以太坊：一种安全去中心化的通用性交易账本

翻译作者：Jorden Gao 邮箱：gaotl33@126.com / tianlu.jorden.gao@gmail.com
本文作者：Dr. Gavin Wood，以太坊&Ethcore 的创始者 邮箱：GAVIN@ETHCORE.IO

摘要：以下是一个实行密码保护交易的区块链范例，而且它已经通过很多项目展示了它的实用性且不亚于比特币。每个单独的项目都能被看成是一个个基于去中心化但独立在电脑资源上的简单应用。我们称这个范例为共享状态和可交易的独立机构。
以太坊完成这个范例只是采用一个通用的方法。在未来，以太坊会提供大量的资源，每一个都是截然不同的状态和可操作的代码，但是它们能通过一个通信框架去和其他的交流。我们讨论了它的设计、实现难题、提供的机会、和我们想象的未来障碍。

1. 简介

随着万能的网络连接在世界上多数城市的发展，全球信息共享的成本越来越便宜。像比特币这样的技术层次的运动已经出现，与此同时它已经与世界分享和完全被免费使用，通过一个可能利用网络设计成的去中心化的价值转移系统的默认的算力、公式机制、和自愿遵守的社会合约。这个系统对于加密安全学来说，能被说成是一个非常特别的特例因为它是基于交易的状态机。域名币（Namecoin）作为进一步发展货币，虽然它是很简单的一种，但是它的系统实现了这样的技术从原有的“货币应用”到其他的应用中。
以太坊是一个去尝试建立通用性技术的项目；这些技术可能全部被建立于基于交易的状态机的概念上。而且以太坊准备去提供一个紧凑地综合的端对端的系统给终端开发者，其目的是为了开发至今还没有发开的电脑软件实例软件：一种通过电脑通信的可信任机制框架。
驱动因素。 以太坊有很多的目标，其中最重要的目标是为了简化个体之间存在互相不信任对方的交易。这些不信任可能是因为地理分离（远距离）、接口对接难度、或者是不兼容、不称职、不情愿、不便宜、不确定、不方便或者现有合法系统的腐败。于是，我们想用一个丰富且清晰的语言去说明一个转台改变的系统，和去搭建一个能自动执行一个争论而且能给我们理想的期望值的系统，最后能给出一个方案去研究这个系统。
在研究这个系统的过程中发现在现实世界有几个属性很难被找到。最难被找到的就是正义清廉化，因为它来自于清廉的算法程序员。其次是透明化，或者说能够清晰的看见一个状态或者一个决断如何传出交易日志和规则或者代码。它几乎没有完美地发生在基于人性的

系统之上，因为自然语言的必要含糊，信息经常性的遗漏，和一成不变的思想导致透明化很能去实现。
总的来说，我希望能发明一个系统，用户能被保证他们的交易是安全的，无论是和其他个体、系统还是组织。而且用户们都能完全放心的处理这些可能的结果和将这些结果实现。
前期事迹。2013 年 9 月下旬，Buterin 第一次提出这种工作的系统机制。虽然现在发展成了很多方法，但是这主要的区块链功能是有一个图灵完备的语言和有效地不受限制和不可改变的互相交易模式下的储存剩余容量。
早在 1992 年，Dwork 和 Naor 就提出了用一种通过网络去传输价值信号和密码学的方法去证明电脑工作量（“proof-of-work”）。在这里，这些价值信号是当成一个阻挡垃圾邮件的机制，而不是任何一种货币。但是在多番证明后，这个潜在的基础数据通道可以携带一个强大的金融信号，同时允许一个接收器去做一个不依赖与信任的物理断言。2002 下半年一种类似血管（vein）的系统被发明。
第一个为强大的金融信号做电脑工作量证明，从而去保护一种货币的范例在 2003 年被 Vish-numurthy et al 使用。在这个案例中，它的令牌被用作去保持 p2p（peer-to-peer）的交易文件在支票中，同时保证“消费者们”能操作微付（pay）到为他们提供服务的“供应商”的账户上。这种通过工作量证明的安全模式在电子签名技术、可以确保历史记录不能破坏的账本技术、和怀有恶意的行为不能欺骗付费或者不满意的服务交货抱怨的技术下被扩大。五年后（2008 年），Nakamoto 介绍了另一种广义下安全的工作量证明的价值令牌。比特币成为这个项目下的产物，而且它是第一个广泛被全球接受的去中心化交易账本。
由于比特币的成功，电子货币项目开始兴起；通过改变它的协议去创建数以千计其他的货币。2013 年 Sprankel 讨论比较知名的币有莱特币（Litocoin）和素数币（Primecoin）。而其他的项目在探索这个协议的价值性的核心内容机制和完善它；例如 2012 年，Aron 讨论了这个域名币（Namecoin）的项目，而且准备去规定一个去中心化的域名争议解决方案系统。
剩下的项目基本都是以共识机制的原则建立于比特币之上，同时借助这个系统的庞大价值和巨大的算力。2013 年，第一个被 Willett 提议的万事达币（Mastercoin）项目，它的目标是通过从属插件的数字使用去创建一个比比特币上层协议还有更多，更高级的特性的丰富的核心协议。2012 年，彩色币（Coloured Coins）项目被 Rosenfeld 提出，去简化这些协议但是保持功能一样，同时润色了这个交易的规则为了打破比特币的基础货币的替代性和允许通过特别的色度钱包（“chroma-wallet”）创造和跟踪这些令牌 – 感知协议的软

件碎片。
在这个区域剩下的工作已经被完成通过放弃一些去中心化基础；2014 年，Boutellier 和 Heinzen 讨论了瑞波币（Ripple）。它已经探索出并创建了一个货币兑换的联邦系统（“federated ”system），同时有效地创建了一个新的金融结算系统。这个金融系统展示了高效的收益能被创造如果去中心化基础能被放弃。
在 1997 年，Szabo 和 Miller 已经完成了早期的智能合约（“smart contract”）。上个世纪 90 年代，算法执行的协议在人类合作中起着重大的作用，这个理念已经变得很清晰。虽然当时没有设计出任何特别的方法去实现这样的系统，但是它已经被提出这样的系统会严重地影响未来的法律。在这个预言下，以太坊可能被看成完成了这样密码学法律的系统的成就。

2.区块链的实例

以太坊，作为一个整体，能被看成是一个基于交易的状态机：我们开始于一个创世块（“Genesis”）状态，然后伴随着执行交易的写入到最后的状态。这个最后状态是我们能接受的权威的版本在以太坊世界。这个状态保存了一下信息，比如账号的余额、名誉度、信誉度、和附属的现实世界的数据。总而言之，最近能被电脑描绘的任何事都是合理的。因此，交易就代表了两个状态的有效桥梁；这个“有效”是很重要的 – 因为这里存在无效的状态改变远远超过有效的状态改变。举一个例子：无效的状态改变可能是这里减少了一个账号的余额，但是没有在其它任何账号上加上同等的额度。然而，一个有效的状态转换是来自一个交易。公式上：
（1） $\sigma(t+1) \equiv \gamma(\sigma t,T)$
其中 γ 是以太坊的状态交换函数。在以太坊中 γ 与 σ 是相当地强大超过现存任何的类似系统；γ 允许部分成分去实现特有的计算，相比下 σ 允许部分成分存贮特有的状态在交易间。
交易是被校对在区块中；区块采用一种密码学的哈希值（hash）作为参考方法被“链子”链接起来。区块充当一个日记，记录着一系列与之前区块的交易和鉴定最后一个状态（虽然它不能通过自己存贮最后状态 – 因为这个区块链会一直增加）。他们也标记这些交易系列为了激励节点去挖矿。这样的激励作为状态转换函数发生，同时增加相应的价值到一个特定的账号。
挖矿是一个奉献努力的过程，这个过程是为了维持交易链（区块链）在任何一个潜在的竞争块中。它是由于一种密码学上安全证明的方式被实现。这个工作量证明的体制

（POW）是众所周知，会在 11.5 小结讨论更多的细节。
公式上，我们张开成：
（2） σ(t+1) ≡ Π(σt,B)
（3） B ≡ (…,(T0,T1,…))
（4） Π(σt,B) ≡Ω(B,γ(γ(σ,T0),T1)…)
其中 Ω 是区块完成状态转换函数（这个函数是奖励一个特有的团体）；B 是这个区块，它包含了某些成分中的一系列交易；Π 是区块等级（“block-level”）状态转换函数。
上述是区块链范例的基础，这种模型不仅仅只是形成了以太坊的基础结构，而是迄今为止所有基于共识去中心化的交易系统。
价值。为了去激励网络的算力，这里需要一种方法去被定义交易价值。为了解决这个问题，以太坊设计了一个固有的币，以太币（“Ether”），ETH 是大家所知的符号，有些时候用古老的 D（“Ð”）描述。以太币最小的面额是位（“Wei”），因为位能在货币中所有整数值进行计数。一个以太币被定义成 10^18 位。这里还有以太币的其他面额值：

Multiplier Name
10^0 Wei
10^12 Szabo
10^15 Finney
10^18 Ether

纵观目前的工作，无论是任何价值的参考、以太币的内容、货币、还是余额或者支付，都应该被看成用“位”来计算。
它的历史？因为这是一个去中心化的系统而且所有的团队都有机会在一些之前的区块上去建立新的区块，这个合成的架构必然是区块链树。为了去形成一个共识，路径通过这种架构从底部（创世块）到叶子（包含最新交易的区块），这就是众所周知的区块链，这里必须都去共识。如果某个时段在最长的区块链中节点之间有一个争论，那么分叉（“fork”）出现。
这个就意味着过去给定的时间点（区块）和系统的多个状态共存：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。这也是为了去避免所以不确定的代价，因为它会破坏整个系统中的所有信用。

计划去产生共识，我们使用了一个简单 GHOST 协议版本。这个协议在 2013 年被 Sompolinsky 和 zohar 提出。我们会在 10 小结描述它的细节。

