以太坊（Ethereum）:下一代智能合约和去中心化应用平台

原文地址：http://ethereum.org/ethereum.html

翻译：巨蟹 比特币咕噜

在过去的几个月里，人们倾注了大量的兴趣去探讨如何把类比特币块链 - 一种能够让整个世界就一个公共拥有的数据库的内容达成一致的机制 - 应用于货币以外的领域。通常被讨论的例子包括“彩色币”- 一种用块链上的数字资产来表示定制货币和金融工具的思想，“智能资产”-象汽车这样的能够通过追踪块链上的彩色币来确定当前合法拥有者的物理实体，此外还有诸如去中心化交易所，金融衍生品，对等投注和基于块链的身份和信誉系统之类的更高级的应用。可能所有这一切之中最具雄心的概念是自治代理或者去中心化自治机构 - 资源和资金被密码学块链上的自我强制的智能合约以自治方式管理，从而避开了对于法律合约和组织规章的依赖，并且没有任何中央控制的在块链上运行的自治实体。

然而，这些应用中的大部分都是难以实施的，原因很简单，因为比特币的脚本系统，以及诸如基于比特币的彩色币和所谓“元币”（metacoins）等下一代加密货币在允许 DACs 所需要的任意复杂的计算这一点上都非常受限。本项目旨在提取上述协议中的创新并将其通用化-创造一个功能完善的，图灵完备的（但是可深度自由调整的）加密账簿，它允许参与者编写任意复杂的完全存在于并且被块链传递的合约，自治代理和关系。用户可以把以太坊当成一种“金融积木”来用，而不是受限于某一特定的交易类型集合。- 这就是说，一个人能够简单地通过协议内置的脚本语言编码去实施他所希望的特性。定制货币，金融衍生品，身份系统和去中心化组织的创建将变得容易，构建以太坊开发者未曾想象得到的交易类型也将是可能的。总之，我们相信这样的设计是迈向“加密货币 2.0”的坚实一步；我们希望以太坊的出现对于加密货币生态系统的标志性意义，正如 1999 年 Web2.0 的对于只有静态内容的互联网一样。

目录

为何需要新的平台

当试图创建新的应用，尤其是在密码学或加密货币领域等如此精妙的领域新建应用的时候，第一，也是正确的反应是尽可能地去使用现存的协议。如果现存的技术能够完全解决问题就没有必要去创建新的货币甚至新的协议。实际上，正是尝试在比特币协议之上构建智能资产，智能合约和去中心化自治机构（DAC）的解决方案所带来的困惑促成了我们对下一代加密货币协议的最初兴趣。在研究中我们发现，虽然比特币协议对于货币，基础多方签名契约以及智能合约的简单版本是足够的，仍有一些根本上的限制使得它只适用于一个非常受限的业务特性范围。

彩色币

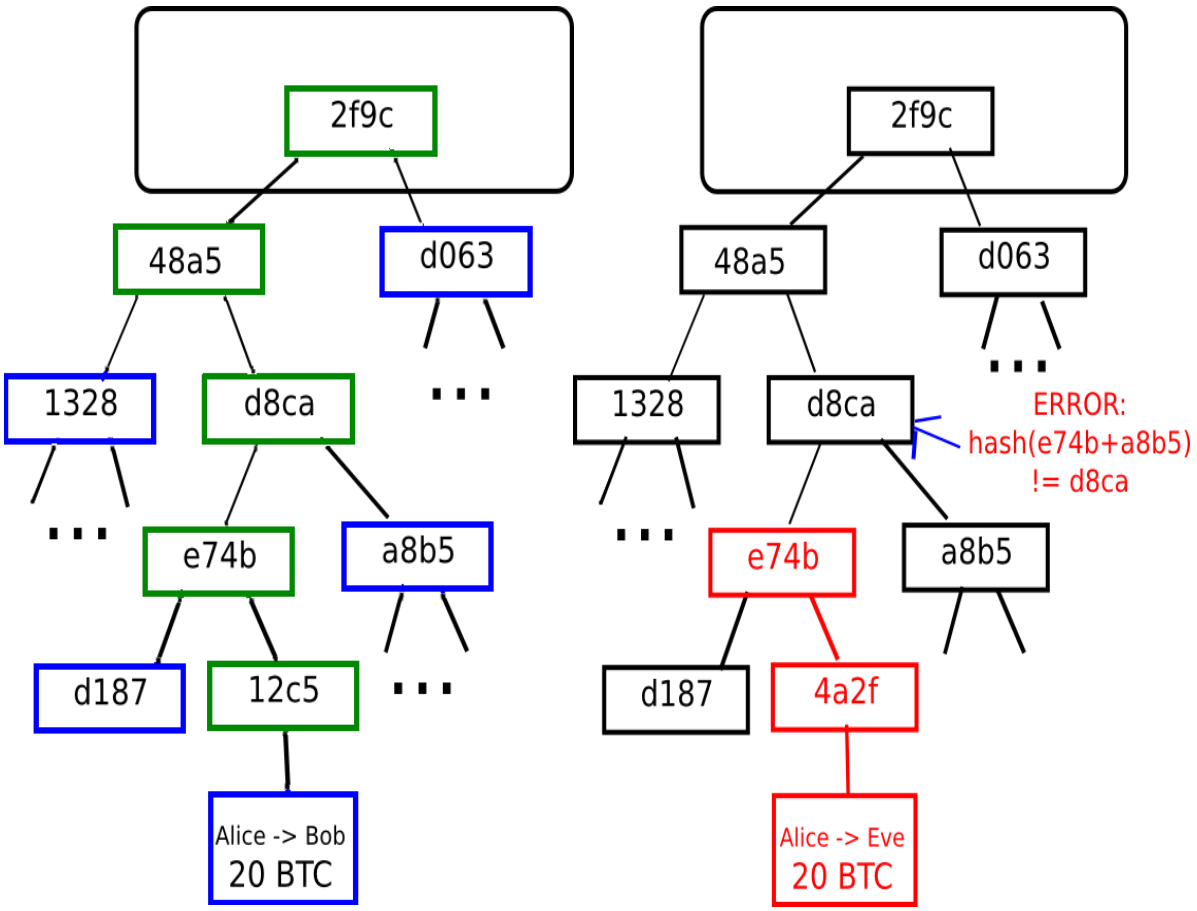
在块链之上构建智能资产和定制货币及资产管理系统的最初尝试是在比特币协议之上叠加新的协议，许多支持者把这种方法与互联网协议栈中 HTTP 叠加于 TCP/IP 之上的方式作类比。大体上彩色币的定义如下：

1. 彩色币发行者确定一个给定的交易输出 H:i(H 为交易的哈希值，i 为输出序号)代表一种特定的资产,并且发布”色彩定义”指定该交易输出代表什么(例如 H:i 中的 1 聪 = 1 盎司可由 amagimetals.com 兑付的黄金)。
2. 其他人在彩色币客户端“安装“色彩定义文件。
3. 当该”色彩定义”首次发布的时候，输出 H:i 是拥有该彩色的唯一交易输出。
4. 如果一个交易花费了带有色彩 X 的输入,则它的输出将同样具有色彩 X。例如，如果输出 H:i 的拥有者创造了一个交易将该输出分发至五个地址，则这些输出将同样具有色彩 X。如果一个交易的输入拥有不同的色彩，则一个“色彩传递规则“或“色彩内核”将确定每一个输出将拥有的色彩（例如一个非常初级的规则可能规定输出 0 将拥有与输入 0 一样的色彩，输出 1 将拥有和输入 1 一样的色彩，依次类推）。
5. 当一个彩色币客户端接受到一个新的交易输出的时候，它将使用基于色彩内核的回溯算法来确定输出的颜色。因为规则是确定的，所有的客户端会一致同意某笔输出应该拥有什么颜色。

然而，这个协议有几个根本性的缺陷：

1. 简化支付验证的困难 - 比特币的默克尔树结构支持一个叫“简化支付验证”的协议， 通过该协议一个没有下载完整块链的客户端也能够通过向其它节点索要包括从交易哈希沿默克尔树上溯至块链头处的根哈希的哈希序列来快速确认交易输出的正确性。为了安全客户端依然需要下载块链头，但相比下载完整块链的做法，所需的带宽和验证时间都成千倍地降低了。而对于彩色币，这样的简化支付验证将变得非常困难。原因在于通过简单地检查默克尔树来确定一个交易输出的色彩是不可能的，这里需要采用后向扫描算法，读取可能成百上千的交易数据并且通过默克尔树来验证每笔交易的正确性。才可能完全地确认某笔交易拥有某种色彩。经过了一年多的研究，包括来自我们的帮助，仍未为此问题寻找到解决方案。

