Ethereum 研究报告

一、 以太坊简介

1、什么是以太坊

以太坊（Ethereum）是一个开源的具有智能合约（smart contract）功能的公共区块链平台。通过其专用的加密货币以太币（Ether）提供去中心化的虚拟机（Ethereum virtual machine）来处理点对点合约。在2013年至2014年间，以太坊的概念首次由Vitalik Buterin提出，收到比特币的启发，其意在建立“下一代加密货币与去中心化的应用平台”，并在2014年通过ICO众筹得以开始发展。

以太坊是将比特币中的技术和概念应用到计算机领域的一项新技术。比特币被广泛理解为一个系统,该系统维护了一个能安全记录比特币余额的共享世界总账本。以太坊采用了比特币中许多相同的技术原理(如区块链和P2P网络)，来维护一个共享的世界计算平台，该平台可以安全灵活地运行用户编码的任何应用程序。

中本聪在2009年发布了比特币，比特币的诞生在当时被誉为货币的激进发展。作为数字资产的首个典型代表，比特币没有任何人给它背书，也没有任何“内在价值”，甚至没有一个中心化的发行者和控制者。但是区块链技术目前已经得到了越来越广泛的关注。如：利用区块链数字资产来替代自定义的货币和金融工具（“彩色硬币”）、底层物理设备（“智能财产”）的所有权，不可互换的资产，如域名（“Namecoin”）以及更复杂的应用程序。

在以太坊之前，已经有很多基于区块链技术的应用项目。但是，它们都有一定的局限性，仅能支持一种或几种特定的应用。而以太坊之所以能超越这些局限性，是因为其核心思想：我们可以拥有一个嵌入式编程语言的区块链协议，而不是许多区块链协议，每个协议支持几个应用程序，甚至一个区块链协议，允许任何应用程序被写在最上面，而其规则由区块链强制执行。这样，协议不仅可以支持已经开发的所有应用程序，而且在未来也可以支持人们还不曾想到过的区块链应用。

比特币经常被称为“全球账簿”，尽管这个账簿只限于记录某一特定货币的全部交易记录。但是以太坊可以被视为 “世界计算机”：任何人都可以上传和执行应用程序，并且程序的有效执行都能得到保障。这种保障依赖于以太坊系统的去中心化特性以及由全球成千上万的计算机组成的高度稳健的共识网络。以太坊使用区块链技术作为基础，同时使用密码学和经济激励来保证其计算的安全性。此外，它支持智能合约，使得以太坊得以开启更大的可能性。

举一个具体的例子,试想使用一个基于以太坊的物联网平台Slock来提供自行车租赁服务的场景。首先车主都会将Slock（智能锁）安装在自己的自行车,并向Ethereum区块链给自己的自行车注册一个智能合约(一种计算机程序)。接下来，任何人都可以向智能合约发起一个发送一定数量的加密货币的请求（交易），合约在接收到请求之后，会自动将这笔数字货币转发给车主，并记录一个状态，该状态用于表明刚刚这位数字货币

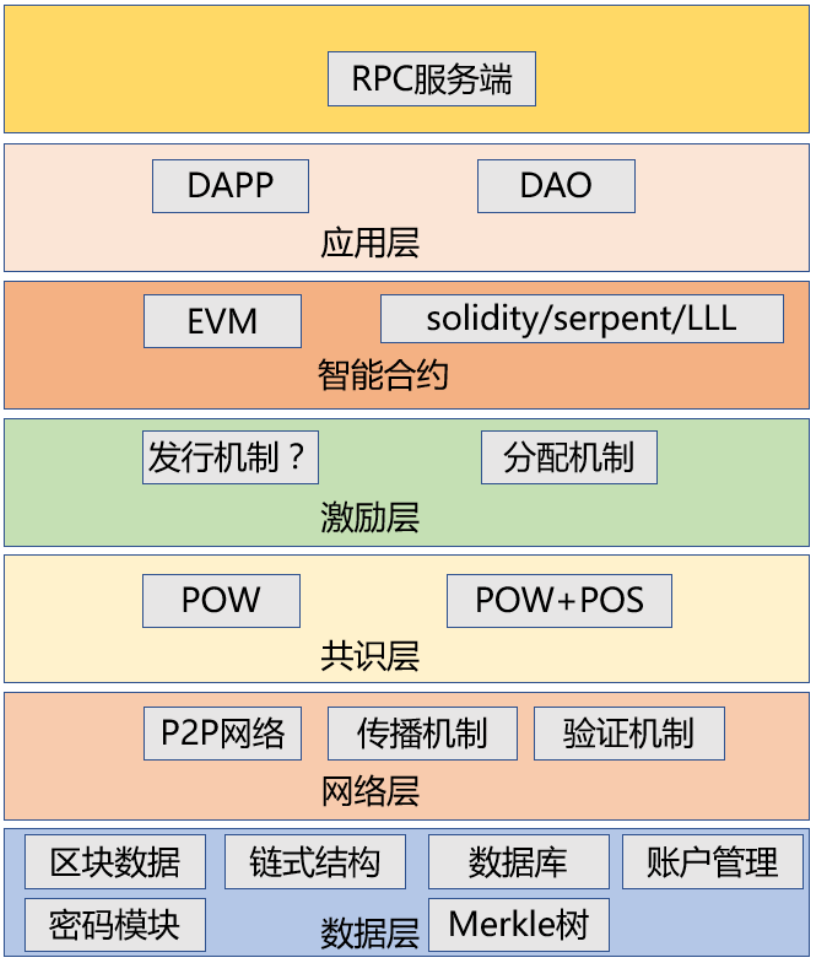
的发送者获得某种所有权，如该辆自行车接下来三个小时的使用权。然后，该车主可以用智能手机向 Slock（智能锁）发送自己的签名的消息，而这条消息能将车上的锁打开。上述的整个过程没有涉及任何中心化的支付机构，服务器或其他第三方，包括 Slock 公司本身。所以，使用 Slock（智能锁）的人，不用担心 Slock 公司倒闭之后锁能否继续使用的问题，也不用担心服务商突然开始收取高昂的费用，更不用担心自己的私人交易信息掌握在某一方的手中。

基于以太坊的区块链应用还包括各种各样的金融合约——将简单的实体资产（黄金、股票）数字化应用，面向互联网基础设施更安全的应用（如 DNS 和证书颁发机构），去中心化的个人线上身份管理等等。

实际上，以太坊旨在将区块链技术去中心化、开放性和安全性引入到几乎所有可以被计算的领域。

2、 以太坊总体框架图

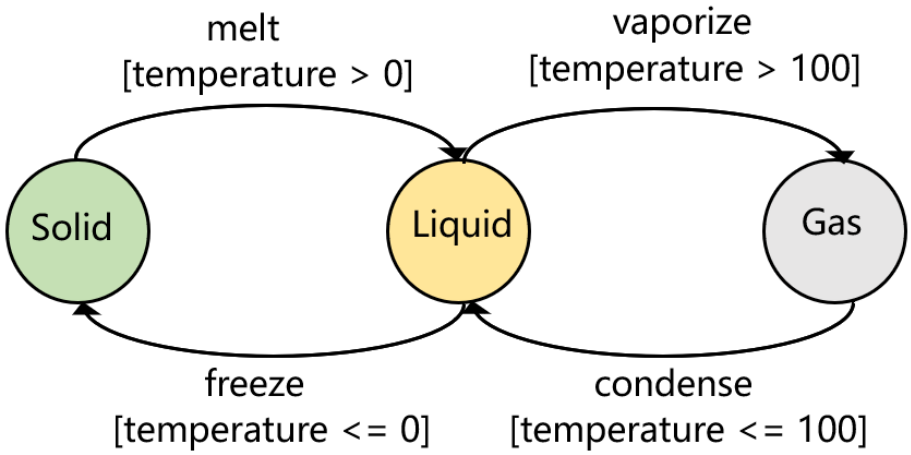
目前我们仅仅给一个简单的图形，后期我们会展示一个 Ethereum 黄皮书中具体详细的流程图，后续可以根据模块化进行章节叙述



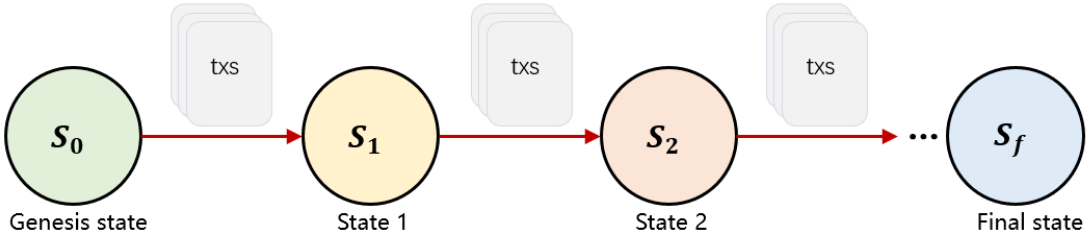
3、 以太坊状态转换函数图

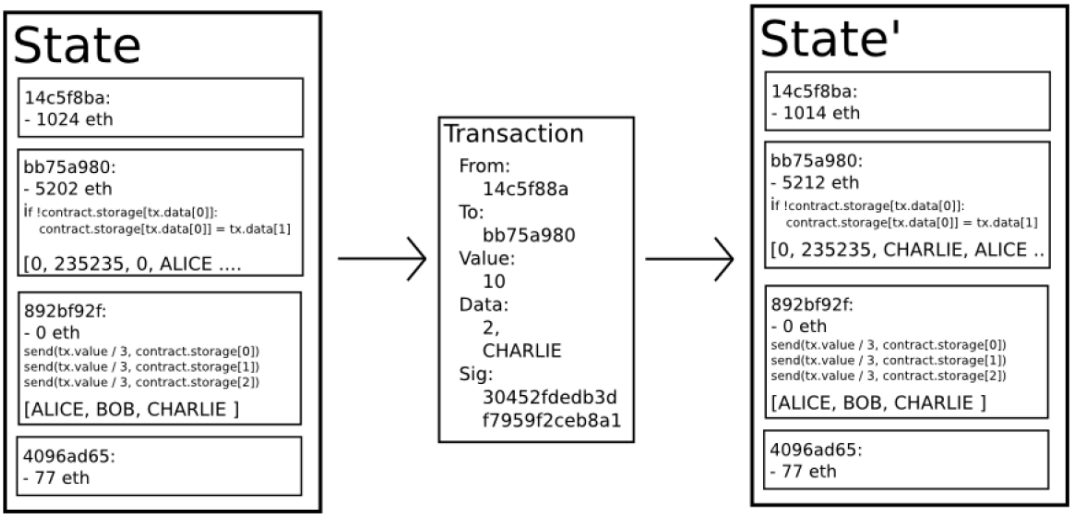
以太坊的本质是一个基于交易的状态机（transaction-based state machine）。所谓状态机是指由状态寄存器和组合逻辑电路构成，能够根据

控制信号按照预先设定的状态进行状态转移，是协调相关信号动作、完成特定操作的控制中心。举例如下：



在以太坊中，从创世状态（genesis state）开始（我们定义为S_0），每执行完交易，状态就会发生改变，直到执行完所有的交易，我们获得最终状态（final state S_f），也就是以太坊的当前状态。

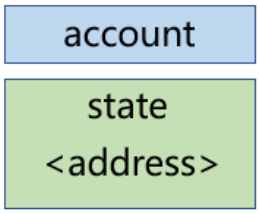




二、 以太坊基本概念

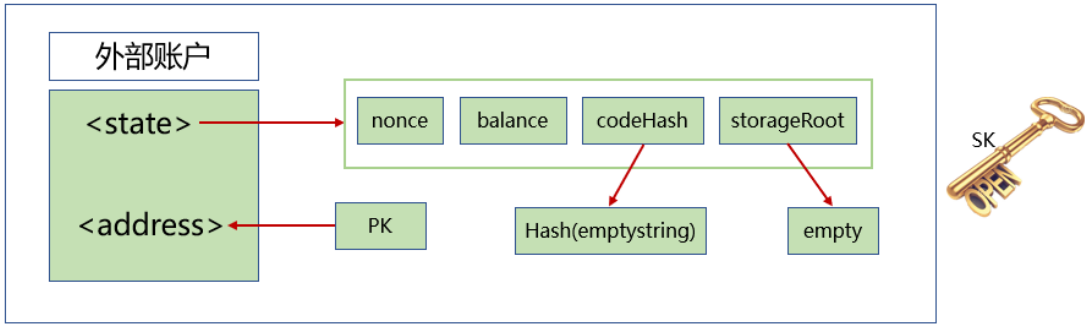
1、账户

比特币的区块链是关于“交易的列表”，而以太坊的基础单元是账户，以太坊区块链跟踪每个账户的状态，以太坊上面所有的状态的转换都是账户之间的价值和信息的转移。以太坊中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）。其中地址是 160 位的标识符，用于识别账户。

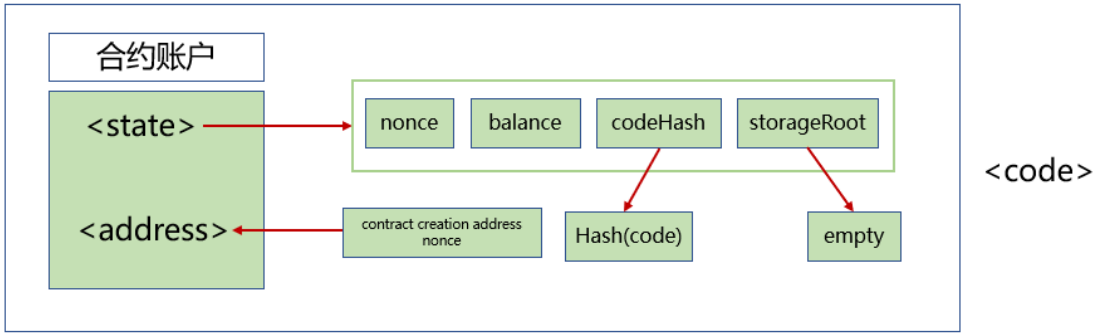


1.1、以太坊区块链中主要有两种类型的账户：

（1）外部账户（EOA）：外部拥有的账户，被私钥控制，没有 code 与之关联（主要人为掌控，因为人掌控着账户的私钥），人们可以通过创建并签名一笔从外部账户发出的交易。



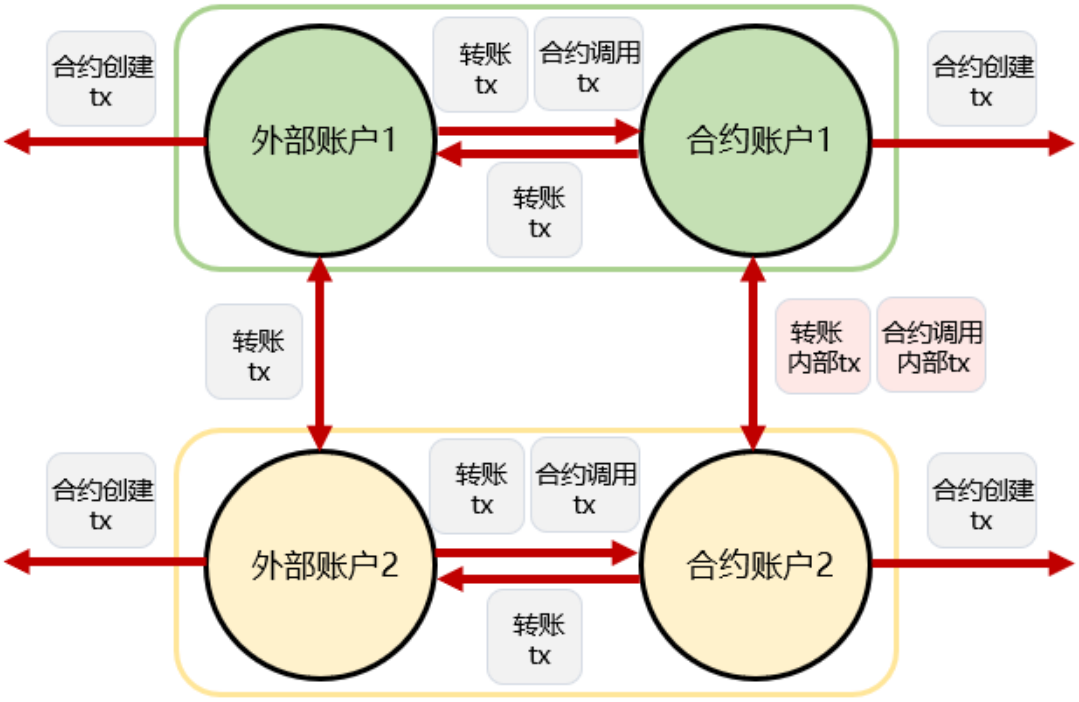
（2）合约账户：由存储在账户中的合约代码控制，并且只能由外部账户“激活”。即只有当外部账户发出指令时，合约账户代码被激活才能执行相应的操作，允许对内部存储进行读取、写入、发送其他消息和创建合约。注意：即使该账户能被人类控制，也是因为合约代码设定了，其可以被人类控制。在合约账户中的 code 就是智能合约（交易发送给该账户时所运行的程序。）



1.2、两种账户的区别：

账户类型	以太币余额	控制权	是否有 code	执行

外部账户	有	私钥	无	发送以太币交易/触发合约代码执行交易
合约账户	有	代码	有	从其它合约接收交易/信息触发来执行代码，由于是图灵完备的，可执行任意的操作，并且操控自己永久存储



1.3、世界状态（world state，也称全局状态）简称 state

世界状态实际上账户地址（address）与账户状态（account state）组成的一个映射。该映射形式为 key-value，其中 key 为地址，value 为账户状态 account state。其中地址是 160 位的标识符，账户状态是按照 RLP 算法（详细算法介绍见第九章第二小节）进行序列化的数据结构。