3.协定

我用了大量的印刷体协定去表示正式的符号，其中一些对于现在的工作来说是非常具有意义：
这两个高度结构化的集合中，高标准（“high-level”），状态值，被指的是粗体小写的希腊字母。他们注入了全局状态（“world-state”）并且被用 σ 代表（或者一个变体的 thereupon）和机器状态（“machine-state”）被用 μ 代表。
在高度结构化值中，可操作函数（“function operating”）被表示成一个大写的希腊字母，例如：γ，同时也是以太坊中的状态转换函数。
对于更多的函数来说，通常用一个大写的字母，例如：C，通用的费用函数。这些也可能作为下角标被表示为一些特别的变量，例如：CSSTORE，费用函数中的 SSTORE 操作。对于特别的和可能外形清晰的函数，我可能设计成了打印机的文字，例如：Keccak-256 哈希函数（按获奖作品 SHA-3 竞赛函数）被用 KEC 表示（通常是用来解释 Keccak）。还有 KEC512 也正在被用作去表示 Keccack-512 哈希函数。
数组通常也是用一个大写字母去代表，例如：T，被用作去表示一个以太坊的交易。如果相对下地定义，这个符号可能成为下角标用去表示一个独立的成分，例如：Tn，表示上述交易的随机数。这些下角标的形式用于表示它们的类型；例如：大写的下角标表式带有下角标成分的数组。
标量和固定大小的字节顺序（或者同义字，数组）都能用小写的字母来表示，例如：n 在数据文件中被用来一个交易的随机数。小写的希腊字母能代表一些特别的含义，例如：δ 表示为所需的项目数在堆栈上所对应的操作。
任意长度的序列通常用加粗的小写字母表示，例如 O 被表示为信息调用中输出数据的字节序列。有时候，加粗的大写字母表示一些特别重要的值。
总体来说，我们认为标量都是正整数而且属于集合 P。字节序列的集合是 B，我们在附件 B 中书写了正式的定义。如果像这样的序列集合被限制它们的特有的长度，它能用下角标符号去代表。因此字节为 32 的所有序列集合命名为 B32，所有正整数小于 2^256 的集合命名为 P256。正规定义将会在 4.4 小结讨论。
正方形括号被用作索引和代表独立的成分或者序列下的子序列，例如：μs [0]表示计算机堆栈中的第一个数字。对于子序列来说，省略号被用作具体去说明一定的范围，且含头尾

的限制元素，例如：μm [0…31]表示计算机内存中的第一个 32 数字。
在全局状态的情况下，σ 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。
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，如果我们认为这些不可修改的输入值将被用□（“占位符”）来表示，可修改和有用的值用□^’表示，中间的值则用□，□*() &c*。*在非常特别的场合下，为了扩大可读性和清晰性，我可能会使用字母-数字下角标表示中间值，尤其他们是特别的。*
当去考虑去用现有的函数时，给定一个函数 f 或者 f，其中序列之间替换了函数之间的映射关系。他们将被正式的定义在 4.4 小节。
全过程中，我定义了很有有用的函数。最常见的是函数是 l (小写 L)，功能是计算给定序列的最后一个数字：
（5） l(X) ≡X[|| X ||- 1]*

公式解释说明：由于网页不默认 word 公式的格式，以下为公式的解释说明：
1.B_20 中*为下角标；*
2. f ^* ((x_0,x_1,…))≡(f_ (x_0),f_ ^ (x_1),…) 中^为上角标；
3. R_u∈P ∩〖 R〗*b ∈B_256 中〖 〗忽略不计；*
4. D(H)≡ {(D_0^ if H_i^( )=0
{ max〖(D_0^ ,P(H)(H_d^)^ +x*ε1+ϵ) if H_i^( )<〗 N_H^( )
{max(D_0^ ,P(H)_(H_d^)^ +x*ε2+ϵ) otherwise)

中 3 { 是一个大个{
若有公式疑问，建议下载原文对应公式看；再次抱歉！

4. 区块，状态和交易

已经介绍了以太坊的基本概念，我们将更详细地讨论一个交易、区块和状态的含义。
4.1 全局状态。全局状态是一种映射在地址（160 位的识别符）和账号之间的状态（一种 RLP 连续的数据结构，见附件 B）。虽然它没有储存在区块链上，但是它被认为它的功劳是全力支撑映射在一个修改过的梅克尔帕特里夏树中（“Modified Merkle Patricia Tree”- 缩写为 trie，见附件 D）。Tire 需要简单的数据库后端去支持字节数组到字节数组的映射；我们命名这底下的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内部的数据，同样地它的哈希在整个系统中能被作为一个安全的身

份。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）去被召回通过简单地改变根部哈希值。因为我们储存了所以这样的根部哈希值在区块链中，我们能过简单地调回以前的状态。

这个账号状态包含以下四个区域：

随机数（“nonce”）：在账号与相关的代码的这种情况下，一个标量值等于这个地址发送的交易数或者，由这个账号产生的合约创造的数量。在状态 σ 中对于地址 a 的账号，可以被表示成 σ[a]n。

平衡（“balance”）：一个标量值等于这个账号拥有的位（“Wei”）数。正式表示为 σ[a]b。

根储存（“storageRoot”）：一个编译在这个账号的储存内容中的（一个包含 256 位整数值的数组）梅克尔帕特里夏树根节点的 256 位哈希值编译进了 trie，作为一个数组从 256 位整数钥匙的 Keccak 256 位哈希值到 RLP 编译的 256 位整数值。这个哈希值被表示为 σ[a]s。

代码哈希值（“codeHash”）：这个账号在 EVM 代码下的哈希值 – 这个代码就是当执行时需要这个地址去接受一个信息调用（“message call”）；它是不可变的，不像在其他的领域一样，因此运行它之后就不能被改变。所有像这样的代码碎片是被包含在它们最近接受回应哈希值的状态数据库下。这个哈希值被正式的表示为 σ[a]c，如果这个代码可能表示为 b，则给定 KEC(b)=σ[a]c。

因为我经常希望不是去暗指这 trie 的根哈希值而是去底层存储的钥匙或者值的集合，我定义一个方便的方程式：

（6） TRIE(LI^* (σ[a]s)) ≡ σ[a]s

在 trie 中这个函数的 LI^*是一对钥匙或者值集合，而且被定义成基于功能 LI 的元素扩展转换，定义为：

（7） LI ((k,v))≡(KEC(k),RLP(v))

其中：

（8） k∈B32 ∩ v∈P

它应该被理解成 σ[a]s 不是这个账号的“本身的”成员而且也不能促成它的系列化。

如果代码哈希值区域是 Keccak-256 的空集合的哈希值，例如： σ[a]c=KEC(()，那么这个代码值表示为一个简单的账号，有些时候简称为一个“非合约”账号。

因此我们可能定义一个世界状态的子函数 Ls：

（9） LS (σ)≡{p(a):σ[a]≠0}

其中：

（10） p(a)≡(KEC(a),RLP((σ[a]n,σ[a]_b,σ[a]_s,σ[a]_c)))

这个函数，L_S，是被跟随使用在 trie 函数后，为了提供一个世界状态的简介身份（哈希值）。我们认为：

（11） ∀a:σ[a]=⌀∪(a ∈ B20 ∩ v ∈(σ[a]))

其中 v 是一个这个账号的有效性功能：

（12） v(x)≡x_n ∈P_256 ∩ x_b ∈P_256 ∩x_s ∈B_32 ∩ x_c ∈B_32

*4.2.家园（“Homestead”）。**一个兼容公共网络的显著区块数字是一个交易数标记在以太坊平台的前沿（“Frontier”）和家园（“Homestead”）阶段，我们用符号表示为 N_H，因此定义：***

（13） N_H≡1,150,000

这个协议过去在这个区块中被升级，因此这个符号在某些方程中出现不同的账号。

****4.3.交易。**交易（符号，T）是一个单一密码署名签名的指令，通过一个永久在以太坊中的操作者创建。当这个永久的操作者含有人类的性质成为假设，软件工具将被用在它的结构和传播 ^1。这里有两种类型的交易：一种是结果在信息调用中、一种是结果在相关的代码（众所周知为“合约创建”）下的新账号创建。两种类型都为公用领域的一个数字。**

(^1 显著地，这样的“工具”最终将会变成那么有原因的迁移从他们基于人类的创造 – 或者人类可能变成有原因的中立 – 这里将会出现一个点当他们正确的被考虑为自治的代理人。例如：合约可能提供好处给人类去发送交易而且去开始他们的行动。)

随机数（“nonce”）：一个标量值等于被发送者发送的交易数；表示为 T_n。

燃料价（“gasPrice”）：一个标量值等于被支付执行这个交易所有计算而产生的每个单位燃料的“位”数；表示为 T_p。

燃料限制（“gasLimit”）：一个标量值等于燃料被用于执行这个交易的最大限度值。它是被预先支付的，在任意计算完成之前和以后不会增加；表示为 T_g。

接受者（“to”）：一个信息调用的 160-位地址的接受者，或者是给合约创建交易，⌀，在这用去指示 B_0 这个数字；表示为 T_t。

价值（“value”）：一个标量值等于被转移到信息调用的接受者，或者合约创建，作为一个新建账号的天赋；表示为 T_v。

v, r, s：交易签名和用去决定交易的发送者的对应值；表示为 T_w，T_r 和 T_s。详细见附件 F。

另外，一个合约创建交易包含：
初始化（“init”）：在 EVM-代码中为了账号初始化的过程而指定的一个没有限制大小的字节数组，表示为 T_i。
初始化是 EVM-代码的碎片；它返回主干，每次执行这个账号的代码第二个碎片都会收到一个消息调用（无论是通过交易还是因为内部执行代码）。初始化仅仅被执行一次在账号创建和之后立即丢弃的时候。
相对之下，一个信息调回交易包含：
数据（“data”）：在信息调用输入数据中指定的一个没有限制大小的字节数组，表示为 T_d。
附件 F 详细描述了这个函数，S，它传输交易给发送者，和画出 SECP-256k1 的 ECDSA 曲线，利用这个交易的哈希值（除了最后三个签名的区域）作为一个数据去签名。暂时我们只是简单地声称这个给定交易的发送者 T 能被看成 S(T)。
（14） LT (T) ≡ { (Tn,Tp,Tg,Tt,Tv,Ti,Tw,Tr,Ts) if Tt=∅

