以太坊：一种安全去中心化的通用交易账本

EIP-150 版本 (db2f7d7 - 2017-09-17)

原文作者: DR. GAVIN WOOD, GAVIN@ETHCORE.IO
译者: 崔广斌, ROOT@CUIGUANGBIN.COM; 高天露, TIANLU.JORDEN.GAO@GMAIL.COM

Abstract. 区块链对交易数据加密以保证安全, 已经通过一系列项目展示了它的实用性, 尤其是比特币。每一个这个的项目都可以看作是一个基于去中心化的单实例且拥有计算资源的应用。我们称这种模式为可以共享状态的单例状态机。

以太坊以更广义的方式实现了这种模式。它提供了大量的资源, 每一个资源都拥有独立的状态和操作码, 并且可以通过消息传递方式和其它资源交互。我们讨论了它的设计、实现难题、它提供的机会以及以后可能有的一些问题。

1. 简介

随着互联网连接了世界上绝大多数地方，全球信息共享的成本越来越低。比特币网络通过共识机制、自愿遵守的社会合约, 实现一个去中心化的价值转移系统且可以在全球范围内自由使用, 这样的技术改革展示了它的巨大力量。这样的系统可以说是加密安全、基于交易的状态机的一种具体应用。后续类似这样的系统, 如域名币（Namecoin）, 从最原先的" 货币应用" 发展到了其它应用, 虽然它只是其中很简单的一种应用。

以太坊是一个尝试实现通用性技术的项目, 所有基于交易的状态机都可以被构建。而且以太坊致力于为开发者提供主流一个紧凑的、整合的端到端系统, 这个系统提供了一种可信的消息传递计算框架让开发者以一种前所未有的范式来构建软件。

1.1. 驱动因素. 这个项目有很多目标, 其中最重要的目标是为了促成不信任对方的个体之间的交易。这些不信任可能是因为地理位置分离、接口对接难度, 或者是不兼容、不称职、不情愿、支出、不确定、不方便, 或现有法律系统的腐败。于是我们想用一个丰富且清晰的语言去实现一个状态变化的系统，期望协议可以自动被执行, 我们可以为此提供一种实现方式。

这个提议系统中的交易, 有一些在现实世界中并不常见的属性。审判廉洁，在现实世界往往很难找到，但对公正的算法解释器是天然的; 透明，或者说通过交易日志和规则或代码指令能够清晰的看见到状态变化或者判决, 但因为人类语言的模糊性、信息的缺乏以及老的偏见难以撼动, 导致基于人的系统中从来没有完美实现透明。

总的来说，我们希望能提供一个系统，能够保证用户无论是和其他个体、系统还是组织交互, 都能对可能的结果及产生结果的过程完全信任。

1.2. 前人工作. ? 在 2013 年 9 月下旬第一次提出了这种系统的核心机制。虽然现在发展出了多种方案, 但最关键的部分, 具备图灵完备语言且不受限制的内部交易存贮容量的区块链, 仍未变化。

? 提出了一种使用计算支出的密码学证明方式 (proof-of-work, 工作量证明) 在互联网上传递信号值。信号值用作阻挡垃圾邮件，而不是任何一种货币, 但展示了一个基本的数据通道可以承载强大经济信号的可能性, 允许接受者无需依赖信任而做出物理断言。? 后来设计了一个类似的系统。

? 最早使用工作量证明作为强大的经济信号保证货币安全。在这个案例中, 代币用作检查点对点 (peer-to-peer,p2p) 文件交易，同时保证“消费者”能支付给为他们提供服务的“供应商”。这种通过工作量证明的安全模型逐步扩展, 包括使用电子签名和账本技术, 以保证历史记录不被篡改, 怀有恶意的用户不能进行欺诈支付或不公平的抱怨服务。五年后（2008 年），中本聪? 介绍了另一种更广泛的工作量证明安全价值代币。这个项目的成果比特币, 成为了第一个被全球广泛认可的去中心化交易账本。

由于比特币的成功，竞争币 (alt-coins) 开始兴起, 通过改变比特币的协议去创建了大量的其他数字货币。比较知名的有莱特币（Litecoin）和素数币（Primecoin）, 参见 ? 。一些项目使用比特币的核心机制并重新改造以应用在其它领域, 例如域名币（Namecoin）, 致力于提供一个去中心化的名字解析系统, 参见 ? 。

其它在比特币网络之上构建的项目, 也是依赖巨大的系统价值和巨大的算力来保证共识机制。万事达币（Mastercoin）项目, 在比特币协议之上, 通过一系列基于核心协议的的辅助插件, 构建一个包含许多高级功能的富协议, 参见 ?。彩色币（Coloured Coins, 参见 ?, 采用了类似的但更简化的协议, 以实现比特币基础货币的可替代性, 并允许通过色度钱包（“chroma-wallet”）来创建和跟踪代币。

其它一些工作通过放弃中心化来进行。瑞波币（Ripple）, 参见 ?, 试图去创建一个货币兑换的联邦系统（“federated ” system）和一个新的金融清算系统。这个系统展示了放弃去中心化特性可以获得性能上的提升。

? 和 ? 进行了智能合约（smart contract）的早期工作。大约在上世纪 90 年代，人们逐渐认识到协议算法的执行可以成为人类合作的重要力量。虽然当时没有这样的系统, 但可以预见未来的法律将会受到这种系统的影响。基于此, 以太坊或许可以成为这种密码学-法律系统的通用实现。

2. 区块链

以太坊在整体上可以看成是一个基于交易的状态机：我们起始于一个创世块（Genesis）状态，然后随着交易的执行状态逐步改变一直到最终状态, 这个最终状态是以太坊世界的权威版本。状态中包含的信息有: 账户余额、名誉度、信誉度、现实世界的附属数据等; 简而言之，能包含电脑可以描绘的任何信息。因此，交易是连接两个状态的有效桥梁；“有效”非常重要—因为无效的状态改变远超过有效的状态改变。例如：无效的状态改变可能是减少账号余额，但是没有在其它账户上加上同等的额度。一个有效的状态转换是通过交易进行的, 表达式如下:

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \tag{1}$$

Υ 是以太坊状态转换函数。在以太坊中, Υ 和 $\boldsymbol{\sigma}$ 比已有的任何比较系统都强; Υ 可以执行任意计算, 而 $\boldsymbol{\sigma}$ 可以存贮交易中的任意状态。

区块中记录着交易信息；区块之间通过密码学哈希 (hash) 链接起来。区块链就像一个分类账, 将一系列交易记录在一起, 并且连接上一个区块及最终状态 (并没有直接保存最终状态本身—否则整个区块链就太大了)。系统激

励节点去挖矿, 挖矿激励时, 有执行状态转移函数, 增加挖矿者的账户余额。

挖矿是和其它潜在区块竞争一系列交易 (一个区块) 的记账权。它是通过密码安全证明的方式来实现的。这个机制称为工作量证明, 会在 **??** 详细讨论。

公式如下:

$$\boldsymbol{\sigma}_{t+1} \equiv \Pi(\boldsymbol{\sigma}_t, B) \tag{2}$$

$$B \equiv (..., (T_0, T_1, ...)) \tag{3}$$

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)...) \tag{4}$$

其中 Ω 是区块定稿状态转换函数（这个函数奖励一个特定的账户）; B 表示包含一系列交易的区块; Π 是区块级的状态转换函数。

上述是区块链的基本内容，这个模型不仅是以太坊的基础，还是迄今为止所有基于共识的去中心化交易系统的基础。

2.1. 面值**.** 为了激励网络中的计算，需要定义一种转账方法。以太坊设计了一个内置货币以太币 (Ether)，ETH 是大家所熟知的符号，有时用 Ð 表示。以太币最小的面额是 Wei(伟)，所有货币值都以 Wei 的整数倍记录。一个以太币被定义成位。一个以太币等于 10^{18} Wei 。不同的面值如下表:

