区块链的数据结构和数据库

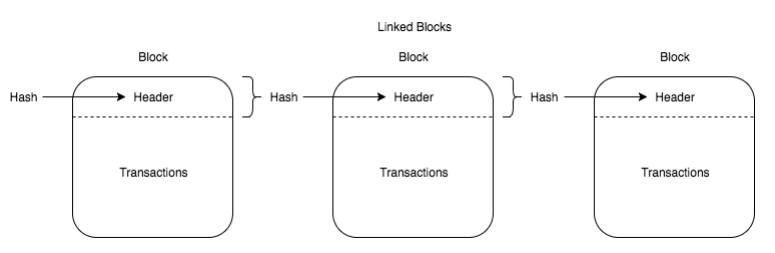
区块链的数据结构和数据库

- 区块链的链式数据结构
- 比特币的数据结构和数据库
- 以太坊的数据结构和数据库

2

区块链的链式数据结构

- 区块链数据结构是一种有序的、后向连接的交易区块列表。
- 区块是“后向”连接的，每个区块都有链接指向链条上的前序区块。
- 区块链通常可想象为一个垂直堆栈，新的区块堆叠在其他区块的顶部，第一个区块是堆栈的基础。
- 区块堆叠在其他区块之上的形象比喻导致了一些名词的引入，比如，“高度（height）”指本区块到第一个区块的距离，“顶部（top）”或“顶端（tip）”指最新加入的区块。



3

区块链的链式数据结构

- 区块链中的每个区块，在其头部使用一个通过哈希算法生成的哈希值进行标识。
- 每个区块头还包含一个“前序区块哈希”的字段，对前序区块（父区块）进行引用。即每个区块在区块头中均存有父区块的哈希。
- 将每个区块连接到其父区块的哈希序列形成了一条可以一直追溯到第一个区块（创世区块）的链条。

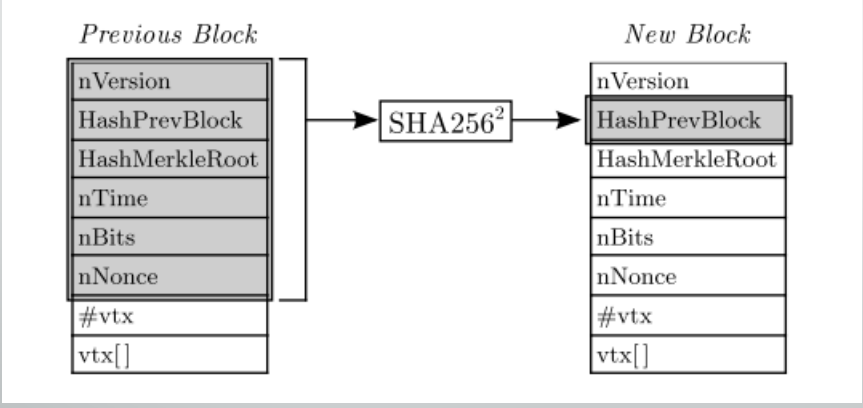
4

区块链的数据结构和数据库

- 区块链的链式数据结构
- 比特币的数据结构和数据库
- 以太坊的数据结构和数据库

5

比特币相邻区块通过区块头的hash进行关联



6

创世区块

- 区块链中的第一个区块叫作创世区块。它是区块链中所有区块的共同祖先，也就是说，如果从任何一个区块开始，沿着区块链回溯，最终都会到达创世区块。
- 每个节点启动时，其区块链中至少包含一个区块，因为创世区块是被静态编码到比特币客户端软件中的，无法被修改。每个节点都“知道”创世区块的哈希和结构、其创建时间，以及它包含的唯一一个交易。这样每个区块都拥有了区块链的起始点，一个安全的“根”，从它开始就可以构建一条可信任的区块链。。

7

比特币的区块结构

大小（字节）	字段名称	数据类型	描述
4	magic_number	uint32	总是oxD9B4BEF9,作为区块之间的分隔符
4	区块大小 block_size	uint32	后面数据到块结束的字节数
80	区块头 block_header	char[]	block header
1-9字节	交易计数器 transaction_cnt	uint	交易数量
可变长度	交易transaction	char[]	交易详情

从原始数据中读取的大概流程：

1. 读取4个字节，比对magic_number
2. 一旦匹配，读取后4个字节，得到块的大小m
3. 读取后面m个字节，得到区块的数据
4. 返回第一步，读取下一个区块