{ (Tn,Tp,Tg,Tt,Tv,Td,Tw,Tr,Ts) otherwise
在这里，我们认为所有的成分是被 RLP 作为一个整数值代替，除了特有长度字节数组的 T_i 和 T_d。

Tn∈P_256∩T_v∈P_256∩ T_p∈P_256∩
（15） Tg∈P_256∩T_w∈P_5 ∩ T_r∈P_256∩
T_s∈P_256∩〖 T〗*d∈B* ∩ *〖T〗i∈B)*
其中，
（16） P_n ={P : P∈P∩P <2^n} P 为元素，属于 P 为集合
这个地址的哈希值 T_t 稍微有点不同：它是一个 20 字节的地址哈希值，或者成为一个创建合约交易（因此表示为∅），它是 RLP 空字节系列，因此表示为数字 B0：
*（17） T_t∈{(B*20 if T_t≠∅

{ B_0 otherwise
4.4.区块。在以太坊中区块是相关信息块的收集（区块头），H，响应的信息聚集一起构造交易，T，和其他区块头数据的集合 U，它众所周知有一个父块，等于这个父区块的父块的父块（这个的区块为 ommers ^2）。这个区块头包含一些信息的碎片：

（^2ommer 是 最流行（但是提起的不多）的性别中立项目，表示为“父母的兄弟”；见 http://nonbinary.org/wiki/Gender_neutral_language#Family_Terms）

父块哈希值（“parentHash”）：在整体中说，父块的区块头 Keccak256 位哈希值，表示为 H_p。
ommers 哈希值（“ommersHash”）：区块中 ommers 列表部分 Keccak256 位哈希值，表示为 H_o。
受惠者（“beneficiary”）：从迁移区块的成功采矿中所有费用收集起来的 160 位地址。
状态根（“stateRoot”）：trie 状态根节点的 Keccak256 位哈希值，在所有交易被执行和定稿被应用；表示为 H_r。
交易根（“transactionsRoot”）：trie 结构根节点的 Keccak256 位哈希值移植到这个区块交易列表中的每个交易；表示为 H_t。
接收人根（“receiptsRoot”）：trie 结构根节点的 Keccak256 位哈希值移植到这个区块交易列表中的每个交易的接收人；表示为 H_r。
日记块（“logsBloom”）：日记块写下含有可索引的信息（写日记人地址和日记主题），这信息包含每个日记里交易列表中每个交易的接收人的登录信息；表示为 H_b。
难度（“difficulty”）：一个响应的标量值为了表示这个区块的难度等级系数。这个等级系数能从之前区块难度等级系数和时间戳中被计算出来；表示为 H_d。
区块数字（“number”）：一个标量值等于产生区块的数字。创世块是数字 0；表示为 H_n。
燃料限制（“gasLimit”）：一个标量值等于每个区块当前燃料开支的限制值；表示为 H_l。
燃料使用（“gasUsed”）：一个标量值等于当前区块中交易所用的所有燃料；表示为 H_g。
时间戳（“timestamp”）：一个标量值等于当前区块产生时的合理 Unix 时间输出；表示为 H_s。
额外数据（“extraData”）：包含这个区块的任意字节相关数据。必须是少于等于 32 为；表示为 H_x。
混合哈希值（“mixHash”）：一个与随机数一起证明已经在这个区块中完成的充分的计算力总量的 256 位哈希值；表示为 H_m。
随机数（“nonce”）：一个与混合哈希值一起证明已经在这个区块中完成的充分的计算力总量的 256 位哈希值；表示为 H_n。

剩下的区块成分是 ommer 区块头中的一个列表（与上面数据格式一样）和一系列的交易。我们用一个区块 B 去表示为：

（18） B≡(B_H,B_T,B_U)

4.4.1.交易收据（“transReceipt”）。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 i 个交易中表示为 B_R [i]）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 H_e。

交易收据是一个包含四个项目的数组：交易后的状态 R_σ；当前区块中交易发生后收据中立即保存积累性燃料使用值 R_u；交易执行中创建的日记集合 R_l；和日记块过滤器中包含这些日记的信息 R_b：

（19） R≡(R_σ,R_u,R_b,R_l)

函数 L_R 是筹备从一个交易收据转变为一个系列化的 RLP 字节的数组：

（20） L_R (R)≡(〖TRIE(L_S (R〗*σ)),R_u,R_b,R_l)*

因此这个交易后的主状态，Rσ 是被编译在 trie 结构中，根值是 trie 格式中的第一个值。我们声称 R_u，积累性的燃料使用值是一个正整数和日记块过滤器，R_b，是一个大小为 2048 位的哈希值（256 字节）：

（21） R_u∈P ∩〖 R〗*b∈B_256 中*

这个日记写入，R_l，是一系列日记写入，例如被称为：（O_0,O_1,…）。一个日记写入 O 包含：O_a，是一个写日记人地址的数据，O_t，是 32 位日记主题的集合和 O_d，是一些数据字节的数字：

（22） O≡(O_a,(O_t,O_t1,…〗),O〗d)

（23） Oa∈B_20∩ ∀(t∈O_t)：t∈B_32 〖∩ O〗*d∈B*

我们定义这个日记过滤器的功能，M，去减少在单个 256 字节哈希值中的一个日记写入：

（24） M(O)≡∨*(t∈{〖 O〗_a ∪〖 O〗_t)*▒*(〖 M〗*(3:2048) (t))

其中〖 M〗*(3:2048)是一个特别的日记过滤器，它的集合是第 3 位到 2048，给定的一个任意的字节序列。它是通过这个字节序列 Keccak-256 哈希值中的每第一个三对字节的低顺序的前 11 位。表示为：*

（25） 〖M〗(3:2048) (x:x∈B)≡ y:y∈B_256 其中：

（26） y≡ (0,0,…,0) 除：

（27） $\forall (i \in \{0,2,4\}) : B(m(x,i))\ (y)=1$

（28） $m(x,i) \equiv KEC(x)[i,i+1] \bmod 2048$

其中 B 是位参考函数，例如：B_j (x)等于字节数组中的 x 索引 j 位（从 0 开始编入索引）。

4.4.2. 历史的有效性（“Holistic Validity”）。如果搞一个区块满足以下几个条件，我们能声称这个区块是有效的：它必须与 ommer 保持内部一致，而且交易区块哈希值和给定的交易 B_T（详细描述在第 11 节），当按基本状态 σ（起源于父块中最后的状态）顺序执行，就会产生一个新的身份状态 H_r：

(29)

$$
\begin{aligned}
&H_r \equiv TRIE(L_S(\Pi(\sigma,B))) \cap \\
&H_o \equiv KEC(RLP(L_H^*(B_U))) \cap \\
&H_t \equiv TRIE(\{\forall i < |(|B_T|)|, i \in P : p(i, L_T(B_T[i]))\}) \cap \\
&H_e \equiv TRIE(\{\forall i < |(|B_R|)|, i \in P : p(i, L_R(B_R[i]))\}) \cap \\
&Hb \equiv \bigvee_{r \in B_R} (r_b)
\end{aligned}
$$

其中 p(k,v)仅仅是与 RLP 互相转化。既然这样，k 为这个区块中的交易索引和 v 为交易收据：

（30） $p(k,v) \equiv (RLP(k), RLP(v))$

此外：

（31） $TRIE(L_S(\sigma)) = \llbracket P(B_H) \rrbracket (H_r)$

因此 TRIE(L_S (σ))是包含状态 σ 和 RLP 编译值的一对钥匙值的梅克尔帕特里夏树的根节点哈希值，〖P(B_H)〗 直接定义为区块 B 的父块。

这些值由交易的计算产生，特别是交易收据 B_R，这个通过交易状态累积函数 Π 定义，在 11.4 小结详细地用公式表示。

4.4.3. 串行化（“serialisation”）。函数 L_B 和 L_H 分别是区块和区块头的准备函数。更像这个交易收据准备函数 L_R，当 RLP 转换被允许时我们声称这个结构的类型和顺序为：

（32） $L_H(H) \equiv (H_P, H_o, H_c, H_r, H_t, H_e, H_b, H_d, H_i, H_l, H_g, H_s, H_x, H_m, H_n)$

（33） $L_B(B) \equiv (L_H(B_H), L_T^*(B_T), L_H^*(B_U))$

随着 L_T^*和 L_H^*成为宽元素序列转换，因此：

（34） f_ ^* ((x_0,x_1,…))≡(f_ (x_0),f_ ^ (x_1),…) 其中 f 可为任何函数

因此这些成分类型被定义为：

（35） H_p∈B_32 ∩ H_o∈B_32 ∩ H_c∈B_(20) ∩

〖 H〗r∈B_32 ∩ H_t ∈B_32 ∩ 〖H〗e∈B_32 ∩

H_b∈B_256∩〖H〗_d∈P ∩〖H〗_i ∈P ∩

H_l∈P ∩ H_g∈P ∩ H_s∈P_256 ∩

H_x∈B ∩〖H〗_m∈B_32∩〖H〗_n∈B_8

其中

（36） B_n≡ {B:B∈B∩|(|B|)|=n})

我们现在知道有一个严格的正式的区块结构的框架说明书。这 RLP 函数（见附件 B）提供了一个标准的方法通过线或者本地的存储空间去转变这种结构成为一个字节的序列。

4.4.4. 区块头的有效性（“Block Header Validity”）。*我们定义 P(BH)为区块 B 的父块, 表示为：*

*（37） P(H)≡ B*H^：KEC(RLP(B_H^')=B_P^

区块数字是这个父块数字加 1：

（38） H_i^ ≡ P〖(H)〗*(H_i^)^ +1*

一个标准的区块头 H 的难度被定义为 D(H)：

（39）

D(H)≡{(D_0^ if H_i^( )=0

*{ max〖(D*0^ ,P(H)*(H_d^)^ +x*ε1+ϵ) if H_i^( )<〗 N_H^( )*

{max(D_0^ ,P(H)(H_d^)^ +x*ε2+ϵ) otherwise)

其中：

（40） D_0^ ≡131072

（41） x≡ (P(H)_(H_d^)^)/2048

（42） ε1≡ {1 if H_s^ <P(H)_(H_s^)^ +13

{-1 otherwise)┤

（43） ε2≡max(1-[(H_s^ -P(H)*(H_s^)^)/10],99)*

（44） ϵ≡[2 ^([(H_i^)/100000]-2)]