倍数	面值
10^0	Wei(伟)
10^{12}	Szabo(萨博)
10^{15}	Finney(芬尼)
10^{18}	Ether(以太)

在整个工作中，任何涉及到价值、以太币相关的、货币、余额或者支付，都以 Wei 作为单位来计算。

2.2. 历史**?** 因为这是一个去中心化的系统, 所有人都有机会在之前的某一个区块创建新的区块并连接在其后, 这会行成一个树状的区块。为了能在这个树状结构上从根节点（创世块）到叶子节点（包含最新交易的区块）能形成一个一致的区块链, 必须有一个共识方案。如果有人认为从根节点到叶子节点的路径不是" 最佳" 的区块链, 那这时候就会发生分叉。

这个就意味着在一个给定的时间点, 系统中会有多个状态共存: 一些节点相信一个区块是包含标准的交易，其他的节点则相信另外一些区块包含标准的交易，其中就包含彻底不同或者不兼容的交易。这一点必须要避免, 因为它会破坏整个系统信用。

我们使用了一个简单 GHOST 协议版本来达成共识, 参见 **?** 。我们会在**??** 详细说明。

有时会从一个特定的区块链高度启用新的协议。本文描述了协议的一个版本, 如果要跟踪历史区块链路径, 可能需要查看这份文档的历史版本。(译者注: 可以到 https://github.com/ethereum/yellowpaper 查看英文版历史版本)

3. 约定

我用了大量的印刷约定表示公式中的符号，其中一些需要特别说明:

有两个高度结构化的顶层状态值, 使用粗体小写希腊字母: $\boldsymbol{\sigma}$ 表示世界状态 (world-state); $\boldsymbol{\mu}$ 表示机器状态 (machine-state)。

作用在高度结构化值上的函数，使用大写的希腊字母，例如: Υ , 是以太坊中的状态转换函数。

对于大部分函数来说，通常用一个大写的字母表示，例如: C, 表示费用函数。可能会使用下角标表示为一些特别的变量，例如: C_{SSTORE}，表示执行 SSTORE 操作的费用函数。对于一些可能是外部定义的函数，可能会使用打印机文字字体，例如: KEC512 哈希函数 (为赢得进入 SHA-3 竞赛) , 使用 KEC 表示。KEC512 表示 Keccack-512 哈希函数。

元组通常使用一个大写字母，例如: T，表示一个以太坊交易。使用下标可能会表示一个独立的变量, 例如: T_n ，表示交易中随机数。角标的形式用于表示它们的类型; 例如: 大写的下角标表示元组包含的下角标变量。

标量和固定大小的字节序列（或数组）都使用小写字母来表示，例如: n 在本文中表示交易随机数。小写的希腊字母一般表示一些特别的含义，例如: δ 表示在栈上一个给定操作需要的条目数量。

任意长度的序列通常用加粗的小写字母表示，例如 $\mathbf{o}$ 表示消息调用中输出的数据字节序列。有时候，会对特别重要的值使用粗体。

我们认为标量都是正整数且属于集合 $\mathbb{P}$。所有的字节序列属于集合 $\mathbb{B}$，附录 **??** 给出了正式的定义。用下角标符号表示这样的序列集合限制在一定长度以内, 长度为 32 的字节序列使用 $\mathbb{B}_{32}$ 表示, 所有比 2^{256} 小的正整数使用 $\mathbb{P}_{256}$ 表示。详细定义见 4.3。

使用方括号表示序列中的一个元素或子序列, 例如: $\boldsymbol{\mu}_{\mathbf{s}}[0]$ 表示计算机堆栈中的第一个条目。对于子序列来说，使用省略号表示一定的范围，且含头尾的限制，例如: $\boldsymbol{\mu}_{\mathbf{m}}[0..31]$ 表示计算机内存中的前 32 个条目。

在全局状态的情况下， 是一个含多个账号的序列，本身的数组，正方形括号被用作去表示一个单独的账号。

以全局状态 $\boldsymbol{\sigma}$ 为例, 它表示一系列的账户, 它们自身的元组, 方括号用于表示一个独立的账户。

当去考虑现有的变量时，我遵循在给定的范围内去定义的原则，我们使用占位符 $\square$ 表示未修改的输入变量, 使用 $\square'$ 表示修改的和可用的变量, $\square^*$, $\square^{**}$ &c 表示中间变量。在特殊情况下，为了提高可读性和清晰性，我可能会使用字母-数字下角标表示中间值。

当使用去已有的函数时，给定一个函数 f, 那么函数 f^* 表示一个相似的、替换序列的函数映射。详细的定义见 4.3。

整个过程中，我定义了大量的函数。一个常见的函数是 ℓ，表示给定序列的最后一个条目:

$$\ell(\mathbf{x}) \equiv \mathbf{x}[\|\mathbf{x}\| - 1] \tag{5}$$

4. 块、状态和交易

介绍了以太坊的基本概念后, 我们将更详细地讨论交易、区块和状态的含义

4.1. 世界状态**.** 世界状态是在地址（160 位的标志符）和账户状态（序列化为 RLP 的数据结构，详见附录 **??**）的映射。虽然世界状态没有直接储存在区块链上，但会假定实施过程中会将这个映射维护在一个修改过的 Merkle Patricia 树 (简称 trie，详见附录 **??**)。trie 需要一个简单的后端数据库去维护字节数组到字节数组的映射; 我们称这个后端数据库为状态数据库。它有一系列的好处: 第一这个结构的根节点是加密的且依赖于所有的内部数据，且它的哈希可以作为整个系统状态的一个安全标志; 第二，作为一个不变的数据结构，因此它允许任何一个之前状态（根部哈希已知的条件下）通过简单地改变根部哈希值而被召回。因为我们在区块链中储存了所以这样的根部哈希值，所以我们能恢复到指定的历史状态。

账户状态包含以下四个字段:

nonce, 随机数**:** 这个值等于账户发出的交易数及这个账户创建的合约数量之和。$\boldsymbol{\sigma}[a]_n$ 表示状态 $\boldsymbol{\sigma}$ 中的地址 a 的 nonce 值。

balance, 余额**:** $\boldsymbol{\sigma}[a]_b$, 表示这个账户拥有多少 Wei。

storageRoot, 存储跟节点**:** 保存账户内容的 Merkle Patricia 树根节点的 256 位哈希编码到 trie 中, 作

为从 256 位整数键值哈希的 Keccak 256 位哈希到 256 位整数的 RLP-编码映射。这个哈希定义为 $\boldsymbol{\sigma}[a]_s$。

codeHash, 代码哈希**:** 这个账户的 EVM(Ethereum Virtual Machine, 以太坊虚拟机) 代码的哈希值—代码执行时, 这个地址会接收一个消息调用; 它和其它字段不同, 创建后不可更改。状态数据库中包含所有像这样的代码片段的哈希, 以便后续使用。这个哈希定义为 $\boldsymbol{\sigma}[a]_c$,$\mathbf{b}$ 表示代码, $\texttt{KEC}(\mathbf{b}) = \boldsymbol{\sigma}[a]_c$.