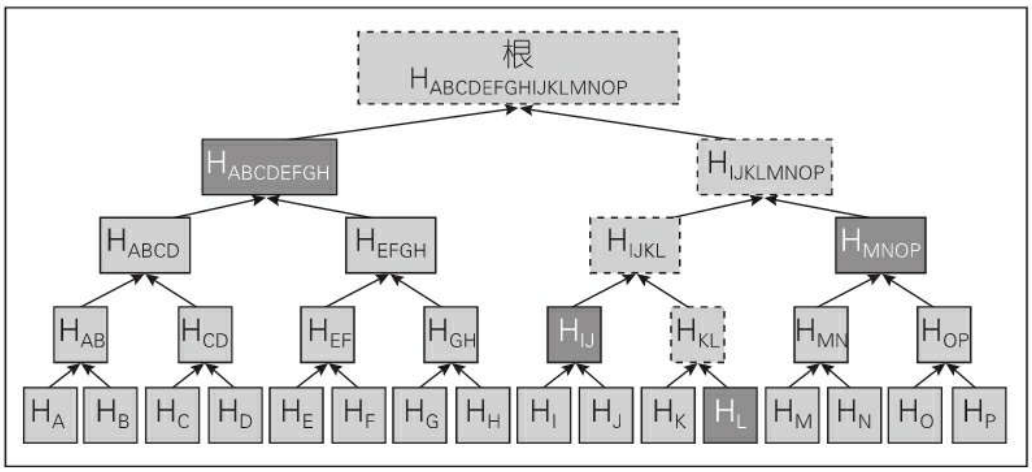
8

比特币的区块头结构

大小	字段	描述
4 字节	版本（Version）	跟踪软件/协议更新的版本号
32 字节	前序区块哈希（Previous Block Hash）	对链中前序（父）区块哈希值的引用
32 字节	默克尔根（Merkle Root）	本区块所有交易的默克尔根的哈希
4 字节	时间戳（Timestamp）	本区块大致的创建时间（Unix 时间戳）
4 字节	难度目标（Difficulty Target）	本区块工作量证明算法的难度目标
4 字节	随机数（Nonce）	用于工作量证明算法的一个计数器

9

区块链中的Merkle树



10

比特币的区块头结构示例

```
02000000 ........................... Block version: 2

b6ff0b1b1680a2862a30ca44d346d9e8
910d334beb48ca0c0000000000000000 ... Hash of previous block's header
9d10aa52ee949386ca9385695f04ede2
70dda20810decd12bc9b048aaab31471 ... Merkle root

24d95a54 ........................... Unix time: 1415239972
30c31b18 ........................... Target: 0x1bc330 * 256**(0x18-3)
fe9f0864 ........................... Nonce
```

11

比特币的交易结构

大小(字节)	字段名称	数据类型	描述
4	version	uint32	交易版本号
varint	tx_in_count	uint	交易输入数量
varies	tx_in	tx_in	交易输入
varint	tx_out_count	uint	交易输出数量
varies	tx_out	tx_out	交易输出
4	lock_time	uint32	锁定时间

从数据中解析流程：

1. 读取4个字节版本号
2. 解析varint，得到输入数量n
3. 执行1～n次循环，解析交易输入
4. 解析varint，得到输出数量m
5. 执行1～m次循环，解析交易输出

12

比特币的交易输入结构

大小	字段	描述
32 字节	交易哈希（Transaction Hash）	指向待花费 UTXO 的指针
4 字节	输出索引（Output Index）	UTXO 的编号，从 0 开始
1 ~9 字节（VarInt）	解锁脚本大小（Unlocking-Script Size）	紧跟的解锁脚本的长度
可变长度	解锁脚本（Unlocking-Script）	满足 UTXO 锁定脚本条件的解锁脚本
4 字节	序号（Sequence Number）	目前未被使用的交易替换功能，设置为 0xFFFFFFFF

一旦UTXO选定后，钱包应用便开始创建包含每个UTXO签名的解锁脚本，使它们满足锁定脚本的条件，从而可以花费。钱包应用加入这些UTXO的引用和解锁脚本作为交易输入。

13

比特币的交易输出结构

大小	字段	描述
8 字节	数量	以聪为单位的比特币价值
1 ~9 字节（VarInt）	锁定脚本大小（Locking-Script Size）	用字节表示的后续锁定脚本长度
可变长度	锁定脚本（Locking-Script）	定义花费输出所需条件的脚本

交易输出包含两部分：

- 比特币金额，“聪”的任意倍数，“聪”是比特币的最小单位。
- 锁定脚本，也被称为“受限”，通过指定花费输出必须符合某种条件，将这个金额锁定。

14

比特币的交易输出限定【锁定脚本】

- 交易输出将特定金额（单位为“聪”）与特定的受限或者锁定脚本相关联，明确了花费这个金额必须满足的条件。
- 在大多数情况下，锁定脚本将输出锁定到一个特定的比特币地址上，从而将这笔资金的所有权转移给新的所有者。
- 当爱丽丝向鲍勃咖啡店支付一杯咖啡的比特币时，她的交易创建了一个0.015比特币的输出，锁定到咖啡店的比特币地址上。这个0.015比特币的输出记录在区块链上，成为UTXO集合的一部分，也意味着在鲍勃的钱包上，这笔输出已成为可使用余额的一部分。当鲍勃选择花费这笔余额时，他的交易将解开这个受限，通过**提供包含私钥签名的解锁脚本**对输出进行解锁。