比特币的简化支付验证



左：仅提供默克尔树（Merkle tree）上的少量节点已经足够给出分支的合法证明。

右：任何对于默克尔树的任何部分进行改变的尝试都会最终导致链上某处的不一致。

2. 与脚本的不兼容 – 如前所述，比特币有一个适度灵活的脚本系统，允许用户签署诸如“我发布此交易输出给任何愿意支付我 1BTC 的人”这样形式的交易。其它的例子包括担保合约，便捷小额支付以及链上拍卖。然而，这个系统固有地识别不了“色彩”；这就是说，你无法发布象“我发布此交易输出给任何愿意支付我在创世交易 H:i 中定义的 1 个金币的人”这样的交易，因为脚本语言根本不知道这里居然还会有不同的色彩。这带来的主要后果是，虽然不依赖于信任的不同色彩的币的交换依然是可能的，但无法下买卖单却导致无法运行一个完全的去中心化的交易所。
3. 与比特币一样的局限 – 理论上，链上协议可以支持高级衍生品，赌约和本文后面将详述的多种条件转让。但彩色币继承的比特币的局限使得上述金融工具中的许多无法实现。

元币

另一个如 HTTP 建立在 TCP 上面那样在比特币协议之上叠加协议建立起来的概念是“元币”。元币的概念很简单：元币协议提供了一种把元币交易数据写入比特币交易输出的方法，一个元币节点会处理所有的比特币交易，评估同时是合法元币交易的比特币交易以确定任何给定时刻的平衡账目。例如，一个简单的元币协议可能要求一个交易有四个输出：MARKER, FROM, TO 和 VALUE。MARKER 是一个特殊的标识符字段用来把一个交易标识成元币交易，FROM 是币的发出地址， TO 是币要发往的地址，VALUE 是表示交易数额的字段。因为比特币协议并未意识到元币协议的存在，因此不会拒绝非法的元币交易，元币协议必须根据比特币交易的第一个输出是否为 MARKER 识别出元币交易并作相应处理。这样的元币协议的相关部分的编码可能是类似这样的：

```
if tx.output[0] != MARKER:
    break
else if balance[tx.output[1]] < decode_value(tx.output[3]):
```

```
        break
    else if not tx.hasSignature(tx.output[1]):
        break
    else:
        balance[tx.output[1]] -= decode_value(tx.output[3]);
        balance[tx.output[2]] += decode_value(tx.output[3]);
```

元币协议的好处是允许更多的高级交易类型，包括定制货币，去中心化交易所，衍生品等，这些交易类型是作为基础的比特币协议本身无法实现的。然而，比特币之上的元币有一个主要的缺点：在彩色币那里已经很困难的简化交易验证，在元币这里干脆就是不可能的。原因是，当使用简化交易验证去确认一个由地址 X 发送 30 个元币的交易时，并不必然意味着地址 X 拥有 30 个元币；如果发送者没有 30 个元币从而该交易是非法的怎么办？找出当前状态的任何一部分都需要对所有交易进行一次彻底的后向扫描直至元币的初始发行交易，以判断当前交易是否合法。这使得任何不下载庞大的完整比特币块链的轻客户端都不会是真正安全的。

以上两个例子推出以下结论：努力在比特币协议之上建立更高级协议，正如在 TCP 之上建立 HTTP，是值得赞赏的，也确实是走向更高级的去中心化应用的正确道路。然而，在比特币协议之上创建彩色币和元币的尝试更象是在 SMTP 上创建 HTTP；SMTP 协议是用来传输电子邮件消息的，而不是用来作为一般互联网通信的基础协议的，（在 SMTP 上创建 HTTP）将不得不使用许多低效的以及结构丑陋的方法使其能够工作。与之类似，虽然比特币在简单交易和价值储存方面是一个出色的协议，但它绝非是被设计用来，而且前述证据表明它不可能被用来，作为通用金融端到端协议的底层协议。

通过建立自己的块链，通过在每一个区块保存一个表示当前每个地址的平衡账目的清晰的“状态树”和一个表示当前区块和上一个区块间交易的“交易表”，以太坊解决了可扩展性问题。以太坊合约将被允许在持久内存中存储数据，这样的内存加上图灵完备的脚本语言将使在单个合约中编码一个完整的货币成为可能。因此，以太坊并不以取代前述的彩色币或元币协议为目的，而是旨在成为一个提供超强脚本系统的优秀底层协议，在其上可以创建任意高级的合约，货币及其它去中心化应用。如果现存的彩色币和元币项目迁移至以太坊平台，它们将从以太坊的简化支付确认，与金融衍生品和去中心化交易所的适应性，以及能够同时在一个网络中存在的能力中获益。使用以太坊，那些想出可能极大改变加密货币应用现状的点子的人将不再需要去启动他们自己的块链，他们可以简单地用以太坊脚本编码实现他们的想法，简而言之，以太坊是创新的基础平台。

哲学

以太坊的设计将遵循以下原则：

1. 简洁原则 - 以太坊协议将尽可能简单，即便以某些数据存储和时间上的低效为代价。一个普通的程序员也能够完美地去实现完整的开发说明。这将最终有助于降低任何特殊个人或精英团体可能对协议的影响并且推进以太坊作为对所有人开放的协议的应用前景。添加复杂性的优化将不会被接受，除非它们提供了非常根本性的益处。
2. 通用原则 – 没有“特性”是以太坊设计哲学中的一个根本性部分。取而代之的是，以太坊提供了一个内部的图灵完备的脚本语言以供用户来构建任何可以精确定义的智能合约或交易类型。想发明你自己的金融衍生品？用以太坊，你可以。想创造你自己的货币？把它做成一个以太坊合约就好。想建立一个全规模的守护程序（Daemon）或天网（Skynet）？你可能需要几千个联锁合约并且确定慷慨地喂养它们，一切皆有可能。
3. 模块化原则 – 以太坊的不同部分应被设计为尽可能模块化的和可分的。开发过程中，应该能够容易地让在协议某处做一个小改动的同时应用层却可以不加改动地继续正常运行。类似“短剑”（Dagger），“帕特里夏树”（Patricia trees） and “递归长度前缀编码“（RLP, recursive length prefix encoding,）等创新应该以独立的库的形式实施并且应该特性完整，以便于让其它的协议同样使用，即便以太坊不需要其中的某些特性。以太坊开发应该最大程度地做好这些事情以助益于整个加密货币生态系统，而不仅是自身。
4. 无歧视原则 – 协议不应主动地试图限制或阻碍特定的类目或用法，协议中的所有监管机制都应被设计为直接监管危害，不应试图反对特定的不受欢迎的应用。你甚至可以在以太坊之上运行一个无限循环脚本，只要你愿意为其支付按计算步骤计算的交易费用。

基础区块创建

在内核中，以太坊的起点是一个相当规则的使用内存困难的工作量证明机制挖矿的不附带多少额外复杂度的加密货币，以太坊在许多方面比我们今天使用的基于比特币的加密货币简单。由多个输入输出构成的交易概念被更直观的基于平衡账目的模型取代了。 序列号和锁定时间都取消了，并且所有的交易和区块数据都用单一格式编码。与比特币中对公钥加上 04 前缀后进行 SHA256 哈希再进行 RIPEMD160 哈希形成地址的方法不同，这里简单地取公钥的 SHA3 哈希的最后 20 字节作为地址。与其它致力于提供大量的“特性”的加密货币不同，以太坊致力于不提供特性，而是通过一个名为“合约”的涵盖所有的机制为用户提供近乎无限强大的功能。

修改实施幽灵协议

“幽灵“协议（"Greedy Heaviest Observed Subtree" (GHOST) protocol）是由 Yonatan Sompolinsky 和 Aviv Zohar 在 2013 年 12 月引入的创新。幽灵协议提出的动机是当前快速确认的块链因为区块的高作废率而受到低安全性困扰；因为区块需要花一定时间（设为 t）扩散至全网，如果矿工 A 挖出了一个区块然后矿工 B 碰巧在 A 的区块扩散至 B 之前挖出了另外一个区块，矿工 B 的区块就会作废并且没有对网络安全作出贡献。此外，这里还有中心化问题：如果 A 是一个拥有全网 30%算力的矿池而 B 拥有 10%的算力，A 将面临 70%的时间都在产生作废区块的风险而 B 在 90%的时间里都在产生作废区块。因此，如果作废率高，A 将简单地因为更高的算力份额而更有效率，综合这两个因素，区块产生速度快的块链很可能导致一个矿池拥有实际上能够控制挖矿过程的算力份额。

正如 Sompolinsky 和 Zohar 所描述的，通过在计算哪条链“最长”的时候把废区块也包含进来，幽灵协议解决了降低网络安全性的第一个问题；这就是说，不仅一个区块的父区块和更早的祖先块，祖先块的作废的后代区块（以太坊术语中称之为“叔区块”）也被加进来以计算哪一个区块拥有支持其的最大工作量证明。我们超越了 Sompolinsky 和 Zohar 所描述的协议以解决第二个问题 – 中心化倾向，以太坊付给以“叔区块”身份为新块确认作出贡献的废区块 87.5%的奖励，把它们纳入计算的“侄子区块”将获得奖励的 12.5%，不过，交易费用不奖励给叔区块。

