

详解以太坊的工作原理

<https://linux.cn/article-9699-1.html>

基于以太坊黄皮书

区块链的定义

区块链就是一个**具有共享状态的密码性安全交易的单机**（cryptographically secure transactional singleton machine with shared-state）。

密码性安全：通过难以破解的复杂数学机制算法保证数字货币生产的安全性，使得欺骗系统的行为不可能发生。

交易的单机：系统中的交易由单一的权威机器实例负责，或是全球真相。

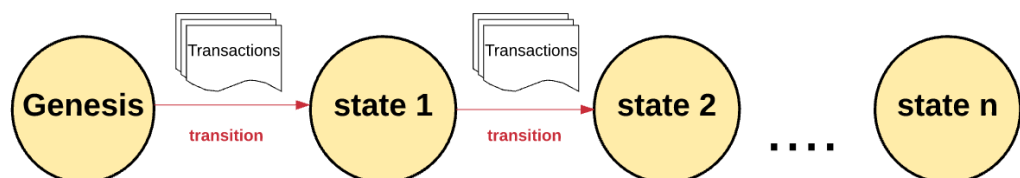
具有共享状态：这台机器上存储的状态共享，对每个人开放。

以太坊模型

1. 状态机

以太坊本质是一个**基于交易的状态机**（transaction-based state machine）。状态机读取一系列的输入并转换成新的状态出来。

从**创世状态**（genesis state）开始，在交易被执行时状态会转变。任何时刻的最终状态都代表着以太坊当前的状态。



2. 链状模型

以太坊的状态包含百万个交易，被打包到一个区块中。每个区块包含着一系列交易，并于前一区块链接。

3. 挖矿

为了让一个状态转换成下一个状态，交易必须是有效的。为了让一个交易被认为是有效的，它必须要经过一个验证过程，此过程也就是挖矿。挖矿就是一组节点（即电脑）用它们的计算资源来创建一个包含有效交易的区块出来。即**利用计算资源换取区块**。

任何在网络上宣称自己是矿工的节点都可以尝试创建和验证区块。世界各地的很多矿工都在同一时间创建和验证区块。每个矿工在提交一个区块到区块链上的时候都会提供一个数

学机制的“证明”，这个证明就像一个保证：如果这个证明存在，那么这个区块一定是有效的。

为了让一个区块添加到主链上，一个矿工必须要比其他矿工更快的提供出这个“证明”。通过矿工提供的一个数学机制的“证明”来证实每个区块的过程称之为工作量证明 PROOF OF WORK。

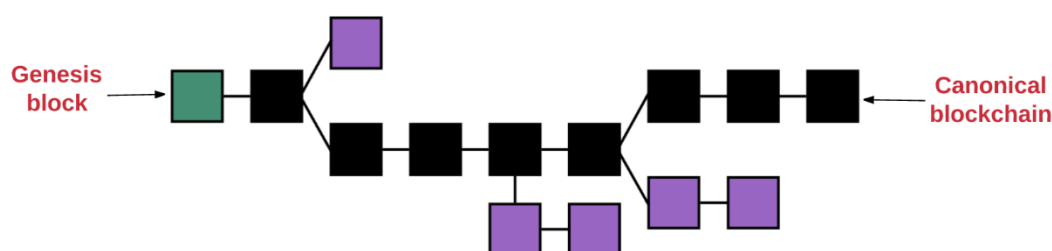
证实了一个新区块的矿工都会被奖励一定价值的奖赏。以太坊使用一种内在数字代币——以太币 ETHER 作为奖赏。每次矿工证明了一个新区块，那么就会产生一个新的以太币并被奖励给矿工。

4. 共识机制

矿工是无法创建另一条链的。由于定义了区块链就是一个具有共享状态的**交易单机**。使用这个定义，我们可以知道正确的当前状态是一个**全球真相**，所有人都必须要接受它。拥有多个状态（或多个链）会摧毁这个系统，因为它在哪个是正确状态的问题上不可能得到统一结果。如果链分叉了，你有可能在一条链上拥有 10 个币，一条链上拥有 20 个币，另一条链上拥有 40 个币。在这种场景下，是没有办法确定哪个链才是最“有效的”。

为了确定哪个路径才是最有效的以及防止多条链的产生，以太坊使用了一个叫做“**GHOST 协议**”的数学机制。（Greedy Heaviest Observed Subtree）

GHOST 协议就是让我们必须选择一个在其上完成**计算最多的路径**。一个方法确定路径就是使用最近一个区块（叶子区块）的区块号，区块号代表着当前路径上总的区块数（不包含创世区块）。区块号越大，路径就会越长，就说明越多的挖矿算力被消耗在此路径上以达到叶子区块。使用这种推理就可以允许我们赞同当前状态的权威版本。



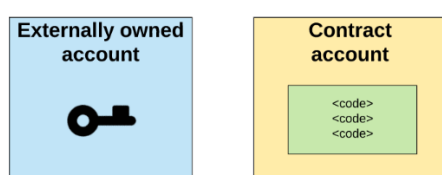
以太坊系统组成

1. 账户

以太坊的全局“共享状态”是有很多小对象（账户）来组成的，这些账户可以通过消息传递来与对方进行交互。**每个账户**都有一个与之关联的**状态 state** 和一个 **20 字节的地址 address**。在以太坊中一个地址是 **160 位的标识符**，用来识别账户。

外部拥有的账户：被**私钥控制**且没有任何代码与之关联。

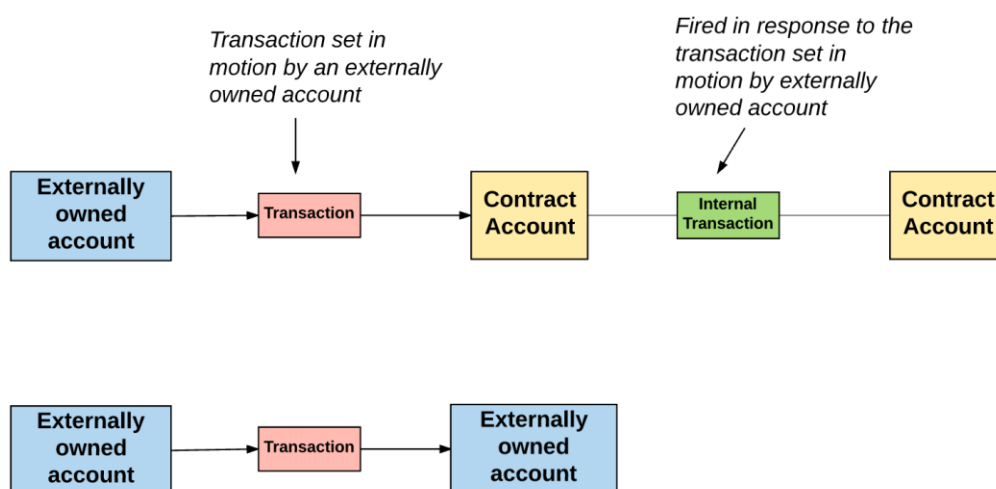
合约账户：被它们的**合约代码控制**且有代码与之关联。



- 比较

一个外部拥有账户可以通过创建和用自己的私钥来对交易进行签名,来发送消息给另一个外部拥有账户或合约账户。在两个外部拥有账户之间传送的消息只是一个简单的**价值转移**。但是从外部拥有账户到合约账户的消息会**激活合约账户的代码**,允许它执行各种动作。(比如转移代币,写入内部存储,挖出一个新代币,执行一些运算,创建一个新的合约等等)。

不像外部拥有账户,**合约账户不可以自己发起一个交易**。相反,合约账户只有在接收到一个交易之后(从一个外部拥有账户或另一个合约账户处),为了响应此交易而触发一个交易。



总的来说, **以太坊上的任何动作都是由外部拥有账户触发的交易所发动的**。

*chatGPT-以太坊的**合约账户地址**:*

以太坊的合约账户也具有地址。与外部账户类似,以太坊的合约账户也使用 20 字节的地址来唯一标识它们。但是,与外部账户不同的是,合约账户的地址不是由公钥经过哈希算法得出的,而是由合约创建时的**交易数据计算出来的**。在以太坊中,每当创建一个新的合约时,都会分配一个新的地址。这个地址是**由创建交易的发送方地址和发送方的 **NONCE** 值计算得出的**。因此,合约地址是**可预测的**,而且不同的合约地址之间是**不相关的**。当一个合约账户执行了一些操作并且需要转账时,它会发出一笔交易,这笔交易的目标地址就是另一个外部账户或者合约账户的地址。

*chatGPT-**公私钥对**说明:*

以太坊合约账户也有对应的公钥和私钥对。但合约账户的公钥并不是直接存储在区块链中的,而是通过**合约代码计算**出来的。在合约账户的创建过程中,需要提供合约代码,并通过代码执行来创建合约账户。创建合约账户时,合约代码中的**构造函数**会计算出合约账户的公钥,然后将公钥存储在账户状态中。这样,当以太坊节点需要验证合约账户的签名时,就可以通过合约代码重新计算出公钥,并使用这个公钥验证签名。

需要注意的是,合约账户的**私钥是由创建合约账户的人保管的**,而不是由合约账户本身保管。这也是因为合约账户并不具备普通账户的所有功能,例如发送交易等。相反,它只能执行其内部编写的合约代码。因此,合约账户的私钥通常由创建合约的人或团队负责保管,以确保合约账户的安全性。

*chatGPT-以太坊中**账户和区块链节点**的关系:*

账户是区块链网络中的**实体**，用于存储以太币（ETH）和其他代币，以及记录与该账户相关的交易历史和智能合约状态。

区块链节点是以太坊网络中的**参与者**，它们运行**以太坊协议**软件以连接网络并与其他节点进行通信。节点之间通过**点对点协议**交换数据，并使用**共识算法**来确认和验证交易和区块。节点可以是完整节点（FULL NODE）、轻量级节点（LIGHT NODE）或 SPV 节点（SIMPLIFIED PAYMENT VERIFICATION NODE），这取决于它们**存储的区块链数据的完整性和广度**。

在以太坊中，节点可以运行一个以太坊客户端，如 GETH 或 PARITY，来与以太坊网络进行交互。当一个节点发送交易时，它将向网络广播该交易，并等待其他节点的确认和验证。一旦交易被打包进区块，所有节点将更新他们的区块链副本以包括最新的交易和状态更改。

以太坊中的账户和区块链节点之间的关系是这样的：账户是以太坊网络中的**实体**，用于存储资产和交易历史；节点是参与网络的**计算机**，用于广播和验证交易，并维护区块链的完整性。节点通过互相交流，共同构建和维护以太坊网络。

*chatGPT-合约账户生成后，内部合约还可以**更改**吗*

可以，合约账户**创建**之后，它的内部代码可以通过智能合约的**更新机制**进行更改。智能合约是以太坊上的一种程序，类似于一个自动执行的合同，它可以在以太坊上运行，并且可以包含一些状态、变量和代码。

当一个智能合约被**部署**到以太坊网络中后，它的代码是**不可变**的。但是，智能合约通常都提供了一些可以用于更新代码和状态的函数。这些函数需要经过**授权才能调用**，通常只有**智能合约的拥有者**可以调用它们。

一旦智能合约的代码被**更新**，它就会被**重新部署**到以太坊网络中，**合约账户的代码**就会被更新。所有之前基于旧合约代码创建的**合约实例也将被更新**，以便它们能够反映最新的智能合约代码。

需要注意的是，智能合约的代码更新可能会对之前已经部署的合约实例产生**影响**，因此需要谨慎考虑代码更新的必要性和影响。此外，更新智能合约代码需要支付燃料费用，因为需要在以太坊网络上执行新的智能合约代码。

*chatGPT-以太坊的**合约与合约账户**：*

在以太坊中，合约指的是一个智能合约**程序**，而合约账户是指存储智能合约程序代码的一种**账户类型**。

智能合约程序是以太坊中的应用程序，由 SOLIDITY 等高级语言编写而成，部署在以太坊网络上。智能合约程序可以被看作是一种计算机程序，它们被存储在合约账户中，并可以被其他账户调用执行。智能合约程序可以用来**实现各种功能**，例如数字货币交易、金融衍生品、投票系统等等。

合约账户是一种特殊类型的账户，**存储了智能合约程序的代码和状态**。与外部账户不同，合约账户的**执行**需要消耗一定数量的**燃气**，以支付计算机资源的成本。在以太坊中，合约账户有一个地址，可以接收以太币和发送交易。但合约账户不能直接对外发送交易，必须通过其他账户对其发起**调用**才能执行其中的代码。

在以太坊中，智能合约程序的执行过程需要消耗燃气，燃气相当于计算机资源的计量单位。当合约账户执行代码时，所消耗的燃气量将从合约账户余额中扣除，如果余额不足，则执行失败。因此，**合约账户的余额必须足够支付其代码执行的成本**。

2. 账户状态

不论是外部拥有账户还是合约账户类型，账户状态都是四个组成部分。其实还有个地址

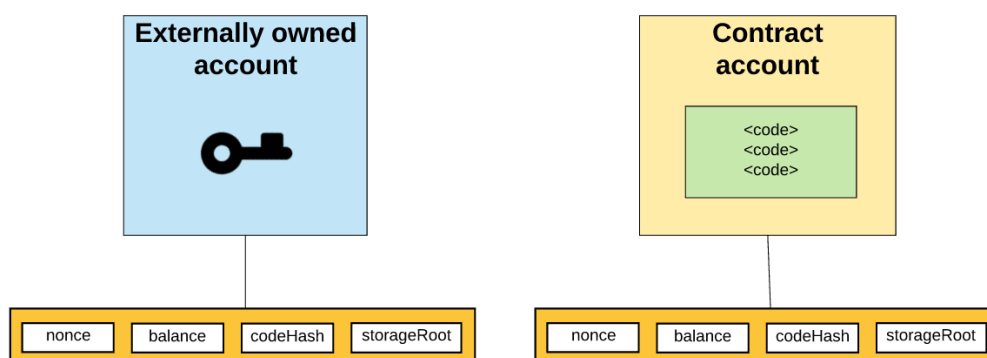
ADDRESS。

序号 (nonce): 如果账户是一个外部拥有账户, 序号代表从此账户地址发送的**交易序号**。如果账户是一个合约账户, 序号代表此账户创建的**合约序号**。

余额 (balance): 此地址拥有 Wei 的数量。1Ether=10¹⁸Wei。

存储根 (storageRoot): Merkle Patricia 树的**根节点哈希值**。梅克尔树会将此账户存储内容的哈希值进行编码, 默认是空值。

代码哈希 (codeHash): 此账户 EVM (以太坊虚拟机) **代码的哈希值**。对于合约账户, 就是被哈希的代码并作为代码根保存。对于外部拥有账户, 代码根这个域是一个空字符串的哈希值。



数据结构为:

```
css Copy code

{
  address: '0x6F1a7698c7b3D3d3C6026Df0D8C7cD62a78C56E6',
  nonce: 8,
  balance: '4700854525000000000',
  storageRoot: '0x05a7301fe7e7086b85c6b05aa64300e43297f9a6a8f6ce1e6d2d6c77ada',
  codeHash: '0x5974ac9b9c4475e5c5bce29d48419a5ec67fbf020e5189263244241e8649e70',
}
```