15

比特币的UTXO数据模型

- 比特币中没有账户或余额，
- 只有散布在区块链中的UTXO

16

比特币的交易数据关联

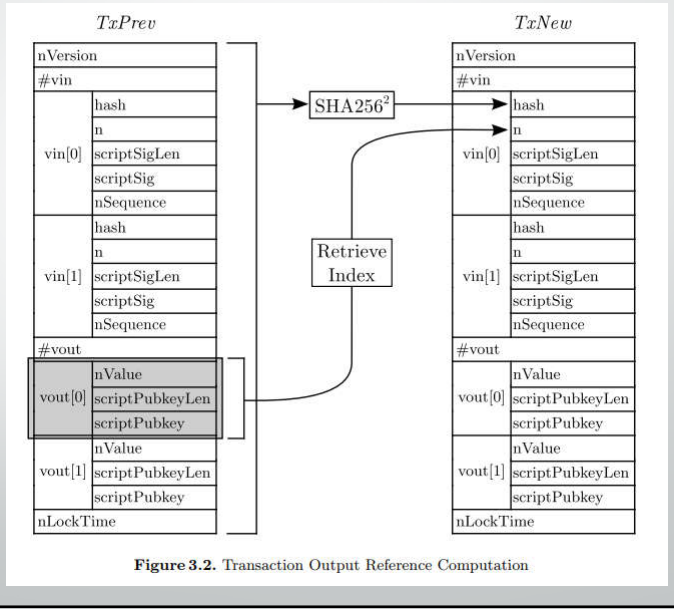
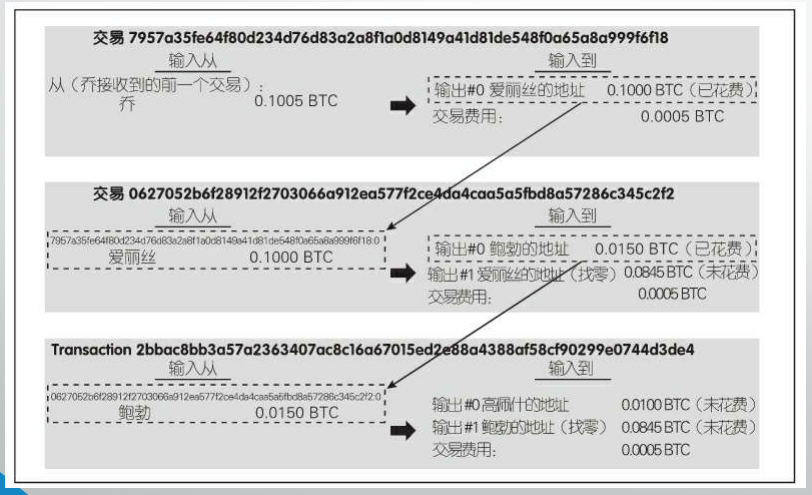


Figure 3.2. Transaction Output Reference Computation

比特币的UTXO数据模型—交易链条



18

比特币的UTXO数据模型

- 每个比特币交易都产生输出，输出将被记录在比特币账本上。除一种情况外［“数据输出（OP_RETURN）”］，几乎所有这些输出都创建可使用的比特币，被称为UTXO，这些UTXO会被全网识别，并可被新的所有者在将来的交易中花费。向某人发送比特币就是创建一个UTXO并注册到他的地址上，随后他就可以花费这笔UTXO。
- UTXO将被所有完全客户端，通过其维护在内存中的数据库的方式进行跟踪，这个数据库叫作UTXO集合或者UTXO池。新交易将从UTXO集合中消费（花费）一个或多个输出。

19

比特币的UTXO数据模型

- 比特币交易的基本结构单元是未花费输出，或者被称为UTXO，UTXO是一个不可拆分的比特币结构，锁定一个特定的所有者，记录在区块链上，并被全网看作一个货币单元。比特币网络跟踪数以百万计的所有有效（未花费）UTXO。
- 当一个用户接收到比特币，金额就以UTXO的形式记录在区块链上。
- 一个用户的比特币资金可能会以UTXO的形式分散存放在数百个交易和区块上。
- 没有任何东西会去记录一个比特币地址或者账户的余额，只有分散的UTXO，锁定到特定的所有者。用户比特币账户余额的概念是钱包应用软件从传统应用中继承而来的。
- 钱包软件通过扫描区块链，收集所有属于这个用户的UTXO，以此来统计用户的余额。

20

比特币的UTXO数据模型

- 一个UTXO可以由任意倍的“聪”构成。比特币能够分割到小数点后8位，被称作“聪”。
- 虽然UTXO可以是任意金额，但是一旦创建，它就是不可分割的，就如同一枚硬币不能剖成两半一样。
- 如果UTXO比交易所需要的金额大，它也需要一次性花完，超出的部分通过在交易中找零被索回。
- 换句话说，如果你有20比特币的UTXO，需要支付1比特币，交易首先要将20比特币的UTXO全部花完，那么就需要创建两个输出：一个是支付1比特币给指定的接收人，另外一个则将19比特币返回到你的钱包。结果是，绝大部分交易都需要创建找零输出。

21

比特币的UTXO数据模型

- 比特币交易不管多大金额，都需要从用户的UTXO中进行创建。
- 用户无法将UTXO拆成两半，就像不能把一张纸票撕成两半来用一样。
- 用户的钱包应用自动从可用的UTXO中选取不同的金额，组合成大于或等于所需金额的交易。

22

比特币的UTXO数据模型

- 交易消费的UTXO叫作交易输入，交易创建的UTXO叫作交易输出。
- 比特币价值不停地从一个所有者转移到另一个所有者，形成一个消费和创建UTXO的交易链条。
- 交易通过当前所有者的签名解锁并消费UTXO，通过将其锁定到新的所有者的方式创建新的UTXO。

