Table of Contents

第三章. Bitcoin Core:参考实现

第四章. 密钥和地址

第五章. 钱包恢复

第六章. 交易

第七章. 授权和认证

第八章. 数字签名

第九章. 交易手续费

第十章. 比特币网络

第十一章. 区块链

第十二章. 挖矿和共识算法

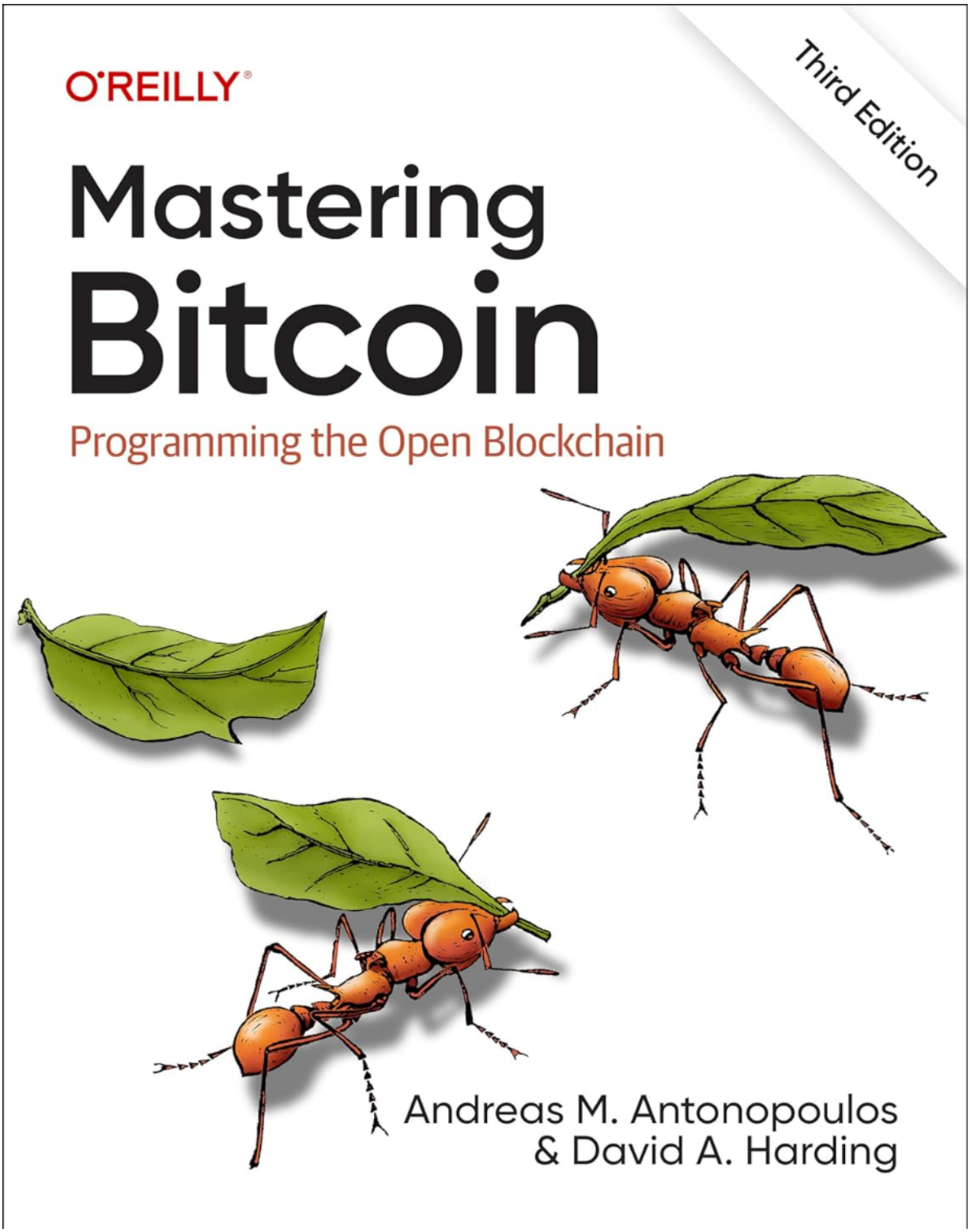
第十三章. 比特币安全

第十四章. 二层应用

附录A. 比特币白皮书（by中本聪）翻译

附录B. 比特币改进建议(Bitcoin Improvement Proposals,BIP)

关于本学习内容



精通比特币第三版封面

精通比特币第三版翻译

https://berryjam.gitbook.io/mastering-bitcoin-3rd

译者介绍

区块链领域从业者

个人编写的区块链相关博客：https://berryjam.github.io/

个人github：https://github.com/berryjam

精通比特币第三版于2023年12月发布，距离2017年发布的第二版过去6年。期间比特币又增加了很多新特性，和实现了不少新的改进提案。如2018年提出的升级提案Taproot，用来增强比特币的可扩展性、隐私性和灵活性，并成为构建BTC 2层重要特性。

另外除了翻译之外，本学习内容会加入一些从技术角度理解的注释，如第4章介绍私钥和公钥时，可能看完就会很快忘记其形式是怎么样，两者如何转换而来。私钥k本质上是一个256比特的大数，而公钥K是椭圆曲线的一个点坐标，通过K=k*G转换而来。那么怎么从大数转为二维(x,y)公钥呢？通过k次椭圆曲线的“加法”操作（切线与曲线相交点的x轴对称点），而G是生成点坐标，通过这就实现了转换。这样加上一些备注，会更方便记忆。

由于很多开发者接触区块链是从以太坊开始的，以太坊开发生态比较完善，实现诸如“拉块”和发起转账和调用智能合约的例子网上也比较多，开发者很容易上手。但是比特币应用开发例子比较少，要实现“拉块”和发起转账就不是那么直观了。所以在原作基础上，第三章加入了《简单的应用开发：使用bitcoinj客户端监听转账，和发起转账》，使用Java客户端实现了一个监听转账和发起转账的demo：https://github.com/berryjam/bitcoinj ，期望能够抛砖引玉。

因此希望本翻译内容能够为大家带来更新的比特币学习内容和一些技术上的思考。

第三版包含以下内容：

- 比特币及其基础区块链的广泛介绍——适合非技术用户、投资者和企业高管。
- 解释比特币的技术基础和加密货币，面向开发人员、工程师和软件及系统架构师
- 比特币去中心化网络、点对点架构、交易生命周期和安全原则的详细信息
- 新发展，如Schnorr 签名、无脚本式多签名、默克尔抽象语法树（MAST）、支付到合约（P2C）、Taproot 协议、Bech32 和 Bech32m 地址格式、手续费管理、无脚本式门限签名、紧凑区块过滤器、FIBRE区块转发系统、BIP8 和 speedy trial 激活软分叉的新方法等
- 深入探讨比特币应用，包括如何将该平台提供的构建块组合成强大的新工具
- 用户故事、类比、示例和代码片段，说明关键技术概念

原书购买链接：

Amazon

Contributors

</table>

Berryjam

lrrrrr2

写作比特币书籍

我（安德烈亚斯）第一次接触比特币是在2011年中期。我的第一反应或多或少是“嘘！这是极客的钱！”我又忽略了它六个月，没有意识到其重要性。我发现这种反应在我认识的很多最聪明的人中间也屡见不鲜，这让我感到有些安慰。第二次遇到比特币时，是在一个邮件列表讨论中，我决定阅读中本聪写的白皮书，看看这是什么。我仍然记得我读完那九页时的时刻，当我意识到比特币不仅仅是一种数字货币，而是一个可以为更多事物提供基础的信任网络。意识到“这不是钱，这是一个去中心化的信任网络”，让我开始了一个长达四个月的旅程，吸收我能找到的关于比特币的每一丁点信息。我变得痴迷而着迷，每天粘在屏幕前读书、写作、编码、学习12个小时甚至更长时间。我从这种迷离状态中走出来时，因为缺乏定期的饮食，体重减轻了20多磅，我下定决心要致力于比特币的工作。

两年后，创建了一些小型创业公司来探索各种与比特币相关的服务和产品之后，我决定是时候写我的第一本书了。比特币是让我狂热创造并占据我的思想的话题；它是我遇到的自互联网以来最令人兴奋的技术。现在是时候与更广泛的观众分享我对这一令人惊叹的技术的热情了。

目标读者

本书主要面向编程人员。如果您能使用编程语言，本书将教会您加密货币的工作原理，如何使用它们，以及如何开发与之配合的软件。前几章也适合非编程人员——那些试图理解比特币和加密货币内部运作原理的人——作为深入介绍。

封面为什么有虫子?

叶切蚁是一种在一个群体超有机体中表现出高度复杂行为的物种，但每只个体蚂蚁都按照社交互动和化学气味（信息素）交换驱动的一组简单规则操作。根据维基百科：“除了人类，叶切蚁构成了地球上最大、最复杂的动物社会。”叶切蚁实际上不吃叶子，而是利用它们种植真菌，这是整个群落的中心食物来源。明白了吗？这些蚂蚁在种植！

虽然蚂蚁形成了一个基于等级的社会，并且有一个用于繁殖后代的女王，但在蚂蚁群中没有中央权威或领袖。一个由数百万个成员组成的群体所展示的高度智能和复杂行为是由社交网络中个体之间的互动产生的 emergent property。

自然表明，去中心化系统可以具有韧性，并且可以产生 emergent complexity 和令人难以置信的复杂性，而无需中央权威、等级制度或复杂的部件。

比特币是一个高度复杂的去中心化信任网络，可以支持多种金融流程。然而，比特币网络中的每个节点都遵循一些简单的规则。许多节点之间的互动导致复杂行为的出现，而不是任何单个节点中的固有复杂性或信任。就像一个蚂蚁群一样，比特币网络是一个由简单节点组成的弹性网络，它们遵循简单规则，可以在没有任何中央协调的情况下做出惊人的事情。

自上一版以来的变化

第三版的一个特别重点是现代化2017年第二版文本和剩余的2014年第一版文本。此外，还添加了许多对2023年当代比特币开发相关的概念：

第4章 我们重新排列了地址信息，以历史顺序逐步解释所有内容，添加了一个新的P2PK部分（在“地址”之前是“IP地址”），更新了之前的P2PKH和P2SH部分，然后添加了segwit/bech32和taproot/bech32m的新部分。

旧的第6章和第7章 从第6章“交易”和第7章“高级交易”的早期版本中重新排列和扩展了文本，跨越了四个新的章节：第6章“交易”（交易结构），第7章“授权和认证”，第8章“数字签名”，和第9章“交易费用”。

第6章 我们添加了关于交易结构的几乎全新文本。

第7章 我们添加了有关MAST、P2C、无脚本多重签名、taproot和tapscript的新文本。

第8章 我们修订了ECDSA文本，并添加了有关schnorr签名、多重签名和阈值签名的新文本。

第9章 我们添加了几乎全新的有关费用、RBF和CPFP费用提升、交易固定、包中继和CPFP切片的文本。

第10章 我们添加了关于紧凑块中继的文本，对布隆过滤器进行了重大更新，更好地描述了它们的隐私问题，并添加了关于紧凑块过滤器的新文本。

第11章 我们添加了有关signet的文本。

第12章 我们添加了有关BIP8和speedy trial的文本。

附录 我们删除了特定于库的附录。在包含原始白皮书的附录之后，我们添加了一个新的附录，描述了比特币的实现和属性与白皮书中提出的不同之处。

综合介绍

比特币是构成数字货币基础的概念和技术的集合。称为比特币的货币单位被用于在比特币网络中的参与者之间存储和传输价值。比特币用户主要通过互联网使用比特币协议进行通信，尽管也可以使用其他传输网络。比特币协议栈作为开源软件可在各种计算设备上运行，包括笔记本电脑和智能手机，使技术易于访问。

用户可以通过网络转移比特币，几乎可以做任何使用传统货币可以做的事情，包括购买和出售商品、向个人或组织发送货币或提供信贷。比特币可以在专门的货币交易所购买、出售和兑换成其他货币。比特币被认为是互联网的理想货币形式，因为它快速、安全且无国界。

与传统货币不同，比特币货币完全虚拟。没有实体硬币甚至个别的数字硬币。硬币在转移价值的交易中暗示着。比特币的用户控制着可以证明他们在比特币网络中拥有比特币的密钥。有了这些密钥，他们可以签署交易以解锁价值，并通过将其转移给新所有者来进行支出。密钥通常存储在每个用户的计算机或智能手机上的数字钱包中。能够签署交易的密钥是花费比特币的唯一先决条件，这样完全将控制权交给了每个用户。

比特币是一个分布式的点对点系统。因此，没有中央服务器或控制点。比特币单位是通过一种称为“挖矿”的过程创建的，该过程涉及反复执行一个参考最近比特币交易列表的计算任务。比特币网络中的任何参与者都可以作为矿工运行，使用他们的计算设备来帮助安全地进行交易。平均每10分钟，一名比特币矿工可以为过去的交易增加安全性，并获得全新的比特币以及最近交易支付的费用。实质上，比特币挖矿去中心化了中央银行的货币发行和清算功能，并取代了任何中央银行的需要。

比特币协议包括内置算法，用于调节网络上挖矿功能。矿工必须执行的计算任务的难度会动态调整，以便平均每10分钟有人成功，无论此时有多少矿工（以及多少处理量）正在竞争。协议还定期减少新创建的比特币数量，将总发行量限制在略低于2100万枚的固定总量。结果是，流通中的比特币数量紧密跟随着一个易于预测的曲线，其中剩余的硬币的一半每四年增加一次。预计在大约第1,411,200个区块，即预计在2035年左右产生时，将发行的所有比特币的99%将被发行。由于比特币发行的减少速度，从长远来看，比特币货币是通缩的。此外，没有人可以强迫您接受超出预期发行速度的任何比特币。

在幕后，比特币也是协议、点对点网络和分布式计算创新的名称。比特币建立在密码学和分布式系统几十年的研究基础上，包括至少四个关键创新，将它们结合在一起形成了独特而强大的组合。比特币包括：

- 一个去中心化的点对点网络（比特币协议）
- 一个公共交易日志（区块链）
- 一组用于独立交易验证和货币发行的规则（共识规则）
- 一种用于在有效的区块链上达成全球分布式共识的机制（工作证明算法）

作为开发者，我将比特币视为货币的互联网，通过分布式计算传播价值并确保数字资产的所有权。比特币远比表面上看到的更加复杂。

在本章中，我们将通过解释一些主要概念和术语、获取必要的软件并使用比特币进行简单交易来开始。在接下来的章节中，我们将开始揭开使比特币成为可能的技术层，并检查比特币网络和协议的内部工作。

\

比特币之前的数字货币

可行数字货币的出现与加密学的发展密切相关。这并不奇怪，当考虑到使用比特表示可以交换货物和服务的价值所涉及的基本挑战时。任何接受数字货币的人都会面临三个基本问题：

• 我能相信这笔钱是真的而不是伪造的吗?

• 我能相信这笔数字货币只能花费一次吗（即“双重支付”问题）？

• 我能确定没有其他人可以声称这笔钱属于他们而不是我吗?

纸币发行者不断地通过使用越来越复杂的纸张和印刷技术来对抗伪造问题。实物货币很容易解决双重支付问题，因为同一张纸币不能同时存在于两个地方。当然，传统货币也经常以数字形式存储和传输。在这些情况下，伪造和双重支付问题通过将所有电子交易清算通过具有对流通中货币的全局视图的中央机构来解决。对于无法利用奇特墨水或全息条的数字货币，加密学提供了信任用户对价值主张合法性的基础。具体来说，加密数字签名使用户能够对数字资产或交易进行签名，证明拥有该资产的所有权。通过适当的架构，数字签名也可以用来解决双重支付问题。

当加密学在1980年代晚期开始变得更加广泛可用和理解时，许多研究人员开始尝试使用加密学构建数字货币。这些早期的数字货币项目发行了数字货币，通常由国家货币或黄金等贵金属支持。

尽管这些早期的数字货币起作用，但它们是中心化的，因此很容易受到政府和黑客的攻击。早期的数字货币使用中央清算所定期结算所有交易，就像传统的银行系统一样。不幸的是，在大多数情况下，这些初期的数字货币受到担心的政府的攻击，并最终因诉讼而消失。一些因母公司突然清算而导致的失败令人瞠目。为了对抗敌对势力的干预，无论是合法政府还是犯罪分子，都需要一个去中心化的数字货币来避免单点攻击。比特币就是这样的系统，通过设计去中心化，并且没有任何可以被攻击或腐败的中央机构或控制点。

\

比特币的历史

比特币首次被描述是在2008年发表的一篇名为《比特币：一种点对点的电子现金系统》的论文中，作者使用了化名中本聪（见附录A）。中本聪结合了几个先前的发明，如数字签名和Hashcash，创建了一个完全去中心化的电子现金系统，不依赖于中央机构进行货币发行、结算和交易验证。一个关键的创新是使用分布式计算系统（称为“工作证明”算法），平均每10分钟进行一次全球彩票，使得去中心化网络能够就交易状态达成共识。这优雅地解决了双重支付问题，即单个货币单位可以被花费两次的问题。以前，双重支付问题是数字货币的弱点，并通过将所有交易通过中央清算所进行清算来解决。

比特币网络始于2009年，基于中本聪发布的参考实现，并且后来由许多其他程序员进行了修订。运行提供比特币安全性和弹性的工作证明算法（挖矿）的计算机的数量和功率已呈指数级增长，它们的综合计算能力现已超过世界顶级超级计算机的综合计算操作数量。

中本聪于2011年4月退出公众视野，将开发代码和网络的责任交给了一个蓬勃发展的志愿者团队。比特币背后的人或团队的身份仍然未知。然而，无论是中本聪还是其他任何人都不能对比特币系统施加个人控制，该系统运行基于完全透明的数学原理、开源代码和参与者之间的共识。这一发明本身具有开创性，并已在分布式计算、经济学和计量经济学领域产生了新的科学。

> 分布式计算问题的解决方案
>
> \ 中本聪的发明还是分布式计算中一个实际且新颖的问题解决方案，即“拜占庭将军问题”。简而言之，该问题包括在没有领导者的情况下，通过在不可靠且可能被 compromise 的网络上交换信息，让多个参与者达成一致行动的尝试。中本聪的解决方案利用工作证明的概念，在没有中央信任机构的情况下实现共识，代表了分布式计算的一项突破。

开始使用比特币

比特币是一个可以通过应用程序访问的协议，该应用程序与该协议进行通信。与HTTP协议最常见的用户界面是Web浏览器一样，“比特币钱包”是比特币系统最常见的用户界面。就像Chrome、Safari和Firefox等许多品牌的Web浏览器一样，比特币钱包有许多实现和品牌。就像我们都有自己喜欢的浏览器一样，比特币钱包在质量、性能、安全性、隐私性和可靠性方面也有所不同。比特币协议还包括一个钱包的参考实现，称为“比特币核心”，它是由中本聪撰写的原始实现衍生而来。

\

选择比特币钱包

比特币钱包是比特币生态系统中最活跃的应用之一。竞争非常激烈，虽然新的钱包可能正在开发中，但去年的几个钱包已经不再积极维护。许多钱包专注于特定平台或特定用途，有些适合初学者，而有些则充满了面向高级用户的功能。选择钱包是非常主观的，取决于使用情况和用户的专业知识。因此，推荐特定品牌或钱包是毫无意义的。然而，我们可以根据其平台和功能对比特币钱包进行分类，并提供有关所有不同类型的钱包的一些澄清。值得尝试几种不同的钱包，直到找到适合自己需求的一个。

比特币钱包的类型

根据平台，比特币钱包可以分为以下几类：

桌面钱包

桌面钱包是最早创建的一种比特币钱包类型，作为参考实现。许多用户使用桌面钱包是因为它们提供的功能、自治权和控制权。但是，作为运行在通用操作系统（如Windows和macOS）上的桌面钱包也存在一定的安全缺陷，因为这些平台通常不安全且配置不良。

移动钱包

移动钱包是最常见的比特币钱包类型。运行在智能手机操作系统（如Apple iOS和Android）上，这些钱包通常是新用户的不错选择。许多钱包设计简单易用，但也有针对高级用户的功能齐全的移动钱包。为了避免下载和存储大量数据，大多数移动钱包从远程服务器检索信息，通过向第三方披露有关您的比特币地址和余额的信息，降低了您的隐私。

Web钱包

Web钱包通过Web浏览器访问，并将用户的钱包存储在第三方拥有的服务器上。这类似于Web邮件，完全依赖于第三方服务器。其中一些服务使用运行在用户浏览器中的客户端代码，这样可以保持比特币密钥控制在用户手中，尽管用户仍然依赖服务器，但这种方式会损害用户的隐私。但大多数服务为了方便起见，从用户那里获取比特币密钥控制权。不建议在第三方系统上存储大量比特币。

硬件签名设备

硬件签名设备是可以使用专用硬件和固件存储密钥并签署交易的设备。它们通常通过USB电缆、近场通信（NFC）或具有QR码的摄像头连接到桌面、移动或Web钱包。通过在专用硬件上处理所有与比特币相关的操作，这些钱包对许多类型的攻击更不易受到影响。有时将硬件签名设备称为“硬件钱包”，但它们需要与功能齐全的钱包配对，以发送和接收交易，并且由配对钱包提供的安全性和隐私性在使用硬件签名设备时起着关键作用。

全节点vs轻节点

另一种分类比特币钱包的方法是根据其自治程度以及与比特币网络的交互方式：

全节点

全节点是一种验证比特币所有历史交易（每个用户的每笔交易）的程序。全节点还可以选择存储先前验证的交易并向其他比特币程序提供数据，无论是在同一台计算机上还是通过互联网。全节点使用大量的计算资源——大约相当于每天观看一小时的流媒体视频——但全节点为其用户提供了完全自治。

轻量级客户端

轻量级客户端，也称为简化支付验证（SPV）客户端，连接到全节点或其他远程服务器以接收和发送比特币交易信息，但将用户钱包存储在本地，部分验证接收到的交易，并独立创建输出交易。

第三方API客户端

第三方API客户端是通过与比特币网络直接连接而不是通过连接到比特币网络来与比特币进行交互的第三方API系统进行交互的客户端。钱包可以由用户或第三方服务器存储，但客户端信任远程服务器向其提供准确的信息并保护其隐私。

比特币是一个点对点（P2P）网络。全节点是对等节点：每个节点都可以验证每个确认的交易，并能够向其用户提供具有完全权限的数据。轻量级钱包和其他软件是客户端：每个客户端依赖于一个或多个对等节点提供有效的数据。比特币客户端可以对接收到的部分数据进行辅助验证，并建立与多个对等节点的连接，以减少对单个对等节点完整性的依赖，但是客户端的安全性最终取决于其对等节点的完整性。

谁控制密钥

一个非常重要的额外考虑因素是谁控制密钥。正如我们将在后续章节中看到的那样，对比特币的访问受到“私钥”的控制，这些私钥类似于非常长的PIN码。如果只有您控制这些私钥，那么您就控制着您的比特币。相反，如果您没有控制权，则您的比特币由第三方管理，最终由第三方代表您控制您的资金和账户。根据控制权，密钥管理软件分为两个重要的类别：钱包，您控制密钥；以及具有托管人的资金和账户，某些第三方控制密钥。为了强调这一点，我（安德烈亚斯）创造了这句话：“你的密钥，你的硬币。不是你的密钥，不是你的硬币。”

结合这些分类，许多比特币钱包分为几个组，其中最常见的三个是桌面全节点（您控制密钥）、移动轻量级钱包（您控制密钥）和与第三方的基于Web的账户（您不控制密钥）。不同类别之间的界限有时会模糊，因为软件在多个平台上运行，并且可以以不同的方式与网络进行交互。\ \

快速开始

艾丽丝不是技术用户，最近才从她的朋友乔那里听说比特币。在一次派对上，乔正在热情地向周围的人解释比特币，并提供演示。艾丽丝感到好奇，问他如何开始使用比特币。乔说移动钱包对新用户来说是最好的选择，并推荐了几款他喜欢的钱包。艾丽丝下载了乔推荐的其中一个，并将其安装在手机上。

当艾丽丝第一次运行她的钱包应用时，她选择了创建一个新的比特币钱包的选项。因为她选择的钱包是非托管钱包，所以艾丽丝（只有艾丽丝）将控制她的密钥。因此，她负责备份密钥，因为丢失密钥意味着她失去了对她的比特币的访问权限。为了方便起见，她的钱包生成了一个恢复码，可以用来恢复她的钱包。

恢复码

大多数现代的非托管比特币钱包都会为用户提供一个恢复码进行备份。恢复码通常由软件随机选择的数字、字母或单词组成，并用作钱包生成的密钥的基础。请参见表1-1中的示例。

表1-1. 恢复码示例

钱包	恢复码
BlueWallet	(1) media (2) suspect (3) effort (4) dish (5) album (6) shaft (7) price (8) junk (9) pizza (10) situate (11) oyster (12) rib
Electrum	nephew dog crane clever quantum crazy purse traffic repeat fruit old clutch
Muun	LAFV TZUN V27E NU4D WPF4 BRJ4 ELLP BNFL

恢复码有时被称为“助记词”或“助记短语”，这暗示您应该记住该短语，但将短语写在纸上比大多数人的记忆更可靠且需要更少的工作。另一个替代名称是“种子短语”，因为它为生成钱包的所有密钥的函数提供了输入（“种子”）。

如果艾丽丝的钱包发生了什么问题，她可以下载一个新的钱包软件，并输入这个恢复码来重建她曾经发送或接收的所有链上交易的钱包数据库。然而，仅凭恢复码进行恢复将不能自动恢复艾丽丝输入到她的钱包中的任何其他数据，比如她与特定地址或交易相关联的标签。虽然失去对这些元数据的访问权并不像失去对钱的访问权那样重要，但在某种程度上它仍然是重要的。想象一下，您需要查看一份旧的银行或信用卡对账单，但每个您支付（或者谁支付给您）的实体的名称都被抹去了。为了防止丢失元数据，许多钱包在恢复码之外提供了额外的备份功能。

对于某些钱包来说，这种额外的备份功能今天比以前更加重要。现在许多比特币支付是使用链下技术进行的，其中并不是每笔支付都会存储在公共区块链上。这降低了用户的成本，并提高了隐私性等方面的利益，但这意味着像恢复码这样依赖于链上数据的机制无法保证完全恢复用户所有的比特币。对于支持链下技术的应用程序，频繁备份钱包数据库非常重要。

值得注意的是，当首次向新的移动钱包收到资金时，许多钱包通常会重新验证您是否已安全备份了恢复码。这可能范围从简单的提示到要求用户手动重新输入代码。尽管许多合法的钱包应用程序会提示您重新输入您的恢复码，但也有许多恶意应用程序会模仿钱包的设计，坚持让您输入恢复码，然后将任何输入的代码传递给恶意开发者，以便他们窃取您的资金。这相当于试图诱骗您提供银行密码的钓鱼网站。对于大多数钱包应用程序来说，它们只会在初始设置（在您收到任何比特币之前）和恢复期间（在您无法访问原始钱包之后）询问您的恢复码。如果应用程序在其他任何时间要求您的恢复码，请咨询专家以确保您没有被钓鱼。

\

比特币地址

艾丽丝现在准备开始使用她的新比特币钱包。她的钱包应用程序随机生成了一个私钥（在第55页“私钥”中有更详细的描述），该私钥将用于生成指向她钱包的比特币地址。此时，她的比特币地址并不为比特币网络所知，也没有在比特币系统的任何部分“注册”。她的比特币地址只是与她的私钥相对应的数字，她可以使用这些地址来控制对资金的访问。这些地址由她的钱包独立生成，不参考或注册任何服务。

有各种各样的比特币地址和发票格式。地址和发票可以与其他比特币用户共享，他们可以使用这些地址直接向您的钱包发送比特币。您可以与其他人分享地址或发票，而不必担心您的比特币的安全性。与银行账户号码不同，任何得知您其中一个比特币地址的人都无法从您的钱包中提取资金——您必须自行发起所有交易。然而，如果您给两个人相同的地址，他们将能够看到对方向您发送了多少比特币。如果您公开发布您的地址，每个人都将能够看到其他人向该地址发送了多少比特币。为了保护您的隐私，您应该每次请求付款时都生成一个新的发票，带有一个新的地址。

\

接收比特币

艾丽丝点击“接收”按钮，显示了一个二维码，如图1-1所示。

\

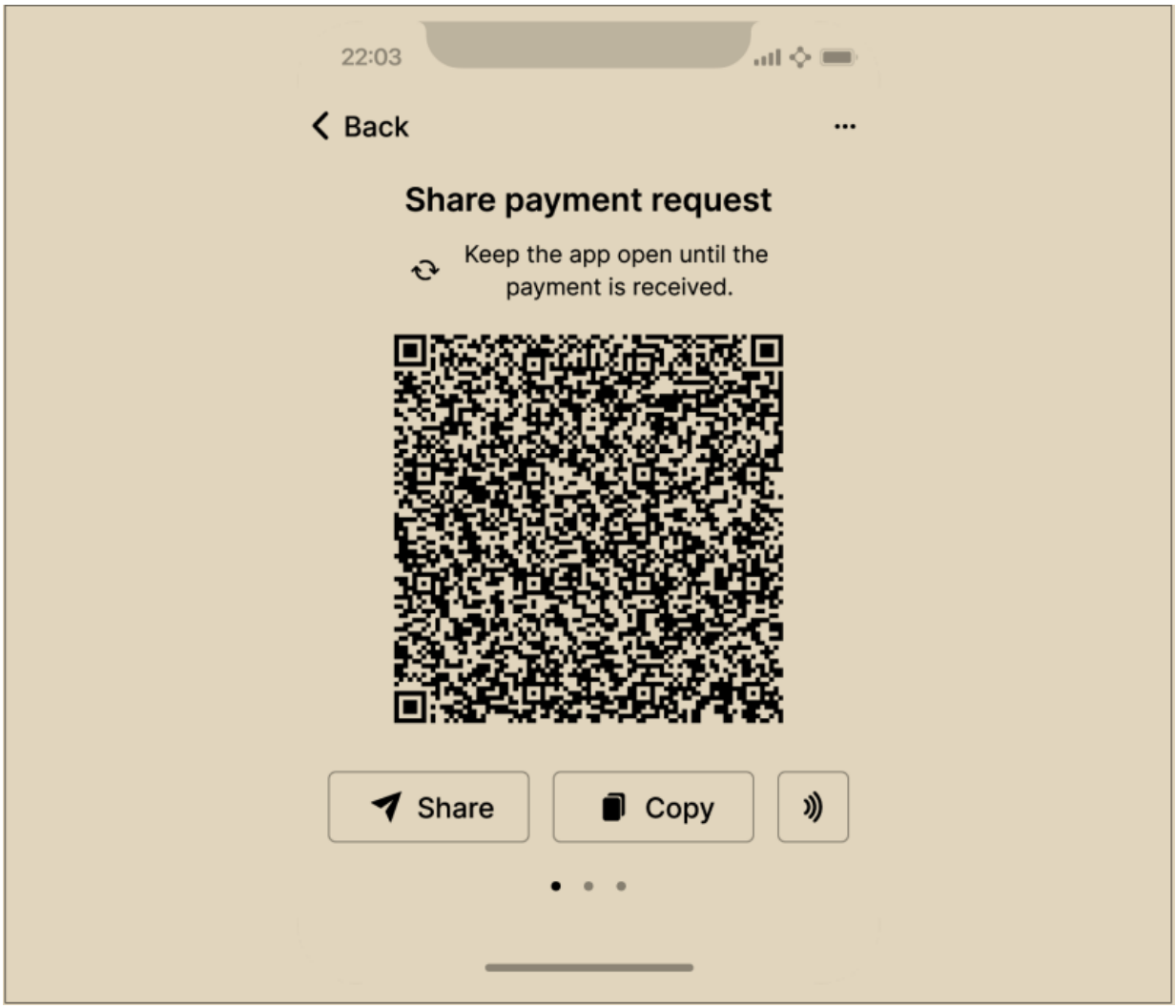


图 1-1. Alice在她的手机比特币钱包上使用“接收”界面，并以二维码格式显示了她的地址。

二维码是一个方形的黑白点阵图案，它包含着相同的信息，可以被Joe的智能手机摄像头扫描。

本书中的地址收到的任何资金都会丢失。如果你想测试发送比特币，请考虑将其捐赠给接受比特币的慈善机构。

获取你的第一枚比特币

新用户的第一个任务是获取一些比特币。

比特币交易是不可逆转的。大多数电子支付网络，如信用卡、借记卡、PayPal和银行账户转账，都是可逆转的。对于出售比特币的人来说，这种差异带来了一个非常高的风险，即买家在收到比特币后会撤销电子支付，实际上欺骗了卖家。为了减轻这种风险，接受传统电子支付换取比特币的公司通常要求买家进行身份验证和信用评估，这可能需要几天甚至几周的时间。作为一个新用户，这意味着你不能立即用信用卡购买比特币。然而，有点耐心和创造性思维，你就不需要这样做。

以下是一些作为新用户获取比特币的方法：

- 找一个有比特币的朋友，直接向他或她购买一些。许多比特币用户都是这样开始的。与持有比特币的人见面的一种方式是参加Meetup.com上列出的当地比特币聚会。
- 通过销售产品或服务换取比特币。如果你是一名程序员，可以用你的编程技能换取比特币。如果你是一名理发师，可以用比特币来剪头发。
- 在你所在城市使用比特币自动取款机。比特币自动取款机是一种接受现金并将比特币发送到你的智能手机比特币钱包的机器。
- 使用与你的银行账户连接的比特币货币交易所。许多国家现在都有货币交易所，提供市场供买家和卖家用本地货币交换比特币。汇率列表服务，如BitcoinAverage，通常会列出每种货币的比特币交易所列表。

比特币相对于其他支付系统的优势之一是，当正确使用时，它为用户提供了更多的隐私。获取、持有和使用比特币不需要你向第三方透露敏感和可识别的个人信息。然而，当比特币与传统系统（如货币交易所）接触时，通常会适用国内外法规。为了将比特币兑换成你的国家货币，通常需要提供身份证明和银行信息。用户应该意识到，一旦一个比特币地址与身份挂钩，其他相关的比特币交易也可能变得容易识别和跟踪，包括之前进行的交易。这也是许多用户选择保持独立于他们的钱包的专用交易账户的原因之一。

爱丽丝是通过朋友介绍认识了比特币，所以她有一个简单的方法来获取她的第一枚比特币。接下来，我们将看看她如何从她的朋友乔那里购买比特币，以及乔如何将比特币发送到她的钱包。

获取比特币现货价格

在爱丽丝能够从乔那里购买比特币之前，他们必须就比特币和美元之间的汇率达成一致。这引出了一个对于新接触比特币的人来说很常见的问题：“谁确定比特币的价格？”简短的答案是市场决定价格。

比特币，像大多数其他货币一样，拥有浮动汇率。这意味着比特币的价值会根据在各种交易市场上的供求情况而波动。例如，比特币对美元的“价格”是根据最近一次比特币和美元的交易而在每个市场上计算出来的。因此，价格往往在每秒钟变动数次。定价服务将从多个市场汇总价格，并计算出一个体现一种货币对（如BTC/USD）广义市场汇率的成交量加权平均值。

有数百个应用程序和网站可以提供当前的市场汇率。以下是一些最受欢迎的：

比特币平均价（Bitcoin Average）

一个提供每种货币成交量加权平均价的简单视图的网站。

CoinCap

一项列出数百种加密货币的市值和汇率的服务，包括比特币在内。

芝加哥商品交易所比特币参考汇率（Chicago Mercantile Exchange Bitcoin Reference Rate）

一种可用于机构和合约参考的参考汇率，由芝加哥商品交易所提供作为投资数据提供之一。

除了这些不同的站点和应用程序外，一些比特币钱包还会自动在比特币和其他货币之间进行金额转换。

发送和接收比特币

爱丽丝决定购买0.001比特币。在她和乔确认了汇率之后，她递给了乔一定金额的现金，打开了她的手机钱包应用程序，并选择了“接收”。这会显示一个带有爱丽丝的第一个比特币地址的二维码。

接着，乔在他的智能手机钱包上选择“发送”并打开了二维码扫描器。这使得乔可以用他的智能手机摄像头扫描条形码，而不必输入爱丽丝的比特币地址，因为地址非常长。

现在，乔已将爱丽丝的比特币地址设置为接收方。乔输入了0.001比特币（BTC）的金额；见图1-2。一些钱包可能会以不同的计量单位显示金额：0.001 BTC相当于1毫比特币（mBTC）或100,000 satoshi（sats）。

一些钱包也许会建议乔为这笔交易输入一个标签；如果是这样，乔输入“Alice”。几周或几个月后，这将帮助乔记住他为什么发送了这0.001比特币。一些钱包也许会提示乔有关手续费的事项。根据钱包以及交易的发送方式，钱包可能会要求乔输入交易费率，或者提示他一个建议的手续费（或费率）。交易手续费越高，交易确认的速度就越快（参见“确认”）。

\

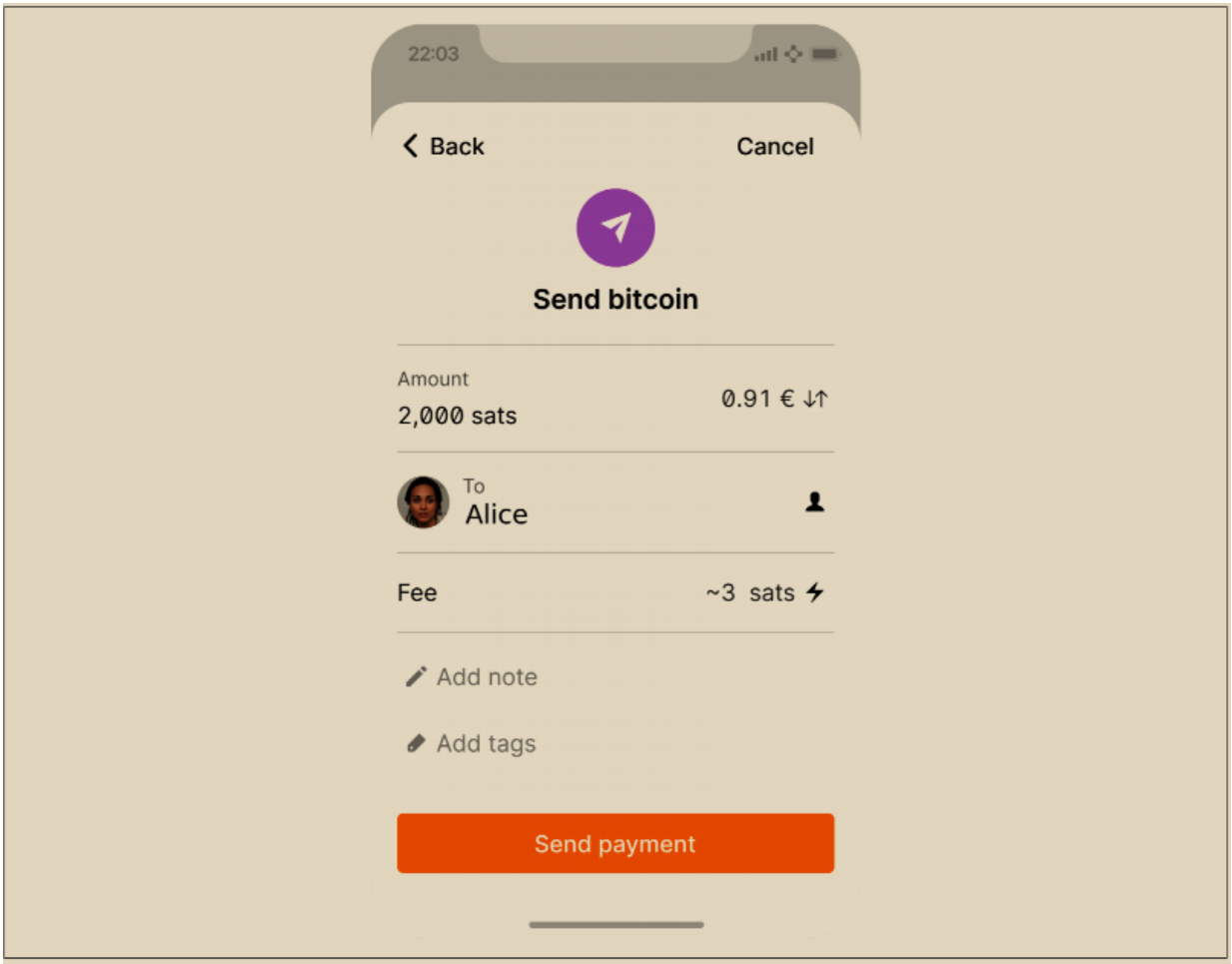


图 1-2. 比特币钱包发送界面

接着，Joe 仔细检查以确保输入的金额正确，因为他即将发送资金，而错误很快将变得不可逆转。在双重检查地址和金额之后，他按下“发送”按钮来发送交易。Joe 的移动比特币钱包构建了一笔交易，将 0.001 BTC 分配到 Alice 提供的地址上，资金来源于 Joe 的钱包，并使用 Joe 的私钥对交易进行签名。这告诉比特币网络，Joe 已经授权将价值转移到 Alice 的新地址。由于交易通过对等协议传输，它很快就会在比特币网络中传播。几秒钟后，网络中的大多数连接良好的节点都会接收到交易，并首次看到 Alice 的地址。

与此同时，Alice 的钱包不断“监听”比特币网络上的新交易，寻找与其包含的地址匹配的任何交易。几秒钟后，Joe 的钱包发送交易，Alice 的钱包将指示正在接收 0.001 BTC。

\ 确认

起初，Alice 的地址会显示来自 Joe 的交易为“未确认”。这意味着交易已经传播到网络上，但尚未记录在比特币交易日志中，即区块链上。要得到确认，交易必须被包含在一个区块中并添加到区块链中，这通常每 10 分钟发生一次。在传统金融术语中，这被称为清算。有关比特币交易的传播、验证和清算（确认）的更多详细信息，请参阅第 12 章。

Alice现在是0.001 BTC的自豪所有者，她可以进行消费。在接下来的几天里，Alice通过自动取款机和交易所购买了更多的比特币。在下一章中，我们将详细探讨她使用比特币进行的第一次购买，并更详细地研究底层的交易和传播技术。

综合介绍

比特币系统与传统银行和支付系统不同，不需要对第三方进行信任。在比特币中，每个用户都可以使用在自己计算机上运行的软件来验证比特币系统的每个方面的正确运行，而不是依赖于中央受信任的权威机构。在本章中，我们将通过追踪单个交易在比特币系统中的流动，并观察其如何记录在区块链上，即所有交易的分布式日志，来从高层次审视比特币。后续章节将深入探讨交易背后的技术、网络和挖矿等内容。

比特币概述

比特币系统由拥有包含密钥的钱包的用户、在网络中传播的交易以及通过竞争性计算生成共识区块链的矿工组成，后者是所有交易的权威账簿。

本章中的每个示例都基于比特币网络上进行的实际交易，通过将资金从一个钱包发送到另一个钱包来模拟用户之间的交互。在跟踪交易通过比特币网络到区块链的过程中，我们将使用区块链浏览器网站来可视化每一步。区块链浏览器是一种网络应用程序，它充当比特币搜索引擎，允许您搜索地址、交易和区块，并查看它们之间的关系和流动。

一些受欢迎的区块链浏览器包括以下几种：

- Blockstream Explorer
- Mempool.Space
- BlockCypher Explorer

这些浏览器都具有搜索功能，可以接受比特币地址、交易哈希、区块编号或区块哈希，并从比特币网络中检索相应的信息。对于每个交易或区块示例，我们将提供一个URL，以便您自行查看并详细研究。

> 区块链浏览器隐私警告
>
> 在区块链浏览器上搜索信息可能会向其运营者透露您对该信息感兴趣，使他们能够将其与您的IP地址、浏览器详细信息、过去的搜索记录或其他可识别信息关联起来。如果您在本书中查找交易，区块链浏览器的运营者可能会猜测您正在学习比特币，这不应该是问题。但是，如果您查找自己的交易，运营者可能会猜测您收到了多少比特币、花费了多少比特币，以及当前拥有多少比特币。

\

\

从在线商店购买

在前一章中介绍的爱丽丝是一位新用户，她刚刚获得了自己的第一笔比特币。在“获取你的第一笔比特币”中，爱丽丝在第11页，与她的朋友乔见面，用一些现金换取了比特币。自那以后，爱丽丝又购买了额外的比特币。现在，爱丽丝将进行她的第一笔消费交易，购买鲍勃在线商店的一个高级播客剧集的访问权限。

鲍勃的网店最近开始接受比特币支付，通过在网站上添加比特币选项。鲍勃商店的价格以当地货币（美元）列出，但在结账时，顾客可以选择以美元或比特币支付。

爱丽丝找到了她想要购买的播客剧集，并前往结账页面。在结账时，除了通常的选项外，爱丽丝还可以选择使用比特币付款。结账购物车显示了以美元和比特币（BTC）计价的价格，以及比特币的当前汇率。

鲍勃的电子商务系统将自动生成一个包含发票的二维码（见图2-1）。

\

图 2-1. 发票二维码

不同于只包含目标比特币地址的二维码，这个发票是一个包含目标地址、支付金额和描述的QR编码URI。这使得比特币钱包应用程序能够预先填写用于发送付款的信息，同时向用户显示可读的描述。您可以使用比特币钱包应用程序扫描QR码，以查看Alice将看到的内容：bitcoin:bc1qk2g6u8p4qm2s2lh3gts5cpt2mrv5skcuu7u3e4?amount=0.01577764\&label=Bob%27s%20Store\&message=Purchase%20at%20Bob%27s%20Store

URI的组成部分：

- 比特币地址："bc1qk2g6u8p4qm2s2lh3gts5cpt2mrv5skcuu7u3e4"
- 支付金额："0.01577764"
- 收款地址的标签："Bob's Store"
- 付款描述："Purchase at Bob's Store"

> 请尝试使用您的钱包扫描此内容，查看地址和金额，但请勿发送资金。

Alice使用她的智能手机扫描显示屏上的条形码。她的智能手机显示了向Bob's Store支付的正确金额，并选择发送以授权支付。几秒钟后（大约与信用卡授权相同的时间），Bob在注册机上看到了交易。

> 比特币网络可以进行分数值的交易，例如，从毫比特币（比特币的1/1000）到1/100,000,000比特币，即被称为“聪” 的单位。本书在谈论大于一个比特币的金额以及使用小数表示法时，使用与美元和其他传统货币相同的复数规则，比如“10比特币”或“0.001比特币”。相同的规则也适用于其他比特币记账单位，如毫比特币和 “聪”。
>
> \

您可以使用区块浏览器来检查区块链数据，比如 Alice 交易中向 Bob 支付的款项。

在接下来的章节中，我们将更详细地研究这笔交易。我们将看到 Alice 的钱包如何构建它，它是如何在网络中传播的，如何被验证的，最后，Bob 如何在随后的交易中花费这笔金额。

\

比特币交易

简单来说，一笔交易告诉网络，某些比特币的所有者已经授权将该价值转移到另一个所有者名下。新所有者现在可以通过创建另一笔交易来授权将比特币转移到另一个所有者名下，如此往复，形成所有权链。

交易输入和输出

交易就像双重记账簿中的条目一样。每笔交易包含一个或多个输入，用于花费资金。在交易的另一侧，有一个或多个输出，用于接收资金。输入和输出不一定相等。相反，输出的总和略小于输入，差额代表隐含的交易费，这是矿工在区块链中包含交易时收取的一笔小费。比特币交易显示为双重记账簿中的条目，如图 2-2 所示。

交易还包含每笔比特币金额（输入）的所有权证明，以数字签名的形式，由所有者提供，任何人都可以进行独立验证。在比特币术语中，花费是指签署一笔交易，将价值从先前的交易转移到由比特币地址标识的新所有者名下。

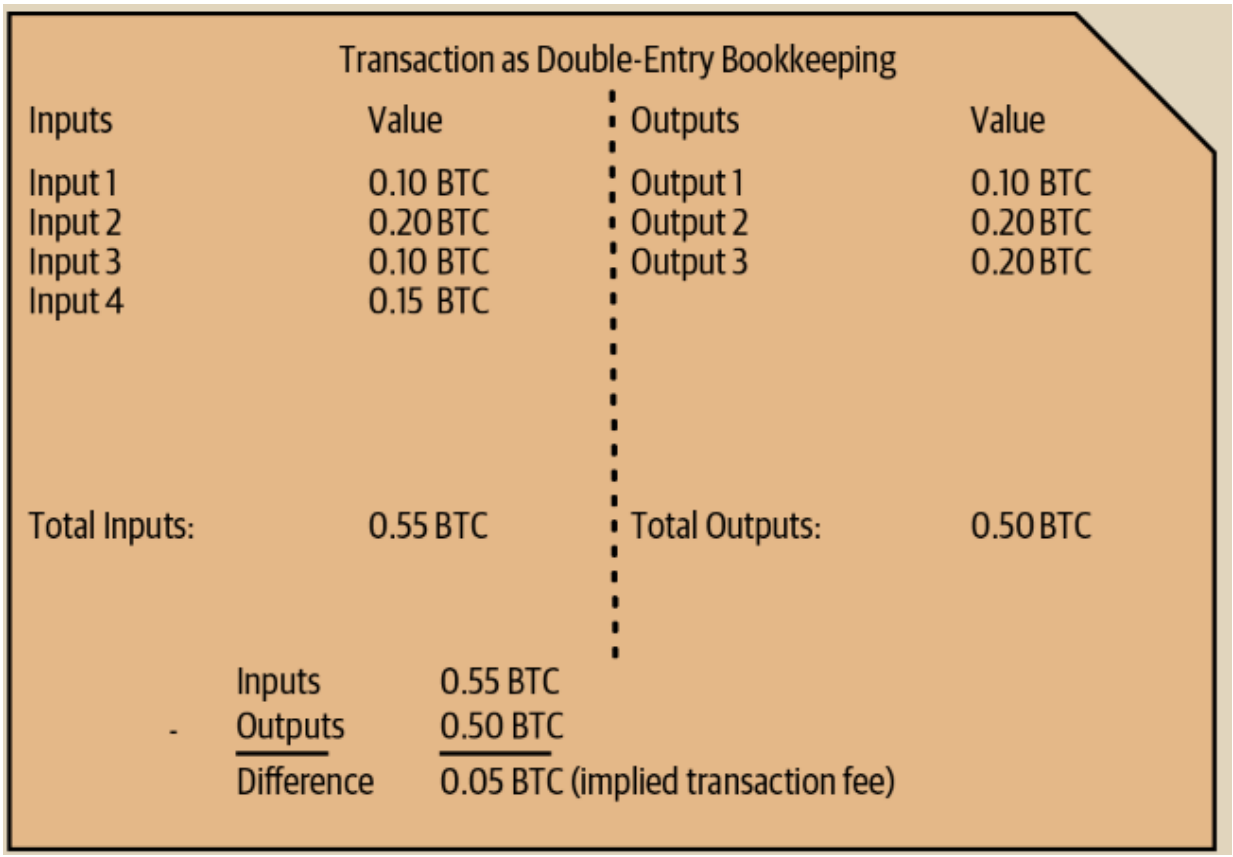


图 2-2. 在比特币交易中，可以将其视为双重记账簿中的条目

交易链

Alice的支付给Bob的商店使用了先前交易的输出作为其输入。在前一章中，Alice以现金换取了她朋友Joe的比特币。我们在图2-3中将其标记为Transaction 1（Tx1）。

Tx1发送了0.001比特币（100,000 satoshi）到由Alice的密钥锁定的输出。她向Bob的商店的新交易（Tx2）引用了先前的输出作为输入。在插图中，我们使用箭头显示该引用，并通过将输入标记为“Tx1:0”来标示。在实际交易中，引用是指Alice从Joe那里收到资金的交易的32字节交易标识符（txid）。“:0”表示Alice收到资金的输出的位置；在这种情况下，是第一个位置（位置0）。