一个标准的区块头 H 的燃料限制为 H_l，它必须满足这个关系：

（45） H_l

（46） H_l>P(H)*(H_l^)^ -[(P(H)*(H_l^)^)/1024] ∩

（47） H_l≥125000

H_s 是区块 H 的时间戳，它必须满足这个关系：

（48） H_S>P(H)*(H_S^)^

这个机制执行了区块间时间的体内平衡；在最后两个区块间的一个更小的周期会导致难度系数的增加，因此额外的计算需求，保持喜爱一个周期的长度。相对下，如果这个周期过长，难度系数和下一个区块的预期时间也会被减少。

这个随机数，H_n 必须满足这个关系：

（49） n≤ 2^256/H_d ∩m= H_m

有(n,m) = PoW (H_n,H_n,d)。

其中，H_n 是这个新区块的区块头，但是里面不存在随机数和混合哈希值得成分，d 是最近的 DAG，一个大数据集合需要去计算这个哈希值。PoW 是工作证明函数（见 11.5 小结）：这个函数评估第一个项目成为混合哈希值的数组，也同时证明一个 DAG 是不是被正确地使用；这个函数使用第二个项目中的 H 和 d 评估是否成为一个假冒随机的密码学数字。给定一个大致统一的分配范围[0, 2^64)，用这个期待的时间去寻找一个方法是和寻找难度系数 H_d 一样。

这就是区块链安全性的基础，同时也是为什么一个怀有恶意节点不能繁殖到最新创建的区块中而重写历史的重要原因。因为这个随机数必须满足这个要求，和因为它的满足取决于这个区块的内容和它依次地交易的组成、新的创建、有效性、区块的难度性、超出的时间、和挖矿可信节点部分大致需要的总算力。

因此我们能够定义这个区块头的有效性函数 V(H)为：

*（50） V(H)≡2^256/H*d ∩m= H_m ∩

（51） 〖 H〗*d=D(H)* ∩

（52） H_g≤H_l ∩

（53） H_l>P(H)(H_l^)^ +[(P(H)*(H_l^)^)/1024]* ∩

（54） H_l>P(H)(H_l^)^ -[(P(H)*(H_l^)^)/1024]* ∩

（55） H_l≥125000 ∩

（56） H_S>P(H)(H_S^)^ ∩

（57） H_i^ ＝ P〖(H)〗*(H_i^)^ +1* ∩

（58） |(|H_x^ |)|≤32

*其中(n,m) = PoW (H*n,H_n,d)。

另外说一句，额外数据（“extraData”）必须最大字节位数为 32。

5. 燃料和支付（“Gas and Payment”）

为了避免网络滥用的问题和回避由于 Turing 不完整性而带来的一些不可避免的问题，在以太坊中所有可编程的计算都是需要费用。这个费用表在燃料单位（“units of gas”）中有特别说明（见附件 G，不同的计算对应着不同的费用）。因此任意给定可编程计算的一部分（包括合约创建、信息调回、有用和可进入的账号存储和在虚拟机上的执行操作）都有一个普遍地同意的燃料消费方式。

每一个交易都有一个特别的燃料总额取决于：燃料限制。这个燃料总额是间接地从发送者的账号余额购买。这个购买力取决于燃料价格，特别是在交易中。如果这个账号余额不能支付这个燃料购买力，这个交易被考虑无效。它被命名为燃料限制，而且任何一个无用的燃料在交易之后都会被退回（以购买时的同样汇率）到这个发送者的账号。燃料不会被使用在交易执行之外。因此对于含有可信任代码的账号，相对下会被单独的设置为一个高的燃料限制。

通常来说，以太币（Ether）用作去购买不能被返回的燃料而且能被转到这个受益人的地址中，通常这个账号的地址都是用矿工控制。交易处理者可以任意地去具体说明他们想要的燃料价格，然而矿工也可以任意地去忽略他们可选择的交易。在一个交易中，高价格的燃料将消费这个发送者的以太币和运输一个更大价值给矿工，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小的燃料价格和交易者们将会免费去覆盖这些由燃料价格去提供的价格。因此这里将会有一个（加权过的）最小可接受的燃料价格分配，交易者们将必须去决定于一个在最低燃料价格和最大及时正确地被挖的机会中间权衡。

6.交易的执行（“Transaction Execution”）

交易的执行是以太坊协议中最复杂的部分：它定义这些状态交易函数 γ。它被认为任何被执行的交易第一次都能通过内部有效性的初始测试。这些包含：

交易是一个结构完整的 RLP，没有多余的后缀字节；

交易的签名是有效的；

交易的随机数是有效的（等于当前发送者账号的当前随机数）；

燃料限制是不小于内部的燃料，g_0^ ， 用来交易；

发送者账号的余额至少大于消耗量，v_0^ ， 需要提前支付。

形式上来说，我们考虑这个函数 γ，配合 T 和状态 σ 成为一个交易：

（59） σ_ ^(’)=γ(σ,T)

因此 σ_ ^(')是交易后的状态。我们也定义 γ_ ^(g)是去估计用于一个交易执行的燃料总额和 γ_ ^(1)去估计交易产生的日记数量，稍后它们都会被定义。

6.1 子状态（“Substate”）。从交易执行全过程来看，我们产生一些特定被立即跟随这个交易执行的信息。我们称为交易子状态，用 A 来表示它的数组：

（60） A≡A_s,A_l,A_r

这个数组的内容包含 A_s，一个自杀集合（“suicide set”）：一个将在交易完成后被抛弃账号的集合。A_l 是一个日记系列：这是一系列在允许通过以太坊内部旁观者容易追踪合约调用的 VM 代码执行的可存储和可索引的‘检查站’（“例如一个前期去中心化的软件”）。A_r，返回的余额，通过使用 SSTORE 结构上升为了重设合约存储从非 0 值到 0。虽然不是立即的返回，它也是被允许去部分抵消整个执行消费。

简短来说，我们定义这个空的子状态 A_ ^0 是没有自杀函数、日记、和 0 返回余额：

（61） A_ ^0≡(∅,( ),0)

6.2 执行（“Execution”）。我们定义内部燃料 g_0^ ，这个交易需要去被首先支付给执行的燃料总额，如下：

（62） g_0^ ≡∑*(i∈T_i^ ,T_d^){(G_txdatazero^ if i=0*

*G*txdatanonzero^ otherwise)┤

（63） + {(G_txcreate^ if T_t^ =∅∩H_i^ ≥N_H^

0 otherwise)┤

（64） + G_transaction^

其中 T_i^ ,T_d^ 表示为一系列与交易相关数据的字节和初始化 EVM 代码，取决于它是合约交易还是信息调回交易。如果这个交易是创建合约而形成，而且不是 EVM 代码的一个结果或者在家园转换（“homestead transition”）之前，则 G_txcreate^ 是会被添加。G 完全定义在附录 G。

这个预见支付的费用 v_0^ 被计算为：

（65） v_0^ ≡ T_g^ T_p^ +T_v^

有效性是被取决于：

（66） S(T)≠∅∩

〖σ[S(T)]〗_ ^ ≠∅∩

〖T〗*n*^ = *〖σ[S(T)]〗n*^ ∩

g_0^ ≤T_g^ ∩
〖v〗0^ ≤〖σ[S(T)]〗b^ ∩

*T_g^ ≤B_Hl^ –l〖(B_R^)〗_u^*
注意最后的那个公式；交易燃料限制的总和 Tg^ 和前一个区块中有用的燃料值 l
〖(BR^)〗_u^ 都必须不能大于区块的燃料限制 B_Hl^ 。
有效交易的执行开始于一个可取消变化的制造状态：发送者账号的随机数 S(T)，它是加 1 的增量而且余额是通过前段消费 Tg^ T_p^ 被返回。算计的过程中有效的燃料 g 是被定义为 T_g^ -g_0^ 。无论是合约创建还是信息调回，这个计算结果是一个最后的状态（合法下可能等于这个当前的状态），这个变化决定论的而且一直无效的：从这个观点看这里能存在一个无效的交易：
我们定义这个检查站状态为 σ0^ ：
（67） σ0^ ≡σ ^ 除：
（68） *σ0^ 〖[S(T)]〗_b^ ≡σ ^* 〖[S(T)]〗*b^ -T_g^ T_p^*
（68） σ0^ 〖[S(T)]〗n^ ≡σ ^ 〖[S(T)]〗*n^ +1*
从 σ0 赋值给 σ_p 取决于交易的类型；不管它是创建合约还是信息调回；我们定义这个过去执行暂时状态数组为 σp，同时留下燃料 g ^(')和子状态 A：
（70） *(σp,g ^(* '),A)≡ {(Λ(σ0^ ,S(T),T_0^ ,g,T_p^ ,T_v^ ,T_i^ ,0) if T_t^ =⌀@Θ3^ (σ0^ ,S(T),T_0^ ,T_t^ ,T_t^ ,g,T_p^ ,T_v^ ,T_d^ ,0) otherwise)⊣*
其中 g 是扣除现有的交易需要去支付的基本燃料总和后剩余燃料总和：
*（71） g≡Tg^ -g_0^*
T_0^ 是原始的交易者，在这个情况下它和信息调用或者合约创建的发送者是不同的，它不去直接地触发交易但是它存在 EVM 执行代码中。
注意，我们使用 Θ3^ 仅仅去表示被用地函数值中的第一个前三个部分；它最后表示为信息调回的输出值（一个字节数组）和没有被用在交易赋值的内容中。
在信息调回或者创建合约被运行后，这个状态被决定通过被返回的决定值总和送给发送者（使用开始的货币汇率）；g ^属于剩余燃料（g_ ^('））加上一些来自于对立的偿还补贴：
（72） *g_ ^≡g_ ^(* ')+min{[(〖T_g^ -g〗_ ^('))/2],A_r^ }
可以返回的总值是合法的剩余燃料 g_ ^(* '），加上 A_r^ ，和最后一个至少超过〖T_g^ -g〗_ ^(')总和一样的部分的最大值。