23

比特币的UTXO数据模型--铸币coinbase交易

- 对于输出输入链来说，也有一个例外，它是一种特殊类型的交易，叫作铸币（coinbase）交易，铸币交易是每个区块的第一笔交易。
- 铸币（coinbase）交易是矿工“赢家”放进区块的，作为矿工挖到区块的奖励。
- 铸币交易并不需要消耗（花费）UTXO。
- 它只有一个输入，叫作币基（coinbase），这个交易从无到有生成了比特币。
- 铸币交易有一个输出，将挖矿和本区块的所有交易手续费汇总支付到矿工的比特币地址。
  - Total Fees = Sum(Inputs) - Sum(Outputs)
- 这也就是比特币系统在挖矿过程中发行新币的过程。

24

比特币数据的本地存储-- LevelDB数据库

比特币程序将数据存在了4个地方

- blocks/blk*.dat的文件中存储了实际的块数据，这些数据以网络格式存储。它们仅用于重新扫描钱包中丢失的交易，将这些交易重新组织到链的不同部分，并将数据块提供给其他正在同步数据的节点。
- blocks/index/*是一个LevelDB数据库，存储着目前已知块的元数据，这些元数据记录所有已知的块以及它们存储在磁盘上的位置。没有这些文件，查找一个块将是非常慢的。
- chainstate/*是一个LevelDB数据库，以紧凑的形式存储所有当前未花费的交易以及它们的元数据。这里的数据对于验证新传入的块和交易是必要的。在理论上，这些数据可以从块数据中重建，但是这需要很长时间。没有这些数据也可以对数据进行验证，但是需要现有块数据进行扫面，这无疑是非常慢的。
- blocks/rev*.dat中包含了“撤销”数据，可以将区块视为链的“补丁”（它们消耗一些未花费的输出并生成新的输出），那么这些撤销数据将是反向补丁。如果需要回滚链，这些数据将是必须的。

比特币程序从网络中接受数据后，会将数据以.dat的形式转储到磁盘上。

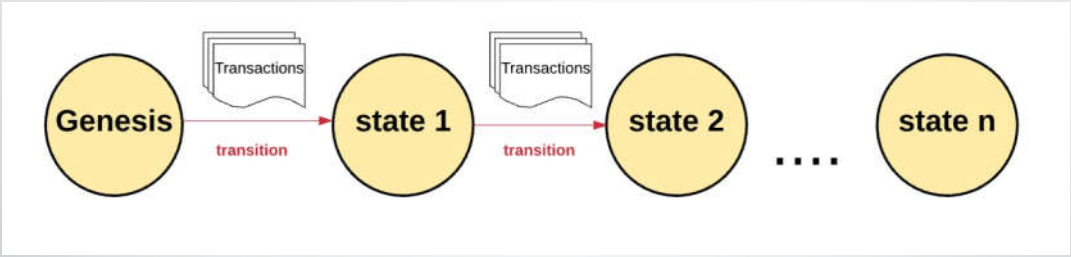
25

区块链的数据结构和数据库

- 区块链的链式数据结构
- 比特币的数据结构和数据库
- **以太坊的数据结构和数据库**

26

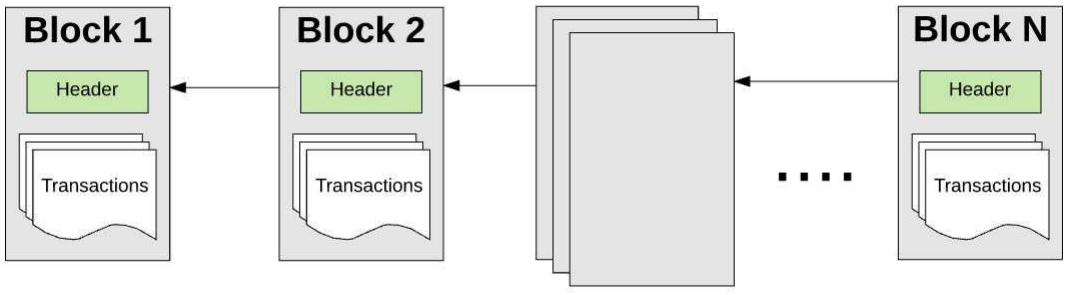
以太坊数据结构



- 以太坊是一个基于交易的**状态机**，其区块链中的每个区块就对应一个状态，每产生一个区块，以太坊中的状态就会转换到下一个状态。通过状态转换使得运行以太坊中的所有节点保持数据的一致性。

