

目录

- 1. 介绍
- 2. 前言
- 3. 快速术语检索
- 4. 第1章 介绍
- 5. 第2章 比特币的原理
- 6. 第3章 比特币客户端
- 7. 第4章 密钥、地址、钱包
- 8. 第5章 交易
- 9. 第6章 比特币网络
- 10. 第7章 区块链
- 11. 第8章 挖矿与共识
- 12. 第9章 竞争币、竞争区块链和应用程序
- 13. 第10章 比特币安全
- 14. 附录1
- 15. 附录2
- 16. 附录3
- 17. 附录4
- 18. 贡献与勘误

精通比特币

作者：Andreas M Antonopoulos

Andreas M. Antonopoulos 是一位著名的技木专家和连续创业企业家，比特币界最著名和倍受尊敬的人物之一。身为一名迷人的公共演说家、教师和作家，他善于把复杂的问题变得简单而易于理解。

Andreas M. Antonopoulos 的成长和互联网密不可分，青少年时期，他便在自己希腊的家中创办了他的首家公司——一个早期电子信息服务系统和原始的互联网服务提供商。他取得了伦敦大学学院的电脑科学学位、数据通信学位以及分布式系统学位，该学校最近跻身于全球大学排名前十。移居美国后，Andreas M. Antonopoulos 和别人合办了一家成功的技术研究公司，管理公司的他，在网络、安全、数据中心和云计算方面，为许多世界五百强公司的首席执行者提供建议。他撰写了200多篇关于安全、云计算和数据中心的文章，已经在世界范围内被印制出版，并在多家报刊发表。此外，他还持有两项网络和安全领域的专利权。

1990年，Andreas M. Antonopoulos 开始在私人、专业和学术等诸多场合讲授IT话题。从五名企业高管的会议室，到千人参与的大型会议，他不断磨练着自己的演讲水平。超过400次的锻炼，不仅使他成为了为人津津乐道的教授，还铸就了他世界级演讲大师的美名。2014年，第一所授予数字货币学士学位的尼科西亚大学邀请他来校任教。上任后，他与别人合作开设了这门课程。他还参与讲授了数字货币导论，后者成了为尼科西亚大学的一门大型网络公开课（MOOC）。

作为一名比特币企业家，Andreas M. Antonopoulos 已成立了不少比特币企业，并推出了一些社区开源项目。他担任好几家比特币和加密货币公司的顾问。他既是一名出版作家，发表了大量关于比特币的文章和博文；也是一档流行博客“Let's Talk Bitcoin”的固定主持人；还是在全球各地的技术安全会议上演说的常客。

译者：

薄荷凉幼；陈萌琦；陈姝吉；程鹏；程西园；达文西；吉鸿帆；李丹；李润熙；李凌豪；李昕阳；刘畅；吕新浩；马峰；牛东晓；秦彤；邱蒙；戎如香；史磊；汪海波；王宏钢；辛颖；杨兵；尹文东；余龙；张林；张琦张大嫂；张亚超；张泽铭；赵冬帅；赵余；YANG YANG

简介

想要加入一场颠覆金融世界的技术革命吗？《精通比特币》会为你参与这个货币网络提供必备知识，引导你进入看似复杂的比特币世界。无论你是正在构建下一个杀手级应用、投资创业，还是单纯对技术好奇，这本实用的书都是你不可或缺的阅读材料。

比特币，作为第一个成功的去中心化数字货币，尽管还处在起步阶段，却已经催生了数十亿美元的全球性经济体。它对任何具备相应知识和参与热情的人都是开放的。《精通比特币》会为您提供必要的知识，但请各位读者自备热情。

本书包括：

- ▷ 针对非技术用户、投资者以及企业高管，概括性地介绍比特币
- ▷ 针对开发人员、工程师以及软件系统架构师，介绍比特币和加密货币的技术基础
- ▷ 详细介绍比特币去中心化网络、点对点体系结构、交易生命周期以及安全原则等细节的
- ▷ 比特币和区块链的发明的衍生物，包括替代链、货币以及应用程序
- ▷ 通过用户故事、简练的类比、示例以及代码段来阐释的关键技术概念

Andreas M. Antonopoulos 是比特币行业最为知名和受人尊敬的技术专家、连续创业家，现担任多家技术创业公司的顾问。他同时还是一名出色的演说家，教师以及作家，经常在世界各地的会议和团体活动中发表演说，用通俗易懂的讲解把复杂的学科普及给广泛听众。

第1章 介绍

1.1 什么是比特币？

比特币是一些概念与技术的集合，这些概念与技术构成了数字货币生态系统的基础。人们称之为“比特币”的货币单位用于存储，也用于在比特币网络参与者之间传输价值。比特币用户之间通过比特币协议进行通讯主要是在互联网上，当然也可以使用其他传输网络进行交流。比特币的协议栈以开源软件的形式实现，这些软件可以在包括笔记本电脑、智能手机在内的多种设备上运行，这使得比特币技术非常亲民。

用户可以通过在网络上传输比特币来实现几乎传统货币能做的全部事情，例如买卖商品、给个人或组织汇款，或提供贷款。比特币可以在专门的交易所里购买，出售，或者与其他币种进行兑换。从一定意义上来说，由于具有快捷、安全以及无国界的特性，比特币正是互联网货币的完美形态。

不同于传统的货币，比特币是完全虚拟的。没有实物的货币，甚至就本质而言，也没有数字货币。比特币隐匿于发送者和接收者间价值传递的交易中。比特币用户拥有能够使他们在比特币网络中证明自己交易权的密钥，解密后可使用比特币，也可以将它购买、出售，以及与其他币种进行兑换。由于比特币快捷、安全以及无国界的特性，在某种意义上，比特币就是互联网货币的完美形态。

用传统货币能做到的事情，用户在网络上利用比特币都可以做到，包括发送给新的接收者。这些密钥通常存储在每个用户的计算机的数字钱包里。每一笔交易都需要密钥解密，这是使用比特币的唯一先决条件，它完完全全掌握在每个用户的手中。

比特币是一个分布式的点对点网络系统。因此，没有“中央”服务器，也没有中央控制点。比特币是通过一个名为“挖矿”的过程产生的，挖矿需要在处理比特币交易的同时参与竞赛来解决一个数学问题。在比特币网络中的任何参与者（比如，任何人使用一个设备来运行完整的比特币协议栈）都是潜在的矿工，用他们电脑算力来验证和记录交易。每隔10分钟，有人能够验证过去10分钟发生的交易，作为回报，将会获得崭新的比特币。从本质上讲，比特币挖矿分散了中央银行的货币发行，也分散了其结算功能，并且能够在全球竞争中取代任何一家中央银行。

比特币协议包括了内置算法，该算法可以调节网络中的挖矿功能。矿工必须完成的任务——在比特币网络中成功地记录一个区块交易——的难度是在动态调整的，因此，无论何时有多少矿工（多少CPU）在挖矿，通常每10分钟就会有人成功。

新比特币开采出的每四年，这项协议也会减半开采速率，并限制比特币的开采总量为一个固定值：2,100万枚。其结果是，在流通中的比特币数量很容易根据预测曲线得出，将会在2140年达到2,100万枚。由于比特币的发行率是递减的，从长期来看，比特币是一种通货紧缩的货币。此外，通过超出预期发行率来“印刷”新比特币，造成通货膨胀是不可实现的。

实质上，比特币本身也是协议，是一种网络，是一种分布式计算革新的代名词。比特币通货仅是这种创新的首次应用。作为一个开发者，我看到比特币类似于互联网货币，一个通过分布式计算来传播价值和保障数字资产所有权的网络。比起初识比特币，这里将知无不言。

在本章中，我们将会从一些主要概念和术语解释开始，获取必备软件，使用比特币进行简单的交易。在接下来的章节里，我们将开始揭开使比特币成为可能的技术面纱，解释比特币网络和比特币协议的内部运行机制。

比特币之前的数字货币

切实可行的数字货币的出现是与密码学发展息息相关的。基本的挑战在于，当一个人考虑到用比特币代表可以兑换商品和服务的价值时，接受数字货币也就不足为奇。任何接受数字货币的人都面临的两个基本问题是：

1. 我能相信这钱是真实可信的，而不是伪造的吗？
2. 我能确定没人说这笔钱是他们的，而不是我的吗？（又名“双重支付”问题）

纸币的发行机构不断的利用日益复杂的纸张和打印技术来遏制造假问题。实物货币很容易解决双重支付问题，因为同一张纸币不可能会同时出现在两个不同的地方。当然，传统货币也经常数字化储存和数字化传输。在这些情况下，假币和双重支付问题是被中央权威机构的处理方式是清除所有的电子交易记录，该中央权威在流通中持有一种全球通货

观。对于数字货币来说，不能有效利用秘制油墨印刷技术或条形全息图，密码学为用户所主张的合法性价值提供了信任的基础。具体地来说，加密数字签名能够使一个用户签署一项能够提供其资产所有权证明的数字资产或数字交易。采用适当的结构，数字签名也可以用于解决双重支付的问题。

在20世纪80年代后期，当密码学开始越来越广泛地使用并被理解时，许多研究人员开始尝试使用密码学来建立数字通货。这些早期的数字通货项目发行的数字货币，通常倚靠一种国家通货或像黄金一样的贵金属。

虽然这些早期的数字通货的运行了，他们却是中心化的，因此，他们很容易遭受到政府和黑客的攻击。早期的数字通货使用了一个中央结算所来处理所有的定期交易，就像一个传统的银行系统。不幸的是，在大多数情况下，这些新兴的数字货币成为了政府担忧的目标，最终从法律程序上消失了。另一些则是在发行这些数字货币的母公司突然违约时颓然失败了。为了坚定的抵制对手的介入，无论这些对手是合法的政府或是犯罪分子，去中心化的数字货币需要的是避免单节点攻击。比特币正是这样的系统，设计完全去中心化，不被任何中央政权或中央点控制，这样的货币系统是不会遭受攻击，也不会变得腐败。

比特币代表了数十年的密码学和分布式系统的巅峰之作，这是一个独特而强大的组合，汇集了四个关键的创新点。比特币由这些构成：

- ▷ 一个去中心化的点对点网络（比特币协议）
- ▷ 一个公共的交易账簿（区块链）
- ▷ 一个去中心化的数学的和确定性的货币发行（分布式挖矿）
- ▷ 一个去中心化的交易验证系统（交易脚本）

1.2 比特币发展史

2008年，一位化名为中本聪的人，在一篇名为《比特币：一个点对点的电子现金系统》的论文中首先提出了比特币。中本聪结合以前的多个数字货币发明，如B-money和HashCash，创建了一个完全去中心化的电子现金系统，不依赖于通货保障或是结算交易验证保障的中央权威。关键的创新是利用分布式计算系统（称为“工作量证明”算法）每隔10分钟进行一次的全网“选拔”，能够使去中心化的网络同步交易记录。这个能优雅的解决双重支付问题，即一个单一的货币单位可以使用两次。此前，双重支付问题是数字货币的一个弱点，并通过一个中央结算机构清除所有交易来处理。

根据中本聪的一篇涉及比特币网络运行的发表论文，比特币网络自从被许多其他的程序员修订之后，于2009年启动。分布式计算，为比特币提供了成倍增长的安全性和韧性，现在超过了世界顶级超级电脑的联合处理能力。根据比特币兑美元汇率，比特币的总市场估值为50至100亿美元。目前从全网来看，比特币处理的最大交易额为1.5亿美元，这笔交易及时处理和转账，没有缴纳任何手续费。

中本聪在2011年4月退出公众视野，将比特币代码开发与网络建设的重任留给了欣欣向荣的社区成员。而“中本聪”究竟是谁，时至今日仍然是未解之谜。然而，比特币系统的运行，既不依赖于中本聪，也不依赖于其他任何人——比特币系统依赖于完全透明的数学原理。这项发明本身就是开创性的，它已经蔓延到了分布式计算、经济学、计量经济学领域。

一个分布式计算问题的解决方案

中本聪的此项发明，对“拜占庭将军”问题也是一个可行的解决方案，这是一个在分布式计算中未曾解决的问题。简单来说，这个问题包括了试图通过在一个不可靠、具有潜在威胁的网络中，通过信息交流来达成一个行动协议共识。中本聪的解决方案是使用工作量证明的概念在没有中央信任机构下达成共识，这代表了分布式计算的科学突破，并已经超越了货币广泛的适用性。它可以用来达成去中心化的网络共识来公正选举、彩票、资产登记，以及数字化公证等等。

1.3 比特币的应用、用户和他们的故事

比特币是一项技术，但它所传递的货币从实质上来说，是一种人与人之间价值交换的基础语言。让我们通过他们的故事，来看看使用比特币的人和一些最常用的通货和协议。我们将会反复用到这些贯穿本书的故事，以此来说明现实生活中数字货币的用途，以及他们是如何通过比特币的各种技术使之成为可能的。

Alice住在北加州的旧金山湾区。她已经从她的科技迷朋友口中得知了比特币，想要开始使用它。我们会跟进她的故事，来了解比特币，获取一些，并在帕洛阿尔托的Bob家咖啡店用比特币购买一杯咖啡。这个故事会从零售的消费者角度向我们介绍比特币软件，交易平台，以及基本的交易。

北美高价零售

Carol是一位旧金山艺术画廊的主人。她出售昂贵的油画来换取比特币。这个故事将介绍高额商品的零售商们所面临的“51%”攻击的风险。

离岸合同服务

Bob是帕洛阿尔托一家咖啡店的老板，正在建设一个新网站。他曾与一个住在印度班加罗尔的网站开发者Gopesh签订了协议。Gopesh已同意比特币支付。这个故事将研究使用比特币进行海外购、合约服务，以及国际电汇。

慈善捐赠

Eugenia是菲律宾一家儿童慈善机构的董事。最近，她发现了比特币，并希望利用它来和一个全新的国内外捐助群体接洽，以此为她的慈善募捐。她还研究如何使用比特币快速优化资金配置。这个故事将会演示用比特币来进行跨币种跨国界的全球融资，展示慈善组织所使用的公开透明账簿。

进口/出口

Mohammed是迪拜一位电子进口商。他试着用比特币来进行快捷支付，进口美国和中国的电子产品到阿联酋。这个故事将示范用比特币来支付大型企业间实物商品的国际收支。

比特币挖矿

Jing是上海一名计算机工程专业的学生。他建了一个用来挖比特币的矿机，使用他的专业技能来为自己创收。这个故事将审查基于比特币的“工业”：用于确保比特币网络安全和发行新货币的特殊设备。

这些故事都是根据真实行业内的真实人物原型来的，他们正在用比特币创造新市场，创造新产业，用比特币这个新兴事物来解决全球经济问题。

1.4 入门

加入比特币网络并开始使用通货，所有用户需要做的就是下载应用程序或使用在线应用程序。因为比特币是一个标准，也有许多运行比特币的客户端软件。还有一个标准客户端，也称为中本聪客户端，这是由一个开发团队管理的一个开源项目，源自于中本聪编写的初始客户端。

比特币客户端的三种主要形式是：

完整客户端

一个完整客户端，或称“全节点”，是存储所有比特币交易的整个交易历史（由每一个用户完成的每一笔交易，曾经所有的每一笔）的客户端，管理用户的钱包，并可以在比特币网络上直接开始交易。这类似于一个独立的电子邮件服务器，因为它处理着协议的各个方面，而不依赖于任何其它的服务器或第三方服务。

轻量级客户端

一个轻量级客户端存储用户的钱包，但需要依赖第三方服务器才能进行比特币交易，才能接入比特币网络。轻量级客户端不保存所有交易的完整副本，因此必须信赖第三方的服务器来获取交易确认。这就类似于一个独立的电子邮件客户端，能够通过邮箱服务器来访问一个邮箱，因为它在网络交流中依赖于一个第三方服务器。

在线客户端

在线客户端通过网页浏览器在第三方服务器上访问和储存该用户的钱包。这类似于在线邮件，因为它完全依赖于第三方服务器。

移动客户端

智能手机的移动客户端，例如基于Android系统，既可以作完整客户端运行，也可作为轻量级客户端或在线客户端。一些移动客户端是与在线客户端或桌面客户端同步的，提供跨多个设备但有一个共同的资金源的多平台钱包。

比特币客户端的选择，取决于用户想要管理资金的数目。一个完整的客户端将为用户提供最高级的管理和独立性。这样钱包的备份和安全责任就转移到了用户身上。另一种选择是在线客户端，其设置和使用是最简单的，但在线客户端的取舍还在于需衡量第三方介入的风险，因为安全性和控制权是由用户和网页服务商所共同承担的。如果一个在线钱包服务遭受了损失，就像已发生过的那样，用户们可能会失去所有的资金。反过来看，如果用户的一个完整客户端没有进行适当的备份，他们可能会因为电脑的操作失误而丢失他们的资金。

这本书的目的在于，我们将演示各种可下载的比特币客户端的使用方法，从原版客户端（中本聪客户端）到在线钱包。一些案例将使用到原版客户端，除了作为一个完整的客户端以外，也会公开钱包的API，网络和交易服务。如果您计划进入比特币系统探索编程的话，将会需要原版客户端。

1.4.1 快速入门

我们在“[1.3 比特币的应用、用户和他们的故事](#)”一节中介绍了Alice，她并不是技术性用户，最近才从一个朋友那听说了比特币。她通过浏览比特币官方网站[bitcoin.org](#)开始了自己的比特币之旅，在官网上，她发现了很多种可供选择的比特币客户端。她根据官网提供的建议，选择了轻量级的Multibit客户端。

Alice通过官网[bitcoin.org](#)上提供的链接，在她的电脑里下载安装了Multibit客户端。目前Multibit电脑客户端有Windows Windows, Mac OS 和 Linux版本。



比特币钱包必须由一个密码或密令来保护。有许多试图破解弱密码的潜在威胁，所以要注意谨慎设置一个不会被轻易破解的密码。密码应使用大小写字母、数字和符号的组合，避免出现生日、球队名字等个人信息。避免使用任何能在字典里轻易找到的词语，不管这个词语是什么语言的。如果条件允许，可以利用密码生成器生成一个完全随机的12位以上的密码。请记住：比特币是一种随时能被转移到世界其他任何地方的货币。如果不加以妥善保管，会很容易被偷走。

Alice下载并安装了Multibit客户端后，打开程序就会出现一个欢迎界面，如图1-1所示：

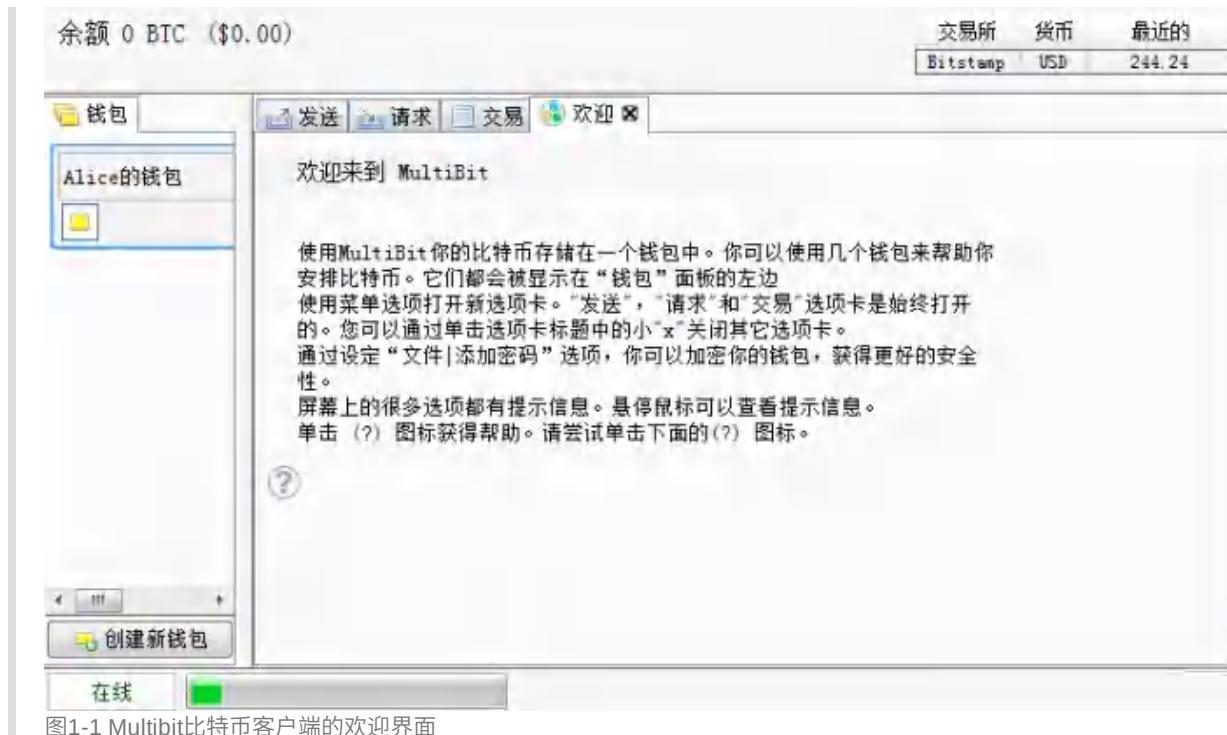


图1-1 Multibit比特币客户端的欢迎界面

Multibit客户端会自动为Alice生成一个钱包和一个全新的比特币地址，点击图1-2所示的请求标签即可看到。



图1-2 Multibit客户端请求标签中Alice的新比特币地址

界面上最重要的是Alice的比特币地址。类似于电子邮件的地址，Alice可以分享这个地址，这样任何人都可以通过这个地址直接将比特币发送到的新钱包里。界面上看起来一长串的字母和数字就是地址：

1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK。

地址旁是一个二维码。这个二维码是可以被智能手机摄像头扫描到该钱包地址的条形码，即窗口右边黑白相间的方块。通过点击比特币地址或二维码旁边的复制按钮，Alice可以将它们分别复制到剪贴板。点击二维码可以将其放大，便于智能手机扫描。

Alice也可以打印出这个地址二维码给别人，这样就不用打一长串字母和数字了。

比特币地址以数字1或3开头。类似于电子邮件地址，这些地址可以分享给其他的比特币的用户，这样他们就可以直接通过这些地址发送比特币到你的钱包里。不同于电邮地址的是，你可以任意地创建新的比特币地址，这些地址都能成功地将资金转到你的钱包。钱包是多个地址和解锁资金密钥的简单集合。每笔交易你都可以使用不同的地址，这有利于提高隐私的安全性。用户可创建地址的数量几乎不受限制。

现在Alice已经准备好开始使用她的新比特币钱包了。

1.4.2 获取你的第一枚比特币

现在你还无法在银行或是外币兑换处买到比特币。截至2014年，在大多数国家，购买比特币还是相当困难的。你可以去一些专门的通货交易所，购买比特币或是出售比特币换取当地货币，交易所是以在线通货市场的方式来运营。包括以下几种：

Bitstamp

一个欧洲通货市场，通过电汇方式，支持包括欧元、美元在内的多币种交易。

Coinbase

美国比特币钱包，也是买家和卖家进行比特币交易的一个平台。Coinbase允许用户通过ACH系统来连接美国支票账户，这样易于购买和出售比特币。

这一类的数字加密货币交易所，在国家货币和加密货币夹缝中求生存。因此，会受各国法规和国际法规的制约，而且往往具体到某单个国家或经济区，并只限于该地区的国家货币。你所选择的货币交易所，只限于你使用的本国货币，也只能是在你国范围内合法运营的交易所。类似于在银行开户，用此类服务来设置这些必要的账户需要花费数日或数周的时间，因为他们需要各种形式来了解你的客户，确认交易将符合反洗钱法。一旦你拥有了交易所的一个账户，你就可以像用代理账户购买出售外币一样的，更快捷地购买或出售比特币了。

[bitcoin charts](#)是提供价格索引的站点，一个包括数十家货币交易所其他市场数据的站点，在这里你可以找到更完整的数据。

新用户有以下四种方法来获取比特币：

- ▷ 找个有比特币的朋友，直接向他购买一些。很多比特币用户都是这样开始的。
- ▷ 利用[localbitcoins.com](#)这样的分类服务网站来寻找你所在地区的卖家，使用现金进行面对面的线下交易。
- ▷ 出售某种产品或服务来换取比特币。如果你是个程序员，可以出售你的编程技能。
- ▷ 使用你所在地区的比特币ATM机。利用CoinDesk里的[在线地图](#)来找到你附近的比特币取款机。

Alice是经朋友介绍认识比特币的，所以在等待加州通货市场上的账号被验证和激活的同时，她轻而易举的就获取了她的第一枚比特币。

1.4.3 发送和接收比特币

Alice已经创建好她的比特币钱包，准备接收资金了。她的钱包程序随机生成了一个私钥（关于私钥的详细介绍见“[4.1.3 私钥](#)”）和对应的比特币地址。这时，她的比特币地址还未在全网公布，也未在在任何比特币系统中“登记”。她的比特币地址只是一串数字，对应一个她可以掌控的资金私钥。在该地址和账户之间还没有任何交易产生，也没有任何关联。直到这个地址接收到在比特币账簿（区块链）中公布过的一笔交易时，才会成为众多可能“有效”的比特币地址之一。一旦该地址接受了一笔交易，就会变成全网所知的地址之一，Alice就可以在公开账簿中查询余额了。

Alice和Joe约在当地的一个饭店里会面，正是Joe把比特币介绍给了Alice的。这样Alice就可以用美金向Joe换取一些比特币，让Joe发送一些比特币到她的账户里。她带来了打印版的比特币地址和钱包二维码。从安全角度来说，比特币地址没什么特别机密的。她可以在任何地方公布自己的地址，而不用担心帐户安全。

Alice只想兑换10美元的比特币，免得在这项新技术上冒险花太多钱。所以她给了Joe 10美元和她地址的打印件，这样Joe就可以给她发送等值的比特币了。

接下来Joe需要弄清楚汇率，以便于发送给Alice相应数额的比特币。有很多应用和网站都会提供当前的市场汇率，下列是一些最流行的：

[Bitcoin Charts](#)

Bitcoin Chart 是一个市场数据服务网站，显示了全球众多交易所的比特币市场汇率，以当地不同的汇率来进行结算。

[Bitcoin Average](#)

Bitcoin Average是一个提供每个币种的交易量加权平均价格的简单视图网站。

[ZeroBlock](#)

ZeroBlock是一个免费的安卓和iOS应用程序，可以显示不同交易所的比特币价格。（见图1-3）

[Bitcoin Wisdom](#)

另一个市场数据索引服务站。

BTC-E

240 \$ 69



High 243.99 | 239.00 Low

Block Time

10.67 Min

Hash Rate

319,729,432

Next Difficulty

14 Days

Market Cap

3.40 B

Volume BTC

4,614

Total BTC

14.12 MM

M1 Rank

115

图1-3 ZeroBlock，一款基于安卓和iOS系统的比特币市场汇率应用程序

Joe使用上述的程序或网站中的其中一个，查到比特币的价格约为每个比特币价值100美元左右。按照这个汇率，Alice给了他10美元，作为交换，他应当给Alice 0.1个比特币，即10,000,000聪。

Joe查到一个市场价后，打开自己的手机钱包应用，选择“发送”比特币。



10:10



BLOCKCHAIN



From

Any Address

To



Amount

0.00

BTC

\$0.00

Send



图1-4 Blockchain 手机钱包的比特币发送界面

例如，如果是在苹果手机上使用Blockchain手机钱包，他会看见屏幕上要求输入：

- ▷ 对方的收款地址
- ▷ 需要发送的比特币数量

在比特币地址的输入栏，有一个看起来像二维码的小图标。Joe可以用他的智能手机摄像头扫描条形码，而不用再输入Alice的比特币地址(1Cdid9KFAatwczBwBtQcwXYCpvK8h7FK)了，不用再打出这个又长又繁琐的地址。Joe轻击二维码图标，启动智能手机的摄像头，然后扫描Alice带来的二维码打印版。手机钱包程序会填好比特币地址，Joe可以通过比较这个地址和Alice打印的地址中的几个数字，来确认已正确地扫描。

接着Joe输入了交易的比特币金额，0.1比特币。他仔细检查，确保已经输入了正确的金额，因为他马上要发送这些钱了，任何一点点的小失误都会导致发送到错误的地址。最后，他按下了发送键来完成这笔交易。Joe的手机钱包会创建一笔交易，将Joe的钱包里的0.10比特币发送到Alice提供的地址，利用Joe的私钥来签名这笔交易。这就公告了比特币全网，Joe已经授权允许从他的一个地址转账比特币到Alice的新地址。由于交易是通过P2P网络协议传输，它会迅速在整个比特币网络传播。不到一秒钟，网络中大多数连接良好的节点都会收到该交易信息，并首次公布Alice的地址。

如果Alice手边有智能手机或笔记本电脑，她也能看到这笔交易。比特币账簿——一个不断膨胀的文件记录，记载了每一笔曾发生过的比特币交易——是公开的，意味着她可以查看所有曾经使用该地址的记录，可以查看是否有人朝这个账户发送了比特币。她只要在blockchain.info网站的搜索框中，输入她自己的地址，就可以轻而易举的知晓了。网页会显示出[该地址所有的交易记录](#)。Joe点击发送后，0.10比特币会很快转到她的钱包里，如果Alice正看这个页面，她就会发现网页更新了这笔新交易信息。

确认

起初，Alice的地址会显示Joe发出这笔的交易为“未确认”，这意味着这笔交易已经被广播到网络，但是尚未列入比特币交易记录账簿，即区块链中。总的来说，交易必须由一个矿工“开采”，交易是包括在区块中的。当新区块创建时，大约十分钟左右，该区域内的交易就会被全网接受为“已确认”，区块中的比特币也都能使用。交易可以立即被其他人看到，但只有当其被包含在新开采的区块中，才是“被信任的”。

现在Alice可以自由支配她所有的0.10个比特币了，感觉很是自豪。在下一章中，我们将看看她第一次使用比特币支付，并会更细致地了解交易和广播的相关技术。

第2章 比特币的原理

2.1 交易、区块、挖矿和区块链

与传统银行和支付系统不同，比特币系统是以去中心化信任为基础的。由于比特币网络中不存在中央权威信任机构，“信任”成为了比特币用户之间存在的一种突出特性。在本章中，我们将从一个较高层面检视比特币，通过追踪一笔比特币系统的单独交易，来看看它在比特币分布式共识机制中变得“被信任”和“被接受”的情形，以及最终成功地被存储到区块链（区块链是一个分布式的公共账簿，包含所有发生在比特币系统中的交易）。

书中每一个例子都是比特币网络中发生的真实交易，通过将资金从一钱包发送到另一钱包来模拟用户（Joe、Alice和Bob）间的交互。我们在追踪一笔通过比特币网络和区块链的交易时，将用到一些区块链数据库查询网站使每个步骤可以方便在网页上直接被呈现。提供区块链数据查询的网站就像是一个比特币的搜索引擎，你可以搜索比特币的地址、交易和区块，以及可以看他们之间的关系和资金流动。

常见的区块链数据查询网站包括：

- ▷ [Blockchain.info](#)
- ▷ [Bitcoin Block Explorer](#)
- ▷ [insight](#)
- ▷ [blockr Block Reader](#)

以上每一个查询网站都有搜索功能，可以通过地址，交易哈希值或区块号，搜索到在比特币网络和区块链中对应的等价数据。我们将给每个例子提供一个链接，可以直接带你到相关条目，方便你做详细研究。

2.1.1 比特币概述



图2-1 比特币概述

如图2-1所示的概述图中，我们可以看到比特币系统由用户（用户通过密钥控制钱包）、交易（每一笔交易都会被广播到整个比特币网络）和矿工（通过竞争计算生成在每个节点达成共识的区块链，区块链是一个分布式的公共权威账簿，包含了比特币网络发生的所有的交易）组成。在本章中，我们将通过追踪在网络上传输的一笔交易，从整个比特币系统的视角检视各个部分之间的交互。后续章节将详细阐述钱包、挖矿、商家系统背后的技术细节。

2.1.2 买咖啡

在之前章节里，Alice是一名刚刚获得第一枚比特币的新用户。在“[1.4.2 获取你的第一枚比特币](#)”一节中，Alice和她的朋友Joe会面时，用现金换取了比特币。由Joe产生的这笔交易使得Alice的钱包拥有了0.10比特币。现在Alice将第一次使用比特币在加利福尼亚州帕罗奥图的Bob咖啡店买一杯咖啡。Bob咖啡店给他的销售网点系统新增加了一个比特币支付选项，价格单上列的是当地货币（美元）的售价，但在收银台，顾客可以选择用美元或比特币支付。此时，Alice点了杯咖啡，然后Bob将交易键入到收银机，之后销售系统将按照当前市场汇率把美元总价转换为比特币，然后同时显示两种货币的价格，并显示一个包含这笔交易支付请求的二维码（如图2-2所示）：



图2-2

总价：

\$1.50 USD

0.0150 BTC

这个二维码中的URL是：

```
bitcoin:1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA?  
amount=0.015&  
label=Bob%27s%20Cafe&  
message=Purchase%20at%20Bob%27s%20Cafe
```

根据BIP0021的定义，此URL的意思是：

```
A bitcoin address: "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"  
The payment amount: "0.015"  
A label for the recipient address: "Bob's Cafe"  
A description for the payment: "Purchase at Bob's Cafe"
```



与一个简单包含目的比特币地址的二维码不同，当前支付请求的是一个二维编码过的URL，它包含有一个目的地址，一笔支付金额，和一个像“Bob咖啡”这样的交易描述。这使比特币钱包应用在发送支付请求时，可以预先填好支付用的特定信息，给用户显示一种友好易懂的描述。你可以用比特币钱包应用查看Alice可能看到的信息。

Bob说到，“总共1.50美元，或15毫比特币”

Alice用她的智能手机扫描了显示的条形码。她的智能手机显示一笔给Bob咖啡的0.0150比特币的支付请求，然后她按下发送键授权了这笔支付。在几秒钟时间内（大约与信用卡授权所需时间相同）Bob将会在收银台看到这笔交易，并完成交易。

在接下来的章节中，我们将更详细地检视这笔交易，观察Alice的钱包是怎样构建交易，交易又是怎样在网络中广播、怎样被验证，以及Bob在后续交易中怎样消费那笔钱。



从千分之一比特币(1毫比特币)到一亿分之一比特币(1聪比特币)，比特币网络可以处理任意小额交易。在本书中，我们将用“比特币”这个术语来表示任意数量的比特币货币，从最小单元(1聪)到可被挖出的所有比特币总数(21,000,000)。

2.2 比特币交易

简单来说，交易告知全网：比特币的持有者已授权把比特币转帐给其他人。而新持有者能够再次授权，转移给该比特币所有权链中的其他人，产生另一笔交易来花掉这些比特币，后面的持有者在花费比特币也是用类似的方式。

交易就像复式记账法账簿中的行。简单来说，每一笔交易包含一个或多个“输入”，输入是针对一个比特币账号的负债。这笔交易的另一面，有一个或多个“输出”，被当成信用点数记入到比特币账户中。这些输入和输出的总额(负债和信用)不需要相等。相反，当输出累加略少于输入量时，两者的差额就代表了一笔隐含的“矿工费”，这也是将交易放进账簿的矿工所收集到的一笔小额支付。如图2-3描述的是一笔作为记账簿记录的比特币交易。

复式记账簿式交易			
输入	值	输出	值
输入1	0.10 BTC	输出1	0.10 BTC
输入2	0.20 BTC	输出2	0.20 BTC
输入3	0.10 BTC	输出3	0.20 BTC
输入4	0.15 BTC		
总输入：	0.55 BTC	总输出：	0.50 BTC
输入 输出 差价	0.55 BTC 0.50 BTC 0.05 BTC (隐含的交易费)		

图2-3

交易也包含了每一笔被转移的比特币(输入)的所有权证明，它以所有者的数字签名形式存在，并可以被任何人独立验证。在比特币术语中，“消费”指的是签署一笔交易：转移一笔以前交易的比特币给以比特币地址所标识的新所有者。



交易是将钱从交易输入移至输出。输入是指钱币的来源，通常是之前一笔交易的输出。交易的输出则是通过关联一个密钥的方式将钱赋予一个新的所有者。目的密钥被称为是安全锁（Encumbrance）。这样就给资金强加了一个要求：有签名才能在以后的交易中赎回资金。一笔交易的输出可以被当做另一笔新交易的输入，这样随着钱从一个地址被移动到另一个地址的同时形成了一条所有权链（如图2-4）。

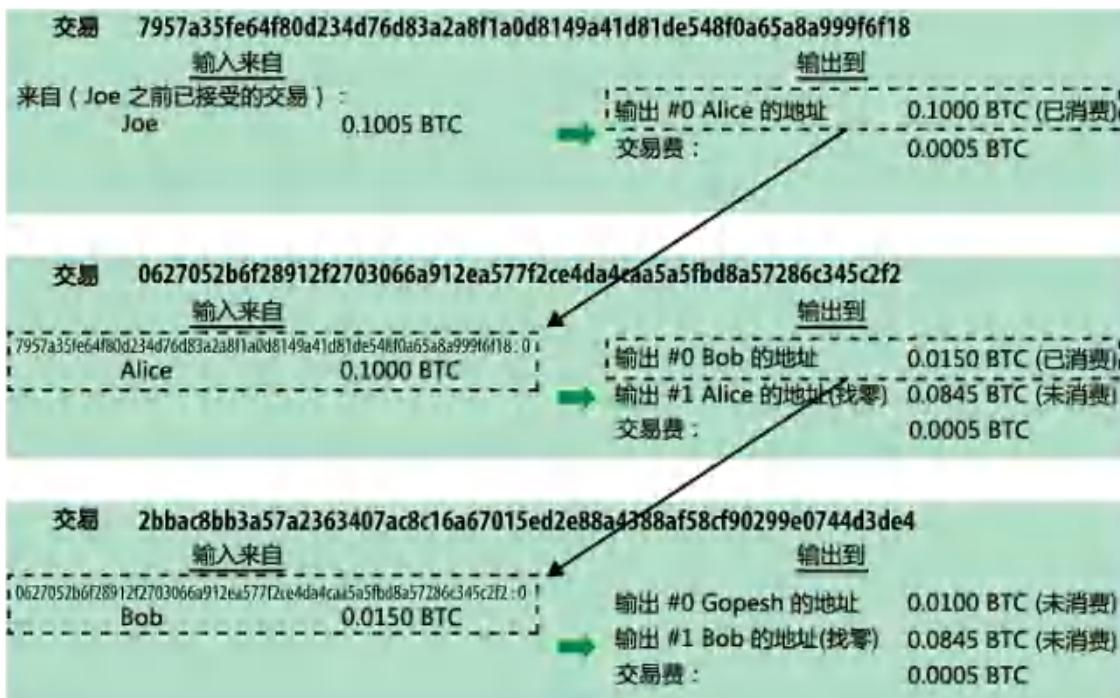


图2-4

Alice支付Bob咖啡时使用一笔之前的交易作为输入。在以前的章节中，Alice从她朋友Joe那里用现金换了点比特币。那笔交易有一些比特币被Alice的密钥锁定（阻塞）。在她支付Bob咖啡店的新交易中使用了之前的交易作为输入，并以支付咖啡和找零作为新的输出。交易形成了一条链，最近交易的输入对应以前交易的输出。Alice的密钥提供了解锁之前交易输出的签名，因此向比特币网络证明她拥有这笔钱。她将咖啡的支付附到Bob的地址上，同时“阻塞”那笔输出，指明要求是Bob签名才能消费这笔钱。这就描述了在Alice和Bob之间钱的转移。上图展示了从Joe到Alice再到Bob的交易链。

2.2.1 常见的交易形式

最常见的交易形式是从一个地址到另一个地址的简单支付，这种交易也常常包含给支付者的“找零”。一般交易有一个输入和两个输出，如图2-5所示：

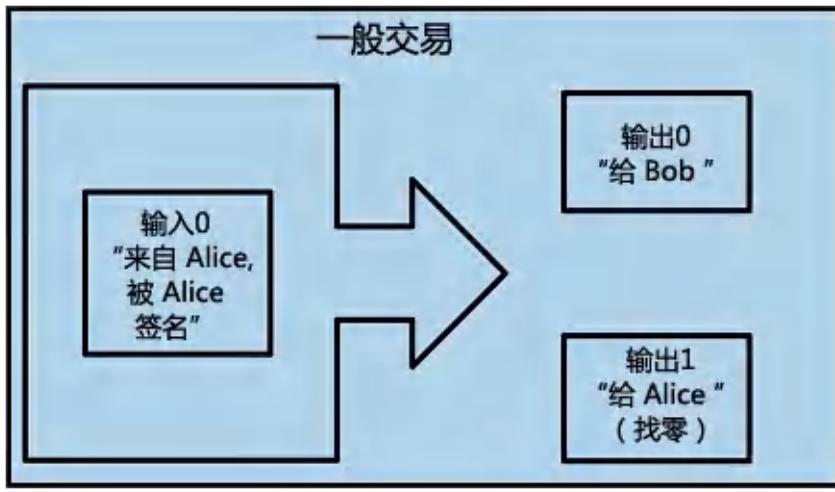


图2-5

另一种常见的交易形式是集合多个输入到一个输出（如图2-6）的模式。这相当于现实生活中将很多硬币和纸币零钱兑换为一个大额面钞。像这样的交易有时由钱包应用产生来清理许多在支付过程收到的小数额的找零。

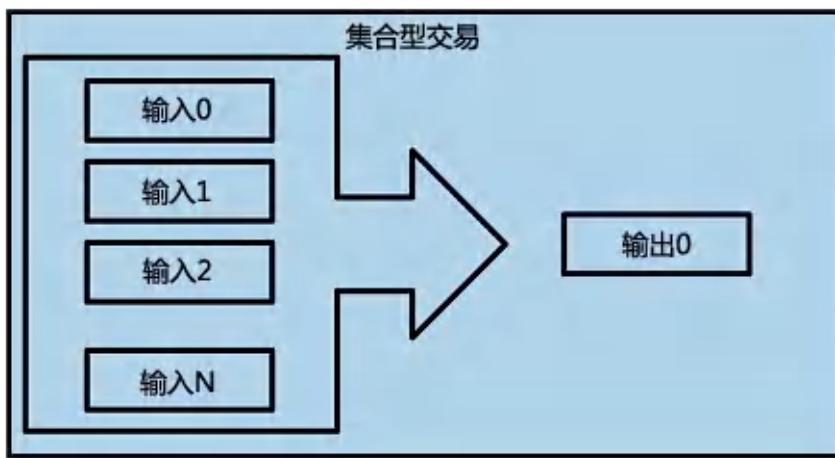


图2-6

最后，另一种在比特币账簿中常见的交易形式是将一个输入分配给多个输出，即多个接收者（如图2-7）的交易。这类交易有时被商业实体用作分配资金，例如给多个雇员发工资的情形。

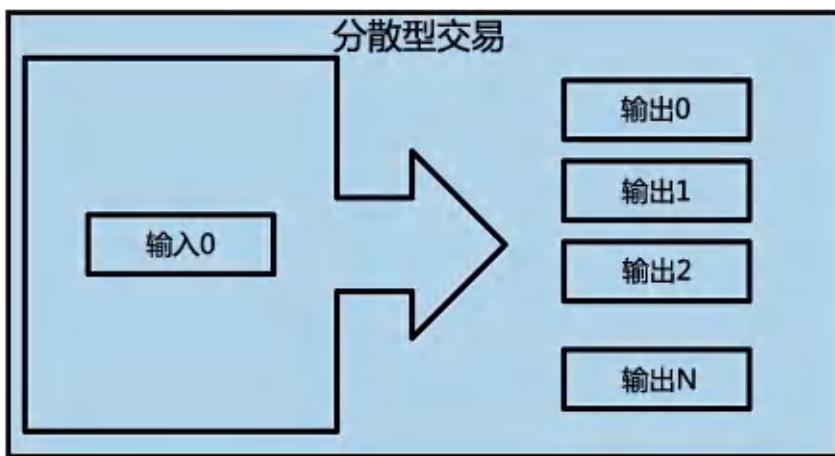


图2-7

2.3 交易的构建

Alice的钱包应用知道如何选取合适的输入和输出以建立Alice所希望的交易。Alice只需要指定目标地址和金额，其余的细节钱包应用会在后台自动完成。很重要的一点是，钱包应用甚至可以在完全离线时建立交易。就像在家里写张支票，之后放到信封发给银行一样，比特币交易建立和签名时不用连接比特币网络。只有在执行交易时才需要将交易发送到网络。

2.3.1 获取正确的输入

Alice的钱包应用首先要找到一些足够支付给Bob所需金额的输入。大多数钱包应用维护着一个含有用钱包自己密钥锁定的“未消费交易输出”小型数据库。因此Alice的钱包会包含她用现金从Joe那里购买的比特币的交易输出副本（参见在“[1.4.2 获取你的第一枚比特币](#)”一节）。完整客户端含有整个区块链中所有交易的所有未消费输出副本。这使得钱包即能拿这些输出构建交易，又能在收到新交易时很快地验证其输入是否正确。然而，完整客户端占太大的硬盘空间，所以大多数钱包使用轻量级的客户端，只保存用户自己的未消费输出。

如果钱包客户端没有某一未消费交易输出，它可以通过不同的服务者提供的各种API或完整索引节点的JSON PRC API从比特币网络中拿到这一交易信息。例子2-1展示了用HTTP GET命令对一个特定URL建立了一个RESTful API的请求。这个URL会返回一个地址的所有未消费交易输出，以提供给需要这些信息的任何应用作为建立新交易的输入而进行消费。我们用一个简单的HTTP命令行客户端 curl 来获得这个响应数据。

例2-1 查找Alice的比特币地址所有的未消费的输出

```
$ curl https://blockchain.info/unspent?active=1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK
```

例2-2 查找返回的响应数据

```
{
  "unspent_outputs": [
    {
      "tx_hash": "186f9f998a5...2836dd734d2804fe65fa35779",
      "tx_index": 104810202,
      "tx_output_n": 0,
      "script": "76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
      "value": 10000000,
      "value_hex": "00989680",
      "confirmations": 0
    }
  ]
}
```

例2-2的响应数据显示了在Alice的地址 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK 上面有一个未消费输出（还未被兑换）。这个响应包含一个交易的索引。而从Joe那里转过来的未消费输入就包含在这个交易里面，它的价值是一千万聪（satoshi），即 0.10 比特币。通过这个信息，Alice的钱包应用就可以创建新的交易将钱转账到新地址。



[点击查看Joe和Alice间的交易信息](#)

如你所见，Alice的钱包在单个未消费的输出中有足够的比特币支付一杯咖啡。假如不够的话，Alice的钱包应用就不得不搜寻一些小的未消费输出，像是从一个存钱罐里找硬币一样，直到找到足够支付咖啡的数量。在两种情境下，可能都需要找回零钱，而这些找零也会是钱包所创建的交易的输出组成部分。我们会在下一节会有所描述。

2.3.2 创建交易输出

交易的输出会被创建成为一个包含这笔数额的脚本的形式，只能被引入这个脚本的一个解答后才能兑换。简单点说就是，Alice的交易输出会包含一个脚本，这个脚本说“这个输出谁能拿出一个签名和Bob的公开地址匹配上，就支付给谁”。因为只有Bob的钱包可以匹配这个地址，所以只有Bob的钱包可以提供这个签名以兑换这笔输出。因此Alice会用需要Bob的签名来包装一个输出。

这个交易还会包含第二个输出。因为Alice的金额是0.10比特币的输出形式，对0.015比特币一杯的咖啡来说太多了，需要找Alice 0.085比特币的零钱。Alice钱包创建给她的零钱的支付就在付给Bob的同一个交易里面。可以说，Alice的钱包将她的金额分成了两个支付：一个给Bob，一个给自己。她可以在以后的交易里消费这笔零钱输出。

最后，为了让这笔交易尽快地被网络处理，Alice的钱包会多付一小笔费用。这个不是明显地包含在交易中的；而是通过输入和输出的差值所隐含的。如果Alice创建找零时只找 0.0845比特币，而不是 0.085比特币的话，这里就有剩下 0.0005比特币（50万聪）。因为加起来小到 0.10，所以这个 0.10 比特币的输入就没有被完整的消费了。这个差值会就被矿工当作交易费放到区块的交易里，最终放进区块链帐簿中。

这个交易的结果信息可以用区块链数据查询站点看到，如图2-8所示。

交易记录 比特币交易的相关信息

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbda57286c345c2f2

1CdId9KFAaatwczBwBttQcwXYCpvK8h7FK	1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA 0.015 BTC
	1CdId9KFAaatwczBwBttQcwXYCpvK8h7FK 0.0845 BTC
	0.0995 BTC

概览	
数据量	258 (字节)
接收时间	2013-12-27 23:03:05
接纳区块	277316 (2013-12-27 23:11:54 + 9 分钟)
确认	74648 确认

转入与转出	
转入总额	0.1 BTC
转出总额	0.0995 BTC
交易费	0.0005 BTC
估计交易金额	0.015 BTC

图2-8

 点击查看Alice支付Bob咖啡的交易的信息

2.3.3 将交易放到总账簿中

这个被Alice钱包应用创建的交易大小为258字节，包含了金额未来所属需要的全部信息。现在，这个交易必须要被传送到比特币网络中以成为分布式账簿（区块链）的一部分。在下一节里，我们来看下一个交易如何成为新区块的一部分，以及区块是如何被挖矿构建的。最后，我们会看看新区块被加进区块链后，是如何随更多区块的添加而增加可信度的。

2.3.3.1 交易的传送

因为这个交易包含处理所需的所有信息，所以这个交易是被如何或从哪里传送到比特币网络的就无所谓了。比特币网络是由参与的比特币客户端联接几个其他比特币客户端组成的P2P网络。比特币网络的目的是将交易和区块传播给所有参与者。

2.3.3.2 如何传播

Alice的钱包应用可以发送新的交易给其它任意一个已联接到互联网的比特币客户端，不论其是由有线网、WiFi、还是通过手机联接的。她的钱包不必直接连着Bob的比特币钱包，且她不必使用咖啡厅提供的网络联网，虽然这两者都是可能的。任何比特币网络节点（其它客户端）收到一个之前没见过的有效交易时会立刻将它转发给联接到自身的其它节点。因此，这个交易迅速地从P2P网络中传播开来，几秒内就能到达大多数节点。

2.3.3.3 Bob的视角

如果Bob的比特币钱包应用是直接连接Alice的钱包应用的话，Bob的钱包应用也许就是第一个收到这个交易的节点。然而，即使Alice的交易是从通过其它节点发过来的，一样可以在几秒钟内到达Bob钱包应用这里。Bob的钱包会立即确认Alice的交易是一个收入支付，因为它包含能用Bob的私钥兑换的输出。Bob的钱包应用也能够独立地用之前未消费输入来确认这个交易是正确构建的，并且由于包含足够交易费会被下一个区块包含进去。这时Bob就可以以一个很小的风险假定这个交易会很快被加到区块且被确认。



一个对比特币交易的常见误解是它们必须要等10分钟后被确认加进一个新区块，或等60分钟以得到六次确认后才是有效的。虽然这些确认可以确保交易已被整个网络接受，但对于像一杯咖啡这样的小额商品来说就没有必要等待那么长时间了。一个商家可以免确认来接受比特币小额支付。这样做的风险不比接受一个不是用有效身份证件领取或没有签名的信用卡的风险更大，而后者是现在商家常做的事情。

2.4 比特币挖矿

这个交易现在在比特币网络传播开来。但只有被一个称为挖矿的过程验证且加进一个区块之后，这个交易才会成为这个共享账簿（区块链）的一部分。关于挖矿的详细描述请见[第8章](#)。

比特币系统的信任是建立在计算的基础上的。交易被包在一起放进区块中时需要极大的计算量来证明，但只需少量计算就能验证它们已被证明。挖矿在比特币系统中起着两个作用：

▷ 挖矿在构建区块时会创造新的比特币，和一个中央银行印发新的纸币很类似。每个区块创造的比特币数量是固定的，随时间会渐渐减少。

▷ 挖矿创建信任。挖矿确保只有在包含交易的区块上贡献了足够的计算量后，这些交易才被确认。区块越多，花费的计算量越大，意味着更多的信任。

描述挖矿的一个好方法是将之类比为一个巨大的多人数独谜题游戏。一旦有人发现正解之后，这个数独游戏会自动调整困难度以使游戏每次需要大约10分钟解决。想象一个有几千行几千列的巨大数独游戏。如果给你一个已经完成的数独，你可以很快地验证它。然而，如果这个数独只有几个方格里有数字其余方格都为空的话，就会花费非常长的时间来解决。这个数独游戏的困难度可以通过改变其大小（更多或更少行列）来调整，但即使它非常大时验证它也是相当容易的。而比特币中的“谜题”是基于哈希加密算法的，其展现了相似的特性：非对称地，它解起来困难而验证很容易，并且它的困难度可以调整。

在[“1.3 比特币的应用、用户和他们的故事”](#)一节中，我们提到了一个叫Jing的在上海学计算机工程的学生。Jing在比特币网络中扮演了一个矿工的角色。大概每10分钟，Jing和其他上千个矿工一起展开一场对一个区块的交易寻找正解的全球竞赛。为寻找这个解，也被称为工作量证明，整个网络需要具有每秒亿万次哈希计算的能力。这个工作量证明算法指的用SHA256加密算法不断地对区块头和一个随机数字进行哈希计算，直到出现一个和预设值相匹配的解。第一个找到这个解的矿工会赢得这局竞赛并会将此区块发布到区块链中。

Jing从2010年开始挖矿，当时他使用一个非常快的桌面电脑来为新区块寻找正解。随着更多的矿工加入比特币网络中，寻找谜题正解的困难度迅速增大。不久，Jing和其他矿工升级成更专业的硬件，比如游戏桌面电脑或控制台专用的高端独享图像处理单元芯片（即显卡GPU）。在写这本书的时候，解题已经变得极其困难，只有使用集成了几百个挖矿专用算法硬件并能同时在一个单独芯片上并行工作的专用集成电路（ASIC）挖矿才会营利。Jing同时加入了一个类似彩票奖池的、能够让多个矿工共享计算力和报酬的矿池。Jing现在运行两个通过USB联接的ASIC机器每天24小时不间断地挖矿。他卖掉一些挖矿所得到的比特币来支付电费，并可以通过营利获得一些收入。作为专用挖矿软件的后台，他的计算机里安装了一个比特币索引客户端，名称为bitcoind。

2.5 区块中的挖矿交易记录

网络中产生的一笔交易直到成为整个比特币大账簿——区块链的一部分时才会被确认有效。平均每10分钟，矿工会将自上一个区块以来发生的所有交易生成一个新的区块。新交易不断地从用户钱包和应用流入比特币网络。当比特币网络上的节点看到这些交易时，会先将它们放到各自节点维护的一个临时的未经验证的交易池中。当矿工构建一个新区块时，会将这些交易从这个交易池中拿出来放到这个新区块中，然后通过尝试解决一个非常困难的问题（也叫工作量证明）以证明这个新区块的合法性。挖矿过程的细节会在“[8.1 简介](#)”一节中详加描述。

这些交易被加进新区块时，以交易费用高的优先以及其它的一些规则进行排序。矿工一旦从网络上收到一个新区块时，会意识到在这个区块上的解题竞赛已经输掉了，会马上开始下一个新区块的挖掘工作。它会立刻将一些交易和这个新区块的数字指纹放在一起开始构建下一个新区块，并开始给它计算工作量证明。每个矿工会在他的区块中包含一个特殊的交易，将新生成的比特币（当前每区块为25比特币）作为报酬支付到他自己的比特币地址。如果他找到了使得新区块有效的解法，他就会得到这笔报酬，因为这个新区块被加入了总区块链中，他添加的这笔报酬交易也会变成可消费的。参与矿池的Jing设置了他的软件，使得构建新区块时会将报酬地址设为矿池的地址。然后根据各自上一轮贡献的工作量将所得的报酬分给Jing和其他参与矿池挖矿的矿工。

Alice的交易被网络拿到后放进未验证交易池中。因为包含足够的交易费，它被Jing的矿池放到了生成的新区块中。大约在Alice的钱包第一次将这个交易发送出来五分钟后，Jing的ASIC矿机发现了新区块的正解并将之发布为第277,316号区块，包含419个其它交易。Jing的ASIC矿机将这个新区块发布到网络上后，其它矿机就会验证它，并投身到构建新区块的竞争中。

你可以查看包含[Alice交易记录](#)的这个区块的信息。

几分钟后，第277,317号新区块诞生在另一个挖矿节点中。因为这个新区块是基于包含Alice交易的第277,316号区块的，在这个区块的基础上增加了更多的计算，因此就加强了这些交易的可信度。包含Alice交易的区块对这个交易来说算一次“证明”。基于这个区块每产生一个新区块，对这个交易来说就会增加了一次“证明”。当区块一个个堆上来时，这个交易变得指数级地越来越难被推翻，因此它在网络中得到更多信任。

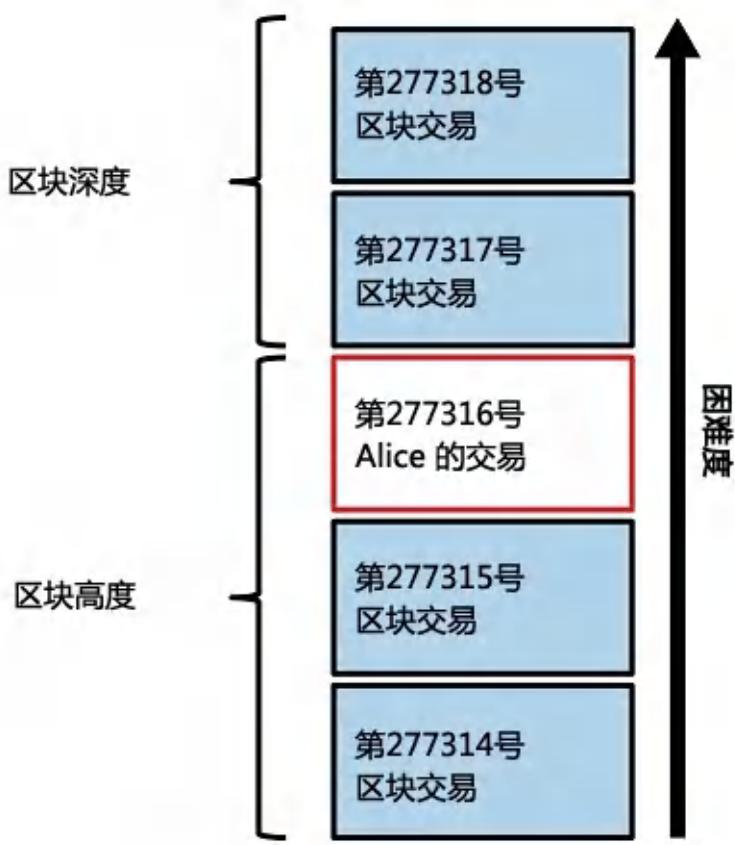


图2-9

在图2-9中，我们可以看到包含Alice的交易的第277,316号区块。在它之下有377,361个区块（包括0号区块），像链子一样一个连着一个（区块链），一直连到0号区块，即创世区块。随着时间变长，这个区块链的高度也随之增长，每个区块和整个链的计算复杂度也随之增加。包含Alice的交易的区块后面形成的新区块使得信任度进一步增加，因为他们叠加了更多的计算在这个越来越长的链子上。按惯例来说，一个区块获得六次以上“证明”时就被认为是不可撤销的了，因为要撤销和重建六个区块需要巨量的计算。在[第8章](#)我们会详细描述挖矿和信任建立的过程。

2.6 消费这笔交易

既然Alice的这笔交易已经成为区块的一部分被嵌入到了区块链中，它就成为了整个分布式比特币账簿的一部分并对所有比特币客户端应用可见。每个比特币客户端都能独立地验证这笔交易是有效且可消费的。全索引客户端可以追钱款的来源，从第一次有比特币在区块里生成的那一刻开始，按交易与交易间的关系顺藤摸瓜，直到Bob的交易地址。轻量级客户端通过确认一个交易在区块链中且在它后面有几个新区块来确认一个支付的合法性。这种方式叫做简易支付验证（参见[“6.7 简易支付验证（SPV）节点”](#)）。

Bob现在可以将此交易和其它交易的结果信息作为输入，创建新的所有权为其他人的交易。这样就实现了对此交易的消费。举个例子，Bob可以用Alice支付咖啡的比特币转账给承包商或供应商以支付相应费用。大多数情况下，Bob用的比特币商户端会将多个小额支付聚合成一个大的支付，也许会将一整天的比特币收入聚合成一个交易。这样会将多个支付合成到咖啡店财务账户的一个单独地址。图2-6为交易集合示例。

当Bob花费从Alice和其他顾客那里赚得的比特币时，他就扩展了比特币的交易链条。而这个链条会被加到整个区块链账簿，使所有人知晓并信任。我们假定Bob向在邦加罗尔的网站设计师Gopesh支付一个新网页的设计费用。那么区块交易链会如图2-10所示。

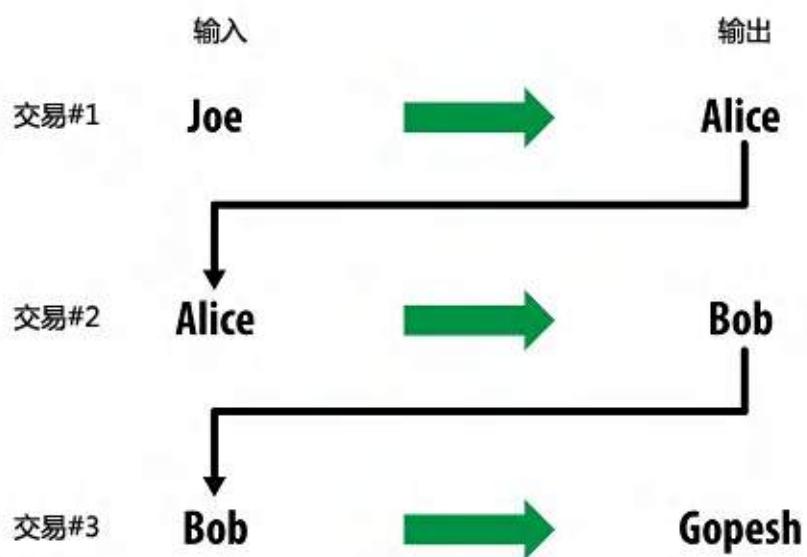


图2-10

第3章 比特币客户端

3.1 比特币核心：参考实现

你可以从bitcoin.org下载标准客户端，即比特币核心，也叫“中本聪客户端”（satoshi client）。它实现了比特币系统的所有方面，包括钱包、对整个交易账簿（区块链）完整拷贝的交易确认引擎，和点对点比特币网络中的一个完整网络节点。

在[Bitcoin网站的选择钱包页面](https://bitcoin.org)，下载参考客户端比特币核心。根据操作系统不同，你将下载不同的可执行安装工具。对于Windows，它是一个ZIP归档文件或.exe格式的可执行文件。对于Mac OS，它是一个.dmg格式的磁盘映像。Linux版本包括用于Ubuntu系统的PPA包，或是tar.gz格式的压缩包。图3-1所示的Bitcoin.org页面上列出了一些推荐的比特币客户端。



图3-1 选择比特币客户端

3.1.1 第一次运行比特币核心

如果你下载了一个安装包，比如.exe、.dmg、或PPA，你可以和安装其它任何应用程序一样，在你的操作系统上安装它。对于Windows，运行.exe文件，按照提示一步步操作。对于Mac OS，启动.dmg文件，然后将Bitcoin-QT图标拖拽到你的应用程序目录就可以了。对于Ubuntu，在文件资源管理器中双击PPA文件，会打开程序包管理器来安装它。一旦完成了安装，在你的应用程序列表中会有一个新的应用叫Bitcoin-QT。双击这个图标就可以启动比特币客户端了。

第一次运行比特币核心时，它会开始下载整个区块链，这个过程可能需要数天（见下图）。让它在后台运行，直到显示“已同步”，并且余额旁边不再显示“数据同步中”。



比特币核心拥有交易账簿（区块链）的一份完整拷贝，里面记录了自2009年比特币网络被发明以来发生在比特币网络上的每一笔交易。这个数据集有几个GB（在2013年底大约是16GB），并且需要几天的时间完成增量形式的下载（从区块0顺次下载到最新区块）。在整个区块链数据集下载完成前，客户端将不能处理任何交易或是更新账户余额。在下载数据集的过程中，客户端账户余额旁会显示“数据同步中”，底部会显示“正在同步”。请确保你有足够的磁盘空间、带宽和时间来完成初始同步。

3.1.2 从源码编译比特币核心比特币核心

对于开发者，可以选择下载包含完整源代码的ZIP包，也可以从Github上克隆权威的源码仓库。在面[GitHub上的比特币页面](#)，在侧边栏选择下载ZIP。或者使用git命令行（git command line）在自己系统上创建源码的本地拷贝。在下面的例子中，我们将通过unix风格的命令行，在Linux或是Mac OS 上克隆源代码：

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 31864, done.
remote: Compressing objects: 100% (12007/12007), done.
remote: Total 31864 (delta 24480), reused 26530 (delta 19621)
Receiving objects: 100% (31864/31864), 18.47 MiB | 119 KiB/s, done.
Resolving deltas: 100% (24480/24480), done.
$
```



终端上的提示和输出结果可能会因版本有所不同。即使你的屏幕上输出的内容跟这里的例子有点不一样，请遵照代码中的文档，这些都是正常的。

在git clone操作完成后，在你本地的bitcoin目录就会有一份完整的源码拷贝。通过在命令提示行输入cd bitcoin切换到这个目录下：

```
$ cd bitcoin
```

默认情况下，本地拷贝将与最新的代码同步，这可能是bitcoin的一个不稳定或是beta版本。在编译这些代码之前，签出发布标签（release tag）以选择某一特定版本（a specific version）。这将通过一个关键的标签标记，让本地拷贝与代码仓库的特定快照同步。开发者用标签来标记代码的特定发行版本号（version numBTCer）。首先，要找到可用的标签，可以通过git tag命令：

```
$ git tag
v0.1.5
v0.1.6test1
v0.2.0
v0.2.10
v0.2.11
v0.2.12

[... many more tags ...]

v0.8.4rc2
v0.8.5
v0.8.6
```

```
v0.8.6rc1  
v0.9.0rc1
```

列出的标签是bitcoin的所有发行版本。按照约定，带有rc后缀的是预发行版本，可以用来测试。没有后缀的稳定版本可以直接在产品环境上运行。从上面的列表中，选择最新的发行版本，目前是v0.9.0rc1。为了让本地代码跟这个版本一致，我们需要用git checkout命令：

```
$ git checkout v0.9.0rc1  
Note: checking out 'v0.9.0rc1'.  
  