用于燃料的以太币被给到在当前区块 B 中地址作为一个受益者的矿工。所以我定义在暂时状态数组 σp^ *中最后状态的前一个状态为* σ ^ *：*

（73） σ_ ^( )≡σ*(P)*^ *除：*

（74） σ ^(*) 〖[S(T)]〗*b*^ ≡σ(P)^ 〖[S(T)]〗*b*^ *-g* ^(*) T_p^

（75） σ_ ^* 〖[m]〗*b*^ ≡σ(P)^ 〖[m]〗*b*^ + *〖(T_g*^ *-g〗* ^)*T_p*^

（76） m≡B_(H_c^ *)*^

在删除所有自杀名单中出现的账号后，最后的状态 σ_ ^'*是被得到：*

（77） σ_ ^(')≡σ_ ^ 除：

（78） 〖∀*i*∈ *:* σ〗 ^' [i]≡⌀

最后，我们具体说明下 γ_ ^(g)是在交易中使用燃料的总和，γ_ ^1 是被这个交易创建的日记：

（79） γ_ ^(g) (σ，T)≡T_g^ -g_ ^(')

（80） γ_ ^1 (σ，T)≡A_l^

它们被使用去帮助定义这个交易收据（transaction receipt），以后还要讨论。

7. 创建合约（“Contract Creation”）

当创建一个账号这里会产生大量的内部参数以一个任意字节长度数组存储，i，：发送者（s），原始交易者（o），有效的燃料（g），燃料价格（p），天赋（“endowment”）（v）和 EVM 的初始化和最后表现出来的信息调用/合约创建堆栈（e）的深度。

我们正式的定义这个创建函数作为一个函数 Λ，用来估计它的值，并且与状态 σ 构成一个新的状态，剩余的燃料和产生的交易子状态（σ_ ^',g_ ^',A），在第六小结：

（81） σ_ ^',g_ ^',A≡Λ(σ,s,o,g,p,v,i,e)

这个新账号的地址被定义为仅仅包含发送者和随机数结构 RLP 编程 Keccak 哈希值的最右边 160 位。然后我们定义这个最后的地址给一个新的账号 a：

（82） a≡B_(96..255)^ (KEC(RLP((s,σ[s]*n*^ *-1))))

其中 KEC 是 Keccak 256 位哈希值函数，RLP 是 RLP 编译函数，B(a..b)^ (X)赋值到包含二进制数据 X 中范围为[a,b]内的位索引值，σ[x]是 x 地址状态，或用⌀表示没有值出现。

注意我们使用的是一个比发送者随机数要小的值；我们声称我们会增加发送者账号的随机数在这个交易呼叫之前，所以在这个有效的交易或者 VM 操作开始之前这个被用过的值则是这个发送者的随机数。

这个账号的随机数开始被定义为 0，余额则为之前的值，存储空间为空和哈希代码为 Keccak 256 位的空字符串哈希值；当前发送者的余额同时会从之前值中被减少。因此这

个变化的状态变成 σ_ ^*:*

（83） *σ_ ^≡σ* except

（84） σ_ ^* [a]≡(0,v+v_ ^',TRIE(⌀),KEC(( )))

（85） σ_ ^* 〖[s]〗*(b)^ ≡σ* ^ 〖[s]〗*(b)^ -v*

其中 v ^'是当前账号中提前出现的值，在事件中它是事先出现的:

（86） v_ ^'≡{ (0 if σ_ ^ 〖[a]〗_ ^ =⌀

{ σ_ ^ 〖[a]〗*(b)^ otherwise)*┤

最后，这个账号是通过执行模型（见小结 9）中初始化的 EVM 代码 i 的执行被初始化。代码执行能影响几个不是在从内部到执行状态事件：当前账号的存储能被改变，未来账号能被创建和未来的信息调用能被产生。同样的，这个代码执行函数 Ξ 赋值到最后的状态 σ ^()的数组中，有效的剩余燃料 g_ ^()，当前产生的子状态 A 和账号 o 的代码本身。

（87） (σ_ ^(),g_ ^(),A,o)≡Ξ(σ_ ^*,g,I)

其中 I 包含这个执行环境的参数，它定义在小结 9：

（88） I_(a)^ ≡ a

（89） I_(o)^ ≡ o

（90） I_(p)^ ≡ p

（91） I_(d)^ ≡( )

（92） I_(s)^ ≡ s

（93） I_(v)^ ≡ v

（94） I_(b)^ ≡ i

（95） I_e^ ≡ e

I_(d)^ 赋值到空的数字中，当做这里没有一个输入数据到这个调用中。I_H^ 没有特别的待遇而且它是从区块链中被决定。

代码执行消耗燃料，而且燃料不可能低于 0，因此执行可能退出在代码已经变成一个自然完整的状态前。在这个（或几个）特例（“exception”）中我们说缺乏燃料（“out-of-gas”）已经产生：这个赋值状态是作为一个空的集合⌀被定义，并且这所有创建的操作不会受到这个状态的影响，有效地离开它就像它是立即去尝试创建。

如果这个初始化代码成功地编译，一个最终的合约创建费用被消费，堆积代码 c 将成与创建合约全代码的大小成比例的消费：

（96） c ≡ G_codedeposit^ ×|o|

如果这里没有充足剩余燃料去支付这个费用，比如 g_ ^() < c，然后我们也声明一个缺乏

燃料异议。
剩余燃料将变为 0 在任意一个像这样的特例中，例如：如果这个创建作为一个交易收据被引导，然后它不会影响合约创建内部消耗的账单；换句话说它无论如何都必须支付。然后，这个交易的数值不是被转变成遗弃交易的地址当我们声明缺乏燃料。
如果像一个这样的特例没有出现，然后这个剩余燃料是被返回到最原始的发送者和新改变的状态是被允许保留。形式上，我们可能具体指明这个最后的状态，燃料，和子状态（σ_ ^',g_ ^',A）：
（97） g_ ^'≡{(0 if σ_ ^()=∅
@g_ ^() if g_ ^()<c∩ H_(i)^ <N_H^
@g_ ^()-c otherwise)┤

（98） σ_ ^'≡{(σ if σ_ ^()=∅
@σ_ ^() if g_ ^()<c∩ H_(i)^ <N_H^
@σ_ ^() except:
@σ_ ^' [a]*(c)^ =KEC(O) otherwise)┤*
在新状态σ ^'中的特例规定了 o，从初始化代码执行中最后的字节顺序，具体说明了最后代码本身
是为了新创建账号。
注意区块 N_H^ （家园版）的目的是成功地产生一个新的合约的能力，或者不是新的账号在没有转移这个值的情况下。在家园版之前，如果这里没有充足的燃料去支付 c，则一个账号在新合约地址下将被创建伴随着所有初始化次要的影响，而且这个值会被转移，但是没有合约代码被编译。

7.1 微妙（“Subtleties”）.注意当这个初始化代码正在执行时，新创建的地址出现但是不会出现内部的本身代码。因此任意信息调用在这段时间内处理因为没有代码被执行。如果这个初始化执行结束于一个自杀（“SUICIDE”）指令，则这件事无意义因为这个账号将被删除在交易被完成交易之前。由于一个正常的停止（“STOP”）代码，或者如果这个返回的代码是其他空形式。然后这个状态是通过一个僵尸账号被遗弃，而且任何剩余的余额将被永远地锁定在账号中。

8.信息调用（“Message Call”）
在信息调用执行的情况下，几个参数是被要求的：发送者（s-sender），交易的创建者

（o-originator），接受者（r-recipient），此账号中被执行过的代码（c-code,通常就是接受者），有效的燃料（g-gas），值（v-value），和燃料价格（p-price）与一个任意长度字节的数组（d），这个调用的输入数据和最后信息调用/合约创建当前深度的堆栈（e）。

除了赋值给一个新的状态和交易子状态，信息调用也有一个额外的元素 —— 用字节数组 o 表示的输出数据。这些将会被忽略当正在执行交易，然而信息调用能被初始化因为在这种情况下 VM 代码的执行和信息被使用。

（99） (σ_ ^(),g_ ^(),A,o)≡Θ(σ,s,o,r,c,g,p,v,v˜,d,e)

注意我们需要从执行内容中 v˜去区分被转移的值 v，为了委托调用（“DELEGATECALL”）指令。

我们定义 *σ(1)^，将第一个交易状态作为原始的状态但是伴随从发送者到接受者的转移值：*

（100） σ(1)^ [r]*(b)^ ≡σ* ^ [r]*(b)^ +v* ∩ *σ*(1)^ [s]*(b)^ ≡σ* ^ [s]*(b)^ -v*

总体来说，当前的工作是被认为如果 σ(1)^ [r]*( )^ 最初是没有被定义，它将被创建作为一个没有代码和状态或者余额为 0 和随机数的账号。因此之前的等式应该会被表示为：*

（101） σ(1)^ ≡σ*(1)^' except*

（102） σ(1)^ [s]*(b)^ ≡σ* ^ [s]*(b)^ -v*

（103） and σ(1)^' ≡ σ_ ^ except

（104） {(*σ(1)^' [r]≡(v,0,KEC(( )),TRIE(⌀)) if σ* ^ [r]= ⌀

@*σ1^' 〖[r]〗*(b)^ ≡ σ_ ^ 〖[r]〗*(b)^ +v o otherwise)*⊣

当前账号关联的代码（定义整个 Keccak 哈希的碎片为 σ ^ 〖[c]〗*(c)^ ）是依靠执行模式（见小结 9）被执行。仅仅作为合约创建，如果这个执行暂停在一个特别的方式中（例如：由于一个消耗完的燃料补给，堆栈溢出，无效的跳转目的地或者无效的指令），然后没有燃料是被返回到调用者，而且这个状态是立即地被回复到这个点在余额转移之前（比如 σ* ^ ）。