*chatGPT-以太坊和比特币中 **nonce** 的差别与联系:*

在以太坊和比特币中, NONCE 都是用于 **PoW 算法** 的一个参数, 但具体实现有一些不同。

比特币的 PoW 算法是 **SHA-256**, 其中 NONCE 是一个 **32 位的随机数**, 用于尝试找到一个满足比特币难度目标的哈希输出。由于比特币中的区块大小是**固定的**, 因此 NONCE 可以用于生成不同的哈希输出, 直到找到符合要求的哈希值为止。这就是比特币挖矿的过程, 挖矿者在尝试不同的 NONCE 值来寻找一个符合条件的哈希值。

在以太坊中, PoW 算法是以太坊哈希算法 (ETHASH), 其中 NONCE 是一个 **64 位的随机数**, 用于尝试找到一个满足以太坊难度目标的哈希输出。以太坊的区块大小是**可变的**, 因此在以太坊中, NONCE 不仅可以用于寻找符合要求的哈希值, 还可以用于生成不同的区块头 (BLOCK HEADER), 从而影响整个区块的哈希值。

CHATGPT-不同的 NONCE:

虽然它们都被称为“NONCE”，但以太坊中账户状态中的 NONCE 和区块链算法中的 NONCE 并**完全不同**。在区块链中，NONCE（随机数）是指为了满足区块哈希的目标难度而需要计算的数值，它是**由矿工随机选择**的，以保证每个矿工都能有公平的机会成为下一个区块的验证者。而在以太坊中，账户状态中的 NONCE 是一个**整数**，它记录了一个账户的**交易数量**，用于**避免重复交易**。每当一个账户发送一笔交易时，它的 NONCE 就会增加 1。这个 NONCE 在交易被打包进区块之前必须是唯一的，因为它会被用来计算交易的哈希值。

chatGPT-合约账户中的 *nonce* 是什么

在以太坊中，合约账户也有一个 NONCE 字段，它与外部账户的 NONCE 字段的含义不同。**合约账户的 NONCE 表示账户执行的交易数**，即在以太坊上执行的交易中，每一个发送到合约账户的交易都会导致 NONCE 增加 1。

与外部账户的 NONCE 不同，**合约账户的 NONCE 不能用于防止交易重播**。这是因为在以太坊中，交易重播的概念与比特币不同，交易在网络中传播和执行是由交易的 HASH 值唯一确定的。换句话说，交易被广播到网络上，所有节点都会按照相同的顺序执行这些交易，而不是像比特币一样通过 NONCE 来防止交易重播。因此，合约账户的 NONCE 仅用于表示账户执行的交易数量，而不适用于防止交易重复执行。

在以太坊中，NONCE 是一个非常重要，它可以防止**重放攻击**，也可以确保交易顺序的正确性。因此，正确地处理 NONCE 是以太坊中非常重要的一部分。

chatGPT-合约账户的**余额**:

合约账户可以**拥有以太币余额**，也可以通过**执行智能合约**而获得更多的以太币余额。

当一个合约被创建时，会在以太坊的区块链上生成一个新的合约账户。合约账户拥有自己的地址，并可以**向其他账户发送和接收以太币**。但是，合约账户**不能通过私钥进行转账，只能通过合约代码中定义的函数来执行转账操作**。

合约账户的余额是存储在以太坊虚拟机 EVM 的**状态数据库**中，以太坊中每一个账户都有一个对应的状态对象，其中包含了账户的余额、代码、存储数据和其他状态信息。合约账户的余额可以通过以太坊的状态转换机制来修改，例如，当一个用户调用合约中的一个函数并支付以太币时，合约账户的余额就会增加相应的数量。

需要注意的是，合约账户**不能像普通账户一样使用私钥进行签名和交易**。因此，在合约代码中，需要定义一些函数来确保账户余额的正确性，并对账户的余额进行合理的管理。

在以太坊中，合约账户的以太币余额是**属于合约账户本身的**，而不是属于合约的创建者或合约执行者。

当创建一个合约时，合约账户的余额初始化为 0，它可以接收以太币的唯一途径是通过**智能合约中的支付函数**。如果一个普通账户向合约账户发送以太币，那么这些以太币将被存储在合约账户的余额中，但这并不意味着这些以太币属于合约创建者或执行者。合约账户的余额只能由合约代码中的函数来修改，因此只有合约代码中明确指定了发送方才能获得合约账户的以太币余额。

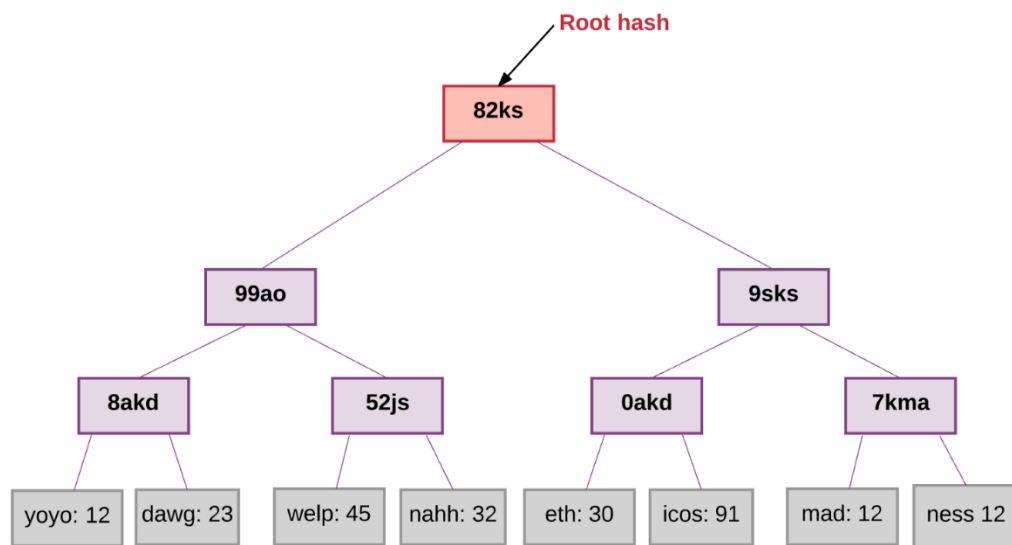
这种设计保证了以太坊的去中心化和智能合约的安全性，因为在以太坊中，智能合约是一种自治的程序，它们运行在以太坊虚拟机上，而不是由中心化的第三方管理。合约账户的余额属于合约本身，而不是属于创建者或执行者，这意味着没有人可以通过控制合约账户的余额来控制合约的执行，从而保障了合约的安全性和可靠性。

3. 世界状态

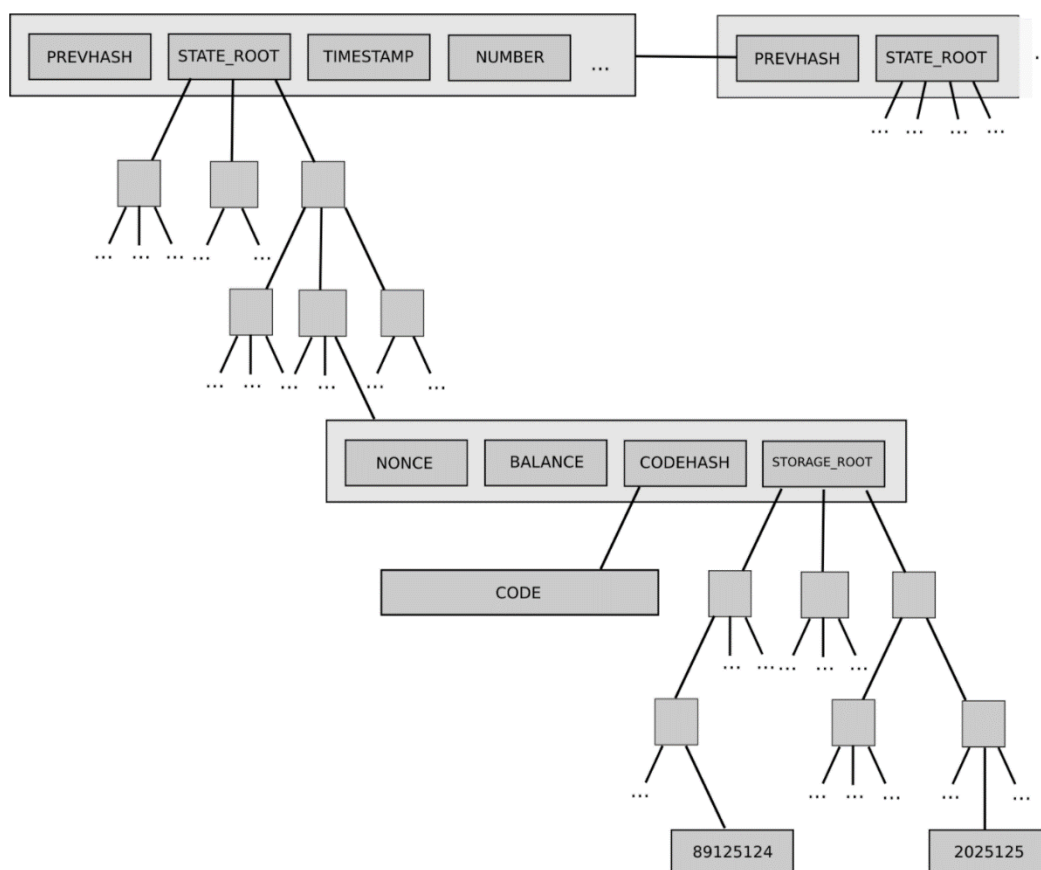
以太坊的全局状态就是由账户地址和账户状态组成的一个映射。这个映射被保存在 Merkle Patricia 树的数据结构中。

chatGPT-merkle 树中间节点的值是怎么算出的：

一个非叶子节点的哈希值是由其左右两个子节点的哈希值**拼接**在一起, 然后再通过一个**哈希**函数 (如 SHA256) 计算得到的。假设 MERKLE 树中一个非叶子节点的左右子节点哈希值分别为 L 和 R , 则该非叶子节点的哈希值 H 可以表示为: $H = \text{HASH}(L || R)$ 。其中 $||$ 表示拼接操作, HASH 表示一个哈希函数。

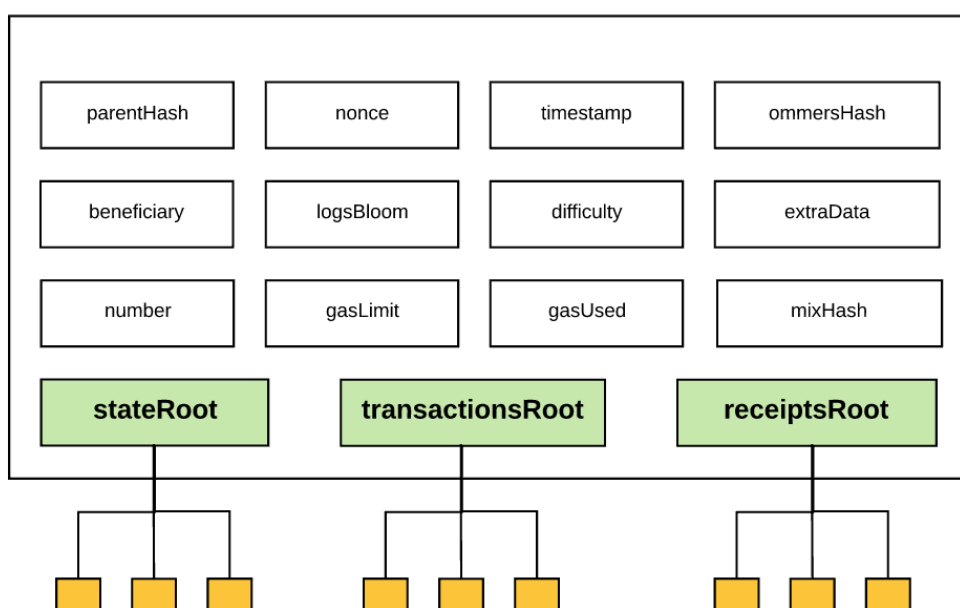


这棵树要求存在里面的值都有一个对应的键。从树的根节点开始, 键会告诉你顺着哪个子节点可以获得对应的值, 这个值存在叶子节点。在以太坊中, 键/值是地址和与地址相关联的账户之间状态的映射, 包括每个账户的余额、序号、代码哈希存储根 (存储根自己就是一棵树)。



相应的，Merkle 树也用来存储交易与收据，每个块头中都保存了三个 MP 树的根节点哈希，三棵树分别为：状态树、交易树与收据树。

Block header



区块链系统中包含两种节点类型：全节点和轻节点。全节点通过下载整条链来进行同步，

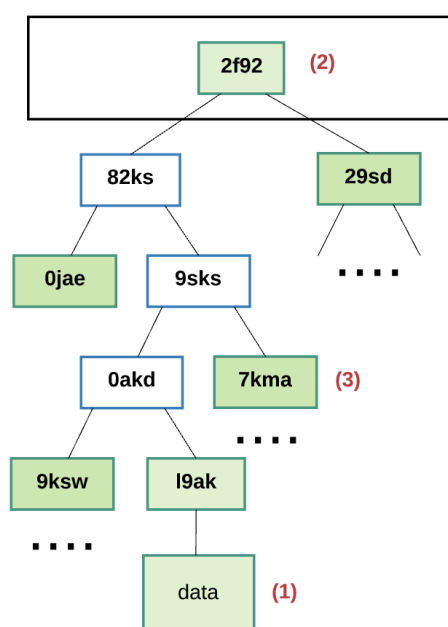
从创世块到当前块，执行其中包含的所有交易。通常，矿工会存储全节点，因为他们在挖矿过程中需要全节点。也有可能下载一个全节点而不用执行所有的交易。无论如何，一个全节点包含了整个链。

不过除非一个节点需要执行所有的交易或轻松访问历史数据，不然没必要保存整条链。这就是轻节点概念的来源。比起下载和存储整个链以及执行其中所有的交易，轻节点仅仅下载链的头，从创世块到当前块的头，不执行任何的交易所检索任何相关联的状态。由于轻节点可以访问区块头，而头中包含了 3 个 MP 树的根哈希值，所有轻节点依然可以很容易生成和接收关于交易、事件、余额等可验证的答案。

这个可以行的通是因为在 MP 树中哈希值是向上传播的——如果一个恶意用户试图用一个假交易来交换梅克尔树底的交易，这个会改变它上面节点的哈希值，而它上面节点的值的变化也会导致上上一个节点哈希值的改变，以此类推，一直到树的根节点。

● Merkle 证明

Merkle 证明包括：一块需要验证的数据、树的根节点的 hash 值、一个分支（从数据块到根的路径上的所有哈希值）。



总之，使用梅克尔树的好处就是该结构的根节点加密取决于存储在树中的数据，而且根节点的哈希值还可以作为该数据的安全标识。由于块的头包含了状态树、交易树、收据树的根 Hash 值，所以任何节点都可以验证以太坊的一小部分状态而不用保存整个状态，这个状态的大小可能是非常大的。