HEAD is now at 15ec451... Merge pull request #3605  
$
```

源代码包含文档，可以在多个文件夹中找到。在命令提示行输入more README.md可以在bitcoin目录下的README.md中查看主文档，用空格键可以翻页。在这一章，我们将构建命令行的比特币客户端，在linux上称作bitcoind。在您的平台上，通过输入more doc/build-unix.md，可以阅读编译bitcoind命令行客户端的说明。Mac OSX和Windows平台的说明可以在doc目录下找到，分别是build-osx.md或是build-msw.md。

仔细阅读build文档第一部分中build的必备条件。这些是在你编译之前你的系统上必须具备的库文件。如果缺少这些必备条件，构建过程将会出现错误并导致失败。如果因为缺少一个必备条件而发生这种情况，你可以先安装它，然后在你停下的地方重新构建。假设这些必备条件已经具备，你就可以开始构建过程，通过autogen.sh脚本，生成一组构建脚本。



从v0.9版本开始，比特币核心的构建过程改用autogen/configure/make体系。旧版本是用一个简单的Makefile，并且和下面的例子稍微有一点不同。请遵照你想要编译版本的说明文档。v0.9版本引入的 autogen/configure/make 构建体系可能用于未来所有版本的代码，是下面的例子中演示的系统

```
$ ./autogen.sh  
configure.ac:12: installing `src/build-aux/config.guess'  
configure.ac:12: installing `src/build-aux/config.sub'  
configure.ac:37: installing `src/build-aux/install-sh'  
configure.ac:37: installing `src/build-aux/missing'  
src/Makefile.am: installing `src/build-aux/depcomp'  
$
```

autogen.sh脚本创建了一系列的自动配置脚本，会询问你的系统以发现正确的设置，确保你已安装必要的库来编译源码。这里面最重要的是configure脚本，它会提供许多不同的选项来定制构建过程。输入./configure --help 可以查看各种不同的选项：

```
$ ./configure --help  
  
'configure' configures Bitcoin Core 0.9.0 to adapt to many kinds of systems.  
  
Usage: ./configure [OPTION]... [VAR=VALUE]...  
  
To assign environment variables (e.g., CC, CFLAGS...), specify them as VAR=VALUE.  
  
See below for descriptions of some of the useful variables.  
  
Defaults for the options are specified in brackets.  
  
Configuration:  
-h, --help           display this help and exit  
--help=short         display options specific to this package
```

```
--help=recursive      display the short help of all the included packages
-V, --version        display version information and exit

[... many more options and variables are displayed below ...]

Optional Features:
--disable-option-checking  ignore unrecognized --enable/--with options
--disable-FEATURE        do not include FEATURE (same as --enable-FEATURE=no)
--enable-FEATURE[=ARG]    include FEATURE [ARG=yes]

[... more options ...]

Use these variables to override the choices made by `configure' or to help it to find libraries and programs with nonstandard na

Report bugs to <info@bitcoin.org>.

$
```

通过使用`--enable-FEATURE`和`--disable-FEATURE`选项，`configure`脚本允许你启用或是禁用某些功能，`FEATURE`需要被替换成功能名称，在上面的帮助输出中可以找到。在这一章，我们将用默认的功能来构建`bitcoind`客户端。这里不会使用配置选项，但是你应该检查一遍，明白哪些可选的功能可以作为客户端的一部分。下一次，运行`configure`脚本就可以自动发现所必要的库，然后为我们的系统创建一个定制的构建脚本。

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes

[... many more system features are tested ...]

configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/test/Makefile
config.status: creating src/qt/Makefile
config.status: creating src/qt/test/Makefile
config.status: creating share/setup.nsi
config.status: creating share/qt/Info.plist
config.status: creating qa/pull-tester/run-bitcoind-for-test.sh
config.status: creating qa/pull-tester/build-tests.sh
config.status: creating src/bitcoin-config.h
config.status: executing depfiles commands
$
```

如果一切顺利，`configure`命令将会以创建可定制的构建脚本结束。这些构建脚本允许我们编译`bitcoind`。如果有缺失的库或是错误，`configur`命令将会以错误信息终止。如果出现了错误，可能是因为缺少库或是有不兼容的库。重新检查构建文档，确认你已经安装缺失的必备条件。然后运行`configure`，看看错误是否消失了。下一步，你将编译源代码，这个过程可能需要1个小时完成。在编译的过程中，你应该过几秒或是几分钟看一下输出结果。如果出现了问题，你会看到错误。如果中断了，编译的过程可以在任何时候恢复。输入`make`命令就可以开始编译了：

```
$ make
Making all in src
make[1]: Entering directory `/home/ubuntu/bitcoin/src'
make  all-recursiv
make[2]: Entering directory `/home/ubuntu/bitcoin/src'
Making all in .
make[3]: Entering directory `/home/ubuntu/bitcoin/src'
      CXX  addrman.o
      CXX  alert.o
      CXX  rpcserver.o
      CXX  bloom.o
```

```
CXX     chainparams.o  
[... many more compilation messages follow ...]  
  
CXX     test_bitcoin-wallet_tests.o  
CXX     test_bitcoin-rpc_wallet_tests.o  
CXXLD  test_bitcoin  
make[4]: Leaving directory `/home/ubuntu/bitcoin/src/test'  
make[3]: Leaving directory `/home/ubuntu/bitcoin/src/test'  
make[2]: Leaving directory `/home/ubuntu/bitcoin/src'  
make[1]: Leaving directory `/home/ubuntu/bitcoin/src'  
make[1]: Entering directory `/home/ubuntu/bitcoin'  
make[1]: Nothing to be done for `all-am'.  
make[1]: Leaving directory `/home/ubuntu/bitcoin'  
$
```

如果一切顺利，`bitcoind`现在已经编译完成。最后一步就是通过`make`命令，安装`bitcoind`可执行文件到你的系统路径下：

```
$ sudo make install  
Making install in src  
Making install in .  
 /bin/mkdir -p '/usr/local/bin'  
 /usr/bin/install -c bitcoind bitcoin-cli '/usr/local/bin'  
Making install in test  
make install-am  
 /bin/mkdir -p '/usr/local/bin'  
 /usr/bin/install -c test_bitcoin '/usr/local/bin'  
$
```

你可以通过询问系统下面2个可执行文件的路径，来确认`bitcoin`是否安装成功。

```
$ which bitcoind  
/usr/local/bin/bitcoind  
  
$ which bitcoin-cli  
/usr/local/bin/bitcoin-cli
```

`bitcoind`默认的安装位置是`/usr/local/bin`。当你第一次运行`bitcoind`时，它会提醒你用一个安全密码给JSON-RPC接口创建一个配置文件。通过在终端输入`bitcoind`就可以运行`bitcoind`了：

```
$ bitcoind  
Error: To use the "-server" option, you must set a rpcpassword in the configuration file:  
/home/ubuntu/.bitcoin/bitcoin.conf  
It is recommended you use the following random password:  
rpcuser=bitcoinrpc  
rpcpassword=2XA4DuKNCbtZXsBQRNDEwEY2nM6M4H9Tx5dFjoAVVbK  
(you do not need to remember this password)  
The username and password MUST NOT be the same.  
If the file does not exist, create it with owner-readable-only file permissions.  
It is also recommended to set alertnotify so you are notified of problems;  
for example: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

在你喜欢的编辑器中编辑配置文件并设置参数，将其中的密码替换成`bitcoind`推荐的强密码。不要使用出现在这里的密码。在`.bitcoin`目录下创建一个名为`.bitcoin/bitcoin.conf`的文件，然后输入用户名和密码：

```
rpcuser=bitcoinrpc  
rpcpassword=2XA4DuKNCbtZXsBQRNDEwEY2nM6M4H9Tx5dFjoAVVbK
```

当你正在编辑配置文件的时候，你可能想要设置一些其他选项，例如`txindex`（见“[交易数据库索引及txindex选项](#)”）。通过输入`bitcoind --help`，可以查看所有可用的选项列表。

现在可以运行比特币核心客户端。当你第一次运行的时候，它会下载所有的区块，重新构建比特币区块链。这是一个好几个GB的文件，可能需要大约2天的时间全部下载完。你可以通过[SourceForge](#)上的BitTorrent客户端下载区块链的部分拷贝来缩短区块链的初始化时间。

选项 -daemon 可以以后台模式运行 bitcoind。

```
$ bitcoind -daemon

Bitcoin version v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
Using OpenSSL version OpenSSL 1.0.1c 10 May 2012
Default data directory /home/bitcoin/.bitcoin
Using data directory /bitcoin/
Using at most 4 connections (1024 file descriptors available)
init message: Verifying wallet...
dbenv.open LogDir=/bitcoin/database ErrorFile=/bitcoin/db.log
Bound to [::]:8333
Bound to 0.0.0.0:8333
init message: Loading block index...
Opening LevelDB in /bitcoin/blocks/index
Opened LevelDB successfully
Opening LevelDB in /bitcoin/chainstate
Opened LevelDB successfully

[... more startup messages ...]
```

3.2 通过命令行使用比特币核心的JSON-RPC API接口

比特币核心客户端实现了JSON-RPC接口，这个接口也可以通过命令行帮助程序bitcoin-cli访问。命令行可以使用API进行编程，让我们有能力进行交互实验。开始前，调用help命令查看可用的比特币RPC命令列表：

```
$ bitcoin-cli help
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decoderawtransaction "hexstring"
decodescript "hex"
dumpprivkey "bitcoinaddress"
dumpwallet "filename"
getaccount "bitcoinaddress"
getaccountaddress "account"
getaddednodeinfo dns ( "node" )
getaddressesbyaccount "account"
getbalance ( "account" minconf )
getbestblockhash
getblock "hash" ( verbose )
getblockchaininfo
getblockcount
getblockhash index
getblocktemplate ( "jsonrequestobject" )
getconnectioncount
getdifficulty
getgenerate
gethashespersec
getinfo
getmininginfo
getnettotals
getnetworkhashps ( blocks height )
getnetworkinfo
getnewaddress ( "account" )
getpeerinfo
getrawchangeaddress
getrawmempool ( verbose )
getrawtransaction "txid" ( verbose )
getreceivedbyaccount "account" ( minconf )
getreceivedbyaddress "bitcoinaddress" ( minconf )
gettransaction "txid"
```

```
gettxout "txid" n ( includemempool )
gettxoutsetinfo
getunconfirmedbalance
getwalletinfo
getwork ( "data" )
help ( "command" )
importprivkey "bitcoinprivkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts ( minconf )
listaddressgroupings
listlockunspent
listreceivedbyaccount ( minconf includeempty )
listreceivedbyaddress ( minconf includeempty )
listsinceblock ( "blockhash" target-confirmations )
listtransactions ( "account" count from )
listunspent ( minconf maxconf ["address",...])
lockunspent unlock [{"txid":"txid","vout":n},...]
move "fromaccount" "toaccount" amount ( minconf "comment" )
ping
sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "commentto" )
)
sendmany "fromaccount" {"address":amount,...} ( minconf "comment" )
sendrawtransaction "hexstring" ( allowhighfees )
sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" )
setaccount "bitcoinaddress" "account"
setgenerate generate ( genproclimit )
settxfee amount
signmessage "bitcoinaddress" "message"
signrawtransaction "hexstring" ( [{"txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex"},...] [{"privatekey1",...}] sighashtype )
stop
submitblock "hexdata" ( "jsonparametersobject" )
validateaddress "bitcoinaddress"
verifychain ( checklevel numBTClocks )
verifymessage "bitcoinaddress" "signature" "message"
walletlock
wallet passphrase "passphrase" timeout
wallet passphrasechange "oldpassphrase" "newpassphrase"
```

3.2.1 获得比特币核心客户端状态的信息

命令： `getinfo`

比特币 `getinfo` RPC命令显示关于比特币网络节点、钱包、区块链数据库状态的基础信息。使用 `bitcoin-cli` 运行它：

```
$ bitcoin-cli getinfo
{
"version" : 90000,
"protocolversion" : 70002,
>walletversion" : 60000,
"balance" : 0.00000000,
"blocks" : 286216,
"timeoffset" : -72,
"connections" : 4,
"proxy" : "",
"difficulty" : 2621404453.06461525,
"testnet" : false,
"keypoololdest" : 1374553827,
"keypoolsize" : 101,
"paytxfee" : 0.00000000,
"errors" : ""
}
```

数据以JSON格式显示，JSON是一种可以很容易被编程语言“消耗”，但同时对人类可读性也很高的格式。在这些数据中，我们看到比特币软件客户端的版本编号（90000），协议编号（70002），钱包编号（60000）。我们看到钱包中的当前余额是0。我们看到当前的区块高度，这可以告诉我们有多少区块对此客户端已知（286216）。我们同样看到关于比特币网络和关于此客户端的各种数据。我们将在其他章节中更具体地探索这些设置。



bitcoind客户端需要花费可能超过一天的时间从其他比特币客户端下载区块以“赶上”当前区块链高度。你可以使用 `getinfo` 命令查看已知区块的数字以检查同步进度。

3.2.2 钱包设置及加密

命令：`encryptwallet`、`walletpassphrase`

在你向前生成秘钥和其他命令之前，你应当先用密码加密钱包。对于本例，将使用 `encryptwallet` 命令，密码为“foo”。很明显，在你自己操作的时候记得使用强且复杂的密码替换“foo”。

```
$ bitcoin-cli encryptwallet foo
wallet encrypted; Bitcoin server stopping, restart to run with encrypted wallet.
The keypool has been flushed, you need to make a new backup.
$
```

你可以再次使用 `getinfo` 命令以验证钱包是否已经加密。这次你将发现有个叫做 `unlocked_until` 的新条目。这是一个计数器，告诉你保持钱包处于解锁状态的解密密码将在内存中存储多久。最初计数器设置为0，意味着钱包是被锁定的：

```
$ bitcoin-cli getinfo
{
    "version" : 90000,
#[... other information...]
    "unlocked_until" : 0,
    "errors" : ""
}
$
```

想解锁钱包，要使用 `walletpassphrase` 命令。`walletpassphrase` 命令需要两个参数——密码，和多久钱包会再次被自动锁定的秒数数字（计时器）：

```
$ bitcoin-cli walletpassphrase foo 360
$
```

你可以确认钱包是解锁状态然后通过再次运行 `getinfo` 查看超过时限：

```
$ bitcoin-cli getinfo
{
    "version" : 90000,
#[... other information ...]
    "unlocked_until" : 1392580909,
    "errors" : ""
}
```

3.2.3 钱包备份、纯文本导出及恢复

命令：`backupwallet`、`importwallet`、`dumpwallet`

下一步，我们将练习创建钱包的备份文件，然后从备份文件重新加载钱包。使用 `backupwallet` 命令备份，提供文件名作为命令参数。这里我们将钱包备份为文件 `wallet.backup`：

```
$ bitcoin-cli backupwallet wallet.backup
$
```

现在，为了重新加载备份文件，我们使用 `importwallet` 命令。如果你的钱包处于锁定状态，你将需要先将钱包解锁（参考下一节的 `wallet passphrase`）以导入备份文件：

```
$ bitcoin-cli importwallet wallet.backup
$
```

`dumpwallet` 命令用来将钱包转储为人类可读的文本文件：

```
$ bitcoin-cli dumpwallet wallet.txt
$ more wallet.txt
# Wallet dump created by Bitcoin v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
# * Created on 2014-02- 8dT20:34:55Z
# * Best block at time of backup was 286234
(000000000000000f74f0bc9d3c186267bc45c7b91c49a0386538ac24c0d3a44),
# mined on 2014-02- 8dT20:24:01Z

KzTg2wn6Z8s7ai5NA9MVX4vstRsqP26QKJClLg4JvFrp6mMaGB9 2013-07- 4dT04:30:27Z
change=1 # addr=16pJ6XkwSQv5ma5FSXMRPaXEYrENCEg47F
Kz3dVz7R6mUpXzdZy4gJEVzxJwA15f198eVu14CuivXotzLBDKY 2013-07- 4dT04:30:27Z
change=1 # addr=17oJds8kaN8LP8kuAkWtco6ZM7BGXFC3gk
[... many more keys ...]

$
```

3.2.4 钱包地址及接收交易

命令：`getnewaddress`、`getreceivedbyaddress`、`listtransactions`、`getaddressesbyaccount`、`getbalance`

比特币参考客户端维护了一个地址池，地址池的大小可以用 `getinfo` 命令 `keypoolsize` 参数获取。这些地址是自动生成的，可以被用作公开接收地址或零钱地址。使用 `getnewaddress` 命令可以获得其中的一个地址：

```
$ bitcoin-cli getnewaddress
1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
```

现在我们可以使用这个地址从一个外部钱包（假设你在其他交易所、在线钱包或其他bitcoind钱包有一些比特币）向我们的bitcoind钱包发送一小笔比特币。在本例中，我们将向下面的地址中发送50mBTC（0.050比特币）。

我们可以询问bitcoind客户端此地址已经接收到的比特币数额，以及指定该数额要被加到余额中所需要的确认数。在本例中，我们指定只需要0个确认。在从另一个钱包发送比特币数秒之后，我们将在这个钱包看到反应。我们用 `getreceivedbyaddress` 命令、这个地址及设置确认数为0：

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL 0
0.05000000
```

如果我们忽略这个命令后面的0，我们将只能在至少 `minconf` 个确认之后才能看到数额，`minconf` 是想要某笔交易出现在余额中所设置的最少确认数。`minconf` 设置在bitcoind配置文件指定。由于这笔发送比特币的交易仅在数秒之前完成，它还没有被确认，因而我们将看到余额是0：

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
```

```
0.00000000
```

整个钱包接收到的交易可以通过使用 `listtransactions` 命令展示出来：

```
$ bitcoin-cli listtransactions
[
{
    "account" : "",
    "address": "1hvzSofGwT8cjb8JU7nBcCSfEVQX5u9CL",
    "category" : "receive",
    "amount" : 0.05000000,
    "confirmations" : 0,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309ac
bae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908
}
]
```

我们可以使用 `getaddressesbyaccount` 命令列出整个钱包的所有地址：

```
$ bitcoin-cli getaddressesbyaccount ""
[
    "1LQoTPYy1TyERbNV4ZzhEmgyfAipC6eqL",
    "17vrg8uwMQUibkvS2ECRX4zpcVJ78iFaZS",
    "1FvRHWhHBBZA8cGRRsGiAeqEzUmjJkJQWR",
    "1NVJK3jsL41BF1KyxrUyJW5XHjunjf2jz",
    "14MZqqzCxjc99M5ipsQSRfieT7qPZcM7Df",
    "1BhrGvtKFjTAhGdPGbrEwP3xvFjkJBuFCa",
    "15nem8CX91XtQE8B1Hdv97jE8X44H3DQMT",
    "1Q3q6taTsUiV3mMemEuQQJ9sGLEGaSjo81",
    "1HoSiTg8sb16oE65rmazQEWcGEv8obv9ns",
    "13fE8BGhBvnOy68yZKuwJ2hheYKovSDjqM",
    "1hvzSofGwT8cjb8JU7nBcCSfEVQX5u9CL",
    "1KHUmVFCJteJ21LmRXHSpPoe23rXKifAb2",
    "1LqJZz1D9yHxG4cLkdujnqG5jNNGmPeAMD"
]
```

最后，`getbalance` 命令将显示所有经过至少 `minconf` 个确认的交易加和后的余额：

```
$ bitcoin-cli getbalance
0.05000000
```



如果交易还未被确认，`getbalance`返回的余额将为0。配置项“`minconf`”决定了交易在余额中体现的最少确认数。

3.2.5 探索及解码交易

命令：`gettransaction`、`getrawtransaction`、`decoderawtransaction`

我们将使用 `gettransaction` 命令探索前面列出的入账交易。我们使用 `gettransaction` 命令通过交易哈希值获取一笔交易，交易哈希值出现在前面的txid条目：

```
$ bitcoin-cli gettransaction
```

```

9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3
{
    "amount" : 0.05000000,
    "confirmations" : 0,
    "txid":"9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908,
    "details" : [
        {
            "account" : "",
            "address":"1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.05000000
        }
    ]
}

```



交易ID在交易确认之前并不权威。区块链中找不到交易哈希值并不意味着此笔交易没有进行。这被称作“交易延展性”，因为交易哈希值在区块确认之前是可以更改的。在确认之后txid是不变且权威的。

用 `gettransaction` 命令显示的交易格式为简化格式。若要得到整个交易代码并且将之解码，我们将使用两个命令：`getrawtransaction` 和 `decoderawtransaction`。第一，`getrawtransaction` 把交易哈希值（txid）当做一个参数，并且把整个交易以一个“原始”的十六进制字符串的形式返回，而这也正是交易在比特币网络上存在的形式：

```
$ bitcoin-cli getrawtransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae30100000001d717279515f88e2f56ce4e
```

要解码这个十六进制字符串，我们使用 `decoderawtransaction` 命令。复制粘贴这个十六进制字符串作为 `decoderawtransaction` 的第一个参数以将整个内容解读为JSON数据格式（由于格式原因，在下面例子中十六进制字符串被缩短）：

```

$ bitcoin-cli decoderawtransaction 0100000001d717...388ac00000000
{
    "txid":"9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid": "d3c7e022ea80c4808e64dd0a1dba009f3eaee2318a4ece562f8ef815952717d7",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "3045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e61a2df0dd0758e227383b302203301768ef878007e9ef7c304f
884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e11d41cc79e05dbfe5cea",
                "hex" : "483045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e61a2df0dd0758e227383b302203301768ef878007e9ef7c304
            },
            "sequence" : 4294967295
        },
        "vout" : [
            {
                "value" : 0.05000000,
                "n" : 0,
                "scriptPubKey" : {
                    "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
                    "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
                    "reqSigs" : 1,
                    "type" : "pubkeyhash",
                    "addresses" : [
                        "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"
                    ]
                }
            }
        ]
}

```

```
        },
        {
            "value" : 1.03362847,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 107b7086b31518935c8d28703d66d09b36231343 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a914107b7086b31518935c8d28703d66d09b3623134388ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "12W9goQ3P7waw5JH8fRVs1e2rVAKoGnvoy"
                ]
            }
        }
    ]
}
```



交易解码展示这笔交易的所有成分，包括交易的输入及输出。在这个例子中，我们可以看到这笔给我们新地址存入50mBTC的交易使用了一个输入并且产生两个输出。这笔交易的输入是前一笔确认交易的输出（展示位以d3c7开头的vin txid）。两个输出则是50mBTC存入额度及返回给发送者的找零。

我们可以使用相同命令（例如 `gettransaction`）通过检查由本次交易的txid索引的前一笔交易进一步探索区块链。通过从一笔交易跳到另外一笔交易，我们可以追溯一连串的交易，因为币值一定是从一个拥有者的地址传送到另一个拥有者的地址。

一旦我们接收到的交易以记录在区块中的方式被确认，`gettransaction` 命令将返回附加信息，显示包含交易的区块的哈希值（标识符）。

```
$ bitcoin-cli gettransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3
{
    "amount" : 0.05000000,
    "confirmations" : 1,
    "blockhash" : "00000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c2a717b",
    "blockindex" : 18,
    "blocktime" : 1392660808,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908,
    "details" : [
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjB8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.05000000
        }
    ]
}
```

这里，我们在区块哈希值（这笔交易所在区块的哈希值）条目看到新信息，以及值为18的区块索引（表明我们的交易为此区块的第18笔交易）。

交易数据库索引及txindex选项

比特币核心默认建立包含仅与用户钱包相关交易的数据库。若你想使用类似 `gettransaction` 的命令访问所有交易，你需要配置比特币核心去建立一个完整的交易索引，这个可以通过txindex选项实现。在比特币核心配置文件中将 `txindex` 赋值为1（通常在安装目录的`.bitcoin/bitcoin.conf`中可以找到）。一旦你改变了此参数，你需要重启bitcoind，并等待其重建索引。

3.2.6 探索区块

命令：`getblock`、`getblockhash`

既然我们知道我们的交易在哪个区块中，我们可以使用 `getblock` 命令，并把区块哈希值作为参数来查询对应的区块：

这个区块包含367笔交易，并且如你所见，列出的第18笔交易（9ca8f9...）就是存入50mBTC到我们地址的txid。我们可以通过height条目来判断：这就是整个区块链中第286,384个区块。

我们同样可以使用 `getblockhash` 命令通过区块高度来检索一个区块，这样需要将区块高度作为参数，并返回那个区块的区块哈希值。

```
$ bitcoin-cli getblockhash 00000000000019d6689c085ae165831e934ff763ae46a2a6c17←  
    2b3f1b60a8ce26f  
$ bitcoin-cli getblockhash
```

这里，我们获得了“创世区块”的区块哈希值，这是被中本聪所挖的第一个区块，高度为0。所获得的区块信息如下：

```
$ bitcoin-cli getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1cb60a8ce26f
{
    "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
    "confirmations" : 286388,
    "size" : 285,
    "height" : 0,
    "version" : 1,
    "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
    "tx" : [
        "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
    ],
    "time" : 1231006505,
    "nonce" : 2083236893,
```

`getblock`、`getblockhash` 和 `gettransaction` 命令可以按照一定编程准则，去探索区块链数据库。

3.2.7 基于UTXO（未花费的交易输出）的建立、签名与提交

命令: listunspent、gettxout、createrawtransaction、decoderawtransaction、signrawtransaction、sendrawtransaction

比特币的交易是基于花费“输出”上的，即上一笔交易的支出，整个交易在地址之间转移所有权。我们的钱包现在收到了一笔向我们钱包地址发来的钱（输出）。一旦它被确定之后，那笔钱就属于我们了。

首先，我们可以使用 `listunspent` 命令去查看我们钱包中所有剩余的从之前交易中已确认的支出：

```
$ bitcoin-cli listunspent
[
  {
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "vout" : 0,
    "address" : "1hvzSofGwT8cjB8JU7nBsCSfEVQX5u9CL",
    "account" : "",
    "scriptPubKey" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
    "amount" : 0.05000000,
    "confirmations" : 7
  }
]
```

我们发现交易 9ca8f9 建立了一个被指派到 1hvzSo 地址的输出（“vout”一项为0）对于50mBTC数量的比特币在这个时间点已经收到了7次确认。通过参考交易之前的txid和vout指数，交易系统将先前的输出变为本次的输入。我们现在可以创立一个花费第0个vout的易 9ca8f9 的账单。利用他的输入分配成新的输出，即给新地址赋值。

首先，让我们仔细观察输出的结果。我们可以使用 `gettxout` 命令来得到未花费的输出的详细信息。交易输出通常可以参考 txid 和 vout 两个指标。以下就是我们通过 `gettxout` 命令得到的结果：

```
$ bitcoin-cli gettxout 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3 0
{
    "bestblock" : "00000000000000001405ce69bd4ceebcdfdb537749cebe89d371eb37e13899fd9",
    "confirmations" : 7,
    "value" : 0.05000000,
    "scriptPubKey" : {
        "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2
OP_EQUALVERIFY OP_CHECKSIG",
        "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
        "reqSigs" : 1,
        "type" : "pubkeyhash",
        "addresses" : [
            "1hvzSofGwT8cjB8JU7nBsCSfEVQX5u9CL"
        ]
    },
    "version" : 1,
    "coinbase" : false
}
```

在这里我们看到由50mBTC分配到我们的账户地址 `1hvz...` 中。如果我们想用掉剩余的比特币，我们要重新建立一笔新的交易。首先，我们为这笔交易建立一个新的地址，告诉它将去往哪里：

```
$ bitcoin-cli getnewaddress 1LnfTndv3qzXGN19Jwscj1T8LR3MVe3JDb
```

我们将25mBTC送往我们钱包中新的地址 `1Lnftn...`。在这笔新的交易中，我们将要花费50mBTC并且放入25mBTC到这个新地址中。因为我们必须花费所有之前交易的输出，同时我们必然产生一些找零。我们将产生的找零放回`1hvz...`的地址之中，即将找零放回到原先产生比特币的地址之中。最后，我们必须为这次交易支出一些费用——我们将0.5mBTC作为交易费，最终再存入24.5mBTC的找零。新的输出（ $25\text{mBTC} + 24.5\text{mBTC} = 49.5\text{mBTC}$ ）和输入（50 mBTC）之间的差额就是奖励给矿工的交易费。

我们用 `createrawtransaction` 命令去建立一笔交易。我们将交易的收入（50已确认未支出的mBTC）和两笔交易的输出（送往新地址的比特币和从原先账户找回的零钱）作为 `createrawtransaction` 的参数。

```
$ bitcoin-cli createrawtransaction '[{"txid": "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3", "vout": 0}]' 0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c0000000000ffffffffff02a02526000000000001976a914d90d36e98f
```

`createrawtransaction` 命令产生了一个原始十六进制字符串，其中编码了这笔交易的诸多细节。我们首先要通过 `decoderawtransaction` 命令来解码这个字符串，以确认所有的细节准确无误：

```
$ bitcoin-cli decoderawtransaction 0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c0000000000ffffffffff { "txid": "0793299cb26246a8d24e468ec285a9520a1c30fc5b6125a102e3fc05d4f3cba", "version": 1, "locktime": 0, "vin": [ { "txid": "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3", "vout": 0, "scriptSig": { "asm": "", "hex": "" }, "sequence": 4294967295 } ], "vout": [ { "value": 0.02500000, "n": 0, "scriptPubKey": { "asm": "OP_DUP OP_HASH160 d90d36e98f62968d2bc9bbd68107564a156a9bcf OP_EQUALVERIFY OP_CHECKSIG", "hex": "76a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac", "reqSigs": 1, "type": "pubkeyhash", "addresses": [ "1Lnftndy3qzXGN19Jwscj1T8LR3MVe3JD" ] } }, { "value": 0.02450000, "n": 1, "scriptPubKey": { "asm": "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP_EQUALVERIFY OP_CHECKSIG", "hex": "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac", "reqSigs": 1, "type": "pubkeyhash", "addresses": [ "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL" ] } } ] }
```

结果无误！我们的交易“消耗了”从我们已确认的交易中未花费的输出，然后将它变成两笔输出，一个是走向了新地址的25mBTC，另一个是从原来地址返回的24.5mBTC零钱。这之间0.5mBTC的差额作为交易费，以奖励挖出包含我们这笔交易

区块的矿工。

你有可能注意到，交易中包含一个空的条目 `scriptSig`，因为我们并没有给它签名。如果没有签名，那么交易是没有意义的；同时我们也不能证明我们拥有未花费的输出的来源地址的所有权。通过签名，我们移除了输出上的障碍同时证明了我们的输出可靠。我们使用 `signrawtransaction` 命令去签名交易。它需要原始十六进制的字符串作为参数：



一个加密的钱包在签名之前必须解密，因为签名需要利用钱包中的秘钥。

```
$ bitcoin-cli wallet passphrase foo 360
$ bitcoin-cli signrawtransaction 01000000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bac
f7db0f6685862aecf53ebd69f9a89c0000000000fffffff02a0252600000000001976a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac506225000000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000
{
    "hex" : "01000000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bac
    "complete" : true
}
```

输入 `signrawtransaction` 命令后，得到另一串十六进制的原始加密交易。我们要对它进行解密，然后去查看发生的变化，请输入 `decoderawtransaction` 命令：

```
$ bitcoin-cli decoderawtransaction 01000000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000006a4730440220
{
    "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "304402203e8a16522da80cef66bacfb0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f
                "hex" : "47304402203e8a16522da80cef66bacfb0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca5
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.02500000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 d90d36e98f62968d2bc9bbd68107564a156a9bcf OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1LnftTndy3qzXGN19Jwscj1T8LR3MVe3JD"
                ]
            }
        },
        {
            "value" : 0.02450000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [

```

```
        "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"
    ]
}
]
}

1
```

现在，交易中的收入包含了 `scriptSig`，一串证明钱包地址 `1hvz...` 所有权的数字签名，同时移除了支出上的障碍，然后我们可以对钱包中的钱进行消费。签名可以让这笔交易被比特币交易网络中的任何节点验证，使它们变得可靠。

现在，该是提交新交易到比特币网络的时候了。我们使用由原始十六进制 `signrawtransaction` 命令生成的 `sendrawtransaction` 命令。以下就是和刚才解码时类似的字符串：

```
$ bitcoin-cli sendrawtransaction 0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e
```

当使用 `sendrawtransaction` 命令发布交易到比特币网络时，它会返回交易的哈希值。我们现在可以通过 `gettransaction` 命令查询交易ID：

```
$ bitcoin-cli gettransaction ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346
{
  "amount" : 0.0000000,
  "fee" : -0.00050000,
  "confirmations" : 0,
  "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
  "time" : 1392666702,
  "timereceived" : 1392666702,
  "details" : [
    {
      "account" : "",
      "address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
      "amount" : -0.02500000,
      "fee" : -0.00050000
    },
    {
      "account" : "",
      "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
      "amount" : -0.02450000,
      "fee" : -0.00050000
    },
    {
      "account" : "",
      "address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
      "amount" : 0.02500000
    },
    {
      "account" : "",
      "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
      "amount" : 0.02450000
    }
  ]
}
```

和以前一样，我们同样可以通过使用 `getrawtransaction` 和 `decoderawtransaction` 命令来检查交易中的细节。这些命令会得到一个在发送到比特币网络之前进行编码和解码并且十分精准的原始十六进制字符串。

3.3 其他替代客户端、资料库、工具包

除了参考客户端（`bitcoind`），还可以使用其他的客户端和资料库去连接比特币网络和数据结构。这些工具都由一系列的编程语言执行，用他们各自的语言为比特币程序提供原生的交互。

其他的执行方式包括：

▷ [libbitcoin和sx tools](#)

一款C++，通过命令行完成的全节点多线程客户端与程序库

▷ [bitcoinj](#)

一款全节点java客户端和程序库

▷ [btcd](#)

一款全节点GO 语言的比特币客户端

▷ [Bits of Proof \(BOP\)](#)

一款Java企业级平台的比特币工具

▷ [picocoin](#)

一款轻量级比特币执行客户端

▷ [pybitcointools](#)

一款Python语言的比特币程序库

▷ [pycoin](#)

另一款Python语言的比特币程序库

在其他的编程语言中，还有许多形式的比特币（程序）库。开发者也尽其所能，一直在尽力创造新的比特币工具。

3.3.1 Libbitcoin和sx Tools

Libbitcoin程序是一款基于C++层面，可扩展、多线程、模块化的执行工具。它可以支持全节点客户端和一款叫做sx的命令行工具，并可以提供我们本章所讨论的比特币命令相同的功能。Sx工具同时提供了管理和操作工具，是bitcoind所不能提供的，包括type-2型确定性密钥和密码助记工具。

安装sx

若要安装sx工具以及相关libbitcoin库，请在Linux操作系统中下载并安装在线安装包：

```
$ wget http://sx.dyne.org/install-sx.sh  
$ sudo bash ./install-sx.sh
```

现在你应当已经安装好了sx工具。输入没有参数的sx命令来显示帮助文档，帮助文档列出了所有可用的命令（详见附录4）。



sx工具提供了许多实用的编码与解码地址的命令，可以从不同的编码方式转化，也可以转化成不同的方式。通过它们，可以探索更多的编码方式，比如Base58, Base58Check, hex，等等。

3.3.2 pycoin

[pycoin](#)最初由Richard Kiss创立并维护，是一款基于Python库，并可以支持比特币密钥的操作和交易的客户端，甚至可以支持编译语言从而处理非标准交易。

Pycoin库同时支持Python2（2.7x）与Python3，以及一些便于使用的命令行工具，比如ku和tx。如果在Python3的虚拟环境中安装 pycoin0.42，请输入以下命令：

```
$ python3 -m venv /tmp/pycoin
$ . /tmp/pycoin/bin/activate
$ pip install pycoin==0.42
Downloading/unpacking pycoin==0.42
  Downloading pycoin-0.42.tar.gz (66kB) 66kB downloaded
    Running setup.py (path:/tmp/pycoin/build/pycoin/setup.py) egg_info for package pycoin

Installing collected packages: pycoin
  Running setup.py install for pycoin

    Installing tx script to /tmp/pycoin/bin
    Installing cache_tx script to /tmp/pycoin/bin
    Installing bu script to /tmp/pycoin/bin
    Installing fetch_unspent script to /tmp/pycoin/bin
    Installing block script to /tmp/pycoin/bin
    Installing spend script to /tmp/pycoin/bin
    Installing ku script to /tmp/pycoin/bin
    Installing genwallet script to /tmp/pycoin/bin
Successfully installed pycoin
Cleaning up...
$
```

这里有一个简单的Python脚本，通过pycoin库来交易比特币：

```
#!/usr/bin/env python

from pycoin.key import Key
from pycoin.key.validate import is_address_valid, is_wif_valid from pycoin.services import spendables_for_address
from pycoin.tx.tx_utils import create_signed_tx

def get_address(which):
    while 1:
        print("enter the %s address=> " % which, end='')
        address = input()
        is_valid = is_address_valid(address)
        if is_valid:
            return address
        print("invalid address, please try again")

src_address = get_address("source")
spendables = spendables_for_address(src_address) print(spendables)

while 1:
    print("enter the WIF for %s=> " % src_address, end='')
    wif = input()
    is_valid = is_wif_valid(wif)
    if is_valid:
        break
    print("invalid wif, please try again")

key = Key.from_text(wif)
if src_address not in (key.address(use_uncompressed=False), key.address(use_uncompressed=True)):
    print("!! WIF doesn't correspond to %s" % src_address)
print("The secret exponent is %d" % key.secret_exponent())

dst_address = get_address("destination")

tx = create_signed_tx(spendables, payables=[dst_address], wifs=[wif])

print("here is the signed output transaction")
print(tx.as_hex())
```

更多的ku与tx命令行样例，请参考附录2。

3.3.3 btcd

btcd是一款基于Go语言的全节点比特币工具。目前，它通过使用精准的规则（包括bugs），下载、验证和服务区块链。它同时依靠新发掘出来的区块来维持交易池，同时依赖没有形成区块的单独交易。在缜密的规则以及检查下，确保了每笔独立交易的安全，并且可以过滤基于矿工需求的交易。

btcd与bitcoind的一个主要区别是btcd不包含比特币钱包的功能，其实这是一个精心的设计。这意味着你不能直接通过btcd进行比特币交易。然而这项功能可以由正在研发的btcwallet与btcgui两个项目提供。另一个显著的区别是btcd同时支持HTTP POST（比如bitcoind）与推荐使用的Websockets两种通信协议的请求。并且btcd的RPC连接默认设置为TLS-开启。

安装btcd

若要安装Windows版btcd，请[从GitHub下载](#)并运行msi；如果你已经安装了Go语言，请在Linux中输入以下命令行：

```
$ go get github.com/conformal/btcd/...
```

若要更新btcd到最新版本，请输入：

```
$ go get -u -v github.com/conformal/btcd/...
```

调试btcd

btcd拥有许多配置选项，可以通过以下命令来查看：

```
$ btcd --help
```

btcd预装了许多好用的功能包，比如btcctl。它是一种可以通过RPC来控制和查询的命令行工具。Btcd并没有默认开启了RPC服务器，你必须通过以下命令行来配置RPC用户名及密码：

▷ btcd.conf:

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

▷ btcctl.conf:

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

若果你想要重写配置，请输入以下命令：

```
$ btcd -u myuser -P SomeDecentp4ssw0rd
$ btcctl -u myuser -P SomeDecentp4ssw0rd
```

可以通过以下命令来查询一系列的选项：

```
$ btcctl --help
```


第4章 密钥、地址、钱包

4.1 简介

比特币的所有权是通过数字密钥、比特币地址和数字签名来确立的。数字密钥实际上并不是存储在网络中，而是由用户生成并存储在一个文件或简单的数据库中，称为钱包。存储在用户钱包中的数字密钥完全独立于比特币协议，可由用户的钱包软件生成并管理，而无需区块链或网络连接。密钥实现了比特币的许多有趣特性，包括去中心化信任和控制、所有权认证和基于密码学证明的安全模型。

每笔比特币交易都需要一个有效的签名才会被存储在区块链。只有有效的数字密钥才能产生有效的数字签名，因此拥有比特币的密钥副本就拥有了该账户的比特币控制权。密钥是成对出现的，由一个私钥和一个公钥所组成。公钥就像银行的帐号，而私钥就像控制账户的PIN码或支票的签名。比特币的用户很少会直接看到数字密钥。一般情况下，它们被存储在钱包文件内，由比特币钱包软件进行管理。

在比特币交易的支付环节，收件人的公钥是通过其数字指纹表示的，称为比特币地址，就像支票上的支付对象的名字（即“收款方”）。一般情况下，比特币地址由一个公钥生成并对应于这个公钥。然而，并非所有比特币地址都是公钥；他们也可以代表其他支付对象，譬如脚本，我们将在本章后面提及。这样一来，比特币地址把收款方抽象起来了，使得交易的目的地更灵活，就像支票一样：这个支付工具可支付到个人账户、公司账户，进行账单支付或现金支付。比特币地址是用户经常看到的密钥的唯一代表，他们只需要把比特币地址告诉其他人即可。

在本章中，我们将介绍钱包，也就是密钥所在之处。我们将了解密钥如何被产生、存储和管理。我们将回顾私钥和公钥、地址和脚本地址的各种编码格式。最后，我们将讲解密钥的特殊用途：生成签名、证明所有权以及创造比特币靓号地址和纸钱包。

4.1.1 公钥加密和加密货币

公钥加密发明于20世纪70年代。它是计算机和信息安全的数学基础。

自从公钥加密被发明之后，一些合适的数学函数被提出，譬如：素数幂和椭圆曲线乘法。这些数学函数都是不可逆的，就是说很容易向一个方向计算，但不可以向相反方向倒推。基于这些数学函数的密码学，使得生成数字密钥和不可伪造的数字签名成为可能。比特币正是使用椭圆曲线乘法作为其公钥加密的基础算法。

在比特币系统中，我们用公钥加密创建一个密钥对，用于控制比特币的获取。密钥对包括一个私钥，和由其衍生出的唯一的公钥。公钥用于接收比特币，而私钥用于比特币支付时的交易签名。

公钥和私钥之间的数学关系，使得私钥可用于生成特定消息的签名。此签名可以在不泄露私钥的同时对公钥进行验证。

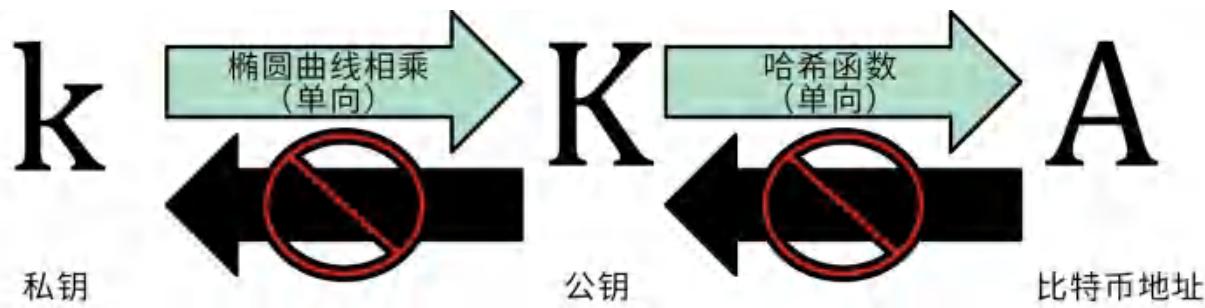
支付比特币时，比特币的当前所有者需要在交易中提交其公钥和签名（每次交易的签名都不同，但均从同一个私钥生成）。比特币网络中的所有人都可以通过所提交的公钥和签名进行验证，并确认该交易是否有效，即确认支付者在该时刻对所交易的比特币拥有所有权。



大多数比特币钱包工具为了方便会将私钥和公钥以密钥对的形式存储在一起。然而，公钥可以由私钥计算得到，所以只存储私钥也是可以的。

4.1.2 私钥和公钥

一个比特币钱包中包含一系列的密钥对，每个密钥对包括一个私钥和一个公钥。私钥（k）是一个数字，通常是随机选出的。有了私钥，我们就可以使用椭圆曲线乘法这个单向加密函数产生一个公钥（K）。有了公钥（K），我们就可以使用一个单向加密哈希函数生成比特币地址（A）。在本节中，我们将从生成私钥开始，讲述如何使用椭圆曲线运算将私钥生成公钥，并最终由公钥生成比特币地址。私钥、公钥和比特币地址之间的关系如下图所示。



4.1.3 私钥

私钥就是一个随机选出的数字而已。一个比特币地址中的所有资金的控制取决于相应私钥的所有权和控制权。在比特币交易中，私钥用于生成支付比特币所必需的签名以证明资金的所有权。私钥必须始终保持机密，因为一旦被泄露给第三方，相当于该私钥保护之下的比特币也拱手相让了。私钥还必须进行备份，以防意外丢失，因为私钥一旦丢失就难以复原，其所保护的比特币也将永远丢失。



比特币私钥只是一个数字。你可以用硬币、铅笔和纸来随机生成你的私钥：掷硬币256次，用纸和笔记录正反面并转换为0和1，随机得到的256位二进制数字可作为比特币钱包的私钥。该私钥可进一步生成公钥。

从一个随机数生成私钥

生成密钥的第一步也是最重要的一步，是要找到足够安全的熵源，即随机性来源。生成一个比特币私钥在本质上与“在1到 2^{256} 之间选一个数字”无异。只要选取的结果是不可预测或不可重复的，那么选取数字的具体方法并不重要。比特币软件使用操作系统底层的随机数生成器来产生256位的熵（随机性）。通常情况下，操作系统随机数生成器由人工的随机源进行初始化，也可能需要通过几秒钟内不停晃动鼠标等方式进行初始化。对于真正的偏执狂，可以使用掷骰子的方法，并用铅笔和纸记录。

更准确地说，私钥可以是1和n-1之间的任何数字，其中n是一个常数（ $n=1.158 \times 10^{77}$ ，略小于 2^{256} ），并由比特币所使用的椭圆曲线的阶所定义（见[4.1.5 椭圆曲线密码学解释](#)）。要生成这样的一个私钥，我们随机选择一个256位的数字，并检查它是否小于n-1。从编程的角度来看，一般是通过在一个密码学安全的随机源中取出一长串随机字节，对其使用SHA256哈希算法进行运算，这样就可以方便地产生一个256位的数字。如果运算结果小于n-1，我们就有了一个合适的私钥。否则，我们就用另一个随机数再重复一次。



本书强烈建议读者不要使用自己写的代码或使用编程语言内建的简易随机数生成器来获得一个随机数。我们建议读者使用密码学安全的伪随机数生成器（CSPRNG），并且需要有一个来自具有足够熵值的源的种子。使用随机数发生器的程序库时，需仔细研读其文档，以确保它是加密安全的。对CSPRNG的正确实现是密钥安全性的关键所在。

以下是一个随机生成的私钥（k），以十六进制格式表示（256位的二进制数，以64位十六进制数显示，每个十六进制数占4位）：

```
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
```



比特币私钥空间的大小是 2^{256} ，这是一个非常大的数字。用十进制表示的话，大约是 10^{77} ，而可见宇宙被估计只含有 10^{80} 个原子。

要使用比特币核心客户端生成一个新的密钥（参见[第3章](#)），可使用 `getnewaddress` 命令。出于安全考虑，命令运行后只显示生成的公钥，而不显示私钥。如果要bitcoind显示私钥，可以使用 `dumpprivatekey` 命令。`dumpprivatekey` 命令会把私钥以Base58校验和编码格式显示，这种私钥格式被称为钱包导入格式（WIF，Wallet Import Format），在“[私钥的格式](#)”一节有详细讲解。下面给出了使用这两个命令生成和显示私钥的例子：

```
$ bitcoind getnewaddress  
1J7mdg5rbQyUHENYdx39WWK7fsLpEoXZy  
$ bitcoind dumpprivatekey 1J7mdg5rbQyUHENYdx39WWK7fsLpEoXZy  
KxFc1jmwwCoACiCAwZ3eXa96mBM6tb3TYzGmf6YwgdGwZgawvrtJ
```

`dumpprivatekey` 命令只是读取钱包里由 `getnewaddress` 命令生成的私钥，然后显示出来。bitcoind的并不能从公钥得知私钥。除非密钥对都存储在钱包里，`dumpprivatekey` 命令才有效。



`dumpprivatekey` 命令无法从公钥得到对应的私钥，因为这是不可能的。这个命令只是提取钱包中已有的私钥，也就是提取由 `getnewaddress` 命令生成的私钥。

你也可以使用命令行sx工具（参见“[3.3.1 Libbitcoin和sx Tools](#)”）用`newkey`命令来生成并显示私钥：

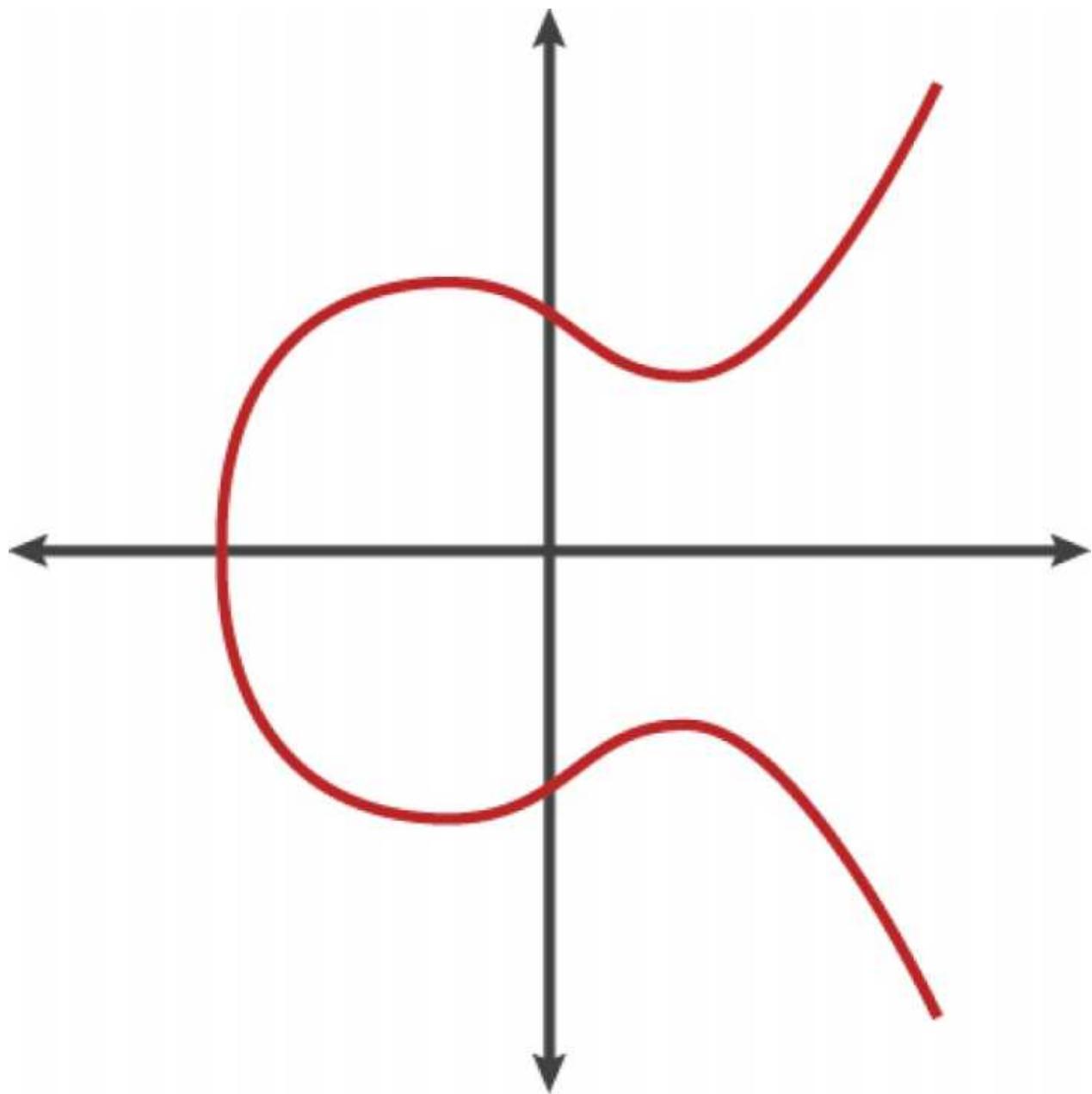
```
$ sx newkey  
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfSYB1Jcn
```

4.1.4 公钥

通过椭圆曲线乘法可以从私钥计算得到公钥，这是不可逆转的过程： $K = k * G$ 。其中 k 是私钥， G 是被称为生成点的常数点，而 K 是所得公钥。其反向运算，被称为“寻找离散对数”——已知公钥 K 来求出私钥 k ——是非常困难的，就像去试验所有可能的 k 值，即暴力搜索。在演示如何从私钥生成公钥之前，我们先稍微详细学习下椭圆曲线加密学。

4.1.5 椭圆曲线密码学解释

椭圆曲线加密法是一种基于离散对数问题的非对称（或公钥）加密法，可以用对椭圆曲线上的点进行加法或乘法运算来表达。



上图是一个椭圆曲线的示例，类似于比特币所用的曲线。

比特币使用了 secp256k1 标准所定义的一条特殊的椭圆曲线和一系列数学常数。该标准由美国国家标准与技术研究院（NIST）设立。 secp256k1 曲线由下述函数定义，该函数可产生一条椭圆曲线：

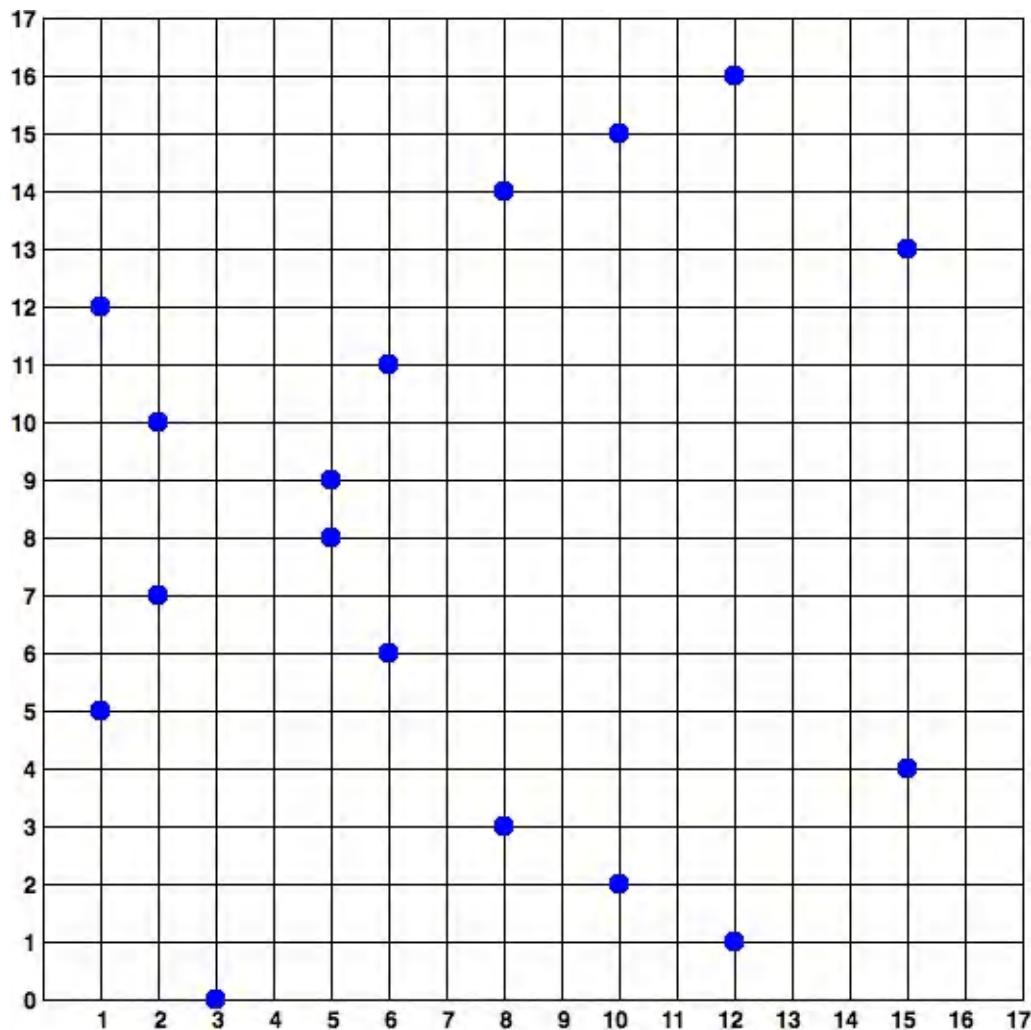
$$y^2 = (x^3 + 7) \pmod{p}$$

或

$$y^2 = (x^3 + 7)$$

上述 \pmod{p} 表明该曲线是在素数阶 p 的有限域内，也写作 \mathbb{F}_p ，其中 $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ ，这是一个非常大的素数。

因为这条曲线被定义在一个素数阶的有限域内，而不是定义在实数范围，它的函数图像看起来像分散在两个维度上的散点图，因此很难画图表示。不过，其中的数学原理与实数范围的椭圆曲线相似。作为一个例子，下图显示了在一个小了很多的素数阶 17 的有限域内的椭圆曲线，其形式为网格上的一系列散点。而 secp256k1 的比特币椭圆曲线可以被想象成一个极大的网格上一系列更为复杂的散点。



图为：椭圆曲线密码学 $F(p)$ 上的椭圆曲线，其中 $p = 17$

下面举一个例子，这是 secp256k1 曲线上的点P，其坐标为(x, y)。可以使用Python对其检验：

```
P = (55066263022277343669578718895168534326250603453777594175500187360389116729240, 326705100207588169780830851305070431844712733
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 = 11579209237316195423570985008687907853269984665640564039457584007908834671663
>>> x = 55066263022277343669578718895168534326250603453777594175500187360389116729240
>>> y = 32670510020758816978083085130507043184471273380659243275938904335757337482424
>>> (x ** 3 + 7 - y**2) % p
0
```

在椭圆曲线的数学原理中，有一个点被称为“无穷远点”，这大致对应于0在加法中的作用。计算机中，它有时表示为 $X = Y = 0$ （虽然这不满足椭圆曲线方程，但可作为特殊情况进行检验）。还有一个+运算符，被称为“加法”，就像小学数学中的实数相加。给定椭圆曲线上的两个点 P_1 和 P_2 ，则椭圆曲线上必定有第三点 $P_3 = P_1 + P_2$ 。

几何图形中，该第三点 P_3 可以在 P_1 和 P_2 之间画一条线来确定。这条直线恰好与椭圆曲线上的一点相交。此点记为 $P_3 = (x, y)$ 。然后，在x轴做映射获得 $P_3 = (x, -y)$ 。

下面是几个可以解释“无穷远点”之存在需要的特殊情况。若 P_1 和 P_2 是同一点， P_1 和 P_2 间的连线则为点 P_1 的切线。曲线上有且只有一个新的点与该切线相交。该切线的斜率可用微分求得。即使限制曲线点为两个整数坐标也可求得斜率！

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），则切线会完全垂直，在这种情况下， $P_3 = “无穷远点”$ 。

若 P_1 就是“无穷远点”，那么其和 $P_1 + P_2 = P_2$ 。类似地，当 P_2 是无穷远点，则 $P_1 + P_2 = P_1$ 。这就是把无穷远点类似于0的作用。

事实证明，在这里+运算符遵守结合律，这意味着 $(A+B)C = A(B+C)$ 。这就是说我们可以直接不加括号书写 $A + B + C$ ，而不至于混淆。

至此，我们已经定义了椭圆加法，为扩展加法下面我们将对乘法进行标准定义。给定椭圆曲线上的点 P ，如果 k 是整数，则 $kP = P + P + P + \dots + P$ （ k 次）。注意， k 被有时被混淆而称为“指数”。

4.1.6 生成公钥

以一个随机生成的私钥 k 为起点，我们将其与曲线上已定义的生成点 G 相乘以获得曲线上的另一点，也就是相应的公钥 K 。生成点是secp256k1标准的一部分，比特币密钥的生成点都是相同的：

```
{K = k * G}
```

其中 k 是私钥， G 是生成点，在该曲线上所得的点 K 是公钥。因为所有比特币用户的生成点是相同的，一个私钥 k 乘以 G 将得到相同的公钥 K 。 k 和 K 之间的关系是固定的，但只能单向运算，即从 k 得到 K 。这就是可以把比特币地址（ K 的衍生）与任何人共享而不会泄露私钥（ k ）的原因。



因为其中的数学运算是单向的，所以私钥可以转换为公钥，但公钥不能转换回私钥。

为实现椭圆曲线乘法，我们以之前产生的私钥 k 和与生成点 G 相乘得到公钥 K ：