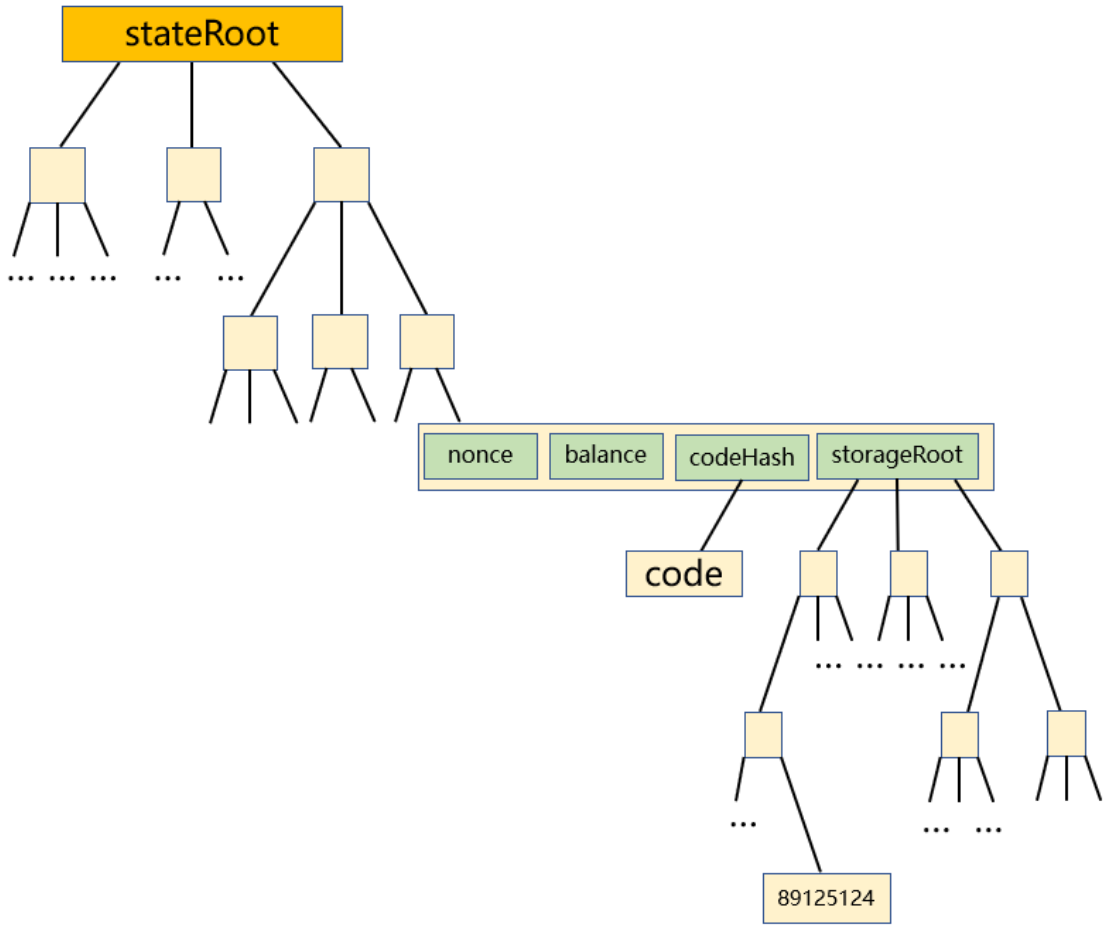
Address	Account state
0x123456...	X
0x1a2b3f...	Y
0xab123d...	Z

其中 world state 并不保存在区块链上，而是被保存在一个叫做 Merkle Patricia 树的数据结构中(也就是区块头的 stateRoot 树对应的 state 中)。（详细内容见第九章第 3 节）

使用 Merkle Patricia 树的数据结构的优点：（1）该数据结构的根节点在密码学意义上依赖于其所有的内部数据，因此它的 hash 值可以用作整个系统状态的安全标识。(2) 作为一个不可变的数据结构，它允许通过相应的改变 root hash 来调用任意之前的状态（previous state，并且 previous state

的 root hash 已知)。

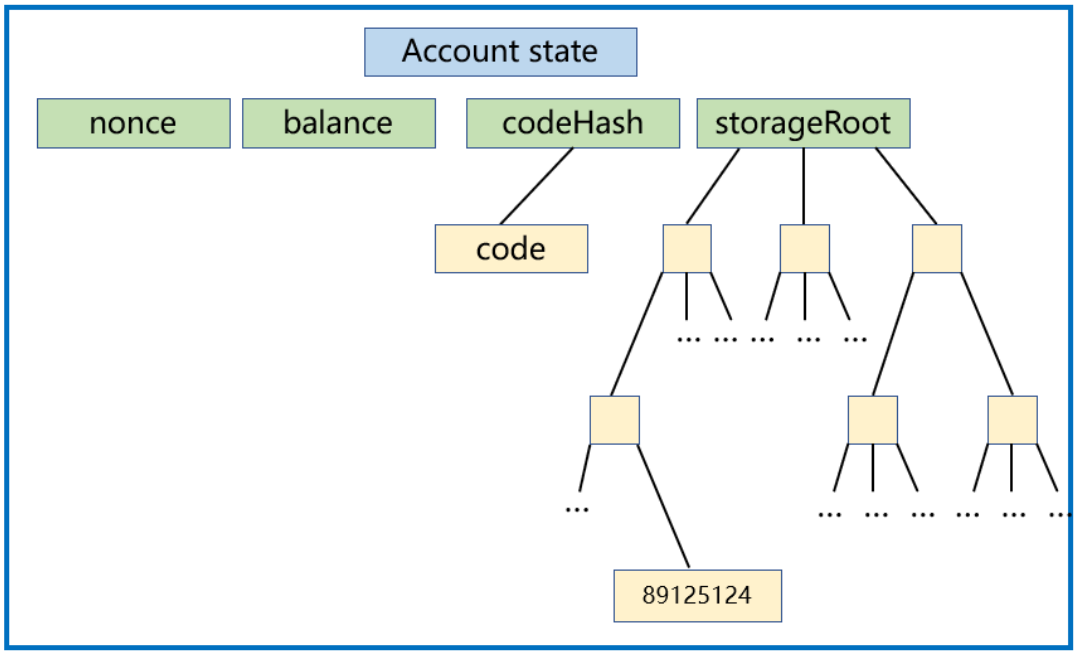
由于我们将所有的 root hashes 存储在区块链上，因此我们可以恢复到 old state。



1.4、账户状态 account state

账户状态主要包含以下四个字段：

- nonce：对于外部账户，nonce 表示的是从该账户地址发送的交易序号；对于合约账户，nonce 表示该账户创建的合约序号
- balance：账户余额，即该账户地址拥有的 Wei 的数量，1Ether = 10^{18}Wei
- codeHash：该账户 EVM code 的 hash 值。对于外部账户，此字段默认为空字符串的 hash 值，即（KECCAK-256(emptystring)）。对于合约账户，此字段为 code 的 hash 值，即（KECCAK-256（code））。由于其是不可变的，因此一旦构造了，是不能改变的。所有的代码 code 都保存在状态数据库的相应的 hash 值之下，以供后续检索（具体见合约账户的图）。
- storageRoot：Merkle Patricia 树的根节点的 hash 值。长度为 256 比特。Merkle Patricia tree 编码账户的存储 storage 内容，而每个账户都有一块永久性的存储 storage，其形式为 key-value，key 和 value 的长度均为 256 比特，该存储的读写操作开销比较大，修改存储甚至更多。在合约里，不能遍历账户的存储，并且一个合约只能对自己的存储进行读写。



2、交易

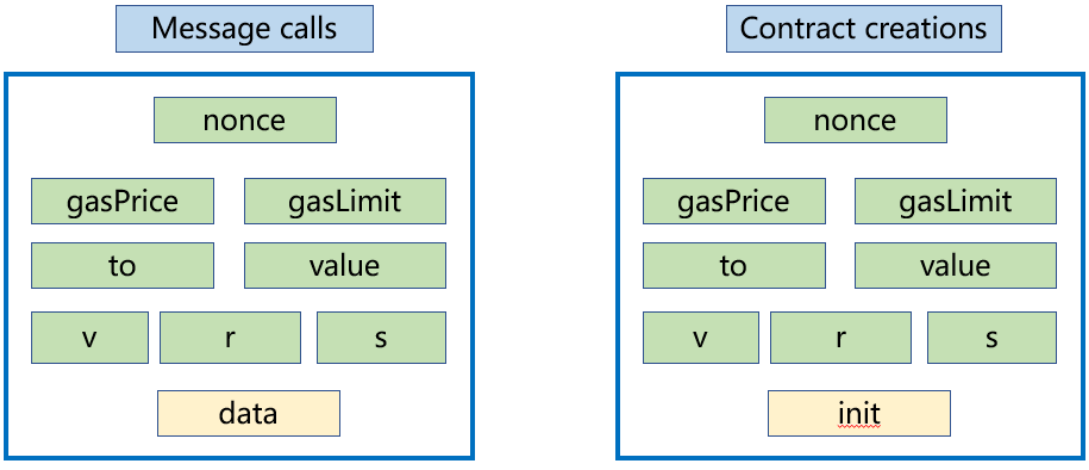
以太坊的本质是基于交易的状态机，即两个不同账户之间发生交易，让以太坊的 world state 从一个状态转换成另一个状态。在以太坊中交易 transaction 指被外部账户生成的加密签名的一段指令，序列化之后提交给区块链。在以太坊中有两种类型的交易：

（1） 消息通信（message calls）：这个又分为 normal transaction 与 transaction to contract。

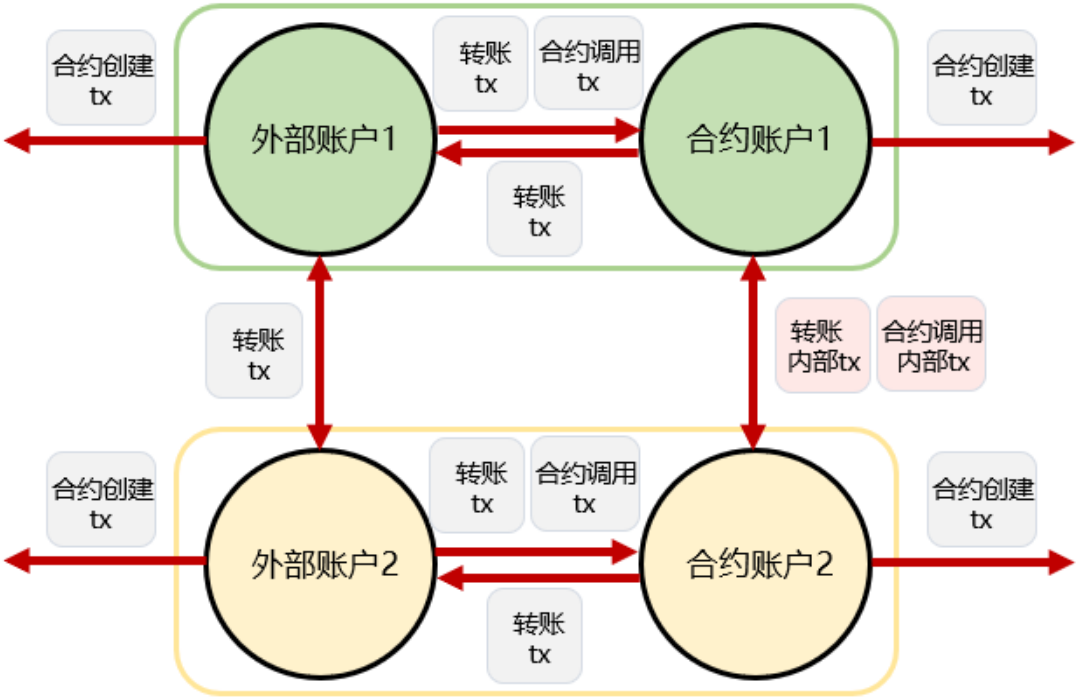
- normal transaction：与比特币的交易类似，仅仅进行账户之间转账
- transaction to contract：向合约账户发送交易，可以选择进行回应，有点类似于函数的概念

（2） 合约创建（contract creation）：为相关的 code 创建新的合约账户，创建合约的过程实际上就是部署合约的过程。

两种交易包含的字段以及区别如下：



- nonce：发送者发送的交易数量（防止重放攻击）
- gasPrice：交易中指定的一个 gas 的价格（换算成以太币）
- gasLimit：发送者执行交易愿意支付的 gas 数量的最大值。该值被设定后，在计算完成之前会被预先支付掉，且之后也不能再增加。
- to：对于（message call）交易，这个表示接收者地址，160bit 标识符；对于合约创建(contract creation)交易，由于合约账户地址尚未创建，初始值为空。
- value：对于 message call 交易，表示从发送者账户地址转移到接收者地址 Wei 的数量。对于合约创建交易，value 作为新创建的合约账户的余额
- v,r,s：交易的签名，用于标识交易的发送者。采用的是椭圆曲线签名算法 ECDSA-256（具体算法见第九章）
- **init**：只有 contract creations 交易才存在该字段，该字段是用来初始化新合约账户的 EVM 代码片段。init 只有在合约账户创建的时候才被执行一次，之后就被丢弃。当 init 第一次执行的时候，它返回一个合约账户的代码体（body），当账户收到一个 message call（外部账户发来的交易/内部代码执行的交易），该代码体（body）就会执行。
- **data**：只有 message call 交易才存在该字段，该字段的大小没有限制，表示的是 message call 的输入数据



注意：

- 合约与合约之间也是可以通信的，只不过它们是通过内部交易（internal transaction）来进行通信的。内部交易类似于交易，但是不同的是内部交易不是由外部账发生的，而是被合约产生的。它们是虚拟的对象，与交易不同，没有被序列化而且只存在于以太坊执行环境。（**简而言之就是外部账户发出一笔调用合约的交易，该交易又能调用其他的合约（内部**

交易/子执行），合约又能调合约）

- 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。
- 此外内部交易不包含 gasLimit。因为 gas limit 是由原始交易的即外部账户创建者决定的。外部账户设置的 gas limit 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”（内部交易），例如合约到合约的内部交易。如果在执行一笔交易的过程中，其中内部交易的执行（子执行）造成了 gas 的不足，那么子执行的状态会回滚。

3、区块

在以太坊中所有的交易都被打包成一个”块”。而区块链包含了一系列这样链在一起的区块。

3.1、在以太坊中，一个区块包含：

（1） 区块头
（2） 包含在区块中交易信息（可以看成区块主体）
（3） 与当前区块的 ommers 相关的一系列其他区块头

3.2、ommers

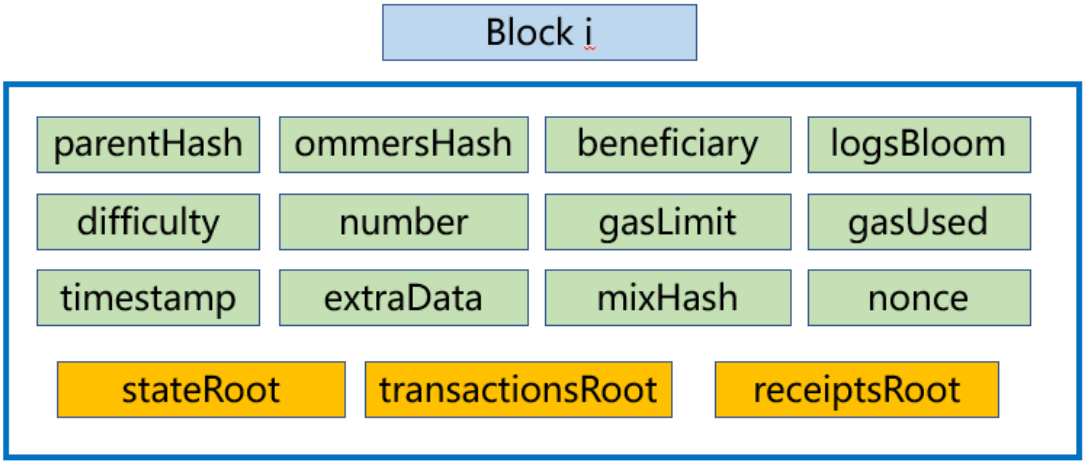
Ommer 指与当前区块的父区块拥有相同的父区块（简单理解，是当前区块的父区块的兄弟姐妹，通常叫叔区块）。叔区块是指那些没有在最长的链上，而是在分叉链上所挖出来的有效区块。

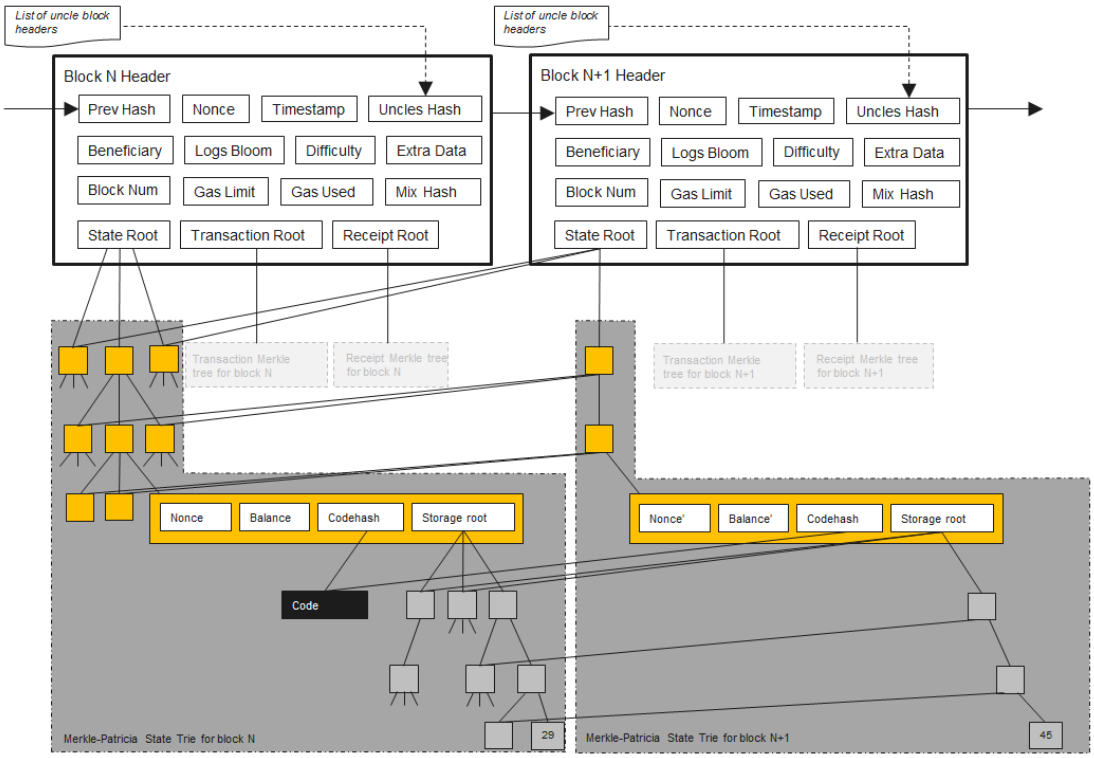
挖到叔区块的矿工可能由于网络延迟的原因而没有同步到最新的区块。以太坊利用这种机制来分散中心挖矿现象（即最大矿池垄断区块生产，导致单个的矿工总是落后于大矿池获得区块信息，因此即使单个矿工找到正确的区块，也无法获得任何收益。）

