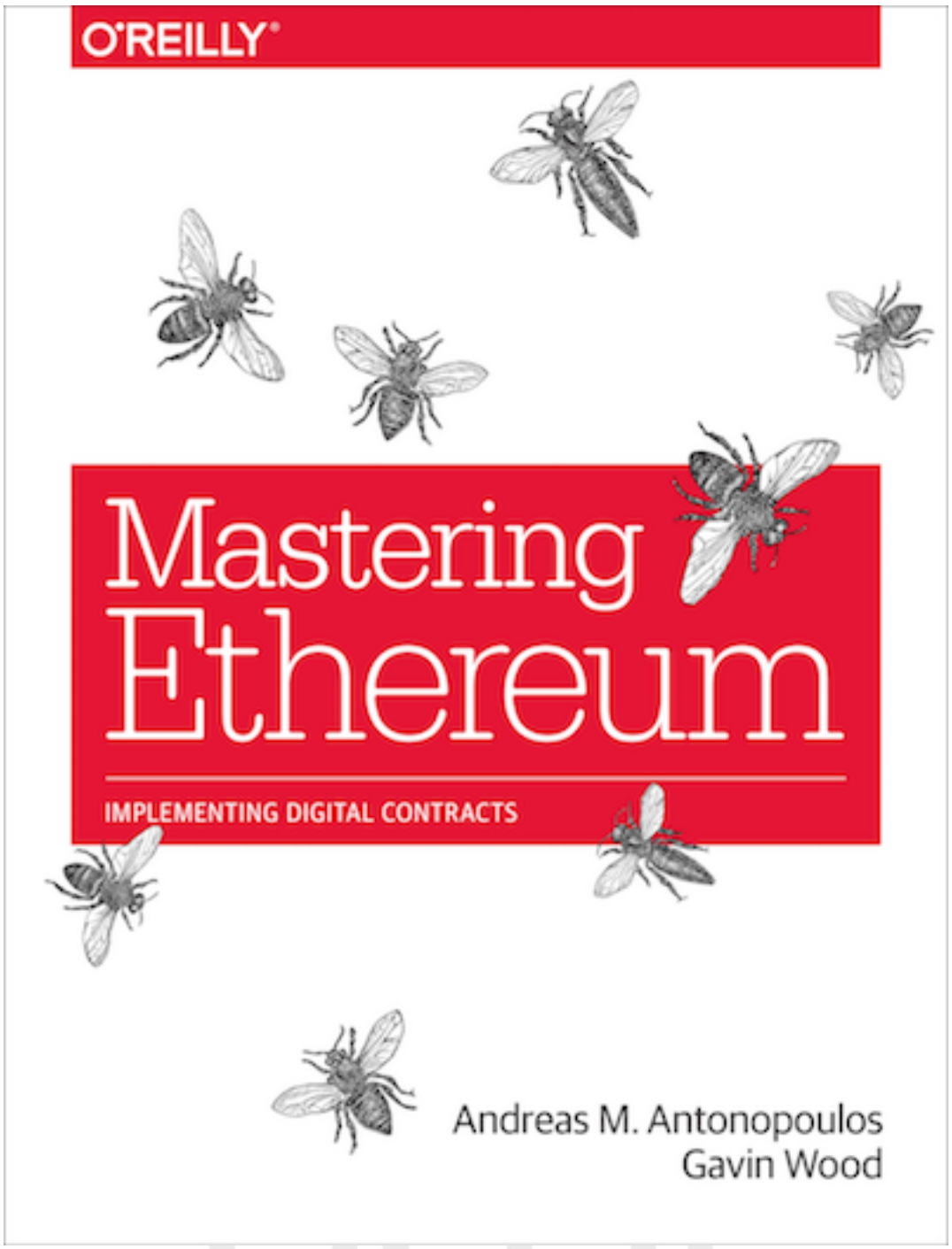

O'REILLY
Mastering
Ethereum
IMPLEMENTING DIGITAL CONTRACTS
Andreas M. Antonopoulos
Gavin Wood

目录

感谢一下所有参与翻译及校对的同学：
第一章：哗哗哗。第二章：左左。第三章：哗哗哗。第四章：joy,奔跑，H 的翻译，284，Raveb，温颖的校对。第五章：游戏 99，Fanlu,常旅客的翻译，Mayday，游戏 99，Iris 的校对。第六章：colors。第七章：奔跑。第八章：Mayday。第九章：哗哗哗，joy，mayday，colors，iris，失眠君，游戏 99，希元，奔跑。第十章：用心，田七，失眠君，Max 的翻译，詹锦，Nini，吴凯，topsun 的校对。第十一章：joy。第十二章：失眠君。第十三章：colos。第十四章：Iris

同样感谢本次参与格式校对的 Joey、Ryan、Iris、用心、奔跑、花生。

以上同学来自 AMT 社区，AMT 社区以“创造价值、传递价值、实现价值互联”为共识，让一群来自不同行业、不同地区的人走到了一起，为了实现共同目标而贡献自我价值。AMT 社区相信区块链技术必将改变人类生活方式，未来社会的联通将会在现有的信息互联的基础上进化成价值互联，在区块链构建的信任体系中，各式各样的人或物将自我价值通过区块链网络进行传递，形成丰富的价值互联网络生态，这最终将极大的提升社会生产效率。了解详情请访问 http://www.vnscoin.org/。

大家可以对以下地址进行捐赠，捐赠的 VNS 会用来作为以后翻译任务的奖励基金。
同时 AMT 社区招募更多对区块链有兴趣，有意参与社区共建的伙伴们，你可以按你所长通过翻译、宣传来贡献力量，而如果你想在 VNS 上面尝试、学习以太坊的智能合约，我们还将视情况赠送给你 VNS 帮助你的提升。
谢谢大家！


接受社区捐赠打赏地址
0x4b2478a48ffcc061d20a512b82415df66bbf691e
复制钱包地址

前言

本书由是 Andreas M. Antonopoulos 和 Gavin Wood 博士的合作完成。一系列幸运的巧合让这两位作者走到一起，携手努力激发数百名贡献者，他们本着开源和创作公共文化的最佳精神，创作了这本书。

Gavin 一直希望写一本扩展黄皮书（他对以太坊协议的技术描述）的书已有一段时间了，主要是为了向更广泛的受众开放，这是原始希腊字母文件不能允许的。

当 Gavin 与 Andreas 交谈时，他的计划正在进行中——出版商已经成立了——Andreas 是他在以太坊工作的一开始就认识的，当时他是该领域的知名人士

Andreas 最近出版了他的着作 *Mastering Bitcoin*（O'Reilly）的第一版，该书迅速成为比特币和加密货币的权威技术指南。几乎在这本书出版后，他的读者就开始问他："你什么时候写'精通以太坊'？" Andreas 已经在考虑他的下一个专题，并发现以太坊是一个引人注目的技术主题。

最后，2016 年 5 月，Gavin 和 Andreas 很巧合的同时在同一个城市。他们聚在一起喝咖啡聊聊这本书。由于 Andreas 和 Gavin 都是开源模式的忠实拥护者，他们都致力于将这项工作作为一项协作努力，并在 Creative Commons 许可下发布。值得庆幸的是，出版商 O'Reilly Media 很高兴同意，*Mastering Ethereum* 项目正式启动。

如何使用本书

本书旨在作为参考手册和以太坊的完整探索。前两章提供了一个温和的介绍，适合新手用户，这些章节中的示例可以由任何具有一定技术的人完成。这两章将使您对基础知识有一个很好的掌握，并教会您使用以太坊的基本工具。[ethereum_clients_chapter]及以后主要面向程序员，包括许多技术主题和编程示例。

作为关于以太坊的参考手册和完整的叙述，这本书不可避免地包含一些重复。有些主题，例如 *gas*，必须尽早引入，以使其余主题有意义，但也会在各自的章节中进行深入研究。

最后，本书的索引能让读者通过关键字轻松找到非常具体的主题和相关部分。

目标受众

本书主要面向程序员。如果您会使用编程语言，本书将教您智能合约区块链如何工作，如何使用它们，以及如何使用它们开发智能合约和分布式应用程序。前几章也适合作为非编码器的以太坊的深入介绍。

本书中使用的约定

本书使用以下印刷约定

斜体

表示新术语，URL，电子邮件地址，文件名和文件扩展名。

等宽

用于程序列表，以及段落内部，用于引用程序元素，如变量或函数名称，数据库，数据类型，环境变量，语句和关键字。

等宽加粗

显示应由用户按字面输入的命令或其他文本。

等宽斜体

显示应替换为用户提供的值或由上下文确定的值的文本。

提示	此图标表示提示或建议。
注释	此图标表示一般注释。
警告	此图标表示警告或警告。

代码示例

这些示例在 Solidity，Vyper 和 JavaScript 中说明，并使用类 Unix 操作系统的命令行。所有代码片段都在*代码*子目录下的 GitHub 存储库中提供。分支书籍代码，尝试代码示例，或通过 GitHub 提交更正：

https：//github.com/ethereumbook/ethereumbook。

所有代码片段都可以在大多数操作系统上进行复制，只需安装少量的编译器，解释器和相应语言的库。必要时，我们提供基本安装说明和这些说明输出的分步示例。

一些代码片段和代码输出已重新格式化以进行打印。在所有这些情况下，行都用反斜杠（\）字符分隔，后跟换行符。在抄写示例时，删除这两个字符并再次连接这些行，您应该看到与示例中显示完全相同的结果。

所有代码片段都尽可能使用实际值和计算，以便您可以从示例构建为示例，并在您编写的任何代码中查看相同的值以计算出相同的结果。例如，私钥和相应的公钥和地址都是真实的。样本交易，合约，区块和区块链参考都已引入实际的以太坊区块链，并且是公共账本的一部分，因此您可以查看它们。

使用代码示例

这本书是为了帮助你完成工作。通常，如果本书提供了示例代码，您可以在程序和文档中使用它。除非您复制了大部分代码，否则您无需与我们联系以获得许可。例如，使用本书中几个代码块编写的程序不需要许可。出售或分发 O'Reilly 书籍中的示例 CD-ROM 需要获得许可。通过引用本书并引用示例代码来回答问题不需要许可。将本书中的大量示例代码合并到产品文档中需要获得许可。

我们感谢，但不要求，署名。署名通常包括标题，作者，出版商，ISBN 和版权。例如：“ 由 Andreas M. Antonopoulos 和 Gavin Wood 博士（O'Reilly）精通以太坊。版权所有 2019 The Ethereum Book LLC 和 Gavin Wood，978-1-491-97194-9。“

精通以太坊是根据知识共享署名 - 非商业性 - 无衍生作品 4.0 国际许可（CC BY-NC-ND 4.0）提供的。

如果您认为您对代码示例的使用超出了合理使用范围或上述许可范围，请随时通过 permissions@oreilly.com 与我们联系。

公司和产品的参考

所有对公司和产品的引用都是为了教育，演示和参考目的。作者并不认可所提及的任何公司或产品。我们尚未测试本书中显示的任何产品，项目或代码段的运算或安全性。使用它们需要您自担风险！

本书中的以太坊地址和交易

本书中使用的以太坊地址，交易，密钥，QR 码和区块链数据在很大程度上都是真实的。这意味着您可以浏览区块链，查看作为示例提供的事务，使用您自己的脚本或程序检索它们等。

但请注意，用于构建本书中打印的地址的私钥已被“公开”。这意味着如果您向这些地址中的任何一个汇款，这笔钱将永久丢失或（更有可能）被挪用，因为任何阅读该书的人都可以使用此处打印的私钥来获取。

警告	不要向本书中的任何地址发送以太坊。因为你的以太坊将被其他读者取走或永久丢失。

O'Reilly Safari

注意	Safari（以前称为 Safari Books Online）是一个基于会员的培训和参考平台，适用于企业，政府，教育工作者和个人。

会员可以访问来自 250 多家出版商的数千本书籍，培训视频，学习渠道，互动教程和策划播放列表，包括 O’Reilly Media，Harvard Business Review，Prentice Hall Professional，Addison-Wesley Professional，Microsoft Press，Sams，Que ，Peachpit Press，Adobe，Focal Press，Cisco Press，John Wiley＆Sons，Syngress，Morgan Kaufmann，IBM Redbooks，Packt，Adobe Press，FT Press，Apress，Manning，New Riders，McGraw-Hill，Jones＆Bartlett 和 Course 技术，等等。

欲了解更多信息，请访问 http://oreilly.com/safari。

如何联系我们

有关掌握以太坊以及开放版和翻译的信息，请访问 https://ethereumbook.info/。

请向出版商提出有关本书的评论和问题：

O’Reilly Media，Inc。
1005 Gravenstein Highway North
Sebastopol，CA 95472
800-998-9938（在美国或加拿大）
707-829-0515（国际或当地）
707-829-0104（传真）

请将有关本书的评论或技术问题发送至 bookquestions@oreilly.com。

有关我们的书籍，课程，会议和新闻的更多信息，请访问我们的网站 https://www.oreilly.com。

在 Facebook 上找到我们：https：//facebook.com/oreilly

在 Twitter 上关注我们：https：//twitter.com/oreillymedia

在 YouTube 上观看我们：https：//www.youtube.com/oreillymedia

联系 Andreas

您可以在他的个人网站上联系 Andreas M. Antonopoulos:https： //antonopoulos.com/

在 YouTube 上订阅 Andreas 的频道：https： //www.youtube.com/aantonop

就像 Andreas 在 Facebook 上的页面一样：
https： //www.facebook.com/AndreasMAntonopoulos

在 Twitter 上关注 Andreas：https： //twitter.com/aantonop

在 LinkedIn 上与 Andreas 联系：https： //linkedin.com/company/aantonop

安德烈亚斯还要感谢所有通过每月捐款支持他的工作的顾客。您可以通过 https://patreon.com/aantonop 在 Patreon 上支持 Andreas 。

联系加文

您可以在他的个人网站上联系 Gavin Wood 博士：http： //gavwood.com/

在 Twitter 上关注 Gavin：https： //twitter.com/gavofyork

Gavin 一般在 Riot.im 上的 Polkadot Watercooler 中闲逛：http：//bit.ly/2xciG68

Andreas 的致谢

我把我对文字和书籍的热爱归功于我的母亲 Theresa，她在一所墙上挂满了书的房子里抚养了我。我的母亲在 1982 年给我买了我的第一台电脑，尽管她说她是一个有技术恐惧症的人。我父亲 Menelaos 是一位土木工程师，他 80 岁时出版了他的第一本书，他教会了我逻辑和分析思维以及对科学和工程的热爱。

谢谢大家支持我走过这段旅程。

Gavin 的致谢

当我 9 岁的时候，我的母亲从邻居那里得到了我的第一台电脑，没有这台电脑我的技术进步无疑会减少。我童年时的电力恐惧，必须感谢 Trevor 和我的祖父母，他们一次又一次地庄严的履行“看着我插上电”的重任，没有他们，电脑对我来说就没用。我还要感谢我一生

幸运的各种教育工作者，从邻居 Sean（他教我第一个计算机程序）到我的小学老师 Quinn 先生，他为我更多练习编程和少学历史，直到 Richard Furlong-Brown 这样的中学教师，他让我更多练习编程而不是打橄榄球。

我必须感谢我的孩子的母亲 Jutta，感谢她的不断支持，以及我生命中的许多人，无论新老朋友，粗略的说，这些让我保持理智。最后，对于 Aeron Buchanan 来说，必须要有一个巨大的感谢，没有他，我生命中的最后五年永远不可能以他们的方式展开，没有他们花时间，支持和指导，这本书将不会像现在这样好。

捐款

许多贡献者在 GitHub 的早期发布草案中提供了评论，修正和补充。

两位 GitHub 编辑为 GitHub 做出了贡献，他们自愿管理，审查，编辑，合并和批准拉取请求和问题：

领导 GitHub 编辑：Francisco Javier Rojas Garcia（fjrojasgarcia）

协助 GitHub 编辑：William Binns（wbnns）

主要贡献包括 DApps，ENS，EVM，fork 历史，gas，oracles，智能合约安全和 Vyper。由于时间和空间的限制，未在第一版中包含的其他贡献可以在 GitHub 仓库的 *contrib* 文件夹中找到。整本书中数以千计的小贡献提高了其质量，易读性和准确性。衷心感谢所有贡献者！

以下是所有 GitHub 贡献者的按字母顺序排序的列表，包括括号中的 GitHub ID：

- Abhishek Shandilya (abhishandy)
- Adam Zaremba (zaremba)
- Adrian Li (adrianmcli)
- Adrian Manning (agemanning)
- Alejandro Santander (ajsantander)
- Alejo Salles (fiiiu)
- Alex Manuskin (amanusk)
- Alex Van de Sande (alexvandesande)
- Anthony Lusardi (pyskell)
- Assaf Yossifoff (assafy)
- Ben Kaufman (ben-kaufman)

- Bok Khoo (bokkypoobah)
- Brandon Arvanaghi (arvanaghi)
- Brian Ethier (dbe)
- Bryant Eisenbach (fubuloubu)
- Chanan Sack (chanan-sack)
- Chris Remus (chris-remus)
- Christopher Gondek (christophergondek)
- Cornell Blockchain (CornellBlockchain)
  - Alex Frolov (sashafrolov)
  - Brian Guo (BrianGuo)
  - Brian Leffew (bleffew99)
  - Giancarlo Pacenza (GPacenza)
  - Lucas Switzer (LucasSwitz)
  - Ohad Koronyo (ohadh123)
  - Richard Sun (richardsfc)
- Cory Solovewicz (CorySolovewicz)
- Dan Shields (NukeManDan)
- Daniel Jiang (WizardOfAus)
- Daniel McClure (danielmcclure)
- Daniel Peterson (danrpts)
- Denis Milicevic (D-Nice)
- Dennis Zasnicoff (zasnicoff)
- Diego H. Gurpegui (diegogurpegui)
- Dimitris Tsapakidis (dimitris-t)
- Enrico Cambiaso (auino)
- Ersin Bayraktar (ersinbyrktr)

- Flash Sheridan (FlashSheridan)
- Franco Daniel Berdun (fMercury)
- Harry Moreno (morenoh149)
- Hon Lau (masterlook)
- Hudson Jameson (Souptacular)
- Iuri Matias (iurimatias)
- Ivan Molto (ivanmolto)
- Jacques Dafflon (jacquesd)
- Jason Hill (denifednu)
- Javier Rojas (fjrojasgarcia)
- Jaycen Horton (jaycenhorton)
- Joel Gugger (guggerjoel)
- Jon Ramvi (ramvi)
- Jonathan Velando (rigzba21)
- Jules Lainé (fakje)
- Karolin Siebert (karolinkas)
- Kevin Carter (kcar1)
- Krzysztof Nowak (krzysztof)
- Lane Rettig (lrettig)
- Leo Arias (elopio)
- Liang Ma (liangma)
- Luke Schoen (ltfschoen)
- Marcelo Creimer (mcreimer)
- Martin Berger (drmartinberger)
- Masi Dawoud (mazewoods)
- Matthew Sedaghatfar (sedaghatfar)

- Michael Freeman (stefek99)
- Miguel Baizan (mbaiigl)
- Mike Pumphrey (bmmpxf)
- Mobin Hosseini (iNDicat0r)
- Nagesh Subrahmanyam (chainhead)
- Nichanan Kesonpat (nichanank)
- Nick Johnson (arachnid)
- Omar Boukli-Hacene (oboukli)
- Paulo Trezentos (paulotrezentos)
- Pet3rpan (pet3r-pan)
- Pierre-Jean Subervie (pjsub)
- Pong Cheecharern (Pongch)
- Qiao Wang (qiaowang26)
- Raul Andres Garcia (manilabay)
- Roger Häusermann (haurog)
- Solomon Victorino (bitsol)
- Steve Klise (sklise)
- Sylvain Tissier (SylTi)
- Taylor Masterson (tjmasterson)
- Tim Nugent (timnugent)
- Timothy McCallum (tpmccallum)
- Tomoya Ishizaki (zaq1tomo)
- Vignesh Karthikeyan (meshugah)
- Will Binns (wbnns)
- Xavier Lavayssière (xalava)
- Yash Bhutwala (yashbhutwala)

- Yeramin Santana (ysfdev)
- Zhen Wang (zmxv)
- ztz (zt2)

如果没有上面列出的每个人提供的帮助，本书就不可能实现。您的贡献展示了开源和开放文化的力量，我们永远感谢您的帮助。谢谢。

来源

本书引用了各种公开和开放许可的来源：

https://github.com/ethereum/vyper/blob/master/README.md
麻省理工学院许可（MIT）

https://vyper.readthedocs.io/en/latest/
麻省理工学院许可（MIT）

https://solidity.readthedocs.io/en/v0.4.21/common-patterns.html
麻省理工学院许可（MIT）

https://arxiv.org/pdf/1802.06038.pdf
Arxiv Non-Exclusive-Distribution

https://github.com/ethereum/solidity/blob/release/docs/contracts.rst#inheritance
麻省理工学院许可（MIT）

https://github.com/trailofbits/evm-opcodes
Apache 2.0

https://github.com/ethereum/EIPs/
Creative Commons CC0

https://blog.sigmaprime.io/solidity-security.html
Creative Commons CC BY 4.0

快速术语表

这个快速术语表包含许多与以太坊相关的术语。这些术语在本书中都有使用，所以请将其加入书签以便快速参考。

账户 Account

包含地址、余额、随机数以及可选存储和代码的对象。账户可以是合约账户或外部拥有账户（EOA，externally owned account）.

地址 Address

一般来说，这代表一个 EOA 或合约，它可以在区块链上接收（目标地址）或发送（源地址）交易。更具体地说，它是 ECDSA 公钥的 Keccak 散列的最右边的 160 位，表现为 16 进制的 40 个字符长度，在前面加上“0x”字符。

断言 Assert

在 Solidity 中，assert(false) 编译为 0xfe, 是一个无效的操作码，用尽所有剩余的燃气（Gas），并恢复所有更改。 当 assert() 语句失败时，说明发生了严重的错误或异常，你必须修复你的代码。 你应该使用 assert 来避免永远不应该发生的条件。

大端序 Big-endian

一种数值的位置表示形式，最高有效位放在最前面。对应小端序（little-endian），最低有效位在前。

比特币改进提议 BIPs

比特币改进提议，Bitcoin Improvement Proposals。比特币社区成员提交的一组提案，旨在改进比特币。例如，BIP-21 是改进比特币统一资源标识符（URI）方案的建议。

区块 Block

区块是关于所包含的交易的所需信息（区块头）的集合，以及称为 ommer 的一组其他区块头。它被矿工添加到以太坊网络中。

区块链 Blockchain

由工作证明系统验证的一系列区块，每个区块都连接到它的前区块，一直到创世区块。这与比特币协议不同，因为它没有块大小限制；它改为使用不同的燃气限制。

拜占庭分叉 Byzantium Fork

拜占庭是大都会（ Metropolis ）发展阶段的两大硬分叉之一。它包括 EIP-649：大都会难度炸弹延迟和区块奖励减少，其中冰河时代（见下文）延迟 1 年，而区块奖励从 5 个以太坊减至 3 个以太坊。

编译 Compiling

将高级编程语言（例如 Solidity）编写的代码转换为低级语言（例如 EVM 字节码）

共识 Consensus

当大量节点，通常是网络上的大多数节点，在其本地验证的最佳区块链中都有相同的区块的情况。 不要与共识规则混淆。

共识规则 Consensus rules

完整节点为了与其他节点保持一致，遵循的区块验证规则。不要与共识混淆。

君士坦丁分叉 Constantinople fork

大都会阶段的第二部分，2018 年中期的计划。预计将包括切换到混合工作证明/权益证明共识算法，以及其他变更。

合约账户 Contract account

包含代码的账户，每当它从另一个账户（EOA 或合约）收到交易时执行。

合约创建交易 Contract creation transaction

一个特殊的交易，以“零地址”作为收件人，用于注册合约并将其记录在以太坊区块链中（请参阅“零地址”）。

去中心化自治组织 DAO

去中心化自治组织 Decentralised Autonomous Organization. 没有层级管理的公司和其他组织。也可能是指 2016 年 4 月 30 日发布的名为“The DAO”的合约，该合约于 2016 年 6 月遭到黑客攻击，最终在第 1,192,000 个区块激起了硬分叉（代号 DAO），恢复了被攻击的 DAO 合约，并导致了以太坊和以太坊经典两个竞争系统。

去中心化应用 DApp

去中心化应用 Decentralised Application. 狭义上，它至少是智能合约和 web 用户界面。更广泛地说，DApp 是一个基于开放式，分散式，点对点基础架构服务的 Web 应用程序。另外，许多 DApp 包括去中心化存储和/或消息协议和平台。

契约 Deed

ERC721 提案中引入了不可替代的标记标准。与 ERC20 代币不同，契约证明了所有权并且不可互换，虽然它们还未在任何管辖区都被认可为合法文件，至少目前不是（另请参阅“NFT”）。

难度 Difficulty

网络范围的设置，控制产生工作量证明需要多少计算。

数字签名 Digital signature

数字签名算法是一个过程，用户可以使用私钥为文档生成称为“签名”的短字符串数据，以便具有签名，文档，和相应公钥的任何人，都可以验证（1 ）该文件由该特定私钥的所有者“签名”，以及（2）该文件在签署后未被更改。

椭圆曲线数字签名算法 ECDSA

椭圆曲线数字签名算法（ Elliptic Curve Digital Signature Algorithm，ECDSA ）是以太坊用来确保资金只能由合法所有者使用的加密算法。

以太坊改进建议 EIP

以太坊改进建议，Ethereum Improvement Proposals，描述以太坊平台的建议标准。EIP 是向以太坊社区提供信息的设计文档，描述新的功能，或处理过程，或环境。有关更多信息，请参见 https://github.com/ethereum/EIPs（另请参见下面的 ERC 定义）。

熵 Entropy

在密码学领域，表示可预测性的缺乏或随机性水平。在生成秘密信息（如主私钥）时，算法通常依赖高熵源来确保输出不可预测。

以太坊名称服务 ENS

以太坊名称服务，Ethereum Name Service. 更多信息，参见 https://github.com/ethereum/ens/.

外部拥有账户 EOA

外部拥有账户，Externally Owned Account. 由或为以太坊的真人用户创建的账户。

以太坊注释请求 ERC

以太坊注释请求 Ethereum Request for Comments. 一些 EIP 被标记为 ERC，表示试图定义以太坊使用的特定标准的建议。

Ethash

以太坊 1.0 的工作量证明算法。更多信息，参见 https://github.com/ethereum/wiki/wiki/Ethash.

以太 Ether

以太 Ether，是以太坊生态系统中使用的本地货币，在执行智能合约时承担燃气费用。它的符合是 Ξ，极客用的大写 Xi 字符.

事件 Event

事件允许 EVM 日志工具的使用，后者可以用来在 DApp 的用户界面中调用 JavaScript 回调来监听这些事件。更多信息，参见 http://solidity.readthedocs.io/en/develop/contracts.html#events。

以太坊虚拟机 EVM

Ethereum Virtual Machine, 基于栈的，执行字节码的虚拟机。在以太坊中，执行模型指定了系统状态如何在给定一系列字节码指令和少量环境数据的情况下发生改变。这是通过虚拟状态机的正式模型指定的。

EVM 汇编语言 EVM Assembly Language

一种人类可读的 EVM 字节码的形式。

回退方法 Fallback function

默认的方法，当缺少数据或声明的方法名时执行。

水龙头 Faucet

一个网站，为想要在 testnet 上做测试的开发人员提供免费测试以太形式的奖励。

前沿 Frontier

以太坊的试验开发阶段，从 2015 年 7 月至 2016 年 3 月。

Ganache

私有以太坊区块链，你可以在上面进行测试，执行命令，在控制区块链如何运作时检查状态。

燃气 Gas

以太坊用于执行智能合约的虚拟燃料。以太坊虚拟机使用会计机制来衡量天然气的消耗量并限制计算资源的消耗。参见“图灵完备”。 燃气是执行智能合约的每条指令产生的计算

单位。燃气与以太加密货币挂钩。燃气类似于蜂窝网络上的通话时间。因此，以法定货币进行交易的价格是 gas （ETH /gas）（法定货币/ETH）。

燃气限制 Gas limit

在谈论区块时，它们也有一个名为燃气限制的区域。它定义了整个区块中所有交易允许消耗的最大燃气量。

加文·伍德 Gavin Wood

Ethereum 的联合创始人和前首席技术官，一个英国程序员。2014 年 8 月他提出了 solidity，一个面向合约的编程语言，用于编写智能合约。

创世区块 Genesis block

区块链中的第一个块，用来初始化特定的网络和加密数字货币。

Geth

Go 语言的以太坊。Go 编写的最突出的以太坊协议实现之一。

硬分叉 Hard fork

硬分叉也称为硬分叉更改，是区块链中的一种永久性分歧，通常发生在非升级节点无法验证升级节点创建的遵循新共识规则的区块时。不要与分叉，软分叉，软件分叉或 Git 分叉混淆。

哈希值 Hash

通过哈希方法为可变大小的数据生成的固定长度的指纹。

分层确定钱包 HD wallet

使用分层确定密钥生成和传输协议的钱包（BIP32）。

分层确定钱包种子 HD wallet seed

HD 钱包种子或根种子是一个可能很短的值，用作生成 HD 钱包的主私钥和主链码的种子。钱包种子可以用助记词表示，使人们更容易复制，备份和恢复私钥。

家园 Homestead

以太坊的第二个发展阶段，于 2016 年 3 月在 1,150,000 区块启动。

互换客户端地址协议 Inter exchange Client Address Protocol (ICAP)

以太坊地址编码，与国际银行帐号（IBAN）编码部分兼容，为以太坊地址提供多样的，校验和的，可互操作的编码。 ICAP 地址可以编码以太坊地址或通过以太坊名称注册表注册的常用名称。他们总是以 XE 开始。其目的是引入一个新的 IBAN 国家代码：XE，X 表示 "extended"， 加上以太坊的 E，用于非管辖货币（例如 XBT，XRP，XCP）。

冰河时代 Ice Age

以太坊在 200,000 区块的硬分叉，提出难度指数级增长（又名难度炸弹），引发了到权益证明 Proof-of-Stake 的过渡。

集成开发环境 IDE (Integrated Development Environment)

集成的用户界面，结合了代码编辑器、编译器、运行时和调试器。

不可变的部署代码问题 Immutable Deployed Code Problem

一旦部署了契约(或库)的代码，它就成为不可变的。修复可能的 bug 并添加新特性是软件开发周期的关键。这对智能合约开发来说是一个挑战。

内部交易（又称“消息”）Internal transaction (also ″message″)

从一个合约地址发送到另一个合约地址或 EOA 的交易。

星际文件系统 IPFS

星际文件系统（InterPlanetary File System）。一种协议、网络和开源项目，旨在创建一种在分布式文件系统中存储和共享超媒体的可寻址的点对点方法。

关键推导函数 KDF

关键推导函数(Key Derivation Function)。又称为“密码扩展算法”， 密钥存储格式使用它反复哈希密码，以防止密码加密受到暴力、字典和 rainbow 表攻击。

Keccak256

以太坊使用的加密哈希方法。虽然在早期 Ethereum 代码中写作 SHA-3，但是由于在 2015 年 8 月 SHA-3 完成标准化时，NIST 调整了填充算法，所以 Keccak256 不同于标准的 NIST-SHA3。Ethereum 也在后续的代码中开始将 SHA-3 的写法替换成 Keccak256 。

密钥推导方法 Key Derivation Function (KDF)

也称为密码扩展算法，它被 keystore 格式使用，以防止对密码加密的暴力破解，字典或彩虹表攻击。它重复对密码进行哈希。

密钥存储文件 Keystore file

JSON 编码的文件，包含一个（随机生成的）私钥，被一个密码加密，以提供额外的安全性。

LevelDB

LevelDB 是一种开源的磁盘键值存储系统。LevelDB 是轻量的，单一目标的持久化库，支持许多平台。

库 Library

以太坊中的库，是特殊类型的合约，没有用于支付的方法，没有后备方法，没有数据存储。所以它不能接收或存储以太，或存储数据。库用作之前部署的代码，其他合约可以调用只读计算。

轻量级客户端 Lightweight client

轻量级客户端是一个以太坊客户端，它不存储区块链的本地副本，也不验证块和事务。它提供了钱包的功能，可以创建和广播交易。

消息 Message

内部交易，从未被序列化，只在 EVM 中发送。

METoken

Mastering Ethereum Token. 本书中用于演示的 ERC20 代币。

大都会阶段 Metropolis Stage

大都会是以太坊的第三个开发阶段，在 2017 年 10 月启动。

矿工 Miner

通过重复哈希计算，为新的区块寻找有效的工作量证明的网络节点。

Mist

Mist 是以太坊基金会创建的第一个以太坊浏览器。它还包含一个基于浏览器的钱包，这是 ERC20 令牌标准的首次实施（Fabian Vogelsteller，ERC20 的作者也是 Mist 的主要开发人员）。Mist 也是第一个引入 camelCase 校验码（EIP-155）的钱包。Mist 运行完整节点，提供完整的 DApp 浏览器，支持基于 Swarm 的存储和 ENS 地址

网络 Network

将交易和区块传播到每个以太坊节点（网络参与者）的对等网络。

不可替代 token NFT

不可替代 token（A non-fungible token）（另参见“deed”）。这是 ERC721 提议引入的 token 标准。NFTs 可以被追踪和交易，但每个 token 都是独一无二的；他们不像 ERC20 tokens 一样可以替换。NTFs 可以表示数字或者物理资产的所有权。

节点 Node

参与到对等网络的软件客户端。

随机数 Nonce

密码学中，随机数指代只可以用一次的数值。在以太坊中用到两类随机数。

Ommer

祖父节点的子节点，但它本身并不是父节点。当矿工找到一个有效的区块时，另一个矿工可能已经发布了一个竞争的区块，并添加到区块链顶部。像比特币一样，以太坊中的孤儿区块可以被新的区块作为 ommers 包含，并获得部分奖励。术语 "ommer" 是对父节点的兄弟姐妹节点的性别中立的称呼，但也可以表示为“叔叔”。

Parity

以太坊客户端软件最突出的支持共同操作（多重签名）的实现之一。

Private key

参见 “scret key”

权益证明 Proof-of-Stake (PoS)

权益证明是加密货币区块链协议旨在实现分布式共识的一种方法。权益证明要求用户证明一定数量的加密货币（网络中的“股份”）的所有权，以便能够参与交易验证。

工作量证明 Proof-of-Work (PoW)

一份需要大量计算才能找到的数据（证明）。在以太坊，矿工必须找到符合网络难度目标的 Ethash 算法的数字解决方案。

公钥 Public key

一串数字，通过私钥的单向函数导出，可以公开共享，任何人都可以使用它来验证使用相应私钥生成的数字签名

收据 Receipt

以太坊客户端返回的数据，表示特定交易的结果，包括交易的哈希值，其区块编号，使用的燃气量，以及在部署智能合约时的合约地址。

重入攻击 Re-entrancy Attack

当攻击者合约（Attacker contracts）调用受害者合约（Victim contracts）的方法时，可以重复这种攻击。让我们称它为 victim.withdraw()，在对该合约函数的原始调用完成之前，再次调用 victim.withdraw()方法，持续递归调用它自己。 递归调用可以通过攻击者合约的后备方法实现。 攻击者必须执行的唯一技巧是在用完燃气之前中断递归调用，并避免盗用的以太被还原。

奖励 Reward

Ether（ETH）的数量，包含在每个新区块中的金额作为网络对找到工作证明解决方案的矿工的奖励。

递归长度前缀 Recursive Length Prefix (RLP)

RLP 是一种编码标准，由以太坊开发人员设计用来编码和序列化任意复杂度和长度的对象（数据结构）。

中本聪 Satoshi Nakamoto

Satoshi Nakamoto 是设计比特币及其原始实现 Bitcoin Core 的个人或团队的名字。作为实现的一部分，他们也设计了第一个区块链。在这个过程中，他们是第一个解决数字货币的双重支付问题的。他们的真实身份至今仍是个谜。

密钥（私钥） Secret key (aka private key)

允许以太坊用户通过创建数字签名（参见公钥，地址，ECDSA）证明账户或合约的所有权的加密数字。

宁静 Serenity

以太坊第四个也是最后一个开发阶段。宁静还没有计划发布的日期。

Serpent

语法类似于 Python 的过程式（命令式）编程语言。也可以用来编写函数式（声明式）代码，尽管它不是完全没有副作用的。首先由 Vitalik Buterin 创建。

SHA

安全哈希算法 Secure Hash Algorithm，SHA 是美国国家标准与技术研究院（NIST）发布的一系列加密哈希函数。

单例 Singleton

计算机程序术语-用来描述只有一个实例可以存在的对象

智能合约 Smart Contract

在以太坊的计算框架上执行的程序。

Solidity

过程式（命令式）编程语言，语法类似于 Javascript, C++ 或 Java。以太坊智能合约最流行和最常使用的语言。由 Gavin Wood（本书的联合作者）首先创造

Solidity inline assembly

内联汇编 Solidity 中包含的使用 EVM 汇编（EVM 代码的人类可读形式）的代码。内联汇编试图解决手动编写汇编时遇到的固有难题和其他问题。

Spurious Dragon

在＃2,675,00 块的硬分叉，来解决更多的拒绝服务攻击向量，以及另一种状态清除。还有转播攻击保护机制。

Swarm

一种去中心化（P2P）的存储网络。与 Web3 和 Whisper 共同使用来构建 DApps。

Szabo

ether 的一种单位，1012 szabo = 1 ether.

Tangerine Whistle

在 #2,463,00 块的硬分叉，改变了某些 I/O 密集操作的燃气计算方式，并从拒绝服务攻击中清除累积状态，这种攻击利用了这些操作的低燃气成本。

测试网 Testnet

一个测试网络（简称 testnet），用于模拟以太网主要网络的行为。

交易 Transaction

由原始帐户签署的提交到以太坊区块链的数据，并以特定地址为目标。交易包含元数据，例如交易的燃气限额。

Truffle

一个最常用的以太坊开发框架。包含一些 NodeJS 包，可以使用 Node Package Manager (NPM) 安装。

图灵完备 Turing Complete

在计算理论中，如果数据操纵规则（如计算机的指令集，程序设计语言或细胞自动机）可用于模拟任何图灵机，则它被称为图灵完备或计算上通用的。这个概念是以英国数学家和计算机科学家阿兰图灵命名的。

Vitalik Buterin

Vitalik Buterin 是俄国-加拿大的程序员和作家，以太坊和 Bitcoin 杂志的联合创始人。

Vyper

一种高级编程语言，类似 Serpent，有 Python 式的语法，旨在接近纯函数式语言。由 Vitalik Buterin 首先创造。

钱包 Wallet

拥有你的所有密钥的软件。作为访问和控制以太坊账户并与智能合约交互的界面。请注意，密钥不需要存储在你的钱包中，并且可以从脱机存储（例如 USB 闪存驱动器或纸张）中检索以提高安全性。尽管名字为钱包，但它从不存储实际的硬币或代币。

Web3

web 的第三个版本。有 Gavin Wood 首先提出，Web3 代表了 Web 应用程序的新愿景和焦点：从集中拥有和管理的应用程序到基于去中心化协议的应用程序。

Wei

以太的最小单位，1018 wei = 1 ether.

Whisper

一种去中心化（P2P）消息系统。与 Web3 和 Swarm 一起使用来构建 DApps。

零地址 Zero address

特殊的以太坊地址，全部是由 0 组成（即 0x0000000000000000000000000000000000000000），被指定为创建一个智能合约所发起的交易（Transaction）的目标地址（即 to 参数的值）。

第一章 什么是以太坊？

什么是以太坊？

以太坊经常被描述为“世界计算机”。但这是什么意思呢？让我们从以计算机科学为中心的描述开始，然后尝试通过对以太坊的能力和特征进行更实际的分析来解读，同时将其与比特币和其他去中心化的信息交换平台（或简称“区块链”）进行比较。

从计算机科学的角度来看，以太坊是一个确定性但实际上无限制的状态机，由全局可访问的单一状态和将更改应用于该状态的虚拟机组成。

从更实际的角度来看，以太坊是一个开源的，全球分散的计算基础设施，执行称为智能合约的程序。 它使用区块链来同步和存储系统的状态变化，以及称为以太网的加密货币来计量和约束执行资源成本。

以太坊平台使开发人员能够构建具有内置经济功能的强大去中心化应用程序。 在提供高可用性，可审计性，透明度和中立性的同时，它还减少或消除了审查并降低了某些交易对手的风险。

与比特币的对比

很多人在接触以太坊时会具备以往加密货币经验，特别是比特币。以太坊与其他开放区块链共享许多共同元素：连接参与者的点对点网络，用于状态更新同步的拜占庭容错一致性算法（工作证明区块链），使用数字等加密原语签名和哈希，以及数字货币（以太）。

然而，在许多方面，以太坊的目的和结构都与之前的开放式区块链（包括比特币）截然不同。

以太坊的目的主要不是成为数字货币支付网络。虽然数字货币以太对于以太坊的运营都是不可或缺的，但以太是用作支付使用以太坊平台作为世界计算机的公用货币。

与具有非常有限的脚本语言的比特币不同，以太坊被设计为通用可编程区块链，其运行能够执行任意和无限复杂代码的虚拟机。 比特币的脚本语言有意地被限制为对支出条件的简单真/假评估，以太坊的语言是图灵完备的，这意味着以太坊可以直接用作通用计算机。

区块链的组成部分

开放的公共区块链主要组成部分（通常）是：

- 基于标准化的“流言算法”协议，连接参与者和传播交易以及已验证交易块的对等（P2P）网络
- 以事务形式表示状态转换的消息
- 一组共识规则，规定交易的构成以及有效的状态转换的原因
- 根据共识规则处理事务的状态机
- 一系列加密安全块，充当所有已验证和接受的状态转换的日志
- 一种共识算法，通过强制参与者合作执行共识规则来分散对区块链的控制
- 游戏理论上的良好激励方案（例如，工作量证明成本加块奖励），以便在开放环境中经济地保护状态机

上述一个或多个开源软件实现（“客户端”）

所有或大多数这些组件通常组合在一个软件客户端中。 例如，在比特币中，参考实现由比特币核心开源项目开发，并作为比特币客户端实现。 在以太坊中，有一个参考规范，而不是参考实现，黄皮书中的系统的数学描述（参见进一步阅读）。 有许多客户端是根据参考规范构建的。

在过去，我们使用术语“区块链”来表示刚刚列出的所有组件，作为包含所描述的所有特征的技术组合的简写参考。 然而，今天，有各种各样的区块链具有不同的属性。 我们需要限定符来帮助我们理解所讨论的区块链的特征，例如开放，公共，全球，分散，中立和审查制度，以确定这些组件允许的“区块链”系统的重要新兴特征。

并非所有区块链都是平等的。 当有人告诉你某些东西是区块链时，你不能确认答案；相反，你需要开始提出很多问题来澄清他们使用“区块链”这个词时的含义。 首先要求对前面列表中的组件进行描述，然后询问这个“区块链”是否具有开放，公开等特征。

以太坊的诞生

所有伟大的创新都能解决实际问题，以太坊也不例外。 以太坊是在人们认识到比特币模型的力量，并试图超越加密货币应用程序的时候构思出来的。 但开发人员面临一个难题：他们要么需要建立在比特币之上，要么开始新的区块链。 以比特币为基础意味着需要在受限的网络上试图找到变通方法。 有限的事务类型，数据类型和数据存储大小似乎限制了可直接在比特币上运行的各种应用程序；任何其他需要额外的脱链层，这立即否定了使用公共区块链的许多优点。 对于需要更多自由和灵活性同时保持链条的项目，新的区块链是唯一的选择。但这意味着很多工作：引导所有基础设施元素，详尽的测试等。

2013 年底，年轻的程序员和比特币爱好者 Vitalik Buterin 开始考虑进一步扩展比特币和 Mastercoin（一种将比特币扩展到提供基本智能合约的覆盖协议）的功能。 同年 10 月，Vitalik 对 Mastercoin 团队提出了一种更为通用的方法，即允许灵活且可编写脚本（但不是图灵完备）的合约取代 Mastercoin 的专业合约语言。 虽然给 Mastercoin 团队留下了深刻的印象，但这个提议过于激进，无法适应他们的发展路线图。

2013 年 12 月，Vitalik 开始分享一份白皮书，概述了以太坊背后的理念：图灵完备的通用区块链。 几十人看到了这个早期的草案并提供了反馈，帮助 Vitalik 改进了提案。

本书的两位作者都收到了白皮书的早期草稿并对其进行了评论。 Andreas M. Antonopoulos 对这个想法很感兴趣，并向 Vitalik 询问了许多关于使用单独的区块链来强制执行关于智能合约执行的共识规则以及图灵完备语言含义的问题。 Andreas 继续非常感兴趣地关注以太坊的进步，但是当时处在写“掌握比特币”一书的早期阶段，直到很久以后才直接参加以太坊。 然而，Gavin Wood 博士是第一批接触 Vitalik 并对他的 C++编程技能提供帮助的人之一。 Gavin 成为以太坊的联合创始人，编码者和 CTO。

正如 Vitalik 在他的“以太坊史前史”中所述：

这是以太坊协议完全是我自己创造的时候。 然而，从现在开始，新的参与者开始加入这一行列。 到目前为止，协议方面最突出的是 Gavin Wood ......

通过将以太坊作为构建可编程货币的平台，以及基于区块链的合同，可以保存数字资产并根据预先设定的规则将其转移到通用计算平台，Gavin 也可以在很大程度上归功于视觉的细微变化。这开始于重点和术语的微妙变化，随后越来越强调“Web3”集合，将以太坊视为一组分散技术的一部分，另外两个是 Whisper 和 Swarm，这种影响变得更强。

从 2013 年 12 月开始，Vitalik 和 Gavin 一起完善和改进了这个想法，共同构建了以太坊的协议层。

以太坊的创始人正在考虑没有特定目的的区块链，通过编程可以支持各种各样的应用程序。这个想法是通过使用像以太坊这样的通用区块链，开发人员可以编程他们的特定应用程序，而无需实现对等网络，区块链，共识算法等的底层机制。以太坊平台旨在抽象这些细节为分散式区块链应用程序提供了确定性和安全的编程环境。

就像 Satoshi，Vitalik 和 Gavin 不仅仅发明了一项新技术；他们以新颖的方式将新发明与现有技术相结合，并提供原型代码，向世界证明他们的想法。

创始人工作多年，建立和完善愿景。2015 年 7 月 30 日，第一个以太坊区块被开采。世界计算机开始为世界服务。

注意：	Vitalik Buterin 的文章“以太坊的史前史”于 2017 年 9 月出版，为以太坊最早的时刻提供了一个迷人的第一人称视角。 您可以在 https://vitalik.ca/general/2017/09/14/prehistory.html 上阅读。

以太坊的四个发展阶段

以太坊的发展计划分为四个不同的阶段，每个阶段都会发生重大变化。阶段中包括被称为“硬分叉”的方式，其以不向后兼容的方式改变功能。

四个主要的发展阶段是代号为 Frontier，Homestead，Metropolis 和 Serenity。 到目前为止(或计划)发生的中间硬分叉代号为 Ice Age,DAO,Tangerine Whistle,Spurious Dragon，Byzantium 和 Constantinople。 开发阶段和中间硬分叉都显示在以下时间轴上，该时间轴按块编号“标注日期”：

Block #0

Frontier—以太坊的初始阶段，持续时间为 2015 年 7 月 30 日至 2016 年 3 月。

Block #200,000

Ice Age—引入指数难度增加的硬分叉，以便在准备好时促使向 PoS 过渡。

Block #1,150,000

Homestead—以太坊的第二阶段于 2016 年 3 月推出。

Block #1,192,000

DAO—这是一个硬分叉，可以挽回被攻击的 DAO 合约的受害者损失，并导致以太坊和以太坊经典分成两个竞争系统。

Block #2,463,000

Tangerine Whistle—一个硬分叉，用于更改某些 I/O 操作的 gas 计算，并清除利用这些操作的低 gas 成本的拒绝服务（DoS）攻击的累积状态。

Block #2,675,000

Spurious Dragon—解决更多 DoS 攻击向量的硬分叉，以及另一个状态清除。另外，还有一种重放攻击保护机制。

Block #4,370,000

Metropolis Byzantium—大都会是以太坊的第三个阶段，目前正在撰写本书时，于 2017 年 10 月推出。拜占庭是大都会计划的两个硬叉中的第一个。

在拜占庭之后，还有一个为大都会计划的硬分叉：君士坦丁堡。大都会之后将是以太坊部署的最后阶段，代号为 Serenity。

以太坊：通用区块链

最初的区块链，即比特币的区块链，跟踪比特币的单位状态及其所有权。 您可以将比特币视为分布式共识状态机，其中交易导致全局状态转换，从而改变币的所有权。状态转换受到共识规则的约束，允许所有参与者在挖掘几个块之后（最终）收敛于系统的共同（共识）状态。

以太坊也是一个分布式状态机。但是，以太网不仅仅跟踪货币所有权状态，而是跟踪通用数据存储的状态转换，即可以保存任何可表示为键值元组的数据的存储。键值数据存储包含任意值，每个值由某个键引用;例如，键“Book Title”引用的值“Mastering Ethereum”。在某些方面，这与大多数通用计算机使用的随机存取存储器（RAM）的数据存储模型具有相同的目的。以太坊拥有存储代码和数据的内存，并使用以太坊区块链来跟踪内存随时间的变化情况。与通用存储程序计算机一样，以太坊可以将代码加载到其状态机中并运行该代码，将结果状态更改存储在其区块链中。与大多数通用计算机存在的两个重要区别是，以太坊状态变化受共识规则的约束，而状态则是全局分布的。以太坊回答了这个问题：“如果我们可以跟踪任意状态并对状态机进行编程以创建一个在共识下运行的全球计算机，该怎么办？”

以太坊的组件

在以太坊中，区块链组件中描述的区块链系统的组件具体为：

P2P 网络

以太坊在以太坊主网络上运行，可在 TCP 端口 30303 上寻址，并运行一个名为 *Ð∃ Vp2p* 的协议。

共识规则

以太坊的共识规则在参考规范黄皮书中定义（参见进一步阅读）。

交易

以太坊交易是包括（包括其他事项）发送者，接收者，价值和数据有效载荷的网络消息。

状态机

以太坊状态转换由以太坊虚拟机（EVM）处理，这是一个执行字节码（机器语言指令）的基于堆栈的虚拟机。 称为“智能合约”的 EVM 程序以高级语言（例如，Solidity）编写，并编译为字节码以在 EVM 上执行。

数据结构

以太坊的状态作为数据库（通常是 Google 的 LevelDB）本地存储在每个节点上，该数据库包含称为 Merkle Patricia Tree 的序列化散列数据结构中的事务和系统状态。

共识算法

以太坊使用比特币的共识模型，Nakamoto Consensus，它使用顺序单一签名块，由 PoW 加权重要性来确定最长链，从而确定当前状态。但是，有计划在不久的将来转向代号为 Casper 的 PoS 加权投票系统。

经济安全

以太坊目前使用一种名为 Ethash 的 PoW 算法，但最终将在未来的某个时刻转向 PoS。

客户端

以太坊有几种可互操作的客户端软件实现，其中最突出的是 Go-Ethereum（Geth）和 Parity。

进一步阅读

以下参考资料提供了有关此处提及的技术的其他信息：

- The Ethereum Yellow Paper: https://ethereum.github.io/yellowpaper/paper.pdf
- The Beige Paper, a rewrite of the Yellow Paper for a broader audience in less formal language: https://github.com/chronaeon/beigepaper
- ÐΞVp2p network protocol: http://bit.ly/2quAlTE
- Ethereum Virtual Machine list of resources: http://bit.ly/2PmtjiS
- LevelDB database (used most often to store the local copy of the blockchain): http://leveldb.org
- Merkle Patricia trees: https://github.com/ethereum/wiki/wiki/Patricia-Tree
- Ethash PoW algorithm: https://github.com/ethereum/wiki/wiki/Ethash
- Casper PoS v1 Implementation Guide: http://bit.ly/2DyPr3l
- Go-Ethereum (Geth) client: https://geth.ethereum.org/
- Parity Ethereum client: https://parity.io/

以太坊和图灵完备性

一旦你开始阅读以太坊，你就会立即遇到“图灵完备”这个词。他们说，以太坊与比特币不同，是图灵完备的。这到底是什么意思呢？

该术语指的是英国数学家阿兰·图灵，他被认为是计算机科学之父。1936 年，他创建了一个计算机的数学模型，该计算机由状态机组成，通过在顺序存储器上读取和写入符号来操纵符号（类似于无限长的纸带）。通过这种结构，图灵继续提供一个数学基础来回答（在否定的）有关通用可计算性的问题，这意味着是否所有问题都是可解决的。他证明了存在一些无法计算的问题。具体来说，他证明了停止问题（无论是否有可能，给定一个任意程序及其输入，以确定程序是否最终会停止运行）是不可解决的。

阿兰·图灵进一步将可以用于模拟任何图灵机的系统定义为图灵完备。这种系统称为通用图灵机（UTM）。

以太坊能够在称为以太坊虚拟机的状态机中执行存储程序，同时向内存读取和写入数据，使其成为图灵完备系统，因此成为 UTM。 考虑到有限存储器的限制，以太坊可以计算任何可以由任何图灵机计算的算法。

以太坊的突破性创新是将存储程序计算机的通用计算体系结构与分散的区块链相结合，从而创建分布式单状态（单例）世界计算机。以太坊计划“无处不在”，但却产生了一个由共识规则保护的共同状态。

图灵完整性作为“特征”

听到以太坊是图灵完备的，你可能会得出结论，这是一个在图灵不完整的系统中某种程度上缺乏的特征。实际上，它恰恰相反。 图灵完备性很容易实现；事实上，已知的最简单的图灵完备状态机有 4 个状态并使用 6 个符号，状态定义只有 22 个指令长。实际上，有时系统被发现是“意外地图灵完备”。这些系统的有趣参考可以在 http://bit.ly/2Og1VgX 找到。

但是，图灵完备性是非常危险的，特别是在公共区块链等开放式访问系统中，因为我们之前提到的暂停问题。例如，现代打印机是图灵完备的，可以给出打印文件，将它们发送到冻结状态。以太坊是图灵完备的事实意味着任何复杂程序都可以由以太坊计算。但这种灵活性带来了一些棘手的安全和资源管理问题。无响应的打印机可以关闭并再次打开。使用公共区块链是不可能的。

图灵完备性的含义

图灵证明，你无法通过在计算机上模拟程序来预测程序是否会终止。简单来说，我们无法在不运行程序的情况下预测程序的路径。图灵完备系统可以在“无限循环”中运行，这是一个术语（过度简化）用于描述不终止的程序。创建一个运行永不结束的循环的程序是微不

足道的。但是由于起始条件和代码之间的复杂交互，无意中永无止境的循环可以在没有警告的情况下出现。在以太坊中，这提出了一个挑战：每个参与节点（客户端）必须验证每个事务，运行它调用的任何智能合约。但正如图灵所证明的那样，以太坊无法预测智能合约是否将终止，或者它将运行多长时间而不实际运行（可能永远运行）。无论是偶然还是故意，都可以创建智能合约，使其在节点尝试验证时永远运行。这实际上是 DoS 攻击。当然，在一个需要一毫秒验证的程序和一个永远运行的程序之间是一系列令人讨厌的资源占用，内存膨胀，CPU 过热的程序，它们只会浪费资源。在世界计算机中，滥用资源的程序会滥用世界资源。如果无法预先预测资源使用情况，以太坊如何限制智能合约使用的资源？

为了应对这一挑战，以太坊引入了一种称为 gas 的计量机制。 当 EVM 执行智能合约时，它会仔细考虑每条指令（计算，数据访问等）。 每个指令具有以 gas 为单位的预定成本。当交易触发智能合约的执行时，它必须包含一定数量的 gas，用于设定运行智能合约所消耗的内容的上限。 如果计算消耗的 gas 量超过交易中可用的 gas，则 EVM 将终止执行。gas 是以太坊用来允许图灵完备计算同时限制任何程序可以消耗的资源的机制。

接下来的问题是，“如何在以太坊世界计算机上获得计算费用？” 你不会在任何交易所找到 gas。 它只能作为交易的一部分购买，并且只能用以太币购买。 以太网需要与交易一起发送，并且需要明确指定用于购买 gas 以及可接受的 gas 价格。就像在泵上一样，gas 的价格也不固定。 为交易购买 gas，执行计算，并将任何未使用的 gas 退还给交易的发送方。

从通用区块链到分散应用（DApps）

以太坊开始作为一种制作通用区块链的方法，该区块链可以被编程用于各种用途。但很快，以太坊的愿景扩展到成为 DApps 编程的平台。DApps 代表了比智能合约更广泛的视角。DApp 至少是一个智能合约和一个 Web 用户界面。更广泛地说，DApp 是一个基于开放，分散的点对点基础设施服务构建的 Web 应用程序。

DApp 至少由以下组成：

区块链上的智能合约

Web 前端用户界面

此外，许多 DApps 还包括其他分散的组件，例如：

分散式（P2P）存储协议和平台

分散式（P2P）消息传递协议和平台

提示 你可能会看到 DApp 拼写为 Ð Apps。 Ð 字符是拉丁字符，称为“ETH”，暗指以太坊。要显示此字符，请使用 Unicode 代码点 0xD0，或者必要时使用 HTML 字符实体 eth（或十进制实体＃208）。

互联网的第三个时代

2004 年，“Web 2.0”这个术语变得突出，描述了 Web 向用户生成的内容，响应式界面和交互性的演变。Web 2.0 不是技术规范，而是描述 Web 应用程序新焦点的术语。

DApps 的概念旨在将万维网带入其下一个自然进化阶段，将对等协议的分散化引入 Web 应用程序的各个方面。用于描述这种演变的术语是 web3，意思是网络的第三个“版本”。Web3 首先由 Gavin Wood 博士提出，它代表了 Web 应用程序的新愿景和重点：从集中拥有和托管应用程序到基于分散协议的应用程序。

在后面的章节中，我们将探索以太坊 web3.js JavaScript 库，它将浏览器中运行的 JavaScript 应用程序与以太坊区块链联系起来。web3.js 库还包括一个名为 Swarm 的 P2P 存储网络接口和一个名为 Whisper 的 P2P 消息服务。通过在 Web 浏览器中运行的 JavaScript 库中包含这三个组件，开发人员可以使用完整的应用程序开发套件来构建 web3 DApp。

以太坊的发展文化

到目前为止，我们已经讨论过以太坊的目标和技术与之前的其他区块链（如比特币）的区别。以太坊也有着截然不同的发展文化。

在比特币中，开发遵循保守原则：所有变更都经过仔细研究，以确保没有任何现有系统中断。在大多数情况下，只有在向后兼容时才会实施更改。允许现有客户选择加入，但如果他们决定不升级，则会继续运营。

相比之下，在以太坊，社区的发展文化关注的是未来而非过去。（并非完全严重）的口头禅是“快速行动并打破局面”。如果需要进行更改，则会实现更改，即使这意味着使先前的假设无效，破坏兼容性或强制客户端更新。以太坊的发展文化的特点是快速创新，快速发展，并愿意部署前瞻性改进，即使这是以牺牲一些向后兼容性为代价的。

作为开发人员，这对您意味着您必须保持灵活性并准备好重建您的基础架构，因为一些基本假设会发生变化。以太坊开发人员面临的一大挑战是将代码部署到不可变系统与仍在发展的开发平台之间固有的矛盾。你不能简单地“升级”智能合约。您必须准备好部署新的，迁移用户，应用程序和资金，然后重新开始。

具有讽刺意味的是，这也意味着构建具有更多自主权和更少集中控制的系统的目标仍未完全实现。自主权和分散化要求平台在未来几年内在以太网中获得的稳定性要高一些。为了

“发展”平台，您必须准备好废弃并重新启动智能合约，这意味着您必须对它们保持一定程度的控制。

但是，从积极的方面来看，以太坊正在快速前进。“自行车脱落”的机会很少，这意味着通过争论一些细节，例如如何在核电站后面建造自行车棚来阻碍发展。如果你开始自行车脱落，你可能会突然发现，当你分心的时候，开发团队的其他人改变了计划并放弃了自行车而转向了自主气垫船。

最终，以太坊平台的开发将变慢，其界面将变得固定。但与此同时，创新是驱动原则。你最好跟上，因为没有人会为你减速。

为什么要学习以太坊？

区块链的学习曲线非常陡峭，因为它们将多个学科组合成一个领域：编程，信息安全，密码学，经济学，分布式系统，点对点网络等。以太网使这一学习曲线不那么陡峭，所以你可以快速入门。但是，在一个看似简单的环境表面之下还有更多。当你学习并开始更深入地思考时，总会有另一层复杂性和奇迹。

以太坊是学习区块链的绝佳平台，它正在建立一个庞大的开发者社区，比任何其他区块链平台都要快。最重要的是，以太坊是开发人员为开发人员构建的区块链。熟悉 JavaScript 应用程序的开发人员可以进入以太坊并开始非常快速地生成工作代码。在以太坊生命的最初几年，通常会看到 T 恤宣布你可以用五行代码创建一个令牌。当然，这是一把双刃剑。编写代码很容易，但编写好的和安全的代码非常困难。

本书将教会你什么

这本书潜入以太坊并检查每个组成部分。您将从一个简单的交易开始，剖析其工作原理，建立一个简单的合约，使其更好，并跟随其在以太坊系统中的旅程。

您不仅将学习如何使用以太坊-它是如何工作的-而且还将学习它为何如此设计。您将能够理解每个部分的工作原理，以及它们如何组合在一起以及为什么。

第二章 以太坊基础知识

以太坊基础知识

在本章中，我们将研究以太坊，学习如何使用钱包，如何创建交易，以及如何运行基本的智能合约。

以太币单位

以太坊币称为以太，通常情况下称为“ETH”，或符号 Ξ （来自希腊字母“Xi”，看起来像一个程式化的大写字母 E），或者♦；列如，1eth，或者 1Ξ ，或者♦1。

以太被分割为更小的单位，分割到的最小单位称为“wei”。一个以太是 1 千万亿“wei”（1*1018 或 1,000,000,000,000,000,000）。你可能也听到别人提及 “以太坊”货币，但这是初学者常见的的一个错误。以太坊是系统，以太是通证。

以太的数值用以“wei”为代表的无符号整数值。当您交易 1 以太时，支付系统会将其编码为 1000000000000000000 “wei”数值。

以太的不同面值可使用国际科学单位（SI）的和俗称表示，在此向伟大的统计学和密码学思想致敬。

以太名称和单位名称代表了不同面值，体现为口语（俗称）名称和 SI 名称。为了与内部值的表示保持一致，该表显示了 wei 中的所有面额（第一行），在第 7 行中以太为 1018wei。

表 1. 以太面值和单位名称

面值（wei）	指数	俗称	SI 名称
1	1	wei	Wei
1，000	10^3	Babbage	Kilowei or femtoether
1,000,000	10^6	Lovelace	Megawei or picoether
1,000,000,000	10^9	Shannon	Gigawei or nanoether
1,000,000,000,000	10^{12}	Szabo	Microether or micro
1,000,000,000,000,000	10^{15}	Finney	Milliether or milli
1,000,000,000,000,000,000	10^{18}	Ether	Ether
1,000,000,000,000,000,000,000	10^{21}	Grand	Kiloether
1,000,000,000,000,000,000,000,000	10^{24}		Megaether

挑选以太坊钱包

“钱包”这个词有很多含义，虽然这些含义都是有联系而且是平时中几乎相同的东西。我们将使用“钱包”这个词来表示软件应用程序，这个程序可帮助您管理以太坊帐户。简而言之，以太坊钱包是您通往以太坊系统的门户。它包含您的密钥，可以代表您创建和广播交易。选择以太坊钱包可能比较难，因为有许多不同的选项，不同的功能和设计。有些更适合初学者，有些更适合专家。以太坊平台本身仍在不断改进，最好的钱包是伴随以太坊平台升级，钱包也会升级。

但别担心！如果你选择了钱包，但不喜欢它的使用方式 - 或者如果你最初喜欢它，但后来想尝试别的东西 - 你可以很轻松地更换钱包。您所要做的就是进行一次交易，将您的资金从旧钱包发送到新钱包，或者导出您的私钥并将其导入新的私钥。

我们选择了三种不同类型的钱包作为本书的例子：手机钱包，桌面钱包和网页的钱包。之所以选择这三个钱包，因为它们体现了综合性和特征性。但是，选这些钱包不是对其品质或安全性的认可，而是它们是演示和测试的好例子。

请记住，要使钱包应用程序正常工作，它必须能够接入您的私钥，因此最重要的是从您信任的资源下载和使用钱包应用程序。一般来说，钱包应用程序越流行，它就越可靠。尽管如此，最好避免“将所有鸡蛋放在一个篮子里”，而是让你的以太坊帐户分散在几个钱包中。

以下是一些不错的首选钱包：

MetaMask：

MetaMask 是一个浏览器扩展钱包，可在您的浏览器（Chrome，Firefox，Opera 或 Brave Browser)中运行。它易于使用且便于测试，因为它能够连接到各种以太坊节点和测试区块链。MetaMask 是一个基于网页的钱包。

Jaxx：

Jaxx 是一个多平台和多币种钱包，可在各种操作系统上运行，包括 Android，iOS，Windows，macOS 和 Linux。对于新用户而言，它通常是一个不错的选择，因为它的设计简单易用。Jaxx 可以是移动或桌面钱包，具体取决于您安装它的位置。

MyEtherWallet（MEW）：

MyEtherWallet 是一个基于网页的钱包，可以在任何浏览器中运行。它具有我们将在许多示例中探索的多种复杂功能。MyEtherWallet 是一个基于网页的钱包。

Emerald Wallet：

Emerald Wallet 旨在与以太坊 Classic 区块链配合使用，但与其他基于以太坊的区块链兼容。它是一个开源桌面应用程序，可在 Windows，macOS 和 Linux 下运行。Emerald Wallet 可以运行完整节点或连接到公共远程节点，以“简捷”模式工作。它还有一个配套工具，可以从命令行执行所有操作。

控制管理

因为以太坊是一个分布式运行的系统，所以作为开源性的区块链有很大意义。这有很多含义，其中一个关键的方面是，以太坊的每个用户都可以通过掌控私钥，控制对资金和智能合约的访问。我们把获得资金和智能合约的组合称为“账户”或“钱包”。这些术语的功能可能会非常复杂，因此我们稍后会详细介绍。基本原则是它就像一个私钥等于一个“帐户”一样简单。有些用户选择通过使用第三方保管人（例如在线交易所）来放弃对其私钥的控制。在本书中，我们将教您如何控制和管理自己的私钥。

控制承担了很大的责任。如果您丢失了私钥，则会失去对您的资金和合约的访问权限。没有人可以帮助您重新获得访问权限 – 您的资金将被永久锁定。以下是一些帮助您管理此责任的提示：

不使用简易安全方式，要使用经过验证和测试的方法。

- 账户越重要（例如，管理的资金价值越高，或智能合约越重要），越采取更高级别的安全措施。
- 最高安全级别是断网设备，但并非每个帐户都需要此级别。
- 切勿以简单方式存储您的私钥，尤其是以数字方式存储。幸运的是，今天的大多数用户界面甚至不会让您看到原始私钥。
- 私钥可以以加密形式存储，作为数字“密钥库”文件。加密后，他们需要密码才能解锁。当系统提示您选择密码时，请将其设置为强（即长且随机），备份密码，不要共享密码。如果您没有密码管理器，请将其记下并存放在安全且保密的地方。要访问您的帐户，您需要密钥库文件和密码。
- 不要将密码存储在数字文档，数码照片，屏幕截图，在线驱动器，加密的 PDF 等中。再者不要即兴创作，要使用密码管理器或笔和纸。
- 当系统提示您备份密钥、助记词序列时，请使用笔和纸进行物理备份。不要把这项任务“留待以后”；你会忘记的。这些备份可用于重建您的私钥，以防您丢失系统中保存的所有数据，或者忘记或丢失密码。但是，它们也可以被攻击者用来获取您的私钥，因此绝不要以数字方式存储它们，并将物理副本安全地存放在锁定的抽屉或保险箱中。

- 在转移任何大额（特别是新地址）之前，首先进行小的测试交易（例如，小于 1 美元的价值）并等待确认收货。
- 创建新帐户时，首先只向新地址发送一个小的测试交易。收到测试交易后，请尝试从该帐户再次发回。创建帐户可能会出错的原因有很多，防止出现问题，最好用一笔小资金来测试。如果测试工作顺利，一切都很好。
- 区块链浏览器是一种独立查看交易是否已被网络接受的简单方法。虽然很便捷，但这会对您的隐私产生负面影响，因为会通过区块链浏览器泄露您的地址，可以以此来追踪你。
- 不要向本书中显示的任何地址汇款。密钥列在书中，有人会立即拿走这笔钱。

现在我们已经介绍了密钥管理和安全性的一些基本操作，让我们开始使用 MetaMask！

MetaMask 入门

打开 Google Chrome 浏览器并导航至 https://chrome.google.com/webstore/category/extensions。

搜索“MetaMask”并单击狐狸的徽标。您应该会看到类似于 MetaMask Chrome 扩展程序的详细信息页面中显示的结果。

图 1. MetaMask Chrome 扩展的详细信息页面

验证您正在下载的 MetaMask 程序是否为正版非常重要，因为有时候人们可以通过 Google 的浏览器推送恶意软件。正确的是：

在地址栏中显示 ID nkbihfbeogaeaoehlefnkodbefgpgknn

由 https://metamask.io 提供

有超过 1,400 条评论

拥有超过 1,000,000 名用户

确认您正在查看正确的程序后，请点击“添加到 Chrome”进行安装。

创建钱包

安装 MetaMask 后，在浏览器的工具栏中看到一个新图标（狐狸头）。点击它开始，选择接受条款和条件，然后通过输入密码创建新的以太坊钱包（请参阅 MetaMask Chrome 扩展程序的密码页面）。

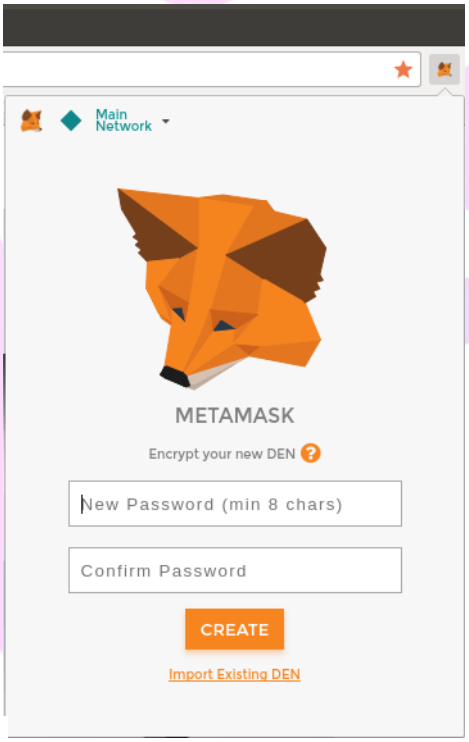


图 2. MetaMask Chrome 扩展的密码页面

小技巧：掌握密码才可对 MetaMask 的访问，因此任何访问您浏览器的人都无法使用它。

设置密码后，MetaMask 将为您生成一个钱包，显示由 12 个英语单词组成的助记符，并备份（请参阅由 MetaMask 创建的钱包的助记符备份）。如果 MetaMask 或您的计算机出现问题，可以在任何兼容的钱包中使用这些单词来恢复对您资金的访问。您不需要此恢复的密码；这 12 个字就足够了。

小技巧：在纸上备份你的助记符（12 个单词）两次。将两个纸张备份存放在两个单独的安全位置，例如防火保险箱，锁定抽屉或保险箱。将纸质备份视为您在以太坊钱包中存储的等值现金。任何能够访问这些单词的人都可以访问并窃取您的资金。

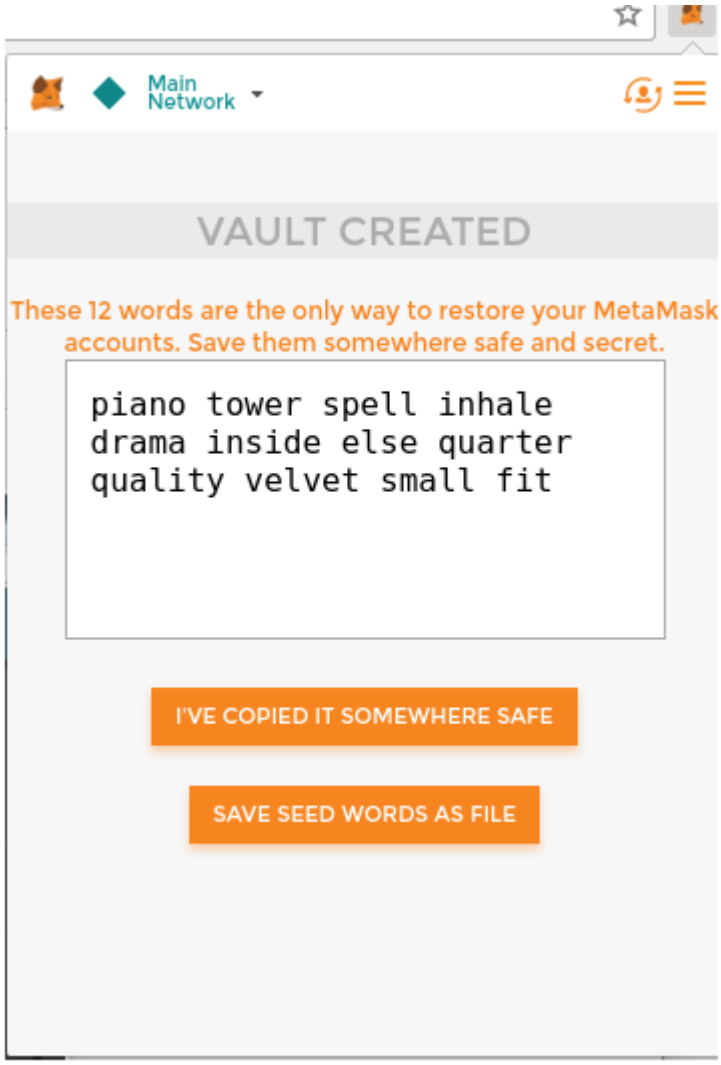


图 3. 由 MetaMask 创建的钱包的助记符备份

确认您已安全存储助记符后，您将能够看到以太坊帐户的详细信息，如 MetaMask 中的以太坊帐户所示。

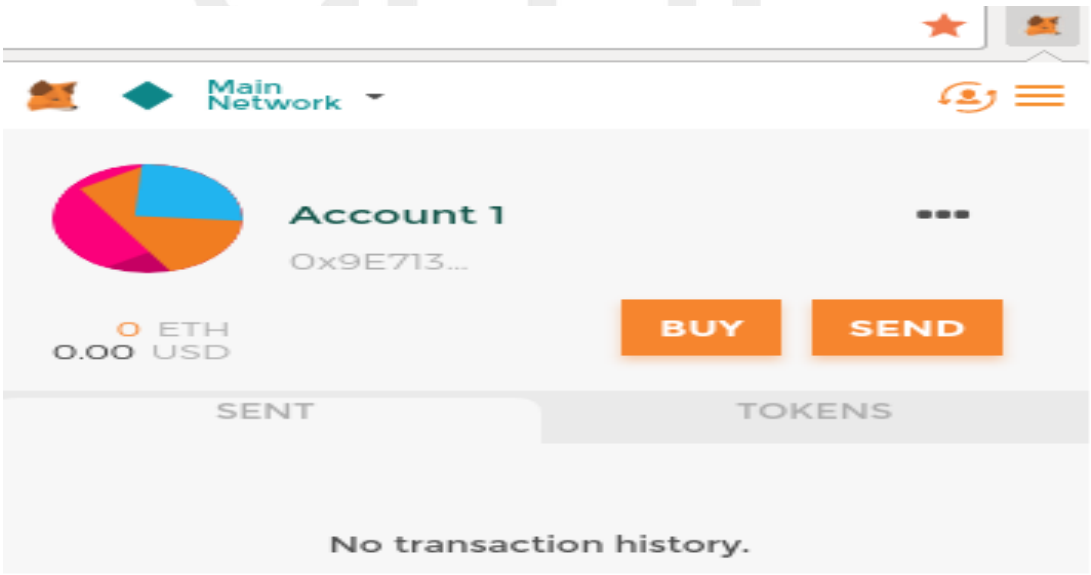


图 4. MetaMask 中的以太坊帐户

帐户页面显示您的帐户名称（默认情况下为“帐户 1”），以太坊地址（示例中为 0x9E713...）以及彩色图标，可帮助您直观地将此帐户与其他帐户区分开来。在帐户页面的顶部，您可以看到当前正在交易的以太（示例中的“主网络”）。

恭喜！您已经设置了第一个以太坊钱包。

交换网络

正如您在 MetaMask 帐户页面上看到的，您可以在多个以太坊网络之间进行选择。默认情况下，MetaMask 将尝试连接到主网络。其他选择是公共测试网、任意以太坊节点或在您自己的计算机上运行的区块链的节点（localhost）：