燃料和费用

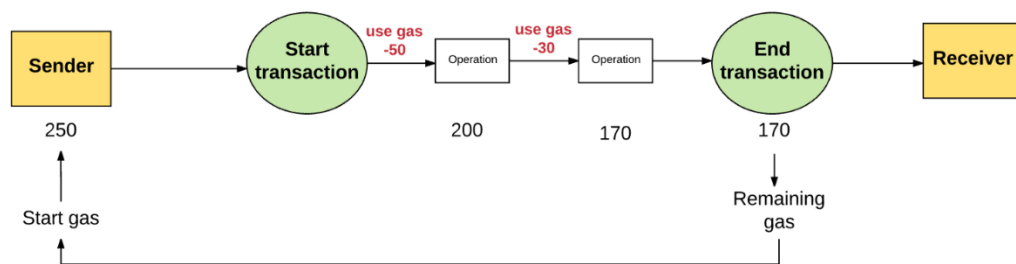
以太坊网络上的交易产生的每一次计算，都会产生费用。

换算关系为 $1\text{Ether}=10^{18}\text{Wei}$ ， $1\text{gwei}=10^9\text{Wei}$ 。

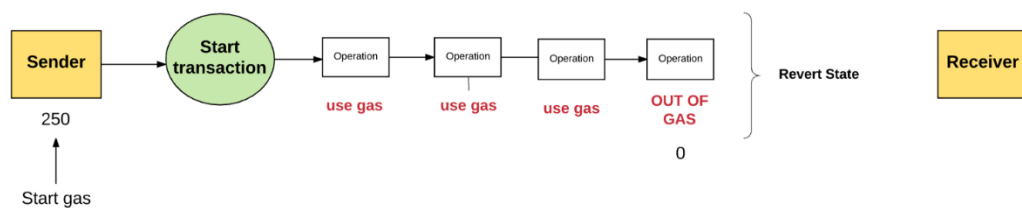
对每个交易而言，发送者需要设置燃料限制（gas limit）和燃料价格（gas price）来表示发送者愿为执行交易支付 Wei 的最大值。

Gas Limit 50,000	X	Gas Price 20 gwei	=	Max transaction fee 0.001 Ether
----------------------------	---	-----------------------------	---	---

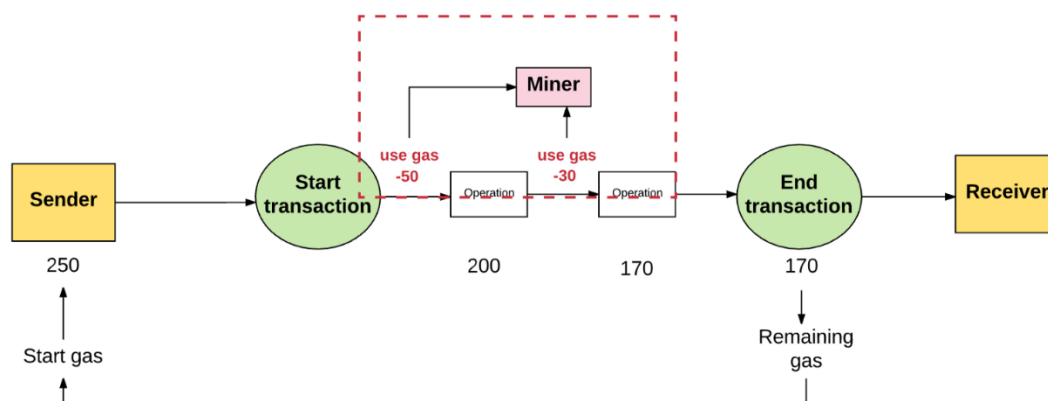
燃料价格代表用户愿意花费在燃料上费用的最大值。如果在他们的账户余额中有足够的以太币来支付这个最大值费用，那么就没问题。在交易结束时任何未使用的燃料都会被返回给发送者，以原始费率兑换。



在发送者没有提供足够的燃料来执行交易，那么交易执行就会出现“燃料不足”然后被认为是无效的。在这种情况下，交易处理就会被终止以及所有已改变的状态将会被恢复，最后我们就又回到了交易之前的状态——完完全全的之前状态就像这笔交易从来没有发生。因为机器在耗尽燃料之前还是为计算做出了努力（算力已经使用过了），所以理论上，将不会有任何的燃料被返回给发送者。



发送者在燃料上花费的所有费用都被发送到“受益人”的地址，通常情况下就是矿工的地址。因为矿工为了计算和验证交易做出了努力，所以矿工接收燃料的费用作为奖励。



通常,发送者愿意支付更高的燃料价格,矿工从这笔交易中就能获得更多的价值。因此,矿工也就更加愿意选择这笔交易。这样的话,矿工可以自由的选择自己愿意验证或忽略的交易。为了引导发送者设置合理的燃料价格,矿工可以选择建议一个最小的燃料值,此值代表自己愿意执行交易的最低价格。(竞价机制)

1. 存储费用

燃料不仅仅是用来支付计算这一步的费用,而且也用来支付存储的费用。存储的总费用与所使用的 32 位字节的最小倍数成比例。

存储费用有一些比较细微的方面。比如,由于增加的存储增加了所有节点上的以太坊状态数据库的大小,所以这种做法可以保持数据存储量较小。出于这个原因,如果一个交易的执行有步骤是清除一个存储实体,那么为执行这个操作的费用就会被放弃,并且由于释放存储空间的退款就会被返回给发送者。

● 费用的作用

以太坊可以运作的一个重要方面就是每个网络执行的操作同时也被全节点所影响。然而,计算的操作在以太坊虚拟机上是非常昂贵的。因此,以太坊智能合约最好是用来执行最简单的任务,比如运行一个简单的业务逻辑或者验证签名和其他密码对象,而不是用于复杂的操作,比如文件存储,电子邮件,或机器学习,这些会给网络造成压力。施加费用防止用户使网络超负荷。

以太坊是一个图灵完备语言(短而言之,图灵机器就是一个可以模拟任何电脑算法的机器。对于图灵机器不太熟悉的人可以看看这个和这个)。这就允许有循环,并使以太坊受到停机问题的影响,这个问题让你无法确定程序是否无限制的运行。如果没有费用的话,恶意的执行者通过执行一个包含无限循环的交易就可以很容易的让网络瘫痪而不会产生任何影响。因此,费用保护网络不受蓄意攻击。

你也许会想,“为什么我们还需要为存储付费?”其实就像计算一样,以太坊网络上的存储是整个网络都必须负担的成本。

交易与消息

在两个不同账户之间发生的交易才让以太坊的全局状态从一个状态转换成另一个状态。

最基本的,一个交易就是指被外部拥有账户生成的加密签名的一段指令,序列化之后提交给区块链。交易可以认为是外部世界和以太坊内部状态的桥梁。

由外部拥有账户触发并提交到区块链的交易有消息通信(message calls)和合约创建文

易 (contract creations) 都包含以下字段：

序号 nonce：发送者发送交易数的计数

燃料价格 gasPrice：发送者愿意支付执行交易所需的每个燃料的 Wei 数量

燃料限制 gasLimit：发送者愿意为执行交易支付燃料数量的最大值。此值设置之后在任何计算完成之前就会被提前扣掉

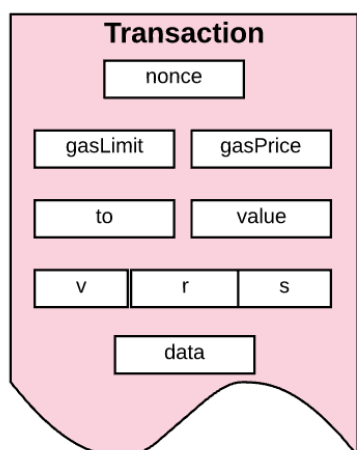
接收者 to：接收者的地址。在合约创建交易中，合约账户的地址还没有存在，所以值先空着

值 value：从发送者转移到接收者 Wei 的数量。在合约创建交易中，值作为新建合约账户的开始余额

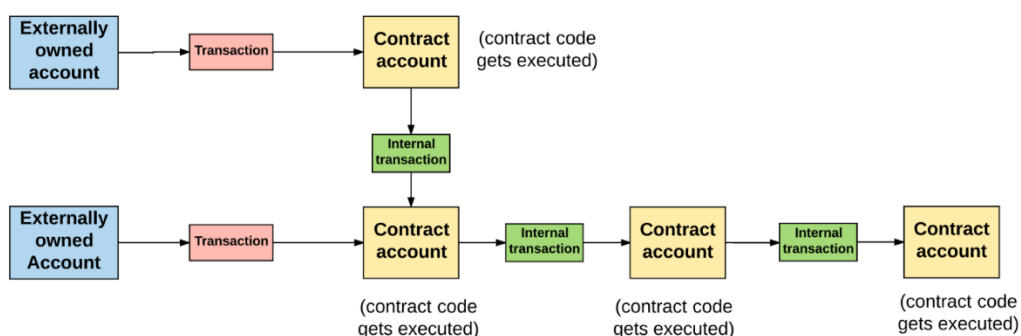
v、r、s：用于产生标识交易发送者的**签名**

初始化值 init（只有在合约创建交易中存在）：用来初始化新合约账户的 EVM 代码片段。初始化值会执行一次，然后就会被丢弃。当初始化值第一次执行的时候，它返回一个账户代码体，也就是**永久与合约账户关联的一段代码**。

数据 data（可选域，只有在消息通信中存在）：消息通信中的输入数据(也就是参数)。例如，如果智能合约就是一个域名注册服务，那么调用合约可能会期待输入参数：域名和 IP 地址。



合约间的通信：在以太坊状态全局范围内的合约可以与在相同范围内的合约进行通信。他们是通过“消息”或者“内部交易”进行通信的。我们可以认为消息或内部交易类似于交易，不过与交易有着最大的不同点——它们不是由外部拥有账户产生的。相反，他们是被合约产生的。它们是虚拟对象，与交易不同，没有被序列化而且只存在于以太坊执行环境。当一个合约发送一个内部交易给另一个合约，存在于接收者合约账户相关联的代码就会被执行。



内部交易或者消息不包含燃料限制 gasLimit。因为燃料限制是由原始交易的外部创建者决定的（也就是外部拥有账户）。外部拥有账户设置的燃料限制必须要高到足够将交易完成，包括由于此交易而产生的任何“子执行”，例如合约到合约的消息。如果，在一个交易或者信息链中，其中一个消息执行造成燃料不足，那么这个消息的执行会被还原，包括任何被此执行触发的子消息。不过，“父执行”没必要被还原。

1. 区块

每个区块包含：

- **区块头**
- 包含在此区块中的**交易集信息**
- 与当前块的 **ommer** 相关一系列其他区块头

● ommer

ommer 是一个区块的父区块与当前区块父区块的父区块是相同的。以太坊的构造使得产生块的速度较快，相应的缺点便是更多的竞争区块会被矿工发现。这种被挖掘出来但不会被加到主链上的区块叫做“**孤区块**”。

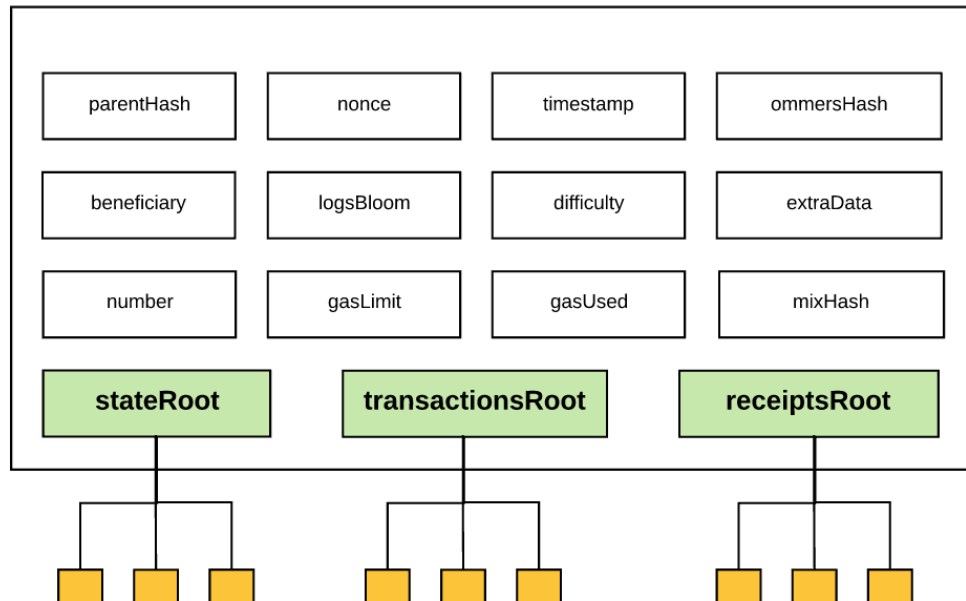
Ommer 的目的就是为了帮助奖励矿工纳入这些孤区块。矿工包含的 ommer 必须是有效的，也就是 ommer 必须是往上数 **6 代之内或更小范围内父区块的子区块**。且 Ommer 区块会收到比全区块**少一点的奖励**。

● 区块头

区块头包含了：

- **父哈希** parentHash：父区块头的哈希值（这也是使得区块变成区块链的原因）
- **ommer 哈希** ommerHash：当前区块 **ommers 列表的哈希值**
- **受益人** beneficiary：接收挖此区块费用的账户地址（矿工）
- **状态根** stateRoot：**状态树根节点**的哈希值
- **交易根** transactionsRoot：包含**此区块所有交易**的梅克尔树的根节点哈希值
- **收据根** receiptsRoot：包含**此区块所有交易收据**的 Merkle 树的根节点 Hash 值
- **日志 Bloom** logsBloom：由日志信息组成的一个 Bloom 过滤器
- **难度** difficulty：此区块的难度级别
- **序号** number：当前区块的计数（创世纪块的区块序号为 0，对于每个后续区块，区块序号都增加 1）
- **燃料限制** gasLimit：每个区块的当前燃料限制
- **所用的燃料** gasUsed：此区块中交易所用的总燃料量
- **时间戳** timestamp：此区块成立时的 Unix 时间戳
- **附加数据** extraData：与此区块相关的附加数据
- **混合哈希** mixHash：一个哈希值，当与随机数 nonce 组合时，证明此区块已经执行了足够的计算
- **随机数** nonce：一个哈希值，当与混合哈希 mixHash 组合时，证明此区块已经执行了足够的计算