```
K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD * G
```

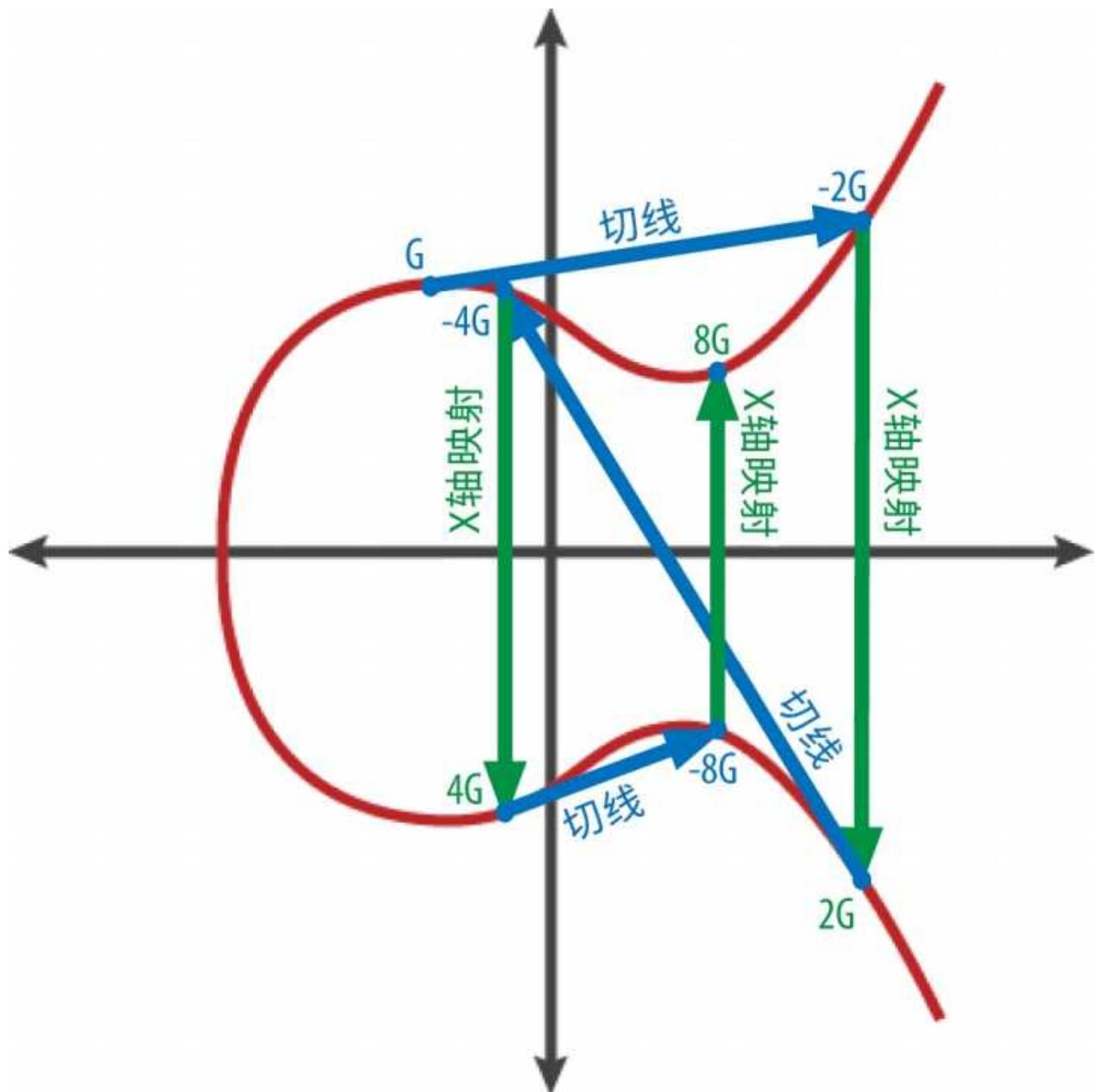
公钥 K 被定义为一个点 $K = (x, y)$ ：

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

其中，

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A  
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

为了展示整数点的乘法，我们将使用较为简单的实数范围的椭圆曲线。请记住，其中的数学原理是相同的。我们的目标是找到生成点 G 的倍数 kG 。也就是将 G 相加 k 次。在椭圆曲线中，点的相加等同于从该点画切线找到与曲线相交的另一点，然后映射到 x 轴。



上图显示了在曲线上得到 G 、 $2G$ 、 $4G$ 的几何操作。



大多数比特币程序使用 OpenSSL 加密库进行椭圆曲线计算。例如，调用 EC_POINT_mul() 函数，可计算得到公钥。

4.2 比特币地址

比特币地址是一个由数字和字母组成的字符串，可以与任何想给你比特币的人分享。由公钥（一个同样由数字和字母组成的字符串）生成的比特币地址以数字“1”开头。下面是一个比特币地址的例子：

1J7mdg5rbQyUHENYdx39WwK7fsLpEoXZy

在交易中，比特币地址通常以收款方出现。如果把比特币交易比作一张支票，比特币地址就是收款人，也就是我们要写入收款人一栏的内容。一张支票的收款人可能是某个银行账户，也可能是某个公司、机构，甚至是现金支票。支票不需要指定一个特定的账户，而是用一个普通的名字作为收款人，这使它成为一种相当灵活的支付工具。与此类似，比特币地址的使用也使比特币交易变得很灵活。比特币地址可以代表一对公钥和私钥的所有者，也可以代表其它东西，比如会在132页的“P2SH (Pay-to-Script-Hash)”一节讲到的付款脚本。现在，让我们来看一个简单的例子，由公钥生成比特币地址。

比特币地址可由公钥经过单向的加密哈希算法得到。哈希算法是一种单向函数，接收任意长度的输入产生指纹摘要。加密哈希函数在比特币中被广泛使用：比特币地址、脚本地址以及在挖矿中的工作量证明算法。由公钥生成比特币地址时使用的算法是Secure Hash Algorithm (SHA)和the RACE Integrity Primitives Evaluation Message Digest (RIPEMD)，特别是SHA256和RIPEMD160。

以公钥 K 为输入，计算其SHA256哈希值，并以此结果计算RIPEMD160 哈希值，得到一个长度为160比特（20字节）的数字：

$$A = \text{RIPEMD160}(\text{SHA256}(K))$$

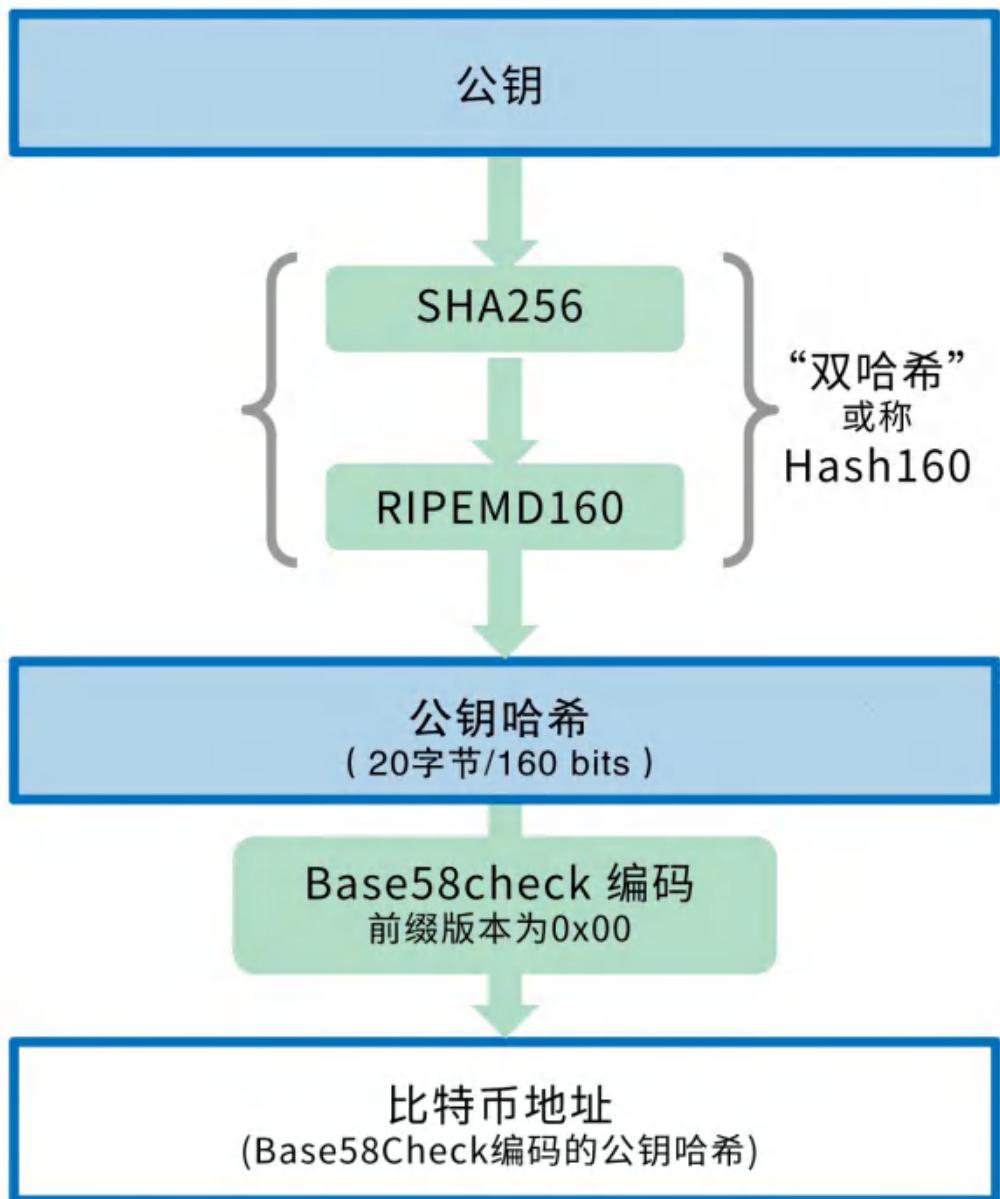
公式中，K是公钥，A是生成的比特币地址。



比特币地址与公钥不同。比特币地址是由公钥经过单向的哈希函数生成的。

通常用户见到的比特币地址是经过“Base58Check”编码的（参见72页“Base58和Base58Check编码”一节），这种编码使用了58个字符（一种Base58数字系统）和校验码，提高了可读性、避免歧义并有效防止了在地址转录和输入中产生的错误。Base58Check编码也被用于比特币的其它地方，例如比特币地址、私钥、加密的密钥和脚本哈希中，用来提高可读性和录入的正确性。下一节中我们会详细解释Base58Check的编码机制，以及它产生的结果。下图描述了如何从公钥生成比特币地址。

从公钥到比特币地址



4.2.1 Base58和Base58Check编码

为了更简洁方便地表示长串的数字，许多计算机系统会使用一种以数字和字母组成的大于十进制的表示法。例如，传统的十进制计数系统使用0-9十个数字，而十六进制系统使用了额外的A-F六个字母。一个同样的数字，它的十六进制表示就会比十进制表示更短。更进一步，Base64使用了26个小写字母、26个大写字母、10个数字以及两个符号（例如“+”和“/”），用于在电子邮件这样的基于文本的媒介中传输二进制数据。Base64通常用于编码邮件中的附件。Base58是一种基于文本的二进制编码格式，用在比特币和其他的加密货币中。这种编码格式不仅实现了数据压缩，保持了易读性，还具有错误诊断功能。Base58是Base64编码格式的子集，同样使用大小写字母和10个数字，但舍弃了一些容易错读和在特定字体中容易混淆的字符。具体地，Base58不含Base64中的0（数字0）、O（大写字母o）、l（小写字母L）、I（大写字母i），以及“+”和“/”两个字符。简而言之，Base58就是由不包括(0, O, l, I)的大小写字母和数字组成。

例4-1 比特币的Base58字母表

```
123456789ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Base58Check是一种常用在比特币中的Base58编码格式，增加了错误校验码来检查数据在转录中出现的错误。校验码长4个字节，添加到需要编码的数据之后。校验码是从需要编码的数据的哈希值中得到的，所以可以用来检测并避免转录和输入中产生的错误。使用Base58check编码格式时，编码软件会计算原始数据的校验码并和结果数据中自带的校验码进行对比。二者不匹配则表明有错误产生，那么这个Base58Check格式的数据就是无效的。例如，一个错误比特币地址就不会被钱包认为是有效的地址，否则这种错误会造成资金的丢失。

为了使用Base58Check编码格式对数据（数字）进行编码，首先我们要对数据添加一个称作“版本字节”的前缀，这个前缀用来明确需要编码的数据的类型。例如，比特币地址的前缀是0（十六进制是0x00），而对私钥编码时前缀是128（十六进制是0x80）。表4-1会列出一些常见版本的前缀。

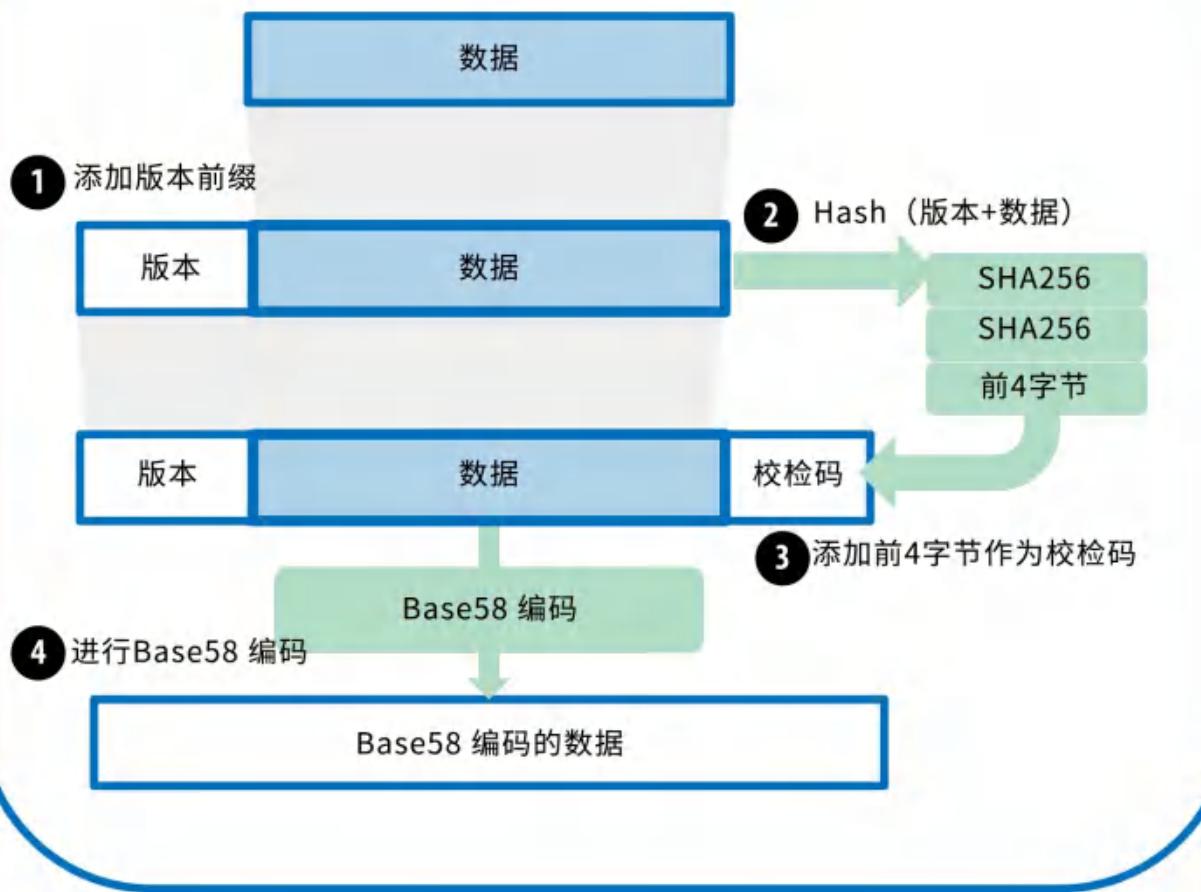
接下来，我们计算“双哈希”校验码，意味着要对之前的结果（前缀和数据）运行两次SHA256哈希算法：

```
checksum = SHA256(SHA256(prefix+data))
```

在产生的长32个字节的哈希值（两次哈希运算）中，我们只取前4个字节。这4个字节就作为校验码。校验码会添加到数据之后。

结果由三部分组成：前缀、数据和校验码。这个结果采用之前描述的Base58字母表编码。下图描述了Base58Check编码的过程。

从公钥到比特币地址



Base58Check编码：一种Base58格式的、有版本的、经过校验的格式，可以明确的对比特币数据编码的编码格式

在比特币中，大多数需要向用户展示的数据都使用Base58Check编码，可以实现数据压缩，易读而且有错误检验。Base58Check编码中的版本前缀是数据的格式易于辨别，编码之后的数据头包含了明确的属性。这些属性使用户可以轻松明确被编码的数据的类型以及如何使用它们。例如我们可以看到他们的不同，Base58Check编码的比特币地址是以1开头的，而Base58Check编码的私钥WIF是以5开头的。表4-1展示了一些版本前缀和他们对应的Base58格式。

表4-1 Base58Check版本前缀和编码后的结果

种类	版本前缀 (hex)	Base58格式
Bitcoin Address	0x00	1
Pay-to-Script-Hash Address	0x05	3
Bitcoin Testnet Address	0x6F	m or n
Private Key WIF	0x80	5, K or L
BIP38 Encrypted Private Key	0x0142	6P
BIP32 Extended Public Key	0x0488B21E	xpub

我们回顾比特币地址产生的完整过程，从私钥、到公钥（椭圆曲线上某个点）、再到两次哈希的地址，最终产生Base58Check格式的比特币地址。例4-2的C++代码完整详细的展示了从私钥到Base58Check编码后的比特币地址的步骤。代码中使用“3.3 其他客户端、资料库、工具包”一节中介绍的libbitcoin library来实现某些辅助功能。

例4-2 从私钥产生一个Base58Check格式编码的比特币地址

```

#include <bitcoin/bitcoin.hpp>

int main() {
    // Private secret key.
    bc::ec_secret secret = bc::decode_hex(
        "038109007313a5807b2ecc082c8c3fb988a973cacf1a7df9ce725c31b14776");
    // Get public key.
    bc::ec_point public_key = bc::secret_to_public_key(secret);
    std::cout << "Public key: " << bc::encode_hex(public_key) << std::endl;

    // Create Bitcoin address.
    // Normally you can use:
    //   bc::payment_address payaddr;
    //   bc::set_public_key(payaddr, public_key);
    //   const std::string address = payaddr.encoded();

    // Compute hash of public key for P2PKH address.
    const bc::short_hash hash = bc::bitcoin_short_hash(public_key);
    bc::data_chunk unencoded_address; // Reserve 25 bytes
    // [ version:1 ]
    // [ hash:20 ]
    // [ checksum:4 ]
    unencoded_address.reserve(25);
    // Version byte, 0 is normal BTC address (P2PKH).      unencoded_address.push_back(0);
    // Hash data
    bc::extend_data(unencoded_address, hash);
    // Checksum is computed by hashing data, and adding 4 bytes from hash. bc::append_checksum(unencoded_address);
    // Finally we must encode the result in Bitcoin's base58 encoding assert(unencoded_address.size() == 25);
    const std::string address = bc::encode_base58(unencoded_address);
    std::cout << "Address: " << address << std::endl;
    return 0;
}

```

正如编译并运行addr代码中展示的，由于代码使用预定义的私钥，所以每次运行都会产生相同的比特币地址。如例4-3所示。

例4-3 编译并运行addr代码

```

# Compile the addr.cpp code
$ g++ -o addr addr.cpp $(pkg-config --cflags --libs libbitcoin)
# Run the addr executable
$ ./addr
Public key: 0202a406624211f2abbd68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa Address: 1PRTTaJesdNovgne6Ehcd1fpEdX7913CK

```

4.2.2 密钥的格式

公钥和私钥的都可以有多种编码格式。一个密钥被不同的格式编码后，虽然结果看起来可能不同，但是密钥所编码数字并没有改变。这些不同的编码格式主要是用来方便人们无误地使用和识别密钥。

私钥的格式

私钥可以以许多不同的格式表示，所有这些都对应于相同的256位的数字。表4-2展示了私钥的三种常见格式。

表4-2 私钥表示法（编码格式）

种类	版本	描述
Hex	None	64 hexadecimal digits
WIF	5	Base58Check encoding: Base58 with version prefix of 128 and 32-bit checksum
WIF-compressed	K or L	As above, with added suffix 0x01 before encoding

表4-3展示了用这三种格式所生成的私钥。

表4-3 示例：同样的私钥，不同的格式

格式	私钥
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF-compressed	KxFc1jmwwCoACiCAwZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawrtJ

这些表示法都是用来表示相同的数字、相同的私钥的不同方法。虽然编码后的字符串看起来不同，但不同的格式彼此之间可以很容易地相互转换。

将Base58Check编码解码为十六进制

sx工具包（参见“[3.3.1 Libbitcoind和sx Tools](#)”）可用来编写一些操作比特币密钥、地址及交易的shell脚本和命令行“管道”。你也可以使用sx工具从命令行对Base58Check格式进行解码。

我们使用的命令是 `base58check-decode`：

```
$ sx base58check-decode 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn  
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd 128
```

所得结果是十六进制的密钥，紧接着是钱包导入格式（Wallet Import Format,WIF）的版本前缀128。

将十六进制转换为Base58Check编码

要转换成Base58Check编码（和之前的命令正好相反），我们需提供十六进制的私钥和钱包导入格式（Wallet Import Format, WIF）的版本号前缀128：

```
$sx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd 128  
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

将十六进制（压缩格式密钥）转换为Base58Check编码

要将压缩格式的私钥编码为Base58Check（参见“[压缩格式私钥](#)”一节），我们需在十六进制私钥的后面添加后缀01，然后使用跟上面一样的方法：

```
$ sx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 128  
KxFc1jmwwCoACiCAwZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawrtJ
```

生成的WIF压缩格式的私钥以字母“K”开头，用以表明被编码的私钥有一个后缀“01”，且该私钥只能被用于生成压缩格式的公钥（参见“[压缩格式公钥](#)”一节）。

公钥的格式

公钥也可以用多种不同格式来表示，最重要的是它们分为非压缩格式或压缩格式公钥这两种形式。

我们从前文可知，公钥是在椭圆曲线上的一个点，由一对坐标（x, y）组成。公钥通常表示为前缀04紧接着两个256比特的数字。其中一个256比特数字是公钥的x坐标，另一个256比特数字是y坐标。前缀04是用来区分非压缩格式公钥，压缩格式公钥是以02或者03开头。

下面是由前文中的私钥所生成的公钥，其坐标x和y如下：

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A  
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

下面是同样的公钥以520比特的数字（130个十六进制数字）来表达。这个520比特的数字以前缀04开头，紧接着是x及y坐标，组成格式为04 x y：

```
K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E
```

压缩格式公钥

引入压缩格式公钥是为了减少比特币交易的字节数，从而可以节省那些运行区块链数据库的节点磁盘空间。大部分比特币交易包含了公钥，用于验证用户的凭据和支付比特币。每个公钥有520比特（包括前缀，x坐标，y坐标）。如果每个区块有数百个交易，每天有成千上万的交易发生，区块链里就会被写入大量的数据。

正如我们在“[4.1.4 公钥](#)”一节所见，一个公钥是一个椭圆曲线上的点(x, y)。而椭圆曲线实际是一个数学方程，曲线上的点实际是该方程的一个解。因此，如果我们知道了公钥的x坐标，就可以通过解方程 $y^2 \bmod p = (x^3 + 7) \bmod p$ 得到y坐标。这种方案可以让我们只存储公钥的x坐标，略去y坐标，从而将公钥的大小和存储空间减少了256比特。每个交易所需要的字节数减少了近一半，随着时间推移，就大大节省了很多数据传输和存储。

未压缩格式公钥使用04作为前缀，而压缩格式公钥是以02或03作为前缀。需要这两种不同前缀的原因是：因为椭圆曲线加密的公式的左边是 y^2 ，也就是说y的解是来自于一个平方根，可能是正值也可能是负值。更形象地说，y坐标可能在x坐标轴的上面或者下面。从图4-2的椭圆曲线图中可以看出，曲线是对称的，从x轴看就像对称的镜子两面。因此，如果我们略去y坐标，就必须储存y的符号（正值或者负值）。换句话说，对于给定的x值，我们需要知道y值在x轴的上面还是下面，因为它们代表椭圆曲线上不同的点，即不同的公钥。当我们在素数p阶的有限域上使用二进制算术计算椭圆曲线的时候，y坐标可能是奇数或者偶数，分别对应前面所讲的y值的正负符号。因此，为了区分y坐标的两种可能值，我们在生成压缩格式公钥时，如果y是偶数，则使用02作为前缀；如果y是奇数，则使用03作为前缀。这样就可以根据公钥中给定的x值，正确推导出对应的y坐标，从而将公钥解压缩为在椭圆曲线上的完整的点坐标。下图阐释了公钥压缩：

公钥压缩

[x , y]

公钥
在曲线上的
x 和 y
坐标



04 x y

非压缩公钥
的十六进制
前缀为04

y为偶数

y为奇数

02 x

若y 为偶数
则压缩公钥
的十六进制
前缀为02

03 x

若y 为奇数
则压缩公钥
的十六进制
前缀为03

下面是前述章节所生成的公钥，使用了264比特（66个十六进制数字）的压缩格式公钥格式，其中前缀03表示y坐标是一个奇数：

K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

这个压缩格式公钥对应着同样的一个私钥，这意味着它是由同样的私钥所生成。但是压缩格式公钥和非压缩格式公钥差别很大。更重要的是，如果我们使用双哈希函数(RIPEMD160(SHA256(K)))将压缩格式公钥转化成比特币地址，得到的地址将会不同于由非压缩格式公钥产生的地址。这种结果会让人迷惑，因为一个私钥可以生成两种不同格式的公钥——压缩格式和非压缩格式，而这两种格式的公钥可以生成两个不同的比特币地址。但是，这两个不同的比特币地址的私钥是一样的。

压缩格式公钥渐渐成为了各种不同的比特币客户端的默认格式，它可以大大减少交易所需的字节数，同时也让存储区块链所需的磁盘空间变小。然而，并非所有的客户端都支持压缩格式公钥，于是那些较新的支持压缩格式公钥的客户端就不得不考虑如何处理那些来自较老的不支持压缩格式公钥的客户端的交易。这在钱包应用导入另一个钱包应用的私钥的时候就会变得尤其重要，因为新钱包需要扫描区块链并找到所有与这些被导入私钥相关的交易。比特币钱包应该扫描哪个比特币地址呢？新客户端不知道应该使用哪个公钥：因为不论是通过压缩的公钥产生的比特币地址，还是通过非压缩的公钥产生的地址，两个都是合法的比特币地址，都可以被私钥正确签名，但是他们是完全不同的比特币地址。

为了解决这个问题，当私钥从钱包中被导出时，较新的比特币客户端将使用一种不同的钱包导入格式（Wallet Import Format）。这种新的钱包导入格式可以用来表明该私钥已经被用来生成压缩的公钥，同时生成的比特币地址也是基于该压缩的公钥。这个方案可以解决导入私钥来自于老钱包还是新钱包的问题，同时也解决了通过公钥生成的比特币地址是来自于压缩格式公钥还是非压缩格式公钥的问题。最后新钱包在扫描区块链时，就可以使用对应的比特币地址去查找该比特币地址在区块链里所发生的交易。我们将在下一节详细解释这种机制是如何工作的。

压缩格式私钥

实际上“压缩格式私钥”是一种名称上的误导，因为当一个私钥被使用WIF压缩格式导出时，不但没有压缩，而且比“非压缩格式”私钥长出一个字节。这个多出来的一个字节是私钥被加了后缀01，用以表明该私钥是来自于一个较新的钱包，只能被用来生成压缩的公钥。私钥是非压缩的，也不能被压缩。“压缩的私钥”实际上只是表示“用于生成压缩格式公钥的私钥”，而“非压缩格式私钥”用来表明“用于生成非压缩格式公钥的私钥”。为避免更多误解，应该只可以说导出格式是“WIF压缩格式”或者“WIF”，而不能说这个私钥是“压缩”的。

要注意的是，这些格式并不是可互换使用的。在较新的实现了压缩格式公钥的钱包中，私钥只能且永远被导出为WIF压缩格式（以K或L为前缀）。对于较老的没有实现压缩格式公钥的钱包，私钥将只能被导出为WIF格式（以5为前缀）导出。这样做的目的就是为了给导入这些私钥的钱包一个信号：到底是使用压缩格式公钥和比特币地址去扫描区块链，还是使用非压缩格式公钥和比特币地址。

如果一个比特币钱包实现了压缩格式公钥，那么它将会在所有交易中使用该压格式缩公钥。钱包中的私钥将会被用来生成压缩格式公钥，压缩格式公钥然后被用来生成交易中的比特币地址。当从一个实现了压缩格式公钥的比特币钱包导出私钥时，钱包导入格式（WIF）将会被修改为WIF压缩格式，该格式将会在私钥的后面附加一个字节大小的后缀01。最终的Base58Check编码格式的私钥被称作WIF（“压缩”）私钥，以字母“K”或“L”开头。而以“5”开头的是从较老的钱包中以WIF（非压缩）格式导出的私钥。

表4-4展示了同样的私钥使用不同的WIF和WIF压缩格式编码。

表4-4 示例：同样的私钥，不同的格式

格式	私钥
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Hex-compressed	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD01
WIF-compressed	KxFc1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawrtJ



“压缩格式私钥”是一个不当用词！私钥不是压缩的。WIF压缩格式的私钥只是用来表明他们只能被生成压缩的公钥和对应的比特币地址。相反地，“WIF压缩”编码的私钥还多出一个字节，因为这种私钥多了后缀“01”。该后缀是用来区分“非压缩格式”私钥和“压缩格式”私钥。

4.3 用Python实现密钥和比特币地址

最全面的比特币Python库是 Vitalik Buterin写的 [pybitcointools](#)。在例4-4中，我们使用pybitcointools库（导入为“bitcoin”）来生成和显示不同格式的密钥和比特币地址。

例4-4 使用pybitcointools库的密钥和比特币地址的生成和格式化过

```
import bitcoin

# Generate a random private key
valid_private_key = False while not valid_private_key:
    private_key = bitcoin.random_key()
    decoded_private_key = bitcoin.decode_privkey(private_key, 'hex')
    valid_private_key = 0 < decoded_private_key < bitcoin.N

print "Private Key (hex) is: ", private_key
print "Private Key (decimal) is: ", decoded_private_key

# Convert private key to WIF format
wif_encoded_private_key = bitcoin.encode_privkey(decoded_private_key, 'wif')
print "Private Key (WIF) is: ", wif_encoded_private_key

# Add suffix "01" to indicate a compressed private key
compressed_private_key = private_key + '01'
print "Private Key Compressed (hex) is: ", compressed_private_key

# Generate a WIF format from the compressed private key (WIF-compressed)
wif_compressed_private_key = bitcoin.encode_privkey(
    bitcoin.decode_privkey(compressed_private_key, 'hex'), 'wif')
print "Private Key (WIF-Compressed) is: ", wif_compressed_private_key

# Multiply the EC generator point G with the private key to get a public key point
public_key = bitcoin.base10_multiply(bitcoin.G, decoded_private_key) print "Public Key (x,y) coordinates is:", public_key

# Encode as hex, prefix 04
hex_encoded_public_key = bitcoin.encode_pubkey(public_key, 'hex') print "Public Key (hex) is:", hex_encoded_public_key

# Compress public key, adjust prefix depending on whether y is even or odd
(public_key_x, public_key_y) = public_key if (public_key_y % 2) == 0:
    compressed_prefix = '02'
else:
    compressed_prefix = '03'
hex_compressed_public_key = compressed_prefix + bitcoin.encode(public_key_x, 16) print "Compressed Public Key (hex) is:", hex_compressed_public_key

# Generate bitcoin address from public key
print "Bitcoin Address (b58check) is:", bitcoin.pubkey_to_address(public_key)

# Generate compressed bitcoin address from compressed public key
print "Compressed Bitcoin Address (b58check) is:", \ bitcoin.pubkey_to_address(hex_compressed_public_key)
```

例4-5显示了上段代码运行结果。

例4-5 运行 key-to-address-ecc-example.py

```
$ python key-to-address-ecc-example.py
Private Key (hex) is:
3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa6
Private Key (decimal) is:
26563230048437957592232553826663696440606756685920117476832299673293013768870
Private Key (WIF) is:
5JG9hT3beGTJuUAmCQEmNaxAuMacCTfXuw1R3FCXig23RQHMr4K
Private Key Compressed (hex) is:
3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa601
Private Key (WIF-Compressed) is:
KyBsPXXtUVD82av65KZkrGrwi5qLMah5SdNq6uftawDbgKa2wv6S
Public Key (x,y) coordinates is:
(41637322786646325214887832269588396900663353932545912953362782457239403430124L,
16388935128781238405526710466724741593761085120864331449066658622400339362166L)
```

```
Public Key (hex) is:  
045c0de3b9c8ab18dd04e3511243ec2952002dbfadec864b9628910169d9b9b00ec-  
243bcfedd4347074d44bd7356d6a53c495737dd96295e2a9374bf5f02ebfc176  
Compressed Public Key (hex) is:  
025c0de3b9c8ab18dd04e3511243ec2952002dbfadec864b9628910169d9b9b00ec  
Bitcoin Address (b58check) is:  
1thMirt546nngXqyPEz532S8fLwbozud8  
Compressed Bitcoin Address (b58check) is:  
14cxpo3MBCYYWcgF74SwTdcxmipnGUspW3
```

例4-6是另外一个示例，使用的是Python ECDSA库来做椭圆曲线计算而非使用bitcoin的库。

例4-6 使用在比特币密钥中的椭圆曲线算法的脚本

例4-7显示了运行脚本的结果。

例4-7 安装Python ECDSA库，运行ec_math.py脚本

```
running the ec_math.py script
$ # Install Python PIP package manager
$ sudo apt-get install python-pip
$ # Install the Python ECDSA library
$ sudo pip install ecdsa
$ # Run the script
$ python ec-math.py
Secret:
38090835015954358862481132628887443905906204995912378278060168703580660294000
EC point:
(70048853531867179489857750497606966272382583471322935454624595540007269312627,
105262206478686743191060800263479589329920209527285803935736021686045542353380)
BTC public key: 029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a1522873
```

4.4 比特币钱包

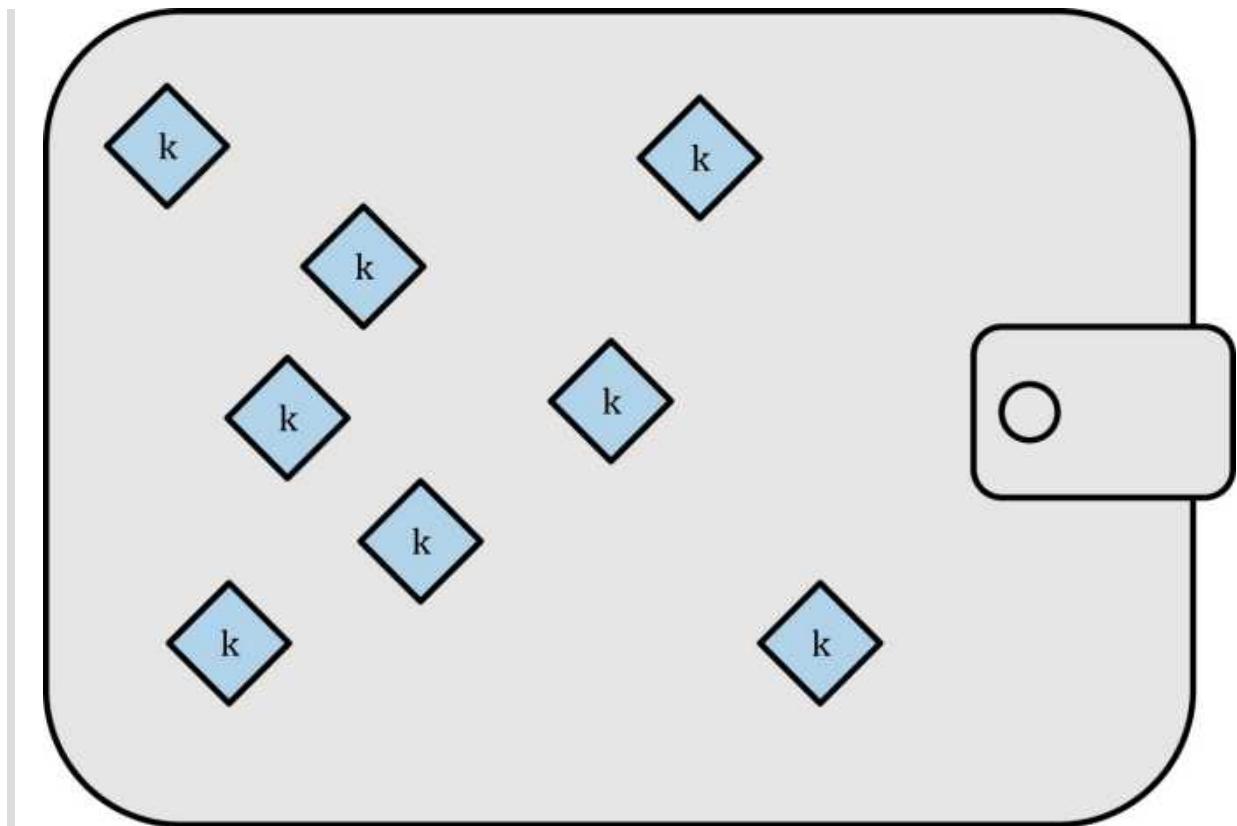
钱包是私钥的容器，通常通过有序文件或者简单的数据库实现。另外一种制作私钥的途径是确定性密钥生成。在这里你可以用原先的私钥，通过单向哈希函数来生成每一个新的私钥，并将新生成的密钥按顺序连接。只要你可以重新创建这个序列，你只需要第一个私钥（称作种子、主私钥）来生成它们。在本节中，我们将会检查不同的私钥生成方法及其钱包结构。



比特币钱包只包含私钥而不是比特币。每一个用户有一个包含多个私钥的钱包。钱包中包含成对的私钥和公钥。用户用这些私钥来签名交易，从而证明它们拥有交易的输出（也就是其中的比特币）。比特币是以交易输出的形式来储存区块链中（通常记为vout或txout）。

4.4.1 非确定性（随机）钱包

在最早的一批比特币客户端中，钱包只是随机生成的私钥集合。这种类型的钱包被称作零型非确定钱包。举个例子，比特币核心客户端预先生成100个随机私钥，从最开始就生成足够多的私钥并且每把钥匙只使用一次。这种类型的钱包有一个昵称“Just a Bunch Of Keys（一堆私钥）”简称JBOK。这种钱包现在正在被确定性钱包替换，因为它们难以管理、备份以及导入。随机钥匙的缺点就是如果你生成很多，你必须保存它们所有的副本。这就意味着这个钱包必须被经常性地备份。每一把钥匙都必须备份，否则一旦钱包不可访问时，钱包所控制的资金就付之东流。这种情况直接与避免地址重复使用的原则相冲突——每个比特币地址只能用一次交易。地址通过关联多重交易和对方的地址重复使用会减少隐私。0型非确定性钱包并不是钱包的好选择，尤其是当你不想重复使用地址而创造过多的私钥并且要保存它们。虽然比特币核心客户包含0型钱包，但比特币的核心开发者并不想鼓励大家使用。下图表示包含有松散结构的随机钥匙的集合的非确定性钱包。



4.4.2 确定性（种子）钱包

确定性，或者“种子”钱包包含通过使用单项离散方程而可从公共的种子生成的私钥。种子是随机生成的数字。这个数字也含有比如索引号码或者可生成私钥的“链码”（参见[“4.4.4 分层确定性钱包（BIP0032/BIP0044）”一节](#)）。在确定性钱包中，种子足够收回所有的已经产生的私钥，所以只用在初始创建时的一个简单备份就足以搞定。并且种子也足够让钱包输入或者输出。这就很容易允许使用者的私钥在钱包之间轻松转移输入。

4.4.3 助记码词汇

助记码词汇是英文单词序列代表（编码）用作种子对应所确定性钱包的随机数。单词的序列足以重新创建种子，并且从种子那里重新创造钱包以及所有私钥。在首次创建钱包时，带有助记码的，运行确定性钱包的钱包的应用程序将会向使用者展示一个12至24个词的顺序。单词的顺序就是钱包的备份。它也可以被用来恢复以及重新创造应用程序相同或者兼容的钱包的钥匙。助记码代码可以让使用者复制钱包更容易一些，因为它们相比较随机数字顺序来说，可以很容易地被读出来并且正确抄写。

助记码被定义在比特币的改进建议39中（参见[“附录 2 比特币改进协议\[BIP0039\]”](#)），目前还处于草案状态。需注意的是，BIP0039是一个建议草案而不是标准方案。具体地来说，电子钱包和BIP0039使用不同的标准且对应不同组的词汇。Trezor钱包以及一些其他钱包使用BIP0039，但是BIP0039和电子钱包的运行不兼容。

BIP0039定义助记码和种子的创建过程如下：

- 1.创造一个128到256位的随机顺序（熵）。
- 2.提出SHA256哈希前几位，就可以创造一个随机序列的校验和。
- 3.把校验和加在随机顺序的后面。
- 4.把顺序分解成11位的不同集合，并用这些集合去和一个预先已经定义的2048个单词字典做对应。
- 5.生成一个12至24个词的助记码。

表4-5表示了熵数据的大小和助记码单词的长度之间的关系。

表4-5 助记码：熵及字段长度

熵 (bits)	校验符 (bits)	熵 + 校验符	字段长
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

助记码表示128至256位数。这可以通过使用私钥抻拉函数PBKDF2来导出更长的（512位）的种子。所得的种子可以用来创造一个确定性钱包以及其所派生的所有钥匙。

表4-6和表4-7展示了一些助记码的例子和它所生成的种子。

表4-6 128位熵的助记码以及所产生的种子

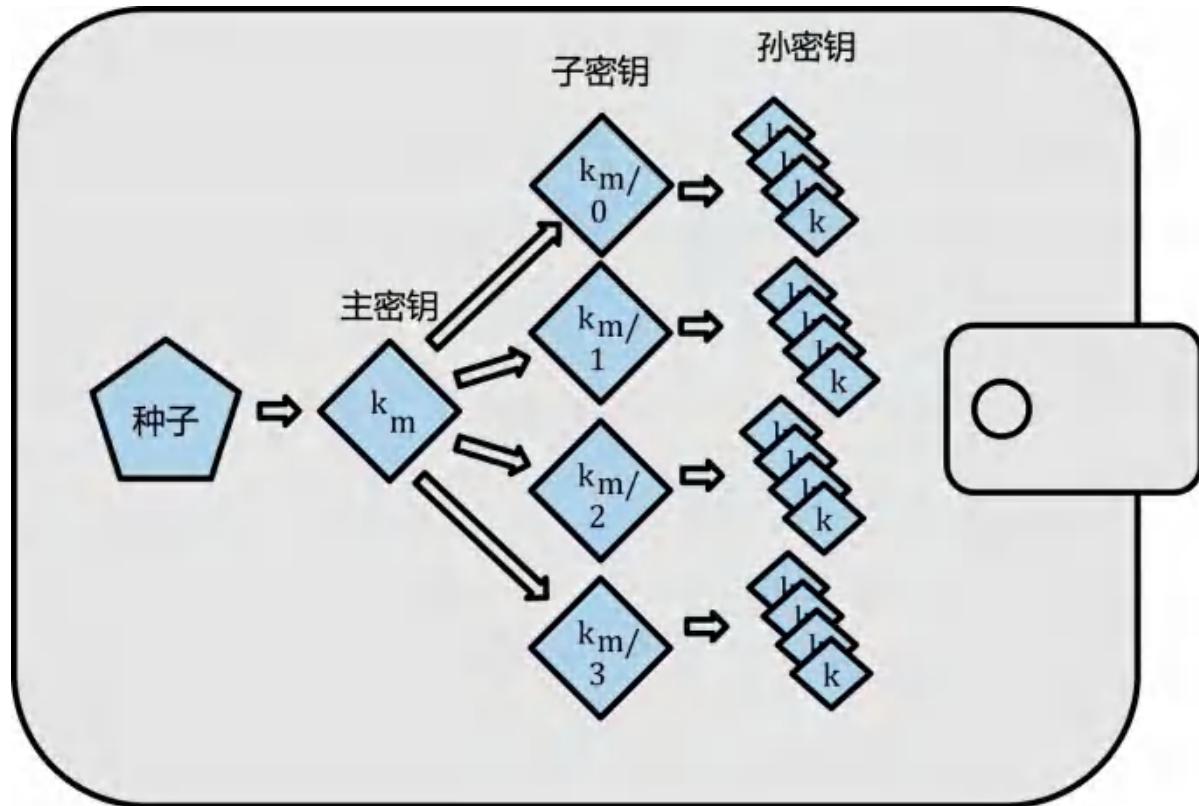
负熵输入 (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
助记码 (12 个单词)	army van defense carry jealous true garbage claim echo media make crunch
种子 (512 bits)	3338a6d2ee71c7f28eb5b882159634cd46a898463e9d2d0980f8e80dfbba5b0fa0291e5fb888a599b44b93187be6ee3ab5fd3ead7dd646341b2cdb8d08d13bf

表4-7 256位熵的助记码以及所产生的种子

负熵输入 (256 bits)	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
助记码 (24个单词)	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
种子 (512 bits)	3972e432e99040f75ebe13a660110c3e29d131a2c808c7ee5f1631d0a977fcf473bee22fce540af281bf7cdeade0dd2c1c795bd02f1e4049e205a0158906c343

4.4.4 分层确定性钱包 (BIP0032/BIP0044)

确定性钱包被开发成更容易从单个“种子”中生成许多关键的钥匙。最高级的来自确定性钱包的形是通过BIP0032标准生成的the hierarchical deterministic wallet or HD wallet defined。分层确定性钱包包含从数结构所生成的钥匙。这种母钥匙可以生成子钥匙的序列。这些子钥匙又可以衍生出孙钥匙，以此无穷类推。这个树结构表如下图所示。



如果你想安装运行一个比特币钱包，你需要建造一个符合BIP0032和BIP0044标准的HD钱包。

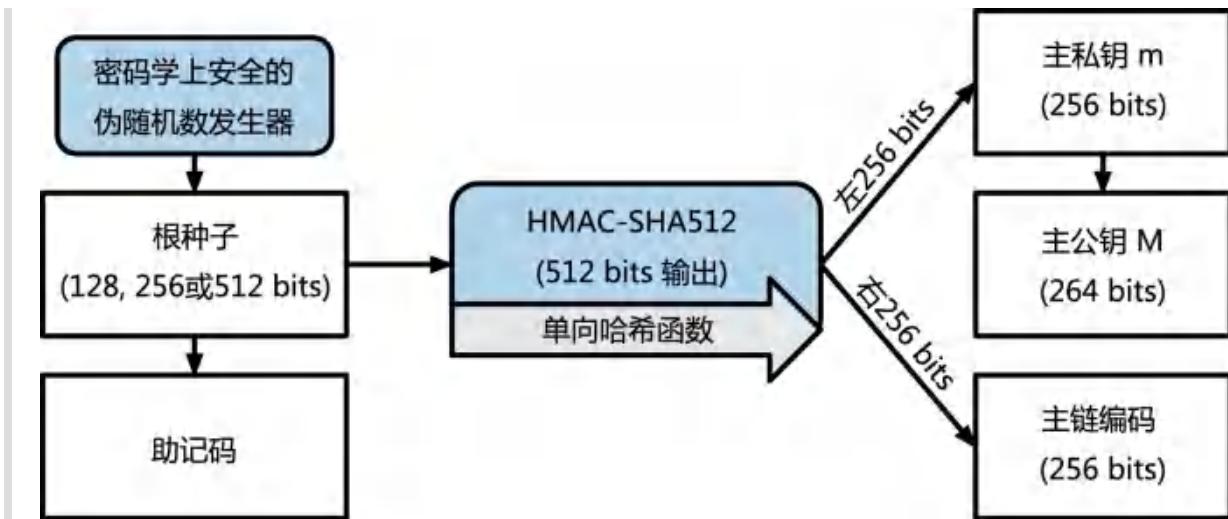
HD钱包提供了随机（不确定性）钥匙有两个主要的优势。第一，树状结构可以被用来表达额外的组织含义。比如当一个特定分支的子密钥被用来接收交易收入并且有另一个分支的子密钥用来负责支付花费。不同分支的密钥都可以被用在企业环境中，这就可以支配不同的分支部门，子公司，具体功能以及会计类别。

HD钱包的第二个好处就是它可以允许让使用者去建立一个公共密钥的序列而不需要访问相对应的私钥。这可允许HD钱包在不安全的服务器中使用或者在每笔交易中发行不同的公共钥匙。公共钥匙不需要被预先加载或者提前衍生，但是在服务器中不具有可用来支付的私钥。

从种子中创造HD钱包

HD钱包从单个root seed中创建，为128到256位的随机数。HD钱包的所有确定性都衍生自这个根种子。任何兼容HD钱包的根种子也可重新创造整个HD钱包。所以简单的转移HD钱包的根种子就让HD钱包中所包含的成千上百万的密钥被复制，储存导出以及导入。根种子一般总是被表示为a mnemonic word sequence，正如[“4.4.3 助记码词汇”](#)一节所表述的，助记码词汇可以让人们更容易地抄写和储存。

创建主密钥以及HD钱包地主链代码的过程如下图所示。



根种子输入到HMAC-SHA512算法中就可以得到一个可用来创造master private key(m) 和 a master chain code的哈希。主私钥 (m) 之后可以通过使用我们在本章先前看到的那个普通椭圆曲线 $m * g$ 过程生来成相对应的主公钥 (M) 。链代码可以从母密钥中创造子密钥的那个方程中引入的熵。

私有子密钥的衍生

分层确定性钱包使用CKD (child key derivation)方程去从母密钥衍生出子密钥。

子密钥衍生方程是基于单项哈希方程。这个方程结合了：

- 一个母私钥或者公共钥匙 (ECDSA未压缩键)
- 一个叫做链码 (256 bits) 的种子
- 一个索引号 (32 bits)

链码是用来给这个过程引入看似的随机数据的，使得索引不能充分衍生其他的子密钥。因此，有了子密钥并不能让它发现自己的相似子密钥，除非你已经有了链码。最初的链码种子（在密码树的根部）是用随机数据构成的，随后链码从各自的母链码中衍生出来。

这三个项目相结合并散列可以生成子密钥，如下。

母公共钥匙——链码——以及索引号合并在一起并且用HMAC-SHA512方程散列之后可以产生512位的散列。所得的散列可被拆分为两部分。散列右半部分的256位产出可以给子链当链码。左半部分256位散列以及索引码被加载在母私钥上来衍生子私钥。在图4-11中，我们看到这种这个说明——索引集被设为0去生产母密钥的第0个子密钥（第一个通过索引）。

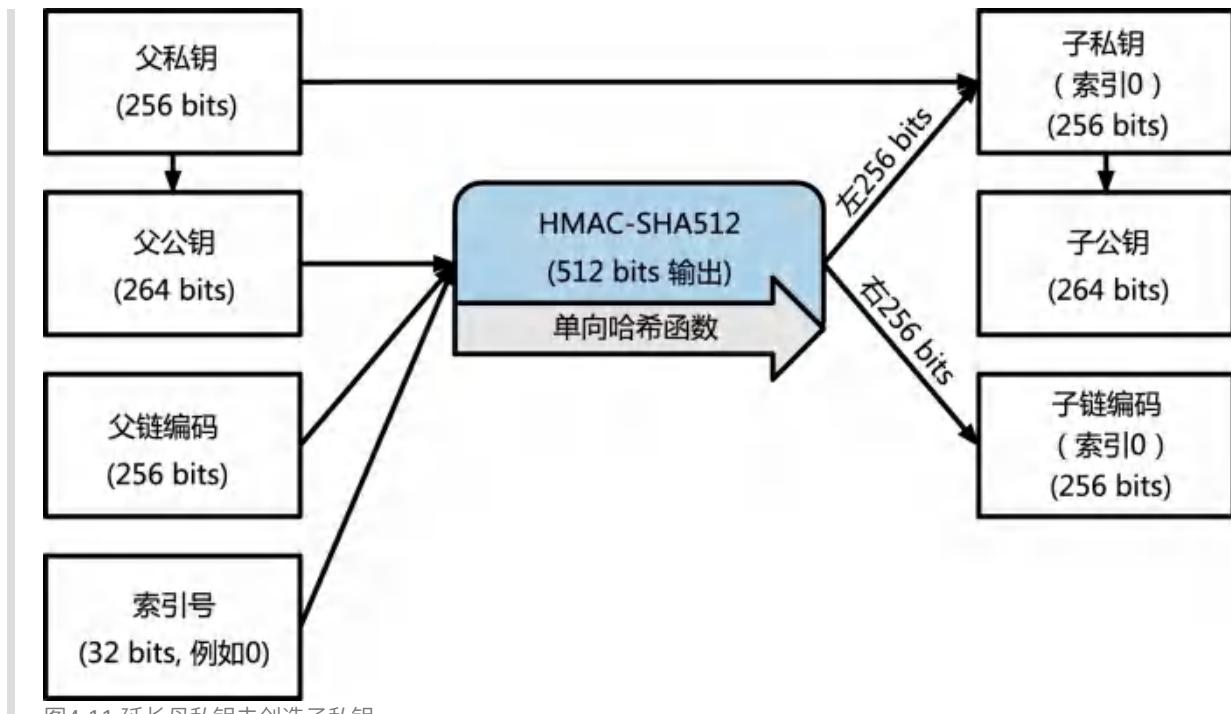


图4-11 延长母私钥去创造子私钥

改变索引可以让我们延长母密钥以及创造序列中的其他子密钥。比如子0，子1，子2等等。每一个母密钥可以有20亿个子密钥。

向密码树下一层重复这个过程，每个子密钥可以依次成为母密钥继续创造它自己的子密钥，直到无限代。

使用衍生的子密钥

子密钥不能从非确定性（随机）密钥中被区分出来。因为衍生方程是单向方程，所以子密钥不能被用来发现他们的母密钥。子密钥也不能用来发现他们的相同层级的姊妹密钥。如果你有第n个子密钥，你不能发现它前面的（第n-1）或者后面的子密钥（n+1）或者在同一顺序中的其他子密钥。只有母密钥以及链码才能得到所有的子密钥。没有子链码的话，子密钥也不能用来衍生出任何孙密钥。你需要同时有子密钥以及对应的链码才能创建一个新的分支来衍生出孙密钥。

那子密钥自己可被用做什么呢？它可以用来做公共钥匙和比特币地址。之后它就可以被用那个地址来签署交易和支付任何东西。



子密钥、对应的公共钥匙以及比特币地址都不能从随机创造的密钥和地址中被区分出来。事实是它们所在的序列，在创造他们的HD钱包方程之外是不可见的。一旦被创造出来，它们就和“正常”钥匙一样运行了。

扩展密钥

正如我们之前看到的，密钥衍生方程可以被用来创造钥匙树上任何层级的子密钥。这只需要三个输入量：一个密钥，一个链码以及想要的子密钥的索引。密钥以及链码这两个重要的部分被结合之后，就叫做extended key。术语“extended key”也被认为是“可扩展的密钥”是因为这种密钥可以用来衍生子密钥。

扩展密钥可以简单地被储存并且表示为简单的将256位密钥与256位链码所并联的512位序列。有两种扩展密钥。扩展的私钥是私钥以及链码的结合。它可被用来衍生子私钥（子私钥可以衍生子公共密钥）公共钥匙以及链码组成扩展公共钥匙。这个钥匙可以用来扩展子公共钥匙，见“[4.1.6 生成公钥](#)”。

想象一个扩展密钥作为HD钱包中钥匙树结构的一个分支的根。你可以衍生出这个分支的剩下所有部分。扩展私人钥匙可以创建一个完整的分支而扩展公共钥匙只能够创造一个公共钥匙的分支。



一个扩展钥匙包括一个私钥（或者公共钥匙）以及一个链码。一个扩展密钥可以创造出子密钥并且能创造出在钥匙树结构中的整个分支。分享扩展钥匙就可以访问整个分支。

扩展密钥通过Base58Check来编码，从而能轻易地在不同的BIP0032-兼容钱包之间导入导出。扩展密钥编码用的Base58Check使用特殊的版本号，这导致在Base58编码字符中，出现前缀“xprv”和“xpub”。这种前缀可以让编码更易被识别。因为扩展密钥是512或者513位，所以它比我们之前所看到的Base58Check-encoded串更长一些。

这是一个在Base58Check中编码的扩展私钥的例子：

```
xprv9tyUQV64JT5qs3RSTJkXcwKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6GoNMKUga5biW6Hx4tws2six3b9c
```

这是在Base58Check中编码的对应的扩展公共钥匙：

```
xpub67xpozcx8pe95XVuZLHXZeG6XwXHpGq6Qv5cmNfi7cS5mtjJ2tgyeQbBs2UAR6KECeeMVKZBPLrtJunSDMstweyLXhRgPxdp14sk9tJPW9
```

公共子钥匙推导

正如之前提到的，分层确定性钱包的一个很有用的特点就是可以不通过私钥而直接从公共母钥匙派生出公共子钥匙的能力。这就给了我们两种去衍生子公共钥匙的方法：或者通过子私钥，再或者就是直接通过母公共钥匙。

因此，扩展的公共钥匙可以再HD钱包结构的分支中，被用来衍生所有的公钥（且只有公共钥匙）。

这种快捷方式可以用来创造非常保密的public-key-only配置。在配置中，服务器或者应用程序不管有没有私钥，都可以有扩展公共钥匙的副本。这种配置可以创造出无限数量的公共钥匙以及比特币地址。但是不可以花送到这个地址里的任何比特币。与此同时，在另一种更保险的服务器上，扩展私钥可以衍生出所有的对应的可签署交易以及花钱的私钥。

这种方案的一个常见的方案是安装一个扩展的公共钥匙在服务电商公共程序的网络服务器上。网络服务器可以使用这个公共钥匙衍生方程去给每一笔交易（比如客户的购物车）创造一个新的比特币地址。但为了避免被偷，网络服务商不会有任何私钥。没有HD钱包的话，唯一的方法就是在不同的安全服务器上造成千上万个比特币地址，之后就提前上传到电商服务器上。这种方法比较繁琐而且要求持续的维护来确保电商服务器不“用光”公共钥匙。

这种解决方案的另一种常见的应用是冷藏或者硬件钱包。在这种情况下，扩展的私钥可以被储存在纸质钱包中或者硬件设备中（比如Trezor硬件钱包）与此同时扩展公共钥匙可以在线保存。使用者可以根据意愿创造“接收”地址而私钥可以安全地在线下被保存。为了支付资金，使用者可以使用扩展的私钥离线签署比特币客户或者通过硬件钱包设备（比如Trezor）签署交易。图4-12阐述了扩展母公共钥匙来衍生子公共钥匙的传递机制。

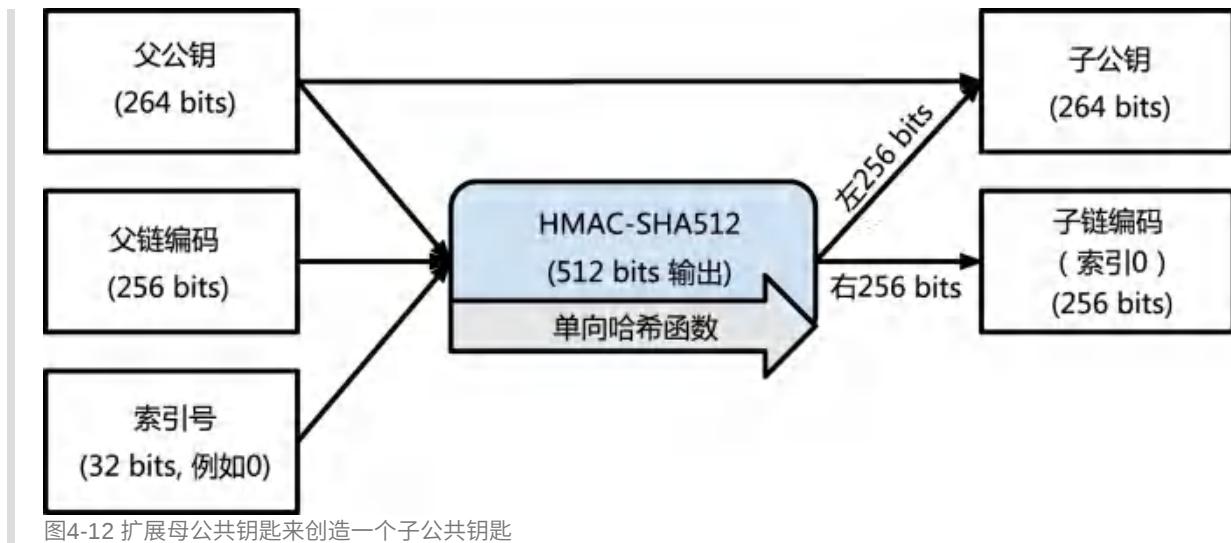


图4-12 扩展母公共钥匙来创造一个子公共钥匙

硬化子密钥的衍生

从扩展公共钥匙衍生一个分支公共钥匙是很重要的，但牵扯一些风险。访问扩展公共钥匙并不能得到访问子私人密钥的途径。但是，因为扩展公共钥匙包含有链码，如果子私钥被知道或者被泄漏的话，链码就可以被用来衍生所有的其他子私钥。一个简单地泄露的私钥以及一个母链码，可以暴露所有的子密钥。更糟糕的是，子私钥与母链码可以用来推断母私钥。

为了应对这种风险，HD钱包使用一种叫做hardened derivation的替代衍生方程。这就“打破”了母公共钥匙以及子链码之间的关系。这个硬化衍生方程使用了母私钥去推到子链码，而不是母公共钥匙。这就在母/子顺序中创造了一道“防火墙”——有链码但并不能够用来推算子链码或者姊妹私钥。强化的衍生方程看起来几乎与一般的衍生的子私钥相同，不同的是母私钥被用来输入散列方程中而不是母公共钥匙，如图4-13所示。

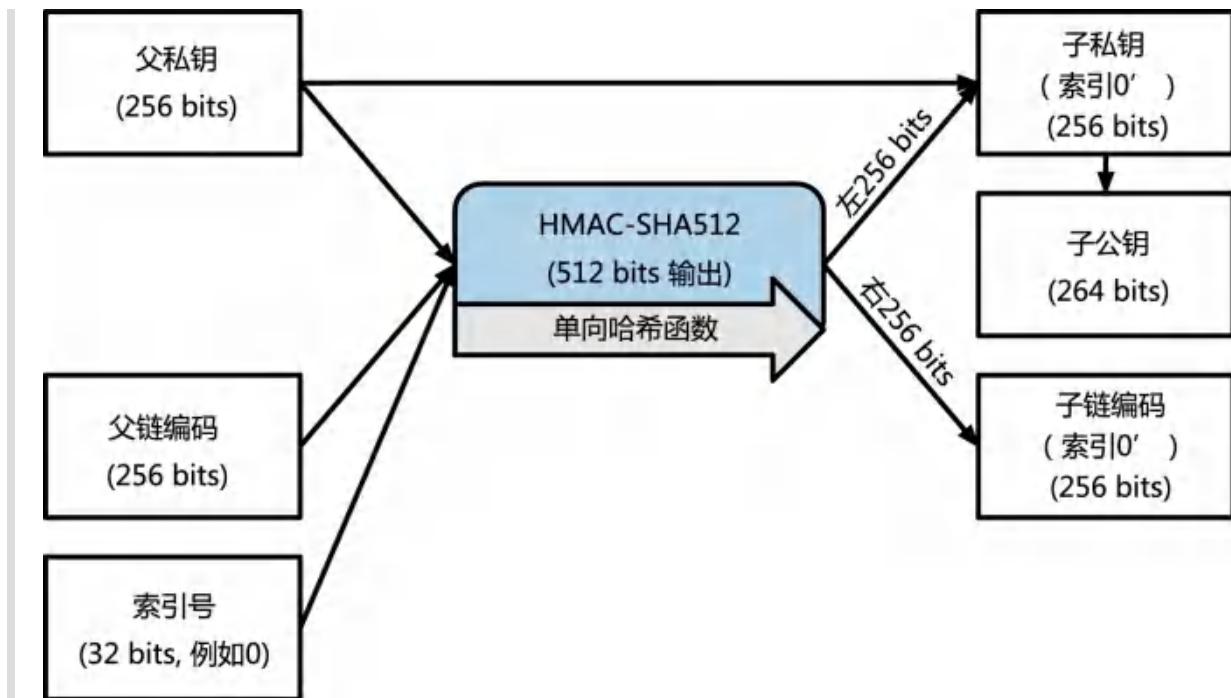


图4-13 子密钥的强化衍生；忽略了母公共密钥

当强化私钥衍生方程被使用时，得到的子私钥以及链码与使用一般衍生方程所得到的结果完全不同的。得到的密钥“分支”可以被用来生产不易被攻击的扩展公共钥匙，因为它所含的链码不能被用来开发或者暴露任何私钥。强化的衍生也因此被用来在上一层级，使用扩展公共钥匙的密钥树中创造“间隙”。

简单地来说，如果你想要利用扩展公共钥匙的便捷来衍生公共钥匙的分支而不将你自己暴露在泄露扩展链码的风险下，你应该从强化母私钥，而不是一般的母私钥，来衍生公共钥匙。最好的方式是，为了避免了推到出主钥匙，主钥匙所衍生的第一

层级的子钥匙最好使用强化衍生。

正常衍生和强化衍生的索引号码

用在衍生方程中的索引号码是32位的整数。为了区分密钥是从正常衍生方程中衍生出来还是从强化衍生方程中产出，这个索引号被分为两个范围。索引号在0和 $2^{31}-1$ (0x0 to 0x7FFFFFFF)之间的是只被用在常规衍生。索引号在 2^{31} 和 $2^{32}-1$ (0x80000000 to 0xFFFFFFFF)之间的只被用在强化衍生方程。因此，索引号小于 2^{31} 就意味着子密钥是常规的，而大于或者等于 2^{31} 的子密钥就是强化型的。

为了让索引号码更容易被阅读和展示，强化子密码的索引号码是从0开始展示的，但是右上角有一个小撇号。第一个常规子密钥因此被表述为0，但是第一个强化子密钥（索引号为0x80000000）就被表示为0'。第二个强化密钥依序有了索引号0x80000001，且被显示为1'，以此类推。当你看到HD钱包索引号*i'*，这就意味着 $2^{31}+i$ 。

HD钱包密钥识别符（路径）

HD钱包中的密钥是用“路径”命名的，且每个级别之间用斜杠 (/) 字符来表示（见表4-8）。由主私钥衍生出的私钥起始以“m”打头。因此，第一个母密钥生成的子私钥是m/0。第一个公共钥匙是M/0。第一个子密钥的子密钥就是m/0/1，以此类推。

密钥的“祖先”是从右向左读，直到你达到了衍生出的它的主密钥。举个例子，标识符m/x/y/z描述的是子密钥m/x/y的第z个子密钥。而子密钥m/x/y又是m/x的第y个子密钥。m/x又是m的第x个子密钥。

表4-8 HD钱包路径的例子

HD path	密钥描述
m/0	从主私钥 (m) 衍生出的第一个 (0) 子密钥。
m/0/0	第一个私人子密钥 (m/0) 的子密钥。
m/0/0	第一个子强化密钥first hardened child (m/0') 的第一个常规子密钥。
m/1/0	第2个子密钥 (m/1) 的第一个常规子密钥
M/23/17/0/0	主密钥衍生出的第24个子密钥所衍生出的第17个子密钥的第一个子密钥所衍生出的第一个子密钥。

HD钱包树状结构的导航

HD钱包树状结构提供了极大的灵活性。每一个母扩展密钥有40已个子密钥：20亿个常规子密钥和20亿个强化子密钥。而每个子密钥又会有40亿个子密钥并且以此类推。只要你愿意，这个树结构可以无限类推到无穷代。但是，又由于有了这个灵活性，对无限的树状结构进行导航就变得异常困难。尤其是对于在不同的HD钱包之间进行转移交易，因为内部组织到内部分支以及亚分支的可能性是无穷的。

两个比特币改进建议（BIPs）提供了这个复杂问题的解决办法——通过创建几个HD钱包树的提议标准。BIP0043提出使用第一个强化子索引作为特殊的标识符表示树状结构的“purpose”。基于BIP0043，HD钱包应该使用且只用第一层级的树的分支，而且有索引号码去识别结构并且有命名空间来定义剩余的树的目的地。举个例子，HD钱包只使用分支m/i'是为了表明那个被索引号“i”定义的特殊目的地。

在BIP0043标准下，为了延长的那个特殊规范，BIP0044提议了多账户结构作为“purpose”。所有遵循BIP0044的HD钱包依据只使用树的第一个分支的要求而被定义：m/44'。

BIP0044指定了包含5个预定义树状层级的结构：

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

第一层的目的地总是被设定为44'。第二层的“coin_type”特指密码货币硬币的种类并且允许多元货币HD钱包中的货币在第二个层级下有自己的亚树状结构。目前有三种货币被定义：Bitcoin is m/44'/0'、Bitcoin Testnet is m/44'/1'，以及Litecoin is m/44'/2'。

树的第三层级是“account”，这可以允许使用者为了会计或者组织目的，而去再细分他们的钱包到独立的逻辑性亚账户。举个例子，一个HD钱包可能包含两个比特币“账户”：m/44'/0'0' 和 m/44'/0'1'。每个账户都是它自己亚树的根。

第四层级就是“change”。每一个HD钱包有两个亚树，一个是用来接收地址一个是用来创造变更地址。注意无论先前的层级是否使用是否使用强化衍生，这一层级使用的都是常规衍生。这是为了允许这一层级的树可以在可供不安全环境下，输出扩展的公共钥匙。被HD钱包衍生的可用的地址是第四层级的子级，就是第五层级的树的“address_index”。比如，第三个层级的主账户收到比特币支付的地址就是 M/44'/0'0'0/2。表4-9展示了更多的例子。

表4-9 BIP0044 HD 钱包结构的例子

HD 路径	主要描述
M/44'/0'0'0/2	第三个收到公共钥匙的主比特币账户
M/44'/0'3/1/14	第十五改变地址公钥的第四个比特币账户
m/44'/2'0'0/1	为了签署交易的在莱特币主账户的第二个私钥

使用比特币浏览器实验比特币钱包

依据第3章介绍的使用比特币浏览器命令工具，你可以试着生产和延伸BIP0032确定性密钥以及将它们用不同的格式进行展示：