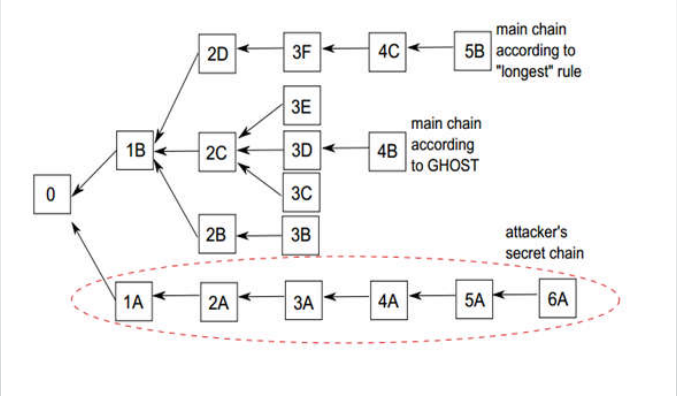
27

以太坊数据结构



28

以太坊数据结构

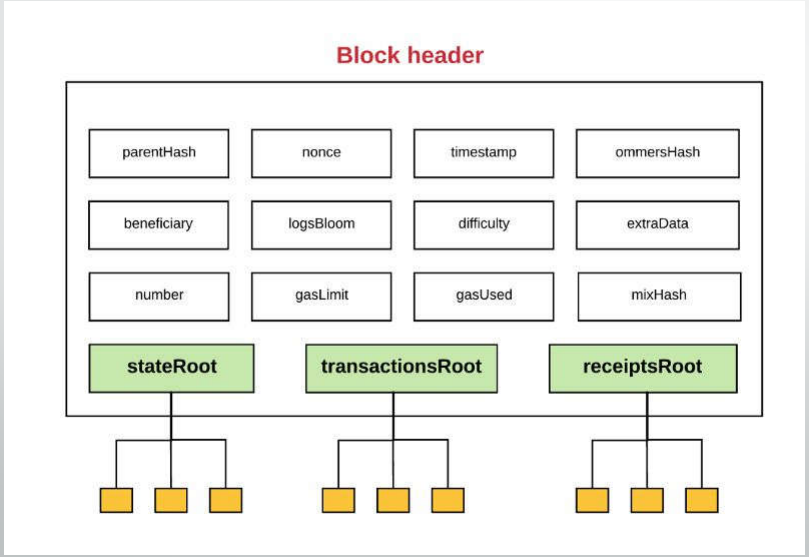


叔区块与GHOST协议

以太坊10-20秒产出一个块

29

以太坊中的区块头



30

以太坊区块头中字段说明

名称	类型	意义
parentHash	common.Hash	父区块的哈希值
UncleHash	common.Hash	叔父区块列表的哈希值
Coinbase	common.Address	打包该区块的矿工的地址，用于接收矿工费
Root	**common.Hash**	**状态树的根哈希值**
TxHash	**common.Hash**	**交易树的根哈希值**
ReceiptHash	**common.Hash**	**收据树的根哈希值**
Bloom	Bloom	交易收据日志组成的Bloom过滤器
Difficulty	*Big.Int	本区块的难度
Number	*Big.Int	本区块块号，区块号从0号开始算起
GasLimit	uint64	本区块中所有交易消耗的Gas上限，这个数值不等于所有交易的中Gas limit字段的和
GasUsed	uint64	本区块中所有交易使用的Gas之和
Timestamp	*big.Int	区块产生的unix时间戳
Extra	[]byte	区块的附加数据
MixDigest	common.Hash	哈希值，与Nonce的组合用于工作量计算
Nonce	BlockNonce	区块产生时的随机值

31

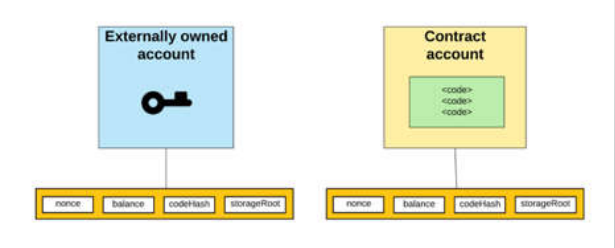
以太坊帐户

- 以太坊的全球「共享状态」是由许多账户组成的，它们能够通过一个消息传递框架相互通信。
- 每个帐户都有一个与它关联的状态和一个20字节的地址。以太坊的地址是一个160位比特的标识符，用于识别帐户。

32

以太坊的两种账户类型

- 外部帐户：由私人密钥控制，没有与之相关的代码。
- 合约账户：由其合约代码控制，并具有与其相关的代码。



33

以太坊的两种账户类型

- 以太坊中这两种账户统称为“状态对象”（存储状态）。
  - **外部账户存储以太币余额状态**
  - **合约账户除了余额还有智能合约及其变量的状态。**
- 通过交易的执行，这些状态对象发生变化，而Merkle树用于索引和验证状态对象的更新。

34

以太坊的外部帐户

- 外部账户一般简称为“账户”，它们都是由人创建的，可以存储以太币，是由公钥和私钥控制的账户。
- 外部账户可以通过创建和使用其私人密钥签署一项交易，向其他外部账户或其他合约账户发送消息。
- 两个外部账户之间的消息只是一种价值转移。
- 但从一个外部帐户到一个合约账户的消息会激活合约账户的代码，使它能够执行各种操作（例如转移代币、写入内存、生成新的代币、执行一些计算、创建新合约等）。

35

以太坊的合约账户

- 合约账户是一个包含合约代码的账户。
- 合约账户不是由私钥文件直接控制，而是由合约代码控制。
- 合约账户的地址是由合约创建时合约创建者的地址，以及该地址发出的交易共同计算得出的。
- 一个合约账户具有下列特性：
  - 拥有一定的以太币余额；
  - 有相关联的代码，代码通过交易或者其他合约发送的调用来激活；
  - 当合约被执行时，只能操作合约账户拥有的特定存储。