如所示，实际的比特币交易并不明确包含其输入的价值。要确定输入的价值，软件需要使用输入的引用来查找要花费的先前交易输出。

Alice的Tx2包含两个新输出，一个支付75,000 satoshi购买播客，另一个支付20,000 satoshi作为找零返回给Alice。

\

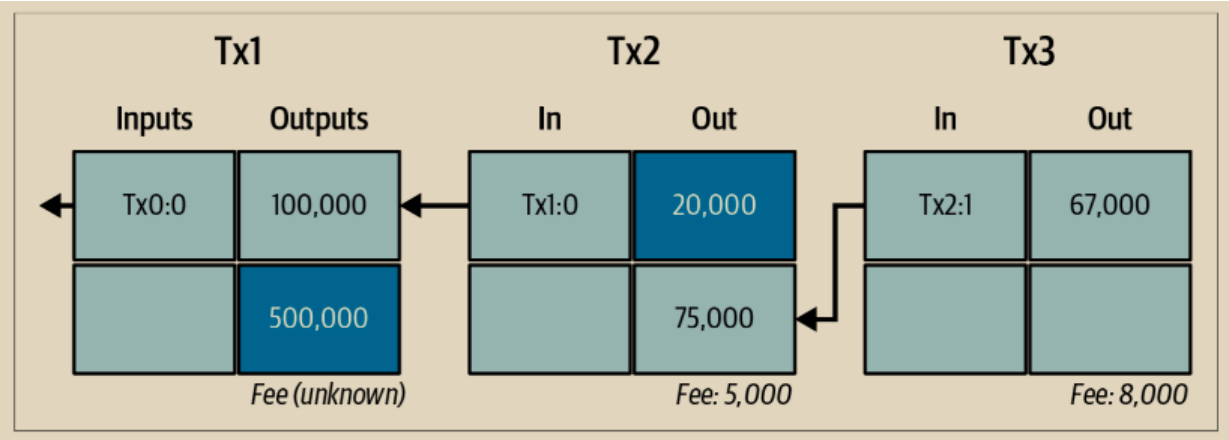


图 2-3. 交易链是指一系列交易，其中一个交易的输出是下一个交易中的输入

比特币交易的序列化格式——软件用于发送交易的数据格式——使用最小的定义在链上价值单位的整数来编码要转移的价值。当比特币刚刚创建时，这个单位没有名称，一些开发者简单地称之为基本单位。后来，许多用户开始将这个单位称为“聪”（satoshi），以纪念比特币的创造者。在本书的图2-3和其他一些插图中，我们使用“聪”值，因为这是协议本身使用的单位。

找零

除了一个或多个用于支付比特币接收者的输出之外，许多交易还将包括一个用于支付比特币花费者的输出，称为找零输出。这是因为交易输入，就像货币票据一样，不能部分使用。如果你在商店购买了价值5美元的物品，但使用了一张20美元的钞票来支付，你希望收到15美元的找零。同样的概念适用于比特币交易输入。如果你购买了价值5比特币的物品，但只有价值20比特币的输入可供使用，那么你会向商店所有者发送一个价值5比特币的输出，以及一个价值15比特币的输出作为找零（不计入交易费用）。

在比特币协议层面上，找零输出（以及支付它的地址，称为找零地址）与支付输出之间没有区别。

重要的是，找零地址不必与输入地址相同，并且出于隐私原因，通常是来自所有者钱包的新地址。在理想情况下，两种不同的输出使用从未见过的地址，并且在其他方面看起来完全相同，防止任何第三方确定哪些输出是找零输出，哪些是支付输出。然而，出于说明的目的，我们在图2-3中为找零输出添加了阴影。

并非每个交易都有找零输出。那些没有找零输出的交易被称为无找零交易，它们只能有一个单一的输出。如果要花费的金额与交易输入减去预期的交易费用的金额大致相同，则无找零交易才是一个实际的选择。在图2-3中，我们看到Bob创建了Tx3作为一个无找零交易，用于花费他在Tx2中收到的输出。

\

币种选择

在支付时，不同的钱包使用不同的策略来选择要使用的输入，称为币种选择。

它们可能会聚合许多小额输入，或者使用一个等于或大于所需支付的输入。除非钱包能够以恰好匹配所需支付加上交易费用的方式聚合输入，否则钱包将需要产生一些找零。这与人们如何处理现金非常相似。如果你总是使用口袋里最大的钞票，你最终会得到满口的零钱。如果你只使用零钱，你经常只会有大钞票。人们下意识地在这两个极端之间找到平衡，而比特币钱包开发者致力于编程这种平衡。

常见的交易形式

一种非常常见的交易形式是简单支付。这种类型的交易有一个输入和两个输出，如图2-4所示。

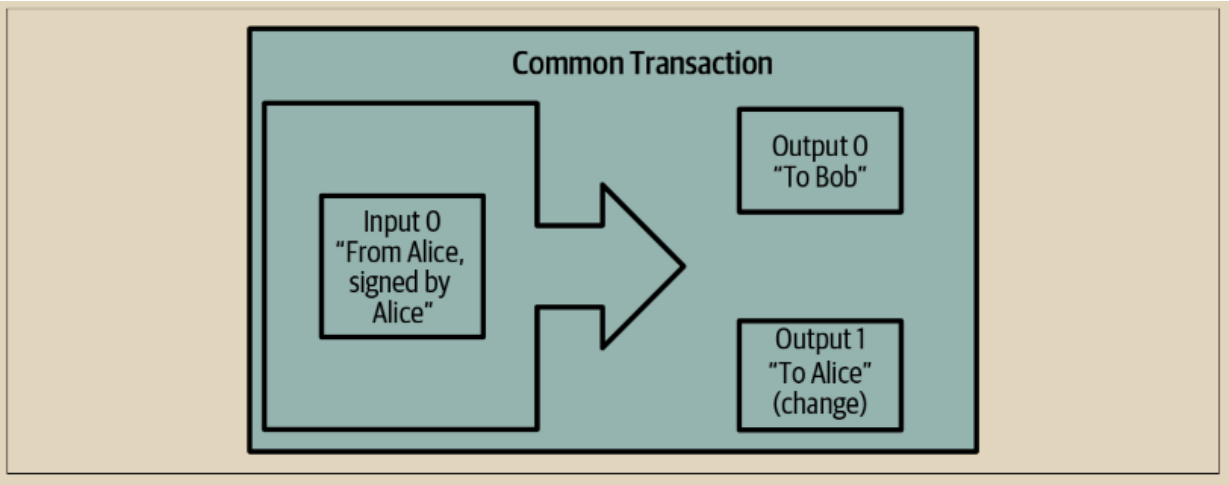


图 2-4. 最常见的交易

另一种常见的交易形式是合并交易，它将多个输入合并成一个输出（图2-5）。这相当于在现实世界中将一堆硬币和纸币换成一张更大的纸币。 有时，钱包和企业会生成这样的交易来清理大量较小的金额。

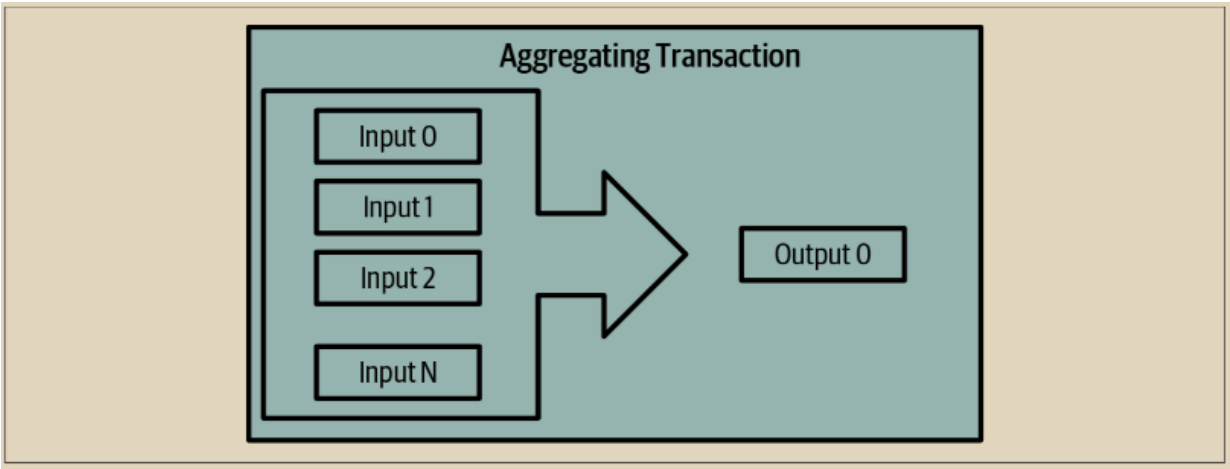


图 2-5. 合并交易聚合资金

最后，区块链上经常看到的另一种交易形式是支付批处理，它支付给多个输出，代表多个收款方（图2-6）。商业实体有时会使用这种类型的交易来分发资金，比如在向多名员工支付工资时。

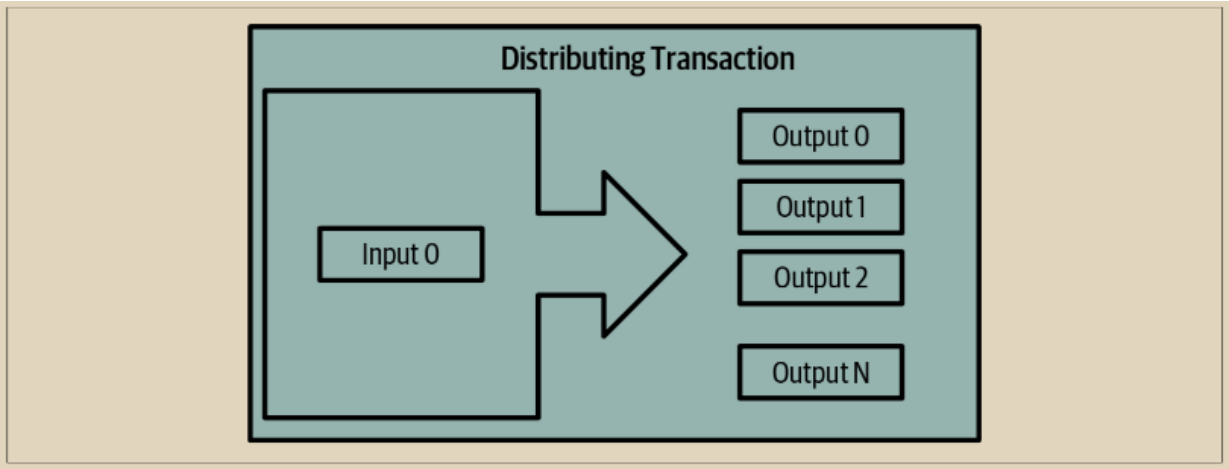


图 2-6. 批处理交易分配资金

构建交易

Alice 的钱包应用程序包含了选择输入和生成输出的所有逻辑，以构建符合 Alice 规格的交易。Alice 只需要选择目的地、金额和交易费用，剩下的事情都在钱包应用程序中进行，她不需要看到详细信息。重要的是，如果一个钱包已经知道它控制的输入，即使它完全离线，它也可以构建交易。就像在家里写支票，然后把它放在信封里寄给银行一样，交易不需要在连接到比特币网络时构建和签名。

获取正确的输入

Alice 的钱包应用程序首先需要找到可以支付给 Bob 的金额的输入。大多数钱包跟踪钱包中地址所有可用的输出。因此，Alice 的钱包会包含来自 Joe 交易的交易输出的副本，该交易是用现金交换的（参见“获取您的第一个比特币”）。在全节点上运行的比特币钱包应用程序实际上包含了每个已确认交易的未花费输出，称为 UTXO（未花费的交易输出）。然而，由于全节点使用的资源更多，许多用户钱包运行轻量级客户端，仅跟踪用户自己的 UTXO。

在这种情况下，这个单一的 UTXO 足以支付播客。如果不是这样，Alice 的钱包应用程序可能需要合并几个较小的 UTXO，就像从钱包中选取硬币一样，直到它找到足够支付播客的金额。在这两种情况下，可能需要得到一些找零，我们将在下一节中看到，当钱包应用程序创建交易输出（支付）时。

创建输出

交易输出是使用一段脚本创建的，该脚本大致表示“此输出支付给可以提供与 Bob 公钥地址对应的密钥的签名的任何人”。因为只有 Bob 拥有与该地址对应的密钥的钱包，所以只有 Bob 的钱包可以提供这样的签名以后花费这个输出。因此，Alice 将输出值与对 Bob 的签名需求绑定起来。

此交易还将包括第二个输出，因为 Alice 的资金比播客的费用更多。Alice 的找零输出在与付款给 Bob 的同一交易中创建。实质上，Alice 的钱包将她的资金分成两个输出：一个给 Bob，一个退回给她自己。然后，她可以在随后的交易中花费找零输出。

最后，为了使交易及时被网络处理，Alice 的钱包应用程序将添加一个小费。这个费用在交易中没有明确说明；它是通过输入和输出之间的差值来隐含的。矿工收取这笔交易费作为将交易包含在区块中的费用。

查看从Alice到Bob商店的交易

将交易添加到区块链中

Alice 的钱包应用程序创建的交易包含确认资金所有权和指定新所有者所需的一切。现在，这个交易必须被传输到比特币网络中，其中它将成为区块链的一部分。在接下来的部分中，我们将看到一个交易如何成为一个新区块的一部分，以及如何进行挖矿。最后，我们将看到，一旦新区块被添加到区块链上，随着更多的区块被添加，网络对其的信任程度也在增加。

传输交易

由于交易包含了处理所需的所有信息，因此它被传输到比特币网络的方式或位置并不重要。比特币网络是一个点对点网络，每个比特币节点通过连接到其他几个比特币节点来参与其中。比特币网络的目的是向所有参与者传播交易和区块。

交易是如何传播的

比特币点对点网络中的节点是具有完全验证新交易正确性所需的软件逻辑和数据的程序。节点之间的连接通常被可视化为图中的边（线），而节点本身则是节点（点）。因此，比特币节点通常被称为“全验证节点”，或简称为全节点。

Alice的钱包应用可以通过任何类型的连接发送新交易到任何比特币节点：有线、WiFi、移动等。它还可以将交易发送到其他程序（如区块浏览器），该程序将其中继到节点。她的比特币钱包不必直接连接到Bob的比特币钱包，并且她也不必使用Bob提供的互联网连接，尽管这两种选择也是可能的。任何收到未见过的有效交易的比特币节点都会将其转发到其连接的所有其他节点，这种传播技术称为gossiping协议。因此，交易迅速传播到点对点网络中的大多数节点，在几秒钟内就能到达。

Bob的视角

如果Bob的比特币钱包应用直接连接到Alice的钱包应用，那么Bob的钱包应用可能是第一个接收到交易的。然而，即使Alice的钱包将交易通过其他节点发送，它也将在几秒钟内到达Bob的钱包。Bob的钱包将立即识别Alice的交易为一笔进账款，因为它包含了一个可由Bob的密钥兑现的输出。Bob的钱包应用程序还可以独立验证交易是否格式良好。如果Bob正在使用自己的完整节点，他的钱包可以进一步验证Alice的交易只花费了有效的UTXO。

比特币挖矿

Alice的交易现在已经在比特币网络上传播开来。它只有在被包括在一个区块中并由全节点验证之后，才会成为区块链的一部分。详细解释请参见第12章。

\ 比特币的防伪系统基于计算。交易被捆绑成区块。区块有一个非常小的头部，必须以非常特定的方式形成，需要巨大的计算量才能得到正确，但验证正确却只需要很少的计算量。挖矿过程在比特币中有两个目的：

- 矿工只能从符合所有比特币共识规则的区块中获得诚实的收入。因此，矿工通常被激励只在他们的区块和他们建立在上面的区块中包含有效的交易。这允许用户可以选择性地做一个基于信任的假设，即在区块中的任何交易都是有效的交易。
- 挖矿目前在每个区块中创建新的比特币，几乎像中央银行印刷新货币一样。每个区块中创建的比特币数量是有限的，并且随着时间的推移而减少，遵循固定的发行时间表。

挖矿实现了成本和回报之间的微妙平衡。挖矿使用电力来解决计算问题。成功的矿工将以新的比特币和交易费的形式获得奖励。然而，只有当矿工仅包含有效交易时，才会收到奖励，并且比特币协议的共识规则决定了什么是有效的。这种微妙的平衡为比特币提供了没有中央权威的安全性。

挖矿被设计成一种分散的抽奖。每个矿工可以通过创建一个包含他们想要挖掘的新交易和一些额外数据字段的候选区块来创建自己的彩票。矿工将候选区块输入到一个专门设计的算法中，该算法会对数据进行混淆（或“哈希”），产生的输出与输入数据完全不同。这个哈希函数对于相同的输入总是产生相同的输出，但是没有人可以预测对于一个新输入，即使与之前的输入略有不同，输出会是什么样子。如果哈希函数的输出与比特币协议确定的模板匹配，那么矿工就赢得了彩票，比特币用户将接受带有其交易的区块作为有效区块。如果输出与模板不匹配，矿工将对其候选区块进行微小更改，然后再次尝试。截至本文撰写时，矿工需要尝试的候选区块数量约为168亿亿。这也是哈希函数需要运行的次数。

然而，一旦找到了一个中奖组合，任何人都可以通过只运行一次哈希函数来验证区块是否有效。这使得有效区块成为需要大量工作才能创建，但只需要微不足道的工作来验证的东西。简单的验证过程能够以概率论的方式证明工作已经完成，因此生成这个证明所需的数据——在这种情况下就是区块——被称为工作证明（PoW）。

交易被添加到新的区块中，优先考虑具有最高费率的交易，并遵循其他几个标准。每个矿工在收到上一个区块后立即开始挖掘一个新的候选区块，知道其他某个矿工赢得了该轮抽奖。他们立即创建一个新的候选区块，并承诺与前一个区块相关联，填充交易，并开始为候选区块计算PoW。每个矿工在他们的候选区块中都包含一个特殊的交易，该交易支付他们自己的比特币地址，即区块奖励加上候选区块中包含的所有交易的交易费之和。如果他们找到了使候选区块成为有效区块的解决方案，那么在他们成功地将区块添加到全球区块链并且包含的奖励交易可以被花费之后，他们将获得这个奖励。参与矿池的矿工已经设置了他们的软件来创建分配奖励给矿池地址的候选区块。从那里，奖励的一部分按照他们贡献的工作量比例分配给矿池矿工的成员。

Alice的交易被网络捕获并包含在未验证交易的池中。一旦被全节点验证，它就被包含在一个候选区块中。大约五分钟后，Alice的交易首次被她的钱包传输后，一个矿工找到了区块的解，并将其宣布给网络。在其他每个矿工验证了获胜区块之后，他们开始生成下一个区块的新抽奖。

包含Alice的交易的获胜区块成为了区块链的一部分。包含Alice的交易的区块被视为对该交易的一个确认。在包含Alice的交易的区块通过网络传播后，要创建一个包含Alice的交易的另一个版本的替代区块（例如一个不支付Bob的交易）将需要执行与将所有比特币矿工创建一个完全新的区块相同的工作量。当存在多个可供选择的替代区块时，比特币全节点选择具有最高总PoW的有效区块链，称为最佳区块链。为了让整个网络接受一个替代区块，还需要在替代区块之上挖掘一个额外的新区块。

这意味着矿工有选择的余地。他们可以与Alice一起寻找一个替代方案，而不是支付给Bob的交易，也许Alice会支付给矿工之前支付给Bob的一部分钱。这种不诚实的行为将需要他们花费精力创建两个新区块。相反，行为诚实的矿工可以创建一个新区块，并收取其中包含的所有交易的费用，加上区块补贴。通常，以不诚实的方式创建两个区块以换取小额额外付款的高成本远远低于诚实地创建一个新区块，这使得确认的交易不太可能被故意更改。对于Bob来说，这意味着他可以开始相信来自Alice的支付是可靠的。

您可以查看包含Alice交易的区块。

\ 大约19分钟后，包含Alice交易的区块被广播，另一个矿工挖掘出了一个新区块。因为这个新区块是建立在包含Alice交易的区块之上的（给予Alice交易两个确认），Alice的交易现在只能在挖掘出两个替代区块——再加上一个建立在它们之上的新区块——的情况下才能被更改，总共需要挖掘三个区块才能让Alice收回她发送给Bob的钱。每个在包含Alice交易的区块之上挖掘的区块都被视为额外的确认。随着区块越来越多地堆叠在一起，撤销交易变得更加困难，从而让Bob对Alice的支付越来越有信心。

在图2-7中，我们可以看到包含Alice交易的区块。在它下面是数十万个区块，彼此链接在一起，形成一个区块链，一直回溯到块＃0，称为创世块。随着新区块的“高度”增加，整个链的计算难度也会增加。按照惯例，任何具有超过六个确认的区块都被认为非常难以更改，因为重新计算六个区块（加上一个新区块）需要大量计算。我们将在第12章中更详细地探讨挖矿过程以及它如何增强信心。

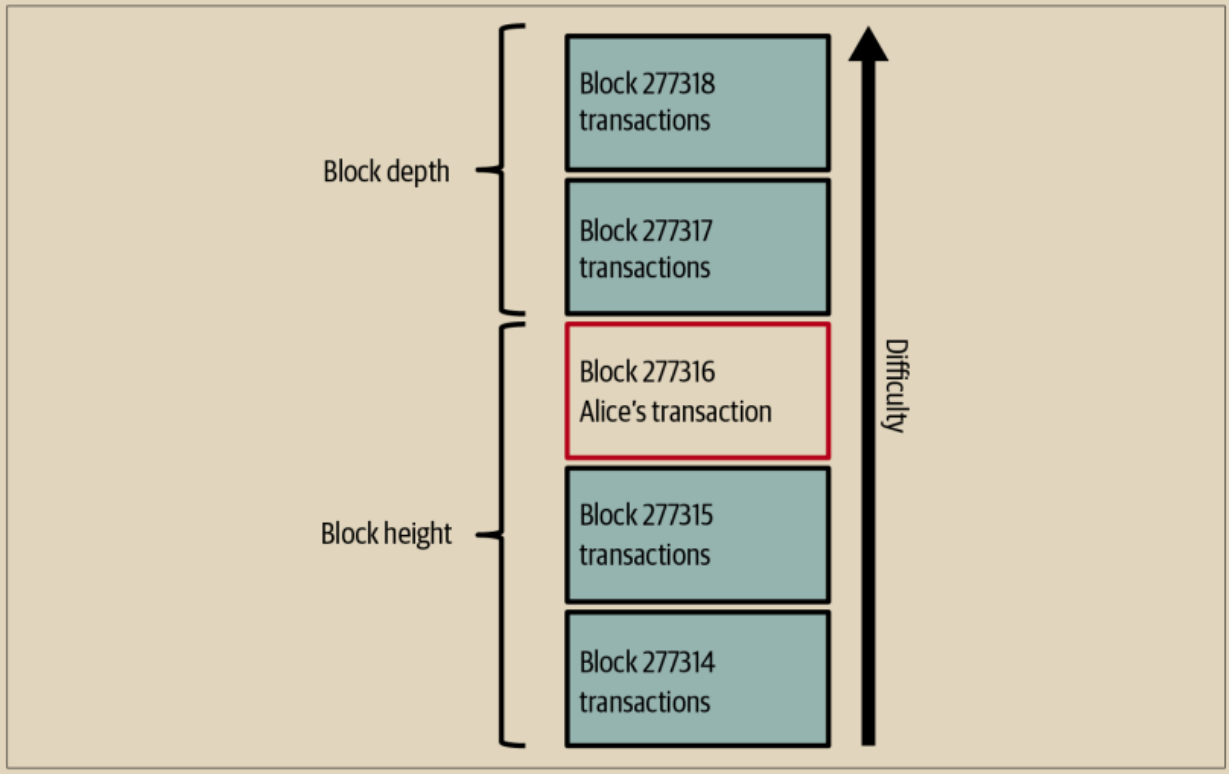


图 2-7. Alice的交易被包含在一个区块中

\ \

\

花费交易

现在，Alice的交易已经嵌入到区块链中作为一个区块的一部分，它对所有比特币应用程序都是可见的。每个比特币完整节点都可以独立验证交易的有效性和可支配性。完整节点验证从比特币首次生成在一个区块中开始的资金转移，直到它们到达Bob的地址的每一笔后续交易。轻量级客户端可以通过确认交易是否在区块链中，并在其后经过了几个区块的挖掘，从而部分验证支付，从而提供了矿工为其付出了大量努力的保证（参见“轻量级客户端”第228页）。

Bob现在可以花费来自此交易和其他交易的输出。例如，Bob可以通过从Alice的播客付款中转移价值来向承包商或供应商支付费用。随着Bob花费来自Alice和其他客户的付款，他扩展了交易链。假设Bob向他的网页设计师Gopesh支付了一个新的网站页面。现在，交易链将如图2-8所示。

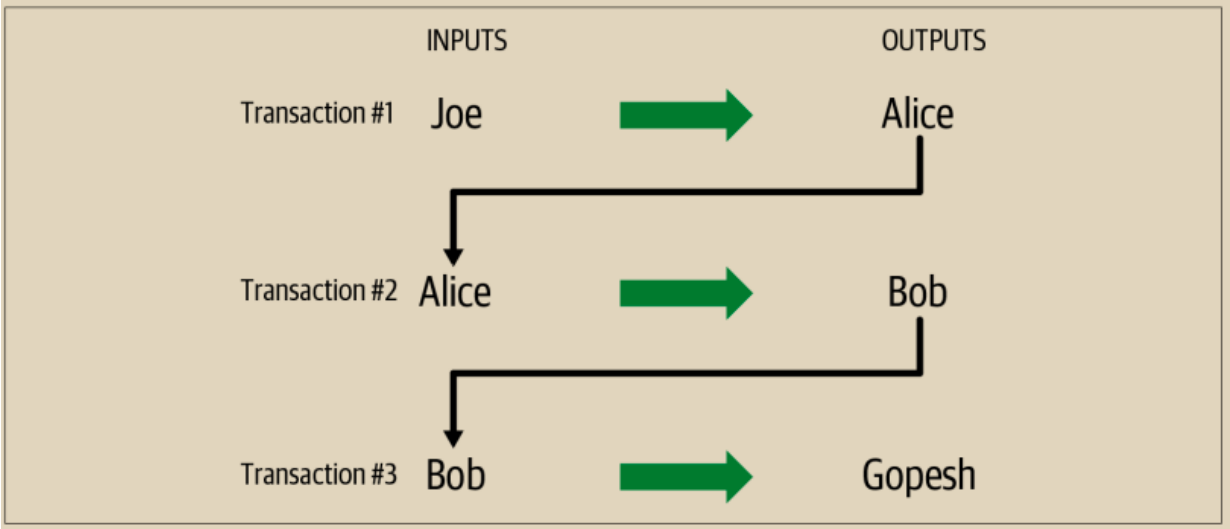


图 2-8. Alice的交易作为从Joe到Gopesh的交易链的一部分

在本章中，我们看到了交易如何构建一个链条，将价值从一个所有者转移到另一个所有者手中。我们还跟踪了Alice的交易，从她的钱包创建的那一刻开始，通过比特币网络，最终到记录在区块链上的矿工。在本书的其余部分，我们将深入探讨钱包、地址、签名、交易、网络和挖矿背后的具体技术。

综合介绍

在人们接受货币以换取他们宝贵的商品和服务时，他们相信自己将来能够使用这笔钱。如果货币是伪造的或者意外地贬值，那么这笔钱未来可能无法使用，因此每个接受比特币的人都有强烈的动机来验证他们收到的比特币的完整性。比特币系统被设计成，完全在您本地计算机上运行的软件可以完美地防止伪造、贬值和其他几个关键问题。提供该功能的软件称为完整验证节点，因为它会对系统中的每个已确认的比特币交易根据每条规则进行验证。完整验证节点，简称为完整节点，还可以提供了解比特币工作原理和当前网络状况的工具和数据。

在本章中，我们将安装比特币核心（Bitcoin Core），这是自比特币网络建立以来大多数完整节点操作员使用的实现。然后，我们将检查来自您节点的区块、交易和其他数据，这些数据是权威性的——不是因为某个强大的实体指定了它，而是因为您的节点独立验证了它。在本书的其余部分，我们将继续使用比特币核心来创建和检查与区块链和网络相关的数据。

从比特币到比特币核心

比特币是一个开源项目，其源代码可在开放（MIT）许可下获取，可供任何目的免费下载和使用。比仅仅是开源，比特币是由一个开放的志愿者社区开发的。起初，该社区只包括中本聪（Satoshi Nakamoto）。到2023年，比特币的源代码已经有了1000多名贡献者，其中大约有十几名开发人员全职工作在代码上，还有几十名在兼职基础上工作。任何人都可以为代码做出贡献——包括您！

当中本聪创建比特币时，软件在白皮书（见附录A）发布之前就已经基本完成了。中本聪希望在发表关于它的论文之前确保实现的可行性。那个最初的实现，当时简称为“比特币”，已经被大幅修改和改进。它已经演变成了我们所知的比特币核心，以区别于其他实现。比特币核心是比特币系统的参考实现，这意味着它提供了如何实现技术的参考。比特币核心实现了比特币的所有方面，包括钱包、交易和区块验证引擎、区块构建工具以及比特币点对点通信的所有现代部分。 图3-1显示了比特币核心的架构。

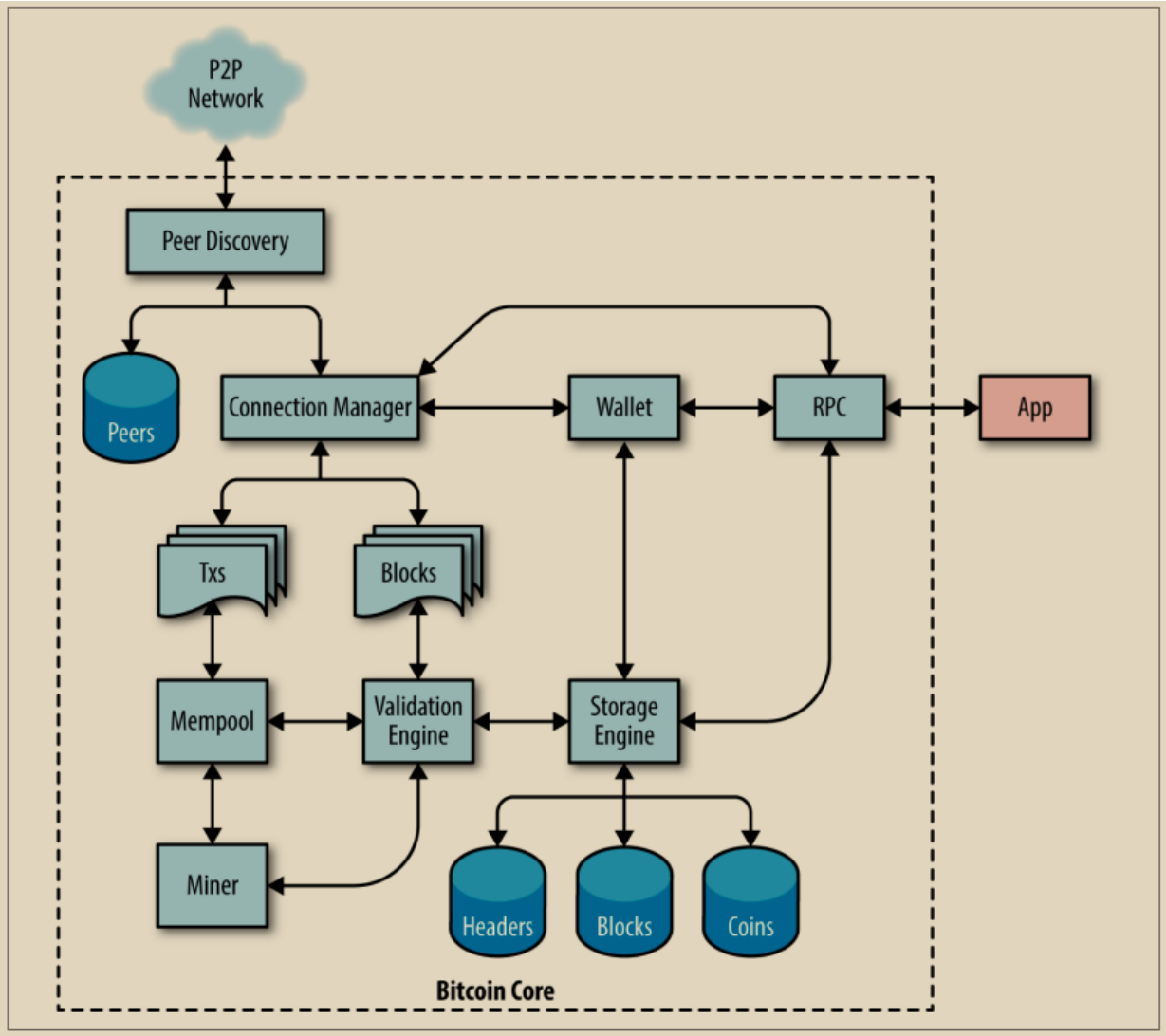


图 3-1. 比特币核心架构

虽然比特币核心作为系统的许多主要部分的参考实现，但比特币白皮书描述了系统的几个早期部分。自2011年以来，系统的大部分重要部分都已在一组比特币改进提案（BIPs）中进行了记录。在本书中，我们通过它们的编号引用BIP规范；例如，BIP9描述了用于几个重要的比特币升级的机制。

比特币开发环境

如果您是一名开发人员，您将希望建立一个开发环境，其中包括编写比特币应用程序所需的所有工具、库和支持软件。在这一高度技术的章节中，我们将逐步介绍这个过程。如果材料变得过于密集（而且您实际上并没有建立开发环境），请随时跳到下一章，那里的内容不那么技术性。

编译比特币核心的源代码

比特币核心的源代码可以通过下载存档文件或从GitHub克隆源代码存储库来获取。在比特币核心下载页面上，选择最新版本并下载源代码的压缩存档。或者，使用Git命令行从GitHub的比特币页面创建源代码的本地副本。

> 在本章的许多示例中，我们将使用操作系统的命令行界面（也称为“shell”），通过“终端”应用程序访问。Shell会显示一个提示符，您输入一个命令，Shell会用一些文本和一个新的提示符来回应您的下一个命令。提示符在您的系统上可能看起来不同，但在以下示例中，它用$符号表示。在示例中，当您看到$符号后面的文本时，不要输入$符号，而是直接输入紧随其后的命令，然后按Enter执行该命令。在示例中，每个命令下面的行是操作系统对该命令的响应。当您看到下一个$前缀时，您会知道这是一个新命令，您应该重复这个过程。

这里，我们使用git命令创建源代码的本地副本（“克隆”）：

```
$git clone https://github.com/bitcoin/bitcoin.git&#x20;

Cloning into 'bitcoin'...&#x20;

remote: Enumerating objects: 245912, done.&#x20;

remote: Counting objects: 100% (3/3), done.&#x20;

remote: Compressing objects: 100% (2/2), done.&#x20;

remote: Total 245912 (delta 1), reused 2 (delta 1), pack-reused 245909

Receiving objects: 100% (245912/245912), 217.74 MiB | 13.05 MiB/s, done.&#x20;

Resolving deltas: 100% (175649/175649), done.
```

> Git是最广泛使用的分布式版本控制系统，是任何软件开发人员工具包的必备部分。如果您尚未安装git命令或Git的图形用户界面，则可能需要在您的操作系统上安装它。

当Git克隆操作完成后，您将在名为bitcoin的目录中拥有完整的源代码存储库的本地副本。使用cd命令切换到此目录：

$cd bitcoin

选择比特币核心版本

默认情况下，本地副本将与最新的代码同步，这可能是比特币的一个不稳定或测试版本。在编译代码之前，请通过检出一个发布标签来选择一个特定版本。这将会将本地副本与代码存储库的特定快照同步，该快照由关键标签标识。标签被开发人员用于按版本号标记代码的特定发布。首先，要查找可用的标签，我们使用git tag命令：

\ $ git tag

v0.1.5

v0.1.6test1

v0.10.0

...

v0.11.2

v0.11.2rc1

v0.12.0rc1

v0.12.0rc2

...

标签列表显示了所有已发布的比特币版本。按照惯例，用于测试的预发布候选版本后缀为“rc”。可以在生产系统上运行的稳定版本没有后缀。从上述列表中选择最高版本的发布版本，在撰写本文时为v24.0.1。要将本地代码与此版本同步，请使用git checkout命令：

$ git checkout v24.0.1

Note: switching to 'v24.0.1'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

HEAD is now at b3f866a8d Merge bitcoin/bitcoin#26647: 24.0.1 final changes

您可以通过输入git status命令来确认您已经“检出”了所需的版本：

HEAD detached at v24.0.1

nothing to commit, working tree clean

配置比特币核心构建

源代码包括文档，可以在许多文件中找到。查看位于bitcoin目录中的README.md中的主要文档。在本章中，我们将在Linux上构建比特币核心守护进程（服务器），也称为bitcoind（类Unix系统）。通过阅读doc/build-unix.md上的指南来审阅有关在您的平台上编译bitcoind命令行客户端的说明。备用说明可在doc目录中找到；例如，Windows说明可在build-windows.md中找到。截至撰写本文时，Android、FreeBSD、NetBSD、OpenBSD、macOS（OSX）、Unix和Windows的说明均可用。

仔细查看构建文档的第一部分，其中包含构建先决条件。这些是必须在您的系统上存在的库，然后您才能开始编译比特币。如果这些先决条件缺失，构建过程将失败并显示错误。如果由于您遗漏了某些先决条件而导致此问题，则可以安装它，然后从中断处恢复构建过程。假设已安装了先决条件，则可以通过使用autogen.sh脚本生成一组构建脚本来启动构建过程：

\ $ ./autogen.sh

libtoolize: putting auxiliary files in AC_CONFIG_AUX_DIR, 'build-aux'.

libtoolize: copying file 'build-aux/ltmain.sh'

libtoolize: putting macros in AC_CONFIG_MACRO_DIRS, 'build-aux/m4'.

...

configure.ac:58: installing 'build-aux/missing'

src/Makefile.am: installing 'build-aux/depcomp'

parallel-tests: installing 'build-aux/test-driver'

autogen.sh脚本创建一组自动配置脚本，将查询您的系统以发现正确的设置，并确保您具有编译代码所需的所有必要库。其中最重要的是configure脚本，它提供了许多不同的选项来定制构建过程。使用--help标志查看各种选项：

$ ./configure --help

`configure' configures Bitcoin Core 24.0.1 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

...

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]

--enable-silent-rules less verbose build output (undo: "make V=1")

--disable-silent-rules verbose build output (undo: "make V=0")

...

configure脚本允许您通过--enable-FEATURE和--disable-FEATURE标志启用或禁用bitcoind的某些功能，其中FEATURE被替换为功能名称，如帮助输出中所列。在本章中，我们将使用所有默认功能构建bitcoind客户端。我们不会使用配置标志，但您应该查看它们以了解客户端的可选功能。如果您处于学术环境中，计算机实验室的限制可能要求您在您的主目录中安装应用程序（例如，使用--prefix=$HOME）。

以下是一些有用的选项，可以覆盖configure脚本的默认行为：

--prefix=$HOME

这将覆盖生成的可执行文件的默认安装位置（默认为/usr/local/）。使用$HOME将所有内容放在您的主目录中，或者使用其他路径。

--disable-wallet

这用于禁用参考钱包实现。

--with-incompatible-bdb

如果要构建钱包，则允许使用不兼容版本的 Berkeley DB 库。

--with-gui=no

不构建图形用户界面，这需要 Qt 库。这将仅构建服务器和命令行 Bitcoin Core。

\ 接下来，运行配置脚本，自动发现所有必要的库，并为您的系统创建一个定制的构建脚本：

\ $ ./configure

checking for pkg-config... /usr/bin/pkg-config

checking pkg-config is at least version 0.9.0... yes

checking build system type... x86_64-pc-linux-gnu

checking host system type... x86_64-pc-linux-gnu checking for a BSD-compatible install... /usr/bin/install -c

...

[many pages of configuration tests follow]

...

如果一切顺利，configure命令将结束，并创建定制的构建脚本，使我们能够编译bitcoind。如果存在任何缺少的库或错误，configure命令将以错误而不是创建构建脚本结束。如果出现错误，这很可能是由于缺少或不兼容的库引起的。再次查看构建文档，确保安装了缺少的先决条件，然后重新运行configure，看看是否解决了错误。\ \

构建比特币核心可执行文件

下一步，您将编译源代码，这个过程可能需要长达一小时才能完成，具体取决于您的CPU速度和可用内存。如果出现错误或编译过程中断，可以随时通过再次键入 make 来恢复。输入 make 开始编译可执行应用程序：

$ make

Making all in src

CXX bitcoind-bitcoind.o

CXX libbitcoin_node_a-addrdb.o

CXX libbitcoin_node_a-addrman.o

CXX libbitcoin_node_a-banman.o

CXX libbitcoin_node_a-blockencodings.o

CXX libbitcoin_node_a-blockfilter.o

[... many more compilation messages follow ...]\ \ 在性能较快且具有多个CPU的系统上，您可能希望设置并行编译作业的数量。例如， `make -j 2` 将使用两个可用的核心。如果一切顺利，比特币核心现在已经编译完成。您应该使用 `make check` 运行单元测试套件，以确保链接的库没有明显的问题。最后一步是使用 `make install` 命令在您的系统上安装各种可执行文件。由于此步骤需要管理员权限，因此您可能需要输入您的用户密码：

$ make check && sudo make install

Password:

Making install in src

../build-aux/install-sh -c -d '/usr/local/lib'

libtool: install: /usr/bin/install -c bitcoind /usr/local/bin/bitcoind

libtool: install: /usr/bin/install -c bitcoin-cli /usr/local/bin/bitcoin-cli

libtool: install: /usr/bin/install -c bitcoin-tx /usr/local/bin/bitcoin-tx

...

默认情况下，bitcoind 的安装位置是 `/usr/local/bin` 。您可以通过询问系统执行文件的路径来确认 Bitcoin Core 是否正确安装，方法如下：

$ which bitcoind

/usr/local/bin/bitcoind

$ which bitcoin-cli

/usr/local/bin/bitcoin-cli

运行比特币核心节点

比特币的点对点网络由网络“节点”组成，主要由个人和提供比特币服务的一些企业运行。那些运行比特币节点的人可以直接且权威地查看比特币区块链，拥有自己系统独立验证的所有可消费比特币的本地副本。通过运行节点，您不必依赖任何第三方来验证交易。此外，通过使用比特币节点对您钱包收到的交易进行全面验证，您可以为比特币网络做出贡献，帮助使其更加强大。

然而，运行节点需要下载和处理超过 500 GB 的数据，以及每天约 400 MB 的比特币交易数据。这些数字是针对 2023 年的情况，并且随着时间的推移可能会增加。如果您关闭节点或者与互联网断开连接了几天，您的节点将需要下载错过的数据。例如，如果您关闭比特币核心 10 天，那么下次启动它时，您将需要下载约 4 GB 的数据。

取决于您是否选择索引所有交易并保留完整的区块链副本，您可能还需要大量的磁盘空间——如果您计划运行比特币核心数年，至少需要 1 TB 的磁盘空间。默认情况下，比特币节点还会将交易和区块传输给其他节点（称为“对等节点”），消耗上传互联网带宽。如果您的互联网连接受限，有低数据上限，或者是按千兆比特收费的，那么您可能不应该在其上运行比特币节点，或者以限制其带宽的方式运行它（请参阅“配置比特币核心节点”）。您可以将节点连接到其他网络，例如免费的卫星数据提供商，如 Blockstream Satellite。

> 比特币核心默认会保留完整的区块链副本，其中包含自2009年以来比特币网络上几乎每笔确认交易的记录。这个数据集的大小达到数百GB，并且会根据您的CPU速度和互联网连接速度在几个小时或几天内逐步下载完成。在完整的区块链数据集下载完成之前，比特币核心将无法处理交易或更新账户余额。确保您有足够的磁盘空间、带宽和时间来完成初始同步。您可以配置比特币核心以减少区块链的大小，通过丢弃旧的区块，但它仍然会下载整个数据集。

尽管存在这些资源要求，成千上万的人都在运行比特币节点。有些人甚至在像树莓派（一台35美元的计算机，大小如一副牌）这样简单的系统上运行比特币节点。

你为什么想要运行一个节点呢？以下是一些最常见的原因：

- 你不想依赖任何第三方来验证你接收到的交易。
- 你不想向第三方披露哪些交易属于你的钱包。
- 你正在开发比特币软件，并且需要依赖比特币节点进行对网络和区块链的可编程（API）访问。
- 你正在构建必须根据比特币共识规则验证交易的应用程序。通常，比特币软件公司会运行多个节点。
- 你想支持比特币。运行一个节点用于验证你接收到的钱包交易，可以使网络更加强大。

如果你正在阅读本书并对强大的安全性、优越的隐私性或开发比特币软件感兴趣，你应该运行自己的节点。

配置比特币核心节点

比特币核心在每次启动时都会在其数据目录中寻找配置文件。在本节中，我们将研究各种配置选项并设置一个配置文件。要查找配置文件，请在终端中运行bitcoind -printtoconsole，并查找前几行：

$ bitcoind -printtoconsole

2023-01-28T03:21:42Z Bitcoin Core version v24.0.1

2023-01-28T03:21:42Z Using the 'x86_shani(1way,2way)' SHA256 implementation

2023-01-28T03:21:42Z Using RdSeed as an additional entropy source

2023-01-28T03:21:42Z Using RdRand as an additional entropy source

2023-01-28T03:21:42Z Default data directory /home/harding/.bitcoin

2023-01-28T03:21:42Z Using data directory /home/harding/.bitcoin

2023-01-28T03:21:42Z Config file: /home/harding/.bitcoin/bitcoin.conf

...

[a lot more debug output]

...

\ 你可以按 Ctrl-C 关闭节点，一旦确定配置文件的位置。通常，配置文件位于用户的主目录下的 .bitcoin 数据目录中。用你喜欢的编辑器打开配置文件。Bitcoin Core 提供了100多个配置选项，可以修改网络节点的行为、区块链的存储以及其操作的许多其他方面。要查看这些选项的列表，请运行 `bitcoind --help`：

$ bitcoind --help

Bitcoin Core version v24.0.1

Usage: bitcoind [options] Start Bitcoin Core

Options:

-?

Print this help message and exit

-alertnotify=\

Execute command when an alert is raised (%s in cmd is replaced by message)

...

[many more options]

以下是您可以在配置文件中设置的一些最重要的选项，或作为 bitcoind 的命令行参数：

alertnotify

运行指定的命令或脚本，向此节点的所有者发送紧急警报。

conf

配置文件的替代位置。这只有作为 bitcoind 的命令行参数才有意义，因为它不能在所指的配置文件内部。

datadir

选择存储所有区块链数据的目录和文件系统。默认情况下，这是您的主目录下的 .bitcoin 子目录。根据您的配置，到目前为止，这可能使用约 10 GB 到接近 1 TB 的空间，预计最大大小每年将增加数百 GB。

prune

通过删除旧的区块，将区块链磁盘空间要求减少到这么多兆字节。在资源受限的节点上使用此选项，无法容纳完整的区块链。系统的其他部分将使用其他磁盘空间，目前无法进行修剪，因此您仍然需要至少 datadir 选项中提到的最小空间量。\

txindex

维护所有交易的索引。这使您能够通过其ID以编程方式检索任何交易，前提是包含该交易的区块尚未被修剪。

dbcache

UTXO 缓存的大小。默认值为 450 Mebibytes（MiB）。在高端硬件上增加此大小，以减少磁盘读写次数，或者在低端硬件上减小大小，以节省内存，但会增加磁盘的使用频率。

blocksonly

通过仅接受经过确认的交易的区块，而不是中继未经确认的交易，来最大限度地减少带宽使用。

maxmempool

限制交易内存池到指定的兆字节数。这可用于减少内存受限节点上的内存使用。

> **交易数据库索引和txindex选项**
>
> 默认情况下，Bitcoin Core构建一个仅包含与用户钱包相关的交易的数据库。如果您想要能够使用诸如getrawtransaction（请参阅“探索和解码交易”）之类的命令访问任何交易，则需要配置Bitcoin Core以构建完整的交易索引，可以通过txindex选项实现。在Bitcoin Core配置文件中设置txindex=1。如果一开始没有设置此选项，而后来将其设置为完整索引，则需要等待其重建索引。

示例3-1 展示了如何将前面的选项与一个完全索引的节点结合起来，作为比特币应用程序的API后端运行。

示例3-1。完全索引节点的示例配置

alertnotify=myemailscript.sh "Alert: %s"

datadir=/lotsofspace/bitcoin

txindex=1

示例3-2 展示了在较小服务器上运行的资源受限节点的示例配置。

示例3-2。资源受限系统的示例配置

alertnotify=myemailscript.sh "Alert: %s"

blocksonly=1

prune=5000

dbcache=150

maxmempool=150\

在编辑配置文件并设置最符合您需求的选项后，您可以使用这个配置测试bitcoind。使用 printtoconsole 选项运行 Bitcoin Core，以在前台输出到控制台：

$ bitcoind -printtoconsole

2023-01-28T03:43:39Z Bitcoin Core version v24.0.1

2023-01-28T03:43:39Z Using the 'x86_shani(1way,2way)' SHA256 implementation

2023-01-28T03:43:39Z Using RdSeed as an additional entropy source

2023-01-28T03:43:39Z Using RdRand as an additional entropy source

2023-01-28T03:43:39Z Default data directory /home/harding/.bitcoin

2023-01-28T03:43:39Z Using data directory /lotsofspace/bitcoin

2023-01-28T03:43:39Z Config file: /home/harding/.bitcoin/bitcoin.conf

2023-01-28T03:43:39Z Config file arg: [main] blockfilterindex="1"

2023-01-28T03:43:39Z Config file arg: [main] maxuploadtarget="1000"

2023-01-28T03:43:39Z Config file arg: [main] txindex="1"

2023-01-28T03:43:39Z Setting file arg: wallet = ["msig0"]

2023-01-28T03:43:39Z Command-line arg: printtoconsole=""

2023-01-28T03:43:39Z Using at most 125 automatic connections

2023-01-28T03:43:39Z Using 16 MiB out of 16 MiB requested for signature cache

2023-01-28T03:43:39Z Using 16 MiB out of 16 MiB requested for script execution

2023-01-28T03:43:39Z Script verification uses 3 additional threads

2023-01-28T03:43:39Z scheduler thread start

2023-01-28T03:43:39Z [http] creating work queue of depth 16

2023-01-28T03:43:39Z Using random cookie authentication.

2023-01-28T03:43:39Z Generated RPC cookie /lotsofspace/bitcoin/.cookie

2023-01-28T03:43:39Z [http] starting 4 worker threads

2023-01-28T03:43:39Z Using wallet directory /lotsofspace/bitcoin/wallets

2023-01-28T03:43:39Z init message: Verifying wallet(s)…

2023-01-28T03:43:39Z Using BerkeleyDB version Berkeley DB 4.8.30

2023-01-28T03:43:39Z Using /16 prefix for IP bucketing

2023-01-28T03:43:39Z init message: Loading P2P addresses…

2023-01-28T03:43:39Z Loaded 63866 addresses from peers.dat 114ms

[... more startup messages ...]

\ 您可以在满意加载了正确设置并且运行如您所期望的情况下，按下 Ctrl-C 来中断该过程。

要将Bitcoin Core作为后台进程运行，请使用守护进程选项启动它，例如 `bitcoind -daemon` 。

要监视您的Bitcoin节点的进度和运行时状态，请以守护进程模式启动它，然后使用命令 `bitcoin-cli getblockchaininfo` 。

$ bitcoin-cli getblockchaininfo

```
{
    "chain": "main",
    "blocks": 0,
    "headers": 83999,
    "bestblockhash": "[...]19d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
    "difficulty": 1,
    "time": 1673379796,
    "mediantime": 1231006505,
    "verificationprogress": 3.783041623201835e-09,
    "initialblockdownload": true,
    "chainwork": "[...]0000000000000000000000000000000000000000000000100010001",
    "size_on_disk": 89087,
    "pruned": false,
    "warnings": ""
}
```


这显示了一个区块链高度为0个区块和83,999个头部的节点。节点首先从其对等节点获取块头，以找到具有最多工作量证明的区块链，然后继续下载完整的块，并在此过程中进行验证。

一旦您对所选择的配置选项感到满意，您应该将Bitcoin Core添加到操作系统的启动脚本中，以便它持续运行，并在操作系统重新启动时重新启动。您会在Bitcoin Core的源目录中的contrib/init下找到各种操作系统的示例启动脚本，以及一个README.md文件，显示哪个系统使用哪个脚本。

\\\

\

比特币核心 API

比特币核心 API 提供了一组命令，用于通过编程方式与运行中的比特币核心节点进行交互。这些命令可以通过 JSON-RPC 接口或使用 `bitcoin-cli` 命令行工具访问。以下是比特币核心 API 中常见的一些命令：

$ bitcoin-cli help

+== Blockchain ==

getbestblockhash

getblock "blockhash" (verbosity)

getblockchaininfo

...

walletpassphrase "passphrase" timeout

walletpassphrasechange "oldpassphrase" "newpassphrase"

walletprocesspsbt "psbt" (sign "sighashtype" bip32derivs finalize)

每个命令都可以接受多个参数。要获取更多帮助、详细描述以及有关参数的信息，请在 `help` 后面添加命令名称。例如，要查看关于 `getblockhash` RPC 命令的帮助：

$ bitcoin-cli help getblockhash

getblockhash height

Returns hash of block in best-block-chain at height provided.