（105） 〖σ_ ^ 〗_ ^'≡{(σ_ ^ if σ_ ^( )=⌀

@σ_ ^( ) otherwise)⊣

（106） (σ_ ^(),g_ ^',s,o)≡{(Ξ*ECREC (σ*(1)^ ,g,I) if r=1

@Ξ*SHA256 (σ*(1)^ ,g,I) if r=2

@Ξ*SHA160 (σ*(1)^ ,g,l) if r=3

@Ξ*ID (σ*(1)^ ,g,l) if r=4

@Ξ_ (σ*(1)^ ,g,l) otherwise)*┤

（107） I(a)^ ≡r

（108） I_(o)^ ≡ o

（109） I_(p)^ ≡ p

（110） I_(d)^ ≡d

（111） I_(s)^ ≡ s

（112） I_(v)^ ≡ v˜

（113） I_e^ ≡ e

（114） Let KEC(I_(b)^) ≡σ_ ^ 〖[c]〗*(c)*^

它是被认为它的客户将在某个点的时候已经存储这对（KEC(I(b)^),I_(b)^ ）数据为了去产生 I_(b)^ 的可行性。

正如我们所见，这里有为了信息调用赋值的一般执行框架 Ξ 的四种惯用特例；他们四个被称为‘提前编译的’（‘precompiled’）合约，意思是作为一个后期可能成为与生俱来的扩展名的最初的设计成本价格。这四个合约地址分别为 1,2,3,和 4，且分别执行椭圆形重获公共钥匙函数（“SCREC”），SHA2 256 位哈希方案，RIPEMD 160 位哈希方案和一致性函数。

他们整个正式的定在是在附件 E。

9.执行模型（“Execution Model”）

执行模型具体说明怎么使用一系列字节代码指令和一个环境数据的小数组去改变这个系统状态。这些是通过一个虚拟状态机的正式形式被具体说明，正如大家所知的以太坊虚拟机（“Ethereum Virtual Machine - EVM”）。它是一个类似完全图灵机器（“quasi-Turing-complete machine”）；这个 quasi 条件是一个这样的事实：算力在内部被限制通过一个参数，燃料，它是限制已经完成的算力总数。

9.1.基础（“Basics”）。EVM 是一个简单基于堆栈结构。机制的字大小（堆栈项目的大小）是 256 位。这些被选择去完成 Keccak-256 位方案和椭圆形曲线算计。记忆模型（“memory”）是一个简单赋有字地址的字节数组。堆栈最大尺寸为 1024。这个机制也有一个不独立存储模型；这类似一个记忆的概念而不是一个字节数组。这个机制是可赋有字地址的字数组。不像可变的记忆模型，存储是不可变的而且是作为系统状态的一部分被保

持。在记忆和存储模型中所有的初始化数据被定义为 0。
这个机制不跟随标准的诺依曼结构（“Neumann architecture”）。它被特别地存储在只和一个特别指令互动的一个虚拟 ROM 中，而不是把存储程序代码放在一般可进入的记忆或存储器中。
在某些情况下这个机制有例外的执行，包括堆栈溢出和无效指令。在缺乏燃料（“out-of-gas - OOG”）特例中，他们不会脱离状态去改变完整性。话句话说，这个机制会立即分别暂定和记述这个问题到执行代理人（无论是交易的处理者，还是重复地大量产生执行环境）。

9.2.费用综述（“Fees Overview”）。费用（命名为燃料）被用在三个不同的情况下，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的本身费用（见附件 G）。第二中情况，燃料可能被减少因为一个下级的信息调用或者合约创建；这个形成创建，调用和调用代码（“CREATE,CALL, and CALLCODE”）费用中的一部分。第三种情况，燃料可能因为一个记忆模型使用量的上升而被支付。
对于一个账号的执行，可支付的记忆模型用量的总费用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。这个为了实时（“just in time”）的基础而被支付的；同样地，参考任意大于之前索引的至少 32 字节记忆区间在某种意义下都将导致一个额外的记忆模型用量费用。由于这个非常不像地址的费用曾经超出过 32 字节的限制。就如上述所说，工作流程必须能够管理这个不可测的事件。
存储的费用有一个极其微小的行为 —— 为了形成存储模型用量的最小化（直接与一个在所有结点上的大型状态数据通信），在存储器中清除一个元素的执行费用不仅仅被免除，而且会返回一些费用；事实上，这个返回的费用是被支付的预先费用，而且消耗的初始化存储模型用量实质上会远远超过正常的用量。
严格的 EVM 燃料消耗定义见附件 H。

9.3.执行环境（“Execution Environment”）。除了系统状态 σ，和计算剩余的燃料 g，这里还有一些在执行代理人必须提供的执行环境下重要使用过的信息块；他们被包含数组 I 中：
I_(a)^，拥有正在执行的代码的账号地址。
I_(o)^，发起执行交易中的发送者地址。
I_(p)^，发起执行交易中的燃料价格。
I_(d)^，执行中输入数据的字节数组；如果这个执行代理人是一个交易，这个将为交易数

据。

I_(s)^，触发这个代码执行的账号地址；如果这个执行代理人是一个交易，这个将为交易发送者。

I_(v)^，价值，单位为 wei，通过这个账号在执行中作为一样工序的部分；如果这个执行代理人是一个交易，这个将为交易的价值

I_(b)^，机制中被执行代码的字节数组

I_H^，当前区块的区块头

I_e^，当前信息调用或者合约创建的深度（比如调用的数值或者当前创建被执行）

执行模型定义为函数 Ξ，能计算出最后的状态〖σ_ ^ 〗_ ^'，剩余燃料 g_ ^'，自杀列表 s，日记序列 l，返回值 r 和最好的输出 o，他们的定义为：

（115）（〖σ_ ^ 〗_ ^',g_ ^',s,l,r,o)≡Ξ(σ,g,I)

9.4.执行综述（"Execution Overview"）。我们现在必须去定义这个函数 Ξ。在很多实际的情况下，这将被模型化以整个系统状态 σ 和机制状态 μ 的对比作为一个重复的过程。正式的来说，我们定义它用函数 X 去重复。这个函数使用一个迭代函数 O（定义状态机制中单循环的结果）和函数 Z，如果当前状态是一个特例的机制暂停状态，和函数 H ——具体说明这个指令的输出数据如果仅仅当前状态是一个正常的机制暂停状态。

空的序列表示为 ( )，它不等于空的集合∅；这是非常重要的当去理解 H 的输出数据，赋值到∅会连续执行但是赋值到系列（潜在的空序列）会暂停执行。

（116） Ξ(σ,g,I)≡X_(0,1,2,4)^ ((σ,μ,A_ ^0,I))

（117） μ*(g)*^ ≡*g*

（118） *μ*(pc)^ ≡0

（119） μ*(m)*^ ≡*(0,0,...)*

（120） *μ*(i)^ ≡0

（121） μ*(s)*^ ≡*( )*

（122） *X((σ,μ,A* ^ ,I))≡{((∅,μ,A_ ^0,I,( )) if Z(σ,μ,I)

O(σ,μ,A_ ^ ,I)•o if o≠∅

X(O(σ,μ,A_ ^ ,I)) otherwise)┤

其中

（123） o≡H(μ,I)

（124） (a,b,c) •d≡(a,b,c,d)

注意我们必须舍弃从函数 X 正确返回和赋值到函数 Ξ 数组的第四个值，因为这里下角标

为 X_(0,1,2,4)^ 。
X 是被循环（重复地操作，但是这些工具通常被期待去使用一个迭代的循环）直到 Z 变成 true，表示当前状态是特例和机制必须被暂停而且舍弃全部变化，或者 H 变成一个系列（是空序列而不是一个空的集合），代表这个机制已经得到了一个可控制的暂停。

9.4.1.机制状态（“Machine State”）。这个机制状态 μ 是作为一个数组（g,pc,m,i,s）被定义，它们分别可用的燃料，程序计数器 pc∈P_(256)^，记忆的内容，在记忆中积极的字数（从 0 不停的计算），和堆栈内容。这个记忆内容 μ*(m)^ 是长度为 2* ^256 值为 0 的一个空系列。
为了阅读的简单性，记忆的指令以小的大写字母创建（例如 ADD），应该被解释为他们数值的等价；整个指令集在给出的附件 H 具体说明。
为了定义 Z，H 和 O，我们定义 w 作为当前被执行的操作：
（125） w ≡{(I_(b)^ [μ*(pc)^] if μ*(pc)^ <||I_(b)^ ||
STOP otherwise)┤
我们也认为被修复的 δ 和 α 总和，具体说明了堆栈项目减少和增加，而且都能用指令去描述，并且一个指令消耗函数 C 将用燃料赋值到给定执行指令的全消耗中。

9.4.2.特例的暂停（“Exceptional Halting”）。特例的暂停函数 Z 被定义为：
（126） Z(σ,μ,I)≡μ*(g)^ <C((σ,μ,I))* ∪
δw^ =∅ ∪
*|(|μ_s^ |)|<δ_w^* ∪
(w ∈ {JUMP,JUMP} ∩
μs^ [0]∉D(I(b)^)) ∪
|(|μ_s^ |)|-δ_w^ +α_w^ >1024
这些执行都属于特例的暂停的状态：如果这里有不充足的燃料，无效的指令（就是为什么 δ 的下角标没有定义），不充分的堆栈项目，一个 JUMP/JUMPI 的目的地是无效或者新的堆栈大小将大于 1024。 机灵的读者将很容易的发现在整个执行过程中没有一个指令能直接进入特例的暂停。

9.4.3.跳转目的地的有效性（“Jump Destination Validity”）。我们之前使用 D 作为一个函数去决定给定
正在允许代码中的有效跳转目的地的集合。我们定义 JUMPDEST 指令能在正在运行代码中的任意位置。

像这样所有的位置必须是在有效指令边界，而不是使用 PUSH 操作进入数据的一部分，而且必须出现在被详细说明定义的部分代码中（而不是使用被详细说明定义的 STOP 指令去跟踪它）。

公式表达为：

（127） D(c)≡D_J^ (c,0)

其中：

（128） D_J^ (c,0)≡{({ } if i≥|c|

{i} ∪ D_J^ (c,N(i,c[i])) if c[i]=JUMPDEST

D_J^ (c,N(i,c[i])) otherwise)┤

其中 N 是代码中下一个有效的指令位置，跳过 PUSH 指令的数据，如果下列任意式子成立：

（129） N(i,w)≡{(i+w-PUSH1+2 if W∈[PUSH1,PUSH32]

i+1 otherwise)┤

9.4.4.正常暂停（“Normal Halting”）。正常暂停函数 H 被定义为：

（130） H(μ,I)≡{(H_RETURN^ (μ) if w=RETURN

( ) if w∈{STOP,SUICIDE}

∅ otherwise)┤

操作中返回的数据，RETUREN，有一个特别的函数 H_RETURN^，详细定义见附件 H。

9.5.执行循环（“The Execution Cycle”）。堆栈项目是从最左边被增加或减少，低索引的序列部分；剩下的所有项目都被保留不变：

（131） O(σ,μ,A_ ^ ,I) ≡ (〖σ_ ^ 〗',〖μ_ ^ 〗_ ^',A_ ^',I)