以太坊仅采用了幽灵协议的最基础部分，即废区块只能以叔区块的身份被它的兄弟区块的子区块，而不是更远关系的后辈区块，纳入计算。这样做有几个原因。首先，无条件的幽灵协议将给计算给定区块的哪一个叔区块合法带来过多的复杂性。其次，带有以太坊所使用的补偿的无条件的幽灵协议剥夺了矿工在主链而不是一个公开攻击者的链上挖矿的激励。最后，计算表明基础部分的幽灵协议已经拥有了幽灵协议 80%以上的益处， 并且在区块生成时间 40 秒的情况下提供了和区块生成时间为 2.5 分钟的莱特币可相比较的废块率。不过，我们将保守一点，依旧选择和质数币相当的 60 秒区块时间因为单个区块需要更长的时间确认。

以太坊客户端 P2P 协议

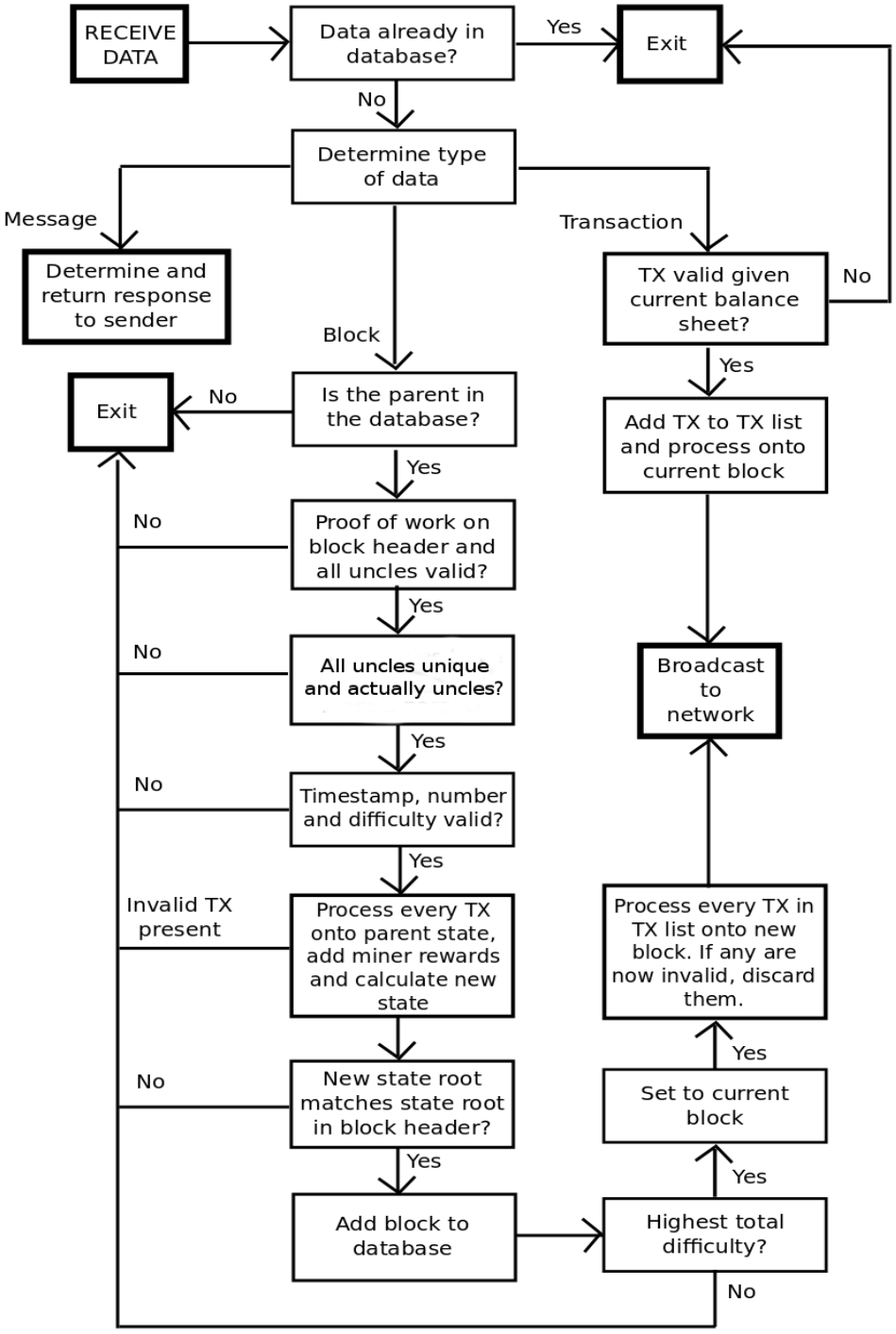
以太坊客户端 P2P 协议是一个相当标准的加密货币协议，并且能够容易地为其它加密货币使用；仅有的改动是引入了上述的“幽灵“协议。以太坊客户端基本上是被动的；如果没有被触发，它自己做的仅有工作是调用网络守护进程维护连接及定期发送消息索要以当前区块为父区块的区块。然而，该客户端同时会更强大；与只存储与块链相关的有限数据的 bitcoind 不同，以太坊客户端将同时扮演一个功能完整的区块浏览器的后台的角色。

当客户端收到一个消息时，它将执行以下布骤：

1. 哈希该数据，并且检查该数据与其哈希是否已经接收过，如果是，退出，否则将数据发送给数据分析器。
2. 确认数据类型。如果该数据项是一个交易，如果交易合法则将其加入本地交易列表，加入当前区块并发布至网络。如果该数据项是一个消息，作出回应。如果该数据项是一个区块，转入步骤 3。
3. 检查区块中的“父区块“参数是否已存储于数据库中。如果没有，退出。
4. 检查该区块头以及其“叔区块列表”中所有区块头中的工作量证明是否合法，如有任意一个非法，退出。
5. 检查“叔区块列表”中每一个区块的区块头以确定其是否以该区块的“祖父区块”为父区块。如有任何否，退出。注意叔区块头并不必须在数据库中；他们只需有共同的父区块并有合法的工作量证明。
6. 检查区块中的时间戳是否最后至未来 15 分钟并且在其父区块的时间戳之后。检查该区块的难度与区块号码匹配。如任何检查失败，退出。
7. 由该区块的父区块的状态开始，加上该区块中的每一笔合法交易。最后，加上矿工奖励。如果结果状态树的根哈希与区块头中的状态根不匹配，退出。如匹配，将该区块加入数据库并前进至下一步。

8. 为新区块确定 TD(block) ("总难度")。TD 由 TD(genesis_block) = 0 及 TD(B) = TD(B.parent) + sum(u.difficulty for u in B.uncles) + B.difficulty 递归定义。如新区块拥有比现区块更高的总难度，则新区块将成为“现区块“并进入下一步，否则，退出。
9. 如果新区块被改动，向其中加入交易列表中的所有交易，废除交易列表中的所有变为不合法的交易，将该区块及这些交易向全网重新广播。

P2P 协议



“现区块”是由矿工存储的一个指针；它指向矿工认为表达了最新的正式的网络状态的区块。所有索要平衡账目，合约状态等的消息都通过查询现区块并计算后回应。如果一个节点在挖矿，过程有一点轻微的改动；在做上述所有步骤的同时，该节点同时在现区块挖矿，将其自己收集的交易列表作为现节点的交易列表。

货币及其发行

以太坊网络包含其内建的货币，以太币，在网络内包含一种货币的原因是双重的。首先，以太币被奖励给矿工以促进网络安全。其次，用它来支付交易费用是一种反欺诈机制。类似 Hashcash 的以交易为单位的工作量证明和放任自由是收取交易费的两个替代方案，前者浪费资源并且对于低档计算机和智能手机是一种不公平的折磨，后者将会导致网络立刻被无限循环的“逻辑炸弹”合约淹没。以太币有一个理论上的最大量 - 2^{128} 单位 (比照比特币的 $2^{50.9}$ 单位)，虽然在可预见的将来不会有超过 2^{100} 被发行。为方便和避免将来的争论（参见现在关于 mBTC/uBTC/聪的争论），这里提前为一些数额设定单位：

- 1: 伟
- 10^3: (未定)
- 10^6: (未定)
- 10^9: (未定)
- 10^{12}: 萨博
- 10^{15}:芬尼
- 10^{18}: 以太

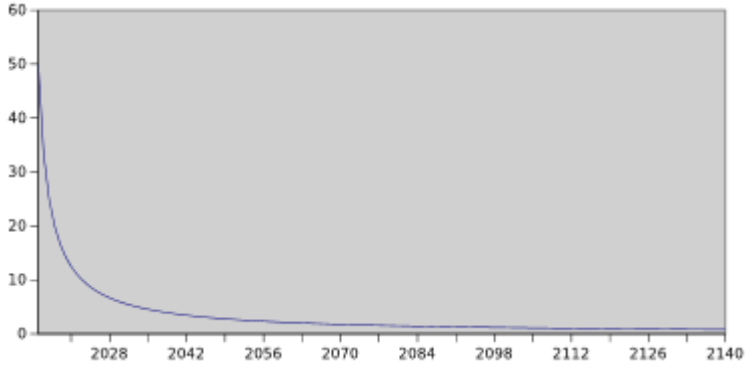
这将是“元”和“分”或者“比特币”和“聪”的概念的扩展版，旨在成为将来的证据；看起来只有萨博，芬尼和以太会在可预见的将来被使用。“以太”将成为系统的主单位，很像元或比特币。为 10^3, 10^6 和 10^9 命名的权力保留，未来将经过我们预批准后作为高级的辅助奖励授予投资者。