36

以太坊的外部帐户 V.S.合约账户

- 合约账户是由外部账户创建的账户。
- 与外部账户不同，合约账户不能自行启动新的交易。相反，合约账户只能根据它们收到的其他交易（从外部账户或从另一个合约账户）进行交易。
- 合约账户和普通账户最大的不同就是它还存有智能合约。
- 在以太坊区块链上发生的任何操作都是由外部控制账户的交易引起的。

37

以太坊的帐户状态

无论帐户是哪种类型，帐户状态都由以下四个部分组成。

- nonce：
  - 如果帐户是一个外部帐户，这个nonce数字代表从帐户地址发送的交易数量。
  - 如果帐户是一个合约帐户，nonce是帐户创建的合约数量。
- balance：这个地址拥有的Wei（以太坊货币单位）数量，每个以太币有10^8 Wei。
- storageRoot：一个Merkle Patricia树根节点的哈希，它对帐户的存储内容的哈希值进行编码，并默认为空。
- codeHash：EVM（以太坊虚拟机）的哈希值代码。
  - 对于合约帐户，这是一个被哈希后并存储为codeHash的代码。
  - 对于外部帐户，codeHash字段是空字符串的哈希。

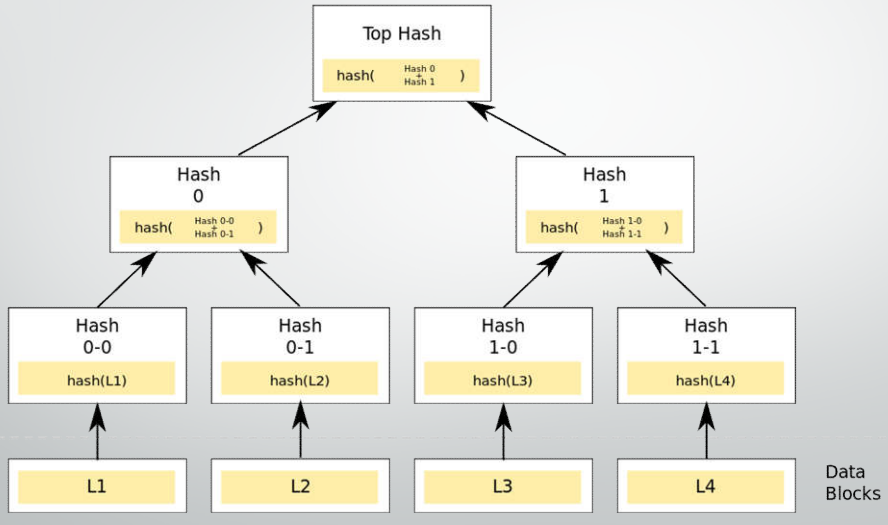
38

以太坊的全局状态存储-- MPT

- 以太坊针对三种对象设计了三棵Merkle Patrcia Tree(MPT)
  - 状态树（Transaction Tree）
  - 交易树（State Tree）
  - 收据树（Receipt Tree）
- MPT 是由 Merkle Tree 与 Patricia Tree结合来的
- 这三种树可以帮助以太坊客户端做一些简易的查询，如查询某个账户的余额、某笔交易是否被包含在区块中等。

39

Merkle Tree



40

Trie 前缀树

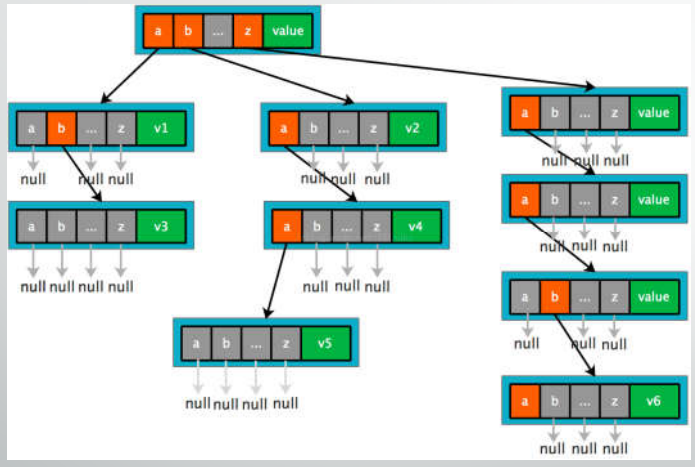
- Trie 前缀树是一种有序树状的数据结构，其中的键通常是字符串，常用语存储 Key-value 数据结构。
- 与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀，节点对应的 key 是根节点到该节点路径上的所有节点 key 值前后拼接而成，节点的 value 值就是该 key 对应的值。根节点对应空字符串 key。

前缀树基本性质

1. 根节点不包含字符，除根节点意外每个节点只包含一个字符。
2. 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符串不相同。

41

Trie 前缀树-示例



- ['a'] = V1,
- ['ab'] = V3,
- ['b'] = V2,
- ['ba'] = V4,
- ['baa'] = V5,
- ['zaab] = V6