```
$ sx hd-seed > m # create a new master private key from a seed and store in file "m"
$ cat m # show the master extended private key
96 | Chapter 4: Keys, Addresses, Wallets
xprv9s21ZrQH143K38iQ9Y5p6qoB8C75TE71NfpyQPdfGvzghDt39DHPFpovvtWZaR- gY5uPwV7RpEgHs7cvdgf1SjLjjbuGKGcjRyU7RGGS8Xa
$ cat m | sx hd-pub 0 # generate the M/0 extended public key xpub67xpozcx8pe95XVuZLHXZeG6XwKHpGq6Qv5cmNfi7cS5mtjJ2tgyeqBbs2UAR6
$ cat m | sx hd-priv 0 # generate the m/0 extended private key xprv9tyUQV64JT5qs3RSTJkXCWkMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy
$ cat m | sx hd-priv 0 | sx hd-to-wif # show the private key of m/0 as a WIF L1pbvV86crAGoDzqmgY85xURkz3c435Z9nirMt52UbnGjYMzKBU
$ cat m | sx hd-pub 0 | sx hd-to-address # show the bitcoin address of M/0 1CHCnCjgMNb6digimckNQ6TBVcTwBAmPHK
$ cat m | sx hd-priv 0 | sx hd-priv 12 --hard | sx hd-priv 4 # generate m/0/12'/4 xprv9yL8ndfdPVeDwJenF18oiHguRUj8jHmVrqD97YQH
```

4.5 高级密钥和地址

在以下部分中，我们将看到高级形式的密钥和地址，诸如加密私钥、脚本和多重签名地址，靓号地址，和纸钱包。

4.5.1 加密私钥（BIP0038）

私钥必须保密。私钥的机密性需求事实情况是，在实践中相当难以实现，因为该需求与同样重要的安全对象可用性相互矛盾。当你需要为了避免私钥丢失而存储备份时，会发现维护私钥私密性是一件相当困难的事情。通过密码加密内有私钥的钱包可能要安全一点，但那个钱包也需要备份。有时，例如用户因为要升级或重装钱包软件，而需要把密钥从一个钱包转移到另一个。私钥备份也可能需要存储在纸张上（参见“4.5.4 纸钱包”一节）或者外部存储介质里，比如U盘。但如果一旦备份文件失窃或丢失呢？这些矛盾的安全目标推进了便携、方便、可以被众多不同钱包和比特币客户端理解的加密私钥标准BIP0038的出台。

BIP0038提出了一个通用标准，使用一个口令加密私钥并使用Base58Check对加密的私钥进行编码，这样加密的私钥就可以安全地保存在备份介质里，安全地在钱包间传输，保持密钥在任何可能被暴露情况下的安全性。这个加密标准使用了AES，这个标准由NIST建立，并广泛应用于商业和军事应用的数据加密。

BIP0038加密方案是：输入一个比特币私钥，通常使用WIF编码过，base58check字符串的前缀“5”。此外BIP0038加密方案需

要一个长密码作为口令，通常由多个单词或一段复杂的数字字母字符串组成。BIP0038加密方案的结果是一个由base58check编码过的加密私钥，前缀为6P。如果你看到一个6P开头的密钥，这就意味着该密钥是被加密过，并需要一个口令来转换（解码）该密钥回到可被用在任何钱包WIT格式的私钥（前缀为5）。许多钱包APP现在能够识别BIP0038加密过的私钥，会要求用户提供口令解码并导入密钥。第三方APP，诸如非常好用基于浏览器的Bit Address，可以被用来解码BIP00038的密钥。

最通常使用BIP0038加密的密钥用例是纸钱包——一张纸张上备份私钥。只要用户选择了强口令，使用BIP0038加密的私钥纸钱包就无比的安全，是一种很棒的线下比特币存储途径（也被称作“冷库”）。

在bitaddress.org上测试表4-10中加密密钥，看看你如何得到输入口令的加密密钥。

表4-10 BIP0038加密私钥例子

私钥 (WIF)	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
密码	MyTestPassphrase
加密私钥 (BIP0038)	6PRTHL6mWa48xSopbU1cKrVjpKbBZxcLRRCdctLJ3z5yxE87MobKoXdTsJ

4.5.2 P2SH (Pay-to-Script Hash)和多重签名地址

正如我们所知，传统的比特币地址从数字1开头，来源于公钥，而公钥来源于私钥。虽然任何人都可以将比特币发送到一个1开头的地址，但比特币只能在通过相应的私钥签名和公钥哈希值后才能消费。

以数字3开头的比特币地址是P2SH地址，有时被错误的称谓多重签名或多重签名地址。他们指定比特币交易中受益人作为哈希的脚本，而不是公钥的所有者。这个特性在2012年1月由BIP0016引进，目前因为BIP0016提供了增加功能到地址本身的机会而被广泛的采纳。不同于发送资金到传统1开头的比特币地址的交易，也被称为P2PKH，资金被发送到3开头的地址时，不仅仅需要一个公钥的哈希值，同时也需要一个私钥签名作为所有者证明。在创建地址的时候，这些要求会被定义在脚本中，所有对地址的输入都会被这些要求阻隔。

一个P2SH地址从事务脚本中创建，它定义谁能消耗这个事务输出。（132页“P2SH (Pay-to-Script-Hash) ”一节对此有详细的介绍）编码一个P2SH地址涉及使用一个在创建比特币地址用到过的双重哈希函数，并且只能应用在脚本而不是公钥：

```
script hash = RIPEMD160(SHA256(script))
```

脚本哈希的结果是由Base58Check编码前缀为5的版本、编码后得到开头为3的编码地址。一个P2SH地址例子是32M8ednmuyZ2zVbes4puqe44NZumgG92sM。



P2SH 不一定是多重签名的交易。虽然P2SH地址通常都是代表多重签名，但也可能是其他类型的交易脚本。

4.5.2.1 多重签名地址和P2SH

目前，P2SH函数最常见的实现是用于多重签名地址脚本。顾名思义，底层脚本需要多个签名来证明所有权，此后才能消费资金。设计比特币多重签名特性是需要从总共N个密钥中需要M个签名（也被称为“阈值”），被称为M-的-N的多签名，其中M是等于或小于N。例如，第一章中提到的咖啡店主鲍勃使用多重签名地址需要1-2签名，一个是属于他的密钥和一个属于他同伴的密钥，以确保其中一方可以签署度过一个事务锁定输出到这个地址。这类似于传统的银行中的一个“联合账户”，其中任何一方配偶可以凭借单一签名消费。或Gopesh，Bob雇佣的网页设计师创立一个网站，可能为他的业务需要一个2-3的多签名地址，确保没有资金会被花费除非至少两个业务合作伙伴签署这笔交易。

我们将会在第五章节探索如何使用P2SH地址创建事务用来消费资金。

4.5.3 比特币靓号地址

靓号地址包含了可读信息的有效比特币地址。例如，1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33就是包含了Base-58字母love的。靓号地址需要生成并通过数十亿的候选私钥测试，直到一个私钥能生成具有所需图案的比特币地址。虽然有一些优化过的靓号生成算法，该方法必须涉及随机上选择一个私钥，生成公钥，再生成比特币地址，并检查是否与所要的靓号图案相匹配，重复数十亿次，直到找到一个匹配。

一旦找到一个匹配所要图案的靓号地址，来自这个靓号地址的私钥可以和其他地址相同的方式被拥有者消费比特币。靓号地址不比其他地址具有更多安全性。它们依靠和其他地址相同的ECC和SHA。你无法比任何别的地址更容易的获得一个靓号图案开头的私钥。

在第一章中，我们介绍了Eugenia，一位在菲律宾工作的儿童慈善总监。我们假设Eugenia组织了一场比特币募捐活动，并希望使用靓号比特币地址来宣布这个募捐活动。Eugenia将会创造一个以1Kids开头的靓号地址来促进儿童慈善募捐的活动。让我们看看这个靓号地址如何被创建，这个靓号地址对Eugenia慈善募捐的安全性又意味着什么。

4.5.3.1 生成靓号地址

我们必须认识到使用来自Base58字母表中简单符号来代表比特币地址是非常重要的。搜索“1kids”开头的图案我们会发现从1Kids111111111111111111111111111111到1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzz的地址。这些以“1kid”开头的地址范围内大约有58的29次方地址。表4-11显示了这些有“1kids”前缀的地址。

表4-11 “1Kids”靓号的范围

From	1Kids111111111111111111111111111111
To	1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzzz

我们把“1Kids”这个前缀当作数字，我们可以看看比特币地址中这个前缀出现的频率。如果是一台普通性能的桌面电脑，没有任何特殊的硬件，可以每秒发现大约10万个密钥。

表4-12 靓号的出现的频率（1KidsCharity）以及生成所需时间

长度	地址前缀	概率	平均生成时间
1	1K	1/58	< 1毫秒
2	1Ki	1/3,364	50毫秒
3	1Kid	1/(195*10 ³)	< 2秒
4	1Kids	1/(11*10 ⁶)	1分钟
5	1KidsC	1/(656*10 ⁶)	1小时
6	1KidsCh	1/(38*10 ⁹)	2天
7	1KidsCha	1/(2.2*10 ¹²)	3–4月
8	1KidsChar	1/(128*10 ¹²)	13–18年
9	1KidsChari	1/(7*10 ¹⁵)	800年
10	1KidsCharit	1/(400*10 ¹⁵)	46,000年
11	1KidsCharity	1/(23*10 ¹⁸)	250万年

正如你所见，Eugenia将不会很快地创建出以“1KidsCharity”开头的靓号地址，即使她有数千台的电脑同时进行运算。每增加

一个字符就会增加58倍的计算难度。超过七个字符的搜索模式通常需要专用的硬件才能被找出，譬如用户定制的具有多图形处理单元（GPU）的桌面级设备。那些通常是无法继续在比特币挖矿中盈利的钻机，被重新赋予了寻找靓号地址的任务。用GPU系统搜索靓号的速度比用通用CPU要快很多个量级。

另一种寻找靓号地址的方法是将工作外包给一个矿池里的靓号矿工们，如[靓号矿池](#)中的矿池。一个矿池是一种允许那些GPU硬件通过为他人寻找靓号地址来获得比特币的服务。对小额的账单，Eugenia可以外包搜索模式为7个字符靓号地址寻找工作，在几个小时内就可以得到结果，而不必用一个CPU搜索上几个月才得到结果。

生成一个靓号地址是一项通过蛮力的过程：尝试一个随机密钥，检查结果地址是否和所需的图案想匹配，重复这个过程直到成功找到为止。例4-8是个靓号矿工的例子，用C++程序来寻找靓号地址。这个例子运用到了我们在56页“其他替代客户端、资料库、工具包”一节介绍过的libbitcoin库。

例4-8 靓号挖掘程序

```
#include <bitcoin/bitcoin.hpp>

// The string we are searching for
const std::string search = "1kid";

// Generate a random secret key. A random 32 bytes.
bc::ec_secret random_secret(std::default_random_engine& engine); // Extract the Bitcoin address from an EC secret.
std::string bitcoin_address(const bc::ec_secret& secret);
// Case insensitive comparison with the search string.
bool match_found(const std::string& address);

int main()
{
    std::random_device random;
    std::default_random_engine engine(random());
    // Loop continuously...
    while (true)
    {
        // Generate a random secret.
        bc::ec_secret secret = random_secret(engine);
        // Get the address.
        std::string address = bitcoin_address(secret);
        // Does it match our search string? (1kid)
        if (match_found(address))
        {
            // Success!
            std::cout << "Found vanity address! " << address << std::endl;
            std::cout << "Secret: " << bc::encode_hex(secret) << std::endl; return 0;
        }
    }
    // Should never reach here!
    return 0;
}

bc::ec_secret random_secret(std::default_random_engine& engine)
{
    // Create new secret...
    bc::ec_secret secret;
    // Iterate through every byte setting a random value... for (uint8_t& byte: secret)
    byte = engine() % std::numeric_limits<uint8_t>::max();
    // Return result.
    return secret;
}

std::string bitcoin_address(const bc::ec_secret& secret)
{
    // Convert secret to pubkey...
    bc::ec_point pubkey = bc::secret_to_public_key(secret);
    // Finally create address.
    bc::payment_address payaddr; bc::set_public_key(payaddr, pubkey);
    // Return encoded form.
    return payaddr.encoded();
}

bool match_found(const std::string& address)
{
    auto addr_it = address.begin();
```

```

// Loop through the search string comparing it to the lower case
// character of the supplied address.
for (auto it = search.begin(); it != search.end(); ++it, ++addr_it)
    if (*it != std::tolower(*addr_it))
        return false;
// Reached end of search string, so address matches.
return true;
}

```

示例程序需要用C编译器链接libbitcoin库（此库需要提前装入该系统）进行编译。直接执行vanity-miner的可执行文件（不用参数，参见例4-9），它就会尝试碰撞以“1kid”开头的比特币地址。

例4-9 编译并运行vanity-miner程序示例

```

$ # Compile the code with g++
$ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin) $ # Run the example
$ ./vanity-miner
Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT
Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f
$ # Run it again for a different result
$ ./vanity-miner
Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn
Secret: 7f65bbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623
# Use "time" to see how long it takes to find a result
$ time ./vanity-miner
Found vanity address! 1KidPWhKgGRQwD5PP5TAnGfDyfWp5yceXM
Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349

real    0m8.868s
user    0m8.828s
sys     0m0.035s

```

正如我们运行Unix命令time所测出的运行时间所示，示例代码要花几秒钟来找出匹配“kid”三个字符模板的结果。读者们可以在源代码中改变search这一搜索模板，看一看如果是四个字符或者五个字符的搜索模板需要花多久时间！

4.5.3.2 靓号地址安全性

靓号地址既可以增加、也可以削弱安全措施，它们着实是一把双刃剑。用于改善安全性时，一个独特的地址使对手难以使用他们自己的地址替代你的地址，以欺骗你的顾客支付他们的账单。不幸的是，靓号地址也可能使得任何人都能创建一个类似于随机地址的地址，甚至另一个靓号地址，从而欺骗你的客户。

Eugenia可以让捐款人捐款到她宣布的一个随机生成地址（例如：1J7mdg5rbQyUHENYdx39WWK7fsLpEoXZy）。或者她可以生成一个以“1Kids”开头的靓号地址以显得更独特。

在这两种情况下，使用单一固定地址（而不是每笔捐款用一个独立的动态地址）的风险之一是小偷有可能会黑进你的网站，用他自己的网址取代你的网址，从而将捐赠转移给自己。如果你在不同的地方公布了你的捐款地址，你的用户可以在付款之前直观地检查以确保这个地址跟在你的网站、邮件和传单上看到的地址是同一个。在随机地址1j7mdg5rbqyuhenydx39wwk7fslpeoxzy的情况下，普通用户可能会只检查头几个字符“1j7mdg”，就认为地址匹配。使用靓号地址生成器，那些想通过替换类似地址来盗窃的人可以快速生成与前几个字符相匹配的地址，如表4-13所示。

表4-13 生成匹配某随机地址的多个靓号

原版随机地址	1J7mdg5rbQyUHENYdx39WWK7fsLpEoXZy
4位字符匹配	1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy
5位字符匹配	1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n
6位字符匹配	1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX

那靓号地址会不会增加安全性？如果Eugenia生成1Kids33q44erFfpeXrmDSz7zEqG2FesZEN的靓号地址，用户可能看到靓

号图案的字母和一些字符在上面，例如在地址部分中注明了1Kids33。这样就会迫使攻击者生成至少6个字母相匹配的靓号地址（比之前多2个字符），就要花费比Eugenia多3364倍的靓号图案。本质上，Eugenia付出的努力（或者靓号池付出的）迫使攻击者不得不生成更长的靓号图案。如果Eugenia花钱请矿池生成8个字符的靓号地址，攻击者将会被逼迫到10字符的境地，那将是个人电脑，甚至昂贵自定义靓号挖掘机或靓号池也无法生成。对Eugenia来说可承担的起支出，对攻击者来说则变成了无法承担支出，特别是如果欺诈的回报不足以支付生成靓号地址所需的费用。

4.5.4 纸钱包

纸钱包是打印在纸张上的比特币私钥。有时纸钱包为了方便起见也包括对应的比特币地址，但这并不是必要的，因为地址可以从私钥中导出。纸钱包是一个非常有效的建立备份或者线下存储比特币（即冷钱包）的方式。作为备份机制，一个纸钱包可以提供安全性，以防在电脑硬盘损坏、失窃或意外删除的情况下造成密钥的丢失。作为一个冷存储的机制，如果纸钱包密钥在线下生成并永久不在电脑系统中存储，他们在应对黑客攻击，键盘记录器，或其他在线电脑欺骗更有安全性。

纸钱包有许多不同的形状，大小，和外观设计，但非常基本的原则是一个密钥和一个地址打印在纸张上。表4-14展现了纸钱包最基本的形式。

表4-14 比特币纸钱包的私钥和公钥的打印形式

公开地址	1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x
私钥 (WIF)	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

通过使用工具，就可以很容易地生成纸钱包，譬如使用bitaddress.org网站上的客户端Javascript生成器。这个页面包含所有必要的代码，甚至在完全失去网络连接的情况下，也可以生成密钥和纸钱包。若要使用它，先将HTML页面保存在本地磁盘或外部U盘。从Internet网络断开，从浏览器中打开文件。更方便的，使用一个原始操作系统启动电脑，比如一个光盘启动的Linux系统。任何在脱机情况下使用这个工具所生成的密钥，都可以通过USB线在本地打印机上打印出来，从而制造了密钥只存在纸张上而从未存储在在线系统上的纸钱包。将这些纸钱包放置在防火容器内，发送比特币到对应的比特币地址上，从而实现了一个简单但非常有效的冷存储解决方案。图4-14展示了通过bitaddress.org生成的纸钱包。



图4-14 通过bitaddress.org 生成的普通纸钱包

这个简单的纸钱包系统的不足之处是那些被打印下来的密钥容易被盗窃。一个能够获取接近这些纸币的小偷可以只需偷走纸币或者用把纸币上密钥拍摄下来，就能获得被这些密钥加密过的比特币的控制权。一个更复杂的纸钱包存储系统使用BIP0038加密的私钥。这些私钥被打印在纸钱包上被所有者记住的口令保护起来。没有口令，这些被加密过的密钥也是毫无用处的。但它们仍旧要比用口令保护的钱包级别要高，因为这些密钥从没有在线过，必须从物理上从保险箱或者其他物理安全存储中导出。图4-15展示了通过bitaddress.org生成的加密纸钱包。



图4-15 通过bitaddress.org 生成的加密纸钱包。密码是“test”。



虽然你可以多次存款到纸钱包中，但是你最好一次性提款，一次性提取里面所有的资金。因为如果你提取的金额少于其中的金额的话，会生成一个找零地址。并且，你所用的电脑可能被病毒感染，那么就有可能泄露私钥。一次性提款可以减少私钥泄露的风险，如果你所需的金额比较少，那么请把余额找零到另一个纸钱包中。

纸钱包有许多设计和大小，并有许多不同的特性。有些作为礼物送给他，有季节性的主题，像圣诞节和新年主题。另外一些则是设计保存在银行金库或通过某种方式隐藏私钥的保险箱内，或者用不透明的刮刮贴，或者折叠和防篡改的铝箔胶密封。图4-16至图4-18展示了几个不同安全和备份功能的纸钱包的例子。

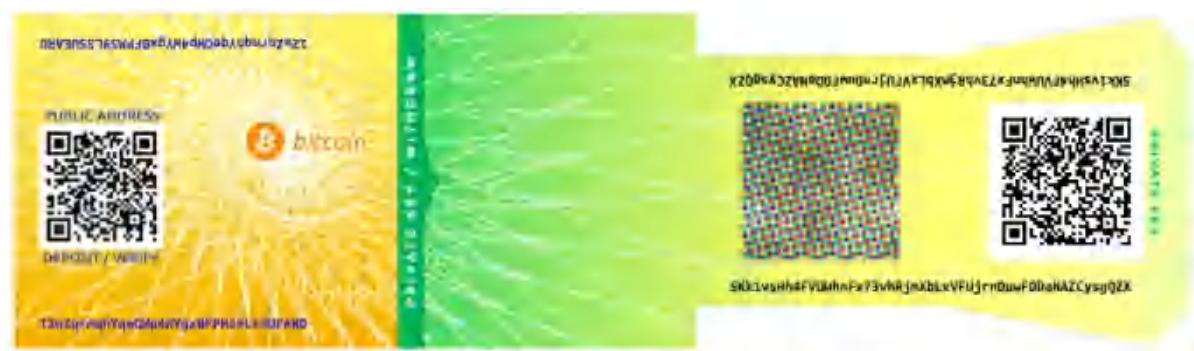


图4-16 通过bitcoinpaperwallet.com生成的、私钥写在折叠皮瓣上的纸钱包



图4-17 通过bitcoinpaperwallet.com 生成的、私钥被密封住的纸钱包

其他设计有密钥和地址的额外副本，类似于票根形式的可以拆卸存根，让你可以存储多个副本以防火灾、洪水或其他自然灾害。



图4-18 在备份“存根”上有多个私钥副本的纸钱包

第5章 交易

5.1 简介

比特币交易是比特币系统中最重要的部分。根据比特币系统的[设计原理](#)，系统中任何其他的部分都是为了确保比特币交易可以被生成、能在比特币网络中得以传播和通过验证，并最终添加到全球比特币交易总账簿（比特币区块链）。比特币交易的本质是数据结构，这些数据结构中含有比特币交易参与者价值转移的相关信息。比特币区块链是全球复式记账总账簿，每个比特币交易都是在比特币区块链上的一个公开记录。

在这一章，我们将会剖析比特币交易的多种形式、所包含的信息、如何被创建、如何被验证以及如何成为所有比特币交易永久记录的一部分。

5.2 比特币交易的生命周期

一笔比特币交易的生命周期起始于它被创建的那一刻，也就是诞生（*origination*）。随后，比特币交易会被一个或者多个签名加密，这些签名标志着对该交易指向的比特币资金的使用许可。接下来，比特币交易被广播到比特币网络中。在比特币网络中，每一个节点（比特币交易参与者）验证、并将交易在网络中进行广播，直到这笔交易被网络中大多数节点接收。最终，比特币交易被一个挖矿节点验证，并被添加到区块链上一个记录着许多比特币交易的区块中。

一笔比特币交易一旦被记录到区块链上并被足够多的后续区块确认，便成为比特币总账簿的一部分，并被所有比特币交易参与者认可为有效交易。于是，被这笔交易分配到一个新所有者名下的比特币资金可以在新的交易中被使用——这使得所有权链得以延伸且再次开启一个新的比特币交易生命周期。

5.2.1 创建比特币交易

将一笔比特币交易理解成纸质支票或许有助于加深我们对它的理解。与支票相似，一笔比特币交易其实是一个有着货币转移目的的工具，这个工具只有在交易被执行时才会在金融体系中体现，而且交易发起人并不一定是签署该笔交易的人。

比特币交易可以被任何人在线上或线下创建，即便创建这笔交易的人不是这个账户的授权签字人。比如，一个负责应付账款的柜员在处理应付票据时可能会需要CEO签名。相似地，这个负责应付账款的柜员可以创建比特币交易，然后让CEO对它进行数字签名，从而使之有效。一张支票是指定一个特定账户作为资金来源的，但是比特币交易指定以往的一笔交易作为其资金来源，而不是一个特定账户。

一旦一笔比特币交易被创建，它会被资金所有者（们）签名。如果它是合法创建并签名的，则该笔交易现在就是有效的，它包含了转移这笔资金所需的所有信息。最终，有效的比特币交易必须能接入比特币网络，从而使之能被传送，直至抵达下一个登记在公共总账簿（区块链）的挖矿节点。

5.2.2 将比特币交易传送至比特币网络

首先，一笔交易需要传递至比特币网络，才能被传播，也才能加入区块链中。本质上，一笔比特币交易只是300到400字节的数据，而且它们必须被发送到成千上万个比特币节点中的任意一个。只要发送者能使用多于一个比特币节点来确保这笔交易被传播，那么发送者并不需要信任用来传播该笔交易的单一节点。相应地，这些节点不需要信任发送者，也不用建立发送者的“身份档案”。由于这笔交易是经过签名且不含任何机密信息、私钥或密码，因此它可被任何潜在的便利网络公开地传播。信用卡交易包含敏感信息，而且依赖加密网络连接完成信息传输，但比特币交易可在任意网络环境下被发送。只要这笔交易可以到达能将它广播到比特币网络的比特币节点，这笔交易是如何被传输至第一个节点的并不重要。

比特币交易因此可以通过未加密网络（例如WiFi、蓝牙、NFC、ChirP、条形码或者复制粘贴至一个网页表格）被发送到比特币网络。在一些极端情况下，一笔比特币交易可以通过封包无线电、卫星或短波、扩频或跳频以避免被侦测或阻塞通信的方式进行传输。一笔比特币交易甚至可被编为文字信息中的表情符号并被发表到在线论坛，或被发送成一条短信或一条Skype聊天信息。因为比特币将金钱变成了一种数据结构，所以在本质上是不可能阻止任何人创建并执行比特币交易的。

5.2.3 比特币交易在比特币网络中的传播

一旦一笔比特币交易被发送到任意一个连接至比特币网络的节点，这笔交易将会被该节点验证。如果交易被验证有效，该节点将会将这笔交易传播到这个节点所连接的其他节点；同时，交易发起者会收到一条表示交易成功的返回信息。如果这笔交易被验证为无效，这个节点会拒绝接受这笔交易且同时返回给交易发起者一条表示交易被拒绝的信息。

比特币网络是一个点对点网络，这意味着每一个比特币节点都连接到一些其他的比特币节点（这些其他的节点是在启动点对点协议时被发现的）。整个比特币网络形成了一个松散地连接、且没有固定拓扑或任何结构的“蛛网”——这使得所有节点的地位都是同等的。比特币交易相关信息（包括交易和区块）被传播——从每一个节点到它连接的其他节点。一笔刚通过验证且并被传递到比特币网络中任意节点的交易会被发送到三到四个相邻节点，而每一个相邻节点又会将交易发送到三至四个与它们相邻的节点。以此类推，在几秒钟之内，一笔有效的交易就会像指数级扩散的波一样在网络中传播，直到所有连接到网络的节点都接收到它。

比特币网络被设计为能高效且灵活地传递交易和区块至所有节点的模式，因而比特币网络能抵御入侵。为了避免垃圾信息的滥发、拒绝服务攻击或其他针对比特币系统的恶意攻击，每一个节点在传播每一笔交易之前均进行独立验证。一个异常交易所能到达的节点不会超过一个。["8.3 交易的独立校验"](#)一节将详细介绍决定比特币交易是否有效的原则。

5.3 交易结构

一笔比特币交易是一个含有输入值和输出值的数据结构，该数据结构植入了将一笔资金从初始点（输入值）转移至目标地址（输出值）的代码信息。比特币交易的输入值和输出值与账号或者身份信息无关。你应该将它们理解成一种被特定秘密信息锁定的一定数量的比特币。只有拥有者或知晓这个秘密信息的人可以解锁。一笔比特币交易包含一些字段，如表5-1所示。

表5-1 交易结构

大小	字段	描述
4字节	版本	明确这笔交易参照的规则
1-9字节	输入计数器	被包含的输入的数量
不定	输入	一个或多个交易输入
1-9字节	输出计数器	被包含的输入的数量
不定	输出	一个或多个交易输出
4字节	时钟时间	一个UNIX时间戳或区块号

交易的锁定时间

锁定时间定义了能被加到区块链里的最早的交易时间。在大多数交易里，它被设置成0，用来表示立即执行。如果锁定时间不是0并且小于5亿，就被视为区块高度，意指在这个指定的区块高度之前的交易没有被包含在这个区块链里。如果锁定时间大于5亿，则它被当作是一个Unix纪元时间戳（从1970年1月1日以来的秒数），并且在这个指定时点之前的交易没有被包含在这个区块链里。锁定时间的使用相当于将一张纸质支票的生效时间予以延后。

5.4 交易的输出和输入

比特币交易的基本单位是未经使用的一个交易输出，简称UTXO。UTXO是不能再分割、被所有者锁住或记录于区块链中的并被整个网络识别成货币单位的一定量的比特币货币。比特币网络监测着以百万为单位的所有可用的（未花费的）UTXO。当一个用户接收比特币时，金额被当作UTXO记录到区块链里。这样，一个用户的比特币会被当作UTXO分散到数百个交易和数百个区块中。实际上，并不存在储存比特币地址或账户余额的地点，只有被所有者锁住的、分散的UTXO。“一个用户的比特币余额”，这个概念是一个通过比特币钱包应用创建的派生之物。比特币钱包通过扫描区块链并聚合所有属于该用户的UTXO来计算该用户的余额。



在比特币的世界里既没有账户，也没有余额，只有分散到区块链里的UTXO。

一个UTXO可以是一“聪”的任意倍。就像美元可以被分割成表示两位小数的“分”一样，比特币可以被分割成表示八位小数的“聪”。尽管UTXO可以是任意值，但只要它被创造出来了，就像不能被切成两半的硬币一样不可再分了。如果一个UTXO比一笔交易所需量大，它仍会被当作一个整体而消耗掉，但同时会在交易中生成零头。例如，你有20比特币的UTXO并且想支付1比特币，那么你的交易必须消耗掉整个20比特币的UTXO并且产生两个输出：一个是支付了1比特币给接收人，另一个是支付19比特币的找零到你的钱包。这样的话，大部分比特币交易都会产生找零。

想象一下，一位顾客要买1.5元的饮料。她掏出她的钱包并努力从所有硬币和钞票中找出一种组合来凑齐她要支付的1.5元。如果可能的话，她会选刚刚好的零钱（比如一张1元纸币和5个一毛硬币）或者是小面额的组合（比如3个五毛硬币）。如果都不行的话，她会用一张大面额的钞票，比如5元纸币。如果她把过多的钱，比如5元，给了商店老板，她会拿到3.5元的找零，并把找零放回她的钱包以供未来使用。

类似的，一笔比特币交易可以有任意数值，但必须从用户可用的UTXO中创建出来。用户不能再把UTXO进一步细分，就像不能把一元纸币撕开而继续当货币使用一样。用户的钱包应用通常会从用户可用的UTXO中选取多个可用的个体来拼凑出一个大于或等于一笔交易所需的比特币量。

就像现实生活中一样，比特币应用可以使用一些策略来满足付款需要：组合若干小的个体，算出准确的找零；或者使用一个比交易值大的个体然后进行找零。所有这些复杂的、由可支付的UTXO完成的组合，都是由用户的钱包自动完成，并不为用户所见。只有当你以编程方式用UTXO来构建原始交易时，这些才与你有关。

被交易消耗的UTXO被称为交易输入，由交易创建的UTXO被称为交易输出。通过这种方式，一定量的比特币价值在不同所有者之间转移，并在交易链中消耗和创建UTXO。一笔比特币交易通过使用所有者的签名来解锁UTXO，并通过使用新的所有者的比特币地址来锁定并创建UTXO。

对于输出和输入链来说，有一个例外，它是一种特殊的交易类型，称为Coinbase交易。这是每个区块中的首个交易。这种交易存在的原因是作为对挖矿的奖励而产生全新的可用于支付的比特币给“赢家”矿工。这也就是为什么比特币可以在挖矿过程中被创造出来，我们将在第8章中进行详述。



输入和输出，哪一个是先产生的呢？先有鸡还是先有蛋呢？严格来讲，先产生输出，因为可以创造新比特币的coinbase交易没有输入，但它可以无中生有地产生输出。

5.4.1 交易输出

每一笔比特币交易创造输出，输出都会被比特币账簿记录下来。除特例之外（见“[5.7.4 数据输出（OP_RETURN操作符）](#)”），几乎所有的输出都能创造一定数量的可用于支付的比特币，也就是UTXO。这些UTXO被整个网络识别，并且所有者可在未来的交易中使用它们。给某人发送比特币实际上是创造新的UTXO，注册到那个人的地址，并且能被他用于新的支付。

UTXO被每一个全节点比特币客户端在一个储存于内存中的数据库所追踪，该数据库也被称为“UTXO集”或者“UTXO池”。新的交易从UTXO集中消耗（支付）一个或多个输出。

交易输出包含两部分：

▷一定量的比特币，被命名为“聪”，是最小的比特币单位；
一个锁定脚本，也被当作是“障碍”，提出支付输出所必须被满足的条件以“锁住”这笔总额。

在前面的锁定脚本中提到的这个交易脚本语言会在后面121页的“交易脚本和脚本语言”一节中详细讨论。表5-2列出了交易输出的结构。

表5-2 交易输出结构

尺寸	字段	说明
8个字节	总量	用聪表示的比特币值（10-8比特币）
1-9个字节（可变整数）	锁定脚本尺寸	用字节表示的后面的锁定脚本长度
变长	锁定脚本	一个定义了支付输出所需条件的脚本

在例5-1中，我们使用blockchain.info应用程序接口来查找特定地址的UTXO。

例5-1 一个调用blockchain.info应用程序接口来查找与一个地址有关的UTXO的脚本

```
# 从blockchain API中得到未花费的输出

import json
import requests

# 样例地址
address = '1Dorian4RoXcnBv9hnQ4Y2C1an6NJ4UrjX'

# API网址是: https://blockchain.info/unspent?active=<address>
# 它返回一个JSON对象，其中包括一个包含着UTXO的“unspent_outputs”列表，就像这样:
#{      "unspent_outputs":[
#{
#    "tx_hash":"ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167",
#    "tx_index":51919767,
#    "tx_output_n": 1,
#    "script":"76a9148c7e252f8d64b0b6e313985915110fcfefcf4a2d88ac",
#    "value": 8000000,
#    "value_hex": "7a1200",
#    "confirmations":28691
#  },
# ...
#}]}

resp = requests.get('https://blockchain.info/unspent?active=%s' % address)
utxo_set = json.loads(resp.text)["unspent_outputs"]

for utxo in utxo_set:
    print "%s:%d - %ld Satoshi" % (utxo['tx_hash'], utxo['tx_output_n'], utxo['value'])
```

运行脚本，我们将会得到“交易ID，冒号，特定UTXO的索引号，以及这个UTXO包含的聪的数额”的列表。在例5-2中，锁定脚本被省略了。

例5-2 运行get-utxo.py脚本

```
$ python get-utxo.py
ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167:1 - 8000000 Satoshi
6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 - 16050000 Satoshi
74d788804e2aae10891d72753d1520da1206e6f4f20481cc1555b7f2cb44aca0:0 - 5000000 Satoshi
b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4:0 - 10000000 Satoshi
...
```

支付条件（障碍）

交易输出把用聪表示的一定数量的比特币，和特定的定义了支付输出所必须被满足的条件的障碍，或者叫锁定脚本，关联到了一起。在大多数情况下，锁定脚本会把输出锁在一个特定的比特币地址上，从而把一定数量的比特币的所有权转移到新的所有者上。当Alice在Bob的咖啡店为一杯咖啡付款时，Alice的交易创造了0.015比特币的输出，在咖啡店的比特币地址上成为一种障碍，或者说是被锁在了咖啡店的比特币地址上。那0.015比特币输出被记录到区块链中，并且成为UTXO的一部分，也就是作为可用余额出现在Bob的钱包里。当Bob选择使用这笔款项进行支付时，他的交易会释放障碍，通过提供一个包含Bob私钥的解锁脚本来解锁输出。

5.4.2 交易输入

简单地说，交易输入是指向UTXO的指针。它们指向特定的UTXO，并被交易哈希和在区块链中记录UTXO的序列号作为参考。若想支付UTXO，一个交易的输入也需要包含一个解锁脚本，用来满足UTXO的支付条件。解锁脚本通常是一个签名，用来证明对于在锁定脚本中的比特币地址拥有所有权。

当用户付款时，他的钱包通过选择可用的UTXO来构造一笔交易。比如说，要支付0.015比特币，钱包应用会选择一个0.01 UTXO和一个0.005 UTXO，使用它们加在一起得到想要的付款额。

在例5-3中，我们展示了一种贪心算法来为了得到特定的付款额而选择可用的UTXO。在例中，可用的UTXO被提供在一个常数数组中。但在实际中，可用的UTXO被一个远程过程调用比特币核心，或者被一个如例5-1中的第三方应用程序接口，来检索出来。

例5-3 一个计算会被发送的比特币总量的脚步

```
# 使用贪心算法从UTXO列表中选择输出。
from sys import argv

class OutputInfo:

    def __init__(self, tx_hash, tx_index, value):
        self.tx_hash = tx_hash
        self.tx_index = tx_index
        self.value = value

    def __repr__(self):
        return "<%s:%s with %s Satoshis>" % (self.tx_hash, self.tx_index,
                                                   self.value)

# 为了发送，从未花费的输出列表中选出最优输出。
# 返回输出列表，并且把其他的改动发送到改变地址。
def select_outputs_greedy(unspent, min_value):
    # 如果是空的话认为是失败了。
    if not unspent: return None
    # 分割成两个列表。
    lessers = [utxo for utxo in unspent if utxo.value < min_value]      greater = [utxo for utxo in unspent if utxo.value >= min_value]
    key_func = lambda utxo: utxo.value
    if greater:
        # 非空。寻找最小的greater。
        min_greater = min(greater)
        change = min_greater.value - min_value
        return [min_greater], change
    # 没有找到greater。重新尝试若干更小的。
    # 从大到小排序。我们需要尽可能地使用最小的输入量。
    lessers.sort(key=key_func, reverse=True)
    result = []
    accum = 0
    for utxo in lessers:
        result.append(utxo)
        accum += utxo.value
        if accum >= min_value:
            change = accum - min_value
            return result, "Change: %d Satoshis" % change
    # 没有找到。
    return None, 0

def main():
    unspent = [
        OutputInfo("ebad
```

```

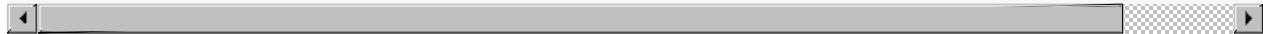
faa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167", 1, 8000000),
    OutputIn
fo("6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf", 0, 16050000),
    OutputInfo("b2af
fea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4", 0, 10000000),
    OutputIn
fo("7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1", 0, 25000000),
    OutputIn
fo("55ea01bd7e9af3d3ab9790199e777d62a0709cf0725e80a7350fdb22d7b8ec6", 17, 5470541),
    OutputIn
fo("12b6a7934c1df821945ee9ee3b3326d07ca7a65fd6416ea44ce8c3db0c078c64", 0, 10000000),
    OutputIn
fo("7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818", 0, 16100000),
]
]

if len(argv) > 1:
    target = long(argv[1])
else:
    target = 55000000

print "For transaction amount %d Satoshi's (%f bitcoin) use: " % (target, target/ 10.0**8)
print select_outputs_greedy(unspent, target)

if __name__ == "__main__":
    main()

```



如果我们不使用参数运行select-utxo.py脚本，它会试图为一笔五千五百万聪（0.55比特币）的付款构造一组UTXO。如果你提供一个指定的付款额作为参数，脚本会选择UTXO来完成指定的付款额。在例5-4中，我们运行脚本来试着完成一笔0.5比特币，或者说是五千万聪的付款。

例5-4 运行select-utxo.py

```

$ python select-utxo.py 50000000
For transaction amount 50000000 Satoshi's (0.500000 bitcoin) use:
([<7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1:0 with 25000000
Satoshi>, <7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818:0 with 16100000 Satoshi>,
<6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 with 16050000 Satoshi>], 'Change: 7150000 Satoshi')

```

一旦UTXO被选中，钱包会为每个UTXO生成包含签名的解锁脚本，由此让它们变得可以通过满足锁定脚本的条件来被支付。钱包把这些UTXO作为参考，并且连同解锁脚本一起作为输入加到交易中。表5-3展示了交易输入的结构。

表5-3 交易输入的结构

尺寸	字段	说明
32个字节	交易	指向交易包含的被花费的UTXO的哈希指针
4个字节	输出索引	被花费的UTXO的索引号，第一个是0
1–9个字节（可变整数）	解锁脚本尺寸	用字节表示的后面的解锁脚本长度
变长	解锁脚本	一个达到UTXO锁定脚本中的条件的脚本
4个字节	序列号	目前未被使用的交易替换功能，设成0xFFFFFFFF

序列号是用来覆盖在交易锁定时间之前失效的交易，这是一项目前没有在比特币中用到的功能。大多数交易把这个值设置成最大的整数（0xFFFFFFFF）并且被比特币网络忽略。如果一次交易有非零的锁定时间，那么它至少需要有一个序列号比0xFFFFFFFF低的输入来激活锁定时间。

5.4.3 交易费

大多数交易包含交易费，这是为了在网络安全方面给比特币矿工一种补偿。在第8章中，对于挖矿、费用和矿工得到的奖励，有更详细的讨论。这一节解释交易费是如何被包含在日常交易中的。大多数钱包自动计算并计入交易费。但是，如果你编程

构造交易，或者使用命令行接口，你必须手动计算并计入这些费用。

交易费可当作是为了包含（挖矿）一笔交易到下一个区块中的一种鼓励，也可当作是对于欺诈交易和任何种类的系统滥用，在每一笔交易上通过征收一笔小成本的税而造成的一种妨碍。交易费被挖出这个区块的矿工得到，并且记录在这个交易的区块链中。

交易费基于交易的尺寸，用千字节来计算，而不是比特币的价值。总的来说，交易费基于市场设置，生效于比特币网络中。矿工依据许多不同的标准，按重要性对交易进行排序，这包括费用，并且甚至可能在某种特定情况下免费处理交易。交易费影响处理优先级，这意味着有足够的费用的交易会更可能地被包含在下一个挖出的区块中；与此同时，交易费不足或者没有交易费的交易可能会被推迟，基于尽力而为的原则在几个区块之后被处理，甚至可能根本不被处理。交易费不是强制的，而且没有交易费的交易也许最终会被处理，但是，包含交易费将提高处理优先级。

随着时间的过去，交易费的计算方式和交易费在交易优先级上的影响一直在发展。起初，交易费是网络中的一个固定常数。渐渐地，交易费的结构被放宽了，以便被市场基于网络容量和交易量而强制影响。目前最小交易费被固定在每千字节0.0001比特币，或者说是每千字节万分之一比特币，最近一次改变是从千分之一比特币减少到这个数值的。大多数交易少于一千字节，但是那些包含多个输入和输出的交易尺寸可能更大。在未来的比特币协议修订版中，钱包应用预计会使用统计学分析，基于最近的几笔交易的平均费用，来计算最恰当的费用并附在交易上。

目前矿工使用的，对包含在一个区块中的交易，基于它们的费用划分优先顺序的算法，在第8章有详细解释。

5.4.4 把交易费加到交易中

交易的数据结构没有交易费的字段。相反地，交易费通过所有输入的总和，以及所有输出的总和之间的差来表示。从所有输入中扣掉所有输出之后的多余的量会被矿工收集走。

交易费被作为输入减输出的余量：

$$\text{交易费} = \text{求和}(\text{所有输入}) - \text{求和}(\text{所有输出})$$

对于交易来说，这是一个很让人摸不着头脑的元素，但又是很重要的问题。因为如果你要构造你自己的交易，你必须确认你没有疏忽地包含了一笔少于输入的、量非常大的费用。这意味着你必须计算所有的输入，如果必要的话进行找零，不然的话，结果就是你给了矿工一笔可观的劳动费！

举例来说，如果你消耗了一个20比特币的UTXO来完成1比特币的付款，你必须包含一笔19比特币的找零回到你的钱包。否则，那剩下的19比特币会被当作交易费，并且会被挖出你的交易到一个区块中的矿工收走。尽管你会受到高优先级的处理，并且让一个矿工喜出望外，但这很可能不是你想要的。



如果你忘记了在手动构造的交易中增加找零的输出，系统会把找零当作交易费来处理。“不用找了！”也许不是你想要的结果。

让我们来看看在实际中它如何工作，重温一下Alice在咖啡店的交易。Alice想为咖啡支付0.015比特币。为了确保这笔交易能立即被处理，Alice想支付一笔交易费，比如说0.001。这意味着总交易成本会变成0.016。因此她的钱包需要凑齐0.016或更多的UTXO。如果需要，还要加上找零。我们假设他的钱包有一个0.2比特币的UTXO可用。他的钱包就会消耗掉这个UTXO，创造一个新的0.015的输出给Bob的咖啡店，另一个0.184比特币的输出作为找零回到Alice拥有的钱包，并留下未分配的0.001比特币内含在交易中。

现在让我们换个例子。Eugenia，我们在菲律宾的儿童募捐项目主管，完成了一次为孩子购买教材的筹款活动。她在世界范围内接收到了好几千个小数额的捐款，总额是50比特币。所以她的钱包塞满了非常小的UTXO。现在她想用比特币从本地的一家出版商购买几百本的教材。

现在Eugenia的钱包应用想要构造一个单笔大额付款交易，它必须从可用的、由很多小数额构成的大的UTXO集合中寻求钱币来源。这意味着交易的结果是从上百个小数额的UTXO中作为输入，但只有一个输出用来付给出版商。输入数量这么巨大的交易会比一千字节要大，也许总尺寸会达到两至三千字节。结果是它需要更高的交易费来满足0.0001比特币的网络费。

Eugenia的钱包应用会通过测量交易的大小，乘以每千字节需要的交易费，来计算适当的交易费。很多钱包会通过多付交易费的方式来确保大交易被立即处理。高交易费不仅是因为Eugenia付的钱很多，还因为她的交易很复杂并且尺寸很大——交易费是与参加交易的比特币值无关的。

5.5 交易链条和孤立交易

正如我们之前所看到的那样，交易形成一条链，这条链的形式是一笔交易消耗了先前的交易（父交易）的输出，并为随后的交易（子交易）创造了输出。有的时候组成整个链条的所有交易依赖于他们自己——比如父交易、子交易和孙交易——而他们又被同时创造出来，来满足复杂交易的工作流程。这需要在一个交易的父交易被签名之前，有一个合法的子交易被签名。举个例子，这是CoinJoin交易使用的一项技术，这项技术可以让多方同时加入交易，从而保护他们的隐私。

当一条交易链被整个网络传送时，他们并不能总是按照相同的顺序到达目的地。有时，子交易在父交易之前到达。在这种情况下，节点会首先收到一个子交易，而不能找到他参考的父交易。节点不会立即抛弃这个子交易，而是放到一个临时池中，并等着接收它的父交易，与此同时广播这个子交易给其他节点。没有父交易的交易池被称作孤立交易池。一旦接收到了父交易，所有与这个父交易创建的UTXO有关的孤块会从池中释放出来，递归地重新验证，然后整条交易链就会被交易池包括进去，并等待着被区块所挖走。交易链可以是任意长度并且可以被任意数量的批次同时传走。在孤立池中保留孤块的机制保证了其他合法的交易不会只是因为父交易被耽误了而被抛弃，并且无论接收顺序，最终整个链会以正确的顺序重新构造出来。

内存中储存的孤立交易数量是有限制的，这是为了防止针对比特币节点的拒绝服务攻击（DoS）。这个限制被定义在比特币涉及到的客户端的源代码中的 `MAX_ORPHAN_TRANSACTIONS`。如果池中的孤立交易数量达到了 `MAX_ORPHAN_TRANSACTIONS`，一个或多个的、被随机选出的孤立交易会被池抛弃，直到池的大小回到限制以内。

5.6 比特币交易脚本和脚本语言

比特币客户端通过执行一个用类Forth脚本语言编写的脚本验证比特币交易。锁定脚本被写入UTXO，同时它往往包含一个用同种脚本语言编写的签名。当一笔比特币交易被验证时，每一个输入值中的解锁脚本被与其对应的锁定脚本同时（互不干扰地）执行，从而查看这笔交易是否满足使用条件。

如今，大多数经比特币网络处理的交易是以“Alice付给Bob”的形式存在的。同时，它们是以一种称为“P2PKH”（Pay-to-Public-Key-Hash）脚本为基础的。然而，通过使用脚本来锁定输出和解锁输入意味着通过使用编程语言，比特币交易可以包含无限数量的条件。当然，比特币交易并不限于“Alice付给Bob”的形式和模式。

这只是这个脚本语言可以表达的可能性的冰山一角。在这一节，我们将会全面展示比特币交易脚本语言的各个组成部分；同时，我们也会演示如何使用它去表达复杂的使用条件以及解锁脚本如何去满足这些花费条件。



比特币交易验证并不基于一个不变的模式，而是通过运行脚本语言来实现。这种语言可以表达出多到数不尽的条件变动。这也是比特币作为一种“可编程的货币”所拥有的权力。

5.6.1 脚本创建（锁定与解锁）

比特币的交易验证引擎依赖于两类脚本来验证比特币交易：一个锁定脚本和一个解锁脚本。

锁定脚本是一个放在一个输出值上的“障碍”，同时它明确了今后花费这笔输出的条件。由于锁定脚本往往含有一个公钥（即比特币地址），在历史上它曾被称作一个脚本公钥代码。由于认识到这种脚本技术存在着更为宽泛的可能性，在本书中，我

们将它称为一个“锁定脚本”。在大多数比特币应用源代码中，脚本公钥代码便是我们所说的锁定脚本。

解锁脚本是一个“解决”或满足被锁定脚本在一个输出上设定的花费条件的脚本，同时它将允许输出被消费。解锁脚本是每一笔比特币交易输出的一部分，而且往往含有一个被用户的比特币钱包（通过用户的私钥）生成的数字签名。由于解锁脚本常常包含一个数字签名，因此它曾被称作 `ScriptSig`。在大多数比特币应用的源代码中，`ScriptSig` 便是我们所说的解锁脚本。考虑到更宽泛的锁定脚本要求，在本书中，我们将它称为“解锁脚本”。但并非所有解锁脚本都一定会包含签名。

每一个比特币客户端会通过同时执行锁定和解锁脚本来验证一笔交易。对于比特币交易中的每一个输入，验证软件会先检索输入所指向的UTXO。这个UTXO包含一个定义了花费条件的锁定脚本。接下来，验证软件会读取试图花费这个UTXO的输入中所包含的解锁脚本，并执行这两个脚本。

在先前的比特币客户端中，解锁和锁定脚本是以连锁的形式存在的，并且是被依次执行的。出于安全因素考虑，在2010年比特币开发者们修改了这个特性——因为存在“允许异常解锁脚本推送数据入栈并且污染锁定脚本”的漏洞。在当今的比特币世界中，这两个脚本是随着堆栈的传递被分别执行的，后续将会详细介绍。

首先，使用堆栈执行引擎执行解锁脚本。如果解锁脚本在执行过程中未报错（没有悬空操作符），主堆栈（非其它堆栈）将被复制，然后脚本将被执行。如果采用从解锁脚本处复制而来的数据执行锁定脚本的结果为真，那么解锁脚本就成功地满足了锁定脚本所设置的条件，因而，该输入是一个能使用该UTXO的有效授权。如果在执行完组合脚本后的结果不是真，那么输入就不是有效的，因为它并未能满足UTXO中所设置的使用该笔资金的条件。注意，UTXO是永久性地记录在区块链中的，因此它不会因一笔新交易所发起的无效尝试而变化或受影响。只有一笔有效的能准确满足UTXO条件的交易才会导致UTXO被标记为“已使用”，然后从有效的（未使用）UTXO集中所移除。

图5-1是最为常见类型的比特币交易（向公钥哈希进行一笔支付）的解锁和锁定脚本样本，该样本展示了在脚本验证之前将解锁脚本和锁定脚本串联而成的组合脚本。

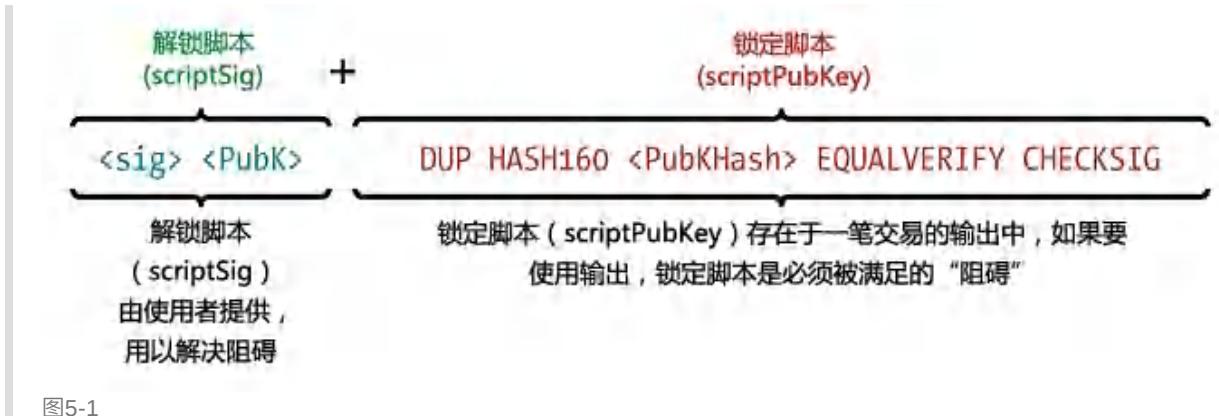


图5-1

5.6.2 脚本语言

比特币交易脚本语言，也成为脚本，是一种基于逆波兰表示法的基于堆栈的执行语言。如果这让您听起来似乎在胡言乱语，很有可能是您没学习过1960年的编程语言的缘故。脚本是一种非常简单的语言，这种语言被设计为能在有限的硬件上执行，这些硬件类似简单的嵌入式设备，如手持计算器。它仅需最少的处理即可，而且不能做许多现代编程语言可以做的事情。当涉及可编程的钱时，这是它的一个基于深思熟虑的安全特性。

比特币脚本语言被称为基于堆栈语言，因为它使用的数据结构被称为堆栈。堆栈是一个非常简单的数据结构，它可以被理解成为一堆卡片。一个堆栈允许两类操作：推送和弹出。推送是在堆栈顶部增加一个项目，弹出则是从堆栈顶部移除一个项目。

脚本语言通过从左至右地处理每个项目的方式执行脚本。数字（常数）被推送至堆栈，操作符向堆栈推送（或移除）一个或多个参数，对它们进行处理，甚至可能会向堆栈推送一个结果。例如，`OP_ADD`将从堆栈移除两个项目，将二者相加，然后再将二者相加之和推送到堆栈。

条件操作符评估一项条件，产生一个真或假的结果。例如，`OP_EQUAL`从堆栈移除两个项目，假如二者相等则推送真（表示为1），假如二者不等则推送为假（表示为0）。比特币交易脚本常含条件操作符，当一笔交易有效时，就会产生真的结果。

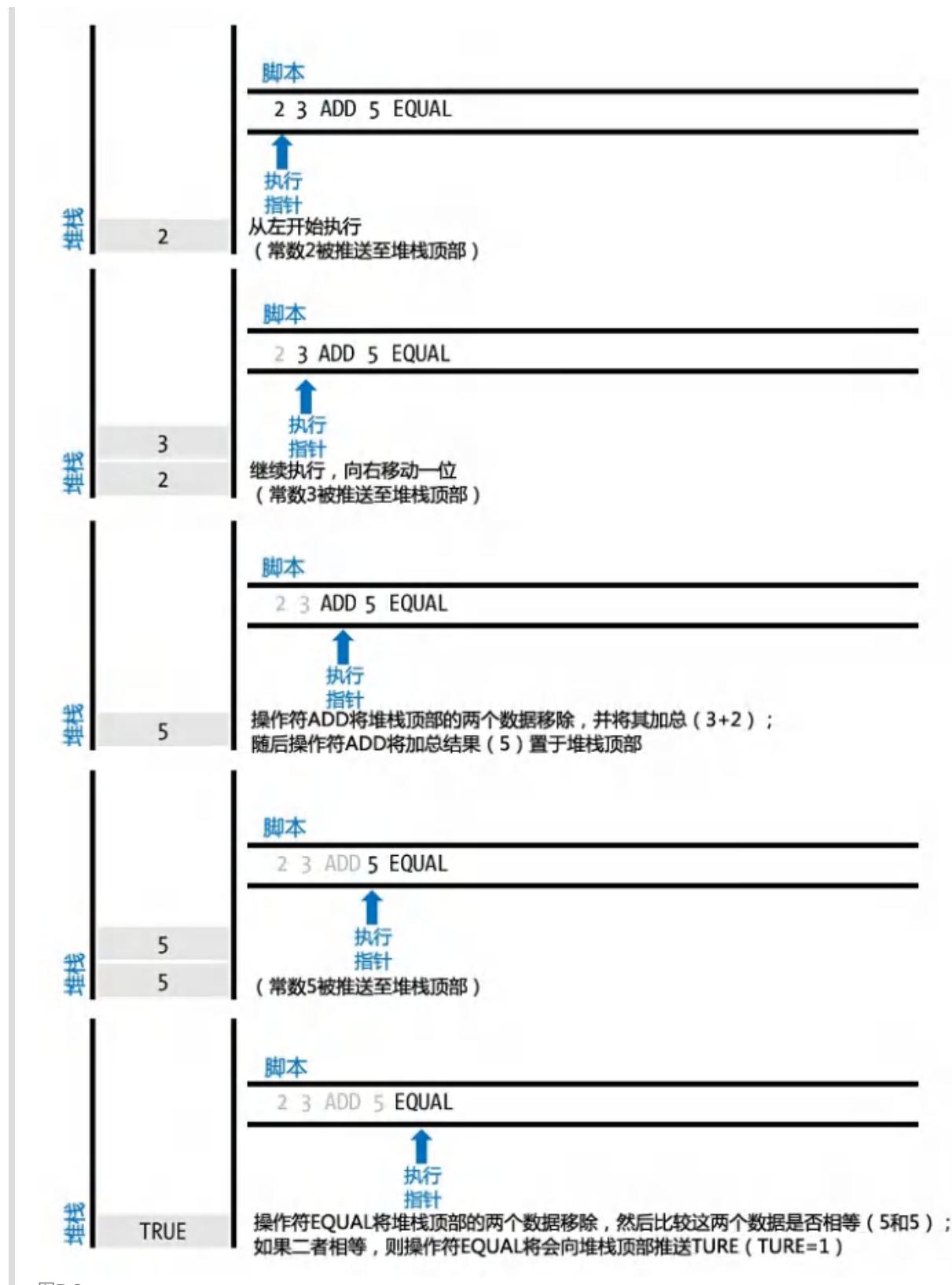


图5-2

在图5-2中，脚本“2 3 OP_ADD 5 OP_EQUAL”演示了算术加法操作符OP_ADD，该操作符将两个数字相加，然后把结果推送到堆栈，OP_EQUAL是验算之前的两数之和是否等于5。为了简化起见，前缀OP_在一步步的演示示例过程中将被省略。

以下是一个稍微有些复杂的脚本，它用于计算 $2+7-3+1$ 。注意，当脚本包含多个操作符时，堆栈允许一个操作符的结果作用于下一个操作符。

2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL

请试着用纸笔自行演算脚本，当脚本执行完毕时，你会在堆栈得到正确的结果。

虽然大多数的解锁脚本都指向一个比特币地址或公钥，因而如果想要使用资金则需验证所有权，但脚本本身并不需要如此的复杂。任何解锁和锁定脚本的组合如果结果为真，则为有效。前面被我们用于说明脚本语言的简单算术运算同样也是一个有效的锁定脚本，该脚本能用于锁定交易输出。

使用部分算数运算示例脚本作用锁定脚本：

```
3 OP_ADD 5 OP_EQUAL
```

该脚本能被以解锁脚本为输入的一笔交易所满足，解锁脚本为：

2

验证软件将锁定和解锁脚本组合起来：

```
2 3 OP_ADD 5 OP_EQUAL
```

正如在图5-2中所看到的，当脚本被执行时，结果是OP_TRUE，从而使得交易有效。不仅该笔交易的输出锁定脚本有效，同时UTXO也能被任何知晓这个运算技巧（知道是数字2）的人所使用。



如果堆栈顶部的结果显示为真（标记为{0x01}），即为任何非零值或脚本执行后堆栈为空情形，则交易有效。如果堆栈顶部的结果显示为假（0字节空值，标记为{}）或脚本执行被操作符禁止，如OP_VERIFY、OP_RETURN，或有条件终止如OP_ENDIF，则交易无效。详见附录1。

5.6.3 图灵非完备性

比特币脚本语言包含许多操作，但都故意限定为一种重要的方式——没有循环或者复杂流控制功能以外的其他条件的流控制。这样就保证了脚本语言的图灵非完备性，这意味着脚本的复杂性有限，交易可执行的次数也可预见。脚本并不是一种通用语言，施加的这些限制确保该语言不被用于创造无限循环或其它类型的逻辑炸弹，这样的炸弹可以植入在一笔交易中，通过引起拒绝服务的方式攻击比特币网络。受限制的语言能防止交易激活机制被人当作薄弱环节而加以利用。

5.6.4 非主权验证

比特币交易脚本语言是无国家主权的，没有国家能凌驾于脚本之上，也没有国家会在脚本被执行后对其进行保存。所以需要执行脚本的所有信息都已包含在脚本中。可以预见的是，一个脚本能在任何系统上以相同的方式执行。如果您的系统对一个脚本进行验证，可以确信的是每一个比特币网络中的其他系统也将对其进行验证，这意味着一个有效的交易对每个人而言都是有效的，而且每一个人都明白这一点。这种对于结果的可预见性是比特币系统的一项重要良性特征。

5.7 标准交易

在比特币最初几年的发展过程中，开发者对可以经由客户端进行操作的脚本类型设置了一些限制。这些限制被编译为一个Standard()函数，该函数定义了五种类型的标准交易。这些限制都是临时性的，当您阅读本书时或许已经更新。截至目前，五种标准交易脚本是仅有的被客户端和大多数运行客户端的矿工们所接受的脚本。虽然创设一个非标准交易（脚本类型非标准化）是有可能的，但前提是必须能找到一个不遵循标准而且能将该非标准交易纳入区块的矿工。

通过检索比特币核心客户端源代码，可以看到当前有哪些交易脚本是被认可的。

五大标准脚本分别为P2PKH、P2PK、MS（限15个密钥）、P2SH和OP_Return，后文将详细介绍这五大脚本。

5.7.1 P2PKH（Pay-to-Public-Key-Hash）

比特币网络上的大多数交易都是P2PKH交易，此类交易都含有一个锁定脚本，该脚本由公钥哈希实现阻止输出功能，公钥哈希即为广为人知的比特币地址。由P2PKH脚本锁定的输出可以通过键入公钥和由相应私钥创设的数字签名得以解锁。

例如，我们可以再次回顾一下Alice向Bob咖啡馆支付的案例。Alice下达了向Bob咖啡馆的比特币地址支付0.015比特币的支付指令，该笔交易的输出内容为以下形式的锁定脚本：

```
OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUAL OP_CHECKSIG
```

脚本中的 `Cafe Public Key Hash` 即为咖啡馆的比特币地址，但这个地址不是基于Base58Check编码的。事实上，大多数比特币地址都显示为十六进制码，而不是大家所熟知的以1开头的基于Base58Check编码的比特币地址。

锁定脚本的解锁版脚本是：

```
<Cafe Signature> <Cafe Public Key>
```

将两个脚本结合起来可以形成如下有效组合脚本：

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160  
<Cafe Public Key Hash> OP_EQUAL OP_CHECKSIG
```

只有当解锁版脚本与锁定版脚本的设定条件相匹配时，执行组合有效脚本时才会显示结果为真（True）。即只有当解锁脚本得到了咖啡馆的有效签名，交易执行结果才会被通过（结果为真），该有效签名是从与公钥哈希相匹配的咖啡馆的私钥中所获取的。

图5-3和图5-4（分两部分）显示了组合脚本一步步检验交易有效性的过程。



图5-3



图5-4

5.7.2 P2PK (Pay-to-Public-Key)

与P2PKH相比，P2PK模式更为简单。与P2PKH模式含有公钥哈希的模式不同，在P2PK脚本模式中，公钥本身已经存储在锁定脚本中，而且代码长度也更短。P2PKH是由Satoshi创建的，主要目的一方面为使比特币地址更简短，另一方面也使之更方便使用。P2PK目前在Coinbase交易中最常见，Coinbase交易由老的采矿软件产生，目前还没更新至P2PKH。

P2PK锁定版脚本形式如下：

```
<Public Key A> OP_CHECKSIG
```

用于解锁的脚本是一个简单签名：

```
<Signature from Private Key A>
```

经由交易验证软件确认的组合脚本为：

```
<Signature from Private Key A> <Public Key A> OP_CHECKSIG
```

该脚本只是CHECKSIG操作符的简单调用，该操作主要是为了验证签名是否正确，如果正确，则返回为真（True）。

5.7.3 多重签名

多重签名脚本设置了这样一个条件，假如记录在脚本中的公钥个数为N，则至少需提供其中的M个公钥才可以解锁。这也被称为M-N组合，其中，N是记录在脚本中的公钥总个数，M是使得多重签名生效的公钥数阈值（最少数目）。例如，对于一个2-3多重签名组合而言，存档公钥数为3个，至少同时使用其中2个或者2个以上的公钥时，才能生成激活交易的签名，通过验证后才可使用这笔资金。最初，标准多重签名脚本的最大存档公钥数被限定为15个，这意味着可采用1-1乃至15-15的任意多重签名组合，或者组合的组合来激活交易。15个存档公钥数的限制也许在本书出版时已有所增加，读者通过检索Standard（）函数可以获得最新存档公钥数上限值的相关信息。

通用的M-N多重签名锁定脚本形式为：

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG
```

其中，N是存档公钥总数，M是要求激活交易的最少公钥数。

2-3多重签名条件：

```
2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG
```

上述锁定脚本可由含有签名和公钥的脚本予以解锁：

```
OP_0 <Signature B> <Signature C>
```

或者由3个存档公钥中的任意2个相一致的私钥签名组合予以解锁。



之所以要加上前缀OP_0，是因为最早的CHECKMULTISIG在处理含有多个项目的过程中有个小漏洞，CHECKMULTISIG会自动忽略这个前缀，它只是占位符而已。

两个脚本组合将形成一个验证脚本：