Block header



- 日志

每个日志实体包含：

- 记录器的账户地址。
- 代表本次交易执行的各种事件的一系列主题以及与这些事件相关的任何数据。

日志保存在了 **bloom 过滤器** 中，高效保存了无尽的日志数据。

- 交易收据

包含着日志信息的交易收据的根哈希值保存在头中。就像你在商店买东西时收到的收据一样，**以太坊为每笔交易都产生一个收据**。每个收据包含关于交易的特定信息：

- 区块序号
- 区块哈希值
- 交易哈希值
- 当前交易使用了的燃料
- 在当前交易执行完之后当前块使用的累计燃料
- 执行当前交易时创建的日志
- 等等

chatGPT- 以太坊为每笔交易都产生收据吗

以太坊为每笔交易都会产生收据。收据记录了交易的执行结果和相关信息，包括**合约地址**、**日志**和**状态变化**等。每个区块的交易收据会被**保存在状态树**中，以供其他交易查询和引用。

- 区块难度

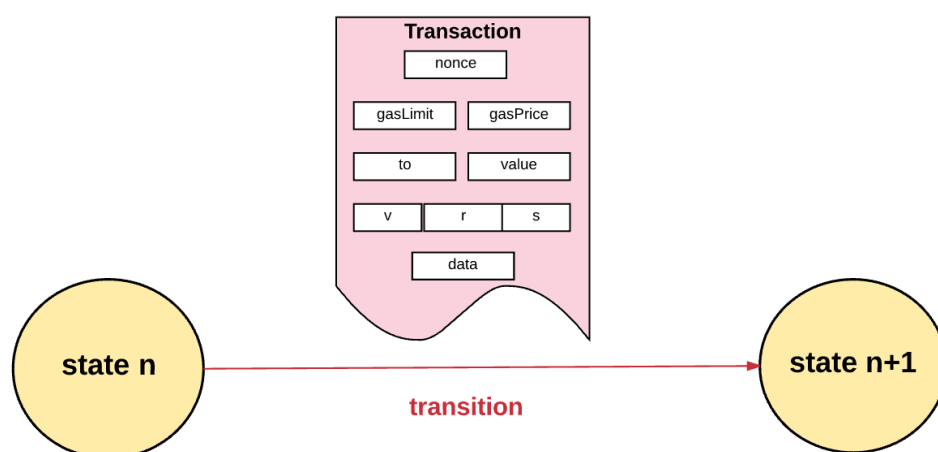
区块难度和**随机数 nonce** 之间的关系用数学形式表达就是：

$$n \leq \frac{2^{256}}{H_d}$$

Hd 为难度，找到符合难度阈值的随机数 nonce **唯一方法**就是使用**工作量证明算法**来列举所有的可能性。找到解决方案**预期时间与难度成正比**，通过调整区块难度，协议可以调整验证区块所需的时间。验证时间**自我调节**以保持恒定的速率——平均每 15 秒一个块。

交易的执行

假设你发送了一笔交易给以太坊网络处理，**将以太坊状态转换成包含你的交易**这个过程到底发生了什么？



1. 交易被执行的前提

一个交易可以被执行需要满足一系列基础要求：

- **交易必须是正确格式化的 RLP**。“RLP”代表 Recursive Length Prefix，它是一种数据格式，用来编码二进制数据嵌套数组。以太坊就是使用 RLP 格式序列化对象。
- **有效的交易签名**。
- **有效的交易序号**。该交易序号一定等于发送账户中的交易序号 nonce。
- **燃料充足**，交易的燃料限制一定要等于或者大于交易使用的内在燃料 intrinsic gas，它包括：

1. 执行交易预订费用为 21,000 个燃料
2. 随交易发送的数据的燃料费用（每字节数据或代码为 0 的费用为 4 个燃料，每个非零字节的数据或代码费用为 68 个燃料）
3. 如果是合约创建交易，还需要额外的 32,000 个燃料

$$\text{Intrinsic gas} = \begin{array}{|c|} \hline \text{Predefined gas fee} \\ \hline \mathbf{21,000} \\ \hline \end{array} + \begin{array}{|c|} \hline \text{Storage fee} \\ \hline \mathbf{4(X) + 68(Y)} \\ \hline \end{array} + \begin{array}{|c|} \hline \text{Contract creation} \\ \hline \mathbf{32,000} \\ \hline \end{array}$$

- **发送账户余额**必须有足够的以太币来支付“前期”燃料费用。前期燃料费用的计算比较简单：首先，交易的燃料限制乘以交易的燃料价格得到最大的燃料费用。然后，

这个最大的燃料费用加上从发送方传送给接收方的总值。

$$\text{Upfront cost} = \begin{array}{|c|} \hline \text{Gas Limit} \\ \hline 50,000 \\ \hline \end{array} \times \begin{array}{|c|} \hline \text{Gas Price} \\ \hline 20 \text{ gwei} \\ \hline \end{array} + \begin{array}{|c|} \hline \text{Value} \\ \hline 0.05 \text{ Ether} \\ \hline \end{array}$$

2. 执行步骤

一、从发送者的余额中扣除执行的**前期费用**，并为当前交易将发送者账户中的交易序号 **nonce 增加 1**。

此时，我们可以计算剩余的燃料，将交易的总燃料减去使用的**内在燃料**。

$$\text{Gas remaining} = \begin{array}{|c|} \hline \text{Gas Limit} \\ \hline 50,000 \\ \hline \end{array} - \begin{array}{|c|} \hline \text{Predefined gas fee} \\ \hline 21,000 \\ \hline \end{array} + \begin{array}{|c|} \hline \text{Storage fee} \\ \hline 4(X) + 68(Y) \\ \hline \end{array} + \begin{array}{|c|} \hline \text{Contract creation} \\ \hline 32,000 \\ \hline \end{array}$$

Intrinsic gas

二、开始**执行交易**。在交易执行的整个过程中，**以太坊保持跟踪“子状态”**。子状态是记录在交易中生成的信息的一种方式，当交易完成时会立即需要这些信息。具体来说，它包含：

自毁集：在交易完成之后会被丢弃的账户集（如果存在的话）

日志系列：虚拟机的代码执行的归档和可检索的检查点

退款余额：交易完成之后需要退还给发送账户的总额。回忆一下我们之前提到的以太坊中的存储需要付费，发送者要是清理了内存就会有退款。以太坊使用退款计数进行跟踪退款余额。退款计数从 0 开始并且每当合约删除了一些存储中的东西都会进行增加。

chatGPT-以太坊交易执行时跟踪的“子状态”是什么

以太坊交易执行时跟踪的“子状态”是在执行合约代码时，记录下每个合约地址的状态变化情况的一个**中间状态**。当一笔交易被执行时，它会涉及到多个合约的调用，这些合约可能会改变自己的状态，例如增加一些代币、更新一些数据等等。这些**状态变化需要被记录下来，以便在最终确定交易结果时进行验证和确认**。

以太坊的交易执行过程中，每个合约都有自己的子状态，其中包含了该合约执行所需的所有信息。这些信息包括了合约地址、合约账户中的余额、合约代码的哈希值、调用者地址、调用参数、返回值、日志信息等等。在交易执行过程中，每个合约的子状态都会被记录下来，并最终合并成一个整体的状态，成为**交易的最终状态**。

子状态的记录可以帮助在交易执行过程中发现**错误**，例如由于合约代码漏洞导致的资金丢失、数据损坏等问题。如果在交易执行过程中出现了错误，就可以根据记录的子状态信息进行**调试和追踪**，从而更好地解决问题。

三、交易所需的各种**计算**开始被处理。

当交易所需的步骤全部处理完成，并假设没有无效状态，通过确定退还给发送者的未使用的燃料量，最终的状态也被确定。除了未使用的燃料，发送者还会得到上面所说的“退款余额”中退还的一些津贴。

一旦发送者得到退款之后：

- **付工钱**：燃料的以太币就会给矿工
 - 交易使用的燃料会被**添加到区块的燃料计数**中(计数一直记录当前区块中所有交易使用的燃料总量，这对于验证区块时是非常有用的)
 - 所有在**自毁集中的账户**（如果存在的话）都会被删除
- 最终，我们就有了一个新的状态以及交易创建的一系列日志。

3. 合约账户创建

为了创建一个新的合约账户，我们使用一个特殊的公式来声明新账户的地址。然后我们使用下面的方法来**初始化**一个账户：

- 设置交易序号 **nonce 为 0**
- 如果发送者通过交易发送了一定量的**以太币**作为**值 value**，那么设置账户的余额为该值
- 将**存储设置为 0**
- 设置合约的**代码哈希 codeHash** 为一个**空字符串的哈希值**

一旦我们完成了账户的初始化，使用交易发送过来的初始化代码，实际上就**创造了一个账户**。初始化代码的执行过程是各种各样的。取决于合约的构造器，可能是更新账户的存储，也可能是创建另一个合约账户，或者发起另一个消息通信等等。

当初始化合约的代码被执行之后，会使用**燃料**。交易不允许使用的燃料超过剩余燃料。如果它使用的燃料超过剩余燃料，那么就会发生**燃料不足异常（OOG）**并退出。如果一个交易由于燃料不足异常而退出，那么**状态会立刻恢复到**交易前的一个点。发送者也不会获得在燃料用完之前所花费的燃料。不过，如果发送者随着交易发送了以太币，即使合约创建失败以太币也会被退回来。总的来说，合约创建失败的话，**燃料费用不退回，附带的以太币退回**。

如果初始化代码成功的执行完成，最后合约创建的花费会被**支付**。这些是**存储成本**，与创建的**合约代码大小成正比**。如果没有足够的剩余燃料来支付最后的花费，那么交易就会再次宣布**燃料不足异常（OOG）**并中断退出。

如果所有的都正常进行没有任何异常出现，那么任何剩余的未使用燃料都会被退回给原始的交易发送者，现在**改变的状态才被允许永久保存**。

4. 消息通信

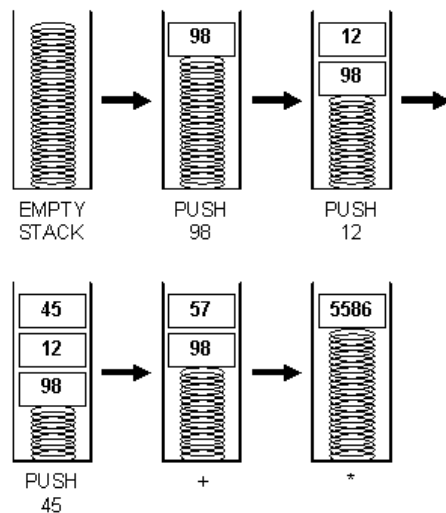
由于没有新账户被创建，所以消息通信的执行**不包含任何的初始化代码**。不过，它可以**包含输入数据**，如果交易发送者提供了此数据的话。一旦执行，消息通信同样会有一个额外的组件来包含输出数据，如果后续执行需要此数据的话组件就会被使用。

就像合约创建一样，如果消息通信执行退出是因为**燃料不足或交易无效**（例如栈溢出，无效跳转目的地或无效指令），那么已使用的燃料是不会被退回给原始触发者的。相反，所有剩余的未使用燃料也会被消耗掉，并且状态会被立刻重置为余额转移之前的那个点。

没有任何方法停止或恢复交易的执行而不让系统消耗你提供的所有燃料，直到最新的以太坊更新。例如，假设你编写了一个合约，当调用者没有授权来执行这些交易的时候抛出一个错误。在以太坊的前一个版本中，剩余的燃料也会被消耗掉，并且没有任何燃料退回给发送者。但是拜占庭更新包括了一个新的“恢复”代码，允许合约停止执行并且恢复改变的状态而不消耗剩余的燃料，此代码还拥有返回交易失败原因的能力。如果一个交易是由于恢复而退出，那么未使用的燃料就会被退回给发送者。

执行模式

交易究竟是如何在虚拟机（VM）中执行的？EVM 是图灵完备虚拟机器。EVM 存在而典型图灵完备机器不存在的唯一限制就是 EVM 本质上是被燃料束缚。因此，可以完成的计算总量本质上是被提供的燃料总量限制的。



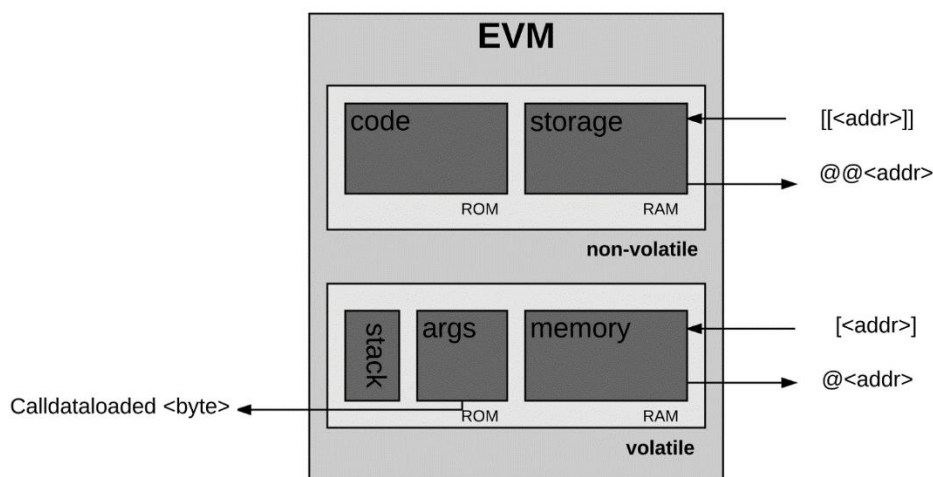
1. EVM 特点

EVM 具有基于堆栈的架构。堆栈机器就是使用后进先出来保存临时值的计算机。

EVM 中每个堆栈项的大小为 256 位，堆栈有一个最大的大小，为 1024 位。

EVM 有内存，各项按照可寻址字节数组来存储。内存是易失性的。“内存是易失性的”是指计算机的内存存在断电或关机时会丢失其中存储的数据。这是因为内存中存储数据的电荷会逐渐泄漏，所以需要定期刷新。如果没有电源供应，内存中存储的数据就会逐渐消失。与之相对的是非易失性存储器，如硬盘、固态硬盘和闪存等，这些存储器可以在断电或关机后继续保存数据。

EVM 也有一个存储器。不像内存，存储器是非易失性的，并作为系统状态的一部分进行维护。EVM 分开保存程序代码，在虚拟 ROM 中只能通过特殊指令来访问。EVM 不同于典型的冯·诺依曼架构，后者将程序的代码存储在内存或存储器中。



EVM 同样有属于它自己的语言：“**EVM 字节码**”，当一个程序员写一个在以太坊上运行的智能合约时，我们通常都是用高级语言例如 Solidity 来编写代码。然后我们可以将它编译成 EVM 可以理解的 EVM 字节码。

2. 执行计算的过程

- **判断**信息是否有效与可取

- 系统**状态**
- 用于计算的剩余**燃料**
- 拥有执行代码的账户**地址**
- 原始触发此次执行的交易**发送者的地址**
- 触发代码执行的账户地址（可能与原始发送者不同）
- 触发此次执行的交易**燃料价格**
- 此次执行的**输入数据**
- **值 Value**（单位为 Wei）作为当前执行的一部分传递给该账户
- 待执行的**机器码**
- 当前区块的**区块头**
- 当前消息通信或合约创建**堆栈的深度**