Ommers 的目的就是为了帮助奖励矿工纳入这些区块。矿工包含的 ommers 必须是有效的，也就是 ommers 必须是往上数 6 代之内或更小范围内父区块的子区块。（因为根据下面的公式，ommers 的 6 代之外也没奖励啦呀。） 一个孤区块在第 6 个子区块之后，这种陈旧的孤区块将不会再被引用。一个叔区块一旦包含在有效的区块链中，挖到该叔区块的矿工可以获得对应以太币作为奖励。这也保证了以太坊能够以很短的时间产生区块（平均 15 秒），而不会因为网络同步的延迟而产生多个分叉。

叔区块奖励公式：$reward_{uncle} = (8 + U_n - B_n) * R/8$。其中$U_n$表示 uncle number；$B_n$表示 block number；$R$表示静态奖励即 5 个 Ether。举例如下：
如果$B_n = 1888, R = 5$
Uncle 0：$U_n = 1887$，$reward_{uncle} = 4.375$
Uncle 1：$U_n = 1886$，$reward_{uncle} = 3.75$
Uncle 2：$U_n = 1885$，$reward_{uncle} = 3.125$

3.3、区块字段





- parentHash：父区块头的 Keccak-256 的 hash 值
- ommersHash：当前区块 ommers 列表的 hash 值
- beneficiary：发送挖到该区块所获得的所有奖励的地址，即矿工的地址，该地址为 20 字节（可以看成 coinbase 交易的地址）
- logsBloom：bloom 过滤器由每个日志条目中包含的可索引的信息（日志地址、日志主题）组成
- difficulty：区块的难度，可以由前一个区块的难度与时间戳计算得来
- number：当前区块技术，创世区块的序号为 0，后续每个区块，区块序号都增加 1（可以理解成区块高度）
- gasLimit：每个区块当前限制的 gas
- gasUsed：当前区块中交易消耗的所有 gas 总量

- timestamp：时间戳，此区块成立是 Unix 时间戳
- extraData：与此区块相关的附加数据，但是必须不超过 32 字节
- mixHash：一个 256 比特 hash 值，当与 nonce 组合时，证明此区块已经执行了足够的计算
- nonce：一个 64 比特 hash 值，当与 mixHash 组合时，证明此区块已经执行了足够的计算
- stateRoot：状态 Merkle Patricia 树的根节点，Keccak-256 的 hash
- transactionRoot：交易 Merkle Patricia 树的根节点
- recipietsRoot：交易收据的根节点，实际上是展示每一笔交易影响的数据条。

根据上述字段，我们可以知道以太坊的每一个区块头，包含了三棵 Merkle Patricia 树。分别对应 state、transaction、receipts。

- ✧ state tree 通常会负责处理：查询账户余额、查询账户是否存在、如果合约中运行了某特定的一笔交易，输出值查询
- ✧ transaction tree 通常会负处理：这笔交易是否被包含在特定的区块中
- ✧ receipts tree：通常负责处理：查询某个地址在过去 30 天内，发出某种类型事件的所有实例（如一个众筹合约完成了它的目标）

3.4、其它字段解释

（1）日志

以太坊允许日志可以跟踪各种交易和信息。一个合约可以通过定义“事件”来显示的生成日志。

一个日志的实体包含：

- 记录仪的账户地址（logger's address）
- 代表本次交易执行的各种事件的一系列主题（log topics 32 字节）以及与这些事件相关的任何数据
- 日志被保存在 bloom 过滤器 中，过滤器高效的保存了无尽的日志数据。(将日志实体压缩成 256 字节 hash 值)**（后期加章节专门讲述过滤器）**

（2）交易收据

每个交易收据主要包含了以下 4 个字段，而每个交易收据按照 MPT 树的格式存储，最终 Merkle Patricia root（交易数据根 hash）值保存在区块头中。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。像你期望的那样，每个收据包含关于交易的特定信息，这些信息为：

- Post-transaction state
- 交易执行完之后，包含交易收据的当前区块累计消耗的 gas
- 执行当前交易使创建的一系列日志，256 字节
- 由这些日志 logs 信息组成的 Bloom 过滤器

（3）区块难度

区块的难度是被用来在验证区块时加强一致性。创世区块的难度是 131072，有一个特殊的公式用来计算之后的每个块的难度。如果某个区

块比前一个区块验证的更快，以太坊协议就会增加区块的难度。**（后期详细解释区块难度的计算）**

区块的难度影响 nonce，它是在挖矿时必须要使用工作量证明算法计算出的一个 Hash 值。区块难度和 nonce 之间的关系用数学形式表达就是：

$$m = H_m \wedge n \leq \frac{2^{256}}{H_d} \; with \; (m,n) = PoW(H_n^*, H_n, \boldsymbol{d})$$

其中m表示新区块的 mixHash 字段，n表示的是新区块的 nonce 字段。H_n^*为没有 nonce 字段和 mixHash 字段的新区块头部，H_n为区块头的 nonce 字段，$\boldsymbol{d}$为一个大的数据集，主要被用于计算 mixHash，而H_d是一个新区块的困难值。

目前找到符合条件的 nonce 唯一方法就是使用工作量证明算法来遍历所有的可能值。找到 nonce 所花费的时间与难度成正比，即难度越高，找到符合条件的 nonce 就越困难，相应地验证一个区块也就相对越难，而这相应地增加了验证新块所需的时间。所以，通过调整区块难度，协议可以调整验证区块所需的时间。另一方面，如果验证时间变的越来越长，协议就会降低难度。这样的话，验证时间自我调节以保持恒定的速率—平均每 15s 一个块。

4、 Gas 和费用

4.1、什么是 gas

Gas 是交易发送方需要为每个以太坊区块链上发生的操作所支付的执行费用，这个概念仅仅在交易执行过程中存在。因此任意一个给定计算机程序，包括创建合约、进行 message call 交易、利用和访问账户的存储，在虚拟机上执行操作均需要消耗一定的 gas。

4.2、gas 设置的目的

为了避免网络资源的滥用和蓄意攻击现象以及回避图灵完备不可避免的问题，以太坊中所有程序的计算都与 gas 有关。一方面限制交易执行所需的工作量，另一方面为交易的执行支付费用。

4.3、需要消耗 gas 的情形

在以太坊中，以下三种情况是需要收费的，而这三者均为执行操作的先决条件。

（1）计算操作需要消耗 gas，这是最常见的一种消耗 gas，（后期将图片粘贴过来，具体可见黄皮书附录 G）

（2）Message call 或者 contract creation 需要消耗 gas，主要有 CREATE，CALL，CALLCODE 这几个指令（简而言之就是交易需要消耗 gas，以太坊中主要分为两种类型交易）

（3）内存 memory 使用需要消耗 gas。内存的总费用通常与所使用的 32 字节的最小倍数成比例，这样使得所有内存都能包含在计费范围内。例如：如果

使用了 33 个字节的内存，那么账户就需要支付两个 32 字节的费用。存储 Storage 的费用会有一些比较细微的方面，例如，由于增加存储会增加所有节点上的以太坊数据库(指的是 Google levelDB 数据库)的大小，因此为了激励保持 storage 小的存储量。如果账户执行存储 storage 清理操作时，该操作不仅不会消耗 gas，还会得到一定数量的 storage 使用费用的折扣，以鼓励用户尽量释放不用的存储。在实际操作中，这种折扣在账户执行操作之前就已经被支付给用户了，这是因为存储 storage 初始化所产生的费用要高于一般的 storage 使用的费用。

4.4、gasLimit 与 gasPrice

每一笔交易都必须包含一个 gasLimit 和 gasPrice（每个 gas 的单位价格）。其中 gasLimit 是根据交易中 gasPrice 以及发送者账户中的余额,购买而来。如果账户的余额不能支付其购买，那么该交易不合理。GasPrice 可以由交易发送者任意指定，但是 gasPrice 越高，意味着需要花费的 Ether 越多，那么最终的交易费也会相对越高。矿工打包时可以任意选择他想打包的交易，通常来说，交易费更高的交易，更容易被矿工打包到下一个区块中。因此通常矿工会发布他愿意将交易打包到区块中的最低的 gasPrice，交易者可以根据矿工提供的这个 gasPrice 来设置自己的 gasPrice。

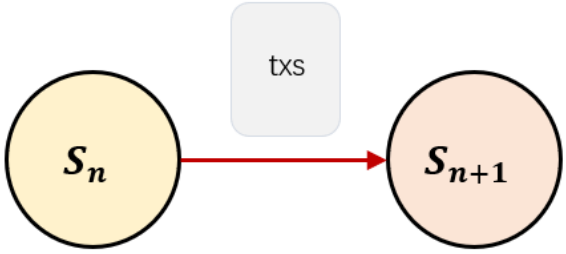
当交易执行时，gas 会按照特定的规则被逐渐消耗。而发送者账户需要预付的交易费用=gasPrice * gas 总量。

- 如果交易产生的用于计算消耗的 gas 总量，不超过 gasLimit，那么交易会进行，最终返回一个 return，并且执行结束如果还有 gas 剩余，这些 gas 会返还给发送者账户。
- 如果 gas 总量超过了 gasLimit，无论计算执行到什么位置，gas 被耗尽时会触发一个 out-of-gas 异常。同时，当前调用是所作的状态修改都将被回滚，即状态返回到交易执行前的状态（简而言之，消耗了 gas，做了无用功，状态也没有任何的改变）。

4.5、例子

（此处最好用一个例子说明 gas 如何减掉的过程）

三、 交易执行



如果你发送了一笔交易给以太坊网络处理。交易的执行主要包含以下几个过程：

- 交易的执行首先要通过合理性的验证（intrinsic validity），主要包括：交易是完全按照 RLP 格式构成（可以参照第九章 RLP 算法）

- 交易的签名是合理的
- 交易中的 nonce 是合理的，即等于发送者账户当前的 nonce，注意这里的 nonce 指的是发送者账户发送交易的序列号
- 交易的 gasLimit 不能小于内在固有的 gas（具体解释内在固有 gas，理解公式）
- 发送者账户的余额要足以支付需要花费的费用，因为以太坊采用的是提前预付的形式

（2）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 suicide set：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 log series：虚拟机的代码执行的归档和可检索的检查点，它允许以太坊外部旁观者（如去中心化的应用前端）很容易地跟踪到“合约调用 contact calls”。
- 退款余额 refund balance：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 storage 需要付费，发送者要是清理了存储就会相应的退款。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。

（3）交易所需的各种计算开始被处理。

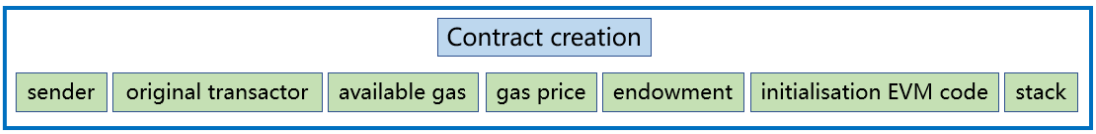
当交易所需的步骤全部处理完成，并假设没有无效状态，通过确定退还给发送者的未使用的 gas 量，最终的状态也被确定。除了未使用的 gas，发送者还会得到上面所说的“退款余额”中退还的一些津贴。一旦发送者得到退款之后：

- gas 的计算得到的 Ether 就会给矿工
- 交易使用的 gas 会被添加到区块的 gas 计数中（计数一直记录当前区块中所有交易使用的 gas 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
- 最后，我们就有了一个新的状态以及交易创建的一系列日志。

四、 合约创建

1、创建一个合约的时候需要以下几个固有的参数：

（1）发送者 sender：s
（2）交易发起人 original transactor：o
（3）可以使用的 gas：g
（4）gasPrice：p
（5）endowment：v
（6）任意长度的字节数组即 EVM code：i
（7）代表 message call 深度/合约创建的栈：e

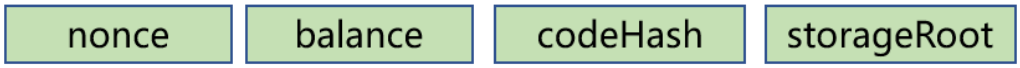


2、合约创建账户的地址计算方式如下：

主要跟发送者地址以及发送者的 nonce 有关。通过将发送者地址与 nonce 进行 RLP 编码之后，进行 Keccak-256 hash 之后，取最右边的 160 比特，作为新创建的合约的地址。

3、初始化账户

我们已经创建了一个新的合约账户，下面我们使用如下方法来初始化一个账户：



- 设置 nonce 为 0
- 如果发送者通过交易发送了一定量的 Ether 作为 value，那么设置账户的余额为 value
- 设置合约的 codeHash 为一个空字符串的 Hash 值，即 Keccak-256(())。
- 将存储设置为 0

4、执行

一旦我们完成了账户的初始化，使用交易发送过来的 init code，实际上就创造了一个账户。init code 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 storage，也可能是创建另一个合约账户，或者发起另一个消息通信 message call 等等。(详细见第六章执行模型)(code execution can effect several events that are not internal to the execution state: the account's storage can be altered, further accounts can be created and further message calls can be made.)

当初始化合约的代码被执行，这个过程会消耗 gas。

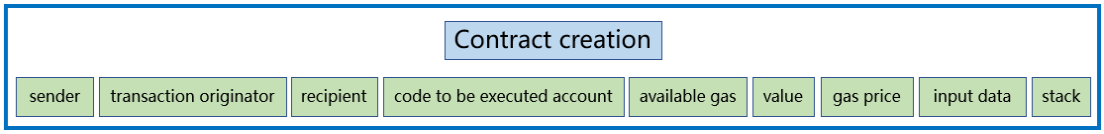
- 如果交易成功执行，那么计算操作、合约创建、内存和存储消耗的 gas 会被支付掉。其中存储费用与创建的合约代码的大小成正比。此时如果还有剩余的 gas，那么剩余未使用的 gas 会被退回给原始的交易发送者，改变的状态也会被永久保存。
- 如果没有足够的 gas 来支付交易的整个花费，那么无论交易执行到哪一步，都会触发一个 out-of-gas 异常，状态回滚并退出。在这个过程中消耗掉的 gas 也不会退还给发送者。但是，如果发送者随着交易发送了 Ether(balance 字段)，如果合约创建失败，Ether 是会被退回来。即发送的以太币会被退回，但是消耗的 gas 不会退回。

注意，当初始化 code 正在执行时，新创建的地址存在但是没有内在的主体代码。因此在此期间收到的任何 message call 都不会导致代码执行。如果初始化执行

以指令 SUICIDE（自毁）结束，则该事项是没有意义的，因为在交易完成之前账户将被删除。对于一个正常的 STOP code，或者是 code 返回值为空，状态将留下一个僵尸账户，任何剩余的余额将永久锁定在账户。

五、 消息通讯 message call

1、执行一个 message call 的时候需要以下几个固有的参数：
（1）发送者 sender：s
（2）交易发起人 transaction originator：o
（3）接收者 recipient：r
（4）code 将被执行的账户 c，通常就是接收者账户
（5）可以使用的 gas：g
（6）value ： v
（7）gasPrice：p
（8）任意长的自己数组 **d**：代表 call 时输入的数据
（9）代表 message call 深度/合约创建的栈：e



2、执行
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 init code。不过，它可以包含输入数据 **d**，如果交易发送者提供了此数据的话。一旦执行，消息通信会包含一个额外的组件即“输出数据”，通常在交易执行被忽略了，但是由于 VM-code 执行时，可以启动 message calls（即需要该组件时就会被使用）。
就像合约创建一样，如果消息通信执行退出是因为 gas 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 gas 是不会被退回给原始交易发送者的。相反，所有剩余的未使用 gas 也会被消耗掉，并且状态会立刻回滚。

六、 执行模型（execution model）

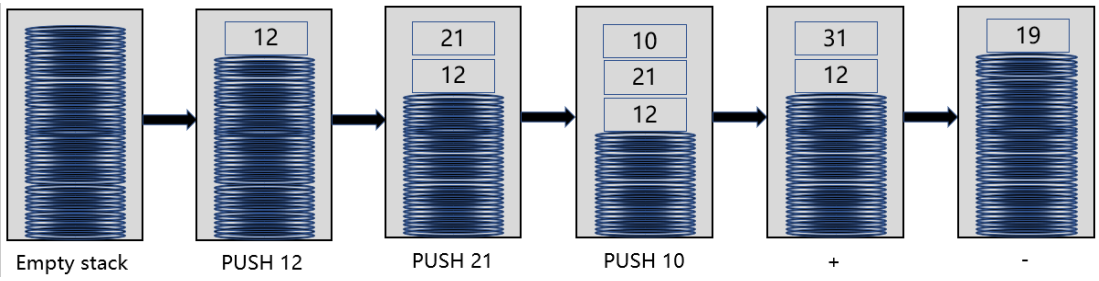
我们知道交易执行会经过一系列的操作，执行模型指明了，给定一系列字节码指令以及一些环境数据的情况下（bytecode instructions + environmental data），系统的状态如何改变的。下面我们重点介绍交易是如何在 EVM 中执行的。