以太坊主网络

以太坊公链就是 ETH，真正拥有价值，并且非常重要。

Ropsten 测试网络

以太坊测试公链网络。该网络上的 ETH 没有任何价值。

Kovan 测试网络

以太坊测试公链网络使用 Aura 共识协议和权威证明（联合签名）。该网络上的 ETH 没有任何价值。Kovan 测试网络仅由 Parity 支持。其他以太坊用户稍后提出的 Clique 共识协议来证明基于授权的验证。

Rinkeby 测试网络

以太坊公共测试区块链和网络，使用 Clique 共识协议和权威证明（联合签名）。该网络上的 ETH 没有任何价值。

本地主机 8545

连接到与浏览器在同一台计算机上运行的节点。该节点可以是任何公共区块链（main 或 testnet）或私有 testnet 的一部分。

自定义 RPC

允许您使用 Geth 兼容的远程过程调用（RPC）接口将 MetaMask 连接到任何节点。该节点可以是任何公共或个人区块链的一部分。

获得一些测试以太

你的首要任务是创建存储钱包。你不用在主网络上这样做，因为真正的以太成本很高，并且处理它需要更多的支出。现在，你将用一些测试以太加载你的钱包。

将 MetaMask 切换到 Ropsten 测试网络。单击“购买”，然后单击“Ropsten 测试网”。MetaMask 将打开一个新的网页，如 MetaMask Ropsten Test Faucet 所示。

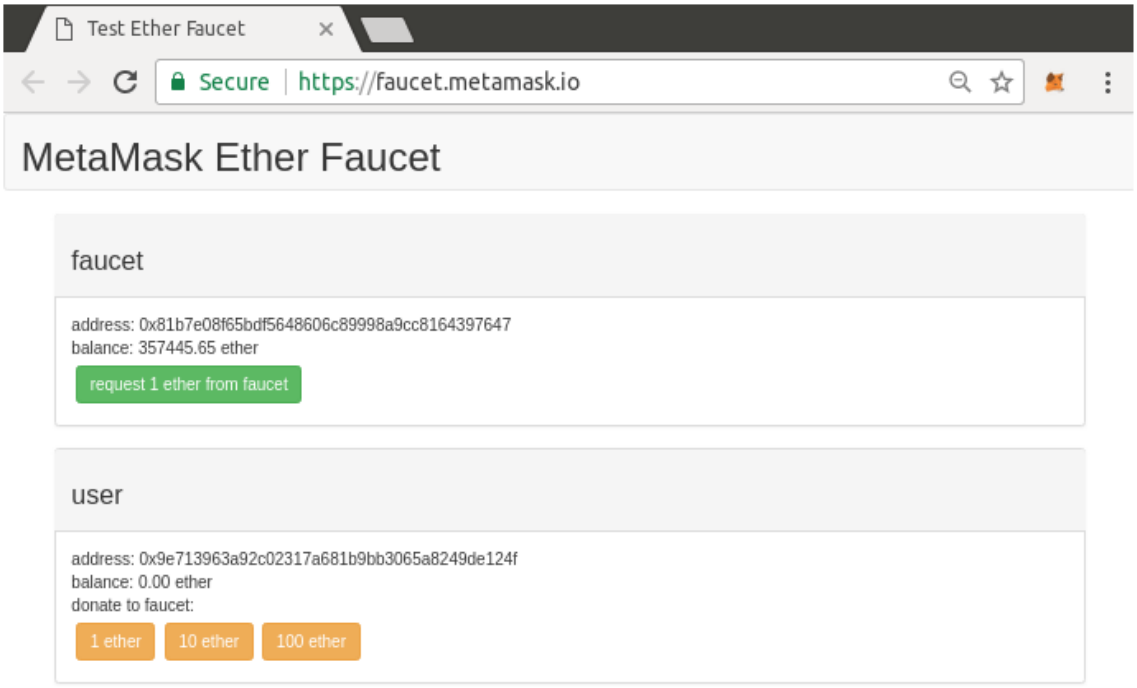


图 5. MetaMask Ropsten 测试网

您可能会注意到该网页已包含您的 MetaMask 钱包的以太坊地址。MetaMask 将启用了以太坊的网页与您的 MetaMask 钱包集成，并可以在网页上“查看”以太坊地址，例如，您可以将付款发送到显示以太坊地址的在线商店。如果页面需要 MetaMask 还可以使用您自己的钱包地址填充网页作为收件人地址。在此页面中，水龙头应用程序通过 MetaMask 向测试钱包地址发送以太。

单击绿色“需求 1 以太接口”按钮。您将在页面的下半部分看到一个交易 ID。开关软件创建了一笔交易 - 向您付款。支付 ID 如下所示：

```
0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57
```

在几秒钟内，新交易将由 Ropsten 矿工开采，您的 MetaMask 钱包将显示 1 ETH 的余额。单击支付 ID，您的浏览器将带您进入区块链浏览器，这是一个可浏览区块、地址和交易的网站。MetaMask 使用 Etherscan 区块链浏览器，这是一个比较流行的以太坊块区块浏览器。包含 Ropsten 测试接口付款的交易显示在 Etherscan Ropsten 区块浏览器中。

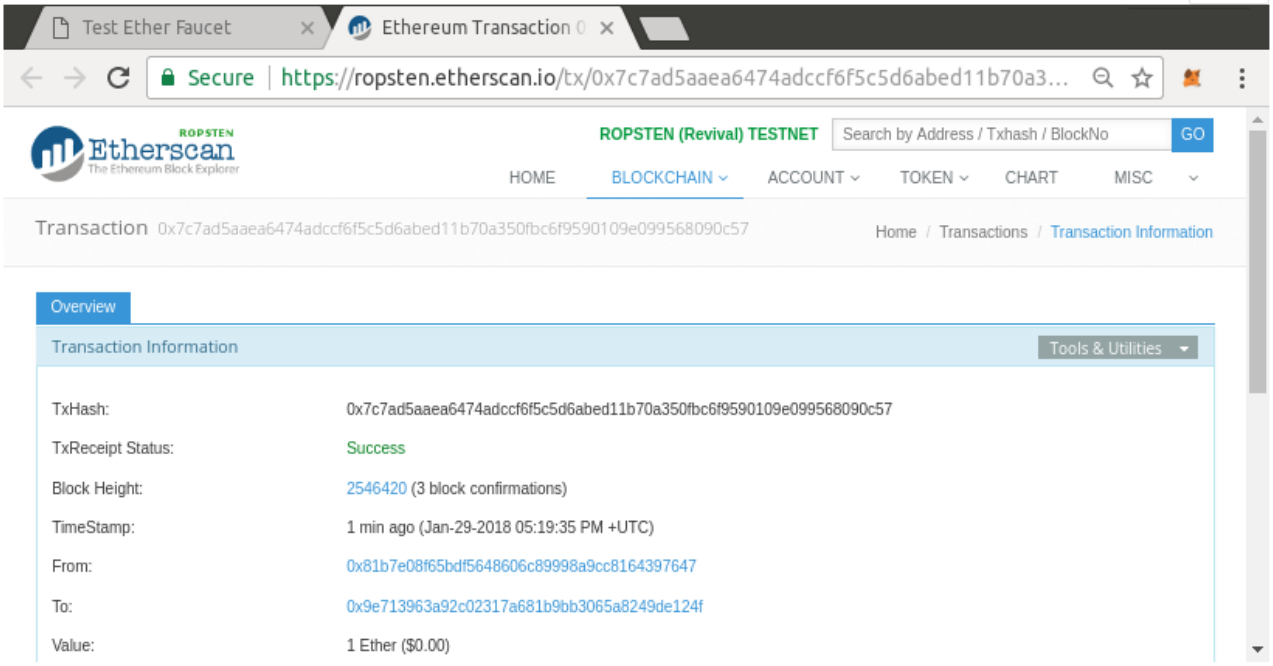


图 6. Etherscan Ropsten 块资源管理器

该交易已记录在 Ropsten 区块链上，任何人都可以随时查看，只需搜索交易 ID 或访问链接即可。

尝试访问该链接，或将支付哈希值输入 ropsten.etherscan.io 网站，即可查看。

从 MetaMask 发送以太网

一旦你从 Ropsten 测试接口收到你的第一个测试以太网，你可以尝试通过尝试将一些回到接口来发送以太。正如您在 Ropsten 测试接口页面上看到的那样，可以选择“发送”1 个

ETH 到接口。此选项可用性以至一旦您完成测试就可以退回剩余部分测试以太，这样其他人可以使用它。尽管有些人囤积它，但是测试以太没有价值，这使得很多人都不怎么用测试网络。因此囤积测试以太不受欢迎！

幸运的是，我们不是测试以太囤积者。单击橙色“1 ether”按钮告诉 MetaMask 创建 1ether 的交易支付接口。MetaMask 将准备一个支付并弹出一个带有确认的窗口，如向接口发送以太网中所示。

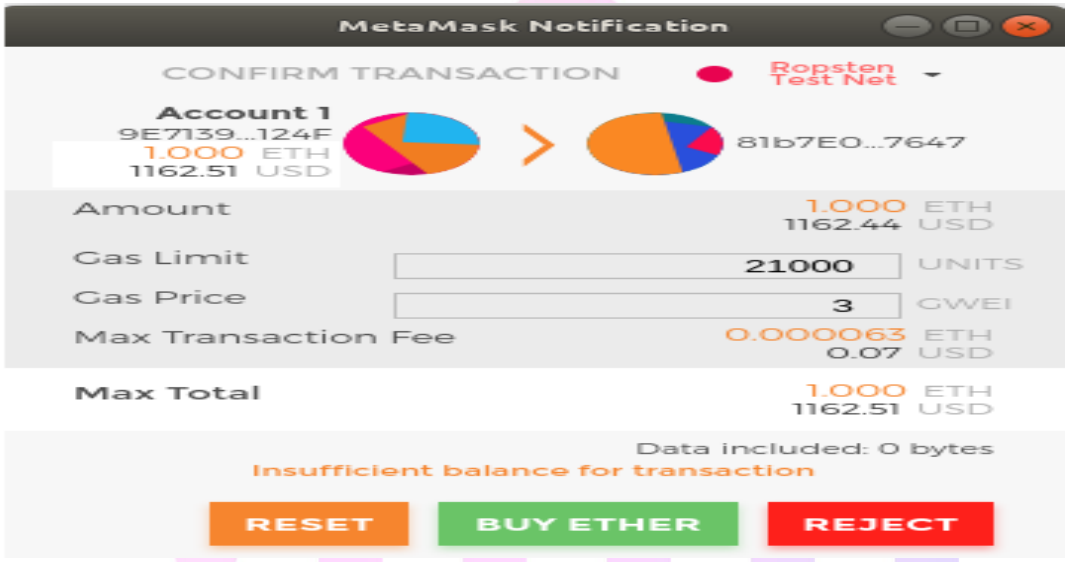


图 7. 向水龙头发送 1 个以太网

您可能已经注意到无法完成交易 - MetaMask 表示您的余额不足。乍一看，这可能会令人困惑：你有 1 个 ETH，你想发送 1 个 ETH，那么为什么 MetaMask 说你没有足够的资金呢？

答案是因为 gas 的成本。每笔以太坊交易都需要支付费用，矿工会收取费用以验证交易。以太坊的支付费用是虚拟货币 gas。作为交易的一部分，您使用以太币支付 gas 费用。

注意：测试网络也需要费用。如果没有费用，测试网络的表现将与主网络不同，使其成为一个不充分的测试平台。费用还保护测试网络免受 DoS 攻击和构造不良的合约（例如，无限循环），与保护主网络一样。

当您进行交易时，MetaMask 计算出最近成功交易的平均 gas 价格为 3 gwei，代表 gigawei。我们在以太币货币组成中所讨论的，wei 是以太币的最小单位。gas 限额是以发送交易为成本设定的，即 21,000 个 gas 单位。因此，您将花费的最大 ETH 值为 3 * 21,000 gwei = 63,000 gwei = 0.000063 ETH。（请注意，平均 gas 价格可能会波动，因为它们主要由矿工决定。我们将在后面的章节中看到如何增加/减少您的 gas 限制，以确保您的交易在需要时优先。）

所有这些都说：1 ETH 交易成本为 1.000063 ETH。当显示总数时，MetaMask 会将其降低到 1 ETH，但实际需要的数量是 1.000063 ETH，并且您只有 1 个 ETH。单击“拒绝”可取消此支付。

让我们再测试一下以太！再次单击绿色“从水龙头要求 1 以太”按钮并等待几秒钟。别担心，水龙头应该有足够的以太，如果你需要的话会给你更多。

你有 2 个 ETH 的余额，你可以再试一次。单击橙色的“1 以太”传输按钮，这次有足够的余额来完成交易。当 MetaMask 弹出付款窗口时，单击“提交”。在完成所有这些操作之后，你的余额为 0.999937 ETH，因为你发送了 1 个 ETH，具有 0.000063 ETH 费用。

探索地址的交易历史

到目前为止，你已熟练使用 MetaMask 发送和接收测试以太。你的钱包已收到至少两笔付款，且发送一次。您可以使用 ropsten.etherscan.io 区块链浏览器查看所有这些交易。复制钱包地址并将其粘贴到块浏览器的搜索框中，也可以让 MetaMask 为您打开页面。在 MetaMask 中的帐户图标旁边，您会看到一个显示三个点的按钮。单击它以显示与帐户相关的选项菜单（请参阅 MetaMask 帐户上下文菜单）。

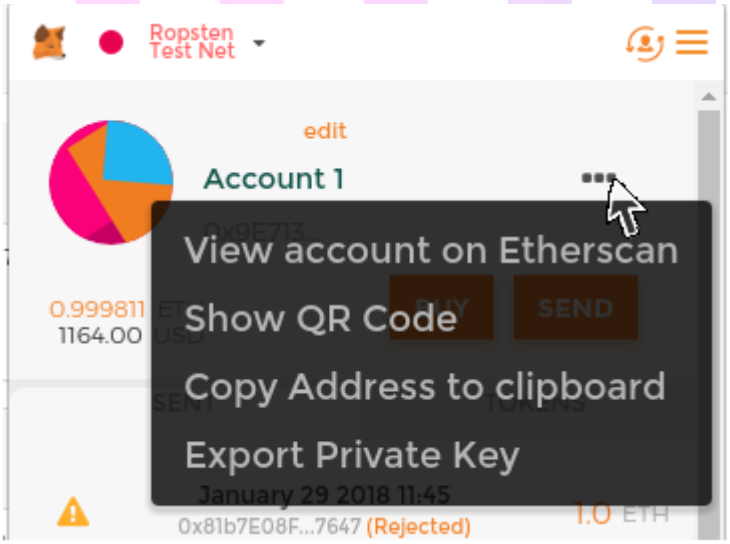


图 8. MetaMask 帐户上下文菜单

选择“在以太上查看帐户”，在区块链浏览器中打开帐户交易历史记录，如 Etherscan 上的地址交易历史记录中所示。

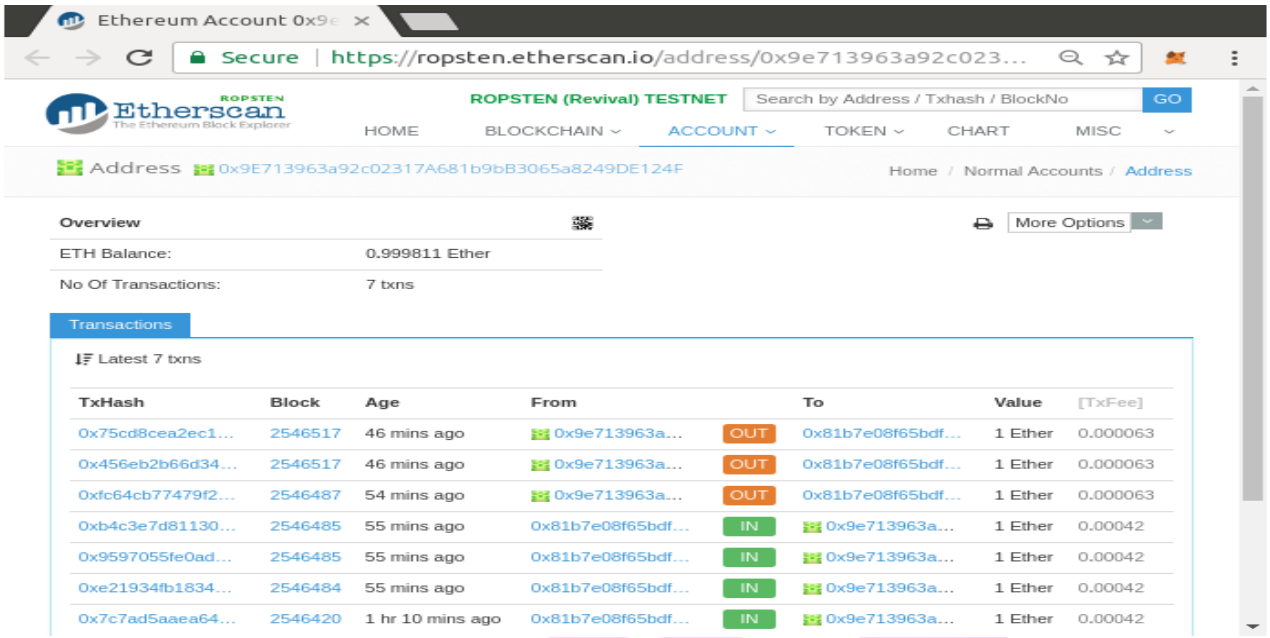


图 9. Etherscan 上的地址交易历史记录

在这可以看到以太坊地址的整个交易历史记录。它显示了 Ropsten 区块链中记录的所有交易，包括发件人或收件人地址。单击其中一些交易即可查看更多信息。

您可以浏览任何地址的交易历史记录。看一下 Ropsten 测试地址的交易历史记录（提示：它是您地址最早支付中列出的“发件人”地址）。您可以看到从水龙头向你和其他地址发送的所有测试以太。每笔交易你都可以发现更多地址和交易记录。不久之后，互联数据将令你眼花缭乱。公共区块链包含大量信息，所有这些信息都可以通过编程方式进行研究，我们将在后面的例子中看到。

介绍全球分布式计算机

你现在已经创建了一个钱包，且发送和接收了以太网。目前，我们已将以太坊视为加密货币，但以太坊更多含义。加密货币从属于作为全球化分布式计算的以太坊。智能合约是在以太坊虚拟机（EVM）的上运行的计算机程序，以太被用于支付运行智能合约的费用。

EVM 在整体可以独立运行。以太坊网络上的每个节点都运行 EVM 的本地副本、验证合约。当处理交易和智能合约时，区块会记录此全球分布式计算机变更。我们将更详细地讨论在[evm_chapter]中这个问题。

外部账户（EOA）与合约

您在 MetaMask 钱包中创建的帐户类型称为外部帐户（EOA）。外部账户是私钥对应的账户;私钥就是控制对资金或合约的访问。还有其他类型的帐户，另一种帐户是合约帐户。合约帐户具有智能合约代码，EOA 没此功能。此外，合约帐户没有私钥。它由其智能合约代码的逻辑所掌控：合约账户创建后，由以太坊区块链的软件程序记录，并由 EVM 运行。

合约具有地址，就像 EOA 一样。合约也可以发送和接收以太，就像 EOA 一样。但是，当交易目标是合约地址时，将使该合约在 EVM 中运行，运行交易后会记录交易数据。除了以太，交易包含合约中运行的特定功能的数据，并传递给该数据。通过这种交易方式可运行智能合约。

请注意，由于合约帐户没有私钥，因此无法进行交易。只有 EOA 才能交易，但可以通过调用其他合约，构建复杂的执行路径来对交易作出反馈。这种情况的典型用途是 EOA 向多重签名智能合约钱包发送交易请求，将一些 ETH 发送到另一地址。典型的 DApp 编程模式是让 A 与 B 建立协约，以便与 A 合约的用户之间共享。

在接下来的几节中，我们将编写第一份合约。然后，您将学习如何使用 MetaMask 钱包创建，保存，使用该合约，并在 Ropsten 网络上测试以太。

一个简单的合约：测试以太水龙头

以太坊有许多不同的高级语言，所有这些语言都可用于编写合约并生成 EVM 字节码。您可以在《高等语言》中阅读许多最着名和最有趣的内容。到目前为止，高级智能合约编程语言的首要选择是：Solidity。Solidity 由本书的作者 Gavin Wood 博士创建，并已成为以太坊（及其他）中最广泛使用的语言。我们将使用 Solidity 编写我们的第一份合约。

我们的第一个例子（Faucet.sol：运行水龙头的 Solidity 合约），我们将编写一份运行水龙头的合约。您已经在 Ropsten 测试网络上使用了一个水龙头来测试以太网。水龙头是一个相对简单的东西：它向任何要求的地址发出以太，并且可以定期填充。可以将水龙头看作被个人或者服务器控制的钱包。

示例 1. Faucet.sol：运行水龙头的 Solidity 合约

```
link:code/Solidity/Faucet.sol[]
```

提示：

您将在本书的 GitHub 存储库的代码子目录中找到本书的所有代码示例。

具体而言，我们的 Faucet.sol　合约是：code/Solidity/Faucet.sol

这是一个非常简单的合约，尽可能做到了最简单。它也是一个有缺陷的合约，出现了一些不良行为和安全漏洞。我们将来学习检查后面部分的漏洞。但就目前而言，让我们逐行了解这份合约的用途及其运作方式。您很快就会注意到 Solidity 的许多元素与现有的编程语言类似，例如 JavaScript，Java 或 C ++。

第一行是评论：

//我们的第一份合约是一个水龙头！

注释供人阅读且不包含可执行 EVM 字节码。我们通常在我们试图解释的代码之前将它们放横线上，或者有时在同一横线上。注释以两个正斜杠开头：//。从第一个斜杠到该行结束的所有内容都被视为空行并被忽略。

下一行是我们的合约开始的地方：

水龙头合约

此行对合约进行了声明，类似于其他面向对象语言中的声明。合约规定范围包括花括号（{}）之间的所有内容，它们规定范围与其他编程语言中使用花括号一样。

接下来，我们宣布水龙头的第一个智能合约：

提现功能（uint withdraw_amount）public {

该功能名为体现，它取自一个名为提现金额的参数。它有共享功能，可以被其他合约调用。提现功能的第一部分就设定了提现限制：

require（withdraw_amount <= 100000000000000000）;

它内置的 Solidity 的功能是前提条件，即提现金额小于或等于 100,000,000,000,000,000 wei，这是 ether 的基本单位（参见以太币和单位名称），相当于 0.1 以太。如果提现大于该账户提现上限，则此处的要求将导致合约停止执行，因异常而失败。注意，编码语言需要在 Solidity 中以分号结束。

这部分合约是我们龙头的主要逻辑。它通过限制提现来控制合约之外的资金流动。这是一个非常简单的控制，但可以让你发现区块链编程的力量：储存资金的分布式软件。

接下来是实际提现：

msg.sender.transfer(提现金额);

这里发生了一些有趣的事情。Msg 的目的是所有合约的访问入口。它表示触发此合约执行的交易。表示发送者是交易的发件人地址。传输功能是一个内置功能，它将以太从当前合约转移到发件人的地址。继续看，这意味着触发此合约执行的 msg 的将转账给发送者。

下一行是结束大括号，表示我们的提现功能的结束。

接下来，我们再宣布一个功能：

此功能是回退或默认功能，即触发合约的交易是未命名合约，或无此功能或未包含数据。合约默认功能是接收以太。因为它可以接受以太合约，所以被认为具有共享和应付职能。除了接受以太之外，它没有做任何事情，如花括号（{}）中的定义所示。这个功能可处理将 ether 发送到合约地址的交易，与钱包一样。

在我们的默认功能下面是最后的结束大括号，它关闭了合约 Faucet 的定义。

编译水龙头合约

现在我们有了第一个示例合约，我们需要使用 Solidity 编译器将 Solidity 代码转换为 EVM 字节码，以便它可以由区块链本身的 EVM 执行。

Solidity 编译器作为独立的可执行文件提供，作为各种框架的一部分，并捆绑在集成开发环境（IDE）中。为了方便起见，我们将使用一种比较流行的 IDE，称为 Remix。

使用您的 Chrome 浏览器（使用之前安装的 MetaMask 钱包）导航到 `https://remix.ethereum.org` 上的 **Remix IDE** 。

当您第一次加载 Remix 时，它将以一个名为 ballot.sol 的示例合约开始。我们不需要它，所以通过单击选项卡一角的 x 来关闭它，如关闭默认示例选项卡中所示。

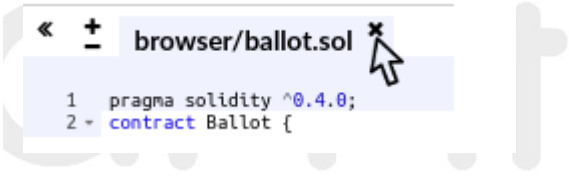


图 10. 关闭默认示例选项卡

现在，通过单击左上角工具栏中的圆形加号添加新选项卡，如单击加号以打开新选项卡中所示。将新文件命名为 Faucet.sol。

图 11. 单击加号以打开新选项卡

打开新选项卡后，复制并粘贴示例 Faucet.sol 中的代码，如将 Faucet 示例代码复制到新选项中所示。

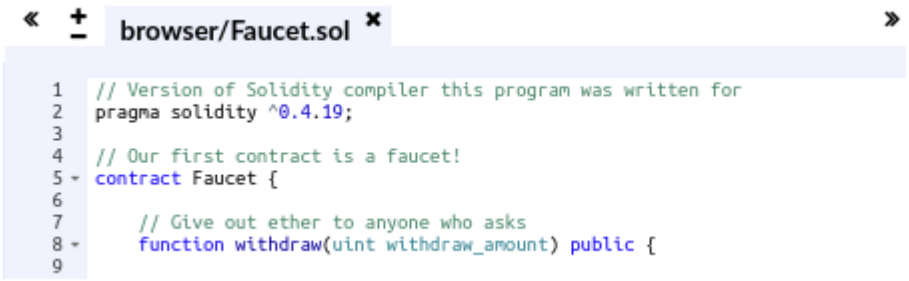


图 12. 将 Faucet 示例代码复制到新选项中

将 Faucet.sol 合约加载到 Remix IDE 后，IDE 将自动编译代码。如果一切顺利，你会在右侧的 Compile 选项卡下看到一个带有“Faucet”的绿色框，确认编译成功（请参阅 Remix 成功编译 Faucet.sol 合约）。

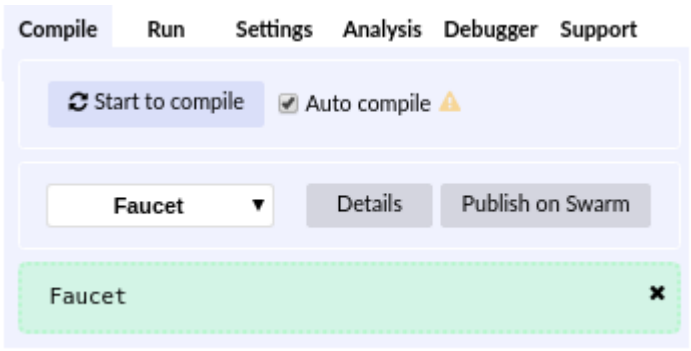


图 13. Remix 成功编译了 Faucet.sol 合约

如果出现问题，最可能的问题是 Remix IDE 使用的是与 0.4.19 不同的 Solidity 编译器版本。在这种情况下，我们的程序指令将阻止 Faucet.sol 编译。要更改编译器版本，请转到“设置”选项卡，将版本设置为 0.4.19，然后重试。

Solidity 编译器现在已将 Faucet.sol 编译为 EVM 字节码。如果你很好奇，字节码看起来像这样：

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN
STOP PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI
PUSH1 0x0 CALLDATALOAD PUSH29
0x100000000000000000000000000000000000000000000000000000
SWAP1 DIV PUSH4 0xFFFFFFFF 和 DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI
JUMPDEST STOP JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST PUSH1 0x5F PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD SWAP1
SWAP2 SWAP1 POP POP PUSH1 0x61 JUMP JUMPDEST STOP JUMPDEST PUSH8
0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO PUSH1 0x77 JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
AND
PUSH2 0x8FC DUP3 SWAP1 DUP2 ISZERO MUL SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1
0x40 MLOAD DUP1 DUP4 SUB DUP2 DUP6 DUP9 DUP9 CALL SWAP4 POP POP POP POP ISZERO
ISZERO PUSH1 0xB6 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP JUMP STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 PUSH9 0x13D1EA839A4438EF75 GASLIMIT CALLVALUE LOG4
0x5f
PUSH24 0x7541F409787592C988A079407FB28B4AD000290000000000
```

您是否很高兴使用像 Solidity 这样的高级语言而不是直接在 EVM 字节码中编程？我也是！

在区块链上创建合约

所以，我们有合约。我们把它编译成字节码。现在，我们需要在以太坊区块链上“注册”合约。我们将使用 Ropsten 测试网来测试我们的合约，这个区块链就是我们想要提交给它的。

在区块链上注册合约涉及创建一个特殊交易，地址是0x0000000000000000000000000000000000000000，也称为零地址。零地址是一个特殊地址，告诉以太坊区块链您想要注册合约。幸运的是，Remix IDE 将为您处理所有这些并将交易发送到 MetaMask。

首先，切换到“运行”选项，然后在“环境”下拉选择框中选择“Injected Web3”。这将 Remix IDE 连接到 MetaMask 钱包，并通过 MetaMask 连接到 Ropsten 测试网络。一旦你这样做，你可以在环境下看到 Ropsten。此外，在“帐户选择”框中，它显示钱包的地址（请参阅“ 混合 IDE 运行”选项卡，选中“Injected Web3”环境）。

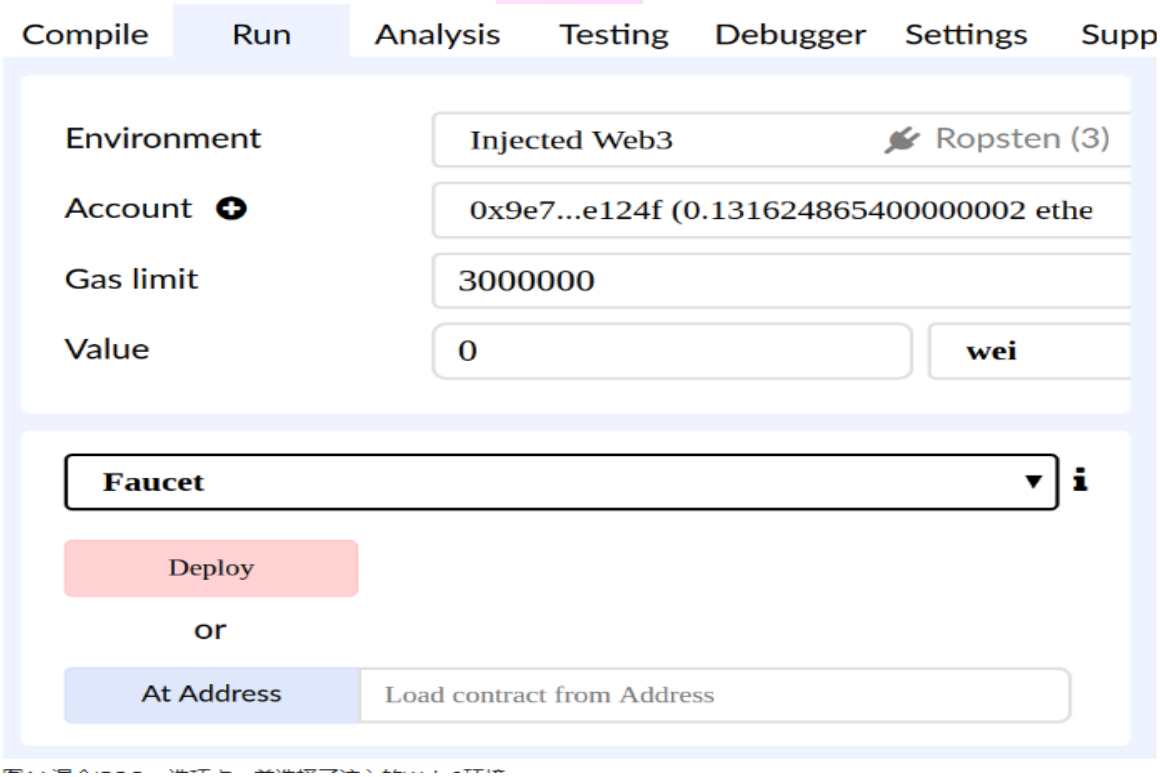


图 14. Remix IDE Run 选项卡，选中 Injected Web3 环境

刚刚确认的运行设置正下方是水龙头合约，准备创建。单击 Remix IDE Run 选项卡中显示的 Deploy 按钮，选中 Injected Web3 环境。

Remix 将构建特殊的“创建”交易，MetaMask 将要求您批准它，如 MetaMask 中所示，显示合约创建交易。您会注意到合约创建交易中没有以太，但它有 258 个字节的数据（已编译的合约）并将消耗 10 gwei 的气体。单击“提交”以批准它。

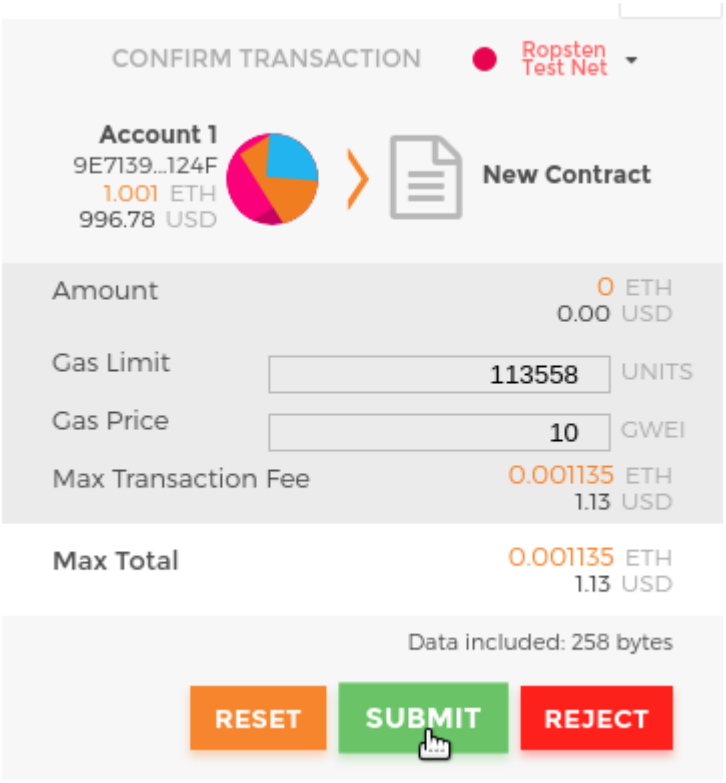


图 15. 显示合约创建交易的 MetaMask

现在你必须等待。合约在 Ropsten 上运行需要 15 到 30 秒。Remix 不用做太多，但请耐心等待。

创建合约后，它将显示在“运行”选项卡的底部（请参阅“龙头合约正在进行中！”）。

图 16. 水龙头合约正在进行中！

请注意，龙头合约现在有一个自己的地址：Remix 将其显示为“Faucet at 0x72e . . . c7829”（尽管您的地址，随机字母和数字会有所不同）。右侧的小剪贴板符号允许您将合约地址复制到剪贴板。我们将在下一节中使用它。

与合约互动

让我们回顾一下迄今为止我们学到的东西：以太坊合约是控制资金的程序，它在称为 EVM 的虚拟机内运行。它们由特殊交易创建，该交易提交其字节码以记录在区块链上。一旦在区

块链上创建了它们，它们就会有一个以太坊地址，就像钱包一样。只要有人将交易发送到合约地址，就会导致合约在 EVM 中运行，并将交易作为输入。发送到合约地址的交易可能具有以太网或数据或两者。如果它们含有乙醚，则将其“存入”合约余额。如果它们包含数据，则数据可以在合约中指定命名功能并调用它，将参数传递给功能。

在区块链资源管理器中查看合约地址

我们现在在区块链上记录了一份合约，我们可以看到它有一个以太坊地址。让我们在 ropsten.etherscan.io 块资源管理器中查看它，看看合约是什么样的。在 Remix IDE 中，通过单击其名称旁边的剪贴板图标来复制合约的地址（请参阅从 Remix 复制合约地址）。

图 17. 从 Remix 复制合约地址

保持 Remix 开启； 我们稍后再回来。现在，将浏览器导航到 ropsten.etherscan.io 并将地址粘贴到搜索框中。您应该看到合约的以太坊地址历史记录，如在 Etherscan 块浏览器中查看水龙头合约地址中所示。

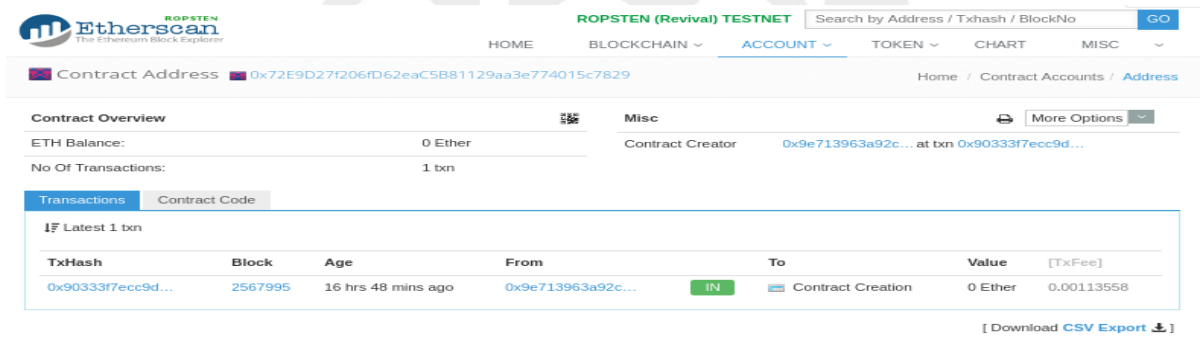


图 18. 在 Etherscan 块资源管理器中查看水龙头合约地址

为合约提供资金

目前，合约在其历史上只有一个交易：合约创建交易。如您所见，合约也没有以太（零余额）。那是因为我们没有在创建交易中向合约发送任何以太，即使我们可以。

我们的水龙头需要资金！我们的第一个项目是使用 MetaMask 将以太送到合约中。您仍然应该在剪贴板中包含合约的地址（如果没有，请从 Remix 再次复制）。打开 MetaMask，并向其发送 1 个以太，就像您对任何其他以太坊地址一样（请参阅将 1 以太网发送到合约地址）。

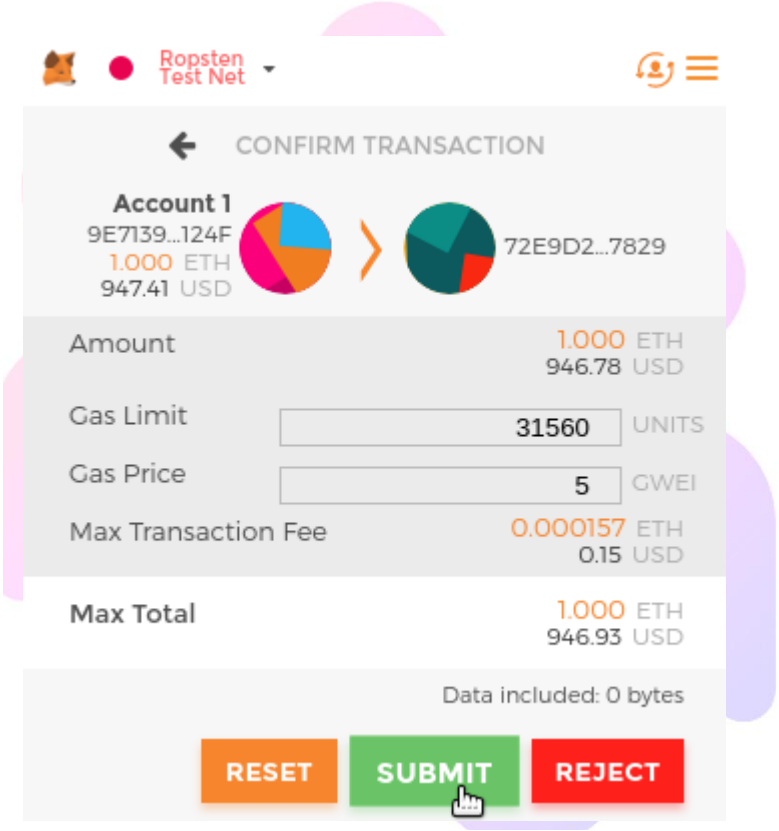


图 19. 将 1 ether 发送到合约地址

在一分钟内，如果您重新加载 Etherscan 块资源管理器，它将显示合约地址的另一个交易以及 1 个以太网的更新余额。

还记得我们的 Faucet.sol 代码中未命名的默认公共支付功能吗？它看起来像这样：

```
function () public  paid {}
```

当您将交易发送到合约地址时，没有数据指示要启用的功能，这是默认功能。因为我们认为它是可支付的，所以它接受并将 1 以太存入合约的账户余额。您的交易导致合约在 EVM 中运行，更新其余额。你资助了你的水龙头！

从合约提现

接下来，让我们从水龙头中提取一些资金。要退出，我们必须构造一个提现的交易，并将提现金额参数输入。简单起见，Remix 将为我们构建该交易，MetaMask 将提供它以供我们批准。

返回 Remix 选项卡并查看 Run 选项卡上的合约。你应该看到一个标有“退出”的红色框，标有 uint256 提现的字段条目（参见 Remix 中 Faucet.sol 的提现功能）。

图 20. Remix 中 Faucet.sol 的提现功能

这是合约的 Remix 界面。它允许我们构造调用合约中交易功能。我们将输入体现金额并单击“提现”按钮以生成交易。

首先，让我们弄清一下提现金额。我们想尝试提现 0.1 以太，这是我们合约允许的最大金额。请记住，以太坊中的所有货币值都在内部以 wei 计价，我们的提现金额也以 wei 计价。我们想要的数量是 0.1 以太，这是 100,000,000,000,000,000 wei（1 后跟 17 个零）。

提示：由于 JavaScript 的限制，Remix 无法处理大到 10 ^ 17 的数字。相反，我们将它括在双引号中，以允许 Remix 将其作为字符串接收并将其作为 BigNumber 进行操作。如果我们不将它括在引号中，则 Remix IDE 将无法处理它并显示“Error encoding arguments:Error: Assertion failed”。

在提现金额框中键入“100000000000000000”（带引号），然后单击“提现”按钮（请参阅“混合”中的“提现”以创建提款交易）。

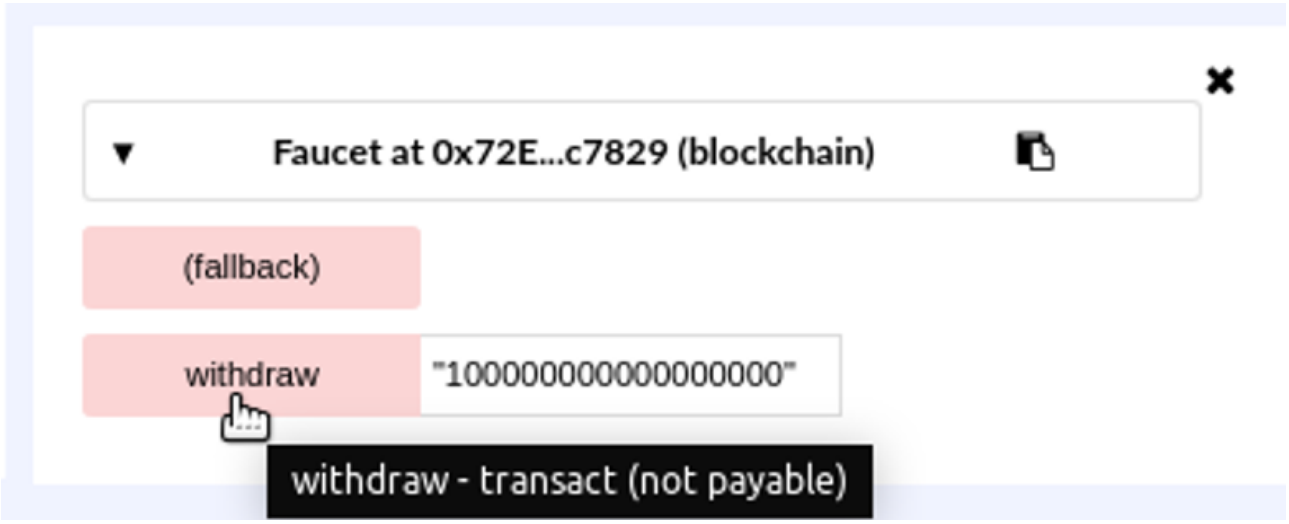


图 21. 单击 Remix 中的“提现”以创建提款交易

MetaMask 将弹出一个交易窗口供您批准。单击“提交”将您的提款请求发送到合约（请参阅 MetaMask 交易以调用提款功能）。

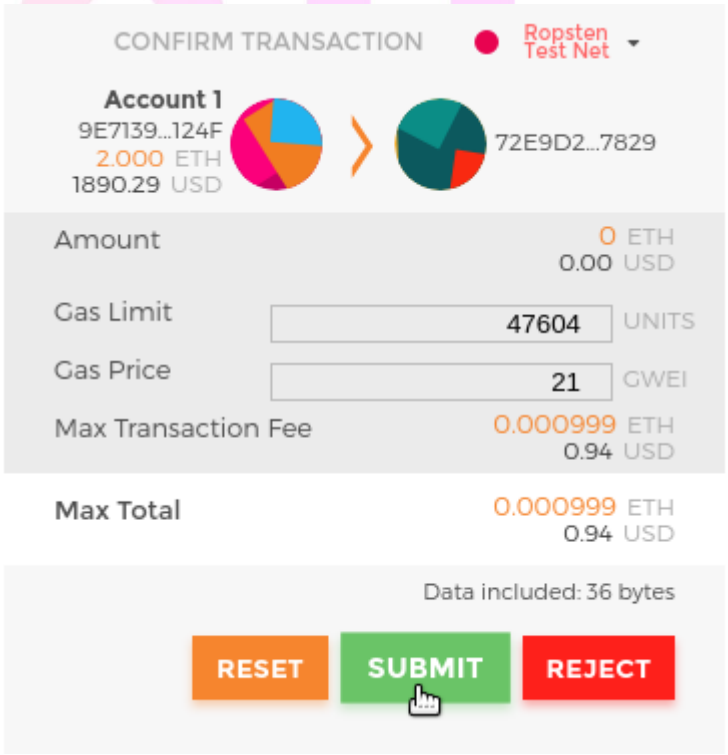


图 22. 调用提现功能的 MetaMask 交易

等一下，然后重新加载 Etherscan 块资源管理器，以查看反映在 Faucet 合约地址历史交易记录（请参阅 Etherscan 显示调用提现功能的交易）。

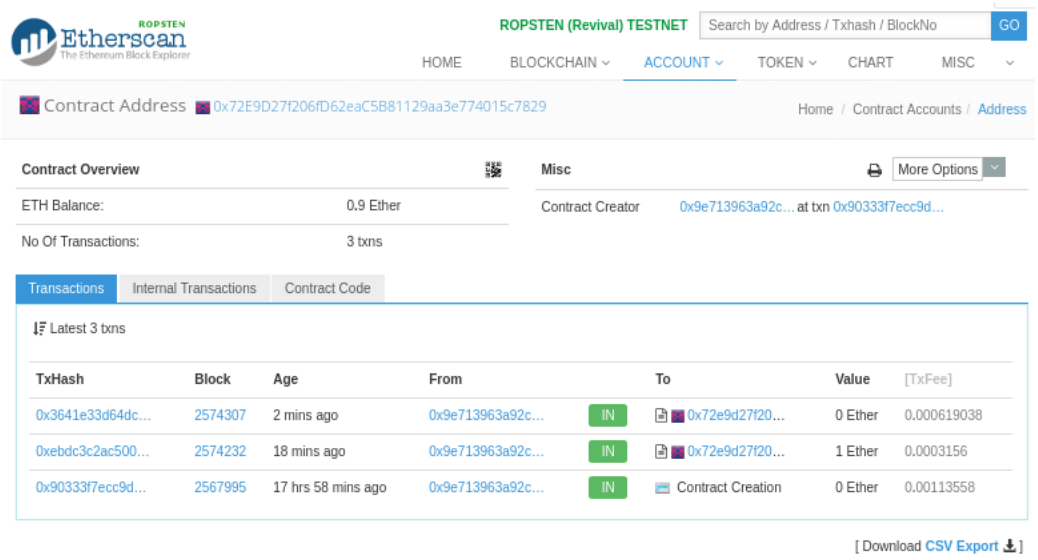


图 23. Etherscan 显示调用提现功能的交易

我们现在看到一个新的交易，以合约中以太为 0 的地址作为目标。合约余额已经改变，现在是 0.9 以太，因为它按要求向发送了 0.1 以太。但我们在合约地址历史记录中看不到“输出”交易。

提现去了哪里？合约的地址历史记录页面上出现了一个新选项卡，名为“内部交易”。因为 0.1 以太转移至合约代码，所以它是内部交易（也称为消息）。单击该选项卡以查看它（请参阅 Etherscan 显示从合约中转出以太的内部交易）。

这个“内部交易”是由合约在这行代码提现中发出的（来自 Faucet.sol 中的功能）：

```
msg.sender.transfer(withdraw_amount);
```

回顾一下：您从 MetaMask 钱包中发送了一个包含数据指令的交易，以使用提现功能将 0.1 ether 提现。该交易导致合约在 EVM 内部运行。当 EVM 运行龙头合约的提现功能时，首先它启用要求功能并验证所请求的数量小于或等于允许的最大提现 0.1 以太。然后它启用传输功能给你发送以太。运行转账功能会产生一个内部交易，从合约的余额中将 0.1 以太币存入您的钱包地址。这是在 Etherscan 的内部交易选项卡上显示的那个。

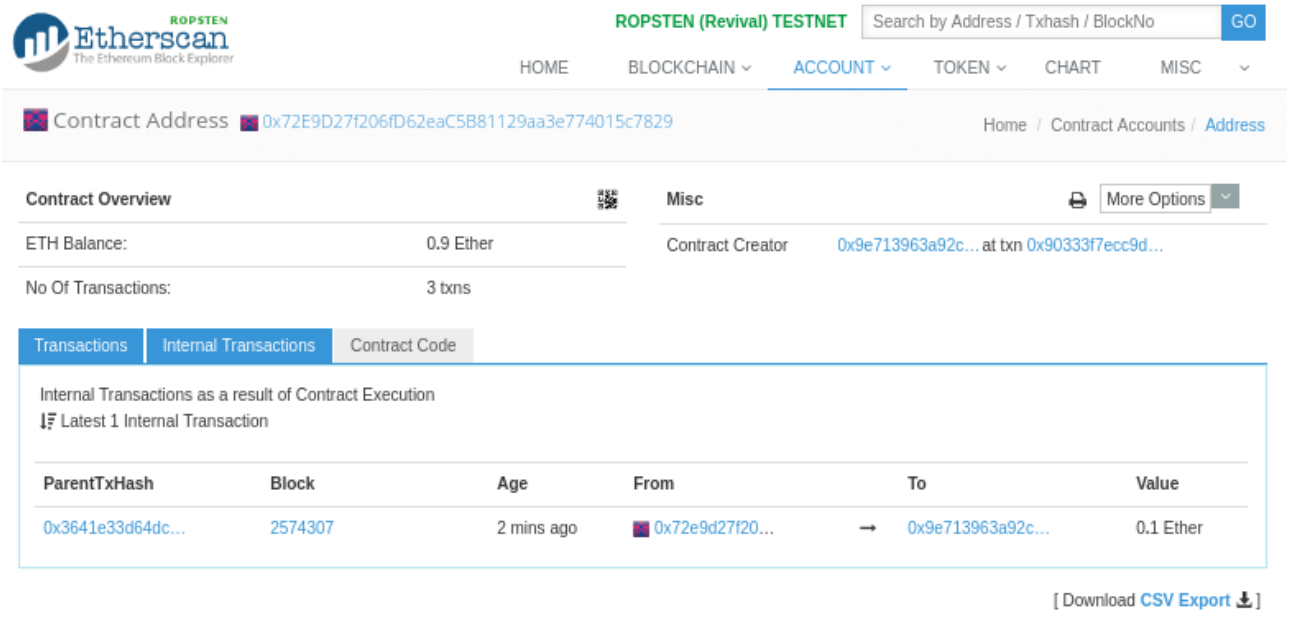


图 24. Etherscan 显示了从合约中转出以太的内部交易

结论

在本章中，您使用 MetaMask 设置了一个钱包，并使用 Ropsten 测试网络上的水龙头为其提供资金。你收到了以太币进入钱包的以太坊地址，然后你把以太送到了水龙头的以太坊地址。

接下来，你在 Solidity 写了一个水龙头合约。您使用 Remix IDE 将合约编译为 EVM 字节码，然后使用 Remix 形成交易并在 Ropsten 区块链上创建 Faucet 合约。一旦创建，水龙头合约就有一个以太坊地址，你发送了一些以太。最后，您构建了一个交易来使用提现功能并成功收到 0.1 ether。合约验证了要求，并通过内部交易向您发送了 0.1 以太。

它可能看起来不怎么样，但您刚刚成功地与在分布式计算机上用控制资金的软件进行了交易。

我们在[smart_contracts_chapter]学习更多优秀的合约式编程，在[smart_contract_security]中学习最好的操作和安全因素。

第三章 以太坊客户端

以太坊客户端

以太坊客户端是一个软件应用程序，它实现以太坊规范并通过对等网络与其他以太坊客户端进行通信。如果不同的以太坊客户端符合参考规范和标准化通信协议，则可以进行交互操作。虽然这些不同的客户端由不同的团队和不同的编程语言实现，但它们都“说”相同的协议并遵循相同的规则。因此，它们都可以操作并交互在同一个太坊网络。

以太坊是一个开源项目，所有主要客户的源代码都可以在开源许可下使用（例如，LGPL v3.0），可以免费下载并用于任何目的。但是，开源意味着不仅仅是免费使用，也意味着以太坊是由一个开放的志愿者社区开发的，任何人都可以修改。更多的关注意味着更值得信赖的代码。

以太坊由称为“黄皮书”的正式规范定义（参见[参考文献]）。

这与例如比特币形成对比，比特币没有以任何正式方式定义。比特币的“规范”是比特币核心的参考实现，以太坊的规范记录在一篇结合了英语和数学（正式）规范的论文中。除了各种以太坊改进提案之外，此正式规范还定义了以太坊客户端的标准行为。随着对以太坊的重大改变，黄皮书会定期更新。

由于以太坊明确的正式规范，以太网客户端有许多独立开发但可互操作的软件实现。与其他任何区块链相比，以太坊在网络上运行的实现更加多样化，通常这被认为是一件好事。事实上，它已被证明是防御网络攻击的绝佳方法，因为利用特定客户端的实施策略只会在开发人员修补漏洞时惹恼开发人员，而其他客户端则保持网络运行几乎不受影响。

以太坊网络

存在各种基于以太坊的网络，这些网络很大程度上符合以太坊黄皮书中定义的形式规范，但相互之间可能可以相互操作，也可能不可以。

在这些基于以太坊的网络中有以太坊，以太坊经典，Ella，Expanse，Ubiq，Musicoin等等。虽然在协议级别大多兼容，但这些网络通常具有要求以太坊客户端软件的维护者进行小的更改以支持每个网络的功能或属性。因此，并非所有版本的以太坊客户端软件都运行基于以太坊的区块链。

目前，以太坊协议有六种主要实现，用六种不同的语言编写：

- Parity，Rust 编写
- Geth，Go 编写
- cpp-ethereum，C++编写
- pyethereum，Python 编写
- Mantis，Scala 编写
- Harmony，Java 编写

在本节中，我们将介绍两个最常见的客户，Parity 和 Geth。我们将展示如何使用每个客户端设置节点，并探索它们的一些命令行选项和应用程序编程接口（API）。

我应该运行完整节点（全节点）吗？

区块链的健康，弹性和审查抵抗力取决于它们具有许多独立操作和地理上分散的完整节点。每个完整节点可以帮助其他新节点获取块数据以引导其操作，并为操作员提供对所有交易和合同的权威和独立验证。

但是，运行完整节点将导致硬件资源和带宽成本增加。完整节点必须下载 80-100 GB 的数据（截至 2018 年 9 月，具体取决于客户端配置）并将其存储在本地硬盘驱动器上。随着新事务和块的增加，这种数据负担每天都在迅速增加。我们将在完整节点的硬件要求中更详细地讨论此主题。

以太坊开发不需要在实时主网上运行的完整节点。您可以使用 testnet 节点（将您连接到较小的公共测试区块链之一），使用本地专用区块链（如 Ganache）或由 Infura 等服务提供商提供的基于云的以太坊客户端执行几乎所有操作。

您还可以选择运行远程客户端，该客户端不存储区块链的本地副本或验证块和事务。这些客户端提供钱包的功能，可以创建和广播交易。远程客户端可用于连接到现有网络，例如您自己的完整节点，公共区块链，公共或许可（证明权限）testnet 或私有本地区块链。实际上，您可能会使用 MetaMask，Emerald Wallet，MyEtherWallet 或 MyCrypto 等远程客户端作为在所有不同节点选项之间切换的便捷方式。

术语“远程客户端”和“钱包”可互换使用，但存在一些差异。通常，除了钱包的事务功能之外，远程客户端还提供 API（例如 web3.js API）。

不要将以太坊中远程钱包的概念与轻客户端（类似于比特币中的简化支付验证客户端）的概念混淆。轻客户端验证块头并使用 Merkle 证明来验证区块链中的事务的包含并确定它们的影响，从而为它们提供与完整节点类似的安全级别。相反，以太坊远程客户端不验证块头

或事务。他们完全信任一个完整的客户，让他们访问区块链，因此失去了重要的安全性和匿名性保证。您可以使用自己运行的完整客户端来缓解这些问题。

全节点的优点和缺点

选择运行完整节点有助于您连接它的网络运行，但也会产生一些轻微到中等的成本。让我们看看一些优点和缺点。

优点：

- 支持基于以太坊的网络的弹性和审查阻力
- 权威地验证所有交易
- 无需中介即可与公共区块链上的任何合约互动
- 可以在没有中介的情况下直接将合约部署到公共区块链中
- 可以离线查询（只读）区块链状态（帐户，合约等）
- 可以在不让第三方知道您正在阅读的信息的情况下查询区块链

缺点：

- 需要大量且不断增长的硬件和带宽资源
- 首次启动时可能需要几天才能完全同步
- 必须维护，升级并保持在线以保持同步

公共 Testnet 的优点和缺点

无论您是否选择运行完整节点，您可能都希望运行公共 testnet 节点。让我们看看使用公共 testnet 的一些优点和缺点。

优点：

- 一个 testnet 节点需要同步和存储更少的数据 - 大约 10 GB，具体取决于网络（截至 2018 年 4 月）。
- 一个 testnet 节点可以在几个小时内完全同步。
- 部署合约或进行交易需要测试以太网，它没有价值，可以从几个“水龙头”免费获得。
- Testnets 是具有许多其他用户和合约的公共区块链，运行“直播（真实）”。

缺点：

- 你不能在 testnet 上使用“真正的”钱； 它运行在测试以太网上。因此，你无法测试真实对手的安全性，因为没有任何利害关系。
- 公共区块链的某些方面无法在 testnet 上进行实际测试。例如，交易费虽然是发送交易所必需的，但并不是测试网上的考虑因素，因为 gas 是免费的。此外，测试网络不会像公共主网有时那样经历网络拥塞。

本地区块链模拟的优点和缺点

对于许多测试目的，最好的选择是启动单实例私有区块链。Ganache（以前称为 testrpc）是您可以与之互动的最受欢迎的本地区块链模拟之一，没有任何其他参与者。它具有公共 testnet 的许多优点和缺点，但也有一些差异。

优点：

- 磁盘上没有同步和几乎没有数据；你自己挖掘第一块
- 无需获得测试以太；你“自己”获得可用于测试的矿业奖励
- 没有其他用户，只有你
- 没有其他合约，只是您在启动后部署的合约

缺点：

- 没有其他用户意味着它的行为与公共区块链的行为不同。交易空间或交易顺序没有竞争。
- 除了你以外，没有矿工意味着采矿更具可预测性；因此，您无法测试公共区块链上发生的某些情况。
- 没有其他合约意味着您必须部署要测试的所有内容，包括依赖项和合约库。
- 您无法重新创建一些公共合约及其地址来测试某些方案（例如，DAO 合约）。

运行以太坊客户端

如果您有时间和资源，则应尝试运行完整节点，即使只是为了了解有关该过程的更多信息。在本节中，我们将介绍如何下载，编译和运行以太坊客户端 Parity 和 Geth。这需要熟悉在操作系统上使用命令行界面。无论您选择将它们作为完整节点，作为 testnet 节点还是作为本地私有区块链的客户端运行，都值得安装这些客户端。

完整节点的硬件要求

在开始之前，您应确保拥有一台具有足够资源的计算机来运行以太坊完整节点。您将需要至少 80 GB 的磁盘空间来存储以太坊区块链的完整副本。如果您还想在以太坊测试网上运行完整节点，则至少需要 15 GB。下载 80 GB 的区块链数据可能需要很长时间，因此建议您使用快速的互联网连接。

同步以太坊区块链是非常输入/输出（I / O）密集型（的操作）。最使用一个固态硬盘（SSD）。如果您有机械硬盘驱动器（HDD），则至少需要 8 GB 的 RAM 用作缓存。否则，您可能会发现系统太慢而无法跟上并完全同步。

最低要求：

- 具有 2 个以上核心的 CPU
- 至少 80GB 的可用存储空间
- SSD 最少 4GB RAM，如果是 HDD 硬盘，则 8GB 以上
- 8 MBit/sec 下载速度

这些是同步基于以太坊的区块链的完整（但已修剪）副本的最低要求。

在撰写本文时，Parity 代码库的资源更轻量，因此如果您使用有限的硬件运行，您可能会看到使用 Parity 的更好结果。

如果您想在合理的时间内同步并存储我们在本书中讨论的所有开发工具，库，客户端和区块链，您将需要一台功能更强大的计算机。

推荐配置：

- 具有 4 核以上的快速 CPU
- 16 GB 以上 RAM
- 具有至少 500 GB 可用空间的快速 SSD
- 25 MBit/sec 以上下载速度

很难预测区块链的大小会增加多快以及何时需要更多磁盘空间，因此建议在开始同步之前检查区块链的最新大小。

注意	此处列出的磁盘大小要求假设您将运行具有默认设置的节点，其中区块链被“修剪”旧状态数据。如果您改为运行一个完整的“归档”节点，其中所有状态都保留在磁盘上，则可能需要超过 1TB 的磁盘空间。

这些链接提供了区块链大小的最新估算：

- Ethereum
- Ethereum Classic

构建和运行客户端的软件要求（节点）

本节介绍 Parity 和 Geth 客户端软件。它还假设您使用的是类 Unix 的命令行环境。这些示例显示了在运行 bash shell（命令行执行环境）的 Ubuntu GNU / Linux 操作系统上出现的命令和输出。

通常，每个区块链都有自己的 Geth 版本，而 Parity 通过相同的客户端下载为多个基于以太坊的区块链（以太坊，以太坊经典，Ellaism，Expanse，Musicoin）提供支持。

提示	在本章的许多示例中，我们将使用操作系统的命令行界面（也称为“shell”），通过“终端”应用程序访问。shell 会显示提示；键入命令，shell 响应一些文本和下一个命令的新提示。您的系统上的提示可能看起来不同，但在以下示例中，它由$符号表示。在示例中，当您在$符号后面看到文本时，请不要键入$符号，而是紧跟其后键入命令（以粗体显示），然后按 Enter 执行命令。在示例中，每个命令下面的行是操作系统对该命令的响应。当你看到下一个$前缀时，你会知道它是一个新命令，你应该重复这个过程。

在我们开始之前，您可能需要安装一些软件。如果您从未在当前使用的计算机上进行任何软件开发，则可能需要安装一些基本工具。对于下面的示例，您需要安装源代码管理系统 git；golang，Go 编程语言和标准库；Rust，一种系统编程语言。

可以按照 https://git-scm.com 上的说明安装 Git

可以按照 https://golang.org 上的说明安装 Go

注意	Geth 要求各不相同，但如果你坚持使用 Go 1.10 或更高版本，你应该能够编译你想要的任何版本的 Geth。当然，您应该始终参考您选择的 Geth 风格的文档。 安装在您的操作系统上或可从系统的软件包管理器获得的 golang 版本可能远远超过 1.10。如果是这样，请将其删除并从 https://golang.org/安装最新版本。

可以按照 https://www.rustup.rs/上的说明安装 Rust 。

注意	Parity 需要 Rust 版本 1.27 或更高版本。

Parity 还需要一些软件库，例如 OpenSSL 和 libudev。要在 Ubuntu 或 Debian GNU / Linux 兼容系统上安装它们，请使用以下命令：

```
$ sudo apt-get install openssl libssl-dev libudev-dev cmake
```

对于其他操作系统，请使用操作系统的软件包管理器或按照 Wiki 说明安装所需的库。

现在您已经安装了 git，golang，Rust 和必要的库，让我们开始工作吧！

Parity 是全节点以太坊客户端和 DApp 浏览器的实现。它是在系统编程语言 Rust 中“从头开始”编写的，旨在构建模块化，安全且可扩展的以太坊客户端。Parity 由英国公司 Parity Tech 开发，并根据 GPLv3 免费软件许可证发布。

注意	披露：本书的作者之一，Gavin Wood 博士，是 Parity Tech 的创始人，并撰写了很多 Parity 客户端。Parity 占安装的以太坊客户群的 25%左右。

要安装 Parity，您可以使用 Rust 包管理器 cargo 或从 GitHub 下载源代码。包管理器还下载源代码，因此两个选项之间没有太大区别。在下一节中，我们将向您展示如何自己下载和编译 Parity。

安装 Parity

该 Parity 维基提供了在不同的环境和容器编译 Parity 的指令。我们将向您展示如何从源代码构建 Parity。这假设您已经使用 rustup 安装了 Rust（请参阅构建和运行客户端（节点）的软件要求）。

首先，从 GitHub 获取源代码：

```
$ git clone https://github.com/paritytech/parity
```

然后切换到 parity 目录并使用 cargo 来构建可执行文件：

```
$ cd parity

$ cargo install
```

如果一切顺利，你应该看到类似的东西：

```
$ cargo install

    Updating git repository
`https://github.com/paritytech/js-precompiled.git`

 Downloading log v0.3.7

 Downloading isatty v0.1.1

 Downloading regex v0.2.1


 [...]


Compiling parity-ipfs-api v1.7.0

Compiling parity-rpc v1.7.0

Compiling parity-rpc-client v1.4.0

Compiling rpc-cli v1.4.0 (file:///home/aantonop/Dev/parity/rpc_cli)

Finished dev [unoptimized + debuginfo] target(s) in 479.12 secs

$

Try and run parity to see if it is installed, by invoking the --version option:
```

通过调用--version 选项尝试并运行 parity 以查看它是否已安装：

```
$ parity --version

Parity

  version
Parity/v1.7.0-unstable-02edc95-20170623/x86_64-linux-gnu/rustc1.18.0
```

```
Copyright 2015, 2016, 2017 Parity Technologies (UK) Ltd

License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>.

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law.


By Wood/Paronyan/Kotewicz/Drwięga/Volf

   Habermeier/Czaban/Greeff/Gotchac/Redmann
```

好极了!现在已经安装了Parity,您可以同步区块链并开始使用一些基本的命令行选项。

Geth是由以太坊基金会积极开发的Go语言实现，因此被认为是以太坊客户端的“官方”实现。通常，每个基于以太坊的区块链都有自己的Geth实现。如果您正在运行Geth，那么您需要确保使用以下存储库链接之一获取区块链的正确版本：

- Ethereum (or https://geth.ethereum.org/)
- Ethereum Classic
- Ellaism
- Expanse
- Musicoin
- Ubiq

注意	您也可以跳过这些说明并为您选择的平台安装预编译的二进制文件。预编译版本更容易安装，可以在此处列出的任何存储库的“版本”部分中找到。但是，您可以通过自行下载和编译软件来了解更多信息。

克隆存储库

第一步是克隆Git存储库，以获取源代码的副本。

要创建所选存储库的本地克隆，请在主目录中或在用于开发的任何目录下使用git命令，如下所示：

```
$ git clone <Repository Link>
```

将存储库复制到本地系统时，您将看到进度报告：

```
Cloning into 'go-ethereum'...

remote: Counting objects: 62587, done.

remote: Compressing objects: 100% (26/26), done.

remote: Total 62587 (delta 10), reused 13 (delta 4), pack-reused 62557

Receiving objects: 100% (62587/62587), 84.51 MiB | 1.40 MiB/s, done.

Resolving deltas: 100% (41554/41554), done.

Checking connectivity... done.
```

好极了！现在您已拥有 Geth 的本地副本，您可以为您的平台编译可执行文件。

从源代码构建 Geth

要构建 Geth，请切换到下载源代码的目录并使用 make 命令：

```
$ cd go-ethereum

$ make geth
```

如果一切顺利，您将看到 Go 编译器构建每个组件，直到它生成 geth 可执行文件：

```
build/env.sh go run build/ci.go install ./cmd/geth

>>> /usr/local/go/bin/go install -ldflags -X
main.gitCommit=58a1e13e6dd7f52a1d...

github.com/ethereum/go-ethereum/common/hexutil

github.com/ethereum/go-ethereum/common/math

github.com/ethereum/go-ethereum/crypto/sha3

github.com/ethereum/go-ethereum/rlp

github.com/ethereum/go-ethereum/crypto/secp256k1

github.com/ethereum/go-ethereum/common

[...]

github.com/ethereum/go-ethereum/cmd/utils
```

```
github.com/ethereum/go-ethereum/cmd/geth

Done building.

Run "build/bin/geth" to launch geth.