发行模型如下：

- 通过发售活动，以太币将以每 BTC 1000-2000 以太的价格发售给投资者。早期投资者会享受较低的价格以补偿不确定性风险，最低的投资额为 0.01BTC，假设此方式售出 X 以太。
- 0.225X 以太将分配给创业团队以及在发售活动启动之前实质性地参与了项目的人员，这些份额将附带时间锁定合同，40%在一年后才能花费，30%两年后，30%三年后。
- 0.05X 以太将分配给一个基金，用来在发售活动后以太币上线前这段时间支付费用和奖励。
- 0.225X 将分配给一个长期的预留资金池，以用来在以太币上线后以以太币的形式支付费用，工资和奖励。
- 每年都将有 0.4X 以太将被矿工挖出。

	1 年后	5 年后
货币单位	1.9X	3.5X
发售活动参与者（投资者）	52.6%	28.6%
创业团队和早期贡献者	11.8%	6.42%
附加以太币上线前分配	2.63%	1.42%
保留	11.8%	6.42%
矿工	21.1%	57.1%

长期通胀率（百分比）



正如比特币选择一种长期来看通胀率趋近于零的发行方式，我们也放弃线性增长的发行方式

举例来说，假设五年内没有发生交易，28.6%的以太币将会在投资者手里，6.42%属于创业团队和早期贡献者，6.42%属于预留池，还有 57.1%属于矿工。永久线性通胀模型降低了在比特币那里看到的财富过度集中的风险，并且给予了生活在现在和将来的个人公平的获取财富的机会，同时能够激励人们获取和持有以太币，因为从长远来看通胀率依然是趋近于零的（例如在第 1000001 年货币供应会从 500001.5 * X 上升到 500002 * X，通胀率 0.0001%）。此外，很多对于以太坊的兴趣是中期的。我们预测如果以太坊成功则它在 1-10 年的时间段内会经历巨大的增长，而在这期间它的供应量是非常有限的。

我们还推论因为从长期来看货币经常因为粗心，死亡等原因而导致遗失，以及遗失的货币量可以经建模成为年货币供应总量的一个百分比，流通中的货币总量事实上将最终稳定在一个等于年货币发行量/遗失率的值上（例如如果遗失率为 1%，一旦货币供应量达到 100 * (0.5X)，则每年有 0.5X 被挖出并且有 0.5X 被丢失，达到均衡）。

数据格式

以太坊中的所有数据都以“递归长度前缀编码（recursive length prefix encoding,RLP）”形式存储，这种编码格式将任意长度和维度的字符串构成的数组串接成字符串。例如，['dog', 'cat']被串接（以字节数组格式）为［130, 67, 100, 111, 103, 67, 99, 97, 116]; 其基本的思想是把数据类型和长度编码成一个单独的字节放在实际数据的前面（例如‘dog’的字节数组编码为[100, 111, 103]，于是串接后就成了[67, 100, 111, 103].）注意 RLP 编码正如其名字表示的一样，是递归的；当 RLP 编码一个数组时，实际上是在对每一个元素的 RLP 编码级联成的字符串编码。需要进一步提请注意的是，以太坊中所有数据都是整数；所以，如果有任何的以一个或多个 0 字节开头的哈希或者地址，这些 0 字节应该在计算出现问题的时候去除。以太坊中没有串接数据结构包含任何以 0 开头的数值。整数以大端基础 256 格式存储（例如 32767 字节数组格式为[127, 255]）。

一个完整的区块的结构是：

```
[
    block_header,
    transaction_list,
    uncle_list
]
```

Where:

```
transaction_list = [
    transaction 1,
    transaction 2,
    ...
]

uncle list = [
    uncle_block_header_1,
    uncle_block_header_2,
    ...
]

block_header = [
    parent hash,
    sha3(rlp_encode(uncle_list)),
    coinbase address,
    state_root,
    sha3(rlp_encode(transaction_list)),
    difficulty,
    timestamp,
```

```
    extra_data,
    nonce
]
```

每个 transaction 和 uncle_block_header 都是一张表。工作量证明数据是区块数据去除掉 nonce（交易数）后的 RLP 编码。

uncle_list 和 transaction_list 分别是叔区块头和区块里的交易构成的表。nonce 和 extra_data 都被限制为最大 32 字节，除了在创世区块中参数 extra_data 会更大。

state_root 是一个包含所有地址的（key, value）对的默克尔-帕特里夏树（Merkle Patricia tree ）的根， 其中每一个地址都由一个 20 字节二进制字符串表示。对于每个地址，储存在默克尔-帕特里夏树的 value 字段是一个对以下格式对象进行 RLP 串接编码形成的字符串：

[balance, nonce, contract_root, storage_deposit]

nonce 是该地址的交易数，每做一次交易都会增加 1。其目的是(1)使每个交易只有一次合法的机会以防范重放攻击，(2)使得构建一个和已存合约有相同哈希的合约成为不可能（更准确地说，密码学意义上不可行）。balance 指的是合约或地址的平衡账目，以伟为单位。contract_root 是另一个帕特里夏树的根，在该地址被一个合约控制的情况下包含该合约的内存。如果一个地址没有被一个合约控制，contract_root 就会是一个空字符串。storage_deposit 是一个计数器，储存花费的储存费用；它的功能将在后面详细讨论。

挖矿算法

挖矿算法的一个令人高度向往的特性是对专业化硬件优势的抵抗力。最初，通过允许每一个人用 CPU 参与挖矿过程，比特币被设想为高度民主的货币。然而在 2010 年，大量矿工利用显卡的快速并行化很快取得了领先，全网算力增长了 100 倍，CPU 已经可以忽略不计。2013 年，通过制造以计算 SHA256 哈希为唯一目的的芯片，更专门化的硬件 ASIC 获取了新一轮 100 倍的算力增长，把显卡远远甩在了后面。今天，没有从专业矿机公司购买的专业矿机，挖矿实际上是不可能的，一些人担心 5-10 年内挖矿将完全被英特尔或 AMD 这样的大型中心化公司控制。

目前为止，达到这一目标的主要途径是“内存困难”- 构造一个不仅需要高算力，而且需要大量内存才能有效工作的工作量证明算法。内存困难工作量证明算法已经有了几个应用案例，每一个都存在缺陷：

- **Scrypt** - Scrypt 是一个设计占用 128KB 内存来计算的函数。该算法本质上是用哈希数填充内存阵列，然后基于内存阵列中的值计算出中间值和最终结果的方式工作的。内存困难特性意味着依赖于大量并行的专门化硬件将需要大量的内存，这降低了这类硬件的效率。然而 128KB 的参数是一个很低的门槛，用于莱特币挖矿的 ASIC 已经在研发中了。此外，因为验证过程与一轮挖矿过程占用同样多的内存和算力，Script 算法难以通过调整达到更高的内存困难特性。
- **生日攻击**- 生日工作量证明背后的思想很简单： 找到值 xn,i,j 使得 i < k, j < k 且 |H(data+xn+i) - H(data+xn+j)| < 2^256 / d^2. 这里 d 为声称区块的难度指数，k 为内存困难指数。任何生日算法都必须把所有的 H(data+xn+i)的计算结果写入内存以便让接下去的计算结果与之比较。这种思想有一个主要的好处: 计算是内存困难的，但验证时内存容易的，这允许极度的内存困难而无需为内存容易的验证做折衷。然而，有两个原因给这种算法带来了问题。首先，在时间-内存置换攻击中用户可以以四倍的时间加上四分之一的内存来解决问题，这使得内存困难不是绝对的。其次，为此问题开发专门芯片是很容易的，尤其是当从传统芯片和处理器架构前行到多种基于硬件的哈希表或者基于概率的模拟计算的时候。
- **短剑（Dagger）** - 短剑- 一个由以太坊团队开发的内部算法 – 背后的思想是拥有一种类似 Scrypt 的算法，它独特的特性是每个单独的随机数只和每个约 1 千万个随机数组成的群构成的树的一小部分有关。有效地计算随机数需要构建整个树（而且该算法是特别为避免小规模的时间-内存置换设计的），占用 100MB 的内存，二验证一个随机数只需约 100KB 的内存。然而，短剑型的算法易受分享内存的多计算电路攻击，虽然这种威胁能够减轻，却可以说是不可能被完全消除的。
- 另一个工作量证明机制（PoW， proof of work）的替代方案，明确地对专用硬件有抵抗力的，是权益证明机制（PoS, proof of stake），在这个方案中，旷工用他们的私钥，而不是算力投票，而且每一个私钥产生合法区块的机会与其持有的货币量成正比。然而，权益证明机制有一个主要的缺点：当有一个尝试中的分叉或攻击的时候，矿工们有能力，而且没有动机不去，在所有竞争块链上同时挖矿。我们曾经开发过一个名为“剑手”（Slasher）的权益证明算法，它是第一个用专门检测和惩罚