如果 key 是英文单词，trie 的每个节点就是一个长度为 27 的指针数组，indexo-25 代表 a-z 字符，26 为标志域。

42

Patricia压缩前缀树

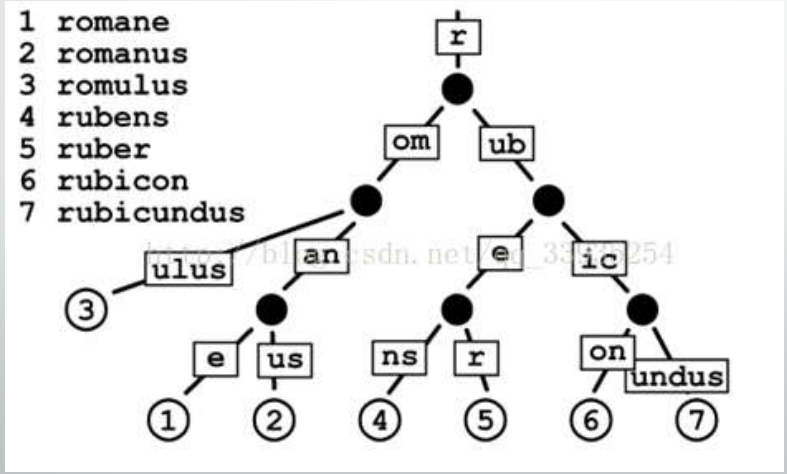
Patricia树，或称Patricia Trie，压缩前缀树

- 一种编码方式，它是传统Trie 前缀树的改进
- 对于基数树的每个节点，如果该节点是唯一的儿子的话，就和父节点合并

43

Patricia压缩前缀树–示例

1 romane
2 romanus
3 romulus
4 rubens
5 ruber
6 rubicon
7 rubicundus



44

Merkle Patrcia Tree(MPT)

- Merkle Tree和Patricia Tree融合，就产生了MPT（Merkle Patricia Tree）
- 在以太坊（ethereum）中，使用了一种特殊的十六进制前缀(hex-prefix, HP)编码，所以在字母表中就有16个字符。这其中的一个字符为一个nibble。
- 半字节nibble，用4比特表示。

45

Merkle Patrcia Tree(MPT)

- MPT树中的节点包括空节点、叶子节点、扩展节点和分支节点
  - 空节点，简单的表示空，在代码中是一个空串。
  - 叶子节点（leaf），表示为[key,value]的一个键值对，其中key是key的一种特殊十六进制编码，value是value的RLP编码。
  - 扩展节点（extension），也是[key，value]的一个键值对，但是这里的value是其他节点的hash值，这个hash可以被用来查询数据库中的节点。也就是说通过hash链接到其他节点。
  - 分支节点（branch），因为MPT树中的key被编码成一种特殊的16进制的表示，再加上最后的value，所以分支节点是一个长度为17的list，前16个元素对应着key中的16个可能的十六进制字符，如果有一个[key,value]对在这个分支节点终止，最后一个元素代表一个值，即分支节点既可以搜索路径的终止也可以是路径的中间节点。

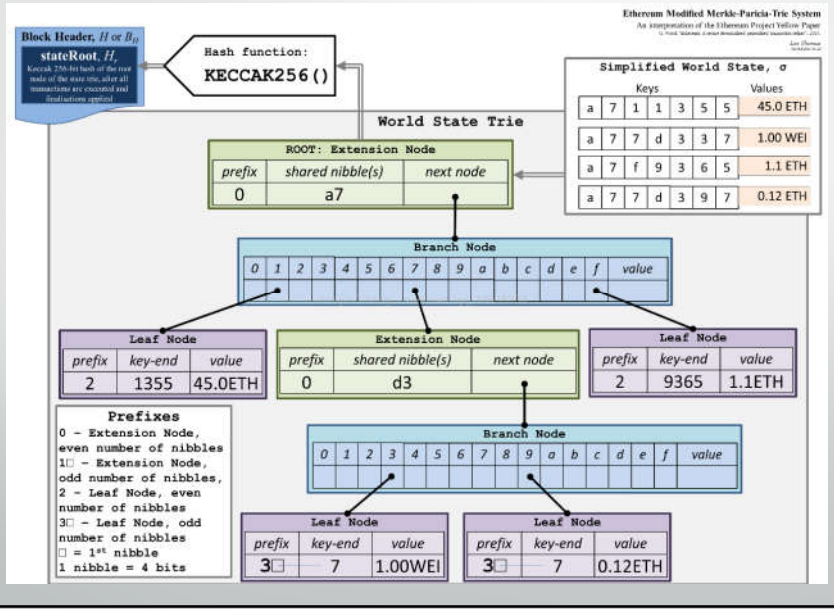
46

Merkle Patrcia Tree(MPT)

- MPT树中另外一个重要的概念是一个特殊的十六进制前缀(hex-prefix, HP)编码，用来对key进行编码。因为字母表是16进制的，所以每个节点可能有16个孩子。
- 因为有两种[key,value]节点(叶节点和扩展节点)，引进一种特殊的终止符标识来标识key所对应的是值是真实的值，还是其他节点的hash。
  - 如果终止符标记被打开，那么key对应的是叶节点，对应的值是真实的value。
  - 如果终止符标记被关闭，那么值就是用于在数据块中查询对应的节点的hash。
- 无论key奇数长度还是偶数长度，HP都可以对其进行编码。最后我们注意到一个单独的hex字符或者4bit二进制数字，即一个nibble。HP编码很简单。一个nibble被加到key前（下图中的prefix），对终止符的状态和奇偶性进行编码。最低位表示奇偶性，第二低位编码终止符状态。如果key是偶数长度，那么加上另外一个nibble，值为0来保持整体的偶特性。