因为我通常希望所指的并不是 trie 树的根哈希, 而是所保存的键值对集合，我做了一个更方便的定义:

$$\texttt{TRIE}\big(L_I^*(\boldsymbol{\sigma}[a]_{\mathbf{s}})\big) \equiv \boldsymbol{\sigma}[a]_s \tag{6}$$

trie 树中的键值对集合函数, L_I^*, 定义为基于基础函数 L_I 的元素转换:

$$L_I\big((k, v)\big) \equiv \big(\texttt{KEC}(k), \texttt{RLP}(v)\big) \tag{7}$$

其中:

$$k \in \mathbb{B}_{32} \quad \wedge \quad v \in \mathbb{P} \tag{8}$$

需要说明的是,$\boldsymbol{\sigma}[a]_{\mathbf{s}}$ 不应算作这个账户的" 物理" 成员, 它不参与序列化。

如果 **codeHash** 字段是一个空字符串的 Keccak-256 哈希, 例如 $\boldsymbol{\sigma}[a]_c = \texttt{KEC}(())$, 则表示对应的节点表示一个简单账户, 有时简称" 非合约" 账户。

因此我们可能定义一个世界状态的函数 L_S:

$$L_S(\boldsymbol{\sigma}) \equiv \{p(a) : \boldsymbol{\sigma}[a] \neq \varnothing\} \tag{9}$$

where

$$p(a) \equiv \big(\texttt{KEC}(a), \texttt{RLP}\big((\boldsymbol{\sigma}[a]_n, \boldsymbol{\sigma}[a]_b, \boldsymbol{\sigma}[a]_s, \boldsymbol{\sigma}[a]_c)\big)\big) \tag{10}$$

函数 L_S 和 trie 函数是为了提供一个世界状态的简短身份 (哈希)。我们假定:

$$\forall a : \boldsymbol{\sigma}[a] = \varnothing \ \vee \ (a \in \mathbb{B}_{20} \ \wedge \ v(\boldsymbol{\sigma}[a])) \tag{11}$$

v 是账户合法性验证函数:

$$v(x) \equiv x_n \in \mathbb{P}_{256} \wedge x_b \in \mathbb{P}_{256} \wedge x_s \in \mathbb{B}_{32} \wedge x_c \in \mathbb{B}_{32} \tag{12}$$

4.2. 交易**.** 交易 (符号, T) 是个单一的加密指令, 通过以太坊中系统之外的操作者创建。我们假设外部的操作者是人, 软件工具用于创建和传播[1]。这里的交易类型有两种: 一种是消息调用, 另一种通过代码创建新的账户 (称为“合约创建”)。两种类型的交易都有的共同字段如下:

nonce, 随机数**:** T_n, 账户发出的交易数量。

gasPrice, 燃料价格**:** T_p, 为执行交易所需要的计算资源付的 *gas* 价格, 以 Wei 为单位。

gasLimit, 燃料上限**:** T_g, 用于执行交易的最大 gas 数量。这个值须在交易前设置, 且设定后不能再修改。

to, 接收者地址**:** 消息调用接收者的 160 位的地址。对与合约创建交易, 无需接收者地址, 使用 $\varnothing$ 表示, $\varnothing$ 是 $\mathbb{B}_0$ 的唯一成员。

value, 转账额度**:** T_v, 转到接收者账户的额度, 以 Wei 为单位。对于合约创建, 表示捐赠到合约地址的额度。

v, r, s: T_w, T_r and T_s, 和交易签名相关的变量, 用于确定交易的发送者。详见附录 **??**。

此外, 合约创建还包含以下字段:

init, 初始化**:** $T_{\mathbf{i}}$, 一个不限制大小的字节数组, 表示账户初始化程序的 EVM 代码。

init 是 EVM 代码片段; 执行 init 后会返回另外一个代码片段, 每次合约接受消息调用 (通过交易或内部调用) 后都会执行这个代码片段。**init** 仅当合约账户创建的时候执行一次。

相比之下, 一个消息调用的交易包括:

data, 数据**:** $T_{\mathbf{d}}$, 一个不限制大小的字节数组, 表示消息调用的输入数据。

附录 **??** 详细描述了映射发送者交易的函数 S, 通过 SECP-256k1 的 ECDSA 曲线, 使用交易 (除了最后的 3 个签名字段) 作为数据来签名。目前我们先简单使用 $S(T)$ 表示发送者的指定交易 T 。

$$L_T(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{i}}, T_w, T_r, T_s) & \text{if } T_t = \varnothing \\ (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{d}}, T_w, T_r, T_s) & \text{otherwise} \end{cases} \tag{13}$$

在这里, 我们假设所有变量都是 RLP 编码的整数, 除了 2 个任意长度的字节数组 $T_{\mathbf{i}}$ 和 $T_{\mathbf{d}}$。

$$\begin{array}{lllllll} T_n \in \mathbb{P}_{256} & \wedge & T_v \in \mathbb{P}_{256} & \wedge & T_p \in \mathbb{P}_{256} & \wedge \\ T_g \in \mathbb{P}_{256} & \wedge & T_w \in \mathbb{P}_5 & \wedge & T_r \in \mathbb{P}_{256} & \wedge \\ T_s \in \mathbb{P}_{256} & \wedge & T_{\mathbf{d}} \in \mathbb{B} & \wedge & T_{\mathbf{i}} \in \mathbb{B} & \end{array} \tag{14}$$

其中

$$\mathbb{P}_n = \{P : P \in \mathbb{P} \wedge P < 2^n\} \tag{15}$$

地址哈希 $T_{\mathbf{t}}$ 稍微有些不同: 它是一个 20 字节的地址哈希值, 但创建合约时它是 RLP 空字节系列, 表示为 $\mathbb{B}_0$:

$$T_{\mathbf{t}} \in \begin{cases} \mathbb{B}_{20} & \text{if} \quad T_t \neq \varnothing \\ \mathbb{B}_0 & \text{otherwise} \end{cases} \tag{16}$$

4.3. 区块**.** 在以太坊中, 区块是相关信息的集合。区块头 H, 与之想对应的交易信息 $\mathbf{T}$, 其它区块头数据的集合 $\mathbf{U}$, $\mathbf{U}$ 表示它的父级区块中有和当前区块的爷爷辈区块是相同的。(这样的区块称为 *ommers*[2], 译者注: 不妨称之为叔链)。区块头包含的的信息如下:

parentHash, 父块哈希**:** H_p, 父区块头的 Keccak 256 位哈希。

ommersHash, 叔链哈希**:** H_o, 当前区块的叔链列表 Keccak 256 位哈希。

beneficiary, 受益者地址**:** H_c, 成功挖到这个区块的 160 位地址, 这个区块中的所有交易费用都会转到这个地址。

stateRoot, 状态字典树根节点哈希**:** 状态字典树根节点的 Keccak 256 位哈希, 交易打包到当前区块且区块定稿后可以生成这个值。

transactionsRoot, 交易字典树根节点哈希**:** 交易字典树根节点的 Keccak 256 位哈希, 在交易字典树含有区块中的所有交易列表。

receiptsRoot, 接受者字典树根节点哈希**:** 接受者字典树根节点的 Keccak 256 位哈希, 在接受者字典树含有区块中的所有交易信息中的接受者。

logsBloom, 日志 **Bloom:** H_b, 日记 Bloom 过滤器由可索引信息 (日志地址和日志主题) 组成, 这个信息包含在每个日志入口, 来自交易列表中的每个交易的接受者。

[1]显著地, 这样的“工具”可以从基于人类行为的初始化中移除—或者人类可能变成有原因的中立—可能有一点他们被视为自治的代理人。例如: 合约可能会给人类好处, 让人发送交易从而触发合约的执行。