双重挖矿行为的方法解决了这个问题的。投票权将提前由区块号决定，所以在块链分叉的时候一个矿工要么能够投给其中任意一条链要么完全失去投票机会。

缺省地，我们目前考虑经过参数调整的类短剑算法来最小化专用硬件攻击，如果确定需要，会和“剑手”算法一起采用以获取更高的安全性。然而，为了能得到一个比现存的都好的 PoW 算法，我们计划使用发售活动筹集的资金来举办一个竞赛，如同 2005 年确定高级加密标准（Advanced Encryption Standard ，AES）和 2013 年确定 SHA3 哈希算法的竞赛那样，全球的研究团队可以竞争开发对 ASIC 有抵抗力的挖矿算法并且由多轮的评选确定最终的优胜者。竞赛有奖金，是开放的，我们鼓励对于内存困难的 PoW 算法，可自修改的 PoW 算法，基于 x86 指令的 PoW 算法，带有与人工驱动动机兼容的经济协议以便在将来轮替的多 PoW 方案，以及能达到目的的任何其它设计。其它替代方案如 PoS，PoB(Proof of Burn)，PoE(Proof of Excellence)都一样有机会。

交易

一笔交易的数据结构是：

[nonce, receiving_address, value, [data item 0, data item 1 ... data item n], v, r, s]

nonce 是该地址已经发送的交易数量，编码为二进制格式(例如 0 -> '', 7 -> '\x07', 1000 -> '\x03\xd8'). (v,r,s)是新生成的不含用发送地址对应的私钥签名的 Electrum 风格的交易签名，v 的范围是 27 <= v <= 30. 从一个 Electrum 风格的签名(65 字节)可以直接提取出公钥和地址。交易合法的条件： (i)签名具有合法格式 (即 27 <= v <= 30, 0 <= r < P, 0 <= s < N), 以及 (ii) 发送地址具有足够的资金支付交易金额和交易费用。 一个区块不能够包含一个非法的交易；如果一个合约产生了一个非法交易则该交易将直接无效。交易费用将被自动包含。如果一个用户自愿支付更高费用，他总可以通过构建一个合约来自动地发送一定数量或一定比例的金额给现区块的矿工同时加速交易确认。

发送给空地址的交易是一种特殊类型的交易，创建了一个“合约”。

难度调整

难度根据下面的公式调整：

D(genesis_block) = 2^36
D(block) =
    if anc(block,1).timestamp >= anc(block,501).timestamp + 60 * 500: D(block.parent) - floor(D(block.parent) / 1000)
    else: D(block.parent) + floor(D(block.parent) / 1000)

anc(block,n) 是该区块（block）的第 n 代祖先；所有创世区块之前的区块都被假定为拥有和创世区块一样的时间戳。这会自动地把产生一个区块的时间稳定在 60 秒左右。选择 500 是考虑到过小的值会导致拥有足够算力的经常挖出两个连续区块的矿工修改时间戳以获取最大收益，而过大的值会导致难度震荡得过于剧烈，当选取常数 500，模拟显示全网算力固定的情况下难度变化幅度是+/-20%。

区块奖励

一个矿工接受三种奖励：静态的产生区块的奖励，交易费用，以及上面幽灵协议部分描述过的叔区块/侄区块奖励。区块奖励将 100%给予矿工自己，但交易费用将被分割，其中的 50%给予矿工，剩下的 50%将在最近的 64 个矿工中平分。这样做是为了防止矿工用制造大量交易的方法创造区块并把交易费用支付给自己，同时依然激励矿工去包含尽可能多的交易。如上面幽灵协议部分描述，叔区块只接受他们的区块奖励的 87.5%，剩余的 12.5%流向把它们包含进来的侄区块，来自作废区块的交易费用不向任何人流动。

合约

在以太坊里，有两种实体可以发起和接收交易：真实的人（或者软件机器人，因为密码协议是不能区分这两者的）与合约。合约事实上可以看成是活在以太坊网络上的自动化代理人，它有以太坊的地址以及账户金额，可以发送和接收交易。每当有人向合约发送交易后，它就被激活了，然后就开始运行它自己的程序，例如改变它自己的内部状态或者甚至放送一些交易，完成后它又休息了。合约自身的程序由特殊的低级语言写成，包括用来暂时存储的堆栈、用来暂时存储的 2^{256} 个内存输入项、以及用来存储合约永久状态的 2^{256} 个存储输入项。注意，以太坊使用者并不需要使用这些低级堆栈语言来编程，我们会提供更为简单的类 C 语言，这里面会包括变量、表达式、条件判断、数组以及 while 循环，我们还会提供编译器，以太坊的脚本代码也可以用它来编译。

应用

这里列举了一些例子，用来展示以太坊合约可以实现些什么，这些代码例子用我们将要发布的类 C 高级语言写成。变量 tx.sender、tx.value、tx.fee、tx.data 和 tx.datan 都是输入交易的属性，变量 contract.storage 和 contract.address 是合约本身的属性，而变量 block.contract_storage、block.account_balance、block.number、block.difficulty、block.parenthash、block.basefee 和 block.timestamp 则是区块的属性。变量 block.basefee 是基准费用，以太坊里面的所有交易费用都是基准费用的整数倍，接下去“费用”章节会详细解释。以大写字母开头（例如 A）的变量都是常量，这些常量的值由合约的创建者在发布合约时确定。

子货币

子货币有很多的应用，例如他们可以用来代表美元或黄金、公司的股票，甚至可以有总数只有一个单位的子货币，它们可以用来代表抵押物或智能资产。一些基于以太坊的具有特殊目的的高级金融协议，可能也会想要拥有内部货币来作为组织形式。 这些子货币在以太坊里实现起来比你想象中要容易，接下去我们会用一个很简单的合约来举例说明。

例如，某人想发送 X 货币单位到 C 货币合约的 A 地址，这个交易的形式会是(C, 100 * block.basefee, [A, X])，之后合约将解析这个交易从而相应地调整各账户金额。为了让这个交易合法，交易的费用必须是基准费用的 100 倍，这样才能“喂饱” 合约（因为对于任何一个合约，从 16 步之后的每一个计算步都会都会花掉微量费用，如果花完了，合约就会罢工不干）。

```
if tx.value < 100 * block.basefee:
    stop
elif contract.storage[1000]:
    from = tx.sender
    to = tx.data[0]
    value = tx.data[1]
    if to <= 1000:
        stop
    if contract.storage[from] < value:
        stop
    contract.storage[from] = contract.storage[from] - value
    contract.storage[to] = contract.storage[to] + value
else:
    contract.storage[mycreator] = 10000000000000000
    contract.storage[1000] = 1
```

Ethereum 的子货币创建人可能还希望增加更多的高级特性：

- 包含一种机制让人们可以在交易所用以太币买入子货币，例如每天拍卖一定数量的子货币。
- 允许交易费以内部货币支付，之后再用以太币返交易费给发送者。这会解决所有子货币协议迄今为止都面临的一个主要问题：子货币的用户花的是子货币，而用来付交易费的却是主货币。这里，一个新账户需要用以太币来激活一次，之后就再也不需要重新充值了。
- 允许建立一个不依赖信任的去中心化交易所来实现子货币与以太币之间的兑换。注意，虽然理论上在以太坊里，任意两个合约之间的去中心化交易所即使没有特殊支持也是可能实现的，但是，有特殊支持就会让它实现的成本便宜 10 倍。

金融衍生品

数据输入是金融衍生品最关键的要素，它提供一种特定的资产以另外一种资产计价的价格（在以太坊里，这另一种资产就是以太币）。有很多方法可以来实现数据输入，其中一种方法已经被万事达币（Mastercoin）的开发者实践过了，那就是把数据输入放进区块链里。实现代码如下：

```
if tx.value < block.basefee:
    stop
if tx.sender != contract.creator:
    stop
contract.storage[data[0]] = data[1]
```

任何其它合约都可以访问数据库 D 的指数 I，只需用 block.contract_storage(D)[I] 即可。另一种更高级地实现数据输入的办法可能是放到链下—数据输入的的提供者对所有数据签名，这就要求任何想要触发合约的人使用最新的签过名的数据，再用以太坊内置的脚本功能来验证签名。几乎任何衍生品都可以通过这个实现，包括杠杆交易、期权，甚至包括更进阶的产品，例如担保债务凭证（CDO）。（小心，这里可没有政府救援计划，所以你要当心黑天鹅事件。）