交易的处理实际上是在自己的以太坊虚拟机中执行的。以太坊虚拟机 Ethereum virtual machine（EVM），以太坊期望构造图灵完备的虚拟机（EVM），但是实际上，它并不是真正意义上的图灵完备。所谓图灵完备是指：一切可计算的问题都能计算，在可计算理论中，当一组数据操作的规则（一组指令集，编程语言，或者元胞自动机）满足任意数据按照一定的顺序可以计算出结果。但是以太坊的 EVM 本质上受 gas 的限制，也就是能完成的计算总量是受到提供的 gas 总量的限制的。

1、基本概念介绍

以太坊是一个简单的基于栈的结构，主要遵循后进先出原则

- EVM 中每个栈的项的大小为 256 比特，这样设计主要是为了 Keccak-256 hash 方案以及椭圆曲线的计算。
- 栈有一个最大的大小，为 1024 位。
- EVM 有内存 memory model：可以按照可寻址字节数组来存储（word-addresses byte array）。是易失性的，内存的数据不是长久存储的。
- EVM 有一个独立的 storage model：与内存不一样，存储器是非易失性的，会作为系统状态的一部分进行维护。初始化的时候，storage 与 memory 均初始为 0
- EVM 与标准的冯诺依曼架构不同，它将程序代码存储在虚拟 ROM 中，并通过特殊的指令来访问，而不是将程序代码存储在一般可访问的 memory 或者 storage 中。
- EVM 有一些异常的执行指令，如：栈溢出，不合理指令，out-of-gas。如果碰到这些指令，机器立马停止并将问题报告给执行代理（要么是交易处理器或者 spawning 执行环境），执行代理会单独处理。



2、费用设置 fees

在以太坊中，以下三种情况是需要收费的，而这三者均为执行操作的先决条件。（详细内容可以参照第二章第四节）

（1）计算操作需要消耗 gas，这是最常见的一种消耗 gas，（后期将图片粘贴过来，具体可见黄皮书附录 G）

（2）Message call 或者 contract creation 需要消耗 gas，主要有 CREATE，CALL，CALLCODE 这几个指令（简而言之就是交易需要消耗 gas，以太坊中主要分为两种类型交易）

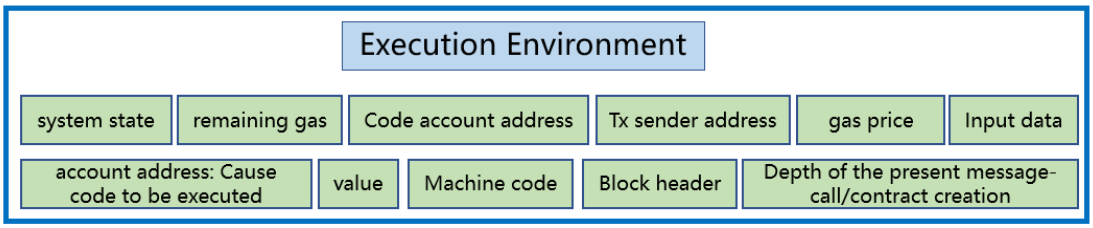
（3）内存 memory 使用需要消耗 gas。内存的总费用通常与所使用的 32 字节的最小倍数成比例，这样使得所有内存都能包含在计费范围内。例如：如果使用了 33 个字节的内存，那么账户就需要支付两个 32 字节的费用。存储 Storage 的费用会有一些比较细微的方面，例如，由于增加存储会增加所有节点上的以太坊数据库（指的是 Google levelDB 数据库）的大小，因此为了激励保持 storage 小的存储量。如果账户执行存储 storage 清理操作时，该操作不仅不会消耗 gas，还会得到一定数量的 storage 使用费用的折扣，以鼓励用户尽量释放不用的存储。在实际操作中，这种折扣在账户执行操作之前就已经被支付给用户了，这是因为存储 storage 初始化所产生的费用要高于一般的 storage 使用的费用。

3、执行环境

除了系统状态σ和合约运算剩余的 gas：g，在执行环境中执行代理还需要提

供以下重要信息

- I_a：拥有需要执行的 code 的账户地址（code account address）
- I_o：交易发送者地址
- I_p：gasPrice
- I_d：输入数据，如果执行代理是一个交易，那么这个就是交易数据
- I_s：触发代码执行的账户的地址，如果执行代理是交易，那么这个就是交易发送者地址
- I_v：value，单位为 Wei
- I_b：字节数组，需要被执行的 code
- I_H：当前区块的区块头
- I_e：当前 message call 或者 contract creation 的深度。即目前执行的 CALL 操作和 CREATE 操作的数量



如果以上信息都包含在一个元组I内，系统状态变化的函数为Ξ，σ'为系统运行之后的状态，g'为运行后剩余的 gas，s为执行终止（suicide）操作的合约列表，l为记录序列（log），r为运行后返还的 gas，o为合约运行后所产生的输出，那么整个以太坊的状态转换可以使用如下公式表示

$$(\sigma', g', s, l, r, o) = \Xi(\sigma, g, I)$$

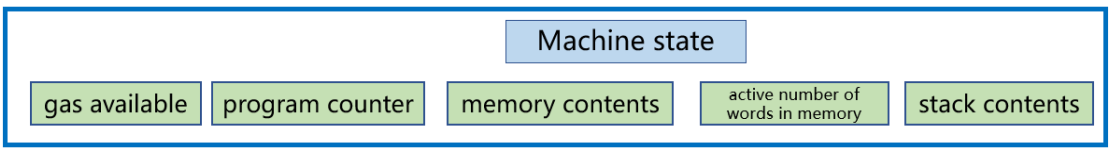
4、执行

根据上述第 3 小节，现在我们需要定义系统状态变化函数Ξ。在大多数实际情况下，整个系统的状态转换是一个不断迭代系统临时状态σ和虚拟机临时状态μ的过程。迭代的过程会进行异常检查和指令输出函数。迭代的终止由一下两个条件决定：

- 系统状态是否异常：gas 不足、指令无效、虚拟机堆栈容量不足等。
- 所有指令执行完毕并返回结果，虚拟机正常停止工作。

在每一次迭代的过程中，智能合约的指令被压入堆栈，虚拟机按照堆栈的索引执行指令。每执行一条指令，将支付相应的 gas，直到所有的指令执行完毕，堆栈被清空。如果在执行过程中遇到异常情况，那么虚拟机将停止动作并返回异常。

4.1、虚拟机状态 machine state：μ



虚拟机的具体指令可以参考 Ethereum yellow paper 附录 H。

4.2、异常停止

以下几个方面会导致执行状态处于异常停止状态：

- 没有足够的 gas
- 指令不合理
- 没有足够的 stack items
- JUMP/JUMPI 目的地不合理或者新的栈的大小超过 1024 位。

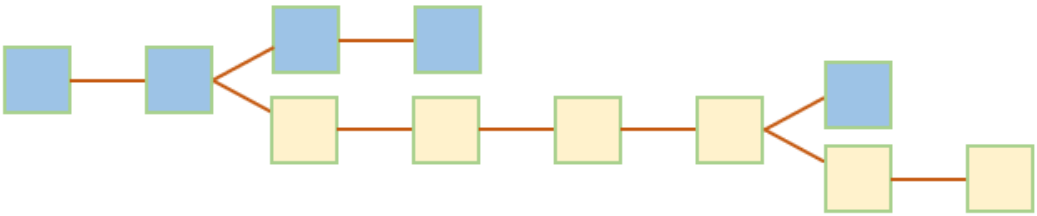
5、执行周期

七、 区块树到区块链

由于以太坊系统是去中心化的，因此每个参与者都有机会在之前存在的区块链上创建新的区块。那么由此产生的结构必然是一个区块树的结构。而规范的区块链是从根（创世区块）到叶子（包含大部分最新交易的区块）通过整个区块树的路径。如果节点对于从区块根到区块叶子节点哪一条路径为最好的区块链（best blockchain）存在分歧，那么此时会产生分叉。

这就意味着在给定的时间点（块）之后，系统的多个状态可以共存。一些节点相信一个区块包含规范的交易，其它节点相信其它一些区块是规范的，它可能包含完全不同的或不兼容的交易。对于这样是要不惜一切代价避免的，因为其随之而来的不确定性会使整个系统不被信任。

为了在哪个路径上达成共识，以太坊中方案中使用 GHOST 协议的简化版本。（**具体见后续介绍**） 在概念上，我们规定完成最多的计算量，或者最终的路径为 best blockchain。显然，确定最重路径的因素之一是叶子区块数，相当于区块的数量。当然不能包括没有挖出来的区块。如果路径越长，为了到达叶子区块必须完成的工作量证明也就越大。这与比特币衍生协议中使用的方案类似。



八、 区块完成

1、区块的完成主要涉及 4 个步骤

（1） 验证（或者，如果挖矿，就是“确定”）ommers
（2） 验证（或者，如果挖矿，就是“确定”）交易
（3） 申请奖励
（4） 验证（或者，如果挖矿，就是“计算合理”）state 和 nonce

2、ommers 验证

在区块头中的每个 ommers 必须是有效的，也就是 ommers 必须是往上数 6 代之内

或更小范围内父区块的子区块。(因为根据下面的公式，ommers 的 6 代之外也没奖励啦呀。) 一个孤区块在第 6 个子区块之后，这种陈旧的孤区块将不会再被引用。

3、交易验证

- 区块中总共使用的 gas 必须与最后交易累加起来使用的 gas 是一样的
- 交易是完全按照 RLP 格式构成（可以参照第九章 RLP 算法）
- 交易的签名是合理的
- 交易中的 nonce 是合理的，即等于发送者账户当前的 nonce，注意这里的 nonce 指的是发送者账户发送交易的序列号

4、奖励

区块的奖励主要包含以下几个方面：

- 静态奖励：矿工根据挖矿所得的 5 个以太币作为奖励，目前是 3 个
- 动态奖励：挖矿所得的交易费归矿工所有，如果区块中包含叔区块，那么矿工还能从每个叔区块中获得额外的 1/32 以太币作为奖励，即矿工奖励为$\left(1+\frac{||B_U||}{32}\right)*R$，其中$R$表示静态奖励即 5 个 Ether，$||B_U||$为叔区块的个数，但是每个区块中最多只能包含 2 个叔区块。

叔区块奖励公式：$reward_{uncle}=(8+U_n-B_n)*R/8$。其中$U_n$表示 uncle number；$B_n$表示 block number；$R$表示静态奖励即 5 个 Ether。举例如下：如果$B_n=1888, R=5$

- Uncle 0：$U_n=1887$, $reward_{uncle}=4.375$
- Uncle 1：$U_n=1886$, $reward_{uncle}=3.75$
- Uncle 2：$U_n=1885$, $reward_{uncle}=3.125$

5、state 和 nonce 验证

- 验证 state 树的根节点的 hash 值
- 验证状态的转移
- 验证区块链的工作量证明，并确认将新区块链连接在权威的区块链上，并将整个系统更新到最新的状态。

九、 以太坊区块链中的密码学技术

1、哈希算法

1.1、Keccak-256

1.2、

2、RLP 算法（编码）

此章节来源于 https://www.jianshu.com/p/da638d0fcea4。想深入了解的朋友

可以详细看看。

RLP(Recursive Length Prefix)是一种编码规则，可用于编码任意结构的二进制数组数据。RLP 编码的结果也是二进制序列。RLP 主要用来序列化/反序列化数据。

2.1、RLP 定义

RLP 编码的定义只处理以下两类数据：

- 字符串（string）：指字节数组。
- 列表（list）：一个可嵌套结构，里面可包含字符串和列表，具体可以参考 Python 的列表。

其他类型的数据需要转成以上的两类数据，才能编码。RLP 编码数据主要有两个特点：可嵌套和递归性。

2.2、RLP 编码规则

RLP 编码的重点是给数据前面添加一个字节的前缀，而这个前缀是和数据的长度相关的。RLP 编码中的长度是数据的实际存储空间的字节大小，去掉首位 0 的正整数，用大端模式表示的二进制格式表示。RLP 编码规定数据（字符串或列表）的长度不得大于 8 字节。因为超过 8 字节后，一个字节的前缀就不能存储了。

- 如果字符串的长度是 1 个字节，并且它的值在[0x00, 0x7f] 范围之间，那么其 RLP 编码就是字符串本身。即前缀为空，用前缀代表字符串本身；
- 否则，如果一个字符串的长度是 0-55 字节，其 RLP 编码是前缀跟上(拼接)字符串本身，前缀的值是 0x80 加上字符串的长度。由于在该规则下，字符串的最大长度是 55,因此前缀的最大值是 0x80+55=0xb7，所以在本规则下前缀(第一个字节)的取值范围是[0x80, 0xb7]；
- 如果字符串的长度大于 55 个字节，其 RLP 编码是前缀级联上字符串的长度再跟上字符串本身。前缀的值是 0xb7 加上字符串长度的二进制形式的字节长度（即字符串长度的存储长度）。即用额外的空间存储字符串的长度，而前缀中只存字符串的长度的长度。

  例如一个长度是 1024 的字符串，字符串长度的二进制形式是\x04\x00，因此字符串长度的长度是 2 个字节，所以前缀应该是 0xb7+2=0xb9，由此得到该字符串的 RLP 编码是\xb9\x04\x00 再跟上字符串本身。因为字符串长度的长度最少需要 1 个字节存储，因此前缀的最小值是 0xb7+1=0xb8；又由于长度的最大值是 8 个字节，因此前缀的最大值是 0xb7+8=0xbf，因此在本规则下前缀的取值范围是[0xb8, 0xbf]；

  以上 3 个规则是针对字符串的，下面的两个规则针对列表的。由于列表的任意嵌套的，因此列表的编码是递归的，先编码最里层列表，再逐步往外层列表编码。
- 如果一个列表的总长度(payload,列表的所有项经过编码后拼接在一起的字节大小)是 0-55 字节，其 RLP 编码是前缀依次跟上列表中各项的 RLP 编码。前缀的值是 0xc0（即 192）加上列表的总长度。在本规则下前缀的取值范围

是[0xc0, 0xf7]。本规则字符串编码的第 2 条规则类似；

- 如果一个列表的总长度大于 55 字节，它的 RLP 编码是前缀跟上列表的长度再依次跟上列表中各元素项的 RLP 编码。前缀的值是 0xf7 加上列表总长度的长度。编码的第一个字节的取值范围是[0xf8, 0xff]。本规则字符串编码的第 3 条规则类似；

2.3、RLP 解码规则

根据 RLP 编码规则和过程，RLP 解码将输入看作二进制字符数组，其过程如下：

- 根据输入首字节数据，解码数据类型、实际数据长度和位置；
- 根据类型和实际数据，解码不同类型的数据；
- 继续解码剩余的数据；

其中，解码数据类型、实际数据类型和位置的规则如下：

- 如果首字节(prefix)的值在[0x00, 0x7f]范围之间，那么该数据是字符串，且字符串就是首字节本身；
- 如果首字节的值在[0x80, 0xb7]范围之间，那么该数据是字符串，且字符串的长度等于首字节减去 0x80，且字符串位于首字节之后；
- 如果首字节的值在[0xb8, 0xbf]范围之间，那么该数据是字符串，且字符串的长度的字节长度等于首字节减去 0xb7，数据的长度位于首字节之后，且字符串位于数据的长度之后；
- 如果首字节的值在[0xc0, 0xf7]范围之间，那么该数据是列表，在这种情况下，需要对列表各项的数据进行递归解码。列表的总长度（列表各项编码后的长度之和）等于首字节减去 0xc0，且列表各项位于首字节之后；
- 如果首字节的值在[0xf8, 0xff]范围之间，那么该数据为列表，列表的总长度的字节长度等于首字节减去 0xf7，列表的总长度位于首字节之后，且列表各项位于列表的总长度之后；

3、Merkle Patricia Tree（MPT）

关于这一部分，大家可以重点参考一下几个链接（主要来源于小 V 的博客）

http://blog.leanote.com/post/hisquery/%E4%BB%A5%E5%A4%AA%E5%9D%8AMPT%E6%A0%91%E8%AF%A6%E8%A7%A3

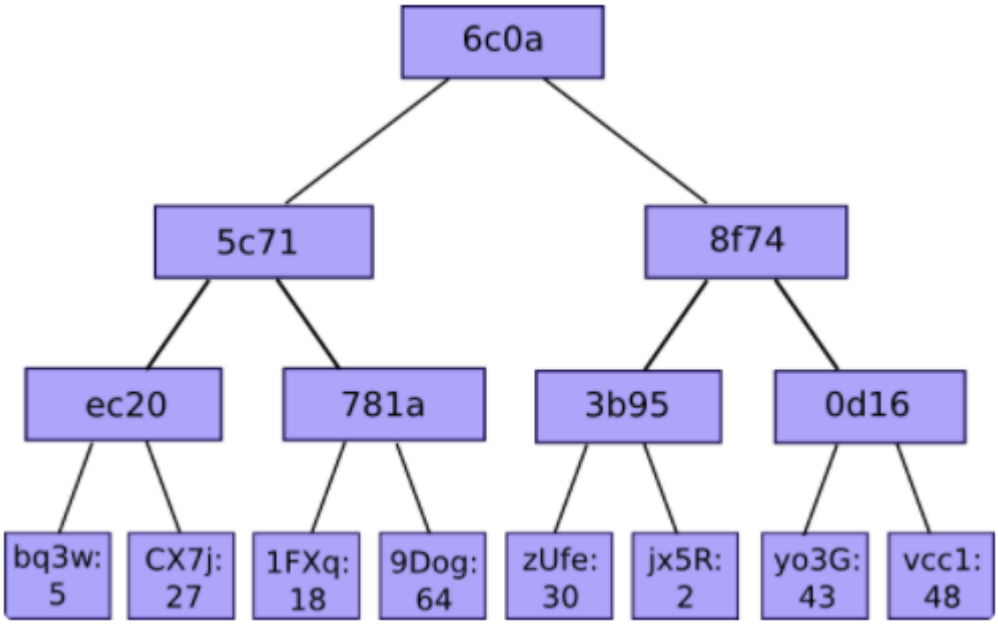
https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie/