[2]*ommer* 的意思和自然界中的" 父母的兄弟姐妹" 最相近, 详见 http://nonbinary.org/wiki/Gender_neutral_language#Family_Terms

difficulty, 难度：H_d, 表示当前区块的难度水平, 这个值根据前一个区块的难度水平和时间戳计算得到。

number, 区块编号：H_i, 等于当前区块的直系前辈区块数量。创始区块的区块编号为 0。

gasLimit, 燃料限制：H_l, 目前每个区块的燃料消耗上限。

gasUsed, 燃料使用量：H_g, 当前区块的所有交易使用燃料之和。

timestamp, 时间戳：H_s, 当前区块初始化时的 Unix 时间戳。

extraData, 附加数据：H_x, 32 字节以内的字节数组。

mixHash, 混合哈希：H_m, 与一个与随机数 (nonce) 相关的 256 位哈希计算, 用于证明针对当前区块已经完成了足够的计算。

nonce, 随机数：H_n, 一个 64 位哈希, 和计算混合哈希相关, 用于证明针对当前区块已经完成了足够的计算。

此外, 当前区块还记录着这个区块的交易列表, 以及 2 个叔链 (ommer) 的区块头列表。我们以 B 表示一个区块:

$$B \equiv (B_H, B_{\mathbf{T}}, B_{\mathbf{U}}) \tag{17}$$

4.3.1. 交易收据. 为了让交易信息编码能有利于零知识证明、索引、搜索, 我们将每个包含一定信息的交易收据进行编码。以 $B_{\mathbf{R}}[i]$ 表示第 i 个交易, 保存在一个索引字典树中,H_e 是这个字典树的根节点。为了去编译信息关于每一个有关系的交易而且可能是一个有用的工具去形成一个零知识证明 (zero-knowledge proof)，或者索引和搜索。我们编译每一个包含当前交易执行的某些信息为交易收据。每个收据 (在第个交易中表示为) 是被放置于一个带关键字索引的 trie 中和这个区块头中被记录下的根值，表示为。

交易收据是一个包含四个条目的元组：交易后的状态,$R_{\boldsymbol{\sigma}}$；当前区块中交易累计燃料使用量, R_u, 交易发生后会立即更新这个值; 交易执行过程中创建的日志集合,$R_{\mathbf{l}}$; 和日志 Bloom 过滤器,R_b :

$$R \equiv (R_{\boldsymbol{\sigma}}, R_u, R_b, R_{\mathbf{l}}) \tag{18}$$

函数 L_R 是一个将交易收据转换为 RLP 编码的预处理函数:

$$L_R(R) \equiv (\texttt{TRIE}(L_S(R_{\boldsymbol{\sigma}})), R_u, R_b, R_{\mathbf{l}}) \tag{19}$$

交易后状态 $R_{\boldsymbol{\sigma}}$ 会编码到一个字典树中, 字典树的根节点组成了第一个条目。

我们假定累计的燃料使用量 R_u 是一个正整数, 日志 Bloom R_b 是 2048 位 (256 字节) 的哈希:

$$R_u \in \mathbb{P} \quad \wedge \quad R_b \in \mathbb{B}_{256} \tag{20}$$

$R_{\mathbf{l}}$ 是一系列的日志入口, 例如 $(O_0, O_1, ...)$。一个日志入口 O 是一个日志记录器的地址 O_a 的元组, $O_{\mathbf{t}}$ 是一系列 32 字节的日志主题, $O_{\mathbf{d}}$ 是一些字节数据:

$$O \equiv (O_a, (O_{\mathbf{t}0}, O_{\mathbf{t}1}, ...), O_{\mathbf{d}}) \tag{21}$$

$$O_a \in \mathbb{B}_{20} \quad \wedge \quad \forall_{t \in O_{\mathbf{t}}} : t \in \mathbb{B}_{32} \quad \wedge \quad O_{\mathbf{d}} \in \mathbb{B} \tag{22}$$

我们定义 Bloom 过滤器函数 M 将一个日志入口转换为一个 256 字节哈希:

$$M(O) \equiv \bigvee_{t \in \{O_a\} \cup O_{\mathbf{t}}} \left(M_{3:2048}(t)\right) \tag{23}$$

其中 $M_{3:2048}$ 是一个特别的 Bloom 过滤器, 针对任意一个字节序列, 它舍弃这个字节序列 2048 位的前三位。它通过对一个字节序列的 Keccak-256 哈希的每一个前三对字节取其的低 11 位来实现:

$$M_{3:2048}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) \equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{256} \quad \text{where:} \tag{24}$$
$$\mathbf{y} = (0, 0, ..., 0) \quad \text{except:} \tag{25}$$
$$\forall_{i \in \{0,2,4\}} : \mathcal{B}_{m(\mathbf{x},i)}(\mathbf{y}) = 1 \tag{26}$$
$$m(\mathbf{x}, i) \equiv \texttt{KEC}(\mathbf{x})[i, i+1] \bmod 2048 \tag{27}$$

其中 $\mathcal{B}$ 是位引用函数, $\mathcal{B}_j(\mathbf{x})$ 等于字节数组 $\mathbf{x}$ 中的索引 j(从 0 开始索引) 的位。

4.3.2. 整体有效性. 如果一个区块同时满足以下几个条件, 我们才能认为这个区块是有效的：当从起始状态 $\boldsymbol{\sigma}$ (父块的最终状态) 按顺序执行完生成新的状态 H_r 后, 在内部上要保持一致, 包括叔链、交易区块哈希、给定的交易 $B_{\mathbf{T}}$（详细描述见 **??**）：

$$\begin{aligned}
H_r &\equiv \texttt{TRIE}(L_S(\Pi(\boldsymbol{\sigma}, B))) &\wedge\\
H_o &\equiv \texttt{KEC}(\texttt{RLP}(L^*_H(B_{\mathbf{U}}))) &\wedge\\
H_t &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{P} : p(i, L_T(B_{\mathbf{T}}[i]))\}) &\wedge\\
H_e &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{R}}\|, i \in \mathbb{P} : p(i, L_R(B_{\mathbf{R}}[i]))\}) &\wedge\\
H_b &\equiv \bigvee_{\mathbf{r} \in B_{\mathbf{R}}} (\mathbf{r}_b)
\end{aligned} \tag{28}$$

其中 $p(k, v)$ 是 RLP 的简单对转换, 在这个例子中, k 为这个区块中的交易索引,v 为交易收据:

$$p(k, v) \equiv \big(\texttt{RLP}(k), \texttt{RLP}(v)\big) \tag{29}$$

此外:

$$\texttt{TRIE}(L_S(\boldsymbol{\sigma})) = P(B_H)_{H_r} \tag{30}$$

$\texttt{TRIE}(L_S(\boldsymbol{\sigma}))$ 是包含以 RLP 编码的状态 $\boldsymbol{\sigma}$ 键值对的 Merkle Patricia 树根节点哈希, $P(B_H)$ 是父节点。

这些值根据交易计算产生，特别是交易收据 $B_{\mathbf{R}}$, 这个通过交易状态累积函数 Π 定义，在 **??** 会详细说明。

4.3.3. 序列化. 函数 L_B 和 L_H 分别是区块和区块头的准备函数。类似交易收据准备函数 L_R，当转换为 RLP 格式时, 假设对应的类型、顺序及结构如下:

$$\begin{aligned}
L_H(H) \equiv (\ & H_p, H_o, H_c, H_r, H_t, H_e, H_b, H_d,\\
& H_i, H_l, H_g, H_s, H_x, H_m, H_n\ )
\end{aligned} \tag{31}$$
$$L_B(B) \equiv \big(L_H(B_H), L^*_T(B_{\mathbf{T}}), L^*_H(B_{\mathbf{U}})\big) \tag{32}$$

其中 L^*_T 和 L^*_H 是元素序列转换函数, 因此:

$$f^*\big((x_0, x_1, ...)\big) \equiv \big(f(x_0), f(x_1), ...\big) \quad \text{对于任何函数 } f \tag{33}$$