Arguments:

1. height (numeric, required) The height index

Result: "hex" (string) The block hash

Examples:

>bitcoin-cli getblockhash 1000

> curl --user myusername --data-binary '{"jsonrpc": "1.0", "id": "curltest",

"method": "getblockhash",

"params": [1000]}' -H 'content-type: text/plain;' http://127.0.0.1:8332/

在帮助信息的末尾，你会看到两个RPC命令的示例，使用了bitcoin-cli助手或HTTP客户端curl。这些示例演示了如何调用该命令。复制第一个示例并查看结果：

$ bitcoin-cli getblockhash 1000 00000000c937983704a73af28acdec37b049d214adbda81d7e2a3dd146f6ed09

结果是一个区块哈希，在接下来的章节中将会详细介绍。但是目前，该命令应该在你的系统上返回相同的结果，表明你的Bitcoin Core节点正在运行，接受命令，并且有关于区块 1,000 的信息返回给你。

在接下来的章节中，我们将演示一些非常有用的RPC命令及其预期的输出。\

获取比特币核心状态信息

比特币核心通过 JSON-RPC 接口提供不同模块的状态报告。最重要的命令包括 getblockchaininfo、getmempoolinfo、getnetworkinfo 和 getwalletinfo。

比特币的 getblockchaininfo RPC 命令早已介绍过。getnetworkinfo 命令显示有关比特币网络节点状态的基本信息。使用 bitcoin-cli 运行它：

$ bitcoin-cli getnetworkinfo

```
{
    "version": 240001,
    "subversion": "/Satoshi:24.0.1/",
    "protocolversion": 70016,
    "localservices": "0000000000000409",
    "localservicesnames": [
        "NETWORK",
        "WITNESS",
        "NETWORK_LIMITED"
    ],
    "localrelay": true,
    "timeoffset": -1,
    "networkactive": true,
    "connections": 10,
    "connections_in": 0,
    "connections_out": 10,
    "networks": [
        "...detailed information about all networks..."
    ],
    "relayfee": 0.00001000,
    "incrementalfee": 0.00001000,
    "localaddresses": [],
    "warnings": ""
}
```

返回的数据采用 JavaScript 对象表示法（JSON）格式，这种格式可以轻松地被所有编程语言“消费”，但也非常易读。在这些数据中，我们看到比特币核心软件和比特币协议的版本号。我们还可以看到当前连接数以及有关比特币网络和与此节点相关的各种信息设置。

> bitcoind 需要一些时间，也许超过一天的时间，来赶上当前的区块链高度，因为它要从其他比特币节点下载区块并验证这些区块中的每一笔交易 —— 在撰写本文时，这些交易已经接近 10 亿笔。您可以使用 getblockchaininfo 来检查其进度，以查看已知区块的数量。本章后续示例假定您至少已经到达了区块 775,072。因为比特币交易的安全性取决于区块，所以以下示例中的一些信息将根据您的节点有多少区块而略有变化

探索和解码交易

在“从在线商店购买”中，Alice从Bob的商店购买了商品。她的交易已记录在区块链上。让我们使用 API 通过传递交易ID（txid）作为参数来检索和检查该交易：

\ $ bitcoin-cli getrawtransaction 466200308696215bbc949d5141a49a41\\ 38ecdfdfaa2a8029c1f9bcecd1f96177

```
01000000000101eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da13569
8679268041c54a0100000000ffffffff02204e00000000000022512 03b41daba
4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068f8240100
0000000016001477 52c165ea7be772b2c0acb7f4d6047ae6f4768e0141cf5efe
2d8ef13ed0af21d4f4cb82422d6252d70324f6f4576b727b7d918e521c00b51b
e739df2f899c49dc267c0ad280aca6dab0d2fa2b42a45182fc83e817130100000000
```

> 交易ID（txid）并不具有权威性。在区块链中找不到txid并不意味着该交易未被处理。这被称为“交易可塑性”，因为在交易确认之前，交易可以被修改，从而改变其txid。一旦交易被包含在一个区块中，其txid就不能更改，除非发生了区块链重组，其中该区块从最佳区块链中被移除。在交易获得几个确认之后，区块链重组变得十分罕见。

命令getrawtransaction以十六进制表示返回一个序列化的交易。为了解码它，我们使用decoderawtransaction命令，并将十六进制数据作为参数传递。您可以复制getrawtransaction返回的十六进制数据，并将其粘贴为decoderawtransaction的参数：

```
$ bitcoin-cli decoderawtransaction 01000000000101eb3ae38f27191aa5f3850dc9cad0\
0492b88b72404f9da135698679268041c54a0100000000ffffffff02204e00000000000022512\
03b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068f82401000000\
00001600147752c165ea7be772b2c0acb7f4d6047ae6f4768e0141cf5efe2d8ef13ed0af21d4f\
4cb82422d6252d70324f6f4576b727b7d918e521c00b51be739df2f899c49dc267c0ad280aca6\
dab0d2fa2b42a45182fc83e817130100000000
```

```
{
    "txid": "466200308696215bbc949d5141a49a4138ecdfdfaa2a8029c1f9bcecd1f96177",
    "hash": "f7cdbc7cf8b910d35cc69962e791138624e4eae7901010a6da4c02e7d238cdac",
    "version": 1,
    "size": 194,
    "vsize": 143,
    "weight": 569,
    "locktime": 0,
    "vin": [{
        "txid": "4ac541802679866935a19d4f40728bb89204d0cac90d85f3a51a19...aeb",
        "vout": 1,
        "scriptSig": {
            "asm": "",
            "hex": ""
        },
        "txinwitness": [
            "cf5efe2d8ef13ed0af21d4f4cb82422d6252d70324f6f4576b727b7d918e5...301"
        ],
        "sequence": 4294967295
    }],
    "vout": [{
            "value": 0.00020000,
            "n": 0,
            "scriptPubKey": {
                "asm": "1 3b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca...068",
                "desc": "rawtr(3b41daba4c9ace578369740f15e5ec880c28279ee7f51b...6ev",
                "hex": "51203b41daba4c9ace578369740f15e5ec880c28279ee7f51b07d...068",
                "address": "bc1p8dqa4wjvnt890qmfws83te0v3qxzsfu7ul63kp7u56w8q...5qn",
                "type": "witness_v1_taproot"
            }
        },
        {
            "value": 0.00075000,
            "n": 1,
            "scriptPubKey": {
                "asm": "0 7752c165ea7be772b2c0acb7f4d6047ae6f4768e",
                "desc": "addr(bc1qwafvze0200nh9vkq4jmlf4sy0tn0ga5w0zpkpg)#qq404gts",
                "hex": "00147752c165ea7be772b2c0acb7f4d6047ae6f4768e",
                "address": "bc1qwafvze0200nh9vkq4jmlf4sy0tn0ga5w0zpkpg",
                "type": "witness_v0_keyhash"
            }
        }
    ]
}
```


\ 交易解码显示了此交易的所有组件，包括交易输入和输出。在这种情况下，我们看到该交易使用了一个输入，并生成了两个输出。这笔交易的输入是来自先前确认的交易的输出（显示为输入txid）。这两个输出对应于付款给Bob和找零返回给Alice。

我们可以通过使用相同的命令（例如getrawtransaction）检查在此交易中引用其txid的先前交易来进一步探索区块链。从一笔交易跳转到另一笔交易，我们可以跟踪一系列交易，从而了解硬币是如何从一个所有者传递到下一个所有者的。

\

探索区块

探索区块与探索交易类似。然而，区块可以通过区块高度或区块哈希来引用。首先，让我们通过区块高度找到一个区块。我们使用getblockhash命令，该命令以区块高度作为参数，并返回该区块的区块头哈希：

$ bitcoin-cli getblockhash 123456

0000000000002917ed80650c6174aac8dfc46f5fe36480aaef682ff6cd83c3ca

现在我们知道了我们选择的区块的区块头哈希，我们可以查询该区块。我们使用getblock命令，并将区块哈希作为参数：

$ bitcoin-cli getblock 0000000000002917ed80650c6174aac8dfc46f5fe36480aaef682f\ f6cd83c3ca


```
{
    "hash": "0000000000002917ed80650c6174aac8dfc46f5fe36480aaef682ff6cd83c3ca",
    "confirmations": 651742,
    "height": 123456,
    "version": 1,
    "versionHex": "00000001",
    "merkleroot": "0e60651a9934e8f0decd1c[...]48fca0cd1c84a21ddfde95033762d86c",
    "time": 1305200806,
    "mediantime": 1305197900,
    "nonce": 2436437219,
    "bits": "1a6a93b3",
    "difficulty": 157416.4018436489,
    "chainwork": "[...]0000000000000000000000000000000000000000541788211ac227bc",
    "nTx": 13,
    "previousblockhash": "[...]60bc96a44724fd72daf9b92cf8ad00510b5224c6253ac40095",
    "nextblockhash": "[...]00129f5f02be247070bf7334d3753e4ddee502780c2acaecec6d66",
    "strippedsize": 4179,
    "size": 4179,
    "weight": 16716,
    "tx": [
        "5b75086dafeede555fc8f9a810d8b10df57c46f9f176ccc3dd8d2fa20edd685b",
        "e3d0425ab346dd5b76f44c222a4bb5d16640a4247050ef82462ab17e229c83b4",
        "137d247eca8b99dee58e1e9232014183a5c5a9e338001a0109df32794cdcc92e",
        "5fd167f7b8c417e59106ef5acfe181b09d71b8353a61a55a2f01aa266af5412d",
        "60925f1948b71f429d514ead7ae7391e0edf965bf5a60331398dae24c6964774",
        "d4d5fc1529487527e9873256934dfb1e4cdcb39f4c0509577ca19bfad6c5d28f",
        "7b29d65e5018c56a33652085dbb13f2df39a1a9942bfe1f7e78e97919a6bdea2",
        "0b89e120efd0a4674c127a76ff5f7590ca304e6a064fbc51adffbd7ce3a3deef",
        "603f2044da9656084174cfb5812feaf510f862d3addcf70cacce3dc55dab446e",
        "9a4ed892b43a4df916a7a1213b78e83cd83f5695f635d535c94b2b65ffb144d3",
        "dda726e3dad9504dce5098dfab5064ecd4a7650bfe854bb2606da3152b60e427",
        "e46ea8b4d68719b65ead930f07f1f3804cb3701014f8e6d76c4bdbc390893b94",
        "864a102aeedf53dd9b2baab4eeb898c5083fde6141113e0606b664c41fe15e1f"
    ]
}
```

确认（confirmations）条目告诉我们这个区块的深度——有多少个区块是在其之上构建的，这表明改变该区块中任何交易的难度。高度告诉我们在该区块之前有多少个区块。我们看到了该区块的版本、创建时间（根据其矿工）、之前的11个区块的中位时间（对矿工来说更难以操纵的时间测量），以及区块的大小以三种不同的度量方式（其传统的剥离大小、完整大小和权重单位中的大小）。我们还看到了一些用于安全性和工作证明的字段（默克尔根、随机数、位、难度和链工作量）；我们将在第12章中对这些进行详细探讨。

\

使用比特币核心的编程接口

比特币命令行助手 bitcoin-cli 对于探索比特币核心API和测试功能非常有用。但是，API的整个目的在于以编程方式访问函数。在本节中，我们将演示如何从另一个程序访问比特币核心。

\ 比特币核心的API是一个JSON-RPC接口。JSON是一种非常方便的数据呈现方式，可以让人类和程序都轻松读取。RPC代表远程过程调用，这意味着我们通过网络协议调用远程（位于比特币核心节点上的）过程（函数）。在这种情况下，网络协议是HTTP。

\ 当我们使用bitcoin-cli命令获取命令的帮助时，它向我们展示了使用curl的示例，这是一种功能多样的命令行HTTP客户端，用于构建这些JSON-RPC调用之一：

\ $ curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest",

"method": "getblockchaininfo",

"params": [] }' -H 'content-type: text/plain;' http://127.0.0.1:8332/

该命令显示了 curl 发送一个 HTTP 请求到本地主机 (127.0.0.1)，连接到默认的比特币 RPC 端口 (8332)，并使用纯文本编码提交一个 jsonrpc 请求，请求 getblockchaininfo 方法。

您可能会注意到 curl 会要求发送凭据以及请求一起发送。比特币核心在每次启动时会创建一个随机密码，并将其放置在数据目录下，命名为 .cookie。bitcoin-cli 辅助工具可以在给定数据目录的情况下读取此密码文件。类似地，您可以复制密码并将其传递给 curl (或任何更高级别的比特币核心 RPC 封装)，如示例 3-3 所示。

示例 3-3. 使用基于 cookie 的身份验证与比特币核心

$ cat .bitcoin/.cookie **cookie**:17c9b71cef21b893e1a019f4bc071950c7942f49796ed061b274031b17b19cd0

$ curl --user **cookie**:17c9b71cef21b893e1a019f4bc071950c7942f49796ed061b274031b17b19cd0 --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getblockchaininfo", "params": [] }' -H 'content-type: text/plain;' http://127.0.0.1:8332/

{"result":{"chain":"main","blocks":799278,"headers":799278,
"bestblockhash":"00000000000000000018387c50988ec705a95d6f765b206b6629971e6978879",
"difficulty":53911173001054.59,"time":1689703111,"mediantime":1689701260,
"verificationprogress":0.9999979206082515,"initialblockdownload":false,
"chainwork":"0000000000000000000000000000000000000004f3e111bf32bcb47f9dfad5b",
"size_on_disk":563894577967,"pruned":false,"warnings":""},"error":null, "id":"curltest"}

或者，您可以使用 Bitcoin Core 源代码目录中提供的辅助脚本 `./share/rpcauth/rpcauth.py` 创建一个静态密码。

如果您正在自己的程序中实现 JSON-RPC 调用，您可以使用通用的 HTTP 库来构建该调用，类似于前面的 curl 示例所示。

然而，大多数流行的编程语言都有库可以以更简单的方式“包装” Bitcoin Core API。我们将使用 python-bitcoinlib 库来简化 API 访问。该库不是 Bitcoin Core 项目的一部分，需要按照通常的 Python 库安装方式进行安装。 请记住，这需要您运行一个 Bitcoin Core 实例，该实例将用于进行 JSON-RPC 调用。

示例 3-4 中的 Python 脚本执行了一个简单的 `getblockchaininfo` 调用，并打印了 Bitcoin Core 返回数据中的块参数。

示例 3-4. 通过 Bitcoin Core 的 JSON-RPC API 运行 getblockchaininfo

```
from bitcoin.rpc import RawProxy
# Create a connection to local Bitcoin Core node
p = RawProxy()
# Run the getblockchaininfo command, store the resulting data in info
info = p.getblockchaininfo()
# Retrieve the 'blocks' element from the info
print(info['blocks'])
```

运行后，我们得到以下结果：

$ python rpc_example.py

773973

它告诉我们本地 Bitcoin Core 节点的区块链中有多少个区块。这并不是令人惊讶的结果，但它演示了使用该库作为简化接口访问 Bitcoin Core 的 JSON-RPC API 的基本用法。

接下来，让我们使用 getrawtransaction 和 decodetransaction 调用来检索 Alice 对 Bob 的付款的详细信息。在示例 3-5 中，我们检索 Alice 的交易并列出交易的输出。对于每个输出，我们显示接收者地址和价值。作为提醒，Alice 的交易有一个输出支付给 Bob，另一个输出是 Alice 的找零。

示例 3-5. 检索交易并迭代其输出

```
from bitcoin.rpc import RawProxy
p = RawProxy()
# Alice's transaction ID
txid = "466200308696215bbc949d5141a49a4138ecdfdfaa2a8029c1f9bcecd1f96177"
# First, retrieve the raw transaction in hex
raw_tx = p.getrawtransaction(txid)
# Decode the transaction hex into a JSON object
decoded_tx = p.decoderawtransaction(raw_tx)
# Retrieve each of the outputs from the transaction
for output in decoded_tx['vout']:
    print(output['scriptPubKey']['address'], output['value'])
```

运行此代码，我们得到：

$ python rpc_transaction.py bc1p8dqa4wjvnt890qmfws83te0v3qxzsfu7ul63kp7u56w8qc0qwp5qv995qn 0.00020000 bc1qwafvze0200nh9vkq4jmlf4sy0tn0ga5w0zpkpg 0.00075000

前面的两个例子都相当简单。你不真的需要一个程序来运行它们；你可以同样使用bitcoin-cli助手。然而，下一个例子需要几百个RPC调用，并更清晰地展示了编程接口的使用。

在示例3-6中，我们首先检索一个块，然后通过引用每个交易ID检索其中的每个交易。接下来，我们遍历每个交易的输出，并累加其值。

示例3-6。检索一个块并添加所有交易输出

```
from bitcoin.rpc import RawProxy

p = RawProxy()

# The block height where Alice's transaction was recorded
blockheight = 775072

# Get the block hash of the block at the given height
blockhash = p.getblockhash(blockheight)

# Retrieve the block by its hash
block = p.getblock(blockhash)

# Element tx contains the list of all transaction IDs in the block
transactions = block['tx']

block_value = 0
# Iterate through each transaction ID in the block
for txid in transactions:
    tx_value = 0
    # Retrieve the raw transaction by ID
    raw_tx = p.getrawtransaction(txid)
    # Decode the transaction
    decoded_tx = p.decoderawtransaction(raw_tx)
    # Iterate through each output in the transaction
    for output in decoded_tx['vout']:
        # Add up the value of each output
        tx_value = tx_value + output['value']
    # Add the value of this transaction to the total
    block_value = block_value + tx_value
print("Total value in block: ", block_value)
```

运行这段代码，我们得到：

\ $ python rpc_block.py

Total value in block: 10322.07722534

我们的示例代码计算出这个区块中的总交易价值为10,322.07722534 BTC（包括25 BTC的奖励和0.0909 BTC的手续费）。与通过搜索区块哈希或高度在区块浏览器网站上报告的金额进行比较。一些区块浏览器报告的总值不包括奖励和手续费。看看你能否发现差异。

\

不同语言客户端、库和工具包

在比特币生态系统中，有许多替代客户端、库、工具包甚至是完整的节点实现。这些实现采用了各种编程语言，为程序员提供了他们喜欢的编程语言的本机接口。

以下各节列出了一些最好的库、客户端和工具包，按照编程语言进行组织。

C/C++

Bitcoin Core

比特币的参考实现

JavaScript

bcoin

一个模块化且可扩展的带有 API 的全节点实现

Bitcore

由 Bitpay 提供的全节点、API 和库

BitcoinJS

一个纯 JavaScript 的 Bitcoin 库，适用于 node.js 和浏览器

Java

bitcoinj

一个 Java 完整节点客户端库

Python

python-bitcoinlib

一个由Peter Todd提供的Python比特币库、共识库和节点

pycoin

由Richard Kiss提供的Python比特币库

Go

btcd

Go语言全节点比特币客户端

Rust

rust-bitcoin

Rust比特币库，用于序列化、解析和API调用

Scala

bitcoin-s

一个用Scala实现的比特币实现

C\

NBitcoin

.NET框架的综合比特币库

总结

\ 许多其他编程语言中还存在许多其他库，而且会有更多库不断被创建。

如果你按照本章的说明进行操作，现在你已经运行了比特币核心，并且已经开始使用你自己的全节点探索网络和区块链。从现在开始，你可以独立使用你控制的软件——在你控制的计算机上——验证你收到的任何比特币是否遵循比特币系统中的每条规则，而无需信任任何外部权威机构。在接下来的章节中，我们将更多地了解系统的规则，以及你的节点和钱包如何利用它们来保护你的资金安全，保护你的隐私，并使支出和接收更加方便。

简单的应用开发：使用bitcoinj客户端监听转账，和发起转账

本章介绍如何使用bitcoinj客户端监听转账，和发起转账。demo工程：https://github.com/berryjam/bitcoinj 是基于bitcoinj的基础上，作了以下小改动：

- 修改了build.gradle，添加了google()仓库，避免编译过程中出现找不到guava包的问题，改动如图3-2所示。

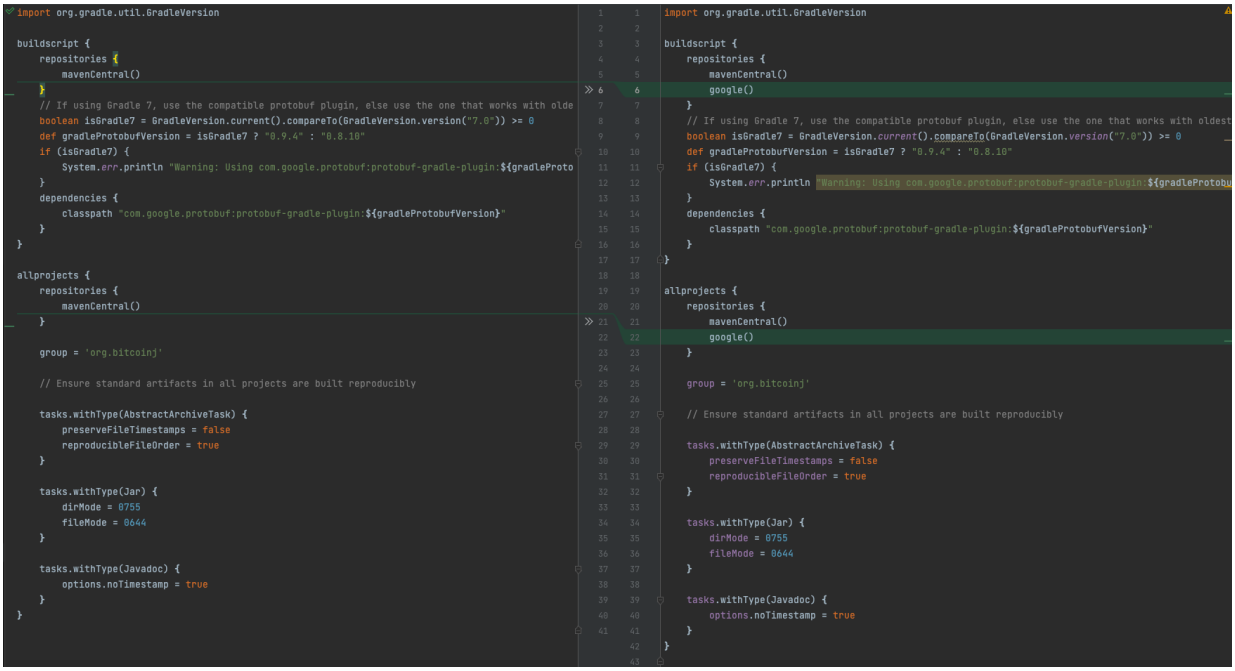


图 3-2. 右边绿色区域为改动内容

- 修改了examples/build.gradle，添加了application插件，使得gradle可以运行指定例子，如通过mainClass指定要运行类，gradle -PmainClass=org.bitcoinj.examples.FetchBlock run --args=000000000000001535467706e8545b339e7e0adca408c5432781f4da0e94d734，改动如图3-3所示。

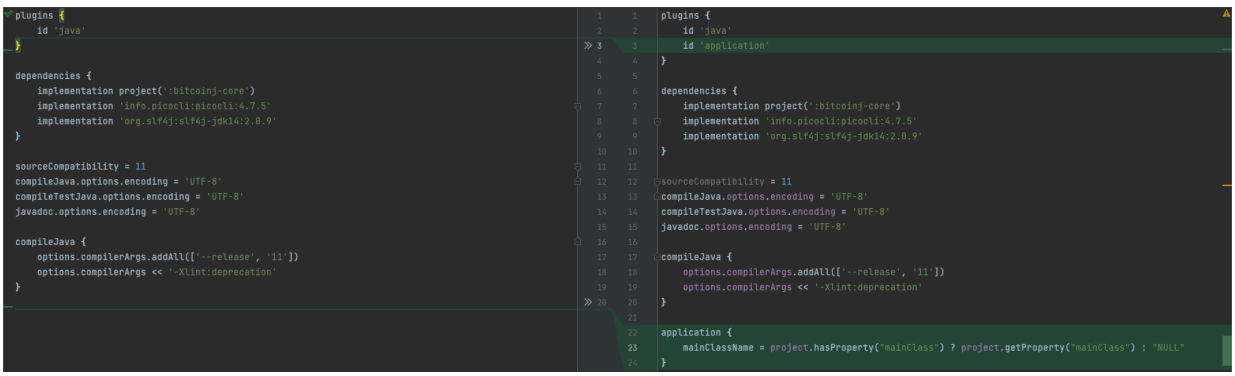


图 3-3. 右边区域为改动内容

- 修改了examples/src/main/java/org.bitcoinj.examples/Kit.java，让Kit后台运行并监听转账后再给个人UniSat钱包地址转账。
- 修改了examples/src/main/java/org.bitcoinj.examples/FetchBlock.java，检索指定块里给个人地址转账交易信息。

准备工作

以下两个例子都是在测试网(testnet)进行，请先安装BTC钱包（如UniSat的chrome插件）和领取测试币，https://altquick.com/swap/ 也可以换取测试币。

> 关于testnet、signet、regtest区别请参考《第十一章：比特币的测试区块链》一节。

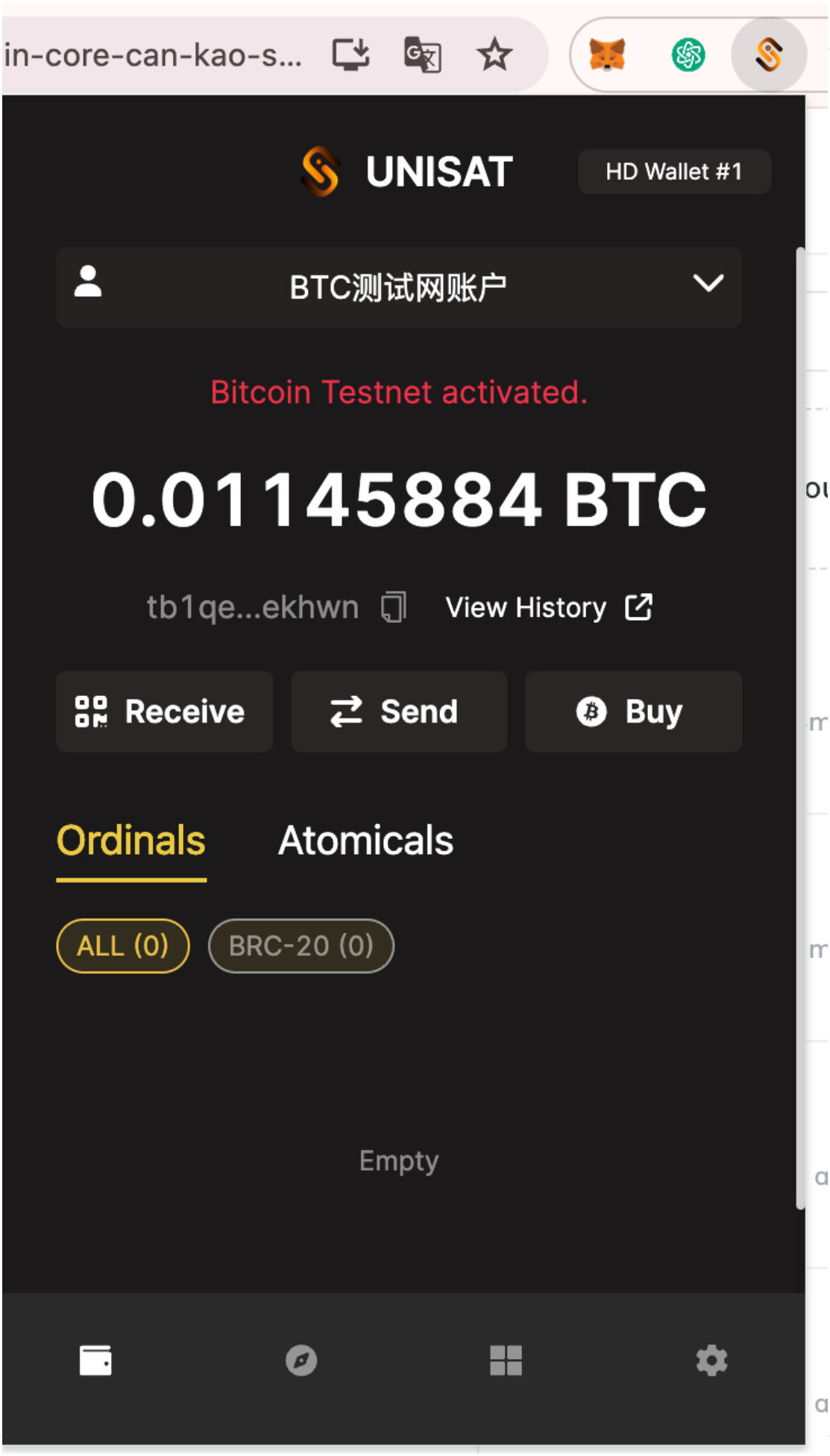

in-core-can-kao-s...
UNISAT
HD Wallet #1
BTC测试网账户
Bitcoin Testnet activated.
0.01145884 BTC
tb1qe...ekhwn
View History
Receive
Send
Buy
Ordinals
Atomicals
ALL (0)
BRC-20 (0)
Empty

图 3-4. UniSat钱包，领取测试币

拉取代码：

```
$git clone git@github.com:berryjam/bitcoinj.git
$cd bitcoinj/examples
```

本地生成钱包和监听转账：

```
$gradle -PmainClass=org.bitcoinj.examples.Kit run
```

此刻如果正常运行，可以看到一些输出信息，并且examples目录下会生成两个文件，分别是：测试链交易信息walletappkit-example.spvchain，及个人钱包walletappkit-example.wallet。

查看本地账户地址：

由于Kit在后台运行，会锁住钱包文件，需要先关掉。使用WalletTool的dump命令可以查看钱包地址，其中参数--chain和--wallet指定上面两个文件路径，--net选择testnet。

```
$cd wallettool
$gradle -PmainClass=org.bitcoinj.wallettool.WalletTool run --args='dump --chain ${本地目录}/bitcoinj/examples/wal
```

正常运行，会看到以下信息，包含钱包地址：

```
Wallet
Balances:
  0.00 BTC ESTIMATED
  0.00 BTC AVAILABLE
  0.00 BTC ESTIMATED_SPENDABLE
  0.00 BTC AVAILABLE_SPENDABLE
Transactions:
  0 pending
  0 unspent
  0 spent
  0 dead
Last seen best block: 2582064 (2024-03-15T12:42:50Z): 00000000000000007836260bd967b437a6a4d993fecafb22f2ab53a5da

Keys:
Earliest creation time: 2024-03-15T12:25:55Z
Seed birthday:     1710505555  [2024-03-15T12:25:55Z]
Ouput script type: P2PKH
Key to watch:      tpubD9XNB5hPxkwtc26UYmLDPLYGZZR1GLQ9KGYPeiWZZbqpWMcQbhc7gogbeSPvsQ6yybiYm9HraXLzYWWKnvASbgVB
Lookahead siz/thr: 100/33
  addr:mwBKsLnLBN2SLsca6gUEyBUa62aE1mDzqV  hash160:abcbb3cacded05d9e55b9b2b63781bb1cf799061  (m, root)
  addr:mnMRvhAaK51FrCFSSnXvmmvwB7qNRFZkDd  hash160:4afbc9628f5cb039d7de8468e99cd90fb02dd6e2  (m/0H, account)
  addr:mzCSon2Bcp2k7fDxEK1eSqTT2SgVMZCdwP  hash160:ccea49a1d36d9c974e674b3e5b1231dae850d116  (m/0H/0, external)
  addr:n4CZ2wLTirF1TLRgdJcL1VEBPcFetaxQ6a  hash160:f8cffd1ccec1f6119122d0ec0b4b9a69f9cc11bd  (m/0H/1, internal)

Seed birthday:     1710505555  [2024-03-15T12:25:55Z]
Ouput script type: P2WPKH
Key to watch:      vpub5VVD3WXsx7uUffZQvJfyq2hQaTvK4iVedZGsBvtXDa9zkZjMwJAKMyVH6xKMApnLchJGMpuMvEcJSmTjYLeqtZ7E
Lookahead siz/thr: 100/33
  addr:tb1q409m8jkda5zane2mnv4kx7qmk88hnyrpemvls6  hash160:abcbb3cacded05d9e55b9b2b63781bb1cf799061  (m, root)
  addr:tb1q7cp7xg0wdm9r46uwckagj8al3p9p6gcggmvyec  hash160:f603e321ee6eca3aeb8ec5ba891fbf884a1d2308  (m/1H, acc
  addr:tb1q30nky20w8gae6ednt2tkt48zakytw6kgckkhr3  hash160:8be76229ee3a3b9d65b35a9765d4e2ed88b76ac8  (m/1H/0, e
  addr:tb1q6k35lxctlknaesw4ckxnngxl7q3f7awhn3de82  hash160:d5a34f9b0bfda7dcc1d5c58d39a0dff0229f75d7  (m/1H/1, i
  addr:tb1qh8h3zgr9g9zgqhed7e9ldp943xqz5vgn62pxnj  hash160:b9ef1120654144805f2df64bf684b589802a3113  (M/1H/0/0)
```

其中一个P2WPKH类型账户：tb1q7cp7xg0wdm9r46uwckagj8al3p9p6gcggmvyec。

> 钱包和地址类型，请参考《第五章：钱包技术栈的详细介绍》一节。

转账

重新让Kit后台运行，监听转账信息，并且余额足够的情况下给UniSat钱包转出0.00001 BTC。

使用UniSat钱包给上面bitcoinj钱包地址转账：

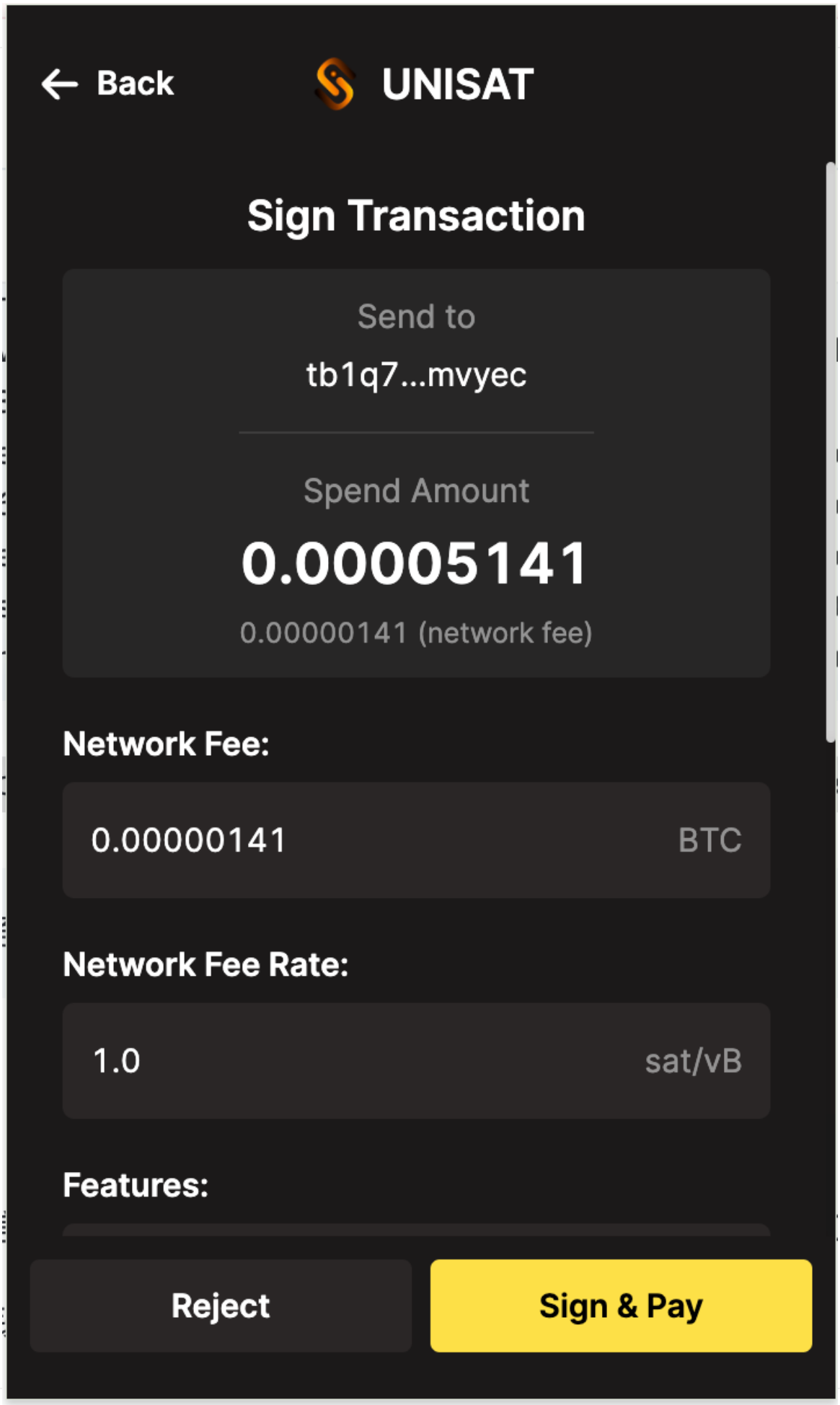

Back
UNISAT
Sign Transaction
Send to
tb1q7...mvyec
Spend Amount
0.00005141
0.00000141 (network fee)
Network Fee:
0.00000141
BTC
Network Fee Rate:
1.0
sat/vB
Features:
Reject
Sign & Pay

图 3-5. 给bitcoinj钱包转账

正常情况下，可以看到Kit后台输出以下打印信息：

```
-----> coins resceived: df2b71b76cfc6ae8f2142204a2460b348d438f5f2a05f61b57b3239ee20cc5b4
3月 15, 2024 9:03:26 下午 org.bitcoinj.wallet.WalletFiles saveNow
信息: Saving wallet; last seen block is height 2582067, date 2024-03-15T12:54:20Z, hash 000000000000000e5646ea50
3月 15, 2024 9:03:26 下午 org.bitcoinj.wallet.WalletFiles saveNowInternal
信息: Save completed in 22 ms
3月 15, 2024 9:03:26 下午 org.bitcoinj.wallet.DeterministicKeyChain maybeLookAhead
信息: 0 keys needed for M/0H/0 = 0 issued + 100 lookahead size + 33 lookahead threshold - 133 num children
3月 15, 2024 9:03:26 下午 org.bitcoinj.wallet.DeterministicKeyChain maybeLookAhead
```

其中df2b71b76cfc6ae8f2142204a2460b348d438f5f2a05f61b57b3239ee20cc5b4为交易hash，在区块链浏览器上可以看到：https://mempool.space/zh/testnet/tx/df2b71b76cfc6ae8f2142204a2460b348d438f5f2a05f61b57b3239ee20cc5b4 具体内容。

在代码里，第一次接收到0.00005 BTC后，会给UniSat转账，却发现钱不够。

```
kit.wallet().addCoinsReceivedEventListener((wallet, tx, prevBalance, newBalance) -> {
            System.out.println("-----> coins resceived: " + tx.getTxId());
            System.out.println("received: " + tx.getValue(wallet));
            Address to = kit.wallet().parseAddress("tb1qeww9d68r9xyka203zpzmwmwc94rp8sqfgekhwn");
            Coin value = Coin.parseCoin("0.00001");
            org.bitcoinj.wallet.SendRequest req = org.bitcoinj.wallet.SendRequest.to(to, value);
            req.setFeePerVkb(Coin.valueOf(1100));
            try {
                Wallet.SendResult result = wallet.sendCoins(req);
                System.out.println("coins sent. transaction hash: " + result.transaction().getTxId());
            } catch (InsufficientMoneyException e) {
                System.out.println("Not enough coins in your wallet. Missing " + e.missing.getValue() + " satos
                System.out.println("Please send enough money to: " + wallet.currentReceiveAddress().toString())
            }
        });
```

```
Not enough coins in your wallet. Missing 1000 satoshis are missing (including fees)
Please send enough money to: tb1qh8h3zgr9g9zgqhed7e9ldp943xqz5vgn62pxnj
-----> confidence changed: df2b71b76cfc6ae8f2142204a2460b348d438f5f2a05f61b57b3239ee20cc5b4
new block depth: 0
-----> confidence changed: df2b71b76cfc6ae8f2142204a2460b348d438f5f2a05f61b57b3239ee20cc5b4
```

深入转账代码，可以发现钱包会从账户的UTXO列表里选择最佳可用的输出（第二章的币种选择一节有过相关介绍），从里面按照最大可用金额排序，然后累加直到满足转账金额。关键是shouldSelect函数，里面会判断交易状态。因为刚接收到的交易，还没得到确认，因此无法用于转账。

```
Wallet.SendResult result = wallet.sendCoins(req);
...
public CoinSelection select(Coin target, List<TransactionOutput> candidates) {
        ArrayList<TransactionOutput> selected = new ArrayList<>();
        // Sort the inputs by age*value so we get the highest "coindays" spent.
        // TODO: Consider changing the wallets internal format to track just outputs and keep them ordered.
        ArrayList<TransactionOutput> sortedOutputs = new ArrayList<>(candidates);
        // When calculating the wallet balance, we may be asked to select all possible coins, if so, avoid sort
        // them in order to improve performance.
        if (!target.equals(BitcoinNetwork.MAX_MONEY)) {
            sortedOutputs.sort(DefaultCoinSelector::compareByDepth);
        }
        // Now iterate over the sorted outputs until we have got as close to the target as possible or a little
        // bit over (excessive value will be change).
        long total = 0;
        for (TransactionOutput output : sortedOutputs) {
            if (total >= target.value) break;
            // Only pick chain-included transactions, or transactions that are ours and pending.
            if (!shouldSelect(output.getParentTransaction())) continue;
            selected.add(output);
            total = Math.addExact(total, output.getValue().value);
        }
        // Total may be lower than target here, if the given candidates were insufficient to create to requeste
        // transaction.
        return new CoinSelection(selected);
    }
```

等待一段时间交易得到确认后，重新转账0.00001 BTC，再次触发转账交易，可以看到输出：

```
coins sent. transaction hash: 910e4fa1e07a192a33986dc45c6857ff33391cabe0ff87e733fda7ea0f71379e
-----> confidence changed: 4618ab0373121312f95bab45ae408e8c5d87acd63262adb17f7af93609c9fb0c
new block depth: 0
-----> coins resceived: 910e4fa1e07a192a33986dc45c6857ff33391cabe0ff87e733fda7ea0f71379e
```

可以在区块链浏览器上看到交易信息：
https://mempool.space/zh/testnet/tx/910e4fa1e07a192a33986dc45c6857ff33391cabe0ff87e733fda7ea0f71379e。

拉块

最后是一个简单的监控转账的小功能，比如我们需求是需要监听链上所有交易信息，然后对感兴趣的地址转账再做一些后处理。

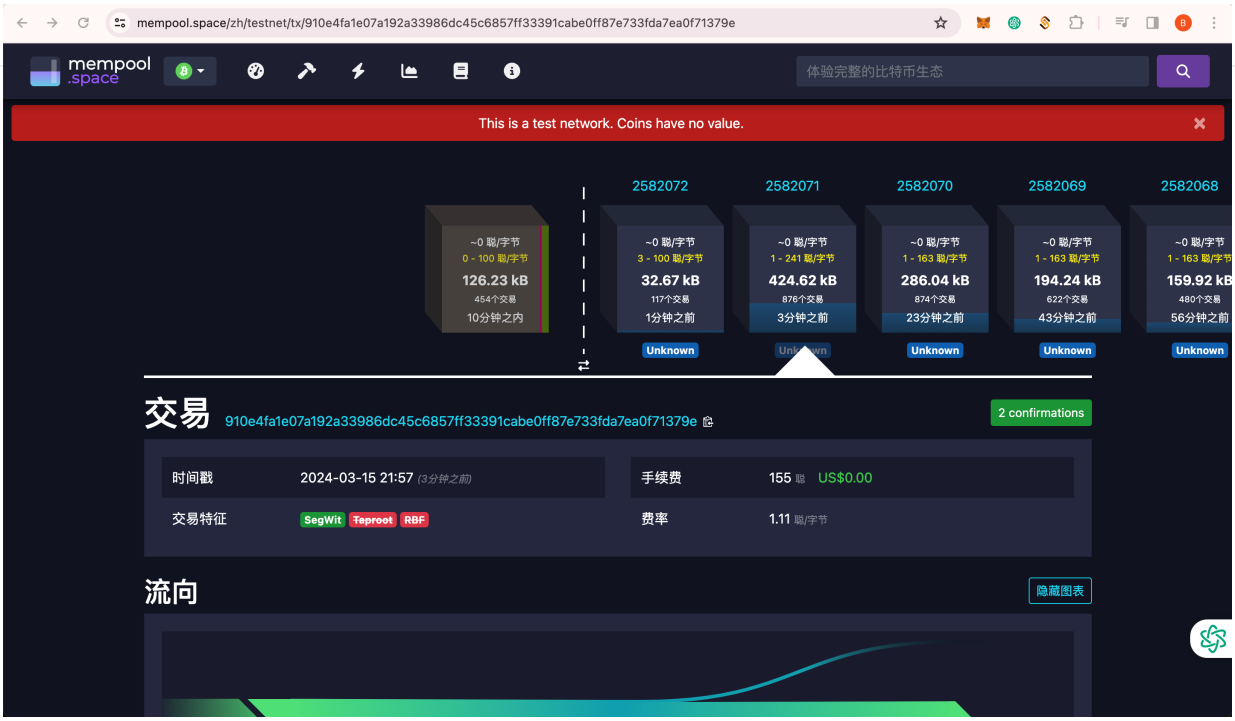


图 3-6. 交易所在块信息

1.获取对应的交易信息

遍历块的所有交易信息，通过交易hash筛选出转账交易。

```
        for (Transaction tx : txs) {
            if ("910e4fa1e07a192a33986dc45c6857ff33391cabe0ff87e733fda7ea0f71379e".equals(tx.getTxId().toString
                System.out.println("tx: " + tx);
                ...
            }
        }
```

2.获取输入地址

观察交易的输入信息，可以看到witness和地址类型信息(这里为P2WPKH)，但是从witness怎么获取到发送方地址呢?

