 交易是由外部拥有的账户发起的签名信息，由以太坊网络传输，并被序列化后永久记录在以太坊区块链上
- 以太坊是一个状态机，交易是唯一可以触发状态更改或合约在 EVM 中执行的事物
- 以太坊是一个全局单例状态机（单例：只有一种状态），交易是唯一可以改变其状态的东西
- 合约不是自己运行的，以太坊也不会“在后台”运行。以太坊上的一切变化都始于交易

交易数据结构

交易是包含以下数据的序列化二进制消息

- nonce：由发起人 EOA 发出的序列号，用于防止交易信息重播（重放攻击），保证信息唯一性
    - 比如：前面已经有 8 笔交易（0-7），然后第 9 笔交易广播（nonce＝8），矿工验证合法，就打包进块了。但是如果广播的ｎｏｎｃｅ＝6，和前面重复，那么就不会验证通过。如果广播的ｎｏｎｃｅ＝16，那么也不会打包进块，矿工会把这个交易放进缓冲池里面，未来可能会打包进块。所以，ｎｏｎｃｅ是依次增加的。
- ｇａｓ　ｐｒｉｃｅ：交易发起人愿意支付的ｇａｓ单价（ｗｅｉ）
- ｓｔａｒｔ　ｇａｓ：交易发起人愿意支付的最大ｇａｓ量
- ｔｏ：目的以太坊地址
- ｖａｌｕｅ：要发送到目的地的以太数量
- ｄａｔａ：可变二长度二进制数据负载（ｐａｙｌｏａｄ），可以理解我们要传播的数据
- ｖ，ｒ，ｓ：发起人 EOA 的 ECDSA 签名的三个组成部分
- 交易信息的数据结构使用递归长度前缀（RLP）编码方案进行序列化，该方案专为在以太坊中准确和字节完美的数据序列化而创建

交易中的 nonce

- 黄皮书定义：nonce 是一个标量值，等于从这个地址发送的交易数，或者对于关联ｃｏｄｅ的账户来说，是这个账户创建合约的数量
- ｎｏｎｃｅ不会明确存储为区块链中账户状态的一部分。相反，它是通过计算发送地址的已确认交易的数量来动态计算的。当需要确认交易的时候，nonce 用来确认，一定要顺序执行
- nonce 还用于防止错误计算账户余额。nonce 强制来自任何地址的交易按顺序处理，没有间隔，无论节点接收它们的顺序如何
- 使用 nonce 确保所有节点计算相同的余额和正确的序列交易，等同于用于防止比特币“双重支付”（“重放攻击”）的机制。但是，由于以太坊跟踪账户余额并且不单独跟踪 UTXO，因此只有在错误地计算账户余额时才会发生“双重支付”。nonce 机制可以防止这种情况发生。

并发和 nonce

- 以太坊是一个允许操作（节点，客户端，ＤＡｐｐｓ）并发的系统，但强制执行单例状态。例如：出块的时候只有一个系统状态
- 假如我们有多个独立的钱包应用或客户端，比如 MetaMask 和Ｇｅｔｈ，它们可以使用相同的地址生成交易。如果我们希望它们都能够同时发送交易，该怎么设置交易的 nonce 呢？
    - 用一台服务器为各个应用分配 nonce，先来先服务——可能出现单点故障，并且失败的交易会将后序交易阻塞。

- 生成交易后不分配 nonce，也不签名，而是把它放入一个队列等待。另起一个节点跟踪 nonce 并签名交易。同样会有单点故障的可能，而且跟踪 nonce 和签名的节点是无法实现真正并发的。

单点故障（英语：single point of failure，缩写 SPOF）是指系统中一点失效，就会让整个系统无法运作的部件，换句话说，单点故障即会整体故障。

高可用性或者高可靠度的系统（商务系统、软件系统或工业系统）不会希望有单点故障造成整体故障的情形。一般可以透过冗余的方式增加多个相同机能的部件，只要这些部件没有同时失效，系统（或至少部分系统）仍可运作，这会让可靠度提高，不过也增加成本和某些设计难度。