我们拿对冲合约来举个例子。大概的意思是，A 存进 4000 个以太币来创建合约，任何人都可以接受这个合约，他只要往里存进 1000 个以太币。假如合约创建时，数据库 D 的指数 I 显示 1000 以太币值 25 美元，如果 B 接受这个合约，那么 30 天后，合约将发送价值 25 美元的以太币给 B，再把剩下的发给 A，任何人都可以通过发送一个交易给合约让它执行。这样 B 就免除了以太币价格波动的风险，且不需依靠任何发行方。B 的唯一风险就只剩下以太币价格在 30 天内下跌 80%以上，但甚至连这个风险都是可以通过建立另一个对冲合约来规避，当然这需要 B 在线 。 而 A 拿到的好处是那隐含的 0.2%的合约费用，至于风险， A 可以通过另外持有等额的美元来对冲（或者 A 看好以太坊的未来，他想通过这个合约用 1.25 倍的杠杆看涨以太币，如果是这样的话，那 A 可能甚至愿意支付合约费给 B ）。

```
state = contract.storage[1000]
if state == 0:
    if tx.value < 1000 * 10^18:
        stop
    contract.storage[1001] = 998 * block.contract_storage(D)[I]
    contract.storage[1002] = block.timestamp + 30 * 86400
    contract.storage[1003] = tx.sender
else:
    if tx.value < 200 * block.basefee:
        stop
    ethervalue = contract.storage[1000] / block.contract_storage(D)[I]
    if ethervalue >= 5000 * 10^18:
        mktx(contract.storage[1003],5000 * 10^18,0,0)
    else if block.timestamp > contract.storage[1002]:
        mktx(contract.storage[1003],ethervalue,0,0)
        mktx(A,5000 - ethervalue,0,0)
```

更复杂的金融合约也是可能的。包含多重条款的复杂期权合约可以通过存储状态变量来定义，就像之前那个合约一样，不过就是多了些条款的代码，每一个条款代表一个可能的状态。例如，一个人可以在 12 月 1 日之前用 2 美元买入一个合约，这个合约让他有权在 12 月 4 日选择，是在 12 月 29 日收到 1.95 美元呢，还是把这选择权推迟到 12 月 11 日，到时候再选择，到底是在 12 月 28 日收到 2.20 欧元呢，还是再次把选择权推迟到 12 月 18 日，那个时候再选择，是在 12 月 30 日收到 1.20 英镑呢，还是在 12 月 29 日收到 3.20 欧元并支付 1 欧元？ 注意，任何形式的金融合约都需要全额担保，因为以太坊网络不控制任何国家机器，也不能帮忙追债。

身份和信誉体系

域名币（Namecoin）在所有替代币中是最早的，它试图通过使用类似比特币的区块里来提供名称注册体系，用户可以把他们的名称与其它信息注册进这个公共数据库。域名币最大的用途被认为可以用来做 DNS 系统，把诸如“bitcoin.org”的域名（在域名币体系里可能会是 bitcoin.bit）映射成 IP 地址。别的用途包括电子邮件认证和潜在的进阶信誉体系。下面的这个例子将展示在以太坊里用一个简单的合约来实现类似域名币的名称注册：

```
if tx.value < 25 * 10^18:
    stop
if contract.storage[tx.data[0]]:
    stop
contract.storage[tx.data[0]] = contract.storage[tx.data[1]]
```

你可以很容易地加入更复杂的功能，来让用户改变映射、自动发送交易到合约并推送，甚至增加信誉体系和信任网络体系。

去中心化自治组织（DAO）

去中心化自治机构（decentralized autonomous corporation）是指由一定数量股东组成的实体，这些投资者可以获得分红，到达一定的大多数（例如 67%）就有权修改它本身的代码。这些股东共同决定该怎么花公司的资金，是以悬赏的形式，还是工资的形式？甚至别的更新奇的方式，如用内部货币奖励贡献，这些资金还是自动分配的，等等。这实际上是复制了传统公司的法律陷阱，不同的是它是以基于密码学的区块链来强制执行的。这是去中心化组织的公司模型，（去中心化自治组织的）另一种形式可被称为“去中心化自治社区”，其成员将在决策时有相等的投票权，添加或者移出成员需要 67%的成员统一。一个成员的加入强制地需要群体共同决定。

一个 DAO 的“代码骨架”可能看上去跟下面的差不多。

有三种类型的交易：

- [0,k] 用以注册投票来修改代码
- [1,k,L,v0,v1...vn] 用以注册 k 处的代码修改，来设置内存使其从 L 处开始至 v0, v1 ... vn 处。
- [2,k] 用以最终确定代码修改。

注意，这个设计依靠地址和哈稀的随机性来保障数据的完好，在大概 2^128 次使用后合约很可能出现某种方式的恶化失效，但这问题不大，因为在可遇见的将来这个数量级是达不到的。2^255 这个数被用来存储（组织的）成员总数，而一个成员的资格以一个“1”的方式存在成员的地址里。合约的最后三行代码用来把 C 加为第一个成员，从这之后就全是 C 的任务了，包括运用基于民主投票的代码修改协议来添加其他成员，以及编写代码来发展壮大这个组织。

```
k = sha3(32,tx.data[1])
if tx.data[0] == 0:
    if contract.storage[tx.sender] == 0:
        stop
    if contract.storage[k + tx.sender] == 0:
        contract.storage[k + tx.sender] = 1
        contract.storage[k] += 1
else if tx.data[0] == 1:
    if tx.value <= tx.datan * block.basefee * 200:
        stop
    if contract.storage[k] > 0:
        stop
    i = 3
    while i < tx.datan:
        contract.storage[k + i] = tx.data[i]
```

```
            i = i + 1
        contract.storage[k] = 1
        contract.storage[k+1] = tx.datan
        contract.storage[k+2] = tx.data[2]
    else if tx.data[0] == 2:
        if contract.storage[k] >= contract.storage[2 ^ 255] * 2 / 3:
            if tx.value <= tx.datan * block.basefee * 200:
                stop
            i = 3
            L = contract.storage[k+1]
            loc = contract.storage[k+2]
            while i < L:
                contract.storage[loc+i-3] = tx.data[i]
                i = i + 1
    if contract.storage[2 ^ 255 + 1] == 0:
        contract.storage[2 ^ 255 + 1] = 1
        contract.storage[C] = 1
```

这个方案采用了“平等主义”的 DAO 模型，你完全可以轻易地把它扩展到股东模型，只需要算清每个股东持了多少股，并提供让他们相互转让股份的渠道。

DAO 和 DAC 将成为未来经济的组织形式 ,这已经是密码货币界的热门话题了。我们也对 DAO 的潜能非常感兴趣 ,并且最终希望把 以太坊组织本身也转变成完全自主的 DAO。

未来应用

1）储蓄钱包。假设 Alice 希望保障她的资金安全，但是担心自己货弄丢私钥或私钥被黑客盗走，那她可以把以太币打进与 Bob（是一家银行）的合约里，规定如下：Alice 每天可以单独取款最多 1%的资金，Alice 和 Bob 一起可以取走全部，而 Bob 单独最多只能取 0.05%。通常一天 1%对于 Alice 是足够的，如果她想取更多可以找 Bob 帮忙；如果 Alice 的私钥被盗了，她可以赶紧跑去找 Bob 一起把资金转移到一个新的合约里；如果她丢了私钥，Bob 最终还是把资金（缓慢地）取出来的；如果最后发现 Bob 是恶人，那 Alice 可以以 20 倍 Bob 的速度把钱转走。

2）农作物保险。你可以很容易地建立金融衍生品合约，这里用的是天气的数据输入，而不是价格指数。如果一个衍生品合约支付出来的金额与爱荷华州的降雨量负相关，那么一个爱荷华州的农民如果买它，就可以在干旱的时候收到补偿；而当降雨充沛的时候，他会很开心因为他的农作物会长得很好。

3）一种以中心化方式管理的数据输入方式，它采用基于权益证明机制投票的最终结果的平均值（更可能是中位值）来代表人们对于某个数据的看法，这个数据可以是某种大宗商品的价格，或者是天气，或是其它相关数据。

4）多重签名智能契约。比特币允许基于多重签名的交易合约，例如，5 把私钥里集齐 3 把就可以使用资金。以太坊可以做得更细化，例如，5 把私钥里集齐 4 把可以花全部资金，如果只有 3 把则每天最多花 10%的资金，只有 2 把就只能每天花 0.5%的资金。另外，以太坊里的多重签名是异步的，意思是说，双方可以在不同时间在区块链上注册签名，最后一个签名到位后就会自动发送交易。

5）点对点赌博。任意数量的点对点赌博协议都可以搬到以太坊的区块链上，例如 Frank Stajano 和 Richard Clayton 的 Cyberdice。最简单的赌博协议事实上是这样一个简单的合约，它用来赌下一个区块的哈稀值与猜测值之间的差额。之后，SatoshiDice 整个赌场都可以搬到区块链上去，这可以通过给每一次赌博创建一个合约来实现，也可以通过半中心化的合约来实现。

6）顺理成章，基于区块链的大规模股票市场，预测市场也很容易实施。

7）利用身份和信用体系，来实现一个基于区块链的去中心化市场。

8)去中心化的 Dropbox。先把文件加密 ,再建立它的 Merkle 树 ,然后把 Merkle 树的根和一定数量的以太币一起根植到一个合约里，最后把文件散布到在某个次级网络上去。这个合约每天都会根据区块的哈稀值来随机选出 Merkle 树的一个分支，然后拿出一些以太币给第一个为合约提供这个分支的节点，这个奖励就鼓励了节点们来长期存储数据。如果你想下载文件的任何一部分，你可以发送支付到某个提供微支付通道的合约，进而从多个节点下载文件，每个区块下载一部分。