```
OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG
```

当执行时，只有当未解锁版脚本与解锁脚本设置条件相匹配时，组合脚本才显示得到结果为真（True）。上述例子中相应的设置条件即为未解锁脚本是否含有与3个公钥中的任意2个相一致的私钥的有效签名。

5.7.4 数据输出（OP_RETURN操作符）

比特币的分发和时间戳账户机制（也即区块链），其潜在运用将大大超越支付领域。许多开发者试图充分发挥交易脚本语言的安全性和可恢复性优势，将其运用于电子公证服务、证券认证和智能协议等领域。比特币脚本语言的早期运用主要包括在区块链上创造出交易输出。例如，为文件记录电子指纹，则任何人都可以通过该机制在特定的日期建立关于文档存在性的证明。

运用比特币区块链存储与比特币支付不相关数据的做法是一个有争议的话题。许多开发者认为其有滥用的嫌疑，因而试图予以阻止。另一些开发者则将之视为区块链技术强大功能的有力证明，从而试图给予大力支持。那些反对非支付相关应用的开发者认为这样做将引致“区块链膨胀”，因为所有的区块链节点都将以消耗磁盘存储空间为成本，负担存储此类数据的任务。更为严重的是，此类交易仅将比特币地址当作自由组合的20个字节而使用，进而会产生不能用于交易的UTXO。因为比特币地址只是被当作数据使用，并不与私钥相匹配，所以会导致UTXO不能被用于交易，因而是一种伪支付行为。这样的做法将使得内存中的UTXO不断增加，而且这些不能被用于交易的数据同样也不能被移除，因此比特币节点将永久性地担负这些数据，这无疑是代价高昂的。

在0.9版的比特币核心客户端上，通过采用 `OP_RETURN` 操作符最终实现了妥协。`OP_RETURN` 允许开发者在交易输出上增加40字节的非交易数据。然后，与伪交易型的UTXO不同，`OP_RETURN` 创造了一种明确的可复查的非交易型输出，此类数据无需存储于UTXO集。`OP_RETURN` 输出被记录在区块链上，它们会消耗磁盘空间，也会导致区块链规模的增加，但它们不存储在UTXO集中，因此也不会使得UTXO内存膨胀，更不会以消耗代价高昂的内存为代价使全节点都不堪重负。

`OP_RETURN` 脚本的样式：

```
OP_RETURN <data>
```

“data”部分被限制为40字节，且多以哈希方式呈现，如32字节的SHA256算法输出。许多应用都在其前面加上前缀以辅助认定。例如，电子公正服务的证明材料采用8个字节的前缀“DOCPROOF”，在十六进制算法中，相应的ASCII码为 `44f4350524f4f46`。

请记住 `OP_RETURN` 不涉及可用于支付的解锁脚本的特点，`OP_RETURN` 不能使用其输出中所锁定的资金，因此它也就没有必要记录在蕴含潜在成本的UTXO集中，所以 `OP_RETURN` 实际是没有成本的。`OP_RETURN` 常为一个金额为0的比特币输出，因为任何与该输出相对应的比特币都会永久消失。假如一笔 `OP_RETURN` 遇到脚本验证软件，它将立即导致验证脚本和标记交易的行为无效。如果你碰巧将 `OP_RETURN` 的输出作为另一笔交易的输入，则该交易是无效的。

一笔标准交易（通过了 `isStandard()` 函数检验的）只能有一个 `OP_RETURN` 输出。但是单个 `OP_RETURN` 输出能与任意类型的输出交易进行组合。

5.7.5 P2SH (Pay-to-Script-Hash)

P2SH在2012年被作为一种新型、强大、且能大大简化复杂交易脚本的交易类型而引入。为进一步解释P2SH的必要性，让我们先看一个实际的例子。

在第1章中，我们曾介绍过Mohammed，一个迪拜的电子产品进口商。Mohammed的公司采用比特币多重签名作为其公司会计账簿记账要求。多重签名脚本是比特币高级脚本最为常见的运用之一，是一种具有相当大影响力的脚本。针对所有的顾客支付（即应收账款），Mohammed的公司要求采用多重签名交易。基于多重签名机制，顾客的任何支付都需要至少两个签名才能解锁，一个来自Mohammed，另一个来自其合伙人或拥有备份钥匙的代理人。这样的多重签名机制能为公司治理提供管控便利，同时也能有效防范盗窃、挪用和遗失。

最终的脚本非常长：

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5  
OP_CHECKMULTISIG
```

虽然多重签名十分强大，但其使用起来还是多有不便。基于之前的脚本，Mohammed必须在客户付款前将该脚本发送给每一位客户，而每一位顾客也必须使用特制的能产生客户交易脚本的比特币钱包软件，每位顾客还得学会如何利用脚本来完成交易。此外，由于脚本可能包含特别长的公钥，最终的交易脚本可能是最初交易脚本长度的5倍之多。额外长度的脚本将给客户造成费用负担。最后，一个长的交易脚本将一直记录在所有节点的随机存储器的UTXO集中，直到该笔资金被使用。所有这些都使得在实际交易中采用复杂输出脚本显得困难重重。

P2SH正是为了解决这一实际难题而被引入的，它旨在使复杂脚本的运用能与直接向比特币地址支付一样简单。在P2SH支付中，复杂的锁定脚本被电子指纹所取代，电子指纹为密码学哈希。当一笔交易试图支付UTXO时，要解锁支付脚本，它必须含有与哈希相匹配的脚本。P2SH的含义是，向与该哈希匹配的脚本支付，当输出被支付时，该脚本将在后续呈现。

在P2SH交易中，锁定脚本由哈希取代，哈希指代的是赎回脚本。因为它在系统中是在赎回时出现而不是以锁定脚本模式出现。表5-4列示了非P2SH脚本，表5-5列示了P2SH脚本。

表5-4 不含P2SH的复杂脚本

Locking Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Unlocking Script	Sig1 Sig2

表5-5 P2SH复杂脚本

Redeem	Script 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Locking Script	OP_HASH160 <20-byte hash of redeem script> OP_EQUAL
Unlocking Script	Sig1 Sig2 redeem script

正如你在表中所看到的，在P2SH中，出现了花费该笔支出（赎回脚本）条件的复杂脚本，而这在锁定脚本中并未出现。取而代之，在锁定脚本中，只出现了哈希，而赎回脚本则在稍后输出被支付时才作为解锁脚本的一部分而出现。

让我们再看下Mohammed公司的例子，复杂的多重签名脚本和相应的P2SH脚本。

首先，Mohammed公司对所有顾客订单采用多重签名脚本：

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5  
OP_CHECKMULTISIG
```

如果占位符由实际的公钥（以04开头的520字节）替代，你将会看到的脚本会非常地长：

```
2  
04C16B8698A9ABF84250A7C3EA7EE-  
DEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C58704A2192968D8655D6A935BEAF2C  
DA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5043752580AFA1EC-  
ED3C68D446BCAB69AC0A7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800 5 OP_CHECKMUL
```



整个脚本都可由仅为20个字节的密码哈希所取代，首先采用SH256哈希算法，随后对其运用RIPEMD160算法。20字节的脚本为：

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97
```

一笔P2SH交易运用锁定脚本将输出与哈希关联，而不是与前面特别长的脚本所关联。使用的锁定脚本为：

```
OP_HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e OP_EQUAL
```

正如你所看到的，这个脚本比前面的长脚本简短多了。取代“向该5个多重签名脚本支付”，这个P2SH等同于“向含该哈希的脚本支付”。顾客在向Mohammed公司支付时，只需在其支付指令中纳入这个非常简短的锁定脚本即可。当Mohammed想要花费这笔UTXO时，附上原始赎回脚本（与UTXO锁定的哈希）和必要的解锁签名即可，如：

```
<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG>
```

两个脚本经由两步实现组合。首先，将赎回脚本与锁定脚本比对以确认其与哈希是否匹配：

```
<2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG> OP_HASH160 <redeem scriptHash> OP_EQUAL
```

假如赎回脚本与哈希匹配，解锁脚本会被执行以释放赎回脚本：

```
<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG
```

5.7.5.1 P2SH地址

P2SH的另一重要特征是它能将脚本哈希编译为一个地址（其定义请见BIP0013）。P2SH地址是基于Base58编码的一个含有20个字节哈希的脚本，就像比特币地址是基于Base58编码的一个含有20个字节的公钥。由于P2SH地址采用5作为前缀，这导致基于Base58编码的地址以“3”开头。例如，Mohammed的脚本，基于Base58编码下的P2SH地址变“39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw”。此时，Mohammed可以将该地址发送给他的客户，这些客户可以采用任何的比特币钱包实现简单支付，就像这是一个比特币地址一样。以“3”为前缀给予客户这是一种特殊类型的地址的暗示，该地址与一个脚本相对应而非与一个公钥相对应，但是它的效果与比特币地址支付别无二致。

P2SH地址隐藏了所有的复杂性，因此，运用其进行支付的人将不会看到脚本。

5.7.5.2 P2SH的优点

与直接使用复杂脚本以锁定输出的方式相比，P2SH具有以下特点：

- ▷ 在交易输出中，复杂脚本由简短电子指纹取代，使得交易代码变短。
- ▷ 脚本能被编译为地址，支付指令的发出者和支付者的比特币钱包不需要复杂工序就可以执行P2SH。
- ▷ P2SH将构建脚本的重担转移至接收方，而非发送方。
- ▷ P2SH将长脚本数据存储的负担从输出方（存储于UTXO集，影响内存）转移至输入方（仅存储于区块链）。
- ▷ P2SH将长脚本数据存储的重担从当前（支付时）转移至未来（花费时）。
- ▷ P2SH将长脚本的交易费成本从发送方转移至接收方，接收方在使用该笔资金时必须含有赎回脚本。

5.7.5.3 赎回脚本和标准确认

在0.9.2版比特币核心客户端之前，P2SH仅限于标准比特币交易脚本类型（即通过标准函数检验的脚本）。这也意味着使用该笔资金的交易中的赎回脚本只能是标准化的P2PK、P2PKH或者多重签名，而非 `OP_RETURN` 和P2SH。

作为0.9.2版的比特币核心客户端，P2SH交易能包含任意有效的脚本，这使得P2SH标准更为灵活，也可以用于多种新的或复杂的交易。

请记住不能将P2SH植入P2SH赎回脚本，因为P2SH不能自循环。也不能在赎回脚本中使用 `OP_RETURN`，因为 `OP_RETURN` 的定义即显示不能赎回。

需要注意的是，因为赎回脚本只有在你试图发送一个P2SH输出时才会在比特币网络中出现，假如你将输出与一个无效的交易哈希锁定，则它将会被忽略。你将不能使用该笔资金，因为交易中含有赎回脚本，该脚本因是一个无效的脚本而不能被接受。这样的处理机制也衍生出一个风险，你能将比特币锁定在一个未来不能被花费的P2SH中。因为比特币网络本身会接受这一P2SH，即便它与无效的赎回脚本所对应（因为该赎回脚本哈希没有对其所表征的脚本给出指令）。

P2SH锁定脚本包含一个赎回脚本哈希，该脚本对于赎回脚本本身未提供任何描述。P2SH交易即便在赎回脚本无效的情况下也会被认为有效。你可能会偶然地将比特币以这样一种未来不能被花费的方式予以锁定。

第6章 比特币网络

6.1 P2P网络架构

比特币采用了基于国际互联网（Internet）的P2P（peer-to-peer）网络架构。P2P是指位于同一网络中的每台计算机都彼此对等，各个节点共同提供网络服务，不存在任何“特殊”节点。每个网络节点以“扁平（flat）”的拓扑结构相互连通。在P2P网络中不存在任何服务端（server）、中央化的服务、以及层级结构。P2P网络的节点之间交互运作、协同处理：每个节点在对外提供服务的同时也使用网络中其他节点所提供的服务。P2P网络也因此具有可靠性、去中心化，以及开放性。早期的国际互联网就是P2P网络架构的一个典型用例：IP网络中的各个节点完全平等。当今的互联网架构具有分层架构，但是IP协议仍然保留了扁平拓扑的结构。在比特币之外，规模最大也最成功的P2P技术应用是在文件分享领域：Napster是该领域的先锋，BitTorrent是其架构的最新演变。

比特币所采用的P2P网络架构不仅仅是选择拓扑结构这样简单。比特币被设计为一种点对点的数字现金系统，它的网络架构既是这种核心特性的反映，也是该特性的基石。去中心化控制是设计时的核心原则，它只能通过维持一种扁平化、去中心化的P2P共识网络来实现。

“比特币网络”是按照比特币P2P协议运行的一系列节点的集合。除了比特币P2P协议之外，比特币网络中也包含其他协议。例如Stratum协议就被应用于挖矿、以及轻量级或移动端比特币钱包之中。网关（gateway）路由服务器提供这些协议，使用比特币P2P协议接入比特币网络，并把网络拓展到运行其他协议的各个节点。例如，Stratum服务器通过Stratum协议将所有的Stratum挖矿节点连接至比特币主网络、并将Stratum协议桥接（bridge）至比特币P2P协议之上。我们使用“扩展比特币网络（extended bitcoin network）”指代所有包含比特币P2P协议、矿池挖矿协议、Stratum协议以及其他连接比特币系统组件相关协议的整体网络结构。

6.2 节点类型及分工

尽管比特币P2P网络中的各个节点相互对等，但是根据所提供的功能不同，各节点可能具有不同的分工。每个比特币节点都是路由、区块链数据库、挖矿、钱包服务的功能集合。一个全节点（full node）包括如图6-1所示的四个功能：

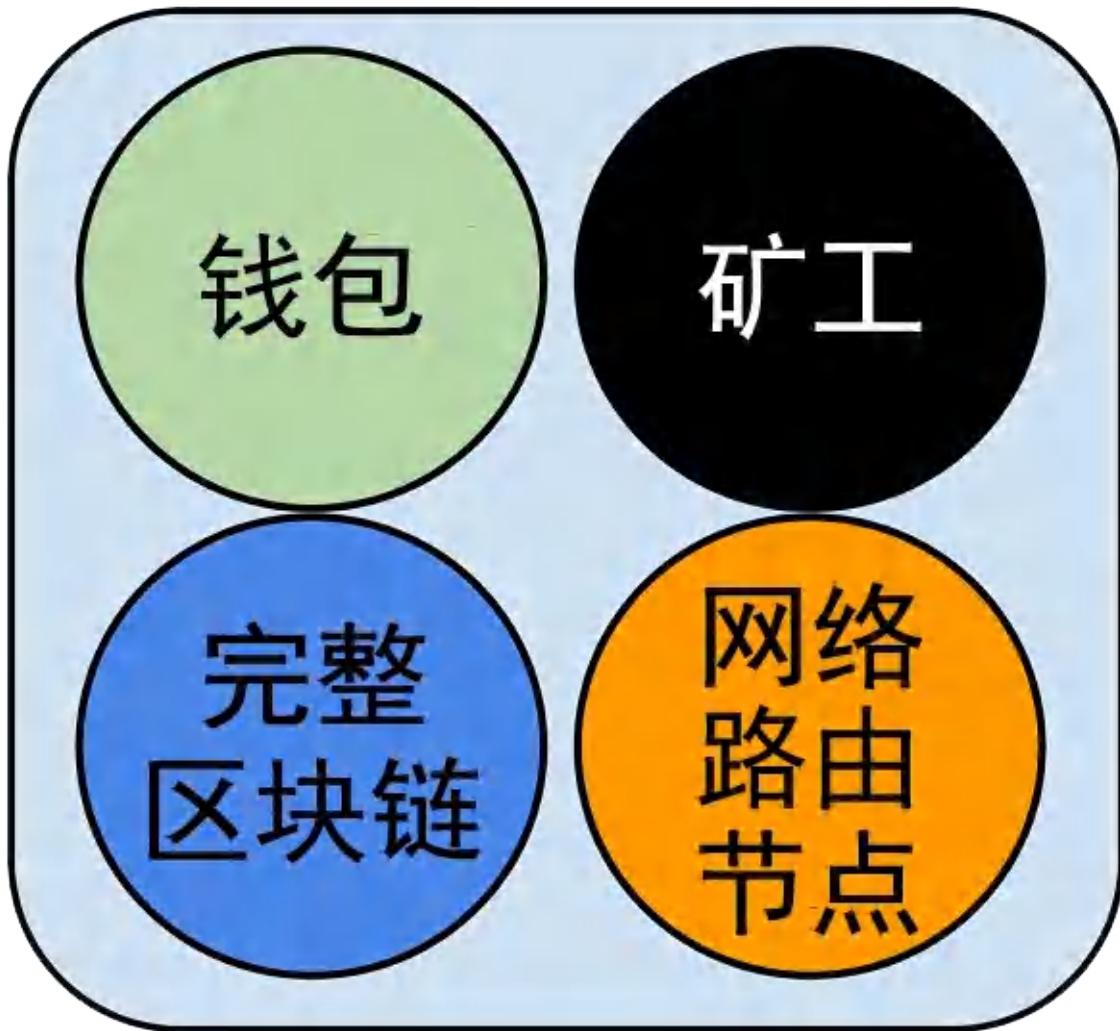


图6-1 一个包含四个完整功能的比特币网络节点：钱包、矿工、完整区块链、网络路由节点

每个节点都参与全网络的路由功能，同时也可能包含其他功能。每个节点都参与验证并传播交易及区块信息，发现并维持与对等节点的连接。在图6-1所示的全节点用例中，名为“网络路由节点”的橙色圆圈即表示该路由功能。

一些节点保有一份完整的、最新的区块链拷贝，这样的节点被称为“全节点”。全节点能够独立自主地校验所有交易，而不需要借由任何外部参照。另外还有一些节点只保留了区块链的一部分，它们通过一种名为“简易支付验证（SPV）”的方式来完成交易验证。这样的节点被称为“SPV节点”，又叫“轻量级节点”。在如上图所示的全节点用例中，名为完整区块链的蓝色圆圈即表示了全节点区块链数据库功能。在图6-3中，SPV节点没有此蓝色圆圈，以示它们没有区块链的完整拷贝。

挖矿节点通过运行在特殊硬件设备上的工作量证明（proof-of-work）算法，以相互竞争的方式创建新的区块。一些挖矿节点同时也是全节点，保有区块链的完整拷贝；还有一些参与矿池挖矿的节点是轻量级节点，它们必须依赖矿池服务器维护的全节点进行工作。在全节点用例中，挖矿功能如图中名为“矿工”的黑色圆圈所示。

用户钱包也可以作为全节点的一部分，这在桌面比特币客户端中比较常见。当前，越来越多的用户钱包都是SPV节点，尤其是运行于诸如智能手机等资源受限设备上的比特币钱包应用；而这正变得越来越普遍。在图6-1中，名为“钱包”的绿色圆圈代表钱包功能。

在比特币P2P协议中，除了这些主要的节点类型之外，还有一些服务器及节点也在运行着其他协议，例如特殊矿池挖矿协议、轻量级客户端访问协议等。

图6-2描述了扩展比特币网络中最为常见的节点类型。



核心客户端 (Bitcoin Core)

在比特币P2P网络中，包含钱包、矿工、完整区块链数据库、网络路由节点。



完整区块链节点

在比特币P2P网络中，包含完整区块链以及网络路由节点。



独立矿工

包含具有完整区块链副本的挖矿功能、以及比特币P2P网络路由节点。



轻量(SPV)钱包

包含不具有区块链的钱包以及比特币P2P网络节点。



矿池协议服务器

将运行其他协议的节点(例如矿池挖矿节点、Stratum节点),连接至P2P网络的网关路由器。



挖矿节点

包含不具有区块链、但具备Stratum协议节点 (S) 或其他矿池挖矿协议节点 (P) 的挖矿功能。



轻量(SPV) Stratum 钱包

包含不具有区块链的钱包、运行 Stratum 协议的网络节点。

图6-2 扩展比特币网络的不同节点类型

6.3 扩展比特币网络

运行比特币P2P协议的比特币主网络由大约7000-10000个运行着不同版本比特币核心客户端（Bitcoin Core）的监听节点、以及几百个运行着各类比特币P2P协议的应用（例如BitcoinJ、Libbitcoin、btcld等）的节点组成。比特币P2P网络中的一小部分节点也是挖矿节点，它们竞争挖矿、验证交易、并创建新的区块。许多连接到比特币网络的大型公司运行着基于Bitcoin核心客户端的全节点客户端，它们具有区块链的完整拷贝及网络节点，但不具备挖矿及钱包功能。这些节点是网络中的边缘路由器（edge routers），通过它们可以搭建其他服务，例如交易所、钱包、区块浏览器、商家支付处理（merchant payment processing）等。

如前文所述，扩展比特币网络既包括了运行比特币P2P协议的网络，又包含运行特殊协议的网络节点。比特币P2P主网络上连接着许多矿池服务器以及协议网关，它们把运行其他协议的节点连接起来。这些节点通常都是矿池挖矿节点（参见第8章）以及轻量级钱包客户端，它们通常不具备区块链的完整备份。

图6-3描述了扩展比特币网络，它包括了多种类型的节点、网关服务器、边缘路由器、钱包客户端以及它们相互连接所需的各类协议。

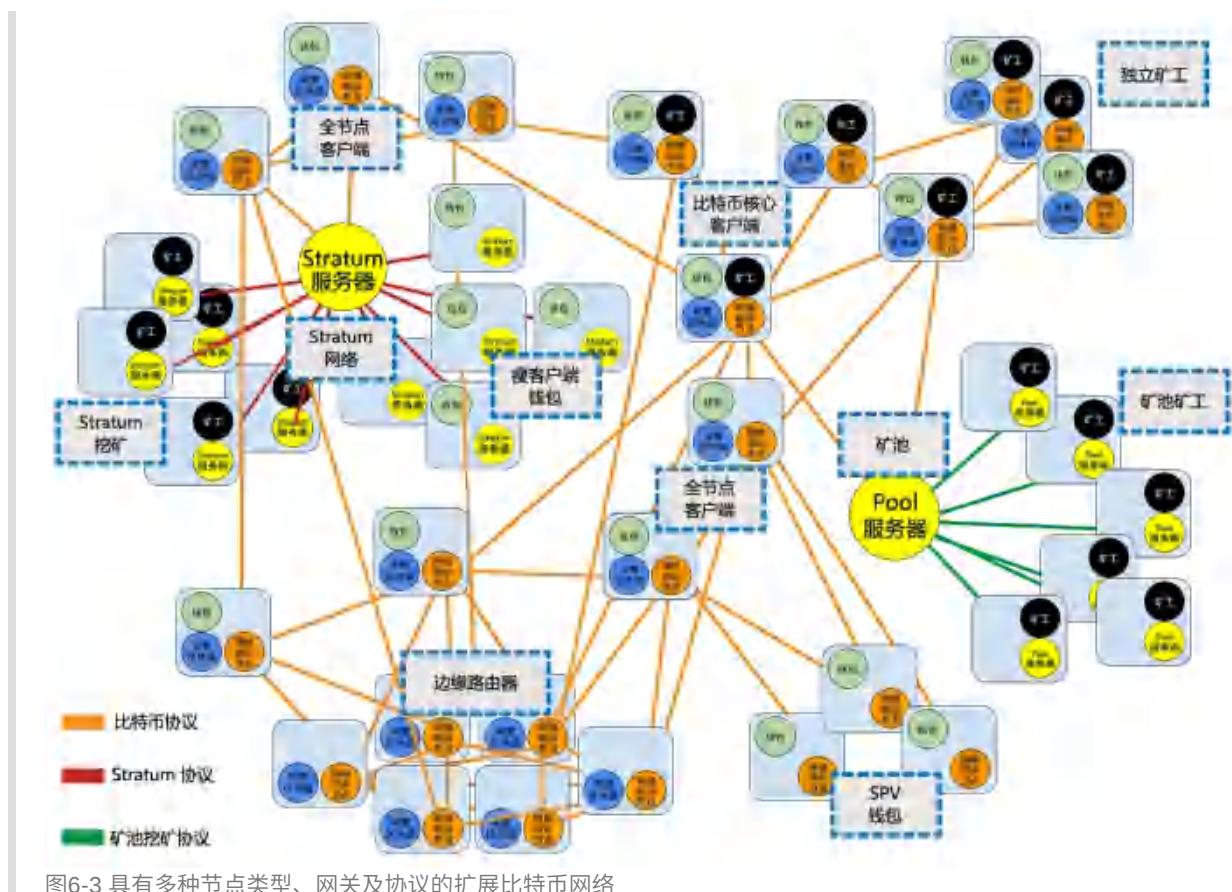


图6-3 具有多种节点类型、网关及协议的扩展比特币网络

6.4 网络发现

当新的网络节点启动后，为了能够参与协同运作，它必须发现网络中的其他比特币节点。新的网络节点必须发现至少一个网络中存在的节点并建立连接。由于比特币网络的拓扑结构并不基于节点间的地理位置，因此各个节点之间的地理信息完全无关。在新节点连接时，可以随机选择网络中存在的比特币节点与之相连。

节点通常采用TCP协议、使用8333端口（该端口号通常是比特币所使用的，除8333端口外也可以指定使用其他端口）与已知的对等节点建立连接。在建立连接时，该节点会通过发送一条包含基本认证内容的version消息开始“握手”通信过程（见图6-4）。这一过程包括如下内容：

▷ PROTOCOL_VERSION

常量，定义了客户端所“说出”的比特币P2P协议所采用的版本（例如：70002）。

▷ **nLocalServices**

一组该节点支持的本地服务列表，当前仅支持NODE_NETWORK

▷ **nTime**

当前时间

▷ **addrYou**

当前节点可见的远程节点的IP地址

▷ **addrMe**

本地节点所发现的本机IP地址

▷ **subver**

指示当前节点运行的软件类型的子版本号（例如：“/Satoshi:0.9.2.1/”）

▷ **BaseHeight**

当前节点区块链的区块高度

(version网络消息的具体用例请参见[GitHub](#))

网络中的对等节点通过对verack消息的响应进行确认并建立连接；有时候，如果接收节点需要互换连接并连回起始节点，也会传回该对等节点的version消息。

新节点是如何发现网络中的对等节点的呢？虽然比特币网络中没有特殊节点，但是客户端会维持一个列表，那里列出了那些长期稳定运行的节点。这样的节点被称为“种子节点（seed nodes）”。新节点并不一定需要与种子节点建立连接，但连接到种子节点的好处是可以通过种子节点来快速发现网络中的其他节点。在比特币核心客户端中，是否使用种子节点是通过“-dnsseed”控制的。默认情况下，该选项设为1，即意味着使用种子节点。另一种方式是，起始时将至少一个比特币节点的IP地址提供给正在启动的节点（该节点不包含任何比特币网络的组成信息）。在这之后，启动节点可以通过后续指令建立新的连接。用户可以使用命令行参数“-seednode”把启动节点“引荐”并连接到一个节点，并将该节点用作DNS种子。在初始种子节点被用于形成“引荐”信息之后，客户端会断开与它的连接、并与新发现的对等节点进行通信。

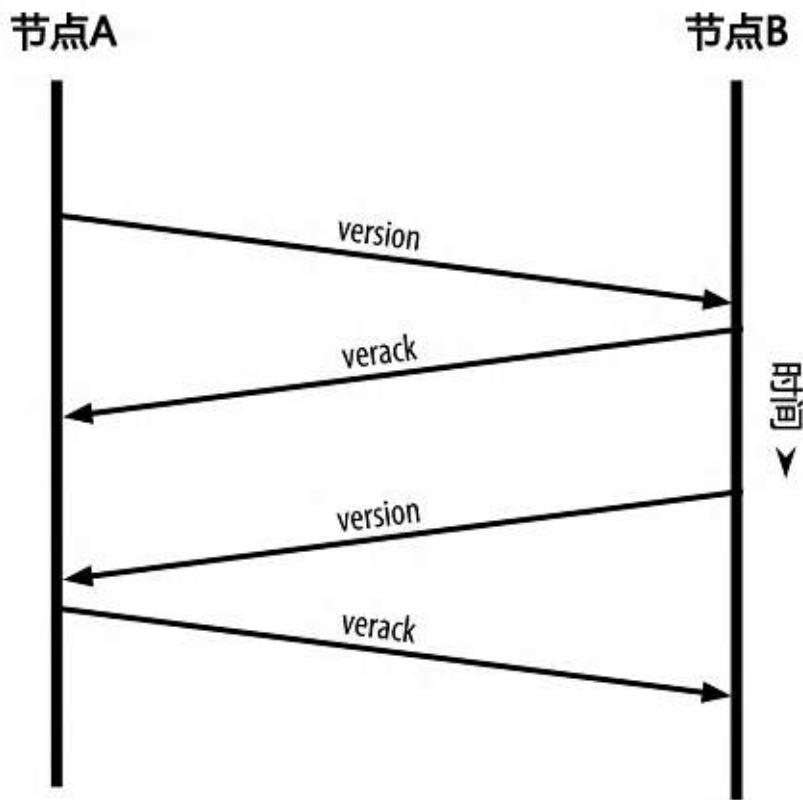


图6-4 对等节点之间的初始“握手”通信

当建立一个或多个连接后，新节点将一条包含自身IP地址的addr消息发送给其相邻节点。相邻节点再将此条addr消息依次转发给它们各自的相邻节点，从而保证新节点信息被多个节点所接收、保证连接更稳定。另外，新接入的节点可以向它的相邻节点发送getaddr消息，要求它们返回其已知对等节点的IP地址列表。通过这种方式，节点可以找到需连接到的对等节点，并向网络发布它的消息以便其他节点查找。图6-5描述了这种地址发现协议。

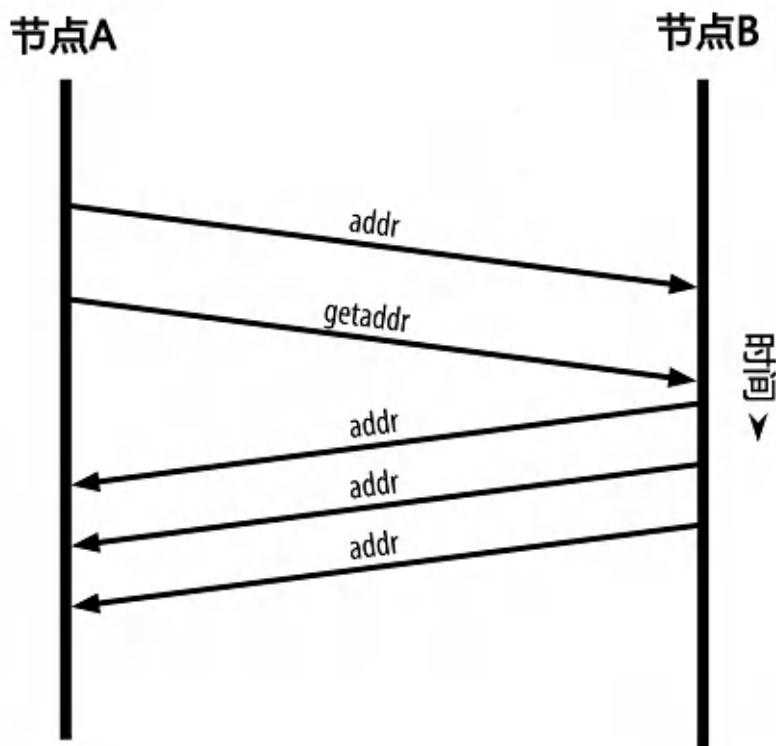


图6-5 地址广播及发现

节点必须连接到若干不同的对等节点才能在比特币网络中建立通向比特币网络的种类各异的路径（path）。由于节点可以随时加入和离开，通讯路径是不可靠的。因此，节点必须持续进行两项工作：在失去已有连接时发现新节点，并在其他节点启动时为其提供帮助。节点启动时只需要一个连接，因为第一个节点可以将它引荐给它的对等节点，而这些节点又会进一步提供引荐。一个节点，如果连接到大量的其他对等节点，这既没必要，也是对网络资源的浪费。在启动完成后，节点会记住它最近成功连接的对等节点；因此，当重新启动后它可以迅速与先前的对等节点重新建立连接。如果先前的网络的对等节点对连接请求无应答，该节点可以使用种子节点进行重启动。

在运行比特币核心客户端的节点上，您可以使用 `getpeerinfo` 命令列出对等节点连接信息：

```
$ bitcoin-cli getpeerinfo
[
  {
    "addr" : "85.213.199.39:8333",
    "services" : "00000001",
    "lastsend" : 1405634126,
    "lastrecv" : 1405634127,
    "bytesent" : 23487651,
    "bytesrecv" : 138679099,
    "conntime" : 1405021768,
    "pingtime" : 0.00000000,
    "version" : 70002,
    "subver" : "/Satoshi:0.9.2.1/",
    "inbound" : false,
    "startingheight" : 310131,
    "banscore" : 0,
    "syncnode" : true
  },
  {
    "addr" : "58.23.244.20:8333",
    "services" : "00000001",
    "lastsend" : 1405634127,
    "lastrecv" : 1405634124,
    "bytesent" : 4460918,
    "bytesrecv" : 8903575,
    "conntime" : 1405559628,
    "pingtime" : 0.00000000,
    "version" : 70001,
    "subver" : "/Satoshi:0.8.6/",
    "inbound" : false,
    "startingheight" : 311074,
    "banscore" : 0,
    "syncnode" : false
  }
]
```

用户可以通过提供 `-connect=<IP地址>` 选项来指定一个或多个IP地址，从而达到复写自动节点管理功能并指定IP地址列表的目的。如果采用此选项，节点只连接到这些选定的节点IP地址，而不会自动发现并维护对等节点之间的连接。

如果已建立的连接没有数据通信，所在的节点会定期发送信息以维持连接。如果节点持续某个连接长达90分钟没有任何通信，它会被认为已经从网络中断开，网络将开始查找一个新的对等节点。因此，比特币网络会随时根据变化的节点及网络问题进行动态调整，不需经过中心化的控制即可进行规模增、减的有机调整。

6.5 全节点

全节点是指维持包含全部交易信息的完整区块链的节点。更加准确地说，这样的节点应当被称为“完整区块链节点”。在比特币发展的早期，所有节点都是全节点；当前的比特币核心客户端也是完整区块链节点。但在过去的两年中出现了许多新型客户端，它们不需要维持完整的区块链，而是作为轻量级客户端运行。在下面的章节里我们会对这些轻量级客户端进行详细介绍。

完整区块链节点保有完整的、最新的包含全部交易信息的比特币区块链拷贝，这样的节点可以独立地进行建立并校验区块链，从第一区块（创世区块）一直建立到网络中最新的区块。完整区块链节点可以独立自主地校验任何交易信息，而不需要借助任何其他节点或其他信息来源。完整区块节点通过比特币网络获取包含交易信息的新区块更新，在验证无误后将此更新

合并至本地的区块链拷贝之中。

运行完整区块链节点可以给您一种纯粹的比特币体验：不需借助或信任其他系统即可独立地对所有交易信息进行验证。辨别您是否在运行全节点是十分容易的：只需要查看您的永久性存储设备（如硬盘）是否有超过20GB的空间被用来存储完整区块链即可。如果您需要很大的磁盘空间、并且同步比特币网络耗时2至3天，那么您使用的正是全节点。这就是摆脱中心化管理、获得完全的独立自由所要付出的代价。

尽管目前还有一些使用不同编程语言及软件架构的其他的完整区块链客户端存在，但是最常用的仍然是比特币核心客户端，它也被称为“Satoshi客户端”。比特币网络中超过90%的节点运行着各个版本的比特币核心客户端。如前文所述，它可以通过节点间发送的version消息或通过getpeerinfo命令所得到的子版本字符串“Satoshi”加以辨识，例如 /Satoshi: 0.8.6/。

6.6 交换“库存清单”

一个全节点连接到对等节点之后，第一件要做的事情就是构建完整的区块链。如果该节点是一个全新节点，那么它就不包含任何区块链信息，它只知道一个区块——静态植入在客户端软件中的创世区块。新节点需要下载从0号区块（创世区块）开始的数十万区块的全部内容，才能跟网络同步、并重建全区块链。

同步区块链的过程从发送version消息开始，这是因为该消息中含有的BestHeight字段标示了一个节点当前的区块链高度（区块数量）。节点可以从它的对等节点中得到版本消息，了解双方各自有多少区块，从而可以与其自身区块链所拥有的区块数量进行比较。对等节点们会交换一个getblocks消息，其中包含他们本地区块链的顶端区块哈希值。如果某个对等节点识别出它接收到的哈希值并不属于顶端区块，而是属于一个非顶端区块的旧区块，那么它就能推断出：其自身的本地区块链比其他对等节点的区块链更长。

拥有更长区块链的对等节点比其他节点有更多的区块，可以识别出哪些区块们是其他节点需要“补充”的。它会识别出第一批可供分享的500个区块，通过使用inv (inventory) 消息把这些区块的哈希值传播出去。缺少这些区块的节点便可以通过各自发送的getdata消息来请求得到全区块信息，用包含在inv消息中的哈希值来确认是否为正确的被请求的区块，从而读取这些缺失的区块。

在下例中，我们假设某节点只含有创世区块。它收到了来自对等节点的inv消息，其中包含了区块链中后500个区块的哈希值。于是它开始向所有与之相连的对等节点请求区块，并通过分摊工作量的方式防止单一对等节点被批量请求所压垮。该节点会追踪记录其每个对等节点连接上“正在传输”（指那些它已经发出了请求但还没有接收到）的区块数量，并且检查该数量有没有超过上限（`MAX_BLOCKS_IN_TRANSIT_PER_PEER`）。用这种办法，如果一个节点需要更新大量区块，它会在上一请求完成后才发送对新区块的请求，从而允许对等节点控制更新速度，不至于压垮网络。每一个区块在被接收后就会被添加至区块链中，这一过程详见第7章。随着本地区块链的逐步建立，越来越多的区块被请求和接收，整个过程将一直持续到该节点与全网络完成同步为止。

每当一个节点离线，不管离线时间有多长，这个与对等节点比较本地区块链并恢复缺失区块的过程就会被触发。如果一个节点只离线几分钟，可能只会缺失几个区块；当它离线长达一个月，可能会缺失上千个区块。但无论哪种情况，它都会从发送 getblocks 消息开始，收到一个inv响应，接着开始下载缺失的区块库存清单和区块广播协议如图6-6所示。

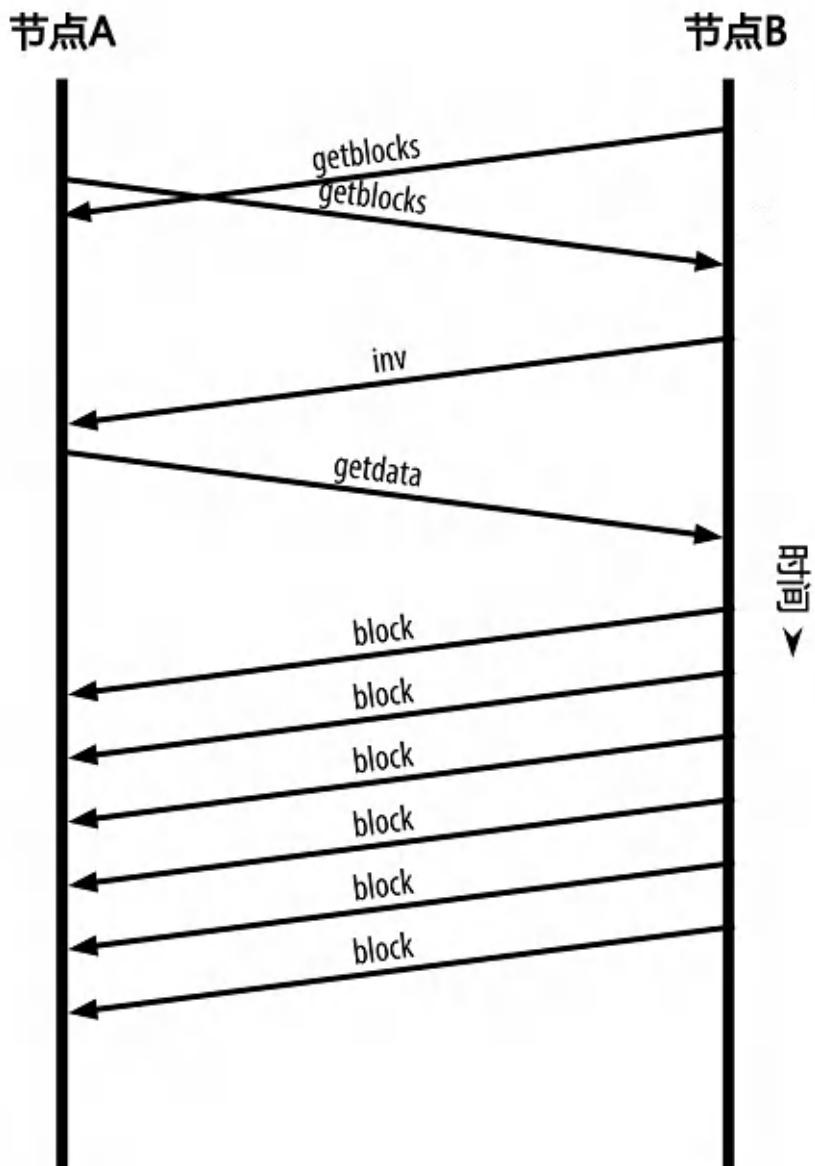


图6-6 节点通过从对等节点读取区块来同步区块链

6.7 简易支付验证（SPV）节点

并非所有的节点都有能力储存完整的区块链。许多比特币客户端被设计成运行在空间和功率受限的设备上，如智能电话、平板电脑、嵌入式系统等。对于这样的设备，通过简化的支付验证（SPV）的方式可以使它们在不必存储完整区块链的情况下进行工作。这种类型的客户端被称为SPV客户端或轻量级客户端。随着比特币的使用热潮，SPV节点逐渐变成比特币节点（尤其是比特币钱包）所采用的最常见的形式。

SPV节点只需下载区块头，而不用下载包含在每个区块中的交易信息。由此产生的不含交易信息的区块链，大小只有完整区块链的1/1000。SPV节点不能构建所有可用于消费的UTXO的全貌，这是由于它们并不知道网络上所有交易的完整信息。SPV节点验证交易时所使用的方法略有不同，这个方法需依赖对等节点“按需”提供区块链相关部分的局部视图。

打个比方来说，每个全节点就像是一个在陌生城市里的游客，他带着一张包含每条街道、每个地址的详细地图。相比之下，SPV节点就像是这名陌生城市里的游客只知道一条主干道的名字，通过随机询问该城市的陌生人来获取分段道路指示。虽然两种游客都可以通过实地考察来验证一条街是否存在，但没有地图的游客不知道每个小巷中有哪些街道，也不知道附近还有什么其他街道。没有地图的游客在“教堂街23号”的前面，并不知道这个城市里是否还有其他若干条“教堂街23号”，也不知道面前的这个是否是要找的那个。对他来说，最好的方式就是向足够多的人问路，并且希望其中一部分人不是要试图抢劫他。

简易支付验证是通过参考交易在区块链中的深度，而不是高度，来验证它们。一个拥有完整区块链的节点会构造一条验证链，这条链是由沿着区块链接按时间倒序一直追溯到创世区块的数千区块及交易组成。而一个SPV节点会验证所有区块的链（但不是所有的交易），并且把区块链和有关交易链接起来。

例如，一个全节点要检查第300,000号区块中的某个交易，它会把从该区块开始一直回溯到创世区块的300,000个区块全部都链接起来，并建立一个完整的UTXO数据库，通过确认该UTXO是否还未被支付来证实交易的有效性。SPV节点则不能验证UTXO是否还未被支付。相反地，SPV节点会在该交易信息和它所在区块之间用merkle路径（见“[7.7 Merkle 树](#)”）建立一条链接。然后SPV节点一直等待，直到序号从300,001到300,006的六个区块堆叠在该交易所在的区块之上，并通过确立交易的深度是在第300,006区块~第300,001区块之下来验证交易的有效性。事实上，如果网络中的其他节点都接受了第300,000区块，并通过足够的工作在该块之上又生成了六个区块，根据代理网关协议，就可以证明该交易不是双重支付。

如果一个交易实际上不存在，SPV节点不会误认为该交易存在于某区块中。SPV节点会通过请求merkle路径证明以及验证区块链中的工作量证明，来证实交易的存在性。可是，一个交易的存在是可能对SPV节点“隐藏”的。SPV节点毫无疑问可以证实某个交易的存在性，但它不能验证某个交易（譬如同一个UTXO的双重支付）不存在，这是因为SPV节点没有一份关于所有交易的记录。这个漏洞会被针对SPV节点的拒绝服务攻击或双重支付型攻击所利用。为了防御这些攻击，SPV节点需要随机连接到多个节点，以增加与至少一个可靠节点相连接的概率。这种随机连接的需求意味着SPV节点也容易受到网络分区攻击或Sybil攻击。在后者情况中，SPV节点被连接到虚假节点或虚假网络中，没有通向可靠节点或真正的比特币网络的连接。

在绝大多数的实际情况中，具有良好连接的SPV节点是足够安全的，它在资源需求、实用性和安全性之间维持恰当的平衡。当然，如果要保证万无一失的安全性，最可靠的方法还是运行完整区块链的节点。



完整的区块链节点是通过检查整个链中在它之下的数千个区块来保证这个UTXO没有被支付，从而验证交易。而SPV节点是通过检查在其上面的区块将它压在下面的深度来验证交易。

SPV节点使用的是一个getheaders消息，而不是getblocks消息来获得区块头。发出响应的对等节点将用一个headers消息发送多达2000个区块头。这一过程和全节点获取所有区块的过程没什么区别。SPV节点还在与对等节点的连接上设置了过滤器，用以过滤从对等节点发来的未来区块和交易数据流。任何目标交易都是通过一个getdata的请求来读取的。对等节点生成一个包含交易信息的tx消息作为响应。区块头的同步过程如图6-7所示。

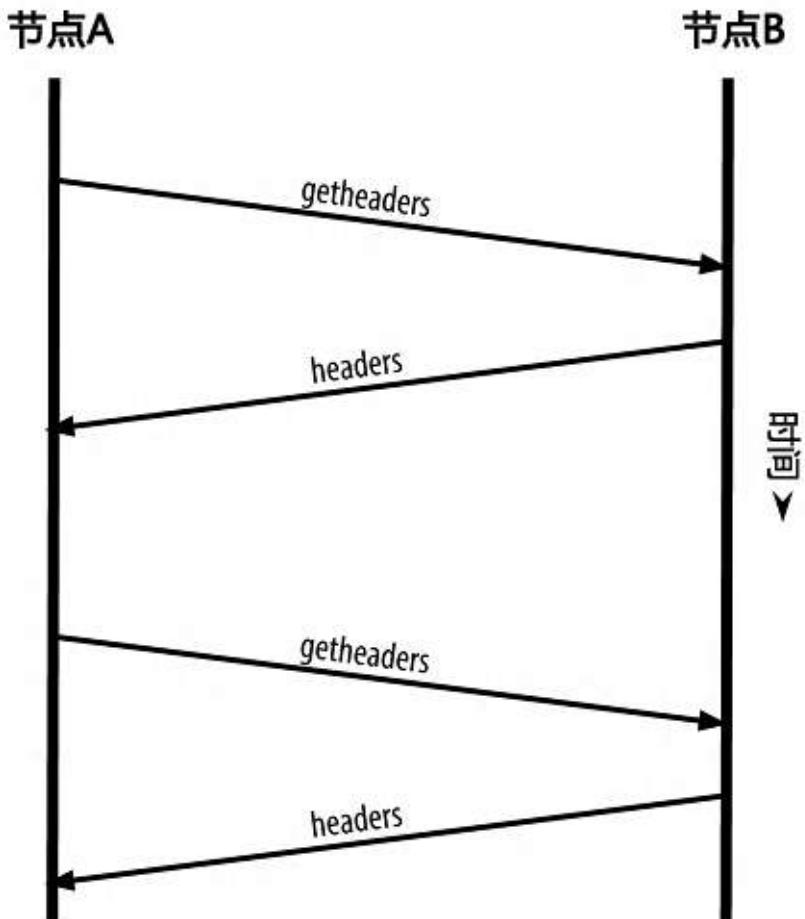


图6-7 SPV节点同步区块头

由于SPV节点需要读取特定交易从而选择性地验证交易，这样就又产生了隐私风险。与全区块链节点收集每一个区块内的全部交易所不同的是，SPV节点对特定数据的请求可能无意中透露了钱包里的地址信息。例如，监控网络的第三方可以跟踪某个SPV节点上的钱包所请求的全部交易信息，并且利用这些交易信息把比特币地址和钱包的用户关联起来，从而损害了用户的隐私。

在引入SPV节点/轻量级节点后不久，比特币开发人员就添加了一个新功能：Bloom过滤器，用以解决SPV节点的隐私风险问题。Bloom过滤器通过一个采用概率而不是固定模式的过滤机制，允许SPV节点只接收交易信息的子集，同时不会精确泄露哪些是它们感兴趣的地址。

6.8 Bloom过滤器

Bloom过滤器是一个允许用户描述特定的关键词组合而不必精确表述的基于概率的过滤方法。它能让用户在有效搜索关键词的同时保护他们的隐私。在SPV节点里，这一方法被用来向对等节点发送交易信息查询请求，同时交易地址不会被暴露。

用我们之前的例子，一位手中没有地图的游客需要询问去特定地方的路线。如果他向陌生人询问“教堂街23号在哪里”，不经意之间，他就暴露了自己的目的地。Bloom过滤器则会这样问，“附近有带‘堂’字的街道吗？”这样的问法包含了比之前略少的关键词。这位游客可以自己选择包含信息的多少，比如“以‘堂街’结尾”或者“教”字开头的街道。如果他问得越少，得到了更多可能的地址，隐私得到了保护，但这些地址里面不乏无关的结果；如果他问得非常具体，他在得到较准确的结果的同时也暴露了自己的隐私。

Bloom过滤器可以让SPV节点指定交易的搜索模式，该搜索模式可以基于准确性或私密性的考虑被调节。一个非常具体的Bloom过滤器会生成更准确的结果，但也会显示该用户钱包里的使用的地址；反之，如果过滤器只包含简单的关键词，更多相应的交易会被搜索出来，在包含若干无关交易的同时有着更高的私密性。

首先，SPV节点会初始化一个不会匹配任何关键词的“空白”Bloom过滤器。接下来，SPV节点会创建一个包含钱包中所有地址

信息的列表，并创建一个与每个地址相对应的交易输出相匹配的搜索模式。通常，这种搜索模式是一个向公钥付款的哈希脚本，该脚本是一个会出现在每一个向公钥哈希地址付款的交易中的锁定脚本。如果SPV节点需要追踪P2SH地址余额，搜索模式就会变成P2SH脚本。然后，SPV节点会把每一个搜索模式添加至Bloom过滤器里，这样只要关键词出现在交易中就能够被过滤器识别出来。最后，对等节点会用收到的Bloom过滤器来匹配传送至SPV节点的交易。

Bloom过滤器的实现是由一个可变长度（N）的二进制数组（N位二进制数构成一个位域）和数量可变（M）的一组哈希函数组成。这些哈希函数的输出值始终在1和N之间，该数值与二进制数组相对应。并且该函数为确定性函数，也就是说任何一个使用相同Bloom过滤器的节点通过该函数都能对特定输入得到同一个结果。Bloom过滤器的准确性和私密性能通过改变长度（N）和哈希函数的数量（M）来调节。

在图6-8中，我们用一个小型的十六位数组和三个哈希函数来演示Bloom过滤器的应用原理。



图6-8 一个由16位数组和三个哈希函数组成的简易Bloom过滤

Bloom过滤器数组里的每一个数的初始值为零。关键词被加到Bloom过滤器中之前，会依次通过每一个哈希函数运算一次。该输入经第一个哈希函数运算后得到了一个在1和N之间的数，它在该数组（编号依次为1至N）中所对应的位被置为1，从而把哈希函数的输出记录下来。接着再进行下一个哈希函数的运算，把另外一位置为1；以此类推。当全部M个哈希函数都运算过之后，一共有M个位的值从0变成了1，这个关键词也被“记录”在了Bloom过滤器里。

图6-9显示了向图6-8里的简易Bloom过滤器添加关键词“A”。

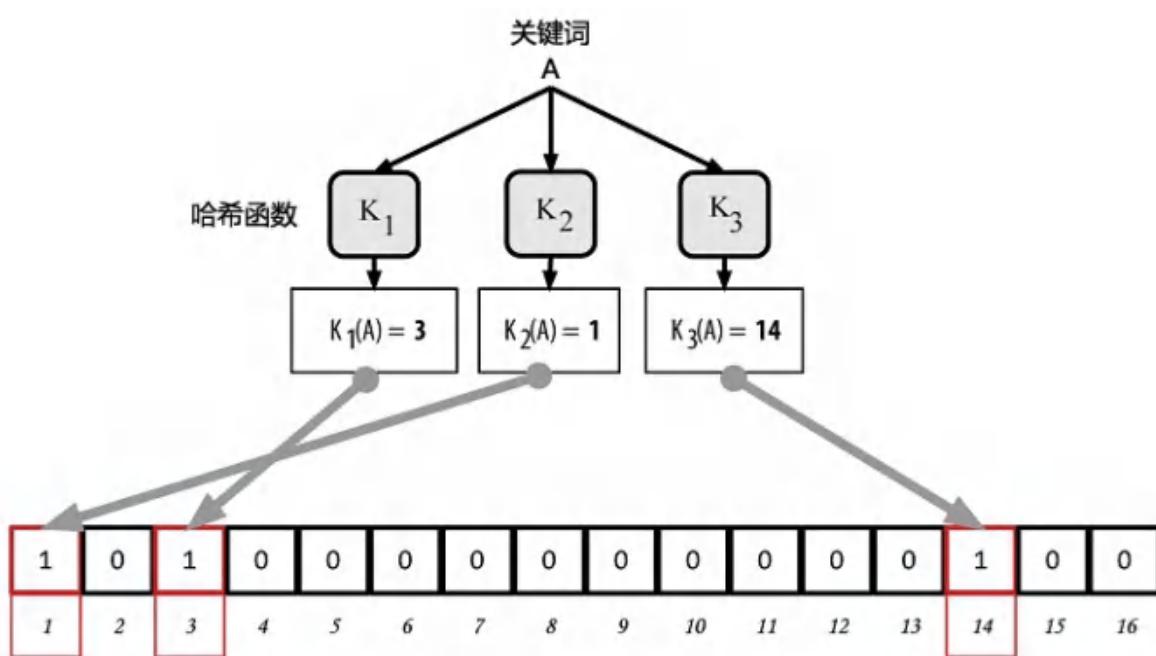


图6-9 向简易Bloom过滤器中增加关键词“A”

增加第二个关键是就是简单地重复之前的步骤。关键词依次通过各哈希函数运算之后，相应的位变为1，Bloom过滤器则记录下该关键词。需要注意的是，当Bloom过滤器里的关键词增加时，它对应的某个哈希函数的输出值的位可能已经是1了，这种情况下，该位不会再次改变。也就是说，随着更多的关键词指向了重复的位，Bloom过滤器随着位1的增加而饱和，准确性也因此降低了。该过滤器之所以是基于概率的数据结构，就是因为关键词的增加会导致准确性的降低。准确性取决于关键字的数量以及数组大小（N）和哈希函数的多少（M）。更大的数组和更多的哈希函数会记录更多的关键词以提高准确性。而小的数组及有限的哈希函数只能记录有限的关键词从而降低准确性。

图6-10显示了向该简易Bloom过滤器里增加第二个关键词“B”。

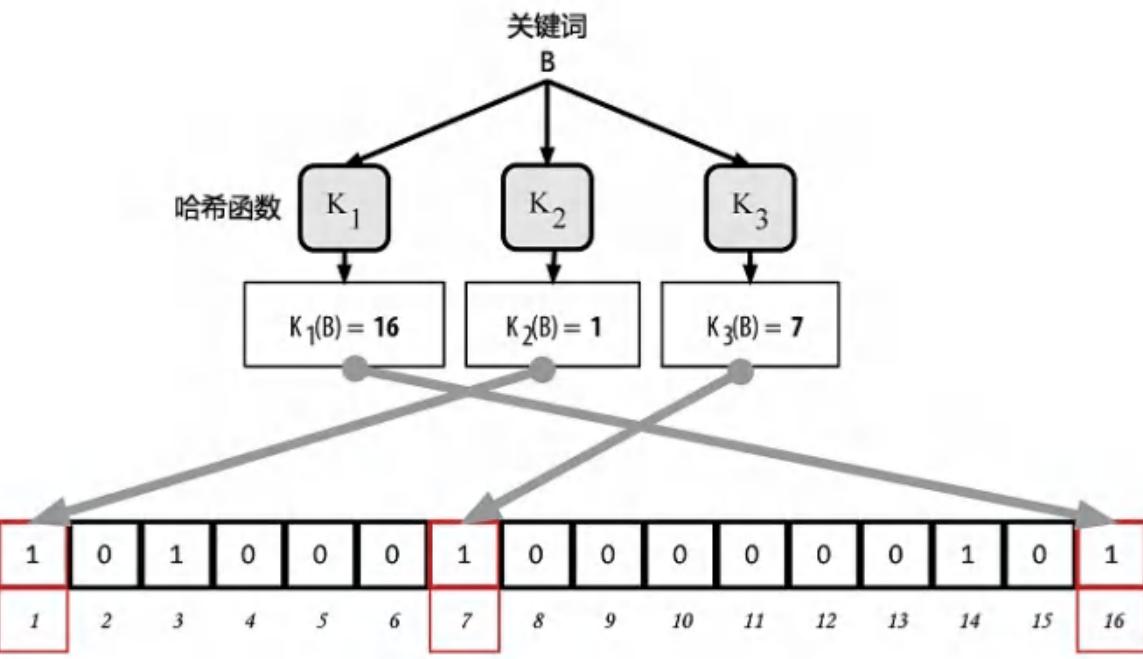


图6-10 向简易Bloom过滤器中增加第二个关键词“B”

为测试某一关键词是否被记录在某个Bloom过滤器中，我们将该关键词逐一代入各哈希函数中运算，并将所得的结果与原数组进行对比。如果所有的结果对应的位都变为了1，则表示这个关键词有可能已被该过滤器记录。之所以这一结论并不确定，是因为这些字节1也有可能是其他关键词运算的重叠结果。简单来说，Bloom过滤器正匹配代表着“可能是”。

图6-11是一个验证关键词“X”是否在前述Bloom过滤器中的图例。相应的比特位都被置为1，所以这个关键词很有可能是匹配的。

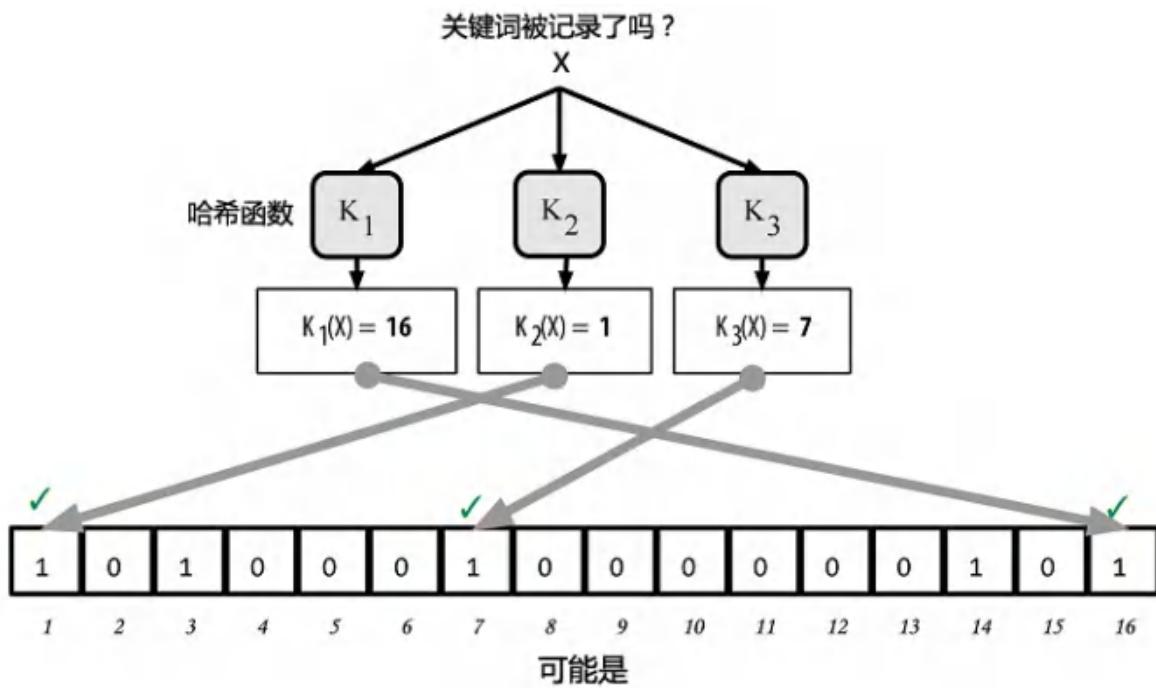


图6-11 验证关键词“X”是否存在于Bloom过滤器中。若结果为或然正匹配，则表示“可能是”。

另一方面，如果我们代入关键词计算后的结果某位为0，说明该关键词并没有被记录在过滤器里。负匹配的结果不是可能，而是一定。也就是说，负匹配代表着“一定不是”。

图6-12是一个验证关键词“Y”是否存在于简易Bloom过滤器中的图例。图中某个结果字段为0，该字段一定没有被匹配。

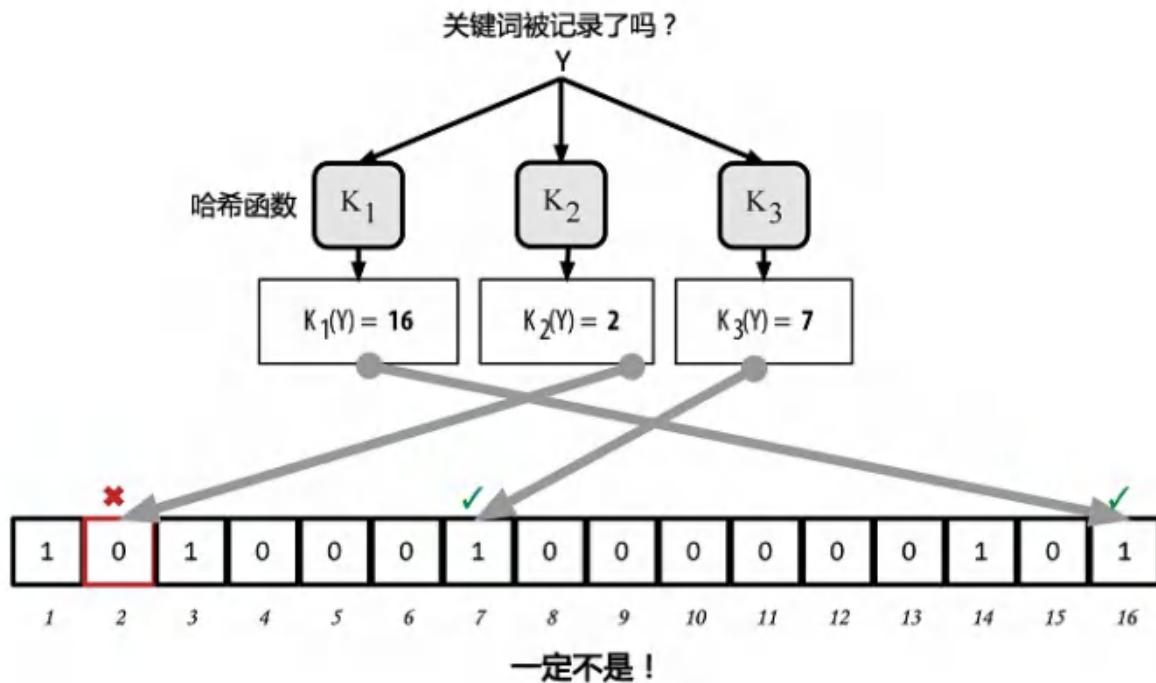


图6-12 验证关键词“Y”是否存在于Bloom过滤器中。若结果为必然负匹配，则表示“一定不是”。

BIP0037里已经对Bloom过滤器的实现有所描述。请参见附录2或访问[GitHub](#)。

6.9 Bloom过滤器和存货更新

Bloom过滤器被用来过滤SPV节点从对等节点里收到的交易信息。SPV会建立一个只能和SPV节点钱包里的地址匹配的过滤

器。随后，SPV节点会向对等节点发送一条包含需在该连接中使用的过滤器的filterload消息。当过滤器建好之后，对等节点将每个交易的输出值代入过滤器中验证。那些正匹配的交易会被传送回SPV节点。

为回应来自SPV节点的getdata信息，对等节点会发出一条只含有和过滤器匹配的区块的区块头信息，以及与之相匹配的交易的merkle树。这一对等节点还会发出一条相匹配的交易的tx消息。

这个节点能够通过发送一条filteradd信息来向它的Bloom过滤器增加关键词，也能够发送filterclear信息来清除整个过滤器。因为不能直接从过滤器里删除关键词，所以如果某关键词不再需要，节点必须通过清除和增加来替换原有的过滤器。

6.10 交易池

比特币网络中几乎每个节点都会维护一份未确认交易的临时列表，被称为内存池或交易池。节点们利用这个池来追踪记录那些被网络所知晓、但还未被区块链所包含的交易。例如，保存用户钱包的节点会利用这个交易池来记录那些网络已经接收但还未被确认的、属于该用户钱包的预支付信息。

随着交易被接收和验证，它们被添加到交易池并通知到相邻节点处，从而传播到网络中。

有些节点的实现还维护一个单独的孤立交易池。如果一个交易的输入与某未知的交易有关，如与缺失的父交易相关，该孤立交易就会被暂时储存在孤立交易池中直到父交易的信息到达。

当一个交易被添加到交易池中，会同时检查孤立交易池，看是否有某个孤立交易引用了此交易的输出（子交易）。任何匹配的孤立交易会被进行验证。如果验证有效，它们会从孤立交易池中删除，并添加到交易池中，使以其父交易开始的链变得完整。对新加入交易池的交易来说，它不再是孤立交易。前述过程重复递归寻找进一步的后代，直至所有的后代都被找到。通过这一过程，一个父交易的到达把整条链中的孤立交易和它们的父级交易重新结合在一起，从而触发了整条独立交易链进行级联重构。

交易池和孤立交易池（如有实施）都是存储在本地内存中，并不是存储在永久性存储设备（如硬盘）里。更准确的说，它们是随网络传入的消息动态填充的。节点启动时，两个池都是空闲的；随着网络中新交易不断被接收，两个池逐渐被填充。

有些比特币客户端的实现还维护一个UTXO数据库，也称UTXO池，是区块链中所有未支付交易输出的集合。“UTXO池”的名字听上去与交易池相似，但它代表了不同的数据集。UTXO池不同于交易池和孤立交易池的地方在于，它在初始化时不为空，而是包含了数以百万计的未支付交易输出条目，有些条目的历史甚至可以追溯至2009年。UTXO池可能会被安置在本地内存，或者作为一个包含索引的数据库表安置在永久性存储设备中。

交易池和孤立交易池代表的是单个节点的本地视角。取决于节点的启动时间或重启时间，不同节点的两池内容可能有很大差别。相反地，UTXO池代表的是网络的突显共识，因此，不同节点间UTXO池的内容差别不大。此外，交易池和孤立交易池只包含未确认交易，而UTXO池之只包含已确认交易。