元素类型定义如下:

$$\begin{aligned}
&H_p \in \mathbb{B}_{32} \wedge H_o \in \mathbb{B}_{32} \wedge H_c \in \mathbb{B}_{20} \wedge\\
&H_r \in \mathbb{B}_{32} \wedge H_t \in \mathbb{B}_{32} \wedge H_e \in \mathbb{B}_{32} \wedge\\
&H_b \in \mathbb{B}_{256} \wedge H_d \in \mathbb{P} \wedge H_i \in \mathbb{P} \wedge\\
&H_l \in \mathbb{P} \wedge H_g \in \mathbb{P} \wedge H_s \in \mathbb{P}_{256} \wedge\\
&H_x \in \mathbb{B} \wedge H_m \in \mathbb{B}_{32} \wedge H_n \in \mathbb{B}_8
\end{aligned} \tag{34}$$

其中

$$\mathbb{B}_n = \{B : B \in \mathbb{B} \wedge \|B\| = n\} \tag{35}$$

我们现在有了一个严密正式的区块结构结构说明。RLP 函数（见附录 **??**）提供了一个标准方法来把这个结构转换为一个字节序列。

4.3.4. 区块头验证. 我们定义 $P(B_H)$ 为 B 的父区块::

$$P(H) \equiv B' : \texttt{KEC}(\texttt{RLP}(B'_H)) = H_p \tag{36}$$

当前区块编号等于它的父块编号加 1:

$$H_i \equiv P(H)_{H_i} + 1 \tag{37}$$

区块难度定义为 $D(H)$:

$$D(H) \equiv \begin{cases} D_0 & \text{if} \quad H_i = 0 \\ \max\left(D_0, P(H)_{H_d} + x \times \varsigma_2 + \epsilon\right) & \text{otherwise} \end{cases} \tag{38}$$

其中:

$$D_0 \equiv 131072 \tag{39}$$

$$x \equiv \left\lfloor \frac{P(H)_{H_d}}{2048} \right\rfloor \tag{40}$$

$$\varsigma_2 \equiv \max\left(1 - \left\lfloor \frac{H_s - P(H)_{H_s}}{10} \right\rfloor, -99\right) \tag{41}$$

$$\epsilon \equiv \left\lfloor 2^{\lfloor H_i \div 100000 \rfloor - 2} \right\rfloor \tag{42}$$

区块的燃料限制 H_l 需要满足下面条件:

$$H_l < P(H)_{H_l} + \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{43}$$

$$H_l > P(H)_{H_l} - \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{44}$$

$$H_l \geqslant 125000 \tag{45}$$

H_s 是区块 H 的时间戳, 需满足下面条件:

$$H_s > P(H)_{H_s} \tag{46}$$

这个机制保证了区块间时间平衡；如果最近的两个区块时间间隔短, 则会导致难度系数增加, 因此需要额外的计算量，大概率会延长下个区块的出块时间。相反，如果最近的 2 个区块时间间隔时间过长，难度系数和下一个区块的出块预期时间也会减少。

随机数 H_n，必须满足下面关系:

$$n \leqslant \frac{2^{256}}{H_d} \quad \wedge \quad m = H_m \tag{47}$$

with $(n, m) = \texttt{PoW}(H_{\not H}, H_n, \mathbf{d})$.

其中 $H_{\not H}$ 是新区块的区块头，但不包含随机数和混合哈希值，$\mathbf{d}$ 是当前的大数据集合 DAG(有向无环图)，需要去计算混合哈希, PoW 是工作量证明函数（见 **??**）：第一个元素用于计算混合哈希值，以证明使用了一个正确的 DAG, 第 2 个元素是伪随机数, 依赖于 H 及 $\mathbf{d}$ 。给定一个范围在 $[0, 2^{64})$ 的均匀分布，则求解时间和难度 H_d 成比例。

这就是区块链安全基础，这也是一个恶意节点不能用其新创建的区块中重写历史数据的重要原因。因为这个随机数必须满足这些条件，且因为条件依赖于这个区块的内容和相关交易, 创建新的合法的区块是困难的、耗时的, 需要超过所有诚实矿工的算力总和。

因此, 我们定义这个区块头的验证函数 $V(H)$ 为:

$$V(H) \equiv n \leqslant \frac{2^{256}}{H_d} \wedge m = H_m \quad \wedge \tag{48}$$

$$H_d = D(H) \quad \wedge \tag{49}$$

$$H_g \leq H_l \quad \wedge \tag{50}$$

$$H_l < P(H)_{H_l} + \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{51}$$

$$H_l > P(H)_{H_l} - \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{52}$$

$$H_l \geqslant 125000 \quad \wedge \tag{53}$$

$$H_s > P(H)_{H_s} \quad \wedge \tag{54}$$

$$H_i = P(H)_{H_i} + 1 \quad \wedge \tag{55}$$

$$\|H_x\| \leq 32 \tag{56}$$

其中 $(n, m) = \texttt{PoW}(H_{\not H}, H_n, \mathbf{d})$

此外, **extraData** 最多 32 字节。

5. 燃料和支付

为了避免网络滥用及回避由于图灵完整性而带来的一些不可避免的问题，在以太坊中所有的编程计算都需要费用。各种操作费用以 *gas* (详见附录 **??**) 为单位计算。任意的程序片段（包括合约创建、信息调回、利用及访问账户存储、在虚拟机上执行操作等）都可以根据规则计算出消耗的燃料。

每一个交易都有一个燃料上限：**gasLimit** (燃料上限)。这些燃料从发送者的账户中扣除。具体从账户上扣除的额度和 **gasPrice**(燃料价格) 有关 (译者注：扣除额度 = **gasLimit** * **gasPrice**), 在执行交易前会指定燃料价格。如果这个账户不能支付起燃料费用，这个交易会被当作无效交易。之所以它被命名为燃料上限，是因为剩余的燃料在交易完成之后会被退回（以购买时的同样价格）到发送者账户。燃料不会被用在交易执行之外。因此对于可信任账户，应该设置一个相对较高的燃料上限。

通常来说，以太币（Ether）用作去购买燃料, 未退回的那部分转到了区块受益人的地址，通常这个账户的地址是由矿工设定。交易者可以任意设定燃料价格，然而矿工也可以任意地忽略某个交易。在一个交易中，高价格的燃料将消费这个发送者更多的以太币, 并转给矿工更多的以太币，因此这个交易会被更多的矿工选择。通常来说，矿工将会选择去通知这是他们执行交易最低燃料价格, 交易者们一般也会些选择一个高过燃料价格下限的价格。因此, 会有一个（加权的）最低燃料可接受价格分布，交易者们需要权衡降低燃料价格和交易快速被矿工打包。

6. 交易执行

交易执行是以太坊协议中最复杂的部分：它定义了状态转换函数 Υ。所有交易在执行时, 都要先通过内部的有效性测试, 这些包含:

(1) 交易是 RLP 格式数据，没有多余的后缀字节;
(2) 交易的签名是有效的;
(3) 交易的随机数是有效的（等于发送者账户的当前随机数）;
(4) 燃料上限不小于实际交易过程中用的燃料 g_0;
(5) 发送者账户的余额至少大于费用 v_0, 需要提前支付。

T 表示交易, $\boldsymbol{\sigma}$ 表示状态, 状态转移函数 Υ 如下:

$$\boldsymbol{\sigma}' = \Upsilon(\boldsymbol{\sigma}, T) \tag{57}$$

$\boldsymbol{\sigma}'$ 是交易后的状态。我们定义 Υ^g 为交易执行所消耗的燃料量, $\Upsilon^{\mathbf{l}}$ 为交易过程中产生的日志记录, 都会在后文正式定义。