$
```

让我们确保 geth 已成功安装而不实际开始运行：

```
$ ./build/bin/geth version


Geth

Version: 1.6.6-unstable

Git Commit: 58a1e13e6dd7f52a1d5e67bee47d23fd6cfdee5c

Architecture: amd64

Protocol Versions: [63 62]

Network Id: 1

Go Version: go1.8.3

Operating System: linux

[...]
```

您的 geth version 命令可能会显示略有不同的信息，但您应该看到的版本报告与此处显示的版本报告非常相似。

不要运行 geth，因为它将以“缓慢的方式”开始同步区块链，这将花费太长时间（几周）。接下来的部分将解释以太坊的区块链初始同步带来的挑战。

基于以太坊区块链的第一次同步

通常，在同步以太坊区块链时，您的客户端将从一开始就下载并验证每个块和每个事务 - 即从创世块开始。

虽然可以通过这种方式完全同步区块链，但同步将花费很长时间并且具有很高的资源要求（它将需要更多的 RAM，如果你没有快速存储，则需要很长时间）。

许多基于以太坊的区块链是 2016 年底拒绝服务攻击的受害者。受影响的区块链在进行完全同步时往往会缓慢同步。

例如，在以太坊上，新客户端（下载进度）将快速进展，直到达到 2,283,397 区块。该区块于 2016 年 9 月 18 日开采，标志着 DoS 攻击的开始。从此块到块 2,700,031（2016 年 11 月 26 日），事务验证变得非常缓慢，内存密集，I / O 密集。这导致每块的验证时间超过 1 分钟。以太坊使用硬叉实现了一系列升级，以解决 DoS 攻击中利用的潜在漏洞。这些升级还通过删除垃圾邮件交易创建的大约 2000 万个空帐户来清理区块链。

如果您正在与完全验证同步，您的客户端将变慢并可能需要几天甚至更长时间来验证受 DoS 攻击影响的块。

幸运的是，大多数以太坊客户端都包含执行“快速”同步的选项，该同步会跳过事务的完整验证，直到它同步到区块链的提示，然后恢复完整验证。

对于 Geth，启用快速同步的选项通常称为 --fast。您可能需要参考所选以太坊链的具体说明。

Parity 默认执行快速同步。

注意	Geth 只能在从空块数据库开始时进行快速同步。如果您已经开始没有快速模式同步，Geth 无法切换。删除区块链数据目录并从头开始快速同步比继续同步完全验证更快。删除区块链数据时，请注意不要删除任何钱包！

运行 Geth 或 Parity

现在您已了解“首次同步”的挑战，您已准备好启动以太坊客户端并同步区块链。对于 Geth 和 Parity，您可以使用--help 选项查看所有配置参数。除了使用--fast for Geth 之外，如上一节所述，默认设置通常是合理的，适合大多数用途。选择如何配置任何可选参数以满足您的需求，然后启动 Geth 或 Parity 来同步链。然后等一下.....

提示	在具有大量 RAM 的非常快速的系统上，同步以太坊区块链将花费半天时间，在较慢系统上的将会花费几天时间。

JSON-RPC 接口

以太坊客户端提供应用程序编程接口和一组远程过程调用（RPC）命令，这些命令被编码为 JavaScript 对象表示法（JSON）。您将看到这称为 JSON-RPC API。本质上，JSON-RPC API 是一个接口，允许我们编写使用以太坊客户端作为以太网网络和区块链的网关的程序。

通常，RPC 接口在端口 8545 上作为 HTTP 服务提供。出于安全原因，默认情况下，它仅限于接受来自 localhost（您自己计算机的 IP 地址，即 127.0.0.1）的连接。

要访问 JSON-RPC API，您可以使用专用库（使用您选择的编程语言编写），该库提供与每个可用 RPC 命令相对应的“存根”函数调用，或者您可以手动构造 HTTP 请求并发送/接收 JSON-编码请求。您甚至可以使用通用命令行 HTTP 客户端（如 curl）来调用 RPC 接口。我们试试吧。首先，确保已配置并运行 Geth，然后切换到新的终端窗口（例如，在现有终端窗口中使用 Ctrl-Shift-N 或 Ctrl-Shift-T），如下所示：

```
$ curl -X POST -H "Content-Type: application/json" --data \
  '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}' \
  http://localhost:8545


{"jsonrpc":"2.0","id":1,
"result":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

在此示例中，我们使用 curl 建立到地址 http://localhost:8545 的 HTTP 连接。我们已经在运行 geth，它在端口 8545 上提供 JSON-RPC API 作为 HTTP 服务。我们指示 curl 使用 HTTP POST 命令并将内容标识为 application / json 类型。最后，我们传递一个 JSON 编码的请求作为 HTTP 请求的数据组件。我们的大多数命令行只是设置 curl 以正确建立 HTTP 连接。有趣的部分是我们发出的实际 JSON-RPC 命令：

```
{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}
```

JSON-RPC 请求根据 JSON-RPC 2.0 规范进行格式化。每个请求包含四个元素：

```
jsonrpc
```

JSON-RPC 协议的版本。这里必须是“2.0”。

```
method
```

要调用的方法的名称

```
params
```

一个结构化值，用于保存在调用方法期间要使用的参数值。该成员可以省略。

```
id
```

客户端建立的标识符，如果包含，必须包含 String，Number 或 NULL 值。如果包含，服务器必须在响应对象中回复相同的值。该成员用于关联两个对象之间的上下文。

提示	id 参数主要用于在单个 JSON-RPC 调用中发出多个请求时，这种做法称为批处理。批处理用于避免每个请求的新 HTTP 和 TCP 连接的开销。例如，在以太坊上下文中，如果我们想通过一个 HTTP 连接检索数千个事务，我们将使用批处理。批处理时，为每个请求设置不同的 ID，然后将其与来自 JSON-RPC 服务器的每个响应中的 id 匹配。实现此目的的最简单方法是维护计数器并增加每个请求的值。

我们收到的回复是：

```
{"jsonrpc":"2.0","id":1,
"result":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

这告诉我们 Geth 客户端版本 1.8.0 正在提供 JSON-RPC API。

让我们尝试更有趣的事情。在下一个示例中，我们向 JSON-RPC API 询问以 wei 为单位的当前 gas 价格：

```
$ curl -X POST -H "Content-Type: application/json" --data \
  '{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":4213}' \
  http://localhost:8545


{"jsonrpc":"2.0","id":4213,"result":"0x430e23400"}
```

响应中的 0x430e23400 告诉我们，目前的 gas 价格是 18 gwei（千兆 wei 或亿 wei）。如果像我们一样，你不用十六进制思考，你可以在命令行上用一点 bash-fu 将它转换为十进制：

```
$ echo $((0x430e23400))


18000000000
```

可以在以太坊 wiki 上查询完整的 JSON-RPC API 。

Parity 的 Geth 兼容模式

Parity 有一个特殊的“Geth 兼容模式”，它提供的 JSON-RPC API 与 Geth 提供的相同。要在此模式下运行 Parity，请使用--geth 开关：

```
$ parity --geth
```

远程以太坊客户端

远程客户端提供完整客户端功能的子集。它们不存储完整的以太坊区块链，因此它们设置起来更快，并且需要更少的数据存储。

这些客户端通常提供执行以下一项或多项操作的功能：

- 管理钱包中的私钥和以太网地址。
- 创建，签署和广播交易。
- 使用数据有效负载与智能合约进行交互。
- 浏览 DApps 并与之交互。
- 提供外部服务的链接，例如块浏览器。
- 转换以太单位并从外部来源检索汇率。
- 将 web3 实例作为 JavaScript 对象注入 Web 浏览器。
- 使用由另一个客户端提供/注入浏览器的 web3 实例。
- 访问本地或远程以太坊节点上的 RPC 服务。

某些远程客户端（例如移动（智能手机）钱包）仅提供基本钱包功能。其他远程客户端是完整的 DApp 浏览器。远程客户端通常提供全节点以太坊客户端的一些功能，而无需通过连接到其他地方运行的完整节点来同步以太坊区块链的本地副本，例如，您在本地机器上或 Web 服务器上，或通过他们服务器上的第三方。

让我们来看看一些最流行的远程客户端及其提供的功能。

移动（智能手机）钱包

所有移动钱包都是远程客户端，因为智能手机没有足够的资源来运行完整的以太坊客户端。轻客户端正在开发中，而不是通常用于以太坊。在 Parity 的情况下，light 客户端被标记为“experimental”，可以通过使用--light 选项运行 Parity 来使用。

流行的移动钱包包括以下内容（我们仅将这些列为示例;这不是对这些钱包的安全性或功能的认可或指示）：

Jaxx

基于 BIP-39 助记符种子的多货币手机钱包，支持比特币，Litecoin，以太坊，以太坊 Classic，ZCash，各种 ERC20 令牌以及许多其他货币。Jaxx 可在 Android 和 iOS 上使用，可作为浏览器插件钱包使用，也可作为各种操作系统的桌面钱包使用。

Status

移动钱包和 DApp 浏览器，支持各种令牌和流行的 DApp。适用于 iOS 和 Android。

Trust Wallet

移动的以太坊和以太坊 Classic 钱包，支持 ERC20 和 ERC223 令牌。Trust Wallet 适用于 iOS 和 Android。

Cipher Browser

全功能的支持以太坊的移动 DApp 浏览器和钱包，允许与以太坊应用程序和令牌集成。适用于 iOS 和 Android。

浏览器钱包

各种钱包和 DApp 浏览器可用作 Chrome 和 Firefox 等 Web 浏览器的插件或扩展。这些是在浏览器中运行的远程客户端。

一些比较流行的是 MetaMask，Jaxx，MyEtherWallet / MyCrypto 和 Mist。

MetaMask

MetaMask，介绍于[intro chapter] ，是一种多用途的基于浏览器的钱包，RPC 客户端，以及基本的合约浏览。它适用于 Chrome，Firefox，Opera 和 Brave Browser。

与其他浏览器钱包不同，MetaMask 将 web3 实例注入浏览器 JavaScript 上下文，充当连接到各种以太网区块链（主网，Ropsten testnet，Kovan testnet，本地 RPC 节点等）的 RPC 客户端。注入 web3 实例并充当外部 RPC 服务的网关的能力使 MetaMask 成为对开发人员和用户来说都非常强大的工具。例如，它可以与 MyEtherWallet 或 MyCrypto 组合，充当这些工具的 web3 提供者和 RPC 网关。

Jaxx

Jaxx 在上一部分作为移动钱包推出，也可作为 Chrome 和 Firefox 扩展以及桌面钱包使用。

MyEtherWallet (MEW)

MyEtherWallet 是一个基于浏览器的 JavaScript 远程客户端，提供：

- 一个用 JavaScript 运行的软件钱包
- 通往 Trezor 和 Ledger 等流行硬件钱包的桥梁
- 一个 web3 接口，可以连接到另一个客户端注入的 web3 实例（例如，MetaMask）
- 可以连接到以太坊完整客户端的 RPC 客户端
- 给定合约地址和应用程序二进制接口（ABI）的基本接口，可与智能合约交互

MyEtherWallet 对于测试和硬件钱包的接口非常有用。它不应该用作主要的软件钱包，因为它通过浏览器环境暴露于威胁，并且不是安全的密钥存储系统。

警告	访问 MyEtherWallet 和其他基于浏览器的 JavaScript 钱包时，您必须非常小心，因为它们经常成为网络钓鱼的目标。始终使用书签而不是搜索引擎或链接来访问正确的 Web URL。

MyCrypto

就在本书出版之前，MyEtherWallet 项目分为两个相互竞争的实现，由两个独立的开发团队指导：一个“分叉”，正如在开源开发中的称号。这两个项目名为 MyEtherWallet（原始品牌）和 MyCrypto。在拆分时，MyCrypto 提供了与 MyEtherWallet 相同的功能，但由于两个开发团队采用不同的目标和优先级，这两个项目很可能会分歧。

Mist

Mist 是由以太坊基金会建造的第一个支持以太坊的浏览器。它包含一个基于浏览器的钱包，它是 ERC20 令牌标准的第一个实现（Fabian Vogelsteller，ERC20 的作者，也是 Mist 的主要开发者）。Mist 也是第一个引入 camelCase 校验和（EIP-55）的钱包。Mist 运行完整节点，并提供完整的 DApp 浏览器，支持基于 Swarm 的存储和 ENS 地址。

结论

在本章中，我们探讨了以太坊客户端。您下载，安装并同步了客户端，成为以太坊网络的参与者，并通过在您自己的计算机上复制区块链来促进系统的健康和稳定发展。

第四章 私钥，地址

密码学

以太坊的基础技术之一就是密码学;密码学是数学的一个分支,广泛用于计算机安全领域。密码学在希腊文中表示为“秘密书写”， 但是密码学背后的科学包含了“加密”这种远比“秘密书写”内涵丰富的知识。密码学也可以通过不泄漏加密文件（数字签名）的手段来验证加密内容；或者验证数据的真实性（数字指纹)。这几种密码学公理不仅是以太坊，同时也是很多种区块链系统以及基于以太坊应用的最为重要的数学工具 讽刺的是,加密其实并不是以太坊最重要的部分，因为以太坊的通信以及交易数据没有加密，并且也不需要加密就可以维护系统的安全。这章我将介绍以太坊中用来控制资产主权的两个概念：私钥以及地址。

私钥，地址

以太坊有两种账户，一种用于拥有以太，另一种用来控制以太，这两种账号分别为：外部拥有账户(EOA)以及协议。在这部分我将研究如何通过密码学中的外部拥有账户,例如私钥,来建立以太的所有权。

外部拥有账户(EOA)中的以太所有权是通过数字秘钥,以太坊地址以及数字签名建立的。数字秘钥并没有被存储到区块链内也并没有在以太坊网络上进行传输，而是由用户创建并存储在一个文件或被称为钱包的简单数据库中。钱包中用户的数字私钥是完全独立于以太坊协议中的，并且这个私钥可以在完全与区块链或者互联网隔绝的情况下通过用户的钱包软件产生或者管理。数字签名使以太坊的很多例如去中心化信任与控制以及所有权验证等特点成为可能。

以太坊交易需要将一个有效的数字签名包含在区块链中，这个数字签名可以通过私钥生成；因此，任何人只要拥有了这个私钥，就拥有了以太的所有权。以太坊交易中的数字签名证明了资产的真正所有者。

数字秘钥成对出现，包含一个私（秘）钥以及一个公钥。可以将公钥看作是银行账户的账号，私钥看作是银行账户的中提供银行账户控制权的密码。这些数字秘钥是以太坊使用者很少看到的。大部分情况下，这些秘钥都是存储到钱包文件中并且通过以太坊钱包管理的。

在一个以太坊交易过程中的付款部分，交易的目标收款方通过以太坊地址表示，类似于支票上填写的受益人名称（即‘以某人的名义付给’）。很多情况下， 以太坊地址由公钥产

生，并关联于公钥。但是，并不是所有的以太坊地址代表公钥；他们也可以代表我们在[合约]部分即将看到的合约。以太坊地址是用户们常见的唯一密钥表示，因为这是用户们与世界分享的一部分。

首先，我们将介绍密码学并解释我们在以太坊中使用到的数学，其次，我们将会阐述密钥是如何生成，存储以及管理的。最后我们将回顾用于表示私钥，公钥以及地址的多种编码格式。

公钥密码学以及加密货币

公钥密码学发明与 19 世纪 70 年代，是计算机以及信息安全的数学基础。

当代密码学更多的是基于拥有同一个独特特点的数学函数：从一个方向计算非常简单但是从相反方向计算却非常难。基于以上数学函数，密码学使得数字秘密的产生以及使得不可丢失的数字签名成为可能。

例如， 将两个很大的质数相乘不是难事。但是给出两个很大的质数相乘的结果，我们很难求出这两个质数（一个被称为质数数分解的问题）。假设我提出一个数为 6895601，我告诉你这个数由两个质数相乘所得。求出这两个质数远比将两个质数相乘得到 6895601 困难得多。

如果给予你一个秘密信息，上面所说的一些问题将不难解决。比如说上面我们讨论的例子，如果我告诉你两个质数中的一个为 1931， 你可以很轻松的找出与之相乘得 6895601 的另一个质数：6895601 / 1931 = 3571。 这类函数被称为陷门函数，因为给出秘密信息中的一部分，你可以找到一条捷径来将问题化简为一个很简单的问题。

另外一类在密码学中用处很广的数学函数基于椭圆曲线上的算术运算。在椭圆曲线上，相乘模块乘以素数是较为简单的，但是除法是不可能的（一个被称为离散对数的问题）。椭圆曲线密码学被广泛用于当代计算机系统，并且是以太坊（以及其他虚拟货币）数字密钥以及数字签名的基础。

注意	阅读更多关于密码学以及现代密码学中的数学函数： 密码学： https://en.wikipedia.org/wiki/Cryptography 陷门函数： https://en.wikipedia.org/wiki/Trapdoor_function 质数分解： https://en.wikipedia.org/wiki/Integer_factorization 离散对数： https://en.wikipedia.org/wiki/Discrete_logarithm 椭圆曲密： https://en.wikipedia.org/wiki/Elliptic_curve_cryptography

以太坊中，我们使用公钥加密来创作一对秘钥，通过这对秘钥我们可以访问以太或者验证合同。这对秘钥由一个私钥和一个由此得来的独一无二的公钥构成。这个公钥用来收取资金，私钥用来制作数字签名以便签署一个交易来使用资产。数字签名同时也被用来验证所有者或者合约的使用者，我们将会在[合约验证]部分详细了解。

公钥和私钥之间有着一种数学关系，这种数学关系可以使得私钥生成信息上的签名。该签名可以在不公开私钥的情况下验证其真实性。

使用以太的时候，当前的拥有者在每一笔交易中展示她的公钥以及签名（每一次都不一样，但是由同一个私钥产生）。在公钥以及签名的展示过程中，以太坊系统内的所有人都可以独自验证并接受交易的有效性，确认交易以太的人在交易过程中为这些以太的拥有者。

提示	在绝大部分钱包软件中，私钥以及公钥为了方便共同存储为密钥对。但是，公钥可以通过私钥很简单的推导出，所以只存储私钥也是可行的。

为什么使用非对称加密（公/私钥）？

为什么在以太坊中使用非对称加密？对称加密并不用来“加密”某笔交易。非对称加密有用的特性是能够生成数字签名。一个私钥可以应用于一笔交易中的数字指纹来生成一个数字签名。这个签名只能由拥有私钥的人生成。然而，任何能够访问公钥以及交易指纹的人都可以使用它们来验证签名。非对称加密的这个特性使得任何人都能够验证任意一笔交易中的签名，同时可以确保只有私钥的拥有者才能生成有效签名。

私钥

一个私钥就是一串随机挑选数字。对私钥的拥有权和控制权是用户控制某个以太坊地址资金的基础，也是访问授权该地址的合同的基础。通过证明在交易过程中用到的资产的所有权，私钥可以用来生成在使用以太过程中需要用到的签名。私钥在任何时刻都要保密，因为向第三方透露私钥就等于给予对方对这些以太资产以及合约的所有权。私钥必须备份并且一定要妥善保管，因为如果将私钥丢失就不能被找回，并且与之关联的所有资产都会永久丢失。

提示	以太坊的私钥只是一串数字。你可以通过任何方式随机生成你的私钥，例如硬币，铅笔以及纸：投掷一枚硬币２５６次之后你就拥有了一串可以在以太坊钱包中使用的２５６位的随机生成的二进制数字。公钥以及地址之后可以通过私钥产生。

通过随机数生成私钥

生成秘钥的第一步，也是最重要的一步就是寻找一个安全的熵或者随机性的来源。生成一个以太坊的私钥等价于从 1 到 2^256 挑选数字。挑选数字的方法并不重要，只要方法是不可预测的并且不是重复的。以太坊软件使用的是底层操作系统的随机数生成器来生成 2 5 6 比特的熵（随机性）。通常来说，操作系统的随机数生成器由人初始化，这就是为什么你会被要求晃动几秒钟你的鼠标或者按一些键盘上的随机键位。

更准确的来说，可用的私钥会稍微比 2^256 少。私钥可以是+1+ and n - 1 中的任何一个数，其中 n 是一个常数（n = 1.158 * 10^{77},稍微小于 2^{256}）定义为以太坊中椭圆曲线的阶数（参考解释椭圆曲线加密）。为了生成一个秘钥，我们随机挑选一个 2 5 6 比特的数字并且检查这个数字是否小于 n-1。在编程方面，这通常是通过提供一大串随机位，从一个密码安全的随机源，转换到一个 2 5 6 位的例如 Keccak-256 或者 SHA256 算法里（参考[加密哈希算法]），这种方式可以产生一个 256 比特的数字。如果结果小于 n-1，我们就得到了一个合适的私钥。否则，我们只需再次尝试使用另一个随机数。

不要自己尝试实现一个随机数生成器，也不要使用一个某种编程语言提供的"简单"的随机数生成器（CSPRNG）。使用从具有高熵的信息源生成的种子并且使用密码学上安全的伪随机数生成器（CSPRNG）。使用随机数生成器前要阅读此生成器的文档来判断它是否密码学上足够安全。正确的 CSPRNG 是对保证秘钥的安全性很重要。

下面所示为一个以十六进制表示的随机生成的私钥（ 2 5 6 比特用 6 4 位 1 6 进制数表示，每 4 个比特表示一个 1 6 进制数）：

```
f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

以太坊的私钥可能性空间（2256）是一个深不可测的数字。用十进制表示大约有 1077 种可能。与之比较的话，可见宇宙中大约包含 1080 原子。

公钥

以太坊公钥是椭圆曲线上的一个点，意味着它是一组满足椭圆曲线方程的 X 和 Y 坐标。

简单来说，以太坊的公钥由两个数组成的。这些数由私钥通过一个只能单向的计算得出。这意味着由私钥计算公钥是很简单的，但是不能从公钥计算私钥。

接下来会有数学部分！别担心。如果你觉得上述部分很难理解，那么你可以跳过接下来的几部分。有很多工具以及库可以帮你处理这些数学部分的内容。

公钥通过私钥使用椭圆曲线相乘来计算得出，这个过程是不可逆的：K = k * G, 其中 k 是私钥，G 是一个称为生成点的常数点，k 是对应生成的公钥。这个被称为“搜寻离散对数”的逆运算，难度堪比找出所有 k 的可能取值，也就是暴力搜索。

用更简单的话来说：椭圆曲线上的代数计算与"普通的“整数代数不同。一个点（G）可以与一个整数（k）相乘得出另一个数（k）。但是这个过程中不存在相除，所以不可能直接用点G除公钥K来得到私钥k。这是一个单向数学方程，在公钥密码学以及加密货币部分会继续描述。

提示	椭圆曲线乘法是密码学家们称为"单向"方程的一类方程：从一个方向（相乘）很简单，但是从反方向（相除）却不可能。私钥的所有者可以轻松地生成公钥然后与世界分享，因为大家知道没有人可以将方程反向来通过公钥计算私钥。这种数学技巧成为证明以太坊资金所有权和合同控制权不可伪造的基础，也是保障数字签名安全的基础。

在我们展示如何从私钥生成公钥之前，我们先来更细节的了解一下椭圆曲线加密。

解释椭圆曲线加密

椭圆曲线加密是一类对称且基于离散对数问题的公钥密码学，这个问题由椭圆曲线上点的相加以及相乘表示。

椭圆曲线的可视化是椭圆曲线的一个例子，与以太坊中使用的椭圆曲线很像。

提示	以太坊使用与比特币完全一样的椭圆曲线，称为 secp256k1。这代表比特币中很多椭圆曲线的库都可以被我们重复运用。

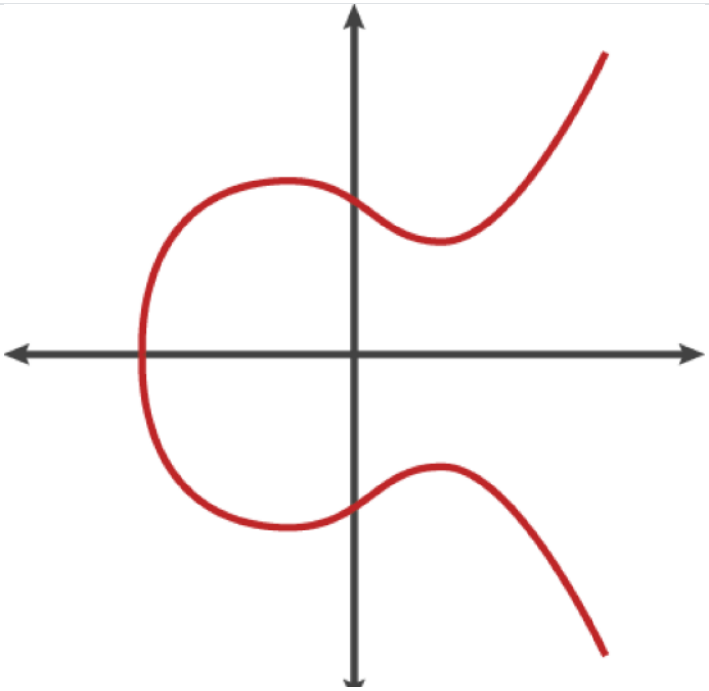


图 1. 椭圆曲线的可视化

以太坊使用的特定椭圆曲线以及数学常数在一个称为 secp256k1 的标准中定义，这个标准由国际标准技术机构建立(NIST).secp256k1 曲线由下面的方程定义。

$$y 2 = (x 3 + 7) \text{ over } (\mathbb{F} p)$$

或

$$y 2 \bmod p = (x 3 + 7) \bmod p$$

mod p（质数 p 模）表明该曲线位于 p 阶素数的有限域上，也可以写为 $\mathbb{F}_p$，其中 $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$,这是一个非常大的质数。

因为这条曲线是在素数阶的有限域上而不是在实数域定义的，所以它看起来像是分散在两个维度上的散点图，使得我们很难观察。 然而，两者数学原理是相同的，散点图与与实数上的椭圆曲线是相同的。比如说，在 F（p）上可视化一个椭圆曲线，p = 17 在一个更小的素数阶 17 的有限域上显示了相同的椭圆曲线，在网格上显示了一个点的模式。secp256k1 以太坊椭圆曲线可以被认为是一个在更大的网格区域内的更加复杂的点的模式。

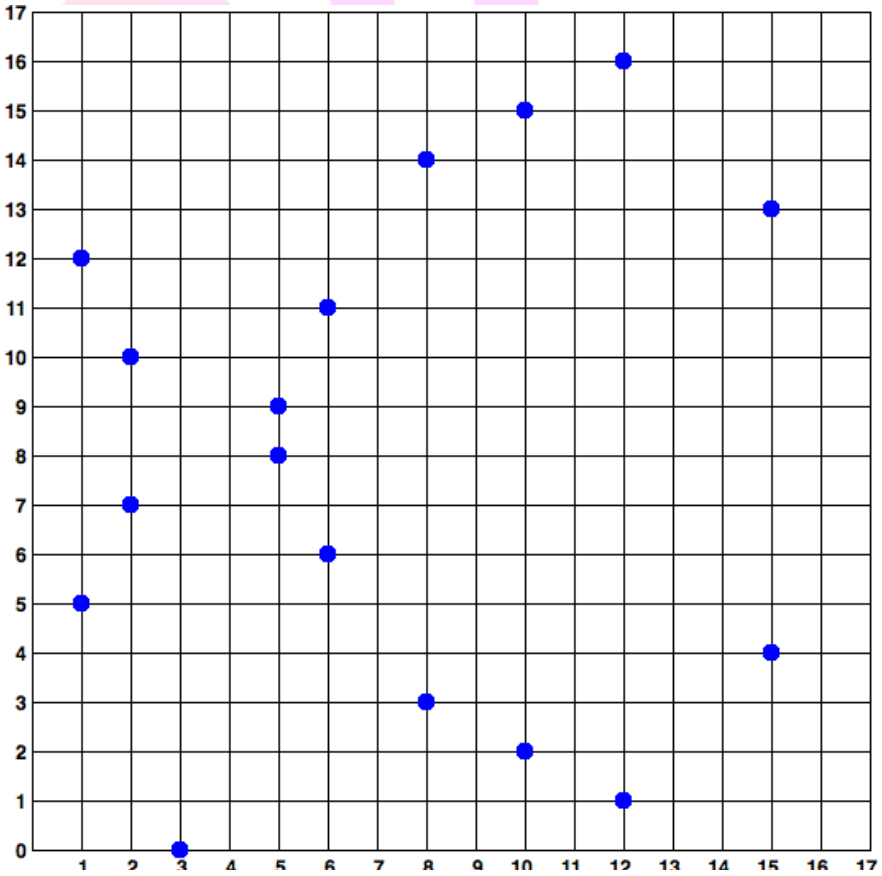


图 2. 椭圆曲线密码学：可视化 F（p）上的椭圆曲线，其中 p = 17

那么，比如说，以下是坐标为（x，y）的某点 Q，它是 secp256k1 曲线上的点：

```
Q =
(49790390825249384486033144355916864607616083520101638681403973749255924
```

```
39515,
595741321618999000458620864939210157800321752917558073992840077210503412973
60)
```

使用 python 证实点在椭圆曲线上展示了如何使用 Python 自己尝试。 变量 x 和 y 是上述点 Q 的坐标。 变量 p 是椭圆曲线的质数阶数（用于所有取模运算的素数）。 Python 的最后一行是椭圆曲线方程（Python 中的%运算符是取模运算符）。 如果 x 和 y 确实是椭圆曲线上的点，那么它们一定满足方程，结果一定为零（+ 0L 是值为零的长整数）。在命令行上输入+python+并复制下面列表中的每一行(>>>+后面的内容)自己尝试一下吧。

Example 1. 使用 python 证实点在椭圆曲线上

```
Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p =
115792089237316195423570985008687907853269984665640564039457584007908834671663
>>> x =
497903908252493844860331443559168646076160835201016386814039737492559245395515
>>> y =
595741321618999000458620864939210157800321752917558073992840077210503412973
60
>>> (x ** 3 + 7 - y**2) % p
0L
```

椭圆曲线算术运算

很多椭圆曲线数学看起来很像我们在学校学到的整数算术。具体而言，我们可以定义一个加法运算符，而不是在曲线上添加数字。一旦有了加法运算符，我们也可以定义一个点和一个整数的乘法，这样它就等于重复加法。

加法被定义为若给定两点 P 1、P 2 都在椭圆曲线上，则在椭圆曲线上也存在第三点 P 3 = P 1 + P 2。

在几何上，通过在 P 1 和 P 2 之间划一条线来计算该第三点 P 3。这条线将在另外一个地方与椭圆曲线相交。称此点 P 3 ’=（x，y）。然后在 x 轴上反射以得到 P 3 =（x，-y）。

在椭圆曲线数学中，有一个叫做无限点的点，它大致对应于零点的作用。在计算机上，它有时用 x = y = 0 表示这并不满足椭圆曲线方程，但它是一个可被检验且简单的独立例子。有一些特殊的例子解释了“无限点”的必要性。

如果 P 1 和 P 2 是相同的点，则在 P 1 和 P 2 之间的线应当延伸成在该点 P 1 处的曲线上的切线。该切线恰好与一条新点中的曲线相交。您可以使用微积分技术来确定切线的斜率。尽管我们将我们的兴趣集中在具有两个整数坐标的曲线上，但这些技巧仍然令人好奇地工作！

在某些情况下（即如果 P 1 和 P 2 具有相同的 x 值但不同的 y 值），切线将完全垂直，在这种情况下，P3 =“无限点”。

如果 P 1 是“无穷远点”（“无限点”），则 P 1 + P 2 = P 2。类似地，如果 P 2 是无限点，则 P 1 + P 2 = P 1。这显示了无限点如何扮演在“正常”算术中的角色。

事实证明+是关联的，这意味着（A + B）+ C = A +（B + C）。这表明我们可以在没有括号的情况下写出 A + B + C 而没有歧义。

现在我们已经定义了加法，乘法也可以用扩展加法的标准方式来定义。对于椭圆曲线上的点 P，如果 k 是整数，则 k * P = P + P + P + … + P（k 次）。请注意，在这种情况下，k 有时会被混淆地称为“指数”

生成公钥

从一个随机生成的数字形式的私有密钥 k 开始，我们把它乘以曲线上被称为发生器点 G 的预定点，以在曲线上的其他位置产生另一个点，这是相应的公钥 K。发生器点被指定为 secp256k1 标准的一部分，并且对于 secp256k1 的所有实现都是相同的，所有从该曲线派生出的密钥都使用相同的点 G：

$$K = k * G$$

其中 k 是私钥，G 是生成器点，K 是结果公钥，即曲线上的一个点。因为所有以太坊用户的生成点始终相同，所以一个乘以 G 的私钥总是得到相同的公钥 K. k 和 K 之间的关系是固定的，但只能按照从 k 到 K 一个方向计算。这就是为什么以太坊地址（从 K 派生）可以与任何人共享，且不会泄露用户的私钥（k）.

正如我们在椭圆曲线算术运算中所描述的那样，k * G 的乘法相当于重复加法，所以 G + G + G + … + G 重复 k 次。总之，为了从私钥 k 产生公钥 K，我们将发生器点 G 自身相加 k 次。

私钥可以转换为公钥，但公钥不能转换回私钥，因为此处只能单向运算。

让我们运用该计算来找到我们在私钥中给出的特定私钥的公钥：

私钥运算产生公钥示例

```
K = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315 * G
```

密码库可以帮助我们使用椭圆曲线乘法计算 K 值。得到的公钥 K 被定义为点 K =(x，y)：

从示例私钥计算的示例公钥

```
K = (x, y)
```

这里,

```
x = 6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b

y = 83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

在以太坊中，您可以看到公钥以 66 个十六进制字符（33 字节）的十六进制序列表示。这采用了行业联盟标准高效密码组(SECG)提出的标准序列化格式，记录在高效密码标准(SEC1)中。该标准定义了四个可用于识别椭圆曲线上点的可能前缀：

字首	含义	长度（字节计数前缀）
0x00	指向无限	1
0x04	未压缩的点	65
0x02	接近 Y 的压缩点	33
0x03	奇数 Y 的压缩点	33

以太坊只使用未压缩的公钥，因此唯一相关的前缀是（十六进制）04。序列化连接公钥的 X 和 Y 坐标：

04 + X 坐标（32 字节/ 64 十六进制）+ Y 坐标（32 字节/ 64 十六进制）

因此，我们在示例私钥计算的示例公钥中计算的公钥被序列化为：

```
046e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38
e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

椭圆曲线库

加密货币相关项目中使用了 secp256k1 椭圆曲线的几个实现：

OpenSSL

TOpenSSL 库提供了一套全面的加密原语，包括 secp256k1 的完整实现。例如，要派生公钥，可以使用函数 EC_POINT_mul（）. 在这里可以找到 https://www.openssl.org/

libsecp256k1

Bitcoin Core 的 libsecp256k1 是 secp256k1 椭圆曲线和其他加密基元的 C 语言实现。椭圆曲线密码学的 libsecp256 是用 scratch 编写的，代替了比特币核心软件中的 OpenSSL，在性能和安全性方面被认为是优越的。在 https://github.com/bitcoin-core/secp256k1 找到它

libsecp256k1

Bitcoin Core 的 libsecp256k1 是 secp256k1 椭圆曲线和其他加密基元的 C 语言实现。椭圆曲线密码学的 libsecp256 是用 scratch 编写的，代替了比特币核心软件中的 OpenSSL，在性能和安全性方面更为优越。在 https://github.com/bitcoin-core/secp256k1 找到它。

加密哈希函数

密码哈希函数在整个以太坊使用。实际上，散列函数在几乎所有的密码系统中都有广泛的应用，这是密码学家布鲁斯•施奈尔（Bruce Schneier）所说的一个事实，他说：“相比加密算法，单向散列函数是现代密码学的主要工具。”

在本节中，我们将讨论散列函数，了解它们的基本属性以及这些属性如何使它们在现代密码学的很多领域广泛运用。我们在这里讨论哈希函数，因为它们是将以太坊公钥转换为地址的一部分。

简而言之，“散列函数是可用于将任意大小的数据映射到固定大小的数据的任意函数。”来源：维基百科。散列函数的输入称为原象或消息。输出被称为散列或摘要。哈希函数的一个特殊子类别是加密哈希函数，它具有对密码学有用的特定属性。

密码散列函数是一种单向散列函数，它将任意大小的数据映射到固定大小的位串，如果知道输出，则在计算上不可能重新创建输入。确定输入的唯一方法是对可能的输入进行暴力搜索，检查匹配输出。

加密哈希函数有五个主要属性（来源：维基百科/加密哈希函数）：

确定性

任何输入消息总是产生相同的散列摘要。

可验证

计算消息的散列是有效的（线性性能）。

不可逆性（抵抗原象）

从哈希计算消息是不可行的，相当于通过可能的消息进行暴力搜索。

不相关

对消息的小改动（例如，一 bit 的改变）应该会全盘改变散列输出，以至于它不能与原始消息的散列相关联。

碰撞保护（抵抗原象）

计算产生相同散列输出的两个不同消息应该是不可行的。

这些属性的组合使加密散列函数可用于广泛的安全应用程序，包括：

- 数据指纹
- 消息完整性（错误检测）
- 工作内容验证
- 身份验证（密码散列和密钥扩展）
- 伪随机数发生器
- 预映像（原象）承诺
- 唯一标识符

当我们在以太坊中通过系统各个层面进行研究时，会发现以上多种属性。

以太坊的加密哈希函数 - Keccak-256

以太坊在许多地方使用 Keccak-256 密码散列函数。Keccak-256 被设为国家科学和技术研究院（NIST）于 2007 年举行的 SHA-3 密码散列函数竞赛的去掉人）。Keccak 在 2015 年成为标准化为联邦信息处理标准（FIPS）202 的获奖算法。

然而，在 Ethereum 开发期间，NIST 标准化工作正在完成。在标准过程完成后，NIST 调整了 Keccak 的一些参数，据称可以提高效率。在此期间，英雄举报者爱德华•斯诺登披露的文件暗示了 NIST 可能受到国家安全局的不当影响，故意削弱 Dual_EC_DRBG 随机数生成器标准，在标准随机数生成器中有效地设置后门。这场争论的结果是以太坊基金会决定实施由其发明人所提议的原来的 Keccak 算法，而不是经 NIST 修改的 SHA-3 标准

虽然您可能在 Ethereum 文档和代码中看到“SHA3”，但很多（可能不是全部）这些实例实际上是指 Keccak-256，而不是最终确定的 FIPS-202 SHA-3 标准。实现差异很小，与填充

参数有关，但它们的重要性在于 Keccak-256 在给定相同输入的情况下产生与 FIPS-202 SHA-3 不同的散列输出。

由于以太坊（Keccak-256）中使用的散列函数与最终标准（FIP-202 SHA-3）之间的差异所导致的混乱，因此在所有代码、操作码和库中重命名所有 sha3 实例到 keccak256 的工作在努力进行中。详情请参阅 ERC-59。

我使用的是哪种散列函数

如何判断您使用的软件库是 FIPS-202 SHA-3 还是 Keccak-256（如果两者都可能被称为“SHA3”）？

一个简单的方法是使用测试矢量，一个给定输入的预期输出。最常用于散列函数的测试是空输入。如果您使用空字符串作为输入运行散列函数，您应该看到以下结果：

测试您使用的 SHA3 库是否是 FIP-202 SHA-3 的 Keccak-256

```
Keccak-256("") =

c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470

SHA-3("") =

a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
```

因此，无论调用什么函数，都可以通过运行上面的简单测试来测试它是否是原始的 Keccak-256 或最终的 NIST 标准 FIPS-202 SHA-3。请记住，以太坊使用 Keccak-256，尽管它在代码中通常被称为 SHA-3。

接下来，我们来研究一下 Ethereum 中 Keccak-256 的第一个应用，它将从公钥生成以太坊地址。

以太坊地址

以太坊地址是使用单向散列函数(特别是Keccak-256)从公钥或合约派生的唯一标识符。在我们之前的例子中，我们从一个私钥开始，使用椭圆曲线乘法来派生一个公钥： 私钥 k：

```
k = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

公钥 K（X 和 Y 坐标连接并显示为十六进制）：

```
K =
6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5
e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

我们使用 Keccak-256 来计算这个公钥的哈希值：

```
Keccak256(K) =
2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

然后我们只保留最后的 20 个字节，这是我们的以太坊地址：

```
001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

大多数情况下，您会看到带有前缀“0x”的以太坊地址，表明它是十六进制编码，如下所示：

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

以太坊地址格式

以太坊地址是一串十六进制数字，通过计算公钥的 Keccak-256 哈希值取其最后 20 个字节作为地址。

比特币通过在客户端的用户界面中编码地址并包含内置校验和，以防止输入错误地址。与之不用的是，以太坊地址以原始十六进制形式呈现，没有经过任何校验和。

这其中的协议设计的理念是，以太坊地址最终会隐藏在系统高层的抽象（如命名服务）之后，并且必要时应在较高层添加校验和。

深思起来，这种设计选择会带来一些问题，包括由于输入错误地址和输入验证错误而导致的资金损失。 此外，以太坊命名服务的开发速度低于最初预期，再诸如 ICAP 之类的替代编码被钱包开发商采纳的进度也十分缓慢。

互通客户端地址协议（ICAP）

客户端地址互换协议（ICAP） 是一种与国际银行帐号（IBAN）编码部分兼容的以太坊地址编码，可以为以太坊地址提供多功能，校验和相互操作编码。 ICAP 地址可以编码以太坊地址或通过以太坊名称注册表注册的常用名称。

详情可以参考以太坊 Wiki 上的 ICAP 原文：
https://github.com/ethereum/wiki/wiki/ICAP:-Inter-exchange-Client-Address-Protocol

IBAN 是识别银行账号的一种国际标准，主要应用于电汇。 它在单一欧元支付区（SEPA）中被广泛采用。 IBAN 是一项集中且严格监管的服务。 对以太坊地址来说，ICAP 可以实现以太坊地址的分散和兼容性。

一个 IBAN 由至多 34 个字母数字字符串（不区分大小写）组成，包括国家代码，校验和以及特定国家银行账户标识符的。

ICAP 采用相同的结构，引入代表"Ethereum"的非标准国家代码"XE"，后面跟着两个字符的校验和以及 3 个可能的账户标识符变体：

直接型（Direct）

最多 30 个字母数字字符、big-endian 或 base-36 整数组成，满足以太坊地址的最低有效位。由于此编码适合小于 155 位，因此它仅适用于以一个或多个零字节开头的以太坊地址。就字段长度和校验和而言，其优点是它能与 IBAN 兼容。 示例：XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD（33 个字符长）

基本型（Basic）

除长度为 31 个字符外，与"直接型（Direct）"编码相同。 这使得它可以编码任何以太坊地址，却和 IBAN 字段验证并不兼容。 示例：XE18CHDJBPLTBCJ03FE9 02NS0BP0JVQCU2P（35 个字符长）

非直接型（Indirect）

编码一个标识符，它通过名称注册表提供去解析以太坊地址。利用 16 个字母数字字符，其中包含资产标识符（例如 ETH），名称服务（例如 XREG）和 9 字符名称（例如 KITTYCATS），这是一个人类可以直接阅读的名称。 示例：XE##ETHXREGKITTYCATS（20 个字符长），"##"代表两个计算校验和字符。

我们可以使用 helpeth 命令行工具来创建 ICAP 地址。 以下为使用私钥示例（前缀为 0x 并作为参数传递给 helpeth）：

```
$ helpeth keyDetails -p
0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315

Address: 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9

ICAP: XE60 HAMI CDXS V5QX VJA7 TJW4 7Q9C HWKJ D

Public key:
0x6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38
e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

helpeth 命令为我们构造了一个十六进制以太坊地址和一个 ICAP 地址。 我们示例密钥的 ICAP 地址是：

```
XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD
```

由于我们的示例的以太坊地址恰好以零字节开始，因此可以使用以满足 IBAN 格式的"直接型"ICAP 编码方法进行编码。 你可以自行判断，因为它是 33 个字符长。

如果我们的地址不是从零开始，那么它将被编码为"基础型"编码，也就是 35 个字符长，同时作为 IBAN 格式是无效。

以零字节开始的任何以太坊地址的概率是 1：256. 要生成这样一个类型的地址，在找到一个作为 IBAN 兼容的"直接"编码 ICAP 地址之前，平均需要尝试 256 次生成 256 个不同的随机私钥。

因此，仅会有极少数的钱包支持 ICAP。

十六进制编码大小写字母的校验和（EIP-55）

由于 ICAP 或命名服务部署缓慢，采用以太坊改进建议-55（**EIP-55**）被提出。以下链接将提供详细信息：

https://github.com/Ethereum/EIPs/blob/master/EIPS/eip-55.md

通过修改十六进制地址的大小写，**EIP-55** 为以太坊地址提供向后兼容的校验和。具体思路如下，以太坊地址不区分大小写，所有钱包都应该接受以大写字母或小写字母表示的以太坊地址，在解释上不存在任何区别。

通过修改地址中字母字符的大小写，我们可以传达一个校验和，用来保护地址的完整性，避免输入或读取错误。不支持 EIP-55 校验和的钱包粗略地忽略地址包含混合大写的事实。但那些支持 EIP-55 校验和的钱包可以验证，还能以 99.986％的准确度检测存在的错误。

混合大写字母的编码变化很微妙，最初你甚至不会注意到它。 我们的示例地址是：

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

通过 EIP-55 混合大写校验和，它变成：

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

你能分辨出来吗？ 一些来自十六进制编码字母的字母（A-F）字符现在是大写字母，而另一些则是小写字母。 除非你仔细观察，否则你甚至不会注意到这种差异的存在。

EIP-55 实施起来十分的简单。 我们采用小写十六进制地址的 Keccak-256 哈希。 这个哈希值作为该地址的数字指纹，为我们提供了一个便捷的校验和。 输入（地址）中的任何小改动都会导致哈希值结果（校验和）发生很大变动，从而使我们能够有效地检测存在的错误。然后我们的地址的哈希值被编码为地址中的大写字母。 我们可以以一步步解析它：

1. 哈希小写地址，不带+0x+前缀：

```
Keccak-256("001d3f1ef827552ae1114027bd3ecf1f086ba0f9")

23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9695d9a19d8f673ca991deae1
```

2. 如果哈希值的相应十六进制数字大于或等于 0x8，则将每个字母地址字符大写。 如果我们对地址和哈希进行排列，这将更容易显示：

```
Address: 001d3f1ef827552ae1114027bd3ecf1f086ba0f9

Hash   : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

地址的第四个位置上是字母 d。 哈希值的第四个字符是 6，它小于 8。所以，我们忽略小写字母 d。 我们地址中的下一个字母字符是 f，在第六位。 十六进制散列的第六个字符是 c，它大于 8。因此，我们在地址中大写了 F，依此类推。 正如您所看到的，我们只使用哈希值的前 20 个字节（40 个十六进制字符）作为校验和，因为我们只有 20 个字节（40 个十六进制字符）能被相应地改为大写。

检查自己生成的混合大写地址，看看您是否可以判断出哪些字符，以及它们对应的哈希值中的字符：

```
Address: 001d3F1ef827552Ae1114027BD3ECF1f086bA0F9

Hash   : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

检测 EIP-55 编码地址中的错误

现在，我们来看看 EIP-55 地址如何帮助我们发现错误。 假设我们已经打印出 EIP-55 编码的以太坊地址：

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

现在，我们故意犯下一个阅读该地址时的基本错误。 最后一个字符之前的字符是大写字母"F"。 对于这个例子，我们假设我们误解为大写"E"。 我们在钱包中输入不正确的地址：

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
```

幸运的是，我们的钱包符合 EIP-55 标准！它注意到混合大写字母进而尝试验证地址。它将其转换为小写，并计算校验和哈希值：

```
Keccak-256("001d3f1ef827552ae1114027bd3ecf1f086ba0e9")

5429b5d9460122fb4b11af9cb88b7bb76d8928862e0a57d46dd18dd8e08a6927
```

正如你所看到的，尽管地址只改变了一个字符（事实上，只有一位"e"和"f"相隔了 1 字符间距），地址的哈希值却发生根本性的改变。 这就是哈希值函数的特性，使得它们对校验和非常有帮助！

现在，让我们排列这两个地址并检查大小写：

```
001d3F1ef827552Ae1114027BD3ECF1f086bA0E9

5429b5d9460122fb4b11af9cb88b7bb76d892886...
```

这些都是错误的！其中的几个字母字符被错误地大写了。 现在请记住，大写字母是"正确"校验和的编码。

我们输入的地址大小写与刚刚计算的校验和不匹配，这就意味着地址中的内容发生了变化，并引入了一些错误。

第五章 钱包

钱包

“钱包”一词用来描述以太坊中一些不同的交易。

从较高层次上看，钱包是作为主要用户界面的应用程序。钱包控制对用户资金的访问，管理密钥和地址，追踪余额，以及创建和签署交易。此外，有些以太坊钱包还可以与合约（如代币）进行交互。

更狭义的来说，从程序员的角度来看，“钱包”一词是指用于存储和管理用户密钥的系统。每种“钱包”都有一个密钥管理组件。对于某些钱包来说，这就是全部了。其他一些钱包广义来说更像是“浏览器”的一部分，它们是基于以太坊的去中心化应用程序或“dapps”的接口。以“钱包”为名之下，钱包以各种形式类型混合存在，它们之间并没有明确的区分。

在本节中，我们将把钱包看作私钥的容器，并将其视为管理私钥的系统。

钱包技术概述

在本节中，我们总结用于构建用户友好，安全和灵活的以太坊钱包的各种技术。

关于以太坊的一个常见误解是以太坊钱包包含了以太坊或代币。实际上，钱包只包含密钥。以太坊或其他代币记录在以太坊区块链中。用户通过使用钱包中的密钥来签署交易从而控制网络上的代币。从某种意义上说，以太坊钱包是一个钥匙串（keychain）。

提示	以太坊钱包包含密钥，而不是以太坊或代币。每个用户都有一个包含密钥的钱包。钱包是真正包含成对私钥/公钥的钥匙串（参见[private_public_keys]）。用户使用密钥签署交易，从而证明他们拥有以太坊。以太坊储存在区块链上。

钱包主要有两种类型，通过它们包含的密钥是否彼此相关来区分。

第一种类型是非确定性钱包（nondeterministic wallet），其中每个密钥都是从随机数独立生成的。密钥之间并不相互关联。这种类型的钱包也被称为 JBOK 钱包，这个词语来自于"Just a Bunch Of Keys"（一堆私钥）。

第二种类型的钱包是确定性钱包（deterministic wallet），其中所有密钥都来自单个主密钥，主密钥也被称为种子。此类钱包中的所有密钥都是相互关联的，如果有原始种子，密钥可以再次生成。确定性钱包中使用了许多不同的密钥推导方法。最常用的推导方法采用树状结构，被称为分层确定性钱包或 HD 钱包。

确定性钱包是从种子初始化的。为了更容易使用，种子被编码为英文单词（或其他语言中的单词），也称为助记码词汇。

接下来的几节将高度介绍这些技术。

非确定性（随机）钱包

在首个以太坊钱包（在以太坊预售时推出）中，钱包文件存储了一个随机生成的私人密钥。这些钱包正在被确定性的钱包取代，因为它们管理、备份和导入很麻烦。随机密钥的缺点是，如果你生成了许多密钥，你必须保留所有密钥的副本。每个密钥都必须备份，否则如果钱包变得不可访问，则其控制的资金将不可撤销地丢失。此外，以太坊地址通过将多重交易和地址相互关联重用来降低隐私。0 型非确定性钱包是一种差劲的选择，特别是如果您想避免地址重用，这意味着管理许多密钥，需要频繁备份。

许多以太坊客户端（包括 go-ethereum 或 geth）使用 keystore(密钥库)文件，这是一个 JSON 编码文件，其中包含一个（随机生成的）私钥，由一个密码加密以提高安全性。JSON 文件内容如下所示：

```
{
    "address": "001d3f1ef827552ae1114027bd3ecf1f086ba0f9",
    "crypto": {
```

```
        "cipher": "aes-128-ctr",
        "ciphertext":
"233a9f4d236ed0c13394b504b6da5df02587c8bf1ad8946f6f2b58f055507ece",
        "cipherparams": {
            "iv": "d10c6ec5bae81b6cb9144de81037fa15"
        },
        "kdf": "scrypt",
        "kdfparams": {
            "dklen": 32,
            "n": 262144,
            "p": 1,
            "r": 8,
            "salt":
"99d37a47c7c9429c66976f643f386a61b78b97f3246adca89abe4245d2788407"
        },
        "mac":
"594c8df1c8ee0ded8255a50caf07e8c12061fd859f4b7c76ab704b17c957e842"
    },
    "id": "4fcb2ba4-ccdb-424f-89d5-26cce304bf9c",
    "version": 3
}
```


密钥库格式使用密钥衍生函数（KDF），也称为密码扩展算法，可防止对密码加密的暴力破解，字典攻击或彩虹表攻击。简而言之，私钥没有直接由密码加密。相反，密码通过重复哈希来延长。哈希函数重复 262144 轮，可以在密钥库 JSON 中作为参数 crypto.kdfparams.n 看到。试图暴力破解密码的攻击者必须对每一个企图破解的密码进行 262144 轮哈希，这足以减缓攻击行为，并因为足够复杂和长度的密码而变得不可行。

有许多软件库可以读写密钥库格式，例如

JavaScript 库 keythereum： https://github.com/ethereumjs/keythereum

提示	对于简单测试以外的任何情况都不鼓励使用非确定性钱包。它们备份和使用太麻烦了。取而代之，请使用带有助记词种子的基于行业标准的 HD 钱包进行备份。

确定性（种子）钱包

确定性钱包，或“种子”钱包是包含私钥的钱包，这些私钥都是通过使用单向哈希函数从公共种子衍生而来的。种子是随机生成的数字，并含有比如索引号码或可生成私钥的“链码”数据（参阅 HD 钱包（BIP-32 / BIP-44））。在确定性钱包中，种子足以恢复所有衍生密钥，因此在初始创建时单个备份就足够了。种子对于钱包的导出或导入也是够用的，这就允许用户在不同的钱包之间轻松迁移所有密钥。

HD 钱包（BIP-32 / BIP-44）

确定性钱包的开发使得从单个“种子”中衍生出许多密钥变得容易。基于比特币 BIP-32 标准的 HD 钱包是最先进的确定性钱包。HD 钱包包含了树形结构衍生的密钥，即一个父密钥可以生成一系列的子密钥，每个子密钥可以生成一系列孙密钥，以此无穷类推。这个树形结构参见图，HD 钱包：从单个种子生成的密钥树.

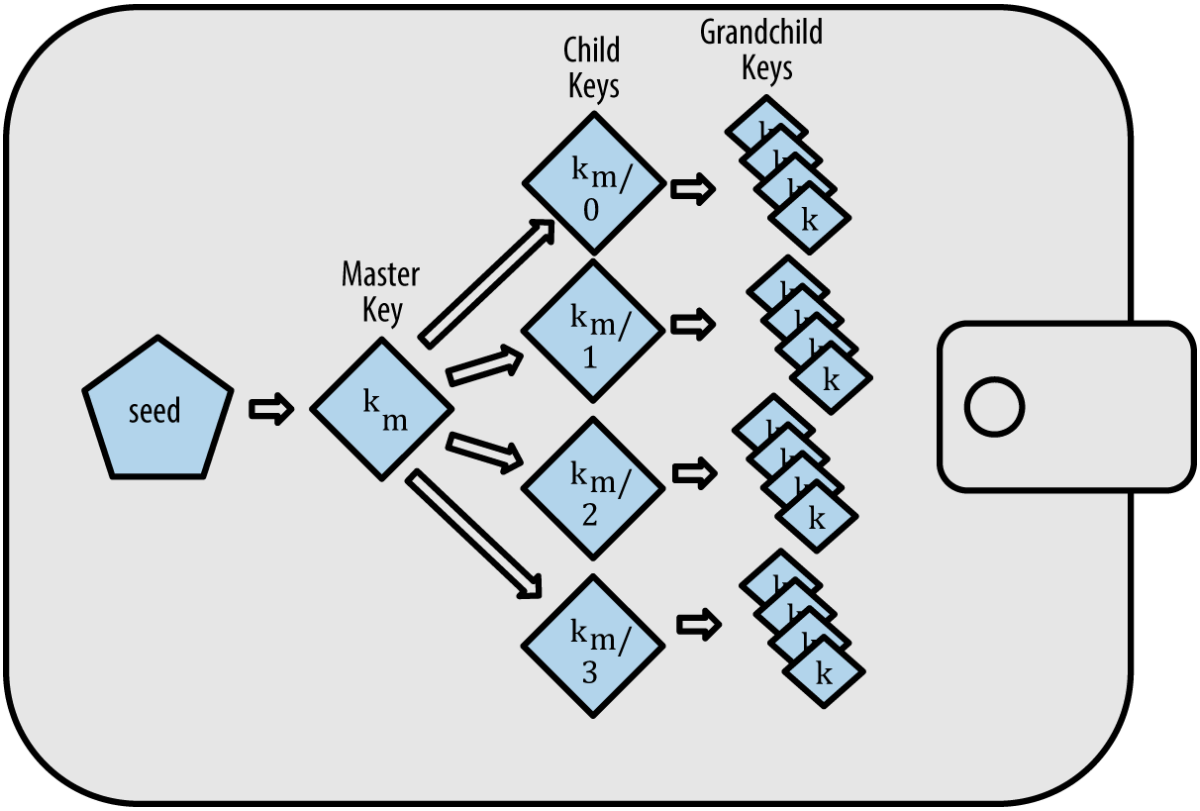


图 1. HD 钱包：从单个种子生成的密钥树

与随机（非确定性）密钥相比，HD 钱包具有两大优势。首先，树结构可以用来表达额外的组织含义，例如，子钥的某个分支用于收款，另一个分支用于付账。密钥的分支结构也可用于企业设置，可将不同分支分配给部门、子公司、特定职位或会计类别。

HD 钱包的第二个优点是用户可以创建一系列公钥，而无需访问相应的私钥。这使得 HD 钱包可以用在不安全的服务器上，或者用于仅查看或仅接收的地方，因为这些地方不需要使用私钥来花费资金。

种子和助记词码（BIP-39）

HD 钱包是管理大量密钥和地址的强大机制。如果它们通过易于转抄，导出、导入钱包的一串英文单词（或其他语言的单词）方式来标准化的创建种子，那么它们就会更加易用。这被称为助记词，标准由 BIP-39 定义。今天，许多以太坊钱包（以及其他加密货币的钱包）都使用此标准，并且使用可互用的助记词来导入和导出种子以进行备份和恢复。

我们从实际的角度来看这个。下列哪种种子更易于转录、在纸上记录、无误地读取、导出并导入另一个钱包？

确定性钱包的种子，十六进制形式

```
FCCF1AB3329FD5DA3DA9577511F8F137
```

确定性钱包的种子，12 个单词的助记词

```
wolf juice proud gown wool unfair

wall cliff insect more detail hub
```

钱包的最佳实践

随着加密货币钱包技术的成熟，一些行业标准的也开始出现了。 这些标准的出现使得钱包更有互通性，容易使用，安全而且更具有灵活性。这些标准适合钱包支持使用一套助记码词组衍生出多个不同类型加密货币的密钥。这些公共标准如下所列：

- 助记码词组（Mnemonic code words），基于 BIP-39
- HD 钱包（HD wallets），基于 BIP-32
- 多用途 HD 钱包结构（Multipurpose HD wallet structure），基于 BIP-43
- 多货币及多账户钱包（Multicurrency and multiaccount wallets），基于 BIP-44

这些标准也许在将来的开发中有所改变或被淘汰，但是就现在而言它们构成了一套连锁技术，成为了大多数加密货币事实上使用的钱包标准。

这些标准被一系列软件钱包和硬件钱包广泛采纳，使得这些钱包间具备互操作性。用户导出其中任意一款钱包生成的助记码词组，导入另外一款钱包同样能够恢复所有的交易记录，密钥和地址。

支持这些标准的软件钱包包括(按照字母排序)Jaxx，MetaMask，MyEtherWallet (MEW)。支持这些标准的硬件钱包包括（按照字母排序） Keepkey， Ledger， 和 Trezor。

下面的章节将逐项检查这些技术的细节。

提示	如果你在实现一个以太坊钱包，它应该被创建为 HD 钱包，使用编码成助记码词组的种子作为备份，并遵循后续描述的 BIP-32， BIP-39，BIP-43 和 BIP-44 标准。

助记码词组（Mnemonic Code Words）（BIP-39）

助记码词组是一组单词序列，它们表示（编码）了一个随机的数字即种子用来派生出一个确定性钱包。这个序列足够重建种子，由种子重建钱包以及所有衍生出的密钥。当第一次创建使用助记码词组实现的确定性钱包时，系统会为用户显示由 12 或 24 个英文单词组成的序列。这个序列就是钱包的备份，可以在同样或者兼容的钱包应用中恢复和重建所有的密钥。相比随机的助记码词组使得备份钱包变得简单，因为对用户来说这些词组容易辨识并较容易正确的誊写。

提示	助记码词组通常容易和“脑钱包（brainwallets）”相混淆。他们是不同的，最主要的区别是脑钱包由用户选择的词组，而助记码词组是由钱包随机生成并展示给用户的。这点重要的区别使得助记码词组钱包更加的安全，因为人类大脑不是很好的随机性来源。

助记码在 BIP-319 中定义。注意 BIP-39 是助记码标准的一种实现。有另外一种标准，使用不同的词组，被 Electrum 比特币钱包使用并且早于 BIP-39。BIP-39 是被 Trezor 硬件钱包背后的公司所提议并且兼容 Electrum 的实现。不过 BIP-39 目前已获得业界几十种可互操作钱包实现的广泛支持，应该被认为是事实上的业界标准。另外 BIP-39 可以被用来生成多币种钱包并支持以太坊，但 Electrum 种子并不可以。

BIP-39 定义了助记码和种子的创建，我们在这里分 9 步描述。为清楚起见，这个过程被分成了两部分：第一到第六步展示的是生成助记码词组　第七到第九步展示的是从助记码到种子

生成助记码词组

助记码词组是钱包依照 BIP-39 定义的标准流程自动生成的。钱包从一点信息熵开始，增加校验码，然后将信息熵关联到一组单词：

1. 创建一段 128 或 256 位的二进制序列（信息熵）。
2. 取这段序列 SHA256 哈希值的前 x 位（x=熵-长度/32）作为校验码。
3. 将校验码添加到序列后端。
4. 将序列进行拆分，每段 11 位。
5. 依据提前定义好的包含 2048 个单词的字典将每段映射为一个单词。
6. 助记码词组就是这些单词构成的序列。

生成信息熵并编码为助记码词组　展示了信息熵如何被用来生成助记码词组的。

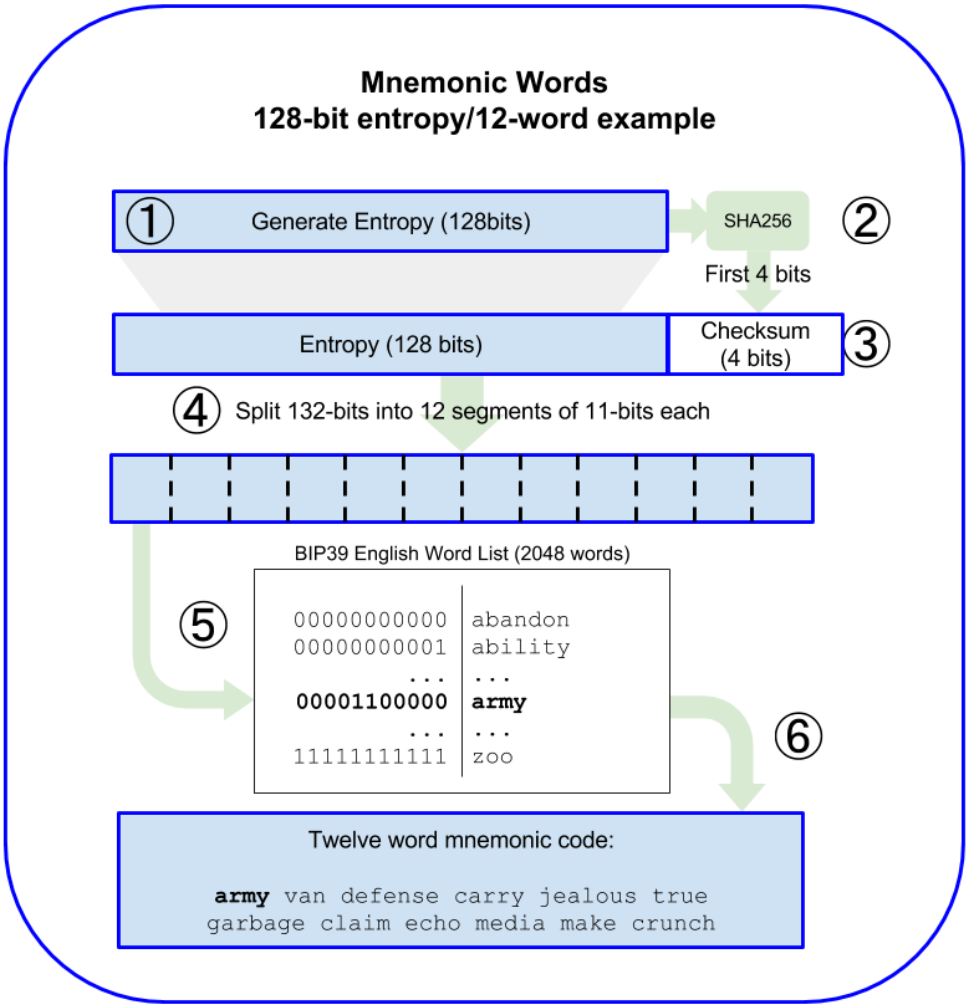


图 2. 生成信息熵并编码为助记码词组

助记码词组：信息熵和助记码数量　展示了信息熵大小与助记码词组数量的关系。

信息熵（bits）	校验码（bits）	信息熵 + 校验码（bits）	助记码数量（words）
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21

信息熵（bits）	校验码（bits）	信息熵 + 校验码（bits）	助记码数量（words）
256	8	264	24

表 1. 助记码词组：信息熵和助记码数量

从助记码到种子

助记码词组代表了 128 位或者 256 位的信息熵。通过 PBKDF2 密钥伸展功能信息熵衍生出一个更长的（512 位）种子。这个种子随后被用来创建确定性钱包以及衍生相关密钥。

密钥伸展功能需要两个参数：助记码和盐（salt）。密钥伸展的目的是为了让创建彩虹表更加困难并以此防止暴力破解。在 BIP-39 标准中，盐有另外的用途—它允许引入密码口令（passphrase）作为一种额外的保护种子的安全机制，我们会在 BIP-39 中可选的密码短语更加详细的描述。

第七到第九步从前面描述的过程，生成助记码词组后继续：

PBKDF2 密钥伸展的第一个参数是第 6 步生成的助记码。

第二个参数是一段盐。盐是由固定字符串"mnemonic"串联一段可选的由用户提供的密码短语字符串构成。

PBKDF2 使用 HMAC-SHA512 算法进行 2048 轮哈希，生成一个 512 位的值作为最终输出. 这个 512 位的值就是种子。

[fig_5_7]展示一段助记码词组如何生成一个种子。

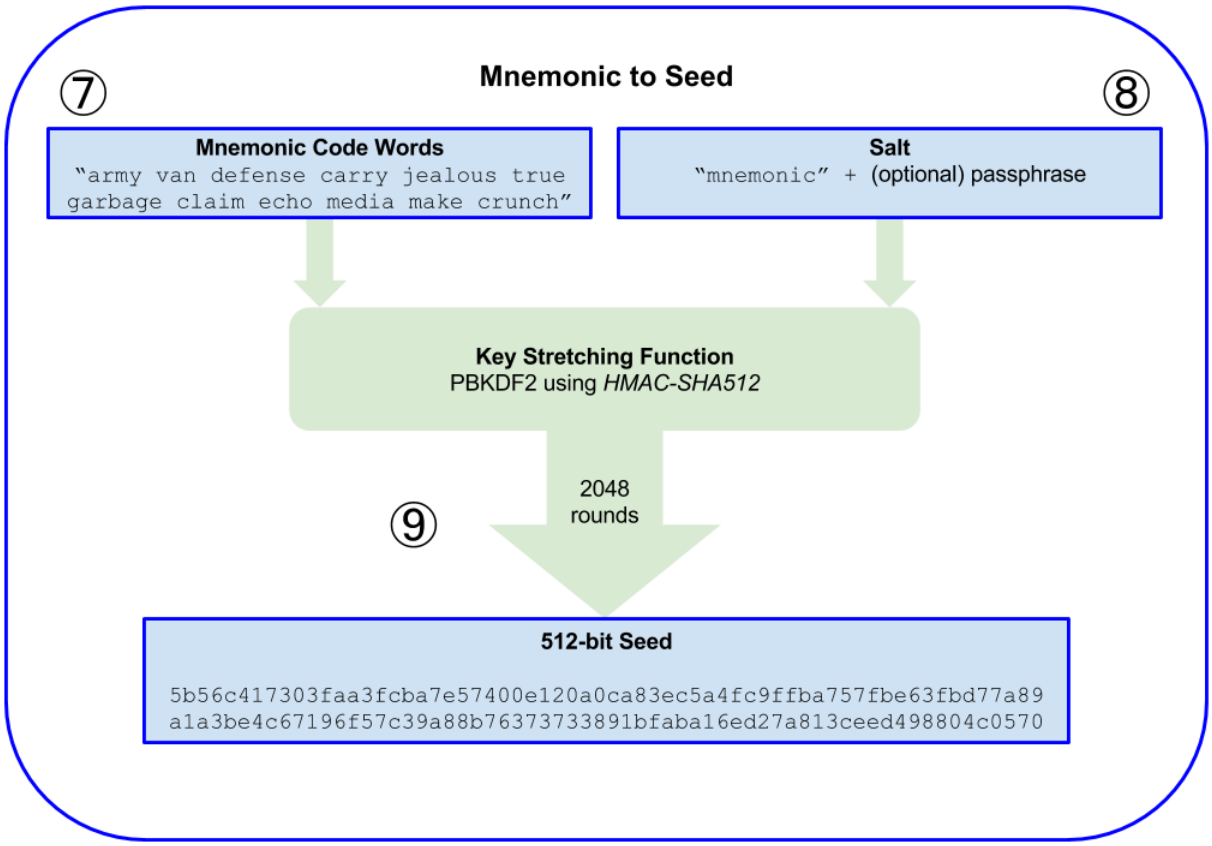


图 3. 从助记码到种子

提示	密钥伸展功能，以及它的 2048 轮哈希运算，在一定程度上是一种有效的针对助记码或密码短语暴力破解保护。它使得尝试几千种密码短语和助记码词组组合比较耗费计算资源，但同时可能衍生出的的种子数量是巨大的(2^{512})。

表格 #mnemonic_128_no_pass, #mnemonic_128_w_pass, and #mnemonic_256_no_pass 展示了一些助记码组合和它们能够生产的种子（不用密码短语）。

信息熵输入（128 位）	0c1e24e5917779d297e14d45f14e1a1a
助记码（12 个词）	army van defense carry jealous true garbage claim echo media make crunch
密码短语	（无）
种子（512 位）	5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39 a88b76373733891bfaba16ed27a813ceed498804c0570

表 2. 128 位信息熵助记码，不使用用密码短语，生成种子

信息熵输入（128 位）	0c1e24e5917779d297e14d45f14e1a1a
助记码（12 个词）	army van defense carry jealous true garbage claim echo media make crunch
密码短语	SuperDuperSecret
种子（512 位）	3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0 715861dc8a18358f80b79d49acf64142ae57037d1d54

表 3. 128 位信息熵助记码，使用密码短语，生成种子

信息熵输入（256 位）	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
助记码（24 个词）	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
密码短语	（无）
种子（512 位）	3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e5 5f1e0deaa082df8d487381379df848a6ad7e98798404

表 4. 256 位信息熵助记码，不使用密码短语，生成种子

BIP-39 中可选的密码短语

BIP-39 标准允许在派生种子的时候使用一个可选的密码短语。如果不使用密码短语，助记码会被以字符串+"mnemonic"+为盐伸展，生成 512 位的种子。如果使用了密码短语，同样的助记码被伸展后会生成一个_不同_的种子。实际上，对于同样的助记码任何可能的密码短语都会生成不同的种子。本质上来看并没有“错误”的密码短语。所有的密码短语都是有效

的并且会导向不同的种子，构成一组数量巨大（2512）潜在未初始化的钱包。数量之多使得没有实际的暴力破解或者偶然猜测获得使用的可能，只要密码短语有足够的长度和复杂性。

提示	BIP-39 中没有“错误的”密码短语，每一个密码短语都能导向一些钱包，它们除非之前用过否则是空的。

可选的密码短语提供了两个重要的特性：

- 可以使助记码词组本身无效的第二保护因素（记忆下来的东西），防止助记码备份被窃贼破解。
- 一种貌似可信的推诿手段或者“被威胁钱包”，使用某个特定的助记短语导向一个资金较小的钱包用来误导攻击者，从而保护真正存有大量资金的钱包。

但是，需要注意的是使用助记短语同样带来了一点损失的风险：

- 如果钱包的所有者丧失行为能力或者死亡，导致没有人知道助记词，种子会变得毫无用处，存在钱包里的所有资金将永远丢失。
- 相反的，如果拥有者将助记短语和种子备份在同样的地方，那么第二保护因素就失去了意义。

尽管助记短语非常有用，但是它们应该仅在有非常小心的备份和恢复计划的情况下使用，考虑到其比所有者存在更长时间的可能，允许他/她的家人恢复加密货币资产。

使用助记码词组

BIP-39 的实现库有多种编程语言版本：

- python-mnemonic

  提出 BIP-39 的中本聪实验室团队的参考实现版本，使用 python 开发。

- Consensys/eth-lightwallet

  轻量级 JS 以太坊钱包提供给节点和浏览器（使用 BIP-39）

- npm/bip39

  比特币 BIP39 的 JavaScript 实现：助记码用来生成确定性密钥

还有一个使用可脱机网页实现的 BIP-39 生成器，对于测试和实验来说非常有用。 一个可脱机的 BIP-39 生成器网页 展示了这个可脱机页面，它能够生成助记码，种子以及扩展出私钥。

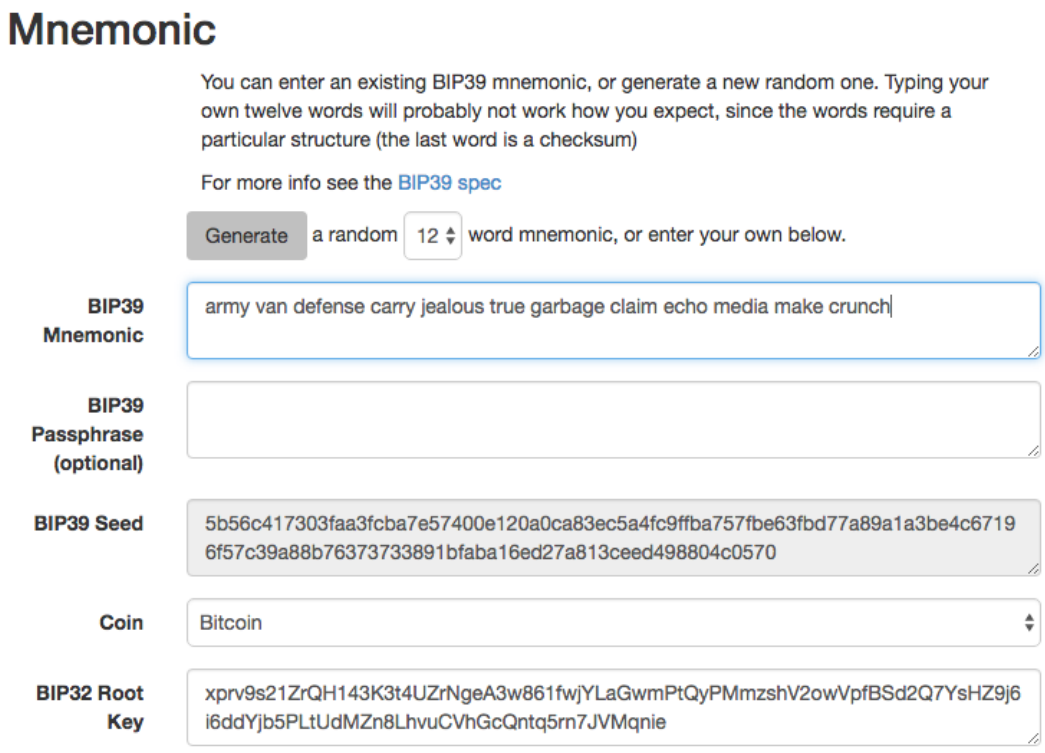


图 4. 一个可脱机的 BIP-39 生成器网页

这个页面(https://iancoleman.github.io/bip39/) 可以在浏览器内离线使用，或者在线访问。

由种子创建一个 HD 钱包

HD 钱包是从一个根种子创建的，它可以是 128-bit,256-bit 或者 512-bit 的随机数。通常情况下，由前面章节所述，这个种子通过助记码生成。

HD 钱包里的每个私钥都是从根种子生成决定的，因此在任何兼容的钱包里可以从这个根种子重新创建整个 HD 钱包。这样备份，保存，导出导入一个包含成千上万个私钥的 HD 钱包都很容易，只要简单地转移一下用来生成根种子的助记码。

分层确定性钱包(BIP-32)和路径(BIP-43/44)

大部分 HD 钱包遵循 BIP-32 标准，这实际上已经成为了确定性密钥生成的行业标准。详细的规格说明请参照：

https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

这里我们不详细展开 BIP-32，只讨论跟理解钱包使用相关的几点。网上各个软件库里有多个可操作的 BIP-32 应用。

Consensys/eth-lightwallet 用于节点和浏览器的轻量级 JS 以太坊钱包下面链接是一个独立的网页生成器可用于测试和试验 BIP-32.

http://bip32.org/

提示	这个独立的 BIP-32 生成器不是一个 HTTPS 网站。提醒您这个工具并不安全。只用于测试用途。不要在实际应用（使用真实资金时）中使用这里生成的密钥

扩展公钥和私钥按照 BIP-32 的定义，用来生成子密钥的父密钥被叫做扩展密钥。如果是私钥，那就是扩展私钥，用前缀 xprv 标识：

```
xprv9s21ZrQH143K2JF8RafpqtKiTbsbaxEeUaMnNHsm5o6wCW3z8ySyH4UxFVSfZ8n7ESu7fgir8imbZKLYVBxFPND1pniTZ81vKfd45EHKX73
```

扩展公钥用前缀 xpub 标识：

```
xpub661MyMwAqRbcEnKbXcCqD2GT1di5zQxVqoHPAgHNe8dv5JP8gWmDproS6kFHJnLZd23tWevhdn4urGJ6b264DfTGKr8zjmYDjyDTi9U7iyT
```

HD 钱包一个非常有用的特点就是在没有私钥的情况下可以用父公钥导出子公钥。这给我们得到子公钥的两个途径：通过子私钥或者直接从父公钥。

所以一个扩展公钥可以用来导出 HD 钱包密钥数结构该分支下的所有公钥(且只有公钥)。在服务器或应用只有一份扩展公钥且没有私钥的情况下，这个特点提供了一条非常安全的公钥部署捷径。

这种方法可以生成无限个公钥和以太坊地址，但不能动用发往这些地址的资金。同时，在另一个更安全的服务器上，扩展私钥可以导出所有对应的私钥给交易签名，运用资金。

这种方案的一个常见应用是安装一个扩展公钥在电商用网站服务器上。网站服务器用公钥的推导函数给每一笔交易生成一个以太坊地址（比如给客户购物车）。网站服务器上不会有任何存在被盗风险的私钥。没有 HD 钱包时，唯一的办法是在物理隔绝服务器上生成成千上万个以太坊地址然后把它们预加载到电子商务服务器上。那样既麻烦而且还要定期维护，保证电商服务器上的密钥不会用完。

这种方案的另一个常见应用是做冷存储或者硬件钱包。这种场景里，扩展私钥可以存在硬件钱包里，而扩展公钥可以在线保存。需要时用户可以随时生成“收币”地址，而私钥安全地离线保存。需要动用资金时，用户可以通过离线的以太客户端或者用硬件钱包服务来签名使用扩展私钥。

增强子密钥推导

从一个 xpub 推导一组公钥的功能非常有用，但是也有潜在风险。Xpub 并没有访问子私钥的权限。但是，xpub 包含了链码，如果一个子私钥已知或者说泄露，两者一起就可以推导

出其它所有的子私钥。泄露一个子私钥，加上父链码，就泄露了所有的子私钥。更糟糕的是，子私钥加上父链码可以用作推算父私钥。

为了防范这种风险，HD 钱包采用了另一种叫做“增强推导”的推导函数，它隔绝了父公钥和子链码的关系。增强推导函数使用父私钥推导子链码，而不是用父公钥。这样通过不影响父私钥和同级子私钥的链码，在父/子结构中间加了一道“防火墙”。

简单的说，如果你用 xpub 的方便性推导一组公钥，同时不想泄露链码，就应该用增强推导，而不是用常规的推导。最正确的做法是，从主密钥推导第一级子密钥时，永远使用增强推导，避免泄露主密钥。

常规和增强推导的索引数

BIP-32 推导函数使用了 32-bit 整数作为索引数。为了简便的区分从常规推导和增强推导生成的密钥，这个索引数分成两个范围，0 到 231 - 1 (0x0 to 0x7FFFFFFF)只用于常规推导。231 到 232 - 1(0x80000000 to 0xFFFFFFFF)之间的索引数只用于增强推导。因此，如果索引数小于 231，子密钥是常规推导，如果索引数大于或等于 231，子密钥是增强推导。

为了便于索引数显示和阅读，增加子密钥的索引数显示从零开始，但多加了一串符号。因此第一个常规推导的子密钥显示为 0，第一个增强推导的子密钥（索引数 0x80000000）显示为 0&#x27；然后按顺序，第二个增强推导的子密钥索引数为 0x80000001，显示为 1&#x27；等等。你看到 HD 钱包索引数 i&#x27，实际表示 231+i。

HD 钱包密钥标识（路径）

HD 钱包里的密钥约定用“路径”标识，树结构里的每一层用“/”分开(参见 HD wallet path examples)。主私钥导出的私钥用“m.”开头，主公钥导出的公钥用“M.”开头。因此，主私钥的第一个子私钥是 m/0。第一个子公钥是 M/0。子私钥的第二个孙私钥是 m/0/1，等等。

一个密钥的“祖先”可以从右到左读出，直到读到主密钥。比如，标识符 m/x/y/z 表示这个密钥是 m/x/y 的第 z+1 个子密钥，而 m/x/y 是密钥 m/x 的第 y+1 个子密钥，m/x,是 m 的第 x+1 个子密钥。

HD 路径	密钥描述
m/0	从主私钥(m) 推导出的第一代的第一个子私钥
m/0/0	从第一代子密钥(m/0)推导出的第二代第一个孙私钥
m/0’/0	从第一代增强密钥 (m/0’)推导出的第二代第一个孙密钥

HD 路径	密钥描述
m/1/0	从第一代的第二个子密钥推导出的第二代第一个孙密钥
M/23/17/0/0	从第一代第 24 个子公钥的第 18 个孙公钥的第一个曾孙公钥推导出的第一个玄孙公钥

表 7. HD 钱包路径示例

遍历 HD 钱包的树结构

HD 钱包的树结构带来了极大便利性。每个扩展父密钥可以有 40 亿子密钥：20 亿常规推导的和 20 亿增强推导的子密钥。每个子密钥又可以有 40 亿个子密钥，诸如此类。树结构可以按需多层衍生，包含无限个层级。有这些便利性的同时，遍历无限层树结构也非常困难。尤其困难的是在不同 HD 钱包应用之间转移时，不同分支和子分支的组织形式变得无穷无尽。

有两个 BIP 提案提出创建 HD 钱包树结构的协议，这提供了解决这类复杂问题的方案。BIP-43 提出用第一个增强子密钥索引作为特殊标识表明树结构的目的。遵循 BIP-43，HD 钱包应当只用树结构的一层分支，用索引数定义目的来标识结构和命名树结构其它部分。比如，一个 HD 钱包只用分支 m/i&#x27，/表明具体目的，该目的用索引数"i."标识。

在这之上， BIP-44 提出在 BIP-43 “purpose”44’ 下的多币种多账户结构。所有按照 BIP-44 协议的 HD 钱包只用数结构的一个分支 m/44’/。