执行开始时，内存与堆栈都是空的，程序计数器为 0：

```
1. | PC: 0 STACK: [] MEM: [], STORAGE: {}
```

- EVM 递归地**执行**交易

为每个循环计算**系统状态**和**机器状态**。

系统状态也就是以太坊的全局状态 global state。

机器状态则包括：

- 可获取的燃料
- 程序计数器
- 内存的内容
- 内存中字的活跃数
- 堆栈的内容

堆栈中的项从系列的**最左边**被删除或者添加。每个循环，剩余的**燃料**都会被**减少**相应的量，**程序计数器**也会增加。在每个循环的结束，都有三种可能性：

- 机器到达异常状态（例如：燃料不足、无效指令、堆栈项不足、堆栈项会溢出 1024、无效的 JUMP/JUMPI 目的地等等）因此**停止并丢弃所有更改**
- 进入后续处理**下一个**循环
- 机器到达了**受控停止**（到达执行过程的终点）

假设执行没有遇到异常状态，达到一个“可控的”或正常的停止，机器就会**产生一个合成状态**，执行之后的**剩余燃料**、产生的**子状态**、以及**组合输出**。

3. 区块的生成

对于新块称为**确定**，对于已存在的块称为**验证**。

- 验证（确定）**ommer**，在区块头中的每个 ommer 都必须是有有效的并且必须在当前块往上 6 代之内。
- 验证（确定）**交易**。区块中的**已用燃料** gasUsed 数量必须与区块中所列交易使用的**累积燃料**量相等。（回忆一下，当执行一个交易的时候，我们会跟踪区块的燃料计数器，也就跟踪了区块中所有交易使用的燃料总数量）
- 申请**奖励**（只有挖矿时）**受益人的地址**会因为挖矿而获得 5 以太币（在以太坊 EIP-649 提案中，5ETH 很快将会被减少为 3ETH）。另外，对于每个 ommer，当前块的受益人会获得额外的 **1/32 当前块奖励金**的奖励。最近，每个 ommer 区块的受益人能够得到一定量的奖励。
- 校验（如果是挖矿的话，就是计算一个有效的）**状态和序号**。确保所有的交易和改变的结果状态都被应用了，然后在区块奖励被应用于最终交易结果状态之后**定义一个新块为状态**。通过检查最终状态与存储在头中的状态树来进行验证。

工作量证明机制

工作量证明是 POW，以太坊的工作量证明用的是“**Ethash**”

$$(m, n) = \text{PoW}(H_m, H_n, d)$$

m 代表混合哈希 (maxhash), n 代表序号 (nonce)，第一个代表的是新区块的头（不包含需要计算的序号 nonce 和混合哈希 mixHash），Hn 是区块头的序号 nonce, d 是 DAG，就是一个大数据集。

“混合哈希”和“随机数”组合时，证明此区块已经执行了足够的计算。PoW 函数即用来估算这两项。

实际的 PoW 函数比较复杂，简单来说，会为每个区块计算一个“种子”。每个“时期”的种子都不一样，每个时期是 30,000 个区块长度。对于第一时期，种子就是 32 位 0 的哈希值。对于后续的每个时期，种子就是前一个种子哈希值的哈希值。使用这个种子，节点可以计算出一个**伪随机“缓存”**。

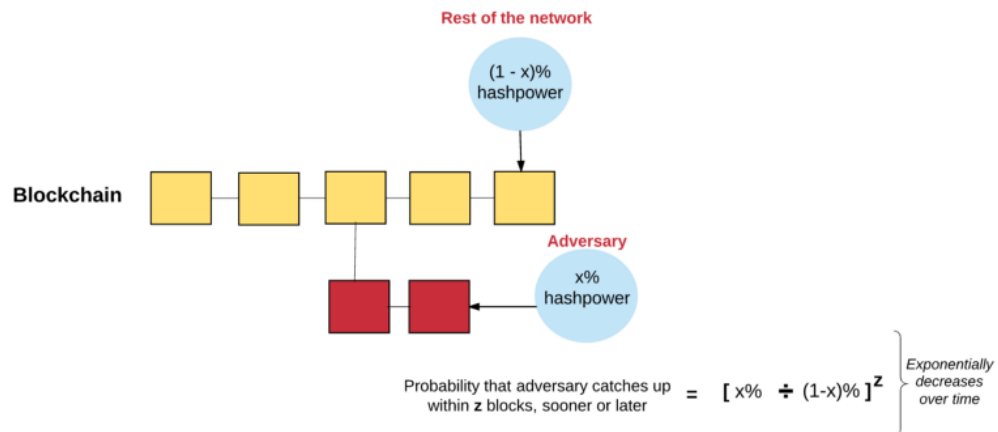
这个缓存使得“**轻节点**”成为现实，一个轻节点可以仅基于缓存来校验一个交易的有效性。而不用存储整个区块链的数据集。

为了成为**矿工**，你需要生成全数据集，所有全客户端和矿工都保存这个数据集，并且这个数据集随着时间**线性增长**。然后矿工可以**随机抽取数据集的部分**并将它们放入一个数学函数中哈希出一个混合哈希 mixHash。矿工会重复生成混合哈希 mixHash 直到输出的值

小于想要的目标值随机数 nonce。当输出的值符合这个条件的时候，随机数 nonce 就被认为是有效的，然后区块就被添加到链中。

1. 挖矿成为了安全机制

PoW 算法确保了特定区块链的权威性，使得攻击者的重写区块或保持分叉十分困难。除非超过网络一半的挖矿能力（51%攻击）。



2. 挖矿提供了财富分配机制

对矿工而言，奖励包括：

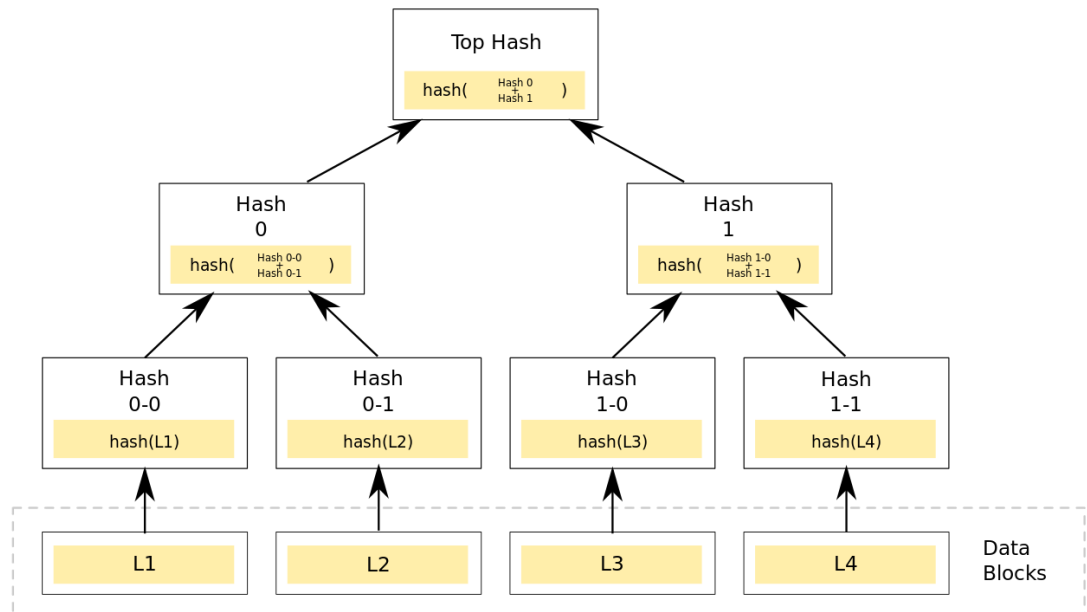
- 出块奖励
- 区块中的交易在区块内所消耗的燃料
- 纳入 ommers 作为区块的一部分的额外奖励

同时，以太坊尽可能地允许更多人访问，并降低任何单个节点能够创造与其不成比例的利润的可能性，以完成去中心化，提高安全性。

Merkle tree 算法解析

<https://blog.csdn.net/wo541075754/article/details/54632929>

原理

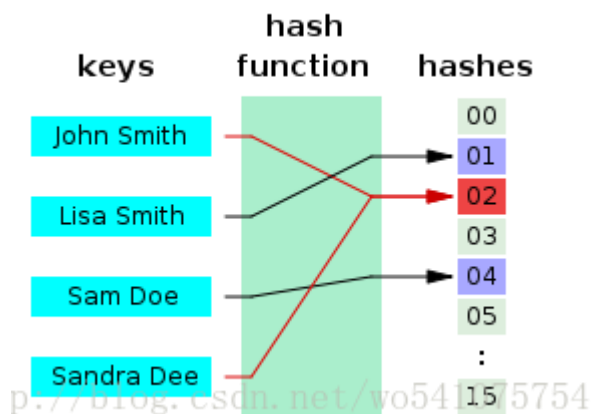


<http://blog.csdn.net/wo541075754>

Merkle Tree 也称 Hash Tree，是存储 hash 值的一棵树。其叶子对应数据块（文件或文件的集合）的 hash 值，非叶子节点对应其串联的字符串的 hash 值。

1. Hash

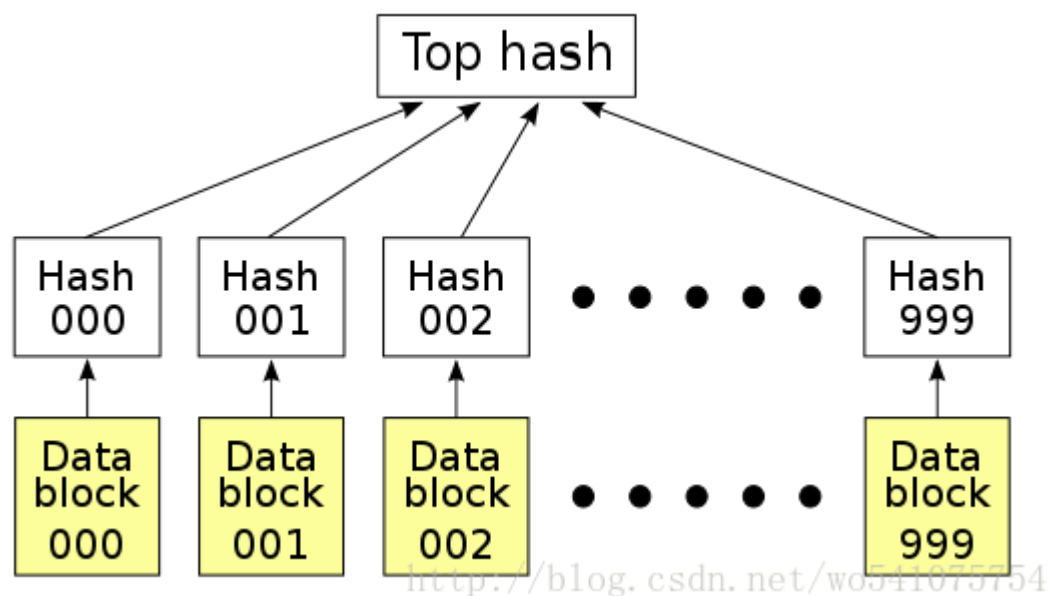
Hash 是一个把任意长度的数据映射成固定长度数据的函数。常用于数据完整性校验，且根据 Hash 值反推原始数据的特征是极其困难的。



2. HashList

在点对点网络中作数据传输的时候，会同时从多个机器上下载数据，而且很多机器可以认为是不稳定或者不可信的。为了校验数据的完整性，更好的办法是把大的文件分割成小的数据块（例如，把分割成 2K 为单元的数据块）。这样的好处是，如果小块数据在传输过程中损坏了，那么只要重新下载这一快数据就行了，不用重新下载整个文件。

怎么确定小的数据块没有损坏哪？只需要为每个数据块做 Hash。BT 下载的时候，在下载真正数据之前，我们会先下载一个 Hash 列表。那么问题又来了，怎么确定这个 Hash 列表本事是正确的哪？答案是把每个小块数据的 Hash 值拼到一起，然后对这个长字符串在作一次 Hash 运算，这样就得到 Hash 列表的根 Hash (Top Hash or Root Hash)。下载数据的时候，首先从可信的数据源得到正确的根 Hash，就可以用它来校验 Hash 列表了，然后通过校验后的 Hash 列表校验数据块。



3. Merkle Tree

更进一步，Merkle Tree 可以看做 Hash List 的泛化 (Hash List 可以看作一种特殊的 Merkle Tree，即树高为 2 的多叉 Merkle Tree)。

最底层是将数据分成小的数据块，对应相应的哈希。向上层走时，并不直接计算根哈希，而是将相邻的两个哈希拼接成一个字符串后计算字符串的哈希。出现单数 hash 时就直接 hash 运算。最终形成一个根哈希 Merkle Root。

在 p2p 网络下载网络之前，先从可信的源获得文件的 Merkle Tree 树根。一旦获得了树根，就可以从其他从不可信的源获取 Merkle tree。通过可信的树根来检查接受到的 Merkle Tree。如果 Merkle Tree 是损坏的或者虚假的，就从其他源获得另一个 Merkle Tree，直到获得一个与可信树根匹配的 Merkle Tree。

Merkle Tree 和 Hash List 的主要区别是，可以直接下载并立即验证 Merkle Tree 的一个分支。因为可以将文件切分成小的数据块，这样如果有一块数据损坏，仅仅重新下载这个数据块就行了。如果文件非常大，那么 Merkle tree 和 Hash list 都很大，但是 Merkle tree 可以一次下载一个分支，然后立即验证这个分支，如果分支验证通过，就可以下载数据了。而 Hash list 只有下载整个 hash list 才能验证。

Merkle Tree 的特点

- MT 是一种树，大多数是二叉树，也可以多叉树，具有树结构的所有特点；
- Merkle Tree 的叶子节点的 value 是数据集合的单元数据或者单元数据 HASH。
- 非叶子节点的 value 是根据它下面所有的叶子节点值 Hash 得出的。