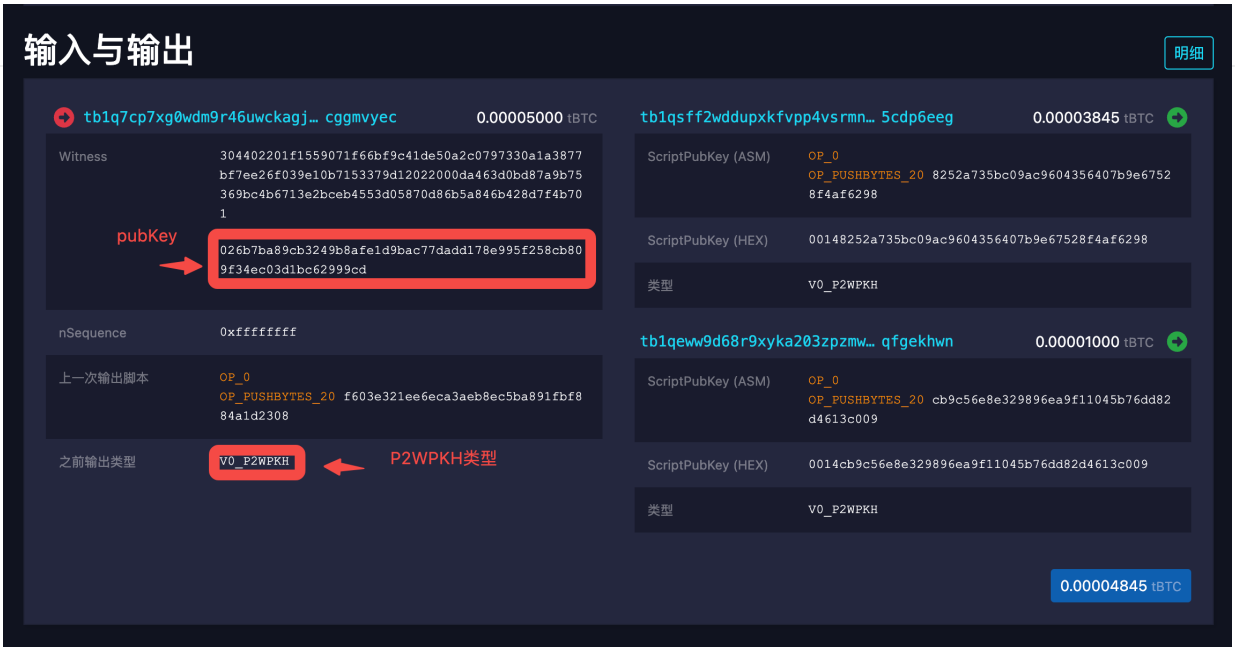


图 3-7. 交易输入信息

我们可以通过Transaction.java的函数了解到input的witness是怎么构造出来的。

```
public TransactionInput addSignedInput(TransactionOutPoint prevOut, Script scriptPubKey, Coin amount, ECKey sig
                                       SigHash sigHash, boolean anyoneCanPay) throws ScriptException {
        ...
        if (ScriptPattern.isP2PK(scriptPubKey)) {
            ...
        } else if (ScriptPattern.isP2PKH(scriptPubKey)) {
            ...
        } else if (ScriptPattern.isP2WPKH(scriptPubKey)) { // 示例为P2WPKH
            Script scriptCode = ScriptBuilder.createP2PKHOutputScript(sigKey);
            TransactionSignature signature = calculateWitnessSignature(inputIndex, sigKey, scriptCode, input.ge
                    sigHash, anyoneCanPay);
            input.setScriptSig(ScriptBuilder.createEmpty());
            input.setWitness(TransactionWitness.redeemP2WPKH(signature, sigKey));
        } else {
            throw new ScriptException(ScriptError.SCRIPT_ERR_UNKNOWN_ERROR, "Don't know how to sign for this ki
        }
        return input;
    }
```

可以看到通过TransactionWitness.redeemP2WPKH(signature, sigKey)构造出来witness，其中包含了sigKey，继续深入redeemP2WPKH函数就可以知道这个参数就是发送方的PubKey，并且放在witness的第二个位置（下标为0）。

```
    public static TransactionWitness redeemP2WPKH(@Nullable TransactionSignature signature, ECKey pubKey) {
        checkArgument(pubKey.isCompressed(), () ->
                "only compressed keys allowed");
        List<byte[]> pushes = new ArrayList<>(2);
        pushes.add(signature != null ? signature.encodeToBitcoin() : new byte[0]); // signature
        pushes.add(pubKey.getPubKey()); // pubkey , witness的第2个位置
        return TransactionWitness.of(pushes);
    }
```

有了公钥，就可以根据账户和网络类型算出对应的地址。

```
byte[] publicKeyBytes = input.getWitness().getPush(1); // 第二个位置，下标为1，存放的的是pubKey
// 创建一个 ECKey 对象并导入公钥
ECKey ecKey = ECKey.fromPublicOnly(publicKeyBytes);

// 将 ECKey 对象转换为 Bitcoin 地址
Address from = ecKey.toAddress(ScriptType.P2WPKH, network); // 如果是测试网络，请改为 TestNetParams.get()
```

3.获取输出地址和金额

遍历所有输出，通过转出地址筛选出我们所关心的输出，类似输入，地址可以从输出脚本解析得到，输出金额直接从output获取到。

```
List<TransactionOutput> outputs = tx.getOutputs();
for (TransactionOutput output : outputs) {
     Script pubKey = output.getScriptPubKey();
     Address to = pubKey.getToAddress(network);
     if ("tb1qeww9d68r9xyka203zpzmwmwc94rp8sqfgekhwn".equals(to.toString())) {
          System.out.printf("from:%s to:%s receive: %s\n", from, to, output.getValue().toFriendlyString());
     }
}
```

4.综合运行

将上面逻辑整合起来，运行examples/src/main/java/org.bitcoinj.examples/FetchBlock.java

```
gradle -PmainClass=org.bitcoinj.examples.FetchBlock run --args=000000009661aeca0d6092068575fe5113455fe2d28d200d
```

输出如下：

```
16:29:40.246 28 PeerGroup.handleNewPeer: Peer{[18.162.45.10]:18333, version=70016, subVer=/Satoshi:0.21.0/, ser
tx: Transaction{910e4fa1e07a192a33986dc45c6857ff33391cabe0ff87e733fda7ea0f71379e, wtxid d69b107bdc9079022593bd9
weight: 561 wu, 141 virtual bytes, 222 bytes
purpose: UNKNOWN
   in   <empty>
        witness:304402201f1559071f66bf9c41de50a2c0797330a1a3877bf7ee26f039e10b7153379d12022000da463d0bd87a9b753
        unconnected  outpoint:df2b71b76cfc6ae8f2142204a2460b348d438f5f2a05f61b57b3239ee20cc5b4:0
   out  0[] PUSHDATA(20)[8252a735bc09ac9604356407b9e67528f4af6298]  0.00003845 BTC
        P2WPKH
   out  0[] PUSHDATA(20)[cb9c56e8e329896ea9f11045b76dd82d4613c009]  0.00001 BTC
        P2WPKH
}
from:tb1q7cp7xg0wdm9r46uwckagj8al3p9p6gcggmvyec to:tb1qeww9d68r9xyka203zpzmwmwc94rp8sqfgekhwn receive: 0.00001
```

可以看到接收来自tb1q7cp7xg0wdm9r46uwckagj8al3p9p6gcggmvyec，发送到tb1qeww9d68r9xyka203zpzmwmwc94rp8sqfgekhwn，一共0.00001 BTC的转账信息。

> 关于交易的结构信息，请参考《第六章：交易》。

使用单独部署的测试全节点

> 由于上述例子使用的测试网公开的节点进行同步区块信息，这些公开节点可能会请求进行限频等，导致交易监听和转账出现问题。因此建议在本地部署一个测试网全节点，专用的节点获取交易信息和处理效率都很会非常高。

1.下载bitcoin core

根据运行操作系统选择合适的版本：https://bitcoincore.org/en/download/

```
$cd ~/download
$wget https://bitcoincore.org/bin/bitcoin-core-26.0/bitcoin-26.0-x86_64-linux-gnu.tar.gz
```

2.解压和设置

```
$tar -xvf bitcoin-26.0-x86_64-linux-gnu.tar.gz
$export PATH=$PATH:/root/download/bitcoin-26.0/bin
```

正常情况下，能够识别节点bitcoind和命令行工具bitcoin-cli

```
$which bitcoind
/root/download/bitcoin-26.0/bin/bitcoind
$which bitcoin-cli
/root/download/bitcoin-26.0/bin/bitcoin-cli
```

3.配置运行参数

接着就可以配置运行参数文件

```
$mkdir -p /data/bitcoin
$touch /data/bitcoin/bitcoin.conf
```

大致参数如下，由于用的测试网，所以只配置了[test]。其中datadir为存储区块交易的数据目录，`txindex` 是一个用于控制是否启用事务索引的参数。如果设置为 `1`，则表示启用事务索引，允许通过事务哈希来快速查找交易的详细信息，建议开启，不过会更占用磁盘空间。peerbloomfilters=1，表示允许开启布隆过滤器，**bitcoinj项目这个参数必须设置为1，否则会拉取不到数据**。

```
# Options for mainnet
[main]

# Options for testnet
[test]
datadir=/data/bitcoin/testnet/data
#prune=51200
shrinkdebuglog=1
rpcuser=berry
rpcpassword=123456
txindex=1
peerbloomfilters=1

# Options for signet
[signet]

# Options for regtest
[regtest]
```

4.运行bitcoind和确认

指定运行配置文件、网络和后台运行

```
$bitcoind --conf=/data/bitcoin/bitcoin.conf -testnet -daemon
```

第一次启动时，节点需要同步最新的区块数据，需要等待一会。正常情况下，通过bitcoin-cli就能查询到上述的交易信息

```
$bitcoin-cli -testnet -rpcuser=berry -rpcpassword=123456 getrawtransaction 910e4fa1e07a192a33986dc45c6857ff3339
01000000000101b4c50ce29e23b3571bf6052a5f8f438d340b46a2042214f2e86afc6cb7712bdf0000000000ffffffff02050f000000000
```

5.代码设置Peer为本地节点

测试节点的P2P默认端口为18333，RPC端口为18332，正式网分别为8333、8332。

```
WalletAppKit kit = WalletAppKit.launch(network, new File("."), "walletappkit-example", (k) -> {
            // In case you want to connect with your local bitcoind tell the kit to connect to localhost.
            // This is done automatically in reg test mode.
            // k.connectToLocalHost();
            try {
                k.setPeerNodes(PeerAddress.simple(InetAddress.getByName("127.0.0.1"), 18333)); // 设置本地连接Pee
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        });
```

正常情况下，使用本地全节点能避免很多问题。

bitcoinj常见问题

1.钱包加载异常

```
Exception in thread "main" java.lang.IllegalStateException: Expected the service WalletAppKit [FAILED] to be RU
    at com.google.common.util.concurrent.AbstractService.checkCurrentState(AbstractService.java:384)
    at com.google.common.util.concurrent.AbstractService.awaitRunning(AbstractService.java:308)
    at com.google.common.util.concurrent.AbstractIdleService.awaitRunning(AbstractIdleService.java:160)
    at org.bitcoinj.kits.WalletAppKit.launch(WalletAppKit.java:220)
    at org.bitcoinj.kits.WalletAppKit.launch(WalletAppKit.java:178)
    at org.bitcoinj.examples.Kit.main(Kit.java:57)
Caused by: org.bitcoinj.wallet.UnreadableWalletException: Could not parse input stream to protobuf
    at org.bitcoinj.wallet.WalletProtobufSerializer.readWallet(WalletProtobufSerializer.java:455)
    at org.bitcoinj.wallet.Wallet.loadFromFileStream(Wallet.java:2076)
    at org.bitcoinj.wallet.Wallet.loadFromFile(Wallet.java:1961)
    at org.bitcoinj.kits.WalletAppKit.loadWallet(WalletAppKit.java:508)
    at org.bitcoinj.kits.WalletAppKit.createOrLoadWallet(WalletAppKit.java:480)
    at org.bitcoinj.kits.WalletAppKit.startUp(WalletAppKit.java:399)
    at com.google.common.util.concurrent.AbstractIdleService$DelegateService.lambda$doStart$0(AbstractIdleServi
    at com.google.common.util.concurrent.Callables.lambda$threadRenaming$3(Callables.java:105)
    at java.base/java.lang.Thread.run(Thread.java:840)
Caused by: java.lang.IllegalStateException: You must construct a Context object before using bitcoinj!
    at org.bitcoinj.core.Context.get(Context.java:124)
    at org.bitcoinj.core.Transaction.getConfidence(Transaction.java:1586)
    at org.bitcoinj.wallet.WalletProtobufSerializer.connectTransactionOutputs(WalletProtobufSerializer.java:771
    at org.bitcoinj.wallet.WalletProtobufSerializer.readWallet(WalletProtobufSerializer.java:551)
    at org.bitcoinj.wallet.WalletProtobufSerializer.readWallet(WalletProtobufSerializer.java:453)
    ... 8 more
```

当出现以上异常时，需要在钱包加载(WalletAppKit.launch)前，设置:Context.propagate(new Context()); 如下所示：

```
        BitcoinNetwork network = BitcoinNetwork.TESTNET;
        Context.propagate(new Context()); // 这里必须要设置，否则会出现加载钱包错误

        // Initialize and start a WalletAppKit. The kit handles all the boilerplate for us and is the easiest wa
        // Look at the WalletAppKit documentation and its source to understand what's happening behind the scen
        // WalletAppKit extends the Guava AbstractIdleService. Have a look at the introduction to Guava service
        WalletAppKit kit = WalletAppKit.launch(network, new File("."), "walletappkit-example", (k) -> {
            // In case you want to connect with your local bitcoind tell the kit to connect to localhost.
            // This is done automatically in reg test mode.
            // k.connectToLocalHost();
            try {
                k.setPeerNodes(PeerAddress.simple(InetAddress.getByName("127.0.0.1"), 18333));
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        });
```

2.节点连接异常

```
WARN  [NioClientManager] [o.b.n.ConnectionHandler](ConnectionHandler.java:251) Error handling SelectionKey: java
java.nio.channels.CancelledKeyException: null
    at sun.nio.ch.SelectionKeyImpl.ensureValid(SelectionKeyImpl.java:73)
    at sun.nio.ch.SelectionKeyImpl.readyOps(SelectionKeyImpl.java:87)
    at java.nio.channels.SelectionKey.isWritable(SelectionKey.java:312)
    at org.bitcoinj.net.ConnectionHandler.handleKey(ConnectionHandler.java:245)
    at org.bitcoinj.net.NioClientManager.handleKey(NioClientManager.java:97)
    at org.bitcoinj.net.NioClientManager.run(NioClientManager.java:133)
    at com.google.common.util.concurrent.AbstractExecutionThreadService$1.lambda$doStart$1(AbstractExecutionThr
    at com.google.common.util.concurrent.Callables.lambda$threadRenaming$3(Callables.java:105)
    at org.bitcoinj.utils.ContextPropagatingThreadFactory.lambda$newThread$0(ContextPropagatingThreadFactory.ja
    at java.lang.Thread.run(Thread.java:750)
2024-03-21 09:56:53.609 INFO  [PeerGroup Thread] [o.b.c.PeerGroup](PeerGroup.java:639) Waiting 1500 ms before n
```

这种情况，一般是因为默认选用了测试网的公开节点，请求被限频。可以参考上面部署单个节点来解决，但必须设置***peerbloomfilters=1，_bitcoinj只能连接支持bloom filter的节点。_***

综合介绍

\ Alice想向Bob支付比特币，但成千上万的比特币全节点将验证她的交易，这些节点不知道Alice或Bob的真实身份，我们希望保持这种状态以保护他们的隐私。Alice需要传达的是，Bob应该收到她的一些比特币，但不应将该交易的任何方面与Bob的真实世界身份或Bob收到的其他比特币支付联系起来。Alice使用的方法必须确保只有Bob能进一步花费他收到的比特币。

最初的比特币论文描述了一种实现这些目标的非常简单的方案，如图4-1所示。

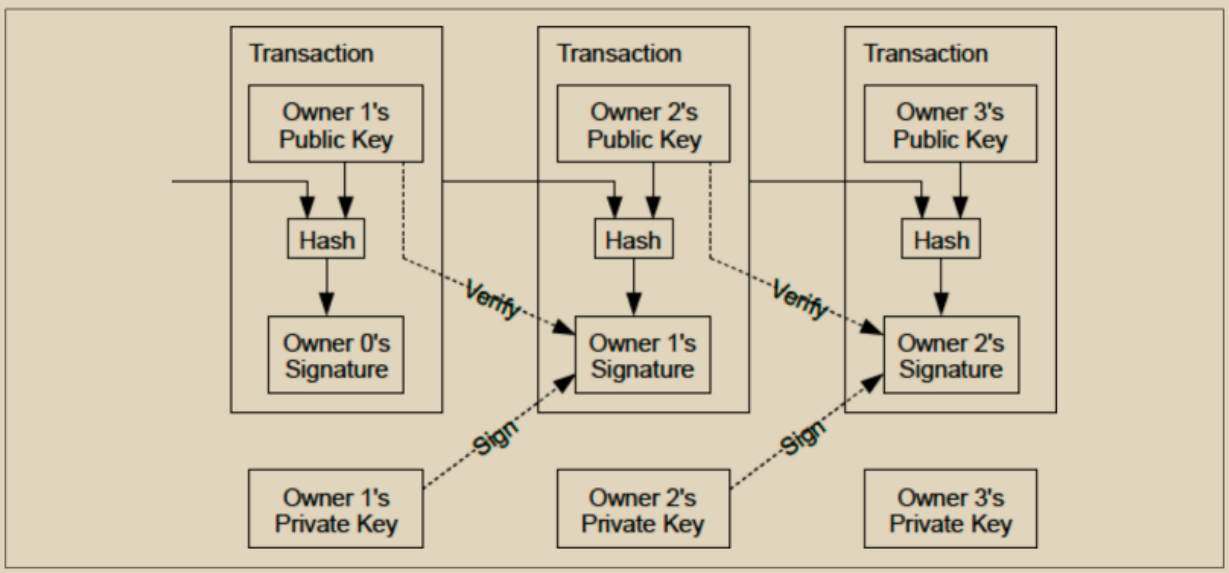


图 4-1. 原始比特币论文中的交易链。

像Bob这样的接收者接受由花费者（如Alice）签名的交易中的比特币到一个公钥。Alice要花费的比特币之前是通过她的一个公钥接收到的，她使用相应的私钥生成她的签名。全节点可以验证Alice的签名是否承诺给出一个哈希函数的输出，该哈希函数本身承诺给出Bob的公钥和其他交易细节。

我们将在本章中讨论公钥、私钥、签名和哈希函数，然后将它们全部结合起来描述现代比特币软件使用的地址。

公钥密码学

公钥密码学是在20世纪70年代发明的，是现代计算机和信息安全的数学基础。

自从公钥密码学的发明以来，已经发现了几种合适的数学函数，比如素数指数和椭圆曲线乘法。这些数学函数在一个方向上易于计算，但在当前计算机和算法下，几乎不可能在反向上进行计算。基于这些数学函数，密码学实现了不可伪造的数字签名。比特币使用椭圆曲线加法和乘法作为其密码学的基础。

在比特币中，我们可以使用公钥密码学来创建一个控制比特币访问权限的密钥对。密钥对由私钥和从私钥派生的公钥组成。公钥用于接收资金，私钥用于对要花费的资金签名的交易。

公钥和私钥之间存在数学关系，使得私钥可以用来对消息生成签名。这些签名可以根据公钥进行验证，而无需揭示私钥。

> 在一些钱包实现中，私钥和公钥一起存储作为一个密钥对是为了方便起见。然而，公钥可以从私钥计算出来，因此仅存储私钥也是可能的。

一个比特币钱包包含了一组密钥对，每个密钥对由一个私钥和一个公钥组成。私钥（k）是一个数字，通常是从随机选择的数字派生而来。通过私钥，我们使用椭圆曲线乘法，这是一个单向加密函数，来生成一个公钥（K），公钥是椭圆曲线上一个二维坐标点。

> **为什么比特币中使用非对称加密（公钥/私钥）？**
>
> 它并不是用于“加密”（保密）交易的。相反，非对称加密的一个有用特性是能够生成数字签名。私钥可以应用到交易中产生一个数字签名。这个签名只能由拥有私钥知识的人产生。然而，任何有公钥和交易访问权限的人都可以使用它们来验证签名。非对称加密的这一有用特性使得任何人都能够验证每一笔交易上的每个签名，同时确保只有私钥的所有者可以生成有效的签名。

私钥

私钥只是一个随机选择的数字。对私钥的控制是用户对与相应的比特币公钥关联的所有资金的控制的根源。私钥用于创建签名，用于在交易中证明对用于支付比特币的资金的控制权。私钥必须始终保密，因为将其透露给第三方等同于将对其保护的比特币的控制权交给他们。私钥还必须备份和保护免受意外丢失，因为如果丢失了，就无法恢复，由其保护的资金也将永远丢失。

比特币私钥只是一个数字。你可以随机选择私钥，只需使用硬币、铅笔和纸：将一枚硬币抛掷 256 次，你就得到了一个随机私钥的二进制数字，可以用于比特币钱包。然后，公钥可以从私钥生成。但要注意，任何不完全随机的过程都可能显著降低私钥及其控制的比特币的安全性。

生成密钥的第一步，也是最重要的一步，是找到一个安全的随机源（计算机科学家称之为熵）。创建比特币密钥几乎与“在 1 到 2^256 之间选择一个数字”相同。只要选择的方法不可预测或不可重复，使用的确切方法就无关紧要。比特币软件使用加密安全的随机数生成器来产生 256 位的熵。

\ 更准确地说，私钥可以是从 0 到 n - 1（包括 n）之间的任何数字，其中 n 是一个常数（n = 1.1578 × 10^77，略小于 2^256），定义为比特币中使用的椭圆曲线的阶（请参阅“椭圆曲线密码学解释”）。为了创建这样的密钥，我们随机选择一个 256 位的数字，并检查它是否小于 n。在编程术语中，通常通过将大量随机位串，从一个加密安全的随机源收集而来，输入到 SHA256 哈希算法中来实现，这将方便地产生一个可以解释为数字的 256 位值。如果结果小于 n，我们就有了一个合适的私钥。否则，我们只需使用另一个随机数再次尝试。

不要编写自己的代码来创建随机数，也不要使用编程语言提供的“简单”随机数生成器。使用具有足够熵源的加密安全伪随机数生成器（CSPRNG）的种子。研究你选择的随机数生成器库的文档，确保它是加密安全的。正确实现 CSPRNG 对密钥的安全性至关重要。

以下是以十六进制格式显示的随机生成的私钥（k）（256位显示为64个十六进制数字，每个4位）：

1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD

比特币的私钥空间大小（2^256）是一个难以想象的大数。它约为10^77，以十进制表示。作为比较，可见宇宙估计包含10^80个原子。

椭圆曲线密码学解释

椭圆曲线密码学（ECC）是一种基于离散对数问题的非对称或公钥密码学，其表达方式是通过椭圆曲线上的点的加法和乘法来实现的。

图4-2是椭圆曲线的一个示例，类似于比特币使用的曲线。

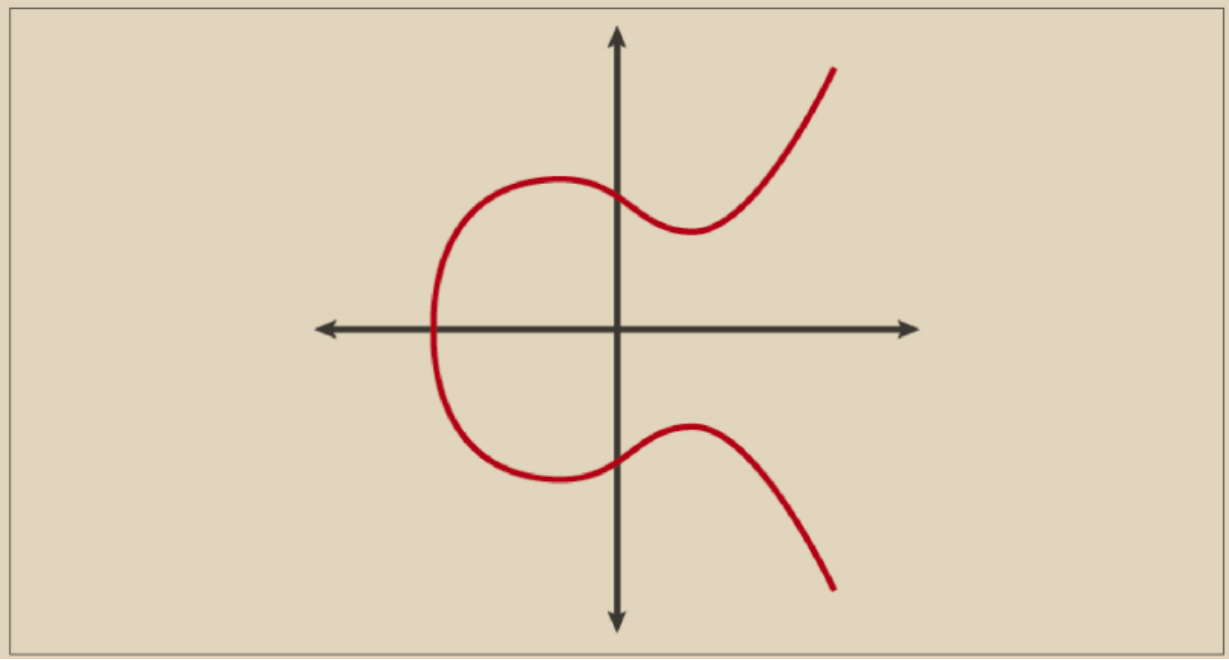


图 4-2. 一种椭圆曲线

比特币使用一种特定的椭圆曲线和一组数学常数，这在一种称为secp256k1的标准中定义，该标准由美国国家标准与技术研究院（NIST）确定。 secp256k1曲线由以下函数定义，该函数产生一个椭圆曲线：

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

$$

或者

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

$$

模 p（模质数 p）表示这个曲线是在一个素数 p 的有限域上，也写作 Fp，其中 p = 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1，是一个非常大的素数。因为这个曲线是在一个素数的有限域上定义的，而不是在实数上，所以它看起来像是在二维空间中散布的点的模式，这使得它难以可视化。然而，数学上与实数上的椭圆曲线完全相同。举个例子，图 4-3 展示了相同的椭圆曲线在一个比较小的素数阶有限域 17 上的情况，显示了一个点在网格上的模式。Bitcoin 的 secp256k1 椭圆曲线可以看作是一个非常复杂的点模式在一个超大网格上。

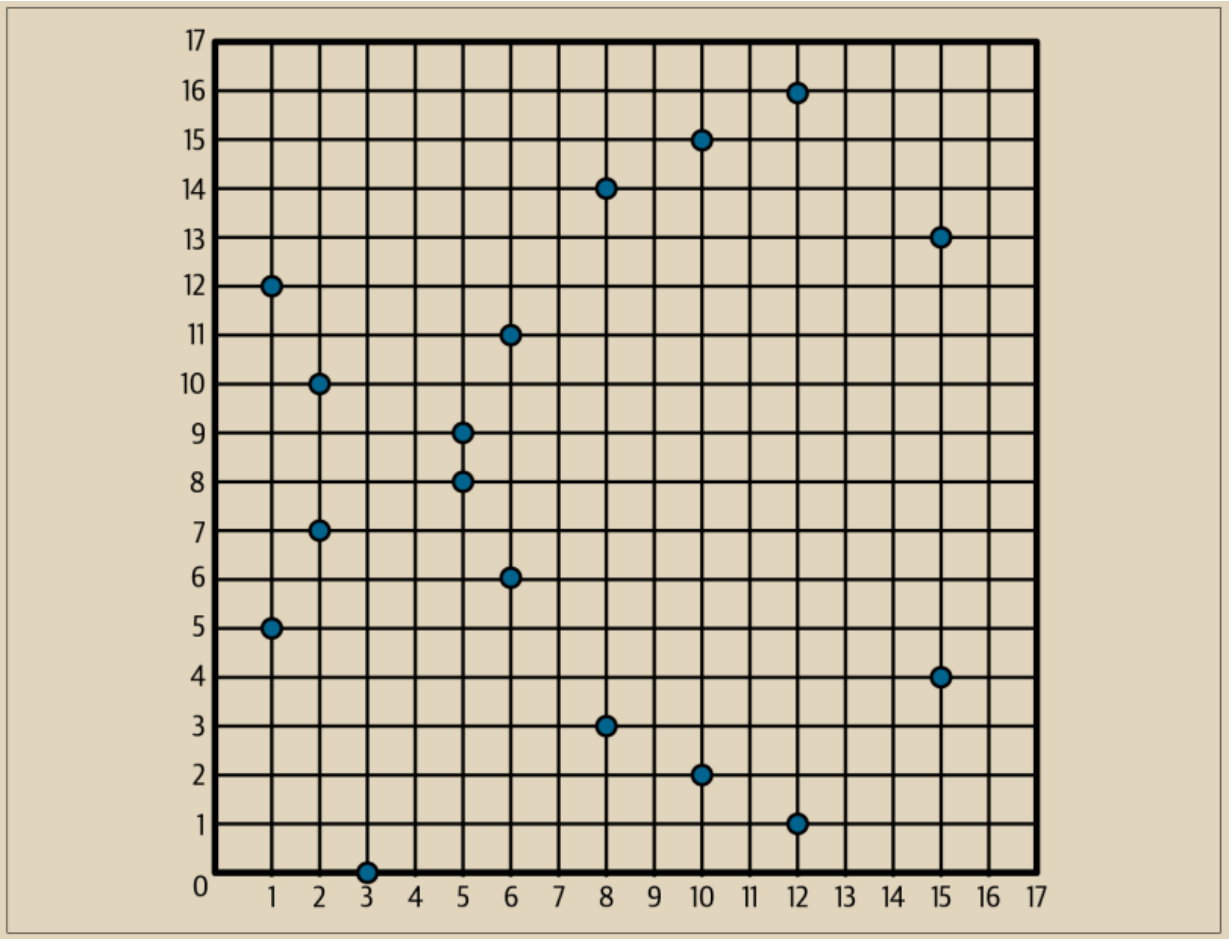


图 4-3. 椭圆曲线密码学：可视化椭圆曲线在 F(p) 上，其中 p=17。

因此，例如，以下是 secp256k1 曲线上的坐标为 (x, y) 的点 P：

\ P = (55066263022277343669578718895168534326250603453777594175500187360389116729240, 32670510020758816978083085130507043184471273380659243275938904335757337482424)

Example 4-1 展示了如何使用 Python 自行验证这一点。

Example 4-1. 使用 Python 确认该点位于椭圆曲线上

```
Python 3.10.6 (main, Nov 14 2022, 16:10:14) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
> p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
> x = 55066263022277343669578718895168534326250603453777594175500187360389116729240
> y = 32670510020758816978083085130507043184471273380659243275938904335757337482424
> (x ** 3 + 7 - y**2) % p
0
```

在椭圆曲线数学中，有一个称为“无穷点”的点，大致对应于加法中的零。在计算机中，它有时以 x = y = 0 表示（虽然不满足椭圆曲线方程，但这是一个容易单独检查的简单情况）。

还有一个称为“加法”的 + 运算符，其性质与小学生学习的实数加法类似。给定椭圆曲线上的两点 P1 和 P2，有第三点 P3 = P1 + P2，也在椭圆曲线上。

几何上，第三点 P3 是通过在 P1 和 P2 之间画一条线来计算的。这条线将在椭圆曲线上的一个额外位置相交。将这一点称为 P3' =（x，y）。然后在 x 轴上反射以得到 P3 =（x，-y）。

有几种特殊情况解释了“无穷点”的必要性。如果 P1 和 P2 是相同的点，则“介于” P1 和 P2 之间的线应延伸成为此点 P1 上的切线。这条切线将在曲线上的一个新点上相交。您可以使用微积分技术确定切线的斜率。尽管我们将兴趣限制在具有两个整数坐标的曲线上的点，但这些技术奇迹般地奏效！

在某些情况下（即，如果 P1 和 P2 具有相同的 x 值但具有不同的 y 值），切线将正好是垂直的，此时 P3 = “无穷点”。

如果 P1 是“无穷点”，则 P1 + P2 = P2。同样，如果 P2 是无穷点，则 P1 + P2 = P1。这显示了无穷点起到了零的作用。

事实证明 + 是可结合的，这意味着（A + B）+ C = A +（B + C）。这意味着我们可以写出 A + B + C，而无需括号或模棱两可。

现在我们已经定义了加法，我们可以按照扩展加法的标准方式来定义乘法。对于椭圆曲线上的点 P，如果 k 是一个整数，则 kP = P + P + P + … + P（k 次）。请注意，在这种情况下，有时会令人困惑地称 k 为“指数”。

公钥

公钥是使用椭圆曲线乘法从私钥计算的，这是不可逆的过程：K = k × G，其中 k 是私钥，G 是一个称为生成点的常量点，而 K 是生成的公钥。反向操作称为“找到离散对数”——即在已知 K 的情况下计算 k，就像尝试所有可能的 k 值一样（即，进行穷举搜索）。在我们演示如何从私钥生成公钥之前，让我们稍微详细地了解一下椭圆曲线密码学。

> 椭圆曲线乘法是密码学家称之为“陷阱门”函数的一种类型：在一个方向上（乘法）很容易进行，而在相反方向（除法）则几乎不可能进行。拥有私钥的人可以轻松地创建公钥，然后将其与世界分享，因为他们知道没有人能够反转该函数并从公钥计算出私钥。这种数学技巧成为无法伪造和安全的数字签名的基础，证明对比特币资金的控制权。

从形式为随机生成的数字 k 的私钥开始，我们将其乘以曲线上的一个预定点，称为生成点 G，以在曲线上的其他位置生成另一个点，即相应的公钥 K。生成点作为 secp256k1 标准的一部分被指定，并且对比特币中的所有密钥始终是相同的：\

K = k × G

其中，k 是私钥，G 是生成点，而 K 是结果公钥，即曲线上的一个点。由于生成点对于所有比特币用户都是相同的，私钥 k 乘以 G 总是会得到相同的公钥 K。k 和 K 之间的关系是固定的，但只能在一个方向上计算，即从 k 到 K。这就是为什么比特币公钥（K）可以与任何人共享而不会暴露用户的私钥（k）。

> 私钥可以转换为公钥，但公钥不能转换回私钥，因为数学运算只能单向进行。

实现椭圆曲线乘法时，我们将之前生成的私钥 k 与生成点 G 相乘，以得到公钥 K：

\ K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD × G

公钥 K 被定义为点 K = (x, y)：

\ K = (x, y)

其中，

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

为了可视化点与整数的乘法，我们将使用实数上的简单椭圆曲线——请记住，数学是一样的。我们的目标是找到生成点 G 的倍数 kG，这等同于将 G 加上自身，连续 k 次。在椭圆曲线中，将一个点加到它自身相当于在该点上画一条切线，并找到它再次与曲线相交的位置，然后将该点在 x 轴上反射。

图 4-4 展示了在曲线上执行几何操作来得到 G、2G、4G 的过程。

> 许多比特币实现使用 libsecp256k1 加密库执行椭圆曲线数学运算。

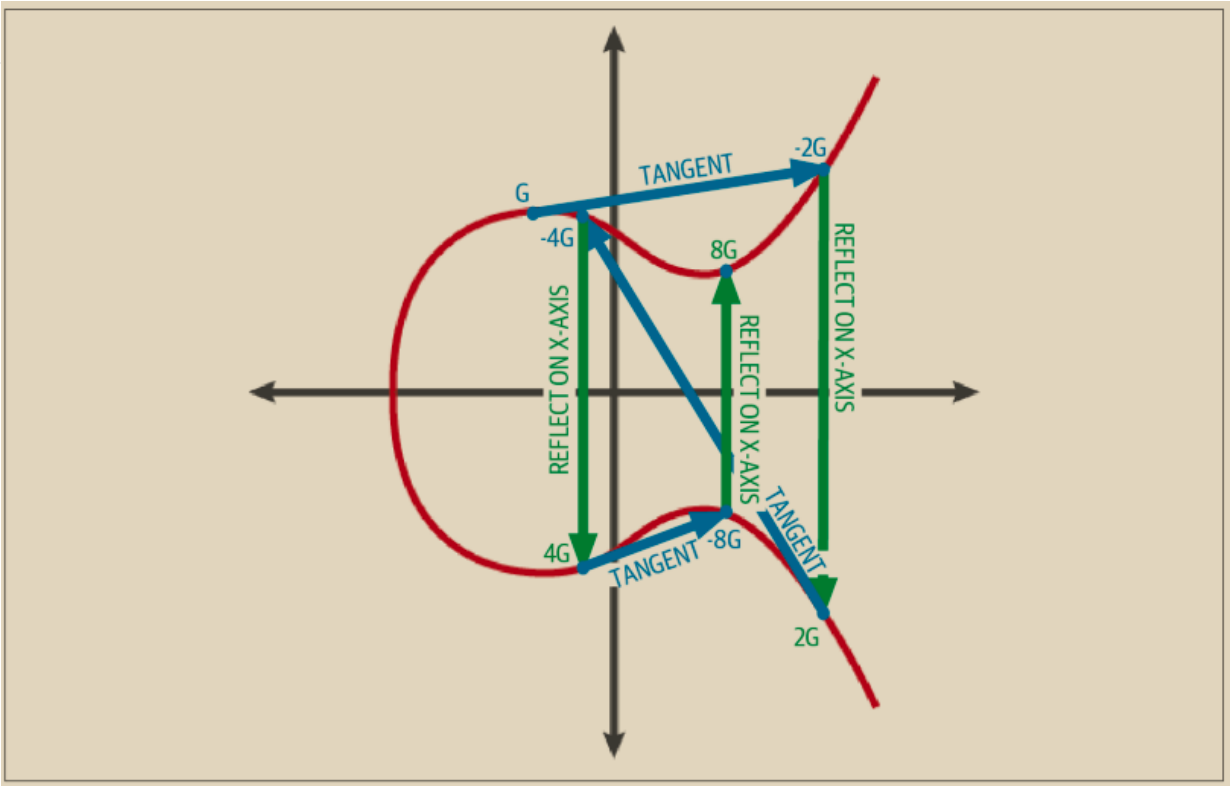


图 4-4. 椭圆曲线密码学：在椭圆曲线上将点 G 与整数 k 相乘的可视化示例。

输出和输入脚本

虽然原始比特币论文中的图 4-1 显示了公钥（pubkeys）和签名（sigs）直接使用的情况，但比特币的第一个版本实际上是将支付发送到一个称为输出脚本的字段，并且这些比特币的花费由一个称为输入脚本的字段授权。这些字段允许执行额外的操作，除了（或代替）验证签名是否对应于公钥。例如，一个输出脚本可以包含两个公钥，并要求在支出的输入脚本中放置两个相应的签名。

\ 稍后，在第 143 页的“交易脚本和脚本语言”中，我们将详细了解脚本。目前，我们只需要理解比特币是接收到一个行为类似于公钥的输出脚本，并且比特币的支出是由一个行为类似于签名的输入脚本授权的。

\

IP 地址：比特币的原始地址（P2PK）

我们已经确定，Alice 可以通过将她的一些比特币分配给 Bob 的一个公钥来支付 Bob。但是，Alice 如何获得 Bob 的一个公钥呢？Bob 可以直接给她一份副本，但让我们再次看一下我们在“公钥”中使用的公钥。注意它非常长。想象一下 Bob 试图通过电话向 Alice 读取公钥的情景：

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

在比特币软件的早期版本中，允许支付者输入接收者的 IP 地址，如图 4-5 所示。这个功能后来被移除了——使用 IP 地址存在很多问题——但对它的快速描述将有助于我们更好地理解为什么某些特性可能被添加到比特币协议中。

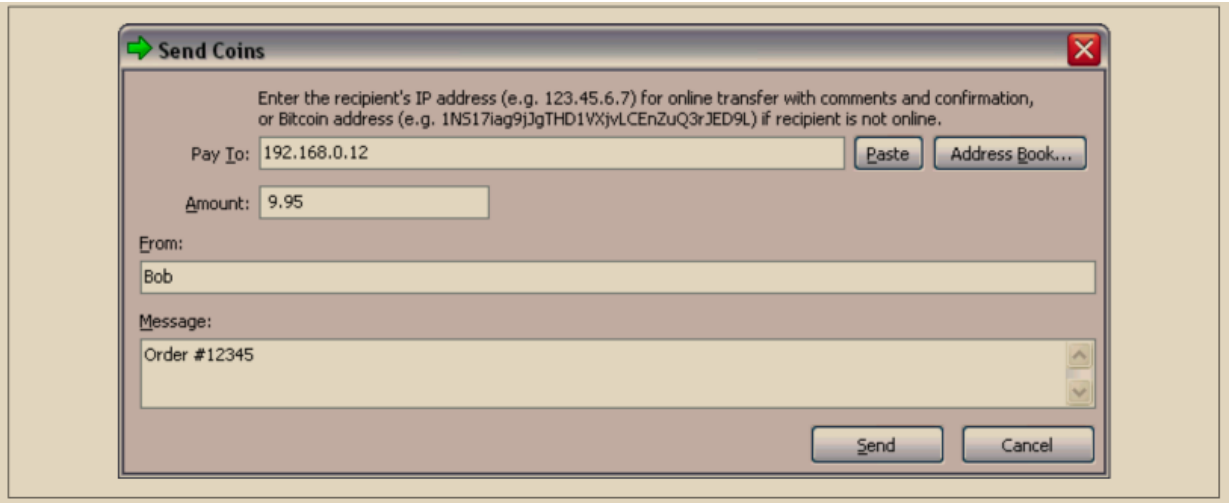


图 4-5. 早期通过互联网档案馆发送比特币的截图。

如果Alice在比特币0.1中输入了Bob的IP地址，她的全节点将与他的全节点建立连接，并从Bob的钱包接收一个新的公钥，这个公钥以前从未给过任何人。这个新的公钥很重要，以确保支付给Bob的不同交易不能被某人通过查看区块链连接在一起，因为他们注意到所有交易都支付给了相同的公钥。

\ 使用她的节点从Bob的节点接收的公钥，Alice的钱包将构造一个非常简单的输出脚本来支付：

\<Bob's public key> OP_CHECKSIG

Bob稍后可以使用完全由他的签名组成的输入脚本来花费该输出：

\<Bob's signature>

为了弄清楚输出和输入脚本的作用，您可以将它们组合在一起（输入脚本在前），然后注意每个数据片段（用尖括号显示）都被放置在一个名为堆栈的项目列表的顶部。当遇到操作码（opcode）时，它使用来自堆栈的项目，从最顶部的项目开始。让我们看看从组合脚本开始是如何工作的：

\ \ OP_CHECKSIG

\ 对于这个脚本，Bob的签名被放置在堆栈上，然后Bob的公钥被放置在签名上面。OP_CHECKSIG操作消耗两个元素，从公钥开始，接着是签名，将它们从堆栈中移除。它验证签名是否对应于公钥，并且对交易中的各个字段进行签名。如果签名正确，OP_CHECKSIG会用值1替换自身在堆栈上的位置；如果签名不正确，它会用0替换自身。如果在评估结束时堆栈顶部有一个非零项，则脚本通过。如果交易中的所有脚本都通过，并且交易的所有其他细节都有效，则完整节点将认为该交易有效。

简而言之，前面的脚本使用了原始论文中描述的相同公钥和签名，但增加了两个脚本字段和一个操作码的复杂性。这似乎是多余的工作，但当我们看到接下来的部分时，我们将开始看到它的好处。

这种类型的输出今天被称为付给公钥，或简称为P2PK。它从未被广泛用于支付，而且几乎十年来没有任何广泛使用的程序支持IP地址支付。\

历史遗留地址P2PKH

为要支付的人输入IP地址有很多优点，但也有很多缺点。一个特别的缺点是接收者需要他们的钱包在线并在其IP地址上可访问。对于很多人来说，这不是一个选择。他们晚上关闭计算机，他们的笔记本电脑进入睡眠模式，他们处于防火墙后面，或者他们正在使用网络地址转换（NAT）。

\ 这让我们回到了接收者（如Bob）不得不向发送者（如Alice）提供一个较长的公钥的问题。早期比特币开发者所知的最短版本的比特币公钥是65字节，用十六进制表示时相当于130个字符。然而，比特币已经包含了几个比65字节大得多的数据结构，需要在比特币的其他部分中以最少的数据安全地引用。

比特币通过哈希函数实现了这一点，哈希函数是一种将潜在大量数据混淆（哈希）并输出固定量数据的函数。当给定相同的输入时，加密哈希函数总是产生相同的输出，并且安全函数还会使得别人很难选择不同的输入以产生先前看到的输出。这使得输出成为对输入的承诺。实际上，这是一种承诺，只有输入x会产生输出X。

例如，假设我想问你一个问题，并且以一种你无法立即阅读的形式给出我的答案。假设问题是，“中本聪什么时候开始研发比特币？”我会以SHA256哈希函数的输出形式给你我的答案的承诺，这是比特币中最常用的函数：

94d7a772612c8f2f2ec609d41f5bd3d04a5aa1dfe3582f04af517d396a302e4e

稍后，当你告诉我你对问题答案的猜测后，我可以揭示我的答案，并向你证明我的答案作为哈希函数的输入产生了与我之前给你的完全相同的输出：

$ echo "2007. He said about a year and a half before Oct 2008" | sha256sum
94d7a772612c8f2f2ec609d41f5bd3d04a5aa1dfe3582f04af517d396a302e4e

现在想象一下，我们向 Bob 提出问题：“你的公钥是什么？” Bob 可以使用哈希函数为他的公钥提供一个具有密码学安全性的承诺。如果他后来透露了他的密钥，而我们验证它产生了与他先前给我们的完全相同的承诺，那么我们可以确定它是用于创建早期承诺的确切相同的密钥。

SHA256 哈希函数被认为非常安全，并产生 256 位（32 字节）的输出，不到原始比特币公钥大小的一半。然而，还有其他略微不太安全的哈希函数可以产生较小的输出，比如 RIPEMD-160 哈希函数，其输出为 160 位（20 字节）。由于 Satoshi Nakamoto 从未说明原因，比特币的原始版本通过首先使用 SHA256 对密钥进行哈希，然后使用 RIPEMD-160 对该输出进行哈希，从而对公钥进行承诺；这产生了对公钥的 20 字节承诺。

我们可以从算法的角度来看。从公钥 K 开始，我们计算 SHA256 哈希，然后计算结果的 RIPEMD-160 哈希，得到一个 160 位（20 字节）的数字：

A = RIPEMD160(SHA256(K))

现在我们知道如何对公钥做出承诺了，接下来我们需要弄清楚如何在交易中使用它。考虑以下输出脚本：

OP_DUP OP_HASH160 \<Bob's commitment> OP_EQUAL OP_CHECKSIG

以及以下输入脚本：

\ \

它们组合在一起形成以下脚本：

OP_DUP OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG

\ 正如我们在“IP地址：比特币的原始地址（P2PK）”上一页所做的那样，我们开始将项目放在堆栈上。 首先放入Bob的签名，然后放入他的公钥。 OP_DUP操作复制顶部项目，因此堆栈顶部和次顶部现在都是Bob的公钥。 OP_HASH160操作消耗（删除）顶部公钥，并用RIPEMD160（SHA256（K））对其进行哈希处理，因此现在堆栈顶部是Bob的公钥的哈希。 接下来，将Bob的公钥的承诺添加到堆栈的顶部。 OP_EQUALVERIFY操作消耗了顶部两个项目，并验证它们是否相等； 如果Bob在输入脚本中提供的公钥与Alice支付中用于创建承诺的公钥相同，那么情况应该是如此。 如果OP_EQUALVERIFY失败，则整个脚本失败。 最后，我们得到的是一个仅包含Bob的签名和他的公钥的堆栈； OP_CHECKSIG操作码验证它们是否相互对应，并且签名是否承诺了交易。

虽然这种支付给公钥哈希（P2PKH）的过程可能看起来有些复杂，但它允许Alice的付款只包含对他的公钥的20字节承诺，而不是公钥本身，在比特币的原始版本中会是65字节。 这使得Bob不必与Alice交流太多的数据。

但是，我们还没有讨论Bob如何从他的比特币钱包将这20个字节传递给Alice的钱包。 字节值有常用的编码，例如十六进制，但如果在复制承诺时出现任何错误，将导致比特币被发送到不可支配的输出，从而永远丢失。 在下一节中，我们将介绍紧凑编码和可靠的校验和。

\ \

Base58check编码

为了以紧凑的方式表示长数字，使用较少的符号，许多计算机系统使用基数高于10的混合字母数字表示。例如，传统的十进制系统使用10个数字，0到9，而十六进制系统使用16个数字，包括字母A到F作为另外六个符号。以十六进制格式表示的数字比等价的十进制表示更短。更紧凑的是，base64表示法使用26个小写字母，26个大写字母，10个数字，以及两个更多的字符，如“+”和“/”，用于在基于文本的媒体（例如电子邮件）上传输二进制数据。

Base58是与base64类似的编码，使用大写和小写字母以及数字，但省略了一些经常被误认为相同并且在某些字体中显示时看起来相同的字符。具体来说，base58是base64去掉了数字0，大写字母O，小写字母l，大写字母I以及符号“+”和“/”。或者更简单地说，它是一个不包含刚提到的四个字符（0、O、l、I）的小写字母、大写字母和数字集合。示例4-2显示了完整的base58字母表。

示例4-2. 比特币的base58字母表

123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

\ 为了增加额外的安全性，防止输入错误或转录错误，base58check在base58字母表中包含了一个校验和。校验和是添加到正在编码的数据末尾的额外的四个字节。校验和是从编码数据的哈希派生出来的，因此可以用来检测转录和输入错误。当提供base58check代码时，解码软件将计算数据的校验和，并将其与代码中包含的校验和进行比较。如果两者不匹配，则引入了错误，base58check数据无效。这可以防止被误输入的比特币地址被钱包软件接受为有效的目标地址，否则会导致资金损失。

要将数据（一个数字）转换为base58check格式，首先我们要为数据添加一个前缀，称为“版本字节”，它用于轻松识别所编码的数据类型。例如，前缀零（十六进制中的0x00）表示数据应该用作传统P2PKH输出脚本中的承诺（哈希）。常见版本前缀列表如表4-1所示：

接下来，我们计算“双SHA”校验和，意思是我们在之前的结果（前缀与数据连接起来）上两次应用SHA256哈希算法：

checksum = SHA256(SHA256(prefix||data))

从结果得到的32字节哈希（哈希的哈希）中，我们只取前四个字节。这四个字节用作错误检查码或校验和。校验和附加到末尾。

结果由三个部分组成：一个前缀、数据和一个校验和。然后，使用先前描述的base58字母表对此结果进行编码。图4-6说明了base58check编码过程。

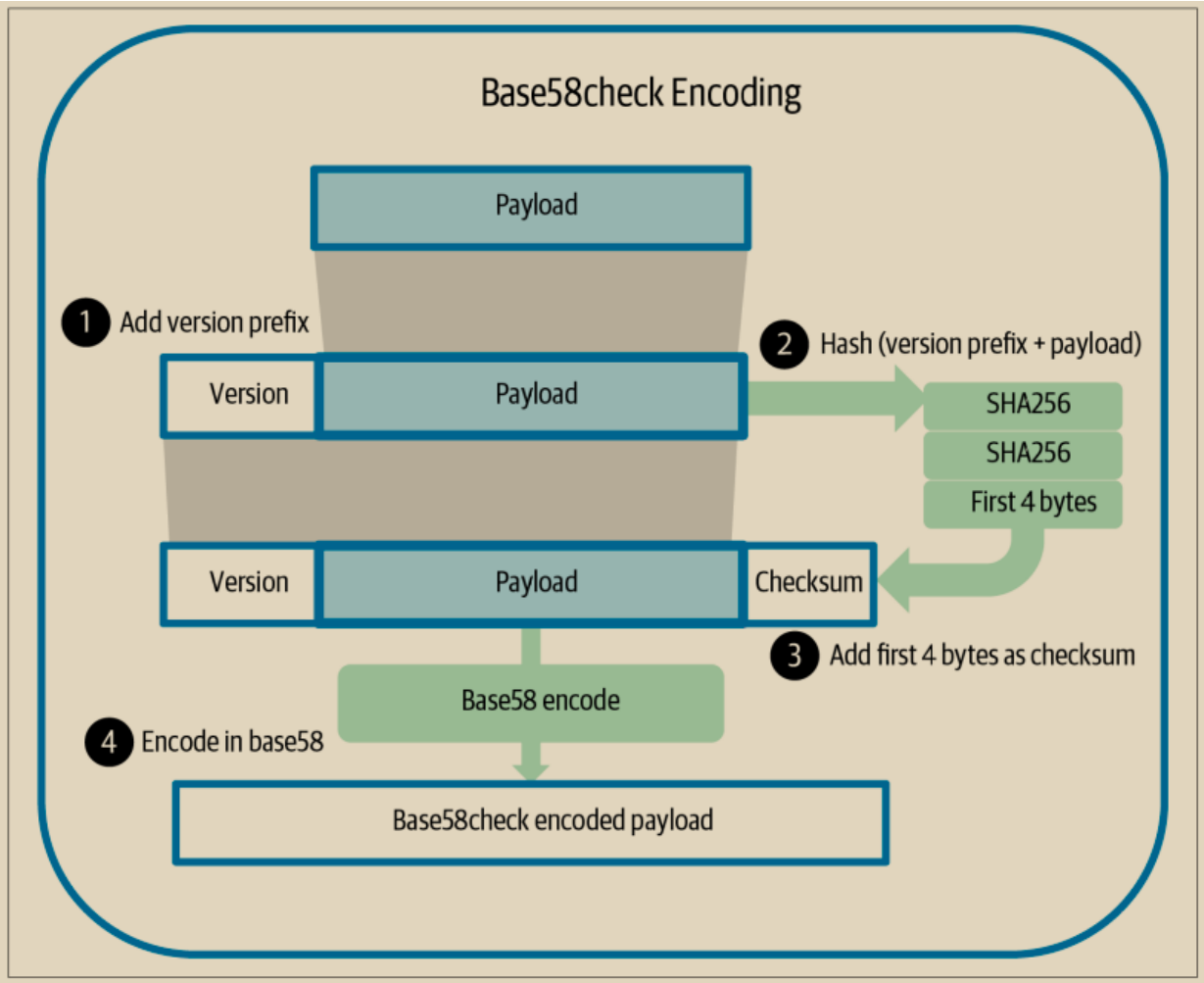


图 4-6. Base58check编码：一种用于明确编码比特币数据的基于Base58、带版本和校验和的格式。

在比特币中，除了公钥承诺之外，其他数据也以base58check编码的方式呈现给用户，以使数据紧凑、易于阅读和易于检测错误。base58check编码中的版本前缀用于创建易于区分的格式，当以base58编码时，在base58check编码的有效负载的开头包含特定的字符。这些字符使人们能够轻松识别编码的数据类型以及如何使用它。例如，以1开头的base58check编码的比特币地址与以5开头的base58check编码的私钥钱包导入格式（WIF）是有区别的。示例版本前缀及其相应的base58字符如下表4-1所示:

表4-1. Base58check版本前缀和编码结果示例

类型	版本前缀（十六进制）	Base58 结果前缀
Address for pay to public key hash (P2PKH)	0x00	1
Address for pay to script hash (P2SH)	0x05	3
Testnet Address for P2PKH	0x6F	m or n
Testnet Address for P2SH	0xC4	2
Private Key WIF	0x80	5, K, or L
BIP32 Extended Public Key	0x0488B21E	xpub

将公钥与基于哈希的承诺和base58check编码相结合，图4-7说明了将公钥转换为比特币地址的过程。

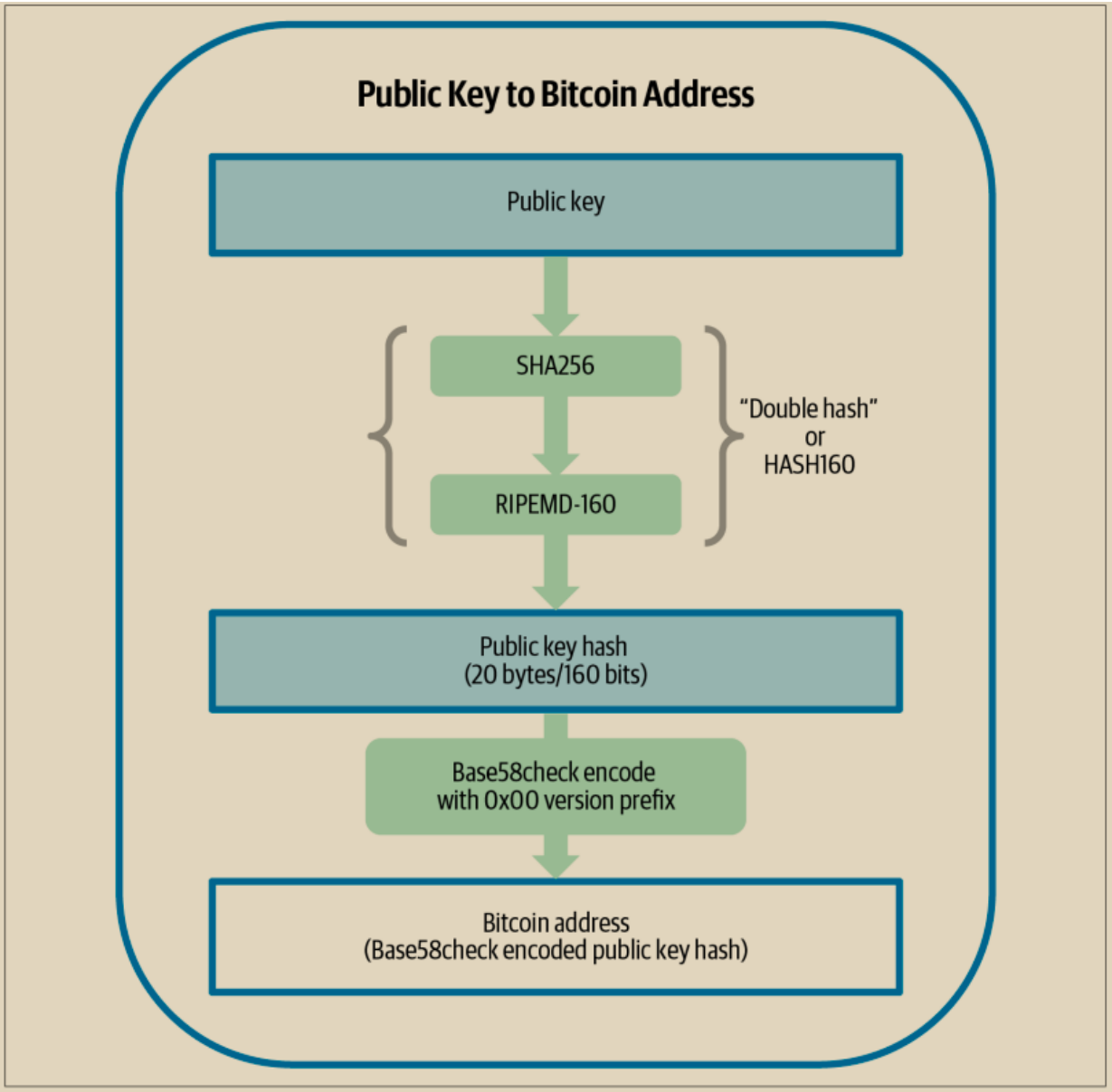


图 4-7. 公钥到比特币地址：将公钥转换为比特币地址的过程

压缩公钥

当比特币刚被设计时，其开发者只知道如何创建 65 字节的公钥。然而，后来的一个开发者发现了一种只使用 33 字节的替代编码公钥的方法，而且这种方法与当时所有的比特币全节点都是向后兼容的，因此不需要改变比特币协议。这些 33 字节的公钥被称为压缩公钥，而原始的 65 字节的公钥被称为非压缩公钥。使用更小的公钥可以减小交易的大小，允许在同一个区块中进行更多的支付。