BIP-44 描述了五层树结构的定义：

```
m / purpose' / coin_type' / account' / change / address_index
```

第一层 "purpose" 总是 44’。第二层"coin_type"描述数字货币类别，允许多币种 HD 钱包里每一种币在第二层下面有自己的树分支结构。多种数字货币定义在一个叫 SLIP0044 的协议文件里。

https://github.com/satoshilabs/slips/blob/master/slip-0044.md

举几个例子：以太币是 m/44’/60’，经典以太币是 m/44’/61’，比特币是 m/44’/0’，所有币种的测试网络是 m/44’/1’。

树的第三层是“account，”允许用户把钱包分割为多个独立的子账户，用于会计或者组织目的。比如，一个 HD 钱包可以有两个以太币账户：m/44’/60’/0’ 和 m/44’/60’/1’. 每个账户都是该树分支的根。

BIP-44 最初是为比特币创建的，它包含了和以太坊不相干的一个异常。路径的第四层 "change,"，一个 HD 钱包里有两个树分支，一个用于产生收币地址，一个用于更改地址。在

以太坊里只用到收币地址，而没有更改地址的说法。要注意的是前面几层都用增强推导，这一层只用常规推导。这允许这层树结构可以导出扩展公钥用在不安全的环境中。可使用的地址从 HD 钱包第四层树结构的子密钥推导出来，这样树结构第五层就是”地址指针”。比如第一个以太账户的第三个收币地址是 M/44’/60’/0’/0/2。 BIP-44 HD wallet structure examples 给出了几个例子。

HD 路径	密钥描述
M/44'/60'/0'/0/2	第 1 个以太币账户的第 3 个收币地址公钥
M/44'/0'/3'/1/14	第 4 个比特币账户的第 15 个更改地址的公钥
m/44'/2'/0'/0/1	用于交易签名的莱特币主账号里第 2 个私钥

表 8. BIP-44 HD 钱包结构示例

第六章 交易

交易

交易就是由外部自有账户发起的签名信息，由以太坊网络传输，并记录保存在以太坊区块链上。这个基本定义隐藏了很多出乎意料而且十分有趣的细节。从另外一个角度来看，我们可以发现这些交易是唯一可以触发状态改变的因素，或者可以利用它们在以太坊虚拟机（EVM）上执行智能合约。以太虚拟机是一个全局单例状态机，在上面执行交易相当于启动这个机器或者改变它的运行状态。合约不是自动执行的。以太坊网络不能自主运行，一切功能的运作从交易开始。

在本章中，我们将一层层论述交易这个概念，展示这个过程是如何工作的，并检验各个细节。请注意，本章的绝大部分内容是针对有兴趣处理低级别交易的人群，也许是因为他们正在编写一个钱包 APP；如果您很乐于使用手头现存的钱包应用的话，那您就没必要操心这些了，尽管，您可能会觉得这些过程细节很有趣！

交易的结构

首先咱们得了解一下交易的基本结构，它是在以太坊网络上按先后顺序处理和传输的。接收到序列化的交易信息之后，每一个客户端和应用程序会用自身的内部数据结构存储这些信息，也许可能会加上一些本来不存在于这个序列化交易网络的元数据，序列化网络是交易的唯一标准形式。

一个交易是一个序列化二进制信息，其中包括以下数据：

Nonce

一个序列号，由原始 EOA 发送，用来防止消息重复发送。

gas price

交易发起人愿意接受的价格。

gas limit

交易发起人愿意为此交易支付的最大数额。

Recipient

交易接收方的以太币地址。

Value

被发送至目的地的以太币的数量。

Data

可变长度二进制数据有效负载。

V, R, S

原始 EOA 的 ECDSA 数字签名的三个组成部分。

在以太坊网络中，交易信息的结构是使用递归长度前缀（RLP）编码方案进行序列化的，该编码方案专门为以太币网络中简捷，字节完善的数据序列化而创建，所有的数字都被编码为长度为 8 的倍数的大端整数。

需要注意的是，为了清楚起见，这里显示了字段标签，但是他们不是序列化数据的一部分，交易的序列化数据是包含字段值 RLP 编码的。通常，RLP 不包含任何字段分隔符或标签。RLP 长度前缀用于标识每个字段的长度。任何超过定义长度的内容都属于结构中的下一个字段。

虽然这是传输中的实际交易结构，但是大部分都含有内部表示法和用户界面可视化这些附加信息，这些信息源自于交易或者区块链上。

例如，您可能会注意到在识别交易发起者 EOA 的地址的数据中没有“from(来自于)”这个单词。这是因为 EOA 的公钥可以从 ECDSA 签名的 v,r,s 组件中导出。反过来，地址也可以由公钥导出。当您看到一个显示出“from”字段的交易时，他是用可视化软件添加的。其他被客户端软件添加的元数据包括区块号（一旦被挖掘出来就会包含在区块链中），还有交易 ID（计算出的哈希值）。同样地，这个数据来自于交易但是并不是交易信息本身的一部分。

nonce

nonce 是交易中最重要同时也最难理解的成分之一，在黄皮书中的定义如下（参见参考文献）：

nonce：一个标量，数值上等同于从这个地址发起的交易的数量，亦或者，在有关联代码的账户的情况下，这个账户创造的合约数量。

严格的说，nonce 是发起交易的地址的属性；也就是说它只在发送地址中的内容中有意义。然而，nonce 并不作为账户状态的一部分显著的存在于区块链中。相反，它是通过计算了源自于交易发起地址的已确认交易数目而得出的。

在这两个场景中，交易计数的 nonce 存在与否很重要：

1. 交易是否可用的特征被包含在创建交易的先后顺序中。

2. 防止交易信息复制。

看我们来分别看看这两个示例场景：

1. 假设您希望进行两笔交易，您有一笔比较重要的 6 个 ETH 的支付，和另一笔价值 8 个 ETH 的支付。您首先签名和广播这个 6 个 ETH 的支付，因为这个是更重要的一笔交易，然后您进行另一笔交易，也就是 8 个 ETH 的那一个。但是很遗憾，您忽略了您的账户仅剩下 10 个

ETH 的这一事实，所以网络中不能同时接受这两笔交易：其中有一个势必要失败。因为您先发起了 6 个 ETH 的交易，所以显而易见，您希望 6 个 ETH 的交易通过而 8 个 ETH 的交易被拒绝。然而，在以太坊这个分布式的系统中，节点是可以以任意的顺序接受到交易信息的，不能保证一个特定节点先接受到交易信息然后再传播。 因此，几乎可以肯定的是，一些节点首先接受到 6 个 ETH 的交易广播而另一些节点首先接受到 8 个 eth 的交易广播。可见如果要是没有 nonce 的话，那么哪一个交易被接受和拒绝都会是随机的。然而，在考虑到 nonce 的情况下，第一个您发送的交易（6 个 ETH）具有 nonce, 如 3， 第二个交易（8 个 ETH）则具有下一个 nonce—4，因此后者将被忽略直到 0 到 3 的 nonce 值被优先处理完成，即使后者先被某些节点接收，放心吧！

2. 现在假设您有个含有 100ETH 的账户，某天您发现网上有人出售您想购买的商品，而且店家接受以太坊支付。然后，您发给店家 2 个 ETH，店家发货给您。为了完成这次交易，您发起了一个从您的账户转移 2ETH 至他们账户的交易信息，然后将其广播到以太坊网络，以供全网验证并记录在区块链中。那么如果没有 nonce 值的情况，那么第二次发送 2ETH 至相同地址的第二次广播会和第一次交易完全一样，这意味着任何在以太坊网络中看到您的交易的人（任何人，包括接收人和别有用心的人）都可以一次又一次的“重播”这次交易，通过复制和粘贴您最初的交易重新广播到以太坊网络直到您所有的以太币用完。然而，如果 nonce 被包含在交易数据中，即使每次都是相同数量的 ETH 被发送至相同的接收人地址，每次交易都是独特的。因此，通过将递增的 nonce 值作为交易的一部分，任何人都不可能“复制”您所完成的交易和支付。

总而言之，与比特币协议的“未使用交易输出”（UTXO）机制相比，我们应该注意到，对于基于账户的协议，使用 nonce 值实际上是至关重要的。

保持 Nonce 的记录

在实践中，nonce 值是对来源于一个账户已确认（即上链）交易的实时计算数值。要了解 nonce 是什么，您可以查询区块链，例如通过 web3 界面。在运行 MetaMask 的浏览器中打开 JavaScript 控制台,或者使用 truffle 控制台命令来访问 JavaScript web3 库,然后输入：

```
>
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f"
)
40
```

提示	nonce 是一个基于零的计数器，这意味着第一笔交易的 nonce 值是 0， 在这个示例中，交易计数是 40，意味着从 0 到 39 已经被识别，下一个交易的 nonce 值需要为 40。

您的钱包将会记录它管理的每一个地址的 nonce；实现起来相当简单，只要您从单个节点发起交易。假设您自己正在编写钱包软件或者是其他能够发起交易的应用，您怎样能保持对 nonce 的记录呢？

当您创建一个新的交易的时候，按次序分配下一个 nonce。但是直到确认之前，这次交易不会算在交易总数（getTransactionCount）当中。

警示	在使用 getTransactionCount 计数待定的交易的时候要当心，因为您如果连续发送一些交易可能会遇到一些问题。

让我们来看个例子：

```
>
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f",
\

"pending")

40

> web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \

"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01,
"ether")});

>
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f",
\

"pending")

41

> web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \

"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01,
"ether")});
```

```
>
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f",
\
"pending")
41
> web3.eth.sendTransaction({from: web3.eth.accounts[0], to: \
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01,
"ether")});
>
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f",
\
"pending")
41
```

正如您所见，我们发起的第一个交易计数到了 41，显示为待定的交易。当我们快速连续的发送三个交易时，getTransactionCount 命令没有计数它们。即使您可能预计在内存池中有三笔待定交易，它也只计数了一笔。如果我们等待几秒钟来允许网络通信稳定下来，getTransactionCount 命令将会返回到预期的 nonce 数值。但在此期间，尽管有多笔交易待定，它也不会响应。

当您构建能产生交易的应用程序的时候，不能取决于 getTransactionCount 来处理待定的交易。只有当待定的交易数目和已确认的交易数目相等时(即所有的未解决交易都已确认时)，您才能够信任 getTransactionCount 的输出值来开始 nonce 的计数。此后，记录应用程序中的 nonce 直到每个交易都被确认。

Parity’s JSON RPC 界面提供的 parity_nextNonce 函数，可以帮助回到交易中应该使用到的下一个 nonce 值。即使当您快速连续的创建几个交易而没有确认它们时，parity_nextNonce 函数也可以正确的计算 nonce 值。

```
$ curl --data '{"method":"parity_nextNonce", \
  "params":["0x9e713963a92c02317a681b9bb3065a8249de124f"],\
  "id":1,"jsonrpc":"2.0"}' -H "Content-Type: application/json" -X POST \
  localhost:8545
{"jsonrpc":"2.0","result":"0x32","id":1}
```

提示	Parity 有一个用于访问 JSON PRC 的网络控制台，但是在这里我们使用命令行 HTTP 客户端来访问它。

nonce 的缺失,复制和确认

如果您以编程的方式创建交易，尤其是同时创建多个独立进程的时候，那么保持记录 nonces 非常重要。

以太坊网络基于 nonce 顺序来处理交易，这意味着当在传输 nonce 为 0 的交易的时候然后也接受到了 nonce 为 2 的交易的时候，后者将不会包含在任何区块中。它将存储在内存池中，同时以太坊网络在等待缺失的 nonce 出现，所有节点都会假定缺失的 nonce 是简单地被延迟了，并且 nonce 为 2 的那个交易只是没有按照顺序来接收而已。

如果随后您传送了一个缺失的 nonce 为 1 的交易，那么两笔交易（nonce 为 1 和 2 的交易）都会被处理和包含在链上（当然得是有效的交易）。一旦填补了缺口，整个网络就可以挖掘那些被保存在内存池中的失序的交易。

这意味着，如果您发起了一些交易并且其中之一没有正式地包含在任何区块中的话，则所有的后续交易都将会被“卡住”，等待那个缺失 nonce。有的时候因为交易无效或者 gas 不足，一个交易会不经意的在 nonce 的次序中产生“缺失”。那么为了让交易再次进行，您必须用缺失的 nonce 来完成一个有效的交易。但是同样地，您应该注意一旦具有“缺失”nonce 的交易被网络验证之后，所有具有后续 nonce 值的交易广播将逐渐变为有效；取消或者召回一个交易将不再可能！

另一方面，如果您意外的复制了一个 nonce，例如，用同样的 nonce 值发送了两笔交易，这两笔交易的接收者和交易数目都不同，那么它们其中一个将会被确认而另一个将会被拒绝。至于哪一个被确认将由它们到达的第一个接受它们的验证节点的顺序来确定，这将是相当随机的。

如您所见，记录 nonce 是十分必要的，并且如果您的应用程序没有正确的处理这个过程，您将会遇到一些麻烦。不幸的是，如果您同时尝试这样去做，事情会变得更加困难，正如我们在下一节中将会看到的。

并发性，交易源头和 nonce

并发性是计算机科学很复杂的层面，而且这个经常在意料之外的情况下突然出现，尤其是在以太坊这样的去中心化和分布式的实时系统中。

简单来说，并发性是指当您拥有通过多个独立系统的同时运算。这些运算可以在相同的程序（如：多线程）、相同的 CPU（如：多处理器）或者不同的计算机上进行（如：分布式系统）。根据定义，以太坊是一个允许通过多端运算（节点、客户端、DApp）的并发性，但是通过共识强制实现单例状态的系统。

现在，假设您拥有多个独立的钱包应用程序，它们分别从相同的地址发起交易。比如说咱们从交易所的热钱包（私钥存储在网上的钱包，对比冷钱包：私钥永不触网）中提取数字货币。在理想情况下，您希望有多个计算机处理取款事项，以便它不发生卡顿或者单点故障的问题。然而这很快就有新的问题出现了，因为多台计算机发起取款将造成一些棘手的并发性问题，尤其是 nonce 值的选择。那么多台计算机是如何从同一个热钱包生成、签名或者广播交易的呢？

您可以使用单个计算机基于先后顺序的原则来为签署交易的计算机分配 nonce。然而，假设这台计算机出现了单点故障的问题。更糟糕的是，如果有已经几个 nonce 值被分配出去了，但是其中有一个值从未被使用（因为计算机使用该 nonce 处理交易的失败），接着所有后续的交易将会被卡住而无法进行。

另一种方式是去生成交易，但是不给这些交易分配 nonce（因此它们处于未签名状态—请记住 nonce 是交易数据的一个组成部分，因此需要被包含在用来验证交易的数字签名中）。然后可以将这些交易排到单个节点上，该节点对它们进行签名并保持记录 nonce，但是这个过程可能会产生瓶颈，对 nonce 的签名和记录是您运算的一部分，这很可能在有负载的情况下变得拥挤卡顿，而未签名的交易的产生是您不需要真正并行化的部分。在这整个过程中会具备一些并发性，但是一些关键部分可以避免这个特性。

最后，除了在独立运算流程中跟踪账户余额和交易确认的难度之外，这些并发性的问题，迫使我们采用了大多数的措施来避免并发同时创造一些瓶颈，例如通过单个过程处理交易所中的所有取款事项，或者设置多个热钱包，它们能完成完全独立的取款事项而且只需要间歇性的再平衡即可。

交易 gas

我们在前面的章节中讨论了 gas，我们在[gas]中更详细的讨论了它。 然而，这里让我们介绍一些关于交易的 gasPrice 和 gasLimit 组件的作用的一些基础知识吧。

Gas 是以太坊的燃料。它不是以太币，它是一种独立的虚拟货币，有着自己对以太的汇率。以太坊使用 gas 来控制交易所使用的资源量，因为它将会在全球数千台计算机上进行处

理。开放式（Turing Complete）计算模型需要某种形式的计量来避免 Dos 攻击或者无意中进行的资源消耗交易。

Gas 和以太是区分开来的系统，为了保护其免受以太价格快速变化时可能出现的波动性影响，同时也是管理用 gas 结算的各种资源（即计算，内存和容量）的成本之间重要和敏感的比率的一种方法。

交易中的 gasPrice 字段允许交易发起人设定他们愿意支付来交换 gas 的价格。价格的计量中每单位的 gas 用 wei 来表示。 例如，在简介章节的示例交易中，您的钱包将 Gasprice 设置为 3 gwei（3Gigawei 或 30 亿 wei）。

提示	一个名为 ETH gas station 的网站提供了一些有关以太坊主网的当前 gas 价格和其他 gas 相关指标的信息。

钱包可以调整 gasprice 的值来加快交易的确认时间。 gasPrice 越高，交易被确认的速度就越快。相反， 低优先级的交易通常伴随着较低的手续费，从而导致确认速度变慢。gasPrice 最小可以被设置成 0，这意味着完成一个免费的交易。当区块空间需求比较低的时候，这样的交易很可能被产生。

注意	最小可接受 gasPrice 值为零。这意味着钱包可以生成完全免费的交易。根据容量的不同,这些交易可能永远不会被确认,但是协议中没有禁止免费交易的内容。您可以在以太坊区块链上找到一些此类交易的成功执行案例。

web3 界面通过计算几个区块的中间价格来提供一个 gasPrice 的建议设置价格。（我们可以通过使用 Truffle 控制台或任何 JaveScript web3 控制台来实现这一点）

```
> web3.eth.getGasPrice(console.log)
> null BigNumber { s: 1, e: 10, c: [ 10000000000 ] }
```

第二个与 gas 有关的字段是 gasLimit。简单来说，gasLimit 给出了交易发起人为了完成交易而愿意购买的最大 gas 单位数值。对于简单的支付，即将以太币从一个 EOA 转账到另一个 EOA 的交易，所需的 gas 值固定为 21000 个 gas 单位。为了计算这次交易应该花费多少以太，您需要将 21000 乘以您愿意支付的 gasPrice，例如：

```
> web3.eth.getGasPrice(function(err, res) {console.log(res*21000)} )
```

```
> 210000000000000
```

如果您的交易目的地是一个合约，那么所需的 gas 数量可以估计，但是不能被准确确定。这是因为合约可以估算导致不同执行路径的不同条件，以及不同的总 gas 花费。合约可能只执行一个简单的计算或一个更复杂一点的计算，这取决于您无法控制且预测的条件。为了证明这一点，让我们来看一个例子：我们可以编写一个智能合约，它在每次调用计数器时递增一个计数器，并执行一个特定的循环，循环次数等于调用的次数。也许在第 100 次调用的过程中，会有一个特殊的奖励机制，就像中彩票一样，但是需要额外的计算来算出奖励。如果您调用了 99 次合约，都是发生相同的事，但是在第 100 次调用中，一些不一样的事情发生了。在您的交易被区块验证之前，您需要支付的 gas 取决于有多少其他的交易调用了该函数。也许您的估算本是基于第 99 笔交易，但是在您的交易被确认之前，有一笔交易第 99 次调用了该合约。那么目前您的交易是第 100 个调用合约的交易，所以计算工作量（和 gas 花费）就要高得多。

借用在以太坊网络中常用的一个类比，您可以把 gasLimit 想象成您车里邮箱的容量（您的车就是交易本身）。您可以在邮箱中加满您认为在旅途中所需要的汽油（验证您的交易所需要的计算量）。您可以在某种程度上估计这个量，但是在您的旅程中可能会有意想不到的变化，例如改道（更复杂的执行路径），这会增加 gas 的消耗。

然而，这个类比可能会有点误导。实际上，它更像一个加油站公司的信用账户，根据您实际使用的汽油量，您可以在旅行结束后付款。当您发起交易时，第一个验证步骤的一部分就是检查交易发起的账户是否有足够的以太来支付 gasPrice 费用。但是在交易执行之前，该金额不会从您的账户中被扣除。您只需支付交易实际消耗的 gas 费用，但在发起交易之前，您必须有足够的余额来支付您愿意支付的最大金额。

交易接收方

交易的接收方在“to”字段中指定。它包含一个 20 字节的以太坊地址。地址可以是 EOA 或者一个合约地址。

以太坊对这个“to”字段没有进一步的验证。任何 20 字节的地址都被认为是有效的。如果 20 字节对应的地址没有对应的私钥，或者没有对应的合约，则交易依然有效。以太坊无法知道一个地址是否正确的从存在的公钥（源自于私钥）中派生。

警告	以太坊协议不验证交易中的接受人地址。您可以发送到一个没有相应私钥或合约的地址，从而“燃烧”以太，使其永远无法交易。验证这个流程应该在用户界面这一级别进行。

将交易发送到错误的地址可能会烧掉被发送的以太币，使其永久性的无法获得（无法支付），因为大多数的地址没有已知的私钥，因此无法生成签名来花费它。默认的是，地址的验证发生在用户界面级别。（见[EIP55]）事实上，销毁以太有一些合理的理由—例如，抑制在支付渠道和智能合约中的欺诈行为。由于以太的数量有限，销毁以太能够有效的将被销毁的以太的价值分配给以太的所有持有者（与他们所持有的以太数量成比例）。

交易的数值和数据

交易的“有效负载”包含在两个字段中：数值和数据。交易能同时具有数值和数据、仅数值、仅数据或者不具备数值或者数据。这四种组合都有效。

只含有数值的交易就是支付。只包含数据的交易就是调用。同时具有数值和数据的交易既是支付又是调用。一个既没有数值又不具备数据的交易，可能就只是单纯的浪费了 gas！但是这个仍然是可能发生的。

让我们来试试所有的这些组合。首先，我们将在钱包中设置源地址和目的地址，为了让这个演示看起来更明朗：

```
src = web3.eth.accounts[0];
dst = web3.eth.accounts[1];
```

我们的第一笔交易只包含数值（支付），但是没有数据的负载：

```
web3.eth.sendTransaction({from: src, to: dst, \
value: web3.toWei(0.01, "ether"), data: ""});
```

我们的钱包出现了一个确认的界面，提示需要被发送的数值，如这个 Parity 钱包所示的，显示有数值却没有数据的交易。

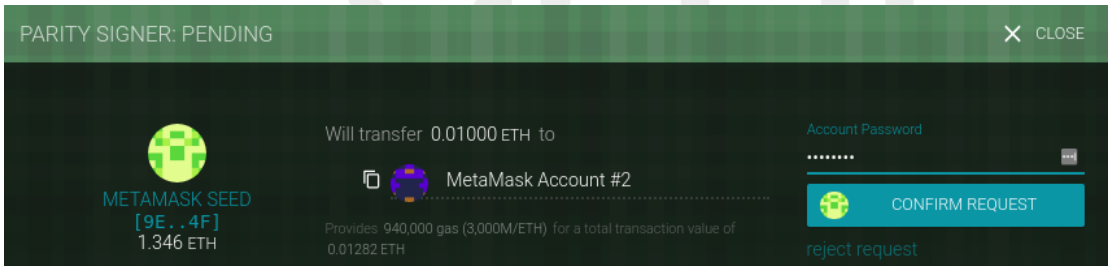


图 1：Parity 钱包显示有数值而无数据的交易

下一个例子展示了既有数值也有数据的交易：

```
web3.eth.sendTransaction({from: src, to: dst, \
```

```
    value: web3.toWei(0.01, "ether"), data: "0x1234"});
```

我们的钱包出现了一个确认的界面，提示需要被发送的数值和数据，如这个 Parity 钱包所示的，显示既有数值又有数据的交易。

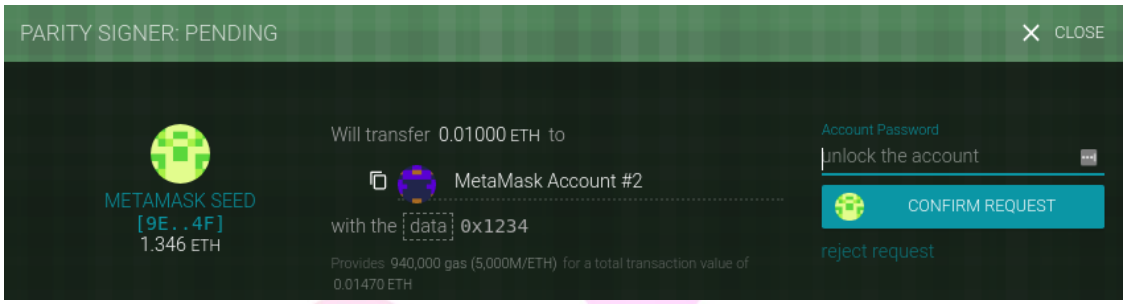


图 2：Parity 钱包显示有数值和数据的交易

下一笔交易包含数据，但是指定数值为零：

```
    web3.eth.sendTransaction({from: src, to: dst, value: 0, data: "0x1234"});
```

我们的钱包展示了一个确认的界面，显示了数值为零但是有数据负载，如这个 Parity 钱包所示的，零数值只有数据的交易。

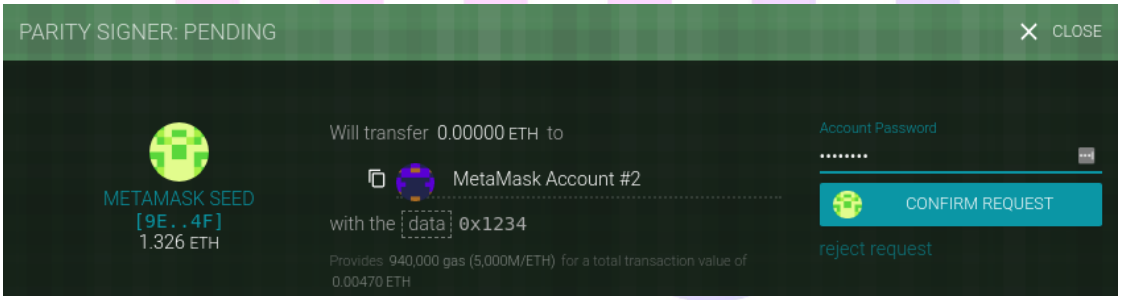


图 3：Parity 钱包显示零数值只含有数据的交易

最后的示例，最后的交易既不包含数值也不具有数据：

```
    web3.eth.sendTransaction({from: src, to: dst, value: 0, data: ""}));
```

我们的钱包展示了一个确认的界面，显示了数值为零，如这个 Parity 钱包所示的，零数值且无数据负载。

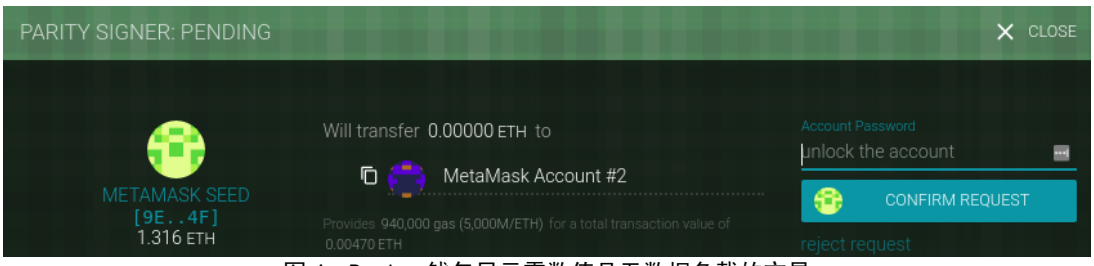


图 4：Parity 钱包显示零数值且无数据负载的交易

给 EOA 和合约地址发送数值

当您构建一个包含数值的以太坊交易时，它就相当于一个支付。根据目的地址是否为合约。此类交易的结果会有不同。

对于 EOA 地址，或者更确切的说，对于区块链上未标记为合约的任意地址，以太坊网络记录这个状态的变更，将您发送的数值添加到地址的余额中。如果之前这个地址没有被确认过，那么它将被添加到客户端的状态内部表示中，其余额被初始化为您的付款数额。

如果目标地址（去往的）是一个合约，那么 EVM 将执行这个合约，并尝试去调用交易数据负载中指定的函数。如果交易中不含有数据，那么 EVM 将会调用一个回退函数，如果这个函数是可支付的，则将会执行该函数以确定下一步要做什么。如果这儿没有回退函数的话，那么这次交易的后果将会增加合约的余额，就像向一个钱包付款一样。

当函数被调用或者其中编码的条件被确定的时候，合约可以通过立即抛出异常来拒绝转入的付款。如果函数成功地终止（无异常抛出），则合约的状态会被更新，以反映合约的以太余额增加。

给 EOA 和合约地址发送数据负载

当您的交易包含数据的时候，它很有可能被发送到一个合约地址。这不意味着您不能将数据负载发送到在以太坊协议中完全有效的 EOA 上。但是，在这种情况下，数据的解译取决于您用来访问 EOA 的钱包。它是被以太坊网络忽略的。大多数的钱包也会忽略掉它们控制的 EOA 交易中所接受到的任何数据。将来，可能会出现一些标准，允许钱包按照合约的方式解析数据，从而允许交易调用用户钱包内运行的函数。关键的区别在于，EOA 对交易的有效负载的任何解译都不受以太坊共识规则的约束，这与合约的执行不同。

现在，假设您的交易正在传送数据去合约地址。在这种情况下，EVM 将数据解译为合约调用。大多数合约将此数据更具体的用作函数调用，调用被指定的函数并将被编译的参数传递给这个函数。

发送到 ABI 兼容合约的数据负载（您可以假定所有合约都是如此）是十六进制的序列编码：

函数选择器

函数原型的 Keccak-256 哈希值的前 4 个字节。这使得合约明确的标识要调用的函数。

函数参数

函数的参数，根据 ABI 规范中定义的各种基本类型的规则进行编码。

在[solidity-faucet-example]中，我们定义了取款的函数：

```
function withdraw(uint withdraw_amount) public {
```

函数的原型被定义为包含函数名的字符串，接着是每个参数的数据类型，用括号括起来，被逗号分隔。这里的函数名称是 withdraw（取款），它采用一个 unit（unit256 的别名）参数，因此 withdraw 的原型是：

```
withdraw(uint256)
```

让我们来算算这个字符串中的 Keccak-256 哈希值：

```
> web3.sha3("withdraw(uint256)");
'0x2e1a7d4d13322e7b96f9a57413e1525c250fb7a9021cf91d1540d5b69f16a49f'
```

哈希值的前 4 个字节是 0x2e1a7d4d。这是我们的函数选择器的值，它将告诉我们合约将会调用哪个函数。

接下来，让我们计算一下要传递的提取金额参数值。我们要提取 0.01 个以太。让我们将其编码为十六进制无符号大端 256 位整数，使用 wei 来表示：

```
> withdraw_amount = web3.toWei(0.01, "ether");
'10000000000000000'
> withdraw_amount_hex = web3.toHex(withdraw_amount);
'0x2386f26fc10000'
```

现在，我们将函数选择器添加到这个数值（填充到 32 字节）：

```
2e1a7d4d0000000000000000000000000000000000000000000002386f26fc10000
```

这就是我们的交易的数据负载，调用取款的函数并且发起 0.01 个以太作为取款数量的申请。

特殊交易：合约创建

我们应该提到的一个特殊情况是，交易在区块链上创建了一个新的合约，并部署好以供将来使用。合约创建交易被发送到一个被称为零地址的特殊目标地址；在合约注册交易的“接收人”字段包含地址 0x0。这个地址既不代表 EOA（没有相应的公私钥）也不代表一个合约。它永远不能花费以太或者启动一个交易。它仅仅被用作目的地址，具有“创建这个合约”的特殊意义。

虽然零地址仅用于创建合约，但有时它会从不同的地址接受付款。对此有两种解释：一是因为意外发送导致以太损失，另一种是故意销毁以太（故意将以太发送到一个永远无法使用的地址）。但是，如果您想要有意的销毁以太，您应该向网络明确地表明您的意图，并使用专门用来销毁以太的特殊地址：

```
0x000000000000000000000000000000000000dEaD
```

警告	任何被发送到此地址的以太将会永久地不可花费和丢失。

一个合约创建的交易只需要包含将会创建合约的含有已编译代码的数据负载，此交易的唯一效果是创建合约。如果您想要使用起始余额设置新的合约，可以在“值”的字段中添加以太的额度，但是这是自主选择的。如果您将以太发送到没有数据负载的（无合约）的合约创建地址中，则后果和发送到销毁的地址是相同的，没有合约作为信用背书，所以以太将会丢失。

例如，我们可以创建在[序章]中提到的水龙头.sol 合约，通过手动创建一个发送到零地址的交易，将该合约放到交易的数据负载中。合约需要被编译成用字节码表示，这可以通过 solidity 编译器完成：

```
$ solc --bin Faucet.sol

Binary:
```

```
6060604052341561000f57600080fd5b60e58061001d6000396000f300606060405260043
61060...
```

同样的信息也可以从 Remix 在线编译器获得。

现在我们可以创建交易：

```
> src = web3.eth.accounts[0];
> faucet_code = \

"0x6060604052341561000f57600080fd5b60e58061001d6000396000f300606...f0029"
;
> web3.eth.sendTransaction({from: src, to: 0, data: faucet_code, \
  gas: 113558, gasPrice: 200000000000});
"0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b"
```

即使在创建零地址合约的情况之下，始终指定一个 to 参数也是一个很好的操作，因为意外地将以太发送到 0x0 地址而永久丢失的成本太高了。您同时应该指定 gasPrice 和 gasLimit。

一旦合约被执行，我们可以在 Etherscan 区块链浏览器上看到它，如 Etherscan 中所示，它显示了被成功挖掘的合约。

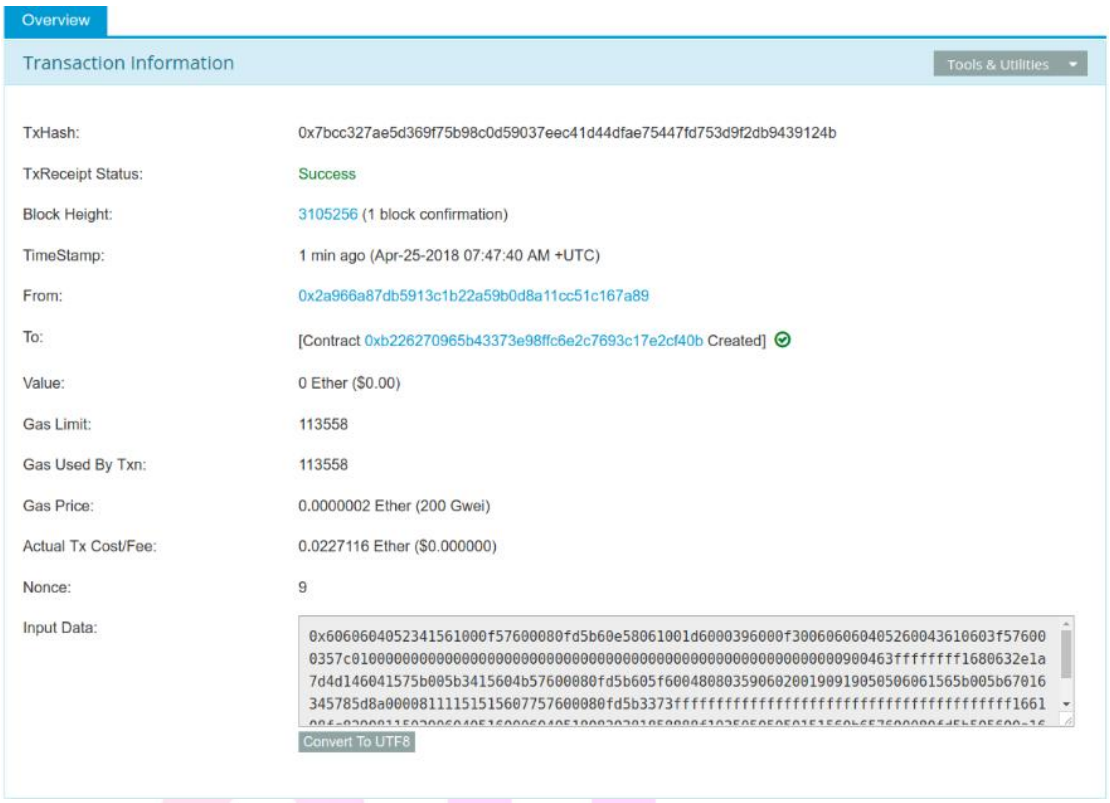


图 5：Etherscan 显示被成功挖掘的合约

我们可以看到交易的接受来得到合约的信息：

```
> eth.getTransactionReceipt( \
  "0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b");
{
  blockHash:
"0x6fa7d8bf982490de6246875deb2c21e5f3665b4422089c060138fc3907a95bb2",
  blockNumber: 3105256,
  contractAddress: "0xb226270965b43373e98ffc6e2c7693c17e2cf40b",
  cumulativeGasUsed: 113558,
  from: "0x2a966a87db5913c1b22a59b0d8a11cc51c167a89",
  gasUsed: 113558,
  logs: [],
  logsBloom: \
    "0x00000000000000000000000000000000000000000000000000...00000",
  status: "0x1",
  to: null,
```

```
  transactionHash: \
    "0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b",
  transactionIndex: 0
}
```

这包括合约地址，我们可以发送或者接受来自于合约的资金，正如前一节所示：

```
> contract_address = "0xb226270965b43373e98ffc6e2c7693c17e2cf40b"
> web3.eth.sendTransaction({from: src, to: contract_address, \
  value: web3.toWei(0.1, "ether"), data: ""});
"0x6ebf2e1fe95cc9c1fe2e1a0dc45678ccd127d374fdf145c5c8e6cd4ea2e6ca9f"
> web3.eth.sendTransaction({from: src, to: contract_address, value: 0, data:
\

"0x2e1a7d4d000000000000000000000000000000000000000000000002386f26fc10000"});
"0x59836029e7ce43e92daf84313816ca31420a76a9a571b69e31ec4bf4b37cd16e"
```

一段时间后，两个交易在以太坊区块链浏览器上都会可见，如 Etherscan 中所示，显示发送和接受资金的交易。

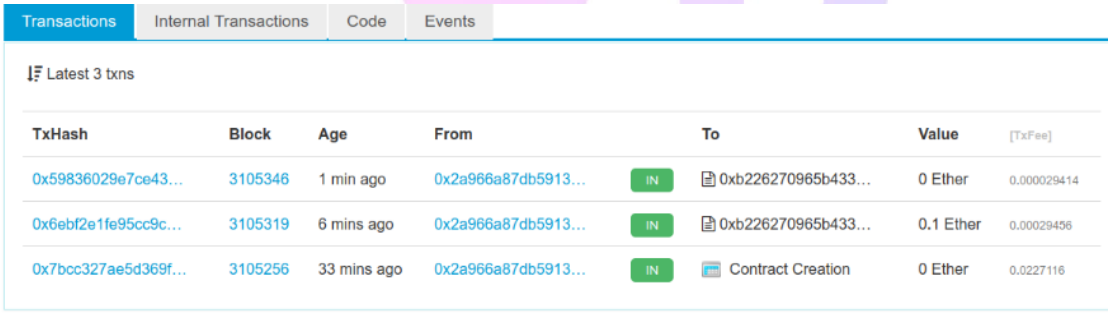


图 6：Etherscan 显示发送和接受资金的交易

数字签名

到目前为止，我们还没有深入的讨论有关数字签名的任何细节。在本节中，我们将介绍数字签名的工作原理，以及在私钥不公开的情况下，它们是如何被使用来提供私钥所有权的证明的。

椭圆曲线数字签名算法

以太坊中使用的数字签名算法是椭圆曲线数字签名算法（ECDSA）。它基于椭圆曲线公钥-私钥对，如[椭圆曲线]中所述。

数字签名在以太坊中有三个用途（参见下面的侧边栏）。首先，签名证明私钥的所有者，也就是暗示这是以太坊账户的所有者，被授权使用以太或者执行合约。第二，它保证了不可否认性：授权的证明是不可否认的。第三：签名证明了在交易签名后这次交易的数据没有也不可能被任何人修改。

维基百科对数字签名的定义

数字签名是表示数字信息或者文档真实性的数学方案。有效的数字签名使接收人有充分的理由来相信信息是已知发起人创建的（身份验证），发送人不能否认已被发出的信息（不可否认性），而且在发送过程中信息没有被更改（完整性）。

来源：https://en.wikipedia.org/wiki/Digital_signature

数字签名工作原理

数字签名是由两部分组成的数学方案。第一部分是使用私钥（签名秘钥）从信息（在我们的例子中是交易）创造签名的算法。第二部分是一个允许任何人只使用信息和公钥来验证签名的算法。

创造数字签名

在以太坊 ECDSA（椭圆曲线数字签名算法）的执行中，被“签名”的信息是交易，或者更确切的说，是从交易数据 RLP 编码的 Keccak-256 哈希值。签名的秘钥是 EOA 的私钥，这个最终结果就是数字签名：

$$Sig = F_{sig}(F_{keccak256}(m), k)$$

1. K 是签名私钥。
2. m 是 RLP 编码的交易。
3. $F_{keccak256}$ 是 Keccak-256 哈希函数。
4. F_{sig} 是签名算法。
5. Sig 是最终的数字签名。

这个函数 F_{sig} 得出的签名 Sig 由两个值组成，一般是 r 和 s：

$$Sig = (r, s)$$

验证数字签名

要验证签名，必须具有签名（r 和 s）、序列化的交易和用于创建签名的私钥对应的公钥。实质上，对签名的验证意味着来验证“只有生成此公钥的私钥所有者才能在此交易上生成此签名。”

签名验证算法获取交易信息（即我们使用的交易的哈希值）、签名者的公钥和签名（r 和 s 值），如果签名对该消息和公钥有效则传回“true”的指令。（验证完毕）

ECDSA 数学方法

如前所述，签名是由一个数学函数 F_{sig} 创建的，它生成一个由两个值 r 和 s 组成的签名。在本节中，我们将更详细的介绍函数 F_{sig} 。

签名算法首先以加密安全的方式生成一个短暂的（临时的）私钥。此临时私钥用于计算 r 和 s 值，以确保在以太坊网络上能够监测到的被签名的交易的网络攻击者无法计算发送方的真实私钥。

正如我们从[公钥]中所介绍的，被使用的临时私钥来源于相应的临时公钥，因此我们有：

1. 加密的安全随机数 q，被用来作为临时的私钥。

2. 由 q 和椭圆曲线生成点 G 生成的相应临时公钥 Q。

数字签名的 r 值是临时公钥 Q 的 x 坐标。从这里，算法计算签名的 s 值，像这样：

$$s \equiv q^{-1} (Keccak256(m) + r * k) (mod\ p)$$

在这里：
1. q 是临时的私钥。

2. r 是临时公钥的 x 轴坐标。

3. k 是签名（EOA 持有者）的私钥。

4. m 是交易数据。

5. p 是椭圆曲线的素数阶。

验证的过程和签名生成的过程恰恰相反，使用 r 和 s 值和发送者的公钥来计算 Q 的值，Q 是椭圆曲线上的一个点（即在签名创建阶段的临时公钥）。步骤如下：

1. 检测所有的输入的格式是否正确。

2. 计算 $w = s^{-1} \mod p$

3. 计算 $u_1 = Keccak256(m) * w \mod p$

4. 计算 $u_2 = r * w \mod p$

5. 最后 计算这个点位于椭圆曲线 $Q \equiv u_1 * \_G + u_2 * K \pmod p$

在这里：
1. r 和 s 是签名的值。

2. k 是签名者（EOA 账户持有者）的公钥。

3. m 是被签名的交易数据。

4. G 是椭圆曲线生成的点。

5. p 是椭圆曲线的素数阶。

如果计算出的点 Q 的 X 轴坐标等于 r，那么验证者可以得出签名是有效的。

注意，在验证签名的时候，私钥既不被知道也不被显示。

提示	提示：ECDSA 确实是一个相当复杂的数学问题；对它完整的解释超出了本书的范围。许多优秀的在线教学指南可以让您一步一步地完成这个过程：您可以搜索“ECDSA explained”或者尝试这个： http://bit.ly/2rOHhGB.

在实践中的交易签名

为了生成有效的交易，交易发起者必须使用椭圆曲线数字签名算法对消息进行数字签名。当我们说“交易签名”时，实际上是指签名 RLP 序列化交易交易数据的 Keccak-256 哈希值。签名是应用于交易数据的哈希值上而不是交易本身。

为了要在以太坊中对交易签名，交易发起者必须：

1. 创建一个交易数据结构，包括九个字段：nonce，gasPrice，gasLimit，to，value，data，chainID，0,0.

2. 生成交易数据结构的 RLP 编码序列化信息。

3. 计算这个序列化信息的 Keccak-256 哈希值。

4. 计算 ECDSA 数字签名，使用原始 EOA 的私钥对交易数据的哈希值签名。

5. 将 ECDSA 签名的值 v,r 和 s 附加上交易上。

特殊的签名变量 v 表示两件事：区块链 ID 和恢复标识符，以帮助 ECDSA 恢复函数来检查签名。它可以被计算为 27 或者 28，或者链 ID 的两倍加上 35 或者 36。有关链 ID 的更多信息，请参阅使用 EIP-155 创建原始交易。恢复标识符（在“旧样式”签名中为 27 或者 28，在完全伪造的“龙式”交易中为 35 或 36）用于表示公钥中 y 组件的奇偶性（请参见签名前缀值 v 和公钥恢复查看更多细节）。

注意	在区块链高度为#2675000 处，以太坊执行了“伪龙”硬分叉，除了其他的变化之外，它引入了一个新的签名方案，其中包括交易重放保护（防止一个网络的交易重放在其他网络上）。EIP-155 中规定了这一新的签名方案。此更改会影响交易的形式和签名，因此必须注意到三个签名变量中的第一个（即 v），它使用了两种形式之一并表示了散列的交易信息中包含的数据字段。

原始交易的创建和签名

在本节中，我们将创建一个原始交易并使用 ethereumjs-tx 库对其进行签名。这将会展示通常在钱包中或者代表用户签署交易的应用程序中被使用到的功能。本例的源代码位于本书 GitHub 存储库中的 raw_tx_demo.js 文件中：

```
link:code/web3js/raw_tx/raw_tx_demo.js[]
```

运行示例代码会产生以下结果：

```
$ node raw_tx_demo.js

RLP-Encoded Tx:
0xe6808609184e72a0008303000094b0920c523d582040f2bcb1bd7fb1c7c1...

Tx Hash:
0xaa7f03f9f4e52fcf69f836a6d2bbc7706580adce0a068ff6525ba337218e6992

Signed Raw Transaction:
0xf866808609184e72a0008303000094b0920c523d582040f2bcb1...
```

使用 EIP-155 构建原始交易

EIP-155“简单重放攻击保护”标准规定了重放攻击保护的交易编码，该编码于签名之前在交易数据中包含了一个链标识符。这样可以确保一个区块链（如以太坊主网）创建的交易在另一个区块链（如以太坊经典或者 Ropsten 测试网）上无效。因此，在一个网络上广播的交易不能在另一个网络上重放，因此这就是标准。

EIP-155 将三个字段添加到交易数据结构的六个主要字段中，即链标识符、0 和 0。这三个字段在被编码或者散列之前被添加到交易数据中。因此，它们会改变交易的哈希值，稍后将对其加上数字签名。通过向正在签名的数据加入链标识符，可以防止交易签名被更改， 因为如果修改了链标识符，签名将会失效。因此，EIP-155 使交易不可能在另一条链上重放，因为签名的有效性取决于链标识符。

链标识符的字段的值和交易的目的网络有关，下面是链标识符的对应表格：

链	链 ID
Ethereum mainnet	1
Morden (obsolete),Expanse	2
Ropsten	3
Rinkeby	4
Rootstock mainnet	30
Rootstock testnet	31
Kovan	42
Ethereum Classic mainnet	61
Ethereum Classic testnet	62
Geth private testnets	1337

图表 1：链标识符

生成的交易结构是 RLP 编码的、散列的和已签名的。签名的算法被轻微的修改，以将链标识符也编码到 v 的前缀中。

获得更多的细节信息，请参考 the EIP-155 specification（EIP-155 规范）.

签名的前缀值 v 和公钥恢复

如交易结构中所述，交易信息并不包含“发件人”字段。这是因为交易发起者的公钥可以直接从 ECDSA 签名中计算出来。一旦拥有了公钥，您便可以轻松的计算出地址。这个恢复签名者的公钥的过程被称为公钥恢复。

给定在 ECDSA 函数中计算出的值 r 和 s，我们可以计算出两个可能的公钥。

首先，我们从签名中的 x 坐标 r 值计算两个椭圆曲线点 R 和 R'。存在两个点，因为椭圆曲线在 x 轴上是对称的，所以对于任何值 x，都有两个符合曲线的可能值，分别位于 x 轴的两侧。

根据 r 我们也能计算出 r^{-1}，它是 r 的乘法逆（互为倒数）。

最后，我们计算 z,它是信息哈希值的 n 个最低位，其中 n 是椭圆曲线的阶。

接着两个可能的公钥就是：

- $K_1 = r^{-1}(sR - zG)$
- $K_2 = r^{-1}(sR' - zG)$

其中：

1. K_1 和 K_2 是签名者的公钥的可能值。

2. r^{-1} 是签名的 r 值的乘法逆（互为倒数）。

3. s 是签名的 s 值。

4. R 和 R'是临时公钥 Q 的两个可能值。

5. z 是交易信息哈希值的 n 个最低位。

6. G 是椭圆曲线生成点。

为了提高效率，交易签名包含一个前缀值 v， 它告诉我们两个可能的 r 值中哪个是临时公钥。如果 v 是偶数，那么 R 是正确的值。如果 v 是奇数，那么 *R'* 就是正确的公钥值。这样的话，我们只需要计算 R 的一个值和 K 的一个值。

隔离签名和传输

一旦一个交易被签名，它就可以被传输到以太坊网络。创建、签名和广播交易的三个步骤通常作为单个操作发生。例如使用 web3.eth.sendTransaction。但是，正如您在原始交易创建和签名中看到的，您可以通过两个单独的步骤分别创建和签名交易。一旦您有了一个已签名的交易，您就可以使用 web3.eth.sendsignedTransaction 传输它，它接受十六进制编码和已签名的交易，并在以太坊网络上传输它。

为什么要将交易的签名和传输隔离开呢？最显而易见的原因是为了更加安全。对交易签名的计算机必须具有加载于内存的未锁定的私钥。进行传输的计算机必须连接到互联网（并运行以太坊客户端）。如果这两个功能在一台计算机上，那么您就有了联网的私钥，这是非常危险的。在不同的设备上（分别在离线设备或者在线设备上）分别签名和传输以及执行这些功能被成为离线签名，这是一种常见的安全措施。

以太坊交易的离线签名过程如下：

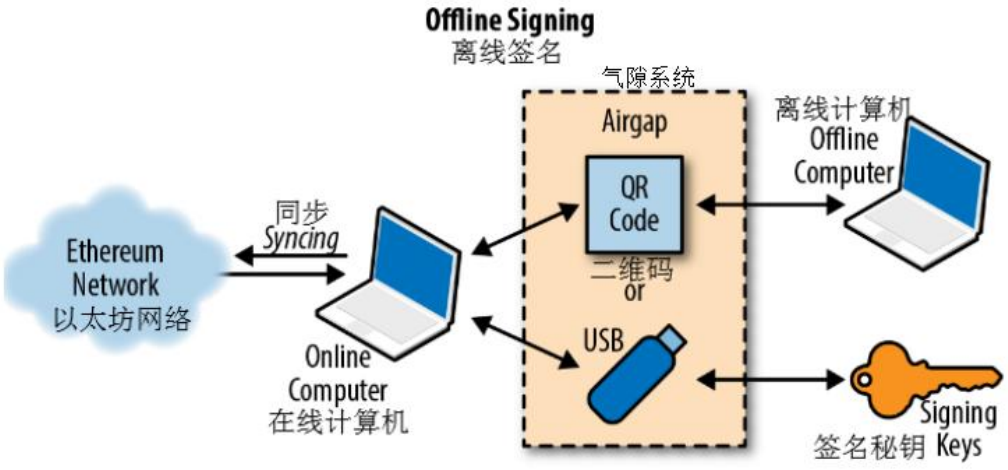


图 7：以太坊交易的离线签名

根据您所需要的安全级别，您的“离线签名”计算机可以与联机计算机有不同程度的分离，从隔离和防火墙子网（联机但隔离）到完全离线的系统（称为气隙系统）。在一个气隙系统中，没有任何网络连接，所有的计算机与在线环境之间都被一个“空隙”分隔开。在气隙系统的计算机之间签署交易的话，您可以使用数据存储介质或者（更好的）网络摄像头和

二维码等工具。当然，这意味着您必须手动传输您想要签名的每个交易，这些过程不能规模化。

虽然没有多少运行环境可以使用完全隔离的系统，但是即使是很小程度的隔离也具有显著的安全效益。例如，带有防火墙的隔离子网只允许信息队列协议通过，与在联机系统上签名相比，它可以提供一个大大减少的攻击面和同时具有更高的安全性。许多公司为了达到这个目的使用了 ZeroMQ (0MQ) 等协议。通过这样的设置，交易被序列化并排队等待签名。队列协议以类似于 TCP socket 的形式将序列化信息传输到签名计算机。签名计算机从队列中读取序列化交易（仔细地），使用适当的秘钥进行签名。并将它们放在传出队列上。传出队列将已签名的交易传输到一台带有以太坊客户端的计算机上，并将它们出列并传输。

交易广播

以太坊网络使用“洪水演算”协议。每个以太坊客户端在 P2P 网络中充当一个节点来（理想情况下）形成一个网状网络。没有一个网络节点是特殊的：他们都是对等的。我们将使用术语“节点”来指连接并参与 P2P 网络的以太坊客户端。

交易广播从发起已签名交易的（或从接受到离线签名的）以太坊节点开始。该交易被验证，然后传输到与发起节点直接连接的所有其他以太坊节点。平均来说，每个以太坊节点都保持与至少 13 个其他节点（被称为邻居）直接连接。每个邻居已收到交易信息便开始验证。如果它们同意这笔交易是有效的，这些节点会存储一个副本并将其传播给它们的所有邻居（信息的来源节点除外）。因此，交易从起始节点向外波动，传遍全网，直到网络中的所有节点都有交易的副本。节点可以过滤它们传播的信息，但在默认情况下是传播它们接收到的所有有效交易信息。

几秒钟之内，以太坊的交易会传播到全球所有的以太坊节点。从每个节点的角度上来说，不能识别交易的起源。发送信息到节点的邻居可能是交易的发起者，也可能是转发从它的邻居那里收到的交易信息。为了能够跟踪交易的起源或者介入干扰这个传播，攻击者必须控制节点的很大一部分。这是 P2P 网络的安全性和隐私性设计的一部分，尤其是当应用到区块链网络的时候。

数据上链

虽然以太坊中的所有节点都是对等的，但其中一些节点是由矿工操作的，来向矿池（即具有高性能图形处理单元 GPUs 的计算机）提供交易和区块。挖矿的计算机将交易添加到候选块，并尝试查找使候选块有效的工作量证明。我们将在[共识]中更详细的讨论这一点。

简单来说，就是有效的交易最终将被包含在一个交易块中，因此被记录在以太坊区块链上。一旦被挖掘到区块中，交易也会通过修改账户余额（在简单的支付中）或调用更改其内部状态的合约来修改以太坊单例机的状态。这些变更以交易凭证的形式与交易一同被记录，同时也可能包含一些事件。我们将在[evm_章节]中更详细地探讨。

总而言之，一个交易要从创建到完整的完成，需要经过 EOA 的数字签名、全网广播和最后通过挖矿改变单例的状态，并在区块链上留下了不可磨灭的标记。

多重签名交易

如果您熟悉比特币的脚本功能，您应该知道可以创建一个比特币多重签名账户，该账户只能在多方签署交易时使用资金（例如，2/2 或 3/4 签名）。以太坊的基本 EOA 值交易没有对多重签名的规定；但是，通过智能合约可以强制执行任意的签名限制，您可以考虑任意条件，来处理以太坊或代币 token 的转账。

为了利用这个功能，以太必须被转移到一个“钱包合约”，该合约是按照所需的支出规则来编译的，如多签名需求或支出限额（或两者的组合）。一旦当支出条件得到满足，钱包合约就会在授权的 EOA 的提示下发送资金。例如，为了在多重签名条件下保护您的以太，请将以太转移到多重签名合约中。无论何时，只要您想将资金发送到另一个账户，所有符合要求的用户都需要使用常规钱包软件发送交易到这个合约，从而有效的授权这个合约来执行最终的交易。

这些合约还可以被设计为在执行本地代码或触发其他合约之前需要多个签名。该方案的安全性最终由多重签名合约代码来决定。

使用智能合约执行多重签名交易的能力体现了以太坊的灵活性。然而，这个能力是一把双刃剑，因为额外的灵活性会导致一些可能破坏多重签名方案的漏洞出现，事实上，在 EVM 中会有很多创建多重签名指令的建议，这会消除对智能合约的需求，至少对于简单的 M-of-N 多重签名方案是如此。这相当于比特币的多重签名系统，该系统是核心共识规则的一部分，并且已经被证明是稳健和安全的。

结语

交易是以太坊系统中每个活动的起点。交易是致使以太坊虚拟机评估合约、更新余额以及更普遍的修改以太坊区块链状态的“输入”。接下来，我们将了解关于智能合约的更多细节，并学习如何使用 Solidity 合约导向的语言来编程。

第七章 智能合约与 Solidity

智能合约与 solidity

正如我们在前言章节所讨论的那样，以太坊有两种不同类型的账户：外部账户（EOA）和合约账户。外部账户由用户控制，通常通过软件，例如以太坊平台外部的钱包应用程序。相反，合约账户由以太坊虚拟机执行的程序代码（通常也称为“智能合约”）控制。简而言之，外部账户是简单的帐户，没有任何相关的代码或数据存储，而合约帐户同时具有相关的代码和数据存储。外部账户由创建 的交易控制，并在协议外部和独立于“真实世界”中使用私钥加密签名，而合约帐户没有私钥，因此通过他们的智能合约代码以既定的方式“控制自己”(control themselves) 。两种类型的帐户都由以太坊地址标识。在本章中，我们将讨论合约账户和控制它们的程序代码。

什么是智能合约？

智能合约一词多年来一直用它来描述各种不同的东西。在 20 世纪 90 年代，密码学家 Nick Szabo 创造了这个术语，并将其定义为“一系列承诺，以数字形式指定，包括各方在其他承

诺上执行的协议。”从那时起，智能合约的概念已经发生变化，尤其是在 2009 年发布比特币的分布式区块链平台之后。在以太坊的背景下，这个词实际上有点用词不当，因为以太坊智能合约既不聪明也不合法，但这个术语已经停滞不前。在本书中，我们使用术语“智能合约”来指代在以太坊虚拟机的上下文中确定性地运行的不可变计算机程序，作为以太坊网络协议的一部分 - 即，在分布式以太坊世界的计算机

让我们解开这个定义：

电脑程序

智能合约只是计算机程序。在这方面，“合约”一词没有法律意义。

不可变

部署后，智能合约的代码无法更改。与传统软件不同，修改智能合约的唯一方法是部署新实例。

确定性

考虑到启动执行的交易的上下文以及执行时的以太坊区块链的状态，执行智能合约的结果对于运行它的每个人都是相同的。

EVM 背景

智能合约的执行环境非常有限。他们可以访问自己的状态，调用它们的交易的上下文，以及有关最新块的一些信息。

分布式世界计算机

EVM 作为每个以太坊节点上的本地实例运行，但由于 EVM 的所有实例都在相同的初始状态下运行并产生相同的最终状态，因此整个系统作为单个“世界计算机”运行。

智能合约的生命周期

智能合约通常用高级语言编写，例如 Solidity。但是为了运行，必须将它们编译为在 EVM 中运行的低级字节码。编译完成后，它们将使用特殊的合约创建交易部署在以太坊平台上，该交易通过发送到特殊合约创建地址（即 0x0）来识别（见[contract_reg]））。每个合约都由以太坊地址标识，该地址是作为始发帐户和现时的函数从合约创建交易中获得的。合约的以太坊地址可以作为接收者在交易中使用，将资金发送给合约或调用合约的一个功能。请注意，与外部账户不同，没有为新智能合约创建的帐户关联的密钥。作为合约创建者，您在协议级别没有获得任何特殊权限（尽管您可以将它们明确地编码到智能合约中）。您当然不会收到合约帐户的私钥，这实际上并不存在 - 我们可以说智能合约帐户拥有自己。

重要的是，合约只有在交易调用时才会运行。因为有外部账户发起的交易，以太坊中的所有智能合约最终都会被执行。合约可以调用另一个合约，该合约又调用另一个合约的合约，依此类推，但是这样一个执行链中的第一个合约将始终由外部账户的交易调用。合约从不“独立运行”或“在后台运行”。合约实际上处于休眠状态，直到交易触发执行，直接或间接作为合约调用链的一部分。同样值得注意的是，智能合约在任何意义上都不是“并行”执行的 - 以太坊世界计算机可以被认为是单线程机器。

交易是原子的，无论他们调用多少合约或这些合约在调用时做什么。交易完整执行，只有在所有最终执行成功时，才会记录全局状态（合约，帐户等）中的任何更改。成功终止意味着程序在没有错误的情况下执行并且执行结束。如果由于错误导致执行失败，则其所有效果（状态更改）都“回滚”，就像交易从未运行一样。失败的交易仍被记录为已经尝试过，并且用于执行的 gas 用于从原始账户中扣除，但除此之外对合约或账户状态没有其他影响。

如前所述，重要的是要记住合约的代码不能改变。但是，可以“删除”合约，从其地址中删除代码及其内部状态（存储），留下空白帐户。在删除合约之后发送到该帐户地址的任何交易都不会导致任何代码执行，因为不再需要执行任何代码。要删除合约，请执行名为自毁（以前称为“自杀”）的 EVM 操作码。该操作花费“负 gas”，即 gas 退款，从而激励网络客户端资源从删除存储状态中释放。以这种方式删除合约不会删除合约的交易历史（过去的），因为区块链本身是不可变的。同样重要的是要注意，只有合约作者对智能合约进行编程才能具有该功能。如果合约的代码没有自毁操作码，或者无法访问，则无法删除智能合约。

以太坊高级语言简介

EVM是一个虚拟机，它运行一种称为EVM字节码的特殊形式的代码，类似于计算机的CPU，它运行机器代码，如 x86_64。我们将在[evm_chapter]中更详细地研究 EVM 的操作和语言。在本节中，我们将了解如何编写智能合约以在 EVM 上运行。

虽然可以直接在字节码中编写智能合约，但 EVM 字节码相当笨拙并且程序员很难阅读和理解。相反，大多数以太坊开发人员使用高级语言编写程序，并使用编译器将它们转换为字节码。

虽然任何高级语言都可以适用于编写智能合约，但是将任意语言调整为可编译的 EVM 字节码是一项非常繁琐的工作，并且通常会导致一些混淆。智能合约在高度约束和简约的执行环境（EVM）中运行。此外，还需要一组特殊的 EVM 特定系统变量和功能。因此，从头开始构建智能合约语言比制作适合编写智能合约的通用语言更容易。结果，出现了许多用于编写智能合约的特殊用途语言。以太坊有几种这样的语言，以及生成 EVM 可执行字节码所需的编译器。

通常，编程语言可以分为两种广泛的编程范例：声明和命令，也分别称为功能和程序。在声明性编程中，我们编写表达程序逻辑的函数，而不是它的流程。声明性编程用于创建没有副作用的程序，意味着函数外部的状态没有变化。声明性编程语言包括 Haskell 和 SQL。相反,命令式编程是程序员编写一组程序逻辑和流程的程序。命令式编程语言包括 C ++和 Java。有些语言是“混合的”,这意味着它们鼓励声明性编程,但也可以用来表达命令式编程范例。这种混合包括 Lisp，JavaScript 和 Python。通常，任何命令式语言都可以用于在声明性范例中编写，但它通常会导致代码不雅。相比之下，纯粹的声明性语言不能用于编写命令式范例。在纯粹的声明性语言中，没有“变量”。

虽然程序员更常使用命令式编程，但编写完全按预期执行的程序可能非常困难。程序的任何部分改变任何其他程序状态的能力使得很难推断程序的执行并引入许多错误机会。相比之下，声明性编程可以更容易理解程序的行为：由于它没有副作用，程序的任何部分都可以被孤立地理解。

在智能合约中，错误确实需要花钱。因此，编写智能合约而没有意外影响至关重要。为此，您必须能够清楚地了解程序的预期行为。因此，声明性语言在智能合约中的作用要大于通用软件中的作用。然而，正如您将看到的，最常用的智能合约语言（Solidity）势在必行。像大多数人一样，程序员抵拒绝改变！

目前支持的智能合约高级编程语言包括（按大致年龄排序）：

- LLL

一种功能性（声明性）编程语言，具有类似 Lisp 的语法。这是以太坊智能合约的第一个高级语言，但今天很少使用。

- Serpent

一种程序性（命令性）编程语言，语法类似于 Python。也可用于编写功能（声明性）代码，尽管它并非完全没有副作用。

- Solidity

一种程序性（命令性）编程语言，其语法类似于 JavaScript，C ++或 Java。以太坊智能合约中最受欢迎和最常用的语言。

- Vyper

一种最近开发的语言，类似于 Serpent，再次使用类似 Python 的语法。旨在接近纯粹的类似 Python 的语言而不是 Serpent，但不能取代 Serpent。

- Bamboo

一种新开发的语言，受 Erlang 的影响，具有显式的状态转换，没有迭代流（循环）。旨在减少副作用并提高可审计性。非常新，但尚未被广泛采用。

如您所见，有许多语言可供选择。然而，在所有这些中，Solidity 是迄今为止最受欢迎的，成为以太坊甚至是其他类似 EVM 的区块链的事实上的高级语言。我们将花费大部分时间使用 Solidity，但也将探索其他高级语言中的一些示例，以了解其不同的哲学。

建立一个 Solidity 智能合约

Solidity 由 Gavin Wood 博士（本书的合着者）创建，它是一种明确用于编写智能合约的语言，其功能是直接支持在分布式的以太坊世界计算机环境中执行。由此产生的属性非常普遍，因此它最终用于在其他几个区块链平台上编写智能合约。它由 Christian Reitiwessner 开发，然后由 Alex Beregszaszi，Liana Husikyan，Yoichi Hirai 和几位前以太坊核心贡献者开发。现在，Solidity 作为 GitHub 上的一个独立项目得以开发和维护。

Solidity 项目的主要“产品”是 Solidity 编译器 solc，它把用 Solidity 语言编写的程序转换为 EVM 字节码。该项目还管理以太坊智能合约的重要应用程序二进制接口（ABI）标准，我们将在本章详细探讨。每个版本的 Solidity 编译器都对应并编译 Solidity 语言的特定版本。

首先，我们将下载 Solidity 编译器的二进制可执行文件。然后我们将开发并编译一个简单的合约，继续我们在[intro_chapter]中开始的例子。

选择一个版本的 Solidity

Solidity 遵循称为语义版本控制的版本控制模型，该模型指定版本号结构为由点分隔的三个数字：MAJOR.MINOR.PATCH。对于主要和向后不兼容的更改，“major”数字会递增，“minor”数字会在主要版本之间添加向后兼容的功能时递增，并且“pathc”编号会递增以用于向后兼容的错误修复。

在撰写本文时，Solidity 的版本为 0.4.24。主要版本 0 的规则（用于项目的初始开发）是不同的：任何时候都可能发生变化。在实践中，Solidity 将“minor”编号视为主要版本，将“patch”编号视为次要版本。因此，在 0.4.24 中，4 被认为是主要版本，24 被认为是次要版本。

Solidity 的 0.5 主要版本即将发布。

正如您在[intro_chapter]中看到的，您的 Solidity 程序可以包含一个 pragma 指令，该指令指定与其兼容的 Solidity 的最小和最大版本，并可用于编译您的合约。

由于 Solidity 正在快速发展，因此安装最新版本通常会更好。

下载并安装

您可以使用许多方法来下载和安装 Solidity，无论是作为二进制版本还是通过源代码编译。您可以在 Solidity 文档中找到详细说明。

以下是使用 apt 包管理器在 Ubuntu / Debian 操作系统上安装 Solidity 的最新二进制版本的方法：

```
$ sudo add-apt-repository ppa: ethereum / ethereum
$ sudo apt update
$ sudo apt install solc
```

安装 solc 后，运行以下命令检查版本：

```
$ solc --version
solc, the solidity compiler commandline interface
Version: 0.4.24+commit.e67f0147.Linux.g++
```

根据您的操作系统和要求，还有许多其他方法可以安装 Solidity，包括直接从源代码编译。有关更多信息，请参阅 https://github.com/ethereum/solidity。

开发环境

要在 Solidity 中进行开发，可以在命令行上使用任何文本编辑器和 solc。但是，您可能会发现一些专为开发而设计的文本编辑器（如 Emacs，Vim 和 Atom）提供了其他功能，如语法突出显示和宏，使 Solidity 开发更容易。

还有基于 Web 的开发环境，例如 Remix IDE 和 EthFiddle。

使用可提高工作效率的工具。最后，Solidity 程序只是纯文本文件。虽然花哨的编辑器和开发环境可以使事情变得更容易，但您不需要简单的文本编辑器，例如 nano（Linux / Unix），TextEdit（macOS）甚至 NotePad（Windows）。只需使用.sol 扩展名保存程序源代码，它就会被 Solidity 编译器识别为 Solidity 程序。

写一个简单的 Solidity 程序

在[intro_chapter]中，我们编写了第一个 Solidity 程序。当我们第一次构建 Faucet 合约时，我们使用 Remix IDE 来编译和部署合约。在本节中，我们将重新审视，改进和修饰龙头。

我们的第一次尝试看起来像 Faucet.sol：实施水龙头的 Solidity 合约。

示例 1. Faucet.sol：部署 faucet 的 Solidity 合约

link:code/Solidity/Faucet.sol[]

使用 Solidity Compiler 进行编译（solc）

现在，我们将在命令行上使用 Solidity 编译器直接编译我们的合约。Solidity 编译器 solc 提供了各种选项，您可以通过传递--help 参数来查看。

我们使用 solc 的--bin 和--optimize 参数来生成示例合约的优化二进制：

```
$ solc --optimize --bin Faucet.sol
======= Faucet.sol:Faucet =======
Binary:
6060604052341561000f57600080fd5b60cf8061001d6000396000f300606060405260043
610603e5
763ffffffff7c0100000000000000000000000000000000000000000000000000000060
00350416
632e1a7d4d81146040575b005b3415604a57600080fd5b603e60043567016345785d8a000
08111156
06357600080fd5b73ffffffffffffffffffffffffffffffffffffffff331681156108fc02
82604051
60006040518083038185888f19350505050151560a057600080fd5b505600a165627a7a7
23058203
556d79355f2da19e773a9551e95f1ca7457f2b5fbbf4eacf7748ab59d2532130029
```

solc 产生的结果是一个十六进制序列化的二进制文件，可以提交给以太坊区块链。

以太坊合约 ABI

在计算机软件中，应用程序二进制接口是两个程序模块之间的接口； 通常，在操作系统和用户程序之间。ABI 定义了如何在机器代码中访问数据结构和函数； 这不应与 API 混淆，后者以高级（通常是人类可读的格式）将此访问定义为源代码。因此，ABI 是将数据编码进出机器代码的主要方式。

在以太坊中，ABI 用于编码 EVM 的合约调用以及从交易中读取数据。ABI 的目的是定义可以调用的合约中的函数，并描述每个函数如何接受参数并返回其结果。

合约的 ABI 被指定为函数描述（参见函数）和事件（参见事件）的 JSON 数组。函数描述是字段的 JSON 对象 type，name，inputs，outputs，constant，和 payable。一个事件描述对象具有字段 type，name，inputs，和 anonymous。

我们使用 solc command-line Solidity 编译器为我们的 Faucet.sol 示例合约生成 ABI ：

```
$ solc --abi Faucet.sol
======= Faucet.sol:Faucet =======
Contract JSON ABI
[{"constant":false,"inputs":[{"name":"withdraw_amount","type":"uint256"}],
\
"name":"withdraw","outputs":[],"payable":false,"stateMutability":"nonpaya
ble", \
"type":"function"},{"payable":true,"stateMutability":"payable", \
"type":"fallback"}]
```

如您所见，编译器生成一个 JSON 数组，用来描述由 Faucet.sol 定义的两个函数。任何想要在部署后访问 Faucet 合约的应用程序都可以使用此 JSON。使用 ABI，诸如钱包或 DApp 浏览器之类的应用程序可以构造使用正确的参数和参数类型调用 Faucet 中的函数的交易。例如，钱包会知道要调用函数 withdraw，它必须提供名为 withdraw_amount 的 uint256 参数。钱包可以提示用户提供该值，然后创建对其进行编码的交易并执行撤销功能。

应用程序与合约交互所需的只是 ABI 和部署合约的地址。

选择 Solidity Compiler 和语言版本

正如我们在前面的代码中看到的那样，我们的 Faucet 合约与 Solidity 版本 0.4.21 成功编译。但是，如果我们使用了不同版本的 Solidity 编译器呢？语言仍在不断变化，事情可能会以意想不到的方式发生变化。我们的合约相当简单，但如果我们的程序使用仅在 Solidity 版本 0.4.19 中添加的功能并且我们尝试使用 0.4.18 编译它会怎样？

要解决这些问题，Solidity 提供了一个编译器指令被称为一个版本编译指示该程序需要一个特定的编译器（和语言）版本的编译器。我们来看一个例子：

```
pragma solidity ^ 0.4.19;
```

Solidity 编译器读取版本编译指示，如果编译器版本与版本编译指示不兼容，则会产生错误。在这种情况下，我们的版本 pragma 表示该程序可以由 Solidity 编译器编译，最小版本为 0.4.19。但是，符号^表示我们允许使用高于 0.4.19 的任何次要修订进行编译； 例如，0.4.20，但不是 0.5.0（这是一个重大修订，而不是一个小修订）。Pragma 指令不会编译为 EVM 字节码。它们仅供编译器用于检查兼容性。

让我们为我们的龙头合约添加一个 pragma 指令。我们将命名新文件 Faucet2.sol，以便在我们从 Faucet2.sol 开始执行这些示例时跟踪我们的更改:将版本编译指示添加到 Faucet。

示例 2. Faucet2.sol：将版本 pragma 添加到 Faucet
link:code/Solidity/Faucet2.sol[]

添加版本编译指示是最佳实践，因为它避免了编译器和语言版本不匹配的问题。我们将探讨其他最佳实践，并在本章中继续改进 faucet 合约。

用 Solidity 编程

在本节中，我们将介绍 Solidity 语言的一些功能。正如我们在[intro_chapter]中提到的，我们的第一个合约示例非常简单，并且存在各种各样的缺陷。我们将逐步改进它，同时探索如何使用 Solidity。然而，这不是一个全面的 Solidity 教程，因为 Solidity 非常复杂且快速发展。我们将介绍基础知识，并为您提供足够的基础，以便能够自己探索其余部分。Solidity 的文档可以在项目网站上找到。

数据类型

首先，让我们看一下 Solidity 中提供的一些基本数据类型：

- 布尔值（布尔值）

布尔值，true 或 false，带有逻辑运算符！（非），&&（与），|| （或），==（等于），和！=（不等于）。

- 整数（int，uint）

有符号（int）和无符号（uint）整数，以从 int8 到 uint256 的 8 位增量声明。如果没有大小后缀，则使用 256 位数量，以匹配 EVM 的字大小。

- Fixed point (fixed, ufixed)

定点数，用（u）声明，其中 M 是以位为单位的大小（增量为 8 到 256），N 是该点之后的小数位数（最多 18 位）； 例如，ufixed32x2。fixedMxN

- 地址

  一个 20 字节的以太坊地址。地址对象有许多有用的成员函数，主要是余额（返回帐户余额）和　转账（将以太币转移到帐户）。

- 字节数组（固定）

  固定大小的字节数组，用 bytes1 到 bytes32 声明。

- 字节数组（动态）

  可变大小的字节数组，用字节或字符串声明。

- 枚举

  用于枚举离散值的用户定义类型：枚举 NAME {LABEL1，LABEL 2，...}。

- 数组

  任何类型的数组，无论是固定的还是动态的：uint32 [] [5]是由五个无符号整数动态数组组成的固定大小的数组。

- 结构

  用于对变量进行分组的用户定义数据容器：。struct NAME {TYPE1　VARIABLE1; TYPE2 VARIABLE2; ...}

- 字典

  key　=>　value 对的哈希查找表：mapping（KEY_TYPE => VALUE_TYPE）NAME。

  除了这些数据类型，Solidity 还提供了各种可用于计算不同单位的值：

- 时间单位

  单位秒，分，小时和天可以用作后缀，转换为基本单位秒的倍数。

- 以太单位

  wei，finney，szabo 和 ether 单位可以用作后缀，转换为基本单位 wei 的倍数。

在我们的 Faucet 合约示例中，我们对 withdraw_amount 变量使用了 uint（uint256 的别名）。我们还间接使用了一个地址变量，我们用 msg.sender 设置了它。在本章的其余部分中，我们将在示例中使用更多这些数据类型。