交易中的ｇａｓ

- 当由于交易或消息触发 EVM 运行时，每个指令都会在网络的每个节点上执行。这具有成本：对于每个执行的操作，都存在固定的成本，我们把这个成本用一定的ｇａｓ表示。
- ｇａｓ是交易发起人需要为 EVM 上的每项操作支付的成本名称。发起交易时，我们需要从执行代码的矿工那里用以太币购买ｇａｓ。
- 为什么要有ｇａｓ这个概念：ｇａｓ于消耗的系统资源对应，这是具有自然成本的。因此在设计上ｇａｓ和ｅｔｈｅｒ有意地解耦，消耗的ｇａｓ数量代表了对资源的占用，而对应的交易费用则还跟ｇａｓ对以太的单价有关。这两者是由自由市场调节的：ｇａｓ的价格实际上是由矿工决定的，他们可以拒绝处理ｇａｓ价格低于最低限额的交易。我们不需要专门购买ｇａｓ，只需将以太币添加到账户即可，客户端在发送交易时会自动用以太币购买ｇａｓ。而以太币本身的价格通常由时长力量而波动。ｇａｓ与消耗的自然资源（电力，算力等）对应，不想被市场的波动而影响

ｇａｓ的计算

- 发起交易时的ｇａｓ　ｌｉｍｉｔ并不是要支付的ｇａｓ数量，而只是给定了一个ｇａｓ的上限，相当于“押金”
- 实际支付的ｇａｓ数量时执行过程中消耗的ｇａｓ（ｇａｓＵｓｅｄ），ｇａｓ　ｌｉｍｉｔ中剩余的部分会返回给发送人
- 最终支付的ｇａｓ费用是ｇａｓUsed对应的以太币费用，单价由设定的ｇａｓPrice而定
- 最终支付费用 totalCost＝ｇａｓPrice　＊　ｇａｓUsed
- totalCost 会作为交易手续费（Ｔｘ　fee）支付给矿工

交易的接收者（ｔｏ）

- 交易接收者在 t o 字段中指定，是一个 20 字节的以太坊地址。地址可以是 EOA 或合约地址。
- 以太坊没有进一步的验证，任何 20 字节的值都被认为是有效的。如果 20 字节值对应于没有相应的私钥的地址，或不存在的合约，则该交易仍然有效。以太坊无法知道地址是否从公钥正确派生的
- 如果将交易发送到无效地址，将销毁发送的以太，使其永远无法访问
- 验证接收人地址是否有效的工作，应该在用户界面一层完成

交易的 value 和 data

- 交易的主要“有效负载”包含在两个字段中：value 和 data。交易可以同时有 value 和 data，仅有 data，或者既没有 value 也没有 data。所有四种组合都有效
- 仅有 value 的交易就是一笔以太的付款
- 仅有 data 的交易一般是合约调用
- 进行合约调用的同时，我们除了传输 data，还可以发送以太，从而交易中同时包含 data 和 value
- 没有 value 也没有 data 的交易，只是在浪费 g a s，但它是有效的

向 EOA 或合约传递 data

- 当交易包含数据有效负载时，它很可能时发送到合约地址的，但它同样可以发送给 EOA
- 如果发送 data 给 EOA，数据负载（data payload）的解释取决于钱包
- 如果发送数据负载给合约地址，EVM 会解释为函数调用，从 payload 里解码出函数名称和参数，调用该函数并传入参数
- 发送给合约的数据有效负载时 32 字节的十六进制序列化编码：
    - 函数选择器：函数原型 Keccak256 哈希的前 4 个字节。这允许 EVM 明确地识别将要调用的函数
    - 函数参数：根据 EVM 定义的各种基本类型的规则进行编码

特殊交易：创建（部署）合约

- 有一种特殊的交易，具有数据负载且没有 value，那就是一个创建新合约的交易
- 合约创建交易被发送到特殊目的地地址，即零地址 0 x 0。该地址既不代表 EOA 也不代表合约。它永远不会花费以太或发起交易，它仅用作目的地，具有特殊含义“创建合约”
- 虽然零地址仅用于合约注册，但它有时会收到来自各种地址的付款。这种情况要么是偶然误操作，导致失去以太；要么是故意销毁以太
- 合约注册交易不应该包含以太值，只包含合约的已编译字节码的数据有效负载。此交易的唯一效果是注册合约