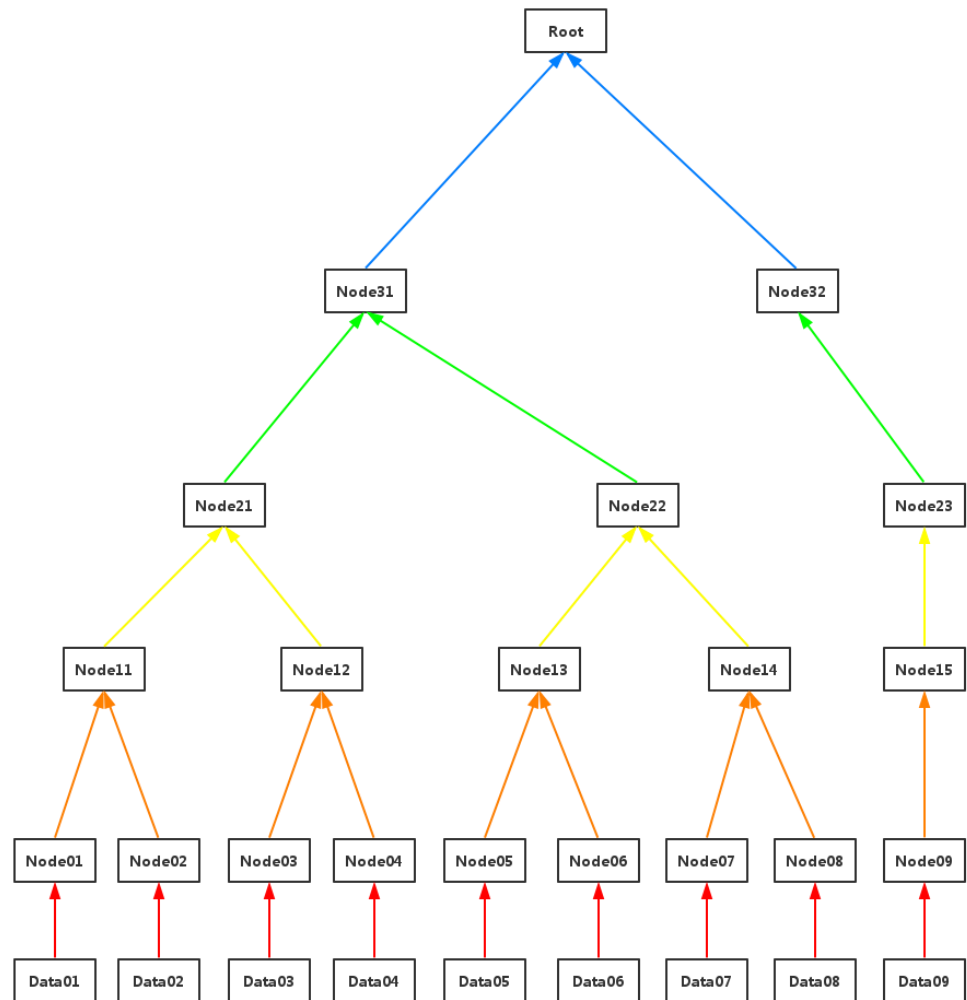
加密的 Hash 可以选 SHA-256 或 MD5 等，但对安全性较低的场合也可以选择高效的算法如 CRC。

1. Second Preimage Attack:

Merkle tree 的树根并不表示树的深度，这可能会导致**第二原像攻击** (second-preimage attack)，即攻击者创建一个具有**相同 Merkle 树根的虚假文档**。一个简单的解决方法在 Certificate Transparency 中定义：当计算叶节点的 hash 时，在 hash 数据前加 0x00。当计算内部节点是，在前面加 0x01。另外一些实现限制 hash tree 的根，通过在 hash 值前面加深度前缀。因此，前缀每一步会减少，只有当到达叶子时前缀依然为正，提取的 hash 链才被定义为有效。

Merkle Tree 的操作

1. 创建 Merkle Tree



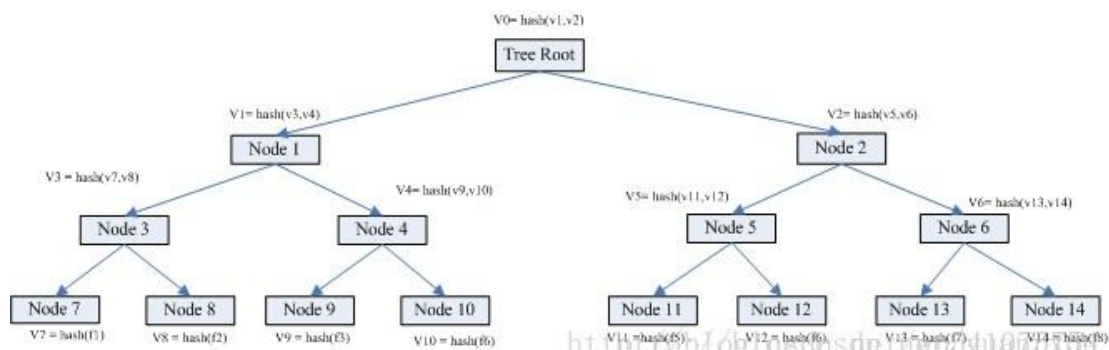
<http://blog.csdn.net/wo541075754>

注: 非叶节点的 hash 运算是两个子节点进行**拼接** (concatenation) 而非相加 (addition)。

创建 Merkle Tree 是 $O(n)$ 复杂度 (这里指 $O(n)$ 次 hash 运算), n 是数据块的大小。得到 Merkle Tree 的树高是 $\log(n)+1$ 。

2. 检索数据块

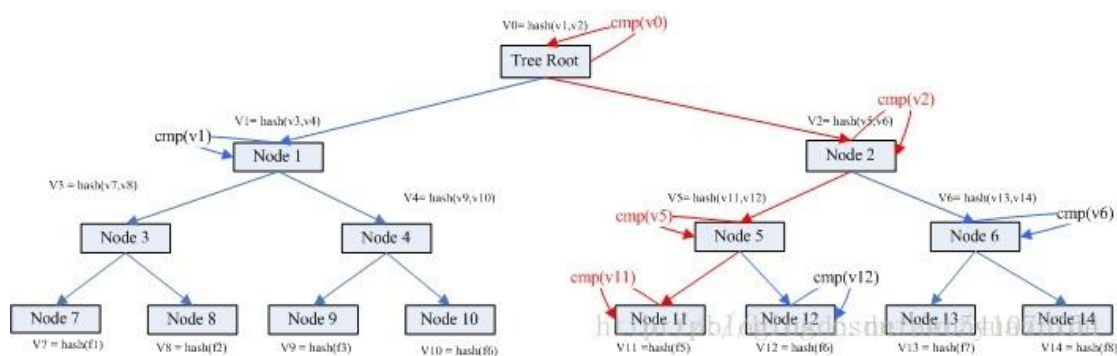
假设有 A 和 B 两台机器, A 需要与 B 相同目录下有 8 个文件, 文件分别是 f1 f2 f3 ... f8。这个时候我们就可以通过 Merkle Tree 来进行快速比较。假设我们在文件创建的时候每个机器都构建了一个 Merkle Tree。具体如下图:



假如 A 上的文件 5 与 B 上的不一样。我们怎么通过两个机器的 merkle tree 信息找到不相同的文件？这个比较检索过程如下：

- Step1. 首先比较 v_0 是否相同,如果不同，检索其孩子 node1 和 node2.
- Step2. v_1 相同， v_2 不同。检索 node2 的孩子 node5 node6;
- Step3. v_5 不同， v_6 相同，检索比较 node5 的孩子 node 11 和 node 12
- Step4. v_{11} 不同， v_{12} 相同。node 11 为叶子节点，获取其目录信息。
- Step5. 检索比较完毕。

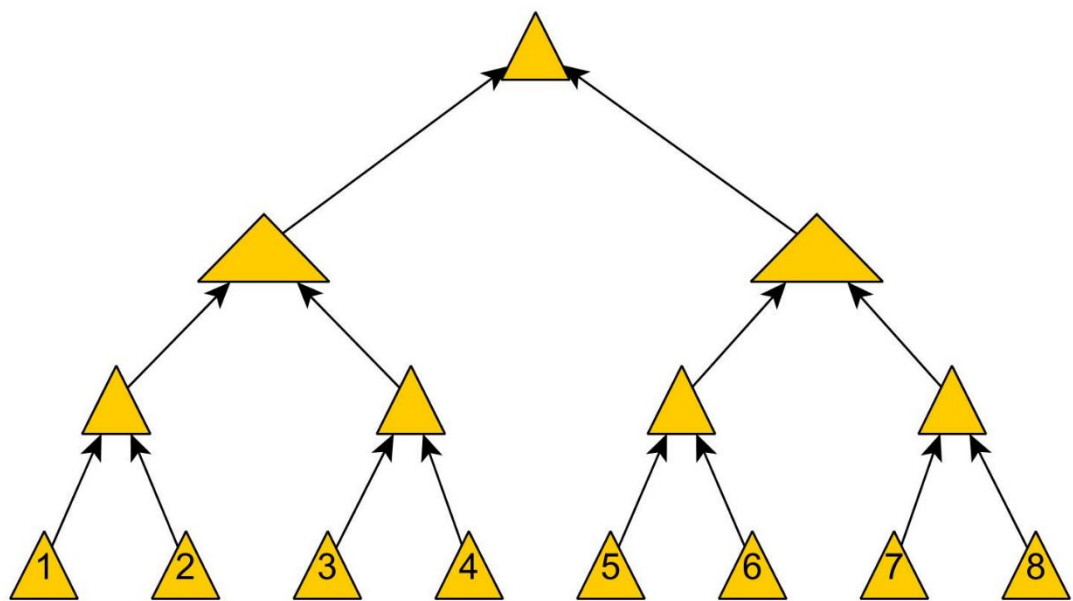
以上过程的理论复杂度是 $\text{Log}(N)$ 。过程描述图如下：



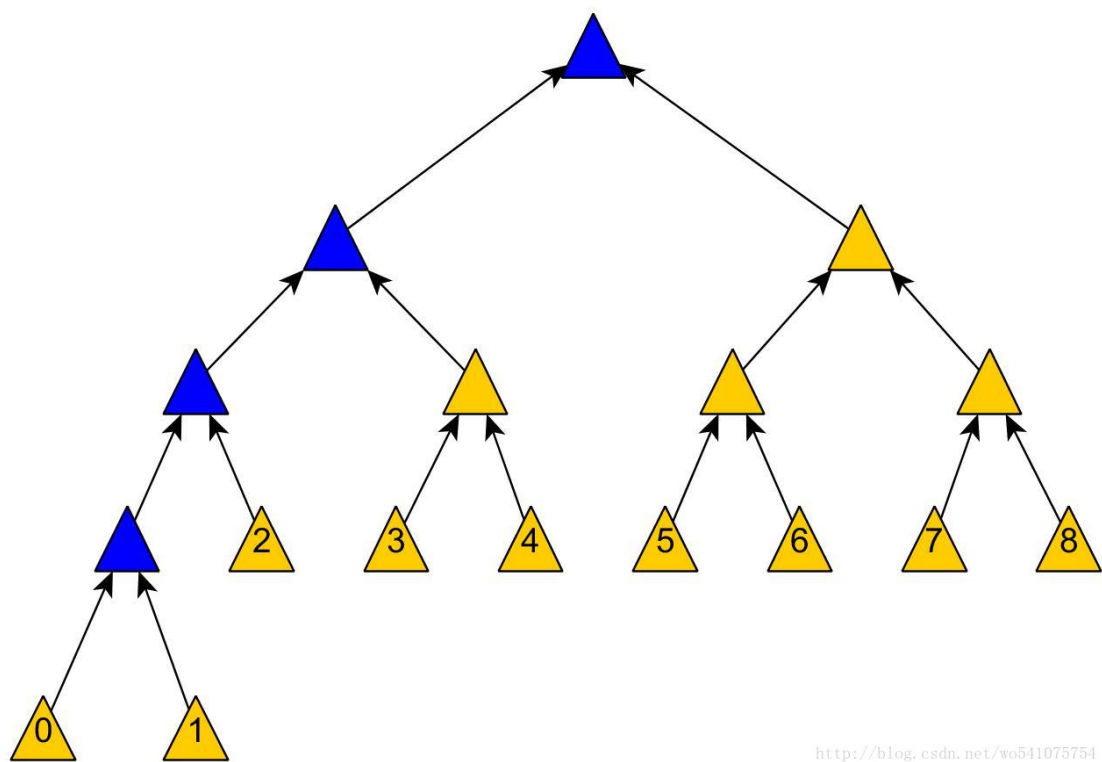
3. 更新、插入与删除

更新 MT 时，只需重新计算其到树根路径上的 Hash 值即可，并不会改变 MT 的结构。

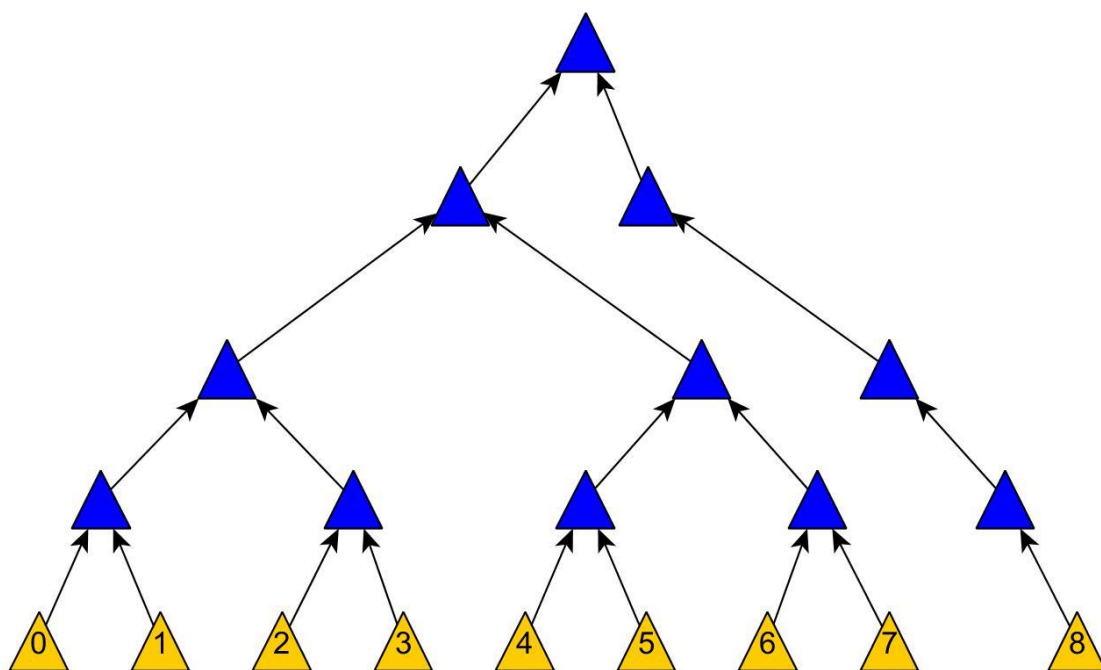
而对于插入与删除，就结论来看，Merkle Tree 的**结构**(是否平衡，树高限制多少)在大多数应用中并不重要，而且保持数据块的**顺序**也在大多数应用中也不需要。Merkle Tree 的插入和删除操作其实是一个工程上的问题，不同问题会有不同的插入方法。因此，可以根据具体应用的情况，设计自己的插入和删除操作。一个通用的 Merkle Tree 插入删除操作是**没有意义的**。以下是两种插入方法的图示：



<http://blog.csdn.net/wo541075754>



<http://blog.csdn.net/wo541075754>



<http://blog.csdn.net/wo541075754>

Merkle Tree 的应用

1. 数字签名

一种高效的数字签名框架：Merkle Signature Scheme。

2. P2P 网络

在 P2P 网络中, Merkle Tree 用来确保从其他节点接受的数据块没有损坏且没有被替换, 甚至检查其他节点不会欺骗或者发布虚假的块。大家所熟悉的 BT 下载就是采用了 P2P 技术来让客户端之间进行数据传输, 一来可以加快数据下载速度, 二来减轻下载服务器的负担。BT 即 **BitTorrent**, 是一种**中心索引式的 P2P 文件分析通信协议**。