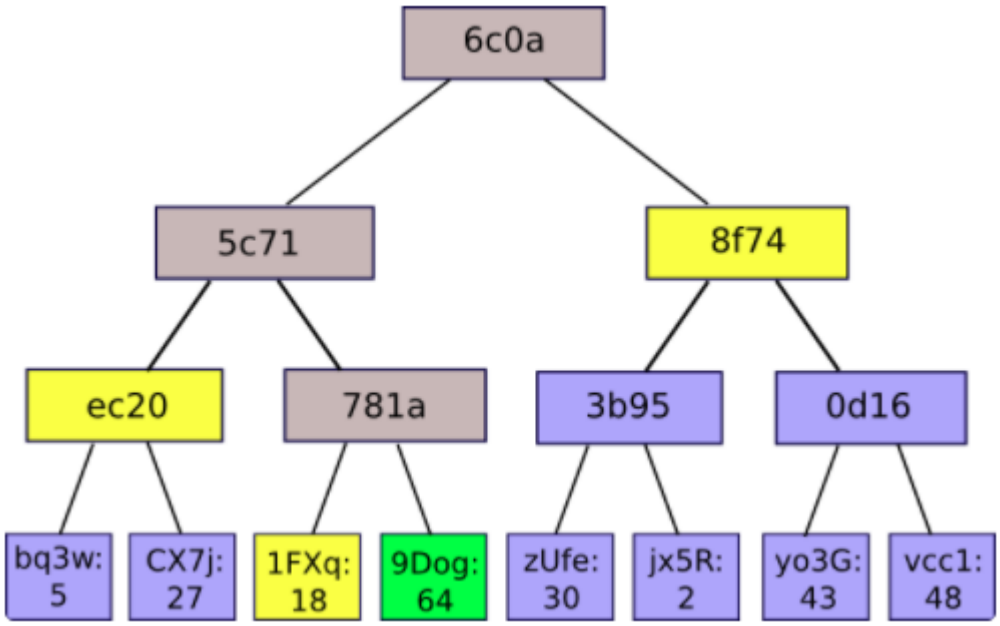
3.1、Merkling in Ethereum

Merkle 树是区块链数据结构的一个重要的部分。 理论上，在没有 Merkle 树的情况下创建区块链也是可行的，但是创建直接包含每个交易（transaction）的大型区块，对其可扩展性会带来巨大的挑战，这样从长远来看，除了超级计算机以外，我们无法信任地使用区块链。Merkle 树的引入，使得所有的大小型计算机和笔记本电脑、智能手机，甚至物联网设备（如由 Slock.it 产生的）都能运行以太坊节点。下面我们详细说明 Merkle 树的工作原理以及它所提供的价值。

广义上讲，Merkle 树是将大量的数据块 chunks 分成多个 buckets，每个 buckets 只包含几个块儿，然后将每个 buckets 做一次 hash 运算，并重复相同的过程，直到算出根 hash。（简而言之，将元数据进行两两 hash，并重复这个过程，直到算出 Merkle root。）

Merkle 树最常见和最简单的形式是二叉 Merkle 树，其中一个 bucket 总是由两个相邻的块 chunk 或 hash 值组成。（每个父节点都有两个叶子节点）描述如下：





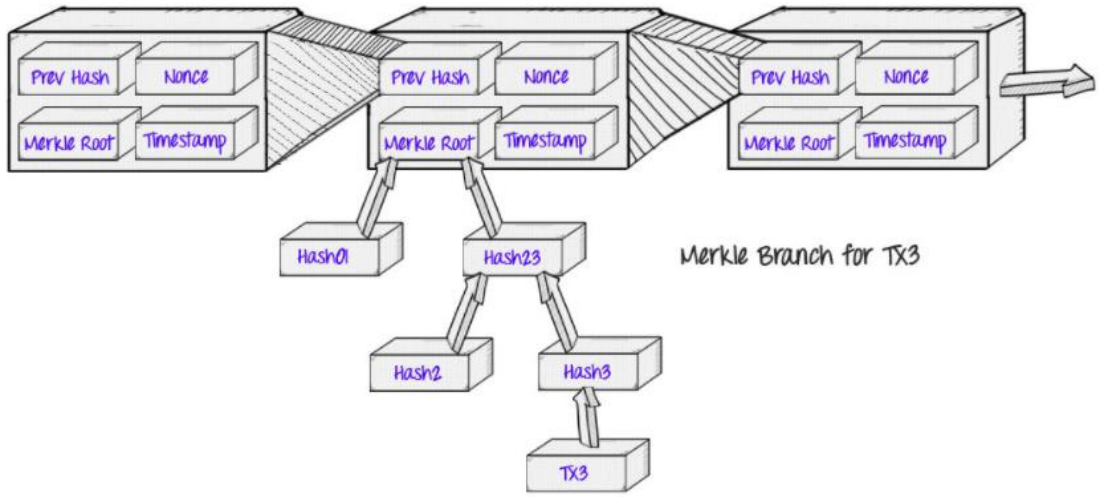
之所以使用这种哈希算法，而不是简单将所有的块儿连接成一个大块，并使用常规的哈希算法。主要是这种数据结构，能提供 Merkle 证明机制（Merkle proof mechanism）。

Merkle 证明由 Merkle root 和由从块（chunk）到根的路径上的所有 hash 值组成的“分支”组成。例如：假设有一个大的数据库，并且数据库的全部内容都存储在 Merkle 树中，Merkle 树的根是众所周知是可信的（例如，它是由足够的信任方进行数字签名的，还是有很多的工作证明）。然后，想要在数据库上进行键值查找的用户（例如“告诉我位于 85273 的对象”）可以要求 Merkle 证明，并且在接收到证明之后验证它是正确的，并且因此实际收到的值是在具有该特定

根的数据库中的位置 85273 处。它允许一个认证少量数据的机制，比如一个散列，可以被扩展来验证可能无限大小的大型数据库。（**可以按照二叉树的形式重新写个言简意赅的版本**）

3.2、Merkle Proofs in Bitcoin

关于 Merkle Proofs 的应用最早出现于比特币中。由 Satoshi Nakamoto 在 2009 年创建。比特币区块链使用 Merkle Proofs 是为了在每个区块中存储交易：



这样做的好处是便于“简单支付验证”(simplified payment verification)。所谓简单支付验证是指：对于“轻客户端/轻节点”，他们只用下载区块链中的区块头，而不用下载每个交易和每个完整区块。其中 80 字节的区块头主要包含以下五个字段：

- 前一个区块头的哈希值
- 时间戳
- 挖矿难度值 difficulty
- nonce 的工作量证明
- 该区块的交易树的 Merkle 根

如果一个轻节点想要确定一笔交易的状态，它可以要求一个 Merkle proof，证明一个特定的交易在一个 Merkle 树中，这个 Merkle 树的根在区块链的区块头中。

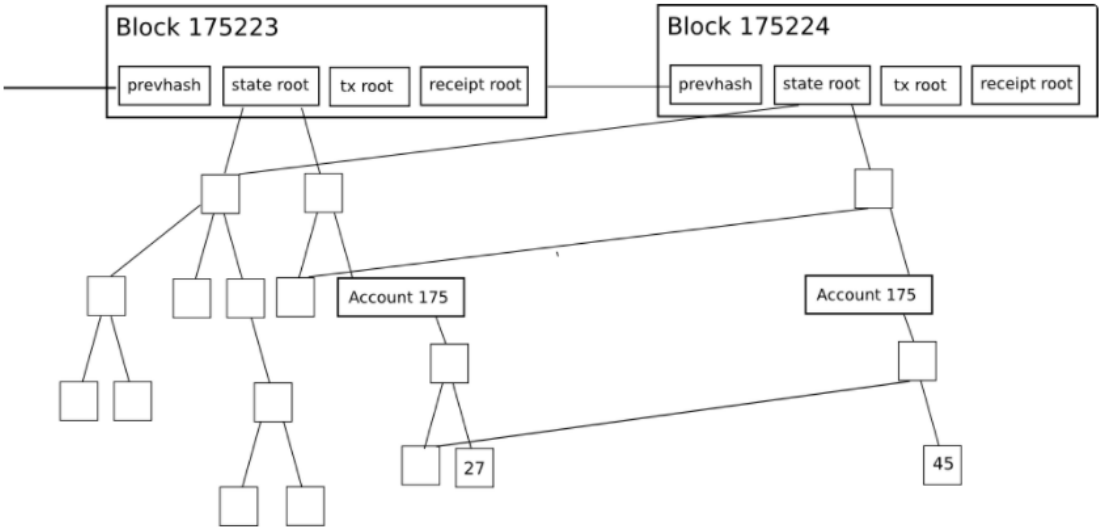
但是在比特币中这种方式也有一定的局限性。(1) 虽然它们可以证明是否包含某特定的交易，但是它们不能证明当前的状态 current state（例如数字资产持有，名称注册，金融合同状态等等）。 例如：查询你现在有多少个比特币。比特币轻节点可以使用一个涉及查询多个节点的协议，并相信至少其中一个会通知它有关它的地址的任何特定交易支出。对于这种简单的用例，当然是很容易实现的。但是对于更加复杂的应用，这种方式还远远不够。

由于一笔交易的实质性效果取决于之前几笔交易的效果，而这些交易本身依赖于以前的交易，因此最终必须验证整个交易链中的每一笔交易。为了解决这个问题，以太坊将 Merkle 树的概念做了进一步的改进。

3.3、Merkle Proofs in Ethereum

在以太坊中每个区块的头部包含了三种对象的三棵 Merkle 树。分别为：

- 交易树 transactions tree
- 收据树 receipts tree（展示每一笔交易影响的数据条）
- 状态树 state tree



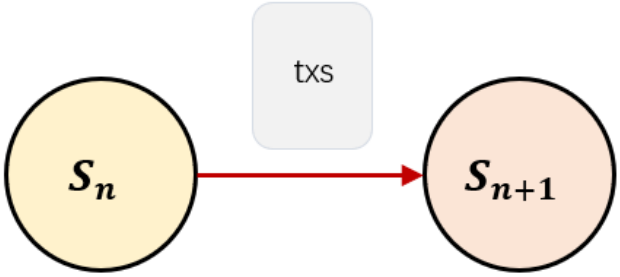
这允许轻节点轻松地做出并得到许多种查询的可验证的答案：如

- 查询某个交易是否被包含在一个特定的区块？（**交易树**）
- 查询某个地址在过去 30 天内发生的所有类型 X 事件（例如众筹合同到达目标）的实例（**收据树**）
- 查询当前账户的余额（**状态树**）
- 查询某个账户是否存在？（**状态树**）
- 如果在某个合约上执行一笔交易，那么输出结果是？（**状态树**）

第一个由交易树处理；第二个由收据树处理；第三和第四个由状态树处理。这四个处理起来相对容易，服务器只需要找到**对象**，就可以获取 Merkle 树的分支（从**对象**到树根的散列列表），并用分支回应轻客户端（轻节点）。

第五种也是由状态树来处理，但是它的计算方式相对比较复杂。

（1）首先我们需要**构建一个 Merkle 状态转移证明**（Merkle state transition proof）。本质上这就是一个证明，如果你在根为 S 的状态上执行一笔交易 T，那么结果将会是根为 S’ 的状态，日志为 log L，输出为 0。（以太坊中，“output”作为一个概念存在，因为每个交易是一个函数调用；这在理论上是不必要的）。



（2）**计算证明 compute the proof**：服务器在本地创建一个假块（fake block），将状态设置为 S，如果轻节点执行一笔交易，如果该交易进程要求客户端确定账

户的余额，那么轻客户端会发出查询余额的询问。如果轻客户端需要检查特定合约存储中的特定项目，则轻客户端为此进行查询，等等。服务器正确地响应所有自己的查询，但是记录/跟踪所有发回的数据。然后服务器向客户端发送来自所有这些请求的组合数据作为证明 proof。然后客户端进行完全相同的程序，但是使用提供的 proof 作为其数据库;如果结果与服务器声明的结果相同，则客户端接受该证明 proof。

3.4、Patricia Trees

我们知道最简单的 Merkle 树是 binary Merkle 树，然而，在以太坊使用的树更加复杂，即 Merkle Patricia Tree。验证“列表”格式的信息，binary Merkle 树是非常合适的数据结构。

对于交易树，binary Merkle 树的这种数据结构也是很好的，因为树一旦被创建，编辑一棵树花费的时间并不重要，因为树一旦被创建，然后永久冻结（as the tree is created once and then forever frozen solid.）。

但是，对于状态树，情况更为复杂。以太坊的状态基本上由一个键值映射组成，其中键 key 是地址，值 value 是账户声明，列出每个账户的余额 balance，随机数 nonce，代码 code 和存储 storage，其中存储 storage 本身就是一棵树。例如，现代测试网络的创世状态如下所示：


```
    {
        "0000000000000000000000000000000000000001": {
            "balance": "1"
        },
        "0000000000000000000000000000000000000002": {
            "balance": "1"
        },
        "0000000000000000000000000000000000000003": {
            "balance": "1"
        },
        "0000000000000000000000000000000000000004": {
            "balance": "1"
        },
        "102e61f5d8f9bc71d0ad4a084df4e65e05ce0e1c": {
            "balance":
"1606938044258990275541962092341162602522202993782792835301376"
        }
    }
```

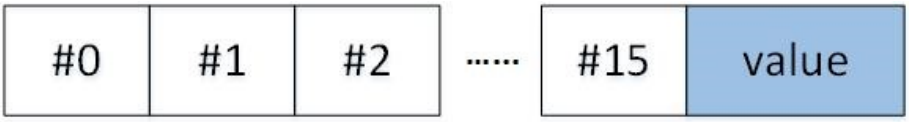

与交易树不同的是，状态需要频繁地更新：账户的余额 balance 和随机数 nonce 经常变化，而且新的账户经常被插入，存储 storage 中键值 keys 也经常被插入和删除。因此以太坊中需要的是一种数据结构，我们能在插入，更新编辑或删除操作之后快速计算新的树根，而无需重新计算整个树。当然对于这种数据结构，还需具备以下两个次要属性：

- 树的深度是有限的，即使是一个攻击者故意制造交易，使树尽可能深。否则，攻击者可以通过操作树来执行拒绝服务攻击，使得每个更新变得非常缓慢。

- 树的根仅取决于数据，而不取决于更新的顺序。以不同的顺序进行更新，甚至从头开始重新计算树也不能改变根。

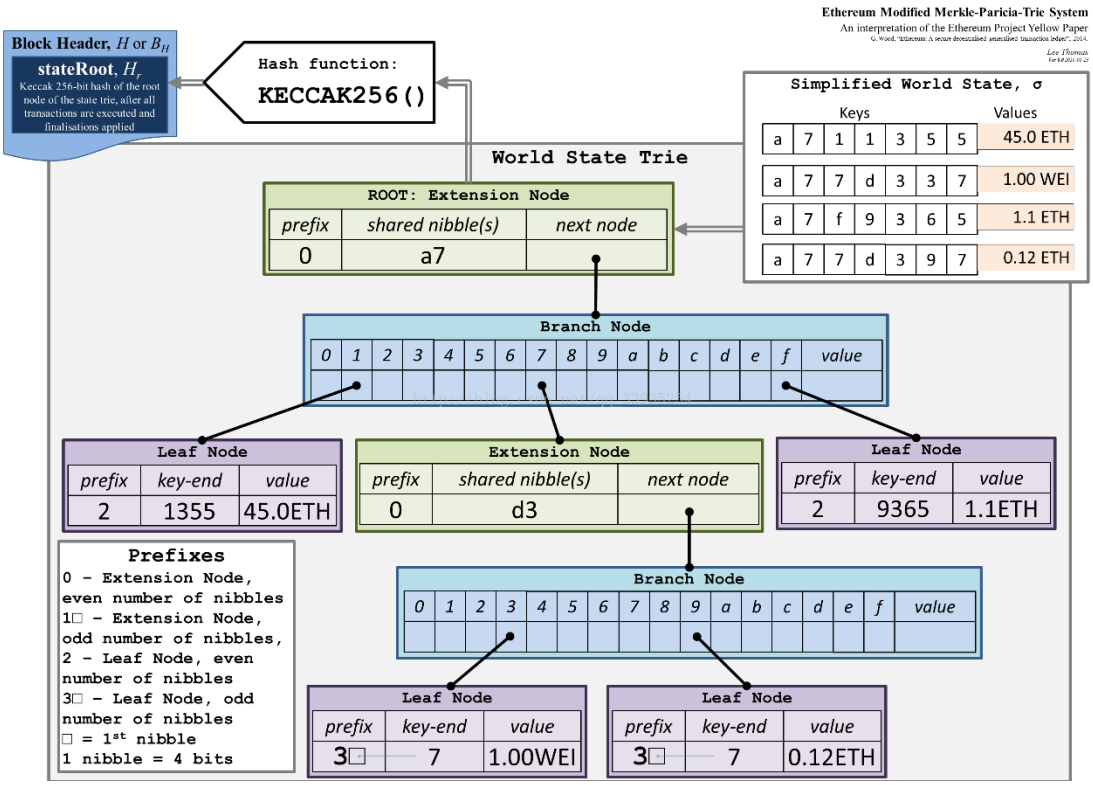
简单来说，Merkle Patricia 树的是满足上述两个特性的最合适的数据结构。在以太坊中，Merkle Patricia 树引入了多种类型节点来提高效率。主要有 4 种类型的节点。即空节点：空节点、叶子节点、扩展节点和分支节点。