合约如何工作？

用合约来实现交易的代码是这样的：

```
[
    nonce,
    '',
    value,
    [
        data item 0,
        data item 1,
        ...
    ],
    v,
    r,
    s
]
```

在大多数情况下，数据项会是脚本代码（后面有更多解释）。来用创建合约的交易是这么验证的：

1. 把交易反串行化，然后从它的签名里提取发送地址。

2. 计算出交易费，确保创建者的账户金额不少于“捐助额+交易费”，如果不是，则退出；如果是，支付交易费。

3. 对于创建合约的交易， 从它 RLP 编码的 sha3 哈稀值中至少提取最后 20 位，如果与这个地址关联的账户已经存在了，则退出；如果不是，则为这个地址创建合约。

4. 为[0 ... n-1]范围的每一个 I, 拷贝数据项 i 到合约的数据槽 i，n 是交易里数据项的总数。

代码语言详述

合约的脚本语言是汇编语言和比特币的基于堆栈的语言的混合体，它里面总是保有一个指数指针，通常每完成一步执行指针就往前走一位，如果指针保持原位则操作会被连续执行。所有的操作符都是[0...63]之间的数，除非这里有特殊说明的操作符，如 STOP、EXTRO 和 BALANCE 这些，它们后面还要定义数值。这种脚本语言可以访问三类存储：

- 堆栈—一种暂时存储，每当合约执行完成，它的列表就会被清零。对于堆栈通常的操作是在其顶部加进或移除数值，所以在程序执行过程中它的长度会增大或缩小。
- 内存—密钥或数值的暂时存储，每当合约执行完成，它就重置为“0”。密钥和数值以[0...2^256-1]之间的整数形式存储。
- 存储—密钥或数值的永久存储，它的初始值是零，除非是合约刚建立的时候插入的一些脚本代码。当它在帕特里夏树里编码时，所有的密钥和数值都以大字节序编码存储，而一个含“0”的密钥则表示这把密钥不在帕特里夏树里。密钥和数值以[0...2^256-1]之间的整数形式存储。

每当一笔交易被发送到某个合约，合约就执行它自己的脚本代码，具体的步骤是这样：

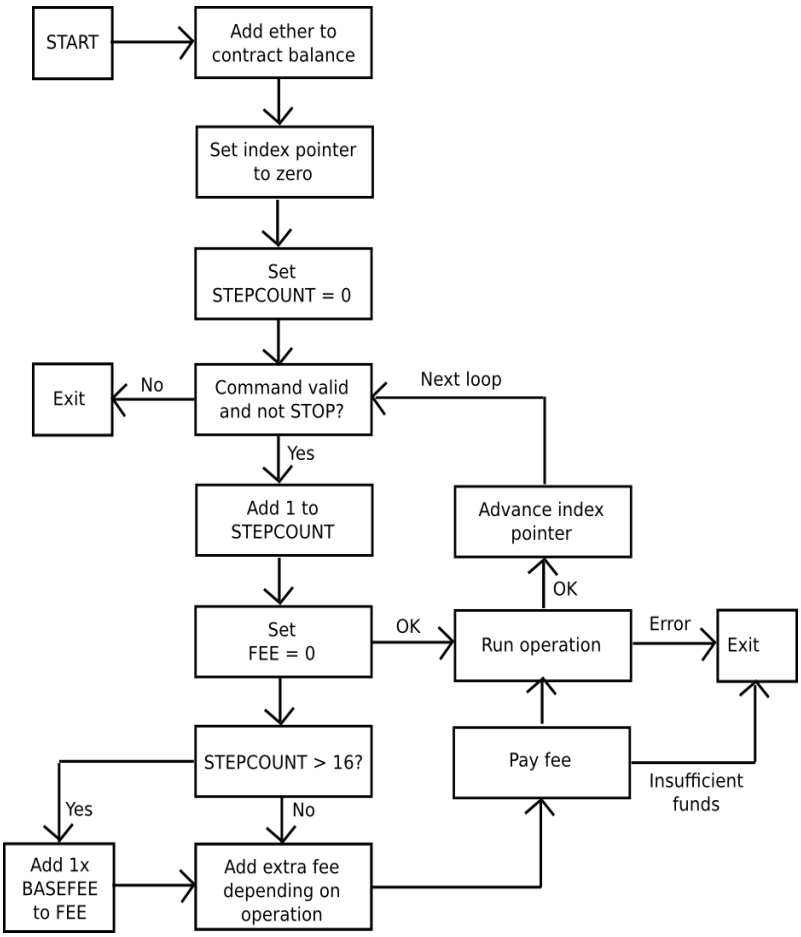
1. 增加发送的以太币数额到合约的账户金额

2. 指数指针设为零，还有 STEPCOUNT = 0

3. 永久重复：

    - 如果在指数指针处的指令是 STOP、非法、或者大于 63，跳出循环。
    - 令 MINERFEE = 0, VOIDFEE = 0
    - 令 STEPCOUNT <- STEPCOUNT + 1
    - 如果 STEPCOUNT > 16, 令 MINERFEE <- MINERFEE + STEPFEE
    - 查看指令是不是 LOAD 或 STORE，如果是，令 MINERFEE <- MINERFEE + DATAFEE
    - 查看指令会不会装满空的内存域，如果会，令 VOIDFEE <- VOIDFEE + MEMORYFEE
    - 查看指令会不会清零已用的内存域，如果会，令 VOIDFEE <- VOIDFEE - MEMORYFEE
    - 查看指令是不是 EXTRO 或 BALANCE，如果是，令 MINERFEE <- MINERFEE + EXTROFEE
    - 查看指令是不是密码运算，如果是，令 MINERFEE <- MINERFEE + CRYPTOFEE
    - 如果 MINERFEE + VOIDFEE > CONTRACT.BALANCE, 暂停并跳出循环。
    - 否则，从合约的账户金额里减去 MINERFEE + VOIDFEE，并把 MINERFEE 加到一个运行中的计数器，计数器的金额会在所有交易都分列完成后加到矿工的账户金额上。注意，在有的情况下 MINERFEE 可能是负数，那么合约的账户金额是增加的。
    - 执行命令
    - 如果命令没有因出错而跳出，那就更新指数指针并回到循环的起点。如果命令因出错而跳出，终止循环。注意，如果合约因出错而跳出，这并不会使交易或区块非法，而只表示合约没执行完就停了。