6.11 警告消息

警告消息并不经常使用，但在大多数节点上都有此功能。警告消息是比特币的“紧急广播系统”，比特币核心开发人员可以借此功能给所有比特币节点发送紧急文本消息。这一功能是为了让核心开发团队将比特币网络的严重问题通知所有的比特币用户，例如一个需要用户采取措施的严重bug。警告系统迄今为止只被用过几次，最严重的一次是在2013年，一个关键的数据缺陷导致比特币区块链中出现了一个多区块分叉。

警告消息是通过alert消息来传播的。警告消息包含几个字段，包括：

▷ **ID**

警告消息序号，用于检测重复警告

▷ **Expiration**

警告到期的时间点

▷ **RelayUntil**

在此时间点之后，警告不再被中继

▷ **MinVer,MaxVer**

此警告所适用的比特币协议版本范围

▷ **subVer**

此警告适用的客户端软件版本

▷ **Priority**

警告消息的优先级（暂未使用）

警告通过公钥进行加密签名。对应的私钥是由核心开发团队的一些特定成员所持有。这样的数字签名可以确保虚假警告不会在网络中传播。

收到警告消息的节点会验证该消息，检查是否过期，并传播给其所有对等节点，从而保证了整个网络中的快速传播。除了传播警告之外，节点也可能会实现一个向用户推送警告的用户级接口函数。

在比特币核心客户端，警告是与命令行选项 `alertnotify` 一起设置的，该选项指定了收到警告时需要执行的命令。警告消息作为参数被传递给 `alertnotify` 命令。最常见的应用是，`alertnotify` 命令被设置为生成一个包含该警告消息的电子邮件并发送给节点管理员。警告也会以弹出对话框的形式显示在图形用户界面（如有运行）上（`bitcoin-Qt`）。

其他比特币协议的实现可能以不同的方式来处理警告。许多硬件嵌入式比特币挖矿系统由于没有用户界面，并没有实现警告消息功能。我们强烈建议运行这类挖矿系统的矿工订阅警告消息，既可以通过矿池运营方来订阅、也可以通过运行一个单独以警告为目的的轻量级节点来订阅。

第7章 区块链

7.1 简介

区块链是由包含交易信息的区块从后向前有序链接起来的数据结构。它可以被存储为flat file（一种包含没有相对关系记录的文件），或是存储在一个简单数据库中。比特币核心客户端使用Google的LevelDB数据库存储区块链元数据。区块被从后向前有序地链接在这个链条里，每个区块都指向前一个区块。区块链经常被视为一个垂直的栈，第一个区块作为栈底的首区块，随后每个区块都被放置在其他区块之上。用栈来形象化表示区块依次堆叠这一概念后，我们便可以使用一些术语，例如：“高度”来表示区块与首区块之间的距离；以及“顶部”或“顶端”来表示最新添加的区块。

对每个区块头进行SHA256加密哈希，可生成一个哈希值。通过这个哈希值，可以识别出区块链中的对应区块。同时，每一个区块都可以通过其区块头的“父区块哈希值”字段引用前一区块（父区块）。也就是说，每个区块头都包含它的父区块哈希值。这样把每个区块链接到各自父区块的哈希值序列就创建了一条一直可以追溯到第一个区块（创世区块）的链条。

虽然每个区块只有一个父区块，但可以暂时拥有多个子区块。每个子区块都将同一区块作为其父区块，并且在“父区块哈希值”字段中具有相同的（父区块）哈希值。一个区块出现多个子区块的情况被称为“区块链分叉”。区块链分叉只是暂时状态，只有当多个不同区块几乎同时被不同的矿工发现时才会发生（参见[8.10.1 区块链分叉](#)）。最终，只有一个子区块会成为区块链的一部分，同时解决了“区块链分叉”的问题。尽管一个区块可能会有不止一个子区块，但每一区块只有一个父区块，这是因为一个区块只有一个“父区块哈希值”字段可以指向它的唯一父区块。

由于区块头里面包含“父区块哈希值”字段，所以当前区块的哈希值因此也受到该字段的影响。如果父区块的身份标识发生变化，子区块的身份标识也会跟着变化。当父区块有任何改动时，父区块的哈希值也发生变化。父区块的哈希值发生改变将迫使子区块的“父区块哈希值”字段发生改变，从而又将导致子区块的哈希值发生改变。而子区块的哈希值发生改变又将迫使孙区块的“父区块哈希值”字段发生改变，又因此改变了孙区块哈希值，等等以此类推。一旦一个区块有很多代以后，这种瀑布效应将保证该区块不会被改变，除非强制重新计算该区块所有后续的区块。正是因为这样的重新计算需要耗费巨大的计算量，所以一个长区块链的存在可以让区块链的历史不可改变，这也是比特币安全性的一个关键特征。

你可以把区块链想象成地质构造中的地质层或者是冰川岩芯样品。表层可能会随着季节而变化，甚至在沉积之前就被风吹走了。但是越往深处，地质层就变得越稳定。到了几百英尺深的地方，你看到的将是保存了数百万年但依然保持历史原状的岩层。在区块链里，最近的几个区块可能会由于区块链分叉所引发的重新计算而被修改。最新的六个区块就像几英寸深的表土层。但是，超过这六块后，区块在区块链中的位置越深，被改变的可能性就越小。在100个区块以后，区块链已经足够稳定，这时Coinbase交易（包含新挖出的比特币的交易）可以被支付。几千个区块（一个月）后的区块链将变成确定的历史，永远不会改变。

7.2 区块结构

区块是一种被包含在公开账簿（区块链）里的聚合了交易信息的容器数据结构。它由一个包含元数据的区块头和紧跟其后的构成区块主体的一长串交易组成。区块头是80字节，而平均每个交易至少是250字节，而且平均每个区块至少包含超过500个交易。因此，一个包含所有交易的完整区块比区块头的1000倍还要大。表7-1描述了一个区块结构。

表7-1 区块结构

大小	字段	描述
4字节	区块大小	用字节表示的该字段之后的区块大小
80字节	区块头	组成区块头的几个字段
1-9（可变整数）	交易计数器	交易的数量
可变的	交易	记录在区块里的交易信息

7.3 区块头

区块头由三组区块元数据组成。首先是一组引用父区块哈希值的数据，这组元数据用于将该区块与区块链中前一区块相连接。第二组元数据，即难度、时间戳和nonce，与挖矿竞争相关，详见第8章。第三组元数据是merkle树根（一种用来有效地总结区块中所有交易的数据结构）。表7-2描述了区块头的数据结构。

表7-2 区块头结构

大小	字段	描述
4字节	版本	版本号，用于跟踪软件/协议的更新
32字节	父区块哈希值	引用区块链中父区块的哈希值
32字节	Merkle根	该区块中交易的merkle树根的哈希值
4字节	时间戳	该区块产生的近似时间（精确到秒的Unix时间戳）
4字节	难度目标	该区块工作量证明算法的难度目标
4字节	Nonce	用于工作量证明算法的计数器

Nonce、难度目标和时间戳会用于挖矿过程，更多细节将在第8章讨论。

7.4 区块标识符：区块头哈希值和区块高度

区块主标识符是它的加密哈希值，一个通过SHA256算法对区块头进行二次哈希计算而得到的数字指纹。产生的32字节哈希值被称为区块哈希值，但是更准确的名称是：区块头哈希值，因为只有区块头被用于计算。例如:000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f是第一个比特币区块的区块哈希值。区块哈希值可以唯一、明确地标识一个区块，并且任何节点通过简单地对区块头进行哈希计算都可以独立地获取该区块哈希值。

请注意，区块哈希值实际上并不包含在区块的数据结构里，不管是该区块在网络上传输时，抑或是它作为区块链的一部分被存储在某节点的永久性存储设备上时。相反，区块哈希值是当该区块从网络被接收时由每个节点计算出来的。区块的哈希值可能会作为区块元数据的一部分被存储在一个独立的数据库表中，以便于索引和更快地从磁盘检索区块。

第二种识别区块的方式是通过该区块在区块链中的位置，即“区块高度（block height）”。第一个区块，其区块高度为0，和之前哈希值000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f所引用的区块为同一个区块。因此，区块可以通过两种方式被识别：区块哈希值或者区块高度。每一个随后被存储在第一个区块之上的区块在区块链中都比前一区块“高”出一个位置，就像箱子一个接一个堆叠在其他箱子之上。2014年1月1日的区块高度大约是278,000，说明已经有278,000个区块被堆叠在2009年1月创建的第一个区块之上。

和区块哈希值不同的是，区块高度并不是唯一的标识符。虽然一个单一的区块总是会有一个明确的、固定的区块高度，但反过来却并不成立，一个区块高度并不总是识别一个单一的区块。两个或两个以上的区块可能有相同的区块高度，在区块链里争夺同一位置。这种情况在“8.10.1 区块链分叉”一节中有详细讨论。区块高度也不是区块数据结构的一部分，它并不被存储在区块里。当节点接收来自比特币网络的区块时，会动态地识别该区块在区块链里的位置（区块高度）。区块高度也可作为元数据存储在一个索引数据库表中以便快速检索。



一个区块的区块哈希值总是能唯一地识别出一个特定区块。一个区块也总是有特定的区块高度。但是，一个特定的区块高度并不一定总是能唯一地识别出一个特定区块。更确切地说，两个或者更多数量的区块也许会为了区块中的一个

位置而竞争。

7.5 创世区块

区块链里的第一个区块创建于2009年，被称为创世区块。它是区块链里面所有区块的共同祖先，这意味着你从任一区块，循链向后回溯，最终都将到达创世区块。

因为创世区块被编入到比特币客户端软件里，所以每一个节点都始于至少包含一个区块的区块链，这能确保创世区块不会被改变。每一个节点都“知道”创世区块的哈希值、结构、被创建的时间和里面的一个交易。因此，每个节点都把该区块作为区块链的首区块，从而构建了一个安全的、可信的区块链的根。

在[chainparams.cpp](#)里可以看到创世区块被编入到比特币核心客户端里。

创世区块的哈希值为：

```
0000000000 19d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

你可以在任何区块浏览网站搜索这个区块哈希值，如blockchain.info，你会发现一个用包含这个哈希值的链接来描述这一区块内容的页面：

<https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f>

<https://blockexplorer.com/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f>

在命令行使用比特币核心客户端：

```
$ bitcoindgetblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
{
  "hash": "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
  "confirmations": 308321,
  "size": 285,
  "height": 0,
  "version": 1,
  "merkleroot": "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
  "tx": ["4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"],
  "time": 1231006505,
  "nonce": 2083236893,
  "bits": "1d00ffff",
  "difficulty": 1.00000000,
  "nextblockhash": "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

创世区块包含一个隐藏的信息。在其Coinbase交易的输入中包含这样一句话“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks.”这句话是泰晤士报当天的头版文章标题，引用这句话，既是对该区块产生时间的说明，也可视为半开玩笑地提醒人们一个独立的货币制度的重要性，同时告诉人们随着比特币的发展，一场前所未有的世界性货币革命将要发生。该消息是由比特币的创立者中本聪嵌入创世区块中。

7.6 区块的连接

比特币的完整节点保存了区块链从创世区块起的一个本地副本。随着新的区块的产生，该区块链的本地副本会不断地更新用于扩展这个链条。当一个节点从网络接收传入的区块时，它会验证这些区块，然后链接到现有的区块链上。为建立一个连接，一个节点将检查传入的区块头并寻找该区块的“父区块哈希值”。

让我们假设，例如，一个节点在区块链的本地副本中有277,314个区块。该节点知道最后一个区块为第277,314个区块，这个区块的区块头哈希值为：000000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249。

然后该比特币节点从网络上接收到一个新的区块，该区块描述如下：

```
{  
    "size":43560,  
    "version":2,  
  
    "previousblockhash":"0000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",  
    "merkleroot":"5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",  
    "time":1388185038,  
    "difficulty":1180923195.25802612,  
    "nonce":4215469401,  
    "tx":["257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",  
        #[...many more transactions omitted...]  
        "05cf38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"  
    ]  
}
```

对于这一新的区块，节点会在“父区块哈希值”字段里找出包含它的父区块的哈希值。这是节点已知的哈希值，也就是第277314块区块的哈希值。故这个区块是这个链条里的最后一个区块的子区块，因此现有的区块链得以扩展。节点将新的区块添加至链条的尾端，使区块链变长到一个新的高度277,315。图7-1显示了通过“父区块哈希值”字段进行连接三个区块的链。

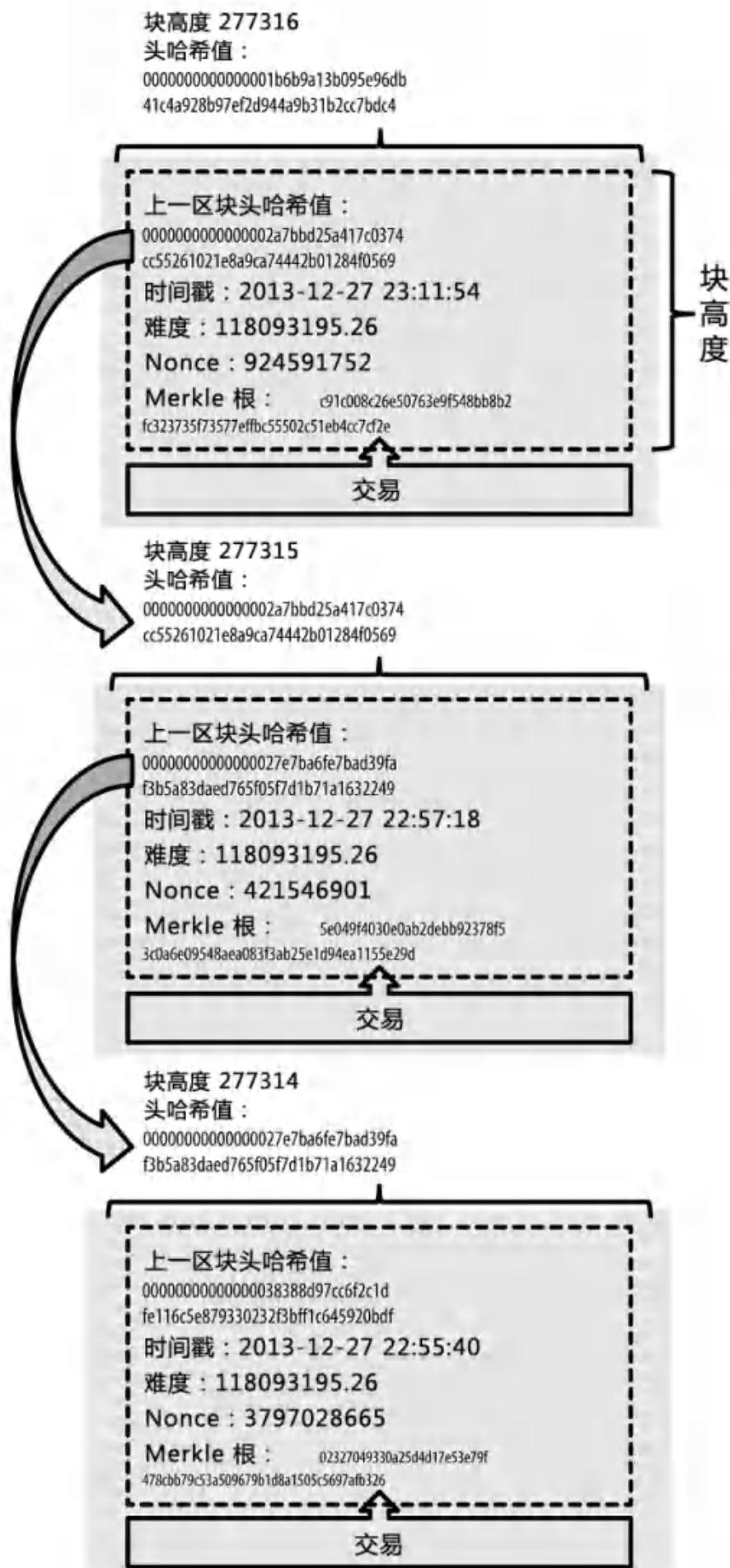


图7-1 区块通过引用父区块的区块头哈希值的方式，以链的形式进行相连

7.7 Merkle 树

区块链中的每个区块都包含了产生于该区块的所有交易，且以Merkle树表示。

Merkle树是一种哈希二叉树，它是一种用作快速归纳和校验大规模数据完整性的数据结构。这种二叉树包含加密哈希值。术语“树”在计算机学科中常被用来描述一种具有分支的数据结构，但是树常常被倒置显示，“根”在图的上部同时“叶子”在图的下部，你会在后续章节中看到相应的例子。

在比特币网络中，Merkle树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹，且提供了一种校验区块是否存在某交易的高效途径。生成一棵完整的Merkle树需要递归地对哈希节点对进行哈希，并将新生成的哈希节点插入到Merkle树中，直到只剩一个哈希节点，该节点就是Merkle树的根。在比特币的Merkle树中两次使用到了SHA256算法，因此其加密哈希算法也被称为double-SHA256。

当N个数据元素经过加密后插入Merkle树时，你至多计算 $2 \times \log_2(N)$ 次就能检查出任意某数据元素是否在该树中，这使得该数据结构非常高效。

Merkle树是自底向上构建的。在如下的例子中，我们从A、B、C、D四个构成Merkle树树叶的交易开始，如图7-2。起始时所有的交易都还未存储在Merkle树中，而是先将数据哈希化，然后将哈希值存储至相应的叶子节点。这些叶子节点分别是HA、HB、HC和HD：

$H_{A\sim} = \text{SHA256}(\text{SHA256}(\text{交易A}))$

通过串联相邻叶子节点的哈希值然后哈希之，这对叶子节点随后被归纳为父节点。例如，为了创建父节点HAB，子节点A和子节点B的两个32字节的哈希值将被串联成64字节的字符串。随后将字符串进行两次哈希来产生父节点的哈希值：

$H_{AB\sim} = \text{SHA256}(\text{SHA256}(H_{A\sim} + H_{B\sim}))$

继续类似的操作直到只剩下顶部的一个节点，即Merkle根。产生的32字节哈希值存储在区块头，同时归纳了四个交易的所有数据。

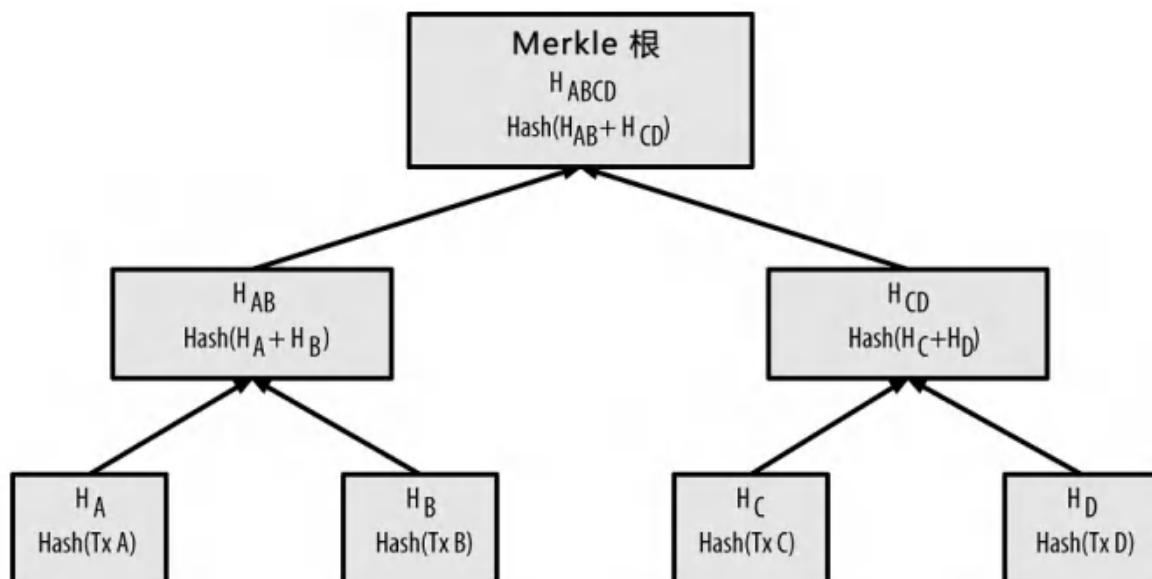


图7-2 在Merkle树中计算节点

因为Merkle树是二叉树，所以它需要偶数个叶子节点。如果仅有奇数个交易需要归纳，那最后的交易就会被复制一份以构成偶数个叶子节点，这种偶数个叶子节点的树也被称为平衡树。如图7-3所示，C节点被复制了一份。

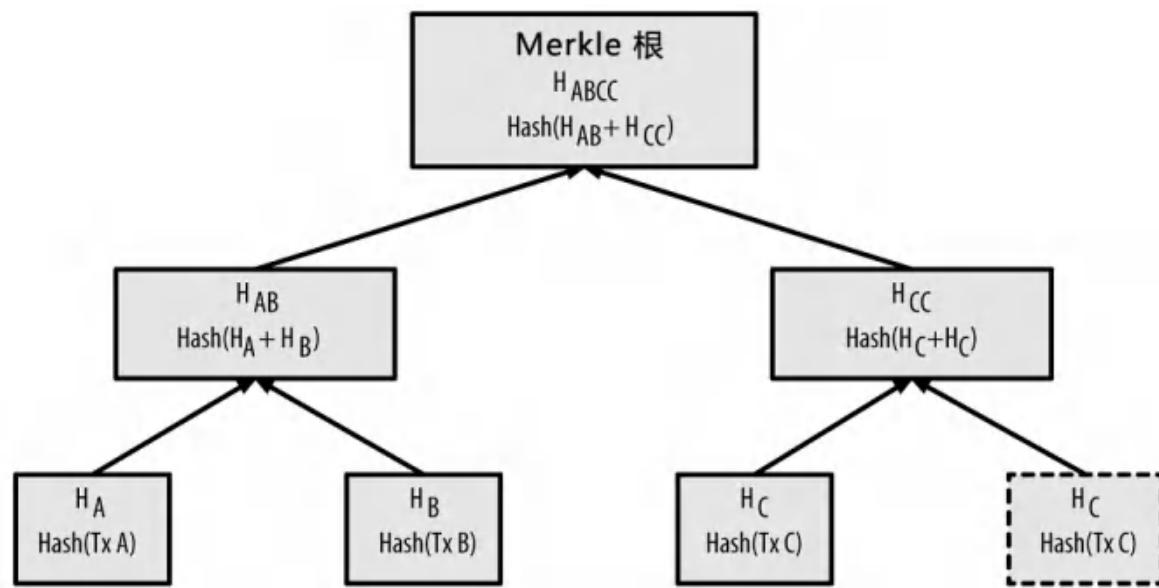


图7-3 复制一份数据节点，使整个树中数据节点个数是偶数

由四个交易构造Merkle树的方法同样适用于从任意交易数量构造Merkle树。在比特币中，在单个区块中有成百上千的交易是非常普遍的，这些交易都会采用同样的方法归纳起来，产生一个仅仅32字节的数据作为Merkle根。在图7-4中，你会看见一个从16个交易形成的树。需要注意的是，尽管图中的根看起来比所有叶子节点都大，但实际上它们都是32字节的相同大小。无论区块中有一个交易或者有十万个交易，Merkle根总会把所有交易归纳为32字节。

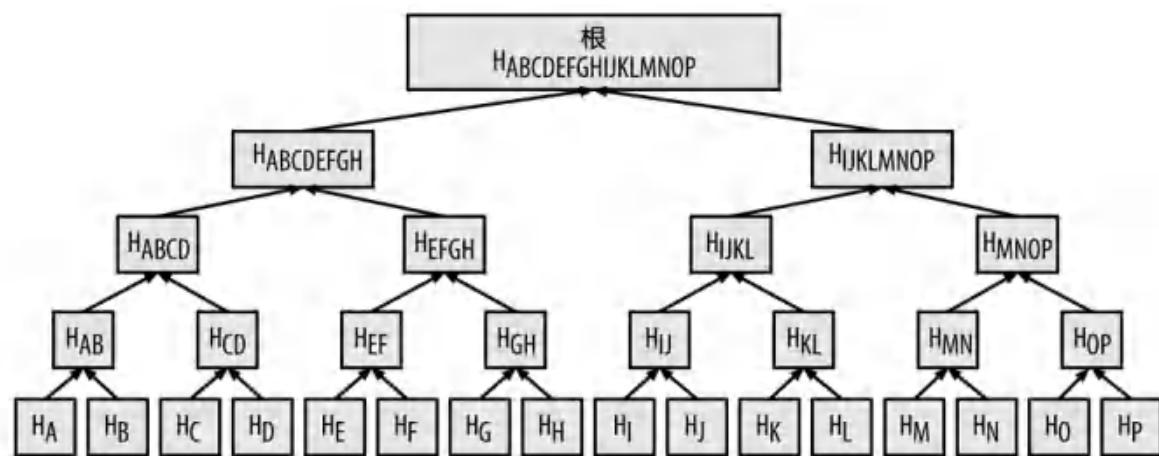


图7-4 一颗囊括了许多数据元素的Merkle树

为了证明区块中存在某个特定的交易，一个节点只需要计算 $\log_2(N)$ 个32字节的哈希值，形成一条从特定交易到树根的认证路径或者Merkle路径即可。随着交易数量的急剧增加，这样的计算量就显得异常重要，因为相对于交易数量的增长，以基底为2的交易数量的对数的增长会缓慢许多。这使得比特币节点能够高效地产生一条10或者12个哈希值（320-384字节）的路径，来证明了在一个巨量字节大小的区块中上千交易中的某笔交易的存在。

在图7-5中，一个节点能够通过生成一条仅有4个32字节哈希值长度（总128字节）的Merkle路径，来证明区块中存在一笔交易K。该路径有4个哈希值（在图7-5中由蓝色标注）H_L、H_IJ、H_MNOP和H_ABCDEFGH。由这4个哈希值产生的认证路径，再通过计算另外四对哈希值HKL、HIJKL、HIJKLMNOP和Merkle树根（在图中由虚线标注），任何节点都能证明HK（在图中由绿色标注）包含在Merkle根中。

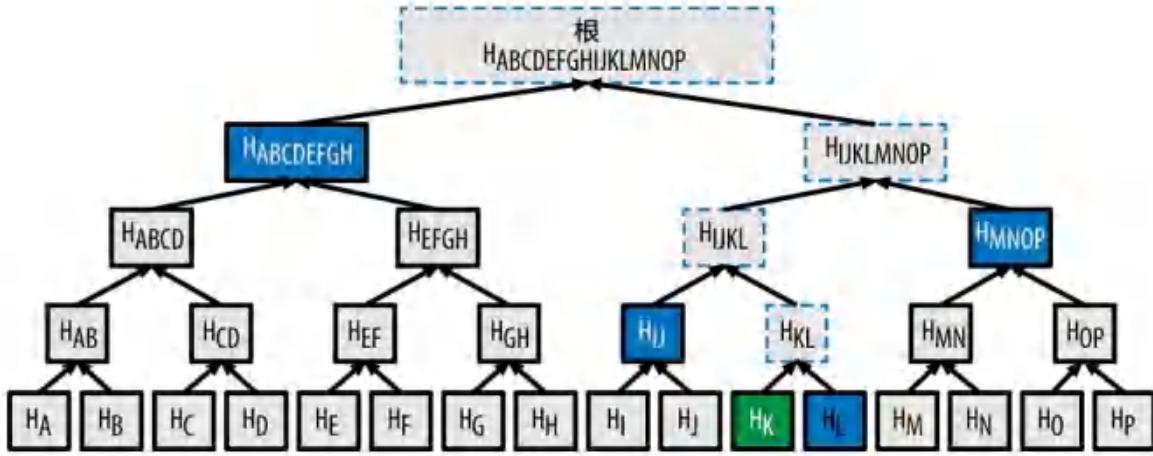


图7-5 一条为了证明树中包含某个数据元素而使用的Merkle路径

例7-1中的代码借用libbitcoin库中的一些辅助程序，演示了从叶子节点哈希至根创建整个Merkle树的过程。

例7-1 构造Merkle树

```
#include <bitcoin/bitcoin.hpp>
bc::hash_digest create_merkle(bc::hash_digest_list& merkle)
{
    // Stop if hash list is empty.
    if (merkle.empty())
        return bc::null_hash;
    else if (merkle.size() == 1)
        return merkle[0];
    // While there is more than 1 hash in the list, keep looping...
    while (merkle.size() > 1)
    {
        // If number of hashes is odd, duplicate last hash in the list.
        if (merkle.size() % 2 != 0)
            merkle.push_back(merkle.back());
        // List size is now even.
        assert(merkle.size() % 2 == 0);

        // New hash list.
        bc::hash_digest_list new_merkle;
        // Loop through hashes 2 at a time.
        for (auto it = merkle.begin(); it != merkle.end(); it += 2)
        {
            // Join both current hashes together (concatenate).
            bc::data_chunk concat_data(bc::hash_size * 2);
            auto concat = bc::make_serializer(concat_data.begin());
            concat.write_hash(*it);
            concat.write_hash(*(it + 1));
            assert(concat.iterator() == concat_data.end());
            // Hash both of the hashes.
            bc::hash_digest new_root = bc::bitcoin_hash(concat_data);
            // Add this to the new list.
            new_merkle.push_back(new_root);
        }
        // This is the new list.
        merkle = new_merkle;

        // DEBUG output -----
        std::cout << "Current merkle hash list:" << std::endl;
        for (const auto& hash: merkle)
            std::cout << " " << bc::encode_hex(hash) << std::endl;
        std::cout << std::endl;
        // -----
    }
    // Finally we end up with a single item.
    return merkle[0];
}

int main()
```

例7-2展示了编译以及运行上述代码后的结果。

例7-2 编译以及运行构造Merkle树代码

```
$ # Compile the merkle.cpp code
$ g++ -o merkle merkle.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the merkle executable
$ ./merkle
Current merkle hash list:
32650049a0418e4380db0af81788635d8b65424d397170b8499cdc28c4d27006
30861db96905c8dc8b99398ca1cd5bd5b84ac3264a4e1b3e65afa1bcee7540c4

Current merkle hash list:
d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

Result: d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3
```

Merkle树的高效随着交易规模的增加而变得异常明显。表7-3展示了为了证明区块中存在某交易而所需转化为Merkle路径的数据量。

表7-3 Merkle树的效率

交易数量	区块的近似大小	路径大小（哈希数量）	路径大小（字节）
16笔交易	4KB	4个哈希	128字节
512笔交易	128KB	9个哈希	288字节
2048笔交易	512KB	11个哈希	352字节
65,535笔交易	16MB	16个哈希	512字节

依表可得，当区块大小由16笔交易（4KB）急剧增加至65,535笔交易（16MB）时，为证明交易存在的Merkle路径长度增长极其缓慢，仅仅从128字节到512字节。有了Merkle树，一个节点能够仅下载区块头（80字节/区块），然后通过从一个满节点回溯一条小的Merkle路径就能认证一笔交易的存在，而不需要存储或者传输大量区块链中大多数内容，这些内容可能有几个G的大小。这种不需要维护一条完整的区块链的节点，又被称作简单支付验证（SPV）节点，它不需要下载整个区块而通过Merkle路径去验证交易的存在。

7.8 Merkle树和简单支付验证 (SPV)

Merkle树被SPV节点广泛使用。SPV节点不保存所有交易也不会下载整个区块，仅仅保存区块头。它们使用认证路径或者Merkle路径来验证交易存在于区块中，而不必下载区块中所有交易。

例如，一个SPV节点欲知它钱包中某个比特币地址即将到达的支付，该节点会在节点间的通信链接上建立起bloom过滤器，限制只接受含有目标比特币地址的交易。当节点探测到某交易符合bloom过滤器，它将以Merkle区块消息的形式发送该区块。Merkle区块消息包含区块头和一条连接目标交易与Merkle根的Merkle路径。SPV节点能够使用该路径找到与该交易相关的区块，进而验证对应区块中该交易的有无。SPV节点同时也使用区块头去关联区块和区块链中的区域区块。这两种关联，

交易与区块、区块和区块链，证明交易存在于区块链。简而言之，SPV节点会收到少于1KB的有关区块头和Merkle路径的数据，其数据量比一个完整的区块（目前大约有1MB）少了一千倍有余。

第8章 挖矿与共识

8.1 简介

挖矿是增加比特币货币供应的一个过程。挖矿同时还保护着比特币系统的安全，防止欺诈交易，避免“双重支付”，“双重支付”是指多次花费同一笔比特币。矿工们通过为比特币网络提供算力来换取获得比特币奖励的机会。

矿工们验证每笔新的交易并把它们记录在总帐簿上。每10分钟就会有一个新的区块被“挖掘”出来，每个区块里包含着从上一个区块产生到目前这段时间内发生的所有交易，这些交易被依次添加到区块链中。我们把包含在区块内且被添加到区块链上的交易称为“确认”交易，交易经过“确认”之后，新的拥有者才能够花费他在交易中得到的比特币。

矿工们在挖矿过程中会得到两种类型的奖励：创建新区块的新币奖励，以及区块中所含交易的交易费。为了得到这些奖励，矿工们争相完成一种基于加密哈希算法的数学难题，这些难题的答案包括在新区块中，作为矿工的计算工作量的证明，被称为“工作量证明”。该算法的竞争的机制以及获胜者有权在区块链上进行交易记录的机制，这二者比特币安全的基石。

新比特币的生成过程被称为挖矿是因为它的奖励机制被设计为速度递减模式，类似于贵重金属的挖矿过程。比特币的货币是通过挖矿发行的，类似于中央银行通过印刷银行纸币来发行货币。矿工通过创造一个新区块得到的比特币数量大约每四年（或准确说是每隔210,000个块）减少一半。开始时为2009年1月每个区块奖励50个比特币，然后到2012年11月减半为每个区块奖励25个比特币。之后将在2016年的某个时刻再次减半为每个新区块奖励12.5个比特币。基于这个公式，比特币挖矿奖励以指数方式递减，直到2140年。届时所有的比特币（20,999,999,980）全部发行完毕。换句话说在2140年之后，不会再有新的比特币产生。

矿工们同时也会获取交易费。每笔交易都可能包含一笔交易费，交易费是每笔交易记录的输入和输出的差额。在挖矿过程中“挖出”新区块的矿工获胜者可以得到该区块中包含的所有交易“小费”。目前，这笔费用占矿工收入的0.5%或更少，大部分收益仍来自挖矿所得的比特币奖励。然而随着挖矿奖励的递减，以及每个区块中包含的交易数量增加，交易费在矿工收益中所占的比重将会逐渐增加。在2140年之后，所有的矿工收益都将由交易费构成。

“挖矿”这个词有一定的误导性。它容易引起对贵重金属采矿的联想，从而使我们的注意力都集中在每个新区块产生的奖励上。尽管挖矿带来的奖励是一种激励，但它最主要的目的并不是奖励本身或者新币的产生。如果只把挖矿看作生产新币的过程，那你是把手段（激励措施）当成了目的。挖矿是一种将结算所去中心化的过程，每个结算所对处理的交易进行验证和结算。挖矿保护了比特币系统的安全，并且实现了在没有中心机构的情况下，也能使整个比特币网络达成共识。

挖矿这个发明使比特币变得很特别，这种去中心化的安全机制是点对点的电子货币的基础。铸造新币的奖励和交易费是一种激励机制，它可以调节矿工行为和网络安全，同时又完成了比特币的货币发行。

在本章中，我们先来审视比特币的货币发行机制，然后再来了解挖矿的最重要的功能：支撑比特币安全的去中心化的自发共识机制。

8.1.1 比特币经济学和货币创造

通过创造出新区块，比特币以一个确定的但不断减慢的速率被铸造出来。大约每十分钟产生一个新区块，每一个新区块都伴随着一定数量从无到有的全新比特币。每开采210,000个块，大约耗时4年，货币发行速率降低50%。在比特币运行的第一个四年中，每个区块创造出50个新比特币。

2012年11月，比特币的新发行速度降低到每区块25个比特币，并且预计会在2016年的某个时刻，在第420,000个区块被“挖掘”出来之后降低到12.5比特币/区块。在第13,230,000个区块（大概在2137年被挖出）之前，新币的发行速度会以指数形式进行64次“二等分”。到那时每区块发行比特币数量变为比特币的最小货币单位——1聪。最终，在经过1,344万个区块之后，所有的共2,099,999,997,690,000聪比特币将全部发行完毕。换句话说，到2140年左右，会存在接近2,100万比特币。在那之后，新的区块不再包含比特币奖励，矿工的收益全部来自交易费。图8-1展示了在发行速度不断降低的情况下，比特币总流通量与时间的关系。

在例8-1的代码展示中，我们计算了比特币的总发行量。

例8-1 比特币发行总量的计算脚本

```
# 初始的块奖励为50BTC
start_block_reward = 50
# 以10分钟为一个区块的间隔，210000个块共约4年时间
reward_interval = 210000

def max_money():
    # 50 BTC = 50 0000 0000 Satoshi
    current_reward = 50 * 10**8
    total = 0
    while current_reward > 0:
        total += reward_interval * current_reward
        current_reward /= 2
    return total

print "Total BTC to ever be created:", max_money(), "Satoshis"
```

例8-2显示了这个脚本的运行结果。

例8-2 运行 max_money.py 脚本

```
$ python max_money.py
Total BTC to ever be created: 209999997690000 Satoshi
```



图8-1 比特币货币供应速度随着时间发生几何级降低

总量有限并且发行速度递减创造了一种抗通胀的货币供应模式。法币可被中央银行无限制地印刷出来，而比特币永远不会因超额印发而出现通胀。

通货紧缩货币

最重要并且最有争议的一个结论是一种事先确定的发行速率递减的货币发行模式会导致货币通货紧缩（简称通缩）。通缩是一种由于货币的供应和需求不匹配导致的货币增值的现象。它与通胀相反，价格通缩意味着货币随着时间有越来越强的购买力。

许多经济学家提出通缩经济是一种无论如何都要避免的灾难型经济。因为在快速通缩时期，人们预期着商品价格会下

跌，人们将会储存货币，避免花掉它。这种现象充斥了日本经济“失去的十年”，就是在需求坍塌之后导致了滞涨状态。

比特币专家们认为通缩本身并不坏。更确切地说，我们将通缩与需求坍塌联系在一起是因为过去出现的一个特例。在法币届，货币是有可能被无限制印刷出来的，除非遇到需求完全崩塌并且毫无发行货币意愿的情形，因此经济很难进入滞涨期。而比特币的通缩并不是需求坍塌引起的，它遵循一种预定且有节制的货币供应模型。

实际上，通缩货币会让卖家考虑到折现的影响，容易诱发过度的囤积本能，除非这部分折现率超过买家的囤积本能。因为买卖双方都有囤积的动机，这两种折现率会因为双方的囤积本能相互抵消，而达成一个平衡价格。因此即使在比特币价格贴现率为30%的情况下，大部分使用比特币的零售商并不会感受到花费比特币很困难，也能因此盈利。当然，比特币这种不是因经济快速衰退而引起的通缩，是否会引起其他问题，仍有待观察。

8.2 去中心化共识

在上一章中我们了解了区块链。可以将区块链看作一本记录所有交易的公开总帐簿（列表），比特币网络中的每个参与者都把它看作一本所有权的权威记录。

但在不考虑相信任何人的情况下，比特币网络中的所有参与者如何达成对任意一个所有权的共识呢？所有的传统支付系统都依赖于一个中心认证机构，依靠中心机构提供的结算服务来验证并处理所有的交易。比特币没有中心机构，几乎所有的完整节点都有一份公共总帐的备份，这份总帐可以被视为认证过的记录。区块链并不是由一个中心机构创造的，它是由比特币网络中的所有节点各自独立竞争完成的。换句话说比特币网络中的所有节点，依靠着节点间的不稳定的网络连接所传输的信息，最终得出同样的结果并维护了同一个公共总帐。这一章将介绍比特币网络不依靠中心机构而达成共识的机制。

中本聪的主要发明就是这种去中心化的自发共识机制。这种自发，是指没有经过明确选举或者没有固定达成的共识的时间。换句话说，共识是数以千计的独立节点遵守了简单的规则通过异步交互自发形成的产物。所有的比特币属性，包括货币、交易、支付以及不依靠中心机构和信任的安全模型等都是这个机制的衍生物。比特币的去中心化共识由所有网络节点的4种独立过程相互作用而产生：

- ▷ 每个全节点依据综合标准对每个交易进行独立验证
- ▷ 通过完成工作量证明算法的验算，挖矿节点将交易记录独立打包进新区块，
- ▷ 每个节点独立的对新区块进行校验并组装进区块链
- ▷ 每个节点对区块链进行独立选择，在工作量证明机制下选择累计工作量最大的区块链

在接下来的几节中，我们将审视这些过程，了解它们之间如何相互作用并达成全网的自发共识，从而使任意节点组合出它自己的权威、可信、公开的总帐。

8.3 交易的独立校验

在第5章中，我们知道了钱包软件通过收集UTXO、提供正确的解锁脚本、构造支付给接收者的输出这一系列的方式来创建交易。产生的交易随后将被发送到比特币网络临近的节点，从而使得该交易能够在整个比特币网络中传播。

然而，在交易传递到临近的节点前，每一个收到交易的比特币节点将会首先验证该交易，这将确保只有有效的交易才会在网络中传播，而无效的交易将会在第一个节点处被废弃。

每一个节点在校验每一笔交易时，都需要对照一个长长的标准列表：

- ▷ 交易的语法和数据结构必须正确。
- ▷ 输入与输出列表都不能为空。
- ▷ 交易的字节大小是小于 `MAX_BLOCK_SIZE` 的。
- ▷ 每一个输出值，以及总量，必须在规定值的范围内（小于2,100万个币，大于0）。
- ▷ 没有哈希等于0，N等于-1的输入（coinbase交易不应当被中继）。
- ▷ `nLockTime` 是小于或等于 `INT_MAX` 的。
- ▷ 交易的字节大小是大于或等于100的。

- ▷交易中的签名数量应小于签名操作数量上限。
- ▷解锁脚本（`scriptSig`）只能够将数字压入栈中，并且锁定脚本（`scriptPubkey`）必须要符合`isStandard`的格式（该格式将会拒绝非标准交易）。
- ▷池中或位于主分支区块中的一个匹配交易必须是存在的。
- ▷对于每一个输入，如果引用的输出存在于池中任何的交易，该交易将被拒绝。
- ▷对于每一个输入，在主分支和交易池中寻找引用的输出交易。如果输出交易缺少任何一个输入，该交易将成为一个孤立的交易。如果与其匹配的交易还没有出现在池中，那么将被加入到孤立交易池中。
- ▷对于每一个输入，如果引用的输出交易是一个coinbase输出，该输入必须至少获得`COINBASE_MATURITY` (100)个确认。
- ▷对于每一个输入，引用的输出是必须存在的，并且没有被花费。
- ▷使用引用的输出交易获得输入值，并检查每一个输入值和总值是否在规定值的范围内（小于2100万个币，大于0）。
- ▷如果输入值的总和小于输出值的总和，交易将被中止。
- ▷如果交易费用太低以至于无法进入一个空的区块，交易将被拒绝。
- ▷每一个输入的解锁脚本必须依据相应输出的锁定脚本来验证。

这些条件能够在比特币标准客户端下的`AcceptToMemoryPool`、`CheckTransaction` 和 `CheckInputs` 函数中获得更详细的阐述。请注意，这些条件会随着时间发生变化，为了处理新型拒绝服务攻击，有时候也为交易类型多样化而放宽规则。

在收到交易后，每一个节点都会在全网广播前对这些交易进行校验，并以接收时的相应顺序，为有效的新交易建立一个池（交易池）。

8.4 挖矿节点

在比特币网络中，一些节点被称为专业节点矿工。第1章中，我们介绍了Jing，在中国上海的计算机工程专业学生，他就是一位矿工。Jing通过矿机挖矿获得比特币，矿机是专门设计用于挖比特币的计算机硬件系统。Jing的这台专业挖矿设备连接着一个运行完整比特币节点的服务器。与Jing不同，一些矿工是在没有完整节点的条件下进行挖矿，正如我们在“[8.11.2 矿池](#)”一节中所述的。与其他任一完整节点相同，Jing的节点在比特币网络中进行接收和传播未确认交易记录。然而，Jing的节点也能够在新区块中整合这些交易记录。

同其他节点一样，Jing的节点时刻监听着传播到比特币网络的新区块。而这些新加入的区块对挖矿节点有着特殊的意义。矿工间的竞争以新区块的传播而结束，如同宣布谁是最后的赢家。对于矿工们来说，获得一个新区块意味着某个参与者赢了，而他们则输了这场竞争。然而，一轮竞争的结束也代表着下一轮竞争的开始。新区块并不仅仅是象征着竞赛结束的方格旗；它也是下一个区块竞赛的发令枪。

8.5 整合交易至区块

验证交易后，比特币节点会将这些交易添加到自己的内存池中。内存池也称作交易池，用来暂存尚未被加入到区块的交易记录。与其他节点一样，Jing的节点会收集、验证并中继新的交易。而与其他节点不同的是，Jing的节点会把这些交易整合到一个候选区块中。

让我们继续跟进，看下Alice从Bob咖啡店购买咖啡时产生的那个区块（参见“[2.1.2 买咖啡](#)”）。Alice的交易在区块277,316。为了演示本章中提到的概念，我们假设这个区块是由Jing的挖矿系统挖出的，并且继续跟进Alice的交易，因为这个交易已经成为了新区块的一部分。

Jing的挖矿节点维护了一个区块链的本地副本，包含了自2009年比特币系统启动运行以来的全部区块。当Alice买咖啡的时候，Jing节点的区块链已经收集到了区块277,314，并继续监听着网络上的交易，在尝试挖掘新区块的同时，也监听着由其他节点发现的区块。当Jing的节点在挖矿时，它从比特币网络收到了区块277,315。这个区块的到来标志着终结了产出区块277,315竞赛，与此同时也是产出区块277,316竞赛的开始。

在上一个10分钟内，当Jing的节点正在寻找区块277,315的解的同时，它也在收集交易记录为下一个区块做准备。目前它已经收到了几百笔交易记录，并将它们放进了内存池。直到接收并验证区块277,315后，Jing的节点会检查内存池中的全部交易，并移除已经在区块277,315中出现过的交易记录，确保任何留在内存池中的交易都是未确认的，等待被记录到新区块中。

Jing的节点立刻构建一个新的空区块，做为区块277,316的候选区块。称作候选区块是因为它还没有包含有效的工作量证明，不是一个有效的区块，而只有在矿工成功找到一个工作量证明解之后，这个区块才生效。

8.5.1 交易块龄，矿工费和优先级

Jing的比特币节点需要为内存池中的每笔交易分配一个优先级，并选择较高优先级的交易记录来构建候选区块。交易的优先级是由交易输入所花费的UTXO的“块龄”决定，交易输入值高、“块龄”大的交易比那些新的、输入值小的交易拥有更高的优先级。如果区块中有足够的空间，高优先级的交易行为将不需要矿工费。

交易的优先级是通过输入值和输入的“块龄”乘积之和除以交易的总长度得到的：

```
Priority = Sum (Value of input * Input Age) / Transaction Size
```

在这个等式中，交易输入的值是由比特币单位“聪”（100万分之1个比特币）来表示的。UTXO的“块龄”是自该UTXO被记录到区块链为止所经历过的区块数，即这个UTXO在区块链中的深度。交易记录的大小由字节来表示。

一个交易想要成为“较高优先级”，需满足的条件：优先值大于57,600,000，相当于一个比特币（即100万聪），年龄为一天（144个区块），交易的大小为250个字节：

```
High Priority > 100,000,000 satoshis * 144 blocks / 250 bytes = 57,600,000
```

区块中用来存储交易的前50K字节是保留给较高优先级交易的。Jing的节点在填充这50K字节的时候，会优先考虑这些最高优先级的交易，不管它们是否包含了矿工费。这种机制使得高优先级交易即便是零矿工费，也可以优先被处理。

然后，Jing的挖矿节点会选出那些包含最小矿工费的交易，并按照“每千字节矿工费”进行排序，优先选择矿工费高的交易来填充剩下的区块，区块大小上限为 `MAX_BLOCK_SIZE`。

如区块中仍有剩余空间，Jing的挖矿节点可以选择那些不含矿工费的交易。有些矿工会竭尽全力将那些不含矿工费的交易整合到区块中，而其他矿工也许会选择忽略这些交易。

在区块被填满后，内存池中的剩余交易会成为下一个区块的候选交易。因为这些交易还留在内存池中，所以随着新的区块被加到链上，这些交易输入时所引用UTXO的深度（即交易“块龄”）也会随着变大。由于交易的优先值取决于它交易输入的“块龄”，所以这个交易的优先值也就随之增长了。最后，一个零矿工费交易的优先值就有可能会满足高优先级的门槛，被免费地打包进区块。

比特币交易中没有过期、超时的概念，一笔交易现在有效，那么它就永远有效。然而，如果一笔交易只在全网广播了一次，那么它只会保存在一个挖矿节点的内存中。因为内存池是以未持久化的方式保存在挖矿节点存储器中的，所以一旦这个节点重新启动，内存池中的数据就会被完全擦除。而且，即便一笔有效交易被传播到了全网，如果它长时间未处理，它将从挖矿节点的内存池中消失。如果交易本应该在一段时间内被处理而实际没有，那么钱包软件应该重新发送交易或重新支付更高的矿工费。

现在，Jing的节点从内存池中整合到了全部的交易，新的候选区块包含有418笔交易，总的矿工费为0.09094925个比特币。你可以通过比特币核心客户端命令行来查看这个区块，如例8-3所示：

例8-3 区块277,316

```
{
  "hash" : "0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
  "confirmations" : 35561,
  "size" : 218629,
  "height" : 277316,
  "version" : 2,
  "merkleroot" :
  "c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",
```

8.5.2 创币交易

区块中的第一笔交易是笔特殊交易，称为创币交易或者coinbase交易。这个交易是由Jing的节点构造并用来奖励矿工们所做的贡献的。Jing的节点会创建“向Jing的地址支付25.09094928个比特币”这样一个交易，把生成交易的奖励发送到自己的钱包。Jing挖出区块获得的奖励金额是coinbase奖励（25个全新的比特币）和区块中全部交易矿工费的总和。如例8-4所示：

```
$ bitcoin-cli getrawtransaction  
d5ada064c6417ca25c4308bd158b34b77e1c0eca2a73cda16c737e7424afba2f 1
```

例8-4 创市交易

与常规交易不同，创币交易没有输入，不消耗UTXO。它只包含一个被称作coinbase的输入，仅仅用来创建新的比特币。创币交易有一个输出，支付到这个矿工的比特币地址。创币交易的输出将这25.09094928个比特币发送到矿工的比特币地址，如本例所示的1MxTkeFP2PmHSMze5tUZ1hAV3YTKU2Gh1N。

8.5.3 Coinbase奖励与矿工费

为了构造创币交易，Jing的节点需要计算矿工费的总额，将这418个已添加到区块交易的输入和输出分别进行加总，然后用输入总额减去输出总额得到矿工费总额，公式如下：

```
Total Fees = Sum(Inputs) - Sum(Outputs)
```

在区块277,316中，矿工费的总额是0.09094925个比特币。

紧接着，Jing的节点计算出这个新区块正确的奖励额。奖励额的计算是基于区块高度的，以每个区块50个比特币为开始，每产生210,000个区块减半一次。这个区块高度是277,316，所以正确的奖励额是25个比特币。

详细的计算过程可以参看比特币核心客户端中的GetBlockValue函数，如例8-5所示：

例8-5 计算区块奖励—Function GetBlockValue, Bitcoin Core Client, main.cpp, line 1305

```
int64_t GetBlockValue(int nHeight, int64_t nFees)
{
    int64_t nSubsidy = 50 * COIN;
    int halvings = nHeight / Params().SubsidyHalvingInterval();

    // 如果右移的次数未定义，区块奖励强制为零
    if (halvings >= 64)
        return nFees;

    // Subsidy每210,000个区块减半一次，大概每4年发生一次
    nSubsidy >>= halvings;

    return nSubsidy + nFees;
}
```

变量nSubsidy表示初始奖励额，值为COIN常量（100,000,000聪）与50的乘积，也就是说初始奖励额为50亿聪。

紧接着，这个函数用当前区块高度除以减半间隔（SubsidyHalvingInterval函数）得到减半次数（变量halvings）。每210,000个区块为一个减半间隔，对应本例中的区块277316，所以减半次数为1。

变量halvings最大值64，如果超出这个值，代码算得的奖励额为0，整个函数将只返回矿工费总额，作为奖励总额。

然后，这个函数会使用二进制右移操作将奖励额（变量nSubsidy）右移一位（等同于除以2），每一轮减半右移一次。在这个例子中，对于区块277,316只需要将值为50亿聪的奖励额右移一次，得到25亿聪，也就是25个比特币的奖励额。之所以采用二进制右移操作，是因为相比于整数或浮点数除法，右移操作的效率更高。

最后，将coinbase奖励额（变量nSubsidy）与矿工费（nFee）总额求和，并返回这个值。

8.5.4 创币交易的结构

经过计算，Jing的节点构造了一个创币交易，支付给自己25.09094928枚比特币。

如例8-4所示，创币交易的结构比较特殊，与一般交易输入需要指定一个先前的UTXO不同，它包含一个“coinbase”输入。在表5-3中，我们已经给出了交易输入的结构。现在让我们来比较一下常规交易输入与创币交易输入。表8-1给出了常规交易输入的结构，表8-2给出的是创币交易输入的结构。

表8-1 “普通”交易输入的结构

长度	字段	描述
32字节	交易哈希	指向包含有将要被花费UTXO的交易

4 字节	交易输出索引	UTXO在交易中的索引，0从0开始计数
1-9 字节	解锁脚本长度	解锁脚本的长度
(VarInt) 可变长度	Unlocking-Script	一段脚本，用来解锁UTXO锁定脚本中的条件
4 bytes	顺序号	当前未启用的TX替换功能，设置为0xFFFFFFFF

表8-2 生成交易输入的结构

长度	字段	描述
32 字节	交易哈希	不引用任何一个交易，值全部为0
4 字节	交易输出索引	值全部为1
1-9 字节	Coinbase数据长度	coinbase数据长度
(VarInt) 可变长度	Coinbase数据	在v2版本的区块中，除了需要以区块高度开始外，其他数据可以任意填写，用于extra nonce和挖矿标签
4 bytes	顺序号	值全部为1，0xFFFFFFFF

在创币交易中，“交易哈希”字段32个字节全部填充0，“交易输出索引”字段全部填充0xFF(十进制的255)，这两个字段的值表示不引用UTXO。“解锁脚本”由coinbase数据代替，数据可以由矿工自定义。

8.5.5 Coinbase数据

创币交易不包含“解锁脚本”(又称作 `scriptSig`)字段，这个字段被coinbase数据替代，长度最小2字节，最大100字节。除了开始的几个字节外，矿工可以任意使用coinbase的其他部分，随意填充任何数据。

以创世块为例，中本聪在coinbase中填入了这样的数据“The Times 03/Jan/ 2009 Chancellor on brink of second bailout for banks”(泰晤士报 2009年1月3日 财政大臣将再次对银行施以援手)，表示对日期的证明，同时也表达了对银行系统的不信任。现在，矿工使用coinbase数据实现extra nonce功能，并嵌入字符串来标识挖出它的矿池，这部分内容会在后面的小节描述。coinbase前几个字节也曾是可以任意填写的，不过在后来的第34号比特币改进提议(BIP34)中规定了版本2的区块(版本字段为2的区块)，这个区块的高度必须跟在脚本操作“push”之后，填充在coinbase字段的起始处。

我们以例8-4中的区块277,316为例，coinbase就是交易输入的“解锁脚本”(或`scriptSig`)字段，这个字段的十六进制值为03443b0403858402062f503253482f。下面让我们来解码这段数据。

第一个字节是03，脚本执行引擎执行这个指令将后面3个字节压入脚本栈(见表4-1)，紧接着的3个字节——0x443b04，是以小端格式(最低有效字节在先)编码的区块高度。翻转字节序得到0x043b44，表示为十进制是277,316。

紧接着的几个十六进制数(03858402062)用于编码extra nonce(参见“[8.11.1 随机值升位方案](#)”)，或者一个随机值，从而求解一个适当的工作量证明。

coinbase数据结尾部分(2f503253482f)是ASCII编码字符 /P2SH/，表示挖出这个区块的挖矿节点支持BIP0016所定义的pay-to-script-hash(P2SH)改进方案。在P2SH功能引入到比特币的时候，曾经有过一场对P2SH不同实现方式的投票，候选者是BIP0016和BIP0017。支持BIP0016的矿工将/P2SH/放入coinbase数据中，支持BIP0017的矿工将 p2sh/CHV放入他们的coinbase数据中。最后，BIP0016在选举中胜出，直到现在依然有很多矿工在他们的coinbase中填入/P2SH/以表示支持这个功能。

例8-6使用了libbitcoin库(在56页“其他替代客户端、资料库、工具包”中提到)从创世块中提取coinbase数据，并显示出中本聪留下的信息。libbitcoin库中自带了一份创世块的静态拷贝，所以这段示例代码可以直接取自库中的创世块数据。

例8-6 从创世区块中提取coinbase数据

```

/*
Display the genesis block message by Satoshi.
*/
#include <iostream>
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Create genesis block.
    bc::block_type block = bc::genesis_block();
    // Genesis block contains a single coinbase transaction. assert(block.transactions.size() == 1);
    // Get first transaction in block (coinbase).
    const bc::transaction_type& coinbase_tx = block.transactions[0];
    // Coinbase tx has a single input.
    assert(coinbase_tx.inputs.size() == 1);
    const bc::transaction_input_type& coinbase_input = coinbase_tx.inputs[0];
    // Convert the input script to its raw format.
    const bc::data_chunk& raw_message = save_script(coinbase_input.script);
    // Convert this to an std::string.
    std::string message;
    message.resize(raw_message.size());
    std::copy(raw_message.begin(), raw_message.end(), message.begin());
    // Display the genesis block message.
    std::cout << message << std::endl;
    return 0;
}

```

在例8-7中，我们使用GNU C++编译器编译源代码并运行得到的可执行文件

例8-7 编译并运行satoshi-words示例代码

```

$ # Compile the code
$ g++ -o satoshi-words satoshi-words.cpp $(pkg-config --cflags --libs libbitcoin) $ # Run the executable
$ ./satoshi-words
^D@<GS>^A^DEThe Times 03/Jan/2009 Chancellor on brink of second bailout for banks

```

8.6 构造区块头

为了构造区块头，挖矿节点需要填充六个字段，如表8-3中所示。

表8-3 区块头的结构

长度	字段	描述
4 字节	版本	版本号，用来跟踪软件或协议的升级
32 字节	前区块哈希	链中前一个区块（父区块）的哈希值
32 字节	Merkle根	一个哈希值，表示这个区块中全部交易构成的merkle树的根
4 字节	时间戳	以Unix纪元开始到当下秒数记录的区块生成的时刻
4 bytes	难度目标	该区块的工作量证明算法难度目标
4 bytes	Nonce	一个用于工作量证明算法的计数器

在区块277,316被挖出的时候，区块结构中用来表示版本号的字段值为2，长度为4字节，以小段格式编码值为0x20000000。接着，挖矿节点需要填充“前区块哈希”，在本例中，这个值为Jing的节点从网络上接收到的区块277,315的区块头哈希值，它是区块277316候选区块的父区块。区块277,315的区块头哈希值为：

```
0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

为了向区块头填充merkle根字段，要将全部的交易组成一个merkle树。创币交易作为区块中的首个交易，后将余下的418笔交易添至其后，这样区块中的交易一共有419笔。在164页，我们已经见到过“Merkle树”，树中必须有偶数个叶子节点，所以需要复制最后一个交易作为第420个节点，每个节点是对应交易的哈希值。这些交易的哈希值逐层地、成对地组合，直到最终组合并成一个根节点。merkle树的根节点将全部交易数据摘要为一个32字节长度的值，例8-3中merkel根的值如下：

```
c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e
```

挖矿节点会继续添加一个4字节的时间戳，以Unix纪元时间编码，即自1970年1月1日0点到当下总共流逝的秒数。本例中的1388185914对应的时间是2013年12月27日，星期五，UTC/GMT。

接下来，节点需要填充难度目标值，为了使得该区块有效，这个字段定义了所需满足的工作量证明的难度。难度在区块中以“尾数-指数”的格式，编码并存储，这种格式称作“难度位”。这种编码的首字节表示指数，后面的3字节表示尾数(系数)。以区块277316为例，难度位的值为0x1903a30c，0x19是指数的十六进制格式，后半部0x03a30c是系数。这部分的概念在第195页的“难度目标与难度调整”和第194的“难度表示”有详细的解释。

最后一个字段是nonce，初始值为0。

区块头完成全部的字段填充后，挖矿就可以开始进行了。挖矿的目标是找到一个使区块头哈希值小于难度目标的nonce。挖矿节点通常需要尝试数十亿甚至数万亿个不同的nonce取值，直到找到一个满足条件的nonce值。

8.7 构建区块

既然Jing的节点已经构建了一个候选区块，那么就轮到Jing的矿机对这个新区块进行“挖掘”，求解工作量证明算法以使这个区块有效。从本书中我们已经学习了比特币系统中不同地方用到的哈希加密函数。比特币挖矿过程使用的是SHA256哈希函数。

用最简单的术语来说，挖矿就是重复计算区块头的哈希值，不断修改该参数，直到与哈希值匹配的一个过程。哈希函数的结果无法提前得知，也没有能得到一个特定哈希值的模式。哈希函数的这个特性意味着：得到哈希值的唯一方法是不断的尝试，每次随机修改输入，直到出现适当的哈希值。

8.7.1 工作量证明算法

哈希函数的输入数据的长度是任意的，将产生一个长度固定且绝不雷同的值，可将其视为输入的数字指纹。对于特定输入，哈希的结果每次都一样，任何实现相同哈希函数的人都可以计算和验证。一个加密哈希函数的主要特征就是不同的输入几乎不可能出现相同的数字指纹。因此，相对于随机选择输入，有意地选择输入去生成一个想要的哈希值几乎是不可能的。

无论输入的大小是多少，SHA256函数的输出的长度总是256bit。在例8-8中，我们将使用Python解释器来计算语句 "I am Satoshi Nakamoto" 的SHA256的哈希值。

例8-8 SHA256示例

```
$ python
Python 2.7.1
>>> import hashlib
>>> print hashlib.sha256("I am Satoshi Nakamoto").hexdigest() 5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e
```

在例8-8中，5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e 是"I am Satoshi Nakamoto"的哈希值。改变原句中的任何一个字母、标点、或增加字母都会产生不同的哈希值。

如果我们改变原句，得到的应该是完全不同的哈希值。例如，我们在句子末尾加上一个数字，运行例8-9中的Python脚本。

例8-9 通过迭代 nonce 来生成不同哈希值的脚本（SHA256）

```
# example of iterating a nonce in a hashing algorithm's input
import hashlib
text = "I am Satoshi Nakamoto"
# iterate nonce from 0 to 19
for nonce in xrange(20):
    # add the nonce to the end of the text
    input = text + str(nonce)
    # calculate the SHA-256 hash of the input (text+nonce)
    hash = hashlib.sha256(input).hexdigest() # show the input and hash result
    print input, '=>', hash
```

执行这个脚本就能生成这些只是末尾数字不同的语句的哈希值。例8-10中显示了我们只是增加了这个数字，却得到了非常不同的哈希值。

例8-10 通过迭代 nonce 来生成不同哈希值的脚本的输出

```
$ python hash_example.py

I am Satoshi Nakamoto0 => a80a81401765c8eddee25df36728d732...
I am Satoshi Nakamoto1 => f7bc9a63094a4647bb41241a677b5345f...
I am Satoshi Nakamoto2 => ea758a8134b115298a1583ffbf80ae629...
I am Satoshi Nakamoto3 => bfa9779618ff072c903d773de30c99bd...
I am Satoshi Nakamoto4 => bce8564de9a83c18c31944a66bde992f...
I am Satoshi Nakamoto5 => eb362c3cf3479be0a97a20163589038e...
I am Satoshi Nakamoto6 => 4a2fd48e3be420d0d28e202360cfbab...
I am Satoshi Nakamoto7 => 790b5a1349a5f2b909bf74d0d166b17a...
I am Satoshi Nakamoto8 => 702c45e5b15aa54b625d68dd947f1597...
I am Satoshi Nakamoto9 => 7007cf7dd40f5e933cd89fff5b791ff0...
I am Satoshi Nakamoto10 => c2f38c81992f4614206a21537bd634a...
I am Satoshi Nakamoto11 => 7045da6ed8a914690f087690e1e8d66...
I am Satoshi Nakamoto12 => 60f01db30c1a0d4cbce2b4b22e88b9b...
I am Satoshi Nakamoto13 => 0ebc56d59a34f5082aaef3d66b37a66...
I am Satoshi Nakamoto14 => 27ead1ca85da66981fd9da01a8c6816...
I am Satoshi Nakamoto15 => 394809fb089c5f83ce97ab554a2812c...
I am Satoshi Nakamoto16 => 8fa4992219df3f50834465d3047429...
I am Satoshi Nakamoto17 => dca8b4b4f8d8e1521fa4ea6f4fc0d...
I am Satoshi Nakamoto18 => 9989a401b2a3a318b01e9ca9a22b0f3...
I am Satoshi Nakamoto19 => cda56022ecb5b67b2bc93a2d764e75f...
```

每个语句都生成了一个完全不同的哈希值。它们看起来是完全随机的，但你在任何计算机上用Python执行上面的脚本都能重现这些完全相同的哈希值。

类似这样在语句末尾的变化的数字叫做nonce。Nonce是用来改变加密函数输出的，在这个示例中改变了这个语句的SHA256指纹。

为了使这个哈希算法变得富有挑战，我们来设定一个具有任意性的目标：找到一个语句，使之哈希值的十六进制表示以0开头。幸运的是，这很容易！在例8-10中语句 "I am Satoshi Nakamoto13" 的哈希值是

`0ebc56d59a34f5082aaef3d66b37a661696c2b618e62432727216ba9531041a5`，刚好满足条件。我们得到它用了13次。用概率的角度来看，如果哈希函数的输出是平均分布的，我们可以期望每16次得到一个以0开头的哈希值（十六进制每一位数字为0到F）。从数字的角度来看，我们要找的是小于

0x10000000000000000000000000000000的哈希值。我们称这个为目标阀值，我们的目的是找到一个小于这个目标的哈希值。如果我们减小这个目标值，那找到一个小于它的哈希值会越来越难。

简单打个比方，想象人们不断扔一对色子以得到小于一个特定点数的游戏。第一局，目标是12。只要你不扔出两个6，你就会赢。然后下一局目标为11。玩家只能扔10或更小的点数才能赢，不过也很简单。假如几局之后目标降低为了5。现在有一半机率以上扔出来的色子加起来点数会超过5，因此无效。随着目标越来越小，要想赢的话，扔色子的次数会指数级的上升。最终当目标为2时（最小可能点数），只有一个人平均扔36次或2%扔的次数中，他才能赢。

在例8-10中，成功的nonce为13，且这个结果能被所有人独立确认。任何人将13加到语句“*I am Satoshi Nakamoto*”后面再

计算哈希值都能确认它比目标值要小。这个正确的结果同时也是工作量证明（Proof of Work），因为它证明我们的确花时间找到了这个nonce。验证这个哈希值只需要一次计算，而我们找到它却花了13次。如果目标值更小（难度更大），那我们需要多得多的哈希计算才能找到合适的nonce，但其他人验证它时只需要一次哈希计算。此外，知道目标值后，任何人都可以用统计学来估算其难度，因此就能知道找到这个nonce需要多少工作。

比特币的工作量证明和例8-10中的挑战非常类似。矿工用一些交易构建一个候选区块。接下来，这个矿工计算这个区块头信息的哈希值，看其是否小于当前目标值。如果这个哈希值不小于目标值，矿工就会修改这个nonce（通常将之加1）然后再试一次。按当前比特币系统的难度，矿工得试 10^{15} 次（ 10 的 15 次方）才能找到一个合适的nonce使区块头信息哈希值足够小。

例8-11是一个简化很多的工作量证明算法的实现。

例8-11 简化的工作量证明算法

```
#!/usr/bin/env python
# example of proof-of-work algorithm

import hashlib
import time

max_nonce = 2 ** 32 # 4 billion

def proof_of_work(header, difficulty_bits):

    # calculate the difficulty target
    target = 2 ** (256-difficulty_bits)

    for nonce in xrange(max_nonce):
        hash_result = hashlib.sha256(str(header)+str(nonce)).hexdigest()

        # check if this is a valid result, below the target
        if long(hash_result, 16) < target:
            print "Success with nonce %d" % nonce
            print "Hash is %s" % hash_result
            return (hash_result,nonce)

    print "Failed after %d (max_nonce) tries" % nonce
    return nonce

if __name__ == '__main__':

    nonce = 0
    hash_result = ''

    # difficulty from 0 to 31 bits
    for difficulty_bits in xrange(32):

        difficulty = 2 ** difficulty_bits
        print "Difficulty: %ld (%d bits)" % (difficulty, difficulty_bits)

        print "Starting search..."