- 空节点，简单的表示空，在代码中是一个空串
- 标准的叶子节点，表示为[key,value]的一个 list，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，也是[key,value]的列表，但是这里的 value 是其他节点的 hash，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。
- 最后分支节点，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。



除了四种节点，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 多可以对其进行编码。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，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。



4、公钥密码算法

4.1、椭圆曲线

4.2、椭圆曲线签名算法（ECDSA signature generation）

- 输入：域参数$D=(q,FR,S,a,b,P,n,h)$；私钥d；消息m
- 输出：签名(r,s)
  1) 随机选择$k\in[1,n-1]$
  2) 计算$kP=(x_1,y_1)$，并将x_1转换成整数$\overline{x}_1$
  3) 计算$r=\overline{x}_1 mod\, n$，如果$r=0$，则返回第一步 1)
  4) 计算$e=H(m)$
  5) 计算$s=k^{-1}(e+dr) mod\, n$，如果$r=0$，则返回第一步 1)
  6) 返回签名(r,s)

4.3、椭圆曲线签名验证算法（ECDSA signature verification）

- 输入：域参数$D=(q,FR,S,a,b,P,n,h)$；公钥Q；消息m；签名(r,s)
- 输出：接受或者拒绝改签名
  1) 验证r和s是属于$[1,n-1]$范围内的整数。如果任意一个数的验证不通过则直接返回“拒绝签名”
  2) 计算$e=H(m)$
  3) 计算$\omega=s^{-1} mod\, n$
  4) 计算$u_1=\ e\omega\ mod\, n$并且$u_2=\ r\omega\ mod\, n$
  5) 计算$X=u_1P+u_2Q$
  6) 如果$X=\infty$，则直接返回“拒绝签名”
  7) 将X的x坐标x_1转换成整数$\overline{x}_1$；计算$v=\overline{x}_1\ mod\, n$

8） 如果$v=r$，则返回接受签名；否则返回拒绝签名

证明签名的验证算法是合理的：如果(r,s)是对消息m的签名，并且该签名是由合法的签名者生成的。那么$s=k^{-1}(e+dr)\ (mod\ n)$

$$k=s^{-1}(e+dr)\equiv s^{-1}e+s^{-1}dr\equiv \omega e+\omega rd\equiv u_1+u_2d\ (mod\ n)$$

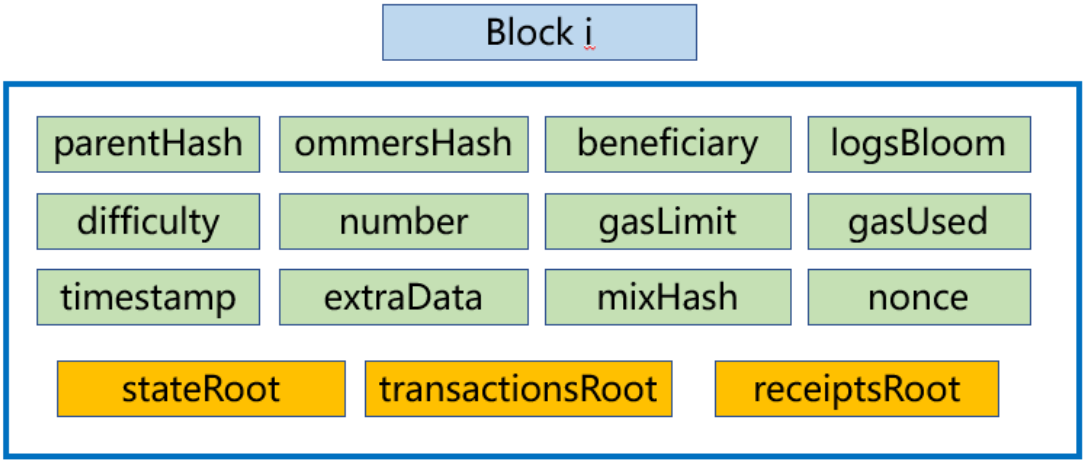
这样$X=u_1P+u_2Q=(u_1+u_2d)P=kP$，因此可得$v=r$。

5、GHOST 协议

十、 共识机制

1、工作量证明机制 POW

工作量证明 proof of work：区块链中的节点通过自己的算力来计算哈希困难性问题来证明工作量，从而获得记账权。简而言之就是挖矿节点通过自己的算力找到一个数（我们通常称之为随机数 nonce），当把这个随机数、当前区块交易等字段，然后用哈希函数计算这一整串字符的输出值，这个输出值正好落在一个相对于这个哈希函数所有可能的输出值中很小的目标区间内。



其中以太坊工作量证明算法的定义如下

$$m=H_m\ \wedge\ n\ \leq\ \frac{2^{256}}{H_d}\ with\ (m,n)=PoW(H_n^*,H_n,\boldsymbol{d})$$

其中m表示新区块的 mixHash 字段，n表示的是新区块的 nonce 字段。H_n^*为没有 nonce 字段和 mixHash 字段的新区块头部，H_n为区块头的 nonce 字段，$\boldsymbol{d}$为一个大的数据集，主要被用于计算 mixHash，而H_d是一个新区块的困难值。

在第八章中我们简单展示了矿工挖出区块所获得的奖励。为了保证 POW 共识算法机制对于安全和财富分配的使用是长期可持续的，以太坊的具有以下这两个特性。

- 尽可能让更多的人可访问：即人们不需要通过不同的硬件设备来运行这个算法，这样便于更多人参与，任何参与人均可以通过提供一些算力而获得一定的以太币作为回报

◆ 降低任何单个节点能够创造与其不成比例的利润可能性。

由于区块链网络中，POW 算法是 SHA-256 哈希函数，在比特币网络中我们曾了解到该种算法缺点就是它能使用 ASIC 这种特殊的芯片，从而可以快速高效的计算出 nonce。因此为了尽量避免这个问题，以太坊的 POW 共识算法的设计为计算出要求的 nonce 需要大量的内存和带宽。大量的内存需要让电脑并行使用内存同时计算出多个 nonce 变得十分困难。而高带宽的需要让同时计算多个 nonce 值也变得异常困难。这在一定程度上降低了中心化的风险，并为网络中的验证节点提供了更加公平的竞争环境。

2、拜占庭容错（需要重写）

拜占庭为过去东罗马帝国的首都，现在位于土耳其的伊斯坦堡。由于当时拜占庭罗马帝国的国土辽阔，基于防御目的，每个军队都分隔遥远，因此将军间只能靠信差传递消息。于战争时，拜占庭帝国军队的将军们必须全体一致的决定是否攻击某一支敌军，因为唯有达成一致的行动才能获致胜利。将军中若存在叛徒，叛徒可以采取行动以欺骗某些将军进行进攻行动，或致使他们无法做出决定，缺乏一致行动的结果则将注定战事的失利。

解决： 一个请求发送到分布式系统时，各个节点可以得出一致的、并且是正确的响应。

当有 f 个内奸，而总成员数>3f+1 时，这个问题才有解！就是说如果有 1 个内奸，那就得 4 个节点一起投票才能把这个内奸找出来；2 个内奸，就需要 7 个节点；

3、权益证明 POS（需要重写）

3.1、概要

我们知道 POW 共识机制中，存在挖矿，并且会耗费大量的电力（据不完全统计，挖矿电力的消耗与整个爱尔兰国家的电力总消耗一样），效率低下，并且一定程度上存在算力集中垄断，与整个分布式系统的去中心化理念有点相悖。此外，这并不是一种节能减排的举措。对于社会能源的可持续性发展而言，难免会受到政府相关的政策的管制，并不适用于长久发展。

因此以太坊生态提出了权益证明机制与分片相结合的方法，来解决目前以太坊网络中存在的这些问题。Proof of stake 本身不是一个完美的解决方法，它自 2011 年开始存在，但是这种算法带来了实质性的好处，既解决了以前系统中的缺陷，甚至引入了新的特性，而这些特性并不存在于工作量证明机制中。Proof of stake 可以看成一种虚拟挖矿“virtual mining”：在 proof of work 中，用户可以花费真是世界中的法币来购买真实的电脑设备，这些设备消耗电力和随机生产块儿的速度大致与所花费的成本成比例。因为

■ 挖矿奖励 = 区块奖励 + 交易费

■ 挖矿成本 = 硬件成本（电脑设备等） + 运营成本（电费、空调费等）

但是在 proof of stake 中，用户花费真实世界中的法币在系统中购买虚拟货币（virtual coins）。然后使用协议 protocol 机制将虚拟货币转换成由协议模拟的虚拟计算机，以与扩大的成本大致成比例的速率随机地产生块：效果差不多，但是没有电力的消耗。

分片 sharding 也不是一个完美的解决方案。它虽然存在的时间很长，但是基于区块链上的应用很受限。基本的方法是通过一种架构来解决可扩展性 scalability 挑战，在这种架构中，来自全局验证器集合的节点被随机分配给特定的分片，其中每个分片并行地处理状态的不同部分中的交易，从而确保工作分布在节点上，而不是由每个人完成

3.2、PoW 与 PoS 的区别：

首先这两个都是区块链上达成共识的算法。我们之所以需要共识，是因为每个人都可以创建区块，但是我们只想要一条链，因此我们需要通过一种方法来决定哪一个区块才是被认可/信赖的。

工作量证明具有很好的性质，可以使用贝叶斯定理和热力学定律来证明给定的块确实需要一定量的工作才能被开采。这样用户可以简单地选择工作量最大的有效链作为正确的链。但是这也暗含了 POW 共识算法的一些弊端，如需要耗费大量的电力、效率低下，并且成本也比较高。这就促使矿工会更加集中 hash 算力,而并不希望有一个分布式网络(该网络为了减少第三方的需要，去中心化)。

POS 不涉及挖矿，仅仅是验证。但是区块仍然需要被创建，那么下一个区块到底由谁来创建（下一个记账权），取决于 proof of stake 算法。但是选择 witness 的过程必须要具有某种随机性，或者至少应该正确的分配投票股份（否则我们又回到了一个集中的系统）

在 POS 中，每个验证者在网络中都具有一些股份 stake。在 PoS 中，每个验证者拥有网络中的一些股份，在以太坊的情况下，他们拥有债券。 债券意味着你将一些资金抵押到网络，并在某种意义上用它作为担保的担保物。在 PoW 中，你知道一个链条是有效的，因为许多工作量证明，而在 PoS 中，你信任最高抵押品的链。

3.3、POS 优势（钱璟整理的资料）

- 不需要浪费算力，同时，进行 51%攻击的代价更高，因为想要进行 51%攻击的话，你得拥有 51%的货币。也就是说，这东西越值钱，攻击的成本就越高。
- 在一定程度上可缩短达成共识的时间，提高转账速度。
- 减少了挖矿难度，在一定程度上能减少能源的消耗。
- 提供了一定的年利率，可缓解加密数字货币系统通货紧缩的问题。

3.4、POS 劣势（钱璟整理的资料）

- **币龄风险：**比如 EPOS 区块的生成是基于币龄的，这是一个随着时间的流逝而线性的增加未花费的币的权重的因子，其证明必须与一个新区块一起提供，并满足以下条件：

  proofhash < 币数 • 币的年龄 | {z }币龄 • 目标

  proofhash 对应于一个取决于权重修正因子、未花费的产出和当前时间的模糊和的哈希值。通过这个系统，攻击者可以把足够的币龄积攒起来，从而成为网络上拥有最高权重的节点。如果攻击是恶意的，攻击者可以对区块链进行分叉并达成双花。但是，此次攻击过后，攻击者必须重新积攒币

龄才能再次发起攻击，因为当区块生成后权益累积就会归零，值得一提的是，这种情况发生的可能性很低，攻击者也没有足够的动机(积攒足够的币龄以成为网络中权重最高的节点，为实现这一目标，需要花费大量时间

另一种情况是这些币龄属于贪婪但却诚实的节点。有一些节点并没有恶意，但是他们的钱包平时都是离线的，只是偶尔进行同步以获得利息。目前的一些系统事实上鼓励了这些节点滥用这一机制，他们平时保持离线，只在累积了可观的币龄以后才连线以获得利息，然后再次关闭，会造成整体网络算力衰减。

- **提前计算和长距离攻击：**对于如何在一个巨大的分布式网当中确保时间戳的安全性还没有已知的解决方案。当前的区块时间戳规则给了攻击者一定程度的自由来选择本小节公式当中提到的 proofhash，并因此提高了让过去几个区块成功分叉的可能性。

此外，目前的权重修正因子没有对哈希功能进行足够的模糊处理以防止攻击者提前计算出未来的权益累积证明。因此恶意的攻击者将能够计算出权益累积证明的解答的下一次间隔，从而能够连续生成多个区块并实施能够危害到整个网络的恶意攻击。

POW 的新增机制是“挖矿”，即矿工每完成一定量的计算，有可能获得一块新 block 中的新增比特币。这个过程是一个纯粹的通胀过程，即无中生有新增比特币。但获得新增的比特币有一定的要求，必须全球第一个找出特定的 HASH 值。

POS 的新增机制是“利息”，即持有一定的 POS 币一定时间，当然得开着客户端，将获得一定量的固定“利息”。这部分“利息”是新增的 POS 币。只要你持有 POS 币并开机，你就能获得一定比例的“利息”，多么类似股票啊，依靠持有获利。

POW 的主要问题是算力过于集中的安全风险，但是相对来说机会的是公平的，算力不公平但机会公平，任何人都可以加入到矿业中，任何人可以研究更高级的矿机，任何人也可以建造任何数量的矿机。

POS 机制最大问题是，看似公平，但里面隐含的其他风险巨大，初始分配是固定的，原始股东以外的人要想获得 POS 币只有转让一条路。转让以后才能获得利息。而原始股东有能力保证其占股比例：不卖就行了，更要命的是，这种保证是制度化的 。只要 POS 持有人不愿意转让其股权，则其他人是不可能获取新的 POS 币的，这点看多么象天朝的股票发行 IPO 啊，很少有人能获得原始股，只能等待交易时被人拉高套牢。

更危险的是如果货币的创造者不安好心的话，隐瞒了大量 POS 货币，伺机抛售，几乎是零成本获利，这也是为什么有很山寨币沦为传销的帮凶的原因，所以本人想说的是，完全的 POS 机制 ICO 根本就是耍流氓，目前看来 POW+POS 方式才是相对合理的方式，例如以太坊目前采用的方式，通过很长一段时间 POW（通过增加算法难度避免专业矿机出现，被破解的时间就是 pow 的时间），让货币充分相对公平分配，然后进入到 POS 阶段。最后，还是那句话，技术永远在发展中，相信人们会找到更公平更公开的方式。

3.5、POS 的应用（钱璟整理的资料）

点点币在SHA256的哈希运算的难度方面引入了币龄的概念,使得难度与交易输入的币龄成反比。在点点币中，币龄被定义为币的数量与币所拥有的天数的乘积。这使得币龄能够反映交易时刻用户所拥有的货币数量。

实际上，点点币的权益证明机制结合了随机化与币龄的概念，未使用至少 30 天的币可以参与竞争下一区块。越久和越大的币集有更大的可能去签名下一区块。然而。一旦币的权益被用于签名一个区块，则币龄将清为零，这样必须等待至少 30 日才能签署另一区块。同时，为防止非常老或非常大的权益控制区块链，寻找下一区块的最大概率在 90 天后达到最大值，这一过程保护了网络，并陆荐时间逐渐生成新的币而无需消耗大屋的计算能力。点点币的开发者声称这将使得恶意攻击变得困难，因为没有中心化的挖矿池需求、而且购买半数以 I 的币的开销似乎超过获得51%的工作地证明的哈希计算能力。

权益证明必须采用某种方法定义任意区块链中的下一合法区块，依据账户结余来选选择将导致中心化，例如单个首富成员可能会拥有长久的优势。为此，人们还设计了其他不同的方法来选择下一合法区块。

早朝的比特币区块链采用高度依赖节点算力的Pow机制，来保证比特币网络分布式记账的二致性,之后又出现了Pos和DPoS等共识机制。除这3类主流共识机制外，实际区块链应用中也衍生出了多个变种机制。这些共识机制各有优劣。例如Pow共识机制在安全性和公平性上比较有优势，也依靠其先发优势已经形成成熟的挖矿产业链，但也因为其对能源的消耗而饱受诟病。而新兴的机制，如pos和DPoS等则更为环保和高效，但在安全性和公平性方面比不上Pow机制。一般来说，pow和pos机制比较适合公共链环境，而PBFT和Raft则比较适合联盟链和私有链的分布式环境。比特币的Pow机制是一种概念性的拜占庭协议，能在定程度上解决拜占庭问题。而Pos等其他机制，目前并没有严格的分析证明其在拜占庭协议方面的属性。

4、委托权益证明DPOS（来自于Emma整理的资料）

4.1、基本概念

- 利益相关者、股权持有者（shareholder）：持有股权的人，每股拥有一份投票可以投一个见证人；
- 见证人（witness）：也是区块生产者，由利益相关者（shareholder）投票选出，允许生成和广播区块，收集 P2P 网络中的交易并使用见证人的私钥进行签名；所提供的服务得到相应的报酬；他们的费率是由股权持有者（shareholder）通过他们选出的委托人（delegates）设置。
- 委托人（Delegates）：有权提出改变参数，从交易费到区块大小、见证人支付、区块间隔等参数，前提是需利益相关者（shareholder）经过2周审查并批准。委托人没有报酬。