合约脚本阐释



在以下描述中，S[-1]、 S[-2]等等表示堆栈最顶端、第二顶端的项。各个操作符定义如下：

- (0) STOP – 停止执行
- (1) ADD - 弹出两项并压入 S[-2] + S[-1] mod 2^256 项
- (2) MUL - 弹出两项并压入 S[-2] * S[-1] mod 2^256 项
- (3) SUB - 弹出两项并压入 S[-2] - S[-1] mod 2^256 项
- (4) DIV - 弹出两项并压入 floor(S[-2] / S[-1])项，如果 S[-1] = 0, 停止执行。
- (5) SDIV - 弹出两项并压入 floor(S[-2] / S[-1])项，但把超出 2^255 的部分看作负数（也就是 x -> 2^256 - x)。如果 S[-1] = 0，停止执行。
- (6) MOD - 弹出两项并压入 S[-2] mod S[-1]项。如果 S[-1] = 0，停止执行。
- (7) SMOD - 弹出两项并压入 S[-2] mod S[-1]项, 但把超出 2^255 的部分看作负数（也就是 x -> 2^256 - x)。如果 S[-1] = 0，停止执行。
- (8) EXP - 弹出两项并压入 S[-2] ^ S[-1] mod 2^256 项
- (9) NEG - 弹出两项并压入 2^256 - S[-1]项
- (10) LT - 弹出两项并压入 1，如果 S[-2] < S[-1]；否则压入 0
- (11) LE - 弹出两项并压入 1，如果 S[-2] <= S[-1]；否则压入 0
- (12) GT - 弹出两项并压入 1，如果 S[-2] > S[-1]；否则压入 0
- (13) GE - 弹出两项并压入 1，如果 S[-2] >= S[-1]；否则压入 0
- (14) EQ - 弹出两项并压入 1，如果 S[-2] == S[-1]；否则压入 0
- (15) NOT - 弹出两项并压入 1，如果 S[-1] = 0；否则压入 0
- (16) MYADDRESS - 把合约地址当作数字压入
- (17) TXSENDER - 把发送者的地址当作数字压入
- (18) TXVALUE - 压入交易金额
- (19) TXDATAN - 压入数据项的总数
- (20) TXDATA - 弹出一项并压入数据项 S[-1]；如果指数溢出范围，则压入 0
- (21) BLK_PREVHASH - 压入前一个区块的哈稀值（不是当前区块，因为这没有可能）
- (22) BLK_COINBASE - 压入当前区块的币基（coinbase）
- (23) BLK_TIMESTAMP - 压入当前区块的时间戳
- (24) BLK_NUMBER - 压入当前区块的号数
- (25) BLK_DIFFICULTY - 压入当前区块的难度值
- (26) BASEFEE - 压入基准费用（乘以，以下费用章节会作定义）
- (32) SHA256 - 弹出两项，然后把内存里处于从指数 S[-2] 到(S[-2] + ceil(S[-1] / 32) - 1) mod 2^256 的所有 ceil(S[-1] / 32)项都提取出来，用它们来生成一个字符串，如有必要，填充“0”字节使得他们的长度是 32 字节，再提取最后的 S[-1] 字节。压入字符串的 SHA256 哈稀值。
- (33) RIPEMD160 - 跟 SHA256 一样执行，只不过用的 RIPEMD-160 哈稀。
- (34) ECMUL - 弹出三项。如果(S[-2],S[-1]) 是在 secp256k1 曲线上合法的点，包括两个座标都要小于 P，则用(0,0)作为无穷处的点，压入(S[-2],S[-1]) * S[-3]项进堆栈；否则压入(2^256 - 1, 2^256 - 1)。注意这里对 S[-3]并无限制。
- (35) ECADD - 弹出四项，压入(S[-4],S[-3]) + (S[-2],S[-1])，如果这两个点都合法；否则，压入(2^256 - 1,2^256 - 1)。
- (36) ECSIGN - 弹出两项，压入(v,r,s) 。这里，(v,r,s) 是哈稀 S[-1]的 Eletrum 类型的 RFC6979 决定性签名，此时私钥是 S[-2] mod N 。
- (37) ECRECOVER - 弹出四项，把(x,y)作为公钥压入，这个公钥来自于哈稀 S[-4]的签名(S[-3],S[-2],S[-1]) 。如果签名里有 v,r,s 的值非法（也就是说，v 不在[27,28]里，r 不在[0,P]里，s 不在[0,N]里），则返回(2^256 - 1,2^256 - 1)。
- (38) ECVALID - 弹出两项，如果(S[-2],S[-1]) 是 secp256k1 曲线上合法的点（包括(0,0)），则压入 1；否则，压入 0。
- (39) SHA3 - 跟 SHA256 一样执行，只不过是 SHA3 哈稀，256 位版本。
- (48) PUSH - 在指数指针+1 处压入项，并让指数指针前进 2。
- (49) POP - 弹出一项。
- (50) DUP - 压入 S[-1]进堆栈。
- (51) SWAP - 弹出两项，先压入 S[-1]，再压入 S[-2]。
- (52) MLOAD - 弹出一项，再把它压入至内存里的指针 S[-1]处。
- (53) MSTORE - 弹出两项，把内存里的指数 S[-1]设为 S[-2]。
- (52) SLOAD -弹出一项，再把它压入至存储里的指针 S[-1]处。

- (53) SSTORE -弹出两项，把存储里的指数 S[-1]设为 S[-2]。
- (54) JMP - 弹出一项，再把指数指针设为 S[-1]。
- (55) JMPI - 弹出两项，只有当 S[-1] 不为零时，把指数指针设为 S[-2]。
- (56) IND - 压入指数指针。
- (57) EXTRO - 弹出两项，压入合约 S[-1]的内存指数 S[-2]。
- (58) BALANCE - 弹出一项，压入地址的账户金额；如果不是地址，则压入 0。
- (59) MKTX - 弹出四项，发起一个交易发送 S[-2] 个以太币到 S[-1]，同时带 S[-3]个数据项。把在内存里从指数 S[-4]到指数(S[-4] + S[-3] - 1) mod 2^256 的项作为交易的数据项。
- (63) SUICIDE - 弹出一项，销毁合约并清空所有存储，从正在清空的内存发送全部的账户金额另加负数的费用到位于 S[-1]处的地址。

如上所述，我们不是要让人们直接写以太坊脚本，而是发布编译器从高级语言来生成以太坊脚本。头两个编译的目标很可能会是之前描述的类 C 语言，而第二个是更全面的头等函数语言，它会支持数组和任意长度的字符串。对于编译器来说，编译类 C 语言比较容易，变量可以被指定到内存指数，编译算术运算符会复杂一些，其方式是转化为反向波兰表示法（例如，(3 + 5) * (x + y) -> PUSH 3 PUSH 5 ADD PUSH 0 MLOAD PUSH 1 MLOAD ADD MUL）。头等函数语言因为变量辖域而更复杂，可能的解决办法在内存里是保留一份连接着的堆栈框架的列表，通过给每个堆栈框架 N 个内存槽来实现，这里 N 是程序里不同变量名的数量。获取变量的方法分以下几步：从上到下搜索堆栈框架列表直到有一个框架包含指向变量的指针；为了记住，拷贝指针至堆栈框架的顶部；然后在指针处发回数值。然而，这些属于长远考虑，因为编译与真正的协议是分离的，所以等网络运行起来很久之后再来研究编译策略都是可能的。

费用

以太坊会有七种主要费用，其中 1 种涉及交易发送者，6 种涉及合约。这些费用是：

- TXFEE (100x) - 发送交易的费用
- NEWCONTRACTFEE (100x) - 新建合约的费用，其中不包括脚本代码里的每一项的内存费
- STEPFEE (x) - 在程序执行时 16 步之后的每一计算步的费用
- MEMORYFEE (100x) - 把一个新项加进合约的内存里的费用，包括新建合约的时候。这个内存费是唯一不支付给矿工的费用，当合约的内存撤销时它会被返还
- DATAFEE (20x) - 从合约内部调用合约内存的费用
- EXTROFEE (40x) - 合约内部的另一合约调用内存的费用
- CRYPTOFEE (20x) - 使用密码运算的费用

以太坊里一个创新是将费用设成与难度的平方根成反比例关系，那就是 x = floor(10^21 / floor(difficulty ^ 0.5))，原因有两点：

1. 这样做建立了一套完全去中心化的设定交易费的机制，这将避免费用随着网络的成长增长过快。例如，如果以太币价值涨到原来的 4 倍，那全网的挖矿算力和挖矿难度都会变 4 倍，也就意味着以以太币计价的交易费变为 2 倍，而真实交易费其实只增加为 2 倍。

2. 如果由于摩尔定律计算机的性能增强为原来的 4 倍，那难度也会增到 4 倍，而费用会减少一半，这反映了计算机处理更高负载的能力的提升 。

这个指数小于 1 的原因有二：1）摩尔定律对各个因素的影响是不一样的，对计算速度的影响可能大于对在交易处理管线中最慢的那个组分的影响；2）如果是由于计算机数量的增加，而不是单机的计算速度的增加，使得以太网络快速成长，那么 指数=1 会让每台计算机负载过高。

结论

从很多方面看，以太坊协议的设计哲学跟现阶段其它密码货币恰好相反，它们致力于增加复杂性并加入各种各样的“特性”，而以太坊却是将特性剥离。此协议不“支持”多重签名交易、多重输入和输出、哈稀代码、锁定时间、以及另外一些就连比特币都支持的特性，相反，它的全部复杂性都来自于一门通用的图灵完备的脚本语言，这语言可以通过合约机制来建立几乎任何可以以数学方式表述

的特性。这样，我们就拥有了一套具有独特潜能的协议。不像那些封闭而专用的协议，它们只能用来创建某组单独的应用，如数据存储、赌博或金融，而以太坊从设计上就是开放的，我们相信它极其适合为将来数量庞大的各种金融或非金融协议充当底层协议。

引用文献和进阶学习

1. Colored coins whitepaper: https://docs.google.com/a/buterin.com/document/d/1AnkP_cVZTCMLIzw4DvsW6M8Q2JC0lIzrTLuoWu2z1BE/edit
2. Mastercoin whitepaper: https://github.com/mastercoin-MSC/spec
3. Decentralized autonomous corporations, Bitcoin Magazine: http://bitcoinmagazine.com/7050/bootstrapping-a-decentralized-autonomous-corporation-part-i/
4. Smart property: https://en.bitcoin.it/wiki/Smart_Property
5. Smart contracts: https://en.bitcoin.it/wiki/Contracts
6. Simplified payment verification: https://en.bitcoin.it/wiki/Scalability#Simplified*payment*verification
7. Merkle trees: http://en.wikipedia.org/wiki/Merkle_tree
8. Patricia trees: http://en.wikipedia.org/wiki/Patricia_tree
9. Bitcoin whitepaper: http://bitcoin.org/bitcoin.pdf
10. GHOST: http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf
11. StorJ and Autonomous Agents, Jeff Garzik: http://garzikrants.blogspot.ca/2013/01/storj-and-bitcoin-autonomous-agents.html
12. Mike Hearn on Smart Property at Turing Festival: http://www.youtube.com/watch?v=Pu4PAMFPo5Y
13. Ethereum RLP: http://wiki.ethereum.org/index.php/RLP
14. Ethereum Merkle Patricia trees: http://wiki.ethereum.org/index.php/Patricia_Tree
15. Ethereum Dagger: http://wiki.ethereum.org/index.php/Dagger
16. Ethereum C-like language: http://wiki.ethereum.org/index.php/CLL