让我们使用其中一个单位乘数来提高示例合约的可读性。在提取函数中，我们限制最大提取，表示在以太的基本单位 wei 的限制：

要求（`withdraw_amount <= 100000000000000000`）；

这不是很容易阅读。我们可以通过使用单位乘数 ether 来改进我们的代码，用以太而不是 wei 来表示值：

要求（`withdraw_amount <= 0.1` 以太）；

预定义的全局变量和函数

在 EVM 中执行合约时，它可以访问一小组全局对象。这些包括块，msg 和 tx 对象。此外，Solidity 将许多 EVM 操作码公开为预定义功能。在本节中，我们将检查您可以在 Solidity 中的智能合约中访问的变量和函数。

交易/消息调用上下文

msg 对象是启动此合约执行的交易调用（EOA 发起）或消息调用（合约发起）。它包含许多有用的属性：

- `msg.sender`
  我们已经使用过这个。它表示发起此合约调用的地址，不一定是发送交易的原始 EOA。如果我们的合约是由 EOA 交易直接调用的，那么这是签署交易的地址，否则它将是合约地址。

- `msg.value`
  通过此调用发送的 ether 的值（在 wei 中）。

- `msg.gas`
  此执行环境的 gas 供应中剩余的 gas 量。这在 Solidity v0.4.21 中已弃用，并由 gasleft 功能取代。

- `msg.data`
  此调用的数据有效载到我们的合约中。

- `msg.sig`
  数据有效负载的前四个字节，即函数选择器。

注意	每当合约调用另一个合约时，msg 的所有属性的值都会更改以反映新调用方的信息。唯一的例外是 delegatecall 函数，它在原始 msg 上下文中运行另一个合约/库的代码。

交易背景

tx 对象提供了一种访问与交易相关的信息的方法：

- tx.gasprice
  调用交易中的 gas 价格。

- tx.origin
  此交易的原始 EOA 的地址。警告：不安全！

- block 上下文
  块对象包含有关当前块的信息：

- block.blockhash（__ blockNumber__）
  指定块编号的块哈希，过去最多 256 个块。在 Solidity v0.4.22 中弃用并替换为 blockhash 函数。

- block.coinbase
  当前区块的接收者地址的费用和区块奖励。

- block.difficulty
  当前块的难度（工作证明）。

- block.gaslimit
  在当前块中包含的所有交易中可以使用的最大 gas 量。

- block.number
  当前块编号（区块链高度）。

- block.timestamp

矿工放置在当前块中的时间戳（自 Unix 纪元以来的秒数）。

- 地址对象
  任何地址，无论是作为输入传递还是从合约对象转换，都有许多属性和方法：

- address.balance
  地址的余额，在 wei。例如，当前合约余额是地址（this）.balance。

- address.transfer（__ amount__）
  将金额（以 wei 为单位）转移到此地址，对任何错误抛出异常。我们在我们的 Faucet 示例中使用此函数作为 msg.sender 地址的方法，如 msg.sender.transfer。

- address.send（__ amount__）
  与 transfer 类似，只是在抛出异常时，它会在出错时返回 false。警告：始终检查发送的返回值。

- address.call（__ payload__）
  低级 CALL 函数 - 可以使用数据有效负载构造任意消息调用。错误时返回 false。警告：不安全的接收者可能（意外或恶意）用尽所有 gas，导致合约因 OOG 异常而停止； 总是检查调用的返回值。

- address.callcode（__ payload__）
  低级 CALLCODE 函数，如地址（this）.call（...），但此合约的代码替换为地址代码。错误时返回 false。警告：仅限高级使用！

- address.delegatecall（）
  低级 DELEGATECALL 函数，如 callcode（...），但具有当前合约所见的完整 msg 上下文。错误时返回 false。警告：仅限高级使用！

内置功能

其他值得注意的功能是：

- addmod，mulmod
  对于模加法和乘法。例如，addmod（x，y，k）计算（x + y）%k。

- keccak256，sha256，sha3，ripemd160
  使用各种标准哈希算法计算哈希值的函数。

- ecrecover
  恢复用于签名消息的地址。

- selfdestrunct（__ recipient_address__）
  删除当前合约，将帐户中任何剩余的以太网发送到收件人地址。

- this
  当前正在执行的合约帐户的地址。

合约定义（Contract Definition）

Solidity的主要数据类型是合约； 我们的Faucet示例只是定义了一个contract对象。与面向对象语言中的任何对象类似，合约是包含数据和方法的容器。

Solidity 提供了两个与合约类似的对象类型：

- 接口（interface）
  接口定义的结构与合约完全相同，除了没有定义任何函数，它们仅被声明。这种类型的声明通常称为存根 ； 它告诉你函数的参数和返回类型，没有任何实现。界面指定合约的“形状”； 在继承时，接口声明的每个函数都必须由子进程定义。

- library
  库合约是指仅使用 delegatecall 方法仅部署一次并由其他合约使用的合约（请参阅地址对象）。

- Functions
  在合约中，我们定义可以由EOA交易或另一个合约调用的函数。在我们的Faucet示例中，我们有两个函数：withdraw 和（未命名）fallback 函数。

我们用于在 Solidity 中声明函数的语法如下：

```
function FunctionName（[ parameters ]）{public | private | internal | external}
[pure | constant | view | payable] [ modifiers ] [ return（return types）]
```

让我们看看每个组件：

- FunctionName

函数的名称，用于在交易（来自 EOA），另一个合约或甚至来自同一合约中调用函数。可以在没有命名的情况下定义每个合约中的一个函数，在这种情况下，它是回调函数，当没有命名其他函数时调用该函数。回调函数不能包含任何参数或返回任何内容。

- 参数

在名称后面，我们指定必须传递给函数的参数及其名称和类型。在我们的 Faucet 示例中，我们将 uint withdraw_amount 定义为 withdraw 函数的唯一参数。

下一组关键字（公共，私有，内部，外部）指定功能的可见性：

- public

public 是默认的； 这些功能可以通过其他合约或 EOA 交易或合约内部来调用。在我们的 Faucet 示例中，两个函数都定义为 public。

- external

external 函数类似于 public 函数，除非它们不能在合约中调用，除非明确地以关键字 this 为前缀。

- internal

internal 功能只能从合约中访问 - 它们不能被另一个合约或 EOA 交易调用。它们可以通过派生合约（那些继承了这个合约）来调用。

- private

私有函数类似于内部函数，但不能由派生合约调用。

请记住，internal 和 private 这两个词有些误导。合约中的任何功能或数据始终在公共区块链上可见，这意味着任何人都可以看到代码或数据。此处描述的关键字仅影响函数的调用方式和时间。

第二组关键字(pure, constant, view, payable)会影响函数的行为：

- constant or view

标记为 view 的函数承诺不修改任何状态。术语常量是 view 的别名，将在以后的版本中弃用。此时，编译器不强制执行 view 修饰符，仅生成警告，但这应该成为 Solidity v0.5 中的强制关键字。

- pure

pure 函数是既不读取也不写入存储中的任何变量的函数。它只能在参数和返回数据上运行，而不需要引用任何存储的数据。pure 函数旨在鼓励没有副作用或状态的声明式编程。

- Payable

payable 是可以接受收款的职能。未声明为应付款的功能将拒绝收款。有两个例外，由于在 EVM 设计决定：coinbase 付款，即使回调函数未声明为可支付，自毁继承将被支付，但是这是有道理的，因为无论如何代码的执行不是这些付款的一部分。

正如您在我们的 Faucet 示例中所看到的，我们有一个应付功能（回调函数），这是唯一可以接收收款的功能。

合约构造函数和自我构造

有一个特殊功能只使用一次。创建合约时，如果存在合约，它还会运行构造函数，以初始化合约的状态。构造函数在与合约创建相同的交易中运行。构造函数是可选的； 你会注意到我们的龙头示例没有。

可以通过两种方式指定构造函数。在 Solidity v0.4.21 中，构造函数是一个名称与合约名称相匹配的函数，如下所示：

```
contract MEContract {
        function MEContract() {
                // This is the constructor
        }
}
```

这种格式的难点在于，如果更改了合约名称并且未更改构造函数名称，则它不再是构造函数。同样，如果在合约和/或构造函数的命名中存在意外错误，则该函数也不再是构造函数。这可能会导致一些非常讨厌，意外和难以发现的错误。想象一下，例如，构造函数是否为了控制目的而设置合约的所有者。如果由于命名错误而该函数实际上不是构造函数，则不仅在创建合约时将所有者保留为未设置，而且该函数也可以部署为合约的永久和“可调用”部分，如正常功能，允许任何第三方劫持合约，并在合约创建后成为“所有者”。

为了解决构造函数基于具有与合约相同的名称的潜在问题，Solidity v0.4.22 引入了一个构造函数关键字，其操作类似于构造函数但没有名称。重命名合约根本不会影响构造函数。此外，更容易识别哪个函数是构造函数。它看起来像这样：

```
pragma ^ 0。4。22
contract MEContract {
        constructor（）{
                //这是构造函数
```

```
        }
}
```

总而言之，合约的生命周期始于 EOA 或合约账户的创建交易。如果存在构造函数，则它将作为合约创建的一部分执行，以在创建合约时初始化合约的状态，然后将其丢弃。

合约生命周期的另一端是合约破坏。合约被称为 SELFDESTRUCT（自毁）的特殊 EVM 操作码破坏。它曾经被称为 SUICIDE(自杀),但由于该词的负面关联,该名称已被弃用。在 Solidity 中，此操作码作为高级内置函数公开，称为 selfdestruct，它接受一个参数：接收合约帐户中剩余的任何以太币余额的地址。它看起来像这样：

```
selfdestruct(address recipient);
```

请注意，如果要将此命令删除，则必须将此命令显式添加到合约中 - 这是删除合约的唯一方法，默认情况下不存在。通过这种方式，可能依赖合约永远存在的合约的用户可以确定如果合约不包含 SELFDESTRUCT 操作码，则不能删除合约。

添加构造函数和 selfdestruct 到我们的 Faucet 示例

我们在[intro_chapter]中介绍的Faucet示例合约没有任何构造函数或selfdestruct函数。这是一份无法删除的永恒合约。让我们通过添加构造函数和 selfdestruct 函数来改变它。我们可能希望 selfdestruct 只能由最初创建合约的 EOA 调用。按照惯例，这通常存储在名为 owner 的地址变量中。我们的构造函数设置所有者变量，selfdestruct 函数将首先检查所有者是否直接调用它。

首先，我们的构造函数：

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.22;

// Our first contract is a faucet!
contract Faucet {

        address owner;

        // Initialize Faucet contract: set owner
        constructor() {
                owner = msg.sender;
        }

[...]
```

我们已经更改了 pragma 指令，将 v0.4.22 指定为此示例的最小版本，因为我们使用了 Solidity v0.4.22 中引入的新构造函数关键字。我们的合约现在有一个名为 owner 的地址类型变量。“所有者”这个名称在任何方面都不是特别的。我们可以将此地址变量称为“马铃薯”，并仍然以相同的方式使用它。名称所有者只是明确了其目的。

接下来，我们的构造函数作为合约创建交易的一部分运行，将 msg.sender 中的地址分配给所有者变量。我们在 withdraw 函数中使用了 msg.sender 来识别提取请求的发起者。但是，在构造函数中，msg.sender 是启动合约创建的 EOA 或合约地址。我们知道这是因为这是一个构造函数：它只在合约创建期间运行一次。

现在我们可以添加一个功能来销毁合约。我们需要确保只有所有者才能运行此函数，因此我们将使用 require 语句来控制访问。以下是它的外观：

```
// Contract destructor
function destroy() public {
        require(msg.sender == owner);
        selfdestruct(owner);
}
```

如果有人从除了所有者之外的地址调用此 destroy 函数，它将失败。但是，如果构造函数存储在所有者中的相同地址调用它，则合约将自毁并将任何剩余余额发送到所有者地址。请注意,我们没有使用不安全的 tx.origin 来确定所有者是否希望销毁合约 - 使用 tx.orgin 会允许恶意合约在未经您许可的情况下销毁您的合约。

功能修饰符

Solidity 提供了一种称为函数修饰符的特殊函数。通过在函数声明中添加修饰符名称，可以将修饰符应用于函数。修饰符通常用于创建适用于合约中许多函数的条件。我们的 destroy 函数中已经有一个访问控制语句。让我们创建一个表达该条件的函数修饰符：

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
```

此函数修饰符（名为 onlyOwner）在其修改的任何函数上设置条件，要求存储为合约所有者的地址与交易的 msg.sender 的地址相同。这是访问控制的基本设计模式，只允许合约的所有者执行具有 onlyOwner 修饰符的任何函数。

你可能已经注意到我们的函数修饰符里面有一个特殊的句法“占位符”，一个下划线后跟一个分号（＆＃95；）。此占位符由正在修改的函数的代码替换。本质上，修饰符“包裹”修改后的函数，将其代码放在由下划线字符标识的位置。

要应用修饰符，请将其名称添加到函数声明中。可以将多个修饰符应用于函数； 它们以声明的顺序应用，以逗号分隔的列表。

让我们重写我们的 destroy 函数以使用 onlyOwner 修饰符：

```
function destroy() public onlyOwner {
    selfdestruct(owner);
}
```

函数修饰符的名称（onlyOwner）位于关键字 public 之后，并告诉我们 destroy 函数由 onlyOwner 修饰符修改。从本质上讲，您可以将其视为“只有所有者可以销毁此合约”。实际上，生成的代码相当于“包装”仅来自 destroyOwner 的代码。

函数修饰符是一个非常有用的工具，因为它们允许我们为函数编写前提条件并一致地应用它们，使代码更易于阅读，因此更容易审计安全性。它们最常用于访问控制，但它们非常通用，可用于各种其他目的。

在修饰符中，您可以访问修改后的函数可见的所有值（变量和参数）。在这种情况下，我们可以访问在合约中声明的所有者变量。但是，反之亦然：您无法访问修改后的函数中的任何修饰符变量。

合约继承

Solidity 的合约对象支持继承，继承是一种具有附加功能的扩展基本合约的机制。要使用继承，请使用关键字指定父合约：

```
contract Child is Parent {
  ...
}
```

使用此构造，Child 合约继承 Parent 的所有方法，功能和变量。Solidity 还支持多重继承，可以在以下关键字后使用逗号分隔的合约名称指定：

```
contract Child is Parent1, Parent2 {
  ...
}
```

合约继承允许我们用实现模块化，可扩展性和重用的方式编写合约。我们从简单的合约开始，实现最通用的功能，然后通过在更专业的合约中继承这些功能来扩展它们。

在我们的 Faucet 合约中，我们介绍了构造函数和析构函数，以及在构造时分配的所有者的访问控制。这些功能非常通用：许多合约都有它们。我们可以将它们定义为通用合约，然后使用继承将它们扩展到 Faucet 合约。

我们首先定义一个拥有的基本合约，它有一个所有者变量，在合约的构造函数中设置它：

```
contract owned {
        address owner;

        // Contract constructor: set owner
        constructor() {
                owner = msg.sender;
        }

        // Access control modifier
        modifier onlyOwner {
            require(msg.sender == owner);
            _;
        }
}
```

接下来，我们定义一个基础合约-mortal，它继承了 owned：

```
contract mortal is owned {
        // Contract destructor
        function destroy() public onlyOwner {
                selfdestruct(owner);
        }
}
```

如您所见，mortal 合约可以使用 onlyOwner 函数修饰符，在自己定义。它间接地还使用所有者地址变量和所拥有的构造函数。继承使每个合约更简单，并专注于其特定功能，允许我们以模块化方式管理细节。

现在我们可以进一步扩展自有合约，继承其在 Faucet 中的能力：

```
contract Faucet is mortal {
    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
```

```
        require(withdraw_amount <= 0.1 ether);
        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
    // Accept any incoming amount
    function () public payable {}
}
```

通过继承 mortal，而 mortal 又继承了 owned，mortal 合约现在拥有构造函数和销毁函数，以及一个已定义的所有者。功能与 Faucet 中的功能相同，但现在我们可以在其他合约中重用这些功能而无需再次编写。代码重用和模块化使我们的代码更清晰，更易于阅读，更易于审计。

错误处理（断言，要求，还原）

合约调用可以终止并返回错误。Solidity 中的错误处理由四个函数处理：assert，require，revert 和 throw（现已弃用）。

当合约以错误终止时，如果调用了多个合约，则所有状态更改（对变量，余额等的更改）都会被恢复，从而一直到合约链调用。这可以确保交易是原子的，这意味着它们要么成功完成，要么对状态没有影响并完全恢复。

assert 和 require 函数以相同的方式运行，评估条件并在条件为假时停止执行错误。按照惯例，当结果预期为真时使用 assert，这意味着我们使用 assert 来测试内部条件。相比之下，在测试输入（例如函数参数或交易字段）时使用 require，设置我们对这些条件的期望。

我们在函数修饰符 onlyOwner 中使用了 require 来测试消息发送者是合约的所有者：

```
require(msg.sender == owner);
```

require 函数充当门控条件，阻止执行函数的其余部分并在不满足时产生错误。

从 Solidity v0.4.22 开始，require 还可以包含一条有用的文本消息，可用于显示错误原因。错误消息记录在交易日志中。因此，我们可以通过在 require 函数中添加错误消息来改进我们的代码：

```
require(msg.sender == owner, "Only the contract owner can call this
function");
```

revert 和 throw 函数停止合约的执行并恢复任何状态更改。throw 函数已过时，将在以后的 Solidity 版本中删除； 你应该使用还原。恢复功能还可以将错误消息作为唯一参数，该参数记录在交易日志中。

无论我们是否明确检查合约中的某些条件，都会产生错误。例如，在我们的水龙头合约中，我们不会检查是否有足够的以满足提款请求。这是因为传输函数会因错误而失败，如果余额不足以进行转移，则还原交易：

```
msg.sender.transfer(withdraw_amount);
```

但是，最好明确检查并在失败时提供明确的错误消息。我们可以通过在传输之前添加 require 语句来做到这一点：

```
require(this.balance >= withdraw_amount,
        "Insufficient balance in faucet for withdrawal request");
msg.sender.transfer(withdraw_amount);
```

像这样的附加错误检查代码会略微增加 gas 消耗，但它提供的错误报告比省略的更好。您需要根据合约的预期用途在 gas 消耗和详细错误检查之间找到适当的平衡点。如果是针对测试网的 Faucet 合约，我们可能会在额外的报告方面犯错，即使它需要更多的 gas。也许对于一个主网络合约，我们选择节省使用我们的 gas。

Events(事件)

当交易完成（成功与否）时，它会产生交易收据，我们将在[evm_chapter]中看到。交易接收包含日志条目，这些日志条目提供有关在执行交易期间发生的操作的信息。事件是用于构造这些日志的 Solidity 高级对象。

事件对轻客户端和 DApp 服务特别有用，它们可以“监视”特定事件并将其报告给用户界面，或者更改应用程序的状态以反映基础合约中的事件。

事件对象接受序列化并记录在区块链中的交易日志中的参数。您可以在参数之前提供索引的关键字，以使值成为可由应用程序搜索或过滤的索引表（哈希表）的一部分。

到目前为止，我们还没有在我们的 Faucet 示例中添加任何事件，所以让我们这样做。我们将添加两个事件，一个用于记录任何提款，另一个用于记录任何存款。我们将分别称这些事件提款和存款。首先，我们在 Faucet 合约中定义事件：

```
contract Faucet is mortal {
        event Withdrawal(address indexed to, uint amount);
        event Deposit(address indexed from, uint amount);
```

```
        [...]
}
```

我们选择将地址编入索引，以允许在任何用于访问我们的 Faucet 的用户界面中进行搜索和过滤。

接下来，我们使用 emit 关键字将事件数据合并到交易日志中：

```
// Give out ether to anyone who asks
function withdraw(uint withdraw_amount) public {
    [...]
    msg.sender.transfer(withdraw_amount);
    emit Withdrawal(msg.sender, withdraw_amount);
}
// Accept any incoming amount
function () public payable {
    emit Deposit(msg.sender, msg.value);
}
```

由此产生的 Faucet.sol 合约看起来像 Faucet8.sol：修改后的 Faucet 合约，包括事件。

示例 3. Faucet8.sol：修改了 faucet 合约，包含事件

```
link:code/Solidity/Faucet8.sol[]
```

捕获事件

好的，所以我们设置合约来发出事件。我们如何看待交易结果并“捕获”事件？web3.js 库提供包含交易日志的数据结构。在那些我们可以看到交易生成的事件。

让我们使用 truffle 在修订后的 Faucet 合约上运行测试交易。按照[truffle]中的说明设置项目目录并编译 Faucet 代码。源代码可以在 code / truffle / FaucetEvents 下的书籍 GitHub 存储库中找到。

```
$ truffle develop
truffle(develop)> compile
truffle(develop)> migrate
Using network 'develop'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xb77ceae7c3f5afb7fbe3a6c5974d352aa844f53f955ee7d707ef6f3f8e6b4e61
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
```

```
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Faucet...
  ... 0xfa850d754314c3fb83f43ca1fa6ee20bc9652d891c00a2f63fd43ab5bfb0d781
  Faucet: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...

truffle(develop)> Faucet.deployed().then(i => {FaucetDeployed = i})
truffle(develop)> FaucetDeployed.send(web3.toWei(1, "ether")).then(res => \
                  { console.log(res.logs[0].event, res.logs[0].args) })
Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }
truffle(develop)> FaucetDeployed.withdraw(web3.toWei(0.1,
"ether")).then(res => \
                  { console.log(res.logs[0].event, res.logs[0].args) })
Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }
```

使用 deployed 函数部署合约后，我们执行两个交易。第一笔交易是存款（使用发送），在交易日志中发出存款事件：

```
Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }
```

接下来，我们使用 withdraw 函数进行提款。这会发出提款事件：

```
Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }
```

为了获得这些事件，我们查看了作为交易结果（res）返回的 logs 数组。第一个日志条目（logs [0]）包含 logs [0] .event 中的事件名称和 logs [0] .args 中的事件参数。通过在控制台上显示这些，我们可以看到发出的事件名称和事件参数。

事件是一种非常有用的机制，不仅用于合约内通信，还用于开发期间的调试。

调用其他合约（发送，调用，调用代码，委托调用）

从合约中调用其他合约是非常有用但有潜在危险的操作。我们将研究实现这一目标的各种方法，并评估每种方法的风险。简而言之，风险源于这样一个事实，即您可能对您正在调

用的合约或调用合约的合约知之甚少。在编写智能合约时，您必须记住，虽然您可能大多数期望处理 EOA，但没有什么可以阻止任意复杂的，也许是恶意的合约不会被您的代码调用。

创建一个新实例

如果您自己创建其他合约，最安全的方式是调用给另一份合约。这样，您就可以确定其界面和行为。为此，您可以使用关键字 new 来简单地实例化它，就像在其他面向对象语言中一样。在 Solidity 中，关键字 new 将在区块链上创建合约并返回可用于引用它的对象。假设您要在另一个名为 Token 的合约中创建并调用 Faucet 合约：

```
contract Token is mortal {
        Faucet _faucet;

        constructor() {
                _faucet = new Faucet();
        }
}
```

这种合约构造机制可确保您了解合约的确切类型及其界面。合约 Faucet 必须在 Token 的范围内定义，如果定义在另一个文件中，您可以使用 import 语句：

```
import "Faucet.sol";

contract Token is mortal {
        Faucet _faucet;

        constructor() {
                _faucet = new Faucet();
        }
}
```

您可以选择在创建时指定 ether transfer 的值，并将参数传递给新合约的构造函数：

```
import "Faucet.sol";

contract Token is mortal {
        Faucet _faucet;

        constructor() {
                _faucet = (new Faucet).value(0.5 ether)();
        }
}
```

您也可以调用 Faucet 函数 。在这个例子中，我们从 Token 的 destroy 函数中调用 Faucet 的 destroy 函数：

```
import "Faucet.sol";

contract Token is mortal {
        Faucet _faucet;

        constructor() {
                _faucet = (new Faucet).value(0.5 ether)();
        }

        function destroy() ownerOnly {
                _faucet.destroy();
        }
}
```

请注意，虽然您是 Token 合约的所有者，但 Token 合约本身拥有新的 Faucet 合约，因此只有 Token 合约才能销毁它。

解决现有实例

您可以调用合约的另一种方法是转换合约现有实例的地址。使用此方法，可以将已知接口应用于现有实例。因此，至关重要的是，您确实知道您正在寻址的实例实际上是您所假设的类型。我们来看一个例子：

```
import "Faucet.sol";
contract Token is mortal {
        Faucet _faucet;
        constructor(address _f) {
                _faucet = Faucet(_f);
                _faucet.withdraw(0.1 ether)
        }
}
```

这里，我们将一个地址作为参数提供给构造函数_f，然后将它转换为 Faucet 对象。这比以前的机制风险更大，因为我们不确定该地址是否实际上是一个 Faucet 对象。当我们调用 withdraw 时，我们假设它接受相同的参数并执行与我们的 Faucet 声明相同的代码，但我们无法确定。据我们所知，这个地址的撤销功能可以执行与我们预期完全不同的东西，即使命名相同。因此，使用作为输入传递的地址并将它们转换为特定对象比自己创建合约要危险得多。

原始调用，代理调用

Solidity 为调用其他合约提供了一些更“低级”的功能。这些直接对应于同名的 EVM 操作码，并允许我们手动构建合约到合约的呼叫。因此，它们代表了调用其他合约的最灵活和最危险的机制。

以下是使用调用方法的相同示例：

```
contract Token is mortal {
        constructor(address _faucet) {
                _faucet.call("withdraw", 0.1 ether);
        }
}
```

正如您所看到的，这种类型的调用是对函数的盲调，非常类似于构建原始交易，仅在合约的上下文中。它可以使您的合约暴露于许多安全风险，最重要的是可重入，我们将在[reentrancy_security]中更详细地讨论。如果出现问题，call 函数将返回 false，因此您可以评估错误处理的返回值：

```
contract Token is mortal {
        constructor(address _faucet) {
                if !(_faucet.call("withdraw", 0.1 ether)) {
                        revert("Withdrawal from faucet failed");
                }
        }
}
```

另一种调用方式是 delegatecall，它取代了更危险的调用代码。该 callcode 方法很快就会被弃用，因此不应该使用它。

正如地址对象中所提到的，代理调用与调用的不同之处在于 msg 上下文不会更改。例如，虽然调用将 msg.sender 的值更改为调用合约，但 delegatecall 保持与调用合约中相同的 msg.sender。从本质上讲，delegatecall 在执行当前合约的上下文中运行另一个合约的代码。它最常用于从库中调用代码。它还允许您使用存储在其他位置的库函数的模式，但使该代码与合约的存储数据一起使用。

应谨慎使用代理调用。它可能会产生一些意想不到的效果，特别是如果您调用的合约不是设计为库。

让我们使用一个示例合约来演示 call 和 delegatecall 用于调用库和合约的各种调用语义。在 CallExamples.sol 中：我们使用一个事件来记录每个调用的详细信息，并根据调用类型查看调用上下文的变化情况。

示例 4. CallExamples.sol：不同调用语义的示例

link:code/truffle/CallExamples/contracts/CallExamples.sol[]

正如您在此示例中所看到的，我们的主要合约是调用者，它调用名为 Library 的库和名为 Contont 的合约。被调用的库和合约都有相同的 calledFunction 函数，它们发出一个名为 Event 的事件。名为 Event 的事件记录了三个数据：msg.sender，tx.origin 和 this。每次调用 calledFunction 时，它都可能具有不同的执行上下文（可能具有所有上下文变量的不同值），具体取决于是直接调用还是通过 delegatecall 调用。

在调用者中，我们首先通过在每个函数中调用 calledFunction 来直接调用合约和库。然后，我们显式地使用低级函数 call 和 delegatecall 来调用 calledContract.calledFunction。通过这种方式，我们可以看到各种调用机制的行为方式。

让我们在 Truffle 开发环境中运行它并捕获事件，看看它的外观：

```
truffle(develop)> migrate
Using network 'develop'.
[...]
Saving artifacts...
truffle(develop)> web3.eth.accounts[0]
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
truffle(develop)> caller.address
'0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
truffle(develop)> calledContract.address
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'
truffle(develop)> calledLibrary.address
'0xf25186b5081ff5ce73482ad761db0eb0d25abfbf'
truffle(develop)> caller.deployed().then( i => { callerDeployed = i })

truffle(develop)>
callerDeployed.make_calls(calledContract.address).then(res => \
                  { res.logs.forEach( log => { console.log(log.args) })})
{ sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' }
{ sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' }
```

让我们看看这里发生了什么。我们调用了 make_calls 函数并传递了 calledContract 的地址，然后捕获了每个不同调用发出的四个事件。让我们看看 make_calls 函数并逐步完成每一步。

第一个调用是：

```
_calledContract.calledFunction（）；
```

在这里，我们使用高级 ABI for calledFunction 直接调用 calledContract.calledFunction。发出的事件是：

```
sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

如您所见，msg.sender 是调用者合约的地址。tx.origin 是我们的帐户 web3.eth.accounts [0]的地址，它将交易发送给调用者。事件是由 calledContract 发出的，我们可以从事件的最后一个参数中看到。

make_calls 中的下一个调用是对库：

```
calledLibrary.calledFunction（）；
```

它看起来与我们称之为合约的方式相同，但行为却截然不同。让我们看看发出的第二个事件：

```
sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
```

这次，msg.sender 不是调用者的地址。相反，它是我们帐户的地址，与交易来源相同。那是因为当你调用一个库时，调用总是代理调用并在调用者的上下文中运行。因此，当被调用的库代码运行时，它继承了调用者的执行上下文，就好像它的代码在调用者内部运行一样。变量 this（在发出的事件中显示）是调用者的地址，即使它是从内部访问的 calledLibrary。

接下来的两个调用，使用低级调用和 delegatecall，验证我们的期望，发出反映我们刚看到的事件。

GAS 考虑因素

在[gas]中更详细地描述的 gas 是智能合约编程中非常重要的考虑因素。Gas 是限制以太坊允许交易消耗的最大计算量的资源。如果在计算过程中超出了 gas 限制，则会发生以下一系列事件：

- 抛出“out of gas”的例外。
  执行前的合约状态将恢复（恢复）。

- 用于支付 gas 的所有以太都作为交易费用； 它没有退款。
  因为 gas 是由发起交易的用户支付的，所以不鼓励用户调用具有高 gas 成本的功能。因此，最小化合约功能的 gas 成本符合程序员的最佳利益。为此，在构建智能合约时建议采用某些做法，以便最小化函数调用的 gas 成本。

- 避免使用动态数组
  任何循环通过动态大小的数组，其中函数对每个元素执行操作或搜索特定元素会引入使用过多 gas 的风险。实际上，合约可能在找到所需结果之前或在对每个元件起作用之前耗尽 gas，从而浪费时间和以太而根本不给出任何结果。

- 避免调用其他合约
  调用其他合约，特别是当其功能的 gas 成本未知时，会引入 gas 耗尽的风险。避免使用未经过充分测试和广泛使用的库。library 从其他程序员那里得到的审查越少，使用它的风险就越大。

- 估算 gas 成本
  如果您需要根据其参数估算执行某种合约方法所需的 gas，您可以使用以下程序：

```
var contract = web3.eth.contract(abi).at(address);
var gasEstimate = contract.myAweSomeMethod.estimateGas(arg1, arg2,
    {from: account});
```

gasEstimate 将告诉您执行所需的 gas 单元数量。这是一个估计，因为 EVM 的图灵完备 - 创建一个功能相对微不足道，这个功能将采用大量不同的 gas 来执行不同的调用。即使是生产代码也可能以微妙的方式改变执行路径，从而导致从一个呼叫到下一个呼叫的 gas 成本大不相同。然而，大多数功能都是合理的，估计 gas 在大多数情况下会得到很好的估计。

要从网络获得 gas 价格，您可以使用：

```
var gasPrice = web3.eth.getGasPrice();
```

从那里你可以估算 gas 费用：

```
var gasCostInEther = web3。fromWei（（gasEstimate * gasPrice），' ether '）;
```

让我们使用我们的 gas 估算功能来估算我们的 Faucet 示例的 gas 成本，使用 book`s 存储库中的代码。

在开发模式启动 truffle 和执行 JavaScript 文件 gas_estimates.js：使用 estimateGas 功能，gas_estimates.js。

示例 5. gas_estimates.js：使用 estimateGas 函数

```
var FaucetContract = artifacts.require("./Faucet.sol");

FaucetContract.web3.eth.getGasPrice(function(error, result) {
    var gasPrice = Number(result);
    console.log("Gas Price is " + gasPrice + " wei"); // "10000000000000"

    // Get the contract instance
    FaucetContract.deployed().then(function(FaucetContractInstance) {

                // Use the keyword 'estimateGas' after the function name to
get the gas
                // estimation for this particular function (aprove)
                FaucetContractInstance.send(web3.toWei(1, "ether"));
        return FaucetContractInstance.withdraw.estimateGas(web3.toWei(0.1,
"ether"));

    }).then(function(result) {
        var gas = Number(result);

        console.log("gas estimation = " + gas + " units");
        console.log("gas cost estimation = " + (gas * gasPrice) + " wei");
        console.log("gas cost estimation = " +
                FaucetContract.web3.fromWei((gas * gasPrice), 'ether') + "
ether");
    });
});
```

以下是 Truffle 开发控制台的样式：

```
$ truffle develop

truffle(develop)> exec gas_estimates.js
Using network 'develop'.

Gas Price is 20000000000 wei
gas estimation = 31397 units
gas cost estimation = 627940000000000 wei
gas cost estimation = 0.00062794 ether
```

建议您在开发工作流程中评估功能的 gas 成本，以避免在将合约部署到主网络时出现任何意外情况。

结论

在本章中，我们开始详细介绍智能合约，并探讨了 Solidity 合约编程语言。我们采用了一个简单的示例合约 Faucet.sol，并逐渐改进它并使其更复杂，使用它来探索 Solidity 语言的各个方面。在[vyper_chap]中，我们将使用另一种面向合约的编程语言 Vyper。我们将比较 Vyper 与 Solidity，展示这两种语言设计的一些差异，并加深我们对智能合约编程的理解。

第八章 智能合约及 Vyper

智能合约及 Vyper

Vyper 是一种用于以太坊虚拟机的，面向合约的，处于实验期的编程语言。它允许开发人员写出易于理解的代码，从而提供了卓越的可审计性。事实上，Vyper 的一个原则是让开发人员几乎不可能编写误导性代码。

在本章中，我们将介绍智能合约的常见问题，介绍 Vyper 合同编程语言，并将其与 Solidity 进行比较以展示差异。

漏洞和 Vyper

最近的一项研究分析了近百万个已经部署的以太坊智能合约，发现其中许多合约都存在严重的漏洞。在分析过程中，研究人员阐述了三种基本的漏洞：

- 自杀合约：有可能被任意地址终止的智能合约
- 贪婪合约：有可能变成无法发送以太的智能合约
- 浪子合约：有可能被滥用，将以太币发布到任意地址的智能合约

漏洞通过代码引入智能合约中。可能有人会说没人会故意在合约中留下漏洞，但无论如何，有瑕疵的智能合约会让用户损失一部分财产，而这并不是我们希望看到的。Vyper 旨在让编写安全的合约代码更加简单，或者用另一句话说，更难编写别人无法看懂或者有漏洞的代码。

与 Solidity 的比较

Vyper 为了更难编写不安全的代码会故意忽略一些 Solidity 的功能。因此想要使用 Vyper 编写合约的人应该清楚的意识到哪些功能 Vyper 是没有的，以及为什么没有这些功能。

修饰符

正如我们在前一章中看到的，在 Solidity 中，您可以用修饰符编写函数。例如，函数 changeOwner 将在名为 onlyBy 的修饰符中运行代码作为其执行的一部分：

```
function changeOwner(address _newOwner)
    public
    onlyBy(owner)
{
    owner = _newOwner;
}
```

此修饰符强制执行关于所有权的规则。如您所见，此特定修饰符代表了 changeOwner 函数并且执行了预检查：

```
modifier onlyBy(address _account)
{
    require(msg.sender == _account);
    _;
}
```

在这我们可以发现，修饰符不仅仅是为了执行预检查。实际上，作为修饰符，它们可以在调用函数的上下文中显着改变智能合约的环境。简而言之，修饰符是普遍存在的。

接下来让我们看看另一个 Solidity 风格的例子：

```
enum Stages {
    SafeStage
    DangerStage,
    FinalStage
}
uint public creationTime = now;
```

```
Stages public stage = Stages.SafeStage;
function nextStage() internal {
    stage = Stages(uint(stage) + 1);
}
modifier stageTimeConfirmation() {
    if (stage == Stages.SafeStage &&
                now >= creationTime + 10 days)
        nextStage();
    _;
}
function a()
    public
    stageTimeConfirmation
    // More code goes here
{
}
```

一方面，开发人员应该始终检查他们自己的代码调用的任何其他代码。但是，在某些情况下（如时间受到限制或者时间比较紧张导致注意力不集中），开发人员可能会忽略某行代码。如果开发人员必须在大型工程源码中不断跳转，同时还要跟踪函数调用逻辑并将智能合约变量的状态提交到内存，则更有可能发生这种上述这种情况。

让我们更深入地看一下前面的例子。假设开发人员正在编写一个名为 a 的公共函数。开发人员对此合约不熟悉，并且正在使用其他人编写的修饰符。乍一看，似乎 stageTimeConfirmation 装饰器只是在简单地执行一些关于合约与调用函数时间的检查。开发人员可能没有意识到装饰器也在调用另一个函数，nextStage。在这种比较简单的情况下，简单地调用公共函数 a 会导致智能合约的 stage 变量从 SafeStage 变为 DangerStage。

Vyper 完全废除了装饰器。Vyper 的建议如下：如果只是使用装饰器进行“断言”，那么只需使用内联检查及断言作为函数的一部分； 如果要修改智能合约状态等，相似的，还是将这些更改显式的作为该函数的一部分。这样做可以提高代码的可审计性和可读性，因为读者不需要费劲费力的将装饰器代码填回函数来查看其功能。

类继承

继承允许程序员通过从现有软件库中获取之前存在的功能、属性和行为来充分利用之前编写完的代码。继承的功能很强大，因为它可以促进代码的复用。Solidity 支持多重继承以及多态，但虽然这些是面向对象编程的一些关键特征，但 Vyper 不支持它们。Vyper 坚持认为继承的实现要求编码人员和审计人员在多个文件之间跳转才能了解程序正在做什么。Vype

r 还认为多重继承可能使代码过于复杂而无法理解 - 这点我们从 Solidity 的文档中可以略知一二，因为文档描述了一个多重继承可能会产生问题的例子。

内联汇编

内联汇编为开发人员提供了对以太坊虚拟机的低级访问，允许 Solidity 程序通过直接访问 EVM 指令来执行操作。例如，以下内联汇编代码将 3 添加到内存位置 0x80：

```
3 0x80 mload add 0x80 mstore
```

Vyper 认为为了这些效率提升极大牺牲了代码的可读性，因此不支持内联汇编。

函数重载

函数重载允许开发人员编写多个同名的函数。在不同的情况下使用哪个函数取决于所提供的参数的类型。例如下面两个例子给出的那样：

```
function f(uint _in) public pure returns (uint out) {
    out = 1;
}
function f(uint _in, bytes32 _key) public pure returns (uint out) {
    out = 2;
}
```

第一个函数（名为 f）接受 uint 类型的输入参数； 第二个函数（也称为 f）接受两个参数，一个是 uint 类型，另一个是 bytes32 类型。具有相同名称的多个函数定义采用不同的参数可能会造成混淆，因此 Vyper 不支持函数重载。

变量类型转换

有两种类型转换：隐式和显式

隐式类型转换通常在编译时执行。例如，如果类型转换在语义上是合理的并且没有信息可能丢失，则编译器可以执行隐式转换，例如将类型为 uint8 的变量转换为 uint16。最早版本的 Vyper 允许对变量进行隐式类型转换，但最新版本却没有。

可以在 Solidity 中插入显式类型转换。不幸的是，它们可能导致意外行为。例如，将 uint32 转换为较小的类型 uint16 只会删除高阶位，如下所示：

```
uint32 a = 0x12345678;
uint16 b = uint16(a);
// Variable b is 0x5678 now
```

但是，Vyper 有一个转换函数可以用来执行显式转换。convert 函数（在 convert.py 的第 82 行）：

```
def convert(expr, context):
    output_type = expr.args[1].s
    if output_type in conversion_table:
        return conversion_table[output_type](expr, context)
    else:
        raise Exception("Conversion to {} is invalid.".format(output_typ
e))
```

注意使用的 conversion_table（在同一文件的第 90 行），如下所示：

```
conversion_table = {
    'int128': to_int128,
    'uint256': to_unint256,
    'decimal': to_decimal,
    'bytes32': to_bytes32,
}
```

当开发人员调用 convert 时，它会通过引用 conversion_table 来确保适当的转换。例如，如果开发人员将 int128 传递给 convert 函数，将会执行 convert.py 文件第 26 行的 to_int128 函数。to_int128 函数如下：

```
@signature(('int128', 'uint256', 'bytes32', 'bytes'), 'str_literal')
def to_int128(expr, args, kwargs, context):
    in_node = args[0]
    typ, len = get_type(in_node)
    if typ in ('int128', 'uint256', 'bytes32'):
        if in_node.typ.is_literal
            and not SizeLimits.MINNUM <= in_node.value <= SizeLimits.MAXN
UM:
            raise InvalidLiteralException(
                "Number out of range: {}".format(in_node.value), expr
```

```
            )
        return LLLnode.from_list(
            ['clamp', ['mload', MemoryPositions.MINNUM], in_node,
            ['mload', MemoryPositions.MAXNUM]], typ=BaseType('int128'),
            pos=getpos(expr)
        )
    else:
        return byte_array_to_num(in_node, expr, 'int128')
```

如您所见，转换过程确保不会有任何信息丢失； 如果可能会有信息丢失，函数则会触发异常。这个转换代码可以防止截断以及隐式类型转换通常允许的其他异常。

选择显式类型转换而不是隐式类型转换意味着开发人员将负责执行所有强制转换。虽然这种方法需要更多的代码，但它也提高了智能合约的安全性和可审计性。

前提条件和后置条件

Vyper 显式处理前置条件，后置条件和状态更改。虽然这会产生冗余代码，但它也允许最大程度的可读性和安全性。在使用 Vyper 中编写智能合约时，开发人员应注意以下三点：

- 条件

以太坊状态变量的当前状态/条件是什么？

- 效果

这个智能合约代码对执行时对状态变量的条件有什么影响？也就是说，什么会受到影响，什么不会受到影响？这些影响是否与智能合约的意图一致？

- 相互作用

在对前两个因素进行详尽考虑之后，是时候运行代码了。在部署之前，从逻辑上逐步遍历执行代码并考虑执行代码的所有可能的永久性结果和场景，包括与其他合同的交互。

理想情况下，应仔细考虑这些要点，然后在代码中进行完整的记录下来。这样做将改进代码的设计，最终使其更具可读性和可审计性。

修饰符

可以在每个函数的开头使用以下修饰符：

- @private：该@private 修饰使函数不能从合约外部访问。
- @public：该@public 修饰使功能可视和可执行的公开。例如，即使是以太坊钱包也会在查看合约时显示这些功能。

- @constant：@constant 不允许使用修饰器的函数更改状态变量。实际上，如果函数尝试更改状态变量，编译器将拒绝整个程序（带有适当的错误）。
- @payable：只有@payable 允许传递值。

Vyper 显式的实现了修饰符的逻辑。例如，如果函数同时具有@payable 修饰符和@constant 修饰符，则 Vyper 编译过程将失败。这是有道理的，因为传递值的函数按定义更新了状态，所以不能@constant。每个 Vyper 函数必须用@public 或@private（但不能同时使用）进行修饰。

函数和变量排序

每个单独的 Vyper 智能合约仅包含一个 Vyper 文件。换句话说，所有给定的 Vyper 智能合约代码，包括所有函数，变量等，都存在于一个地方。Vyper 要求每个智能合约的功能和变量声明都按特定的顺序进行写入。Solidity 完全没有这个要求。让我们快速看一下 Solidity 示例：

```
pragma solidity ^0.4.0;
contract ordering {
    function topFunction()
    external
    returns (bool) {
        initiatizedBelowTopFunction = this.lowerFunction();
        return initiatizedBelowTopFunction;
    }
    bool initiatizedBelowTopFunction;
    bool lowerFunctionVar;
    function lowerFunction()
    external
    returns (bool) {
        lowerFunctionVar = true;
        return lowerFunctionVar;
    }
}
```

在这个例子中，名为 topFunction 的函数正在调用另一个函数 lowerFunction。topFunction 还为名为 initiatizedBelowTopFunction 的变量赋值。如您所见，Solidity 不需要在执行代码调用之前声明这些函数和变量。这是成功编译的有效 Solidity 代码。

Vyper 的命令要求并不是一件新鲜事； 事实上，这些顺序要求一直存在于 Python 编程中。Vyper 所需的排序简单明了，如下一个示例所示：

```
# Declare a variable called theBool
theBool: public(bool)
# Declare a function called topFunction
@public
def topFunction() -> bool:
    # Assign a value to the already declared function called theBool
    self.theBool = True
    return self.theBool
# Declare a function called lowerFunction
@public
def lowerFunction():
    # Call the already declared function called topFunction
  assert self.topFunction()
```

这显示了 Vyper 智能合约中函数和变量的正确排序。注意变量 theBool 和函数 topFunction 如何分别在赋值和调用之前声明的。如果 theBool 在 topFunction 下面声明，或者如果 topFunction 在 lowerFunction 下面声明，则此合约将不会编译。

汇编

Vyper 有自己的在线代码编辑器和编译器，它允许您仅使用 Web 浏览器编写智能合约，然后将其编译为字节码，ABI 和 LLL。Vyper 在线编译器具有各种预先编写的智能合约，方便您使用，包括简单公开拍卖，安全的远程购买，ERC20 令牌等合约。

注意：Vyper 将 ERC20 作为预编译合约实施，从而是这些智能合约很易于使用。Vyper 中的合约必须声明为全局变量。声明 ERC20 变量的示例如下：

```
token: address(ERC20)
```

您还可以使用命令行编译合约。每个 Vyper 合约都保存在一个扩展名为.vy 的文件中。安装后，您可以通过运行以下命令与 Vyper 编译合约：

```
vyper  ~/hello_world.vy
```

然后，可以通过运行以下命令获取人类可读的 ABI 描述（采用 JSON 格式）：

```
vyper -f json ~/hello_world.v.py
```

在编译器级别防止溢出错误

在处理实际值时，软件中的溢出错误可能是灾难性的。例如，2018 年 4 月中旬的一项交易显示 恶意转移超过 57,596,044,618,658,100,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000 个 BEC 代币。此事件是 BeautyChain 的 ERC20 代币合同（BecToken.sol）中的整数溢出问题导致的。Solidity 开发人员可以访问像 SafeMath 这样的库以及像 Mythril OSS 这样的以太坊智能合约安全分析工具。但是，开发人员并没有使用安全工具。简而言之，如果语言没有强制执行安全性，开发人员可以编写不安全的代码，这些代码将成功编译并“成功”执行。

Vyper 具有内置溢出保护功能，通过“双管齐下”的方法实现。首先，Vyper 提供了一个 SafeMath 等价物，其中包括整数算术所必需的异常情况。其次，只要加载了文字常量，将值传递给函数或赋值变量，Vyper 就会使用 clamps。clamps 是通过低级别类 Lisp 语言（LLL）编译器中的自定义函数实现的，无法禁用。（Vyper 编译器输出 LLL 而不是 EVM 字节码;这简化了 Vyper 本身的开发。）

读写数据

虽然存储，读取和修改数据的成本很高，但这些存储操作是大多数智能合约的必要组成部分。智能合约可以将数据写入两个地方：

全局状态

给定智能合约中的状态变量存储在以太坊的全局状态 trie 中； 智能合约只能存储，读取和修改与特定合约地址相关的数据（即智能合约无法读取或写入其他智能合约）。

日志

智能合约还可以通过日志事件写入以太坊的链数据。虽然 Vyper 最初使用__log__语法来声明这些事件，但已经进行了更新，使其事件声明更符合 Solidity 的原始语法。例如，VyperMyLog 最初声明的事件声明 MyLog: __log__({arg1: indexed(bytes[3])})。语法现在已成为 MyLog: event({arg1: indexed(bytes[3])})。需要注意的是，在 Vyper 中执行日志事件的过程仍然如下：log.MyLog("123")。

虽然智能合约可以写入以太坊的链数据（通过日志事件），但他们无法读取他们创建的链上日志事件。尽管如此，通过日志事件写入以太坊的链数据的一个优点是，可以在公链上由轻客户端发现和读取日志。例如，挖掘块中的 logsBloom 值可以指示是否存在日志事件。一旦确定了日志事件的存在，就可以从给定的事务接收中获得日志数据。

结论

Vyper 是一种功能强大且有趣的新合约导向编程语言。它的设计偏向于“正确性”，但牺牲了一些灵活性。这将允许程序员编写更好的智能合约，并避免某些导致严重漏洞出现的陷阱。接下来，我们将更详细地介绍智能合约安全性。一旦了解了智能合约中可能出现的所有可能的安全问题，Vyper 设计的一些细微差别可能会变得更加明显。

第九章 智能合约的安全

智能合约的安全

在编写智能合约时，需要考虑的最重要因素之一就是安全性。在智能合约编程领域，发生错误（忽略安全性）会付出高昂的代价并且很容易被利用。在本章中，我们将介绍最佳的安全性实现和设计模式，以及“反安全性模式”，这些实现和模式可能会在智能合约中导致漏洞。

与其他程序一样，智能合约将完全执行所有写下的代码内容，这通常不一定是程序员希望代码执行的初衷。除此之外，所有智能合约都是公开的，任何用户都可以通过创建交易来与他们进行交互。这意味着任何漏洞都可以被利用，并且损失通常无法挽回。因此，遵循最佳实现并使用经过完善测试的设计模式至关重要。

安全最佳实现

防御性编程是一种特别适合智能合约的编程风格。它强调以下内容，所有这些都是最佳实现 ：

简约/简单

复杂性是安全的敌人。代码越简单，越少，发生错误或产生无法预料的后果的可能性就越小。第一次参与智能合约编程时，开发人员往往试图编写大量代码。最佳方法恰恰相反，您应该查看自己的智能合约代码，并尝试使用更少的代码，更短的代码行数，更低的复杂性以及更简单的“功能”。如果有人告诉你他们的项目为完成智能合约功能生成了“数千行代码”，那么你应该质疑该项目的安全性。越简单越安全。

代码重用

尽量不要重新发明轮子（编程中把他人已经做好的类库或者实现的功能，自己再重新做一遍，译者注）。如果已经存在满足您需要的大部分功能的库或智能合约，请重复使用它。在你自己的代码中，遵循 DRY 原则：不要重复自己。如果你看到任何代码片段重复多次，请问自己是否可以将其编写为函数或库并重复使用。已经广泛使用和测试的代码可能比你编写的任何新代码更安全。谨防“非我发明”综合症（即不愿意用外人发明的东西，译者注），虽然你可以通过从头开始构建它来“改进”某个功能或组件但是安全风险通常大于改进所带来的价值。

代码质量

智能合约代码出现错误是不可原谅的。每个 bug 都可能导致金钱损失。你不应该像普通编程那样对待智能合约编程。在 Solidity 中编写 DApps 与在 JavaScript 中创建 Web 小部件不同。相反，您应该使用严格的工程和软件开发方法，就像在航空航天工程或任何类似的无情原则中那样。一旦你“启动”你的代码，你几乎无法解决任何（错误）问题。

可读性/审计性

你所写的代码应该清晰易懂。阅读越容易，审计就越容易。智能合约是公开的，所以每个人都可以读取字节码，任何人都可以对其进行逆向工程。因此，使用协作和开源方法在公共场合进行开发是有益的，可以利用开发人员社区的集体智慧，并从开源开发的共同点中受益。你应该按照以太坊社区中已有的样式和命名约定编写记录良好且易于阅读的代码。

测试范围

测试你能做的一切。智能合约在公共执行环境中运行，任何人都可以使用他们想要的任何输入来执行它们。你永远不应该假设输入，例如函数参数的格式是规范的，有取值限制或没有恶意的。应测试所有参数以确保它们在预期范围内并正确格式化，然后才允许继续执行代码。

安全风险和反模式

作为一名智能合约编程者，你应该熟悉最常见的安全风险，以便能够检测并避免使合约暴露于这些风险的编程实现中。在接下来的几节中，我们将讨论一些例子来展示不同的安全风险，漏洞是如何产生的，以及可用于解决这些漏洞的对策或预防性解决方案。

重入

以太坊智能合约的一个特点是能够调用和利用其他外部合约的代码。合约通常也处理以太（币），因此经常将以太(币)发送到各种外部用户地址。这些操作要求合约提交外部呼叫。这些外部调用可能被攻击者劫持，攻击者可以强制合约执行更多代码（通过回滚功能），包括回调自身。在臭名昭着的 DAO 事件中，黑客中使用了这种攻击。

有关重入攻击的进一步阅读，请参阅 Gus Guimareas 关于该主题的博客文章和以太坊智能合约最佳实现。

漏洞

当合约将以太币发送到未知地址时，可能会发生此类攻击。攻击者可以在回滚函数中将包含恶意代码的外部地址写入合约。当合约将以太币发送到此地址时，它将调用恶意代码。通常，恶意代码在易受攻击的合约上执行功能，执行开发人员不期望的操作。“重入”一词来自于外部恶意合约调用，易受攻击的合约上的函数以及代码执行路径“重新运行”。

为了解释这一点，请思考 EtherStore.sol 中的简单易受攻击的合约，该合约作为以太坊金库，允许存款人每周仅提取 1 个以太币。

```
Example 1. EtherStore.sol
contract EtherStore {

    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

该合约有两个公共功能，depositFunds 和 withdrawFunds。 depositFunds 函数只是递增发送者的余额。 withdrawFunds 函数允许发送者指定要提取的 wei 数量。此功能仅在要求提取的金额少于 1 以太且上周未发生提款时才能成功。

漏洞位于第 17 行，其中合约向用户发送其请求的以太币（个数）。思考一个在 Attack.sol 中创建合约的攻击者。

Example 2. Attack.sol

```
import "EtherStore.sol";

contract Attack {
  EtherStore public etherStore;

  // intialize the etherStore variable with the contract address
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function attackEtherStore() public payable {
      // attack to the nearest ether
      require(msg.value >= 1 ether);
      // send eth to the depositFunds() function
      etherStore.depositFunds.value(1 ether)();
      // start the magic
      etherStore.withdrawFunds(1 ether);
  }

  function collectEther() public {
      msg.sender.transfer(this.balance);
  }

  // fallback function - where the magic happens
  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

漏洞是如何利用的？首先，攻击者会使用 EtherStore 的合约地址作为唯一的构造函数参数来创建恶意合约（假设在地址 0x0 ... 123 处）。这将初始化并将公共变量 etherStore 指向要受攻击的合约。

然后，攻击者将调用 attackEtherStore 函数，其中一些大于或等于 1 的以太假设暂时为 1。在这个例子中，我们还假设许多其他用户已将以太币存入此合约，以使其当前余额为 10 以太。然后会发生以下情况：

Attack.sol，第 15 行：EtherStore 合约的 depositFunds 函数将被调用，其中 msg 值为 1 以太（以及大量 gas）。发件人（msg.sender）将是恶意合约（地址）（0x0 ... 123）。因此，余额[0x0..123] = 1 以太。

Attack.sol，第 17 行：然后，恶意合约将使用 1 以太的参数调用 EtherStore 合约的 withdrawFunds 函数。这将通过所有要求（EtherStore 合约的第 12-16 行），因为之前没有提取

第 17 行：EtherStore.sol：合约将向恶意合约发送 1 个以太。

Attack.sol，第 25 行：对恶意合约的支付将执行回退功能。

Attack.sol，第 26 行：EtherStore 合约的总余额为 10 以太，现在为 9 以太，因此 if 语句通过。

Attack.sol，第 27 行：回退函数再次调用 EtherStore withdrawFunds 函数并“重新进入”EtherStore 合约。

EtherStore.sol，第 11 行：在第二次调用 withdrawFunds 时，攻击合约的余额仍为 1 以太，因为第 18 行尚未执行。因此，我们攻击地址仍然有余额[0x0..123] = 1 以太。lastWithdrawTime 变量也是如此。我们再次通过了所有条件。

第 17 行：EtherStore.sol：攻击合约提取另外 1 个以太币。

步骤 4-8 重复，直到不再是 EtherStore.balance > 1 的情况，如 Attack.sol 中的第 26 行所示。

Attack.sol，第 26 行：一旦 EtherStore 合约中剩下 1 个或更少以太，这个 if 语句将失败。然后，这将允许执行 EtherStore 合同的第 18 和 19 行（对于每次调用 withdrawFunds 函数）。

EtherStore.sol，第 18 行和第 19 行：将设置 balances 和 lastWithdrawTime 映射，执行将结束。

最终结果是攻击者在一次交易中从EtherStore 合同中提取了除了 1 太以外的所有以太币。

预防手段

有许多常用技术可以帮助避免智能合约中潜在的重入漏洞。第一种是尽可能在向外部合约发送以太币时使用内置传输功能。转移功能仅通过外部呼叫发送 2300gas，这不足以使目的地地址/合约调用另一个合约（即重新输入发送合约）。

第二种技术是确保所有改变状态变量的逻辑在以太网被发送出合约或任何外部调用之前发生。在 EtherStore 示例中，EtherStore.sol 的第 18 行和第 19 行应放在第 17 行之前。对于执行未知地址的外部调用的任何代码来说，最好用本地化函数或代码执行最后一个操作。这被称为检查效果 - 交互模式。

第三种技术是引入一个互斥体，即添加一个状态变量，在代码执行期间锁定契约，防止可重入调用。

应用全部这些技术（同时使用这三种技术是没有必要的，我们这样做是出于演示目的）到 EtherStore.sol，以下给出了无重入的合约：

```
contract EtherStore {

    // 初始化锁
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
       // 设置取钱限额
        require(_weiToWithdraw <= withdrawalLimit);
```

```
        // 限制取钱允许时间
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        // 在外部调用之前设定 reEntrancy 锁
        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        // 外部调用结束后释放锁
        reEntrancyMutex = false;
    }
}
```

真实案例：DAO

DAO（去中心化自治组织）攻击是以太坊早期发展中受到的主要攻击之一。当时，合约持有了超过 1.5 亿美元。 重入（Reentrancy）是这次攻击的主要原因，最终引发了创造以太坊经典（ETC）的硬分叉。 有关 DAO 事件的详细分析，请参考 http://bit.ly/2EQaLCI。 有关以太坊的分叉历史，DAO 事件的时间线以及关于 ETC 在硬分叉中的诞生等信息可以在[ethereum_standards]中找到。

算术上/下溢

以太坊虚拟机为整数指定了固定大小的数据类型。 这意味着一个整数变量只能表示一定范围内的数字。 例如，uint8 类型的变量只能存储[0,255]范围内的数字。尝试将 256 赋值给 uint8 类型变量将导致变量结果为 0. 不加以小心的话，如果没有对用户输入数据进行检查并在此基础上进行了计算，Solidity 中的变量可能会被滥用，导致数字超出存储它们的变量的数据类型范围。

有关算术上/下溢的更多信息，请参阅“如何保护智能合约”，以太坊智能合约最佳实践，以及 “Ethereum, Solidity and integer overflows: programming blockchains like 1970”（以太坊，Solidity 以及整数溢出：如 1970 年那样进行区块链编程）。

漏洞

当需要固定大小的变量来存储超出变量数据类型范围的数字（或数据）时，会发生上溢/下溢，。

例如，从值为 0 的 uint8 变量中（8 位无符号整数;即非负数）变量中减去 1，将会生成数字：255. 此时的情况为下溢。 我们分配了一个低于 uint8 范围的数字，因此结果会给出 uint8 可以存储的最大数字。 类似地，将 2^8 = 256 加到 uint8 类型的变量将使变量保持不变，因为我们已经绕过了 uint 的整个长度。这种行为的两个简单类比是汽车中的里程表，里程表能测量行驶的距离（它们在超过最大值，即 999999 之后会重置为 000000）以及周期性数学函数（在 sin 的参数上加 2π 将使值保持不变）。

添加大于数据类型范围的数字称为溢出。 比如说，将 257 添加到当前值为 0 的 uint8 变量将得到数字 1.将固定大小的变量视为循环变量有时是有益的，如果我们将数字添加到最大值以上，我们将从零开始存储数字，如果我们从零减去，则从最大数字开始向下计数。 对于可以表示负数的有符号 int 类型，一旦达到最大负值，我们就会重新开始计数; 例如，如果我们尝试从值为-128 的 uint8 变量中减去 1，我们将得到 127。

这些数字陷阱允许攻击者滥用代码并造成意外的逻辑行为。 例如 TimeLock.sol 中的 TimeLock 合约。

例子 3. TimeLock.sol

```
contract TimeLock {

    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        balances[msg.sender] = 0;
        msg.sender.transfer(balance);
    }
}
```

此合约被设计为一种类似时间保险库的机制：用户可以将以太存入合约，合约将会把以太锁定至少一周。用户可以选择是否将锁定时间延长超过 1 周，但是一旦将以太存入合约，用户可以确保他们的以太被至少安全的锁定了一周 ——如合约所希望的那样。

如果用户被迫交出他们的私钥，那么这样的合约可以用来确保他们的以太在短时间内无法被取出。但是，如果用户在此合约中锁定了 100 以太，并将其私钥交给了攻击者，则无论 lockTime 如何，攻击者都可以使用溢出来获取这些以太。

攻击者可以决定他们当前持有私钥的地址的 lockTime（因为它是一个公共变量）。我们称之为 userLockTime。然后他们可以调用 increaseLockTime 函数并将 2^256 作为参数传递进去 - userLockTime。此数字将被添加到当前 userLockTime 并且会导致上溢，从而将 lockTime [msg.sender]重置为 0.然后，攻击者可以简单地调用 withdraw 函数来获取奖励。

让我们看一下另一个例子（来自 Ethernaut 挑战赛的下溢漏洞演示），这个例子来自 Ethernaut 挑战赛。

剧透警告：如果您尚未完成 Ethernaut 挑战赛，下面的内容可以为其中一个关卡提供答案。

例 4.来自 Ethernaut 挑战赛的下溢漏洞演示

```
pragma solidity ^ 0.4.18;

contract Token {

  mapping(address => uint) balances;
  uint public totalSupply;

  function Token(uint _initialSupply) {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    return true;
  }
```

```
  function balanceOf(address _owner) public constant returns (uint balanc
e) {
    return balances[_owner];
  }
}
```

这是一个简单的可以调用 transfer 函数的代币合约，此合约允许参与者转移它的代币。你能发现合约中的错误吗？

问题来自与 transfer 函数。第 13 行的 require 语句可以使用下溢来绕过。考虑一位具有 0 余额的用户。他们可以用任何非零_value 调用 transfer 函数并在第 13 行略过 require 语句。这是因为 balances [msg.sender]是 0（数据类型为 uint256），所以从此数减去任何正数（不包括 2^256）结果为一个正数。对于第 14 行也是如此，其中余额将被记为正数。因此，在此示例中，攻击者可以用下溢漏洞获得免费代币。

预防手段

目前防止下溢/上溢漏洞的传统方法是使用或新建一个数学计算库来代替标准数学运算符的加法，减法和乘法（除法除外，因为它不会导致上溢/下溢，EVM 会在除以 0 时恢复）。

OpenZeppelin 在为以太坊社区构建及审核安全库方面做得非常出色。特别是它的 SafeMath 库可用于避免下/上溢漏洞。

为了演示如何在 Solidity 中使用这些库，让我们使用 SafeMath 库对 TimeLock 合约进行修改。此时没有溢出问题的合约是：

```
library SafeMath {

  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
```

```
    uint256 c = a / b;
    // assert(a == b * c + a % b); // This holds in all cases
    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}

contract TimeLock {
    using SafeMath for uint; // 使用 SafeMath 定义的 uint
    mapping(address => uint256) public balances;
    mapping(address => uint256) public lockTime;

    function deposit() public payable {
        balances[msg.sender] = balances[msg.sender].add(msg.value);
        lockTime[msg.sender] = now.add(1 weeks);
    }

    function increaseLockTime(uint256 _secondsToIncrease) public {
        lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncreas
e);
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        balances[msg.sender] = 0;
        msg.sender.transfer(balance);
    }
}
```

注意，所有的标准数学运算符都已替换为 SafeMath 库中定义的运算符。 TimeLock 合约将不能执行任何能够出现下/上溢的计算。

真实案例：PoWHC 及批传输上溢（CVE-2018-10299）

Proof of Weak Hands Coin（PoWHC，最初被当做一种玩笑设计出来，是由网络用户写的一种庞氏骗局。不幸的是，似乎合约的作者之前不了解上/下溢，因此 866 枚以太被从合约中取走。 Eric Banisadr 在他的博客文章中很好地概述了下溢是如何发生的（这与前面描述的 Ethernaut 挑战并不太相似）。

另一个例子是用 batchTransfe()函数实现一组 ERC20 代币合约。此合约包含上溢漏洞;你可以在 PeckShield 的帐户中阅读有关详细信息。

异常以太

通常，当发送以太到合约时，必须执行回退函数或合约中定义的其他函数。但是有两种例外情况，此时以太可以不执行任何代码就能存在合约中。依赖执行代码将以太发送给它们的合约当以太强制发送的时候很容易受到攻击。

更多内容，请参阅“如何保护您的智能合约”和“Solidity 安全模式 - 强制转移以太到合约”。

漏洞

在确保执行正确的状态转换或验证操作时一种常见并有用的防御性编程方法为不变性检查。 该技术涉及定义一组不变量（保持不变的尺度或参数），并检查在单个（或多个）操作之后参数是否保持不变。 只要检查的不变量实际上是真的不变量，那么通常来说这会是很好的设计。 不变量的一个例子固定发行的 ERC20 代币的 totalSupply（总量）。 因为没有函数可以修改此不变量,因此可以在传递函数中添加一个检查,以确保 totalSupply 保持不变,从而确保函数按预期工作。

有一种特殊情况，一种不变量可能看起来适合不变性检查，但实际上可以被外部用户操纵（无论智能合约的规则如何）。 这个变量就是存储在合约中的当前以太币。 通常，当开发人员首次学习 Solidity 时，他们会误解合约只能通过应付函数来接收或获得以太。 这种误解会导致合约对其中的以太量做出错误的假设，而这可能会导致一系列漏洞。 这个漏洞根源是（不正确）使用 this.balance。

在不使用应付函数或在合约上执行任何代码的情况下，有两种方式可以（强制）将以太发送到合约：

自毁/自杀

任何合约都能够实现 self-destruct 函数，该函数从合约地址中删除所有字节码，并将存储在那里的所有以太发送到指定的参数地址。 如果此指定的地址也是合约，则不会调用任何函数（包括回退）。 因此，无论合约中可能存在的任何代码，甚至合约没有支付功能，也可以使用 self-destruct 函数强制向任何合约发送以太。 这意味着任何攻击者都可以创建具有 self-destruct 函数的合约，向其发送以太，调用 self-destruct（目标）并强制发送以太到目标合约。 Martin Swende 发表了一篇优秀的博客文章，描述了 self-destruct 指令（Quirk＃2）的一些怪事，以及客户端节点如何检查不正确的不变量的报告，这可能导致以太坊网络的灾难性崩溃。

预发送的以太

另一种让以太加入合约的方法是用以太预先加载合约地址。合同地址是确定的 - 实际上，地址是根据创建合同的地址的 Keccak-256（通常与 SHA-3 同义）散列和创建合同的当前交易计算的。 具体来说，它的格式为
address = sha3（rlp.encode（[account_address，transaction_nonce]））（参见 Adrian Manning 关于“无钥匙以太”的讨论，了解一些有趣的使用案例）。 这意味着任何人都可以在创建合同之前计算合同的地址，并将以太网发送到该地址。 创建合同时，它将具有非零以太平衡。

让我们探讨一下根据这些知识可能出现的一些陷阱。考虑 EtherGame.sol 中过于简单的合约：

Example 5. EtherGame.sol

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;
```

```
    mapping(address => uint) redeemableEther;
    // Users pay 0.5 ether. At specific milestones, credit their account
s.
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game has finished
        require(currentBalance <= finalMileStone);
        // if at a milestone, credit the player's account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
}
```

这个合约代表一个简单的游戏（这自然会包括竞争条件），玩家将 0.5 以太发送到合约中，希望成为首先达到三个里程碑之一的玩家。 里程碑以以太计价。 达到里程碑的第一个玩家可以在游戏结束时领取一部分以太。 当达到最终里程碑（10 以太）时游戏结束；用户随后可以领取奖励。

EtherGame 合约的问题来自于第 14（以及通过关联 16）和 32 行的 this.balance 的恶劣使用。恶意攻击者可以通过 self-destruct 函数强行发送少量以太 - 比如 0.1 以太 - 前面讨论过）以防止任何未来的玩家达到里程碑。 由于这个 0.1 以太的作用，this.balance 永

远不会是 0.5 以太的倍数，因为所有合法的玩家只能发送 0.5 以太的增量。 这可以防止第 18,21 和 24 行上的所有 if 条件成立。

更糟糕的是，一个错过里程碑的报复性攻击者可以强制发送 10 以太（或将合同的平衡推到最终里程碑之上的同等数量以太，），将永远锁定合同中的所有奖励。这是因为，由于第 32 行的要求（即，因为 this.balance 大于 finalMileStone），ClaimReward 函数将始终恢复。

预防技术

这种漏洞通常来自滥用 this.balance。在可能的情况下，合约逻辑应该避免依赖于合同余额的确切值，因为它可以被人为操纵。如果基于 this.balance 应用逻辑，则必须应对意外的余额。

如果需要精确的以太存款，则应使用在支付函数中递增的自定义变量，以安全地跟踪一台存款。 此变量不受通过 self-destruct 调用发送的强制以太的影响。

考虑到这一点，EtherGame 合同的更正版本可能如下所示：

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;
    uint public depositedWei;

    mapping (address => uint) redeemableEther;

    function play() public payable {
        require(msg.value == 0.5 ether);
        uint currentBalance = depositedWei + msg.value;
        // ensure no players after the game has finished
        require(currentBalance <= finalMileStone);
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
```

```
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        depositedWei += msg.value;
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(depositedWei == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
}
```

在这里，我们创建了一个新的变量，depositEther，它跟踪已知的以太存款，这是我们用于测试的变量。 请注意，我们不再引用 this.balance。

其他案例

在暗箱 Solidity 编码竞赛（Underhanded Solidity Coding Contest）中给出了一些可利用合约的案例，它还提供了本节中提出的一些陷阱的扩展案例。

DELEGATECALL

CALL 和 DELEGATECALL 指令可以让以太坊开发人员模块化他们的代码。对合约的标准外部消息调用由 CALL 指令处理，其中代码在外部合约/函数的上下文中运行。DELEGATECALL 指令几乎相同，只是在目标地址执行的代码在调用协定的上下文中运行，并且 msg.sender 和 msg.value 保持不变。此功能支持库的实现，允许开发人员一次部署可重用代码，并从未来的合同中调用它。

虽然这两个指令之间的差异简单直观，但使用 DELEGATECALL 会导致意外的代码执行。

详细说明，请参阅 Loi.Luu 关于此主题的以太坊堆栈交换问题和 Solidity 文档。

漏洞

由于 DELEGATECALL 的上下文保护特性，构建无漏洞的自定义库并不像人们想象的那么容易。 库中的代码本身可以是安全且无漏洞的；但是，当在另一个应用程序的上下文中运行时，可能会出现新的漏洞。 让我们看一个相当复杂的使用斐波纳契数的案例。

考虑 FibonacciLib.sol 中的库，它可以生成 Fibonacci 序列和类似形式的序列。 （注意：此代码已从 https://bit.ly/2MReuii 修改。）

Example 6. FibonacciLib.sol

```
// library contract - calculates Fibonacci-like numbers
contract FibonacciLib {
    // initializing the standard Fibonacci sequence
    uint public start;
    uint public calculatedFibNumber;

    // modify the zeroth number in the sequence
    function setStart(uint _start) public {
        start = _start;
    }

    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n);
    }

    function fibonacci(uint n) internal returns (uint) {
        if (n == 0) return start;
        else if (n == 1) return start + 1;
        else return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

该库提供了一个可以在序列中生成第 n 个斐波纳契数的函数。 它允许用户更改序列（开始）的起始编号，并计算此新序列中的第 n 个斐波那契数。

现在让我们考虑使用这个库的合约，如 FibonacciBalance.sol 所示。

```
Example 7. FibonacciBalance.sol
contract FibonacciBalance {

    address public fibonacciLibrary;
    // the current Fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting Fibonacci sequence number
    uint public start = 3;
    uint public withdrawalCounter;
    // the Fibonancci function selector
    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));

    // constructor - loads the contract with ether
    constructor(address _fibonacciLibrary) public payable {
        fibonacciLibrary = _fibonacciLibrary;
    }

    function withdraw() {
        withdrawalCounter += 1;
        // calculate the Fibonacci number for the current withdrawal user
-
        // this sets calculatedFibNumber
        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounte
r));
        msg.sender.transfer(calculatedFibNumber * 1 ether);
    }

    // allow users to call Fibonacci library functions
    function() public {
        require(fibonacciLibrary.delegatecall(msg.data));
    }
}
```

该合约允许参与者从合约中撤回以太，其中以太等于对应于参与者提款订单的斐波纳契数；即，第一个参与者获得 1 以太，第二个也获得 1 以太，第三个获得 2 以太，第四个获得 3 以太，第五个获得 5 以太，依此类推（直到合同的余额小于提款斐波纳契数）。

本合约中有许多现象可能需要一些解释。首先，有一个看起来很有趣的变量 fibSig。这保存了字符串'setFibonacci（uint256）'的 Keccak-256（SHA-3）散列的前 4 个字节。这被

称为函数选择器，并被放入 calldata 以指定将调用智能合约的函数。它在第 21 行的 delegatecall 函数中用于指定我们希望运行 fibonacci（uint256）函数。delegatecall 中的第二个参数是我们传递给函数的参数。其次，我们假设在构造函数中正确引用了 FibonacciLib 库的地址（外部合约引用讨论了与此类合约引用初始化相关的一些潜在漏洞）。

你能发现这份合约中的任何错误吗？如果一个人要部署这个合同，发送以太，并拨打撤回，它可能会归还。你可能已经注意到状态变量 start 在库和主调用合约中都使用。在库合约中，start 用于指定斐波纳契序列的开头并设置为 0，而在调用合约中设置 3。您可能还注意到 FibonacciBalance 合约中的后退函数允许将所有调用传递给库合约，这允许调用库合约的 setStart 函数。回想一下，我们保留了合约的状态，看起来这个函数可能允许你改变本地 FibonnacciBalance 合约中的起始变量的状态。如果是这样，这将允许提取更多以太，因为得到的 calculatedFibNumber 依赖于起始变量（如库合约中所示）。实际上，setStart 函数不会（也不能）修改 FibonacciBalance 合约中的起始变量。此合同中的潜在漏洞明显比仅修改起始变量更糟糕。

在讨论实际问题之前，让我们先把另一个问题快速过一遍：了解状态变量如何储存在智能合约中。状态变量或者存储变量（在单个交易中保持不变的变量）按照它们被引入合约的顺序放入槽中。（这里有些复杂，请查阅 Solidity 文档以获得更深入的了解。）

举个例子，看我们来看看 library 合约。它有两个状态变量: start 和 calculatedFibNumber,第一个变量 start 被存储在合约的存储槽[0]中（即第一个槽），第二个变量 calculatedFibNumber 被置于下一个可用的存储槽[1]中。函数段 setStart 接受一个输入值并且将任意输入值设置为变量 start。因此，此函数段将存储槽[0]设置为在 setStart 函数段中提供的任意输入值。同样地，setFibonacci 函数段将 calculatedFibNumber 设置为 fibonacci(n)的结果，也就是将存储槽[1]设置为 fibonacci(n)的值。

现在让我们来看看 FibonacciBalance 合约。存储槽[0]现在与 fibonacciLibrary 地址对应，存储槽[1]对应着 calculatedFibNumber。正是在这种不正确的映射中，漏洞才会出现。Delegatecall 可以保留合约内容，这意味着通过 delegatecall 执行的代码将作用于调用的合约状态（即存储）。

现在注意，在第 21 行的 withdraw 中，我们执行 fibonacciLibrary.delegatecall(fibSig,withdrawalCounter)，这将会调用 setFibonacci 函数段，正如我们上述所探讨的，它修改了存储槽[1]，在我们当前的环境中，存储槽[1]是 calculatedFibNumber。这和预期的一致（即，在执行之后，calculatedFibNumber 被修改）。然而，召回的在 FibonacciLibrary 合约中的 start 变量位于存储槽[0]中,该存储槽就是当前合约中的 fibonacciLibrary 地址。这意味着 fibonacci 函数段将给出一个意料之外的结果，因为它引用了 start 变量（存储槽[1]），这个变量在目前的调用环境中是 fibonacciLibrary 地址（当被编译为 uint 的时候,