4.2、基本原理

PoW机制和PoS机制虽然都能有效解决记账行为的一致性问题，但是比特币的PoW机制纯粹依赖于算力，并且会耗费大量的电力，效率低下，并且一定程度上存在算力集中垄断，与整个分布式系统的去中心化理念有点相悖。PoS机制虽然考虑到了PoW的不足，但是依据权益来选择，会导致股份比较多的人权益相对较大，有可能支配记账权。而委托权益证明delegate proof

of stake 的出现正是基于解决 PoW 和 PoS 这两种共识机制的不足。

BitShares 采用的正是 DPoS 这种共识机制。比特股的 DPoS 机制，中文名叫做股份授权证明机制（又称受托人机制），它的原理是让每一个持有比特股的人（利益相关者 shareholder）进行投票，每个账户每一股投一票投一个见证人，这个过程称为投票制。51%股东投票的结果将是不可逆且有约束力的。由此产生投票数最多的前 101 位代表，我们可以将其理解为 101 个超级节点或者矿池，而这 101 个超级节点彼此的权利是完全相等的。从某种角度来看，DPOS 有点像是议会制度或人民代表大会制度。如果代表不能履行他们的职责（当轮到他们时，没能生成区块），他们会被除名，网络会投票选出新的超级节点来取代他们。在 DPOS 的领导下，我们可以真正地说，行政权力掌握在用户手中，而不是委托人或见证人。

DPOS 利用利益相关者投票的力量，以公平和民主的方式解决共识的问题，以抵消中心化所带来的负面效应。

4.3、见证人随机排序与见证人参与率

见证人随机排序；当统计选票时，每个维护周期（1 天）将会更新活跃的见证人名单。见证人被随机排序，每个见证人在固定的时刻每 2 秒就轮流出块。当所有见证人轮流一遍之后又会随机排序。如果一个见证人在他们给定的时间中没有产生一个块，那么这个时间就被跳过了，下一个证人会产生下一个块。

见证人参与率：任何人都可以通过观察见证人参与率来监测网络健康。历史上，BitShares 保持了 99%的见证人参与率。见证人参与率下降到某一水平，用户网络允许花费更多时间来确认交易并对网络连接性能保持警惕。这一特性使BitShares在故障发生后不到1分钟就能提醒用户潜在问题的独特优势。

4.4、改变规则（硬分叉）

在 DPOS 中，所有的变更都必须由活跃的利益相关者（shareholder）批准触发，即使是最轻微的修改。

虽然在技术上，见证人可以单方面串通并改变他们的软件，但这样做并不符合他们的利益。见证人是根据他们对区块链政策保持中立的承诺而选中的。见证人只是利益相关者的雇员。

开发人员可以实现他们认为合适的任何更改，只要这些更改是基于利益相关者批准的。这种策略保护了开发人员，同时也保护了利益相关者，并确保没有人能够单方面控制网络的方向。

改变规则的阈值和更换 51%的选定见证人是相同的。参与选举见证人的利益相关者（shareholder）越多，改变规则就越难。

4.5、双花攻击概率低

双花，简而言之就是同一笔钱花费两次：拿着币，在 A 商店买了瓶水，在 B 商店买了包瓜子。两个商店几乎同时花，假设商店都不等 1 确认。那么可能 A 或 B 商店最后有一家没有能收到币。那么就实现一次双花。

其中区块链分叉/双重支付攻击指的是攻击者通过不承认最近的某个交易，并在这个交易之前重构新的块，从而生成新的分叉，继而实现双重支付。充

足算力的前提下，攻击者可以一次性更改最近的 6 个区块或者更多的区块。需要注意的是双重支付只能在攻击者拥有的钱包所发生的交易上进行，因为只有钱包的拥有者才能生成一个合法的签名用于双重支付，即攻击者只能在自己的交易上进行双重支付攻击。这意味着见证人由于互联网基础设施的中断而造成通讯中断。

在 DPOS 中，由于通信阻断而产生双花的概率非常低。

该网络能够监控自己的健康状况，当见证人未能按时完成工作，网络能立即检测出在通信中出现的任何损失。当这种情况发生时，用户需要等待，直到有一半的见证人确认他们的交易，这可能会持续一到两分钟。

4.6、区块链重组

由于所有的见证人都是经过选举产生的，高度负责任的，并给予专门的时间段来生产区块，所以很少有任何情况下会存在两个相互竞争的链。

网络延迟将阻止一个见证人及时收到先前的块。如果发生这种情况，下一个见证人将基于他们先收到的区块来解决这个问题。在 99%的见证人参与率的情况下，一笔交易有 99%的概率被单个见证人证实。

虽然该系统对自然链重组事件是稳健的，但仍有一些潜在的软件缺陷，网络中断，或不称职的或恶意的证人创建多个相互竞争的历史，比一个或两个区块长。软件总是选择最高的见证人参与率的区块。一个独立的见证人只能每轮生产一个区块，而且其参与率总是比大多数人低。任何单一的见证人(或少数证人)都不能产生一个参与率更高的区块。参与率是通过比较预期产生的区块数量和实际产生的区块数量来计算的。

4.7、DPoS 背后的基本原理

- 给持股人一把可以开启他们所持股份对应的表决权的钥匙，而不是给他们一把能挖矿的铲子；
- 最大化持股人的盈利；
- 最小化维护网络安全的费用；
- 最大化网络的效能；
- 最小化运行网络的成本（带宽、CPU 等）；

4.8、DPoS 优势与劣势

- 优点：大幅缩小参与验证和记账节点的数量，可以达到秒级的共识验证；
- 缺点：整个共识机制还是依赖于代币，很多商业应用是不需要代币存在的；

4.9、DPoS 应用场景

Bitshares：比特股（BitShares，简称 BTS）是一个结合了去中心化的全球支付系统，去中心化的数字货币交易所(如比特币中国)，去中心化的证券交易所(如纳斯达克)三者的综合系统。

Steemit：用区块链技术搭建的社交媒体平台。其创建服务于两重目标：成为数字代币（Digital Token）的处理系统，以及成为主流社交媒体平台。

EOS：分布式的区块链操作系统，为智能合约提供并行处理，并实现异步通信和建立去中心化组织等操作，它的交易处理速度提高到了每秒钟 100000 笔，还提供了数据库，账号许可，调度，认证和互联网应用通信。

十一、 smart contracts

1、智能合约的历史

2、什么是智能合约

关于智能合约的相关描述有很多，我们首先将参考的一些关于智能合约的描述放在下面，然后我们重点本文档关于智能合约的理解。

- ✧ 法律范畴上看，智能合约是否是真正意义上的合约还有待研究，在计算机领域，智能合约是指一种计算机协议，这类协议一旦制定和部署就能实现自我执行（self-executing）和自我验证（self-verifying），并且不需要人为的干预。从技术层面来看，智能合约可以看做是一种计算机程序，这种程序可以自主的执行全部或者部分和合约相关的操作，并产生相应的可以被验证的数据，以此来说明执行合约操作的有效性。
- ✧ 智能合约是代码（它的功能）和数据（它的状态）的集合，存在于以太坊区块链的特定地址。合约账户能够彼此之间传递信息，进行图灵完备的运算。合约依靠以太坊虚拟机字节代码（以太坊特有的二进制形式）上的区块链来运行
- ✧ 智能合约的理念可以追溯到 1995 年，几乎与互联网（world wide web）同时出现。因为比特币打下基础而受到广泛赞誉的密码学家尼克·萨博（Nick Szabo）首次提出了“智能合约”这一术语。从本质上讲，这些自动合约的工作原理类似于其它计算机程序的 if-then 语句。智能合约只是以这种方式与真实世界的资产进行交互。当一个预先编好的条件被触发时，智能合约执行相应的合同条款。（**百度百科**）
- ✧ A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts allow the performance of credible transactions without third parties. These transactions are trackable and irreversible. Smart contracts were first proposed by Nick Szabo in 1994. (**Wikipedia**)

在这里我们给出本文档关于智能合约的解释。

A smart contract is a **computer program** executed in a **secure environment** that **directly controls digital assets**。

这句话看似简单，但是有四个关键的词我们需要重点关注。

（1） computer program：在这里我们给出维基百科上关于 computer program 的详细解释

A computer program is a collection of instructions that performs a specific task when executed by a computer. A computer requires programs to function, and typically executes the program's instructions in a central processing unit.

（2） secure environment

1）安全的环境。什么样的环境才是安全的，或者说安全的环境有哪些特点呢？

- 能使 computer program 正确执行（execute correctly），没有被篡改（tampered）
- 代码和数据的完整性（integrity of code and data）
- 其他的：代码和数据的保密性；执行的可验证性；内部运行程序的可用性（可调用）

2）安全环境的示例

- 由可信第三方运行的服务器
- 分布式计算机网络（blockchain）
- Quasi-decentralized 计算机网络（联盟区块链）
- 受可信任硬件保护的服务器（如 SGX）

（3） directly controls

字面意思直接控制，顾名思义，不假手于他人，不存在于第三方，完全去中心化。

（4） digital assets

数字资产。广义的范畴上有：

- domain name 域名
- Website
- Money 钱啊
- Anything tokenisable 如黄金、白银、股票等
- Game items
- Network bandwidth, computation cycles.

3、以太坊虚拟机

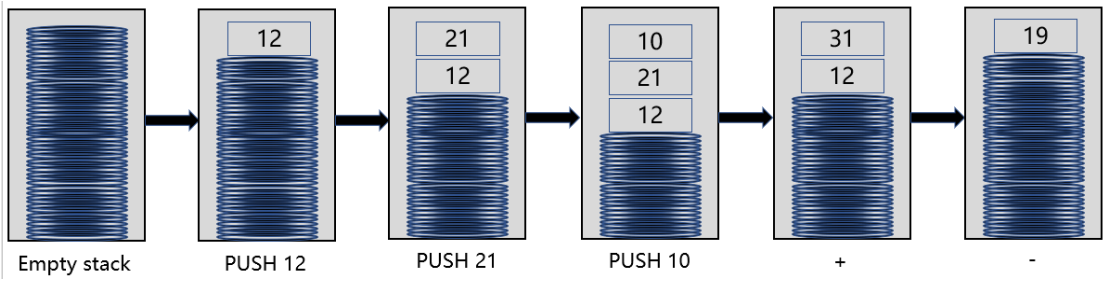
以太坊并不是唯一一个可以在区块链上部署智能合约的平台。如：Fabric、Neo、counterparty、Monax、Codius、Tezos、Rootstock、Corda 等也可以部署智能合约。从狭义上讲，以太坊是一系列去定义去中心化应用平台的协议，其核心在于以太坊虚拟机 Ethereum virtual machine（EVM）。EVM 的引入使得编写智能合约变得更加容易，高度脚本化的程序设计语言使得普通用户也能开发自己的智能合约。

以太坊虚拟机是建立在以太坊区块链上的智能合约代码的运行环境，但是虚拟机本身并没有存储在区块链内，而是和区块链一样同时存储于各个节点（全节点）的计算机上。每个参与以太坊网络中的验证节点都会运行虚拟机，并将其作为区块有效性验证的一部分。每个节点都会对合约的部署和调用进行相同的计算，并存储相同的数据，以确保最权威（最真实）的结果记录在区块链上。

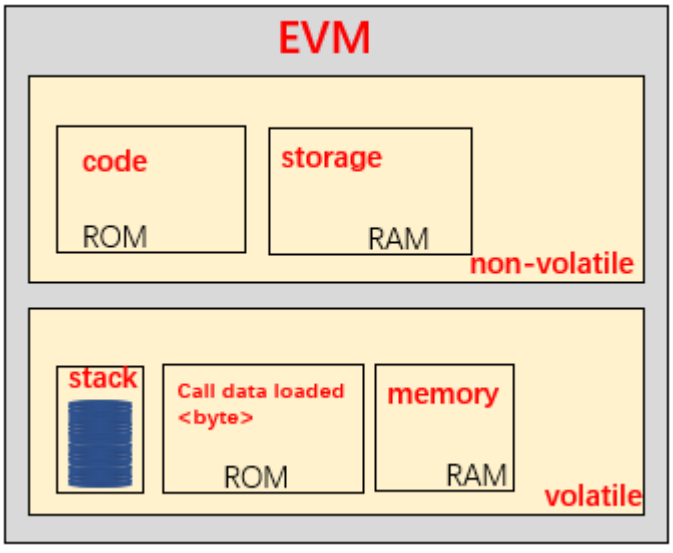
以太坊虚拟机是一个图灵完备的 256 位虚拟机，所谓图灵完备是指：一切可计算的问题都能计算，在可计算理论中，当一组数据操作的规则（一组指令集，编程语言，或者元胞自动机）满足任意数据按照一定的顺序可以计算出结果。但

是为了防止恶意用户设计无限循环的代码致使虚拟机运行瘫痪，以太坊虚拟机中执行的代码严格收到 gas 的制约。这规定了可运行的计算指令的数量上限，即使出现无限循环也会因为 gas 的耗尽而终止。对于 gas 的具体消耗，详细请见第五章。

以太坊虚拟机是一个简单的基于栈的结构，主要遵循后进先出原则。



- EVM 中每个栈的项的大小为 256 位，即虚拟机的位宽为 256 位。所谓位宽就是内存或显存一次能传输的数据量。简单地讲就是一次能传递的数据宽度。这样设计主要目的是能够方便应用于 256 位 Keccak 哈希算法和椭圆曲线的计算。栈的存储 storage 是一个基于字段地址的数组，其最大包含 1024 个元素，每个元素为 256 位。
- 栈的访问只限于其顶端，允许复制最顶端的 16 个元素中的一个到栈顶，或者是交换栈顶元素和下面 16 个元素中的一个。所有其他操作只能去最顶的一个或者几个元素，并将结果压入栈顶。（再查查）
- EVM 有内存 memory model：各项可以按照可寻址字节数组来存储。内存是易失性（volatile）的，内存的数据不是长久保存的。
- EVM 有一个独立的 storage model：与内存不一样，存储器是非易失性（non-volatile）的，即当虚拟机不运行时，其所存储的数据也不会丢失，并且该存储中的记录会作为整个以太坊系统状态的一部分进行维护。
- 虚拟机的存储 storage 与内存 memory 初始时都被设置为 0
- EVM 与标准的冯诺依曼架构不同，它将程序代码存储在虚拟 ROM 中，并通过特殊的指令来访问，而不是将程序代码存储在一般可访问的 memory 或者 storage 中。



◆ EVM 可以处理一些异常的执行指令，如：栈溢出，无效指令，out-of-gas。如果碰到这些指令，虚拟机会立即停止工作并将问题报告给执行代理（要么是交易处理器或者运行环境的代理程序），执行代理会单独处理。

4、智能合约编写与编译

4.1、以太坊智能合约的开发语言

以太坊具有三种编写智能合约的语言：Solidity、Serpent 和 LLL。其中 Solidity 是一种与 JavaScript 相似的语言，也是目前以太坊最受欢迎的语言。Serpent 是一种与 Python 类似的语言，LLL 是一种与 Assemly 类似的低级语言。

4.2、智能合约的编译

Solidity 合约的编译可以通过很多机制来完成，具体有如下几种方法：

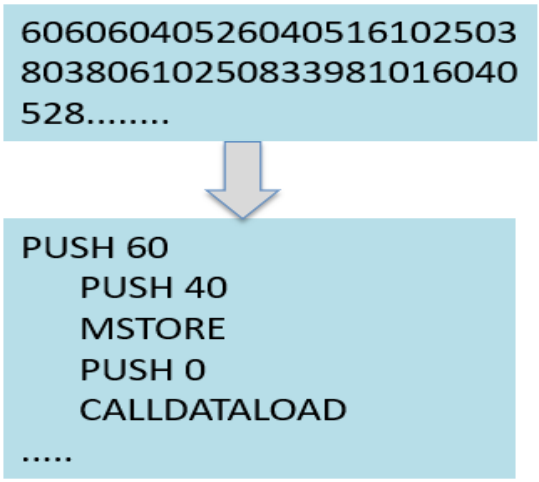
- 通过命令行使用 solc 编译器实现
- 在 geth 或 eth 提供的 JavaScript 控制台使用 web3.eh.compile.solidity（这里仍然需要安装 solc 编译器）实现
- 通过在线 Solidity 实时编译器实现
- 通过建立 Solidity 合约的 Meteor dapp Cosmo 实现
- 通过 Mix IDE 实现
- 通过以太坊钱包实现

举例如下：下面是一段用 Solidity 编写的智能合约

```
contract Greetings {
    string greeting;
    function Greetings (string _greeting) public {
        greeting = _greeting;
    }

    /* main function */
    function greet() constant returns (string) {
        return greeting;
    }
}
```

通过编译器编译之后，我们在区块链上看到的实际是编译之后的 bytecode，即所有的。即：所有的二进制数据都以十六进制的格式进行序列化。

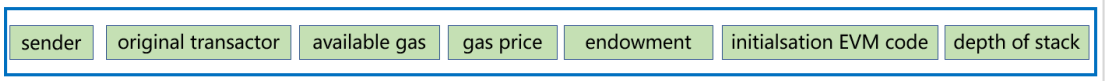


5、智能合约部署流程

智能合约的部署过程实际上就是创建合约账户（contract creation）的过程。这里我们假设全节点 A 想要部署智能合约（也就是为智能合约创建合约账户）。

5.1、 首先全节点 A 连接到以太坊网络（如何连接到网络需要具体详细说明），并同步自己想要的区块链

5.2、 同步完成后可以创建自己的外部账户，（涉及密钥备份的问题）并由外部账户发起创建合约账户的交易（具体可以参考第四章“合约创建”）。创建一个合约的时候需要以下几个固有的参数：