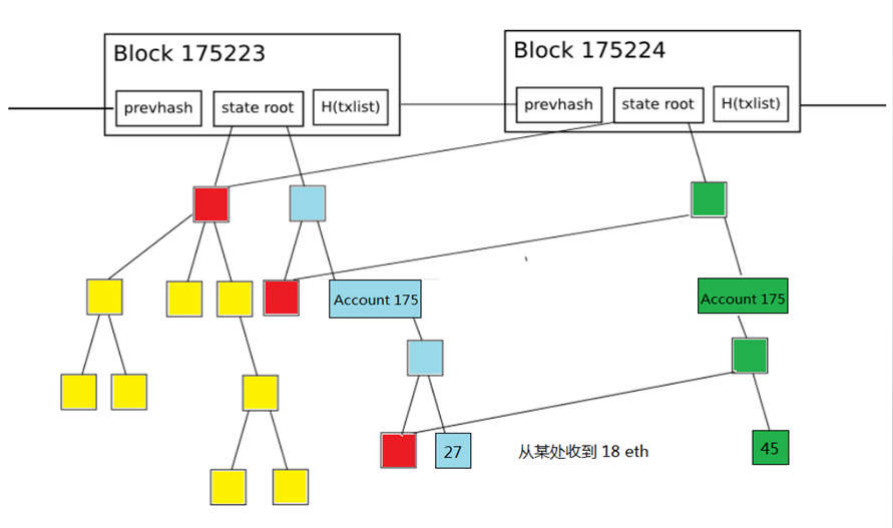
47

Merkle Patrcia Tree(MPT)



48

Transaction Tree 的状态变迁



49

以太坊的状态树

- 状态树中的每个节点有16个孩子节点，每个叶节点表示一个账户，这些叶节点的父节点由叶节点的散列组成，而这些父节点再组成更高一层的父节点，直至到形成根节点。
- 状态树包含一个键值映射，其中键是账户地址，值是账户内容，主要是｛nonce, balance, codeHash, storageRoot｝。
  - nonce是账户交易的序数
  - balance是账户余额
  - codeHash是代码的散列值
  - storageRoot是另一棵树的根节点。
- 状态树代表访问区块后的整个状态。

50

以太坊的交易树

- 每个区块都有一棵独立的交易树。区块中交易的顺序主要由“矿工”决定，在这个块被挖出前这些数据都是未知的。不过“矿工”一般会根据交易的GasPrice和nonce对交易进行排序。
  - 首先会将交易列表中的交易划分到各个发送账户，每个账户的交易根据这些交易的nonce来排序。
  - 每个账户的交易排序完成后，再通过比较每个账户的第一条交易，选出最高价格的交易，这些是通过一个堆（heap）来实现的。
- 每挖出一个新块，更新一次交易树。
- 在交易树包含的键值对中，其中每个键是交易的编号，值是交易内容。

51

以太坊的收据树

- 每个区块都有自己的收据树，收据树不需要更新，收据树代表每笔交易相应的收据。
- 收据树也包含一个键值映射，其中键是索引编号，用来指引这条收据相关交易的位置，值是收据的内容。

52

以太坊的数据库支持--LevelDB

- LevelDB是Google实现的一个非常高效的键值对数据库，其中键值都是二进制的，目前能够支持十亿级别的数据量，在这个数据量下还有着非常高的性能。以太坊中共有三个LevelDB数据库，
  - BlockDB保存了块的主体内容，包括块头和交易；
  - StateDB保存了账户的状态数据；
  - ExtrasDB保存了收据信息和其他辅助信息。
- LevelDB的用户接口非常简单，包括put(k,v)、get(k,v)和delete(k,v)

53

LevelDB数据库概述（续）

LevelDB特点：

- 1、key和value都是任意长度的字节数组；
- 2、entry（即一条K-V记录）默认是按照key的字典顺序存储的，当然开发者也可以重载这个排序函数；
- 3、提供的基本操作接口：Put()、Delete()、Get()、Batch()；
- 4、支持批量操作以原子操作进行；
- 5、可以创建数据全景的snapshot(快照)，并允许在快照中查找数据；
- 6、可以通过前向（或后向）迭代器遍历数据（迭代器会隐含的创建一个snapshot）；
- 7、自动使用Snappy压缩数据；
- 8、可移植性；

LevelDB限制：

- 1、非关系型数据模型（NoSQL），不支持sql语句，也不支持索引；
- 2、一次只允许一个进程访问一个特定的数据库；
- 3、没有内置的C/S架构，但开发者可以使用LevelDB库自己封装一个server；

54

以太坊的数据存储层次

- 【levelDB数据库】

  在以太坊每个节点上，都有一个levelDB数据库，所有的信息，包括区块信息、账户状态、交易信息等等，都会以key/value对的方式存储在这个数据库里。

- 【MPT树】

  在levelDB的基础上，构造并存储各种MPT树。

- 【区块】

  比如交易信息、各种树的根哈希等等，以区块的结构呈现。

55

下课

56