        # checkpoint the current time
        start_time = time.time()

        # make a new block which includes the hash from the previous block
        # we fake a block of transactions - just a string
        new_block = 'test block with transactions' + hash_result

        # find a valid nonce for the new block
        (hash_result, nonce) = proof_of_work(new_block, difficulty_bits)

        # checkpoint how long it took to find a result
        end_time = time.time()

        elapsed_time = end_time - start_time
        print "Elapsed Time: %.4f seconds" % elapsed_time

        if elapsed_time > 0:
```

```
# estimate the hashes per second
hash_power = float(long(nonce)/elapsed_time)
print "Hashing Power: %ld hashes per second" % hash_power
```

你可以任意调整难度值（按二进制bit数来设定，即哈希值开头多少个bit必须是0）。然后执行代码，看看在你的计算机上求解需要多久。在例8-12中，你可以看到该程序在一个普通笔记本电脑上的执行情况。

例8-12 多种难度值的工作量证明算法的运行输出

```
$ python proof-of-work-example.py*
Difficulty: 1 (0 bits)

[...]

Difficulty: 8 (3 bits)
Starting search...
Success with nonce 9
Hash is 1c1c105e65b47142f028a8f93ddf3dabb9260491bc64474738133ce5256cb3c1
Elapsed Time: 0.0004 seconds
Hashing Power: 25065 hashes per second
Difficulty: 16 (4 bits)
Starting search...
Success with nonce 25
Hash is 0f7becfd3bcd1a82e06663c97176add89e7cae0268de46f94e7e11bc3863e148
Elapsed Time: 0.0005 seconds
Hashing Power: 52507 hashes per second
Difficulty: 32 (5 bits)
Starting search...
Success with nonce 36
Hash is 029ae6e5004302a120630adcbb808452346ab1cf0b94c5189ba8bac1d47e7903
Elapsed Time: 0.0006 seconds
Hashing Power: 58164 hashes per second

[...]

Difficulty: 4194304 (22 bits)
Starting search...
Success with nonce 1759164
Hash is 0000008bb8f0e731f0496b8e530da984e85fb3cd2bd81882fe8ba3610b6cef3
Elapsed Time: 13.3201 seconds
Hashing Power: 132068 hashes per second
Difficulty: 8388608 (23 bits)
Starting search...
Success with nonce 14214729
Hash is 000001408cf12dbd20fcba6372a223e098d58786c6ff93488a9f74f5df4df0a3
Elapsed Time: 110.1507 seconds
Hashing Power: 129048 hashes per second
Difficulty: 16777216 (24 bits)
Starting search...
Success with nonce 24586379
Hash is 0000002c3d6b370fccd699708d1b7cb4a94388595171366b944d68b2acce8b95
Elapsed Time: 195.2991 seconds
Hashing Power: 125890 hashes per second

[...]

Difficulty: 67108864 (26 bits)
Starting search...
Success with nonce 84561291
Hash is 0000001f0ea21e676b6dde5ad429b9d131a9f2b000802ab2f169cbca22b1e21a
Elapsed Time: 665.0949 seconds
Hashing Power: 127141 hashes per second
```

你可以看出，随着难度位一位一位地增加，查找正确结果的时间会呈指数级增长。如果你考虑整个256bit数字空间，每次要求多一个0，你就把哈希查找空间缩减了一半。在例8-12中，为寻找一个nonce使得哈希值开头的26位值为0，一共尝试了8千多万次。即使家用笔记本每秒可以达270,000多次哈希计算，这个查找依然需要6分钟。

在写这本书的时候，比特币网络要寻找区块头信息哈希值小于

0000000000000004c296e6376db3a241271f43fd3f5de7ba18986e517a243baa7。可以看出，这个目标哈希值开头的0多了很多。这意味着可接受的哈希值范围大幅缩减，因而找到正确的哈希值更加困难。生成下一个区块需要网络每秒计算 1.5×10^{17} 次哈希。这看起来像是不可能的任务，但幸运的是比特币网络已经拥有100PH每秒（petahashes per second, peta-为 10^{15} ）的处理能力，平均每10分钟就可以找到一个新区块。

8.7.2 难度表示

在例8-3中，我们在区块中看到难度目标，其被称“难度位”或简称“bits”。在区块277,316中，它的值为0x1903a30c。这个标记的值被存为系数/指数格式，前两位十六进制数字为幂，接下来得六位为系数。在这个区块里，0x19为幂，而0x03a30c为系数。

计算难度目标的公式为：

```
target = coefficient * 2^(8 * (exponent - 3))
```

由此公式及难度位的值 0x1903a30c，可得：

```
target = 0x03a30c * 2^(0x08 * (0x19 - 0x03))

=> target = 0x03a30c * 2^(0x08 * 0x16)

=> target = 0x03a30c * 2^0xB0
```

按十进制计算为：

```
=> target = 238,348 * 2^176  
=> target =  
22,829,202,948,393,929,850,749,706,076,701,368,331,072,452,018,388,575,715,328
```

转化为十六进制后为：

也就是说高度为277,316的有效区块的头信息哈希值是小于这个目标值的。这个数字的二进制表示中前60位都是0。在这个难度上，一个每秒可以处理1万亿个哈希计算的矿工（1 tera-hash per second 或 1 TH/sec）平均每8,496个区块才能找到一个正确结果，换句话说，平均每59天，才能为某一个区块找到正确的哈希值。

8.7.3 难度目标与难度调整

如前所述，目标决定了难度，进而影响求解工作量证明算法所需要的时间。那么问题来了：为什么这个难度值是可调整的？由谁来调整？如何调整？

比特币的区块平均每10分钟生成一个。这就是比特币的心跳，是货币发行速率和交易达成速度的基础。不仅是在短期内，而是在几十年内它都必须要保持恒定。在此期间，计算机性能将飞速提升。此外，参与挖矿的人和计算机也会不断变化。为了能让新区块的保持10分钟一个的产生速率，挖矿的难度必须根据这些变化进行调整。事实上，难度是一个动态的参数，会定期调整以达到每10分钟一个新区块的目标。简单地说，难度被设定在，无论挖矿能力如何，新区块产生速率都保持在10分钟一个。

那么，在一个完全去中心化的网络中，这样的调整是如何做到的呢？难度的调整是在每个完整节点中独立自动发生的。每2,016个区块中的所有节点都会调整难度。难度的调整公式是由最新2,016个区块的花费时长与20,160分钟（两周，即这些区块以10分钟一个速率所期望花费的时长）比较得出的。难度是根据实际时长与期望时长的比值进行相应调整的（或变难或变

易）。简单来说，如果网络发现区块产生速率比10分钟要快时会增加难度。如果发现比10分钟慢时则降低难度。

这个公式可以总结为如下形式：

```
New Difficulty = Old Difficulty * (Actual Time of Last 2016 Blocks / 20160 minutes)
```

例8-13展示了比特币核心客户端中的难度调整代码。

例8-13 工作量证明的难度调整 源文件 pow.cpp 第43行函数 GetNextWorkRequired()

```
// Go back by what we want to be 14 days worth of blocks
const CBlockIndex* pindexFirst = pindexLast;
for (int i = 0; pindexFirst && i < Params().Interval()-1; i++)
    pindexFirst = pindexFirst->pprev;
assert(pindexFirst);
// Limit adjustment step
int64_t nActualTimespan = pindexLast->GetBlockTime() - pindexFirst->GetBlockTime(); LogPrintf(" nActualTimespan = %d before bound\n"
if (nActualTimespan < Params().TargetTimespan()/4)
    nActualTimespan = Params().TargetTimespan()/4;
if (nActualTimespan > Params().TargetTimespan()*4)
    nActualTimespan = Params().TargetTimespan()*4;

// Retarget
uint256 bnNew;
uint256 bnOld;
bnNew.SetCompact(pindexLast->nBits);
bnOld = bnNew;
bnNew *= nActualTimespan;
bnNew /= Params().TargetTimespan();

if (bnNew > Params().ProofOfWorkLimit())
    bnNew = Params().ProofOfWorkLimit();
```

参数Interval(2,016区块)和TergetTimespan(1,209,600秒及两周) 的定义在文件chainparams.cpp中。

为了防止难度的变化过快，每个周期的调整幅度必须小于一个因子（值为4）。如果要调整的幅度大于4倍，则按4倍调整。由于在下一个2,016区块的周期不平衡的情况会继续存在，所以进一步的难度调整会在下一周期进行。因此平衡哈希计算能力和难度的巨大差异有可能需要花费几个2,016区块周期才会完成。



寻找一个比特币区块需要整个网络花费10分钟来处理，每发现2,016个区块时会根据前2,016个区块完成的时间对难度进行调整。

值得注意的是目标难度与交易的数量和金额无关。这意味着哈希算力的强弱，即让比特币更安全的电力投入量，与交易的数量完全无关。换句话说，当比特币的规模变得更大，使用它的人数更多时，即使哈希算力保持当前的水平，比特币的安全性也不会受到影响。哈希算力的增加表明更多的人为得到比特币回报而加入了挖矿队伍。只要为了回报，公平正当地从事挖矿的矿工群体保持足够的哈希算力，“接管”攻击就不会得逞，让比特币的安全无虞。

目标难度和挖矿电力消耗与将比特币兑换成现金以支付这些电力之间的关系密切相关。高性能挖矿系统就是要用当前硅芯片以最高效的方式将电力转化为哈希算力。挖矿市场的关键因素就是每度电转换为比特币后的价格。因为这决定着挖矿活动的营利性，也因此刺激着人们选择进入或退出挖矿市场。

8.8 成功构建区块

前面已经看到，Jing的节点创建了一个候选区块，准备拿它来挖矿。Jing有几个安装了ASIC（专用集成电路）的矿机，上面有成千上万个集成电路可以超高速地并行运行SHA256算法。这些定制的硬件通过USB连接到他的挖矿节点上。接下来，运行在Jing的桌面电脑上的挖矿节点将区块头信息传送给这些硬件，让它们以每秒亿万次的速度进行nonce测试。

在对区块277,316的挖矿工作开始大概11分钟后，这些硬件里的其中一个求得了解并发回挖矿节点。当把这个结果放进区块头时，nonce 4,215,469,401 就会产生一个区块哈希值：

00000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569

而这个值小于难度目标值：

Jing的挖矿节点立刻将这个区块发给它的所有相邻节点。这些节点在接收并验证这个新区块后，也会继续传播此区块。当这个新区块在网络中扩散时，每个节点都会将它作为区块277,316加到自身节点的区块链副本中。当挖矿节点收到并验证了这个新区块后，它们会放弃之前对构建这个相同高度区块的计算，并立即开始计算区块链中下一个区块的工作。

下节将介绍节点进行区块验证、最长链选择、达成共识，并以此形成一个去中心化区块链的过程。

8.9 校验新区块

比特币共识机制的第三步是通过网络中的每个节点独立校验每个新区块。当新区块在网络中传播时，每一个节点在将它转发到其节点之前，会进行一系列的测试去验证它。这确保了只有有效的区块会在网络中传播。独立校验还确保了诚实的矿工生成的区块可以被纳入到区块链中，从而获得奖励。行为不诚实的矿工所产生的区块将被拒绝，这不但使他们失去了奖励，而且也浪费了本来可以去寻找工作量证明解的机会，因而导致其电费亏损。

当一个节点接收到一个新的区块，它将对照一个长长的标准清单对该区块进行验证，若没有通过验证，这个区块将被拒绝。这些标准可以在比特币核心客户端的CheckBlock函数和CheckBlockHead函数中获得，它包括：

- ▷ 区块的数据结构语法上有效
 - ▷ 区块头的哈希值小于目标难度（确认包含足够的工作量证明）
 - ▷ 区块时间戳早于验证时刻未来两个小时（允许时间错误）
 - ▷ 区块大小在长度限制之内
 - ▷ 第一个交易（且只有第一个）是coinbase交易
 - ▷ 使用检查清单验证区块内的交易并确保它们的有效性，本书177页
 - ▷ “交易的独立校验”一节已经讨论过这个清单。

每一个节点对每一个新区块的独立校验，确保了矿工无法欺诈。在前面的章节中，我们看到了矿工们如何去记录一笔交易，以获得在此区块中创造的新比特币和交易费。为什么矿工不为他们自己记录一笔交易去获得数以千计的比特币？这是因为每一个节点根据相同的规则对区块进行校验。一个无效的coinbase交易将使整个区块无效，这将导致该区块被拒绝，因此，该交易就不会成为总账的一部分。矿工们必须构建一个完美的区块，基于所有节点共享的规则，并且根据正确工作量证明的解决方案进行挖矿，他们要花费大量的电力挖矿才能做到这一点。如果他们作弊，所有的电力和努力都会浪费。这就是为什么独立校验是去中心化共识的重要组成部分。

8.10 区块链的组装与选择

比特币去中心化的共识机制的最后一步是将区块集合至有最大工作量证明的链中。一旦一个节点验证了一个新的区块，它将尝试将新的区块连接到到现存的区块链，将它们组装起来。

节点维护三种区块：第一种是连接到主链上的，第二种是从主链上产生分支的（备用链），最后一种是在已知链中没有找到已知父区块的。在验证过程中，一旦发现有不符合标准的地方，验证就会失败，这样区块会被节点拒绝，所以也不会加入到任何一条链中。

任何时候，主链都是累计了最多难度的区块链。在一般情况下，主链也是包含最多区块的那个链，除非有两个等长的链并且其中一个有更多的工作量证明。主链也会有一些分支，这些分支中的区块与主链上的区块互为“兄弟”区块。这些区块是有效的，但不是主链的一部分。保留这些分支的目的是如果在未来的某个时刻它们中的一个延长了并在难度值上超过了主链，那么后续的区块就会引用它们。在[“8.10.1 区块链分叉”](#)，我们将会看到在同样的区块高度，几乎同时挖出区块时，候选链是如何产生的。

当节点接收到新区块，它会尝试将这个区块插入到现有区块链中。节点会看一下这个区块的“previous block hash”字段，这个字段是该区块对其父区块的引用。同时，新的节点将尝试在已存在的区块链中找出这个父区块。大多数情况下，父区块是主块链的“顶点”，这就意味着这个新的区块延长了主链。举个例子，一个新的区块——区块277,316引用了它的父区块——区块277,315。大部分收到了区块277,316的节点将区块277,315作为主链的顶点，连接这个新区块并延长区块链。

有时候，新区块所延长的区块链并不是主链，这一点我们将在[“8.10.1 区块链分叉”](#)中看到。在这种情况下，节点将新的区块添加到备用链，同时比较备用链与主链的难度。如果备用链比主链积累了更多的难度，节点将收敛于备用链，意味着节点将选择备用链作为其新的主链，而之前那个老的主链则成为了备用链。如果节点是一个矿工，它将开始构造新的区块，来延长这个更新更长的区块链。

如果节点收到了一个有效的区块，而在现有的区块链中却未找到它的父区块，那么这个区块被认为是“孤块”。孤块会被保存在孤块池中，直到它们的父区块被节点收到。一旦收到了父区块并且将其连接到现有区块链上，节点就会将孤块从孤块池中取出，并且连接到它的父区块，让它作为区块链的一部分。当两个区块在很短的时间间隔内被挖出来，节点有可能会以相反的顺序接收到它们，这个时候孤块现象就会出现。

选择了最大难度的区块链后，所有的节点最终在全网范围内达成共识。随着更多的工作量证明被添加到链中，链的暂时性差异最终会得到解决。挖矿节点通过“投票”来选择它们想要延长的区块链，当它们挖出一个新块并且延长了一个链，新块本身就代表它们的投票。

相互竞争的链之间是存在差异的，下节我们将看到节点是怎样通过独立选择最长难度链来解决这种差异的。

8.10.1 区块链分叉

因为区块链是去中心化的数据结构，所以不同副本之间不能总是保持一致。区块有可能在不同时间到达不同节点，导致节点有不同的区块链视角。解决的办法是，每一个节点总是选择并尝试延长代表累计了最大工作量证明的区块链，也就是最长的或最大累计难度的链。节点通过将记录在每个区块中的难度加总起来，得到建立这个链所要付出的工作量证明的总量。只要所有的节点选择最长累计难度的区块链，整个比特币网络最终会收敛到一致的状态。分叉即在不同区块链间发生的临时差异，当更多的区块添加到了某个分叉中，这个问题便会迎刃而解。

在下面的图例中，我们可以了解网络中发生分叉的过程。图例代表简单的全球比特币网络，在真实的情况下，比特币网络的拓扑结构不是基于地理位置组织起来的。相反，在同一个网络中相互连接的节点，可能在地理位置上相距遥远，我们采用基于地理的拓扑是为了更加简洁地描述分叉。在真实比特币网络里，节点间的距离按“跳”而不是按照真实位置来衡量。为了便于描述，不同的区块被标示为不同的颜色，传播这些区块的节点网络也被标上颜色。

在第一张图（图8-2）中，网络有一个统一的区块链视角，以蓝色区块为主链的“顶点”。



图8-2 形象化的区块链分叉事件——分叉之前

当有两个候选区块同时想要延长最长区块链时，分叉事件就会发生。正常情况下，分叉发生在两名矿工在较短的时间内，各自都算得了工作量证明解的时候。两个矿工在各自的候选区块一发现解，便立即传播自己的“获胜”区块到网络中，先是传播给邻近的节点而后传播到整个网络。每个收到有效区块的节点都会将其并入并延长区块链。如果该节点在随后又收到了另一个候选区块，而这个区块又拥有同样父区块，那么节点会将这个区块连接到候选链上。其结果是，一些节点收到了一个候选区块，而另一些节点收到了另一个候选区块，这时两个不同版本的区块链就出现了。

在图8-3中，我们看到两个矿工几乎同时挖到了两个不同的区块。这两个区块是顶点区块——蓝色区块的子区块，可以延长这个区块链。为了便于跟踪这个分叉事件，我们设定有一个被标记为红色的、来自加拿大的区块，还有一个被标记为绿色的、来自澳大利亚的区块。



图8-3 形象化的区块链分叉事件：同时发现两个区块

假设有这样一种情况，一个在加拿大的矿工发现了“红色”区块的工作量证明解，在“蓝色”的父区块上延长了块链。几乎同一时刻，一个澳大利亚的矿工找到了“绿色”区块的解，也延长了“蓝色”区块。那么现在我们就有了两个区块：一个是源于加拿大的“红色”区块；另一个是源于澳大利亚的“绿色”。这两个区块都是有效的，均包含有效的工作量证明解并延长同一个父区块。这个两个区块可能包含了几乎相同的交易，只是在交易的排序上有些许不同。

当这两个区块传播时，一些节点首先收到“红色”区块，一些节点收到“绿色”区块。如图8-4所示，比特币网络上的节点对于

区块链的顶点产生了分歧，一派以红色区块为顶点，而另一派以绿色区块为顶点。



图8-4 形象化的区块链分叉事件：两个区块的传播将网络分裂了

从那时起，比特币网络中邻近（网络拓扑上的邻近，而非地理上的）加拿大的节点会首先收到“红色”区块，并建立一个最大累计难度的区块，“红色”区块为这个链的最后一个区块（蓝色-红色），同时忽略晚一些到达的“绿色”区块。相比之下，离澳大利亚更近的节点会判定“绿色”区块胜出，并以它为最后一个区块来延长区块链（蓝色-绿色），忽略晚几秒到达的“红色”区块。那些首先收到“红色”区块的节点，会即刻以这个区块为父区块来产生新的候选区块，并尝试寻找这个候选区块的工作量证明解。同样地，接受“绿色”区块的节点会以这个区块为链的顶点开始生成新块，延长这个链。

分叉问题几乎总是在一个区块内就被解决了。网络中的一部分算力专注于“红色”区块为父区块，在其之上建立新的区块；另一部分算力则专注于“绿色”区块上。即便算力在这两个阵营中平均分配，也总有一个阵营抢在另一个阵营前发现工作量证明解并将其传播出去。在这个例子中我们可以打个比方，假如工作在“绿色”区块上的矿工找到了一个“粉色”区块延长了区块链（蓝色-绿色-粉色），他们会立刻传播这个新区块，整个网络会都会认为这个区块是有效的，如图8-5所示。



图8-5 形象化的区块链分叉事件：新区块延长了分支

所有在上一轮选择“绿色”区块为胜出者的节点会直接将这条链延长一个区块。然而，那些选择“红色”区块为胜出者的节点现在会看到两个链：“蓝色-绿色-粉色”和“蓝色-红色”。如图8-6所示，这些节点会根据结果将“蓝色-绿色-粉色”这条链设置为主链，将“蓝色-红色”这条链设置为备用链。这些节点接纳了新的更长的链，被迫改变了原有对区块链的观点，这就叫做链的重新共

识。因为“红”区块做为父区块已经不在最长链上，导致了他们的候选区块已经成为了“孤块”，所以现在任何原本想要在“蓝色-红色”链上延长区块链的矿工都会停下来。全网将“蓝色-绿色-粉色”这条链识别为主链，“粉色”区块为这条链的最后一个区块。全部矿工立刻将他们产生的候选区块的父区块切换为“粉色”，来延长“蓝色-绿色-粉色”这条链。



图8-6 形象化的区块链分叉事件：全网在最长链上重新共识

从理论上来说，两个区块的分叉是有可能的，这种情况发生在因先前分叉而相互对立起来的矿工，又几乎同时发现了两个不同区块的解。然而，这种情况发生的几率是很低的。单区块分叉每周都会发生，而双块分叉则非常罕见。

比特币将区块间隔设计为10分钟，是在更快速的交易确认和更低的分叉概率间作出的妥协。更短的区块产生间隔会让交易清算更快地完成，也会导致更加频繁地区块链分叉。与之相对地，更长的间隔会减少分叉数量，却会导致更长的清算时间。

8.11 挖矿和算力竞赛

比特币挖矿是一个极富竞争性的行业。自从比特币存在开始，每年比特币算力都成指数增长。一些年份的增长还体现出技术的变革，比如在2010年和2011年，很多矿工开始从使用CPU升级到使用GPU，进而使用FPGA（现场可编程门阵列）挖矿。在2013年，ASIC挖矿的引入，把SHA256算法直接固化在挖矿专用的硅芯片上，引起了算力的另一次巨大飞跃。一台采用这种芯片的矿机可以提供的算力，比2010年比特币网络的整体算力还要大。

下表表示了比特币网络开始运行后最初五年的总算力：

2009

0.5 MH/秒–8 MH/秒 (16倍增长)

2010

8 MH/秒–116 GH/秒 (14,500倍增长)

2011

16 GH/秒–9 TH/秒 (562倍增长)

2012

9 TH/秒–23 TH/秒 (2.5倍增长)

2013

23 TH/秒–10 PH/秒 (450倍增长)

2014

10 PH/秒–150 PH/秒 到8月为止 (15倍增长)

在图8-7的图表中，我们可以看到近两年里，矿业和比特币的成长引起了比特币网络算力的指数增长（每秒网络总算力）。

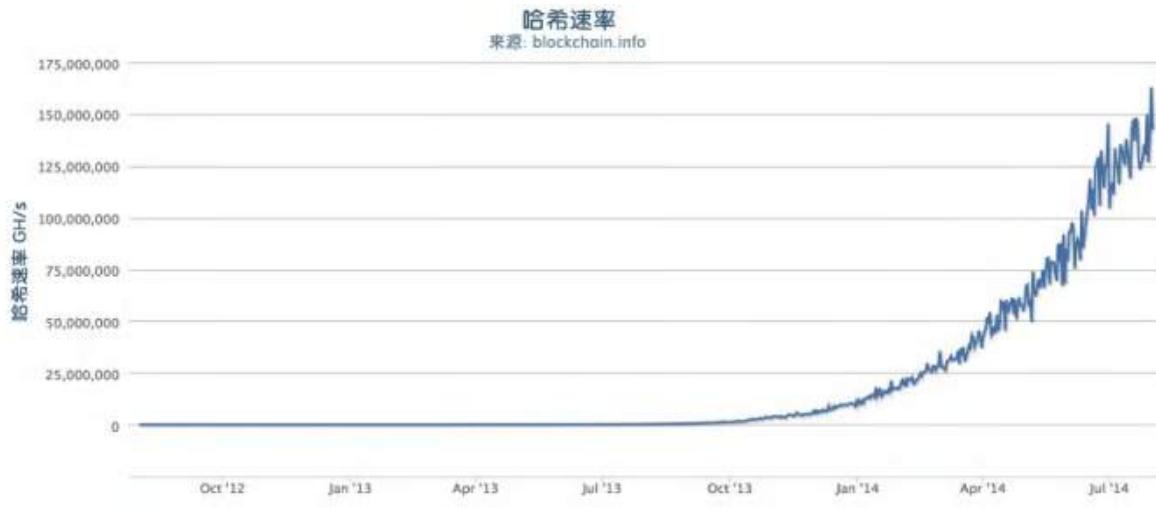


图8-7 近两年的总算力，G次hash/秒

随着比特币挖矿算力的爆炸性增长，与之匹配的难度也相应增长。图8-8中的相对难度值显示了当前难度与最小难度（第一个块的难度）的比例。

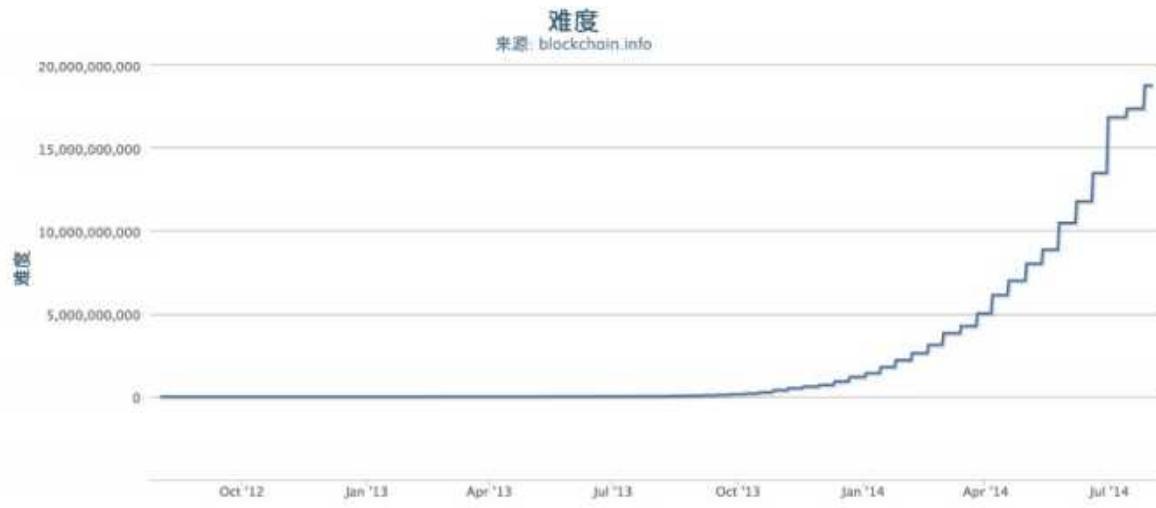


图8-8 近两年的比特币难度值

近两年，ASIC芯片变得更加密集，特征尺寸接近芯片制造业前沿的22纳米。挖矿的利润率驱动这个行业以比通用计算更快的速度发展。目前，ASIC制造商的目标是超越通用CPU芯片制造商，设计特征尺寸为16纳米的芯片。对比特币挖矿而言，已经没有更多飞跃的空间，因为这个行业已经触及了摩尔定律的最前沿。摩尔定律指出计算能力每18个月增加一倍。尽管如此，随着更高密度的芯片和数据中心的部署竞赛，网络算力继续保持同步的指数增长。现在的竞争已经不再是单一芯片的能力，而是一个矿场能塞进多少芯片，并处理好散热和供电问题。

8.11.1 随机值升位方案

2012年以来，比特币挖矿发展出一个解决区块头基本结构限制的方案。在比特币的早期，矿工可以通过遍历随机数(Nonce)获得符合要求的hash来挖出一个块。难度增长后，矿工经常在尝试了40亿个值后仍然没有出块。然而，这很容易通过读取块的时间戳并计算经过的时间来解决。因为时间戳是区块头的一部分，它的变化可以让矿工用不同的随机值再次遍历。当挖矿硬件的速度达到了4GH/秒，这种方法变得越来越困难，因为随机数的取值在一秒内就被用尽了。当出现ASIC矿机并很快达到了TH/秒的hash速率后，挖矿软件为了找到有效的块，需要更多的空间来储存nonce值。可以把时间戳延后一点，但将来如果把它移动得太远，会导致区块变为无效。区块头需要一个新的“差异性”的信息来源。解决方案是使用coinbase交易作为额外的随机值来源，因为coinbase脚本可以储存2-100字节的数据，矿工们开始使用这个空间作为额外随机值的来源，允许

他们去探索一个大得多的区块头值范围来找到有效的块。这个coinbase交易包含在merkle树中，这意味着任何coinbase脚本的变化将导致Merkle根的变化。8个字节的额外随机数，加上4个字节的“标准”随机数，允许矿工每秒尝试 2^{296} （8后面跟28个零）种可能性而无需修改时间戳。如果未来矿工可以尝试所有的可能性，他们还可以通过修改时间戳来解决。同样，coinbase脚本中也有更多额外的空间可以为将来随机数的扩展做准备。

8.11.2 矿池

在这个激烈竞争的环境中，个体矿工独立工作（也就是solo挖矿）没有一点机会。他们找到一个区块以抵消电力和硬件成本的可能性非常小，以至于可以称得上是赌博，就像是买彩票。就算是最快的消费型ASIC也不能和那些在巨大机房里拥有数万芯片并靠近水电站的商业矿场竞争。现在矿工们合作组成矿池，汇集数以千计参与者们的算力并分享奖励。通过参加矿池，矿工们得到整体回报的一小部分，但通常每天都能得到，因而减少了不确定性。

让我们来看一个具体的例子。假设一名矿工已经购买了算力共计6,000GH/S，或6TH/S的设备，在2014年8月，它的价值大约是1万美元。该设备运行功率为3千瓦（KW），每日耗电72度，每日平均成本7或8美元。以目前的比特币难度，该矿工平均每155天或5个月可能solo出一个块。如果这个矿工确实在这个时限内挖出一个区块，奖励25比特币，如果每个比特币价格约为600美元，可以得到15,000美元的收入。这可以覆盖整个时间周期内的设备和电力成本，还剩下大约3,000美元的净利润。然而，在5个月的时间周期内能否挖出一个块主要靠矿工的运气。他有可能在五个月中得到两个块从而赚到非常大的利润。或者，他可能10个月都找不到一个块，从而遭受经济损失。更糟的是，比特币的工作证明（POW）算法的难度可能在这段时间内显著上升，按照目前算力增长的速度，这意味着矿工在设备被下一代更有效率的矿机取代之前，最多有6个月的时间取得成果。如果这个矿工加入矿池，而不是等待5个月内可能出现一次的暴利，他每周能赚取大约500-700美元。矿池的常规收入能帮他随时间摊销硬件和电力的成本，并且不用承担巨大的风险。在7到9个月后，硬件仍然会过时，风险仍然很高，但在此期间的收入至少是定期的和可靠的。

矿池通过专用挖矿协议协调成百上千的矿工。个人矿工在建立矿池账号后，设置他们的矿机连接到矿池服务器。他们的挖矿设备在挖矿时保持和矿池服务器的连接，和其他矿工同步各自的工作。这样，矿池中的矿工分享挖矿任务，之后分享奖励。

成功出块的奖励支付到矿池的比特币地址，而不是单个矿工的。一旦奖励达到一个特定的阈值，矿池服务器便会定期支付奖励到矿工的比特币地址。通常情况下，矿池服务器会为提供矿池服务收取一个百分比的费用。

参加矿池的矿工把搜寻候选区块的工作量分割，并根据他们挖矿的贡献赚取“份额”。矿池为赚取“份额”设置了一个低难度的目标，通常比比特币网络难度低1000倍以上。当矿池中有人成功挖出一块，矿池获得奖励，并和所有矿工按照他们做出贡献的“份额”数的比例分配。

矿池对任何矿工开放，无论大小、专业或业余。一个矿池的参与者中，有人只有一台小矿机，而有些人有一车库高端挖矿硬件。有人只用几十度电挖矿，也有人会用一个数据中心消耗兆瓦级的电量。矿池如何衡量每个人的贡献，既能公平分配奖励，又避免作弊的可能？答案是在设置一个较低难度的前提下，使用比特币的工作量证明算法来衡量每个矿工的贡献。因此，即使是池中最小的矿工也经常能分得奖励，这足以激励他们为矿池做出贡献。通过设置一个较低的取得份额的难度，矿池可以计量出每个矿工完成的工作量。每当矿工发现一个小于矿池难度的区块头hash，就证明了它已经完成了寻找结果所需的hash计算。更重要的是，这些为取得份额贡献而做的工作，能以一个统计学上可衡量的方法，整体寻找一个比特币网络的目标散列值。成千上万的矿工尝试较小区间的hash值，最终可以找到符合比特币网络要求的结果。

让我们回到骰子游戏的比喻。如果骰子玩家的目标是扔骰子结果都小于4（整体网络难度），一个矿池可以设置一个更容易的目标，统计有多少次池中的玩家扔出的结果小于8。当池中的玩家扔出的结果小于8（矿池份额目标），他们得到份额，但他们没有赢得游戏，因为没有完成游戏目标（小于4）。但池中的玩家会更经常地达到较容易的矿池份额目标，规律地赚取他们的份额，尽管他们没有完成更难的赢得比赛的目标。

时不时地，池中的一个成员有可能会扔出一个小于4的结果，矿池获胜。然后，收益可以在池中玩家获得的份额基础上分配。尽管目标设置为8或更少并没有赢得游戏，但是这是一个衡量玩家们扔出的点数的公平方法，同时它偶尔会产生一个小于4的结果。

同样的，一个矿池会将矿池难度设置在保证一个单独的矿工能够频繁地找到一个符合矿池难度的区块头hash来赢取份额。时不时的，某次尝试会产生一个符合比特币网络目标的区块头hash，产生一个有效块，然后整个矿池获胜。

8.11.2.1 托管矿池

大部分矿池是“托管的”，意思是有一个公司或者个人经营一个矿池服务器。矿池服务器的所有者叫矿池管理员，同时他从矿工的收入中收取一个百分比的费用。

矿池服务器运行专业软件以及协调池中矿工们活动的矿池采矿协议。矿池服务器同时也连接到一个或更多比特币完全节点并直接访问一个块链数据库的完整副本。这使得矿池服务器可以代替矿池中的矿工验证区块和交易，缓解他们运行一个完整节点的负担。对于池中的矿工，这是一个重要的考量，因为一个完整节点要求一个拥有最少15-20GB的永久储存空间（磁盘）和最少2GB内存（RAM）的专用计算机。此外，运行一个完整节点的比特币软件需要监控、维护和频繁升级。由于缺乏维护或资源导致的任何宕机都会伤害到矿工的利润。对于很多矿工来说，不需要跑一个完整节点就能采矿，也是加入托管矿池的一大好处。

矿工连接到矿池服务器使用一个采矿协议比如Stratum (STM)或者 GetBlockTemplate (GBT)。一个旧标准GetWork (Gwk) 自从2012年底已经基本上过时了，因为它不支持在hash速度超过4GH/S时采矿。STM和GBT协议都创建包含候选区块头模板的区块模板。矿池服务器通过聚集交易，添加coinbase交易（和额外的随机值空间），计算MERKLE根，并连接到上一个块hash来建立一个候选区块。这个候选区块的头部作为模板分发给每个矿工。矿工用这个区块模板在低于比特币网络的难度下采矿，并发送成功的结果返回矿池服务器赚取份额。

8.11.2.2 P2P矿池

托管矿池存在管理人作弊的可能，管理人可以利用矿池进行双重支付或使区块无效。（参见“8.12 共识攻击”）此外，中心化的矿池服务器代表着单点故障。如果因为拒绝服务攻击服务器挂了或者被减慢，池中矿工就不能采矿。在2011年，为了解决由中心化造成的这些问题，提出和实施了一个新的矿池挖矿方法。P2Pool是一个点对点的矿池，没有中心管理人。

P2Pool通过将矿池服务器的功能去中心化，实现一个并行的类似区块链的系统，名叫份额链。一个份额链是一个难度低于比特币区块链的区块链系统。份额链允许池中矿工在一个去中心化的池中合作，以每30秒一个份额区块的速度在份额链上采矿，并获得份额。份额链上的区块记录了贡献工作的矿工的份额，并且继承了之前份额区块上的份额记录。当一个份额区块上还实现了比特币网络的难度目标时，它将被广播并包含到比特币的区块链上，并奖励所有已经在份额链区块中取得份额的池中矿工。本质上说，比起用一个矿池服务器记录矿工的份额和奖励，份额链允许所有矿工通过类似比特币区块链系统的去中心化的共识机制跟踪所有份额。

P2Pool采矿方式比在矿池中采矿要复杂的多，因为它要求矿工运行空间、内存、带宽充足的专用计算机来支持一个比特币的完整节点和P2Pool节点软件。P2Pool矿工连接他们的采矿硬件到本地P2Pool节点，它通过发送区块模板到矿机来模拟一个矿池服务器的功能。在P2Pool中，单独的矿工创建自己的候选区块，聚合交易，非常类似于solo矿工，但是他们在份额链上合作采矿。P2Pool是一种比单独挖矿有更细粒度收入优势的混合方法。但是不需要像托管矿池那样给管理人太多权力。

最近，在集中式矿池已经接近产生51%攻击的担忧下，P2Pool的份额增长显著。（参见参见“8.12 共识攻击”）P2Pool协议的进一步发展有望去除对完整节点的需要，这将使去中心化采矿更容易。

8.12 共识攻击

比特币的共识机制指的是，被矿工（或矿池）试图使用自己的算力实行欺骗或破坏的难度很大，至少理论上是这样。就像我们前面讲的，比特币的共识机制依赖于这样一个前提，那就是绝大多数的矿工，出于自己利益最大化的考虑，都会通过诚实地挖矿来维持整个比特币系统。然而，当一个或者一群拥有了整个系统中大量算力的矿工出现之后，他们就可以通过攻击比特币的共识机制来达到破坏比特币网络的安全性和可靠性的目的。

值得注意的是，共识攻击只能影响整个区块链未来的共识，或者说，最多能影响不久的过去几个区块的共识（最多影响过去10个块）。而且随着时间的推移，整个比特币块链被篡改的可能性越来越低。理论上，一个区块链分叉可以变得很长，但实际上，要想实现一个非常长的区块链分叉需要的算力非常非常大，随着整个比特币区块链逐渐增长，过去的区块基本可以认为是无法被分叉篡改的。同时，共识攻击也不会影响用户的私钥以及加密算法（ECDSA）。共识攻击也不能从其他的钱包那里偷到比特币、不签名地支付比特币、重新分配比特币、改变过去的交易或者改变比特币持有纪录。共识攻击能够造成的唯一影响是影响最近的区块（最多10个）并且通过拒绝服务来影响未来区块的生成。

共识攻击的一个典型场景就是“51%攻击”。想象这么一个场景，一群矿工控制了整个比特币网络51%的算力，他们联合起来打算攻击整个比特币系统。由于这群矿工可以生成绝大多数的块，他们就可以通过故意制造块链分叉来实现“双重支付”或者

通过拒绝服务的方式来阻止特定的交易或者攻击特定的钱包地址。区块链分叉/双重支付攻击指的是攻击者通过不承认最近的某个交易，并在这个交易之前重构新的块，从而生成新的分叉，继而实现双重支付。有了充足算力的保证，一个攻击者可以一次性篡改最近的6个或者更多的区块，从而使得这些区块包含的本应无法篡改的交易消失。值得注意的是，双重支付只能在攻击者拥有的钱包所发生的交易上进行，因为只有钱包的拥有者才能生成一个合法的签名用于双重支付交易。攻击者只能在自己的交易上进行双重支付攻击，但当这笔交易对应的是不可逆转的购买行为的时候，这种攻击就是有利可图的。

让我们看一个“51%攻击”的实际案例吧。在第1章我们讲到，Alice 和 Bob 之间使用比特币完成了一杯咖啡的交易。咖啡店老板 Bob 愿意在 Alice 给自己的转账交易确认数为零的时候就向其提供咖啡，这是因为这种小额交易遭遇“51%攻击”的风险和顾客购物的即时性（Alice 能立即拿到咖啡）比起来，显得微不足道。这就和大部分的咖啡店对低于25美元的信用卡消费不会费时费力地向顾客索要签名是一样的，因为和顾客有可能撤销这笔信用卡支付的风险比起来，向用户索要信用卡签名的成本更高。相应的，使用比特币支付的大额交易被双重支付的风险就高得多了，因为买家（攻击者）可以通过在全网广播一个和真实交易的UTXO一样的伪造交易，以达到取消真实交易的目的。双重支付可以有两种方式：要么是在交易被确认之前，要么攻击者通过块链分叉来完成。进行51%攻击的人，可以取消在旧分叉上的交易记录，然后在新分叉上重新生成一个同样金额的交易，从而实现双重支付。

再举个例子：攻击者Mallory在Carol的画廊买了描绘伟大的中本聪的三联组画，Mallory通过转账价值25万美金的比特币与Carol进行交易。在等到一个而不是六个交易确认之后，Carol放心地将这幅组画包好，交给了Mallory。这时，Mallory的一个同伙，一个拥有大量算力的矿池的人Paul，在这笔交易写进区块链的时候，开始了51%攻击。首先，Paul利用自己矿池的算力重新计算包含这笔交易的块，并且在新块里将原来的交易替换成了另外一笔交易（比如直接转给了Mallory的另一个钱包而不是Carol的），从而实现了“双重支付”。这笔“双重支付”交易使用了跟原有交易一致的UTXO，但收款人被替换成了Mallory的钱包地址。然后，Paul利用矿池在伪造的块的基础上，又计算出一个更新的块，这样，包含这笔“双重支付”交易的块链比原有的块链高出了一个块。到此，高度更高的分叉区块链取代了原有的区块链，“双重支付”交易取代了原来给Carol的交易，Carol既没有收到价值25万美金的比特币，原本拥有的三幅价值连城的画也被Mallory白白拿走了。在整个过程中，Paul矿池里的其他矿工可能自始至终都没有觉察到这笔“双重支付”交易有什么异常，因为挖矿程序都是自动在运行，并且不会时时监控每一个区块中的每一笔交易。

为了避免这类攻击，售卖大宗商品的商家应该在交易得到全网的6个确认之后再交付商品。或者，商家应该使用第三方的多方签名的账户进行交易，并且也要等到交易账户获得全网多个确认之后再交付商品。一条交易的确认数越多，越难被攻击者通过51%攻击篡改。对于大宗商品的交易，即使在付款24小时之后再发货，对买卖双方来说使用比特币支付也是方便并且有效率的。而24小时之后，这笔交易的全网确认数将达到至少144个（能有效降低被51%攻击的可能性）。

共识攻击中除了“双重支付”攻击，还有一种攻击场景就是拒绝对某个特定的比特币地址提供服务。一个拥有了系统中绝大多数算力的攻击者，可以轻易地忽略某一笔特定的交易。如果这笔交易存在于另一个矿工所产生的区块中，该攻击者可以故意分叉，然后重新产生这个区块，并且把想忽略的交易从这个区块中移除。这种攻击造成的结果就是，只要这名攻击者拥有系统中的绝大多数算力，那么他就可以持续地干预某一个或某一批特定钱包地址产生的所有交易，从而达到拒绝为这些地址服务的目的。

需要注意的是，51%攻击并不是像它的命名里说的那样，攻击者需要至少51%的算力才能发起，实际上，即使其拥有不到51%的系统算力，依然可以尝试发起这种攻击。之所以命名为51%攻击，只是因为在攻击者的算力达到51%这个阈值的时候，其发起的攻击尝试几乎肯定会成功。本质上来看，共识攻击，就像是系统中所有矿工的算力被分成了两组，一组为诚实算力，一组为攻击者算力，两组人都在争先恐后地计算块链上的新块，只是攻击者算力算出来的是精心构造的、包含或者剔除了某些交易的块。因此，攻击者拥有的算力越少，在这场决逐中获胜的可能性就越小。从另一个角度讲，一个攻击者拥有的算力越多，其故意创造的分叉块链就可能越长，可能被篡改的最近的块或者或者受其控制的未来的块就会越多。一些安全研究组织利用统计模型得出的结论是，算力达到全网的30%就足以发动51%攻击了。

全网算力的急剧增长已经使得比特币系统不再可能被某一个矿工攻击，因为一个矿工已经不可能占据全网哪怕的1%算力。但是中心化控制的矿池则引入了矿池操作者出于利益而施行攻击的风险。矿池操作者控制了候选块的生成，同时也控制哪些交易会被放到新生成的块中。这样一来，矿池操作者就拥有了剔除特定交易或者双重支付的权力。如果这种权利被矿池操作者以微妙而有节制的方式滥用的话，那么矿池操作者就可以在不为人知的情况下发动共识攻击并获益。

但是，并不是所有的攻击者都是为了利益。一个可能的场景就是，攻击者仅仅是为了破坏整个比特币系统而发动攻击，而不是为了利益。这种意在破坏比特币系统的攻击者需要巨大的投入和精心的计划，因此可以想象，这种攻击很有可能来自政府资助的组织。同样的，这类攻击者或许也会购买矿机，运营矿池，通过滥用矿池操作者的上述权力来施行拒绝服务等共识攻击。但是，随着比特币网络的算力呈几何级数快速增长，上述这些理论上可行的攻击场景，实际操作起来已经越来越困难。

近期比特币系统的一些升级，比如旨在进一步将挖矿控制去中心化的P2Pool挖矿协议，也都正在让这些理论上可行的攻击变得越来越困难。

毫无疑问，一次严重的共识攻击事件势必会降低人们对比特币系统的信心，进而可能导致比特币价格的跳水。然而，比特币系统和相关软件也一直在持续改进，所以比特币社区也势必会对任何一次共识攻击快速做出响应，以使整个比特币系统比以往更加稳健和可靠。

第9章 竞争币、竞争区块链和应用程序

比特币是20多年的分布式系统和货币研究的结果，是一项具有革命性的新技术：一种基于工作量证明的去中心化的一致性机制。这项比特币的核心发明引领了一场包括货币体系、金融服务、经济学、分布式系统、投票系统、联合监管和合同体系在内的创新浪潮。

本章将探讨比特币和区块链的发明的衍生物：2009年比特币诞生以来所涌现出来的竞争币、竞争区块链和应用程序。大部分篇幅将要探讨竞争币（alt coin），这些电子货币有着与比特币相似的构建模式出来的，但它们完全独立地运行在自己的网络和块链系统之上。

除了我们将在本章讨论到的竞争币以外，至少还有50种以上的竞争币未能论及，这或许会引来这些竞争币的创造者或粉丝们的不满。但这一章节的目的并不是为了给提到的竞争币背书，讨论到的竞争币也不是作者根据主观判断选择的。相反，这一章节讨论的是每一种不同的创新性的竞争币里的代表币种，以此来展现整个竞争币生态系统的宏大和多样。从货币的角度考虑，有些很有趣竞争币实际上是完全失败的，从而让这些例子研究起来变得更有趣，同时也表明本章所讲并非投资建议。

由于每天都会有新的竞争币出现，因此我们的讨论难免会有遗漏，甚至漏掉的就是改变历史的币种。目前这个领域的创新速度快得令人兴奋，同时也预示着，从本书发布之日起，这一章的讨论就将不再有时效性和完备性。

9.1 竞争币和竞争区块链的分类

比特币是一个开源项目，其源代码也作为其他的一些软件项目的基础。由比特币衍生出来的最常见的形式，就是替代性去中心化货币，简称“竞争币”，这类货币使用跟比特币同样的创建块的方式来实现自己的电子货币系统。

在比特币的块链上层，可以实现一系列的协议层。元币、元块链或者块链应用程序以块链为平台，或通过增加协议层的方式扩展比特币协议。如彩色币，万事达币以及合约币。

下一部分我们将介绍一些值得注意的竞争币，比如Litecoin, Dogecoin, Freicoin, Primecoin, Peercoin, Darkcoin和Zerocoin。之所以提到这些，并不是因为它们是最好的或是市值最高的竞争币，而是因为他们是历史上某种竞争币创新的典型代表。

除了竞争币，还有一些关于块链其他实现，他们并不是“币”，可以称之为“竞争块链”（alt chains）。竞争块链通过实现一致性和分布式账簿机制来给诸如合同、名字注册和其他一些应用提供服务。竞争块链使用的是和比特币一样的创建块的机制，有时也会采用货币或代币的支付机制，但它们的主要目的不是为了维持一个货币系统。我们后续将探讨竞争块链的典型代表：域名币、以太坊和 NXT。

除了比特币系统使用的基于工作量证明的一致性机制这种协议以外，还有基于资源证明或发布证明的一些试验性协议。后续将探讨以MaidSafe和Twister为代表的这类协议。

最后，有一些比特币的竞争者，比如Ripple等，也提供电子货币和交易网络，但并没有像比特币一样使用分布式账簿或者一致性机制。这些不基于块链技术的电子货币系统不在本书的讨论范畴，故不会在本章节出现。

9.2 元币平台

元币和元块链是在比特币之上实现的软件层，也可以认为是覆盖在比特币系统之上的平台/协议，或者是在一个币中币的实现。这些功能层拓展了核心比特币协议，使得在比特币交易和比特币地址中编码附加信息成为可能。元币的第一个实现利用了大量的 hack 技巧把元数据添加到比特币块链中，比如使用比特币地址编码数据，或者利用空白的交易字段存放新协议层增加的这些元数据。自从交易脚本操作码问世之后，元币得以直接将信息存放在块链之中。

9.2.1 染色币

染色币是一种在少量比特币上存储信息的一种元协议。一个“被染色的”币，是一定数额的重新用于表达另一种资产的比特币

币。想象一下，在一张一美金的纸币上盖上写有一行“这是Acme公司的一份股权的证明”的印章。现在这张一美金的纸币就有了两层意义，它既是流通的货币，同时又是一份股权证明。由于它作为一份股权证明的价值更大一些，因此你肯定不大会用它来购买糖果吃了（而是保留着），这也让这张纸币不再具有货币的流通属性。染色币也是这个工作原理，通过将一笔数额不大的具体比特币交易转化为某种证明来指征另外一笔财产。所谓“染色”也仅仅是一种隐喻，并非字面意思，而是指增加属性（比如给个颜色）的方式。因此，染色币并没有颜色。

染色币由特殊的钱包管理，这类钱包存储和解析依附在染色币上的元信息。用户在使用这类钱包的时候，可以通过增加有着某种特殊含义的标签的方式，将一般的比特币“染色”为染色币。比如说，这种标签的内容可以表示股票证明、优惠券信息、实际财产、商品或者可收集的代币等等。如何书写和解读这类标签，完全取决于给这枚比特币“染色”的人，他可以决定附着在这部分比特币上的元信息属性。比如信息类型、能不能再分割、某种符号或描述，或者其他的相关信息。这部分比特币一旦被染色，这些币可以用来交易、分割、合并和获取利息等。被染色的比特币也可用通过删除附着信息的方式，也能将“被染色的”比特币恢复为普通比特币。

如例9-1所示，为了演示染色币的使用，我们创建了20单位带有元信息“MasterBTC”的染色币，其中“MasterBTC”代表了可以获取本书免费拷贝的兑换码。每一单位的这种染色币，都可以被出售或赠予给任何装有兼容染色币协议钱包的人，拥有这种染色币的人可以继续转手或者用它来兑换本书的免费拷贝。[染色币的例子](#)如下。

例9-1 The metadata profile of the colored coins recorded as a coupon for a free copy of the book

```
{  
    "source_addresses": [  
        "3NpZmvSPLmN2cVFw1pY7gxEAVPCVfnWfVD"  
    ],  
    "contract_url": "https://www.coinprism.info/asset/_3NpZmvSPLmN2cVFw1pY7gxEAVPCVfnWfVD",  
    "name_short": "MasterBTC",  
    "name": "Free copy of \"Mastering Bitcoin\"",  
    "issuer": "Andreas M. Antonopoulos",  
    "description": "This token is redeemable for a free copy of the book \"Mastering  
Bitcoin\"",  
    "description_mime": "text/x-markdown; charset=UTF-8",  
    "type": "Other",  
    "divisibility": 0,  
    "link_to_website": false,  
    "icon_url": null,  
    "image_url": null,  
    "version": "1.0"  
}
```

9.2.2 万事达币

万事达币是另一个建立在比特币之上的协议，该协议支持多个平台对比特币系统的扩展。万事达币使用名为MST的代币来指导交易，但它并不是一种通货。相反的，它服务于其他应用平台，比如用户货币，智能财产代币，去中心化的财产交易和合约系统等等。就像HTTP协议是TCP协议的应用层一样，Mastercoin是比特币协议的应用层协议。

类似HTTP协议利用TCP协议的80端口和其他协议的TCP流量加以区别，万事达币通过一个名为“exodus”的比特币地址（1EXoDusjGvnjZUyKkxZ4UHEf77z6A5S4P）的进出交易机制来维持协议的运行。万事达币协议正从利用“exodus”地址和多方签名的机制转向利用OP_RETURN比特币操作符来编码信息。

9.2.3 合约币

合约币是另一个建立在比特币系统之上的协议层。合约币拥有用户货币、可交易代币、金融手段、去中心化财产交易和其他一些功能。合约币利用比特币脚本语言中的OP_RETURN操作符记录元信息来增加比特币交易的额外信息。合约币使用名为XCP的代币维持整个系统的运行。

9.3 竞争币/山寨币

绝大多数的山寨币都来自比特币源代码的克隆，少数则没有使用比特币的任何源码，仅仅是借鉴了块链的模型后自己实现。竞争币或竞争块链（下一节会讲到）都是运行在自己块链上的独立的块链实现。之所以以命名区分，主要是因为竞争币主要

用做货币，而竞争块链则不是。

严格意义上讲，比特币的第一个克隆并不是一个竞争币而是一个名为Namecoin的竞争块链，我们将在下一节讨论。

从发布时间来看，第一款竞争币名为IXCoin，出现于2011年8月。IXCoin更改了比特币的一些参数，尤其是通过调整每个新块的奖励为96个币，从而增加了货币的发行量。

2011年9月，Tenebrix发布。Tenebrix是第一款使用了其他工作量证明算法（script）的加密货币，这种算法起初是为了防止密码遭暴力破解而设计的。Tenebrix的目标是通过使用这种消耗内存的算法来实现一种不依赖GPU和ASIC芯片的电子货币。

除了使用这种算法，莱特币还把新块产生的时间从比特币的10分钟缩短为2分半钟。如果把比特币看作电子货币中的金币的话，那么莱特币的愿景就是当电子货币系统中的银币，谋求成为比特币的一种轻量的替代货币。考虑到莱特币8,400万的货币总量和相对更快的确认速度，很多莱特币的拥趸相信与比特币相比，莱特币更适合零售业的交易。

以比特币和莱特币为基础的竞争币数量在2011和2012年呈持续增长状态。到了2013年，有20种竞争币在市场中谋求一席之地。到2013年年底，这个数字增至200种，2013年也因此被誉为“竞争币之年”。竞争币的增长在2014年依然没有放缓，截至本书截稿，市场上的竞争币数量已经达到了500种以上，其中超过一半的竞争币克隆自莱特币。

之所以市面上的竞争币有超过500种之多，是因为创造一种新的竞争币非常简单。因此，大多数的竞争币跟比特币区别非常小，并没有多少研究价值。但在这些通过毫无创意的抄袭和圈钱模式产生的竞争币中间，依然有一些值得一提的非常重要的创新。这些特殊的竞争币，要么采用完全不同的实现方式，要么在比特币现有的设计模式上加入了重大的创新。下面所列出的就是这些竞争币区别于比特币的三点主要不同：

- ▷ 货币策略不同
- ▷ 基于工作量证明的一致性机制不同
- ▷ 一些特殊的功能，比如更强的匿名性等等

For more information, see this [graphical timeline of alt coins and alt chains](#).

9.3.1 评估竞争币的价值

市面上这么多竞争币，该如何决定关注哪些呢？一些竞争币旨在成为广泛流通的主流货币，还有一些是实验室项目，仅仅是为了测试不同的特性和货币模型，更多的仅仅是那些发起者们创富的手段。我一般通过某款竞争币的决定性特性和市场规模来对其进行价值评估。

以下是关于竞争币和比特币的不同之处的几个问题：

- ▷ 这款竞争币有没有引入重大的创新？
- ▷ 如果有，那么这项创新是不是足够吸引使用比特币的用户转移过来？
- ▷ 这款竞争币是不是致力于某一细分领域或应用？
- ▷ 这款竞争币可以吸引到足够多的矿工来抵御一致性攻击吗？

还有一些有关关键财务和市场指标的问题：

- ▷ 这款竞争币的市场总值是多少？
- ▷ 整个系统的用户/钱包规模大概是多少？
- ▷ 接受其支付的商家有多少？
- ▷ 整个系统每日的交易数是多少？
- ▷ 交易总量是多少？

本节，我们将主要在技术和创新层面上就上述第一组的四个问题进行讨论。

9.3.2 货币属性不同于比特币的竞争币：莱特币、狗狗币和Freicoin

比特币本身所具有的一些货币属性令其成为总额固定并且不通货膨胀的货币。比如，比特币的总量为固定的2,100万枚，新币

的生成速度随时间递减，块生成速度为十分钟一块，这个频率也控制了整个比特币系统交易的确认速度和新币的生成。很多竞争币通过对这些货币属性的微调，来达到实现不同的货币政策的目的。在这类竞争币中，值得一提的有以下几种。

莱特币

莱特币是最早的一批竞争币中的一员，自2011年发布至今，已经成为继比特币之后的第二成功的电子货币。它的主要创新在于两点，一是使用了scrypt作为工作量证明算法（继承自前文提到的Tenebrix），二是更快的货币参数。

- ▷ 出块速度：2分钟
- ▷ 货币总量：到2140年达到8,400万
- ▷ 一致性算法：scrypt
- ▷ 市场总值：1亿6,000万美金（截至2014年年中）

狗狗币

狗狗币是基于莱特币的一款竞争币，于2013年12月发布。狗狗币之所以值得一提，主要是因为其飞快的出块速度和惊人的货币总量，其目的也是为了鼓励用户交易和给小费等。狗狗币始于一个玩笑，在其2014年快速衰退之前，一经发布就风行于巨大而活跃的用户社区。下面是狗狗币的一些特性：

- ▷ 出块速度：60秒
- ▷ 货币总量：到2015年达到100,000,000,000（1,000亿）
- ▷ 一致性算法：scrypt
- ▷ 市场总值：1,200万美金（截至2014年年中）

Freicoin

Freicoin于2012年7月发布。它是一种滞留性通货，可以理解为存在钱包中的货币的利率为负数。为了鼓励用户消费和减少储蓄，Freicoin拟定了一个4.5%的APR fee。Freicoin值得一提的原因是它的货币策略跟比特币的通货紧缩策略恰恰相反。作为货币，Freicoin并不是非常成功，但它是竞争币所能表现的多样性货币策略的生动体现。

- ▷ 出块速度：10分钟
- ▷ 货币总量：到2140年达到1亿
- ▷ 一致性算法：SHA256
- ▷ 市场总值：13万美金（截至2014年年中）

9.3.3 一致性机制创新：peercoin, Myriad, Blackcoin, vericoin 和 NXT

比特币的一致性机制建立在基于SHA256算法的工作量证明之上。第一款引入scrypt算法作为一致性机制的竞争币是为了便于CPU挖矿，避免ASIC矿机可能导致的算力集中化的问题。在那之后，对于一致性机制的创新一直很活跃。诸多竞争币陆续引进了包括scrypt, scrypt-N, Skein, Groestl, SHA3, X11, Blake在内的算法来实现工作量证明的一致性机制。而在2013年，作为工作量证明的一种替代机制——权益证明的出现，成为现代竞争币的基础。

权益证明系统中，货币的所有人可以将自己的通货做利息抵押。类似于存款证明(CD)，参与者可以保有他们货币的一部分，通过利息和矿工费的方式获取回报。

Peercoin

Peercoin于2012年8月发布，是首款工作量证明和权益证明混用的竞争币。

- ▷ 出块速度：10分钟
- ▷ 货币总量：没有上限
- ▷ 一致性算法：工作量证明和权益证明混用
- ▷ 市场总值：140万美金（截至2014年年中）

Myriad

Myriad于2014年2月发布，值得一提的是，它同时使用5种工作量证明算法（HA256d, Scrypt, Qubit, Skein, or Myriad-Groestl），根据参与矿工的情况动态选择。这是为了让整个Myriad系统不受集中化的ASIC矿机的影响，同时也加强了其抵御一致性攻击的能力。

- ▷ 出块速度：平均30秒
- ▷ 货币总量：到2024年达到 20 亿
- ▷ 一致性算法：多重算法的工作量证明机制
- ▷ 市场总值：12万美金（截至2014年中）

Blackcoin

Blackcoin发布于2014年2月，使用的是权益证明的一致性机制。同时，它引入的可以根据受益自动切换到不同竞争币的“多矿池”机制也值得一提。

- ▷ 出块速度：1分钟
- ▷ 货币总量：没有上限
- ▷ 一致性算法：权益证明机制
- ▷ 市场总值：370万美金（截至2014年年中）

VeriCoin

VeriCoin于2014年5月发布。它使用了权益证明机制，并辅以随着市场供需关系动态调整的利率。它也是首款可以直接在钱包中兑换比特币支付的竞争币。

- ▷ 出块速度：1分钟
- ▷ 货币总量：没有上限
- ▷ 一致性算法：权益证明机制
- ▷ 市场总值：110万美金（截至2014年年中）

NXT

NXT（发音同Next）是一种“纯”权益证明的竞争币，它甚至不采用工作量证明的挖矿机制。NXT是一款完全自己实现的加密货币，并非衍生自比特币或其他竞争币。NXT具有很多先进的功能，包括名字注册、去中心化资产交易、集成的去中心化加密信息和权益委托。NXT的拥趸称NXT为新一代加密货币或者或者加密货币2.0。

- ▷ 出块速度：1分钟
- ▷ 货币总量：没有上限
- ▷ 一致性算法：权益证明机制
- ▷ 市场总值：3,000万美金（截至2014年年中）

9.3.4 多目的挖矿创新：Primecoin, Curecoin, Gridcoin

比特币的工作量证明机制只有一个目的：维护比特币系统的安全。跟维护一个传统货币系统比起来，挖矿的成本并不高。然而，某些批评者认为某些批评者认为挖矿这一行为是一种浪费。新一代的加密货币试图解决这个争议。多目的挖矿算法就是为了解决工作量证明导致的“浪费”问题而出现的。多目的挖矿在为货币系统的安全加入额外需求的同时，也为该系统的供需关系加入了额外的变量。

Primecoin

Primecoin是在2013年7月发布的。它的工作量证明算法可以搜索质数，计算孪生素数表。素数在科研领域有广泛的应用。Primecoin的块链中包含其发现的质数，因此Primecoin的块链在用于维护公共交易账簿的同时，还会产生一份公开的科学发现（素数表）。

- ▷ 出块速度：1分钟
- ▷ 货币总量：没有上限
- ▷ 一致性算法：含有素数计算功能的工作量证明算法
- ▷ 市场总值：130万美金（截至2014年年中）

Curecoin

Curecoin于2013年5月发布。通过Folding@Home项目，它将SHA256工作量证明算法和蛋白质褶皱结构的研究结合了起来。蛋白质褶皱研究需要对蛋白质进行生化反应的模拟，用于发现治愈疾病的新药，但这一过程需要大量的计算资源。

- ▷ 出块速度：10分钟
- ▷ 货币总量：没有上限
- ▷ 一致性算法：含有蛋白质结构研究功能的工作量证明算法
- ▷ 市场总值：6.8万美金（截至2014年年中）

Gridcoin

Gridcoin是2013年10月对外发布的。它结合了以scrypt为基础的工作量证明算法和参与BOINC计算项目的补贴机制。BOINC——伯克利开发网络计算系统——是一项用于科学研究网格计算的开放协议。Gridcoin网络输出算力给BOINC这个计算平台，而不是自己直接用算力去解决某一个具体的科学问题。

- ▷ 出块速度：150秒
- ▷ 货币总量：没有上限
- ▷ 一致性算法：整合了BOINC网格计算的工作量证明算法
- ▷ 市场总值：12.2万美金（截至2014年年中）

9.3.5 致力于匿名性的竞争币：CryptoNote, Bytecoin, Monero, Zerocash/Zerocoin, Darkcoin

比特币一直被误解为匿名货币。事实上，将个人和比特币地址关联起来，是一件相对容易的事情。利用大数据分析可以很容易地得到某一比特币地址的消费习惯。一些竞争币试图通过增强匿名性来解决这个问题。最初尝试的是Zerocoin，它是一种建立在比特币协议之上的元币协议，最早发布于2013 IEEE安全隐私讨论会上。截至本书完稿时，基于这个协议的Zerocash的竞争币系统还在开发当中。匿名性的另一种实现名为CryptoNote，初见于2013年10月的一篇论文。CryptoNote是一种由多个竞争币一起实现的基础技术，稍后将重点讨论。除了上述两种实现之外，还有一些其他的独立的匿名币，比如利用影子地址和交易混淆来达到匿名性目的的Darkcoin。

Zerocoin/Zerocash

Zerocoin是2013年由Johns Hopkins发表的电子货币匿名性的一种理论实现。截至本书完稿时，基于这一理论的Zerocash的竞争币系统还在开发当中。

CryptoNote

CryptoNote是一种提供了电子货币基础的匿名性的参考实现，于2013年10月发布。它可以被克隆继而衍生出其他实现，并且内建了一个周期性的重置机制使其不能用作货币。很多竞争币是基于CryptoNote实现的。比如Bytecoin (BCN), Aeon (AEON), Boolberry (BBR), duckNote (DUCK), Fantomcoin (FCN), Monero (XMR), MonetaVerde (MCN), 和 Quazarcoin(QCN)。值得指出的是，CryptoNote是一个没有借鉴比特币的完全独立的实现。

Bytecoin

Bytecoin是CryptoNote的第一个实现，基于CryptoNote技术提供切实可行的匿名货币方案。Bytecoin于2012年发布。这里要留意一下，在基于CryptoNote的Bytecoin发布之前，有一个名字同样为Bytecoin的电子货币，货币符号为BTE，而基础CryptoNote的Bytecoin的货币符号为BCN。Bytecoin使用了基于Cryptonight的工作量证明机制，每个实例需要至少2MB的

RAM，这使得GPU和ASIC矿机无法在Bytecoin网络中运行。Bytecoin继承了CryptoNote的环签名、不可链接交易和块链抗分析匿名性等机制。

- ▷ 出块速度：2分钟
- ▷ 货币总量：1,840亿BCN
- ▷ 一致性算法：基于Cryptonight的工作量证明机制
- ▷ 市场总值：300万美金（截至2014年年中）

Monero

Monero是CryptoNote的另一个实现。其货币曲线比Bytecoin稍显平缓，在系统运行的最开始四年发行80%的货币。它提供一些基于CryptoNote的匿名性特性。

- ▷ 出块速度：1分钟
- ▷ 货币总量：1,840万XMR
- ▷ 一致性算法：基于Cryptonight的工作量证明机制
- ▷ 市场总值：500万美金（截至2014年年中）

Darkcoin

Darkcoin在2014年1月发布。Darkcoin通过一个名为DarkSend的混淆协议来实现匿名货币。值得一提的是，Darkcoin在工作量证明算法中使用了11轮不同的哈希函数（blake, bmw, groestl, jh, keccak, skein, luffa, cubehash, shavite, simd, echo）

- ▷ 出块速度：2.5分钟
- ▷ 货币总量：最高2,200万DRK
- ▷ 一致性算法：基于多轮哈希的工作量证明算法
- ▷ 市场总值：1,900万美金（截至2014年年中）

9.4 非货币型竞争区块链

非货币型竞争币区块链是区块链设计模式的另类实现，并不主要作为货币使用。当然不少这种区块链的确含有货币，只不过它们的货币仅是一种象征，用于分配其他东西，比如一种资源或者一份合约。换句话说，货币并不是非货币型竞争币区块链的要点，仅仅是一种次要特征。

9.4.1 域名币

域名币是比特币源代码的首个克隆产物，它是一种使用区块链的去中心化平台，用来注册和转让键-值对。域名币支持全球的域一名注册，类似因特网上的域一名注册系统。目前域名币作为根域名.bit的替代性域名服务（DNS）使用。域名币也可以用来注册其他命名空间下的名称和键-值对，例如存储邮件地址、密钥、SSL证书、文件签名、投票系统和股票凭证之类，以及许多其他应用。

域名币系统也有它自己的货币（符号为NMC），用于支付域名注册及转让的交易费用。依照当前价格（2014年8月），注册一个域名的费用是0.01NMC，大约相当于1美分。与比特币类似，这些费用支付给域名币的矿工。

域名币的基本参数与比特币相同：

- ▷ 出块速度：10分钟
- ▷ 货币总量：2140年将达2,100万NMC
- ▷ 共识算法：SHA256工作量证明法
- ▷ 市场总值：1,000万美元（截至2014年年中）

域名币的命名空间不受限制，任何人都可以以任意方式使用任意命名空间。不过，一些特定的命名空间因为有着一致认可的规范，因此当从区块链读取它们的时候，应用层的软件知道如何进行后续操作。无论使用何种软件，假如区块链遭到篡改，

读取这个特定命名空间的软件都会报错。域名币一些流行的命名空间有：

- ▷ d/ 是 .bit 域名的域一名命名空间
- ▷ id/ 是存储诸如邮件地址、PDP 密钥等个人身份验证的命名空间
- ▷ u/ 是一个补充性的、更加结构化的存储身份的规范（基于公开规范）

域名币的客户端与比特币核心十分类似，因为前者的代码是从后者衍生而来的。在安装过程中，域名币客户端会下载其区块链的完整拷贝，下载完成之后便可进行查询和注册域名了。域名币客户端有3条可用命令：

name_new 查询并提前注册一个域名

name_firstupdate 公开注册一个域名

name_update 改变域名的信息或刷新域名

例如，注册mastering-bitcoin.bit这个域名，需使用按如下方法使用name_new指令：