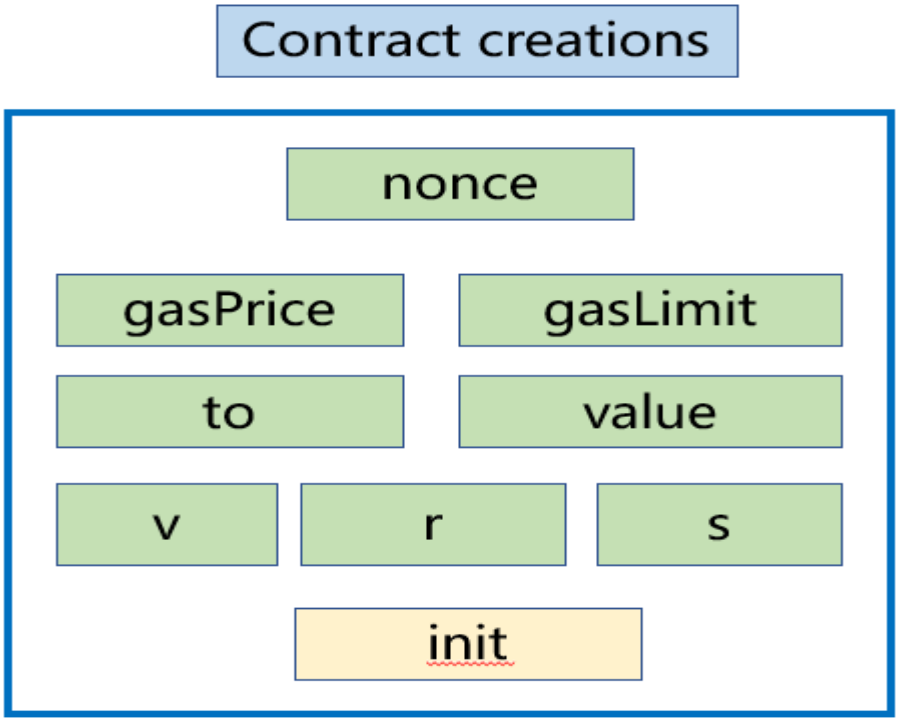


5.3、 现在假设全节点 A 想要部署智能合约，他先用 Solidity 编写好智能合约，并通过 4.2 提及的方法进行将智能合约编译成 bytecode。

5.4、 全节点 A 发送 contract creation 交易（也就是将 bytecode 作为数据给空地址发送交易），并进行全网广播（P2P）。

这一步是需要支付执行的（消耗 gas），根据第二章第 5 节中需要消耗 gas 三种情形，我们知道 contract creation 是需要消耗 gas 的。

假设该节点发送的交易为 Tx-1，此时 nonce 值将加 1，由于合约创建操作是要消耗 gas 的，因此节点 A 会设置 gasPrice 与 gasLimit 字段，并且由于此时合约账户地址还并未创建，to 字段为空。init code 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 storage，也可能是创建另一个合约账户，或者发起另一个消息通信 message call 等等。



5.5、 此时以太坊网络中的全节点也会依据综合标准对每个未确认的交易进行独立校验。每个验证节点会运行以太坊虚拟机（EVM）（**可以参照第六章执行模型和本章第 3 小节**），我们知道协议实际操作交易处理的部分是以太坊虚拟机，验证节点会根据收到的交易的字段，在本地 EVM 中执行相应的操作。首先处理器会确认以下信息是否有效和是否可获取(画图)

- 系统状态 system state
- 用于计算的剩余 gas
- 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 gas price
- 此次执行的输入数据
- 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
- 执行刚开始时，内存和堆栈都是空的，程序计数器为 0。

然后 EVM 开始递归的执行交易，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（machine state）包含:

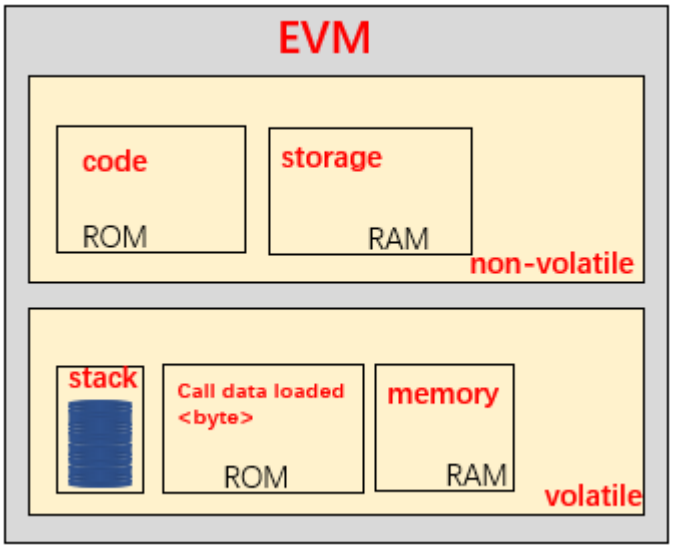
- 可获取的 gas
- 程序计数器
- 内存的内容

- 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。

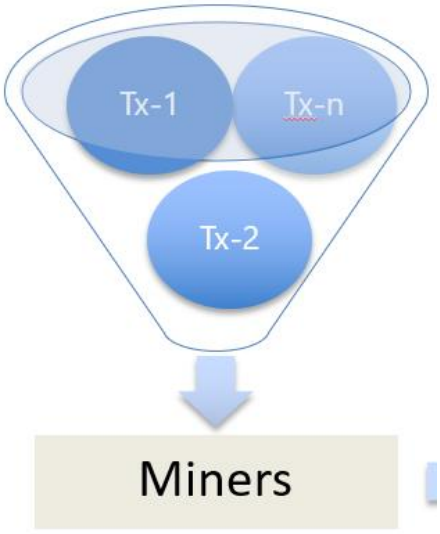
每个循环，剩余的 gas 都会被减少相应的量，程序计数器也会增加。
在每个循环的结束，都有三种可能性：

- 机器到达异常状态（例如 gas 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）

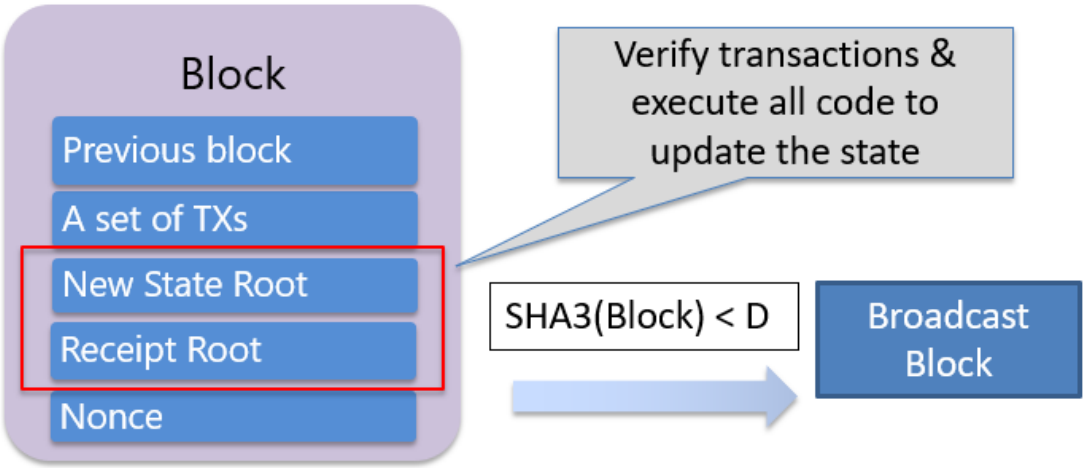
假设执行没有遇到异常状态，达到一个“可控的”或正常的停止，机器就会产生一个合成状态，执行之后的剩余 gas、产生的子状态、以及组合输出。



5.6、 以太坊网络中矿工收到交易后，验证每一笔交易并将收到的交易放到自己的交易池（内存池）中。交易池主要用来暂时存放尚未被加入到区块的交

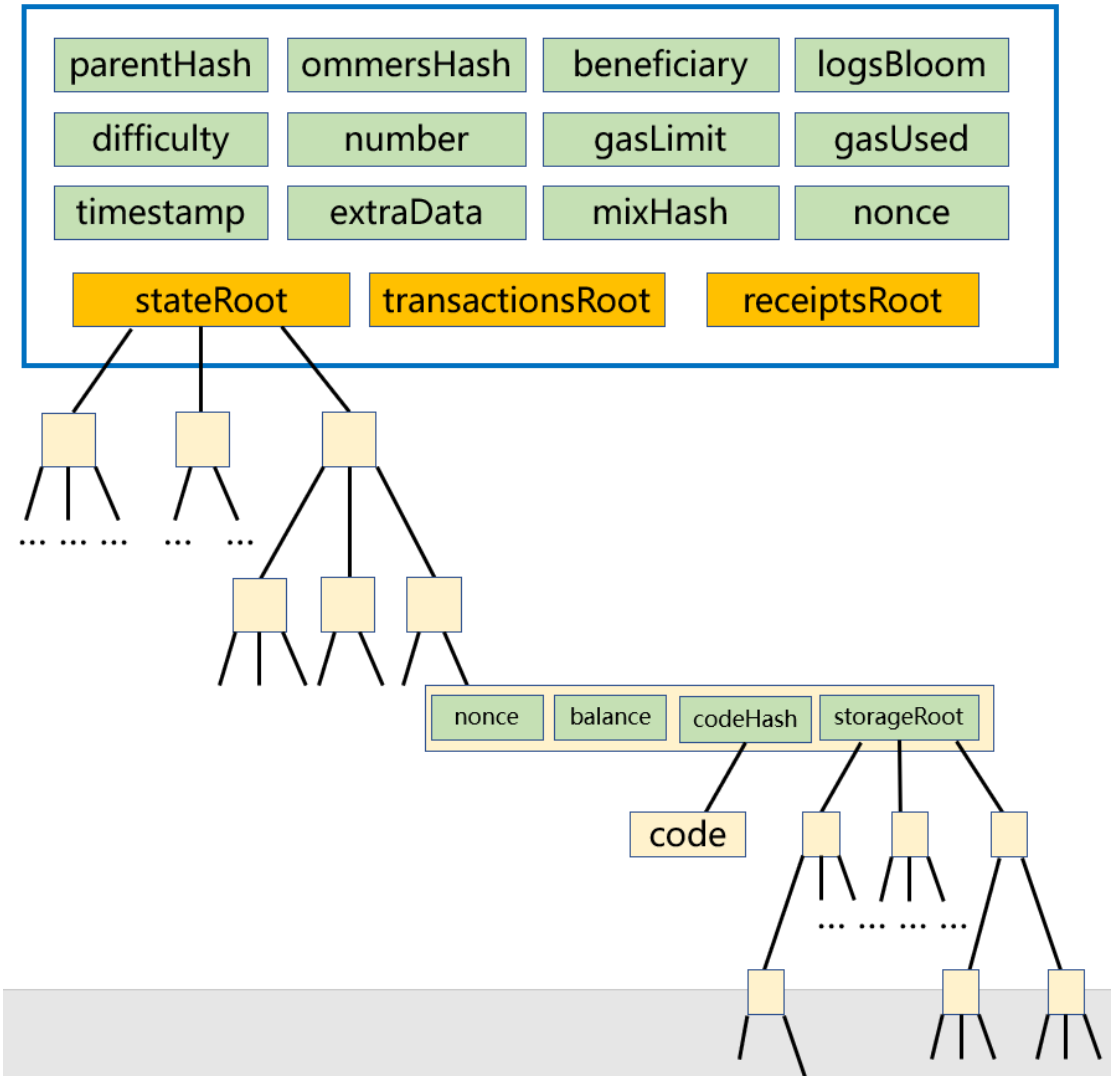


易记录，因此矿工会收集、验证新的交易，然后矿工会完成工作量证明算法的验证（具体参照工作量证明机制），之后矿工会广播区块。

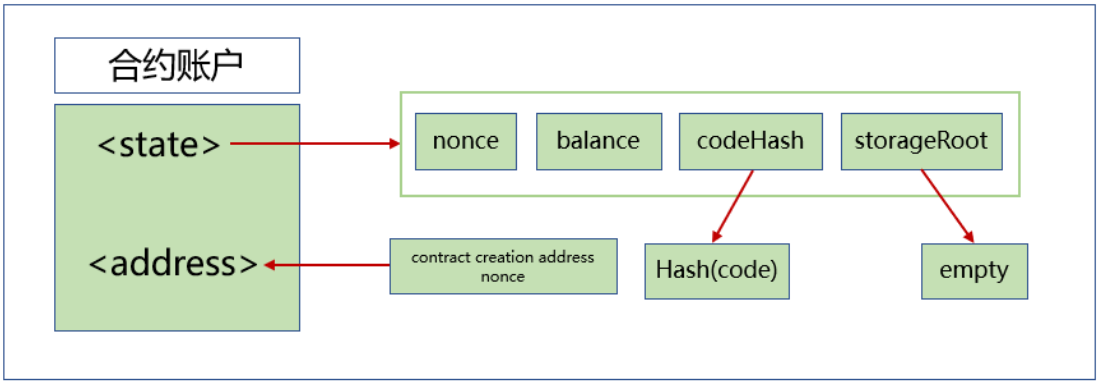


5.7、 网络中其他节点验证新生成的区块，验证通过则开始下一个区块的争夺。主要分为以下四个步骤（具体可参照第 8 章）

- 验证 ommers：在区块头中的每个 ommer 都必须是有效的并且必须在当前块往上 6 代之内
- 验证交易：区块中总共使用的 gas 必须与最后交易累加起来使用的 gas 是一样的。
- 申请奖励（挖矿奖励）：主要包含两方面的奖励
  - 静态奖励：矿工根据挖矿所得的 5 个以太币作为奖励
  - 动态奖励：挖矿所得的交易费归矿工所有，如果区块中包含叔区块，那么矿工还能从每个叔区块中获得额外的 1/32 以太币作为奖励，但是每个区块中最多只能包含 2 个叔区块。
- 验证 state 和 nonce：验证区块链的工作量证明，并确认将新区块链连接在权威的区块链上，并将整个系统更新到最新的状态。以太坊中要求交易必须在 12 个区块产生之后才能得到最终确认。因此当 12 个区块确认完成时，合约才能被真正保存到区块链中，即部署到以太坊的网络上，并可以被调用。



5.8、 合约部署成功后，会返回合约的地址，也就是新创建的合约账户的地址。改地址是由全节点 A 发送 contract creation 交易时账户地址与该地址发送过交易数 nonce 计算得来。以及由 web3.js 库提供的 API 接口。web3.js 是以太坊提供的一个 Javascript 库，它封装了以太坊的 JSON RPC API，提供了一系列与区块链交互的 Javascript 对象和函数，包括查看网络状态，查看本地账户、查看交易和区块、发送交易、编译/部署智能合约、调用智能合约等，其中最重要的就是与智能合约交互的 API。而合约的调用则会需要（合约地址，合约接口）。

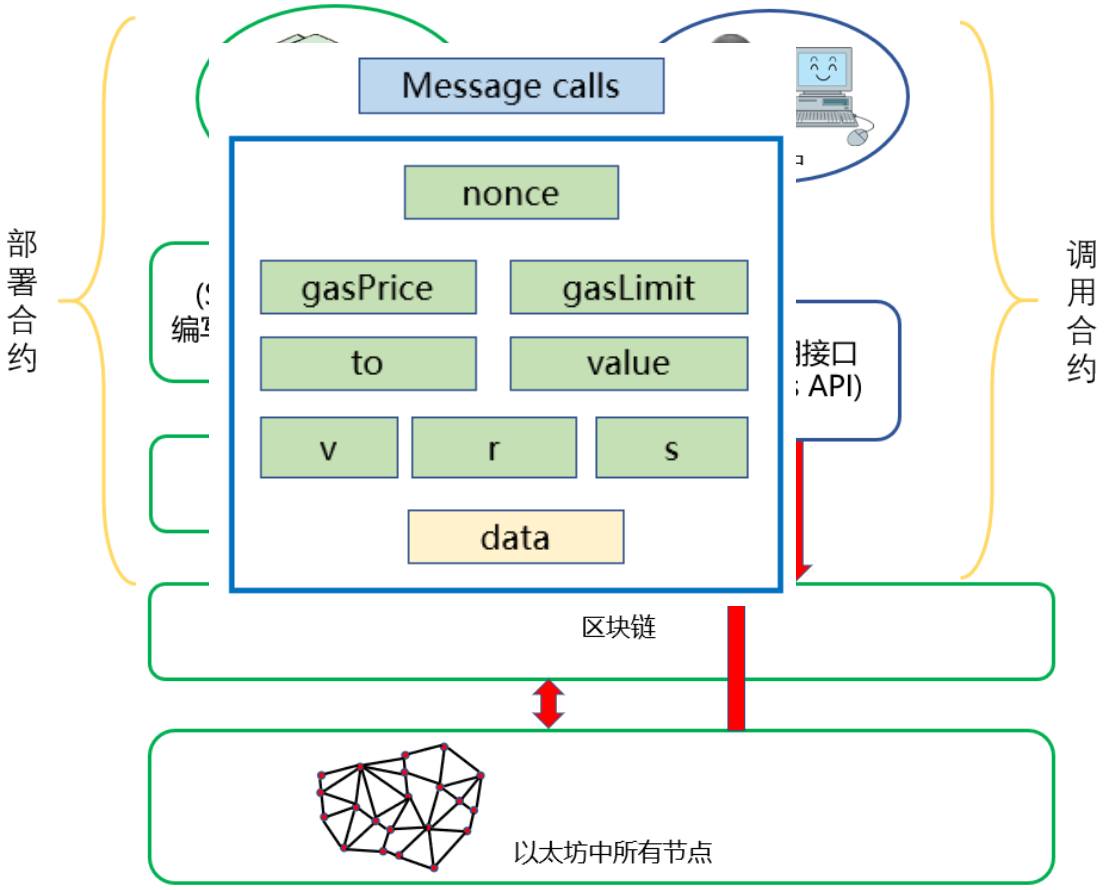


注意：合约的拥有者是可以调用智能合约自带的自毁程序的，因为一个交易需要被发送到网络上是需要支付费用的，自毁程序是对网络的补充，花费的费用远小于一个常用的交易。一旦自毁程序调用后，该地址剩余的以太币就会返回到设定的账户中，storage 和代码都从状态中移除了。

6、智能合约调用

如果用户希望在另一个以太坊节点调用刚刚创建的智能合约，合约的调用主要使用如下方法：

发送调用合约的交易，此时 data 字段包括了调用合约的哪个方法和哪个参数的负载量。此时我们需要使用 web3.js 库提供的 API 接口。根据合约的地址以及对应的 API 接口进行调用。 与创建合约一样，经过 12 个区块的确认之后，该交易会被写入到区块链上。



十二、 以太坊上安全性问题

1、以太坊钱包安全性问题：Parity 多签钱包

2、