正如我们在第 59 页的“公钥”部分中看到的那样，公钥是椭圆曲线上的一个点 (x, y)。因为曲线表示一个数学函数，曲线上的一个点代表了方程的一个解，因此如果我们知道 x 坐标，我们可以通过求解方程 y^2 mod p = (x^3 + 7) mod p 来计算出 y 坐标。这使得我们只需存储公钥点的 x 坐标，省略 y 坐标，从而减小了密钥的大小和存储它所需的空间，节省了 256 位的空间。在每个交易中几乎减小了 50% 的大小，随着时间的推移，节省了大量的数据！

以下是我们在第 59 页的“公钥”部分创建的私钥生成的公钥：

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

同样的公钥可以表示为一个 520 位的数（130 个十六进制数字），带有前缀 04，后面跟着 x 和 y 坐标，格式为 04 x y：

K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A\\
07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

与未压缩的公钥以 04 作为前缀不同，压缩的公钥以 02 或 03 作为前缀。让我们看看为什么会有两个可能的前缀：因为等式的左侧是 y2，所以 y 的解是一个平方根，可以有正值或负值。从视觉上看，这意味着得到的 y 坐标可以在 x 轴上方或下方。正如您从图 4-2 中的椭圆曲线图中所看到的那样，曲线是对称的，这意味着它像镜子一样被 x 轴反射。因此，虽然我们可以省略 y 坐标，但我们必须存储 y 的符号（正或负）；换句话说，我们必须记住它是否在 x 轴上方或下方，因为这两种选项代表不同的点和不同的公钥。在二进制算术上计算素数阶 p 的有限域上的椭圆曲线时，y 坐标要么是偶数，要么是奇数，这对应于前面解释的正负号。因此，为了区分 y 的两个可能值，如果 y 是偶数，则存储带有前缀 02 的压缩公钥，如果 y 是奇数，则存储带有前缀 03 的压缩公钥，从而使软件能够从 x 坐标推断出 y 坐标，并将公钥展开为点的完整坐标。公钥压缩如图 4-8 所示。

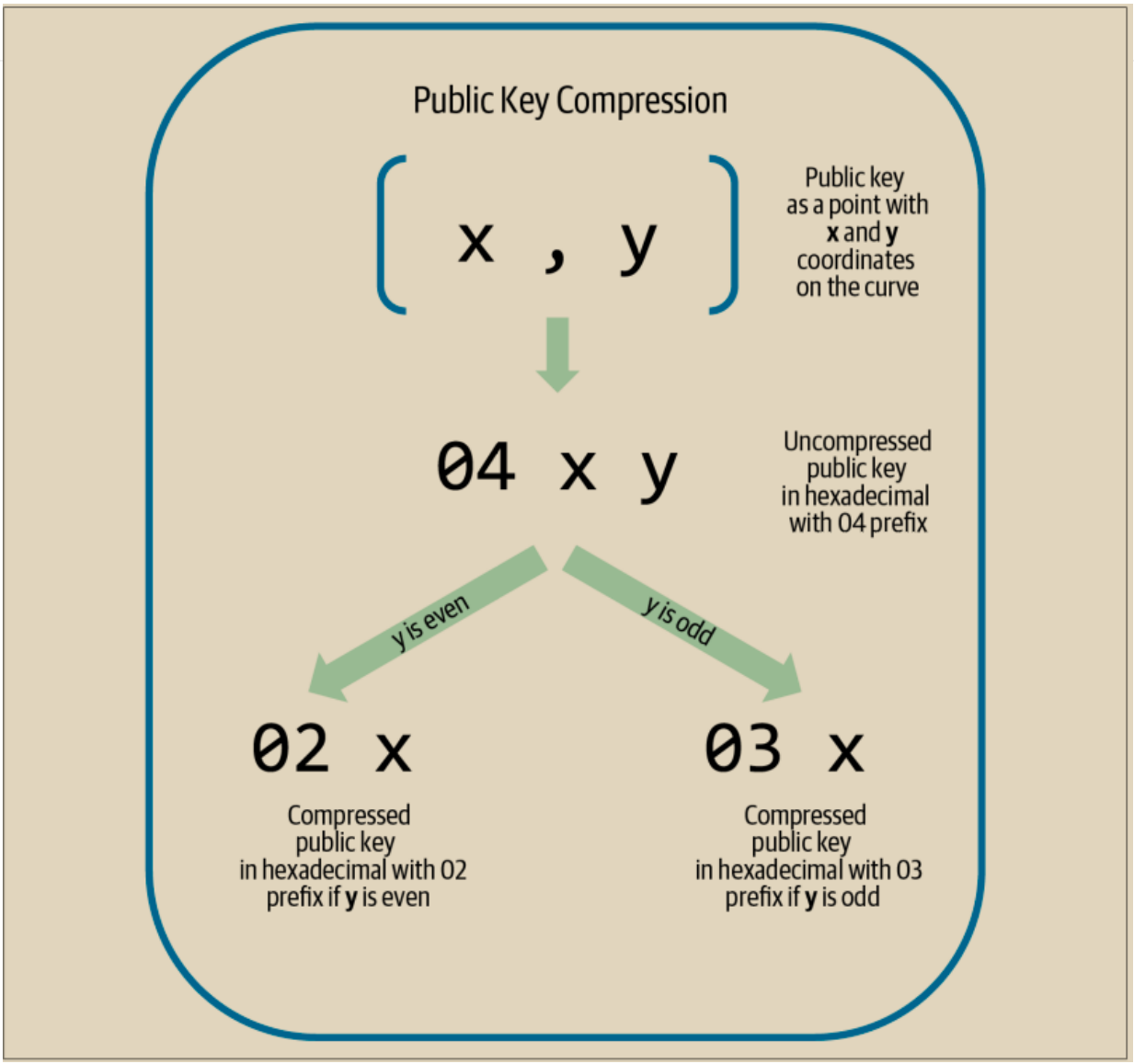


图 4-8. 公钥压缩

这里是与在第59页生成的相同的公钥，显示为压缩的公钥，以264位（66个十六进制数字）存储，并带有前缀03，表示y坐标为奇数：

K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

\ 这个压缩的公钥对应于相同的私钥，意味着它是由相同的私钥生成的。然而，它与未压缩的公钥看起来不同。更重要的是，如果我们将这个压缩的公钥转换为一个承诺，使用HASH160函数（RIPEMD160(SHA256(K)）），它将产生一个不同的承诺，而不是未压缩的公钥，从而导致不同的地址。这可能会让人困惑，因为这意味着一个单一的私钥可以产生以两种不同格式（压缩和未压缩）表示的公钥，从而产生两个不同的比特币地址。然而，私钥对于两个比特币地址是相同的。

几乎所有比特币软件现在默认使用压缩的公钥，并且在稍后的协议升级中添加了某些新功能时需要使用压缩的公钥。

然而，一些软件仍然需要支持未压缩的公钥，比如从旧钱包导入私钥的钱包应用程序。当新钱包扫描区块链以查找旧的P2PKH输出和输入时，它需要知道是扫描65字节的密钥（以及这些密钥的承诺）还是33字节的密钥（以及它们的承诺）。如果没有扫描正确的类型，用户可能无法花费他们的全部余额。为了解决这个问题，当私钥从钱包中导出时，在新的比特币钱包中使用的WIF有所不同，以指示这些私钥已被用于生成压缩的公钥。

历史遗留的支付到脚本哈希地址（P2SH）

正如我们在前面的章节中所看到的，接收比特币的人（比如Bob）可以要求支付给他的比特币在其输出脚本中包含某些约束条件。当Bob花费这些比特币时，他将需要使用输入脚本来满足这些约束条件。在“IP地址：比特币的原始地址（P2PK）”中，约束条件简单地是输入脚本需要提供适当的签名。在“用于P2PKH的遗留地址”中，还需要提供适当的公钥。

对于支付给Bob的输出脚本中放置Bob想要的约束条件的支出者（比如Alice），Bob需要将这些约束条件传达给她。这类似于Bob需要将他的公钥传达给她的问题。就像这个问题一样，公钥可以相当大，Bob使用的约束条件也可能非常大——潜在地有数千字节。这不仅是需要传达给Alice的数千字节，而且是她每次想向Bob支付款项时都需要支付交易费用的数千字节。然而，使用哈希函数为大量数据创建小承诺的解决方案在这里也适用。

2012年比特币协议的BIP16升级允许输出脚本承诺一个赎回脚本（redeem script）。当Bob花费他的比特币时，他的输入脚本需要提供一个与承诺匹配的赎回脚本，以及满足赎回脚本所需的任何数据（例如签名）。让我们首先想象一下，Bob想要要求两个签名来花费他的比特币，一个来自他的桌面钱包，另一个来自硬件签名设备。他将这些条件放入一个赎回脚本中：

\ OP_CHECKSIGVERIFY \ OP_CHECKSIG

然后，他使用与P2PKH承诺相同的HASH160机制创建对赎回脚本的承诺，即RIPEMD160（SHA256（脚本））。该承诺被放置到输出脚本中，使用特殊的模板：

OP_HASH160 OP_EQUAL

> 使用支付至脚本哈希（P2SH）时，必须使用特定的P2SH模板，在输出脚本中没有额外的数据或条件。如果输出脚本不是完全是 OP_HASH160 <20字节> OP_EQUAL，赎回脚本将不会被使用，而且任何比特币可能要么无法支配，要么可以被任何人支配（意味着任何人都可以取走它们）。

当Bob要花费他收到的支付，用于他脚本的承诺时，他会使用一个包含赎回脚本的输入脚本，将其序列化为一个单独的数据元素。他还提供了满足赎回脚本所需的签名，按照它们被操作码消耗的顺序放置：

\ \ \

当比特币全节点接收到Bob的交易时，它们将验证序列化的赎回脚本是否会哈希为与承诺相同的值。然后，它们将其替换为堆栈上的反序列化值：

\ \ \ OP_CHECKSIGVERIFY \ OP_CHECKSIG

脚本被执行，如果通过并且所有其他交易细节都正确，则交易有效。

P2SH的地址也使用base58check创建。版本前缀设置为5，这会导致编码地址以3开头。P2SH地址的示例是3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM。

> P2SH不一定等同于多重签名交易。P2SH地址通常代表一个多重签名脚本，但也可能代表编码其他类型交易的脚本。

\ P2PKH和P2SH是使用base58check编码的仅有两种脚本模板。它们现在被称为传统地址，并且随着时间的推移变得越来越不常见。传统地址已经被bech32地址家族所取代。

> **P2SH碰撞攻击**
>
> 基于哈希函数的所有地址理论上都容易受到攻击者独立找到产生哈希函数输出（承诺）的相同输入的影响。在比特币的情况下，如果攻击者以与原始用户相同的方式找到输入，他们将知道用户的私钥并能够花费该用户的比特币。攻击者独立生成现有承诺的输入的机会与哈希算法的强度成正比。对于像HASH160这样的安全160位算法，这种可能性是1/2^160。这是一种原像攻击。
>
> 攻击者还可以尝试生成两个不同的输入（例如，赎回脚本），这些输入会产生相同的承诺。对于完全由单方创建的地址，攻击者生成现有承诺的不同输入的机会也大约是1/2^160，对于HASH160算法而言也是如此。这是二阶原像攻击。
>
> 然而，当攻击者能够影响原始输入值时情况就会改变。例如，攻击者参与了多方签名脚本的创建，在这种情况下，他们在了解所有其他参与方的公钥之后不需要提交自己的公钥。在这种情况下，哈希算法的强度降低到其平方根。对于HASH160，概率变为1/2^80。这是一种碰撞攻击。 为了将这些数字放入上下文，截至2023年初，所有比特币矿工每小时执行大约2^80个哈希函数。他们运行与HASH160不同的哈希函数，因此他们现有的硬件无法为其创建碰撞攻击，但比特币网络的存在证明了对HASH160等160位函数的碰撞攻击是切实可行的。比特币矿工已经花费了数十亿美元的特殊硬件，因此创建碰撞攻击不会很便宜，但有些组织预计将获得数十亿美元的比特币到与多方参与的过程相关的地址，这可能会使攻击变得有利可图。 有着成熟的密码协议用于预防碰撞攻击，但一个简单的解决方案，不需要钱包开发人员具有任何特殊的知识，就是简单地使用更强大的哈希函数。比特币的后续升级使这成为可能，新的比特币地址提供了至少128位的碰撞抵抗能力。执行2^128次哈希操作将需要所有当前的比特币矿工大约320亿年。
>
> 尽管我们认为没有任何立即威胁到任何人创建新的P2SH地址，但我们建议所有新钱包使用更新类型的地址，以消除地址碰撞攻击的担忧。

Bech32地址

2017年，比特币协议进行了升级。当使用该升级时，它可以防止交易标识符（txids）在未经付款用户（或在需要多个签名时的签名者群体）的同意下被更改。该升级被称为隔离见证（简称segwit），还为区块中的交易数据提供了额外的容量以及其他几项好处。然而，想要直接获得segwit好处的用户必须接受新输出脚本的支付。

正如在“支付到脚本哈希”中提到的那样，P2SH输出类型的一个优点是支付者（比如Alice）不需要知道接收者（比如Bob）使用的脚本的详细信息。隔离见证升级旨在利用这种机制，允许用户立即开始通过使用P2SH地址来访问许多新的好处。但是，要让Bob获得所有的好处，他需要Alice的钱包使用不同类型的脚本支付给他。这需要Alice的钱包进行升级，以支持新的脚本。

起初，比特币开发人员提出了BIP142，它将继续使用base58check，但带有一个新的版本字节，类似于P2SH的升级。但是，让所有钱包升级到具有新base58check版本的新脚本预计将需要几乎与让它们升级到全新地址格式一样多的工作量，因此几位比特币贡献者着手设计最佳可能的地址格式。他们确定了base58check存在的几个问题：

- 其大小写混合的表示形式使得朗读或转录时不方便。试着向一位朋友朗读本章中的某个传统地址，让他们把它记录下来。注意到你不得不在每个字母前面加上“大写”和“小写”这样的词语。此外，请注意当你审阅他们写下的内容时，一些字母的大写和小写版本在许多人的手写中可能看起来相似。
- 可以检测错误，但无法帮助用户纠正这些错误。例如，如果你在手动输入地址时意外地交换了两个字符的位置，你的钱包几乎肯定会警告存在错误，但它不会帮助你找出错误出现在哪里。你可能需要花费几分钟才能最终发现错误。
- 使用大小写混合字母表还需要额外的空间来在QR码中进行编码，而QR码通常用于在钱包之间分享地址和发票。这额外的空间意味着在相同分辨率下，QR码需要更大才能被快速扫描。

在开发segwit地址格式的开发人员找到了一种新的地址格式，称为bech32（发音为软“ch”，如“besh thirty-two”）。“bech”代表BCH，是三位于1959年和1960年发现bech32基于的循环码的缩写人物的首字母。 “32”代表bech32字母表中的字符数（类似于base58check中的58）：

- Bech32仅使用数字和一种大小写字母（最好以小写字母呈现）。尽管其字母表几乎是base58check字母表的一半大小，但用于付款给见证公钥哈希（P2WPKH）脚本的bech32地址只比等效的P2PKH脚本的传统地址稍长。
- Bech32既能检测错误，也能帮助纠正错误。在长度符合预期的地址中，它在数学上保证可以检测到影响四个字符或更少的任何错误；这比base58check更可靠。对于更长的错误，它将不会在十亿次操作中检测到它们的出现，这大致与base58check的可靠性相同。更重要的是，对于仅有少量错误的地址，它可以告诉用户错误发生的位置，使他们能够快速纠正轻微的抄录错误。示例4-3显示了输入有错误的地址的示例。

示例4-3。Bech32错别字检测

地址：

bc1p9nh05ha8wrljf7ru236aw~~n~~4t2x0d5ctkkywm~~v~~9sclnm4t0av2vgs4k3au7

检测到的错误以粗体和删除线显示。使用bech32地址解码器演示生成。

- Bech32最好只用小写字母书写，但在将地址编码成QR码之前，这些小写字母可以被大写字母替换。这样可以使用一种特殊的QR编码模式，从而占用更少的空间。注意图4-9中相同地址的两个QR码的大小和复杂度的差异。

图 4-9. 相同的Bech32地址，一个使用小写字母编码，另一个使用大写字母编码

- Bech32利用了SegWit设计的一种升级机制，使得支付者钱包能够支付尚未使用的输出类型。其目标是允许开发者构建一个钱包，使其能够在今天支付到Bech32地址，并且该钱包仍然能够支付给未来协议升级中添加的新功能的Bech32地址的用户。人们希望再也不需要经历系统范围的升级周期，以使人们能够完全使用P2SH和SegWit。

Bech32地址所存在的问题

Bech32地址在每个方面都可能取得成功，除了一个问题。关于它们能够检测错误的数学保证只有在你输入到钱包中的地址长度与原始地址的长度相同时才适用。如果在转录过程中添加或删除任何字符，则该保证将不适用，你的钱包可能会将资金发送到错误的地址。然而，即使没有这个保证，人们仍然认为用户添加或删除字符几乎不可能产生带有有效校验和的字符串，从而确保用户的资金安全。

不幸的是，Bech32算法中一个常量的选择恰巧使得在以字母“p”结尾的地址的倒数第二个位置添加或删除字母“q”变得非常容易。在这种情况下，你还可以多次添加或删除字母“q”。这样做有时会被校验和检测到，但远不如Bech32对字符替换错误的预期频率那样少见。请参见示例4-4。

示例4-4. 扩展Bech32地址的长度而不使其校验和失效

预期的 Bech32 地址:

bc1pqqqsq9txsqp

具有有效校验和的不正确地址:

bc1pqqqsq9txsqqqqp

bc1pqqqsq9txsqqqqqqp

bc1pqqqsq9txsqqqqqqqqp

bc1pqqqsq9txsqqqqqqqqqp

bc1pqqqsq9txsqqqqqqqqqqp

对于初始版本的隔离见证（版本0），这并不是一个实际的问题。对于v0隔离见证输出，只定义了两个有效的长度：22字节和34字节。这对应于bech32地址的长度为42个字符或62个字符，因此某人需要在bech32地址的倒数第二个位置添加或删除字母“q”20次，才能向无效地址发送资金，而钱包无法检测到。然而，如果未来实施了基于隔离见证的升级，这将成为用户的一个问题。

Bech32m

尽管Bech32对于隔离见证v0表现良好，但开发人员不希望在隔离见证的后续版本中不必要地限制输出大小。在没有限制的情况下，在Bech32地址中添加或删除一个“q”可能导致用户意外将资金发送到一个不可花费的输出，或者是可以由任何人花费的输出（允许任何人夺取比特币）。开发人员对Bech32问题进行了彻底的分析，并发现改变算法中的一个常数将消除这个问题，确保对最多五个字符的任何插入或删除的检测失败的频率低于十亿分之一。

具有单个不同常数的Bech32版本被称为修改过的Bech32（Bech32m）。对于相同底层数据，Bech32和Bech32m地址中的所有字符都将相同，除了最后六个字符（校验和）。这意味着钱包需要知道正在使用哪个版本才能验证校验和，但是两种地址类型都包含一个内部版本字节，使得确定这一点变得容易。

为了同时处理Bech32和Bech32m，我们将查看Bech32m比特币地址的编码和解析规则，因为它们包括了解析Bech32地址的能力，并且是比特币钱包的当前推荐地址格式。

Bech32m地址以人类可读部分（HRP）开头。BIP173中有关于创建自己的HRP的规则，但是对于比特币，您只需要了解已选择的HRP，如表4-2所示。

表 4-2. 比特币中Bech32 人类可读部分前缀

人类可读部分前缀	网络类型
bc	比特币主网
tb	比特币测试网

HRP 之后是一个分隔符，即数字 "1"。早期的协议分隔符使用了冒号，但某些操作系统和应用程序允许用户双击单词以突出显示以进行复制和粘贴，这样的功能将不会延伸到并且会经过冒号。数字确保双击高亮功能可以与支持 bech32m 字符串的任何程序一起工作（其中包括其他数字）。选择数字 "1" 是因为 bech32 字符串在其他情况下不使用它，以防止数字 "1" 与小写字母 "l" 之间的意外音译。

Bech32m 地址的另一部分称为 "数据部分"。这部分包括三个元素：

\ 见证版本

一个单字节，在bech32m比特币地址中作为分隔符之后的单个字符编码。该字母表示SegWit版本。字母“q”是SegWit v0的编码，其中引入了bech32地址的初始版本。字母“p”是SegWit v1的编码（也称为Taproot），在那里开始使用bech32m。SegWit有17个可能的版本，对于比特币来说，bech32m数据部分的第一个字节必须解码为数字0到16（包括0和16）。

见证程序

长度为2到40个字节。对于SegWit v0，见证程序必须是20或32个字节；其他长度都无效。截至目前，对于SegWit v1，唯一定义的长度是32个字节，但以后可能会定义其他长度。

校验和

为6个字符。这是使用 BCH 码创建的，一种错误纠正码（尽管对于比特币地址，稍后我们将看到它仅用于错误检测，而不是纠正）。

\ 让我们通过一个示例来说明这些规则，创建bech32和bech32m地址。在以下所有示例中，我们将使用Python的bech32m参考代码。

我们将首先生成四个输出脚本，每个脚本对应于出版时使用的不同segwit输出类型，以及一个对应于尚未定义含义的未来segwit版本。这些脚本列在表4-3中。

表4-3 不同类型segwit输出的脚本

输出类型	示例脚本
P2WPKH	OP_0 2b626ed108ad00a944bb2922a309844611d25468
P2WSH	OP_0 648a32e50b6fb7c5233b228f60a6a2ca4158400268844c4bc295ed5e8c3d626f
P2TR	OP_1 2ceefa5fa770ff24f87c5475d76eab519eda6176b11dbe1618fcf755bfac5311
Future Example	OP_16 0000

\ 对于P2WPKH输出，见证程序包含一个承诺，其构造方式与“P2PKH的传统地址”（见第63页）中的P2PKH输出的承诺完全相同。将公钥传递到SHA256哈希函数中。然后，将结果的32字节摘要传递到RIPEMD-160哈希函数中。该函数的摘要（承诺）放置在见证程序中。

对于付款给见证脚本哈希（P2WSH）输出，我们不使用P2SH算法。而是将脚本传递到SHA256哈希函数中，并使用该函数的32字节摘要作为见证程序。对于P2SH，SHA256摘要再次与RIPEMD-160哈希，但在某些情况下可能不安全；有关详细信息，请参见“P2SH碰撞攻击”（第73页）。使用SHA256而不是RIPEMD-160的结果是P2WSH承诺为32字节（256位），而不是20字节（160位）。

对于付款到taproot（P2TR）输出，见证程序是secp256k1曲线上的一个点。它可以是一个简单的公钥，但在大多数情况下，它应该是一个公钥，该公钥承诺了一些附加数据。我们将在“Taproot”（第178页）中详细了解这个承诺。

对于未来segwit版本的示例，我们简单地使用最高可能的segwit版本号（16）和允许的最小见证程序（2字节）以及空值。

现在我们知道了版本号和见证程序，我们可以将它们转换为bech32地址。让我们使用Python的bech32m参考库快速生成这些地址，然后深入了解发生了什么：

$ github="https://raw.githubusercontent.com"

$ wget $github/sipa/bech32/master/ref/python/segwit_addr.py

$ python

```
>>> from segwit_addr import *
>>> from binascii import unhexlify
>>> help(encode)
encode(hrp, witver, witprog)
 Encode a segwit address.
>>> encode('bc', 0, unhexlify('2b626ed108ad00a944bb2922a309844611d25468'))
'bc1q9d3xa5gg45q2j39m9y32xzvygcgay4rgc6aaee'
>>> encode('bc', 0,unhexlify('648a32e50b6fb7c5233b228f60a6a2ca4158400268844c4bc295ed5e8c3d626f'))
'bc1qvj9r9egtd7mu2gemy28kpf4zefq4ssqzdzzycj7zjhk4arpavfhsct5a3p'
>>> encode('bc', 1,
unhexlify('2ceefa5fa770ff24f87c5475d76eab519eda6176b11dbe1618fcf755bfac5311'))
'bc1p9nh05ha8wrljf7ru236awm4t2x0d5ctkkywmu9sclnm4t0av2vgs4k3au7'
>>> encode('bc', 16, unhexlify('0000'))
'bc1sqqqqkfw08p'
```

如果我们打开文件segwit_addr.py并查看代码的操作，我们会注意到bech32（用于segwit v0）和bech32m（用于后续segwit版本）之间唯一的区别是常数：

BECH32_CONSTANT = 1

BECH32M_CONSTANT = 0x2bc830a3

接下来我们注意到生成校验和的代码。在校验和的最后一步中，适当的常数通过异或运算合并到值中。这个单一的值是bech32和bech32m之间唯一的区别。

校验和创建完成后，数据部分（包括见证版本、见证程序和校验和）中的每个5位字符都转换为字母数字字符。

要解码回输出脚本，我们反向工作。首先让我们使用参考库来解码我们的两个地址：

```
>>> help(decode)
decode(hrp, addr)
 Decode a segwit address.
>>> _ = decode("bc", "bc1q9d3xa5gg45q2j39m9y32xzvygcgay4rgc6aaee")
>>> _[0], bytes(_[1]).hex()
(0, '2b626ed108ad00a944bb2922a309844611d25468')
>>> _ = decode("bc",
 "bc1p9nh05ha8wrljf7ru236awm4t2x0d5ctkkywmu9sclnm4t0av2vgs4k3au7")
>>> _[0], bytes(_[1]).hex()
(1, '2ceefa5fa770ff24f87c5475d76eab519eda6176b11dbe1618fcf755bfac5311')
```

我们获得了见证版本和见证程序。 这些可以插入到我们输出脚本的模板中：

\ \

例如：

OP_0 2b626ed108ad00a944bb2922a309844611d25468

OP_1 2ceefa5fa770ff24f87c5475d76eab519eda6176b11dbe1618fcf755bfac5311

这里需要注意的一个可能错误是，见证版本0对应的是OP_0，使用的是字节0x00，但见证版本1对应的是OP_1，使用的是字节0x51。而见证版本2到16分别对应0x52到0x60。

当实现bech32m编码或解码时，我们强烈建议您使用BIP350提供的测试向量。我们也要求您确保您的代码通过与支付尚未定义的未来segwit版本相关的测试向量。即使您无法在新的比特币功能推出时立即添加支持，这也将帮助您的软件在未来多年保持可用。

私钥格式

私钥可以用多种不同格式表示，所有这些格式都对应于相同的256位数。表4-4显示了用于表示私钥的几种常见格式。不同的格式在不同的情况下使用。十六进制和原始二进制格式在软件内部使用，很少显示给用户。WIF用于钱包之间的密钥导入/导出，并经常用于私钥的QR码（条形码）表示。

> **私钥格式的现代相关性**
>
> 早期的比特币钱包软件在初始化新用户钱包时生成一个或多个独立的私钥。当初始密钥集已全部使用完毕时，钱包可能会生成额外的私钥。单个私钥可以导出或导入。每当生成新的私钥或导入私钥时，都需要创建钱包的新备份。
>
> 后来的比特币钱包开始使用确定性钱包，其中所有私钥都是从单个种子值生成的。这些钱包只需要在典型的链上使用中备份一次。但是，如果用户从其中一个这些钱包中导出一个单独的私钥，而攻击者获得了该密钥以及钱包的一些非私密数据，他们可能会推导出钱包中的任何私钥，从而允许攻击者窃取所有的钱包资金。此外，无法将密钥导入确定性钱包。这意味着几乎没有现代钱包支持导出或导入单个密钥的功能。本节中的信息主要是对需要与早期比特币钱包兼容的人感兴趣。
>
> 有关更多信息，请参阅“分层确定性（HD）密钥生成（BIP32）”。

表 4-4. 不同类型私钥（编码格式）

类型	前缀	描述
Hex	None	64位十六进制字符
WIF	5	Base58check 编码: 具有版本前缀 128 和 32 位校验和的 base58。
WIF-compressed	K 或 L	与上述相同，但在编码之前添加了后缀 0x01。

表4-5 展示了以几种不同格式生成的私钥。

表4-5。示例：相同的密钥，不同的格式

格式	私钥
Hex	1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

所有这些表示法都是展示相同数字、相同私钥的不同方式。它们看起来不同，但任何一个格式都可以轻松转换为任何其他格式。

压缩私钥

常用的术语“压缩私钥”是一个错误的说法，因为当一个私钥被导出为WIF-压缩时，它实际上比“未压缩”私钥长一个字节。这是因为私钥有一个额外的字节后缀（在表4-6中以十六进制表示为01），表示该私钥来自一个较新的钱包，应该只用于生成压缩的公钥。私钥本身并没有被压缩，也不能被压缩。术语“压缩私钥”实际上意味着“只应该从中派生压缩的公钥的私钥”，而“未压缩私钥”实际上意味着“只应该从中派生未压缩的公钥的私钥”。为了避免进一步混淆，您应该只将导出格式称为“WIF-压缩”或“WIF”，而不是将私钥本身称为“压缩”。

表4-6显示了相同的密钥，以WIF和WIF-压缩格式编码

表 4-6. 示例：相同的私钥，不同格式

格式	私钥
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Hex-compressed	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD01
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

请注意，十六进制压缩私钥格式在末尾多了一个额外的字节（十六进制中的01）。虽然基于Base58编码的版本前缀对于WIF和WIF-compressed格式是相同的（0x80），但数字末尾添加了一个字节会导致Base58编码的第一个字符从5变为K或L。可以将这视为十进制编码中数字100和数字99之间的差异。虽然100比99多一位数字，但它的前缀也是1而不是9。随着长度的变化，前缀也会受到影响。在Base58中，数字前缀5会随着数字长度增加一个字节而变为K或L。

请记住，这些格式不能互换使用。在实现了压缩公钥的新钱包中，私钥只能导出为WIF-compressed格式（带有K或L前缀）。如果钱包是旧版本，并且不使用压缩公钥，则私钥只能导出为WIF格式（带有5前缀）。这里的目标是向导入这些私钥的钱包传达一个信号，告诉它是否必须在区块链中搜索压缩或非压缩的公钥和地址。

如果一个比特币钱包能够实现压缩公钥，它将在所有交易中使用它们。钱包中的私钥将用于从曲线派生出压缩的公钥点。压缩的公钥将用于生成比特币地址，并在交易中使用。从实现了压缩公钥的新钱包中导出私钥时，WIF将被修改，私钥将添加一个字节的后缀01。结果生成的Base58check编码的私钥称为“压缩WIF”，并以K或L字母开头，而不是以5开头，这是来自旧钱包的WIF编码（非压缩）私钥的情况。

\

高级密钥和地址

在接下来的章节中，我们将探讨更高级的密钥和地址形式，比如虚拟地址和纸钱包。

自定义地址

\ 自定义地址是有效的比特币地址，其中包含可读的人类可读信息。例如，1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33是一个有效的地址，其中的前四个base58字母形成了单词“Love”。自定义地址需要生成和测试数十亿个候选私钥，直到找到一个具有所需模式的比特币地址。虽然自定义地址生成算法中有一些优化，但该过程基本上涉及随机选择一个私钥，派生出公钥，派生出比特币地址，并检查它是否与所需的自定义模式匹配，重复数十亿次直到找到匹配项。

一旦找到与所需模式匹配的自定义地址，就可以使用派生自该地址的私钥来花费比特币，方式与任何其他地址完全相同。自定义地址与任何其他地址一样安全。它们依赖于与任何其他地址相同的椭圆曲线密码学（ECC）和安全哈希算法（SHA）。您无法更轻松地找到具有自定义模式起始的地址的私钥，就像您无法更轻松地找到任何其他地址的私钥一样。

Eugenia是一位在菲律宾从事儿童慈善工作的主任。假设Eugenia正在组织一场筹款活动，并希望使用一个自定义的比特币地址来宣传筹款活动。Eugenia将创建一个以“1Kids”开头的自定义地址，以促进儿童慈善筹款活动。让我们看看这个自定义地址将如何创建，以及对Eugenia的慈善机构安全性意味着什么。

生成自定义地址

重要的是要意识到，比特币地址只是用base58字母表中的符号表示的数字。像“1Kids”这样的模式搜索可以看作是在范围从1Kids111111111111111111111111111111到1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzzz之间搜索地址。在这个范围内有大约5829个（大约1.4 × 1051）地址，所有地址都以“1Kids”开头。

表4-7显示了以“1Kids”开头的自定义地址范围。

From 1Kids11111111111111111111111111111

1Kids11111111111111111111111111112

1Kids11111111111111111111111111113

…

To 1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz

让我们将模式“1Kids”视为一个数字，并看看在比特币地址中可能找到这个模式的频率（见表4-8）。一台普通的桌面电脑，没有任何专门的硬件，可以每秒搜索大约100,000个密钥。

表4-8显示了一个自定义模式（1KidsCharity）的频率和在桌面PC上的平均搜索时间。

长度	模式	频率	平均搜索时间
1	1K	58次	< 1毫秒
2	1Ki	3364次	50 毫秒
3	1Kid	195,000 次	< 2 秒
4	1Kids	1100万次	1分钟
5	1KidsC	6.56亿	1小时
6	1KidsCh	380亿	2天
7	1KidsCha	2.2兆（一兆=10^12）	3-4个月
8	1KidsChar	128兆	13-18年
9	1KidsChari	7千兆	800年
10	1KidsCharit	400千兆	46000年
11	1KidsCharity	23京（一京=10^16）	250万年

如您所见，即使是有几千台计算机，Eugenia 也不可能很快就创建出 "1KidsCharity" 这个自定义地址。每增加一个字符，搜索的难度就增加了58倍。通常情况下，具有超过七个字符的模式通常是由专用硬件完成的，例如具有多个图形处理单元（GPU）的定制台式机。在 GPU 系统上进行的自定义搜索比在通用 CPU 上快得多。

另一种找到自定义地址的方法是将工作外包给一组自定义矿工。自定义矿工池是一种服务，允许那些拥有快速硬件的人通过为他人搜索自定义地址而赚取比特币。以一定费用，Eugenia 可以将对七个字符模式自定义地址的搜索外包，并在几小时内获得结果，而不必花费数月时间在 CPU 上进行搜索。

生成自定义地址是一种蛮力练习：尝试一个随机密钥，检查生成的地址是否与所需模式匹配，重复直至成功。

自定义地址安全和隐私

自定义地址在比特币早期很受欢迎，但截至2023年几乎完全从使用中消失。这种趋势有两个可能的原因：

确定性钱包

正如我们在“恢复码”中看到的那样，大多数现代钱包可以通过简单地写下几个单词或字符来备份每个密钥。这是通过使用确定性算法从这些单词或字符派生钱包中的每个密钥来实现的。除非用户为创建的每个自定义地址备份了额外的数据，否则无法在确定性钱包中使用自定义地址。更实际的是，大多数使用确定性密钥生成的钱包根本不允许从自定义地址生成器导入私钥或密钥扭曲。

避免地址重用

使用自定义地址接收多笔支付到同一个地址会在所有这些支付之间创建一个联系。如果Eugenia的非营利组织需要向税务机构报告其收入和支出，那么这可能是可以接受的。然而，这也会降低那些支付给Eugenia或从她那里接收支付的人的隐私。例如，Alice可能想匿名捐赠，Bob可能不希望他的其他客户知道他给Eugenia提供折扣价格。

我们不指望在未来会看到很多自定义地址，除非上述问题得到解决。

纸钱包

纸钱包是印在纸上的私钥。通常，纸钱包还包括相应的比特币地址以便使用，但这并非必要，因为地址可以从私钥派生出来。

> 纸钱包是一种已经过时的技术，对大多数用户来说是危险的。生成它们涉及许多微妙的陷阱，其中最重要的是生成代码可能被“后门”入侵。很多比特币就是这样被盗走的。这里仅仅展示纸钱包供信息参考，不应用于存储比特币。使用恢复码备份您的密钥，可能还可以使用硬件签名设备来存储密钥和签署交易。请勿使用纸钱包。

纸钱包有许多不同的设计和尺寸，具有许多不同的功能。图4-10显示了一个样品纸钱包。

图 4-10. 简单纸钱包的示例

\ 有些纸钱包设计成礼物赠送，具有季节性主题，如圣诞节和新年。其他设计用于存放在银行保险箱或保险柜中，私钥以某种方式隐藏，可以使用不透明的刮开贴纸或折叠并用防篡改的粘贴铝箔密封。其他设计则包含私钥和地址的额外副本，形式类似于可拆卸的车票副本，允许您存储多个副本以防火灾、水灾或其他自然灾害。

从比特币最初的公钥设计到现代地址和脚本，如bech32m和支付到taproot，甚至是未来比特币升级的地址，您已经了解到比特币协议如何允许支付者识别应该接收他们支付的钱包。但当实际上是您的钱包接收支付时，您肯定希望确保即使发生了钱包数据丢失，您仍然能够访问这笔资金。在下一章中，我们将看看比特币钱包是如何设计来保护其资金免受各种威胁的。

综合介绍

创建私钥和公钥对是使比特币钱包能够接收和花费比特币的关键部分。但是，如果丢失了私钥，任何人都无法再花费相应的公钥接收到的比特币。 多年来，钱包和协议开发人员一直致力于设计系统，使用户能够在出现问题后恢复对比特币的访问，而不会在其他时间损害安全性。

在本章中，我们将探讨钱包采用的一些不同方法，以防止数据丢失导致资金损失。一些解决方案几乎没有缺点，并被现代钱包普遍采用。我们将简单地将这些解决方案推荐为最佳实践。其他解决方案既有优势也有劣势，导致不同的钱包作者做出不同的权衡。在这些情况下，我们将描述各种可用的选项。

\

独立密钥生成

物理现金的钱包持有现金，因此许多人错误地认为比特币钱包包含比特币。事实上，许多人称为比特币钱包的东西（我们称之为钱包数据库，以区别于钱包应用程序）只包含密钥。这些密钥与记录在区块链上的比特币相关联。通过向比特币全节点证明您控制这些密钥，您可以花费相关的比特币。

简单的钱包数据库包含接收比特币的公钥以及允许创建授权支出这些比特币所需的私钥。其他钱包的数据库可能仅包含公钥，或者仅包含授权支出交易所需的某些私钥。它们的钱包应用程序通过与外部工具（如硬件签名设备或多重签名方案中的其他钱包）合作来生成必要的签名。

钱包应用程序可以独立生成后续计划使用的每个钱包密钥，如图5-1所示。所有早期的比特币钱包应用程序都这样做，但是用户需要在每次生成和分发新密钥时备份钱包数据库，这可能频繁到每次生成新地址以接收新支付时。如果未能及时备份钱包数据库，用户将无法访问未备份的密钥收到的任何资金。

对于每个独立生成的密钥，用户需要备份大约32字节，再加上开销。一些用户和钱包应用程序尝试通过仅使用单个密钥来最小化需要备份的数据量。虽然这样做可能是安全的，但严重降低了该用户及其所有交易对象的隐私。重视隐私的人和他们的同行为每笔交易创建新的密钥对，生成的钱包数据库只能合理地使用数字媒体进行备份。

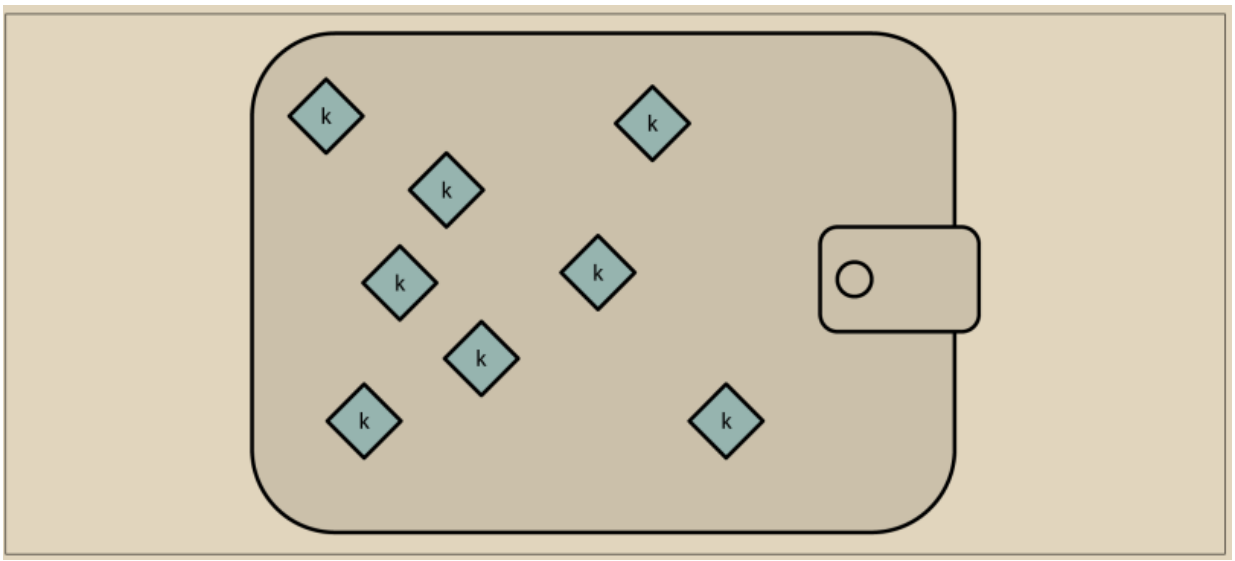


图 5-1. 非确定性密钥生成：存储在钱包数据库中的一组独立生成的密钥。

现代钱包应用程序不会独立生成密钥，而是使用可重复（确定性）算法从单个随机种子派生密钥。

确定性密钥生成

当给定相同的输入时，哈希函数总是会产生相同的输出，但是如果输入稍有改变，输出将会不同。如果函数是加密安全的，那么没有人应该能够预测新的输出，即使他们知道新的输入。

这使我们能够将一个随机值转换为几乎无限数量的看似随机值。更有用的是，稍后使用相同的哈希函数和相同的输入（称为种子）将产生相同的看似随机值：

```
# Collect some entropy (randomness)
$ dd if=/dev/random count=1 status=none | sha256sum
f1cc3bc03ef51cb43ee7844460fa5049e779e7425a6349c8e89dfbb0fd97bb73 -
# Set our seed to the random value
$ seed=f1cc3bc03ef51cb43ee7844460fa5049e779e7425a6349c8e89dfbb0fd97bb73
# Deterministically generate derived values
$ for i in {0..2} ; do echo "$seed + $i" | sha256sum ; done
50b18e0bd9508310b8f699bad425efdf67d668cb2462b909fdb6b9bd2437beb3 -
a965dbcd901a9e3d66af11759e64a58d0ed5c6863e901dfda43adcd5f8c744f3 -
19580c97eb9048599f069472744e51ab2213f687d4720b0efc5bb344d624c3aa -
```

如果我们将这些派生的值用作我们的私钥，那么稍后我们可以使用之前使用的算法和我们的种子值生成完全相同的私钥。使用确定性密钥生成的用户可以通过简单地记录他们的种子和他们使用的确定性算法的引用来备份他们钱包中的每个密钥。例如，即使Alice有一百万个比特币分别存放在一百万个不同的地址中，她只需要备份以下内容，以便稍后恢复对这些比特币的访问：

f1cc 3bc0 3ef5 1cb4 3ee7 8444 60fa 5049

e779 e742 5a63 49c8 e89d fbb0 fd97 bb73

基本顺序确定性密钥生成的逻辑图如图5-2所示。然而，现代钱包应用程序有一种更聪明的方法来实现这一点，允许公钥与其相应的私钥分别派生，从而使得私钥比公钥更安全地存储成为可能。

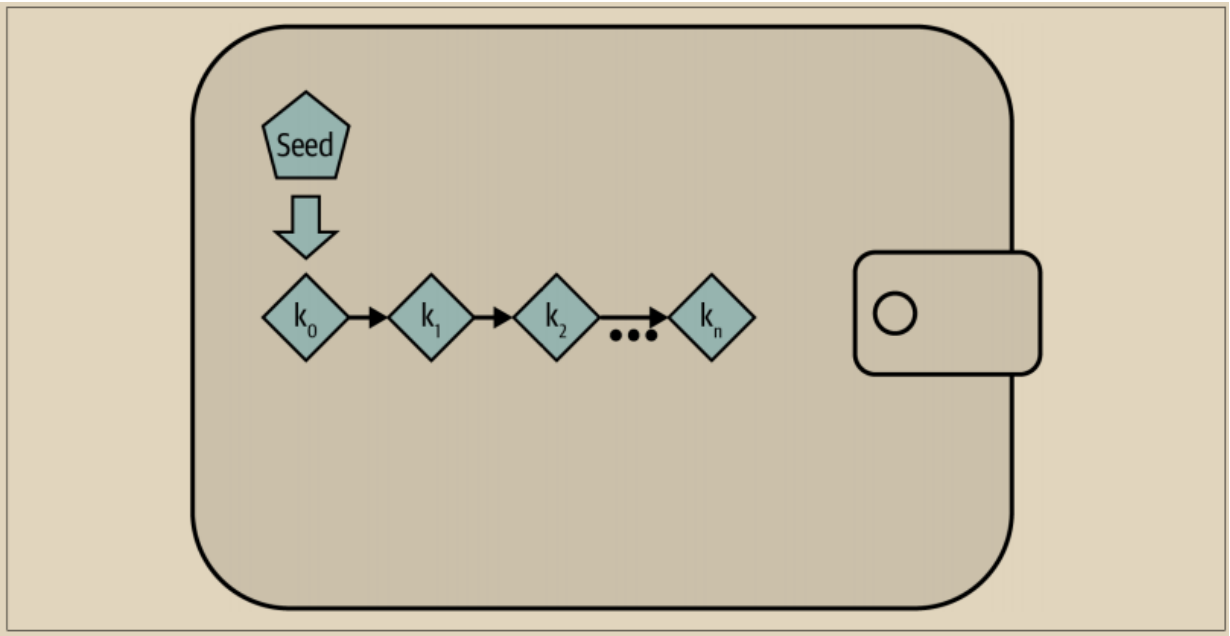


图 5-2. 确定性密钥生成：从钱包数据库的种子派生出的确定性密钥序列。

子公钥派生

在第59页的“公钥”部分，我们学习了如何使用椭圆曲线密码学（ECC）从私钥创建公钥。虽然在椭圆曲线上的操作并不直观，但它们类似于常规算术中使用的加法、减法和乘法操作。换句话说，可以对公钥进行加法、减法或乘法操作。考虑我们在第59页的“公钥”部分中使用的操作，用主私钥（k）和生成器点（G）生成公钥（K）：

\ K = k × G

通过简单地将相同的值添加到等式的两边，可以创建一个派生密钥对，称为子密钥对：

K + 123 × G == k + 123 × G

> 在本书中的方程中，我们使用单个等号表示诸如 K = k × G 这样的操作，其中计算了变量的值。我们使用双重等号来显示方程两边是等价的，或者如果两边不等价，则操作应返回 false（而不是 true）。

\ 这样做的一个有趣的结果是，可以使用完全公开的信息将123添加到公钥中。例如，Alice生成了公钥K并将其提供给Bob。Bob不知道私钥，但他知道全局常数G，因此他可以向公钥添加任何值，以生成一个派生的公共子键。然后，如果他告诉Alice他向公钥添加的值，她可以将相同的值添加到私钥中，从而生成与Bob创建的公共子键相对应的派生私有子键。

换句话说，即使你不知道父私钥的任何信息，也可以创建子公钥。添加到公钥的值称为密钥调整值。如果使用确定性算法来生成密钥调整值，那么即使某人不知道私钥，也可以从单个公共父键创建一系列基本上无限的公共子键。控制私有父键的人可以使用相同的密钥调整值创建所有相应的私有子键。

这种技术通常用于将钱包应用的前端（不需要私钥）与签名操作（需要私钥）分离。例如，Alice的前端向想要支付她的人分发她的公钥。稍后，当她想要花费收到的钱时，她可以向硬件签名设备（有时令人困惑地称为硬件钱包）提供她使用的密钥调整值，该设备安全地存储她的原始私钥。硬件签名者使用这些调整值来推导所需的子私钥，并使用它们来签名交易，然后将已签名的交易返回给较不安全的前端以广播到比特币网络。

公共子键派生可以产生与先前看到的图5-2类似的线性键序列，但现代钱包应用程序使用一种更巧妙的技巧来提供键树而不是单个序列，如下一节所述。

分层确定性(HD)密钥生成(BIP32)

我们所知道的每个现代比特币钱包默认都使用分层确定性（HD）密钥生成。这个标准在BIP32中定义，使用确定性密钥生成和可选的公共子密钥派生算法，生成一组密钥的树。在这个树中，任何密钥都可以是一系列子密钥的父密钥，而这些子密钥中的任何一个都可以是另一系列子密钥（原始密钥的子孙）的父密钥。树的深度没有任何任意限制。这种树结构如图5-3所示

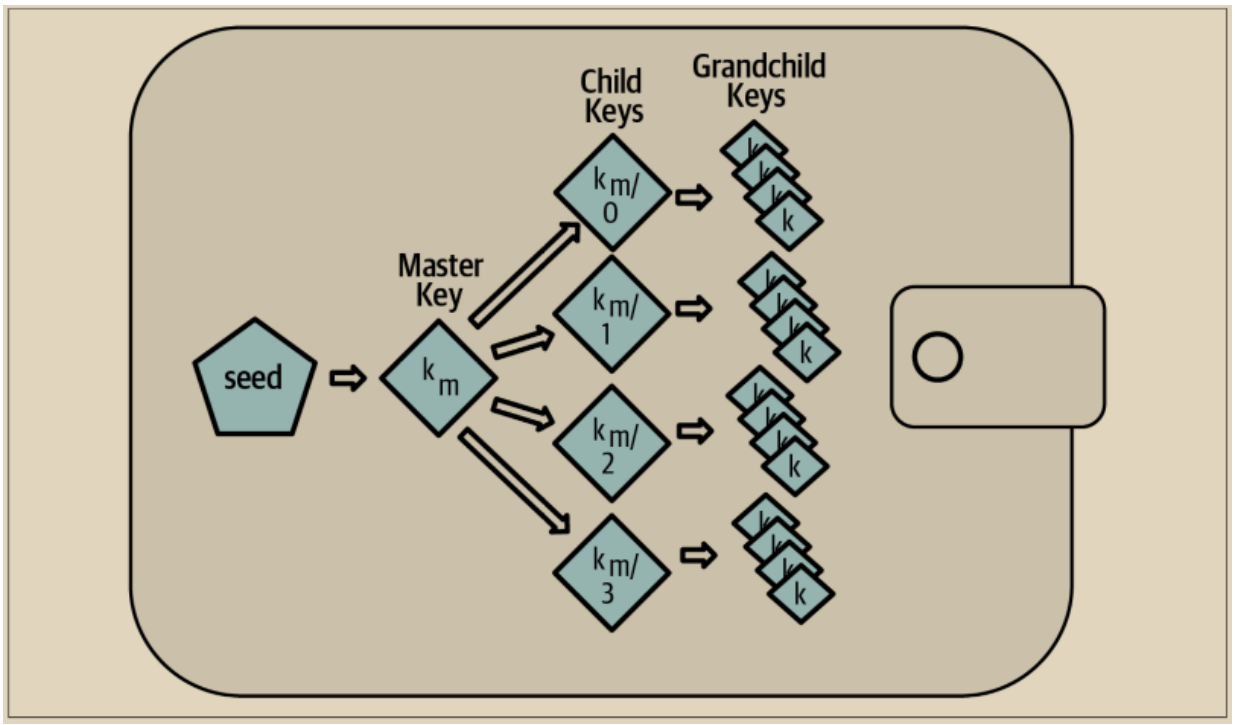


图 5-3. HD钱包：从单个种子生成的密钥树

\ 树结构可以用来表示额外的组织含义，比如特定的子密钥分支用于接收传入付款，另一个分支用于接收传出付款的找零。密钥的分支也可以在企业设置中使用，将不同的分支分配给部门、子公司、特定功能或会计类别。

我们将在“从种子创建HD钱包”一节中进行对HD钱包的详细探讨。

种子和恢复码

HD钱包是一种非常强大的机制，可以管理从单个种子派生出的许多密钥。如果您的钱包数据库曾经损坏或丢失，您可以使用原始种子重新生成钱包的所有私钥。但是，如果其他人获取了您的种子，他们也可以生成所有的私钥，从而能够窃取单签名钱包中的所有比特币，并降低多重签名钱包中比特币的安全性。在本节中，我们将讨论几种恢复码，旨在使备份更加简便和安全。