```
$ namecoind name_new d/mastering-bitcoin
[
    "21cbab5b1241c6d1a6ad70a2416b3124eb883ac38e423e5ff591d1968eb6664a",
    "a05555e0fc56c023"
]
```

name_new 通过给该域名创建一个哈希数和一个随机密钥来做一个对域名的声明。命令执行完毕后返回的两个字符串分别是哈希数和随机密钥（上例中的a05555e0fc56c023），二者可用于公开此次域名注册。一旦上述声明记录于域名币的区块链上，该声明便可转换为一个公开的注册。使用 name_firstupdate 命令便可达到此目的，当然，要提供随机密钥：

```
$ namecoind name_firstupdate d/mastering-bitcoin a05555e0fc56c023 "{\"map\": {\"www\": {\"ip\": \"1.2.3.4\"}}}"
b7a2e59c0a26e5e2664948946ebeca1260985c2f616ba579e6bc7f35ec234b01
```

这个例子将会把域名 www.mastering-bitcoin.bit 映射到1.2.3.4这个IP地址上，返回的哈希数则是交易ID，能够用于追踪此次注册。你可以运行name_list命令来查看自己名下注册了哪些域名：

```
$ namecoind name_list
[
    {
        "name" : "d/mastering-bitcoin",
        "value" : "{map: {www: {ip:1.2.3.4}}}",
        "address" : "NCccBXrRUahAGrisBA1BLPWQfSrups8Geh",
        "expires_in" : 35929
    }
]
```

每生成36,000个区块（大约200到250天），域名币上的注册就需要更新一次。不过name_update命令不收取费用，因此续约域名是免费的。也有第三方提供商提供一个网页界面来帮助处理注册、自动续约及更新等事宜，当然，这要花费你少许费用。使用第三方提供商的好处是你不需要运行一个域名币客户端了，坏处是你失去了对域名币提供去中心化的域名注册服务的自主控制。

9.4.2 Bitmessage

Bitmessage是一个实现了去中心化安全消息服务的比特币竞争币区块链，其本质上是一个无服务器的加密电子邮件系统。Bitmessage可以让用户通过一个Bitmessage地址来编写和发送消息。这些消息的运作方式与比特币交易大致相同，但区别在于消息是短暂瞬态的——如果超过两天还没被传送至目的节点，消息将会丢失。发送方和接收方都是假名，除了一个bitmessage地址外，他们没有其他的身份标识。但发送方和接收方有严格的身份验证，这意味着不会出现“欺骗”消息。Bitmessage都是经加密再发送给接收方，Bitmessage网络也因此可以抵御全面监视。除非网络偷听者破坏了接收方的

设备，否则他们无法截取邮件消息。

9.4.3 以太坊

以太坊是一种图灵完备的平台，基于区块链账簿，用于合约的处理和执行。它不是比特币的一个克隆，而是完完全全独立的一种设计和实现。以太坊内置一种叫做ether的货币，该货币是付合约执行之费用所必须的。以太坊区块链记录的东西叫做合约，所谓合约，就是一种低级二进制码，也是一种图灵完备语言。本质上，合约其实是运行在以太坊系统中各个节点上的程序。这些程序可以存储数据、支付及收取、存储ether以及执行无穷范围（因此才叫图灵完备）的计算行为，在系统中充当去中心化的自治软件代理。

以太坊能够实现一些颇为复杂的系统，这些系统甚至还能自我实现为其他的竞争币区块链。举例来说，下面就是一个类域名币的域名注册合约，使用以太坊代码编写（或者更准确地说，使用一种可编译为以太坊代码的高级代码编写）：

```
if !contract.storage[msg.data[0]]: # Is the key not yet taken?  
    # Then take it!  
    contract.storage[msg.data[0]] = msg.data[1]  
    return(1)  
else:  
  
    return(0) // Otherwise do nothing
```

9.5 加密货币的未来

总体来看，加密货币的未来甚至比比特币还要光明。这是因为，比特币引入了这样一种全新的形式，那就是去中心化的组织和共识，而且这种形式已经催生了大量不可思议的创新。这些创新很有可能影响到社会中相当广泛的行业，从分布式系统科学到金融、经济、货币、中央银行以及企业管理，不一而足。在以前，很多人类活动都需要一个中心化的机构或组织来实现权威或可信控制点的功能，现在，这些都可以去中心化了。区块链和共识系统的发明，还会显著降低大型系统在组织及协调上的花销，同时也将消除权力攫取、腐败及管制俘获的可趁之机。

第10章 比特币安全

保护比特币是很具有挑战性的事，因为比特币不像银行账户余额那样体现抽象价值。比特币其实更像数字现金或黄金。你可能听过这样的说法，“现实持有，败一胜九。”好吧，在比特币的世界里，这样的持有只能让你有一成胜率。而只有拥有解锁比特币的密钥才相当于持有现金或一块贵重金属。你可能会将密钥丢失，会放错地方，会被盗或者不小心错支了数额。无论是哪种场景，用户都没有办法撤回，因为这就像是将现金丢在了车水马龙的大街上。

不过，与现金、黄金或者银行账户相比，比特币有着一个独一无二的优势。你不能“备份”你的现金、黄金或者银行账户，但你可以像备份其他文件一样，备份含有密钥的比特币钱包。它可以被复制成很多份，放到不同的地方保存起来，甚至能打印到纸上进行实体备份。比特币与至今为止的其他货币是如此不同，以致于我们需要以一种全新的思维方式来衡量比特币的安全性。

10.1 安全准则

比特币的核心准则是去中心化，这一点对安全性具有重要意义。在中心化的模式下，例如传统的银行或支付网络，需要依赖于访问控制和审查制度将不良行为者拒之门外。相比之下，比特币这样的去中心化系统则将责任和控制权都移交给了用户。由于网络的安全性是基于工作量证明而非访问控制，比特币网络可以对所有人开放，也无需对比特币流量进行加密。

在一个传统的支付网络中，例如信用卡系统，支付是终端开放式的，因为它包含了用户的个人标识（信用卡号）。在初次支付后，任何能获得该标识的人都可以从所有者那里反复“提取”资金。因此，该支付网络必须采取端对端加密的方式，以确保没有窃听者或中间人可以在资金流通或存储过程中将交易数据截获。如果坏人获得该系统的控制权，他将能破获当前的交易和支付令牌，他还可以随意动用这笔资金。更糟的是，当客户数据被泄露时，顾客的个人信息将被盗窃者们一览无余。客户这时必须立即采取措施，以防失窃帐户被盗窃者用于欺诈。

比特币则截然不同，一笔比特币交易只授权向指定接收方发送一个指定数额，并且不能被修改或伪造。它不会透露任何个人信息，例如当事人的身份，也不能用于权限外的支付。因此，比特币的支付网络并不需要加密或防窃听保护。事实上，你可以在任何公开的网络上广播比特币交易的数据，例如在不安全的WiFi或蓝牙网络上公开传播比特币交易的数据，这对安全性没有任何影响。

比特币的去中心化安全模型很大程度上将权力移交到用户手上，随之而来的是用户们保管好密钥的责任。这对于大多数用户来说并非一件易事，特别是在像智能手机或笔记本电脑这种能时刻联网的通用设备上。虽然比特币的去中心化模型避免了常见的信用卡盗用等情况，但很多用户由于无法保管好密钥从而被黑客攻击。

10.1.1 比特币系统安全开发

对于比特币开发者而言最重要的是去中心化原则。大多数开发者对中心化的安全模型很熟悉，并可能试图将中心化的模型运用到借鉴比特币的应用中去，这将给比特币带来灭顶之灾。

比特币的安全性依赖于密钥的分散性控制，并且需要矿工们各自独立地进行交易验证。如果你想利用好比特币的安全性，你需要确保自己处于比特币的安全模型里。简而言之，不要将用户的密钥控制权拿走，不要接受非区块链交易信息。

例如，许多早期的比特币交易所将所有用户的资金集中在一个包含着私钥的“热钱包”里，并存放在服务器上。这样的设计夺取了用户的掌控权，并将密钥集中到单个系统里。很多这样的系统都被黑客攻破了，并给客户带来灾难性后果。

另一个常见的错误是接受区块链离线交易，妄图减少交易费或加速交易处理速度。一个“区块链离线交易”系统将交易数据记录在一个内部的中心化账本上，然后偶尔将它们同步到比特币区块链中。这种做法，再一次，用专制和集中的方式取代比特币的去中心化安全模型。当数据处于离线的区块链上的时候，保护不当的中心化账本里的资金可能会不知不觉被伪造、被挪用、被消耗。

除非你是准备大力投资运营安全，叠加多层访问控制，或（像传统的银行那样）加强审计，否则在将资金从比特币的去中心化安全场景中抽离出来之前，你应该慎重考虑一番。即使你有足够的资金和纪律去实现一个可靠的安全模型，这样的设计也

仅仅是复制了一个脆弱不堪，深受账户盗窃威胁、贪污和挪用公款困扰的传统金融网络而已。要想充分利用比特币特有的去中心化安全模型，你必须避免中心化架构的常见诱惑，因它最终将摧毁比特币的安全性。

10.1.2 信任根源

传统的安全体系基于一个称为信任根源的概念，它指的总体系统或应用程序中一个可信赖的安全核心。安全体系像一圈同心圆一样围绕着信任根源来进行开发，像层层包裹的洋葱一样，信任从内至外依次延伸。每一层都构建于更可信的内层之上，通过访问控制，数字签名，加密和其他安全方式确保可信。随着软件系统变得越来越复杂，它们更可能出现问题，安全更容易受到威胁。其结果是，软件系统变得越复杂，就越难维护安全性。信任根源的概念确保绝大多数的信任被置于系统一个不是过于复杂的部分，因此该系统的这部分也相对坚固，而更复杂的软件则在它之上构建。这样的安全体系随着规模扩大而不断重复出现，首先信任根源建立于单个系统的硬件内，然后将该信任根源通过操作系统扩展到更高级别的系统服务，最后逐次扩散到圈内多台服务器上。

比特币的安全体系与这不同。在比特币里，共识系统创建了一个可信的完全去中心化的公开账本，一个正确验证过的区块使用创世块作为信任的根源，建立一条直至当前区块的可信任链。比特币系统可以并应该使用区块链作为它们的信任根源。在设计一个多系统服务机制的比特币应用时，你应该仔细确认安全体系，以确保对它的信任能有据可依。最终，唯一可确信无疑的是一条完全有效的区块链。如果你的应用程序或明或暗地信赖于区块链以外的东西，就该引起重视，因为它可能会引入漏洞。一个不错的方法评估你应用程序的安全体系：单独考量每个组件，设想该组件被完全攻破并被坏人掌控的场景。依次取出应用程序的每个组件，并评估它被攻破时对整体安全的影响。如果你的应用程序的安全性在该组件沦陷后大打折扣，那就说明你已经对这些组件过度信任了。一个没有漏洞的比特币应用程序应该只受限于比特币的共识机制，这意味着其安全体系的信任源于比特币最坚固的部分。

无数个黑客攻击比特币交易所的例子都是因为轻视了这一点，他们的安全体系和设计甚至无法通过基本的审查。这种中心化的实现方式将信任置于比特币区块链之外的诸多组件之上，例如热钱包，中心化的账本数据库，简易加密的密钥，以及许多类似的方案。

10.2 用户最佳安全实践

人类使用物理的安全控制已经有数千年之久。相比之下，我们的数字化安全经验的年纪还不满50岁。现代通用的操作系统并不是十分安全，亦不特别适合用来存储数字货币。我们的电脑通过一直连接的互联网长时间暴露在外，它们运行着成千上万第三方软件组件，这些软件往往可以不受约束地访问用户的文件。你电脑上安装的众多软件只要有一个恶意软件，就会威胁到你的文件，可窃取你钱包里的所有比特币。想要杜绝病毒和木马对电脑的威胁，用户要达到一定的计算机维护水平，只有小部分人能做到。

尽管信息安全经过了数十年的研究和发展，数字资产在绵延不绝的攻势下还是十分脆弱。纵使是像金融服务公司，情报机构或国防承包商这样拥有高度防护和限制的系统，也经常会被攻破。比特币创造了具有内在价值的数字资产，它可以被窃取，并立即转移给他人而无法收回。这让黑客有了强烈的作案动机。至今为止，黑客都不得不在套现后更换身份信息或帐户口令，例如信用卡或银行账户。尽管掩饰和洗白这部分财务信息的难度不小，但越来越多的窃贼从于此道。而比特币使这个问题加剧了，因为它不需要掩饰或洗白，它本身就是具有内在价值的数字资产。

幸运的是，比特币也有着激励机制，以提高计算机的安全性。如前所述，计算机受威胁的风险是模糊的，间接的，而比特币让这些风险变得明确清晰。在电脑上保存比特币让用户时刻注意他们需要提高计算机的安全性，结果便是这使得比特币和其它数字货币得以传播和扩散，我们已经看到在黑客技术和安全解决方案双方的提升。简单来说，黑客现在有着一个非常诱人的目标，而用户也有明确的激励性去保卫自己。

在过去的三年里，随着比特币不断被接纳，一个直接的结果是，我们已经看到信息安全领域取得了巨大创新，例如硬件加密，密钥存储和硬件钱包，多重签名技术和数字托管。在下面的章节中，我们将研究各种实际用户安全中的实践经验。

10.2.1 比特币物理存储

相比数字信息的安全，大多数用户对物理安全更加熟悉，一个非常有效保护比特币的方法是，将它们转换为物理形式。比特币密钥不过是串长数字而已。这意味着它们可以以物理形式存储起来，如印在纸上或蚀刻成金属硬币上。这样保护密钥就变成了简单地保护印着比特币密钥的物理实体。一组打印在纸上的比特币密钥被称为“纸钱包”，有许多可以用来创建它们的免

费工具。我个人将大部分（99%以上）的比特币存储在纸钱包上，并用BIP0038加密，复制了多份并锁在保险箱里。将比特币离线保存被称为冷存储，它是最有效的安全技术之一。冷存储系统是在一个离线系统（一个从来没有连接过互联网的系统）上生成密钥，并离线存储到纸上或者U盘等电子媒介。

10.2.2 硬件钱包

从长远来看，比特币安全将越来越多地以硬件防篡改钱包的形式出现。与智能手机或台式电脑不同，一个比特币硬件钱包只有一个目的，安全地存储比特币。不像容易受害的常用软件那样，硬件钱包只提供了有限的接口，从而可以给非专业用户提供近乎万无一失的安全等级。我预期将看到硬件钱包成为比特币储存的主要方式。要想看硬件钱包的实例，请查阅[Trezor](#)。

10.2.3 平衡风险

虽然大多数用户都非常关注比特币防盗，其实还有一个更大的风险存在。数据文件丢失的情况时有发生。如果比特币的数据也在其中，损失将会让人痛苦不堪。为了保护好比特币钱包，用户必须非常注意不要剑走偏锋，这样不至于会搞丢比特币。在2011年7月，一个著名的比特币认知教育项目损失了近7,000枚比特币。为了防止被盗窃，其主人曾之前采取了一系列复杂的操作去加密备份。结果他们不慎丢失了加密的密钥，使得备份变得毫无价值，白白失去了一大笔财富。如果你保护比特币的方式太过了，这好比于把钱藏在沙漠里，你可能不能再把它找回来了。

10.2.4 分散风险

你会将你的全部家当换成现金放在钱包里随身携带么？大多数人会认为这非常不明智，但比特币用户经常会将所有的比特币放在一个钱包里。用户应该将风险分散到不同类型的比特币钱包。审慎的用户应该只留一小部分（或许低于5%）的比特币在一个在线的或手机钱包，就像零用钱一样，其余的部分应该采用不同存储机制分散开来，诸如电脑钱包和离线（冷存储）钱包。

10.2.5 多重签名管理

当一个公司或个人持有大量比特币时，他们应该考虑采用多重签名的比特币地址。多重签名比特币地址需要多个签名才能支付，从而保证资金的安全。多重签名的密钥应存储在多个不同的地方，并由不同的人掌控。打个比方，在企业环境中，密钥应该分别生成并由若干公司管理人员持有，以确保没有任何一个人可以独自占有资金。多重签名的地址也可以提供冗余，例如一个人持有多个密钥，并将它们分别存储在不同的地方。

10.2.6 存活能力

一个非常重要却又常常被忽视的安全性考虑是可用性，尤其是在密钥持有者丧失工作能力或死亡的情况下。比特币的用户被告知应该使用复杂的密码，并保证他们的密钥安全且不为他人所知。不幸的是，这种做法使得在用户无法解锁时，用户的家人几乎无法将该财产恢复。事实上，比特币用户的家人可能完全不知道这笔比特币资金的存在。

如果你有很多的比特币，你应该考虑与一个值得信赖的亲属或律师分享解密的细节。一个更复杂的比特币生还计划，可以通过设置多重签名，做好遗产规划，并通过专门的“数字资产执行者”律师处理后事。

10.3 总结

比特币是一项全新的，前所未有的，复杂的技术。随着时间的推移，我们将开发出更好的安全工具，而且更容易被专业人士使用的做法。而现在，比特币用户可以使用许多这里所讨论的技巧，享受安全而无困扰的比特币生活。

附录1 交易脚本的操作符、常量和符号

表A-1列出了将脚本压入堆栈的操作符。

表A-1 入栈操作

关键字	值（十六进制）	描述
OP_0 or OP_FALSE	0x00	一个字节空串被压入堆栈中
1-75	0x01-0x4b	把接下来的N个字节压入堆栈中，N 的取值在 1 到 75 之间
OP_PUSHDATA1	0x4c	下一个脚本字节包括N，会将接下来的N个字节压入堆栈
OP_PUSHDATA2	0x4d	下两个脚本字节包括N，会将接下来的N个字节压入堆栈
OP_PUSHDATA4	0x4e	下四个脚本字节包括N，会将接下来的N个字节压入堆栈
OP_1NEGATE	0x4f	将脚本-1压入堆栈
OP_RESERVED	0x50	终止 - 交易无效（除非在未执行的 OP_IF 语句中）
OP_1 or OP_TRUE	0x51	将脚本1压入堆栈
OP_2 to OP_16	0x52	将脚本N压入堆栈，例如 OP_2 压入脚本“2”

表A-2列出了有条件的流量控制的操作符。

表A-2 有条件的流量控制操作

关键字	值（十六进制）	描述
OP_NOP	0x61	无操作
OP_VER	0x62	终止 - 交易无效（除非在未执行的 OP_IF 语句中）
OP_I	0x63	如果栈项元素值为0，语句将被执行
OP_NOTIF	0x64	如果栈项元素值不为0，语句将被执行
OP_VERIF	0x65	终止 - 交易无效
OP_VERNOTIF	0x66	终止 - 交易无效
OP_ELSE	0x67	如果前述的OP_IF或OP_NOTIF或OP_ELSE未被执行，这些语句就会被执行
OP_ENDIF	0x68	终止 OP_IF, OP_NOTIF, OP_ELSE 区块
OP_VERIFY	0x69	如果栈项元素值非真，则标记交易无效
OP_RETURN	0x6a	标记交易无效

表A-3列出了控制堆栈的操作符。

表A-3 堆栈操作

关键字	值（十六进制）	描述
OP_TOALTSTACK	0x6b	从主堆栈中取出元素，推入辅堆栈。
OP_FROMALTSTACK	0x6c	从辅堆栈中取出元素，推入主堆栈
OP_2DROP	0x6d	删除栈顶两个元素

OP_2DUP	0x6e	复制栈顶两个元素
OP_3DUP	0x6f	复制栈顶三个元素
OP_2OVER	0x70	把栈底的第三、第四个元素拷贝到栈顶
OP_2ROT	0x71	移动第五、第六元素到栈顶
OP_2SWAP	0x72	如果栈项元素值不为0，复制该元素值
OP_IFDUP	0x73	如果栈项元素值不为0，复制该元素值
OP_DEPTH	0x74	把堆栈元素的个数压入堆栈
OP_DROP	0x75	删除栈顶元素
OP_DUP	0x76	复制栈顶元素
OP_NIP	0x77	删除栈顶的下一个元素
OP_OVER	0x78	复制栈顶的下一个元素到栈顶
OP_PICK	0x79	把堆栈的第n个元素拷贝到栈顶
OP_ROLL	0x7a	把堆栈的第n个元素移动到栈顶
OP_ROT	0x7b	翻转栈顶的三个元素
OP_SWAP	0x7c	栈顶的三个元素交换
OP_TUCK	0x7d	拷贝栈顶元素并插入到栈顶第二个元素之后

表A-4列出了字符串操作符。

表A-4 字符串接操作

关键字	值 (十六进制)	描述
OP_CAT	0x7e	连接两个字符串，已禁用
OP_SUBSTR	0x7f	返回字符串的一部分，已禁用
OP_LEFT	0x80	在一个字符串中保留左边指定长度的子串，已禁用
OP_RIGHT	0x81	在一个字符串中保留右边指定长度的子串，已禁用
OP_SIZE	0x82	把栈顶元素的字符串长度压入堆栈

表A-5列出了二进制算术和布尔逻辑运算符演示。

表A-5 二进制算术和条件

关键字	值 (十六进制)	描述
OP_INVERT	0x83	所有输入的位取反，已禁用
OP_AND	0x84	对输入的所有位进行布尔与运算，已禁用
OP_OR	0x85	对输入的每一位进行布尔或运算，已禁用
OP_XOR	0x86	对输入的每一位进行布尔异或运算，已禁用
OP_EQUAL	0x87	如果输入的两个数相等，返回1，否则返回0
OP_EQUALVERIFY	0x88	与OP_EQUAL一样，如结果为0，之后运行OP_VERIFY
OP_RESERVED1	0x89	终止 - 无效交易（除非在未执行的OP_IF语句中）
OP_RESERVED2	0x8a	终止-无效交易（除非在未执行的OP_IF语句中）

表A-6列出了数值(算法)操作符。

表A-6 数值操作

关键字	值 (十六进制)	描述
OP_1ADD	0x8b	输入值加1
OP_1SUB	0x8c	输入值减1
OP_2MUL	0x8d	无效 (输入值乘2)
OP_2DIV	0x8e	无效 (输入值除2)
OP_NEGATE	0x8f	输入值符号取反
OP_ABS	0x90	输入值符号取正
OP_NOT	0x91	如果输入值为0或1，则输出1或0；否则输出0
OP_0NOTEQUAL	0x92	输入值为0输出0；否则输出1
OP_ADD	0x93	输出输入两项之和
OP_SUB	0x94	输出输入 (第二项减去第一项) 之差
OP_MUL	0x95	禁用 (输出输入两项的积)
OP_DIV	0x96	禁用 (输出用第二项除以第一项的倍数)
OP_MOD	0x97	禁用 (输出用第二项除以第一项得到的余数)
OP_LSHIFT	0x98	禁用 (左移第二项，移动位数为第一项的字节数)
OP_RSHIFT	0x99	禁用 (右移第二项，移动位数为第一项的字节数)
OP_BOOLAND	0x9a	两项都不会为0，输出1，否则输出0
OP_BOOLOR	0x9b	两项有一个不为0，输出1，否则输出0
OP_NUMEQUAL	0x9c	两项相等则输出1，否则输出为0
OP_NUMEQUALVERIFY	0x9d	和 NUMEQUAL 相同，如结果为0运行OP_VERIFY
OP_NUMNOTEQUAL	0x9e	如果栈顶两项不是相等数的话，则输出1
OP_LESS THAN	0x9f	如果第二项小于栈顶项，则输出1
OP_GREATER THAN	0xa0	如果第二项大于栈顶项，则输出1
OP_LESS THANOREQUAL	0xa1	如果第二项小于或等于第一项，则输出1
OP_GREATER THANOREQUAL	0xa2	如果第二项大于或等于第一项，则输出1
OP_MIN	1:26	输出栈顶两项中较小的一项
OP_MAX	1:27	输出栈顶两项中较大的一项
OP_WITHIN	1:28	如果第三项的数值介于前两项之间，则输出1

表A-7列出了加密函数操作符。

表A-7 加密和散列操作

关键字	值 (十六进制)	描述
OP_RIPEMD160	0xa6	返回栈顶项的 RIPEMD160 哈希值

OP_SHA1	0xa7	返回栈顶项 SHA1 哈希值
OP_SHA256	0xa8	返回栈顶项 SHA256 哈希值
OP_HASH160	0xa9	栈顶项进行两次HASH，先用SHA-256，再用RIPEMD-160
OP_HASH256	0xaa	栈顶项用SHA-256算法HASH两次
OP_CODESEPARATOR	0xab	标记已进行签名验证的数据
OP_CHECKSIG	0xac	交易所用的签名必须是哈希值和公钥的有效签名，如果为真，则返回1
OP_CHECKSIGVERIFY	0xad	与CHECKSIG一样，但之后运行OP_VERIFY
OP_VERIFY	0xae	对于每对签名和公钥运行CHECKSIG。所有的签名要与公钥匹配。因为存在BUG，一个未使用的外部值会从堆栈中删除。
OP_CHECKMULTISIGVERIFY	0xaf	与 CHECKMULTISIG 一样，但之后运行OP_VERIFY

表A-8列出了非操作符。

表A-8 非操作

关键字	值（十六进制）	描述
OP_NOP1-OP_NOP10	0xb0-0xb9	无操作 忽略

表A-9保留关键字，仅供内部脚本调试。

表A-9 仅供内部使用的保留关键字

关键字	值（十六进制）	描述
OP_SMALLDATA	0xf9	代表小数据域
OP_SMALLINTEGER	0xfa	代表小整数数据域
OP_PUBKEYS	0xfb	代表公钥域
OP_PUBKEYHASH	0xfd	代表公钥哈希域
OP_PUBKEY	0xfe	代表公钥域
OP_INVALIDOPCODE	0xff	代表当前未指定的操作码

附录2 比特币改进协议

比特币改进协议（Bitcoin improvement proposals 简称BIP）是为比特币社区提供规范，完善比特币及其运行进程和外部环境特性的设计指导文件。依据 BIP0001 协议即比特币改进协议的目的与指南，比特币改进协议有以下三种类型：

标准协议（Standard BIP）

描述任何影响大多或全部比特币应用的变化，比如网络协议、交易有效性规则的变化，或者任何影响使用比特币交互操作性变化或补充。

信息补充协议（Informational BIP）

描述比特币的设计事项而不是为其提供新特性，或者为比特币社区提供一般性的指南或信息。信息补充型协议不一定需要比特币社区达成共识或推荐，因此用户和开发人员可以选择忽略或者接受信息补充型协议的建议。

开发指导协议（Process BIP）

描述比特币进程，或者提议更改进程或事项。Process BIP与Standard BIP相似，但是也可以应用于除比特币协议以外的领域。在普遍达成共识的情况下，它可以向比特币以外的代码库提出改进建议。与Informational BIP不同，Process BIP是强制性的，用户必须遵守。例如针对决策进程的过程、指南、改变，在比特币开发过程中使用的工具、环境的改变。任何meta-BIP也应被认为是Process BIP。比特币改进协议在 GitHub 中更新版本。

表B-1为比特币改进协议一览表（更新至2014年底）。需要有关目前 BIP 内容的最新信息，请参考官方版本。

表B-1 BIP一览表

BIP#	链接	标题	作者	类型
1	https://github.com/bitcoin/bips/blob/master/bip-0001.mediawiki	BIP Purpose and Guidelines	Amir Taaki	Standard
10	https://github.com/bitcoin/bips/blob/master/bip-0010.mediawiki	Multi-Sig Transaction Distribution	Alan Reiner	Informational
11	https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki	M-of-N Standard Transactions	Gavin Andresen	Standard
12	https://github.com/bitcoin/bips/blob/master/bip-0012.mediawiki	OP_EVAL	Gavin Andresen	Standard
13	https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki	Address Format for pay-to-script-hash	Gavin Andresen	Standard
14	https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki	Protocol Version and User Agent	Amir Taaki, Patrick	Standard
15	https://github.com/bitcoin/bips/blob/master/bip-0015.mediawiki	Aliases	Amir Taaki	Standard
16	https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki	Pay To Script Hash	Gavin Andresen	Standard
17	https://github.com/bitcoin/bips/blob/master/bip-0017.mediawiki	OP_CHECKHASHVERIFY (CHV)	Luke Dashjr	Withdrawal
18	https://github.com/bitcoin/bips/blob/master/bip-0018.mediawiki	hashScriptCheck	Luke Dashjr	Standard
19	https://github.com/bitcoin/bips/blob/master/bip-0019.mediawiki	M-of-N Standard Transactions (Low SigOp)	Luke Dashjr	Standard
20	https://github.com/bitcoin/bips/blob/master/bip-0020.mediawiki	URI Scheme	Luke Dashjr	Standard
			Nils	

21	https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki	URI Scheme	Schneider, Matt Corallo	Standard
22	https://github.com/bitcoin/bips/blob/master/bip-0022.mediawiki	getblocktemplate - Fundamentals	Luke Dashjr	Standard
23	https://github.com/bitcoin/bips/blob/master/bip-0023.mediawiki	getblocktemplate - Pooled Mining	Luke Dashjr	Standard
30	https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki	Duplicate transactions	Pieter Wuille	Standard
31	https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki	Pong message	Mike Hearn	Standard
32	https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki	Hierarchical Deterministic Wallets	Pieter Wuille	Informatic
33	https://github.com/bitcoin/bips/blob/master/bip-0033.mediawiki	Stratized Nodes	Amir Taaki	Standard
34	https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki	Block v2, Height in coinbase	Gavin Andresen	Standard
35	https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki	mempool message	Jeff Garzik	Standard
36	https://github.com/bitcoin/bips/blob/master/bip-0036.mediawiki	Custom Services	Stefan Thomas	Standard
37	https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki	Bloom filtering	Mike Hearn and Matt Corallo	Standard
38	https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki	Passphrase-protected private key	Mike Caldwell	Standard
39	https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki	Mnemonic code for generating deterministic keys	Slush	Standard
40	-	Stratum wire protocol	Slush	Standard
41	-	Stratum mining protocol	Slush	Standard
42	https://github.com/bitcoin/bips/blob/master/bip-0042.mediawiki	A finite monetary supply for bitcoin	Pieter Wuille	Standard
43	https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki	Purpose Field for Deterministic Wallets	Slush	Standard
44	https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki	Multi-Account Hierarchy for Deterministic Wallets	Slush	Standard
50	https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki	March 2013 Chain Fork Post- Mortem	Gavin Andresen	Informatic
60	https://github.com/bitcoin/bips/blob/master/bip-0060.mediawiki	Fixed Length "version" Message (Relay-Transactions Field)	Amir Taaki	Standard
61	https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki	"reject" P2P message	Gavin Andresen	Standard
62	https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki	Dealing with malleability	Pieter Wuille	Standard

63	-	Stealth Addresses	Peter Todd	Standard
64	https://github.com/bitcoin/bips/blob/master/bip-0064.mediawiki	getutxos message	Mike Hearn	Standard
70	https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki	Payment protocol	Gavin Andresen	Standard
71	https://github.com/bitcoin/bips/blob/master/bip-0071.mediawiki	Payment protocol MIME types	Gavin Andresen	Standard
72	https://github.com/bitcoin/bips/blob/master/bip-0072.mediawiki	Payment protocol URIs	Gavin Andresen	Standard
73	https://github.com/bitcoin/bips/blob/master/bip-0073.mediawiki	Use "Accept" header with Payment Request URLs	Stephen Pair	Standard

附录3 pycoin库、实用密钥程序ku和交易程序tx

pycoin库起初由Richard Kiss 撰写和维护，基于 Python 脚本的处理交易密钥，不仅支持比特币交易，也支持其他非标准脚本语言的交易类型。

pycoin 库支持 Python 2 (版本 2.7.x) 和Python 3 (版本 3.3 以后)。下面介绍一些好用的命令行使用程序——ku 和 tx。

实用密钥程序 (KU)

命令行实用程序 KU (key utility 缩写)对于处理密钥而言，就如同瑞士军刀一样灵活有用。它支持 BIP32 密钥、WIF 和地址 (比特币以及竞争币均可)。下面是一些例子。

使用默认的 GPG 热池和系统随机数设备 (/dev/random) 来创建BIP32 密钥，如下：

```
$ ku create
input : create
network : Bitcoin
wallet key : xprv9s21ZrQH143K3LU5ctPZTBnb9kTjA5Su9DcWhVXJemiJBsY7VqXUG7hipgdWaU
m2nhnzdvxJf5KJ09vjP2nABX65c5sFsWsV8oXcbpehtJi
public version : xpub661MyMwAqRbcFpYYiuvZpKjKhnJDZYAkWSY76JvvD7FH4fsG3Nqiov2CfxzxY8
DGcpfT56AMFeo8M8KPkFMfLutvwjwb6WPv8rY65L2q8Hz
tree depth : 0
fingerprint : 9d9c6092
parent f'print : 00000000
child index : 0
chain code :
80574fb260edaa4905bc86c9a47d30c697c50047ed466c0d4a5167f6821e8f3c
private key : yes
secret exponent :
112471538590155650688604752840386134637231974546906847202389294096567806844862
hex :
f8a8a28b28a916e1043cc0aca52033a18a13cab1638d544006469bc171fddfbe
wif : L5Z54xi6qJusQT42JHA44mfPVZGjyb4XBRWfxAzUWwRiGx1kV4sP
uncompressed : 5KhoEavGNHH4GHKoy2Ptu4KfdNp4r56L5B5un8FP6RZnbsz5Nmb
public pair x :
76460638240546478364843397478278468101877117767873462127021560368290114016034
public pair y :
59807879657469774102040120298272207730921291736633247737077406753676825777701
x as hex :
a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
y as hex :
843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcfd625
y parity : odd
key pair as sec :
03a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
uncompressed :
04a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcfd625
hash160 : 9d9c609247174ae323acfc96c852753fe3c8819d
uncompressed : 8870d869800c9b91ce1eb460f4c60540f87c15d7
Bitcoin address : 1FNNRQ5fSv1wBi5gyfVBs2rkNheMGt86sp
uncompressed : 1DSS5isnh4FsvaLVjevXewvSpfqktdiQAM
```

使用口令创建一个 BIP 32 密钥：



本例中的口令很容易猜到。

```

$ ku P:foo
input : P:foo
network : Bitcoin
wallet key :
xprv9s21ZrQH143K31AgNK5pyVvw23gHnkBq2wh5aEk6g1s496M8ZMjxncCKZKgb5j
ZoY5eSJMJ2Vbyvi2hbmQnCuHBujZ2wXGTux1X2k9Krdtq
public version : xpub661MyMwAqRbcFVF9ULcqlLdsEa5wCCugQAcghd91EMQ31tgH6u4DLQWQayvtS
VYFvxZ2vPPpbXE1qpjoUFidhjFj82pVShwu9curWmb2zy
tree depth : 0
fingerprint : 5d353a2e
parent f'print : 00000000
child index : 0
chain code :
5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc
private key : yes
secret exponent :
65825730547097305716057160437970790220123864299761908948746835886007793998275
hex :
91880b0e3017ba586b735fe7d04f1790f3c46b818a2151fb2def5f14dd2fd9c3
wif : L2c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
uncompressed : 5jVNzA5vXDoKYJdw8SwwLHxUxalvn9mDea6k1vRPCX7KLUVWa7W
public pair x :
81821982719381104061777349269130419024493616650993589394553404347774393168191
public pair y :
58994218069605424278320703250689780154785099509277691723126325051200459038290
x as hex :
b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
y as hex :
826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
y parity : even
key pair as sec :
02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
uncompressed :
04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
hash160 : 5d353a2ecdb262477172852d57a3f11de0c19286
uncompressed : e5bd3a7e6cb62b4c820e51200fb1c148d79e67da
Bitcoin address : 19Qc8uLTfUonmxUEZac7fz1M5c5ZZbAii
uncompressed : 1MwkRkogzBRMehBntgcq2aJhXCStJTXHT

```

以 JSON 格式得到信息：

```

$ ku P:foo -P -j
{
  "y_parity": "even",
  "public_pair_y_hex": "826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52" ,
  "private_key": "no" ,
  "parent_fingerprint": "00000000" ,
  "tree_depth": "0" ,
  "network": "Bitcoin" ,
  "btc_address_uncompressed": "1MwkRkogzBRMehBntgcq2aJhXCStJTXHT" ,
  "key_pair_as_sec_uncompressed": "04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f826d8b4d3010aea16ff4c1c1d3" ,
  "public_pair_x_hex": "b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f" ,
  "wallet_key": "xpub661MyMwAqRbcFVF9ULcqlLdsEa5wCCugQAcghd91EMQ31tgH6u4DLQWQayvtSVYFvxZ2vPPpbXE1qpjoUFidhjFj82pVShwu9curWmb2" ,
  "chain_code": "5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc" ,
  "child_index": "0" ,
  "hash160_uncompressed": "e5bd3a7e6cb62b4c820e51200fb1c148d79e67da" ,
  "btc_address": "19Qc8uLTfUonmxUEZac7fz1M5c5ZZbAii" ,
  "fingerprint": "5d353a2e" ,
  "hash160": "5d353a2ecdb262477172852d57a3f11de0c19286" ,
  "input": "P:foo" ,
  "public_pair_x": "81821982719381104061777349269130419024493616650993589394553404347774393168191" ,
  "public_pair_y": "58994218069605424278320703250689780154785099509277691723126325051200459038290" ,
  "key_pair_as_sec": "02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f"
}

```

BIP32 公钥：

```
$ ku -w -P P:foo
```

```
xpub661MyMwAqRbcFVF9ULcqLdsEa5WhCCugQAcgNd9iEMQ31tgH6u4DLQoQayvtSVYFvXz2vPPpbXE1qpjouFidhjFj82pVShlu9curWmb2zy
```

生成一个子密钥：

```
$ ku -w -s3/2 P:foo  
xprv9wTERTSkjVyJa1v4cUTMFkWMe5eu8ErbQcs9xajnsUzCBT7ykHAwdrxvG3g3f6BFk7ms5hHBvmbdutNmyg6iogWKxx6mefEw4M8EroLgKj
```

加强型子密钥：

```
$ ku -w -s3/2H P:foo  
xprv9wTERTSu5AWGkDeUPmqBcbZWx1xq85ZNX9iQRQw9DXwygFp7iRGJ079dsVctcsCHsnZ3XU3DhsuaGZbDh8iDkBN45k67UKsJUXM1JfRCdn1
```

WIF：

```
$ ku -W P:foo  
L26C3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
```

地址：

```
$ ku -a P:foo  
19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
```

生成一串子密钥：

```
$ ku P:foo -s 0/0-5 -w  
xprv9wkbDfyBXmZjBG9EiBp67KK72fphUp9utJokEBFtjsjiuKUUDF5V3TU8U8cDzytqYnSekc8bYuJS8G3bhXxKwB89Ggn2dzLcoJsuEdRK  
xprv9wkbDfyBXmZnzkf3bAGifK593gT7WJZPrnYAmvc77gUQVej5QHckc5Adtxxa28ACmAni9XhCrRvtFqQcUxt8rUgFz3souMiDdwxDZnQzx  
xprv9wkbDfyBXmZqdxA8y4Swqfbdy71gsW9sjx93pC1JE1bwSMQyRxan6srXUPBtj3PTxQFKZJAiwoUpmvtrxkZu4zfnsr3pqyy2vthpkwuoVq  
xprv9wkbDfyBXmZsA85GyWj9uYPyoQv826YAadKwMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p  
xprv9wkbDfyBXmZv2q3N6hhZ8DACEnQDnXML1J62krJAcf7xb1Hjwu2VMJQrCofY2jtFXdiEY8UsRNJfqK6DAdyZXoMvtaLHywQx3FS4A9zw  
xprv9wkbDfyBXmZw4jEYXUHYc9fT25k9irP87n2rqfJ5bqbjKdT84Mm7Wtc2xmzFuKg71Yf7XFHkkSsaYKwKjbR54bnvAD9GzjUYbAYTtN4ruo
```

生成相关地址：

```
$ ku P:foo -s 0/0-5 -a 1MrjE78H1R1rqdFrmkjdhnPUDLCJALbv3x 1AnYyVEcuqeoVzH96zj1eYKwoWfwte2pxu  
1GXr1kZfxE1FcK6ZRD5sqqq5YfvuzA1Lb 116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK  
1Cz2rTLjRM6pMnxPNrRKp9ZSvRtj5dDUML 1WstdwPnU6HEUPme1DQayN9nm6j7nDVEM
```

生成对应的 WIF：

```
$ ku P:foo -s 0/0-5 -w  
L5a4iE5k9gcJKGqX3FwmxzBYQc29PvZ6pgBaePLVqT5YByEnBomx  
Kyjgne6GZwPGB66kJEhoPbmyjMP7D5d3zRbHvjwcq4iQD9QqKQ  
L4B3ygQxK6zh2NQxLDee2H9v4Lvg14cLJw7QwWPzCtKhdwMaQz  
L2L2PZdorybUqkPjrhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF  
L2oD6vA4TUyqPF8G4vhUFSGwCyuuuvFZ3v8SKHYFDwkbM765Nrfd  
KzChTbc3kZFxUSJ3Kt54cxsogefAD9CCM4zGB22s18nfKcThQn8c
```

通过选择 BIP32 字符串（和子密钥 0/3 相关的那个串）检查是否起作用：

```
$ ku -W xprv9wkbDfyBXmZsA85GyWj9uYPyoQv826YAadKwMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p  
L2L2PZdorybUqkPjrhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
```

```
$ ku -a xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKwMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p  
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
```

好了，看上去很熟悉了。

从秘密指数：

```
$ ku 1  
input : 1  
network : Bitcoin  
secret exponent : 1  
hex : 1  
wif : KwDiBF890gGbjEhKnhXJuH7LrciVrZi3qYjgd9M7rFU73sVHnowh  
uncompressed : 5HpHagT65TZZG1PH3CSu63k8DbpvD8s5ip4nEB3kEsreAnchuDf  
public pair x :  
5506626302227734366957871889516853432625060345377594175500187360389116729240  
public pair y :  
32670510020758816978083085130507043184471273380659243275938904335757337482424  
x as hex :  
79be667ef9dcbbac55a06295ce870b07029bfcd2dce28d959f2815b16f81798  
y as hex :  
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8  
y parity : even  
key pair as sec :  
0279be667ef9dcbbac55a06295ce870b07029bfcd2dce28d959f2815b16f81798  
uncompressed :  
0479be667ef9dcbbac55a06295ce870b07029bfcd2dce28d959f2815b16f81798  
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8  
hash160 : 751e76e8199196d454941c45d1b3a323f1433bd6  
uncompressed : 91b24bf9f5288532960ac687abb035127b1d28a5  
Bitcoin address : 1BqGZ9tcN4rm9KBzDn7KprQz87Sz26SAMH  
uncompressed : 1EHNa6Q4Jz2uvNEExL497mE43ikXhwF6kZm
```

莱特币版本：

```
$ ku -nL 1  
input : 1  
network : Litecoin  
secret exponent : 1  
hex : 1  
wif : T33ydQRKp4FCw5LCLLUB7deioUMoveiwekdwUwyfRDeGzm76aUjV  
uncompressed : 6u823ozcyt2rjPH8Z2ErsSXJB5PPQwK7VVTwwN4mxLBFrao69XQ  
public pair x :  
5506626302227734366957871889516853432625060345377594175500187360389116729240  
public pair y :  
32670510020758816978083085130507043184471273380659243275938904335757337482424  
x as hex :  
79be667ef9dcbbac55a06295ce870b07029bfcd2dce28d959f2815b16f81798  
y as hex :  
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8  
y parity : even  
key pair as sec :  
0279be667ef9dcbbac55a06295ce870b07029bfcd2dce28d959f2815b16f81798  
uncompressed :  
0479be667ef9dcbbac55a06295ce870b07029bfcd2dce28d959f2815b16f81798  
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8  
hash160 : 751e76e8199196d454941c45d1b3a323f1433bd6  
uncompressed : 91b24bf9f5288532960ac687abb035127b1d28a5  
Litecoin address : LVuDpNCSSj6pQ7t9Pv6d6SukLKoqDEVnJ  
uncompressed : LYwKqJhtPeGyBAw7WC8R3F7ovxtzAiubdM
```

狗狗币 WIF:

```
$ ku -nD -W 1  
QNcdLVw8fHkixm6NNyN6nVwxKek4u7qrioRbQmjxac5TVoTtZuot
```

来自公用对 (Testnet 上) :

```
$ ku -nT
55066263022277343669578718895168534326250603453777594175500187360389116729240, ev
en
input :
55066263022277343669578718895168534326250603453777594175500187360389116729240
89116729240, even
network : Bitcoin testnet
public pair x :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y :
32670510020758816978083085130507043184471273380659243275938904335757337482424
252 | Appendix C: pycoin, ku, and tx
x as hex :
79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
y as hex :
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity : even
key pair as sec :
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
uncompressed :
0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160 : 751e76e8199196d454941c45d1b3a323f1433bd6
uncompressed : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin testnet address : mrCDrCyB6J1vRfbwM5hemdJz73FwDBC8r
uncompressed : mtoKs9V381UAhUia3d7Vb9GNak8Qvmcsme
```

来自 hash160:

```
$ ku 751e76e8199196d454941c45d1b3a323f1433bd6
input : 751e76e8199196d454941c45d1b3a323f1433bd6
network : Bitcoin
hash160 : 751e76e8199196d454941c45d1b3a323f1433bd6
Bitcoin address : 1Bg6Z9tcN4rm9KBzDn7KprQz87Sz26SAMH
```

作为狗狗币地址

```
$ ku -nD 751e76e8199196d454941c45d1b3a323f1433bd6
input : 751e76e8199196d454941c45d1b3a323f1433bd6
network : Dogecoin
hash160 : 751e76e8199196d454941c45d1b3a323f1433bd6
Dogecoin address : DFpN6QqFfUm3gKNaxN6tNcab1FArL9cZL
```

实用交易程序 (TX)

命令行实用程序 tx 可将交易以一种易读的方式呈现，还可以从 pycoin 的交易缓存或者网络服务（目前支持 blockchain.info, blockr.io, and biteeasy.com）中获取原始交易，合并交易，添加或删除输入或输出，以及签署交易。

下面是一些例子。

看看有名的“披萨”交易[PIZZA]:

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: consider setting environment variable PYCOIN_CACHE_DIR=~/pycoin_cache
to cache transactions fetched via web services
warning: no service providers found for get_tx; consider setting environment
variable PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
usage: tx [-h] [-t TRANSACTION_VERSION] [-l LOCK_TIME] [-n NETWORK] [-a]
[-i address] [-f path-to-private-keys] [-g GPG_ARGUMENT]
Key Utility (KU) | 253
[--remove-tx-in tx_in_index_to_delete]
```

```
[--remove-tx-out tx_out_index_to_delete] [-F transaction-fee] [-u]
[-b BITCOIND_URL] [-o path-to-output-file]
argument [argument ...]
tx: error: can't find Tx with id
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
```

糟糕！我们没有设置好网络服务。让我们现在设置：

```
$ PYCOIN_CACHE_DIR=~/pycoin_cache
$ PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
$ export PYCOIN_CACHE_DIR PYCOIN_SERVICE_PROVIDERS
```

这不是自动完成的，所以这种命令行工具不会泄漏你在第三方网站交易的隐私信息。如果你想忽略这个提醒，就可以把这些命令行加入到profile文件。

我们再试一次：

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
Version: 1 tx hash
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a 159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  0: (unknown) from
  1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total output 10000000.00000 mBTC
including unspents in hex dump since transaction not fully signed
01000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a493046022100a7f26eda8749
31999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e9199238eb73f07c8f2
09504c84b80f03e30ed8169edd44f80ed17ddf451901fffffff010010a5d4e80000001976a9147e
c1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000
** can't validate transaction as source transactions missing
```

出现最后一行是为了验证交易签名，严格说来您需要源交易。因此让我们通过添加 -a 指令来给交易补充源信息：

```
$ tx -a 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: transaction fees recommendations casually calculated and estimates may
be incorrect
warning: transaction fee lower than (casually calculated) expected value of 0.1
mBTC, transaction might not propagate
Version: 1 tx hash
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a 159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  254 | Appendix C: pycoin, ku, and tx
  0: 17WFx2GQZUmh6Up2NDNCEdk3deYomdNCfk from
  1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0
  10000000.00000 mBTC sig ok
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total input 10000000.00000 mBTC
Total output 10000000.00000 mBTC
Total fees 0.00000 mBTC
01000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a493046022100a7f26eda8749
31999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e9199238eb73f07c8f2
09504c84b80f03e30ed8169edd44f80ed17ddf451901fffffff010010a5d4e80000001976a9147e
c1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000
all incoming transaction values validated
```

接下来，我们看看尚未使用完的输出的专用地址的（UTXO）。在区块 #1，我们看到到 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX 的 coinbase 交易。我们用 fetch_unspent 命令找到这个地址下的所有比特币。

```
$ fetch_unspent 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX
a3a6f902a51a2cbebede144e48a88c05e608c2cce28024041a5b9874013a1e2a/
0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/333000
cea36d008badf5c7866894b191d3239de9582d89b6b452b596f1f1b76347f8cb/
31/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/10000
065ef6b1463f552f675622a5d1fd2c08d6324b4402049f68e767a719e2049e8d/
86/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/10000
a66ddd42f9f2491d3c336ce5527d45cc5c2163aaed3158f81dc054447f447a2/0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/10000
ffd901679de65d4398d090cefe68d2c3ef073c41f7e8dbec2fb5cd75fe71dfe7/0/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/100
d658ab87cc053b8dbcf4aa2717fd23cc3edfe90ec75351fadd6a0f7993b461d/
5/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/911
36ebe0ca3237002acb12e1474a3859bde0ac84b419ec4ae373e63363ebef731c/
1/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/100000
fd87f9addeb17f4eb1673da76ff48ad29e64b7afa02fda0f2c14e43d220fe24/0/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/1
fdf0b375a987f17056e5e919ee6eadd87dad36c09c4016d4a03cea15e5c05e3/1/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/1337
cb2679bfd0a557b2dc0d8a6116822f3fcbe281ca3f3e18d3855aa7ea378fa373/0/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/1337
d6be34ccf6edddc3cf69842dce99fe503bf632ba2c2adb0f95c63f6706ae0c52/1/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/2000000
0e3e2357e806b6cd1f70b54c3a3a17b6714ee1f0e68beb44a74b1efd512098/0/410496b538e85
3519c726a2c91e61ec11600ae1390813a627c66fb8be7947be63c52da7589379515d4e0a604f8141
781e62294721166bf621e73a82cbf2342c858eeac/50000000000
```

附录4 sx工具下一些的命令

sx 命令如下：

不赞成 DEPRECATED ELECTRUM 钱包式的确定密钥和地址 genaddr 由钱包种子或主公钥来生成一个比特币地址。 genpriv 只由种子来生成一个私钥。 genpub 由钱包种子或主公钥来生成一个公钥。 mpk 只由钱包种子来生成主公钥。 newseed 创建一个新的决定性的钱包种子。

试验中的 EXPERIMENTAL APPS wallet 试验性的命令行钱包。

离线区块链 OFFLINE BLOCKCHAIN 区块头 showblkhead 显示区块头细节

离线密钥和地址 OFFLINE KEYS AND ADDRESSES 基础操作 addr 查看公钥或私钥的比特币地址。 embed-addr 将记录数据嵌入区块链，生成新的地址。 get-pubkey 如果可行的话，得到地址公钥。 newkey 创建一个新私钥。 pubkey 查看私钥的公开部分。 validaddr 确认一个地址有效。

记忆存储 BRAIN STORAGE brainwallet 从任意口令生成256位比特币私钥。 mnemonic 从128位electrum钱包或bip32种子生成12助记词。

HD / BIP32 hd-priv 从一个HD私钥创建新的HD 私钥。 hd-pub 从一个HD私钥或公钥创建一个新的HD公钥。 hd-seed 创建一个随机的新HD密钥。 hd-to-address 将 HD 公钥或私钥转为比特币地址。 hd-to-wif 将HD私钥转为WIF私钥。

MULTISIG ADDRESSES scripthash 从原始十六进制脚本创建BIP16脚本哈希值。

隐身STEALTH stealth-addr 从给定输入查看隐身地址。 stealth-initiate 初始化新的隐身支付。 stealth-newkey 生成新的隐身密钥和地址。 stealth-show-addr 显示具体隐身地址。 stealth-uncover 揭开隐身地址。 stealth-uncover-secret 放弃隐身。

离线交易 OFFLINE TRANSACTIONS mktx 创建一个未签名的tx。 rawscript 从脚本创建未处理的十六进制请求。 set-input 设置交易输入。 showscript 显示脚本细节。 showtx 显示交易细节。 sign-input 输入交易签名。 unwrap 使验证码有效，从原始十六进制串恢复版本字节和原始数据。 validsig 确认交易输入签名有效。 wrap 向十六进制串添加版本字节和验证码。

ONLINE (BITCOIN P2P) 区块链更新 BLOCKCHAIN UPDATES sendtx-node 把交易导入单个节点。 sendtx-p2p 把tx传向比特币网络。

ONLINE (BLOCKCHAIN.INFO) BLOCKCHAIN QUERIES (blockchain.info) bci-fetch-last-height 使用 blockchain.info 获取最后一个区块高度。 bci-history 从 blockchain.info 得到输出点、价值、消费总额的列表。

区块链更新 BLOCKCHAIN UPDATES sendtx-bci 把tx传向blockchain.info/pushtx。

ONLINE (BLOCKEXPLORER.COM) BLOCKCHAIN QUERIES (blockexplorer.com) blke-fetch-transaction 从 blockexplorer.com 获取交易

ONLINE (OBELISK) BLOCKCHAIN QUERIES balance 以聪为单位显示一个比特币地址的余额。 fetch-block-header 获取区块头。 fetch-last-height 获取最后的区块高度。 fetch-stealth 网络连接obelisk load balancer后台发出请求以获取隐身信息。 fetch-transaction 用网络连接向 obelisk load balancer后台请求以获取未处理交易。 fetch-transaction-index 在交易区块里获取区块高度和索引。 get-utxo 从给定地址集合里得到足够的尚未动用的交易输出，用以支付给定数量的聪。 history 从地址得到输出点、价值、消费总额列表。 grep命令可以过滤未动用的输出，其结果可以被mktx命令调用。 validtx 确认交易有效。 BLOCKCHAIN UPDATES sendtx-obelisk 将tx传送到 obelisk 服务器。 BLOCKCHAIN WATCHING monitor 监控一个地址。 watchtx 通过网络搜索特定哈希值来查看交易。

OBELISK ADMIN initchain 初始化新区块链。

UTILITY EC MATH ec-add-modp 计算整数和整数相加结果。 ec-multiply 整数和点的乘积 ec-tweak-add 计算 POINT +

INTEGER * G 结果。

FORMAT (BASE 58) base58-decode 从 base58 转为十六进制。 base58-encode 从 16 进制转为 base58。

FORMAT (BASE58CHECK) base58check-decode 从 base58check 转为十六进制。 base58check-encode 从十六进制转为 base58check。 decode-addr 将地址从 base58check 形式解码为内部 RIPEMD 表达。 encode-addr 将地址从内部 RIPEMD 编码为 base58check 形式。

FORMAT (WIF) secret-to-wif 将秘密指数转为WIF。 wif-to-secret 将WIF转为秘密指数。

HASHES ripemd-hash 从 STDIN 转为RIPEMD 哈希值。 sha256 取得数据的SHA256哈希值。

MISC qrcode 生成比特币离线二维码。 SATOSHI MATH btc 转换聪币为比特币 satoshi 将比特币转换为聪币。

输入“sx help COMMAND”，可以了解命令的具体信息。接下来，我们看一些使用 sx 工具操作密钥和地址的例子。用 newkey 命令利用系统的随机数生成器生成一个新的私钥。将标准输出存入 private_key 文件。

```
$ sx newkey > private_key $ cat private_key 5Jgx3UAaXw8AcCQCi1j7uaTaqpz2fqNR9K3r4apxdYn6rTzR1PL
```

现在，用 pubkey 命令将前面生成的私钥转成公钥。将 private_key 文件作为标准输入，以标准输出的方式导出到新文件 public_key 。

```
$ sx pubkey < private_key > public_key $ cat public_key  
02fca46a6006a62dfdd2dbb2149359d0d97a04f430f12a7626dd409256c12be500
```

我们可以用 addr 命令将 public_key 的格式重新设定为地址。将 public_key 作为标准输入。

```
$ sx addr < public_key 17re1S4Q8ZHyCP8Kw7xQad1Lr6XUzWUnkG
```

生成的密钥是所谓的type-0型非决定性密钥，也就是说每个密钥都是从一个随机数生成器生成的。 sx 工具也支持 type-2 决定型密钥，先创建了一个主密钥，然后扩展生成一个子密钥链。首先，我们生成种子，这是整个密钥链的基础。这也可应用于Electrum钱包以及其他类似应用。我们使用 new seed 命令来生成种子值。

```
$ sx newseed > seed $ cat seed eb68ee9f3df6bd4441a9feadec179ff1
```

种子值可以通过 mnemonic 命令转化为助记词，这比十六进制数字更方便易读，也更容易存储和输入。

```
$ sx mnemonic < seed > words $ cat words adore repeat vision worst especially veil inch woman cast recall dwell  
appreciate
```

使用助记词也可以用 mnemonic 命令重新生成种子。

```
$ sx mnemonic < words eb68ee9f3df6bd4441a9feadec179ff1
```

利用这个种子，我们现在可以生成一系列私钥和公钥，也就是一个密钥链。genpriv 命令可以从一个种子生成一系列私钥，addr 命令可以生成对应的公钥。

```
$ sx genpriv 0 < seed 5JzY2cPZGViPGgXZ4Syb9Y4eUGjJpVt6sR8noxrpEcqgyj7LK7i $ sx genpriv 0 < seed | sx addr  
1esVQV2vR9JZPhFeRaeWkAhzmWq7Fi7t7 $ sx genpriv 1 < seed  
5JdtL7ckAn3iFBFyVG1Bs3A5TqziFTaB9f8NeyNo8crnE2Sw5Mz $ sx genpriv 1 < seed | sx addr  
1G1oTeXitk76c2fvQWny4pryTdH1RTqSPW
```

使用这种决定性的密钥，我们可以生成和再造成千上万的新密钥，它以一种关联链的形式从一个唯一的种子生成的。这项技术已经使用在许多钱包应用中，仅用简单几个助记词，就可以备份和重现密钥，这种方式要比创建密钥时需要把钱包的随机生成的所有密钥一起备份要方便的多。

以下是参与本书翻译的志愿者名单，他们用自己宝贵的时间与精力为本书的发布做出了贡献（排名不分先后）：

- 薄荷凉幼
 - 陈萌琦
 - 陈姝吉
 - 程鹏
 - 程西园
 - 达文西
 - 吉鸿帆
 - 李丹
 - 李润熙
 - 李凌豪
 - 李昕阳
 - 刘畅
 - 吕新浩
 - 马峰
 - 牛东晓
 - 秦彤
 - 邱頔
 - 邱蒙
 - 戎如香
 - 史磊
 - 汪海波
 - 王宏钢
 - 王秒
 - 辛颖
 - 杨兵
 - 尹文东
 - 余龙
 - 张林
 - 张琦张大嫂
 - 张亚超
 - 张泽铭
 - 赵冬帅
 - 赵余
 - YANG YANG
 - 申屠青春
 - 李建源
 - 黄世亮
 - 小钻风
 - 接盘手杰克
 - Eric 闵
 - 刘昌用
 - 吕西安
-

如果您发现了书中的错误，请点击[此处](#)填写《勘误收集表》。目前，我们已经收到部分读者的反馈，更新了书中的一些错误，在此对各位的火眼金睛表示感谢。