这个数值很大）。因此撤回函数段可能重置，因为它将不包含 uint 数量的以太,而这也是 calculatedFibNumberd 将要返还的以太数量。

更糟糕的情况是，FibonacciBalance 合约允许用户通过第 26 行的回退函数段调用所有的 fibonacciLibrary 函数段。这包括在前文所讨论到的 setStart 函数段，我们讨论过这个函数段允许任何人修改或者设置存储槽[0]。在这种情况下，存储槽[0]是 fibonacciLibrary 地址。因此，攻击者可以创建一些恶意的合约，将地址转换成 uint（这可以在 Python 中使用 int('<address>',16)轻易做到），然后调用 setStart(<attack_contract_address_as_uint>)。就就会将 fibonacciLibrary 更改为攻击合约的地址，接着每当使用撤回或者回退函数段的时候，这个恶意合约就会运行（这能够窃取合约的全部余额），因为我们已经修改了 fibonacciLibrary 的实际地址。这种攻击合约的一个案例是：

```
contract Attack {
    uint storageSlot0; // corresponds to fibonacciLibrary
    uint storageSlot1; // corresponds to calculatedFibNumber

    // fallback - this will run if a specified function is not found
    function() public {
        storageSlot1 = 0; // we set calculatedFibNumber to 0, so if withdr
aw
        // is called we don't send out any ether
        <attacker_address>.transfer(this.balance); // we take all the eth
er
    }
}
```

请注意，此类攻击合约通过更改存储槽[1]来修改 calculatedFibNumber。原则上，攻击者可以修改他们所选择的任意其他存储插槽，来执行对这个合约的各种形式的攻击。我们推荐您将这些合约放入 Remix 中，并通过这些 delegatecall 函数段尝试一下不同的攻击合约和状态更改。

同样地，我们也应该注意：当我们说 delegatecall 是状态保持的时候，我们不是在讨论合约的变量名称，而是那些名称指向的实际存储槽。从这个案例中可以看到，一个轻微的小错误就可能会导致攻击者控制整个合约和其中包含的以太。

预防手段

Solidity 提供用于执行库合约的库关键词（有关详细信息，请参阅文档）。以此确保库合约是无状态的而且不可自毁。强制库无状态化可以减轻在本章节中演示的存储环境的复杂

性。无状态库还可以防止攻击者通过直接修改库的状态来影响依赖于库代码的合约的攻击行为。按照惯例，在使用 DELEGATECALL 时，请注意库合约和调用合约的可能调用环境，并尽可能地构建无状态库。

真实案例：Parity 多重签名钱包（第二次黑客入侵）

第二次 Parity 多重签名钱包的黑客入侵就是一个真实案例，它展示了如果在预期之外的环境中运行，编写良好的库代码是如果被利用的。针对这种黑客行为，有很多很好的解释，如：“再次 Parity 多重签名入侵”和“对 Parity 多重签名漏洞的深入研究”。

为了增加这类参考资料，让我们来探究一下那些被利用的合约。大家可以在 GitHub 中找到这些库和钱包合约。

库合约如下：

```
contract WalletLibrary is WalletEvents {
  ...
  // throw unless the contract is not yet initialized.
  modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
  // constructor - just pass on the owner array to multiowned and
  // the limit to daylimit
  function initWallet(address[] _owners, uint _required, uint _daylimit)
      only_uninitialized {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }
  // kills the contract sending everything to `_to`.
  function kill(address _to) onlymanyowners(sha3(msg.data)) external {
    suicide(_to);
  }
  ...
}
And here’s the wallet contract:
```

接着这里是钱包合约：

```
contract Wallet is WalletEvents {
  ...
  // METHODS
```

```
  // gets called when no other function matches
  function() payable {
    // just being sent some cash?
    if (msg.value > 0)
      Deposit(msg.sender, msg.value);
    else if (msg.data.length > 0)
      _walletLibrary.delegatecall(msg.data);
  }
  ...
  // FIELDS
  address constant _walletLibrary =
    0xcafecafecafecafecafecafecafecafecafecafe;
}
```

请注意，钱包合约基本上通过委托调用将所有调用传递给钱包库合约。此代码段中的常量_walletLibrary 地址用作实际部署的 WalletLibrary 合约的占位符。（位于 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4 地址上。）

这些合约的预期操作是去拥有一个简单的低成本可部署钱包合约，其代码库和主要函数段在 WalletLibrary 合约中。可惜的是，WalletLibrary 本身就是个合约，而且它需要保持自身的状态。您能明白这为什么可能是一个问题吗？

这使得向 WalletLibrary 合约本身发送信息调用成为可能。具体地说，WalletLibrary 合约可以初始化并变为已拥有的状态。事实上，一个用户这么去做了，在 WalletLibrary 合约上调用 initWallet 函数段，并成为了库合约的所有者，而这个用户随后又调用了 kill 函数段。因为该用户是库合约的所有者，所以传递了修饰符并且库合约自毁。由于现有的所有钱包合约都引用了这个库合约，并且不包含更改此引用的方法，因此它们所有的功能，包括提取以太的功能都与 WalletLibrary 合约一起丢失。因此，这种类型的所有 Parity 多重签名钱包立即丢失并且永久不可恢复。

默认可见

Solidity 中的函数段具有可见性说明符，该说明符指示如何调用它们。可见性决定了一个函数段是否可以由用户从外部调用，或被其他派生合约调用，亦或者只能由内部调用还是只能由外部调用。有四种可见性说明符，它们在 Solidity 文档中有详细描述。函数段默认为公开状态，允许用户从外部调用它们。我们现在将关注的是：不正确地使用可见性说明符会如何导致智能合约中出现一些破坏性的漏洞。

漏洞

函数段的默认可见性是公开的，因此外部用户可以调用一些未指定其可见性的函数段。当开发人员错误地忽略了应该是私有的函数段（或者是只能在合约本身调用的函数段）的可见性标识符时，就会出现一些问题。

让我们来迅速地探究这样一个简单的案例：

```
contract HashForEther {
    function withdrawWinnings() {
        // Winner if the last 8 hex characters of the address are 0
        require(uint32(msg.sender) == 0);
        _sendWinnings();
     }
     function _sendWinnings() {
         msg.sender.transfer(this.balance);
     }
}
```

这个简单的合约被设计成一个猜测地址的赏金游戏。为了获得合约中的余额，用户必须创建一个最后 8 位十六进制字符为零的以太坊地址。一旦实现，他们就可以调用 withdrawWinnings 函数段来获得奖金。

遗憾的是，这个函数段的可见性还没有被指定。特别是，sendWinnings 函数段是公开的（默认公开），因此任何地址都可以通过调用该函数段来窃取奖金。

预防手段

最好提前指定合约中所有函数段的可见性，即使其中有些函数段是有意公开的。最新版本的 Solc 会对没有明确可见性设定的函数段提出警示信息，以鼓励这种做法。

真实案例：Parity 多重签名钱包（第一次黑客入侵）

在黑客对 Parity 多重签名的第一次侵入中，价值大约 3100 万美元的以太币被偷，大部分源自于 3 个钱包。哈塞布·库雷希（Haseeb Qureshi）对这次过程进行过详细的回顾。

基本上，多重签名钱包是由一个基本钱包合约构成的，它调用一个包含核心功能的库合约（如：真实的案例：Parity 多重签名钱包（第二次黑客入侵））。库合约包含能够初始化钱包的代码，如以下代码片段所示：

```
contract WalletLibrary is WalletEvents {
  ...
  // METHODS
  ...
  // constructor is given number of sigs required to do protected
  // "onlymanyowners" transactionsas well as the selection of addresses
  // capable of confirming them
  function initMultiowned(address[] _owners, uint _required) {
    m_numOwners = _owners.length + 1;
    m_owners[1] = uint(msg.sender);
    m_ownerIndex[uint(msg.sender)] = 1;
    for (uint i = 0; i < _owners.length; ++i)
    {
      m_owners[2 + i] = uint(_owners[i]);
      m_ownerIndex[uint(_owners[i])] = 2 + i;
    }
    m_required = _required;
  }
  ...
  // constructor - just pass on the owner array to multiowned and
  // the limit to daylimit
  function initWallet(address[] _owners, uint _required, uint _daylimit)
{
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }
}
```

注意	这两个函数段都没有指定它们的可见性，因此都默认为公开状态。initWallet 函数段在钱包的构造函数中被调用，并可以设定多重签名钱包的所有者，如在 initMultiowned 函数段中所示。由于这些函数段被意外地公开，攻击者能够在部署的合约上调用这些函数，从而将钱包所有权重置为攻击者的地址。作为钱包所有者，攻击者随后用尽了这些钱包中的所有以太余额。

熵错觉

以太坊上所有的交易都是确定状态的转移操作，可以确定的是，这意味着每一笔交易都以一种可计算的方式修正以太坊生态系统的全局状态，没有不确定性。即以太坊中没有熵或随机性的来源。实现去中心化的熵已经成为一个颇具规模的问题，已经有许多解决方案被提出，包括 RANDAO，或使用哈希链，正如 V 神在博客文章“验证器排序和 PoS 中的随机性”中描述的那样。

脆弱性

基于以太坊平台的首批合约中，有一些是基于赌博的。根本来说，赌博需要不确定性(可以下注的东西)，这使得在区块链(一个确定性的系统)上构建赌博系统相当困难。显然，不确定性必须来源区块链的外部资源。这对于玩家之间的投注是可能的(例如 commit-reveal 技术)；然而，如果您想要执行一个合同来扮演“房子”(比如 21 点或轮盘)，则要困难得多。一个常见的圈套是使用未来的块变量——也就是说，变量包含了关于尚未知道价值的交易区块的信息，比如 hashes（哈希值）， timestamps（时间戳）， block numbers（块编号）或 gas limit。这些问题在于，它们是由开采区块的矿工控制的，因此并不是真正随机的。例如，考虑一个有逻辑的轮盘赌智能合约，如果下一个区块哈希值（区块散列值）以偶数结尾，则返回一个黑色数字。一矿工（或矿池）可以在黑色下注$ 100 万。如果他们挖出一个块，并发现区块哈希值以奇数结尾，他们便可以不公布他们的块而去寻找下一个，直到他们找到了区块哈希值为偶数（假设区块的报酬和费用的解决方案都小于$ 1M）。而使用过去或现在块变量破坏性将会更大，正如 Martin Swende 在他的博客文章所展示的。此外，使用单独块变量意味着伪随机数将与块中的所有交易一样，因此攻击者可以通过一个块中执行多个交易(如果存在最大的赌注）增加他们胜率。

预防手段

熵源（随机性）必须来源于区块链外部。这可以在拥有诸如 commit-reveal 等系统的同行之间完成，也可以通过将信任模型更改为一组参与者（如 RandDAO。这也可以通过一个中心化的实体来充当随机性的 Oracle 来实现。块变量（一般情况下，也有一些例外）不应该被用来当熵源，因为他们可以被矿工操纵。

真实案例：PRNG 合约

在 2018 年 2 月 Arseny Reutov 在博客上发表了他对使用某种伪随机数生成器（prng）的 3649 个实时智能合约的分析；他发现了 43 个可以被利用的合约。

外部合约的调用

以太坊“计算机世界”的其中一个好处是能够重用代码，并与已经部署在网络上的合同进行交互。因此，大量的合同引用外部合约，通常通过外部消息调用。恶意行为者的意图可以隐藏在这些不起眼的外部消息调用之下，下面我们就来探讨这些方法。

漏洞

在 Solidity 中，任何地址都可以被当作合约，无论地址上的代码是否表示需要用到合约类型。这可能是骗人的，特别是当合约的作者试图隐藏恶意代码时。让我们以一个例子来说明这一点：

考虑像 Rot13Encryption.sol 一段代码，它初步地实现了 Rot13 密码。

Example 8. Rot13Encryption.sol

```
// encryption contract
contract Rot13Encryption {

   event Result(string convertedString);

    // rot13-encrypt a string
    function rot13Encrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            // inline assembly to modify the string
            assembly {
                // get the first byte
                char := byte(0,char)
                // if the character is in [n,z], i.e. wrapping
                if and(gt(char,0x6D), lt(char,0x7B))
                // subtract from the ASCII number 'a',
                // the difference between character <char> and 'z'
                { char:= sub(0x60, sub(0x7A,char)) }
                if iszero(eq(char, 0x20)) // ignore spaces
                // add 13 to char
```

```
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))}
            }
        }
        emit Result(text);
    }

    // rot13-decrypt a string
    function rot13Decrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61)) }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,13))}
            }
        }
        emit Result(text);
    }
}
```

得到一串字符（字母 a-z，没有验证）之后，上述代码通过将每个字符向右移动 13 个位置（围绕 'z'）来加密该字符串；即 'a' 转换为 'n'，'x' 转换为 'k'。这里的集合并不重要，所以不熟悉的读者可忽略它。

考虑以下使用此代码进行加密的合约：

```
import "Rot13Encryption.sol";

// encrypt your top-secret info
contract EncryptionContract {
    // library for encryption
    Rot13Encryption encryptionLibrary;

    // constructor - initialize the library
    constructor(Rot13Encryption _encryptionLibrary) {
        encryptionLibrary = _encryptionLibrary;
```

```
    }

    function encryptPrivateData(string privateInfo) {
        // potentially do some operations here
        encryptionLibrary.rot13Encrypt(privateInfo);
     }
}
```

该合约的问题是，~~encryption Library~~加密库地址不是公开或不变的。因此，合约的配置人员可以在指向该合约的构造函数中给出一个地址：

```
// encryption contract
contract Rot26Encryption {

   event Result(string convertedString);

    // rot13-encrypt a string
    function rot13Encrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            // inline assembly to modify the string
            assembly {
                // get the first byte
                char := byte(0,char)
                // if the character is in [n,z], i.e. wrapping
                if and(gt(char,0x6D), lt(char,0x7B))
                // subtract from the ASCII number 'a',
                // the difference between character <char> and 'z'
                { char:= sub(0x60, sub(0x7A,char)) }
                // ignore spaces
                if iszero(eq(char, 0x20))
                // add 26 to char!
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,26))}
            }
        }
        emit Result(text);
    }

    // rot13-decrypt a string
```

```
    function rot13Decrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61)) }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,26))}
            }
        }
        emit Result(text);
    }
}
```

这个契约实现了 ROT26 密码，它将每个字符移动 26 位（即，什么都不做）。同样，不需要理解本合同中的组件。更简单地说，攻击者能够链接以下合同达到相同效果：

```
contract Print{
    event Print(string text);

    function rot13Encrypt(string text) public {
        emit Print(text);
    }
}
```

如果在构造函数中提供了这些合约的地址，那么 encryptprivatedata 函数只会生成一个事件，即打印未加密的私有数据。

尽管在本例中，在构造函数中设置了类似库的合约，但通常情况下，特权用户（如所有者）可以更改库合约地址。如果链接的合约不包含被调用的函数，则回退函数将被执行。例如，在 encryptionLibrary.rot13Encrypt（）中，如果指定的合约 encryptionLibrary 是;

```
contract Blank {
     event Print(string text);
     function () {
         emit Print("Here");
         // put malicious code here and it will run
     }
```

}

那么会发出一个带有“Here”文字的事件。因此，如果用户可以更改合约库，原则上可以让用户在不知不觉中运行任意代码。

注意	这里的合同仅用于说明的目的，而不代表正确的加密。同样他们也不应被用于加密。

预防手段

正如前面所证明的，安全合约可以（在某些情况下），以这样的方式恶意部署。审计人员可以公开验证合约并让其所有者以恶意方式进行部署，从而产生具有漏洞或恶意的公开审计合约。

有许多的防止这些情况的技术。

一种方法是使用 new 关键字来创建合约。在前面的例子中，构造函数可以被写为：

```
constructor() {
    encryptionLibrary = new Rot13Encryption();
}
```

这样，引用合约的一个实例就会在部署时创建，并且部署者无法在不修改智能合约的情况下用其他任何东西替换 `Rot13Encryption` 合约。另一个解决方案是硬编码外部合约地址。

通常，调用外部合约的代码应该始终被仔细审查。作为开发人员，在定义外部合约时，最好将合约地址公开(在下一节的 honey-pot 示例中不是这样)，以允许用户方便地检查合约引用的代码。相反，如果一个合约有一个私有可变的合约地址，那么它可能意味着某人存在恶意行为。(如实际示例所示)。如果特权（或任何）用户能够更改用于调用外部函数的合约地址，（在去中心化系统的情境中）实现时间锁定或投票机制就变得很重要，为要允许用户查看哪些代码正在改变，或让参与者有机会选择加入/退出新的合约地址。

真实案例:可重入蜜罐

最近主网上出现了一些“蜜罐合约”，这些合约试图打败那些想要利用合约漏洞的黑客，让他们反过来在想要利用的合约中损失 Ether。一个例子是通过在构造函数中用恶意合约代替期望的合约来发动上述攻击。代码可以在这里找到：

```
pragma solidity ^0.4.19;

contract Private_Bank
{
    mapping (address => uint) public balances;
    uint public MinDeposit = 1 ether;
    Log TransferLog;

    function Private_Bank(address _log)
    {
        TransferLog = Log(_log);
    }

    function Deposit()
    public
    payable
    {
        if(msg.value >= MinDeposit)
        {
            balances[msg.sender]+=msg.value;
            TransferLog.AddMessage(msg.sender,msg.value,"Deposit");
        }
    }

    function CashOut(uint _am)
    {
        if(_am<=balances[msg.sender])
        {
            if(msg.sender.call.value(_am)())
            {
                balances[msg.sender]-=_am;
                TransferLog.AddMessage(msg.sender,_am,"CashOut");
            }
        }
    }

    function() public payable{}
```

```
}

contract Log
{
    struct Message
    {
        address Sender;
        string  Data;
        uint Val;
        uint  Time;
    }

    Message[] public History;
    Message LastMsg;

    function AddMessage(address _adr,uint _val,string _data)
    public
    {
        LastMsg.Sender = _adr;
        LastMsg.Time = now;
        LastMsg.Val = _val;
        LastMsg.Data = _data;
        History.push(LastMsg);
    }
}
```

reddit 的一位用户在这篇文章中解释了他们是如何试图利用合约中存在的可重入漏洞，从而在合同中丢失了一个以太。

短地址/参数攻击

这个攻击不是在 Solidity 合约本身上执行，而是可能在与它们交互的第三方应用程序上执行。 添加此部分是为了 Solidity 合约的完整性，并让读者知道在合约中如何操作参数。

有关进一步阅读，请参阅“ERC20 短地址攻击解释”，“ICO 智能合约漏洞：短地址攻击”或这份 Reddit 帖子。

漏洞

将参数传递给智能合约时，参数根据 ABI 规范进行编码。 发送短于预期参数长度的编码参数是可以的（例如，发送仅为 38 个十六进制字符（19 个字节）的地址，而不是标准的 40 个十六进制字符（20 个字节））。 在这种情况下，EVM 将在编码参数的末尾添加零以符合预期长度。

当第三方应用程序未验证输入时，这就会成为一个问题。最明显的例子是用户请求提款时不会验证该 ERC20 通证地址的交换。 Peter Vessenes 的帖子“ERC20 短地址攻击解释”中详细介绍了这个例子。

需要考虑标准的 ERC20 传递函数接口，注意参数的顺序：

```
function transfer(address to, uint tokens) public returns (bool succes
s);
```

现在设想一个持有大量通证的交易所（比如通证 REP）和一个想提取 100 个 REP 的用户（该用户在交易所的 REP 中占有一定份额）。

该用户需提交他们的地址：0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead，以及想提取的数量：100。

交易所将按照传递函数指定的顺序对这些参数进行编码；也就是说，先是通证的地址，再是通证。

编码的结果将是：

```
a9059cbb000000000000000000000000deaddeaddea \
ddeaddeaddeadeaddeaddeaddead0000000000000
0000000000000000000000000000000056bc75e2d63100000
```

```
The first 4 bytes (a9059cbb) are the transfer function signature/selector,
 the next 32 bytes are the address, and the final 32 bytes represent the u
int256 number of tokens. Notice that the hex 56bc75e2d63100000 at the end
corresponds to 100 tokens (with 18 decimal places, as specified by the REP
 token contract).
```

前 4 个字节（a9059cbb）是传递函数签名/选择器，接下来的 32 个字节是地址，最后的 32 个字节代表通证的 uint256 数。 请注意，末尾的十六进制 56bc75e2d63100000 对应于 100 个通证（具有 18 个小数位，由 REP 通证合约指定）。

现在我们看看如果发送一个缺少 1 个字节（2 个十六进制数位）的地址会发生什么。 具体来说，假设攻击者发送 0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde 作为地址（缺少最后两个数位）以及输入相同的想提取的通证数：100。
如果交易所不验证此输入，它将被编码为：

```
a9059cbb000000000000000000000000deaddeaddea \
ddeaddeaddeaddeaddeaddeadde00000000000000
0000000000000000000000000000000056bc75e2d6310000000
```

这个差异不易察觉。请注意，00 已添加到编码结尾，以弥补发送的短地址。 当它被发送到智能合约时，地址参数将被读取为 0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde00，以及提取的值将被读取为 56bc75e2d6310000000（注意两个额外的 0）。于是提取的数现在为 25600 个通证（该值已乘以 256）。 在该示例中，如果交易所持有这么多（REP）通证，则用户将提取 25600 个通证（当交易所认为用户仅提取 100 时）到修改的地址。显然，攻击者在这个例子中不会拥有修改后的地址，但是如果攻击者要生成任何以 0 结尾的地址（可以很容易地强制使用）并使用这个生成的地址，他们就可以从毫无疑问地从交易所中偷取通证。

预防技术

所有外部应用程序中的输入参数都应在发送到区块链之前进行验证。 还应注意，在这里起着重要作用的参数排序。 由于填充只发生在最后，因此在智能合约中谨慎地排序参数可以减轻某些形式的这种攻击。

未检查的调用返回值

在 Solidity 中有多种方法可以执行外部调用。通常通过 transfer 方法将以太发送到外部帐户。当然，也可以使用 send 功能，对于更通用的外部调用， CALL 操作码可以直接用于 Solidity。

call 和 send 函数将返回一个布尔值以指示调用是成功还是失败。因此，这些函数有一个简单的警告，即如果外部调用（通过 call 或 send 进行初始化）失败，那么执行这些函数的交易将不会恢复;相反，函数将返回 false。
常见错误是开发人员期望在外部调用失败时发生恢复，并且不检查返回值。

如需进一步阅读，请参阅 DASP 2018 年 Top10 的“＃4”和“未检查发送’错误扫描实时以太坊合约”。

漏洞

考虑下面的例子:

```
contract Lotto {

    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    // ... extra functionality here

    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount);
        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);
        msg.sender.send(this.balance);
    }
}
```

这代表了一种类似于乐透彩票的合约，中奖者将获得一定数额的以太，也通常会留下少量以太留给任何人提取。

该漏洞存在于第 11 行，在第 11 行中使用 send 而不检查响应。在这个简单的示例中，如果交易失败(要么是耗尽了手续费，要么是合约故意加入了回退函数)，则不管是否发送了以太，都允许将 payedOut 设置为 true。

在这种情况下，任何人都可以通过 withdraw leftover 函数来提取中奖者的奖金。

预防手段

只要可能，使用 transfer 函数而不是 send，因为外部交易一旦恢复，transfer 就恢复。如果被要求使用 send 函数，请始终检查返回值。

一个更有力的建议是采用撤退模式。

在这个解决方案中，每个用户必须调用一个隔离的提取函数，该函数处理合约外的以太发送，并处理交易发送失败的后果。其思想是逻辑上将外部发送功能与代码库的其余部分隔离开来，并将潜在交易失败的负担放在调用 withdraw 函数的最终用户身上。

真实案例：以太网和以太之王

Etherpot（以太网）是一个聪明的合同彩票，与前面示例提到的合同并没有太大的不同。该合同的失败主要是由于不正确地使用了块散列(只有最后 256 个块散列可用;参见阿基尔•费尔南德斯(Aakil Fernandes)关于以太网未能正确考虑这点的文章)。

除此之外，该合约还存在未检查的调用值。需考虑 lotto.sol: Code snippet 中的现金功能：

示例 9. 洛托.索尔:代码片段

```
...
  function cash(uint roundIndex, uint subpotIndex){

        var subpotsCount = getSubpotsCount(roundIndex);

        if(subpotIndex>=subpotsCount)
            return;

        var decisionBlockNumber = getDecisionBlockNumber(roundIndex,sub
potIndex);

        if(decisionBlockNumber>block.number)
            return;

        if(rounds[roundIndex].isCashed[subpotIndex])
            return;
        //Subpots can only be cashed once. This is to prevent double pa
youts

        var winner = calculateWinner(roundIndex,subpotIndex);
        var subpot = getSubpot(roundIndex);

        winner.send(subpot);
```

```
            rounds[roundIndex].isCashed[subpotIndex] = true;
            //Mark the round as cashed
    }
    ...
```

注意	在第 21 行，send 函数的返回值没有被选中，接下来的行设置了一个布尔值，指示中奖者已经将他们的资金发送了

这个 bug 允许中奖者没有收到以太，但是合约的状态却表明中奖者已经被支付了。（即已经收到以太）

更严重的版本发生在以太之王（King of the Ether）。
一份针对该合约的极好的事后分析，详细说明了如何用一个未检查的失败发送去攻击该合约。

竞态条件/前运行

外部调用其他合约和底层区块链的多用户特性，这两者的结合，导致了各种潜在的可靠性的缺陷，用户竞相执行代码以获得意外状态。

可重入性(本章前面讨论过)就是这种竞争条件的一个例子。

在本节中，我们将讨论在以太坊区块链上可能发生的其他竞态条件。关于这个主题有很多好文章，包括 Ethereum Wiki 上的“竞赛条件”、2018 年 DASP Top10 上的第 7 条以及 Ethereum Smart Contract Best Practices。

漏洞

与大多数区块链一样，Ethereum 节点汇集事务并将其形成区块。
只有当矿工解决了协商一致机制(目前 Ethereum 的 Ethash PoW)后，这些事务才被认为是有效的。

处理区块的矿工还将选择池中的哪些事务将包含在该块中，通常根据每个事务给予的手续费高低进行排序。

这是一个潜在的攻击向量。攻击者可以监视事务池，寻找可能包含问题解决方案的事务，并修改或撤销解决程序的权限，或故意更改合约中的状态。然后攻击者可以从这个事务中获

取数据，并创建一个自己的具有更高手续费的事务，这样他们的事务就包含在原始事务之前的一个块中。

让我们用一个简单的例子来看看这是如何工作的。

考虑一个在 FindThisHash.sol 中显示的合约。

Example 10. FindThisHash.sol
示例 10.FindThisHash.sol

```
contract FindThisHash {
    bytes32 constant public hash =
      0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    constructor() public payable {} // load with ether

    function solve(string solution) public {
        // If you can find the pre-image of the hash, receive 1000 ether
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

假设这个合约包含 1000 以太。用户可以找到以下 SHA-3 散列的原图像:

```
0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a
```

可以提交解决方案并检索 1000 以太。

假设一个用户找到了解决方案我们叫它「Ethereum!」。他们以「Ethereum!」作为参数调用 solve。不幸的是，攻击者非常聪明，能够监视提交解决方案的任何人的事务池。他们看到了这个解决方案，检查了它的有效性，然后提交了一个等效的事务，但他的手续费比原来的事务要高得多。

处理该区块的矿工可能会因为较高的手续费而给予攻击者优先权，并在原始解决者之前打包处理他们的事务。

攻击者将拿走 1000 以太，而解决问题的用户将一无所获。请记住，在这种“领先”的脆弱性中，矿工有独特的动机会自己发动攻击(或者可以贿赂矿工，让他们以高昂的费用发动这些攻击)。攻击者本身是矿工的可能性不应被低估。

预防手段

有两类参与者可以执行这种预先运行的攻击:用户(修改他们的交易的手续费)和矿工(他们可以按照自己认为合适的方式在一个块中重新排序交易)。

易受第一类攻击（用户）的合约要比易受第二类攻击（矿工）的合约糟糕得多，因为矿工只有在解出一个区块时才能执行攻击，这对于任何针对特定块的矿工来说都不太可能。在这里，我们将列出一些与这两类攻击者相关的抑制措施。

一种方法是在 gasPrice 上放置一个上限。这可以避免用户通过增加 gasPrice 来获得超过上限的优先交易顺序。此措施仅防范第一类攻击者（任意用户）。在这种情况下，矿工仍然可以攻击合约，因为无论 gas price 是多少，他们都可以在他们的区块内发送交易。

更健康的方法是使用提交-显示方案。这种方案规定用户使用隐藏信息（通常是哈希）发送交易。在交易已包含在区块中之后，用户发送显示数据已发送的交易（显示阶段）。此方法可防止矿工和用户进行抢先交易，因为他们不能确定交易的内容。然而，此方法无法隐藏交易值（在某些情况下，这是需要隐藏的有价值信息）。ENS 智能合约允许用户发送包含他们愿意花费多少以太坊数量数据的交易。然后，用户可以发送任意值的交易。在显示阶段，用户会被退还交易中发送的金额与他们愿意花费的金额之间的差额。

Lorenz Breidenbach，Phil Daian，Ari Juels 和 FlorianTramèr 提出的改进建议是使用“潜艇发射”（submarine sends）。这个想法的有效实现需要 CREATE2 操作码，目前尚未被采用，但似乎可能会在未来的硬分叉中出现。

真实案例：ERC20 和 Bancor

ERC20 标准以在以太坊上构建代币而闻名。由于 approve 函数，此标准具有潜在的抢先交易漏洞。Mikhail Vladimirov 和 Dmitry Khovratovich 已经对这个漏洞做了很好的解释(以及减轻攻击的方法）。

Approve 函数的标准定义为：

```
function approve(address _spender, uint256 _value) returns (bool success)
```

这个函数允许一个用户许可其他用户以他们的名义转移代币。抢先交易漏洞发生情况如下：用户 Alice 批准她的朋友 Bob 花费 100 个代币。Alice 后来决定撤销 Bob 的 100 个代币的权限，因此她创建了一个交易，并将 Bob 的上限设置为 50 个代币。一直在关注以太链的 Bob 看到了这笔交易并创建了一笔自己花费 100 代币的交易。他给自己的交易提供了比 Alice 更高的 gasPrice，因此他的交易优先于她的交易。Approve 函数的实现将允许 Bob 转移他的

100 个代币，然后，当 Alice 的交易被提交时，将 Bob 的接入权限重置为 50 个代币，实际上允许 Bob 有权使用总共 150 个代币。

另一个突出的现实世界的例子是 Bancor。Ivan Bogatyy 和他的团队记录了对最初 Bancor 实施的有利可图的攻击。他的博客文章和 DevCon3 讲话详细讨论了这是如何完成的。本质上来说，代币的价格是根据交易价值确定的；用户可以观察 Bancor 交易的交易池，并抢先运行它们以从价格差异中获利。Bancor 团队现在已经解决了这一攻击。

拒绝服务攻击（DoS）

此类别非常广泛，但概括来说就是使用户可能在一段时间内或在某些情况下永久性地无法使用合约的攻击。这可以永久地获得这些合约中的以太坊，就像现实世界中的例子一样，比如 Parity 多签名钱包（第二次黑客攻击）。

漏洞

有许多方法让合约无法运作。在这里，我们只重点介绍一些可能会导致 DoS 漏洞的不太引人注意的 Solidity 编码模式：

通过外部操作的映射或数组循环

这种模式通常出现在所有者希望利用分发类的函数，向投资者分发代币时，如此示例合约中所示：

```
contract DistributeTokens {
    address public owner; // 所有者的代币地址
    address[] investors; // 投资者的地址数组
    uint[] investorTokens; // 每个投资者获得的代币数量

    // …… 一些额外功能，包括发送代币 transfertoken()

    function invest() public payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 发送 5 倍的 wei
        }

    function distribute() public {
        require(msg.sender == owner); // 仅所有者
        for(uint i = 0; i < investors.length; i++) {
```

```
            //  transferToken(to,amount) 发送代币数量
            // 代币打到投资者地址
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

请注意，此合约中的循环遍历数组数量可能极为夸张。攻击者可以创建许多用户帐户，使投资者地址数组巨大。大体上，这样就可以使执行 for 循环所需的 gas 超过区块 gas 上限，基本上使分发函数无法使用。

所有者操作

另一种常见模式是所有者在合约中具有特定权限，并且必须执行某些任务才能使合约进入下一个状态。一个例子是初始货币发行（ICO）合约，要求所有者最终确定合约，然后才允许代币可流转。例如：

```
bool public isFinalized = false;
address public owner; // 所有者的代币地址

function finalize() public {
    require(msg.sender == owner);
    isFinalized == true;
}

// …… 额外的 ICO 功能

// 重载的传输（transfer）函数
function transfer(address _to, uint _value) returns (bool) {
    require(isFinalized);
    super.transfer(_to,_value)
}

……
```

在这种情况下，如果特权用户丢失私钥或变为非活动状态，则整个代币合约将变得无法使用。在这样的情况下，如果所有者无法调用 finalize 函数，则代币都无法被流转；代币生态系统的整个操作都取决于单个地址。

基于外部调用的进程状态

合约有时会被写为发送以太坊到一个地址以进入到新状态，或者是等待来自外部源的某些输入。当外部调用失败或由于外部原因而被阻止时，这些模式可能导致 DoS 攻击。在发送以太坊的示例中，用户可以创建一个不接受以太坊的合约。假如合约需要取出以太坊以便进入新状态（比如一个时间锁合约，要求在再次使用之前取出所有以太坊），合约将永远不会达到新状态，因为以太坊不可能发送给不接受以太坊的用户合约上。

预防手段

在第一个示例中，合约不应循环通过可由外部用户人为操纵的数据结构。建议采用转出（withdrawal）模式，即每个投资者都调用转出函数来独立的认领代币。

在第二个示例中，特权用户需要更改合约的状态。在这样的示例中，可以在所有者变得丧失能力或资格的情况下使用失效保护。一种解决方案是使所有者成为多重签名的合约。另一种解决方案是使用时间锁：在上文给定的示例中，第 13 行上的 require 函数可以包括基于时间的机制，例如 require（msg.sender == owner || now> unlockTime），允许任何用户经过指定的一段时间 unlockTime 之后 finalize。这种纾缓技术也可用于第三个例子。如果需要外部调用以进入新状态，请考虑其可能会失败，并在可能永远不会发生所需调用的情况下添加基于时间的状态进展。

当然，也有一些中心化的替代建议：比如添加一个维护用户（maintenanceUser），如有需要，他可以一直关注跟进并解决基于 DoS 的攻击向量问题。通常来说由于这种实体力量的介入，这种类型的合约具有信任问题。

真实案例：GovernMental

GovernMental 是一个古老的庞氏骗局，它收获了大量的以太坊（一度达到了 1100 个以太坊）。不幸的是，它很容易受到本节中提到的 DoS 漏洞的影响。 Etherik 在 Reddit 论坛网站上的帖子描述了合约需要删除大型映射以便取出以太坊。删除该映射的 gas 成本超过了当时的区块 gas 限制，因此不可能取出 1,100 以太坊。该合约地址为 0xF45717552f12Ef7cb65e95476F217Ea008167Ae3，你能从交易记录 0x0d80d67202bd9cb6773df8dd2020e719 0a1b0793e8ec4fc105257e8128f0506b 中看到 1100 个以太坊最终通过 2.5M 的 gas 获得（当区块的 gas 上限提升到足够允许这样的交易时）。

区块时间戳操作

区块时间戳历来被用于各种应用场景中，例如随机数的熵（有关更多详细信息，请参见 Entropy Illusion），锁定一段时间的资金，以及各种与时间相关的状态变化条件语句。矿工有能力略微调整时间戳，而如果在智能合约中又不当地使用区块时间戳，可能会导致较大的风险。

相关的参考资料包括 Solidity 文档和 Joris Bontje 关于以太坊堆栈交换问题的相关论题。

漏洞

如果矿工有动机，block.timestamp 及其别名现在可以被他们进行操纵。让我们构建一个简单的游戏，如 roulette.sol 所示，它很容易被矿工利用。

案例 11：roulette.sol
Example 11. roulette.sol

```
contract Roulette {
    uint public pastBlockTime; // 强制每个区块只能下注一次

    constructor() public payable {} // 最初的资金合约

    // 用于下注的 fallback 函数
    function () public payable {
        require(msg.value == 10 ether); // 必须发送 10 个以太坊进行游戏
        require(now != pastBlockTime); // 每个区块只有 1 次交易
        pastBlockTime = now;
        if(now % 15 == 0) { // 获胜者
            msg.sender.transfer(this.balance);
        }
    }
}
```

这份合约就像一个简单的彩票。每个区块仅有一次交易可以下注 10 个以太坊，每次下注有机会赢得合约中的资金。这里的假设是 block.timestamp 的最后两位数是均匀分布的。按照这样来假设，那将有 15%的几率赢得这个彩票。

但是，正如我们所知，矿工可以根据需要调整时间戳。在这种特殊情况下，如果合约中有足够多的以太坊，则发现区块的矿工有动机选择一个时间戳，使其 block.timestamp 或现在 15%的几率为 0。这样做矿工可以随着区块奖励一同赢得锁定在此合约中的以太坊。由于每

个区块只允许一个人下注，因此这也容易受到抢先交易攻击（有关详细信息，请参阅 Race Conditions/Front Running）。

现实中，区块时间戳是自动增加的，所以矿工不能随意选择区块时间戳（他们必须比前一个处理者晚一个时间戳）。同样的，他们的区块时间也不被允许提前，不然区块就会被网络拒绝（如果那个区块的时间戳提前了，节点就会失效）

预防手段

区块链时间戳不会被用于熵或生成随机数字 - 比如，他们可以是游戏中取得胜利或改变关键状态的决定性因素（不管是直接的还是通过其他派生方式）

指定一个区块号

时间敏感逻辑有一些必须条件，比如：解锁限制（时间上锁算法），在几周后完成一个ICO，或者某一强制结束时间点，一周约等于 60480 个区块。因此，指定区块号在一个指定的区间里会更安全，这样矿工就不可以轻易操纵区块号。BAT ICO 规则中制定了这个规范。

在矿工操作时间戳的时候并不是必须要求有的，但是在开发约束的时候必须要注意到。

真实案例：GovernMental

GoverMental 是上面提过的旧的 Ponzi 算法，也是最容易受攻击的基于时间戳的算法.这个约束只支付给同一时间最后一个加入的用户（最少一分钟）。因此，矿工可以调整时间戳到未来的某个时间使之看起来在那一分钟已经被运行了（尽管可能在现实中并不是真实运行的）。关于这里更多的细节可以在 Tanya Bahrvnovska 编著的《以太网安全漏洞黑客攻击及修复历史》一书中找到。

保护构造函数

在创建合约时，构造函数时一个非常特殊的至关重要的方法，有特殊的任务需要执行。在可靠版本 v0.4.22 之前，构造函数被定义为和合约名相同并包括的方法。在这种例子里，开发过程中合约名被改变的话，如果构造函数的名字没有一起改变就会成为一个普通的可调用的方法。所以你可以想象，这回引起非常有趣的合约攻击。

为了进一步了解，读者可以尝试一下 Ethernaut 挑战（在特定的影响层）

漏洞

如果构造名被修改了，或者构造名被输入错误，使它和合约名不同，构造函数就会表现的和一般方法一样。这会导致非常严重的后果，特别是如果构造函数执行了特权操作。参考下面的这个合约：

```
contract OwnerWallet {
    address public owner;

    // constructor
    function ownerWallet(address _owner) public {
        owner = _owner;
    }

    // Fallback. Collect ether.
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

这个合约收集 ether 并且只有持有者通过调用这个方法可以撤回。这个问题出现的原因是构造函数和合约名并不是一模一样的：首字母不同！因此任何用户都可以调用 ownerWallet 方法，使自己成为钱包主人，这样就可以通过撤回拿到合约中所有的 ether。

预防手段

这个主题已经在 0.4.22 的稳定版编译器中实现了。在这个版本中引入了一个可以指定构造函数的构造函数关键字而不是要求构造函数必须和合约名相同。通过在构造函数中使用这个关键字的方法可以有效避免命名问题。

真实案例：Rubixi

Rubixi 是暴漏机制弱点的另一个极端例子。原本被叫做 DynamicPyrmid，但是合约名在部署到 Rubixi 之前被改变了。构造函数的名字并没有被改变，允许其他用户成为构造者。关于这个 bug 的一些有趣的决策可以在 Bitcointalk 这本书中找到。根本上，它允许用户争夺

构造者状态来声明极端机制的费用。关于这个特定 bug 的更多些姐可以在《以太网安全漏洞黑客攻击及修复历史》一书中找到。

未初始化的存储指针

EVM 在固态存贮空间或者内存中存储数据。在合约开发中准确理解执行方式的以及方法的本地变量的默认类型书被强烈推荐的。这是因为它在不正常的初始化变量时可能会产生易受攻击的合约

想阅读更多关于 EVM 存储和内存的信息，可以看数据位置的可靠性文档，存储层变量状态和内存层。

备注：

这个章节是基于 Sefan Beyer 的一个精彩发表。想了解这个话题的更多知识，受 Setfan 的影响，可以在这个 Reddit 线程中找到。

弱点：

方法内的本地变量默认根据他们本身的类型存在存储层和内存中。味精初始化的本地存储变量可能会包含合约中其他存储变量的值。这个事实可以引起为初始化风险，或者被恶意使用。

让我们看一下 NameRegistrar.sol 中的相对简单的名字注册合约。

Example 12. NameRegistrar.sol

```
// A locked name registrar
contract NameRegistrar {

    bool public unlocked = false;  // registrar locked, no name updates

    struct NameRecord { // map hashes to addresses
        bytes32 name;
        address mappedAddress;
    }

    // records who registered names
    mapping(address => NameRecord) public registeredNameRecord;
    // resolves hashes to addresses
    mapping(bytes32 => address) public resolve;
```

```
    function register(bytes32 _name, address _mappedAddress) public {
        // set up the new NameRecord
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;

        resolve[_name] = _mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;

        require(unlocked); // only allow registrations if contract is unl
ocked
    }
}
```

这个简单的名字注册系统只有一个方法。当合约被解锁时，它允许任何人注册一个名字（用 32 位）然后把这个名字分配给一个地址。这个注册系统时默认被锁的，并且在第 25 行需要从添加名字记录中保护注册系统。看起来合约并不能被使用，因为没有方法可以解锁注册处，但是有一个弱点可以无视被锁变量而进行名字注册。

想要探讨这个弱点，首先我们需要理解固态存贮的存储机制。作为一个高级概述（没有任何合适的技术方法 - 我们推荐阅读可靠性文档），状态参数被顺序存储在合约（他们可以一起出现，但不是在这个例子里，所以我们不用担心这个问题）。因此，非锁存在在 slot[0],registeredNameRecord in slot[1],并且在 slot[2]中被解决，等等。这些槽每个长度时 32 位（在映射的时候增加了复杂性，但是我们从现在开始忽略不计）。布尔解锁时 0x000…0（64 0s，包括 0x）表示假或者 0x000…1 (63 0s)表示真。正如你看到的那样，在特定例子下，有一个很大的存储资源浪费。

因为默认存储，这个谜题的下一部分是默认情况下稳定性放入复杂数据类型，比如结构体，在存储空间中，初始化它们为本地函数。因此，在第 18 行默认存入存储空间新纪录。newRecord 没有被初始化的事实会引起弱点。因为默认存储，被映射到存储 slot[0]，这里当前包含一个没有上锁的指针。注意在第 19 和 20 行中，我们设置 ewRecord.name 为 _name，同时设置 newRecord.mappedAddress 为_mappedAddress；这样就更新了存储地址为 slot[0]和 slot[1]，它们都修改为解锁状态并且存储槽和 registeredNameRecord 相关联。

0x0000000000000000000000000000000000000000000000000000000000000001

这就意味着 unlocked 的可以简单的通过注册方法的 32 位 _name 参数可以被直接修改。因此如果_name 到最后一位是非零的，它会修改存储空间 slot[0]的最后一位并直接修改 unlocked 为 true。这样_name 值回引起回掉第 25 行成功，这样我们就设置 unlocked 为真。在

Remix 中尝试这个，注意如果你使用_name 的格式为 0x0000000000000000000000000000000000000000000000000000000000000001 的话方法将会通过。

预防手段

固态编译器展示了未初始化存储变量的警告；开发者在构建智能合约时需要非常小心的注意这些警告。当前版本的 Mist (0.10) 并不允许这些合约被编译。当处理复杂类型时通常这是一个好的练习来明确的使用内存和存储说明符，来保证系统能按预期执行。

现实例子：OpenAddressLottery 和 CryptoRoulette Honey Pots。一个名为 OpenAddressLottery 的 honey pot 被部署用于非初始化存储变量 quirk 用于收集从可能的攻击者那里收集 ether。合约相比被包含，我们会把分析留给攻击非常容易解释的 Reddit 线程。

另一个 honey pot：CryptoRoulette，同样被这个花招所利用来试图收集一些 ether。如果你不能发现这些攻击是怎么进行的，可以参考《一些以太网 Honeypot 攻击分析》来了解合约。

浮点和精度

就像这里所写的一样（v0.4.24），固态不支持定点和浮点睡着。这意味着浮点型在固态存储中鼻血被勾践为整数类型。如果没有别正确实现，就会导致出现错误和弱点

备注：

想要了解更多信息，可以看以太网合约安全技术和和技巧百科

弱点：

在完整性中没有定点类型，开发者需要实现自己的标准整数类型。有很多漏洞开发者可以在过程中执行进去。我们在这一章节需要高亮这些。
让我们用一段代码例子开始（我们会忽略超出/下溢问题，我们回在这一章节先简易的讨论一下这个问题，）

```
contract FunWithNumbers {
    uint constant public tokensPerEth = 10;
    uint constant public weiPerEth = 1e18;
    mapping(address => uint) public balances;

    function buyTokens() public payable {
        // convert wei to eth, then multiply by token rate
```

```
        uint tokens = msg.value/weiPerEth*tokensPerEth;
        balances[msg.sender] += tokens;
    }

    function sellTokens(uint tokens) public {
        require(balances[msg.sender] >= tokens);
        uint eth = tokens/tokensPerEth;
        balances[msg.sender] -= tokens;
        msg.sender.transfer(eth*weiPerEth);
    }
}
```

这个简单的 token 买卖合约有一些明显的问题。虽然购买和出售 tokens 的数学计算是正确的，但缺少浮点数会得到错误的结果。例如，当在第 8 行购买 tokens 时，如果该值小于 1 ether 初始除法将导致 0，则将最终乘法的结果保留为 0（例如，200 wei 除以 1e18 weiPerEth 等于 0）。同样，在出售 tokens 时，任何数量的、token 小于 10 都会产生 0 ether。事实上，这里的四舍五入总是下降的，所以卖出 29 tokens 会得到 2 ether。

这个合约的问题在于精度只取近似的以太值（即 1e18 wei）。 在处理 ERC20 tokens 中的小数时，而你需要更高的精度，这会变得很麻烦 。

预防手段

保持智能合约的正确精确度非常重要，尤其是在处理反映经济决策的比例和利率时。

你应该确保你使用的任何比例或比率都允许分数中有较大的分子，例如，我们 tokensPerEth 在示例中使用了比例。使用 weiPerTokens 会更好，这是一个很大的数字。 要计算相应的令牌数量，我们可以使用 msg.sender/ weipertoken。这样可以得到更精确的结果。

要记住的另一个策略是注意操作的顺序。在我们的示例中，购买 token 的计算是 msg.value/weiPerEth*tokenPerEth。请注意，除法发生在乘法之前。（与某些语言不同，Solidity 保证按照它们的写入顺序去执行操作。）如果计算时首先执行乘法再执行除法，则此示例将会有更高的精度； 即，msg.value*tokenPerEth/weiPerEth。

最后，在为数字定义任意精度时，最好将值转换为更高的精度，执行所有数学运算，然后最终转换回输出所需的精度。通常使用 uint256s（因为它们对于 gas 使用是最佳的）；在它们的范围内给出了大约 60 个数量级，其中一些数量级可以用于精确的数学运算可能的情况是，最好将所有变量保持在 Solidity 中的高精度并转换回外部应用程序中较低的精度（这实际上是 decimals 变量在 ERC20 token 合约中的工作方式）。要查看如何执行此操作的示

例，我们建议您查看 DS-Math。它使用了一些时髦的命名（“wads”和“rays”），但这个概念很有用。

真实案例：Ethstick

该 Ethstick 合约不使用扩展精度； 然而，它处理的是 wei。因此，这个合约将有四舍五入的问题，但只有在精确的 wei 水平会出现。它有一些更严重的缺陷， 但这些都与在区块链上获得熵的难度有关（参见 Entropy Illusion ）。有关 Ethstick 合约的进一步讨论，我们将向您推荐 Peter Vessenes 写的另一篇文章， ″以太坊合约将成为黑客的糖果″。

Tx.Origin 验证

Solidity 有一个全局变量，tx.origin 它遍历整个调用堆栈并包含最初发送调用（或事务）的帐户的地址。在智能合约中使用此变量进行身份验证会使合约容易受到类似网络钓鱼的攻击。

注意	有关进一步阅读，请参阅 dbryson 的以太坊堆交换问题， Peter Vessenes 的“Tx.Origin 和 Ethereum Oh My!”以及 Chris Coverdale 的“ Solidity : Tx Origin Attacks”。

漏洞

授权用户使用 tx.origin 变量的合约通常容易受到网络钓鱼攻击，这些攻击会诱使用户对易受攻击的合约执行经过身份验证的操作。

考虑 Phishable.sol 中的简单契约。

示例 13.Phishable.sol

```
contract Phishable {
address public owner;
constructor (address _owner) {
owner = _owner;
}
function () public payable {} // collect ether
function withdrawAll(address _recipient) public {
require(tx.origin == owner);
_recipient.transfer(this.balance);
```

```
}
}
```

请注意，在第 11 行，合约授权 withdrawAll 使用该功能 tx.origin。此合约允许攻击者创建以下形式的攻击合约：

```
import "Phishable.sol";
contract AttackContract {
Phishable phishableContract;
address attacker; // The attacker's address to receive funds
constructor (Phishable _phishableContract, address _attackerAddress) {
phishableContract = _phishableContract;
attacker = _attackerAddress;
}
function () payable {
phishableContract.withdrawAll(attacker);
}
}
```

攻击者可能将此合约伪装成他们自己的私人地址，并将受害者（ Phishable 合约的所有者）进行社会工程设计，以便将某种形式的交易发送到该地址 - 也许可以向该合约发送一定数量的以太币。除非小心，否则受害者可能不会注意到攻击者的地址上有代码，或者攻击者可能将其作为多重签名钱包或某些高级存储钱包传递（请记住默认情况下公共合约的源代码是不可用的） 。

在任何情况下，如果受害人将有足够的 gas 交易的 AttackContract 地址，它会调用回退功能，这反过来又调用 withdrawAll 的功能 Phishable 与参数合约 attacker。这将导致所有资金从 Phishable 合约中撤回到该 attacker 地址。这是因为首次初始化呼叫的地址是受害者（即 Phishable 合约的所有者）。因此，tx.origin 将等于 owner 和合约的 require 第 11 Phishable 行将通过。

预防手段

tx.origin 不应该用于智能合约的授权。这并不是说永远不应该使用 tx.origin 变量。它在智能合约中确实有一些合法的用例。例如，如果想要拒绝外部合约调用当前合约，则可以实现 require 该表单的一个 require(tx.origin == msg.sender)。这可以防止使用中间合约来调用当前合约，从而将合约限制为常规无代码地址。

合约库

有许多现有代码可供重用，它们都作为可调用库部署在链上，而作为代码模板库部署在链外。 平台上的库已被部署，它们作为字节码智能合约存在，因此在生产中使用它们之前应该非常小心。但是，使用成熟的现有平台库具有许多优势，例如能够从最新升级中受益，并通过减少以太坊中的实时合约总数来节省资金并使以太坊生态系统受益。

在以太坊，最广泛使用的资源是 OpenZeppelin 套装，一个丰富的合约库，从 ERC20 和 ERC721 tokens 的实现到多种包模型，在合约中常见的简单的行为，如 Ownable，Pausable 或 LimitBalance。此存储库中的合约已经过广泛测试，在某些情况下甚至可以作为事实上的标准实现。它们是免费使用的，由 Zeppelin 和不断增长的外部贡献者一起构建和维护。

来自 Zeppelin 的还有 ZeppelinOS，这是一个开源的服务和工具平台，可以安全地开发和管理智能合约应用程序。ZeppelinOS 在 EVM 之上提供了一个层，使开发人员可以轻松启动可升级的 DApp，这些 DApp 链接到经过良好测试的合约组成的链上库，这些合约本身可以升级。这些库的不同版本可以在以太坊平台上共存，并且审核系统允许用户提出或推进不同方向的改进。该平台还提供了一组用于调试，测试，部署和监视分布式应用程序的脱链工具。

项目 ethpm 旨在通过提供包管理系统来组织生态系统中正在开发的各种资源。因此，他们的注册表提供了更多示例供你浏览：

- 网站：https：//www.ethpm.com/
- 仓库链接：https：//www.ethpm.com/registry
- GitHub 链接：https：//github.com/ethpm
- 文档：https：//www.ethpm.com/docs/integration-guide

结论

这里对于任何从事智能契约领域的开发人员来说，都有很多东西需要了解和理解。通过在你的智能合约设计和代码编写方面遵循最佳的实践，你将避免许多严重的误区和陷阱。

也许最基本的软件安全原则是最大化可信代码的重用。在密码学中，这一点非常重要，它被浓缩成一句格言：“不要自己写加密包。” 在智能合约的情况下，这相当于从社区彻底审查的库获得尽可能多的收益。

第十章 令牌

什么是令牌

“令牌”一词来源于古英语(“tacen”)，意思是标志或符号。通常用来表示私人发行且价值无关紧要的类似货币的东西，如运输令牌，洗衣令牌，拱廊令牌。

近年来，随着区块链技术的兴起，基于区块链的令牌被赋予了新的含义。即令牌被抽象为区块链上有明确所有权属性的实物，这一实物可以代表资产、货币或者访问权限。

“令牌”与其微不足道的价值之间的关系取决于与物理令牌受限的使用场景。由于时常受限于特定的商业活动、组织机构或者地理位置，物理令牌的接受程度较差，且仅能用于单一用途。但对于区块链令牌而言，这些限制性障碍被扫除了。许多区块链令牌都具有多种用途，可以在全球作为交易媒介或者在全球流动性市场兑换其他法定货币。

这一节我们将了解令牌的各种用途和起源。我们也将讨论令牌的属性，比如可替代性和内在机制，等等。最后，我们了解令牌所基于的标准和技术并动手发布自己的令牌。

令牌有什么用

令牌最明显的用途是数字私人货币。然而，这仅仅只是其中一项用途。令牌在被编程后能够提供许多不同的功能，它们通常是重叠的。例如，令牌可以同时传递投票权，访问权和资源所有权。 货币只是它的第一例“应用”。

货币

令牌可以作为一种货币形式，其价值通过私人交易来确定。比如以太坊或比特币。

资源

令牌可以表示在共享经济或资源共享环境中获得或产生的资源。例如，存储类或 CPU 类令牌代表的资源可通过网络共享。

资产

令牌可以代表内在或外在，有形或无形资产的所有权。例如，黄金，房地产，汽车，石油，能源等。

访问

一个令牌可以代表访问权限，甚至可以访问数字或实体资产，例如论坛，专属网站，酒店房间，租车。

公平
令牌可以代表数字组织（如 DAO）或合法机构（如公司）的股东权益。

投票
令牌可以代表数字或合法系统中的投票权。

可收藏品
令牌可以代表数字（例如 CryptoPunks）或现实收藏品（例如绘画）。

身份
令牌可以表示数字（例如化身）或合法身份（例如国家 ID）。

公证
令牌可以代表某个权威机构或分布式信誉系统的认证或事实证明。

通常，单一令牌具有这些重叠功能中的多项功能。在某些情况下很难区分它们，因为物理等价物总是有着千丝万缕的联系。例如，在现实世界中，驾驶执照（证明）也是一种身份证件（身份证），两者不能分开。在数字领域，以前的混合功能可以区分开来并单独开发的（例如匿名证明）。

令牌和可替代性

维基百科解释：

在经济学中，可替代性是物品或商品的一种属性，其个体单位本质上是可以互换的。
当两种令牌价值或功能没有任何差异时，我们可以用一种令牌单位代替另一种代，此时我们可以说令牌是可替换的。例如，以太是一个可替代的令牌，因为以太的任何单位具有与任何其他以太单位相同的价值和用途。

严格地说，如果令牌的历史踪迹是可追溯的，那么它就不是完全可替代的。追踪踪迹的能力可能导致黑名单和白名单，从而降低或消除可替代性。我们将[隐私]中进一步研究。

不可替代的令牌指代表独特的有形或无形商品的令牌，因此不可互换。例如，代表一副特定的梵高画作的令牌与代表毕加索画作的另一个令牌不能互相替换。同样，表示特定数字收藏品的令牌，如一只唯一的加密猫（请参阅[cryptoKitties]）就与任何其他的加密猫是不可互换的。

接下来我们将看到可替换和不可替换令牌的例子。

交易对手风险

交易对手风险是交易中的其他方不能履行其义务的风险。由于在交易中增加了两个以上的交易方，某些类型的交易会产生额外的交易对手风险。例如，如果您持有贵金属存款证并将其出售给某人，则该交易中至少有三方：卖方，买方和贵金属的保管人。某人持有有形资产，必要时他们成为涉及该资产交易的一方，并增加交易对手风险。当任何资产通过交换拥有所有权的令牌实现间接交易时，资产托管人会承担额外的交易对手风险。他们是否真的拥有资产？他们是否会根据令牌的转让（例如证书，契据，标题或数字令牌）来认可（或允许）所有权转让。在数字令牌的世界中，了解谁持有由令牌表示的资产以及适用于该基础资产的规则很重要。

令牌及其内在机制

“内在”一词源于拉丁语“内部”，意思是“从内部”。

一些令牌代表区块链上的数字资产。这些数字资产受制于共识规则，就像令牌本身一样。这具有重要意义：代表内在资产的令牌不会带来额外的交易对手风险。如果你拥有 1 个以太币的私钥，那么就没有其他人拥有这个以太币的所有权。区块链共识规则生效，您对私钥的所有权（控制权）等同于资产的所有权且无需任何中介。

相反，许多令牌被用来代表区块链以外的现实世界中的资产，如房地产，公司投票权股份，商标，金条，等等。这些资产的所有权并不在区块链上，而是受法律，道德和政策的约束，这与形成令牌的共识规则是分离的。因此，这些外部资产会带来额外的交易对手风险，因为它们由托管人持有，记录在外部注册管理机构中，或由区块链环境以外的法律和政策保障。

基于区块链的令牌的最重要的后果之一是能够将外部资产替换为内部资产，从而消除交易对手风险。 一个很好的例子就是将一家公司的股权（外部）转换为一个分布式自治组织或类似的（内部）组织的股权或投票权。

令牌应用：实用性或权益性

今天几乎所有的以太坊项目都以某种令牌的形式出现。但是，所有这些项目真的就一定需要一个令牌吗？使用令牌到底有什么缺点，或者我们应该考虑“将所有事物代币化”的口号是否成熟？

首先，我们阐述令牌在新项目中的作用。大多数项目中令牌主要发挥一下两方面作用：“实用令牌”或者“股权令牌”。很多时候，这两个角色是混合在一起的，难以区分。

应用令牌指被用来支付服务费用，应用程序或资源的令牌。应用令牌的例子包括代表资源的令牌，如共享存储，或者访问社交媒体网络等服务，或将自己作为以太坊平台的 GAS。 相比之下，股权令牌指代表创业公司股票的令牌。

股权令牌与股票和利润分配的无投票权股份一样是数量有限的，也可以像分布式自治组织中的投票股一样广泛，平台的管理是通过令牌持有者的大多数投票来决定的。

It’s a Duck!

仅仅因为令牌用于为初创公司筹款，并不意味着它必须用作服务的支付，反之亦然。 然而，许多创业公司面临一个棘手的问题：令牌是一个很好的筹资机制，但向大众提供证券（股票）在大多数国家和地区都是受监管的活动。通过将股权令牌伪装成应用令牌，许多创业公司希望能够绕过这些监管限制，并从预售应用令牌中公开募股筹集资金。这些被变相伪装的股权产品是否能够摆脱监管机构仍有待观察。

正如谚语所说：“如果它像鸭子一样走路，像鸭子一样嘎嘎叫 - 它就是一只鸭子。” 监管机构不会因这些语义游戏而转移注意力，恰恰相反，他们更有可能将这种法律诡辩看作是企图欺骗公众。

应用令牌：谁需要它们？

然而，真正的问题在于，应用型令牌为创业公司带来了重大风险和采用障碍。 也许在不久的将来，“所有事物令牌化”将成为现实。但是，目前来看，能够掌握，理解，并愿意使用令牌的人数仅仅只是规模已经非常小的加密货币市场中的一小部分人。

对于创业公司而言，每项创新都代表风险和市场过滤器。创新就是探索前人没走过的道路，走出与传统迥异的道路。它注定是一段孤独的旅程。如果一家初创公司试图在一个新的技术领域进行创新，比如 P2P 网络上的存储共享，那么这是一条相当孤单的道路。 为该创新添加效用令牌并要求用户采用令牌以使用该服务会增加风险并增加采用的障碍。 它走出了已经孤独的 P2P 存储创新之路，进入荒野。

把每一项创新想象成一个过滤器。它限制了可以成为这种创新的早期采用者的市场子集的采用。 添加第二个过滤器化合物，进一步限制了可寻址市场。您在询问您的早期采用者采用的不是一种，而是两种全新的技术：您所创建的新型应用程序/平台/服务以及令牌经济。

对于创业公司而言，每项创新都会带来风险，增加创业失败的机会。如果您采取已经冒险的创业想法并添加实用标记，则会增加底层平台（以太坊），更广泛的经济（交易所，流

动性），监管环境（股票/商品监管机构）和技术（智能合约，令牌标准）。 这对创业公司来说是一个很大的风险。

“所有事物令牌化”的倡导者可能会通过采用令牌来抵制这种情况，它们也继承了整个令牌经济的市场热情，早期采用者，技术，创新和流动性。这也确实如此。问题在于利益和热情是否超过风险和不确定性。

最后，在本章开始介绍令牌，我们说令牌的口语意义即“具有无关紧要价值的事物”。大多数令牌的价值微不足道的根本原因是它们只能用在非常狭窄的场景中：一家巴士公司，一家洗衣店，一家商场，一家酒店，一家公司商店。有限的流动性，有限的实用性和高转化成本将会降低令牌的价值，直到它只是一种“标记”价值。因此，当您将应用型令牌添加到您的平台上，但该令牌只能在您自己的一个平台上使用且市场很小时，则会重新出现使现实世界法币变得毫无价值的情况。如果为了使用您的平台，用户必须将某些东西转换为您的应用型令牌，以使用它，然后将余数转换回更常用的东西，您已创建公司代码。数字令牌的转换成本比没有市场的物理令牌低几个数量级。但转换成本不是零。在整个行业部门工作的应用令牌将非常有趣，可能相当有价值。但是，如果您将创业公司设置为必须引导整个行业标准才能获得成功，那么您可能已经失败了。

为了正确的理由做出这个决定。因为你的应用程序无法使用令牌_（例如以太坊），所以采用令牌。采用它是因为令牌解决了基本的市场障碍或访问问题。不要引入实用标记，因为它是您快速筹集资金的唯一方式，您需要假装它不是公共证券。

Token 标准

区块链 token 在以太坊之前就已存在。在某些方面，第一种区块链货币比特币本身就是一种代币。在以太坊之前，许多令牌平台也是在比特币和其他加密货币上开发的。然而，在以太坊上引入第一个 token 标准导致令牌爆发。

Vitalik Buterin 建议将 token 作为广义可编程区块链以太坊最明显和最有用的应用之一。事实上，在以太坊的第一年，人们常常看到 Vitalik 和其他人穿着背面印有醒目的以太坊标志的 T 恤和智能合约示例。这有几种不同的 T 恤，但最常见的是展示 token 的实现。

ERC20 令牌标准

第一个标准由 Fabian Vogelsteller 于 2015 年 11 月引入，作为以太坊通用征求意见协议（ERC）。它被自动分配给 GitHub 第 20 号 issue，产生名称“ERC20 token”。绝大多数 token 目前都基于 ERC20。ERC20 的征求意见最终成为了以太坊改进提案 EIP20，但大部分仍然被称为 ERC20。你可以在这里阅读标准：

https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

ERC20 是可替代 tokens 的标准，意味着 ERC20token 的不同单元是可互换的并且没有独特的属性。

ERC20 标准为实现 token 的合约定义了一个公共接口，以便可以以相同的方式访问和使用任何兼容的 token。该接口包含许多必须存在于每个标准的实现的函数中，以及可由开发人员添加的一些可选函数和属性。

ERC20 需要的函数和事件

totalSupply

返回当前存在的此 token 的总单位。ERC20 令牌可以有固定或可变的供应。

balanceOf

给定一个地址，返回该地址的 token 余额。

Transfer

给定地址和金额，将该 token 数量从执行发送的地址的余额发送到该地址。

transferFrom

给定发件人，收件人和金额，将 token 从一个帐户转移到另一个帐户。与下面的批准结合使用。

Approve

给定收件人地址和金额，授权该地址从发出批准的帐户执行多次，直到到达该金额的转帐。

Allowance

给定所有者地址和消费者地址，返回消费者被允许从所有者中提取的剩余金额。

Transfer event

成功支付后触发事件（调用 transfer 或 transferFrom）（即使是零值转移）。

Approval event

成功调用 approve 后记录事件。

ERC20 可选函数

Name

返回 token 的可读名称（例如“US Dollars”）。

Symbol

返回 token 的可读符号（例如“USD”）。

Decimals

返回用于划分 token 金额的小数位数。例如，如果小数为 2，则将 token 金额除以 100 以获取其用户表示。

ERC20 接口在 Solidity 中定义

这是 ERC20 接口规范在 Solidity 中的展示：

```
contract ERC20 {
   function totalSupply() constant returns (uint theTotalSupply);
   function balanceOf(address _owner) constant returns (uint balance);
   function transfer(address _to, uint _value) returns (bool success);
   function transferFrom(address _from, address _to, uint _value) returns
(bool success);
   function approve(address _spender, uint _value) returns (bool success);
   function allowance(address _owner, address _spender) constant returns
(uint remaining);
   event Transfer(address indexed _from, address indexed _to, uint _value);
   event Approval(address indexed _owner, address indexed _spender, uint
_value);
```

```
}
```

ERC20 数据结构

如果您检查任何 ERC20 实现，它将包含两个数据结构，一个用于跟踪余额，另一个用于跟踪 allowance。在 Solidity 中，它们通过数据映射实现。

第一个数据映射由所有者实现 token 余额的内部表。这允许 token 合约跟踪 token 的拥有者。每次转账都是一个余额减少，另一个余额增加。

余额：从地址（所有者）到金额（余额）的映射

```
mapping(address => uint256) balances;
```

第二个数据结构是配额的数据映射。正如我们将在 ERC20 workflows: "transfer" and "approve & transferFrom",使用 ERC20 token，token 的所有者可以将权限委托给消费者，允许他们从所有者的余额中支出特定金额（allowance）。ERC20 合同通过二维映射跟踪 allowance，主键是 token 所有者的地址，映射到消费者地址和 allowance 金额：

Allowances：从地址（所有者）到地址（消费者）到金额（allowance）的映射

```
mapping（address => mapping（address => uint256））public allowed;
```

ERC20 工作流程："transfer" and "approve & transferFrom"

ERC20token 标准具有两种转账功能。你可能想知道为什么？
ERC20 允许两种不同的工作流程。第一种是使用 transfer 函数的单事务，直接的工作流程。此工作流程是钱包用于将 token 发送到其他钱包的工作流程。绝大多数 token 交易都发生在转账工作流程中。

执行转账合约非常简单。如果 Alice 想要向 Bob 发送 10 个 token，她的钱包会向 token 合约的地址发送一个交易，用 Bob 的地址和“10”作为参数调用 transfer 函数。token 合约调整 Alice 的余额（-10）和 Bob 的余额（10）并发出+转账事件。

第二个工作流是使用 approve 的双事务工作流，然后是 transferFrom。此工作流允许 token 所有者将他们的控制权委托给另一个地址。它通常用于将控制委托给合约以分发 token，但它也可以由交易所使用。例如，如果一家公司为 ICO 出售代币，他们可以批准一个众包合约地址来分发一定数量的代币。然后，众包合约可以从 token 合约所有者余额转账到 token 的每个买方。

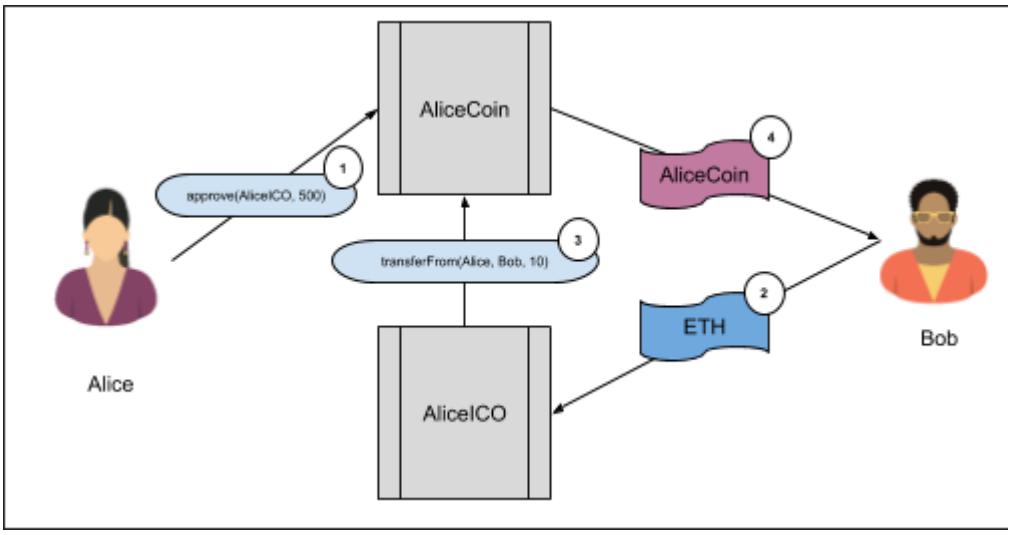


图 1. ERC20token 的两步 approve 和转账工作流程

对于 approve&transferFrom 工作流程，需要两个事务。假设爱丽丝希望允许 AliceICO 合约将所有 AliceCoin 代币的 50%出售给 Bob 和 Charlie 这样的买家。首先，Alice 启动 AliceCoin ERC20 合约，将所有 AliceCoin 发布到她自己的地址。然后，Alice 启动 AliceICO 合约，该合约可以出售以太币的 token。接下来，Alice 启动 approve& transferFrom 工作流程。她向 AliceCoin 发送了一笔交易，称为 approve，其中包含 AliceICO 的地址和 totalSupply 的 50%。这将触发 Approval 事件。现在，AliceICO 合约可以出售 AliceCoin。当 AliceICO 从 Bob 接收以太坊时，它需要向 Bob 发送一些 AliceCoin 作为回报。为此，AliceICO 调用 AliceCoin transferFrom 函数，Alice 的地址作为发送者，Bob 的收件人地址和给 Bob 的 token 数量。AliceCoin 合约将余额从 Alice 的地转账到 Bob 的地址，并触发 transfer 事件。AliceICO 合约可以无限次地调用 transferFrom，只要它不超过 Alice 设置的批准限制。AliceICO 合约可以通过调用 allowance 函数来跟踪它可以销售多少 AliceCoin 代币。

ERC20 实现

虽然可以在大约 30 行 Solidity 代码中实现与 ERC20 兼容的 token，但大多数实现都比较复杂，以解决潜在的安全漏洞。EIP20 标准中提到了两种实现：

Consensys EIP20

一个简单易读的 ERC20 兼容 token 实现。

您可以在此处阅读 Consensys 实施的 Solidity 码：

https：//github.com/ConsenSys/Tokens/blob/master/contracts/eip20/EIP20.sol

OpenZeppelin StandardToken

该实现与 ERC20 兼容，具有额外的安全预防措施。它构成了 OpenZeppelin 库的基础，实现了更复杂的 ERC20 兼容 token，包括众筹上限，拍卖，发送时间表和其他功能。
您可以在此处查看 OpenZeppelin StandardToken 的 Solidity 代码：

https： //github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol

发布我们自己的 ERC20token

让我们创建并启动我们自己的 token。对于这个例子，我们将使用 truffle 框架（参见[truffle]）。该示例假定您已经安装了 truffle，配置它，并且熟悉其基本操作。

我们将使用符号“MET”调用我们的 token“Mastering Ethereum Token”。

您可以在本书的 GitHub 存储库中找到此示例：https：//github.com/ethereumbook/ethereumbook/blob/first_edition/code/METoken

首先，让我们创建并初始化一个 truffle 项目目录，就像我们在[truffle_project_directory]中所做的那样。运行这四个命令并同意任何问题的默认答案：

```
$ mkdir METoken
$ cd METoken
METoken $ truffle init
METoken $ npm init
```

您现在应该具有以下目录结构：

```
METoken/
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── package.json
├── test
├── truffle-config.js
```

```
└── truffle.js
```

编辑 truffle.js 配置文件以设置 truffle 环境，或复制我们使用的那个：

https://github.com/ethereumbook/ethereumbook/blob/first_edition/code/Faucet/truffle.js

如果您使用示例 truffle.js，请记住在包含测试私钥的 METoken 文件夹中创建 file .env，以便在公共以太坊测试网络（例如 ganache 或 Kovan）上进行测试和部署。您可以从 MetaMask 导出测试网络私钥。

警告	仅限没有在以太坊主网络上持有资金的测试密钥或测试助记符使用。切勿使用持有真钱的密钥进行测试。

对于我们的示例，我们将导入 OpenZeppelin StandardContract，它实现了一些重要的安全检查并且易于扩展。让我们导入该库：

```
$ npm install zeppelin-solidity

+ zeppelin-solidity@1.6.0
added 8 packages in 2.504s
```

zeppelin-solidity 包将在 node_modules 目录下添加大约 250 个文件。OpenZeppelin 库包含的内容远远超过 ERC20 token，但我们只使用它的一小部分。

接下来，让我们写下我们的 token 合约。创建一个新文件 METoken.sol 并从 GitHub 复制示例代码：

https://github.com/ethereumbook/ethereumbook/blob/first_edition/code/METoken/contracts/METoken.sol

我们的合约非常简单，因为它继承了 OpenZeppelin StandardToken 库的所有功能：
METoken.sol：实施 ERC20token 的 Solidity 合约
link:code/METoken/contracts/METoken.sol[]

在这里，我们定义了可选变量 name，symbol 和 decimals。我们还定义了一个 _initial_supply 变量，设置为 2100 万个 token，以及两个小数的细分（总计 21 亿）。

在合约的初始化（构造函数）函数中，我们将 totalSupply 设置为等于_initial_supply，并将所有_initial_supply 分配给创建 METoken 合约的帐户余额（msg.sender）。

我们现在使用 truffle 来编译 METoken 代码：

```
$ truffle compile
Compiling ./contracts/METoken.sol...
Compiling ./contracts/Migrations.sol...
Compiling zeppelin-solidity/contracts/math/SafeMath.sol...
Compiling zeppelin-solidity/contracts/token/ERC20/BasicToken.sol...
Compiling zeppelin-solidity/contracts/token/ERC20/ERC20.sol...
Compiling zeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol...
Compiling zeppelin-solidity/contracts/token/ERC20/StandardToken.sol...
```

如您所见，truffle 合并了 OpenZeppelin 库中的必要依赖项，并编译了这些合约。让我们设置一个迁移脚本来部署 METoken 合约。在 METoken / migrations 文件夹中创建一个新文件 2_deploy_contracts.js。复制从 Github 存储库中的示例内容：

https://github.com/ethereumbook/ethereumbook/blob/first_edition/code/METoken/migrations/2_deploy_contracts.js

这是它包含的内容：

2_deploy_contracts：迁移以部署 METoken
link:code/METoken/migrations/2_deploy_contracts.js[]

在我们部署其中一个以太坊测试网络之前，让我们开始一个本地区块链来测试一切。从 ganache-cli 的命令行或图形用户界面启动 ganache 区块链，就像我们在[using_ganache]中所做的那样。

一旦 ganache 启动，我们就可以部署我们的 METoken 合约，看看是否一切都按预期工作：

```
$ truffle migrate --network ganache
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xb2e90a056dc6ad8e654683921fc613c796a03b89df6760ec1db1084ea4a084eb
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
```