6.1. 子状态. 从交易执行过程来看，伴随交易会产生一些特定的信息，我们称为交易子状态，用 A 来表示：

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_r) \tag{58}$$

元组内容包含自毁集合 $A_{\mathbf{s}}$：一个将在交易完成后被删除的账户集合。$A_{\mathbf{l}}$ 是一系列的日志：在代码执行过程中是可归档，可索引的检查点，允许以太坊世界的外部旁观者 (例如去中心化应用的前端) 跟踪合约调用。A_r 表示返回的余额，当使用 SSTORE 指令将非 0 的合约存贮空间置为 0 时，返回余额会增加。虽然不是立即返回余额，但可以部分抵消整个执行费用。

我们定义空的子状态 A^0，它没有自毁、没有日记、返回余额为 0：

$$A^0 \equiv (\varnothing, (), 0) \tag{59}$$

6.2. 执行. 执行过程中需要的燃料需要在交易进行之前支付，定义 g_0：

$$g_0 \equiv \sum_{i \in T_{\mathbf{i}}, T_{\mathbf{d}}} \begin{cases} G_{txdatazero} & \text{if} \quad i = 0 \\ G_{txdatanonzero} & \text{otherwise} \end{cases} \tag{60}$$

$$+ \begin{cases} G_{\text{txcreate}} & \text{if} \quad T_t = \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{61}$$

$$+ G_{transaction} \tag{62}$$

$T_{\mathbf{i}}$ 合约初始化 EVM 代码，$T_{\mathbf{d}}$ 是交易附带的关联数据，取决于交易是合约创建还是消息调用。如果这个交易是合约创建，那么会增加 G_{txcreate}，其它情况则不增加。G 的详细定义见附录 **??**。

预支付的费用 v_0 计算如下：

$$v_0 \equiv T_g T_p + T_v \tag{63}$$

需满足下面关系：

$$\begin{aligned} S(T) &\neq \varnothing \quad \wedge \\ \boldsymbol{\sigma}[S(T)] &\neq \varnothing \quad \wedge \\ T_n &= \boldsymbol{\sigma}[S(T)]_n \quad \wedge \\ g_0 &\leqslant T_g \quad \wedge \\ v_0 &\leqslant \boldsymbol{\sigma}[S(T)]_b \quad \wedge \\ T_g &\leqslant B_{Hl} - \ell(B_{\mathbf{R}})_u \end{aligned} \tag{64}$$

注意最后的那个条件：交易燃料限制 T_g <= 这个区块的燃料限制 B_{Hl} - 这个区块之前的所有交易已使用的燃料 $\ell(B_{\mathbf{R}})_u$。

有效交易的执行起始于一个对状态不能撤回的改变：发送者账户 $S(T)$ 的随机数会加 1，账户余额扣减部分预付费用 $T_g T_p$。计算过程中有效的燃料 $g = T_g - g_0$。无论是合约创建还是消息调用，得到的最终这状态 (可能等于当前的状态)，这个改变是确定的且永久无效：从这个观点看不会存在无效的交易。

我们定义检查点状态 $\boldsymbol{\sigma}_0$：

$$\boldsymbol{\sigma}_0 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{65}$$

$$\boldsymbol{\sigma}_0[S(T)]_b \equiv \boldsymbol{\sigma}[S(T)]_b - T_g T_p \tag{66}$$

$$\boldsymbol{\sigma}_0[S(T)]_n \equiv \boldsymbol{\sigma}[S(T)]_n + 1 \tag{67}$$

从 $\boldsymbol{\sigma}_0$ 转变为 $\boldsymbol{\sigma}_P$ 和交易类型相关。不管它是创建合约还是消息调用，我们定义元组包括执行后的临时状态 $\boldsymbol{\sigma}_P$，剩余的燃料 g' 和子状态 A：

$$(\boldsymbol{\sigma}_P, g', A) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}_0, S(T), T_o, \\ \quad g, T_p, T_v, T_{\mathbf{i}}, 0) & \text{if} \quad T_t = \varnothing \\ \Theta_3(\boldsymbol{\sigma}_0, S(T), T_o, \\ \quad T_t, T_t, g, T_p, T_v, T_v, T_{\mathbf{d}}, 0) & \text{otherwise} \end{cases} \tag{68}$$

其中 g 是燃料上限扣除已有的交易消耗的燃料：

$$g \equiv T_g - g_0 \tag{69}$$

T_o 是原始执行者，与和合约创建或消息调用 (直接由交易触发) 不同，可以通过执行 EVM 代码内部调用来执行。

注意，我们使用 Θ_3 表示取函数值的前三个元素；最终结果是消息调用的输出 (一个字节数组)，最终结果并没有在交易的上下文使用。

在消息调回或者创建合约被处理后，再确定退回的燃料后最终状态就确定了。剩余的燃料 g'，再加上一些补偿，得到最终需要退回发送者的燃料 g^*：

$$g^* \equiv g' + \min\{\left\lfloor \frac{T_g - g'}{2} \right\rfloor, A_r\} \tag{70}$$

终需要退回发送者的燃料 g^* 等于当前剩余的燃料 g'，再加一个补偿，这个补偿是 A_r 和总使用燃料量 $T_g - g'$ 的一半中的小者。

燃料对应的以太币给了矿工，矿工地址是当前区块 B 的受益者地址。我们定义预备最终状态 $\boldsymbol{\sigma}^*$，临时状态 $\boldsymbol{\sigma}_P$：

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}_P \quad \text{except} \tag{71}$$

$$\boldsymbol{\sigma}^*[S(T)]_b \equiv \boldsymbol{\sigma}_P[S(T)]_b + g^* T_p \tag{72}$$

$$\boldsymbol{\sigma}^*[m]_b \equiv \boldsymbol{\sigma}_P[m]_b + (T_g - g^*) T_p \tag{73}$$

$$m \equiv B_{Hc} \tag{74}$$

删除所有出现在自毁集合中的账户后，达到了最终状态 $\boldsymbol{\sigma}'$：

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma}^* \quad \text{except} \tag{75}$$

$$\forall i \in A_{\mathbf{s}} : \boldsymbol{\sigma}'[i] \equiv \varnothing \tag{76}$$

最后，我们定义这次交易中总共使用的燃料 Υ^g，这次交易中创建的日志 $\Upsilon^{\mathbf{l}}$：

$$\Upsilon^g(\boldsymbol{\sigma}, T) \equiv T_g - g' \tag{77}$$

$$\Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}, T) \equiv A_{\mathbf{l}} \tag{78}$$

这些有助于后续讨论的交易收据。

7. 合约创建

当创建一个账户时会有很多参数：发送者 (s)、原始执行者 (o)、可用的燃料 (g)、燃料价格 (p)、转账额度 (v)、任意长度的字节数组 $\mathbf{i}$、EVM 初始化代码、消息调用/合约创建的当前栈深度 (e)。

我们定义创建函数为函数 Λ，相关的变量有状态 $\boldsymbol{\sigma}$，包含新状态、剩余燃料及交易子状态 $(\boldsymbol{\sigma}', g', A)$ (详见第 6 小结)：:

$$(\boldsymbol{\sigma}', g', A) \equiv \Lambda(\boldsymbol{\sigma}, s, o, g, p, v, \mathbf{i}, e) \tag{79}$$

这个新账户的地址被定义为包含发送者和随机数 RLP 编码的 Keccak 哈希的最边 160 位。因此我们定义新帐户 a 的地址：

$$a \equiv \mathcal{B}_{96..255}\Big(\texttt{KEC}\Big(\texttt{RLP}\big(\ (s, \boldsymbol{\sigma}[s]_n - 1)\ \big)\Big)\Big) \tag{80}$$