（132） Δ ≡α*w*^ -*δw*^

（133） |(|μ_s^(′) |)| ≡ |(|μ_s^ |)|+Δ

（134） ∀x∈[α_w^ ,|(|μ_s^(^′) |)|]:μ_s^(^′) [x] 〖≡ μ〗_s^ [x+Δ]

燃料通过指令的燃料消耗和大量的指令被减少，程序计数器在每一个循环上增加，对于这三个特例，我们假设了一个函数 J，下角标是二个指令中的一个，根据以下的式子赋值：

（135） μg^′ ≡ μ_g^ -C(〖σ ^ 〗_ ^ ,〖μ_ ^ 〗_ ^ ,I)

（136） μ*(pc)^(′) ≡ {(J_JUMP^ (μ) if w=JUMP*

J_JUMPI^ (μ) if w=JUMPI

N(μ(pc)^(^),w) otherwise)┤

在一般情况下，我们认为这个记忆（“memory”），自杀（“suicide”）列表和系统状态不改变：

（137） μ(m)^(') ≡ μ(m)^

（138） 〖μi^(') ≡ μ〗(i)^

（139） A_ ^(') ≡ A

（140） σ_ ^' ≡ 〖σ_ ^ 〗_ ^

然而，指令通常会改变一个到几个元素的值。改变的元素会被通过附件 H 中强调的指令一一列举出来，α 和 δ 边上的值和一个燃料需求的公式描述。

10.区块树到区块链（“Blocktree To Blockchain”）

标准的区块链是一个区块树上所有元素从根到叶子的路径（“path”）。为了形成什么是路径的共识，我们概念性地定义这个路径是已经计算过最长的路径，或者是最重的路径。叶子当前的数值是最清晰的因素去辨别最重的路径，它也是路径中当前区块的数值，不算上没有挖的创世块。随着路径越来越长，总挖矿的影响越来越大而且为了到达叶子上它必须被完成。这是一个已有方案的改进，比如它已经应用在比特币协议（“Bitcoin-derived protocols”）中。

因为一个区块头包含难度（difficult），区块头能充分证明已完成的计算有效性。任意一个区块贡献到总的计算或者区块链的整个难度中。

因此我们重复地定义区块 B 的难度为：

（141） B_t^ ≡ B_t^'+B_d^

（142） B_ ^' ≡ P(B_H^)

在给定一个区块 B，B_t^ 是它本身总难度，B_ ^'是它本身当前区块和 B_d^ 是它本身的难度。

区块定稿（“Block Finalisation”）

区块定稿的步骤包含了以下四个步骤：

确认 ommers 的有效性（如果挖矿，确定）；

确认交易的有效性（如果挖矿，确定）；

应用奖励；

证实状态和随机数（如果挖矿，计算有效性）。

Ommer Validation（“有效性”）。Ommer 头文件的有效性表示去证明每一个 ommer 头文件都是一个有效的文件而且满足第 N 代 Ommer 和当前区块 N≤6 的关系。Ommer 头文

件的最大值是 2.公式表达为：

（143） |(|B_U^ |)| ≤ 2^(U ∈B_U^)▒〖V(U)∩k(U,P(B_H^)H^ ,6)〗

其中 k 表示是亲戚关系的（“is-kin”）性质：

（144） k(U,H,n) ≡ {▒(false if n=0@s(U,H) ∨k(U,P(H)_H^ ,n-1) otherwise)┤

其中 s 表示是兄妹的（“is-sibling”）性质：

（145） s(U,H)≡(P(H)≡P(U)∩H≠U∩U∉B(H)U^)

其中 B(H)是响应头文件 H 的区块。

交易的有效性（“Transaction Validation”）。给定使用过的燃料（“gasUsed”）必须诚实地回应到交易列表：B(H_g^)^，在区块中使用的全部燃料，必须等于最后交易所积累下来被使用过燃料总和：

（146） B_(H_g^)^ = 〖l(R)〗 *u^*

奖励应用（“Reward Application”）。对一个区块来说，奖励的应用包含当前区块收益账号余额中上升的账号列表和一个随机总和的 ommer。我们通过 R_b^ 上升当前区块的收益账号；对于每一个 ommer，我们是采取区块额外的 1/32 奖励去上升一个区块收益的账号，而且 ommer 的受益者是根据区块当前的数值去决定奖励。我们定义函数 Ω 去表达：

（147） Ω(B,σ) ≡ σ ^':σ_ ^'= σ except

（148） σ_ ^' 〖[B_(H_g^)^]〗 *b^ = σ* ^ 〖[B_(H_g^)^]〗 *b^ +(1+(|(|B_U^*

*|)|)/32)R_b^*

（149） ∀(〖U∈B〗 *U^)^ :*

σ ^' 〖[U_c^]〗 *b^ = σ* ^ 〖[U_c^]〗 *b^ +(1+(1)/8(U_i^ -B*(H_i^)^))R_b^

如果这里有在 ommers 和区块之间的受益者账号冲突问题（比如：两个相同的 ommer 受益者账号或者一个 ommer 受益者账号作为当前的区块），剩下的都被积累地应用。

我们定义区块奖励为 5 以太币：

（150） Let R_b^ = 5×〖10〗^18

状态&随机数的有效性（“State & Nonce Validation”）。我们现在可能定义这个函数，Γ，去绘制一个区块 B 到它本身初始化状态：

（151） Γ(B) ≡ {▒(σ0^ *if P(B_H^)=∅@σi^ :TRIE(L_S^ (σ_i^))=* 〖*P(BH^)*〗(H_r^)^ otherwise)┤

其中，TRIE(L_S^ (σ*i^))表示状态 σ_i^ 的梅克尔帕特里夏树（“trie”）根节点的哈希值；它被认为某些工具将存储这个状态的数据库，它是繁琐和有效的因为这个梅克尔帕特里夏树是一个永不会改变的数据结构。

最后定义 Φ，区块交易函数，是一个绘制不完整区块 B 到一个完整区块 B ^'的过程：

（152） Φ(B) 〖≡ B〗_ ^' ： 〖 B〗_ ^' = 〖 B〗_ ^* except:

（153） 〖 B〗*n^' = n ：x≤ (〖2〗* ^256)/(〖 H〗*d^)*

（154） 〖B〗(m)^' = m with (x,m)= PoW(〖 B〗*(≠n)^*,n,d)*

（155） 〖B〗 ^* ≡ B except ： 〖 B〗*r^* = r(Π(Γ(B),B))*

具体的解释在附件 J，d 成为一个数据集。

去详细介绍当前工作的前期，Π 是一个状态交易函数，被以区块定稿函数 Ω 和交易升级函数 Y 去定义。

在之前的定义中，〖R[n]〗σ^ ， 〖 R[n]〗I^ 和〖R[n]〗_u^ 是在每一个交易后的第 n 个响应的状态，日记和积累所使用的燃料（〖R[n]〗_b^ ，是在数组中的第四个元素，它已经被用日记去定义）。之前这些只是作为从应用中响应交易的最后状态到先前交易的最后状态而被简单地定义（或者说，在第一个像这样的交易情况下的区块初始化状态）。

（156） 〖R[n]〗σ^ = {(Γ(B) if n<0@Y(〖 R[n-1]〗σ^ ,〖B〗T^ [n]) otherwise)┤

在〖B〗_R^ 〖[n]〗_u^ 的情况下，我们采取一个类似用每一个作为升级响应总和的交易使用的燃料和前一个项目的方法去定义这个项目（或者说，是第 0 个，如果它是第 1 个的话），给出一个运转的总和：

（157） 〖R[n]〗u^ = {(0 if n<0@〖Y〗 ^g (〖 R[n-1]〗σ^ ,〖B〗T^ [n]) @ + 〖R[n-1]〗_u^ otherwise)┤

对于〖R[n]〗I^ ，我们使用〖Y〗 ^I 函数方便的定义这个交易执行函数。

（158） 〖 R[n]〗*I^ = 〖Y〗 ^I (〖 R[n-1]〗σ^ ,〖B〗T^ [n])*

最后，我们定义 Π 作为一个新的状态去给定区块奖励函数 Ω 去应用到最终交易的最后状态，〖 〖I(B〗R^)〗σ^ ：

（159） Π(σ,B) ≡ Ω(B,〖I(R)〗σ^)

因此这个完整的区块交易机制，less PoW，工作机制证明（“proof-of-work”）函数被定义。

工作机制证明挖矿（“Mining Proof-of-Work”）。工作机制证明（PoW）作为一个密码地保密随机数存在，提供不可被怀疑而且被早已悬挂标记过的一些数值 n 去决定的一个特别的计算力总和。它是被使用去执行区块的安全性，通过给定的意义和信用去改变难度（“difficulty”）（还有其他元素，扩展名（“extension”），总的难度和）。然而，因为挖出一个新的区块会附带一些奖励，工作机制证明不仅是一个未来区块链继续存在的标准安全信心的方法函数，而且也是一个健康的分配机制。

对于以上的原因，这里有两个重要的工作机制证明目标：首先，它应该是尽可能的被更多的人去接受的。特别而且罕见的硬件的条件，或者说奖励，应该被减到最小。理想说，这些导致这个分配模型尽可能的开源，和形成一个简单的挖矿动作与大致在全世界各个角落下是一样的比例的电子以太币交换。

第二，它应该不能去尽可能的制造出超线性的利益，而且它没有特别高地初始化障碍。像如此的机制允许一个资金充足的敌人去获得令人讨厌的网络挖矿总权利（算力）和给他们一个超线性的奖励（在他们这种情况下会导致曲线形的分配）而且减少网络的安全性。