虽然种子是大型随机数，通常为128至256位，但大多数恢复码使用人类语言单词。使用单词的主要动机之一是使恢复码易于记忆。例如，考虑使用十六进制和单词编码的恢复码，如示例5-1所示。

示例5-1。一个以十六进制和英语单词编码的种子

Hex-encoded:

0C1E 24E5 9177 79D2 97E1 4D45 F14E 1A1A

Word-encoded:

army van defense carry jealous true garbage claim echo media make crunch

可能存在一些情况，记住恢复码是一个强大的功能，比如当您无法携带物理财产（比如写在纸上的恢复码），以免它们被外部可能窃取您比特币的当事人扣押或检查。然而，大多数情况下，仅依赖记忆是危险的：

- 如果您忘记了恢复码并且无法访问原始的钱包数据库，您的比特币将永远丢失。
- 如果您死亡或遭受严重伤害，您的继承人没有访问您原始钱包数据库的权限，他们将无法继承您的比特币。
- 如果有人认为您记住了一个恢复码，可以让他们访问比特币，他们可能会试图强迫您透露该代码。 截至撰写本文时，比特币贡献者詹姆逊·洛普（Jameson Lopp）记录了100多起针对比特币和其他数字资产的潜在所有者的物理攻击，包括至少三起死亡案件和许多事件，其中有人遭受折磨、被扣为人质，或者家人受到威胁。

> 即使您使用设计用于轻松记忆的恢复码类型，我们强烈建议您考虑将其写下来。

\ 截至撰写本文时，有几种不同类型的恢复码正在广泛使用：

BIP39

作为过去十年生成恢复码最流行的方法，BIP39涉及生成一个随机的字节序列，为其添加一个校验和，然后将数据编码成一系列12到24个单词（可以根据用户的母语进行本地化）。这些单词（加上一个可选的密码）经过一种键拉伸函数处理，输出被用作种子。BIP39恢复码有几个缺点，后来的方案试图解决这些问题。

Electrum v2

在Electrum钱包（版本2.0及以上）中使用的这种基于单词的恢复码比BIP39具有几个优点。它不依赖于一个全球性的单词列表，该列表必须由每个兼容程序的每个版本实现，而且其恢复码包括一个版本号，提高了可靠性和效率。与BIP39类似，它支持一个可选的密码（Electrum称之为种子扩展），并使用相同的键拉伸函数。

Aezeed

在LND钱包中使用的这种基于单词的恢复码比BIP39有所改进。它包括两个版本号：一个是内部版本号，消除了升级钱包应用程序（如Electrum v2的版本号）时的一些问题；另一个版本号是外部的，可以递增以改变恢复码的底层加密属性。它还在恢复码中包含了一个钱包生日，即用户创建钱包数据库的日期的参考。这允许恢复过程找到与钱包关联的所有资金，而无需扫描整个区块链，这对于注重隐私的轻量级客户端特别有用。它支持更改密码或更改恢复码的其他方面，而无需将资金转移到新的种子——用户只需备份一个新的恢复码。与Electrum v2相比的一个缺点是，与BIP39一样，它取决于备份和恢复软件支持相同的单词列表。

Muun

Muun钱包使用的是非单词代码，该钱包默认要求多个密钥签名来进行交易，必须与其他额外信息一起使用（Muun目前提供在PDF文件中）。这个恢复码与种子无关，而是用于解密PDF中包含的私钥。虽然与BIP39、Electrum v2和Aezeed恢复码相比，这种方法相对麻烦，但它支持新技术和标准，这些技术和标准在新钱包中变得越来越普遍，如闪电网络（LN）支持、输出脚本描述符和小脚本。

SLIP39

SLIP39是BIP39的一个继任者，由一些相同的作者编写，它允许使用多个恢复码来分发单个种子，这些恢复码可以存储在不同的地方（或由不同的人保存）。当您创建恢复码时，可以指定需要多少个来恢复种子。例如，您创建了五个恢复码，但只需要其中三个来恢复种子。SLIP39支持可选的密码，依赖于一个全局的单词列表，并且不直接提供版本控制。

一种类似于SLIP39的新的分发恢复码系统在书写过程中被提出。Codex32允许只使用打印说明书、剪刀、精密刀、黄铜卡扣和笔即可创建和验证恢复码，再加上隐私和一些空闲时间。另外，那些信任计算机的人可以使用数字设备上的软件立即创建恢复码。您可以创建多达31个恢复码，并将它们存储在不同的地方，指定需要多少个恢复码才能恢复种子。作为一个新的提议，Codex32的详细信息在本书出版前可能会发生重大变化，因此我们鼓励对分布式恢复码感兴趣的读者调查其当前状态。

> **恢复码密码**
>
> BIP39、Electrum v2、Aezeed 和 SLIP39 方案都可以与可选的口令一起使用。如果您唯一保存口令的地方是您的记忆中，那么它具有与记忆恢复码相同的优点和缺点。然而，还有一组针对口令在恢复码中使用方式的特定权衡。
>
> 这三种方案（BIP39、Electrum v2 和 SLIP39）在用于防止数据输入错误的校验和中不包括可选的口令。每个口令（包括不使用口令）都将产生一个用于 BIP32 密钥树的种子，但它们不会是相同的树。不同的口令将产生不同的密钥。这可能是积极的或消极的，这取决于您的观点：
>
> • 积极的一面是，如果有人获取了您的恢复码（但没有您的口令），他们将看到一个有效的 BIP32 密钥树。如果您为这种可能性做好了准备，并向非口令树发送了一些比特币，那么他们将窃取那笔钱。尽管有些比特币被盗通常是一件坏事，但它也可以提醒您恢复码已被泄露，从而让您调查并采取纠正措施。为相同的恢复码创建多个口令，所有这些口令看起来都有效，这是一种合理的否认的类型。
>
> • 消极的一面是，如果您被迫向攻击者提供恢复码（带有或不带有口令），但它没有产生他们期望的比特币数量，他们可能会继续试图强迫您，直到您给出另一个口令，以访问更多的比特币。设计合理的否认意味着没有办法向攻击者证明您已经泄露了所有信息，因此即使在您给出所有比特币后，他们可能会继续试图强迫您。
>
> • 另一个负面因素是错误检测的减少。如果在从备份恢复时输入了略有错误的口令，您的钱包将无法警告您有关错误。如果您期望有余额，当您的钱包应用程序为重新生成的密钥树显示零余额时，您会知道出现了问题。然而，新手用户可能会认为他们的钱永远丢失了，并做出愚蠢的事情，比如放弃并丢掉他们的恢复码。或者，如果您实际上期望零余额，那么在您犯错后的几年里，您可能会继续使用钱包应用程序，直到下次使用正确的口令进行恢复并看到零余额。除非您能够找出先前犯下的拼写错误，否则您的资金将不复存在。
>
> 与其他方案不同，Aezeed 种子加密方案对其可选口令进行身份验证，并在提供错误值时返回错误。这消除了合理的否认，增加了错误检测，并使得可以证明口令已被泄露。
>
> 许多用户和开发人员对哪种方法更好存在不同意见，有些人坚决支持合理的否认，而其他人则更喜欢增加安全性，以便新手用户和受压迫者。我们认为，只要恢复码继续被广泛使用，这种辩论就会持续下去。

备份非密钥数据

钱包数据库中最重要的数据是其私钥。如果您无法访问私钥，您将无法花费比特币。确定性密钥派生和恢复码为备份和恢复您的密钥及其控制的比特币提供了一个相当健壮的解决方案。然而，重要的是要考虑到许多钱包数据库存储的不仅仅是密钥，它们还存储了关于每笔交易的用户提供的信息。

例如，当 Bob 在向 Alice 发送发票的一部分时创建一个新地址时，他会向他生成的地址添加一个标签，以便他可以区分她的付款与他收到的其他付款。当 Alice 支付 Bob 的地址时，她也会将交易标记为支付给 Bob 的原因相同。一些钱包还会向交易中添加其他有用的信息，例如当前的汇率，在某些司法管辖区计算税收时可能很有用。这些标签完全存储在它们自己的钱包中，而不与网络共享，保护了它们的隐私，并将不必要的个人数据排除在区块链之外。请参阅表 5-1 中的示例。

表 5-1. Alice 的每笔交易历史及其标记

日期	标签	BTC
2023-01-01	Bought bitcoins from Joe	+0.00100
2023-01-02	Paid Bob for podcast	−0.00075

\ 然而，由于地址和交易标签仅存储在每个用户的钱包数据库中，并且因为它们不是确定性的，所以它们不能仅通过使用恢复码来恢复。如果唯一的恢复方式是基于种子的，那么用户将只能看到一个近似的交易时间和比特币金额的列表。这可能会让人很难弄清楚过去如何使用自己的钱。想象一下回顾一年前的银行或信用卡对账单，上面列出了每笔交易的日期和金额，但“描述”字段为空白。

钱包应该为用户提供一种方便的方式来备份标签数据。这似乎是显而易见的，但有许多广泛使用的钱包应用程序可以轻松创建和使用恢复码，但却没有提供备份或恢复标签数据的方式。

此外，钱包应用程序提供一种标准化格式导出标签可能会很有用，以便它们可以在其他应用程序（例如会计软件）中使用。该格式的标准提议在BIP329中。

钱包应用程序实现了超出基本比特币支持的其他协议，可能也需要或希望存储其他数据。例如，截至2023年，越来越多的应用程序已经添加了支持通过闪电网络（LN）发送和接收交易的功能。尽管闪电网络协议提供了一种在数据丢失时恢复资金的方法，称为静态通道备份，但它无法保证结果。如果您的钱包连接的节点意识到您丢失了数据，它可能会窃取您的比特币。如果它在您丢失数据库的同时丢失了其钱包数据库，并且您两者都没有充足的备份，那么您两者都将损失资金。\ \ 再次强调，这意味着用户和钱包应用程序需要做的不仅仅是备份恢复码。

一些钱包应用程序实现的解决方案是频繁地自动创建完整的备份，将其钱包数据库加密，加密密钥是从其种子派生的。比特币密钥必须是无法猜测的，而现代加密算法被认为是非常安全的，因此除了能够生成种子的人之外，没有人应该能够打开加密备份。这使得在不受信任的计算机上存储备份成为可能，例如云托管服务甚至是随机网络节点。

稍后，如果原始的钱包数据库丢失了，用户可以将其恢复码输入到钱包应用程序中以恢复其种子。然后，应用程序可以检索最新的备份文件，重新生成加密密钥，解密备份，并恢复用户的所有标签和其他协议数据。

备份密钥派生路径

在BIP32密钥树中，大约有40亿个一级密钥；每个这些密钥都可以有自己的40亿个子级，而每个子级又可以有自己的40亿个子级，依此类推。钱包应用程序不可能生成BIP32树中的所有可能密钥的一小部分，这意味着从数据丢失中恢复需要知道的不仅仅是恢复码、获取种子的算法（例如，BIP39）和确定性密钥派生算法（例如，BIP32） - 还需要知道您的钱包应用程序用于生成其分发的特定密钥的密钥树中的路径。

已经采用了两种解决方案来解决这个问题。第一种是使用标准路径。每当与钱包应用程序可能想要生成的地址相关的变化发生时，某人都会创建一个BIP，定义要使用的密钥派生路径。例如，BIP44定义了m/44'/0'/0'作为用于P2PKH脚本（传统地址）中的密钥的路径。实现此标准的钱包应用程序在首次启动时和从恢复码进行恢复后都使用该路径中的密钥。我们称这种解决方案为隐式路径。由BIP定义的几个流行的隐式路径如表5-2所示。

表5-2. 由各种BIP定义的隐式脚本路径

标准	脚本	BIP32路径
BIP44	P2PKH	m/44'/0'/0'
BIP49	Nested P2WPKH	m/49'/1'/0'
BIP84	P2WPKH	m/84'/0'/0'
BIP86	P2TR Single-key	m/86'/0'/0'

第二种解决方案是将路径信息与恢复码一起备份，明确指出哪些路径与哪些脚本一起使用。我们称之为显式路径。

隐式路径的优点在于用户无需记录他们使用的路径。如果用户将恢复码输入到他们之前使用的相同版本或更高版本的钱包应用程序中，它将自动为之前使用的相同路径重新生成密钥。

隐式脚本的缺点在于其不灵活性。当输入恢复码时，钱包应用程序必须为其支持的每个路径生成密钥，并且必须扫描区块链以查找涉及这些密钥的交易，否则可能找不到所有用户的交易。对于支持许多具有各自路径的功能的钱包来说，这种做法是浪费的，如果用户只尝试了其中的一些功能。

对于不包含版本号的隐式路径恢复码，如BIP39和SLIP39，新版本的钱包应用程序如果不再支持旧路径，在恢复过程中无法警告用户可能无法找到部分资金。反过来，如果用户将恢复码输入到旧软件中，则无法找到用户可能已经收到资金的新路径。包含版本信息的恢复码，如Electrum v2和Aezeed，可以检测到用户输入的是旧版本还是新版本的恢复码，并引导用户到相应的资源。

隐式路径的最终结果是它们只能包含通用的信息（例如标准化路径）或从种子派生的信息（例如密钥）。某些用户特定的重要非确定性信息无法使用恢复码进行恢复。例如，Alice、Bob和Carol收到的资金只能通过其中两个人的签名来花费。尽管Alice只需要Bob或Carol的签名来花费，但她需要他们两个人的公钥才能在区块链上找到他们的共同资金。这意味着他们每个人都必须备份所有三个人的公钥。随着多重签名和其他高级脚本在比特币上变得更加普遍，隐式路径的不灵活性变得更加重要。

显式路径的优点在于它们可以准确描述应该与何种脚本一起使用的密钥。无需支持过时的脚本，也不会出现向后或向前兼容性的问题，而且任何额外的信息（例如其他用户的公钥）都可以直接包含在内。它们的缺点是它们需要用户将额外信息与恢复码一起备份。这些额外的信息通常不会危及用户的安全性，因此不需要像恢复码那样多的保护，尽管它可能会减少用户的隐私，并且需要一些保护。

\ 几乎所有截至目前为止使用显式路径的钱包应用程序都使用了输出脚本描述符标准（简称描述符），该标准在BIP380、BIP381、BIP382、BIP383、BIP384、BIP385、BIP386和BIP389中指定。描述符描述了一个脚本以及与之一起使用的密钥（或密钥路径）。Bitcoin Core文档中的一些示例描述符如下所示（省略部分）：

Table 5-3. Bitcoin Core文档中的示例描述符（部分省略）

描述符	解释
pkh(02c6…9ee5)	提供的公钥的P2PKH（支付至公钥哈希）脚本
sh(multi(2,022f… 2a01,03ac… ccbe))	需要两个签名与这两个密钥相对应的P2SH（支付至脚本哈希）多重签名
pkh([d34db33f/44'/0'/0']xpub6ERA…RcEL/1/*)	BIP32 d34db33f的扩展公钥（xpub）在路径M/44'/0'/0'上的P2PKH脚本，使用该xpub的M/1/*路径上的密钥

长期以来，专为单签名脚本设计的钱包应用程序倾向于使用隐式路径。而为多重签名或其他高级脚本设计的钱包应用程序则越来越多地采用支持描述符的显式路径。同时支持两种方式的应用程序通常会遵循隐式路径的标准，并提供描述符。

钱包技术栈的详细介绍

简介

现代钱包的开发者可以从各种不同的技术中选择，来帮助用户创建和使用备份，而且每年都会出现新的解决方案。与其详细介绍本章前面描述的每个选项，我们将把重点放在截至2023年初被认为是最广泛使用的技术堆栈上：

- BIP39恢复代码
- BIP32 HD密钥派生
- BIP44风格的隐式路径

所有这些标准早在2014年或之前就已经存在了，你可以轻松找到更多关于它们的使用资源。然而，如果你感到有兴趣，我们鼓励你调查一些可能提供额外功能或安全性的更现代的标准。

BIP39恢复码

BIP39恢复代码是表示（编码）用作种子的随机数的单词序列，用于派生确定性钱包。这个单词序列足以重新创建种子，从而重新创建所有派生密钥。实现了具有BIP39恢复代码的确定性钱包的钱包应用程序在首次创建钱包时会向用户显示一个由12到24个单词组成的序列。这个单词序列是钱包的备份，可以用来在相同或任何兼容的钱包应用程序中恢复和重新创建所有密钥。恢复代码使用户更容易备份，因为它们易于阅读和正确转录。

> 恢复代码经常与“脑钱包”混淆。它们并不相同。主要区别在于脑钱包由用户选择的单词组成，而恢复代码是由钱包随机创建并呈现给用户的。这个重要区别使得恢复代码更加安全，因为人类是非常不好的随机源。

\ 请注意，BIP39是恢复代码标准的一种实现。BIP39由Trezor硬件钱包背后的公司提出，并与许多其他钱包应用兼容，尽管肯定不是所有钱包都兼容。

BIP39定义了恢复代码和种子的创建过程，我们在这里分成两部分描述，步骤1到6在“生成恢复代码”中展示，步骤7到9在“从恢复代码到种子”中展示。

生成恢复码

恢复代码是由钱包应用程序使用BIP39中定义的标准化流程自动生成的。钱包从熵源开始，添加一个校验和，然后将熵映射到一个单词列表中：

1. 创建一个128到256位的随机序列（熵）。
2. 通过取其SHA256哈希的前（熵长度/32）位来创建随机序列的校验和。
3. 将校验和添加到随机序列的末尾。
4. 将结果分割成11位长度的段。
5. 将每个11位值映射到预定义的2,048个单词字典中的一个单词。
6. 恢复代码是单词序列。

图5-4显示了熵是如何用来生成BIP39恢复代码的。

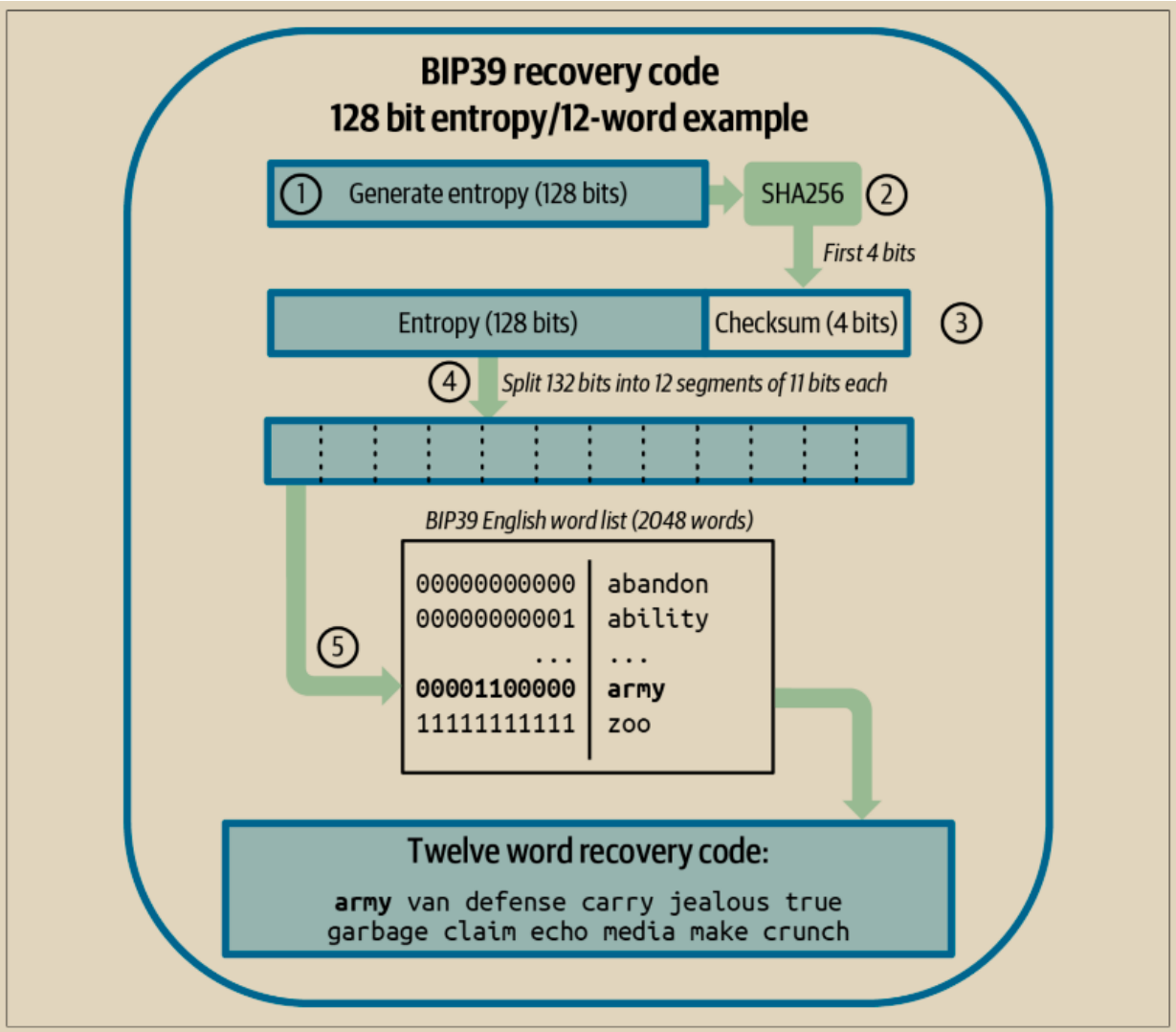


图 5-4. 生成熵并将其编码为恢复代码

表5-4显示了熵数据的大小与恢复代码长度（以字为单位）之间的关系。

表5-4. BIP39：熵和字长度

熵(比特数)	校验码(比特数)	熵+校验码(比特数)	恢复码单词数
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

从恢复码到种子

恢复码代表长度为 128 到 256 位的熵。然后使用熵通过密钥拉伸函数 PBKDF2 派生一个更长的（512 位）种子。生成的种子然后用于构建确定性钱包并派生其密钥。

密钥拉伸函数接受两个参数：熵和盐。盐在密钥拉伸函数中的目的是使构建查找表以进行暴力攻击变得困难。在BIP39 标准中，盐还有另一个目的——它允许引入一个密码短语，作为额外的安全因素来保护种子，我们将在“BIP39中的可选密码短语”（第 107 页）中更详细地描述。

> 密钥拉伸函数，具有 2,048 轮哈希，使使用软件进行暴力攻击恢复码变得稍微困难一些。特殊用途硬件没有受到明显影响。对于需要猜测用户完整恢复码的攻击者来说，代码长度（至少 128 位）提供了足够的安全性。但是对于攻击者可能了解用户代码的一小部分的情况，密钥拉伸通过减慢攻击者检查不同恢复码组合的速度，增加了一些安全性。即使在将近十年前首次发布时，BIP39 的参数也被现代标准认为是脆弱的，尽管这可能是为了与低功耗 CPU 的硬件签名设备兼容而设计的后果。一些替代方案使用更强大的密钥拉伸参数，例如 Aezeed 的 32,768 轮哈希使用更复杂的 Scrypt 算法，尽管它们可能在硬件签名设备上运行时不太方便

\ 步骤7至9描述的过程是从前文“生成恢复代码”中描述的过程继续的：

7. PBKDF2密钥拉伸函数的第一个参数是步骤6生成的熵。

1. PBKDF2密钥拉伸函数的第二个参数是盐。盐由字符串常量“助记词”与可选的用户提供的密码字符串连接而成。
2. PBKDF2使用HMAC-SHA512算法对恢复代码和盐参数进行2048轮哈希拉伸，生成一个512位的值作为其最终输出。这个512位的值就是种子。

图5-5展示了如何使用恢复代码生成种子。

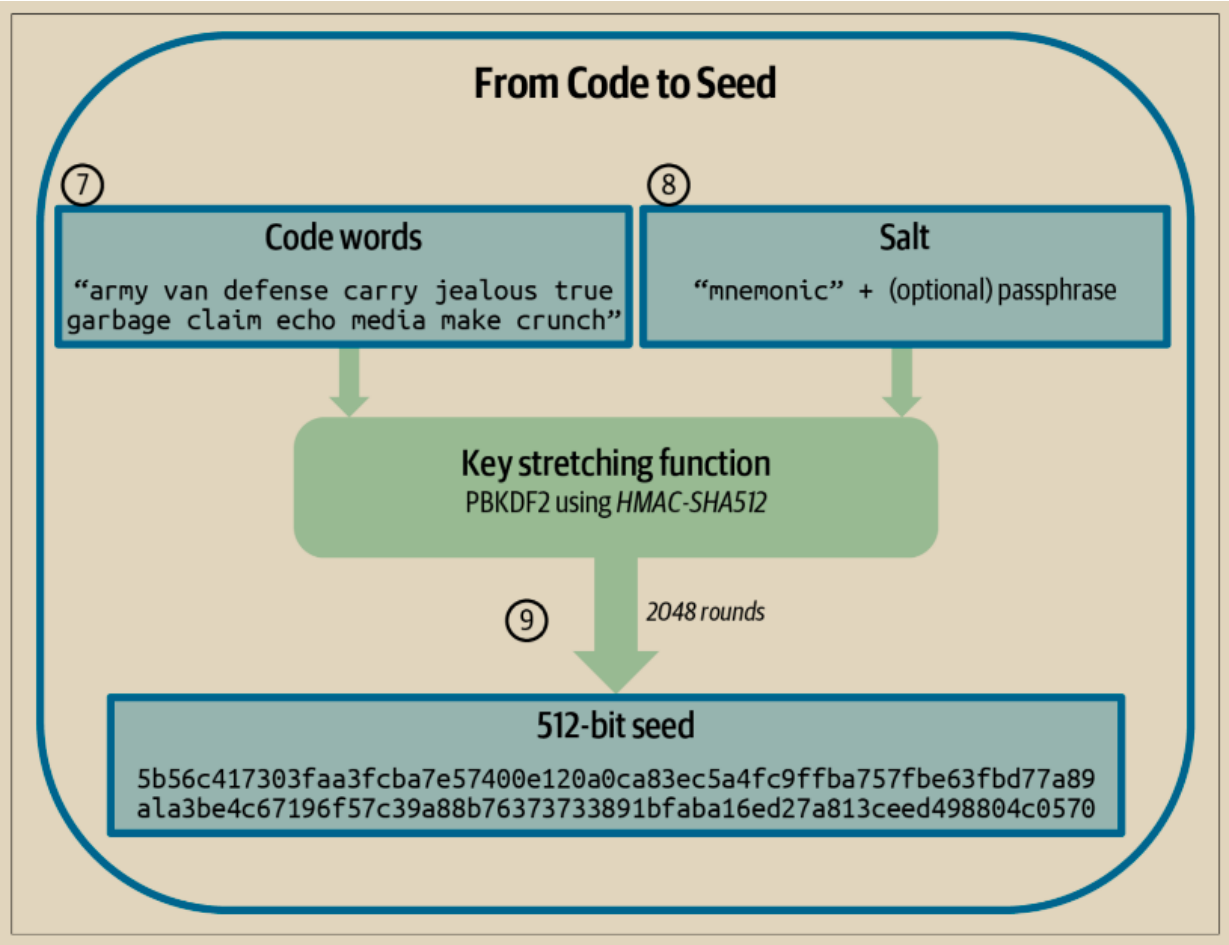


图 5-5. 从恢复码到种子

表5-5、5-6和5-7显示了一些恢复代码及其生成的种子的示例。

表5-5. 128位熵BIP39恢复代码，无口令，生成的种子

类型	示例
熵输入(128比特)	0c1e24e5917779d297e14d45f14e1a1a
恢复码(12个单词)	army van defense carry jealous true garbage claim echo media make crunch
密码	无
种子(512比特)	5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39a88b76373733891bfaba16ed27a813ceed498804c0570

表 5-6. 128位熵BIP39恢复代码，带有口令，生成的种子

类型	示例
熵输入(128比特)	0c1e24e5917779d297e14d45f14e1a1a
恢复码(12个单词)	army van defense carry jealous true garbage claim echo media make crunch
密码	SuperDuperSecret
种子(512比特)	3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0715861dc8a18358f80b79d49acf64142ae57037d1d54

表 5-7. 256位熵BIP39恢复代码，无口令，生成的种子

类型	示例
熵输入(256比特)	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
恢复码(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
密码	无
种子(512比特)	3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e55f1e0deaa082df8d487381379df848a6ad7e98798404

> **需要多少熵?**
>
> BIP32允许种子从128位到512位。BIP39接受从128位到256位的熵；Electrum v2接受132位熵；Aezeed接受128位熵；SLIP39接受128位或256位熵。这些数字的变化使得确定安全所需的熵量变得不清晰。我们将尝试澄清这一点。
>
> BIP32扩展私钥由256位密钥和256位链代码组成，总共512位。这意味着最多有2512个不同的可能扩展私钥。如果您的熵超过512位，您仍将获得包含512位熵的扩展私钥，因此即使我们提到的任何标准允许这样做，也没有使用超过512位的必要。
>
> 然而，即使有2512个不同的扩展私钥，也只有(稍少于)2256个常规私钥——而实际上正是这些私钥保护了您的比特币。这意味着，如果您的种子使用了超过256位的熵，您仍然会得到仅包含256位熵的私钥。可能会有未来与比特币相关的协议，在这些协议中，扩展密钥中的额外熵提供额外的安全性，但目前并非如此。
>
> 比特币公钥的安全强度为128位。使用经典计算机（截至目前，这是唯一可以进行实际攻击的计算机）的攻击者需要对比特币的椭圆曲线执行约2^128次操作，以找到另一个用户的公钥的私钥。128位的安全强度意味着在使用128位熵以上的熵时没有明显的好处（尽管您需要确保您生成的私钥在整个2^256范围内均匀选择）。
>
> 更大熵的一个额外好处是：如果攻击者看到了恢复代码的固定百分比（但不是整个代码），那么熵越大，他们就越难以弄清楚他们没有看到的代码的部分。例如，如果攻击者看到了128位代码的一半（64位），则他们可能会能够破解剩余的64位。如果他们看到了256位代码的一半（128位），则他们不太可能能够破解另一半。我们不建议依赖这种防御——要么将您的恢复代码保管得非常安全，要么使用像SLIP39这样的方法，让您将恢复代码分布到多个位置，而不依赖于任何单个代码的安全性。
>
> 截至2023年，大多数现代钱包为其恢复代码生成128位熵（或接近128位的值，例如Electrum v2的132位）。

BIP39可选口令

\ BIP39标准允许在生成种子的派生过程中使用可选的口令。如果没有使用口令，恢复代码将与由常量字符串“mnemonic”组成的盐一起进行拉伸，从而从给定的任何恢复代码生成特定的512位种子。如果使用口令，则拉伸函数将从相同的恢复代码生成不同的种子。事实上，对于单个恢复代码，每个可能的口令都会导致不同的种子。基本上，不存在“错误”的口令。所有口令都是有效的，并且它们都导致不同的种子，形成了一个庞大的可能未初始化的钱包集合。可能的钱包集合如此之大（2^512），以至于没有实际可能性对其进行穷举或意外猜测使用中的钱包。

> 在BIP39中，不存在“错误”的口令。每个口令都会导致某个钱包，除非该钱包之前已被使用，否则该钱包将为空。

\ 可选的口令创建了两个重要的功能：

- 第二因素（记忆的东西），使得仅凭恢复码无法单独使用，从而保护恢复码免受随意窃贼的威胁。为了防止技术娴熟的窃贼，您需要使用一个非常强大的口令。
- 可以提供一种合理的否认或“胁迫钱包”，其中选定的口令导致一个只包含少量资金的钱包，用于转移攻击者对“真实”钱包的注意力，而“真实”钱包包含了大部分资金。

需要注意的是，使用口令也会引入丢失的风险：

- 如果钱包所有者丧失能力或去世，而没有其他人知道口令，那么种子就无法使用，钱包中存储的所有资金都将永远丢失。
- 相反，如果所有者将口令备份在与种子相同的地方，那么就失去了第二因素的作用。

虽然口令非常有用，但应仅与备份和恢复的精心计划流程结合使用，考虑到可能存活的所有者并允许其家人恢复加密货币资产的可能性。

从种子创建HD钱包

HD钱包是从单个根种子创建的，该种子是一个128位、256位或512位的随机数。通常情况下，这个种子是通过前面部分详细介绍的恢复码生成或解密的。

HD钱包中的每个密钥都是从这个根种子确定性地派生出来的，这使得在任何兼容的HD钱包中都可以从该种子重新创建整个HD钱包。这使得通过仅转移从根种子派生的恢复码，即可轻松备份、恢复、导出和导入包含数千甚至数百万个密钥的HD钱包。创建HD钱包的主密钥和主链码的过程如图5-6所示。\

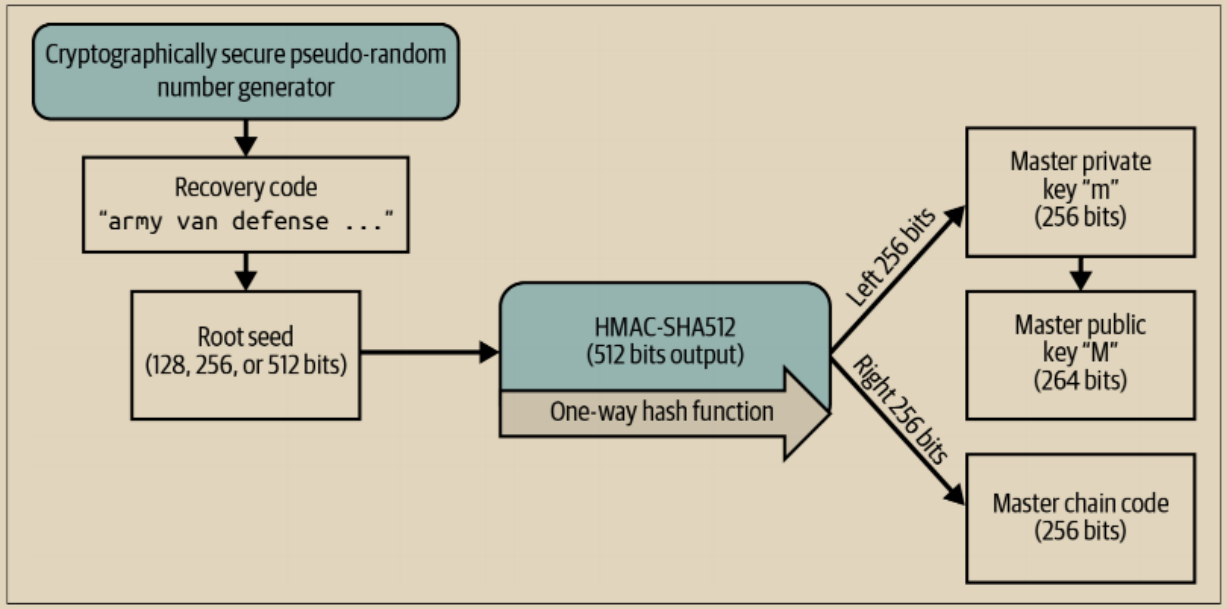


图 5-6. 从根种子创建主密钥和链码

根种子被输入到HMAC-SHA512算法中，生成的哈希用于创建一个主私钥（m）和一个主链码（c）。

主私钥（m）然后使用正常的椭圆曲线乘法过程 m × G 生成相应的主公钥（M），就像我们在“公钥”中看到的一样。

主链码（c）用于在从父密钥创建子密钥的函数中引入熵，正如我们将在下一节中看到的那样。

子私钥派生

\ HD钱包使用子密钥派生（CKD）函数从父密钥派生子密钥。

子密钥派生函数基于一种单向哈希函数，结合了：

- 父私钥或公钥（未压缩密钥）
- 一个称为链码的种子（256位）
- 一个索引号（32位）

链码用于在过程中引入确定性随机数据，因此仅知道索引和一个子密钥不足以推导出其他子密钥。仅知道一个子密钥也不足以找到它的兄弟，除非你也有链码。初始链码种子（在树的根部）由种子生成，而后续的子链码从每个父链码派生而来。

这三个项目（父密钥、链码和索引）被组合并进行哈希处理以生成子密钥，具体如下所示。

父公钥、链码和索引号被组合并使用HMAC-SHA512算法进行哈希处理，生成512位哈希。这512位哈希被分成两个256位的半部分。哈希输出的右半部分256位成为子的链码。哈希的左半部分256位被添加到父私钥上，生成子私钥。在图5-7中，我们看到了这个过程的示例，将索引设置为0以生成父的“零”（按索引排序的第一个）子密钥。

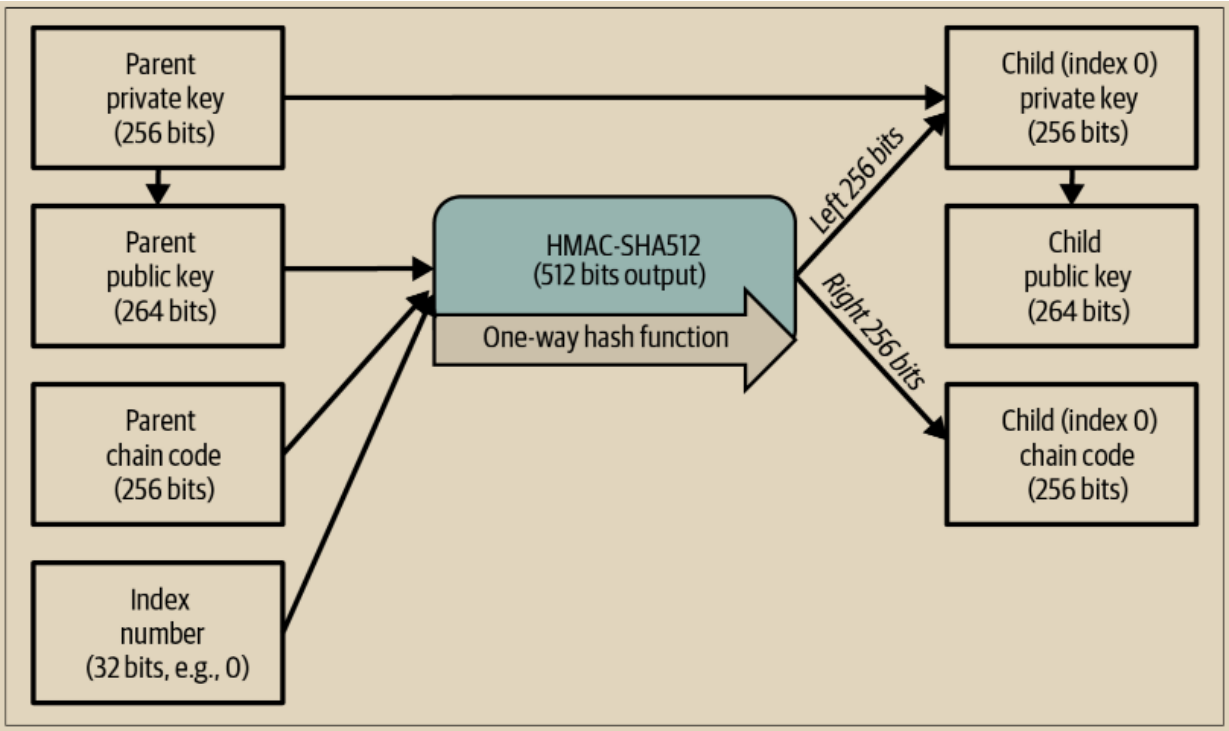


图 5-7. 扩展父私钥以创建子私钥

更改索引使我们能够扩展父节点并创建序列中的其他子节点（例如，Child 0、Child 1、Child 2 等）。每个父密钥可以拥有 2,147,483,647（2^31）个子节点（2^31 是整个 2^32 范围的一半，因为另一半保留用于我们稍后将在本章中讨论的一种特殊派生类型）。

重复此过程一级下来，每个子节点可以进一步成为一个父节点，并创建自己的子节点，以无限的代数。

使用派生子私钥

子私钥与不确定性（随机）密钥无法区分。 因为派生函数是一个单向函数，所以子密钥不能用于找到父密钥。子密钥也不能用于找到任何兄弟节点。如果你有第 n 个子节点，你无法找到它的兄弟节点，比如第 n-1 个子节点或第 n+1 个子节点，或者序列中的其他任何子节点。只有父密钥和链码才能派生出所有的子节点。如果没有子链码，子密钥也不能用于派生任何孙节点。您需要子私钥和子链码才能启动一个新的分支并派生出孙子节点。

那么子私钥单独能用来做什么呢？它可以用来生成公钥和比特币地址。然后，它可以用来签署交易以花费支付给该地址的任何内容。

> 子私钥、对应的公钥和比特币地址与随机创建的密钥和地址完全相同。它们属于一个序列的事实在创建它们的 HD 钱包功能之外是不可见的。一旦创建，它们的操作方式与“普通”密钥完全相同。

扩展密钥

正如我们之前所见，密钥派生函数可以根据三个输入在树的任何级别上创建子级，这三个输入是：一个密钥、一个链码和所需子级的索引。这两个基本要素是密钥和链码，它们的组合称为扩展密钥。术语“扩展密钥”也可以被视为“可扩展密钥”，因为这样的密钥可以用来派生子级。

扩展密钥简单地存储和表示为密钥和链码的串联。有两种类型的扩展密钥。扩展私钥是私钥和链码的组合，可用于派生子私钥（以及从中派生的子公钥）。扩展公钥是公钥和链码，可用于创建子公钥（仅公钥），如第59页的“公钥”中描述的。

将扩展密钥视为 HD 钱包树结构中分支的根。有了分支的根，您可以派生出分支的其余部分。扩展私钥可以创建完整的分支，而扩展公钥只能创建公钥分支。

扩展密钥使用 base58check 进行编码，以便在不同的 BIP32 兼容钱包之间轻松导出和导入。扩展密钥的 base58check 编码使用特殊的版本号，在编码为 base58 字符时会生成前缀“xprv”和“xpub”，以使其易于识别。因为扩展密钥包含的字节比常规地址多得多，所以它也比我们之前见过的其他 base58check 编码字符串长得多。

这里是一个扩展私钥的 base58check 编码示例：

xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CA
WrUE9i6GoNMKUga5biW6Hx4tws2six3b9c

这是相应的扩展公钥，使用base58check编码：

xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBP
LrtJunSDMstweyLXhRgPxdp14sk9tJPW9

公钥子派生

正如前面提到的，HD钱包的一个非常有用的特性是能够从公钥父级键派生公钥子级键，而无需私钥。这使我们有两种方式来派生子公钥：要么从子私钥，要么直接从父公钥。

因此，扩展公钥可以用来派生该HD钱包结构分支中的所有公钥（仅限公钥）。

这种快捷方式可用于创建仅公钥的部署，其中服务器或应用程序具有扩展公钥的副本，但完全没有私钥。这种部署方式可以产生无限数量的公钥和比特币地址，但不能花费发送到这些地址的任何资金。同时，在另一个更安全的服务器上，扩展私钥可以派生所有相应的私钥来签署交易并花费资金。

这种解决方案的一个常见应用是在提供电子商务应用程序的Web服务器上安装扩展公钥。Web服务器可以使用公钥派生函数为每笔交易（例如，为客户购物车）创建一个新的比特币地址。Web服务器将不具有可能被盗的私钥。没有HD钱包，唯一的方法是在一个单独的安全服务器上生成数千个比特币地址，然后预加载它们到电子商务服务器上。这种方法繁琐，并且需要不断维护，以确保电子商务服务器不会“用完”密钥。

> **注意间隙**
>
> 扩展公钥可以生成大约40亿个直接子密钥，远远超过任何商店或应用程序可能需要的数量。然而，一个钱包应用程序要生成所有40亿个密钥并扫描区块链以查找涉及这些密钥的交易，需要的时间是不合理的。因此，大多数钱包一次只生成几个密钥，扫描涉及这些密钥的支付，并在使用之前的密钥时按顺序生成更多的密钥。例如，Alice的钱包生成100个密钥。当它看到对第一个密钥的支付时，它会生成第101个密钥。 有时，一个钱包应用程序会向某人分发一个密钥，后来决定不支付，从而在密钥链中创建一个间隙。只要钱包在间隙之后已经生成了密钥，以便找到后续的支付并继续生成更多的密钥，这是可以接受的。连续未收到支付的最大未使用密钥数量而不会引起问题称为间隙限制。 当一个钱包应用程序已经分发了所有的密钥直到它的间隙限制，而且这些密钥都没有收到支付时，它有三个选项来处理未来请求的新密钥：
>
> 1. 它可以拒绝请求，阻止其接收任何进一步的支付。这显然是一个不受欢迎的选项，尽管它是最简单的实现方式。
> 2. 它可以生成超出间隙限制的新密钥。这样确保每个请求支付的人都获得一个唯一的密钥，防止地址重用并提高隐私。然而，如果需要从恢复代码中恢复钱包，或者如果钱包所有者正在使用加载了相同扩展公钥的其他软件，那么这些其他钱包将看不到扩展间隙之后收到的任何支付。
> 3. 它可以分发先前分发的密钥，确保平稳恢复，但可能降低钱包所有者和与其交易的人的隐私。
>
> 在线商户的开源生产系统，例如BTCPay Server，试图通过使用非常大的间隙限制和限制生成发票的速率来回避这个问题。其他解决方案已经被提出，例如在支付者收到实际交易的新地址之前，要求支付者的钱包构建（但不广播）一笔向可能重用的地址支付的交易。然而，截至目前，这些其他解决方案尚未在生产中使用。

另一个常见的应用是用于冷存储或硬件签名设备。在这种情况下，扩展私钥可以存储在纸钱包或硬件设备上，而扩展公钥可以在线上保存。用户可以随意创建“接收”地址，而私钥则安全地存储在离线状态下。要花费资金，用户可以在离线软件钱包应用程序或硬件签名设备上使用扩展私钥。图5-8说明了将父公钥扩展以派生子公钥的机制。

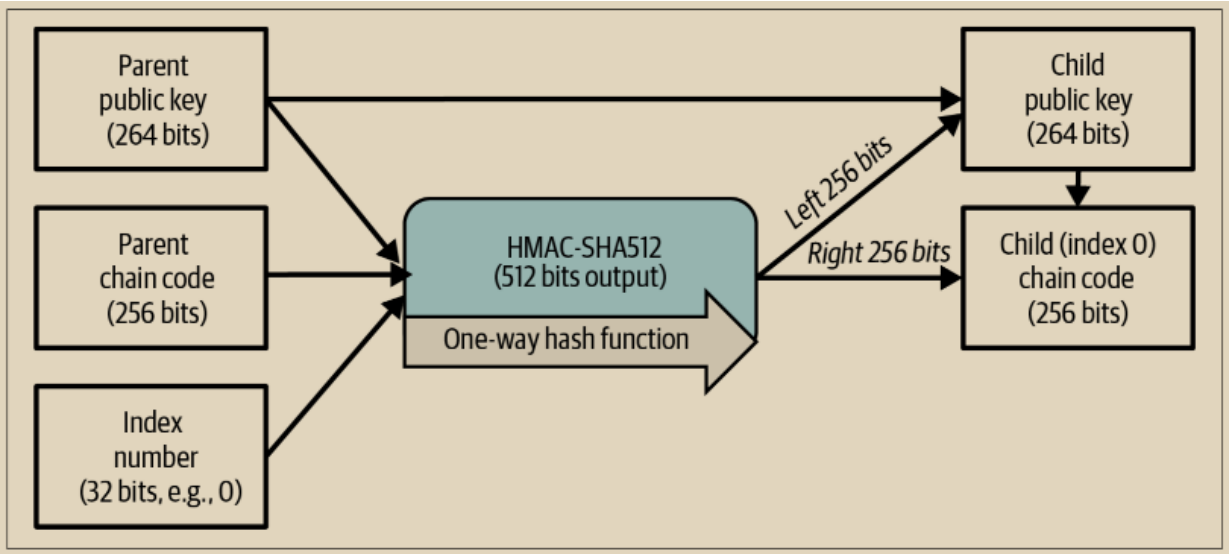


图 5-8. 扩展父公钥以创建子公钥

在网络商店使用扩展公钥

让我们通过看看加布里埃尔的网络商店来了解HD钱包是如何使用的。

加布里埃尔最初是抱着兴趣的心态建立了他的网络商店，基于一个简单的托管的WordPress页面。他的商店非常基础，只有几个页面和一个带有单一比特币地址的订单表单。

加布里埃尔使用他的常规钱包生成的第一个比特币地址作为他的商店的主要比特币地址。顾客会使用表单提交订单并向加布里埃尔发布的比特币地址付款，触发一封包含订单详情的电子邮件，供加布里埃尔处理。尽管每周只有几个订单，但这个系统足够满足需求，尽管它削弱了加布里埃尔、他的客户和他支付的人的隐私。

然而，这个小型网络商店变得相当成功，吸引了很多来自当地社区的订单。很快，加布里埃尔就不堪重负。由于所有订单支付同一个地址，因此在正确匹配订单和交易时变得困难，特别是当相同金额的多个订单相继而来时。

典型比特币交易接收者选择的唯一元数据是金额和付款地址。没有主题或消息字段可用于保存唯一标识发票号。

通过HD钱包，加布里埃尔的问题得到了更好的解决方案，因为他可以在不知道私钥的情况下生成公共子密钥。加布里埃尔可以在他的网站上加载一个扩展公钥（xpub），用于为每个客户订单派生一个唯一地址。这个唯一地址立即提高了隐私，还为每个订单提供了一个可以用于跟踪已支付发票的唯一标识。

使用HD钱包允许加布里埃尔从他的个人钱包应用程序中支出资金，但加载在网站上的xpub只能生成地址并接收资金。这是HD钱包的一个很好的安全功能。加布里埃尔的网站不包含任何私钥，因此对其的任何黑客攻击都只能窃取加布里埃尔未来可能收到的资金，而不是他过去收到的任何资金。

为了从他的Trezor硬件签名设备中导出xpub，加布里埃尔使用基于Web的Trezor钱包应用程序。要导出公钥，Trezor设备必须插入。请注意，大多数硬件签名设备永远不会导出私钥——它们始终保留在设备上。

加布里埃尔将xpub复制到他的网店的比特币支付处理软件中，比如广泛使用的开源BTCPay服务器。

强化子密钥衍生

从xpub推导出公钥分支的能力非常有用，但也带来了潜在的风险。访问xpub并不会给予对子私钥的访问权限。然而，因为xpub包含了链代码，如果某个子私钥已知或某种方式泄露，它可以与链代码一起用于推导出所有其他子私钥。一个泄露的子私钥，加上父链代码，会暴露出所有子节点的私钥。更糟糕的是，子私钥与父链代码一起可以用于推断出父私钥。

为了应对这种风险，HD钱包提供了一种另类的衍生函数，称为硬化衍生，它打破了父公钥与子链代码之间的关系。硬化衍生函数使用父私钥来推导子链代码，而不是使用父公钥。这在父子序列中创建了一个“防火墙”，其链代码不能用于危及父或同级私钥。硬化衍生函数看起来几乎与普通的子私钥衍生相同，只是父私钥被用作哈希函数的输入，而不是父公钥，如图5-9中的图表所示。

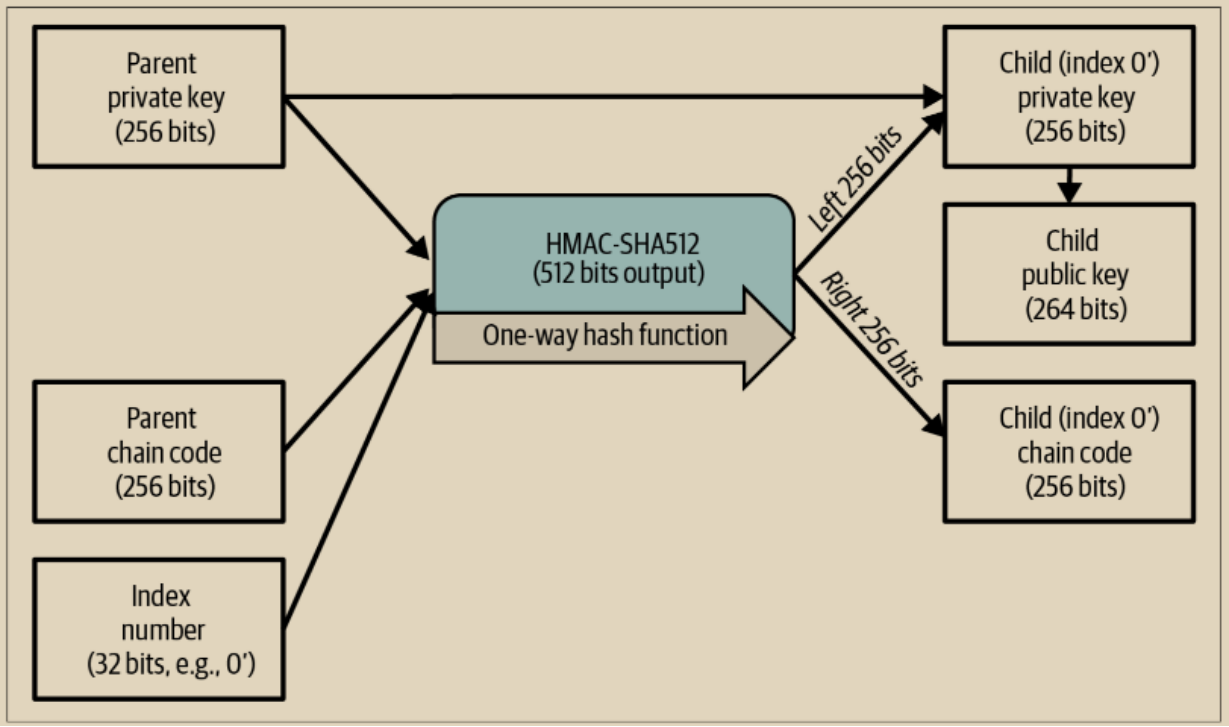


图 5-9. 子密钥的硬化衍生；省略了父公钥。

当使用硬化私有派生函数时，得到的子私钥和链码与使用常规派生函数得到的完全不同。结果的“分支”密钥可以用来生成扩展公钥，因为它们包含的链码不能被利用来揭示它们的兄弟或父母的任何私钥。因此，硬化派生用于在扩展公钥使用的级别上创建“间隙”。