```
Saving successful migration to network...
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying METoken...
  ... 0xbe9290d59678b412e60ed6aefedb17364f4ad2977cfb2076b9b8ad415c5dc9f0
  METoken: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
```

在 ganache 控制台上，我们应该看到我们的部署创建了 4 个新事务：

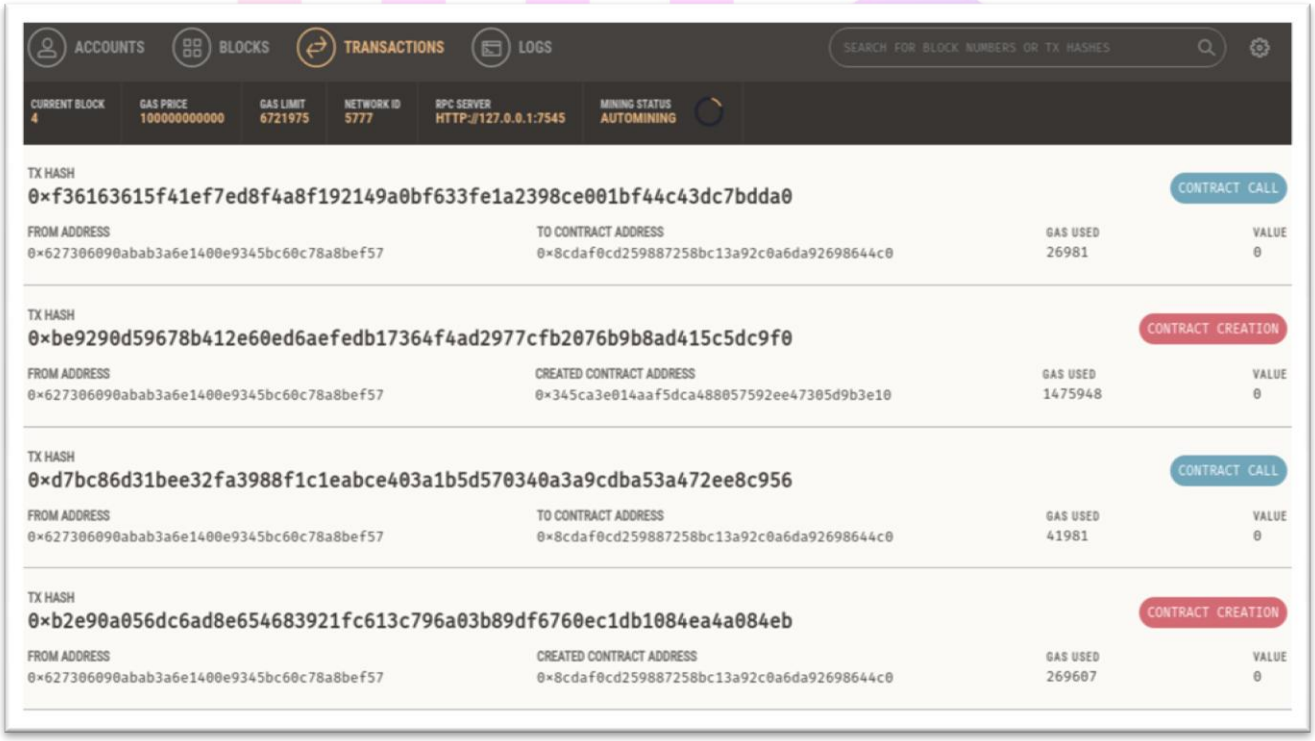


图 2 使用 truffle console 与 METoken 进行交互

我们可以使用 truffle console 与 ganache 区块链的合约进行交互。这是一个交互式的 JavaScript 环境，它提供了对 truffle 环境的访问，并通过 Web3 向区块链提供了访问。在这种情况下，我们将 truffle console 连接到 ganache 区块链：

```
$ truffle console --network ganache
```

truffle(ganache)>

truffle(ganache)>表示我们已经连接到 ganache 区块链，并准备好输入命令。truffle console 支持所有的 truffle 命令，因此我们可以从 console 中进行编译(compile)和迁移(migrate)。我们已经运行了这些命令，直接进入合约。在 truffle 环境中，METoken 合约作为一个 JavaScript 对象存在。在提示符处输入 METoken，它将转储整个合约的定义:

```
truffle(ganache)> METoken
{ [Function: TruffleContract]
  _static_methods:

[...]

currentProvider:
 HttpProvider {
   host: 'http://localhost:7545',
   timeout: 0,
   user: undefined,
   password: undefined,
   headers: undefined,
   send: [Function],
   sendAsync: [Function],
   _alreadyWrapped: true },
network_id: '5777' }
```

METoken 还公开了几个属性，例如合约的地址（由转移命令(migrate)部署）：

```
truffle(ganache)> METoken.address
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

如果我们想要与已部署的合约进行交互，必须用 JavaScript 语言“promise”的形式进行异步调用。 我们用 deployed 函数来获取合约实例，然后调用 totalSupply 函数：

```
truffle(ganache)> METoken.deployed().then(instance => instance.totalSupply())
BigNumber { s: 1, e: 9, c: [ 2100000000 ] }
```

接下来，让我们使用由 ganache 创建的账户来检查我们的 METoken 余额并将一些 METoken 发送到另一个地址。 首先，让我们获取帐户地址：

```
truffle(ganache)> let accounts
undefined
truffle(ganache)> web3.eth.getAccounts((err,res) => { accounts = res })
```

```
undefined
truffle(ganache)> accounts[0]
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
```

帐户列表现在包含由 ganache 创建的所有帐户，以及账户[0]是部署了该 METoken 合约的帐户。 它应该有一个 METoken 的 balance，因为 METoken 构造函数将整个令牌提供给创建它的地址。 让我们核对：

```
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.balanceOf(accounts[0]).then(console.log) })
undefined
BigNumber { s: 1, e: 9, c: [ 2100000000 ] }
```

最后，让我们调用合约的传递函数，将 1000.00 METoken 从帐户[0]转移到帐户[1]：

```
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.transfer(accounts[1], 100000) })
undefined
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.balanceOf(accounts[0]).then(console.log) })
undefined
truffle(ganache)> BigNumber { s: 1, e: 9, c: [ 2099900000 ] }

undefined
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.balanceOf(accounts[1]).then(console.log) })
undefined
truffle(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

提示	METoken 具有 2 位精度的小数，这意味着 1 个 METoken 在合约中是 100 个单位。 当我们转移 1000 个 METoken 时，我们在传递函数中将该值指定为 100,000。

正如你所见，在 console 中，帐户[0]现在拥有 20,999,000 MET，帐户[1]拥有 1000 MET。如果你切换到 ganache 图形用户界面，您将看到称为传递函数（transfer function）的记录：

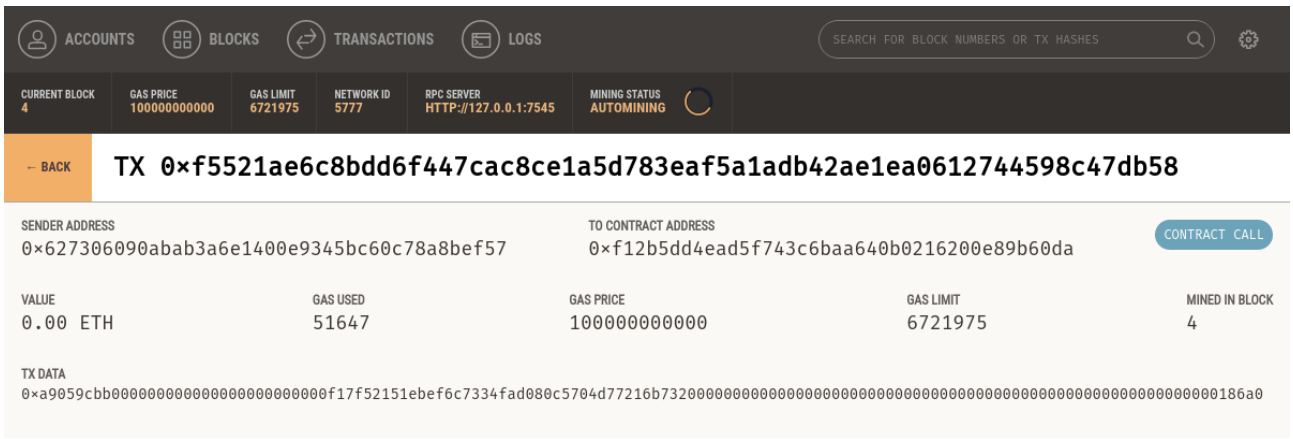


图 3. 发送 ERC20 代币到合约地址

到目前为止，我们已经设置了 ERC20 令牌并从一个帐户转移到另一个帐户。 我们用于这些示范的所有账户都是外部拥有账户（EOAs），这意味着它们由私钥控制，而不是合约。 如果我们将 MET 发送到合约地址会发生什么？ 让我们一起发现！

首先，我们将其他合约部署到我们的测试环境中。 对于这个例子，我们将使用我们的第一份合约 Faucet.sol。 我们将它添加到 METoken 项目中，方法是将其复制到合约目录中。 我们的目录应该是这样的：

```
METoken/
├── contracts
│   ├── Faucet.sol
│   ├── METoken.sol
│   └── Migrations.sol
```

我们也添加了一个 migration, 以便将 Faucet 和 METoken 分离：

```
var Faucet = artifacts.require("Faucet");

module.exports = function(deployer) {
  // Deploy the Faucet contract as our only task
  deployer.deploy(Faucet);
};
```

让我们在 truffle console 中编译(compile)和传递(migrate)合约。

```
$ truffle console --network ganache
truffle(ganache)> compile
Compiling ./contracts/Faucet.sol...
Writing artifacts to ./build/contracts
```

```
truffle(ganache)> migrate
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x89f6a7bd2a596829c60a483ec99665c7af71e68c77a417fab503c394fcd7a0c9
  Migrations: 0xa1ccce36fb823810e729dce293b75f40fb6ea9c9
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Replacing METoken...
  ... 0x28d0da26f48765f67e133e99dd275fac6a25fdfec6594060fd1a0e09a99b44ba
  METoken: 0x7d6bf9d5914d37bcba9d46df7107e71c59f3791f
Saving artifacts...
Running migration: 3_deploy_faucet.js
  Deploying Faucet...
  ... 0x6fbf283bcc97d7c52d92fd91f6ac02d565f5fded483a6a0f824f66edc6fa90c3
  Faucet: 0xb18a42e9468f7f1342fa3c329ec339f254bc7524
Saving artifacts...
```

太棒了！现在让我们发送一些 MET 到 Faucet 合约吧：

```
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.transfer(Faucet.address, 100000) })
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.balanceOf(Faucet.address).then(console.log)})
truffle(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

好了，我们已经把 1000 个 MET 转移到 Faucet 合约上了。现在，我们怎么从 Faucet 上提取这些 MET 呢？

请牢记,Faucet.sol 是一个非常简单的合约。它只有一个功能,那就是提取以太(ether)。它并没有一个提取 MET 或者其他任何 ERC20 代币的功能。如果我们让它行使提取（非以太）的动作，它会尝试发送以太，但由于 faucet 不会达到以太的平衡，这个动作将会失败。METoken 合约确知 Faucet 有余额，但它唯一能转移余额的方式是接收到来自合约地址的转账命令。 不管怎样，我们需要让 Faucet 合约调用 METoken 的传递函数。

如果你想知道下一步该做什么的话，就别想了。 这个问题没有解决办法。 发送到 Faucet 的 MET 卡住了，永久的。 只有 Faucet 合约才能转移它，Faucet 合约没有代码来调用 ERC20 代币合约的传递函数。

也许你预料到了这个问题， 但很大可能是你并没有。 实际上，数百名以太坊用户也无意将各种代币转移到没有任何 ERC20 能力的合约上。 据估计，价值超过 250 万美元的代币像这样被“卡住”，并且永远丢失。

ERC20 代币的用户在转移代币时无意中丢失代币的方式之一是他们尝试转移到具有交易或其他服务的地方。 他们从交易所网站复制了以太坊地址，认为只需发送代币即可。 但是，许多交易所实际上公布的都是合约的收货地址！ 这些合约具有许多不同的功能，通常将所有发送给他们的资金都清扫到“冷库”或其他集中的钱包。尽管有许多警告说“不要将代币发送到这个地址”，但依然有许多代币以这种方式丢失。

演示批准（approve）和转让（transferForm）的工作流程

我们的 Faucet 合约无法处理 ERC20 代币，使用传递函数向它发送代币会导致这些代币丢失。 我们重写合约，并让它能处理 ERC20 代币。 具体而言，我们将把它变成一个可以把 MET 发给任何访问者的 faucet。

在这个例子中，我们制作了一个 truffle 程序目录的副本，称之为 METoken_METFaucet，初始化 truffle，npm，安装 OpenZeppelin 相关项并复制 METoken.sol 合约。 请参阅我们的第一个示例 Launching our own ERC20 token，已有详细说明。

现在，让我们创建一个新的 faucet 合约，称之为 METFaucet.sol.看起来像这样：

METFaucet.sol: 一份为了 METoken 的 faucet 合约。

METFaucet.sol: a faucet for METoken

include::code/METoken_METFaucet/contracts/METFaucet.sol

我们对基本的 faucet 样本做了一些改变。 由于 METFaucet 将使用 METoken 中的 transferFrom 函数，因此它需要两个额外的变量。 一个将保存已部署 METoken 合约的地址。另一个将保存允许 faucet 提款的 MET 所有者的地址。 METFaucet 将调用 METoken.transferFrom 并指示它将 MET 从所有者处移至 faucet 中收到提取请求的地址。

我们在这里声明这两个变量：

```
StandardToken public METoken;
address public METOwner;
```

由于 faucet 需要使用 METoken 和 METOwner 的正确地址进行初始化，因此我们需要声明一个自定义构造函数：

```
// METFaucet constructor, provide the address of METoken contract and
// the owner address we will be approved to transferFrom
function METFaucet(address _METoken, address _METOwner) public {
```

```
    // Initialize the METoken from the address provided
    METoken = StandardToken(_METoken);
    METOwner = _METOwner;
}
```

下一个改变是提取功能。METFaucet 使用 METoken 中的 transferFrom 函数而不是调用传递函数，并要求 METoken 将 MET 传递给 faucet 接收人：

```
// Use the transferFrom function of METoken
METoken.transferFrom(METOwner, msg.sender, withdraw_amount);
```

最后，由于 faucet 不再发送以太，所以我们应该可以防止任何人将以太发送到 METFaucet，因为我们不希望它卡住。 我们更改回退应付功能以拒绝传入的以太网，并使用回复功能来还原所有收款：

```
// REJECT any incoming ether
function () public payable { revert(); }
```

现在 METFaucet.sol 代码已准备就绪，我们需要修改迁移脚本来部署它。 这个迁移脚本会更复杂一点，因为 METFaucet 依赖于 METoken 的地址。 我们将使用 JavaScript 承诺按顺序部署这两个合约。 创建 2_deply_contracts.js，如下所示：

```
var METoken = artifacts.require("METoken");
var METFaucet = artifacts.require("METFaucet");
var owner = web3.eth.accounts[0];

module.exports = function(deployer) {

    // Deploy the METoken contract first
    deployer.deploy(METoken, {from: owner}).then(function() {
        // then deploy METFaucet and pass the address of METoken
        // and the address of the owner of all the MET who will approve METFaucet
        return deployer.deploy(METFaucet, METoken.address, owner);
    });
}
```

现在，我们可以测试 truffle console 中的所有内容。 首先，使用迁移函数来部署合约。当 METoken 被部署时，它会将所有 MET 分配给创建它的帐户，web3.eth.accounts [0]。 然

后，在 METoken 中调用批准函数来批准 METFaucet 发送高达 1000 的 MET，用 web3.eth.accounts [0]表示。 最后，为了测试 faucet，我们调用 web3.eth.accounts [1] 中的 METFaucet.withdraw 并尝试提取 10 MET。 以下是控制台命令：

```
$ truffle console --network ganache
truffle(ganache)> migrate
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x79352b43e18cc46b023a779e9a0d16b30f127bfa40266c02f9871d63c26542c7
  Migrations: 0xaa588d3737b611bafd7bd713445b314bd453a5c8
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Replacing METoken...
  ... 0xc42a57f22cddf95f6f8c19d794c8af3b2491f568b38b96fef15b13b6e8bfff21
  METoken: 0xf204a4ef082f5c04bb89f7d5e6568b796096735a
  Replacing METFaucet...
  ... 0xd9615cae2fa4f1e8a377de87f86162832cf4d31098779e6e00df1ae7f1b7f864
  METFaucet: 0x75c35c980c0d37ef46df04d31a140b65503c0eed
Saving artifacts...
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.approve(METFaucet.address, 100000) })
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.balanceOf(web3.eth.accounts[1]).then(console.log) })
truffle(ganache)> BigNumber { s: 1, e: 0, c: [ 0 ] }
truffle(ganache)>          METFaucet.deployed().then(instance          =>
{ instance.withdraw(1000, {from:web3.eth.accounts[1]}) } )
truffle(ganache)>           METoken.deployed().then(instance           =>
{ instance.balanceOf(web3.eth.accounts[1]).then(console.log) })
truffle(ganache)> BigNumber { s: 1, e: 3, c: [ 1000 ] }
```

正如从结果所见，我们可以使用批准和转让工作流程来授权一个合约转移另一个代币中中定义的代币。 如果使用得当，ERC20 代币可以由外部拥有的地址和其他合约使用。
然而这样正确管理 ERC20 代币的负担会推送到用户界面。如果用户错误地尝试将 ERC20 代币转移到合约地址，并且该合约不能接收 ERC20 代币，则代币将丢失。

ERC20 代币的问题

ERC20 代币标准已经被极其广泛地采用。成千上万的代币发行即是为了实验新的特性也是为了通过众筹拍卖或 ICO 来融资。不过，正如我们看到，采用 ERC20 代币在将代币转移到合约地址时会有一些问题。(topsun: 即是为了实验新的特性 --→ 即是为了实验新的性能)其中一个不太明显的问题是 ERC20 代币和以太(ether)本身是有细微差别的。以太的转移是靠一笔有收款人地址的交易(transaction)来完成，而代币的转移是在某个具体的代币合约里完成，其接收者是这个代币合约而不是一个收款人地址。代币合约会跟踪余额以及发布事件。在一次代币转移中，实际上没有一笔真正的交易发到代币的接收者那里，而是把收款人地址加到了代币合约中。一笔通常意义上发送以太到某个地址的交易是在改变那个地址的状态。一笔转移代币到某个地址的交易只更改了合约的状态而没有改变收款人地址的状态。即便是一个支持 ERC20 代币的钱包也不会变得能够知道代币余额，除非用户特地添加了个监控代币余额的合约。有些钱包会监听最热门的合约来获取用户所掌握的代币余额信息，但是这仅限于非常小部分的 ERC20 合约。 (topsun:而是把收款人地址加到了代币合约中 --→而是把收款人地址加到了包含于代币合约中映射）。

事实上，一个用户不太可能会想监听所有的代币合约里的代币余额。很多 ERC20 代币更像是垃圾邮件而不是有价值的代币。这些代币会为了吸引用户，自动为有以太活动的账户创建余额。如果你拥有一个有长期活动历史的以太坊地址，特别是如果这个地址是在预售期被创建的，你会发现这里面充满了不知道从哪冒出来的『垃圾』代币。当然，这些地址里不是真正地充满了代币，而是那些代币合约里有你的这些地址。你只有当你用区块链浏览器或者钱包去监听这些合约里你的地址时才会看到这些余额。 （topsun:你只有当你用区块链浏览器或者钱包去监听这些合约里你的地址时才会看到这些余额 --→ 只有当你用区块链浏览器或者钱包去监听这些合约里你的地址时才会看到这些余额代币和以太的具体表现是不一样的。以太是通过发送函数(send function)进行发送并且可以被任意合约里的应付函数(payable function)或任意外部拥有账户(EOA)所接收。代币是通过只存在于 ERC20 合约里的 transfer 或者 approve & transferFrom 函数进行发送，并且至少在 ERC20 中并不会触发任何接收人合约的应付函数。代币的职能本来是应该和以太这样的加密货币一样，但是他们本身和以太一些细微的不同打破了这样的幻想。 (topsun: 1.代币和以太的具体表现是不一样的 --→代币和以太的运作方式是不一样的。 2.任意外部拥有账户(EOA) --→任意外部持有账户(EOA) 3.代币是通过只存在鱼 ERC20 合约里的 --→代币是通过只存在于 ERC20 合约里的 ）现在我们来看另一个问题。想要给别人转以太或者使用以太坊合约，你需要以太来支付相应的交易工作量(gas)。想要给别人转代币，你同样需要以太。你无法用代币来支付一笔交易所需的计算工作量，你的代币合约也同样无法为你支付所需的计算工作量。这就会导致一些非常奇怪的用户体验。打比方说你用某个交易所把一些比特币换成了某种代币。你『接收』到了这些代币并且用钱包监听了这个代币的合约以显示你的余额。这跟你在钱包里拥有的其他加密货币看起没有什么不同。但是如果你现在想要把这些代币转出去，你的钱包就会提醒你说你需要有以太才能转账。你可能会觉得非常奇怪，毕竟你接收这些代币的时候都没有需要用到以太。你可能没有以太。你可能根本就不知道这个代币是基于以太坊 ERC20 的一种代币，你可能还觉得它是一种拥有自己区块链的加密货币。不管怎样，这样

的臆想在你转不出去账的时候破灭了。(topsun:1.打比方说你用某个交易所把一些比特币换成了某种代币 -→ 打比方说你用某个交易所或"Shapeshift"把一些比特币换成了某种代币) 有些问题是针对于 ERC20 代币的，而其他的是更宽泛的、跟以太坊抽象和接口边界有关的问题。有些问题能通过修改代币接口解决，有的可能需要改变以太坊的基础架构（比如外部拥有账户(EOA)和合约(contract)之间的差异，交易(transaction)和消息(message)之间的差异）。有些差异可能并不能被消除，它们需要通过一些用户界面的隐藏使得用户感知不到内部的细微差异。 (topsun:1.跟以太坊抽象和接口边界有关的问题 -→ 是一些和以太坊本身的抽象和接口边界有关的问题 2.有些差异可能并不能被消除，它们需要通过一些用户界面的隐藏使得用户感知不到内部的细微差异。 -→ 有些问题可能并不会被完美解决，但通过用户界面设计的技巧，这些问题得以隐藏，从而使用户感知不到内部的细微差异。）在下面一个章节我们将会介绍一些想要解决以上部分问题的提案。 （topsun:在下面一个章节我们将会介绍一些想要解决以上部分问题的提案。-→ 在下面一个章节我们将会介绍一些针对解决以上部分问题的提案。） ==== ERC223 - a proposed token contract interface standard

ERC223 - 一个代币合约接口标准提案

ERC223 提案试图解决通过检测一次交易的目标地址是否是合约来解决用户因为粗心而错把代币转到一个合约（此合约可能不支持代币转账）。ERC223 强制接收代币的合约实现一个叫做 tokenFallback 的函数。如果一次交易的目标地址是一个合约，而这个合约并没有支持代币（即没有实现 tokenFallback），那么这次交易就会失败。
为了检测一笔交易的目标地址是否是合约，ERC223 的参考实现采用了一个很聪明的方法：利用小段的内联字节码：

```
function isContract(address _addr) private view returns (bool is_contract)
{
   uint length;
   assembly {
        //retrieve the size of the code on target address, this needs assembly
        length := extcodesize(_addr)
   }
   return (length>0);
}
```

关于 ERC223 的讨论可以在这里找到：
https://github.com/ethereum/EIPs/issues/223

ERC223 合约的接口规格：

```
interface ERC223Token {
```

```
  uint public totalSupply;
  function balanceOf(address who) public view returns (uint);

  function name() public view returns (string _name);
  function symbol() public view returns (string _symbol);
  function decimals() public view returns (uint8 _decimals);
  function totalSupply() public view returns (uint256 _supply);

  function transfer(address to, uint value) public returns (bool ok);
  function transfer(address to, uint value, bytes data) public returns (bool ok);
  function transfer(address to, uint value, bytes data, string custom_fallback) public returns (bool ok);

  event Transfer(address indexed from, address indexed to, uint value, bytes indexed data);
}
```

ERC223 并没有被广泛采用。关于 ERC 的讨论里有对 ERC223 向后兼容性以及到底是修改合约接口还是用户界面的争论。这样的争论还在持续。

ERC777 - 一个代币合约接口标准提案

ERC777 是另外一个致力于改善代币合约的提案。这个提案有如下几个目标：

To offer an ERC20 compatibility interface
提供 ERC20 兼容的接口

使用一个类似于以太转账的发送函数来进行代币转账

兼容 ERC820 的代币合约注册

在发送代币之前，合约和地址可以控制哪些代币会通过 tokensToSend 函数发送

接收者可以通过调用 tokensReceived 函数来通知合约和地址 （topsun:接收者可以通过调用 tokensReceived 函数来通知合约和地址 --→ 可以通过调用接收方的 tokensReceived 函数来通知合约和地址）

代币转账交易会在 userData 和 operatorData 中包含元数据

转账到合约或外部拥有地址（EOA）将会是同样的操作

https://github.com/ethereum/EIPs/issues/777

这是关于 ERC777 的细节以及正在进行的讨论的链接：

https://github.com/ethereum/EIPs/issues/777

ERC777 合约的接口规格：

```
interface ERC777Token {
    function name() public constant returns (string);
    function symbol() public constant returns (string);
    function totalSupply() public constant returns (uint256);
    function granularity() public constant returns (uint256);
    function balanceOf(address owner) public constant returns (uint256);

    function send(address to, uint256 amount) public;
    function send(address to, uint256 amount, bytes userData) public;

    function authorizeOperator(address operator) public;
    function revokeOperator(address operator) public;
    function isOperatorFor(address operator, address tokenHolder) public
constant returns (bool);
    function operatorSend(address from, address to, uint256 amount, bytes
userData, bytes operatorData) public;

    event Sent(address indexed operator, address indexed from, address
indexed to, uint256 amount, bytes userData, bytes operatorData);
    event Minted(address indexed operator, address indexed to, uint256 amount,
bytes operatorData);
    event Burned(address indexed operator, address indexed from, uint256
amount, bytes userData, bytes operatorData);
    event AuthorizedOperator(address indexed operator, address indexed
tokenHolder);
    event RevokedOperator(address indexed operator, address indexed
tokenHolder);
}
```

ERC777 的提案提供了一份参考实现。ERC777 依赖于 ERC820 所提出的合约注册。因此一些关于 ERC777 的争论围绕着一次性采纳两个大的变化（一个新的代币标准和一个注册标准）的复杂度。这样的争论还在继续。（topsun: ERC777 依赖于 ERC820 所提出的合约注册 -→ERC777 依赖于 一份在 ERC820 中提到的注册合约提案）

ERC721 - 不可替换的代币标准（契据）

所有我们目前见到过的代币标准都是可替代的代币，这代表着一个代币里的每一个最小单位都是可以互相转换的。ERC20 代币标准只追踪每个账户的最终余额，它并不显式地追踪代币从哪里来的。

ERC721 提案是关于不可替换代币（又被称为契据）的标准
牛津词典这样定义：

契据：契据是一种签署并履行了的法律文件，通常指财产或法律权利的所有权。
把契据这个词用在这里是想表达其『财产所有权』的部分，虽然这些契据现在还没有被任何法律认为是有效法律文件。

不可替换的代币追踪对某个独特事物的所有权。被拥有的事物可以是数字虚拟物品，比如一款游戏或是数字藏品。被拥有的事物也可以是现实世界里的物品，比如一栋房子，一辆车或是一件艺术品。一个契据也可以用来代表负向的拥有权比如借款，抵押，地役权等。ERC721 标准并没有对契据可以追踪所有权的事物做任何限制。ERC721 通过 256 位的标识符来实现了被追踪的事物的唯一标识。（topsun: ERC721 标准并没有对契据可以追踪所有权的事物做任何限制 -→ERC721 标准并没有对契据可以追踪所有权的事物做任何限制和期望） The 此标准的细节和相关讨论记录在下面两个 GitHub 地址里：

初版提案: https://github.com/ethereum/EIPs/issues/721

持续讨论： https://github.com/ethereum/EIPs/pull/841

通过查看 ERC721 的内部数据结构，我们可以较好地体会 ERC20 和 ERC721 的区别：
// Mapping from deed ID to owner

ERC20 通过把拥有者作为映射的主键来实现追踪拥有者的余额，而 ERC721 通过把契据 ID 作为映射的主键来实现追踪每个契据的它的拥有者。这个基本的区别影响着不可替换代币的所有属性。（topsun:而 ERC721 通过把契据 ID 作为映射的主键来实现追踪每个契据的它的拥有者 -→ 而 ERC721 通过把契据 ID 作为映射的主键来实现追踪每个契据和它的拥有者）

ERC721 合约的接口规格：

```
interface ERC721 /* is ERC165 */ {
    event Transfer(address indexed _from, address indexed _to, uint256 _deedId);
    event Approval(address indexed _owner, address indexed _approved, uint256 _deedId);
    event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved);

    function balanceOf(address _owner) external view returns (uint256 _balance);
    function ownerOf(uint256 _deedId) external view returns (address _owner);
    function transfer(address _to, uint256 _deedId) external payable;
    function transferFrom(address _from, address _to, uint256 _deedId) external payable;
    function approve(address _approved, uint256 _deedId) external payable;
    function setApprovalForAll(address _operateor, boolean _approved) payable;
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

ERC721 同时也支持两个可选的接口，一个关于元数据，另一个则是为了遍历契据和其拥有者。

ERC721 可选的元数据接口标准：

```
interface ERC721Metadata /* is ERC721 */ {
    function name() external pure returns (string _name);
    function symbol() external pure returns (string _symbol);
    function deedUri(uint256 _deedId) external view returns (string _deedUri);
}
```

ERC721 可选的遍历接口是：

```
interface ERC721Enumerable /* is ERC721 */ {
    function totalSupply() external view returns (uint256 _count);
    function deedByIndex(uint256 _index) external view returns (uint256 _deedId);
```

```
    function countOfOwners() external view returns (uint256 _count);
    function ownerByIndex(uint256 _index) external view returns (address
_owner);
    function deedOfOwnerByIndex(address _owner, uint256 _index) external
view returns (uint256 _deedId);
}
```

代币标准

在这一章里，我们介绍了几个标准提案以及一些已经大规模采纳的代币合约标准。这些标准到底是做什么的？你应该使用这些标准吗？你应该添加标准之外的功能吗？你到底应该使用哪个标准？我们接下来会一一回答这些问题。 （topsun:你应该使用这些标准吗？你应该添加标准之外的功能吗？ -→ 你应该使用这些标准吗？怎样使用这些标准？你应该添加标准之外的功能吗？）

什么是代币标准？它们的目的是什么？

代币标准是一套最少需要具体实现的功能。这代表着想要遵循，比如 ERC20 标准，你至少需要实现那些 ERC20 所规定的函数和行为。你同时也可以自由地添加并不在标准里规定的功能。 （topsun:1.代币标准是一套最少需要具体实现的功能。 -→ 代币标准是一系列执行的最简化说明。 2.这代表着想要遵循 -→ 这意味着为了合规）
这些标准的主要目标是增加合约之间的协作能力。这样的话，所有的钱包、交易所、用户界面和其他的基础设施可以使用可预测的接口来和任何符合标准的合约交流。 （topsun:可以使用可预测的接口来和任何符合标准的合约交流 -→ 可以使用一种可预测的方式来和任何符合标准的合约交流）

这些标准应该是描述性的而不是指定性的。你如何实现那些函数完全取决于你 — 合约的内部实现和标准无关。标准会规定功能性需求，这规范了在特定场景下的行为，但是并未规定要如何实现。一个例子是当值被设为零时转移函数的行为。

你应该使用这些标准吗？

每个开发者在考虑使用这些标准的时候都面临着一个两难的境地：采用现有的标准还是摆脱标准的限制自主创新？

这个问题并不能轻易地解决。标准会把你固定在一个很窄的『车道』上来限制你创新的空间。但是从另一个角度来看，这些总结于数百个应用的基本标准通常适用于 99%的用例。(topsun:这些总结于数百个应用的基本标准通常适用于 99%的用例。 -→ 这些总结自数百个应用的基本标准通常适用于 99%的用例。）

另外一个更大的考虑是：兼容性和广泛的使用率。如果你使用一个现有的标准，你就能够利用到所有遵循这个标准的系统。如果你不选择使用任何标准，你必须要考虑到自己开发所有基础设施的成本或者是劝说他人认可你的新标准。这种忽略现有标准而自行创造新标准的倾向被称为『非我所创』，与开源文化正好相反。在另一方面，进步和创新有时正是离经叛道的结果。所以这是个不太好抉择的问题，仔细考虑吧！（topsun:1.如果你不选择使用任何标准 -→如果你不选择使用这个标准 2.所以这是个不太好抉择的问题 -→ 所以这是个棘手的选择。
维基百科 『非我所创』(https://en.wikipedia.org/wiki/Not_invented_here)
非我所创或 NIH 综合症（英文：Not Invented Here Syndrome），指的是社会、公司和组织中的一种文化现象，人们不愿意使用、购买或者接受某种产品、研究成果或者知识，不是出于技术或者法律等因素，而只是因为它源自其他地方。（topsun:1.人们不愿意使用、购买或者接受某种产品、研究成果或者知识，不是出于技术或者法律等因素，而只是因为它源自其他地方。 -→ 由于来源于外面以及经费因素（比如版权）人们不愿意使用、购买既有的产品、研究成果 标准或者知识。）

安全性成熟度

除了标准的选择，实现同样也需要选择。当你决定采用某个标准时，比如说 ERC20，你必须接下来决定怎样实现一个 ERC20 兼容的代币。现在已经有一些被以太坊生态广泛使用的『参考』实现。或者你也可以从头开始自己写。这样的选择面临着能够严重影响安全性的两难境地。 （topsun:或者你也可以从头开始自己写。这样的选择面临着能够严重影响安全性的两难境地。 -→或者你也可以从头开始自己写，但安全性有可能因此受到严重影响。）

现有的标准是『久经沙场』的。虽然说我们不能完全证明这些标准就一定是绝对安全的，但大多数标准都支撑着价值数百万美元的代币。它们一直被不断地、猛烈地攻击着。至少到目前为止，还没有任何明显的漏洞出现。开发一个你自己的标准并不是件容易的事 — 有太多不太容易察觉的方式可以侵入你的合约。使用一个彻底测试的、广泛采用的实现是相对来说更安全的做法。在我们之前的例子里，我们使用的是 OpenZeppelin 对 ERC20 的实现，因为这个实现从头到尾都非常注重安全性。 （topsun:1.现有的标准是『久经沙场』的 -→ 现有的很多实现是『久经沙场』的 2.虽然说我们不能完全证明这些标准就一定是绝对安全的，但大多数标准都支撑着价值数百万美元的代币。 -→ 虽然说我们不能完全证明这些实现就一定是绝对安全的，但其中大多数支撑着价值数百万美元的代币。）

如果你使用一个现有的实现，你也可以扩展它。同样，对于这样的冲动需要保持冷静。复杂度是安全性的敌人。每一行新加入的代码都扩大了合约的被攻击面或者引入了新的等待被发现的漏洞。你可能并不能意识到某个漏洞知道你在这个合约上面放了非常多的价值，然后别人黑了你的合约。 （topsun: 你可能并不能意识到某个漏洞知道你在这个合约上面放了

非常多的价值，然后别人黑了你的合约。 -→你可能并不能意识到某个漏洞直到这个合约价值高升，然后别人黑了你的合约。）

对代币接口标准的扩展

在这一章里讨论过的代币标准一开始都只有非常少的接口，能做非常少的事情。很多项目都做了自己的扩展实现来支持它们自身项目的需求。一些扩展如下： （topsun:在这一章里讨论过的代币标准一开始都只有非常少的接口，能做非常少的事情。 -→在这一章里讨论过的代币标准一开始只是一个功能有限的简易接口。）

拥有者控制：

特定地址或者特定地址集（多重签名）被赋予特殊功能，例如黑名单、白名单、铸币、恢复等等。

销毁：

销毁一个代币是指当代币被故意转到某个无法使用的地址或者减少余额来达到减少供应的目的。（topsun:或者减少余额来达到减少供应的目的 -→ 或者抹去余额来达到减少供应的目的 ）

铸币：

以一个可预测的比率增加代币总供应量的能力，或者说代币创造者『发行』代币的能力。（topsun:铸币：以一个可预测的比率增加代币总供应量的能力，或者说代币创造者『发行』代币的能力。 -→ 铸币：增加代币总供应量的能力，这种增长或遵循一个可预测的比率，或取决于代币创造者『发行』代币的意愿/能力。）

众筹：

将代币进行售卖的能力，比如通过拍卖，市场销售，逆向拍卖等。

上限：

预先设定、不可更改的对总供应量的限制，是『铸币』功能的反向功能。

恢复『后门』：

可以恢复资金、转账或者废除某个代币的函数，这通常可以被一个或多个指定的地址激活。 （topsun:恢复『后门』：可以恢复资金、转账或者废除某个代币的函数，这通常可以被一个或多个指定的地址激活。 -→恢复『后门』：可以恢复资金、撤销转账或者废除某个代币的函数，此类代币通常可以被一个或多个指定的地址激活。）

白名单：

限制只能将代币转给特定列表里地址的功能。最常用在各种审查通过后提供代币给『可信任的投资人』。通常来说会有个特定机制来更新白名单。

黑名单：

限制将代币转给某些特定地址。通常来讲会有个特定机制来更新黑名单。

这些以上的功能很多都有参考实现，比如说 OpenZeppelin 库。但有些功能因为是场景相关的，所以只在很少的代币里实现了。现在来说这些场景相关的功能并没有一个广泛采纳的接口标准。

正如之前所讨论的，是否向一个代币标准增加新功能的决策过程实际上是在权衡创新/冒险和兼容性/安全性。（topsun:是否向一个代币标准增加新功能的决策过程实际上是在权衡创新/冒险和兼容性/安全性。-→ 是否向一个代币标准增加新功能的决策过程实际上是在创新与冒险,兼容性与安全性之间的权衡取舍。）

代币和 ICO

代币在以太坊生态里已经有非常爆炸性的进展。很有可能代币会成为所有像以太坊这样的智能合约平台里的一个重要的、基础的组件。

尽管如此，本书所认可的这些标准的重要性和对未来的影响不应该被误认为对现在这些 ICO 的背书。作为一个还处于早期的技术，第一波产品和公司基本上全部都会失败，而且有的会死得非常壮丽。很多现在在以太坊上的代币都是不加掩饰的骗局、非法传销以及抢钱工具。

最重要的是能区分开那些很热门的有着长期愿景和对这项技术的影响的代币，与短期流行的一些 ICO 泡沫。这两者可以同时都成立。但是代币标准和平台会最终挺过这个 ICO 狂热，并且它们很有可能改变这个世界。（topsun: 最重要的是能区分开那些很热门的有着长期愿景和对这项技术的影响的代币，与短期流行的一些 ICO 泡沫。 -→最重要的是能区分开那些有着长期愿景和对这项技术会产生深远影响的代币，与短期流行的一些 ICO 泡沫，前者的增长潜力巨大。）

第十一章 Oracles

Oracles

在本章中，我们将讨论 *oracles*，这些系统可以为以太坊智能合约提供外部数据源。“oracle”一词来自希腊神话，指代一个有能力与神灵交流并且可以预示未来的人。在区块链中，oracle 是一个可以解决以太坊外部需求的系统。 理想情况下，oracles 是*不可信*系统，意味着它们不需要被信任，因为它们是按照去中心化原则运行的。

为什么需要 oracles

以太坊平台的关键组件是以太坊虚拟机，它能够在分散网络中的任何节点上执行程序并更新受共识规则约束的以太坊的状态。为了保持共识，EVM 的所有执行结果必须完全“确定”的，并且结果仅基于共享的太坊状态和已签名的交易。 这有两个特别重要的后果：第一个是 EVM 和智能合约中内没有随机性来源；第二，外部数据只能作为交易的数据有效载荷引入系统。

让我们进一步了解一下这两种后果。要理解在 EVM 中禁止真正的随机函数为智能合约提供随机性，考虑执行这样一个函数后系统再次尝试达成共识的结果：节点 A 将执行命令并代表智能合约存储 3，而节点 B，执行相同的智能合约，反而将存储 7。因此，尽管在相同的环境中中运行完全相同的代码，节点 A 和 B 会对结果状态产生不同的结果。实际上，每次评估智能合约时，可能达到的状态都会不一样。因此，由于其众多节点在世界各地独立运行，网络将永远不可能得到一个关于最终状态的共识。现实世界里， 因为包括以太转移在内的连锁效应会以指数方式增长， 结果很可能比例子中的情况更快的变得更差。

注意，伪随机函数，例如加密安全的哈希函数（它是确定性的，因此它可以，实际上也是 EVM 的一部分），对于许多应用来说是不够的。 举一个抛硬币游戏作为例子，这需要随机生成头部还是尾部— 矿工可以通过玩游戏并且只包括他们获胜的区块中的交易来获得利益。那么我们如何解决这个问题呢？ 所有节点都可以就签名的交易达成一致，因此可以引入外部信息，包括随机性，价格信息，天气预报等，作为发送到网络的交易的数据部分。 但是，这些数据根本无法被信任，因为他们的来源本身无法被核实。因此，我们刚刚推迟了这个问题。我们将使用 oracles 来尝试解决这些问题，我们将在本章的其余部分详细讨论这些问题。

Oracle 使用实例

理想情况下，Oracles 提供了一种无信任（或至少近乎无信任）的获取外在（即“真实世界”或“链外”）信息的方式，比如足球比赛的结果，黄金的价格，或真正随机的数字，以便以太坊平台上的智能合约使用。它们还可直接将数据安全地中继到 DApp 前端。因此，Oracles 可以被当做一种桥接链外世界与智能合约的桥梁。允许智能合约基于真实世界的事件和数据来执行合约，从而大大扩展了它们的使用范围。但是，这也会给以太坊的安全模型带来外部风险。比如说“智能遗嘱”合约，当一个人去世时分配资产。这是智能合约中经常讨论的内容，并突出了可信任的 oracle 的风险。如果由这样的合约控制的遗嘱金额足够高，那么在遗嘱所有者死亡之前攻击 oracle 并触发资产分配的动机是很强的。

请注意，某些 oracles 提供特定的私有数据，例如学术证书或政府 ID。 这些数据的来源，如大学或政府部门，是完全可信的，数据的真实性是主观的（真实性只能通过咨询数据源来确定）。 因此，不能无信地提供这样的数据 - 即，不信任来源 - 因为没有独立可验证的客观事实。 因此，我们将这些数据源包含在我们对“oracles”的定义中，因为它们还为智能合约提供了数据桥梁。 他们提供的数据通常采用证明的形式，如护照或获奖记录。 考试将成为未来区块链平台成功的重要组成部分，特别是在验证身份或声望的相关问题方面，因此探索区块链平台如何为其提供服务非常重要。

oracles 提供的数据可包括：

- 来自诸如量子/热过程等物理来源的随机数/熵：例如，在彩票智能合约中公平地选择赢家
- 自然灾害触发器：例如，触发灾难债券智能合约，例如地震债券的里氏震级测量
- 汇率数据：例如，用于将加密货币准确地挂钩到法定货币
- 资本市场数据：例如，代币化资产/证券的定价
- 基准参考数据：例如，将利率纳入智能金融衍生工具中
- 静态/伪静态数据：安全标识符，国家/地区代码，货币代码等
- 时间和间隔数据：用于基于精确时间测量的事件触发器
- 天气数据：例如，基于天气预报的保险费计算
- 政治事件：用于预测市场决议
- 体育赛事：用于预测市场决议和体育合同
- 地理位置数据：例如，用于供应链跟踪
- 损坏验证：用于保险合同
- 其他区块链上发生的事件：互关功能
- 以太的市场价格：例如，基于法币 gas 价格的 oracles
- 航班统计信息：例如，由团体和俱乐部用于机票汇集

在接下来的部分中，我们将研究实现 oracles 的一些方法，包括基本的 oracle 设计模式，计算 oracles，去中心化 oracles 以及 Solidity 中的一些 oracle 客户端的实现。

Oracle 设计模式

所有的 Oracle 都会根据定义提供一些基本的功能。基本功能包括：

- 从链下收集数据。
- 使用签名消息在链上传输数据。
- 将数据放入智能合约的存储空间，使数据供给其他人使用。

一旦数据在智能合约的存储空间中变为可用，其他智能合约就可以通过调用 oracle 智能合约的“检索”功能来访问它；它也可以通过“查看” oracle 的存储直接访问以太坊节点或支持网络的客户端。

设置 oracle 的三种主要方法可分为 *request-response*， *publish-subscribe* 以及 *immediate-read*。

从最简单的，immediate-read oracles 开始，此种 oracle 为需要立即做出决定的请求提供数据，比如“ethereumbook.info 的地址是什么？”或者“这个人超过 18 岁吗？”那些希望查询此类数据的人倾向于得到“立即”的结果；查找是在需要信息时完成的，可能永远不会再次进行。这种 oracles 的例子包括那些持有组织数据或由组织发布的数据，例如学术证书，拨号代码，机构的会员资格，机场标识符，ID 等。这种类型的 oracle 将数据存储一次到其合约的存储空间中，其他智能合约可以通过从 oracles 请求调用来查找这些信息。这些信息是可能会更新的。 oracle 存储中的数据也可由通过区块链启用的应用程序直接查找，而无需复杂麻烦的工作以及承担交易的 gas 费用。一家商店如果想查看一位想购买酒精制品的顾客年龄，可以使用这种方式。这种类型的 oracle 对于可能需要运行和维护服务器来回答此类问题的组织或公司是非常具有吸引力的。注意，出于例如效率或隐私等原因，由 oracle 存储的数据可能不是 oracle 提供的原始数据，。大学可能会为过去学生的学业成绩设立一个 oracle。但是，存储证书的完整信息（可能包含非常多的的课程信息以及成绩信息）没有太大必要。相反，存储证书的哈希值就足够了。同样，政府可能希望将公民身份证放入以太坊平台，显然需要对细节信息保密。进一步来说，将数据用哈希值代替(更详细的说，in Merkle trees with salts)并且仅将根哈希值存储在智能合约的存储空间中将是组织这种服务的一种有效方式。

下一种设置方式是 publish-subscribe，oracle 将可能会改变的数据以广播的形式发布出来（可能是定期且频繁的），而此 oracle 可能通过智能合约在链上进行查询，也可以通过链下的守护进程观测更新。此类功能类似于 RSS，WebSub 等，其中 oracle 使用新信息进行更新，并且进行标记表示有新数据可供那些“订阅”的人使用。有兴趣的人需要对 oracle 投票来检查最新的信息是否已更改，或者监听 oracle 合约的更新并在更新时时采取行动。例子包括售价报送，天气信息，经济或社会统计数据，交通数据等。在 Web 服务器领域，投票的效率非常低，但在区块链环境下的 peer-to-peer 环境中却不是这样：以太坊的客户必须跟所有状态更新进行同步，包括对合约存储的更改，因此对数据更改投票是对已同步客户端的一种本地调用。以太坊事件日志使应用程序可以特别容易注意的发现 oracle 的更新，

因此这种模式在某些方面甚至可以被视为一种“推送”服务。但是，如果投票是通过智能合同完成的，这对于某些去中心化的应用可能是必要的（例如，在无法激活激励的情况下），则可能产生大量的 gas 支出。

request-response 类别是最复杂的：此种情况数据太大而无法存储到智能合约内，用户一次只能使用整个数据集的一小部分。它也是数据提供商的一种适用模型。实际上，这样的 oracle 可以实现为一种包括链上智能合约和用于监视请求，检索以及返回数据的链外基础结构的系统。来自去中心化的应用程序的数据请求通常是涉及许多步骤的一种异步过程。在这种模式中，首先，EOA 与去中心化的的应用程序一起进行交易，从而与 oracle 智能合约中定义的功能进行交互。此函数会初始化对 oracle 的请求，除了可能包含回调函数和调度参数等补充信息之外，还使用相关参数详细说明所请求的数据。一旦交易得到验证，就可以将 oracle 请求视为 oracle 合约发出的一种 EVM 事件，或者作为一种状态更新;参数可以取回并用于执行链下数据源的查询。 oracle 可能还需收费来处理请求，为回调支付的 gas 费用，以及访问所请求数据的权限。最后，结果会有由 oracle 的所有者签名，证明在给定时间中的数据有效性，并在交易中传递给发出请求的去中心化应用程序 -直接或者通过 oracle 合约。根据调度参数，oracle 可以定期广播交易来更新这些数据（例如，一天结束后的定价信息）。

Request-response oracle 的步骤可以总结如下：

1.从 DApp 接收查询指令。
2.解析查询指令。
3.检查是否提供了付款和数据访问权限。
4.从链外源检索相关数据（并在必要时加密）。
5.对交易进行签名并将数据附上。
6.将交易广播到网络。
7.安排例如通知等任何进一步必要的交易。

其他方案也是可能的;例如,可以从 EOA 请求数据并直接返回数据,从而无需使用 oracle 智能合约。类似地，可以向支持物联网的硬件传感器发出请求和响应。因此，oracles 可以是人，软件或硬件。

此处描述的 request-response 模式常见于客户端-服务器体系结构中。虽然这是一种有用的消息传递模式，允许应用程序进行双向对话，但在某些情况下这可能是不合适的。例如，需要来自 oracle 的利率的智能债券可能必须在请求-响应模式下每天请求数据，以确保利率始终是正确的。鉴于利率不经常变化，此时 publish-subscribe 模式可能更合适 -尤其是考虑到以太坊的带宽有限。

Publish-subscribe 是一种模式，其中发布者（这里指 oracles）不直接向接收者发送消息，而是将发布的消息划分到不同的类中。订阅者只能订阅一个或多个类并仅检索那些感兴趣的类。在这种模式下，oracle 可能会在每次更改时将利率写入其自己的内部存储。多个订阅的 DApp 可以简单地从 oracle 合约中读取它，从而减少对网络带宽的影响，同时最大限度地降低存储成本。

在广播或组播模式中，一个 oracle 将所有消息发布到频道，订阅合约将在各种订阅模式下收听频道。例如，oracle 可能会将消息发布到加密货币汇率通道。订阅智能合约如果需要时间序列，例如移动平均计算，则可以请求信道的全部内容;另一个可能只需要最新的现货价格计算费率。广播模式适用于 oracle 不需要知道订阅者身份的情况.

数据认证

如果我们假设 DApp 查询的数据源既具有权威性又值得信赖（这是很重要的假设），那么一个悬而未决的问题仍然存在：鉴于 oracle 和请求 - 响应机制可能由不同的实体操作，我们如何相信这种机制？ 数据在传输过程中是存在被篡改的可能性的，因此离线方法需要证明返回数据的完整性。 数据验证的两种常用方法是_authenticity proofs_和_trusted 执行环境（TEEs）。

真实性证明是数据未被篡改的加密保证。基于各种证明技术（例如，数字签名），它们有效地将信任从数据载体转移到证明者（即证明的提供者）。通过验证链上的真实性证明，智能合约能够在对其进行操作之前验证数据的完整性。 http://www.oraclize.it/ [Oraclize]是利用各种真实性证明的 oracle 服务的一个例子。 目前可用于以太坊主网上数据查询的一个验证是 TLSNotary 验证。 TLSNotary 证明允许客户端向第三方提供客户端与服务器之间发生的 HTTPS Web 流量证据。虽然 HTTPS 本身是安全的，但它不支持数据签名。因此，TLSNotary 证明依赖于 TLSNotary（通过 PageSigner）签名。 TLSNotary 证明利用传输层安全性（TLS）协议，使得 TLS 主密钥在访问数据后对数据进行签名， 并且允许其在三方之间分配：服务器（oracle），受审计方（Oraclize）和审核员。 Oraclize 使用 Amazon Web Services（AWS）虚拟机实例作为审核员，可以验证自实例化以来未经修改。此 AWS 实例会存储 TLSNotary 机密，允许它提供诚实证据。尽管它提供了比纯粹的请求 - 响应机制更高的数据防篡改保证，但这种方法确实需要假设 Amazon 本身不会篡改 VM 实例。

http：//www.town-crier.org/ [Town Crier]是基于 TEE 方法的经过验证的数据馈送 oracle 系统;这些方法利用基于硬件的安全区域来确保数据的完整性。 Town Crier 使用 Intel 的“(SGX（软件卫士扩展))”软件卫士扩展（SGX）)软件卫士扩展（SGX），以确保来自 HTTPS 的查询结果是真的。 SGX 提供完整性保证，确保在安全区内运行的应用程序可以受到 CPU 的保护，防止被任何其他进程篡改。它还提供保密性，在安全区内运行的程序状态对于其他同区域的进程来说都是不可见的。最后，SGX 允许证明，通过生成数字签名的证据，

证明应用程序 - 通过其构建的散列安全地识别 - 实际上在飞地内运行。通过验证此数字签名，分散式应用程序可以证明 Town Crier 实例在 SGX 飞地内安全运行。反过来，这证明实例没有被篡改，因此 Town Crier 发出的数据是真实的。此外，机密性还允许 Town Crier 通过允许使用 Town Crier 实例的公钥加密数据查询来处理私有数据。在诸如 SGX 之类的飞地内运行 oracle 的查询/响应机制有效地允许我们将其视为在受信任的第三方硬件上安全运行，确保所请求的数据被返回到未被篡改（假设我们信任 Intel / SGX）。

计算 Oracles

到目前为止，我们仅在请求和提供数据的背景下讨论了 oracles。 然而，oracles 也可以用于执行任意计算，这个功能在以太坊固有的块气限制和相对昂贵的计算成本的情况下特别有用。 计算 oracles 不仅可以中继查询结果，还可以用于对一组输入执行计算，并返回计算结果，这可能是不可行的，无法计算链上计算结果。 例如，可以使用计算 oracle 执行计算密集型回归计算，以估计债券合约的收益率。

如果您相信一个中心化但可审计的服务，您可以再次访问 Oraclize。它们提供的服务允许去中心化应用程序请求沙盒 AWS 虚拟机中计算的结果。 AWS 实例根据存放在存档中的，经过用户配置的 Dockerfile 创建出一个可执行容器，而该存档会被上传到星际文件系统（IPFS; 请参阅 [data_storage_sec] ）。Oraclize 会根据请求，基于哈希值检索此存档，然后在 AWS 上初始化并执行此 Docker 容器，将提供的参数当做环境变量。被容器化的应用程序会根据时间限制来执行计算，并将结果以标准输出的形式写出，Oraclize 可以将此结果检索并返回给去中心化应用程序。Oraclize 目前只在可审计的 t2.micro AWS 实例上提供此服务，因此如果一次计算的价值比较高，则需要检查是否执行了正确的 Docker 容器。尽管如此，这不是一个真正意义上的去中心化解决方案。

作为可验证 oracle 真假的标准，“cryptlet”的概念已融为 MicrosoftESC 框架的一部分。 Cryptlet 在经过加密的环境内执行，该封装可以抽象出基础配置，例如 I/O，并附加 CryptoDelegate，以便自动的对传入和传出的消息签名，确认以及验证。 Cryptlet 支持分布式交易，因此合约逻辑可以以复杂的多步骤，多区块链，外部系统交易的 ACID 方式处理。这允许开发人员创建便携，隔离的以及私有的解决方案以便在智能合约中使用。

Cryptlet 遵循如下的格式：

```
public class SampleContractCryptlet : Cryptlet
{
public SampleContractCryptlet(Guid id, Guid bindingId, string name,
       string address, IContainerServices hostContainer, bool contract)
       : base(id, bindingId, name, address, hostContainer, contract)
```

```
        {
            MessageApi = new CryptletMessageApi(GetType().FullName,
                new SampleContractConstructor())
```

对于更加去中心化的的解决方案，我们可以使用 https://truebit.io/[TrueBit]，它提供了可扩展并可验证的离线计算解决方案。他们使用一个包含解算器和验证器的系统，这些解算器和验证器被激励分别执行计算并验证这些计算。如果一个解受到怀疑，系统会开始进行子集的迭代验证，而这个过程会在链上执行 —一种“验证游戏”。游戏通过一系列循环进行，每个循环递归地计算的越来越小的子集。游戏最终进入最后一轮，此时的验证是微不足道的，以至于评委-也就是以太坊矿工 - 可以对结果做出最终裁决。实际上，TrueBit 是计算市场的一种实现，允许去中心化的应用程序付费使用可在线下执行的验证计算，但依靠以太坊来强制制定验证游戏的规则。理论上，这使无信任的智能合约能够安全地执行任何计算任务。

TrueBit 等系统有广泛的应用，从机器学习到 PoW 验证。后者的一个例子是 Doge-Ethereum 桥，它使用 TrueBit 来验证 Dogecoin 的 PoW 工作证明（Scrypt），此工作占用内存大，计算量大，因此无法在以太坊块 gas 限制内完成。通过在 TrueBit 上执行此验证，可以在以太坊的 Rinkeby testnet 上的智能合约中安全地验证 Dogecoin 交易。

去中心化 Oracles

*虽*然中心化数据或计算 oracles 足以满足许多应用，它们在以太坊网络中是很多潜在的单点故障。围绕去中心化 oracles 已经提出了许多方案，为了确保数据可用性以及个体数据提供者网通过链上数据聚合系统创建网络。

https://www.smartcontract.com/link [ChainLink] 提出了一个去中心化的 oracle 网络，包括三个关键的智能合约＆＃x2014; 一个声望合约，一个订单匹配合约，以及一个聚合合约—以及数据提供者的链下注册表。声望合约用于跟踪数据提供商的行为。声望合约中的分数用于填充链下注册表。订单匹配合约根据声望合约从 oracles 中选择最佳出价。然后它会最终确定一个服务级别的协议，其中包括查询参数和所需的 oracles 数量。这意味着购买者无需直接与个人 oracle 进行交易。聚合合约从多个 oracles 收集响应（根据“提交-显示”逻辑提交），计算查询的最终汇总结果，最后将结果反馈到声望合约中。

这种去中心化方法的主要挑战之一是汇总函数的制定。ChainLink 建议计算加权响应，允许为每个 oracle 的相应提供一个有效性分数。此时检测“无效”分数是非常重要的，因为它依赖于这样的前提：通过与其他人提供的数据计算偏离程度来是不正确的。根据响应分布中的 oracle 响应的位置计算有效性分数可能会错误的惩罚正确相应。因此，ChainLink 提供了一组标准的聚合合约，但也允许指定自定义的聚合合约。

一个相关的想法是 SchellingCoin 协议。在这里，多个参与者报告响应值，中位数被视为 “正确”回答。汇报者需要提供重新分配的存款，以支持更接近中位数的响应值，从而激励汇报与其他响应值相似的响应。一个共同的响应值，也称为谢林点，回复者可认为谢林点是一个自然而明显的协调目标，预计会更接近实际价值。

TrueBit 的 Jason Teutsch 最近提出了一种新的去中心化链下数据可用性设计。此设计利用了一个专用的 PoW 工作证明区块链，此区块链能够正确报告在给定时期内注册的数据是否可用。矿工尝试下载，存储和传播所有当前注册的数据，从而保证数据在本地可用。虽然这样的系统在每个挖掘节点存储并传播所有注册的数据非常昂贵，但是系统允许通过在注册周期结束之后释放数据来重用存储空间。

Oracle 客户端基于 Solidity 的接口

[using_oraclize_to_update_the_eth_usd] 是一个 Solidity 示例，演示了如何使用 Oraclize 从一个 API 连续查询 ETH/USD 价格并以可用的方式存储结果。

Example 1. 通过 Oraclize 从外部数据源更新 ETH/USB 汇率

```
/*
   ETH/USD price ticker leveraging CryptoCompare API
   This contract keeps in storage an updated ETH/USD price,
   which is updated every 10 minutes.
 */
pragma solidity ^0.4.1;
import “github.com/oraclize/ethereum-api/oraclizeAPI.sol”;
/*
   “ oraclize_ ”  prepended  methods  indicate  inheritance  from
“usingOraclize”
 */
contract EthUsdPriceTicker is usingOraclize {
    uint public ethUsd;
    event newOraclizeQuery(string description);
    event newCallbackResult(string result);
    function EthUsdPriceTicker() payable {
        // signals TLSN proof generation and storage on IPFS
        oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);
        // requests query
        queryTicker();
    }
```

```
        function __callback(bytes32 _queryId, string _result, bytes _proof)
public {
            if (msg.sender != oraclize_cbAddress()) throw;
            newCallbackResult(_result);
            /*
             * Parse the result string into an unsigned integer for on-chain
use.
             * Uses inherited “parseInt” helper from “usingOraclize”,
allowing for
             * a string result such as “123.45” to be converted to uint 12345.
             */
            ethUsd = parseInt(_result, 2);
            // called from callback since we're polling the price
            queryTicker();
        }
        function queryTicker() public payable {
            if (oraclize_getPrice(“URL”) > this.balance) {
                newOraclizeQuery(“Oraclize query was NOT sent, please add some
ETH to cover for the query fee”);
            } else {
                newOraclizeQuery(“Oraclize query was sent, standing by for
the answer...”);
                // query params are (delay in seconds, datasource type,
                // datasource argument)
                // specifies JSONPath, to fetch specific portion of JSON API
result
                oraclize_query(60 * 10, “URL”,
                    “json(https://min-api.cryptocompare.com/data/price?\
                    fsym=ETH&tsyms=USD,EUR,GBP).USD”);
            }
        }
    }
```

如果要与Oraclize集成，合约 EthUsdPriceTicker 必须是pass的子项；usingOraclize 合约在 oraclizeAPI 文件中定义。 数据请求是通过 oraclize_query API实现的，该函数继承自 usingOraclize 合约。 这是一个重载函数，至少需要两个参数：

- 支持的数据源，例如URL，WolframAlpha，IPFS或computation
- 数据源的参数，可能包括JSON或XML的解析助手

价格查询通过 queryTicker 函数实现。 若要执行查询，Oraclize 要求支付少量以太，包括处理结果的gas成本，将结果传输到 __callback 函数的成本，以及附加的服务费。 此数量取决于数据源，如果指定，还取决于所需的证明类型。 一旦检索到数据，__callback 函数将由 Oraclize 控制的帐户调用，该帐户被允许进行回调；回调将会传递响应值以及一个唯一的 queryId ，queryId 参数可用于处理和跟踪oraclize中挂起的回调。

金融数据提供商 Thomson Reuters 也为以太坊提供了oracle服务，称为BlockOne IQ，允许私有的或经过允许的网络上运行的智能合约请求市场数据以及参考数据。 [合约调用 blockone_iq 来获取市场数据] 展示了 oracle 的接口，以及客户端进行请求的合约。

Example 2. 合约调用blockone_iq获取市场数据

```
pragma solidity ^0.4.11;
contract Oracle {
    uint256 public divisor;
    function initRequest(
       uint256   queryType,   function(uint256)   external
onSuccess,
       function(uint256
    ) external onFailure) public returns (uint256 id);
    function  addArgumentToRequestUint(uint256  id,  bytes32
name, uint256 arg) public;
    function addArgumentToRequestString(uint256 id, bytes32
name, bytes32 arg)
        public;
    function executeRequest(uint256 id) public;
    function getResponseUint(uint256 id, bytes32 name) public
constant
        returns(uint256);
```

```
        function getResponseString(uint256 id, bytes32 name)
public constant
            returns(bytes32);
        function getResponseError(uint256 id) public constant
returns(bytes32);
        function deleteResponse(uint256 id) public constant;
    }
    contract OracleB1IQClient {
        Oracle private oracle;
        event LogError(bytes32 description);
        function OracleB1IQClient(address addr) public payable {
            oracle = Oracle(addr);
            getIntraday(“IBM”, now);
        }
      function getIntraday(bytes32 ric, uint256 timestamp) public
{
            uint256 id = oracle.initRequest(0, this.handleSuccess,
this.handleFailure);
            oracle.addArgumentToRequestString(id, “symbol”,
ric);
            oracle.addArgumentToRequestUint(id, “timestamp”,
timestamp);
            oracle.executeRequest(id);
        }
       function handleSuccess(uint256 id) public {
            assert(msg.sender == address(oracle));
            bytes32 ric = oracle.getResponseString(id, “symbol”);
            uint256 open = oracle.getResponseUint(id, “open”);
            uint256 high = oracle.getResponseUint(id, “high”);
            uint256 low = oracle.getResponseUint(id, “low”);
            uint256 close = oracle.getResponseUint(id, “close”);
            uint256 bid = oracle.getResponseUint(id, “bid”);
            uint256 ask = oracle.getResponseUint(id, “ask”);
            uint256 timestamp = oracle.getResponseUint(id,
“timestamp”);
            oracle.deleteResponse(id);
            // Do something with the price data
        }
```

```
    function handleFailure(uint256 id) public {
        assert(msg.sender == address(oracle));
        bytes32 error = oracle.getResponseError(id);
        oracle.deleteResponse(id);
        emit LogError(error);
    }
```

使用 initRequest 函数初始化数据请求，除了两个回调函数之外，还允许指定查询类型（本示例中是对日内价格的请求）。 这将返回一个 uint256 标识符，然后可用于提供其他参数。 addArgumentToRequestString 函数用于指定 Reuters Instrument Code（RIC），此处为 IBM 库存，并且传递 addArgumentToRequestUint 允许要指定的时间戳。现在，传入 block.timestamp 的别名将检索 IBM 的当前价格。然后由 executeRequest 函数执行该请求。处理完请求后，oracle 合约将使用查询标识符调用 onSuccess 回调函数来判断结果；如果检索失败， onFailure 回调将返回错误代码。成功检索的可用字段包括 open，high，low，close（OHLC）以及 bid/ask 价格。

总结

如你所见，oracles 为智能合约提供了重要的服务：它将外部信息引入合同执行过程中。 因此，oracles 也会带来很大的风险，如果它们是受信任的并且被修改的信息源，oracles 可能导致智能合约的执行受到损害。

一般来说，在考虑使用 oracle 时要非常小心 trust model。 如果你认为 oracle 是可信的，那么你可能将其暴露给了潜在的错误输入，而这个错误输入会破坏智能合约的安全性。 换一个角度看，如果所有的安全前提都考虑到的话的话，oracles 会非常有用。

分散式 oracles 可以解决其中一些问题，并为以太坊智能合约提供无信任的外部数据。 仔细选择后，你可以开始探索以太坊与 oracles 提供的“真实世界”之间的桥梁。

第十二章 分布式应用

分布式应用

在本章中，我们将探讨分布式应用或 DApps 的世界。从以太坊的早期开始，创始人的愿景远比“智能合约”更广阔：不亚于重新创建网络并创建一个新的 DApp 世界，称为 web3 恰如其分。智能合约是分布式应用程序的控制逻辑和支付功能的一种方式。Web3 DApps 是关于分布式应用程序的其他所有方面：存储，消息传递，命名等（参见 Web3：使用智能合约和 P2P 技术的分布式 Web）

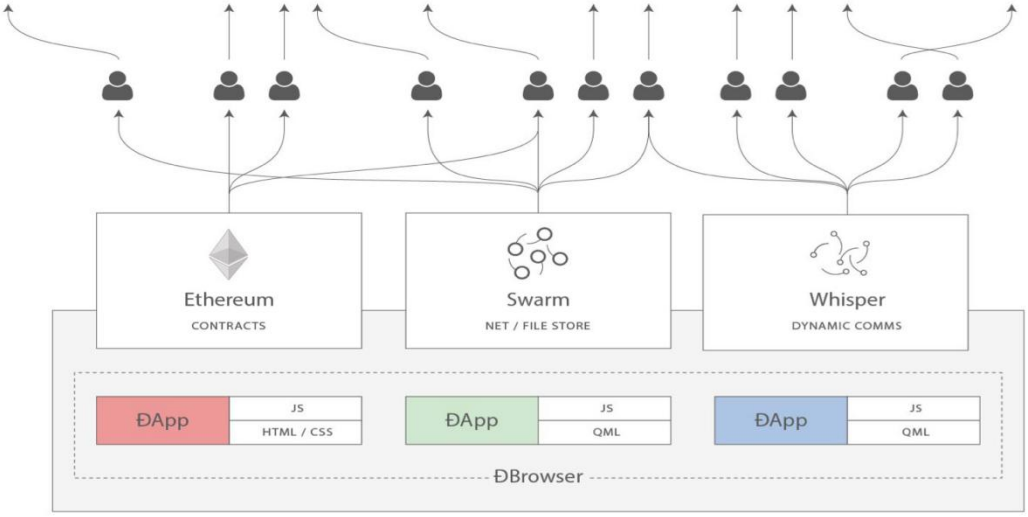


图 1. Web3：使用智能合约和 P2P 技术的分布式 Web

Web3：使用智能合约和 P2P 技术的分布式 Web

虽然“分布式应用程序”是对未来的大胆愿景，但“DApp”一词通常适用于任何具有 Web 前端的智能合约。其中一些所谓的 DApps 是高度集中的应用程序（CApps？）。小心虚假的 DApps！在本章中，我们将开发和部署一个样本 DApp：一个拍卖平台。您可以在 code/auction_dapp 下的图书库中找到源代码。我们将查看拍卖应用程序的每个方面，并了解我们如何尽可能地分散应用程序。首先，让我们仔细研究一下 DApps 的定义特征和优势。

什么是 DApp？

DApp 是一个大部分或完全分散的应用程序。 考虑一个应用程序能分散的所有可能的方面：后端软件（应用程序逻辑），前端软件，数据存储，消息通信，名称解析。

这些中的每一个都可以在某种程度上集中或稍微分散。例如，可以将前端开发为在中央服务器上运行的 Web 应用程序，或者作为在您的设备上运行的移动应用程序。后端和存储可以位于私有服务器和专有数据库上，也可以使用智能合约和 P2P 存储。 创建的 DApp 有很多典型集中式架构无法提供的优点： 弹性：由于业务逻辑由智能合约控制，因此 DApp 后端将在区块链平台上完全分发和管理。与部署在集中式服务器上的应用程序不同，DApp 将没有停机时间，只要平台仍在运行，它就会继续可用。 透明度：DApp 的链上特性允许每个人检查代码并更加确定其功能。与 DApp 的任何交互都将永久存储在区块链中。 审查阻力：只要用户可以访问以太坊节点（必要时运行一个），用户将始终能够与 DApp 交互而不受任何集中控制的干扰。在网络上部署代码后，任何服务提供商，甚至智能合约的所有者都无法更改代码。

在今天的以太坊生态系统中，很少有真正分散的应用程序 - 大多数仍然依赖于集中服务和服务器来完成其部分操作。在未来，我们希望任何 DApp 的每个部分都可以以完全分散的方式运行。

后端（智能合约）

在 DApp 中，智能合约用于存储业务逻辑（程序代码）和应用程序的相关状态。您可以认为在常规应用程序中替换服务器端（也称为“后端”）组件的智能合约。当然，这是一种过于简单化的做法。其中一个主要区别是智能合约中执行的任何计算都非常昂贵，因此应尽可能保持最小化。因此，确定应用程序的哪些方面需要可信、分散的执行平台非常重要。 以太坊智能合约允许您构建架构，其中智能合约网络在彼此之间调用和传递数据，随时读取和写入自己的状态变量，其复杂性仅区块 gas limit 的限制。部署智能合约后，未来许多其他开发人员都可以使用您的业务逻辑。 智能合约架构设计的一个主要考虑因素是部署后无法更改智能合约的代码。如果使用可访问的 SELFDESTRUCT 操作码对其进行编程，则可以将其删除，但除完全删除外，不能以任何方式更改代码。 智能合约架构设计的第二个主要考虑因素是 DApp 大小。一个非常庞大的单片智能合约可能会花费大量的 gas 来部署和使用。因此，某些应用程序可能会选择进行离线计算和外部数据源。但请记住，让 DApp 的核心业务逻辑依赖于外部数据（例如，来自中央服务器）意味着您的用户必须信任这些外部资源。

前端（Web 用户界面）

与 DApp 的业务逻辑不同，DApp 需要开发人员理解 EVM 和新语言（如 Solidity），DApp 的客户端接口可以使用标准 Web 技术（HTML，CSS，JavaScript 等）。这允许传统的 Web 开发人员使用熟悉的工具，库和框架。与以太坊的交互，例如签名消息，发送交易和管理密钥，通常通过网络浏览器，通过 MetaMask 等扩展进行（参见[intro_chapter]）。 虽然也可以创建移动 DApp，但目前很少有资源可以帮助创建移动 DApp 前端，这主要是因为缺少可以作为具有密钥管理功能的轻客户端的移动客户端。 前端通常通过 web3.js JavaScript 库链接到以太坊，该库与前端资源捆绑在一起，并由 Web 服务器提供给浏览器。

数据存储

由于高昂的 gas 成本和目前较低的区块 gas limit，智能合约不太适合存储或处理大量数据。因此，大多数 DApps 利用离线数据存储服务，这意味着它们将数据存储平台上的庞大数据存储在以太坊链中。该数据存储平台可以是集中的（例如，典型的云数据库），或者数据可以分散，存储在诸如 IPFS 的 P2P 平台或者以太坊自己的 Swarm 平台上。 分散式 P2P 存储非常适合存储和分发大型静态资产，如图像，视频和应用程序前端 Web 界面（HTML，CSS，JavaScript 等）的资源。我们接下来会看几个选项。 星际文件系统（IPFS）是一种分散的内容可寻址存储系统，它在 P2P 网络中的对等体之间分配存储的对象。 “内容可寻址”意味着对每条内容（文件）进行散列并使用散列来标识该文件。然后，您可以通过其哈希请求从任何 IPFS 节点检索任何文件。 IPFS 旨在取代 HTTP 作为交付 Web 应用程序的首选协议。这些文件不是在单个服务器上存储 Web 应用程序，而是存储在 IPFS 上，可以从任何 IPFS 节点检索。 有关 IPFS 的更多信息，请访问 https://ipfs.io。Swarm 是另一种内容可寻址的 P2P 存储系统，类似于 IPFS。 Swarm 由以太坊基金会创建，作为 Go-Ethereum 工具套件的一部分。与 IPFS 一样，它允许您存储由 Swarm 节点传播和复制的文件。您可以通过哈希引用任何 Swarm 文件来访问它。 Swarm 允许您从分散的 P2P 系统访问网站，而不是中央 Web 服务器。 Swarm 的主页本身存储在 Swarm 上，可以在 Swarm 节点或网关上访问：https：//swarm-gateways.net/bzz：/theswarm.eth/。

分散的消息通信协议

任何应用程序的另一个主要组件是进程间通信。这意味着能够在应用程序之间，应用程序的不同实例之间或应用程序的用户之间交换消息。传统上，这是通

过依赖中央服务器来实现的。但是，基于服务器的协议有各种分散的替代方案，通过 P2P 网络提供消息传递。最值得注意的 DApps P2P 消息传递协议是 Whisper，它是以太坊基金会 Go-Ethereum 工具套件的一部分。可以分散的应用程序的最后一方面是名称解析。 我们将在本章后面仔细研究以太坊的名称服务;现在，让我们深入研究一个例子。

基本 DApp 示例：拍卖 DApp

在本节中，我们将开始构建一个示例 DApp，以探索各种分散工具。我们的 DApp 将实施分散式拍卖。 拍卖 DApp 允许用户注册“契约”token，该 token 代表一些独特的资产，例如房屋，汽车，商标等。一旦 token 被注册，token 的所有权将转移到拍卖 DApp，允许它上市销售。 Auction DApp 列出了每个注册的 token，允许其他用户进行出价。在每次拍卖期间，用户可以加入专门为该拍卖创建的聊天室。拍卖完成后，契约 token 的所有权将转移给拍卖的获胜者。整个拍卖过程可以在 Auction DApp 中看到：一个简单的拍卖 DApp 示例。 拍卖 DApp 的主要组成部分是： 实施 ERC721 不可替代“契约”token 的智能合约（DeedRepository）； 实施拍卖（AuctionRepository）以出售契约的智能合约； 使用 Vue / Vuetify JavaScript 框架的 Web 前端； web3.js 库连接到以太坊链（通过 MetaMask 或其他客户端）； 一个 Swarm 客户端，用于存储图像等资源； 一个 Whisper 客户端，为所有参与者创建每个拍卖聊天室。

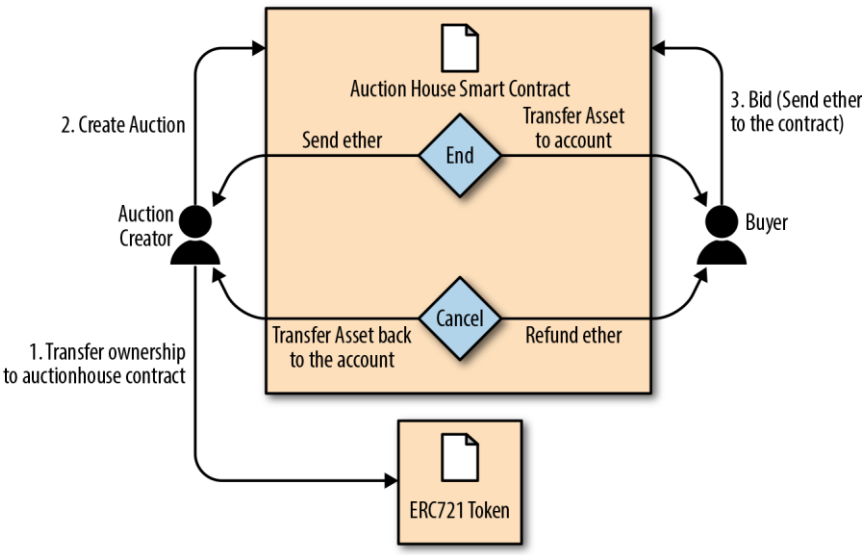


图 2.拍卖 DApp：一个简单的拍卖 DApp 示例 您可以在图书库中找到拍卖 DApp 的源代码。

拍卖 DApp：后端智能合约

我们的 Auction DApp 示例由两个智能合约支持，我们需要在以太坊区块链上部署它们以支持应用程序：AuctionRepository 和 DeedRepository。 让我们首先看看 DeedRepository 中显示的 DeedRepository：一个用于拍卖的 ERC721 契约令牌。该合同是与 ERC721 兼容的不可互换的令牌（参见[erc721]）。