比特币的世界中一个灾难是 ASICs。这些是被具体解释为电脑硬件仅仅是为了处理一个简单的任务而存在。在比特币的案例中，这个任务就是 SHA256 哈希函数。当 ASICs 为了工作机制证明函数而存在，它们的目标都是被替代损失。因为这样，一个可抵抗 ASIC 的工作机制证明函数（比如：难度或者经济的无效率去执行特别的电脑硬件）已经作为著名的神奇般的解决方法去识别。

二个防 ASIC 漏洞的方法：第一就是去让它变成有序列的硬件记忆（"memory-hard"），比如：设计一个函数 —— 随机数的决定需要大量的记忆和带宽以至于这些记忆不能被并行同时地去发现多个随机数。第二个方向就是去制造去执行同样目的不同计算类型；同样的来说，特别硬件的意思对于同样目的任务集合来说就是通常的硬件，和桌面上的附件都非常类似特别的硬件对于任务来说。在以太坊 1.0 时代，我们已经选择了第一个方法。

更公式化的说，这个工作机制证明函数来自 PoW：

（160） m = 〖H〗(m)^ ∩ n≤ (〖2〗_ ^256)/(〖H〗d^) with (m,n)= PoW(H(≠n)^ , 〖H〗n^ ,d)

*其中 H(≠n)^ 是一个新的区块头文件，但是它没有随机数和混合哈希元素；〖H〗n^ 是区块头文件的随机数；d 是一个需要计算呼和哈希值的大数据集合和〖H〗(d)^ 是一个新区块的难度值（例如：这个区块难度见小结 10）。PoW 一个工作机制证明函数，它是用成为混合哈希值的第一个项目赋值到一个数组和取决于 H 和 d 密码地成为冒充的随机数的第二个项目。这以下的算法被称为 Ethash 和被在下文描述。

Ethash。Ethash 是以太坊 1.0 计划中的 PoW 算法。它是 Dagger-Hashimoto 的最后版本，被 Buterin

[2013b]和 Dryja[2014] 介绍。虽然它不再大概地被称为 Dagger-Hashimoto 因为很多算法的最初特性已经被大量地修改这升级在上个月的研究中。算法的通常路径采取以下方法：这里出现了一个种子（"seed"），它能被每一个区块计算而找到通过扫描区块头文件。在种子中，一个能计算假冒的随机缓冲，〖J〗_cacheinit^ 字节在初始化尺寸中。轻节

点客户（“light clients”）存储这个缓冲。在缓冲中，我们能产生一个数据集合，〖J〗_datasetinit^ 字节在初始化尺寸中，伴随着在数据集合中每个从缓冲中仅仅是一个小数字的项目的特性。全节点客户和矿工保存着数据集合。数据集合随着时间线性增长。
挖矿包含着抓取数据集合的随机碎片，再将它们哈希在一起。证明能通过使用缓冲去再次产生你所需要的数据集合的某个碎片在低端记忆中被完成，所以你仅仅需要存储的是缓冲。较大的数据集合是被升级一次到〖J〗_epoch^ 区块中，所以大多数矿工的努力都是阅读这些数据集合，而不是尝试改变它。这些被提及到的参数和算法都在附件 J 中被详细地解释。
执行合约（“Implementing Contracts”）
这里有几种设计合约的形式，并且允许使用特别有用的行为；我将简单地讨论它们中的二个是数据喂养（“data feeds”）和随机数字（“random numbers”）。
数据喂养（“Data Feeds”）。一个数据喂养合约是提供简单的服务：它允许外部的信息可能进入以太坊内。这个信息的精确度和及时性不被保证，而且它是第二个合约作者的任务——这个合约被用作数据喂养——去决定单个数据喂养是否能被替代的可信度。
通常的模式包含了在以太坊中的单个合约，当给定一个信息调用，关于一个外部现象有规律的信息的回应。一个例子可能是纽约城市的本地温度。这个可能作为返回在存储器中一些总所周知的点的合约被执行。当然在存储器的中这些点必须是正确的温度，而且这个模式的第二个部分将作为外部的服务器去运行一个以太坊节点，和立即地在新的区块中被发现，产生一个新的有效的交易，发送到合约，同时更新存储器中上述的值。合约的代码将仅仅同意在包含上述的服务器上的更新。
随机数字（“Random Numbers”）。显然，在决定论的系统中提供的随机数字是一个不可能实现的任务。然而，我们能在交易的同时使用一些普通不可知的数据去估计这个假冒的随机数字。这样的数据可能包含区块的哈希值，区块的时间戳，和区块受益人的地址。加强它的难度为了不让恶意的矿工去控制这些值，其中使用区块哈希（“BLOCKHASH”）操作为了去使用之前的 256 个区块哈希值去冒充一个随机数字。对于这样一系列的数字，一个微不足道的方法就是去添加一些不变的总额和切分结果。
未来的方向（“Future Directions”）
在以后，状态数据库将不被强行维持所有之前的状态 trie（梅克尔帕特里夏树）结构。它应该保持每一个节点的年龄和最后抛弃岂不是当前的节点也不是检查点的节点；检查点，或者一系列允许特别区块状态 trie 被旋转数据库的节点，可以被使用去替代一个需要的算力总和的最大限度值，它是为了重新获得整个区块链中的任意一个状态。

区块链巩固（“consolidation”）被使用为了减少每个客户需要下载的全节点和挖矿节点的区块总数。及时在给定节点上的一个 trie 结构的压缩档案（也许是第 10000 个中一个）可能被保持下来通过伙伴（“peer”）网络，且有效地重置这个创世块。这将减少去下载单个档案加上有难度的最大限度区块的总和。

最后，区块链压缩也许被引导：在一些不变的区块总和中没有发送或接受交易的状态 trie 节点可能被抛弃，同时减少以太币的遗漏和状态数据库的增长。

可扩展性（“Scalability”）。可扩展性剩下一个永恒不变的忧虑。随着一个广义的状态交易函数，它变成困难的一部分而且平行交易的去应用分而治之的策略。仍未解决的是，这个系统的动态范围值保留了必要固定的和作为一个普通交易值上升，其中较少价值的将被忽略，成为经济性不重要的去保存在主要的记账本中。然而，一些策略存在可能潜在地被发掘而去提供一个更为重要地可扩展性的协议。

一些来自于等级体系的结构，通过巩固在主链中最小较轻最大的链去实现或者通过最小交易集合中增长的联盟和附着（通过 proof-of-work）去建立一个主区块，从而可能允许交易联盟的平行化和区块建立。平行机制也可能来自一个优先的平行区块链集合，同时巩固每一个区块和复制或者无效交易的抛弃。

最后，证明算力，如果创造出逐渐足够有效和有效率的，可能提供一个路径去允许这个工作机制证明去成为最后状态的 证明。

总结

我已经介绍，讨论和形式上定义了以太坊的协议。通过这个协议，读者可能在以太坊网络上执行了一个节点和加入其它人在一个去中心化安全社会开放的系统中。合约可能被实名化为了计算性地具体说明和自动化地强力执行互动的规则。

致谢（“Acknowledgements”）

重要的维护，有用的矫正和建议是被很多来自以太坊 DEV 组织和以太坊社区的工作人员提供，其中包括 Christoph Jentzsch, Gustav Simonsson, Aeron Buchanan, Pawel Bylica, Jutta Steiner, Nick Savers, Viktor Tron, Marko Simovic and Vitalik Buterin。

参考

Jacob Aron. BitCoin software *nds new life. New Scientist, 213(2847):20, 2012.*

Adam Back. Hashcash - Amortizable Publicly Auditable Cost-Functions. 2002.

URL{http://www.hashcash.org/papers/amortizable.pdf}.

Roman Boutellier and Mareike Heinzen. Pirates, Pioneers,Innovators and Imitators. In Growth Through Innovation, pages 85{96. Springer, 2014.

Vitalik Buterin. Ethereum: A Next-Generation SmartContract and Decentralized Application Platform.2013a. URL {http://ethereum.org/ethereum.html}.

Vitalik Buterin. Dagger: A Memory-Hard to Compute,Memory-Easy to Verify Scrypt Alternative. 2013b. URL{http://vitalik.ca/ethereum/dagger.html}.

Thaddeus Dryja. Hashimoto: I/O bound proof of work.2014. URL {https://mirrorx.com/files/hashimoto.pdf}.

Cynthia Dwork and Moni Naor. Pricing via processing orcombatting junk mail. In In 12th Annual International Cryptology Conference, pages 139{147, 1992.

Phong Vo Glenn Fowler, Landon Curt Noll. FowlerNollVo hash function. 1991. URL {https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function # cite_note-2}.

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle,and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In Cryptographic Hardware and Embedded Systems-CHES 2004,pages 119{132. Springer, 2004.

Sergio Demian Lerner. Strict Memory Hard Hashing Functions.2014. URL {http://www.hashcash.org/papers/ memohash.pdf}.

Mark Miller. The Future of Law. In paper delivered at the Extro 3 Conference (August 9), 1997.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Consulted, 1:2012, 2008.

Meni Rosenfeld. Overview of Colored Coins. 2012. URL {https://bitcoil.co.il/BitcoinX.pdf}.

Yonatan Sompolinsky and Aviv Zohar. Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains, 2013. URL {CryptologyePrintArchive,Report2013/881}. http://eprint.iacr.org/.

Simon Sprankel. Technical Basis of Digital Currencies,2013.

Nick Szabo. Formalizing and securing relationships on public networks. First Monday, 2(9), 1997.

Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gn Sirer. Karma: A secure economic framework for peer-to-peer resource sharing, 2003.

*J. R. Willett. MasterCoin Complete Specic*ation. 2013. URL {https://github.com/mastercoin-MSC/spec}.

来源：
以太坊黄皮书连载（一）http://ethfans.org/jordengao/articles/223
以太坊黄皮书连载（二）http://ethfans.org/jordengao/articles/233
以太坊黄皮书连载（三）http://ethfans.org/jordengao/articles/241
以太坊黄皮书连载（四）http://ethfans.org/jordengao/articles/270
以太坊黄皮书连载（五）http://ethfans.org/jordengao/articles/271
以太坊黄皮书连载（六）http://ethfans.org/jordengao/articles/274