简而言之，如果您想要使用xpub方便地派生公钥分支，而又不暴露于泄漏链码的风险之中，那么您应该从硬化父节点派生它，而不是从普通父节点派生。作为最佳实践，主密钥的一级子代始终通过硬化派生来派生，以防止主密钥被破坏。

普通和强化衍生的索引号

\ 派生函数中使用的索引号是一个32位整数。为了区分通过常规派生函数创建的密钥和通过硬化派生创建的密钥，该索引号被分为两个范围。

索引号介于0和2^31-1（0x0到0x7FFFFFFF）之间，仅用于常规派生。索引号介于2^31和2^32-1（0x80000000到0xFFFFFFFF）之间，仅用于硬化派生。因此，如果索引号小于2^31，则子代是常规的，而如果索引号等于或大于2^31，则子代是硬化的。

为了使索引号更易于阅读和显示，硬化子代的索引号从零开始显示，但带有一个质数符号。因此，第一个常规子代密钥显示为0，而第一个硬化子代（索引0x80000000）显示为0'。然后，在序列中，第二个硬化密钥的索引将为0x80000001，并显示为1'，依此类推。当您看到HD钱包索引i'时，意味着2^31+i。在常规ASCII文本中，质数符号被替换为单引号或字母h。对于可能在shell或其他上下文中使用文本的情况（其中单引号具有特殊含义），建议使用字母h。

HD钱包密钥标志符(路径)

\ HD钱包中的密钥使用“路径”命名约定来标识，树的每个级别之间用斜杠（/）字符分隔（见表5-8）。从主私钥派生的私钥以“m.”开头。从主公钥派生的公钥以“M.”开头。因此，主私钥的第一个子私钥是m/0。第一个子公钥是M/0。第一个子代的第二个孙子是m/0/1，依此类推。

密钥的“祖先”从右向左读取，直到达到派生它的主密钥为止。例如，标识符m/x/y/z描述的是密钥m/x/y的第z个子代，它是密钥m/x/y的第y个子代，它是密钥m/x的第x个子代，它是m的子代。

表 5-8. 以下是 HD 钱包路径的一些示例

HD路径	私钥描述
m/0	The first (0) child private key from the master private key (m)
m/0/0	The first grandchild private key from the first child (m/0)
m/0'/0	The first normal grandchild private key from the firsthardened child (m/0')
m/1/0	The first grandchild private key from the second child (m/1)
M/23/17/0/0	The first great-great-grandchild public key from the first great-grandchild from the 18th grandchild from the 24th child

HD钱包树状结构导航

在 HD 钱包树结构中导航提供了巨大的灵活性。每个父扩展密钥可以有 40 亿个子级：20 亿个普通子级和 20 亿个硬化子级。每个子级都可以有另外 40 亿个子级，依此类推。树的深度可以任意，有无限的代数。然而，尽管有如此大的灵活性，但在这个无限树中导航变得非常困难。将 HD 钱包在实现之间转移尤其困难，因为将其内部组织成分支和子分支的可能性是无穷无尽的。

两个 BIP 提出了这种复杂性的解决方案，通过创建一些关于 HD 钱包树结构的标准提出了解决方案。BIP43 提议使用第一个硬化子级索引作为特殊标识符，表示树结构的“目的”。根据 BIP43，HD 钱包应该只使用树的一级分支，通过定义其目的来使用索引号来识别树的其余部分的结构和命名空间。例如，只使用分支 m/i' 的 HD 钱包旨在表示特定目的，并且该目的由索引号 “i” 来标识。

在此规范的基础上，BIP44 提议作为 BIP43 下的 “目的” 编号 44' 的多账户结构。遵循 BIP44 结构的所有 HD 钱包都被识别为它们只使用树的一个分支：m/44'。

BIP44 指定了结构，由五个预定义的树级别组成：

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

第一级 “purpose” 总是设为 44'。第二级 “coin_type” 指定了加密货币币种的类型，允许创建多币种 HD 钱包，其中每种币种都有自己的子树位于第二级之下。比特币为 m/44'/0'，比特币测试网为 m/44'/1'。

树的第三级是 “account”，它允许用户将他们的钱包细分为不同的逻辑子账户，用于会计或组织目的。例如，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'。每个账户都是其自己子树的根。

\ 在第四级 “change” 上，HD 钱包有两个子树，一个用于创建接收地址，另一个用于创建找零地址。需要注意的是，前面的级别使用了硬化派生，而这一级别使用了普通派生。这是为了允许这一级别的树导出扩展公钥供非安全环境使用。 可用地址由 HD 钱包作为第四级的子代派生，使得树的第五级成为 “address_index”。例如，主帐户中用于付款的第三个接收地址将是 M/44'/0'/0'/0/2。下表显示了一些更多的示例。

Table 5-9. BIP44 HD 钱包结构示例

HD 钱包路径	描述
m/44'/0'/0'/0/2	主比特币账户的第三个接收公钥
M/44'/0'/3'/1/14	第四个比特币账户的第十五个找零地址公钥
m/44'/2'/0'/0/1	Litecoin主账户的第二个私钥，用于签署交易

\ 许多人关注保护比特币免受盗窃和其他攻击，但丢失比特币的主要原因之一，也许是主要原因之一，是数据丢失。如果丢失了用于花费比特币的密钥和其他必要数据，那么这些比特币将永远无法花费。没有人可以为您取回它们。在本章中，我们看了现代钱包应用程序使用的系统，帮助您防止丢失这些数据。然而，请记住，实际上要使用可用的系统来制作好的备份并定期测试它们是由您来决定的。

综合介绍

通常情况下，我们转移实物现金的方式与转移比特币的方式几乎毫无相似之处。实物现金是一种持有人凭证。Alice支付Bob的方式是将一定数量的代币，比如美元纸币，交给他。相比之下，比特币既不存在实物形式，也不以数字数据形式存在，Alice不能将比特币交给Bob，也无法通过电子邮件发送。

相反，想象一下Alice如何将对一块土地的控制权转移给Bob。她无法实际拿起土地并交给Bob。相反，存在一种记录（通常由地方政府维护），记录着Alice拥有的土地。Alice通过说服政府更新记录，将土地转让给Bob。

比特币的运作方式类似。每个比特币全节点都维护着一个数据库，显示Alice控制着一定数量的比特币。Alice通过说服这些全节点更新其数据库，将部分比特币转移到Bob手中。Alice用于说服全节点更新其数据库的信息称为交易。值得注意的是，这个过程在不直接使用Alice或Bob的身份的情况下进行，如第7章中所述。

在本章中，我们将解构一个比特币交易，并检查其各个部分，以了解它们如何以高度表现力和令人惊叹的可靠性促成价值转移。

\

序列化的比特币交易

在“探索和解码交易”第43页中，我们使用了启用了 txindex 选项的 Bitcoin Core 来检索 Alice 支付给 Bob 的交易的副本。让我们再次检索包含该付款的交易，如示例 6-1 所示。

示例 6-1. Alice 的序列化交易

```
$ bitcoin-cli getrawtransaction 466200308696215bbc949d5141a49a41\\ 38ecdfdfaa2a8029c1f9bcecd1f96177

01000000000101eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da13569
8679268041c54a0100000000ffffffff02204e0000000000002251203b41daba
4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068f8240100
0000000016001477 52c165ea7be772b2c0acb7f4d6047ae6f4768e0141cf5efe
2d8ef13ed0af21d4f4cb82422d6252d70324f6f4576b727b7d918e521c00b51b
e739df2f899c49dc267c0ad280aca6dab0d2fa2b42a45182fc83e81713010000 0000\
```

Bitcoin Core 的序列化格式很特殊，因为它是用于对交易进行承诺并在比特币的 P2P 网络中传递它们的格式，但是其他程序可以使用不同的格式，只要它们传输了所有相同的数据。然而，Bitcoin Core 的格式对于传输的数据来说相当紧凑且易于解析，因此许多其他比特币程序都使用这种格式。

> 我们知道的唯一其他广泛使用的交易序列化格式是部分签名比特币交易（PSBT）格式，其在 BIP 174 和 370 中有文档记录（其他 BIP 中有扩展文档记录）。PSBT 允许一个不受信任的程序生成一个交易模板，可以由具有必要的私钥或其他敏感数据填充模板的受信任程序（例如硬件签名设备）进行验证和更新。为了实现这一点，PSBT 允许存储关于交易的大量元数据，使其比标准序列化格式要不紧凑得多。本书不会详细介绍 PSBT，但我们强烈建议支持使用多个密钥进行签名的钱包开发人员使用它。

示例 6-1 中以十六进制显示的交易在图 6-1 中被复制为一个字节映射。请注意，显示 32 字节需要 64 个十六进制字符。此映射仅显示顶级字段。我们将按照它们在交易中出现的顺序逐个检查它们，并描述它们包含的任何其他字段。

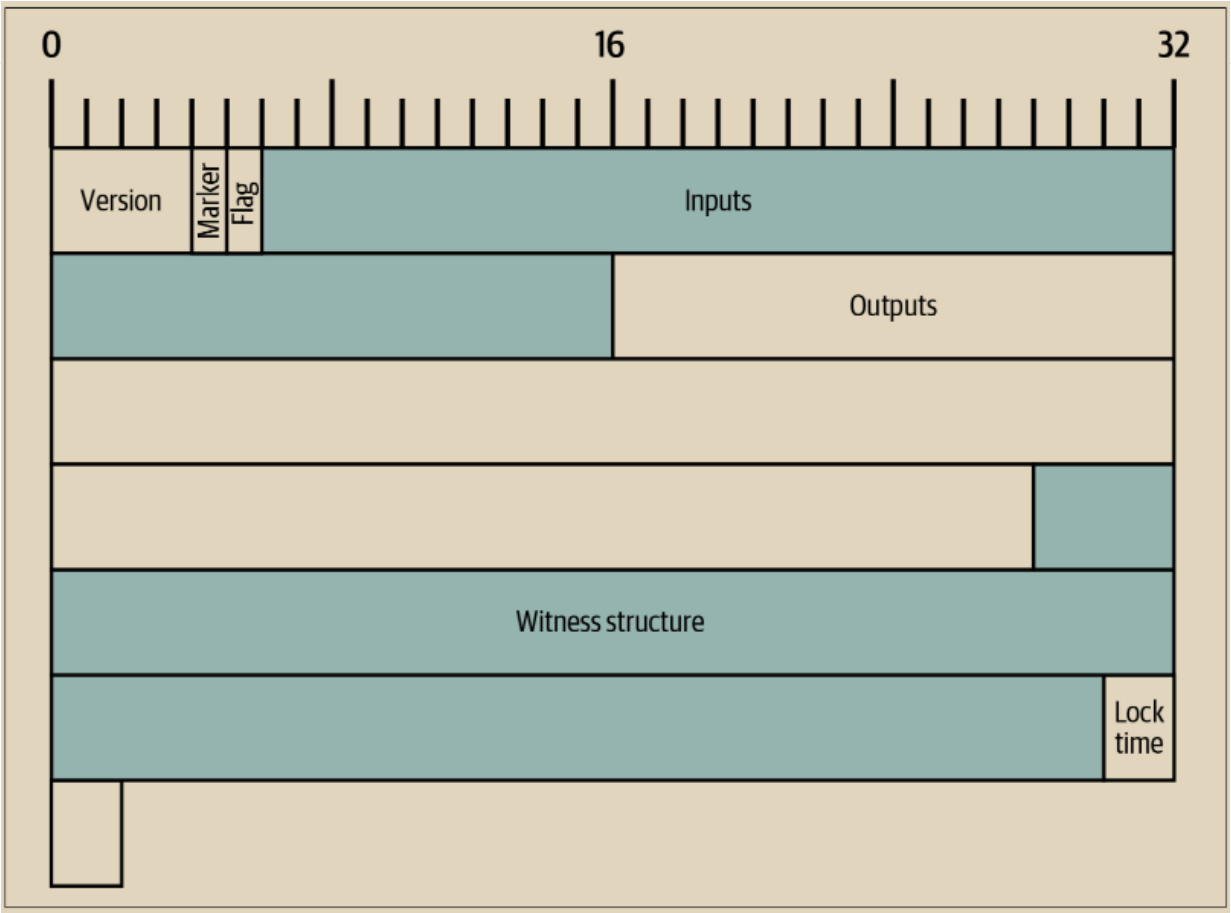


图 6-1. 爱丽丝的交易的字节映射。

版本

序列化比特币交易的前四个字节是其版本号。比特币交易的最初版本是版本 1（0x01000000）。所有比特币交易必须遵循版本 1 交易的规则，其中许多规则在本书中有所描述。

版本 2 的比特币交易是在 BIP68 软分叉更改比特币共识规则时引入的。BIP68 对序列字段增加了额外的约束条件，但这些约束条件仅适用于版本为 2 或更高的交易。版本 1 的交易不受影响。BIP112 也是与 BIP68 同一软分叉的一部分，它升级了一个操作码（OP_CHECKSEQUENCEVERIFY），如果它作为版本小于 2 的交易的一部分进行评估，将会失败。除了这两个更改之外，版本 2 的交易与版本 1 的交易相同。\

> **保护预签名交易**
>
> \ 在将交易广播到网络以包含在区块链中之前的最后一步是对其进行签名。然而，可以在不立即广播的情况下对交易进行签名。您可以将这种预签名交易保存几个月甚至几年，认为在以后广播时可以将其添加到区块链中。在此期间，甚至可能丢失访问签署另一笔消费资金所需的私钥（或密钥）。这并非是假设性的：建立在比特币之上的几个协议，包括闪电网络，都依赖于预签名交易。
>
> 对于协议开发人员来说，当他们帮助用户升级比特币共识协议时，这就产生了挑战。添加新的约束（例如 BIP68 对序列字段所做的）可能会使一些预签名交易无效。如果没有办法为等价交易创建新的签名，那么在预签名交易中使用的资金就永久丢失了。
>
> 这个问题通过将一些交易特性保留给升级解决，比如版本号。在 BIP68 之前创建预签名交易的任何人都应该使用版本 1 的交易，因此仅将 BIP68 的额外约束应用到版本 2 或更高版本的交易上不应该使任何预签名交易无效。
>
> 如果您实施了一个使用预签名交易的协议，请确保它不使用任何保留给未来升级的功能。比特币核心的默认交易中继策略不允许使用保留的功能。您可以通过在比特币主网上使用比特币核心的 testmempoolaccept RPC 来测试交易是否符合该策略。

截至撰写本文时，普遍考虑开始使用版本 3 交易的提案正在广泛讨论中。该提案并不旨在改变共识规则，而只是改变比特币全节点用于中继交易的策略。根据该提案，版本 3 交易将受到额外约束，以防止某些拒绝服务（DoS）攻击，我们将在第 212 页的“交易固定”中进一步讨论这些攻击。

扩展Marker和Flag字段

示例序列化交易的下两个字段是作为隔离见证（SegWit）软分叉对比特币共识规则进行的修改的一部分添加的。这些规则根据 BIP 141 和 BIP 143 进行了修改，但扩展的序列化格式在 BIP 144 中定义。

如果交易包含见证结构（我们将在第133页“见证结构”中描述），则标记必须为零（0x00），标志必须为非零。在当前的P2P协议中，标志应始终为1（0x01）；备用标志保留用于以后的协议升级。

如果交易不需要见证堆栈，则标记和标志不得存在。这与比特币交易序列化格式的原始版本兼容，现在称为传统序列化。有关详细信息，请参阅第142页“传统序列化”。

在传统序列化中，标记字节将被解释为输入数量（零）。一笔交易不能有零个输入，因此标记向现代程序发出了使用扩展序列化的信号。标志字段提供了类似的信号，并且还简化了将来更新序列化格式的过程。

输入

输入字段包含其他几个字段，因此让我们首先在图6-2中显示这些字节的映射。\

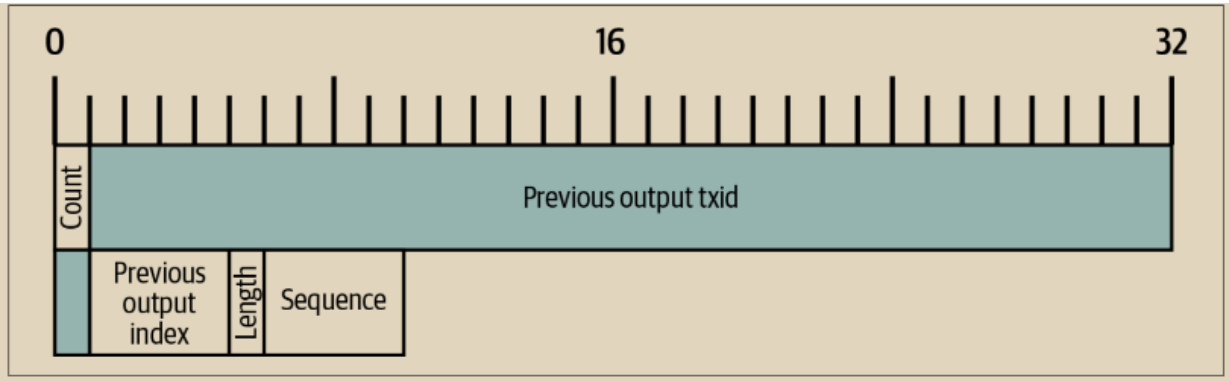


图 6-2. Alice的交易输入字段中的字节映射

交易输入列表长度字段

交易输入列表以一个整数开始，该整数表示交易中输入的数量。最小值为一个。虽然没有明确的最大值，但是对于交易大小的最大限制实际上将交易限制在几千个输入之内。该数字被编码为一个紧凑型无符号整数。

> **紧凑型无符号整数**
>
> 比特币中的无符号整数通常具有较低的值，但有时可能具有较高的值，这些值通常使用CompactSize数据类型进行编码。CompactSize是可变长度整数的一种版本，因此有时被称为var_int或varint（例如，参见BIP 37和144的文档）。
>
> \ 在不同的程序中，包括不同的比特币程序中，使用了几种不同的可变长度整数。例如，比特币核心使用称为VarInts的数据类型来序列化其UTXO数据库，这与CompactSize不同。此外，在比特币区块头中，nBits字段使用称为Compact的自定义数据类型进行编码，这与compactSize无关。在讨论比特币交易序列化和比特币P2P协议的其他部分中使用的可变长度整数时，我们将始终使用全名compactSize。
>
> 对于0到252之间的数字，紧凑大小无符号整数与C语言数据类型uint8_t相同，这可能是任何程序员熟悉的本地编码方式。对于其他数字，可以将一个字节作为前缀添加到数字前，以指示其长度，但除此之外，这些数字看起来就像常规的C语言编码的无符号整数：

值范围	使用字节数	格式
≥ 0 && ≤ 252 (0xfc)	1	uint8_t
≥ 253 && ≤ 0xffff	3	0xfd后面跟着一个uint16_t类型的数字。
≥ 0x10000 && ≤ 0xffffffff	5	0xfd后面跟着一个uint32_t类型的数字。
≥ 0x100000000 && ≤ 0xffffffffffffffff	9	0xfd后面跟着一个uint64_t类型的数字。

每个交易输入必须包含三个字段：outpoint字段、带长度前缀的输入脚本字段和序列字段。我们将在接下来的章节中逐一查看这些字段。一些输入还包括见证堆栈，但这些在交易的末尾进行序列化，因此我们稍后会对其进行检查。

Outpoint字段

比特币交易是要求全节点更新其包含的货币所有权信息的请求。为了将她的一些比特币转让给鲍勃，Alice 首先需要告诉全节点如何找到她接收这些比特币的先前交易。由于比特币的控制是通过交易输出进行分配的，Alice 使用一个输出点字段指向先前的输出。每个输入必须包含一个输出点。

输出点包含一个32字节的 txid，用于指向 Alice 接收比特币的交易。这个 txid 使用比特币内部的字节顺序表示（参见“内部和显示字节顺序”）。

因为交易可能包含多个输出，Alice 还需要标识该交易的特定输出索引，称为其输出索引。输出索引是从零开始的4字节无符号整数。

当一个全节点遇到一个输出点时，它会使用这些信息来查找被引用的输出。全节点只需要查看区块链中的先前交易。例如，Alice 的交易包含在块774,958中。验证她的交易的全节点只会在该块及之前的块中查找她的输出点引用的先前输出，而不会查找任何更晚的块。在块774,958中，它们只会查看排在 Alice 交易之前的区块中的交易，这由块的默克尔树中叶子节点的顺序决定。

在找到先前的输出后，全节点从中获取了几个关键的信息：

- 被分配给先前输出的比特币数量。这些比特币将在本次交易中转移。在示例交易中，先前输出的价值为100,000 satoshis。
- 先前输出的授权条件。这些是必须满足的条件，才能花费分配给先前输出的比特币。
- 对于已确认的交易，确认它的区块高度和该区块的中位时间过去（MTP）。这对于相对时间锁（在“作为共识强制相对时间锁的序列”中描述）和 coinbase 交易的输出（在“Coinbase 交易”中描述）是必要的。
- 先前输出存在于区块链中（或作为已知的未确认交易），并且没有其他交易花费它的证据。比特币的一个共识规则禁止任何输出在有效的区块链中被花费超过一次。这是针对双重支付的规则：Alice 不能使用同一个先前输出在不同的交易中分别支付给 Bob 和 Carol。试图花费相同先前输出的两个交易称为冲突交易，因为只有一个可以包含在有效的区块链中。

不同的全节点实现在不同的时期尝试了跟踪先前输出的不同方法。比特币核心目前使用被认为是最有效的解决方案，即保留所有必要信息同时最小化磁盘空间的解决方案：它维护一个数据库，存储每个未花费交易输出（UTXO）及其关键元数据（如其确认块高度）。每当新的交易块到达时，它们花费的所有输出都将从 UTXO 数据库中移除，而它们创建的所有输出都将添加到数据库中。

> **内部和显示字节顺序**
>
> 比特币在各种情况下使用哈希函数的输出，称为摘要。摘要为区块和交易提供唯一标识符；它们用于地址、区块、交易、签名等的承诺；并且摘要在比特币的工作量证明功能中进行迭代。在某些情况下，哈希摘要以一种字节顺序显示给用户，但在内部使用另一种字节顺序，这可能会引起混淆。例如，考虑我们示例交易中的前一个输出的事务ID（txid）：
>
> eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da135698679268041c54a
>
> 如果我们尝试使用Bitcoin Core来使用该txid检索该交易，我们会收到一个错误，并且必须反转其字节顺序：
>
> $ bitcoin-cli getrawtransaction \\
> eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da135698679268041c54a
>
> error code: -5
>
> error message:
>
> No such mempool or blockchain transaction.
>
> Use gettransaction for wallet transactions.
>
> $ echo eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da135698679268041c54a | fold -w2 | tac | tr -d "\n" 4ac541802679866935a19d4f40728bb89204d0cac90d85f3a51a19278fe33aeb
>
> $ bitcoin-cli getrawtransaction \\
> 4ac541802679866935a19d4f40728bb89204d0cac90d85f3a51a19278fe33aeb
> 02000000000101c25ae90c9f3d40cc1fc509ecfd54b06e35450702...
>
> \ 这种奇怪的行为可能是早期比特币软件设计决策的无意后果。从实际角度来看，这意味着比特币软件的开发者需要记住将显示给用户的交易和区块标识符中的字节顺序反转。
>
> 在本书中，我们使用内部字节顺序一词来表示出现在交易和区块中的数据。我们使用显示字节顺序来表示显示给用户的形式。另一组常见术语是内部版本的小端字节顺序和显示版本的大端字节顺序。

输入脚本字段

\ 输入脚本字段是遗留交易格式的残余。我们的示例交易输入花费了一个不需要输入脚本中的任何数据的本地隔离见证输出，因此输入脚本的长度前缀设置为零（0x00）。

对于一个长度前缀的输入脚本示例，它花费了一个遗留输出，我们使用了一个在此书撰写时的最新区块中的任意交易： 6b483045022100a6cc4e8cd0847951a71fad3bc9b14f24d44ba59d19094e0a8c fa2580bb664b020220366060ea8203d766722ed0a02d1599b99d3c95b97dab8e 41d3e4d3fe33a5706201210369e03e2c91f0badec46c9c903d9e9edae67c167b 9ef9b550356ee791c9a40896

长度前缀是一个紧凑型无符号整数，表示序列化的输入脚本字段的长度。在这种情况下，它是一个字节（0x6b），表示输入脚本为107字节。我们将在第7章详细介绍解析和使用脚本。

序列号字段

输入的最后四个字节是它的序列号。这个字段的用途和含义随着时间的推移而发生了变化。

最开始是基于序列号的交易替换

\ 最初，序列字段旨在允许创建相同交易的多个版本，随后的版本可以替换之前的版本作为确认的候选项。序列号跟踪交易的版本。

例如，假设Alice和Bob想要在一场纸牌游戏上进行赌注。他们首先各自签署一笔交易，将一些资金存入一个脚本，该脚本要求双方签名才能支出，这是一个多重签名脚本（简称多签）。这被称为建立交易。然后，他们创建一个花费该输出的交易：

- 交易的第一个版本，带有nSequence 0（0x00000000），将最初存入的资金返还给Alice和Bob。这被称为退款交易。此时他们都不会广播退款交易。只有在出现问题时才需要它。
- Alice赢得了纸牌游戏的第一轮，因此交易的第二个版本，带有序列1，增加了支付给Alice的金额，并减少了Bob的份额。他们都签署了更新后的交易。同样，除非出现问题，否则他们不需要广播此版本的交易。
- Bob赢得了第二轮，所以序列递增为2，减少了Alice的份额，增加了Bob的份额。他们再次签署但不广播。
- 在经过了许多轮游戏，序列被递增，资金被重新分配，得到的交易被签署但不广播后，他们决定完成交易。创建一个具有最终资金余额的交易，他们将序列设置为其最大值（0xffffffff），以完成交易。他们广播了这个版本的交易，它在网络中传播，并最终被矿工确认。

我们可以看到序列的替换规则是如何工作的，如果我们考虑替代情况：

\ • 想象一下，如果Alice广播具有0xffffffff序列的最终交易，然后Bob广播其中一笔余额较高的早期交易。由于Bob的交易版本具有较低的序列号，使用原始比特币代码的全节点不会将其中继到矿工，而且也使用原始代码的矿工也不会挖掘它。

• 在另一种情况下，想象一下，Bob在Alice广播最终版本之前几秒钟广播了早期版本的交易。节点将中继Bob的版本，并且矿工将尝试对其进行挖掘，但是当带有较高序列号的Alice的版本到达时，节点也将中继它，并且使用原始比特币代码的矿工将尝试挖掘它，而不是Bob的版本。除非Bob幸运地在Alice的版本到达之前发现了一个区块，否则将确认Alice的交易版本。

这种类型的协议现在被称为支付通道。比特币的创始人在一封被归因于他的电子邮件中称这些交易为高频交易，并描述了协议中添加的一些功能以支持它们。我们稍后将了解到其他几个特性，也将发现当代支付通道的现代版本如何越来越多地在比特币中使用。

纯序列基础的支付通道存在一些问题。第一个问题是替换较低序列交易的规则仅仅是软件策略的问题。没有直接的激励使矿工更喜欢某一版本的交易而不是其他版本。第二个问题是第一个发送交易的人可能会很幸运，即使它不是最高序列的交易，也可能被确认。由于不幸的原因而导致的安全协议失败并不是一个非常有效的协议。

第三个问题是可以无限次替换交易的一个版本。每次替换都会消耗网络上所有中继全节点的带宽。例如，截至本书写作时，有约50,000个中继全节点；一个攻击者每分钟创建1,000个200字节的替代交易，每分钟将使用约10 GB的全节点网络带宽。除了他们每分钟20 KB的带宽成本和偶尔确认交易时的费用之外，攻击者不需要支付任何费用来承担对全节点运营者造成的巨大负担。

为了消除这种攻击的风险，早期比特币软件中禁用了原始类型的序列基础交易替换。数年来，比特币全节点不允许包含特定输入（由其输出点指示）的未确认交易被包含相同输入的不同交易替换。然而，这种情况并没有持续很久。

选择性交易替换信号

由于原始基于序列的交易替换存在滥用潜力而被禁用后，提出了一种解决方案：编写比特币核心和其他中继全节点软件，允许支付更高交易费率的交易替换支付更低费率的冲突交易。这被称为替代费率，简称RBF。一些用户和企业反对将交易替换支持重新添加到比特币核心中，因此达成了一项妥协，再次利用序列字段来支持替换。

正如BIP125所记录的那样，具有任何输入的未确认交易，其序列设置为低于0xfffffffe的值（即至少低于最大值2），向网络发出信号，表明其签名者希望它可以被支付更高费率的冲突交易替换。比特币核心允许这些未确认交易被替换，并继续禁止其他交易被替换。 这允许反对替换的用户和企业简单地忽略包含BIP125信号的未确认交易，直到它们被确认。

现代交易替换政策不仅涉及费率和序列信号，还有更多内容，我们将在“替代费率 (RBF) 费率提升”中看到。

相对时间锁定的一致性强制序列

\ 在“版本”一节中，我们了解到BIP68软分叉为版本号为2或更高的交易添加了一个新的约束。该约束适用于序列字段。

序列值小于2^31的交易输入被解释为具有相对时间锁定。这样的交易只能在前一个输出（由outpoint引用）的相对时间锁定量经过一段时间后才能包含在区块链中。例如，具有一个输入的交易，其相对时间锁定为30个区块，只能在至少有29个区块的区块中确认，这些区块在同一区块链中花费相同的输出。由于序列是每个输入的字段，一个交易可能包含任意数量的带有相对时间锁定的输入，所有这些输入都必须经过足够长的时间才能使交易有效。一个禁用标志允许一个交易包含具有相对时间锁定（序列<2^31）和不具有相对时间锁定（序列≥2^31）的输入。

序列值可以以区块或秒为单位指定。一个类型标志用于区分计算区块的值和计算以秒为单位的时间的值。类型标志设置在第23个最低有效位（即值1<<22）。如果类型标志被设置，则序列值将被解释为512秒的倍数。如果类型标志未设置，则序列值将被解释为区块数。

在将序列解释为相对时间锁定时，只考虑16位最低有效位。一旦标志（位32和23）被评估，序列值通常会被使用16位掩码“掩盖”（例如，sequence & 0x0000FFFF）。512秒的倍数大致相当于区块之间的平均时间间隔，因此从16位（2^16）中得到的最大相对时间锁定的值，无论是以区块还是秒为单位，都略多于一年。

图6-3显示了由BIP68定义的序列值的二进制布局。

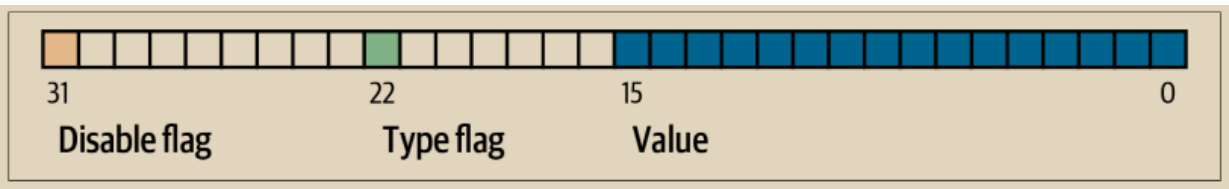


图 6-3. BIP68对序列编码的定义（来源：BIP68）

请注意，任何使用序列设置相对时间锁定的交易也会发送用于选择性接受费用替换的信号，如“选择性交易替换信号”中所述。

\

输出

交易的输出字段包含与特定输出相关的几个字段。与我们对输入字段所做的一样，我们将从示例交易中的输出字段的特定字节开始，并将这些字节显示为图6-4中的映射。

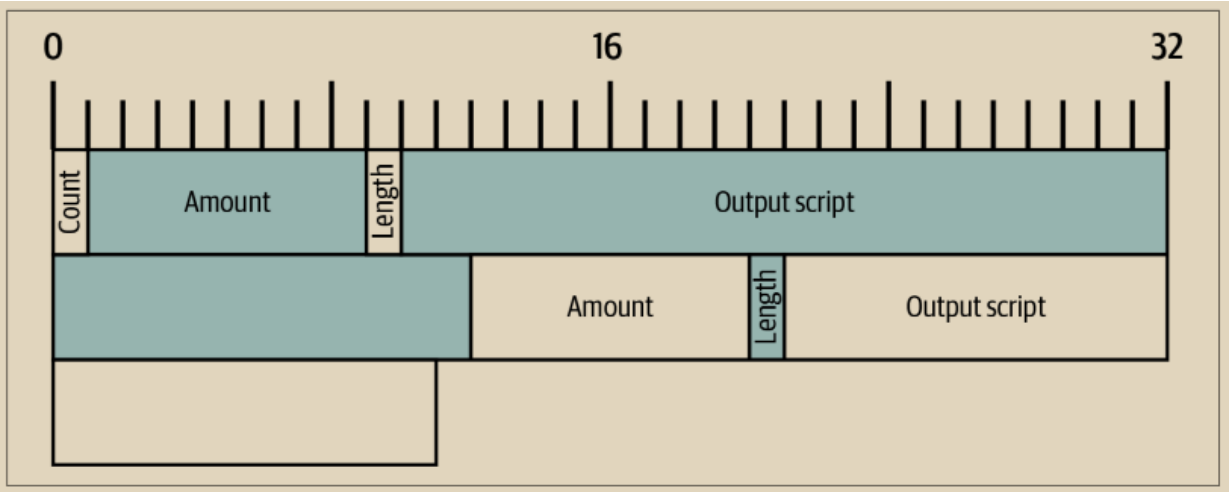


图 6-4. 来自Alice交易的outputs字段的字节映射

交易输出列表长度字段

与交易的inputs部分相似，outputs字段的开头包含一个计数，表示此交易中的输出数量。它是一个compactSize整数，必须大于零。

示例交易有两个输出。

转出数量字段

特定输出的第一个字段是其金额，在Bitcoin Core中也称为“value”。这是一个8字节的有符号整数，表示要转移的satoshis数量。Satoshis是比特币链上交易中可以表示的最小单位。1比特币等于1亿satoshis。

比特币的共识规则允许输出的值最小为零，最大为2100万比特币（2.1千万亿satoshis，比特币限制了总量为2100万比特币不变）。

不经济的输出和不允许的小额资金

尽管没有价值，但零价值的输出可以根据与任何其他输出相同的规则进行消费。但是，消费输出（将其用作交易的输入）会增加交易的大小，从而增加需要支付的费用。如果输出的价值小于额外费用的成本，则消费它就没有经济意义。这样的输出被称为不经济的输出。

零价值的输出始终是不经济的输出；即使交易的费率为零，它也不会为花费它的交易提供任何价值。然而，许多其他价值较低的输出也可能是不经济的，甚至是无意中的。例如，在当前网络上的典型费率下，一个输出可能会给交易增加更多的价值，而花费它的成本则更高——但是明天，费率可能会上升，使输出变得不经济。

全节点需要跟踪所有的未使用交易输出（UTXO），如“Outpoint”中所述，这意味着每个UTXO都会使全节点的运行变得稍微困难。对于包含大量价值的UTXO，有动机最终将它们花费掉，因此它们不会成为问题。但是，对于控制不经济的UTXO的人来说，没有动机花费它，这可能使其成为全节点运营者的永久负担。因此，比特币核心等几种全节点实现通过影响未确认交易的中继和挖矿来阻止不经济的输出的创建。\ \ 针对中继或挖矿创建新不经济输出的政策被称为尘埃政策，基于将具有非常小价值的输出与非常小尺寸的粒子进行比喻。比特币核心的尘埃政策很复杂，包含了几个任意的数字，因此我们所知道的许多程序简单地假定小于546个聪的输出是尘埃，不会默认中继或挖矿。有时会有降低尘埃限制的提案，以及提高尘埃限制的反对提案，因此我们鼓励使用预签名交易或多方协议的开发人员检查政策是否在本书出版后发生了变化。

> 自比特币问世以来，每个全节点都需要保留每个UTXO的副本，但这可能并不总是情况。几位开发人员一直在开发Utreexo项目，该项目允许全节点存储对UTXO集合的承诺，而不是数据本身。最小的承诺可能只有一两千字节大小——与比特币核心截至本书撰写时存储的逾五千兆字节相比，这是非常小的。
>
> 然而，Utreexo仍然需要一些节点存储所有UTXO数据，特别是为矿工和其他需要快速验证新区块的操作提供服务的节点。这意味着即使在可能的未来大多数节点使用Utreexo的情况下，不经济的输出仍然可能成为全节点的问题。

比特币核心关于尘埃的策略规则有一个例外：以OP_RETURN开头的输出脚本，称为数据载体输出，可以具有零值。OP_RETURN操作码导致脚本立即失败，无论之后发生什么，因此这些输出永远无法被花费。这意味着全节点不需要跟踪它们，比特币核心利用这一特性，允许用户在区块链中存储少量任意数据，而不会增加其UTXO数据库的大小。由于这些输出是不可花费的，它们并不是不经济的——分配给它们的任何聪将永久不可花费——因此允许金额为零可确保没有聪被销毁。

输出脚本

输出金额后面是一个compactSize整数，指示输出脚本的长度，该脚本包含了需要满足的条件，以便花费比特币。根据比特币的共识规则，输出脚本的最小大小为零。

输出脚本的共识最大允许大小取决于何时进行检查。在交易输出中，输出脚本的大小没有明确的限制，但后续交易只能花费具有大小不超过10,000字节的先前输出。隐含地，输出脚本几乎可以与包含它的交易一样大，而交易几乎可以与包含它的区块一样大。

> 长度为零的输出脚本可以由包含OP_TRUE的输入脚本花费。任何人都可以创建该输入脚本，这意味着任何人都可以花费空的输出脚本。有一种基本上无限数量的脚本，任何人都可以花费，并且比特币协议开发人员将其称为任何人都可以花费。比特币的脚本语言升级通常会将现有的任何人都可以花费的脚本添加新的约束条件，使其仅在新条件下可花费。应用程序开发人员不应该需要使用任何人都可以花费的脚本，但如果您确实需要，我们强烈建议您大声宣布您的计划给比特币用户和开发人员，以便未来的升级不会意外地干扰您的系统。

比特币核心对于中继和挖矿交易的策略有效地将输出脚本限制在几个模板上，称为标准交易输出。这最初是在发现了与脚本语言相关的几个早期错误之后实施的，并在现代比特币核心中保留下来，以支持任何人都可以花费的升级，并鼓励将脚本条件放置在P2SH赎回脚本、segwit v0见证脚本和segwit v1（taproot）叶脚本中的最佳实践。

我们将在第7章中查看每个当前标准交易模板，并学习如何解析脚本。

见证结构(Witness Structure)

在法庭上，证人是指证明他们目睹了重要事件发生的人。人类证人并不总是可靠的，因此法院有各种程序来审问证人，以便（理想情况下）只接受那些可靠的证据。

想象一下数学问题的证人会是什么样子。例如，如果重要的问题是 x + 2 == 4，某人声称他们目睹了解决方案，我们会问什么？我们想要一个数学证明，证明存在一个值，将其与2相加得到4。我们甚至可以省去人的需要，只使用提议的 x 值作为我们的证人。如果我们被告知证人是2，那么我们可以填写方程，检查它是否正确，并决定重要的问题已经解决了。

当花费比特币时，我们想要解决的重要问题是确定该花费是否得到了控制这些比特币的人或团体的授权。执行比特币共识规则的成千上万的全节点无法审问人类证人，但他们可以接受完全由用于解决数学问题的数据组成的证人。例如，一个值为2的见证将允许花费由以下脚本保护的比特币：

2 OP_ADD 4 OP_EQUAL

\ 显然，允许任何能够解决简单方程的人花费您的比特币是不安全的。正如我们将在第8章中看到的那样，一个不可伪造的数字签名方案使用的方程只能由某人在掌握某些他们能够保密的数据时解决。他们可以使用公共标识符引用该秘密数据。该公共标识符称为公钥，方程的解称为签名。

以下脚本包含一个公钥和一个操作码，要求与花费交易中的数据相对应的签名。就像我们简单示例中的数字2一样，签名就是我们的见证：

\ OP_CHECKSIG\

见证者是用于解决保护比特币的数学问题的值，在使用它们的交易中需要包含它们，以便全节点进行验证。在所有早期比特币交易中使用的传统交易格式中，签名和其他数据放置在输入脚本字段中。然而，当开发者开始在比特币上实现合约协议时，就像我们在“原始基于序列的交易替换”中所看到的那样，他们发现将见证者放置在输入脚本字段中存在几个重大问题。

循环依赖

很多比特币的合约协议涉及一系列按顺序签名的交易。例如，Alice和Bob想要将资金存入一个只能通过两人签名才能花费的脚本，但他们每个人也希望在对方不响应时能够拿回自己的钱。一个简单的解决方案是按顺序签署交易：

- Tx0支付了来自Alice和Bob的资金到一个脚本的输出，该脚本要求Alice和Bob的签名才能花费。
- Tx1花费了前一个输出到两个输出，一个是退还Alice她的钱，另一个是退还Bob他的钱（减去一笔小额用于交易费）。
- 如果Alice和Bob在签署Tx0之前签署Tx1，那么他们都能随时获得退款。这个协议不需要他们中的任何一个相信另一个，因此是一个无需信任的协议。

在传统的交易格式中，这种构造存在一个问题，即每个字段，包括包含签名的输入脚本字段，都用于生成交易的标识符（txid）。Tx1的txid是Tx1中输入的outpoint的一部分。这意味着在Alice和Bob了解Tx0的签名之前，他们无法构造Tx1。但如果他们知道Tx0的签名，其中一个人可以在签署退款交易之前广播该交易，从而消除退款的保证。这就是一个循环依赖。

第三方交易篡改

一个更复杂的交易系列有时可以消除循环依赖，但是许多协议随后会遇到一个新的问题：通常可以用不同的方式解决相同的脚本。例如，考虑我们从“见证结构”中的简单脚本：

\ 2 OP_ADD 4 OP_EQUAL

我们可以通过在输入脚本中提供值2来使此脚本通过，但在比特币中有几种方法可以将该值放入堆栈中。以下只是其中几种方式：

OP_2

OP_PUSH1 0x02

OP_PUSH2 0x0002

OP_PUSH3 0x000002

...

OP_PUSHDATA1 0x0102

OP_PUSHDATA1 0x020002

...

OP_PUSHDATA2 0x000102

OP_PUSHDATA2 0x00020002

...

OP_PUSHDATA4 0x0000000102

OP_PUSHDATA4 0x000000020002

...

\ 每种对数字2在输入脚本中的替代编码都会生成一个略有不同的交易，具有完全不同的交易ID（txid）。每个不同版本的交易都花费了与其他版本相同的输入（outpoint），使它们彼此冲突。在有效的区块链中，一组冲突交易只能包含其中的一个版本。

想象一下，Alice创建了一个在输入脚本中带有OP_2的交易版本，并有一个输出支付给Bob。然后Bob立即将该输出花费给了Carol。任何网络上的人都可以用OP_PUSH1 0x02替换OP_2，与Alice的原始版本产生冲突。如果那个冲突的交易得到确认，那么就没有办法将Alice的原始版本包含在同一区块链中，这意味着Bob的交易无法花费其输出。尽管Alice、Bob和Carol都没有做错任何事情，但Bob向Carol的付款变得无效了。某人（第三方）能够改变（突变）Alice的交易，这就是所谓的不受欢迎的第三方交易篡改问题。

> 有些情况下，人们希望他们的交易是可篡改的，比特币提供了几个功能来支持这一点，其中最重要的是我们将在“签名哈希类型（SIGHASH）”中学到的签名哈希（sighash）。例如，Alice可以使用签名哈希允许Bob帮助她支付一些交易费用。这会改变Alice的交易，但只以Alice想要的方式。因此，我们有时会在术语“交易可篡改性”前加上“不受欢迎的”一词。即使我们和其他比特币技术撰写者使用更短的术语，我们几乎肯定是在谈论不受欢迎的篡改变体。

第二方交易篡改

\ 当传统交易格式是唯一的交易格式时，开发人员致力于提出方案来最小化第三方篡改，例如 BIP62。然而，即使他们能够完全消除第三方篡改，合约协议的用户仍然面临另一个问题：如果他们需要来自协议中其他参与者的签名，那么该人可以生成替代签名并更改 txid。

例如，Alice 和 Bob 将他们的资金存入一个需要双方签名才能支取的脚本中。他们还创建了一个退款交易，允许他们任何一方随时取回自己的资金。Alice 决定只花费部分资金，因此她与 Bob 合作创建了一系列交易：

- Tx0 包含来自 Alice 和 Bob 的签名，将比特币花费到两个输出。第一个输出支付了部分 Alice 的资金；第二个输出将剩余的比特币退还给需要 Alice 和 Bob 签名的脚本。在签署此交易之前，他们创建了一个新的退款交易 Tx1。
- Tx1 将 Tx0 的第二个输出花费到两个新的输出，一个给 Alice 作为她在联合资金中的份额，另一个给 Bob。Alice 和 Bob 在签署 Tx0 之前都签署了这笔交易。

这里没有循环依赖，如果我们忽略第三方交易篡改，这看起来应该为我们提供一个无需信任的协议。然而，Bitcoin 签名的一个特性是签名者在创建签名时必须选择一个大的随机数。即使签名的一切保持不变，选择不同的随机数也会产生不同的签名，就像如果你为两份相同合同的副本提供手写签名，每个物理签名都会略有不同一样。

签名的可变性意味着，如果 Alice 尝试广播 Tx0（其中包含 Bob 的签名），Bob 可以生成另一种签名，创建一个与不同 txid 的冲突交易。如果 Bob 的替代版本的 Tx0 被确认，那么 Alice 就无法使用预签名版本的 Tx1 来要求退款。这种变异称为不受欢迎的第二方交易篡改。

隔离见证

\ 随着早在2011年，协议开发者就知道如何解决循环依赖、第三方篡改和第二方篡改等问题。这个想法是避免将输入脚本包含在生成交易 txid 的计算中。回想一下，输入脚本中保存的数据的抽象名称是见证。将交易的其余数据与其见证分开，以便生成 txid 的想法称为隔离见证（Segregated Witness，简称 SegWit）。

明显的实现 SegWit 的方法需要对比特币的共识规则进行更改，这将与旧的全节点不兼容，也称为硬分叉。硬分叉带来了很多挑战，我们将在第 291 页的“硬分叉”中进一步讨论。

另一种实现 SegWit 的方法是在2015年底提出的。这将使用向后兼容的方式更改共识规则，称为软分叉。向后兼容意味着实现更改的全节点不得接受旧节点认为无效的任何区块。只要它们遵守这个规则，较新的全节点就可以拒绝旧全节点会接受的区块，从而使它们有能力执行新的共识规则（但前提是较新的全节点代表比特币用户之间的经济共识——我们将在第 12 章中探讨更新比特币的共识规则的细节）。

软分叉 SegWit 方法基于任何人可支配的输出脚本。以数字 0 到 16 中的任何一个数字开头，后跟 2 到 40 字节的数据的脚本被定义为隔离见证输出脚本模板。数字表示其版本（例如，0 是隔离见证版本 0，或者 SegWit v0）。数据称为见证程序。还可以将 SegWit 模板包装在 P2SH 承诺中，但我们在本章中不处理这个。

从旧节点的角度来看，这些输出脚本模板可以使用空的输入脚本进行支出。从了解新 SegWit 规则的新节点的角度来看，对隔离见证输出脚本模板的任何支付必须仅使用空的输入脚本进行支出。请注意这里的区别：旧节点允许空的输入脚本；新节点要求空的输入脚本。

\ 空的输入脚本使得见证不会影响 txid，消除了循环依赖、第三方交易篡改和第二方交易篡改。但是，由于不能在输入脚本中放置数据，使用隔离见证输出脚本模板的用户需要一个新的字段。该字段称为见证结构（Witness Structure）。

引入见证程序和见证结构使比特币变得更加复杂，但它遵循了增加抽象的现有趋势。回想一下第 4 章，最初的比特币白皮书描述了一个系统，比特币被接收到公钥（pubkeys）并使用签名（sigs）进行支出。公钥定义了谁有权花费比特币（控制相应私钥的人），而签名提供了验证，即支出交易来自控制私钥的人。为了使该系统更加灵活，比特币的初始版本引入了脚本，允许比特币被接收到输出脚本并使用输入脚本进行支出。后来对合同协议的经验启发了允许比特币被接收到见证程序并使用见证结构进行支出。不同版本比特币中使用的术语和字段如表 6-1 所示。

表6-1. 比特币中不同部分用于授权和认证数据的术语

	授权（Authorization）	认证（Authentication）
Whitepaper	Publick key	Signature
Original(Legacy)	Output script	Input script
Segwit	Witness program	Witness structure

见证结构序列化

见证结构类似于输入和输出字段，包含其他字段，因此我们将从Alice交易的字节映射开始，如图6-5所示。

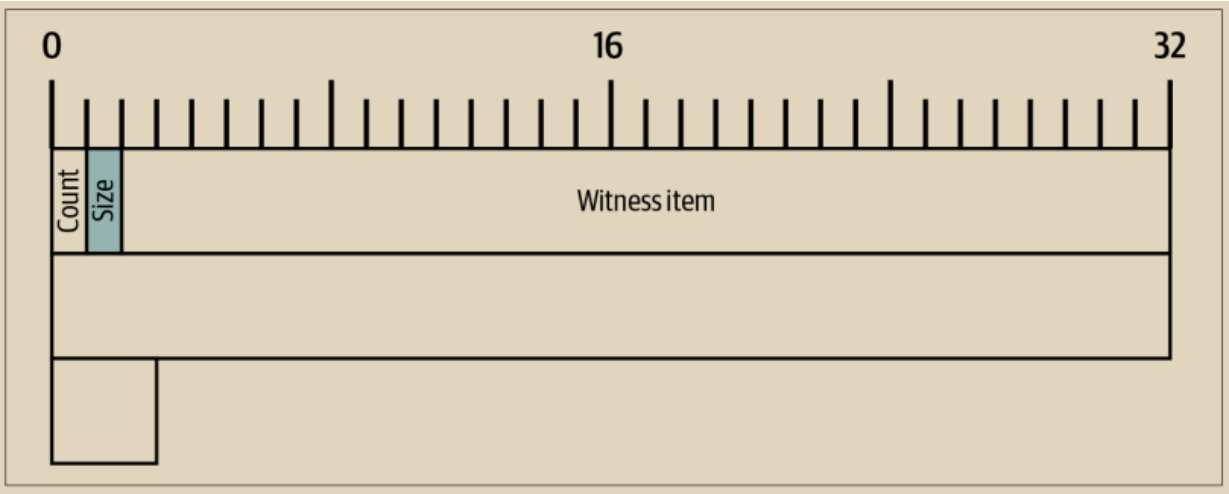


图 6-5. Alice的交易见证结构的字节映射

与输入和输出字段不同，整体的见证结构没有以包含的见证堆栈总数为开始。相反，这是由输入字段隐含的——每个交易的输入都有一个见证堆栈。

特定输入的见证结构确实以一个元素数量的计数开头。这些元素称为见证项。我们将在第7章详细探讨它们，但现在我们需要知道每个见证项都以一个表示其大小的紧凑尺寸整数为前缀。

传统的输入不包含任何见证项，因此它们的见证堆栈完全由一个计数为零的元素组成（0x00）。

Alice的交易包含一个输入和一个见证项。

\

锁定时间

序列化交易的最后一个字段是其锁定时间。这个字段是比特币最初的序列化格式的一部分，但最初只由比特币选择哪些交易挖矿的政策来执行。比特币最早的已知软分叉添加了一个规则，从区块高度31,000开始，禁止在区块中包含一个交易，除非满足以下规则之一：