其中 KEC 是 Keccak 256 位哈希函数，RLP 是 RLP 编码函数，$\mathcal{B}_{a..b}(X)$ 表示取二进制数据 X 位数为范围 $[a, b]$ 的值,$\boldsymbol{\sigma}[x]$ 是地址 x 的状态，$\varnothing$ 表示地址不存在时的状态。注意，我们使用的是一个比发送者随机数要小的值；我们断言会在在这个合约创建是前会增加发送者账户的随机数，因此在刚开始在交易或 VM 操作中使用的随机数就是发送者的随机数。

账户的随机数开始被定义为 0，余额为传递的转账值，存储空间为空，哈希代码为 Keccak 256 位的空字符串哈希值；发送者的余额会减去转账值。这个变化的状态变成 $\boldsymbol{\sigma}^*$：

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{81}$$

$$\boldsymbol{\sigma}^*[a] \equiv \left(0, v + v', \texttt{TRIE}(\varnothing), \texttt{KEC}(())\right) \tag{82}$$

$$\boldsymbol{\sigma}^*[s]_b \equiv \boldsymbol{\sigma}[s]_b - v \tag{83}$$

其中 v' 是账户在交易之前就有的余额：

$$v' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}[a] = \varnothing \\ \boldsymbol{\sigma}[a]_b & \text{otherwise} \end{cases} \tag{84}$$

最后，这个账户是通过执行初始化账户的 EVM 代码 $\mathbf{i}$ (执行模型详见小结 9) 来初始化的。代码执行可以影响一些事件：可以改变当前账户的存储，能创建更多的账户，执行更多的消息调用。代码执行函数 Ξ 可以得到一个元组，包括结果状态 $\boldsymbol{\sigma}^{**}$，还可以用的燃料剩余燃料 g^{**}，子状态 A，以及账户 $\mathbf{o}$ 的代码。赋值到最后的状态的数组中，有效的剩余燃料，当前产生的子状态 A 和账号代码本身 $\mathbf{o}$。

$$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}^*, g, I) \tag{85}$$

其中 I 包含执行环境的相关参数，这些参数在小结 9 中有详细定义：

$$I_a \equiv a \tag{86}$$

$$I_o \equiv o \tag{87}$$

$$I_p \equiv p \tag{88}$$

$$I_{\mathbf{d}} \equiv () \tag{89}$$

$$I_s \equiv s \tag{90}$$

$$I_v \equiv v \tag{91}$$

$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{92}$$

$$I_e \equiv e \tag{93}$$

如果没有输入数据，使用 $I_{\mathbf{d}}$ 表示空的元组。I_H 没有什么特别的，它由区块链决定。

代码执行消耗燃料，且燃料不能低于 0，因此程序执行可能会在自然终止之前会退出。在这个（以及其他几个）异常情况，我们称发生了燃料不足（out-of-gas, OOG）异常：演进的状态变成空集合 $\varnothing$，整个创建操作对状态没有影响，状态就像尝试创建合作之前一样。

如果这个初始化代码成功地执行完，那么对应的合约创建的也会被支付。代码保存费用 c 和创建的合约代码大小成正比：

$$c \equiv G_{codedeposit} \times |\mathbf{o}| \tag{94}$$

如果没有足够的燃料支付这些，例如 $g^{**} < c$，就会抛出燃料不足异常。

发生这样的异常发生后，剩余燃料将变为 0。如果合约创建是用于接受一个交易，它不会影响合约创建内部消耗的燃料，无论如何都必须支付。然而，当燃料不足时，并不会给这个合约地址转账。

如果没有出现这样的异常，那么剩余的燃料会退会给最原始的发送者，改变后的新状态也会永久保存。最终状态、燃料和子状态 $(\boldsymbol{\sigma}', g', A)$ 的关系如下：

$$g' \equiv \begin{cases} 0 & \text{if} \quad F \\ g^{**} - c & \text{otherwise} \end{cases} \tag{95}$$

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad F \\ \boldsymbol{\sigma}^{**} \quad \text{except:} & \\ \quad \boldsymbol{\sigma}'[a]_c = \texttt{KEC}(\mathbf{o}) & \text{otherwise} \end{cases} \tag{96}$$

where

$$F \equiv \left(\boldsymbol{\sigma}^{**} = \varnothing \ \vee \ g^{**} < c \ \vee \ |\mathbf{o}| > 24576\right) \tag{97}$$

上述 $\boldsymbol{\sigma}'$ 的公式，表明了执行代码初始化得到的最终字节序列 $\mathbf{o}$，就是新创建账户的代码体。

当合约成功创建时，如果有转账，对应的转账也会转到合约中；如果没有转账，则仅是合约创建。

7.1. 细微之处. 有一种情况需要注意，当初始化代码正在执行时，新创建的地址出现了，但还没有内部的代码时。在这个时间内，任意消息调用不会引起代码执行。如果这个初始化执行结束于一个 SELFDESTRUCT 指令，这个账号会在交易执行完前将被删除，这个行为是无意义的。对于一个正常的 STOP 指令代码，或者返回的代码是空的，这时候会出现一个僵尸账户，而且账户中剩余的余额将被永远被锁定在这个僵尸账户中。

8. 消息调用

当执行消息调用时需要多个参数：发送者（s）、交易发起人（o）、接受者（r）、执行代码的账户（c，通常就是接受者）、可用的燃料（g）、转账额度（v）、燃料价格（p）、函数调用的一个任意长度字节的数组 $\mathbf{d}$ 的输入数据以及消息调用/合约创建的当前栈 (e) 深度。。

除了转变到新的状态和交易子状态外，消息调用还有一个额外的元素 — 用字节数组 $\mathbf{o}$ 表示的输出数据。执行交易时输出数据是被忽略的，但在消息调用时，输出的数据由取决于虚拟机代码执行，在这种情况下也使用了这些信息。

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Theta(\boldsymbol{\sigma}, s, o, r, c, g, p, v, \tilde{v}, \mathbf{d}, e) \tag{98}$$

注意我们的是，当执行 DELEGATECALL 指令时，我们需要区别转账额度 v 和执行上下文中的 $\tilde{v}$ 。

我们定义，在原始状态基础上进行了发送者向接收者转账后为的状态第一个转变状态 $\boldsymbol{\sigma}_1$：

$$\boldsymbol{\sigma}_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v \quad \wedge \quad \boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}[s]_b - v \tag{99}$$

unless $s = r$.

我们假设如果 $\boldsymbol{\sigma}_1[r]$ 还未定义，则会创建一个没有代码或没有状态，且余额和随机数都为 0 的账户。我们改进上一个方程式：

$$\boldsymbol{\sigma}_1 \equiv \boldsymbol{\sigma}'_1 \quad \text{except:} \tag{100}$$

$$\boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}'_1[s]_b - v \tag{101}$$

$$\text{and} \quad \boldsymbol{\sigma}'_1 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{102}$$

$$\begin{cases} \boldsymbol{\sigma}'_1[r] \equiv (v, 0, \texttt{KEC}(()), \texttt{TRIE}(\varnothing)) & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \\ \boldsymbol{\sigma}'_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v & \text{otherwise} \end{cases} \tag{103}$$

账户关联的代码（整个代码碎片，Keccak 哈希为 $\boldsymbol{\sigma}[c]_c$）依靠执行模式（见小结 9）执行。合约创建时，如果执行因为一个异常（例如：耗尽了燃料、堆栈溢出、无效的跳转目的地或者无效的指令）而被终止，燃料不会返回到调用者，且状态也会立即恢复到转账之前的状态 (例如 $\boldsymbol{\sigma}$)。