要进下载必须从中心索引服务器获取一个扩展名为 torrent 的索引文件 (即大家所说的种子), torrent 文件包含了要共享文件的信息, 包括文件名, 大小, 文件的 Hash 信息和一个指向 Tracker 的 URL。Torrent 文件中的 Hash 信息是每一块要下载的文件内容的加密摘要, 这些摘要也可运行在下载的时候进行验证。大的 torrent 文件是 Web 服务器的瓶颈, 而且也不能直接被包含在 RSS 或 gossiped around(用流言传播协议进行传播)。一个相关的问题是大数据块的使用, 因为为了保持 torrent 文件的非常小, 那么数据块 Hash 的数量也得很小, 这就意味着每个数据块相对较大。大数据块影响节点之间进行交易的效率, 因为只有当大数据块全部下载下来并校验通过后, 才能与其他节点进行交易。

3. 可信计算 Trusted Computing

可信计算是可信计算组为分布式计算环境中参与节点的计算平台提供端点可信性而提出的。可信计算技术在计算平台的硬件层引入**可信平台模块**(Trusted Platform, TPM), 实际上为计算平台提供了基于硬件的**可信根**(Root of trust, RoT)。从可信根出发, 使用信任链传

递机制,可信计算技术可对本地平台的硬件及软件实施逐层的完整性度量,并将度量结果可靠地保存在 TPM 的**平台配置寄存器**(Platform configuration register, PCR)中,此后远程计算平台可通过**远程验证机制**(Remote Attestation)比对本地 PCR 中度量结果,从而验证本地计算平台的可信性。可信计算技术让分布式应用的参与节点摆脱了对中心服务器的依赖,而直接通过用户机器上的 TPM 芯片来建立信任,使得创建扩展性更好、可靠性更高、可用性更强的安全分布式应用成为可能。可信计算技术的核心机制是远程验证(remote attestation),分布式应用的参与结点正是通过远程验证机制来建立互信,从而保障应用的安全。

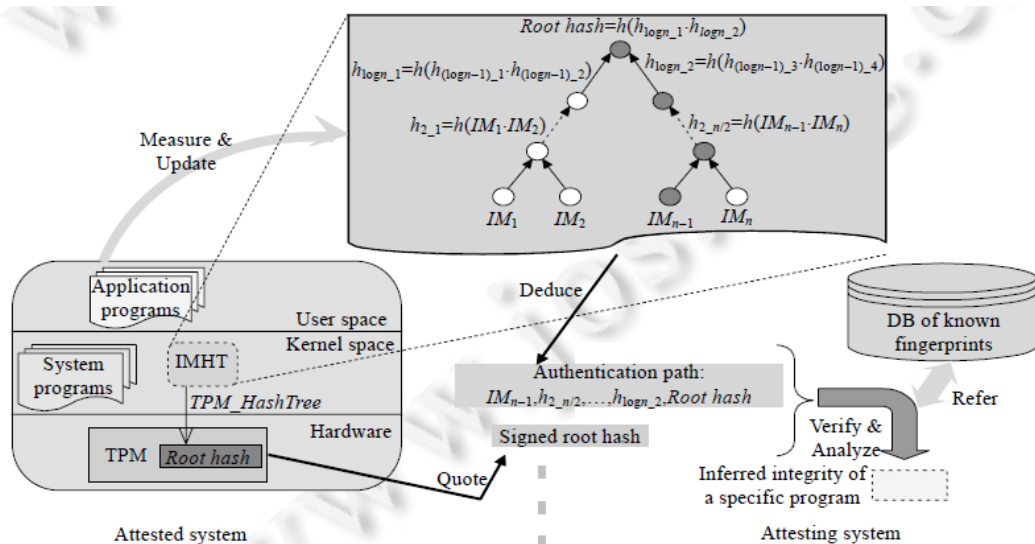


Fig.3 Architecture of remote attestation mechanism based on Merkle hash tree

图3 基于Merkle哈希树的远程验证机制的体系架构 [csdn.net/wo541075754](https://www.csdn.net/wo541075754)

图示一种基于Merkle Tree的远程验证机制,核心是完整性度量值的哈希树。

首先,RAMT 在内核中维护的不再是一张完整性度量值列表(ML),而是一棵**完整性度量值哈希树**(integrity measurement hash tree,简称IMHT)。其中,IMHT的叶子结点存储的数据对象是待验证计算平台上被度量的各种程序的完整性哈希值,而其内部结点则依据Merkle哈希树的构建规则由子结点的连接的哈希值动态生成。

其次,为了维护IMHT叶子结点的完整性,RAMT需要使用TPM中的一段存储器来保存IMHT可信根哈希的值。

再次,RAMT的完整性验证过程基于认证路径(authentication path)实施。认证路径是指IMHT上从待验证叶子结点到根哈希的路径。

4. 星际文件系统 IPFS(InterPlanetary File System)

IPFS是众多超强的互联网技术的综合体,如**分布式哈希表DHT**(Distributed Hash Table),**Git版本控制系统**,**Bittorrent**等。它创建了一个P2P的集群,这个集群允许IPFS对象的交换。全部的IPFS对象形成了一个被称作**Merkle DAG**的加密认证数据结构。

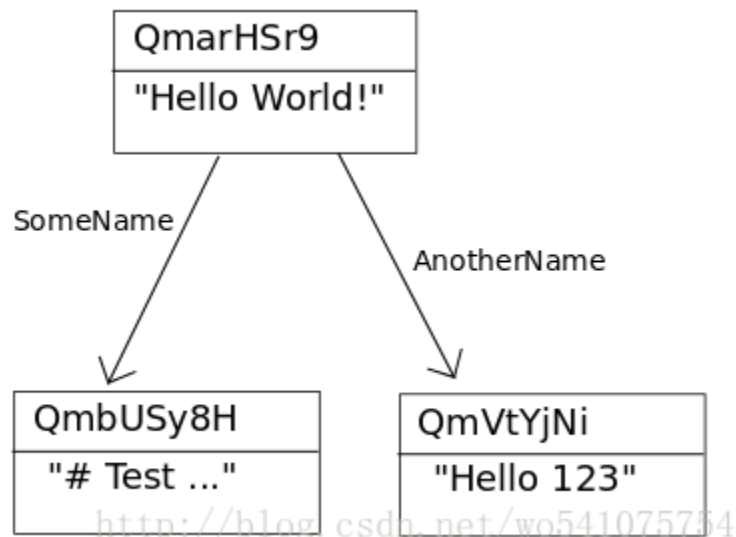
IPFS对象是一个含有两个域的数据结构:

- **Data** – 非结构的二进制数据,小于256kB
 - **Links** – 一个Link数据结构的数组。IPFS对象通过他们链接到其他对象
- Link数据结构包含三个域:
- **Name** – Link的名字
 - **Hash** – Link链接到对象的Hash
 - **Size** – Link链接到对象的累积大小,包括它的Links

```
> ipfs object get QmarHSr9aSNaPSR6G9KFPbuLV9aEqJfTk1y9B8pdwqK4Rq
{
  "Links": [
    {
      "Name": "AnotherName",
      "Hash": "QmVtYjNi3KeyGmcgg7yVXWskLaBtov3UYL9pgcGK3MCWu",
      "Size": 18
    },
    {
      "Name": "SomeName",
      "Hash": "QmbUSy8HCn8J4TMDRRdxCbK2uCCtkQyZtY6XYv3y7kLgDC",
      "Size": 58
    }
  ],
  "Data": "Hello World!"
}
```

<http://blog.csdn.net/wo541075754>

这样，IPFS 的集合组成了一个 **Merkle DAG**（有向无环图）。



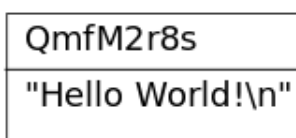
<http://blog.csdn.net/wo541075754>

而没有 Links 的 IPFS 对象是这样的：

Viewing the underlying structure with `ipfs object get` yields:

```
chris@chris-VBox:~/tmp$ ipfs object get
QmfM2r8seH2GiRaC4esTjeraXEachRt8ZsSeGaWTPLyMoG
{
  "Links": [],
  "Data": "\u0008\u0002\u0012\rHello World!\n\u0018\r"
}
```

We visualize this file as follows:



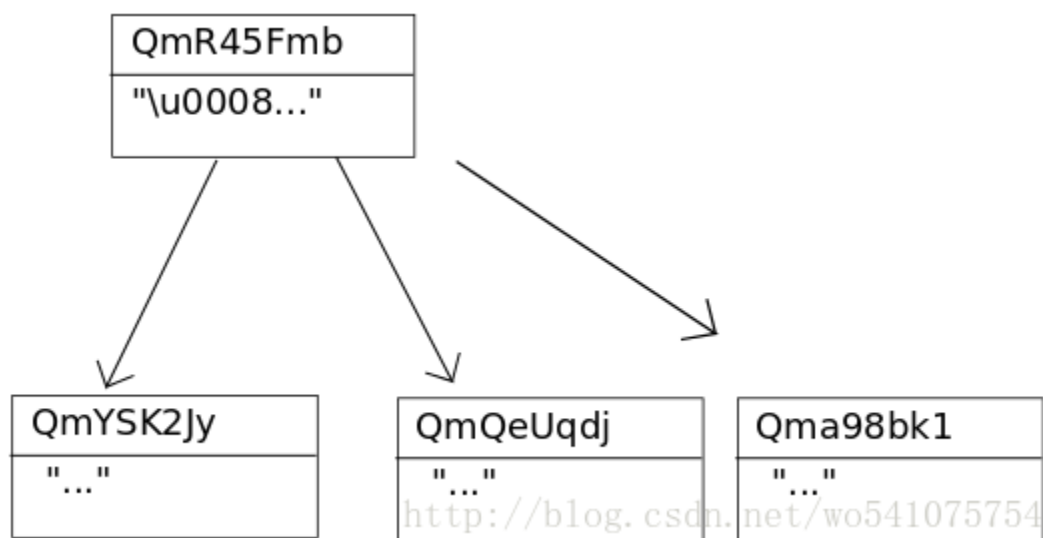
<http://blog.csdn.net/wo541075754>

对于大文件，由于 256KB 的限制，故只能用最小的 Data 对象来代表大文件，且 Links 的名字都为空字符串。

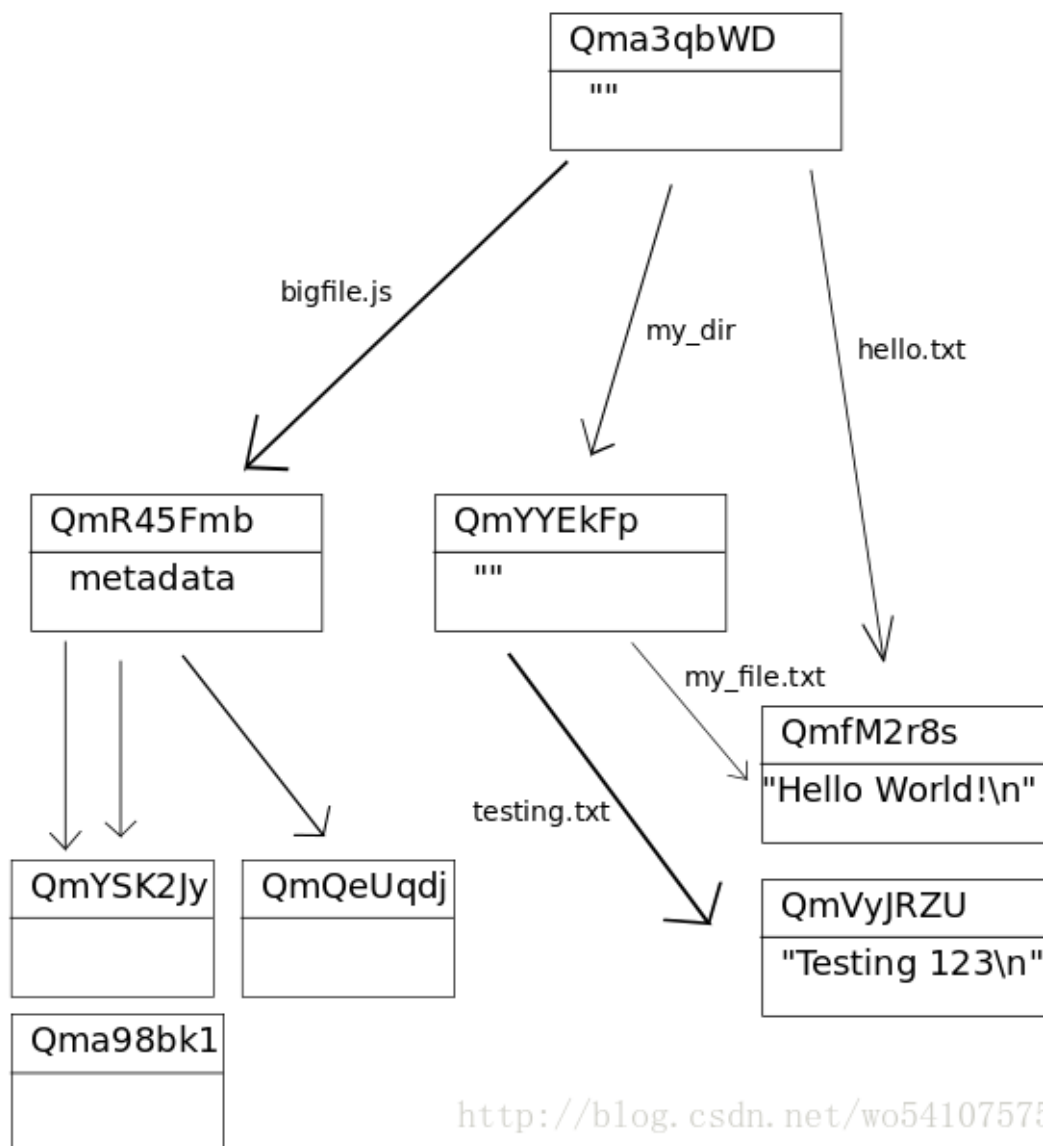
```
chris@chris-VBox:~/tmp$ ipfs add test_dir/bigfile.js
added QmR45FmbVVrixReBwJkhEKde2qwHYaQzGxu4ZoDeswuF9w test_dir/bigfile.js
chris@chris-VBox:~/tmp$ ipfs object get
QmR45FmbVVrixReBwJkhEKde2qwHYaQzGxu4ZoDeswuF9w
{
  "Links": [
    {
      "Name": "",
      "Hash": "QmYSK2JyM3RyDyB52caZCTKFR3HKniEcMnNJYdk8DQ6KKB",
      "Size": 262158
    },
    {
      "Name": "",
      "Hash": "QmQeUqdjFmaxuJewStqCLUoKrR9khqb4Edw9TfRQQdfWz3",
      "Size": 262158
    },
    {
      "Name": "",
      "Hash": "Qma98bk1hj1RZDTmYmfiUXDj8hXXt7uGA5roU5mfUb3sVG",
      "Size": 178947
    }
  ],
  "Data": "\u0008\u0002\u0018* \u0010 \u0010 \n"
}
```