```
link:code/auction_dapp/backend/contracts/DeedRepository.sol[]
```

示例 1. DeedRepository.sol：用于拍卖的 ERC721 契约令牌

如您所见，DeedRepository 契约是 ERC721 兼容令牌的直接实现。 我们的拍卖 DApp 使用 DeedRepository 合同为每次拍卖发放和跟踪令牌。拍卖本身由 AuctionRepository 合同编排。这个合同太长了，不能完全包含在这里，但 AuctionRepository.sol：主要的 Auction DApp 智能合约显示了合同和数据结构的主要定义。整个合同可以在本书的 GitHub 存储库中找到。

```
contract AuctionRepository {

    // Array with all auctions
    Auction[] public auctions;

    // Mapping from auction index to user bids
    mapping(uint256 => Bid[]) public auctionBids;

    // Mapping from owner to a list of owned auctions
    mapping(address => uint[]) public auctionOwner;

    // Bid struct to hold bidder and amount
    struct Bid {
        address from;
        uint256 amount;
    }

    // Auction struct which holds all the required info
    struct Auction {
        string name;
        uint256 blockDeadline;
        uint256 startPrice;
        string metadata;
        uint256 deedId;
        address deedRepositoryAddress;
        address owner;
        bool active;
        bool finalized;
    }
```

示例 2. AuctionRepository.sol：主要的 Auction DApp 智能合约

AuctionRepository 合约使用以下功能管理所有拍卖：您可以使用书籍存储库中的松露将这些合同部署到您选择的以太坊区块链中（例如，Ropsten）。

DApp 治理

如果你仔细阅读 Auction DApp 的两个智能合约，你会注意到一些重要的事情：没有特殊的帐户或角色对 DApp 有特殊的权限。每个拍卖都有一个具有一些特殊功能的所有者，但 Auction DApp 本身没有特权用户。 这是分散 DApp 治理并在部署后放弃任何控制的有意选择。相比之下，一些 DApp 具有一个或多个具有特殊功能的特权帐户，例如终止 DApp 合同，覆盖或更改其配置或“否决”某些操作的能力。通常，这些治理功能在 DApp 中引入，以避免由于错误而可能出现的未知问题。 治理问题是一个特别难以解决的问题，因为它代表了一把双刃剑。一方面，特权账户是危险的;如果受到损害，他们可以破坏 DApp 的安全性。另一方面，没有任何特权帐户，如果发现错误，则没有恢复选项。我们已经看到这些风险在以太坊 DApps 中都有所体现。在 DAO（[real_world_example_the_dao]和[ethereum_fork_history]）的情况下，有一些特权帐户称为“策展人”，但他们的能力非常有限。这些帐户无法覆盖 DAO 攻击者撤回资金。在最近的一个案例中，分散的交易所 Bancor 经历了大规模的盗窃，因为特权管理账户遭到了破坏。事实证明，Bancor 没有像最初假设的那样分散。 在构建 DApp 时，您必须决定是否要使智能合约真正独立，启动它们然后无法控制，或者创建特权帐户并冒险受到攻击。这两种选择都存在风险，但从长远来看，真正的 DApps 无法对特权帐户进行专门访问 - 这种权限不是分散的。

拍卖 DApp：前端用户界面

一旦部署了 Auction DApp 的合同，您就可以使用自己喜欢的 JavaScript 控制台和 web3.js 或其他 web3 库与它们进行交互。但是，大多数用户需要易于使用的界面。我们的 Auction DApp 用户界面是使用 Google 的 Vue2 / Vuetify JavaScript 框架构建的。 您可以在本书的存储库的 code / auction_dapp / frontend 文件夹中找到用户界面代码。该目录具有以下结构和内容： 部署合同后，编辑 frontend / src / config.js 中的前端配置，并在部署时输入 DeedRepository 和 AuctionRepository 合同的地址。前端应用程序还需要访问提供 JSON-RPC 和 WebSockets 接口的以太坊节点。配置完前端后，在本地计算机上使用 Web 服务器启动它： Auction DApp 前端将启动，可通过 http：// localhost：8080 上的任何 Web 浏览器访问。 如果一切顺利，您应该会看到 Auction DApp 用户界面中显示的屏幕，其中显示了在 Web 浏览器中运行的 Auction DApp。

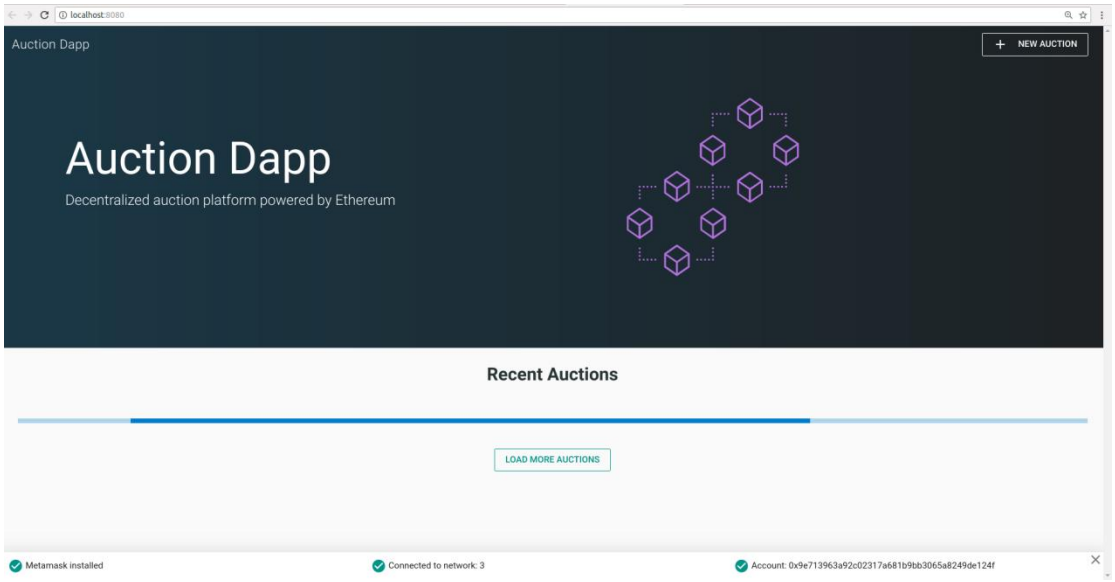


图 3. Auction DApp 用户界面

进一步下放拍卖 DApp 我们的 DApp 已经非常分散，但我们可以改进。AuctionRepository 合同独立于任何监督，对任何人开放。一旦部署，它就无法停止，也不能控制任何拍卖。每次拍卖都有一个单独的聊天室，允许任何人在没有审查或识别的情况下就拍卖进行沟通。各种拍卖资产（例如描述和相关图像）存储在 Swarm 上，使得它们难以进行审查或阻止。 任何人都可以通过手动构建事务或在本地计算机上运行 Vue 前端来与 DApp 交互。DApp 代码本身是开源的，并在公共存储库上协作开发。 我们可以做两件事来使这个 DApp 分散和弹性：将所有应用程序代码存储在 Swarm 或 IPFS 上。 使用以太坊名称服务通过引用名称访问 DApp。

我们将在下一节中探讨第一个选项，我们将深入研究以太坊名称服务（ENS）中的第二个选项。

在 Swarm 上存储拍卖 Dapp

我们在本章前面介绍了 Swarm in Swarm。我们的拍卖 DApp 已经使用 Swarm 存储每次拍卖的图标图像。这是一种比尝试在以太坊上存储数据更有效的解决方案，这是昂贵的。与将这些图像存储在 Web 服务器或文件服务器等集中式服务中相比，它也更具弹性。 但我们可以更进一步。我们可以将 DApp 本身的整个前端存储在 Swarm 中，并直接从 Swarm 节点运行它，而不是运行 Web 服务器。

准备 Swarm

首先，您需要安装 Swarm 并初始化 Swarm 节点。 Swarm 是以太坊基金会 Go-Ethereum 工具套件的一部分。请参阅[go_ethereum_geth]中安装 Go-Ethereum 的说明，或安装 Swarm 二进制版本，按照 Swarm 文档中的说明进行操作。 一旦安装了 Swarm，就可以通过使用 version 命令运行它来检查它是否正常工作： 要开始运行 Swarm，您必须告诉它如何连接到 Geth 实例，以访问 JSON-RPC API。按照“入门指南”中的说明开始使用。 当你启动 Swarm 时，你应该看到这样的东西： 您可以通过连接到本地 Swarm 网关 Web 界面来确认您的 Swarm 节点是否正常运行：http：// localhost：8500。 您应该看到类似于 localhost 上的 Swarm 网关中的屏幕，并且能够查询任何 Swarm 哈希或 ENS 名称。

图 4. localhost 上的 Swarm 网关

将文件上传到 Swarm 一旦你的本地 Swarm 节点和网关运行，你就可以上传到 Swarm，只需参考文件哈希就可以在任何 Swarm 节点上访问这些文件。 让我们通过上传文件来测试： Swarm 已上传 README.md 文件并返回一个哈希，您可以使用该哈希从任何 Swarm 节点访问该文件。例如，您可以使用公共 Swarm 网关。 虽然上传一个文件相对简单，但上传整个 DApp 前端会有点复杂。这是因为各种 DApp 资源（HTML，CSS，JavaScript，库等）都嵌入了彼此的引用。通常，Web 服务器将 URL 转换为本地文件并提供正确的资源。我们可以通过打包我们的 DApp 来实现 Swarm 的相同功能。

在拍卖 DApp 中，有一个用于打包所有资源的脚本： 这个命令的结果将是一个新目录，代码/ auction_dapp / frontend / dist，它包含整个 Auction DApp 前端，打包在一起： 现在，您可以使用 up 命令和–recursive 选项将整个 DApp 上传到 Swarm。在这里，我们还告诉 Swarm index.html 是加载此 DApp 的默认路径： 现在，我们的整个 Auction DApp 都托管在 Swarm 上，并可通过 Swarm URL 访问：BZZ：// ab164cf37dc10647e43a233486cdeffa8334

b026e32a480dd9cbd020c12d4581 我们在分散 DApp 方面取得了一些进展，但我们使用起来更加困难。像这样的 URL 比像 auction_dapp.com 这样漂亮的名字更不方便用户使用。为了获得权力下放，我们是否被迫牺牲可用性？不必要。在下一节中，我们将研究以太坊的名称服务，它允许我们使用易于阅读的名称，但仍然保留了我们应用程序的分散性。

以太坊名称服务（ENS）

您可以设计世界上最好的智能合约，但如果您没有为用户提供良好的界面，他们将无法访问它。 在传统的互联网上，域名系统（DNS）允许我们在浏览器中使用人类可读的名称，同时将这些名称解析为 IP 地址或幕后的其他标识符。在以太坊区块链上，以太坊命名系统（ENS）以分散的方式解决了同样的问题。例如，以太坊基金会捐赠地址为 0xfB6916095ca1df60 & thinsp; bB79Ce92cE3Ea74c37c5d359;在支持 ENS 的钱包中，它只是以太坊.eth。 ENS 不仅仅是一份智能合约;它是 DApp 的基础，提供分散的名称服务。此外，ENS 由许多 DApps 支持，用于注册，管理和注册名称的拍卖。 ENS 演示了 DApps 如何协同工作：它是为其他 DApp 服务的 DApp，由 DApps 生态系统支持，嵌入在其他 DApp 中，等等。

在本节中，我们将了解 ENS 的工作原理。我们将演示如何设置自己的名称并将其链接到钱包或以太坊地址，如何将 ENS 嵌入到另一个 DApp 中，以及如何使用 ENS 命名 DApp 资源以使其更易于使用。

以太坊名称服务的历史

名称注册是区块链的第一个非货币应用程序，由 Namecoin 开创。以太坊白皮书给出了一个两行的 Namecoin 注册系统作为其示例应用程序之一。 Geth 和 C ++以太网客户端的早期版本有一个内置的 namereg 合约（不再使用），并且提供了许多用于名称服务的提议和 ERC，但只有当 Nick Johnson 在 2016 年开始为以太坊基金会工作时才在他的支持下，该项目开始了对注册商的认真工作。ENS 于 2017 年 5 月 4 日星球大战日发布(在 3 月 15 日 Pi 日试图推出之后失败)。

ENS 规范

ENS 主要在三个以太坊改进提案中指定：EIP-137，其规定了 ENS 的基本功能; EIP-162，描述了.eth 根的拍卖系统;和 EIP-181，它规定了地址的反向登

记。 ENS 遵循“三明治”设计理念：底部是一个非常简单的层，其次是更复杂但可替换的代码层，顶层非常简单，可以将所有资金保存在不同的帐户中。

底层：名称所有者和解析器

ENS 操作“节点”而不是人类可读的名称：使用“Namehash”算法将人类可读的名称转换为节点。 ENS 的基础层是 ERC137 定义的一个非常简单的合同（少于 50 行代码），它只允许节点的所有者设置有关其名称的信息并创建子节点（ENS 等效于 DNS 子域）。 基础层上的唯一功能是使节点所有者能够设置有关其自身节点的信息（特别是解析器，生存时间或转移所有权）以及创建新子节点的所有者。

Namehash 算法

Namehash 是一种递归算法，可以将任何名称转换为标识名称的哈希。“递归”意味着我们通过解决同一类型的较小问题的子问题来解决问题，然后使用子问题的解来解决原始问题。 Namehash 以递归方式散列名称的组件，为任何有效的输入域生成唯一的固定长度字符串（或“节点”）。例如，subdomain.example.eth 的 Namehash 节点是 keccak（'<example.eth>'node）+ keccak（'<subdomain>'）。我们必须解决的子问题是计算 example.eth 的节点，即 keccak（'<。eth>'节点）+ keccak（'<example>'）。首先，我们必须计算 eth 的节点，即 keccak（<root node>）+ keccak（'<eth>'）。 根节点就是我们称之为递归的“基本情况”，我们显然无法递归地定义它，或者算法永远不会终止！根节点定义为`0x0000000000000000000000000000000000000000000000000000000000000000` (32 zero bytes). 把这些都放在一起，subdomain.example.eth 的节点因此是 keccak（keccak（0xcc … 0 + keccak（'eth'））+ keccak（'example'））+ keccak（'subdomain'）） 。 通用化，我们可以按如下方式定义 Namehash 函数（根节点的基本情况，或空名称，后跟递归步骤）：

在 Python 中，这变为： 因此，mastering-ethereum.eth 将按如下方式处理： 当然，子域本身可以有子域：subdomain.example.eth 之后可以有 sub.subdomain.example.eth，然后是 sub.sub.subdomain.example.eth，依此类推。为了避免昂贵的重新计算，由于 Namehash 仅依赖于名称本身，因此可以预先计算给定名称的节点并将其插入到合同中，从而无需进行字符串操作并允许立即查找 ENS 记录，而不管其中的组件数量是多少。原始名称。

如何选择有效的名称

名称由一系列点分隔标签组成。虽然允许使用大写和小写字母，但所有标签都应遵循 UTS＃46 规范化过程，在对其进行散列之前对其进行大小写折叠，因此具有不同大小但拼写相同的名称最终将使用相同的 Namehash。 您可以使用任何长度的标签和域，但为了与传统 DNS 兼容，建议使用以下规则： 标签每个不应超过 64 个字符。 完整的 ENS 名称不应超过 255 个字符。 标签不应以连字符开头或结尾，也不应以数字开头。

根节点所有权

这种分层系统的结果之一是它依赖于根节点的所有者，他们能够创建顶级域（TLD）。虽然最终的目标是为新 TLD 采用分散的决策流程，但在撰写本文时，根节点由 7 个 4 的多重控制控制，由不同国家的人们持有（构建为 7 的反映）DNS 系统的关键人员）。因此，7 个关键持有人中至少有 4 个人中的大多数都需要进行任何变更。 目前，这些关键人物的目的和目标是与社区达成共识： 在评估系统后，将.eth TLD 的临时所有权迁移并升级为更长久的合同。 如果社群同意需要，则允许添加新 TLD。 在同意，测试和实施此类系统时，将根 multisig 的所有权迁移到更分散的合同。 作为处理顶级注册表中的任何错误或漏洞的最后手段。

解析器

基本的 ENS 合同无法向名称添加元数据;这就是所谓的“解析合同”的工作。这些是用户创建的合同，可以回答有关名称的问题，例如 Swarm 地址与应用程序关联的内容，接收应用程序付款的地址（以太币或代币），或应用程序的哈希值（验证）它的完整性）。

中间层：.eth 节点 在撰写本文时，唯一可以在智能合约中注册的顶级域名是.eth。

注意

目前正在努力使传统的 DNS 域名所有者能够获得 ENS 所有权。虽然理论上这可能适用于.com，但到目前为止，唯一实现此目的的域是.xyz，并且仅在 Ropsten testnet 上。 .eth 域名通过拍卖系统分发。没有保留列表或优先级，获取名称的唯一方法是使用系统。拍卖系统是一段复杂的代码（超过 500 行）;

ENS 中的大部分早期开发工作（和错误！）都在系统的这一部分。然而，它也是可替换和可升级的，没有资金的风险 - 更多的是后来。

Vickrey 拍卖

名称通过修改后的 Vickrey 拍卖分发。在传统的 Vickrey 拍卖中，每个投标人都提交密封投标，并且所有投标人同时被公开，此时最高出价者赢得拍卖，但只支付第二高的出价。因此，竞标者被激励不要向他们出价低于名称的真实价值，因为竞标他们的真实价值会增加他们赢得的机会，但不会影响他们最终支付的价格。 在区块链上，需要进行一些更改： 为确保投标人不提交投标，他们无意支付，他们必须事先锁定等于或高于其投标价值，以保证投标有效。 由于您无法隐藏区块链上的机密信息，因此投标人必须至少执行两项交易（提交揭示流程），以隐藏其出价的原始值和名称。 由于您无法在分散系统中同时显示所有出价，因此投标人必须自行展示自己的出价;如果他们不这样做，他们就会丧失锁定的资金。如果没有这种没收，人们可以做出很多出价，并选择只披露一两个，将密封拍卖变成传统的增加价格拍卖。

因此，拍卖分为四个步骤：

1. 开始拍卖。这是广播注册名称的意图所必需的。这将创建所有拍卖截止日期。这些名称是经过哈希处理的，因此只有在字典中具有该名称的人才能知道哪个拍卖被打开了。这允许一些隐私，这在您创建新项目并且不想共享其详细信息时非常有用。您可以同时打开多个虚拟拍卖，因此如果有人关注您，他们就不能简单地对您打开的所有拍卖进行出价。

2. 密封投标。您必须在投标截止日期之前执行此操作，方法是将一定数量的以太币与秘密消息的哈希值相关联（其中包括名称的哈希值，实际的出价金额和盐值）。您可以锁定比您实际出价更多的以太，以掩盖您的真实估值。

3. 显示出价。在显示期间，您必须进行一次显示出价的交易，然后计算最高出价和第二高出价，并将以太送回不成功的出价人。每次出价都会重新计算当前的赢家;因此，在揭示截止日期到期之前设置的最后一个成为总冠军。

4. 之后清理。如果您是赢家，则可以最终确定竞价，以便取消您的出价与第二高出价之间的差额。如果您忘记透露，您可以进行延迟披露并收回一些出价。

顶层：契约

ENS 的最高层是另一个超级简单的合同，只有一个目的：持有资金。 当您赢得一个名字时，资金实际上并没有被发送到任何地方，而是在您想要保留名称的期间（至少一年）被锁定。这有点像保证回购：如果所有者不再需要该名称，他们可以将其卖回系统并恢复他们的以太（因此持有该名称的成本是做出回报大于零的事情的机会成本） ）。 当然，单一合同持有数百万美元的以太币已被证明是非常危险的，因此 ENS 为每个新名称创建契约。契约合约非常简单（大约 50 行代码），它只允许将资金转回单个账户（契约所有者）并由单个实体（注册商合同）调用。这种方法大大减少了攻击面，因为错误可能会使资金面临风险。

注册名称

正如我们在 Vickrey 拍卖会上看到的那样，在 ENS 中注册名称是一个分为四个步骤的过程。首先我们对任何可用的名称进行出价，然后我们会在 48 小时后显示我们的出价以确保名称。 ENS 注册时间表是显示注册时间表的图表。 让我们注册我们的名字！ 我们将使用几个可用的用户友好界面中的一个来搜索可用的名称，对名称 ethereumbook.eth 进行出价，显示出价并保护名称。 ENS 有许多基于 Web 的界面，允许我们与 ENS DApp 进行交互。对于这个例子，我们将使用 MyCrypto 接口和 MetaMask 作为我们的钱包。 图 5.注册的 ENS 时间表

首先，我们需要确保我们想要的名称可用。在写这本书时，我们真的想注册名称 mastering.eth，但是，在 MyCrypto.com 上搜索 ENS 名称显示它已经被采用了！由于 ENS 注册仅持续一年，因此将来可能可以保护该名称。在此期间，让我们搜索 ethereumbook.eth（在 MyCrypto.com 上搜索 ENS 名称）。 图 6.在 MyCrypto.com 上搜索 ENS 名称 太棒了！这个名字可用。为了注册它，我们需要继续开始拍卖 ENS 名称。让我们解锁 MetaMask 并开始为 ethereumbook.eth 拍卖。 图 7.开始 ENS 名称的拍卖 让我们出价。为此，我们需要按照放置 ENS 名称的出价中的步骤进行操作。 图 8.为 ENS 名称设置出价

警告

如 Vickrey 拍卖中所述，您必须在拍卖完成后 48 小时内公布您的出价，否则您将失去投标中的资金。我们忘记这样做并且自己减掉 0.01 ETH 吗？你打赌我们做到了。 截取屏幕截图，保存您的密码（作为出价的备份），并在日历中添加提醒以显示日期和时间，这样您就不会忘记并丢失资金。

第十三章 以太坊虚拟机(EVM)

以太坊虚拟机(EVM)

以太坊协议和运行系统的核心是以太坊虚拟机，简称 EVM。 顾名思义，它是一个运算引擎，与微软的.net 框架的虚拟机或其他字节代码编译的变成语言（如 java）没有很大的不同。在本章中，我们将在以太坊状态更新的内容中详细介绍 EVM，包括其指令集，结构和操作系统。

什么是 EVM?

EVM 是以太坊的一部分，负责智能合约的部署和执行。实际上，从一个 EOA 到另一个 EOA 的简单价值转移交易不需要涉及它，但其他所有交易都将涉及到由 EVM 计算的状态更新。从高级别来看，运行在以太坊区块链上的 EVM 可以被认为是一台包含数百万个可执行对象的全球分散式计算机，每个对象都具备自身的永久数据存储。

EVM 是一个类似图灵完备状态机，“类似”是因为被任何给定智能合约的 gas 可用量限定了，因此所有执行过程都被限制在有限数量的计算步骤内。因此，中断问题是“解决的”（所有进程都将停止），并且避免了执行程序可能（意外或者恶意）永久运行，从而使以太坊平台完全停止的情况。

EVM 有基于堆栈的体系结构，并将所有的内存值存储在堆栈上。它的字节大小是 256 位（主要是为了方便本地散列和椭圆曲线操作），并具有几个可寻址的数据组件：

1. 一种不可改变的程序代码 ROM，与要执行的智能合约的字节码一起加载。

2．易失存储器，每个位置都显式初始化为零。
3．作为以太坊状态的一部分的永久存储器，其值也被初始化为零。

还有一组在执行期间可用的环境变量和数据。我们将在本章后面更加详细地讨论这些问题。

以太坊虚拟机（EVM）架构和执行的文本显示 EVM 架构和执行的内容。

与现存技术的比较

术语“虚拟机”通常应用于实际计算机的虚拟化，通常由“管理程序”如 VirtualBox 或 QEMU，或整个操作系统实例如 Linux 的 KVM 实现。它们必须分别提供实际硬件，系统调用和其它内核功能的软件抽象。

EVM 在一个更有限的领域中运行： 它只是一个计算引擎，因此提供了一个抽象的计算和存储，类似于 Java 虚拟机（JVM）规范。从高级别角度来看，JVM 旨在提供一个与底层主机操作系统或者硬件无关的运行环境，从而实现各种跨系统的兼容性。高级编程语言如 Java 或 Scala（使用 JVM）或 C#（使用.NET）被编译成它们各自虚拟机的字节码指令集。以同样的方式，EVM 执行自己的字节码指令集(在下一节中描述)，这些指令集被编译成更高级的智能合约编程语言，如 LLL，Serpent,Mutan,或者 Solidity。

因此，EVM 没有调度能力，因为执行顺序是外部组织的，以太坊客户端通过运行经过验证的区块交易来确定哪些智能合约需要执行或者按照什么顺序执行。从这点上来说，以太坊世界中的计算机是单线程的，就像 JavaScript 一样。EVM 也没有任何“系统接口”处理或者“硬件支持”——没有实际的机器可以与之相接。以太坊世界的计算机是完全虚拟的。

EVM 指令集（字节码运算）

EVM 指令集提供了您可能期望的大多数操作，包括：
1.算术和位逻辑运算
2.执行内容查询
3.堆栈、内存和存储访问
4.控制流操作
5.日志记录、调用和其他算子

除了典型的字节码操作之外，EVM 还可以访问账户信息（如地址和余额）和区块信息（如区块编号和当前 gas 价格。）

让我们从可用的操作码和他们的作用开始更加详细的探索 EVM。如您可能所预料到的，所有操作数都是从堆栈中获取的，结果（适用的结果）通常放回堆栈顶部。

提示：完整的操作代码列表及其相应的 gas 成本可在[evm_opcodes]中被找到。

可用操作码可被分为如下类别：

算术运算

```
算数运算代码指令：
ADD         //添加前两个堆栈项
MUL         //将前两个堆栈项相乘
SUB         //减去前两个堆栈项
DIV         //整数除法
SDIV        //有符号整数除法
MOD         //模（余数）运算
SMOD        //有符号模运算
ADDMOD      //任意数的加法模
MULMOD      //任意数的乘法模
EXP         //指数运算
SIGNEXTEND  //扩展两个有符号补整数的长度
SHA3        //计算内存块的 keccak-256 哈希值
```

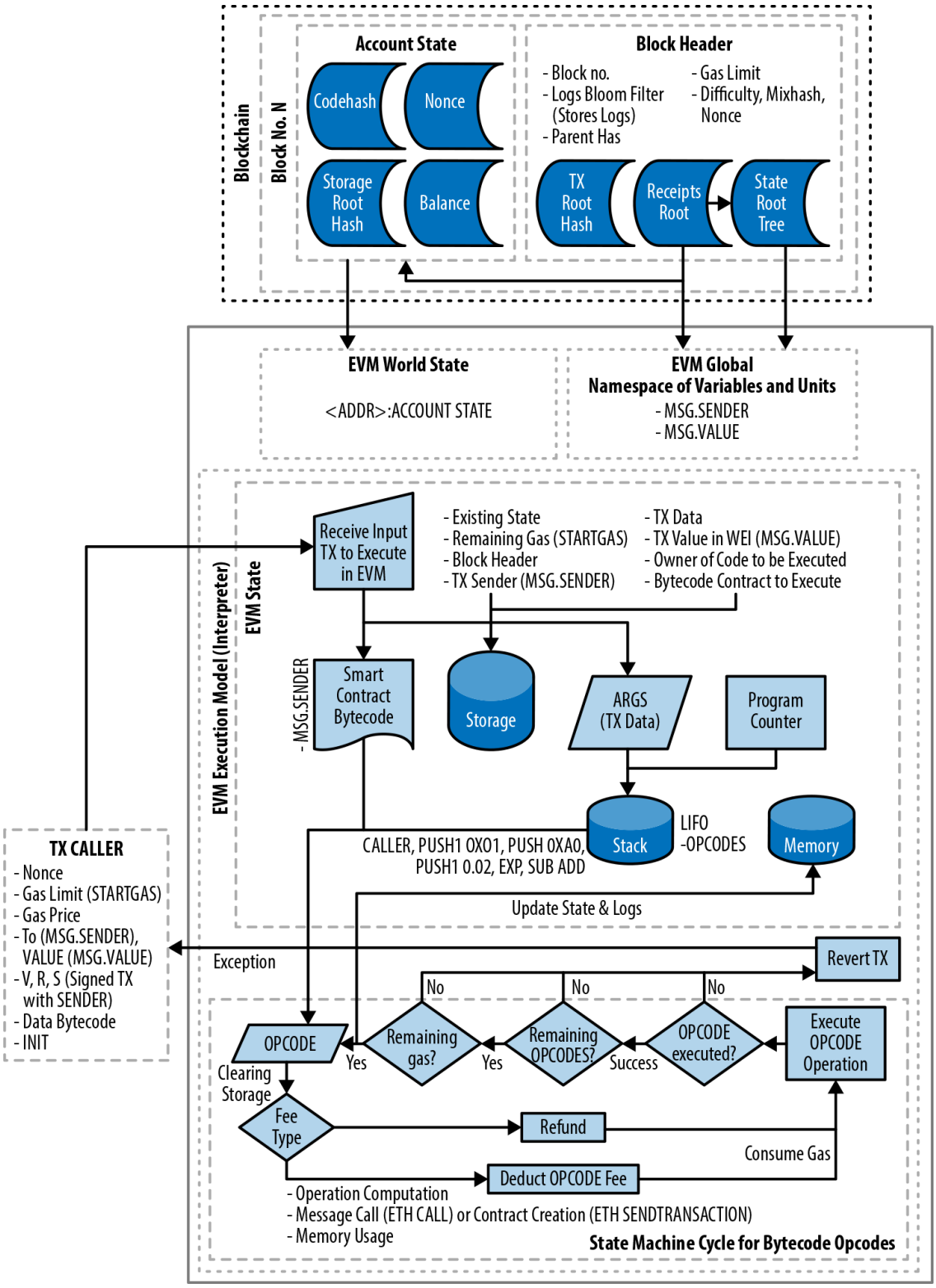


图 1：以太坊虚拟机（EVM）架构和执行的相关文本

要注意的是所有的算数运算都是以 2256 为模（除非另外说明），零的零次幂取 1。

堆栈运算

堆栈，内存和存储管理指令：

```
POP      //从堆栈中移除顶部项
MLOAD    //从内存中加载一个单词
MSTORE   //将单词保存在内存中
MSTORE8  //将字节保存到存储空间
SLOAD    //从存储空间中加载一个单词
SSTORE   //将单词保存到存储空间
MSIZE    //获取可用内存的大小（以字节为单位）
PUSHx    //将 x 字节放在堆栈上，其中 x 可以是
         // 1 到 32 的整数（含 1 和 32）
DUPx     //复制第 X 个堆叠项，其中 X 可以是来自
         // 1 到 16 的整数（含 1 和 16）
SWAPx    //交换第 1 个和（x+1）个堆栈项，其中 X 可以是任何
         // 1 到 16 之间的整数（含 1 和 16）
```

工艺流程运算

控制流程指令：

```
STOP       //停止执行
JUMP       //将程序计数器设置为任意值
JUMPI      //有条件的改变程序计数器
PC         //获取程序计数器的值（在递增之前
           //对应于此指令)
JUMPDEST   //标记有效的跳转目标
```

系统运算

执行程序的系统操作指令：

```
LOGx           //附加一个带有 x 主题的日志记录，其中 x 是任意整数。
               //从 0 到 4（含 0 和 4）
CREATE          //创建带有关联代码的新账户
CALL           //消息呼叫到另一个账户，即运行另一个账户
               //账户代码
CALLCODE        //用另一个账户调用此账户的信息
               //账户代码
RETURN          //停止执行并返回输出数据
DELEGATECALL   //用另一个账户调用此账户的信息
               //账户代码，但是为了发送方和数值
               //维持现有的值
```

```
STATICCALL    //静态消息调用账户
REVERT        //停止执行，回复状态更改但是返回数据和剩余 gas
INVALID       //指定的无效指令
SELFDESTRUCT  //停止执行并且注册要删除的账户
```

逻辑运算

```
用于比较和位逻辑的操作指令：
LT     //小于比较
GT     //大于比较
SLT    //有符号小于比较
SGT    //有符号大于比较
EQ     //等式比较
ISZERO //简单非运算符
AND    //位与运算
OR     //按位或运算
XOR    //位 xor 操作
NOT    //位非操作
BYTE   //从全宽 256 位字中检索单个字节
```

环境运算

```
处理执行环境信息的操作指令：256 位
GAS            //得到的可用 gas 量 (对于本条是在减少后)
ADDRESS        //获取当前执行账户的地址
BALANCE        //获取任何给定账户的账户余额
ORIGIN         //获取启动此 EVM 的 EOA 地址
               //执行
CALLER         //立即获取调用方的地址
               //执行
CALLVALUE      //获取调用方存入的以太金额
               //执行
CALLDATALOAD   //获取调用发发送的输入数据
               //执行
CALLDATASIZE   //获取输入数据的大小
CALLDATACOPY   //将输入数据复制到内存
CODESIZE       //获取当前环境中运行的代码大小
CODECOPY       //将当前环境中运行的代码复制到
```

```
                //内存
GASPRICE        //获取发送方指定的 gas 价格
                //交易
EXTCODESIZE     //获取任意账户的代码大小
EXTCODECOPY     //将任何账户的代码复制到内存中
RETURNDATASIZE //获取上一个调用的输出数据大小
                //在当前环境中
RETURNDATACOPY //将上一次调用的输出数据复制到内存
```

区块运算

```
访问当前区块信息的操作指令：
BLOCKHASH  //获取最近完成的 256 个哈希值之一
          //区块
COINBASE     //获取区块奖励的受益人地址
TIMESTAMP    //获取区块的时间戳
NUMBER      //获取区块的编号
DIFFICULTY    //获取区块的难度
GASLIMIT     //获取区块的 gas 限制值
```

以太坊状态

EVM 的作用是根据以太坊协议定义，通过计算智能合约代码执行后的有效状态转换来更新以太坊状态。这一层面导致了以太坊作为基于交易的状态机的描述，反映了外部参与者（即账户拥有者和矿工）通过创建、接受和订购交易来启动转换状态的事实。在这一点上，考虑什么构成以太坊状态是有用的。

在最高级别中，我们有以太坊世界状态。世界状态是以太坊地址（160 位值）到账户的映射。在较低的级别，每个以太坊地址代表一个包含以太余额的账户（存储为账户拥有的 Wei 数量）、nonce（如果这是 EOA，就代表从该账户成功发送的交易数量，如果这是合约账户，就代表被创建的合约数量）、该账户的存储（如果是一个永久性的数据存储，仅供智能合约使用），以及该账户的程序代码（同样地，仅当账户是只能合约账户时）。EOA 将始终没有代码和空存储。

当一笔交易导致只能合约代码执行时，EVM 将实例化，其中包含与正在创建的当前块和正在处理的特定交易相关的所有必需信息。尤其是，EVM 的程序代码 ROM 与正在被调用的合约账户的代码一起加载，程序计数器被设定为零，存储从合约账户的存储中加载，内存设置为全零，所有区块和环境变量都被设置。一个关键变量是执行此操作所需的 gas 供应量，它被设置为交易开始时发送方支付

的 gas 量（有关更多详细信息，请参考 Gas）。随着代码的执行，根据被执行的运算的 gas 量，gas 供应量减少。如果在任意时刻，Gas 供给量减少到零，我们会得到“gas 不足”的异常；此时执行过程立刻停止而且交易被废弃。不应用到对以太坊状态的任何更改，除非发送方的 nonce 正在增加，并且其以太坊余额下降以向区块受益人支付用于执行代码到停止点的资源。此时，您可以认为 EVM 运行在以太坊世界的沙盒副本上，如由于任何原因无法完成执行，则系统完全放弃此沙盒副本。但是，如果成功的执行完成，那么系统将更新实际状态以匹配沙盒版本，包括被调用的合约数据的各种更改、任何被创建的新合约以及任何被发起的以太坊余额传输。

需要注意的是，由于智能合约本身可以有效的启动交易，代码的执行是一个递归的处理过程。一个合约可以调用其他合约，每次调用都会导致围绕调用的新目标实例化另一个 EVM。每个实例都有其沙盒世界状态，从上级 EVM 的沙盒来启动新的执行过程。每一个实例也被赋予一个特定的 gas 供应量（当然，不超过上级的 gas 剩余量），因此，当由于 gas 太少而无法完成其执行时，可能会自行停止。同样地，在这种情况下，沙盒状态被丢弃，执行过程回到上级 EVM。

将 solidity 编译为 EVM 字节码

将 solidity 源文件编译为 EVM 字节码可以通过几种方法完成。在[简介章节]中，我们使用了在线 Remix 编译器。在本章中，我们将在命令行中使用 solc 可执行文件，有关选项列表，运行如下命令：

```
$ solc –help
```

使用-opcodes 命令行选项可以很容易的生成 solidity 源文件的原始操作码流。这个操作码遗漏了一些信息（asm 选项产生完整的信息），但是对于这个讨论来说已经足够了。例如，编译一个 solidity 示例文件 example.sol， 并将操作码输出发送到名为 BytecodeDir 的目录中，可以使用以下命令完成：

```
$ solc -o BytecodeDir --opcodes Example.sol
```

或者

```
$ solc -o BytecodeDir --asm Example.sol
```

下面的命令将为我们的示例程序生成字节码二进制：

```
$ solc -o BytecodeDir --bin Example.sol
```

生成的输出操作码文件将取决于 solidity 源文件中包含的特定合约。我们的简单 solidity 文件 Example.sol 只含有一个合约，名为 example：

```
pragma solidity ^0.4.19;
contract example {
  address contractOwner;
  function example() {
    contractOwner = msg.sender;
  }
}
```

如您所见，这个合约所做的就是保存一个持久状态变量，它被设置成运行这个合约的最后一个账户的地址。

如果您查看 BytecodeDir，您将看到 opcade 文件 example.opcode,其中包含示例合约的 EVM 操作码指令。在文本编辑器中打开 example.opcode，其中包含示例合约的 EVM 操作码指令。在文本编辑器中打开 example.opcode 文件将显示以下内容：

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH1 0xE JUMPI
PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH1 0x0 DUP1 PUSH2 0x100 EXP DUP2 SLOAD
DUP2 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF MUL NOT AND SWAP1
DUP4 PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND MUL OR SWAP1
SSTORE POP PUSH1
0x35 DUP1 PUSH1 0x5B PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP
PUSH1 0x60 PUSH1
0x40 MSTORE PUSH1 0x0 DUP1 REVERT STOP LOG1 PUSH6 0x627A7A723058
KECCAK256 JUMP
0xb9 SWAP14 0xcb 0x1e 0xdd RETURNDATACOPY 0xec 0xe0 0x1f 0x27
0xc9 PUSH5
0x9C5ABCC14A NUMBER 0x5e INVALID EXTCODESIZE 0xdb 0xcf
EXTCODESIZE 0x27
EXTCODESIZE 0xe2 0xb8 SWAP10 0xed 0x
```

使用-asm 选项编译示例会在字节标识符目录中生成一个名为 example.evm 的文件。这包含对 EVM 字节码指令的稍高级别描述，以及一些有帮助的注释：

```
/* “Example.sol”:26:132  contract example {... */
  mstore(0x40, 0x60)
    /* “Example.sol”:74:130  function example() {... */
  jumpi(tag_1, iszero(callvalue))
  0x0
  dup1
  Revert
tag_1:
    /* “Example.sol”:115:125  msg.sender */
  caller
    /* “Example.sol”:99:112  contractOwner */
  0x0
  dup1
    /* “Example.sol”:99:125  contractOwner = msg.sender */
  0x100
  exp
  dup2
  sload
  dup2
  0xffffffffffffffffffffffffffffffffffffffff
  mul
  not
  and
  swap1
  dup4
  0xffffffffffffffffffffffffffffffffffffffff
  and
  mul
  or
  swap1
  sstore
  pop
    /* “Example.sol”:26:132  contract example {... */
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x0
  codecopy
```

```
      0x0
      return
    stop
    sub_0: assembly {
           /* “Example.sol”:26:132  contract example {... */
          mstore(0x40, 0x60)
          0x0
          dup1
          revert
        auxdata:
0xa165627a7a7230582056b99dcb1edd3eece01f27c9649c5abcc14a435efe
3b...
    }
    --bin 运行时选项生成机器可读的十六进制字节码：
    60606040523415600e57600080fd5b336000806101000a81548173
    ffffffffffffffffffffffffffffffffffffffff
    021916908373
    ffffffffffffffffffffffffffffffffffffffff
    16021790555060358060 5b6000396000f300606060405260080fd00a16
5627a7a7230582056b...
```

您可以使用 EVM 指令集中给出的操作码列表（字节码运算）来详细了解这里的细节信息。但是，这是一项非常艰巨的任务，因此我们先从检查前四条开始：

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
```

这里我们可以看到 PUSH1，后面跟一个值 0x60 的原始字节。EVM 指令接受程序代码中操作码后面的单个字节（作为一个文字值），并将其推送至堆栈上。可以将大小不超过 32 个字节的值推送到堆栈上，如：

```
    PUSH32
0x436f6e67726174756c6174696f6e732120536f6f6e20746f206d61737465
7221
```

这里来自于 example.opcode 的第二个 PUSH1 操作码将 0x40 存储到堆栈顶部。（0x60 已经被呈现在下面的插槽。）

接下来是 MSTORE，它是一个内存存储运算，用于将值存于 EVM 的内存中。它接受两个参数，并且像大多数 EVM 运算一样，从堆栈获取它们。对于每个参数，堆栈是“弹出”的；即， 堆栈上的顶值被取下，堆栈上所有的其他值都向上移动一个位置。MSTORE 的第一个参数是内存中要被保存的值的语言地址。对于这个程序，我们在堆栈的顶部有 0x40，所以它被从堆栈中去除，并用做内存地址。第二个参数是要保存的值，这是是 0x60。在 MSTORE 运算被执行后，堆栈再次空置，但内存位置 0x40 处的值为 0x60（96，十进制）。

下一个操作码是 CALLVALUE，它是一个环境操作码，它将把发起这次执行的信息调用和被发送的以太数量（以 wei 为单位）推送到堆栈顶部。

我们可以继续以这种方式逐步推演这整个程序，直到我们完全理解这个代码所影响的低级状态更改，但是在这个阶段我们没必要做太深入的讨论，相关内容我们将在本章后面再讨论。

合约部署代码

在以太坊平台上创建和部署新合约时使用的代码与合约本身的代码之间存在重要但是微妙的差异。为了创建一个新的合约，我们需要一个特殊合约，并将 to 字段设置成为 0x0 的特殊地址，将数据字段设置合约的初始代码。处理此类合约创建的交易时，新合约账户的代码不是交易数据字段中的代码。相反，当像这样的智能合约被执行时，用加载到其程序代码 ROM 中的交易的数据字段来实例化 EVM，然后将执行该部署代码的输出作为新合约账户的代码。因此。新合约可以在部署时使用以太坊世界状态以编程方式初始化，在合约的存储中设置值，甚至发送以太坊或者创建更多的新合约。

当编译离线合约时，例如在命令行上使用 solc，您可以获取部署字节码或者运行时间字节码。

部署字节码用于新合约账户初始化的各个方面，包括当新交易调用此合约（即运行时的字节码）时将实际执行的字节码，以及基于合约的构造函数初始化所有内容的代码。

另一方面，运行时字节码恰恰是在调用新合约时被执行的字节码，而不包括在合约部署期间初始化合约所需的字节码。

我们以前面创建的简单 Faucet.sol 合约为例：

```
    // Version of Solidity compiler this program was written for
pragma solidity ^0.4.19;
    // Our first contract is a faucet!
    contract Faucet {
      // Give out ether to anyone who asks
      function withdraw(uint withdraw_amount) public {
          // Limit withdrawal amount
          require(withdraw_amount <= 100000000000000000);
          // Send the amount to the address that requested it
          msg.sender.transfer(withdraw_amount);
        }
      // Accept any incoming amount
      function () public payable {}
    }
```

为了获得部署字节码，我们将运行 `solc-bin Faucet.sol`。如果我们只需要运行时字节码，那么我们将运行 `solc-bin runtime Faucet.sol`。

如果您比较这些命令的输出，您将会知道运行时字节码是部署字节码的一个子集。换句话说，运行时字节码完全包含在部署字节码中。

分解字节码

分解 EVM 字节码是理解 EVM 中高级别的 Solidity 是如何工作的一个很好的方法。您可以使用一些拆装器来执行这个操作。

1. Porosity 是一种流行的开源反编译器。
2. Ethersplay 是一个用于 Binary Ninja 的 EVM 插件，一个反汇编程序。
3. IDA-Evm 是另一个反汇编程序 IDA 的 EVM 插件。

在本节中，我们将使用 Binary Ninja 的 Ethersplay 插件开始分解 the Faucet 的运行时字节码。在获得了 Facet.sol 的运行时字节码之后，我们可以把它输入 Binary Ninja （在加载了 Ethersplay 插件之后），看看 EVM 指令是什么样子的。

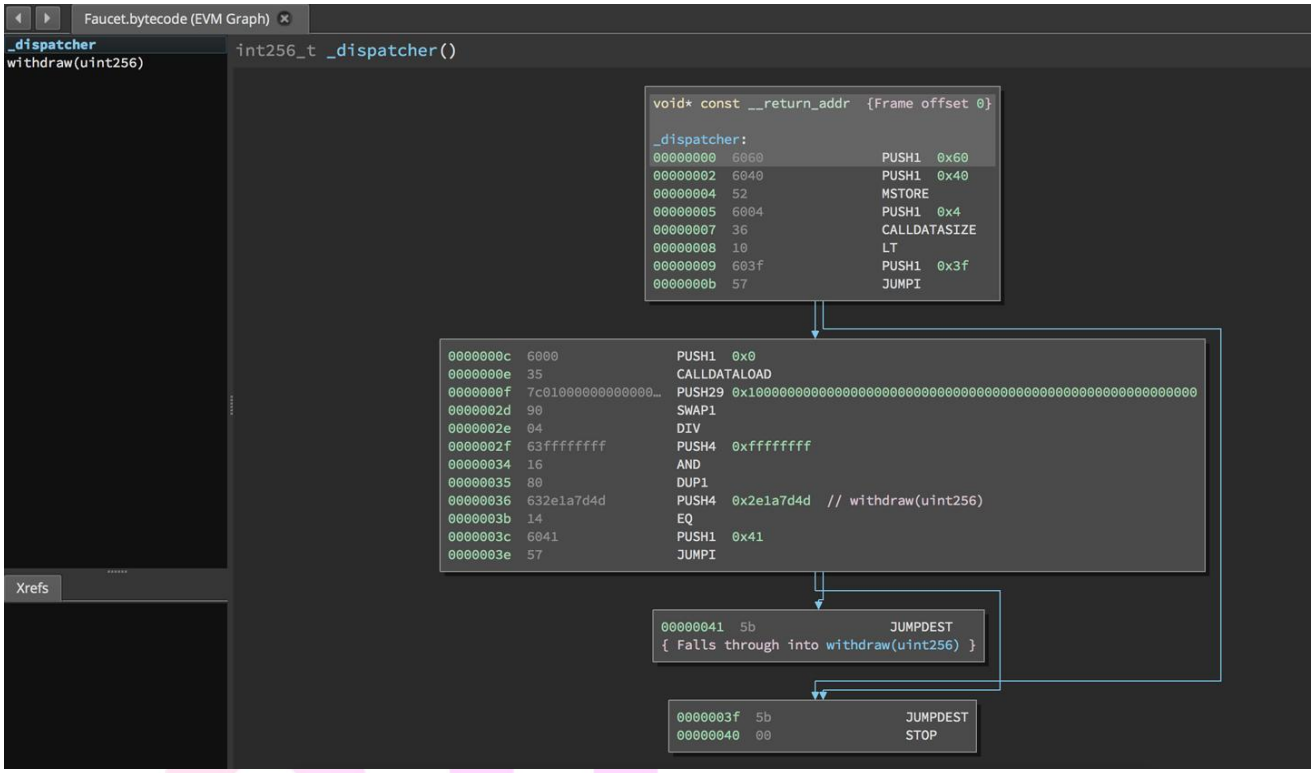


图 2：分解 the Faucet 运行时字节码

当您将交易发送到与 ABI 兼容的智能合约（您可以假定所有的合约都是）时，该交易首先与智能合约的调度程序进行交互。调度器读取交易的数据字段，并将相关部分发送给相应的函数。我们可以在被分解 Faucet.sol 运行时字节码的开头看到一个调度器的例子。在熟悉的 MSTORE 指令之后，我们可以看到以下指令：

```
PUSH1 0x4
CALLDATASIZE
LT
PUSH1 0x3f
JUMPI
```

如我们所见，PUSH1 0x4 将 0x4 放在堆栈顶部，否则该堆栈为空。CALLDATASIZE 获取交易发送的数据（称为 calldata），并将该数字推送到堆栈中。执行这些操作后，堆栈如下所示：

Stack
<length of calldata from tx>
0x4

下一条指令是 LT，它是小于的缩写。IT 指令负责检查堆栈上的顶项是否小于堆栈上的下一项，在我们的例子中，它检查 calldataSize 的结果是否小于 4 个字节。

为什么 EVM 要检查交易的 CallData 是否至少为 4 个字节呢？因为函数标识符是这样工作的。每个函数都凭借其 keccak-256 哈希的前 4 个字节来识别。通过将函数名及其参数放入 keccak256 哈希函数中，我们可以推断出它的函数标识符。在我们的案例中，我们有：

```
keccak256(“withdraw(uint256)”) = 0x2e1a7d4d...
```

因此，撤销（uint256）函数的函数标识符是 0x2E1A7D4D，因为它们是生成哈希的前 4 个字节。函数标识符总是 4 个字节长，因此，如果发送到协定的交易的整个数据字段小于 4 个字节，那么就没有可以与交易通信的函数，除非定义了回退函数，所以当 calldata 的长度小于 4 个字节时，EVM 将跳转到这个函数。

LT 指令弹出了堆栈中的前二个值，如果交易的数据字段小于 4 个字节，则将 1 推到堆栈上。否则，它将推 0。在我们的示例中，假设发送到我们的合约的交易的数据字段小于 4 个字节。

PUSH1 0x3f 指令将字节 0x3f 推送到堆栈上。在该指令之后，堆栈如下所示：

Stack
0x3f
1

下一条指令是 JUMPI，它代表“jump if”。它的工作原理如下：

```
jumpi(label, cond) // Jump to “label” if “cond” is true
```

在我们的例子中，标签是 0x3f，这是我们的回退功能在智能合约中的位置。Cond 参数是 1，这是前面的 LT 指令的结果。要将整个序列放入语言中，如果交易数据小于 4 个字节，合约将跳转到 Fallback 函数。

在 0x3f，只有一个 stop 指令跟随，即使我们声明了一个回退函数，我们也仍然保持它为空缺状态。正如您在导致回退函数的 JUMPI 指令中看到的，如果我们没有执行回退函数，合约将会抛出一个异常。

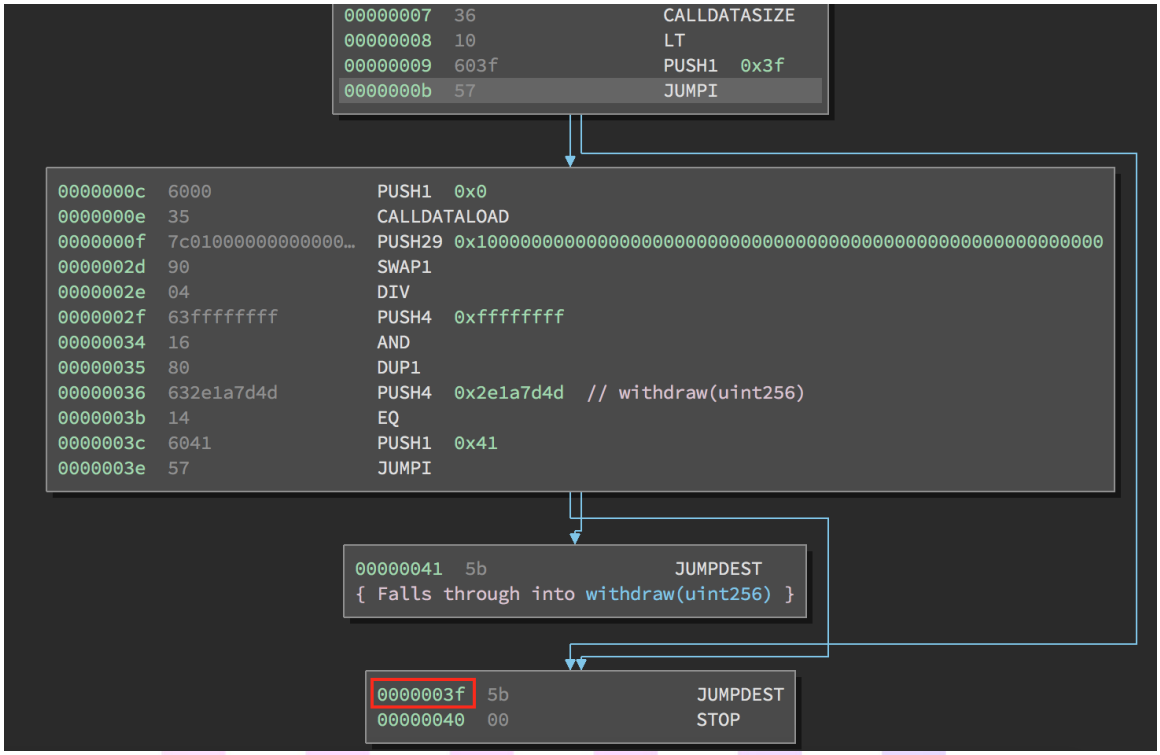


图 3：导致回退功能的 Jumpo 指令

让我们检查一下调度器的中心块。假设我们收到的 calldata 长度大于 4 个字节，JUMPI 指令将不会跳转到 fallback 函数。相反，代码执行将继续执行以下指令：

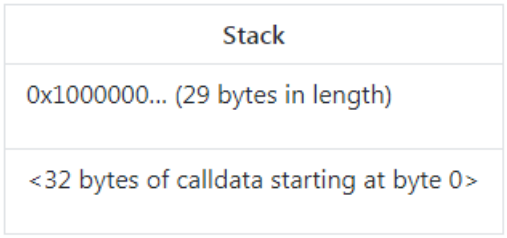
```
PUSH1 0x0
CALLDATALOAD
PUSH29 0x1000000...
SWAP1
DIV
PUSH4 0xffffffff
AND
DUP1
PUSH4 0x2e1a7d4d
EQ
PUSH1 0x41
JUMPI
```

PUSH1 0x0 将 0 推到堆栈上，否则堆栈将再次为空。CALLDATALOAD 接受发送到智能合约的 calldata 中的索引作为参数，并从该索引中读取 32 个字节，

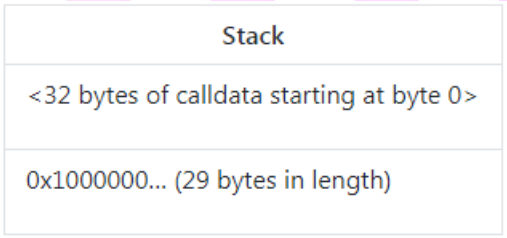
如下所示：

```
calldataload(p) //load 32 bytes of calldata starting from byte position p
```

因为 0 是从 PUSH1 0x0 命令传递给它的索引，所以 CALLDATALOAD 从字节 0 开始读取 32 字节的 calldata，然后将其推送到堆栈的顶部（在弹出原始的 0x0 之后）。在 push29 0x1000000……指令之后，堆栈为：



SWAP1 切换堆栈上的顶部元素，第 i 个元素在其之后。在这种情况下，它用 calldata 交换 0x1000000…。新的堆栈是：



下一条指令是 DIV，其运行如下：

```
div(x, y) // integer division x / y
```

在这种情况下，x=32 字节的 calldata 从 0 字节开始，y=0x100000000...（总共 29 字节）。这里有一个提示：我们之前从 calldata 读取了 32 个字节，从索引 0 开始。调用数据的前 4 个字节是函数标识符。

我们之前推过的 0x100000000...有 29 个字节长，由一个 1 开头，后面跟着的所有数字都是 0。将我们 32 个字节 calldata 除以这个值将只剩下我们的 calldata 加载的最前面的 4 个字节，从索引 0 开始。这 4 个字节，从索引 0 开始的 calldata 中的前 4 个字节是函数标识符，这就是 EVM 提取该字段的过程。

如果您不清楚这一部分，可以这样思考：在 10 进制中，123400/100=1234，在 16 进制中，这是相同的原理。我们用 16 进制的运算来取代 10 进制的运算，

正如我们之前的示例中除以 103（1000）只保留最前面的数字一样，将 32 字节的 16 进制值除以 1629 也是如此。

DIV（函数标识符）的结果被推送到堆栈上，现在我们的堆栈是：

Stack
<function identifier sent in data>

由于 PUSH4 0xffffffff 和 AND 指令是冗余的，我们完全可以忽略它们，因为完成它们之后，堆栈将保持不变。DUP1 指令复制堆栈上的第一项，即函数标识符。下一条指令 PUSH4 0x2e1a7d4d 将 withdraw（unit256）函数的预先计算函数标识符推送到堆栈上。现在堆栈是：

Stack
0x2e1a7d4d
<function identifier sent in data>
<function identifier sent in data>

下一条指令 EQ，将弹出堆栈的前两项并进行比较。调度器在这里执行其主要功能：它比较在交易的 msg.data 字段中发送的函数标识符是否与 withdraw（unit256）的函数标识符匹配。如果它们相等，则 EQ 将 1 推到堆栈上，该堆栈最终将用于跳转到 withdraw 函数。否则，EQ 将 0 推到堆栈上。

假设发送给我们的合约确实是以函数标识符 withdraw(unit 256)开始的，那么我们的堆栈就变成了：

Stack
1
<function identifier sent in data> (now known to be 0x2e1a7d4d)

接下来，我们有 PUSH1 0x41，它是 withdraw（unit256）函数在合约中的地址。在该指令之后，堆栈如下所示：

Stack
0x41
1
function identifier sent in msg.data

接下来是 JUMPI 指令，它再次接受堆栈中最前面的两个元素作为参数。在这种情况下，我们有 JUMPI(0x41,1)，它告诉 EVM 执行到 withdraw(unit256) 函数位置的跳转，并且可以继续执行该函数的代码。

图灵完备和 gas

正如我们已经提到的，简单来说，如果一个系统或者编程语言能够运行任何程序，那么它就是图灵完备的。然而这种性能带来了一个非常重要的警告：有些程序需要永久运行。一个很重要的层面是，我们不能仅仅通过查看一个程序来判断它是否需要永远执行。我们必须实际的完成程序的执行，并等待它完成来确认这一点。当然，如果一个程序需要永久的执行，我们必须等待到永久来确认。这被称为 halting 问题，如果不加以解决，以太坊系统运行将成为一个巨大的问题。

由于 halting 问题，以太坊世界计算机面临着要求被执行一个永不停止的程序的风险。这可能是偶然或者恶意的。我们已经讨论过，以太坊就像一个单线程机器，没有任何调度程序，因此如果它陷入无限循环，这意味着它将变得不可用。

但是，对于 gas，这儿有一个解决方案：如果在执行预先制定的最大计算量之后，执行没有结束，则 EVM 将停止程序的执行。这使得 EVM 成为一个准图灵完备的机器：它可以运行任何您输入的程序，但前提是程序在特定的计算量内终止。以太坊并没有固定的限额，您可以支付最高限额（称为“区块 gas 限额”），每个人都可以同意随着时间的推移增加该限额。然而，在任何时候，都有一个限制，在执行过程中消耗过多 gas 的交易将被停止。

在接下来的章节中，我们将了解 gas 并且检验它工作的细节。

Gas

Gas 是以太坊的单位，用于测量在以太坊区块链上执行操作所需的计算和存储资源。与比特币相比，比特币的交易费用只考虑交易的大小（以千字节为单位），以太坊币必须考虑交易和智能合约代码执行所执行的每一个计算步骤。

通过交易或合约执行的每项操作都需要固定数量的 gas。一些例子，来自以太坊黄皮书：

1．增加两个数字需要 3gas。

2．计算一个 keccak-256 散列值需要花费 30 个 gas+6 个 gas，每一个 256 位哈希数据需要花费 6 个 gas。

3．发送交易成本为 21000gas。

gas 是以太坊的一个重要组成成分，具有双重作用：在以太坊的（不稳定）价格和矿工工作报酬之间起到缓冲作用，并作为防御拒绝服务攻击的手段。为了防止网络中意外或者恶意的无限循环或其它的计算浪费。每个交易的发起方都需要对他们愿意支付的计算量设置限额。因此，gas 系统可以组织攻击者发送“垃圾邮件”交易，因为他们必须按照比例支付所消耗的计算，带宽和存储资源。

执行期间 gas 计费

当需要 EVM 来完成交易时，首先会提供与交易中 gas 限制规定数量相等的 gas。执行过程中的每一个操作码都有 gas 成本，因此随着 EVM 逐步执行该程序，EVM 的 gas 供应会减少。在每次运算之前，EVM 都会检查是否有足够的 gas 来支付操作的执行费用。如果没有足够的 gas，则停止执行，并回到交易。

如果 EVM 成功地达到执行结束的环节，而没有耗尽 gas，则使用的 gas 成本作为交易费用支付给矿工，并根据交易中规定的 gas 价格转换为以太。

矿工费用=gas 成本*gas 价格

Gas 供应中剩余的 gas 将退还给发送方，并根据交易中规定的 gas 价格再次转换成以太。

剩余 gas=gas 限值*gas 成本

退回的以太=剩余 gas*gas 价格

如果交易在执行过程中耗尽 gas,则立即终止操作，引发“gas 耗尽”异常。交易将被还原，状态的所有更改都将回滚。

尽管交易失败，但发送方将支付交易费用，因为矿工已经完成了计算工作，必须为此获得补偿。

gas 计费的考虑

可由 EVM 执行的各种运算的相对 gas 成本已经过仔细的选择，来确保最好地来保护以太坊区块链免受攻击。您可以在[evm_opcodes_table]中看到不同 EVM 操作码的 gas 成本详细表。

更多计算密集型运算需要更多的 gas。例如，执行 SHA3 函数的成本（30gas）是 ADD 运算(3gas)的 10 倍。更重要的是，一些操作（如 EXP）需要根据运算量的大小支付额外的费用。使用 EVM 内存和数据存储在合约的链上存储中也会产生 gas 成本。

2016 年，一名攻击者发现并利用了成本不匹配的问题，证明了将 gas 成本与实际资源成本进行匹配的重要性。攻击产生了需要非常巨大的运算量的交易，使得以太坊主网几乎陷入停顿。这种不匹配是通过一个硬分叉（代号为“Tangerine Whistle”）来解决的，它调整了相对 gas 成本。

gas 成本和 gas 价格

虽然 gas 成本是在 EVM 中计算和存储所用的计量单位，但 gas 本身也有一个用以太计量的价格。在执行交易时，发送方指定他们愿意为每单位 gas 支付的 gas 价格（以以太坊计算），允许市场决定以太价格与操作成本之间（以 gas 计）的关系：

交易费=总 gas 使用量*支付的 gas 价格（以以太计算）

在创造一个新的区块时，以太坊网络上的矿工可以通过选择那些愿意支付更高 gas 价格的交易来在未决交易中进行选择。因此，提供更高的 gas 价格将激励矿工将您的交易包括在内，并更快得到确认。

在实践中，交易的发送方将设定一个高于或者等于预期使用的 gas 数量的 gas 限额。如果设定的 gas 限值高于所消耗的 gas 量，则发送方将收到超额金额的退款，因为矿工只会得到与实际工作量相当的补偿。

必须明确 gas 成本和 gas 价格之间的区别。重述：

gas 成本是执行特定运算所需的 gas 的单位数。

gas 价格是当您将交易发送到以太坊网络时，您愿意支付每单位 gas 的以太金额。

提示：虽然 gas 价格昂贵，但它既不能“拥有”，也不能“消耗”。gas 只存在于 EVM 内部，作为正在被执行的计算工作量的计数。交易发送者在以太网上被收取交易费，然后交易费被转换为充当 EVM 计数的 gas，然后作为支付给矿工的交易费返回到以太网上。

负 gas 成本

以太坊鼓励删除已使用的存储变量和账户，通过退还合约执行期间使用的部分 gas。

EVM 中有两个操作的 gas 成本为负：

删除合约（自毁）是价值 24000gas 的退款。

将存储地址从非零值更改为零(SSTORE[x] = 0) 是价值为 15000gas 的退款。

为避免使用退款机制，交易退款的最大额设置为所用 gas 总量的一半（四舍五入）。

区块 gas 限额

区块 gas 限额是一个区块内所有交易可能消耗的最大 gas 量，并限制一个区块内可容纳多少交易。

例如，假设我们有 5 个交易，其 gas 限值分别为 30000，30000，40000 和 50000。如果区块容量为 180000，那么这些交易中的任何四个都可以放在一个区块内，而第五个交易则必须等待未来的区块。正如前面讨论的，矿工决定在一个区块内包含哪些交易。不同的矿工可能会选择不同的组合，主要是因为他们以不同的顺序从网络接收交易。

如果一个矿工试图包括一个需要比当前区块 gas 限额更多 gas 的交易，该区块将被网络拒绝。根据 https://etherscan.io，大多数以太坊客户端都会给出“交易超过区块 gas 限额”这一行信息作为警告，阻止您发布此类交易。根据 https://etherscan.io，在编写时，以太坊主网上的 gas 限额为 800 万 gas，这意味着大约 380 个基本交易（每个消耗大约 21000gas）可以满足进入一个区块。

什么决定了区块限额？

网络上的矿工共同决定了区块 gas 的限额。以太坊网络上进行挖掘的个人如果需要使用挖矿程序，例如链接到 Geth 或者对等以太坊客户端的 Ethminer。以太坊协议有一个内置机制，矿工可以在该机制中对 gas 限制进行投票，以便在后续区块中增加或者减少容量。一个区块的开采者可以进行投票，以 1/1024（0.0976%）的系数调整区块 gas 限值。其结果是根据当时网络的需求来调整块的大小。这一机制与默认的开采策略相结合，矿工投票决定的 gas 限额至少为 470 万，但是目标值为每个区块最近总 gas 使用量平均值的 150%。（使用 1024 个区块指数移动平均值）

结语

在本章中，我们探讨了以太坊虚拟机（EVM），跟踪了各种智能合约的执行，并研究了 EVM 如何执行字节码。我们还研究了 EVM 的会计机制 gas，并了解了它如何解决暂停问题并保护以太坊免于受到拒绝服务攻击。接下来，在[共识]中，我们将研究以太坊用来实现去中心化共识的机制。

第十四章 共识

共识

在本书中，我们讨论了“共识规则”，即每个人都必须同意的规则，以便使系统以分散的但具有确定性的方式运行。在计算机科学中，“共识”一词早于区块链，且与分布式系统中更广泛的同步状态问题有关，分布式系统中的不同参与者都（最终）同意一个系统范围的状态。这被称为“达成共识”。

当涉及到分散式记录保存和验证的核心功能时，仅依靠信任机制来确保从状态更新中获得的信息是正确的方式存在一定的问题。这种相当普遍的挑战在分散的网络中尤其明显，因为没有中心实体来决定什么是正确的。缺乏中心决策实体是区块链平台的主要吸引力之一，其因此具有抵制审查的能力且无需依赖权限即可访问信息。然而，这些好处是有代价的：如果没有一个可信的仲裁员，任何分歧、欺骗或分歧都需要通过其他方式加以调和。共识算法是用于协调安全和去中心化的机制。

在区块链中，共识是系统的一个关键属性。简而言之，钱是有风险的！因此，在区块链的背景下，共识是能够达成一个共同的状态，同时保持去中心化。换言之，共识旨在产生一个没有统治者的严格规则体系。没有任何一个人、组织或团体“掌权”；相反，权力和控制分散在广泛的参与者网络中，他们的自我利益是通过遵守规则和诚实行为来实现的。

所有开放公共区块链的核心原则是，在没有集中控制的情况下，于对抗条件中在分布式网络上达成共识的。为了应对这一挑战且保持去中心化，社区继续尝试不同的共识模型。本章探讨了这些共识模型及其对区块链智能合约（如以太坊）的预期影响。

注意　虽然共识算法是区块链运作的重要组成部分，但它们在基础层运行，远比智能合约抽象。换句话说，大多数共识的细节对智能合约的作者来说都是隐藏的。当你使用以太坊时，不需要知道它们是如何工作的，就像你使用互联网不需要知道路由是如何工作的一样。

通过工作证明达成共识

比特币，区块链最初的创始者发明了一种称为工作证明（POW）的共识算法。可以说，POW 是支撑比特币最重要的机制。我们通常称 POW 为“挖矿”， 这也造成了对共识主要目的的误解。人们通常认为挖矿的目的是创造新的货币，因为现实世界中的挖矿是为了开采贵金属或其他资源。与此相反，POW（以及所有其他共识模型）的真正目的是确保区块链的安全，同时保持对系统的控制分散，并在尽可能多的参与者之间分散。挖矿的奖励是对那些为体系安全做出贡献的人的一种激励：一种达到目的的手段。从这个意义上说，奖励是手段，分散的安全是目的。在 POW 共识中，也有相应的“惩罚”，即挖矿所需的能源成本。如果参加者不遵守规则并获得奖励，他们有可能会损失花在电力开采上的资金。因此，POW 共识是一种谨慎的风险和回报平衡，促使参与者诚实行事。

以太坊目前同样是 PoW 区块链，因为它使用具有相同基础激励系统的 PoW 算法来实现相同的目标：在去中心化的同时保护区块链。以太坊的 PoW 算法与比特

币略有不同，被称为 Ethash。 我们将在 Ethash：Ethereum 的工作证明算法中研究该算法功能和设计特点。

通过股权证明（PoS）达成的共识

从历史上看，工作证明并不是第一个提出的共识算法。在引入工作证明之前，许多研究人员提出了基于金融股权的共识算法的变体，现在称为股权证明（PoS）。在某些方面，工作证明被发明作为股权证明的替代方案。随着比特币的成功，许多区块链已经开始效仿工作证明机制。然而，对共识算法研究的爆炸式增长使股权证明再次获得关注，极大的推动了技术的发展。从一开始，以太坊的创始人就希望最终将其共识算法迁移到股权证明上。事实上，以太坊的工作证明存在一个蓄意障碍，称为难度炸弹，旨在逐渐使以太坊的工作证明挖掘变得越来越困难，从而迫使过渡到股权证明。

在这本书出版之时，以太坊仍在使用工作证明，但正在进行的，关于股权证明替代方案的研究接近完成。以太坊计划的 POS 算法称为 casper。在过去两年中，引入 Casper 来替代 Ethash 被推迟了好几次，因此需要采取必要的措施解决难度炸弹，并推迟放弃工作证明。

一般来说，pos 算法的工作原理如下。区块链跟踪一组验证器，任何持有区块链基本加密货币的人（在以太坊的情况下，以太）都可以通过发送特殊类型的交易来将其以太币锁定到存款中，从而成为验证者。验证者轮流提议并对下一个有效块进行投票，每个验证者的投票权重取决于其存款的大小（即股权）。重要的是，如果验证者选择的区块被大多数验证人拒绝，则有可能失去他们的存款。相反，对于被大多数人接受的每个区块，验证者都会获得与其存放的股份成比例的小额奖励。因此，PoS 通过奖励和惩罚制度，迫使验证者诚实行事并遵循共识规则，。 PoS 和 PoW 之间的主要区别在于 PoS 中的惩罚是区块链固有的（例如，失去的以太），而在 PoW 中，惩罚是外在的（例如，用于电力的资金损失）。

Ethash：以太坊的工作证明算法

Ethash 是以太坊 PoW 算法。 它使用了 Dagger-Hashimoto 算法的演化，该算法是 Vitalik Buterin 的 Dagger 算法和 Thaddeus Dryja 的 Hashimoto 算法的结合。Ethash 依赖于大数据集的生成和分析，称为有向无环图（或更简单地说，“DAG”）。 DAG 的初始大小约为 1 GB，并且将继续缓慢且线性地增长，每个周期更新一次（30,000 个块，或大约 125 个小时）。

DAG 的目的是使 Ethash PoW 算法维持大型，频繁访问的数据结构。这反过来旨在使 Ethash“抗 ASIC”，这意味着制造比快速图形处理单元（GPU）快几个数量级的专用集成电路（ASIC）采矿设备更加困难。以太坊的创始人希望避免在 PoW 挖掘中集中化，因为专门的硅制造工厂和巨额预算的公司可能主导采矿基础设施，从而破坏共识算法的安全性。

使用消费级 GPU 在以太坊网络上执行 PoW 意味着全球更多人可以参与挖矿过程。矿工越是独立，矿业权就越分散，从而意味着我们可以避免像比特币这样的局面，大部分矿业都集中在一些大型矿厂中。将 GPU 用于采矿的缺点在于它导致了 2017 年全球的 GPU 短缺，其价格飙升，并引起游戏玩家的强烈抗议。从而导致了购买限制，每个客户只能购买一到两个 GPU。

直到最近，以太坊网络上的 ASIC 矿工的威胁基本不存在了。在以太坊中使用 ASIC 需要设计，制造和分发高度定制的硬件。 生产它们需要大量的时间和金钱投入。以太坊的开发者长期以来表示想要迁移至 PoS 共识算法，使 ASIC 供应商一直以来远离以太网络。因为

一旦以太坊移动到 POS，为 POW 算法设计的 ASIC 将变得无用，除非矿工们可以使用它们来开采其他加密货币。后一种可能性现在已经成为现实，即开采其他基于 ethash 的共识硬币，如 PIRL 和 Ubiq 并且以太坊经典已经承诺在可预见的未来保持 POW 区块链。

这意味着我们很可能会看到 ASIC 挖掘在以太坊网络上开始成为一股力量，尽管它仍以 POW 共识运作。

Casper：以太坊的股份证明算法

Casper 是以太坊 PoS 共识算法的拟议名称。 它仍处于积极的研究和开发阶段，并且在本书出版时尚未在以太坊区块链上实施。 Casper 正在开发两种相互竞争的“特点”。

casper ffg：“友善的小精灵”。

casper cbc：“友好幽灵/使用构建修正”。

最初，casper-ffg 被提议作为一种混合的 pow/pos 算法，作为一种向更持久的“纯 pos”算法的过渡来实现。但是在 2018 年 6 月，领导 casper-ffg 研究工作的 Vitalik Buterin 决定放弃混合模型，转而采用纯 pos 算法。现在，

casper-ffg 和 casper-cbc 都是并行开发的。正如 Vitalik 所解释的：FFG 和 CBC 之间的主要折衷是 CBC 似乎具有更好的理论性质，但 FFG 似乎更容易实现。

有关 Casper 的历史、正在进行的研究和未来计划的更多信息，请访问以下链接：

Ethereum Casper（股权证明）
Casper 的历史，第一章
Casper 的历史，第二章
Casper 的历史，第三章
Casper 的历史，第四章
Casper 的历史，第五章

共识原则

通过几个关键问题，可以更清楚地理解共识算法的原理和假设：

谁能改变过去，如何改变？（这也被称为不可变。）
谁能改变未来，如何改变？（这也被称为最终性。）
做出这种改变的代价是什么？
做出这种改变的权力是如何分散的？
如果发生改变谁将知道？他们又是如何知道的呢？

共识算法正在迅速发展，试图以越来越多的创新方式来回答这些问题。

争议与竞争

在这一点上，你可能会想：为什么我们需要这么多不同的共识算法？哪一个更好？后一个问题的答案是过去十年分布式系统中最激动人心的研究领域的核心。归根结底，这一切都归结为你认为“更好”的东西——在计算机科学的背景下，它是关于假设，目标和不可避免的权衡取舍。

似乎没有一种算法能够在分散共识问题的所有维度上进行优化。当有人建议一种共识算法比其他算法“更好”时，你应该开始问一些问题来澄清：哪方面更好？不变性，终结性，分散性，成本？ 这些问题没有明确的答案，至少现在还没有。此外，共识算法的设计是数十亿美元产业的中心且引起了巨大的争议和激烈的争论。最后，可能没有一个“准确”的答案，就像不同的应用程序可能有不同的答案。

整个区块链行业是一个很大的实验，这些问题将伴随着巨大的财产风险在对抗条件下进行测试。最终，历史将回答这些争议。

结论

本书完成时，以太坊的共识算法仍在不断变化。在未来的版本中，随着这些技术的成熟和部署在 以太坊上，我们可能会添加更多关于 casper 和其他相关技术的详细信息。这一章代表着我们旅程的结束。附加参考资料见附录。感谢您阅读本书，恭喜您阅读完毕！

《精通以太坊》

Andreas M Antonopoulos，Gavin Wood 著，AMT 社区翻译组等译

由于时间和各译者对专有词汇理解等原因，初稿中存在一些格式及翻译错误。经 AMT 社区翻译组成员的多次审核与校验，修正了多处错误与表意不明的地方，在此对辛苦付出的伙伴们表示衷心的感谢。

即便如此，当前版本仍可能存在部分错误，欢迎读者们在 GitHub 上提交勘误，也可以发至邮箱：

vns@amt.life