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if } \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{otherwise} \end{cases} \tag{104}$$

$$g' \equiv \begin{cases} 0 & \text{if } \boldsymbol{\sigma}^{**} = \varnothing \\ g^{**} & \text{otherwise} \end{cases} \tag{105}$$

$$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \begin{cases} \Xi_{\texttt{ECREC}}(\boldsymbol{\sigma}_1, g, I) & \text{if } r = 1 \\ \Xi_{\texttt{SHA256}}(\boldsymbol{\sigma}_1, g, I) & \text{if } r = 2 \\ \Xi_{\texttt{RIP160}}(\boldsymbol{\sigma}_1, g, I) & \text{if } r = 3 \\ \Xi_{\texttt{ID}}(\boldsymbol{\sigma}_1, g, I) & \text{if } r = 4 \\ \Xi(\boldsymbol{\sigma}_1, g, I) & \text{otherwise} \end{cases} \tag{106}$$

$$I_a \equiv r \tag{107}$$

$$I_o \equiv o \tag{108}$$

$$I_p \equiv p \tag{109}$$

$$I_{\mathbf{d}} \equiv \mathbf{d} \tag{110}$$

$$I_s \equiv s \tag{111}$$

$$I_v \equiv \tilde{v} \tag{112}$$

$$I_e \equiv e \tag{113}$$

$$\text{Let } \texttt{KEC}(I_{\mathbf{b}}) = \boldsymbol{\sigma}[c]_c \tag{114}$$

我们假设客户端会先保存数据对 $(\texttt{KEC}(I_{\mathbf{b}}), I_{\mathbf{b}})$，以便更方便地根据 $I_{\mathbf{b}}$ 做出决定。

如我们所见，消息调用执行框架 Ξ 中有 4 个特例: 这四个是‘预编译’合约，作为最初架构中的一部分后续可能会变成本地扩展。这四个合约地址分别为 1、2、3 和 4，分别是用来执行椭圆曲线公钥恢复函数、SHA2 256 位哈希方案、RIPEMD 160 位哈希方案和身份函数。

这 4 个合约的详细定义见附录 **??**。

9. 执行模型

执行模型具体说明怎么使用一系列字节代码指令和一个小的环境数据元组去改变这个系统状态。这些是通过以太坊虚拟机 (Ethereum Virtual Machine - EVM) 这个虚拟的状态来实现的。它是一个准图灵机, 说” 准” 是因为计算会被燃料所限制。

9.1. 基础. EVM 基于栈结构, 机器的字大小（以及栈中数据的大小）是 256 位。主要是便于执行 Keccak-256 位哈希及椭圆曲线计算。内存模型基于字寻址的字节数据。栈的最大深度为 1024。EVM 也有一个独立的存储模型；类似内存但更新一个字节数组, 一个基于字寻址的字数组。不像易变的内存，存储是非易变的且是作为系统状态的一部分被维护。所有在内存和存储中的数据会初始化为 0。

EVM 不是标准的诺依曼结构。它通过一个特别的指令把程序代码保存在一个虚拟的可以交互的 ROM 中, 而不是保存在一般性可访问的内存或存储中。

EVM 在某些情况下会有异常发生，包括堆栈溢出和非法指令。就像缺乏燃料异常那样, 不会改变状态。有异常时, EVM 立即停止并告知执行代理 (交易的处理者，或执行环境), 代理会单独处理异常。

9.2. 费用概述. 在三个不同的情况下使用费用（命名为燃料），在这三种情况下，费用都是执行任何操作的必备条件。第一种情况也是最普通的情况就是计算操作费用 (详见附录 **??**)。第二种情况，燃料可能因为一个子消息调用或者合约创建而被消耗, 这是执行 CREATE, CALL and CALLCODE 费用中的一部分。第三种情况，燃料可能因为内存使用的增多而付费。

对于一个账户的执行，内存总费用和最少需要的 32 字节倍数的内存量成正比, 所有的内存是按在一定范围中被包含的所有记忆索引（无论是读还是写）下要求的 32 位字节的最小倍数成比例的。这个为了 just-in-time (JIT) 的基础而去支付的；访问一个大于索引量 32 字节的地址的, 就会需要额外的内存使用费。地址很容易超过 32 字节的限制, 但 EVM 不同。综上所述, 必须能够去管理这些可能发生的事件。

存储费用有一个微妙的行为——为最小化存储费用（直接与一个在所有结点上的大型状态数据通信），清除存储的执行费用指令不仅仅免除, 而且会返回一些费用；因为初始化的存储费用往往比实际实际使用的多, 在清除存储空间后会有返回费用, 这个返回的费用是预先支付的费用中的一部分。

详细的 EVM 燃料消耗定义见附录 **??**。

9.3. 执行环境. 除了系统状态 $\boldsymbol{\sigma}$ 和计算过程中剩余的燃料 g 外, 还有一些在计算过程中需要提供的信息, 这些信息组成了元组 I:

- I_a, 拥有正在执行的代码的账户地址。
- I_o, 发起这次交易的发送者地址。
- I_p, 发起这次交易中的燃料价格。
- $I_{\mathbf{d}}$, 执行过程中的输入字节数组；如果这次执行是一个交易，这个就是交易数据。
- I_s, 触发代码执行的账户地址；如果这次执行是一个交易，则为交易发送者地址。
- I_v, 以 Wei 为单位的值, 作为计算中的一部分传递给这个账户; 如果这次执行是一个交易, 这个值为转账额度。
- $I_{\mathbf{b}}$, 机器代码字节数组, 会用来执行。
- I_H, 当前区块的区块头。
- I_e, 当前消息调用或合约创建的深度 (例如: 当前 CALLs 或 CREATEs 被执行的次数)。

执行模型定义了函数 Ξ, 用来计算结果状态 $\boldsymbol{\sigma}'$, 剩余的燃料 g', 产生的子状态 A 和结果输出 $\mathbf{o}$, 根据当前的上下文, 我们定义:

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, I) \tag{115}$$

应该还记得 A, 这个产生的子状态定义为自毁集合 $\mathbf{s}$、日志系列 $\mathbf{l}$ 和回推金额 r 的元组:

$$A \equiv (\mathbf{s}, \mathbf{l}, r) \tag{116}$$

9.4. 执行概述. 我们现在必须去定义函数 Ξ。在大多实际执行过程中, 将会整个系统状态 $\boldsymbol{\sigma}$ 和机器状态 $\boldsymbol{\mu}$ 的迭代过程建模。我们定义一个递归函数 X, 它使用这个迭代函数 O(定义状态机中单循环的结果), 定义函数 Z 用来表示当前状态是一个异常终止状态, 定义函数 H 表示当前状态是正常终止时得到的结果数据。

空序列使用 () 表示, 它不等于空的集合 $\varnothing$；这对理解 H 的输出结果非常重要, 当 H 的输出结果是 $\varnothing$ 时需要继续执行, 但当 H 的输出结果是时一个空的序列时执行应该终止。

$$\Xi(\boldsymbol{\sigma}, g, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A, \mathbf{o}) \tag{117}$$

$$(\boldsymbol{\sigma}, \boldsymbol{\mu}', A, ..., \mathbf{o}) \equiv X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\big) \tag{118}$$

$$\boldsymbol{\mu}_g \equiv g \tag{119}$$

$$\boldsymbol{\mu}_{pc} \equiv 0 \tag{120}$$

$$\boldsymbol{\mu}_{\mathbf{m}} \equiv (0, 0, ...) \tag{121}$$

$$\boldsymbol{\mu}_i \equiv 0 \tag{122}$$

$$\boldsymbol{\mu}_{\mathbf{s}} \equiv () \tag{123}$$

$$X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv \begin{cases} \big(\varnothing, \boldsymbol{\mu}, A^0, I, ()\big) & \text{if } Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \\ O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \cdot \mathbf{o} & \text{if } \mathbf{o} \neq \varnothing \\ X\big(O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) & \text{otherwise} \end{cases} \tag{124}$$