<http://blog.csdn.net/wo541075754>

结构图为：



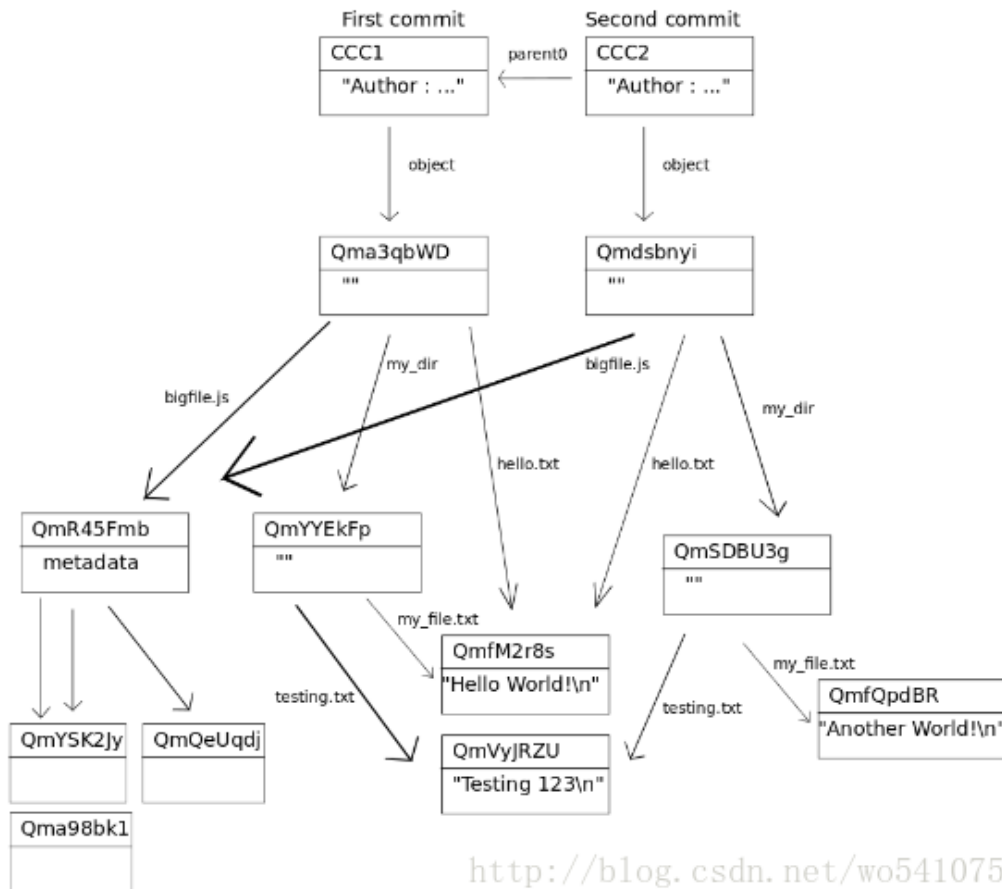
还有一种特殊的结构，**目录结构**：目录是没有数据的 IPFS 对象，它的链接指向其包含的文件和目录。



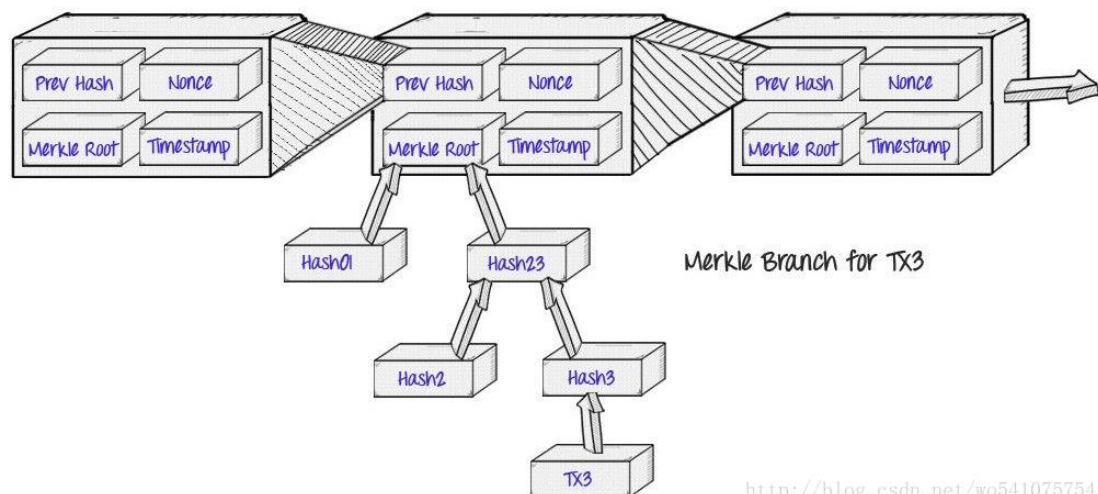
<http://blog.csdn.net/wo541075754>

IPFS 可以表示 **Git** 使用的数据结构，Git commit object。Commit Object 主要的特点是他有一个或多个名为'parent0'和'parent1'等的链接（这些链接指向前一个版本），以及一个名为 object 的对象(在 Git 中成为 tree)指向引用这个 commit 的文件系统结构。

We give as an example our previous file system directory structure, along with two commits: The first commit is the original structure, and in the second commit we've updated the file `my_file.txt` to say `Another World!` instead of the original `Hello World!`.



5. 比特币 BitCoin



Bitcoin 的 Blockchain 利用 Merkle proofs 来存储每个区块的交易。好处就是中本聪描述到的“**简化支付验证**” (Simplified Payment Verification, SPV) 的概念:一个“**轻客户端**” (light client) 可以仅下载链的区块头即每个区块中的 80byte 的数据块, 仅包含五个元素, 而不是下载每一笔交易以及每一个区块。

- 上一区块头的哈希值
- 时间戳
- 挖矿难度值
- 工作量证明随机数 (nonce)
- 包含该区块交易的 Merkle Tree 的根哈希

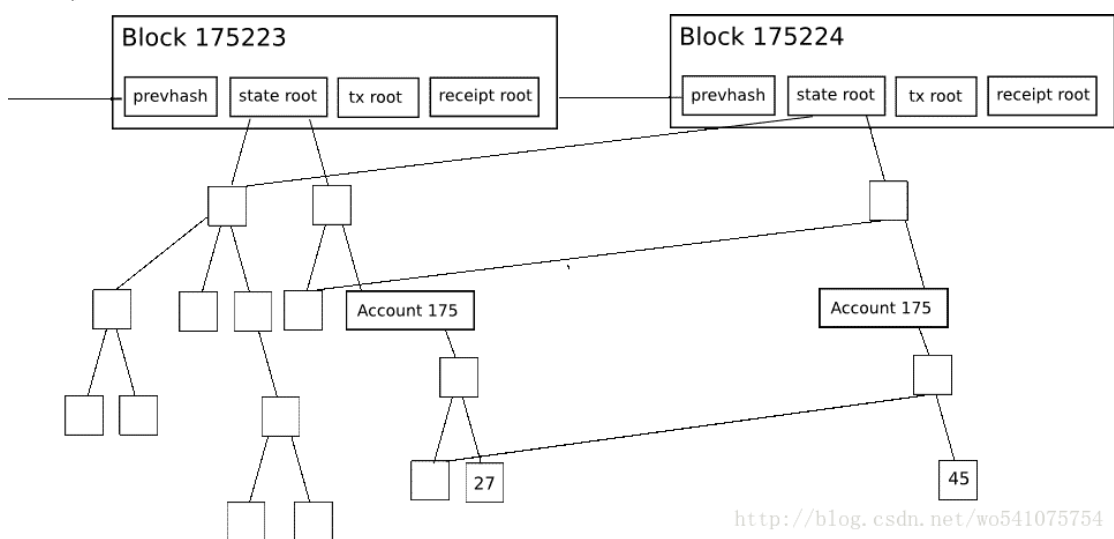
如果客户端想要确认一个交易的状态, 它只需简单的发起一个 **Merkle proof 请求**, 这个请求显示出这个特定的交易在 Merkle trees 的一个之中, 而且这个 Merkle Tree 的树根在主链的一个区块头中。

尽管它可以证明包含的交易, 但是它不能进行涉及**当前状态**的证明(如数字资产的持有, 名称注册, 金融合约的状态等) 这引出了更加完善的以太坊。

6. 以太坊 Ethereum

以太坊的区块头不只有一个 Merkle 树, 而是为**三种对象**设计的三棵树:

- 交易 Transaction
- 收据 Receipts(本质上是显示每个交易影响的多块数据)
- 状态 State



<http://blog.csdn.net/wo541075754>

这样做的好处, 使**轻客户端**可以轻松查询以下类型的答案:

- 这笔交易被包含在特定的区块中了么?
- 告诉我这个地址在过去 30 天中, 发出 X 类型事件的所有实例 (例如, 一个众筹合约完成了它的目标)
- 目前我的账户余额是多少?
- 这个账户是否存在?
- 假如在这个合约中运行这笔交易, 它的输出会是什么?

第一种是由**交易树** (transaction tree) 来处理的; 第三和第四种则是由**状态树** (state tree) 负责处理, 第二种则由**收据树** (receipt tree) 处理。计算前四个查询任务是相当简单的。服务器简单地找到对象, 获取 Merkle 分支, 并通过分支来回轻客户端。

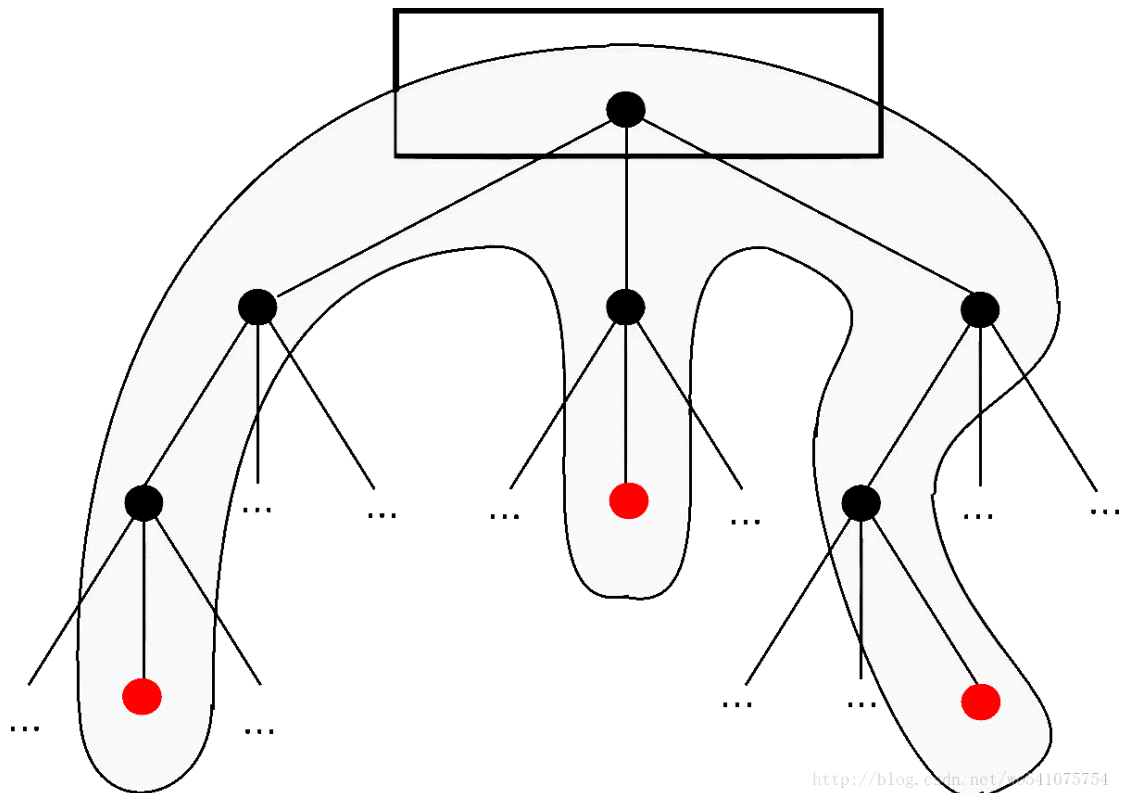
第五种查询任务同样也是由状态树处理, 但它的计算方式会比较复杂。这里, 我们需要

构建一个 **Merkle 状态转变证明** (Merkle state transition proof)。从本质上来讲，这样的证明也就是在说“如果你在根 S 的状态树上运行交易 T，其结果状态树将是根为 S'，log 为 L，输出为 O”（“输出”作为存在于以太坊的一种概念，因为每一笔交易都是一个函数调用；它在理论上并不是必要的）。

为了推断这个证明，服务器在本地创建了一个假的区块，将状态设为 S，并在请求这笔交易时假装是一个轻客户端。也就是说，如果请求这笔交易的过程，需要客户端确定一个账户的余额，这个轻客户端(由服务器模拟的)会发出一个余额查询请求。如果需要轻客户端在特点某个合约的存储中查询特定的条目，这个轻客户端就会发出这样的请求。也就是说服务器(通过模拟一个轻客户端)正确回应所有自己的请求，但服务器也会跟踪它所有发回的数据。

然后，服务器从上述的这些请求中把数据合并并把数据以一个证明的方式发送给客户端。

然后，客户端会进行相同的步骤，但会将服务器提供的证明作为一个数据库来使用。如果客户端进行步骤的结果和服务器提供的是一样的话，客户端就接受这个证明。



7. MPT(Merkle Patricia Trees)

最为简单的一种 Merkle Tree 大多数情况下都是一棵二叉树。然而，Ethereum 所使用的 Merkle Tree 则更为复杂，我们称之为“**梅克尔.帕特里夏树**” (Merkle Patricia tree)。

对于验证属于 **list 格式**（本质上来讲，它就是一系列前后相连的数据块）的信息而言，二叉 Merkle Tree 是非常好的数据结构。对于交易树来说，它们也同样是不错的，因为一旦树已经建立，花多少时间来编辑这棵树并不重要，树一旦建立了，它就会永远存在并且不会改变。但是，对于**状态树**，情况会更复杂些。以太坊中的状态树基本上包含了一个**键值映射**，其中的键是地址，而**值**包括账户的声明、余额、随机数 nonce、代码以及每一个账户的存储（其中存储本身就是一颗树）。例如，摩登测试网络（the Morden testnet ）的创始状态如下所示：

- 树的**深度是有限制的**，即使考虑攻击者会故意地制造一些交易，使得这颗树尽可能地深。不然，攻击者可以通过操纵树的深度，执行拒绝服务攻击（DOS attack），使得更新变得极其缓慢。
- 树的根只取决于数据，**和其中的更新顺序无关**。换个顺序进行更新，甚至重新从头计算树，并不会改变根。

实践中也可以对稀疏树有额外的优化。