- 交易指示应该有资格包含在任何块中，将其锁定时间设置为0。
- 交易指示它想要限制可以包含在其中的块，将其锁定时间设置为小于500,000,000的值。在这种情况下，该交易只能包含在高度等于锁定时间或更高的块中。例如，具有锁定时间为123,456的交易可以包含在块123,456或任何更高的块中。
- 交易指示它想要限制可以包含在区块链中的时间，将其锁定时间设置为500,000,000或更高的值。在这种情况下，该字段被解析为纪元时间（自1970年01月01日T00:00 UTC以来的秒数），并且该交易只能包含在具有大于锁定时间的中位时间过去（MTP）的块中。 MTP通常比当前时间晚大约一两个小时。有关MTP的规则，请参阅“过去的中位时间（MTP）”第280页。

Coinbase交易

每个区块中的第一笔交易都是一个特例。大多数较旧的文档称之为生成交易，但大多数较新的文档称之为Coinbase交易（不要与名为“Coinbase”的公司创建的交易混淆）。

Coinbase交易是由包含它们的区块的矿工创建的，允许矿工领取该区块中所有交易支付的任何费用。此外，在区块6,720,000之前，矿工被允许领取一笔由以前从未流通过的比特币组成的补贴，称为区块补贴。矿工可以领取的区块的总金额——包括费用和补贴——称为区块奖励。 Coinbase交易的一些特殊规则包括：

- 它们只能有一个输入。
- 单个输入的输出点必须具有空的txid（完全由零组成）和最大的输出索引（0xffffffff）。这样做是为了防止Coinbase交易引用先前的交易输出，因为Coinbase交易支付费用和补贴，引用先前的交易输出至少会令人困惑。
- 在普通交易中包含输入脚本的字段称为coinbase。正是这个字段赋予了Coinbase交易其名称。coinbase字段的长度必须至少为两个字节，但不超过100个字节。虽然此脚本不会执行，但是对于签名检查操作（sigops）的传统交易限制仍然适用，因此应在其中放置的任意数据应以数据推送操作码为前缀。自2013年BIP34定义的一个软分叉以来，此字段的前几个字节必须遵循我们将在“Coinbase数据”中描述的其他规则。
- 输出的总和不得超过区块中所有交易所收取的费用和补贴的价值。补贴从每个区块的50个比特币开始，每210,000个区块减半一次（大约每四年）。补贴值会向下舍入到最近的satoshis。
- 自2017年在BIP141中记录的segwit软分叉以来，包含消费segwit输出的交易的任何区块都必须包含一个输出到Coinbase交易的输出，该输出承诺该区块中所有交易（包括它们的见证）。我们将在第12章探讨这种承诺。Coinbase交易可以具有任何在普通交易中有效的其他输出。但是，在Coinbase交易获得100次确认之前，消费其中一笔输出的交易不能包含在任何区块中。这称为成熟规则，尚未获得100次确认的Coinbase交易输出称为未成熟。 大多数比特币软件不需要处理Coinbase交易，但它们的特殊性质意味着它们偶尔可能会导致未经设计以接受它们的软件出现异常问题。

\

权重和V字节(Vbytes)

每个比特币区块在包含的交易数据量方面都有限制，因此大多数比特币软件都需要能够测量其创建或处理的交易。比特币的现代计量单位称为权重。权重的另一种版本是V字节(Vbytes)，其中四个权重单位等于一个V字节(Vbytes)，这样可以轻松地与传统比特币区块中使用的原始字节测量单位进行比较。

区块的权重限制为4百万。区块头占据240个权重。另外一个字段，交易计数，使用4或12个权重。所有剩余的权重可以用于交易数据。

要计算交易中特定字段的权重，需要将该序列化字段的大小（以字节为单位）乘以一个因子。要计算交易的权重，请将其所有字段的权重相加。交易中每个字段的因子如表6-2所示。为了提供示例，我们还计算了本章中从Alice到Bob的示例交易中每个字段的权重。

选择这些因子以减少花费未使用交易输出（UTXO）时使用的权重。这有助于避免不经济的输出的创建，如“不经济的输出和禁止灰尘”中所述。

表6-2. 比特币交易中所有字段的权重因子

字段	因子	Alice的交易权重
Version	4	16 （因为version占4个字节，乘以4后即为16），16权重=4Vbytes
Marker & Flag	1	2
Inputs Count	4	4
Outpoint	4	144
Input script	4	14
Sequence	4	16
Output script	4	4
Amount	4	64 (2 outputs)
Output script	4	232 (2 outputs with different scripts)
Witness Count	1	1
Witness items	1	66
Lock time	4	16
Total	*N/A*	**569**

我们可以通过从比特币核心获取Alice的交易总额来验证我们的权重计算：

$ bitcoin-cli getrawtransaction 466200308696215bbc949d5141a49a41\\ 38ecdfdfaa2a8029c1f9bcecd1f96177 2 | jq .weight

569

Alice在本章开头的示例6-1中的交易以权重单位表示如图6-6所示。通过比较两个图像中各个字段的大小差异，可以看到因子的作用。\

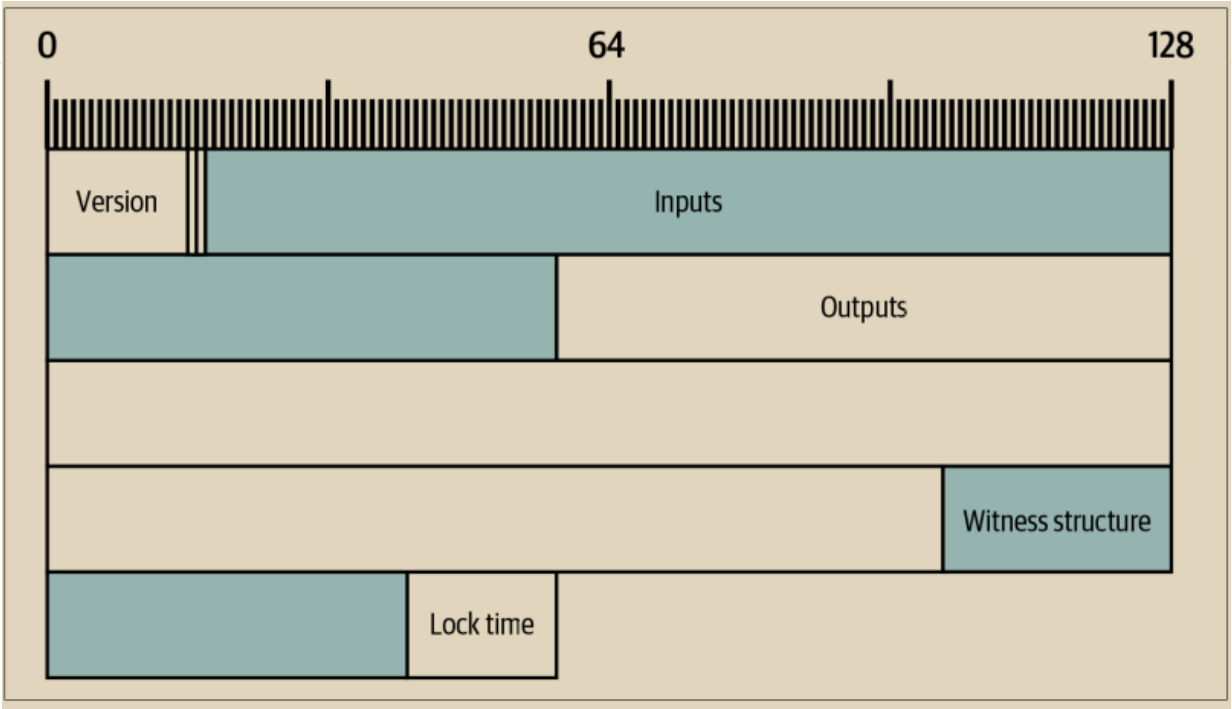


图 6-6. Alice交易的字节映射

比特币权重详细介绍可参见：比特币权重维基百科。

历史遗留序列化

\ 历史遗留序列化格式是比特币交易中较旧的序列化格式，仍然用于许多交易。对于任何具有空见证结构的交易（仅在交易不花费任何见证程序时有效），必须在比特币P2P网络上使用历史遗留序列化。

历史遗留序列化不包括标记、标志和见证结构字段。在本章中，我们看了交易中的每个字段，并了解了它们如何向全节点传达有关要在用户之间转移的比特币的详细信息。我们只是简要地看了输出脚本、输入脚本和见证结构，这些脚本允许指定和满足限制谁可以花费什么比特币的条件。

了解如何构建和使用这些条件对确保只有Alice能够花费她的比特币至关重要，因此它们将是下一章的主题。

综合介绍

当你收到比特币时，你必须决定谁有权花费它们，这称为授权。你还必须决定全节点如何区分经授权的花费者和其他人，这称为认证。你的授权指示和花费者的认证证明将由成千上万个独立的全节点进行检查，所有这些节点都需要得出相同的结论，即一笔交易的花费是经过授权和认证的，才能使包含它的交易有效。

比特币的最初描述使用了公钥进行授权。艾丽斯通过将鲍勃的公钥放入交易的输出中向他支付了比特币。认证来自鲍勃，形式是一笔承诺进行支出的签名，例如从鲍勃到卡罗尔。

最初发布的比特币实际版本提供了更灵活的授权和认证机制。此后的改进进一步增强了这种灵活性。在本章中，我们将探讨这些功能，并了解它们最常见的用法。

交易脚本和脚本语言

比特币的原始版本引入了一种称为脚本（Script）的新编程语言，它是一种类似于Forth的基于堆栈的语言。在交易输出中放置的脚本以及用于支出交易的传统输入脚本都是用这种脚本语言编写的。

脚本是一种非常简单的语言。它需要最少的处理，并且不能轻松地做现代编程语言可以做的许多花哨的事情。

\ 当遗留交易是比特币网络中最常见的交易类型时，大多数通过比特币网络处理的交易都具有“支付给Bob的比特币地址”形式，并使用一种称为支付到公钥哈希（P2PKH）脚本的脚本。然而，比特币交易并不限于“支付给Bob的比特币地址”脚本。事实上，脚本可以编写成表达各种复杂条件的形式。为了理解这些更复杂的脚本，我们必须首先了解交易脚本和脚本语言的基础。

在本节中，我们将演示比特币交易脚本语言的基本组成部分，并展示如何使用它来表达花费条件以及如何满足这些条件。

> 比特币交易验证不是基于静态模式，而是通过执行脚本语言来实现的。这种语言允许表达几乎无限种类的条件。

图灵不完备

比特币交易脚本语言包含许多运算符，但在一个重要方面有意限制——它没有循环或复杂的流程控制能力，除了条件流程控制。这确保了该语言不是图灵完备的，也就是说，脚本具有有限的复杂性和可预测的执行时间。脚本不是通用的语言。这些限制确保了语言不能用于创建无限循环或其他形式的“逻辑炸弹”，这可能会以导致针对比特币网络的拒绝服务攻击的方式嵌入到交易中。请记住，比特币网络上的每个全节点都会验证每个交易。有限的语言防止了交易验证机制被用作漏洞。

无状态验证

比特币交易脚本语言包含许多运算符，但在一个重要方面有意限制——它没有循环或复杂的流程控制能力，除了条件流程控制。这确保了该语言不是图灵完备的，也就是说，脚本具有有限的复杂性和可预测的执行时间。脚本不是通用的语言。这些限制确保了语言不能用于创建无限循环或其他形式的“逻辑炸弹”，这可能会以导致针对比特币网络的拒绝服务攻击的方式嵌入到交易中。请记住，比特币网络上的每个全节点都会验证每个交易。有限的语言防止了交易验证机制被用作漏洞。

脚本构建

比特币的传统交易验证引擎依赖于脚本的两个部分来验证交易：输出脚本和输入脚本。

输出脚本指定了必须满足的条件，以便在未来花费输出，例如谁被授权花费输出以及如何进行身份验证。

输入脚本是满足输出脚本中设定条件并允许花费输出的脚本。输入脚本是每个交易输入的一部分。在传统交易中，大部分情况下它们包含用户钱包从私钥生成的数字签名，但并非所有输入脚本都必须包含签名。

每个比特币验证节点将通过执行输出脚本和输入脚本来验证交易。正如我们在第6章中看到的那样，每个输入包含一个指向先前交易输出的输出点。输入还包含一个输入脚本。验证软件将复制输入脚本，检索由输入引用的UTXO，并从该UTXO中复制输出脚本。然后一起执行输入和输出脚本。如果输入脚本满足输出脚本的条件（参见“分别执行输出和输入脚本”），则输入有效。所有输入都独立验证，作为交易的总体验证的一部分。

请注意，上述步骤涉及复制所有数据。先前输出和当前输入的原始数据永远不会更改。特别是，先前的输出是不变的，不受未能花费它的尝试的影响。只有一个有效的交易才能正确满足输出脚本的条件，才能将输出视为“已花费”。

图7-1是传统比特币交易（付款给公钥哈希）最常见类型的输出和输入脚本的示例，显示了在验证之前对脚本进行串联的组合脚本。

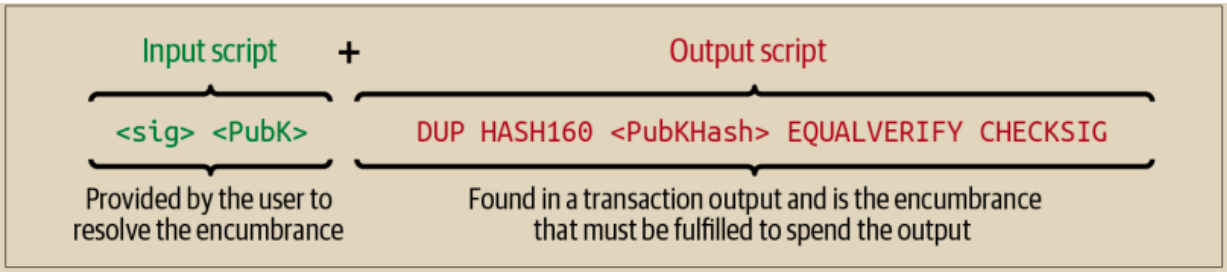


图 7-1. 组合输入脚本和输出脚本以评估交易脚本

脚本执行堆栈

\ 比特币的脚本语言被称为基于栈的语言，因为它使用一种称为栈的数据结构。栈是一种非常简单的数据结构，可以被视为一叠卡片。栈具有两个基本操作：push 和 pop。push 将一个项目添加到栈顶。pop 从栈顶移除项目。

脚本语言通过从左到右处理每个项目来执行脚本。数字（数据常量）被推送到栈上。操作符从栈中推送或弹出一个或多个参数，对它们进行操作，并可能将结果推送到栈上。例如，OP_ADD 将从栈上弹出两个项目，将它们相加，并将结果推送到栈上。

条件运算符评估一个条件，产生一个布尔结果，为 TRUE 或 FALSE。例如，OP_EQUAL 从栈中弹出两个项目，并在它们相等时推送 TRUE（TRUE 用数字 1 表示），在它们不相等时推送 FALSE（用数字 0 表示）。比特币交易脚本通常包含一个条件运算符，以便产生表示有效交易的 TRUE 结果。

一个简单的脚本

现在让我们将所学到的关于脚本和堆栈的知识应用到一些简单的示例中。

正如我们将在图7-2中看到的那样，脚本 2 3 OP_ADD 5 OP_EQUAL 展示了算术加法操作符 OP_ADD，将两个数字相加并将结果放入堆栈中，然后是条件操作符 OP_EQUAL，它检查结果的总和是否等于5。为了简洁起见，本书中的示例有时会省略 OP_ 前缀。有关可用的脚本操作符和函数的更多详细信息，请参阅比特币维基的脚本页面。

尽管大多数传统的输出脚本都引用了一个公钥哈希（本质上是一个传统的比特币地址），从而要求证明拥有权才能花费这些资金，但脚本并不一定要那么复杂。任何输出和输入脚本的组合，只要产生 TRUE 值，都是有效的。我们用作脚本语言示例的简单算术也是有效的脚本。

使用算术示例脚本的一部分作为输出脚本：

```
3 OP_ADD 5 OP_EQUAL
```

这可以通过包含以下输入脚本的交易来满足：

```
2
```

验证软件结合了这些脚本：

```
2 3 OP_ADD 5 OP_EQUAL
```

\ 正如我们在图7-2中所看到的那样，当执行此脚本时，结果为OP_TRUE，使得交易有效。尽管这是一个有效的交易输出脚本，但请注意，得到的UTXO可以被任何具有算术技能的人花费，以知道数字2满足脚本。

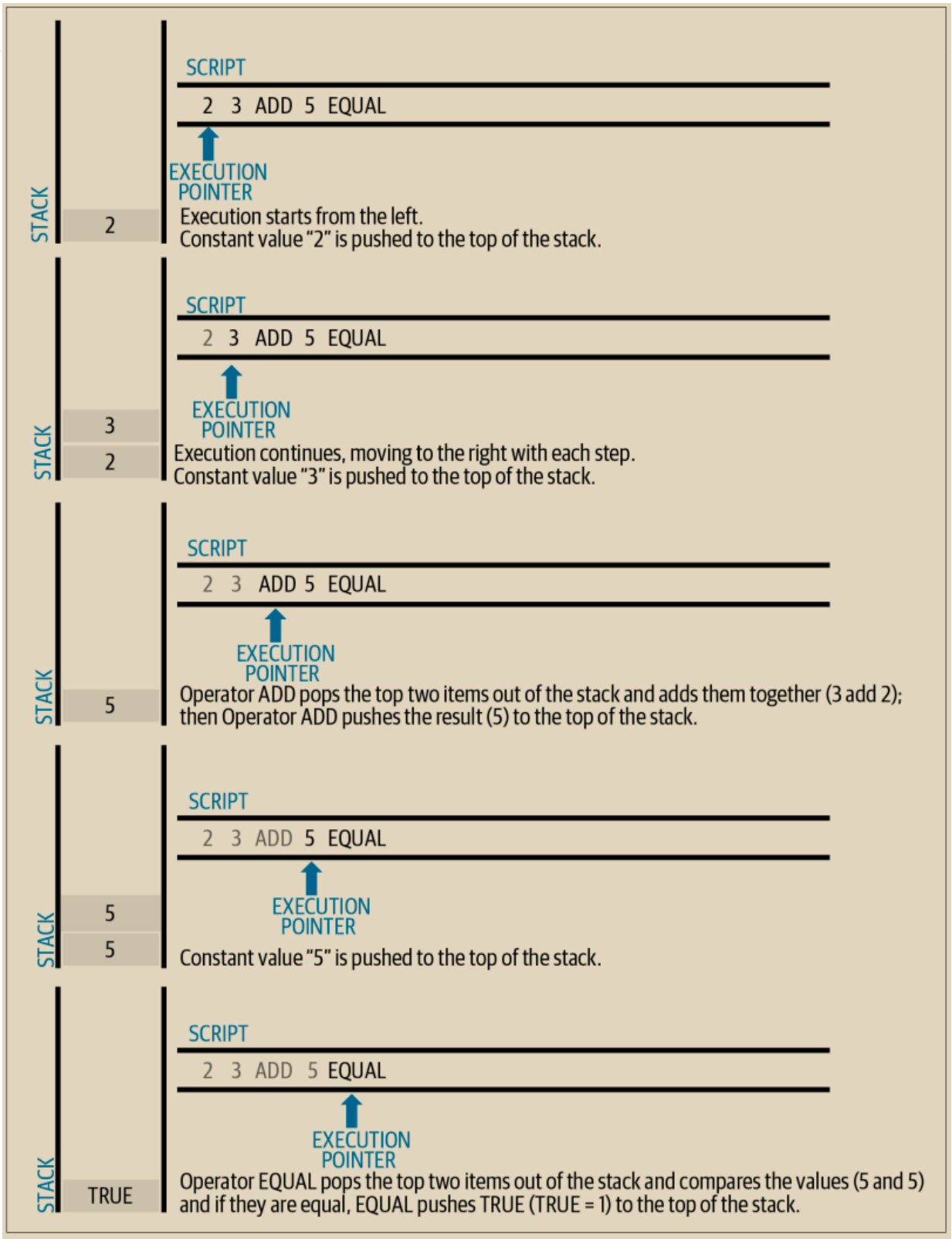


图 7-2. 比特币的脚本验证进行简单的数学运算

如果栈顶的结果为TRUE（任何非零值），则交易有效。如果栈顶的值为FALSE（值为零或空栈），脚本执行被明确中止（例如使用VERIFY、OP_RETURN等操作符），或者脚本不是语义上有效的（例如包含未由OP_ENDIF操作码终止的OP_IF语句），则交易无效。详情请参阅比特币维基的脚本页面。

以下是一个稍微复杂一些的脚本，它计算了2 + 7 - 3 + 1。请注意，当脚本中包含多个连续的操作符时，栈允许一个操作符的结果被下一个操作符使用：

2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL

尝试使用铅笔和纸验证前面的脚本。当脚本执行结束时，您应该在堆栈上留下一个TRUE值。

输出脚本和输入脚本的分离执行

在最初的比特币客户端中，输出脚本和输入脚本被连接在一起并按顺序执行。出于安全原因，这在2010年发生了改变，因为存在一个漏洞，即1 OP_RETURN bug。在当前的实现中，这些脚本是分开执行的，同时在两次执行之间传递栈数据。

首先，使用栈执行引擎执行输入脚本。如果输入脚本在执行过程中没有出现错误并且没有剩余操作，则栈数据会被复制，然后执行输出脚本。如果使用从输入脚本复制的栈数据执行输出脚本的结果为TRUE，则表示输入脚本已成功解析了输出脚本所施加的条件，因此输入是有效的，可以用于花费UTXO。如果在合并脚本执行后仍然存在除TRUE之外的任何结果，则表示输入是无效的，因为它未能满足对输出所施加的花费条件。

支付至公钥哈希（P2PKH）

支付至公钥哈希（P2PKH）脚本使用一个包含哈希值的输出脚本，该哈希值与一个公钥相关联。P2PKH 最为人熟知的是作为传统比特币地址的基础。一个 P2PKH 输出可以通过提供与哈希值相匹配的公钥以及由相应私钥创建的数字签名来进行消费（参见第 8 章）。让我们看一个 P2PKH 输出脚本的例子：

\ OP_DUP OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG

密钥哈希是将编码到传统的base58check地址中的数据。大多数应用程序会使用十六进制编码显示脚本中的公钥哈希，而不是以“1”开头的熟悉的比特币地址base58check格式。

前面的输出脚本可以通过以下形式的输入脚本满足：

\ \\

两个脚本合在一起将形成以下组合验证脚本：

\ OP_DUP OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG

如果输入脚本具有与设置为限制条件的公钥哈希相对应的Bob的私钥的有效签名，结果将为TRUE。

图 7-3 和 7-4 分两部分展示了组合脚本的逐步执行过程，证明这是一个有效的交易。\

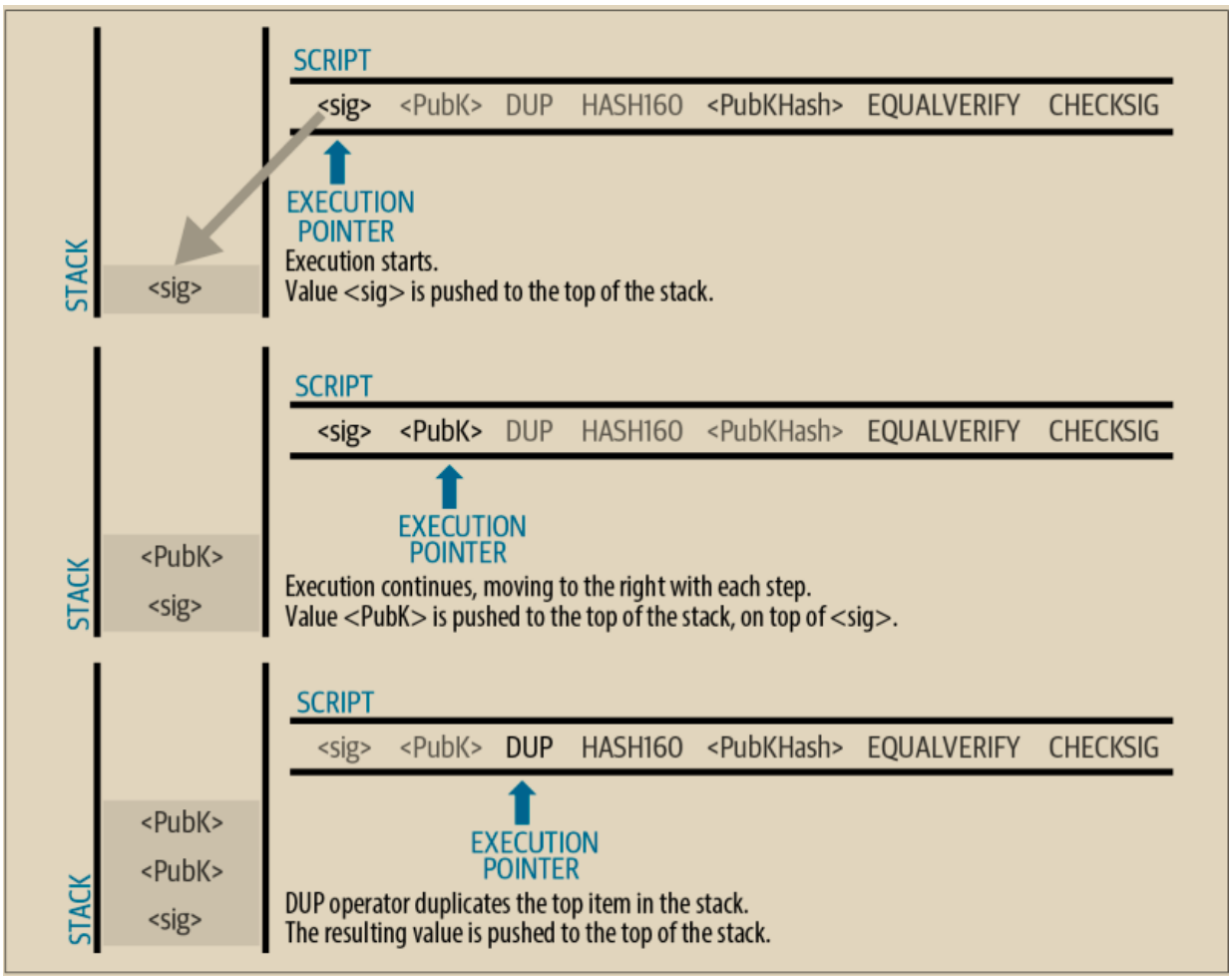


图 7-3. 执行P2PKH交易的脚本（1/2）

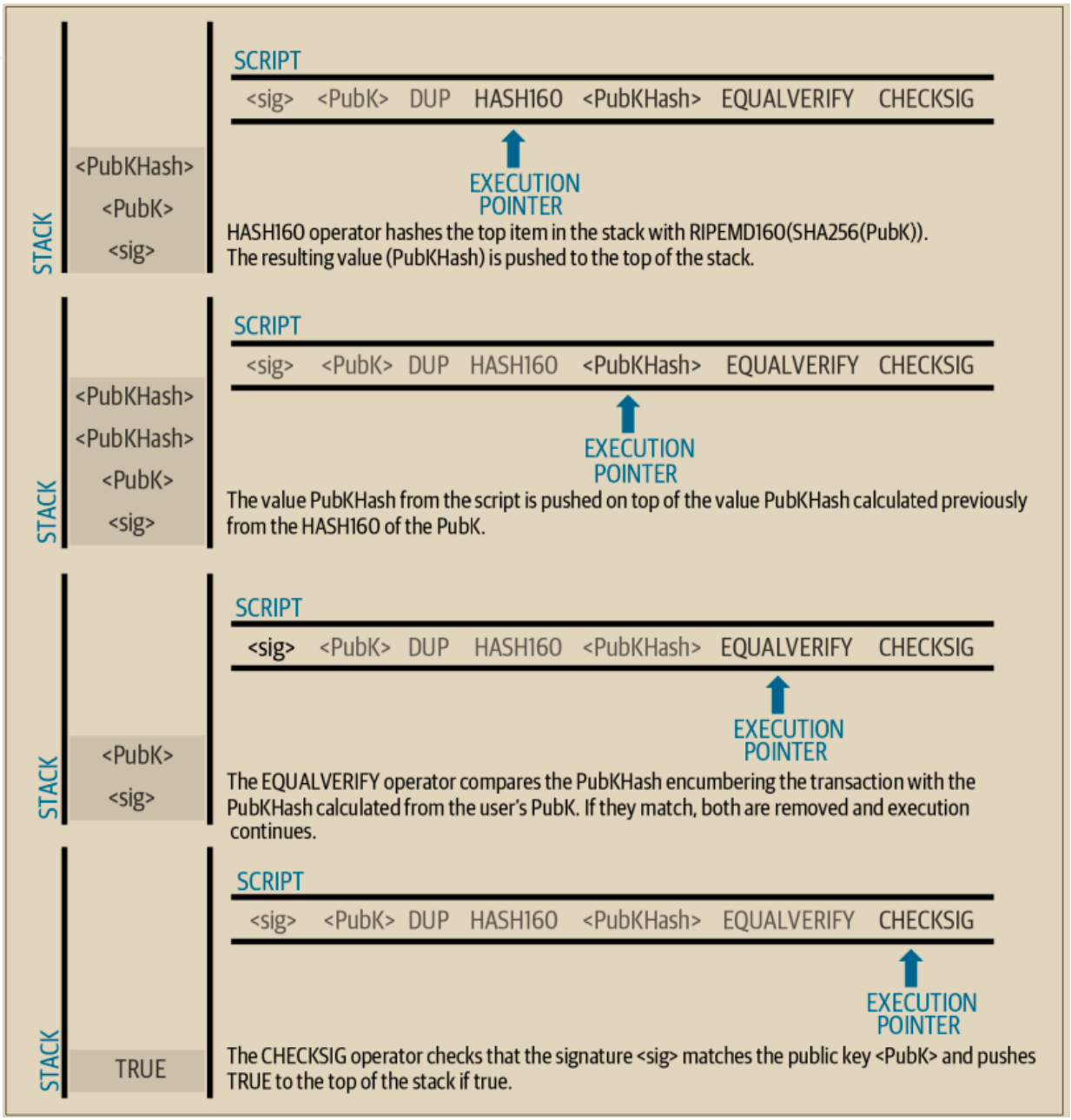


图 7-4. 执行P2PKH交易的脚本（2/2）

脚本化的多重签名

多重签名脚本设置了一个条件，其中 k 个公钥被记录在脚本中，至少其中的 t 个必须提供签名才能花费资金，称为 t-of-k。 例如，一个 2-of-3 多重签名是指列出了三个潜在签名者的公钥，并且至少必须使用其中的两个来为有效交易创建签名以花费资金。

> 一些比特币文档，包括本书早期的版本，使用术语“m-of-n”来表示传统的多重签名。然而，当它们被说出时，“m”和“n”很难区分，因此我们使用替代的 t-of-k。这两个短语都指的是相同类型的签名方案。

设置 t-of-k 多重签名条件的输出脚本的一般形式是：

t \ \ ... \ k OP_CHECKMULTISIG

其中，k 是列出的公钥总数，t 是需要的签名门槛，以花费输出。

一个设置 2-of-3 多重签名条件的输出脚本如下所示：

2 \ \ \ 3 OP_CHECKMULTISIG

前面的输出脚本可以通过包含签名的输入脚本满足：

\ \

或者是来自所列的3个公钥对应的任意两个私钥的签名组合。

两个脚本一起形成组合验证脚本：

\ \ 2 \ \ \ 3 OP_CHECKMULTISIG

执行此组合脚本时，如果输入脚本具有与设置为限制条件的三个公钥中的两个相对应的两个有效签名，则评估结果将为TRUE。

目前，Bitcoin Core的交易中继策略将多重签名输出脚本限制为最多三个列出的公钥，这意味着您可以在范围内执行从1-of-1到3-of-3的任何操作，或者在该范围内的任何组合。您可能希望检查IsStandard()函数，以查看网络当前接受的内容。请注意，三个密钥的限制仅适用于标准（也称为“裸”）多重签名脚本，而不适用于包含在其他结构（如P2SH、P2WSH或P2TR）中的脚本。P2SH多重签名脚本受到政策和共识的限制，最多可包含15个密钥，允许最多15-of-15的多重签名。我们将在“支付到脚本哈希”中了解有关P2SH的信息。所有其他脚本根据每个OP_CHECKMULTISIG或OP_CHECKMULTISIGVERIFY操作码的共识限制为20个密钥，尽管一个脚本可能包含多个这些操作码。\

CHECKMULTISIG执行中的一些怪异情况

\ CHECKMULTISIG执行中存在一个奇怪的问题，需要做一些小的调整。当执行CHECKMULTISIG操作时，它应该消耗t + k + 2个堆栈项作为参数。然而，由于这个奇怪的问题，CHECKMULTISIG会多弹出一个值，或者比预期多一个值。

让我们通过之前的验证示例更详细地了解这个问题：

\ \ 2 \ \ \3 OP_CHECKMULTISIG

首先，OP_CHECKMULTISIG弹出顶部项目，即k（在此示例中为“3”）。然后，它弹出k个项目，即可以签名的公钥；在此示例中，为公钥A、B和C。然后，它弹出一个项目，即t，即需要的签名数（即有多少个签名是必需的）。在这里t = 2。此时，OP_CHECKMULTISIG应该弹出最后的t个项目，即签名，并查看它们是否有效。然而，不幸的是，实现中的一个异常导致OP_CHECKMULTISIG弹出了比应该多一个项目（总共是t + 1）。这个额外的项目被称为虚拟堆栈元素，当检查签名时，它被忽略，因此对OP_CHECKMULTISIG本身没有直接影响。然而，虚拟元素必须存在，因为如果在OP_CHECKMULTISIG尝试在空堆栈上弹出时不存在，它将导致堆栈错误和脚本失败（将交易标记为无效）。由于虚拟元素被忽略，它可以是任何值。早期的惯例是使用OP_0，后来成为一个中继策略规则，并最终成为一个共识规则（通过BIP147的执行）。

由于弹出虚拟元素是共识规则的一部分，因此必须永远复制。因此，脚本应如下所示：

OP_0 \ \ 2 \ \ \ 3 OP_CHECKMULTISIG

因此，实际上在多签名中使用的输入脚本不是：

\ \

而是：

OP_0 \ \

\ 一些人认为这种奇怪的现象是比特币原始代码中的一个错误，但还存在一个合理的替代解释。验证 t-of-k 签名可能需要比 t 或 k 更多的签名检查操作。让我们考虑一个简单的例子，1-of-5，具有以下组合脚本：

\ \ 1 \ \ \ \ 5 OP_CHECKMULTISIG

首先对签名进行与 key0 的比对，然后是 key1，然后再对其前面的其他键进行比对，最后才与其对应的 key4 进行比对。这意味着即使只有一个签名，也需要执行五次签名检查操作。消除这种冗余的一种方法是为 OP_CHECKMULTISIG 提供一个映射，指示提供的签名与哪个公钥对应，从而使 OP_CHECKMULTISIG 操作只执行确切的 t 次签名检查操作。可能比特币的原始开发者在比特币的原始版本中添加了额外的元素（现在称为虚拟堆栈元素），以便他们可以在以后的软分叉中添加传递映射的功能。然而，该功能从未实现，并且2017年对共识规则的BIP147更新使得将来不可能添加该功能。

只有比特币的原始开发者能告诉我们虚拟堆栈元素是一个错误还是一个未来升级的计划。在本书中，我们只是将其称为奇异现象。

从现在开始，如果您看到一个多重签名脚本，您应该期望在开头看到一个额外的 OP_0，它唯一的目的是解决共识规则中的一个奇异现象。

\

支付到脚本哈希（Pay to Script Hash，P2SH）

\ 支付到脚本哈希（Pay to Script Hash，P2SH）于2012年引入，作为一种强大的新型操作，极大地简化了复杂脚本的使用。为了解释P2SH的必要性，让我们看一个实际的例子。

穆罕默德是一家总部位于迪拜的电子产品进口商。穆罕默德的公司广泛使用比特币的多重签名功能来管理其企业账户。多重签名脚本是比特币高级脚本功能中最常见的用法之一，也是一种非常强大的功能。穆罕默德的公司为所有客户支付使用了多重签名脚本。客户支付的任何款项都被锁定，需要至少两个签名才能释放。穆罕默德、他的三位合伙人和他们的律师每人都可以提供一个签名。这样的多重签名方案提供了企业治理控制，并防止了盗窃、侵占或丢失。

由此产生的脚本相当长，看起来像这样：

\ 2 \<Mohammed's Public Key> \ \ \ \ 5 OP_CHECKMULTISIG

\ 虽然多重签名脚本是一种强大的功能，但它们使用起来很麻烦。考虑到前面的脚本，穆罕默德必须在支付之前将该脚本传达给每个客户。每个客户都必须使用具有创建自定义交易脚本能力的特殊比特币钱包软件。此外，由于该脚本包含非常长的公钥，因此生成的交易将比简单的支付交易大约多五倍。额外的数据负担将以额外的交易费用的形式由客户承担。最后，像这样的大型交易脚本将在每个完整节点的UTXO集中保存，直到被花费。所有这些问题使得在实践中使用复杂的输出脚本变得困难。

P2SH的开发旨在解决这些实际困难，并使使用复杂脚本像向单个密钥比特币地址支付一样简单。在P2SH支付中，复杂脚本被替换为一个承诺，即加密哈希的摘要。当稍后尝试花费UTXO的交易被提交时，它必须包含与承诺匹配的脚本，以及满足脚本的数据。简单地说，P2SH意味着“支付到与此哈希匹配的脚本，一个将在此输出被花费时稍后呈现的脚本”。

在P2SH交易中，被哈希替换的脚本被称为赎回脚本，因为它是在赎回时向系统呈现的，而不是作为输出脚本。表7-1显示了不使用P2SH的脚本，表7-2显示了使用P2SH编码的相同脚本。

表 7-1. 不使用P2SH的复杂脚本

Output script	**2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG**
Input script	Sig1 Sig2

表 7-2. 不使用P2SH的复杂脚本

Redeem script	**2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG**
Output script	OP_HASH160 <20-byte hash of redeem script> OP_EQUAL
Input script	Sig1 Sig2

你可以从表格中看到，使用 P2SH，用于描述支出条件（赎回脚本）的复杂脚本并不包含在输出脚本中。相反，输出脚本中只包含其哈希值，而赎回脚本本身会在支出输出时作为输入脚本的一部分呈现。这将费用和复杂性的负担从交易发起者转移到了交易接收者身上。

让我们来看看 Mohammed 公司的情况，以及用于所有客户付款的复杂多重签名脚本及其生成的 P2SH 脚本。

首先，是 Mohammed 公司用于所有客户付款的多重签名脚本：

2 \<Mohammed's Public Key> \ \

\ \ 5 OP_CHECKMULTISIG

整个脚本可以通过首先应用 SHA256 哈希算法，然后在结果上应用 RIPEMD-160 算法来表示为一个 20 字节的加密哈希。例如，以 Mohammed 的赎回脚本的哈希值为起点：

54c557e07dde5bb6cb791c7a540e0a4796f5e97e

一个 P2SH 交易将输出锁定到此哈希值，而不是较长的赎回脚本，使用了一个特殊的输出脚本模板：

```
OP_HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e OP_EQUAL
```

正如您所看到的，这个输出脚本要短得多。与“支付给这个 5 个密钥的多重签名脚本”相比，P2SH 等效交易是“支付给一个具有此哈希的脚本”。客户向 Mohammed 的公司付款时只需在付款中包含这个更短的输出脚本。当 Mohammed 和他的合作伙伴想要花费这个 UTXO 时，他们必须呈现原始的赎回脚本（锁定该 UTXO 的哈希）以及解锁它所需的签名，就像这样：

\ \ <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG>

两个脚本在两个阶段进行合并。首先，检查赎回脚本与输出脚本是否匹配：

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

如果赎回脚本的哈希值匹配，则执行赎回脚本：

\ \ 2 \ \ \ \ \ 5 OP_CHECKMULTISIG\

P2SH地址

P2SH功能的另一个重要部分是将脚本哈希编码为地址，这在BIP13中定义。P2SH地址是脚本的20字节哈希的base58check编码，就像比特币地址是公钥的20字节哈希的base58check编码一样。P2SH地址使用版本前缀“5”，这导致了以“3”开头的base58check编码地址。

例如，Mohammed的复杂脚本，经过哈希和base58check编码成为P2SH地址，变成了39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw。

现在，Mohammed可以将这个“地址”提供给他的客户，他们可以使用几乎任何比特币钱包进行简单支付，就像对任何其他比特币地址进行支付一样。前缀3提示他们这是一种特殊类型的地址，对应于脚本而不是公钥，但除此之外，它的工作方式与对任何其他比特币地址的支付完全相同。

P2SH地址隐藏了所有复杂性，因此进行支付的人看不到脚本。

P2SH的优点

P2SH功能相比直接在输出中使用复杂脚本具有以下优势：

• 与原始遗留地址的相似性意味着发送者和发送者的钱包不需要复杂的工程来实现P2SH。

• P2SH将长脚本的数据存储负担从输出（除了存储在区块链上之外，还在UTXO集合中）转移到输入（仅存储在区块链上）。

• P2SH将长脚本的数据存储负担从当前时间（支付）转移到将来时间（在其被花费时）。

• P2SH将长脚本的交易费用成本从发送者转移到收件人，后者必须包含长的赎回脚本才能花费它。

赎回脚本和验证

您不能在P2SH赎回脚本中放置一个P2SH，因为P2SH规范不支持递归。此外，虽然技术上可以在赎回脚本中包含OP_RETURN（参见“数据记录输出（OP_RETURN）”），因为规则中没有阻止这样做，但实际上没有任何实际用途，因为在验证期间执行OP_RETURN会导致交易被标记为无效。

请注意，因为直到您尝试花费P2SH输出时才会向网络展示赎回脚本，所以如果您创建一个带有无效赎回脚本哈希的输出，您将无法花费它。包含赎回脚本的花费交易将不会被接受，因为它是一个无效的脚本。这会带来风险，因为您可以将比特币发送到一个以后无法花费的P2SH地址。

> P2SH输出脚本包含赎回脚本的哈希，这不会给出有关赎回脚本内容的线索。即使赎回脚本无效，P2SH输出也会被视为有效并被接受。您可能会意外地以一种无法稍后花费的方式接收比特币。

数据记录输出（OP_RETURN）

比特币的分布式和时间戳的区块链具有超越支付的潜力。许多开发人员尝试利用交易脚本语言，利用系统的安全性和弹性进行数字公证服务等应用。早期尝试使用比特币的脚本语言进行这些用途的做法包括创建记录数据到区块链的交易输出；例如，以一种任何人都能通过引用该交易来建立文件存在的证据的方式记录对文件的承诺，以在特定日期证明该文件的存在。

\ 使用比特币的区块链存储与比特币支付无关的数据是一个颇具争议的议题。许多人认为这种使用是滥用的，希望予以遏制。而另一些人则将其视为区块链技术强大功能的展示，并希望鼓励这种实验。反对包含非支付数据的人士认为，这给运行完整比特币节点的人带来了负担，因为他们需要承担为区块链不打算承载的数据而进行的磁盘存储成本。此外，这样的交易可能会创建无法被花费的UTXO，使用传统比特币地址作为自由格式的20字节字段。由于该地址用于存储数据，它不对应于私钥，因此生成的UTXO永远无法被花费；这是一个虚假的支付。因此，永远无法被花费的这些交易从未从UTXO集中移除，并导致UTXO数据库的大小永远增加，或者说“膨胀”。

达成的一个妥协是允许以OP_RETURN开头的输出脚本向交易输出添加非支付数据。然而，与“虚假”UTXO的使用不同，OP_RETURN操作符创建的是明确可证明无法花费的输出，不需要存储在UTXO集中。OP_RETURN输出被记录在区块链上，因此它们占用磁盘空间并导致区块链大小的增加，但它们不存储在UTXO集中，因此不会使完整节点承担更昂贵的数据库操作成本。

OP_RETURN脚本如下所示：

OP_RETURN \

数据部分通常代表一个哈希值，比如来自SHA256算法的输出（32字节）。一些应用程序在数据前面添加前缀以帮助识别该应用程序。例如，证明存在数字公证服务使用了8字节的前缀DOCPROOF，其ASCII编码为44 4f 43 50 52 4f 4f 46（十六进制）。

请记住，没有与OP_RETURN相对应的输入脚本，因此无法用于“花费”OP_RETURN输出。OP_RETURN输出的整个意义在于，您无法花费锁定在该输出中的资金，因此不需要将其保存在UTXO集中作为可能花费的金额：OP_RETURN输出可以明确证明无法花费。OP_RETURN输出通常具有零金额，因为分配给此类输出的任何比特币实际上都永远丢失了。如果在交易中将OP_RETURN输出作为输入引用，脚本验证引擎将停止执行验证脚本，并将交易标记为无效。执行OP_RETURN本质上会导致脚本“RETURN”并终止。因此，如果您意外地将OP_RETURN输出作为交易的输入引用，则该交易将无效。

交易锁定时间的限制

使用锁定时间允许花费者限制一个交易直到特定的区块高度才能被包含在区块中，但它并不阻止在那之前的另一个交易中花费这些资金。让我们通过以下示例来解释这一点。

Alice签署了一笔交易，将她的一个输出支付到Bob的地址，并将交易锁定时间设置为未来3个月。Alice将这笔交易发送给Bob保管。通过这笔交易，Alice和Bob知道：

- Bob在3个月之内不能传输这笔交易以赎回资金。
- Bob可以在3个月后传输这笔交易。

然而：

- Alice可以创建一笔冲突的交易，花费相同的输入但没有锁定时间。因此，Alice可以在3个月之前花费相同的UTXO。
- Bob无法保证Alice不会这样做。

了解交易锁定时间的限制是很重要的。唯一的保证是在3个月之前Bob将无法赎回预先签名的交易。没有保证Bob会得到资金。确保Bob会收到资金但不能在3个月之前花费它们的一种方法是将时锁限制作为脚本的一部分放置在UTXO本身上，而不是在交易上。这是通过下一个形式的时锁实现的，称为检查锁定时间验证。

检查锁定时间验证（OP_CLTV）

2015年12月，一种新形式的时间锁被引入比特币作为软分叉升级。根据BIP65规范，新增了一个名为OP_CHECK LOCKTIMEVERIFY（OP_CLTV）的脚本操作符。OP_CLTV是一种基于输出的时间锁，而不是基于交易的时间锁，这与锁定时间不同。这样做可以增加时间锁的应用灵活性。

简单来说，通过在输出中使用OP_CLTV操作码，该输出受限制，只能在指定的时间过去后才能花费。

OP_CLTV并不取代锁定时间，而是限制特定的UTXO，使它们只能在具有设置为大于或等于指定值的锁定时间的未来交易中花费。

\ OP_CLTV操作码接受一个参数作为输入，以与锁定时间相同的格式表示（可以是区块高度或Unix纪元时间）。如同VERIFY后缀所示，OP_CLTV是一种如果结果为FALSE就会停止脚本执行的类型的操作码。如果结果为TRUE，则继续执行。

要使用OP_CLTV，您需要将其插入到创建输出的交易中输出的赎回脚本中。例如，如果Alice要支付Bob，他可能通常会接受以下P2SH脚本支付：

\ \<Bob's public key> OP_CHECKSIG

为了将其锁定到一个时间，比如说从现在开始的3个月后，他的P2SH脚本将改为：

\<Bob's pubkey> OP_CHECKSIGVERIFY \ OP_CHECKLOCKTIMEVERIFY

其中 \ 是一个块高度或时间值，从交易被挖掘时起预计的3个月后的值：当前块高度 + 12,960（块）或当前 Unix 时间 + 7,760,000（秒）。

当 Bob 尝试花费这个 UTXO 时，他构造一个引用该 UTXO 作为输入的交易。他在该输入的输入脚本中使用他的签名和公钥，并将交易锁定时间设置为等于或大于 Alice 设置的 OP_CHECKLOCKTIMEVERIFY 的 timelock。然后 Bob 将交易广播到比特币网络。

Bob 的交易如下评估。如果 Alice 设置的 OP_CHECKLOCKTIMEVERIFY 参数小于或等于花费交易的锁定时间，则脚本继续执行（就好像执行了无操作或 OP_NOP 操作码一样）。否则，脚本执行停止，并且交易被视为无效。

更确切地说，BIP65 解释了如果发生以下情况，OP_CHECKLOCKTIMEVERIFY 将失败并停止执行：

• 栈为空。

• 栈顶项小于 0。

• 栈顶项的锁定时间类型（高度与时间戳）与锁定时间字段不相同。

• 栈顶项大于交易的锁定时间字段。

• 输入的序列字段为 0xffffffff。

> **时间锁冲突**
>
> OP_CLTV和lock time使用相同的格式来描述timelocks，即块高度或自Unix纪元以来经过的秒数。关键是，当两者一起使用时，锁定时间的格式必须与输出中的OP_CLTV匹配——它们必须都引用块高度或以秒为单位的时间。
>
> 这意味着如果脚本必须执行两次不同的OP_CLTV调用，一次使用高度，一次使用时间，那么该脚本永远不会有效。在编写高级脚本时可能会犯这种错误，因此请务必在测试网络上彻底测试您的脚本，或者使用专为防止此问题而设计的工具，例如Miniscript编译器。
>
> 另一个影响是，在交易的任何脚本中只能使用一种类型的OP_CLTV。如果一个输入的脚本使用高度类型，而另一个输入的不同脚本使用时间类型，那么就无法构建一个有效的交易来花费这两个输入。

\ 在执行后，如果OP_CLTV得到满足，那么它之前的参数将保持为堆栈中的顶部项目，并且可能需要使用OP_DROP将其移除，以便正确执行后续的脚本操作码。出于这个原因，您经常会看到脚本中的OP_CHECKLOCKTIMEVERIFY后面跟着OP_DROP。OP_CLTV和OP_CSV（参见“相对Timelocks”）不同于其他CHECKVERIFY操作码，因为它们保留了堆栈上的项目。这是因为引入它们的软分叉重新定义了旧的操作码，而这些旧的操作码不会丢弃堆栈中的项目，因此必须保留这些以前的无操作（NOP）操作码的行为。

通过将锁定时间与OP_CLTV结合使用，页面158中描述的情景将发生改变。“交易锁定时间限制”中所述的情况。Alice立即发送她的交易，将资金分配给Bob的密钥。Alice不再能够花费这笔钱，但Bob在3个月的锁定时间到期之前也不能花费它。

通过直接将Timelock功能引入脚本语言，OP_CLTV使我们能够开发一些非常有趣的复杂脚本。该标准在BIP65（OP_CHECKLOCKTIMEVERIFY）中定义。\

相对时间锁

锁定时间和OP_CLTV都是绝对时间锁定，因为它们指定了一个绝对的时间点。我们将要检查的下两个时间锁定功能是相对时间锁定，因为它们指定了从区块链中确认输出以来的经过时间作为花费输出的条件。

相对时间锁定很有用，因为它们允许在一个交易上施加一个时间约束，这个约束取决于从之前交易的确认开始的经过时间。换句话说，时钟直到UTXO被记录在区块链上才开始计时。这种功能在双向状态通道和闪电网络（LN）中特别有用，我们将在“支付通道和状态通道”中看到。

相对时间锁定与绝对时间锁定一样，都有交易级别的特性和脚本级别的操作码实现。交易级别的相对时间锁定是通过对序列值的共识规则实现的，序列值是设置在每个交易输入中的一个字段。脚本级别的相对时间锁定是通过OP_CHECKSEQUENCEVERIFY（OP_CSV）操作码实现的。

相对时间锁定是根据BIP68《使用强制共识序列号的相对锁定时间》和BIP112《OP_CHECKSEQUENCEVERIFY》的规范实现的。

BIP68和BIP112于2016年5月作为共识规则的软分叉升级激活。

使用OP_CSV的相对时间锁定

与OP_CLTV和锁定时间一样，有一个用于相对时间锁定的脚本操作码，利用了脚本中的序列值。这个操作码称为OP_CHECKSEQUENCEVERIFY，简称为OP_CSV。

当在UTXO的脚本中评估OP_CSV操作码时，只允许在输入序列值大于或等于OP_CSV参数的交易中花费。基本上，这限制了在相对于UTXO被挖掘的时间已经过去一定数量的区块或秒数之前花费UTXO。

与CLTV一样，OP_CSV中的值必须与相应序列值的格式匹配。如果OP_CSV以区块为单位指定，那么序列值也必须是这样。如果OP_CSV以秒为单位指定，那么序列值也必须是这样。

> 执行多个OP_CSV操作码的脚本必须只使用相同的类型，即基于时间或基于高度。混合类型将产生一个无法花费的无效脚本，这与我们在“时间锁定冲突”中看到的OP_CLTV的问题相同。 然而，OP_CSV允许将任意两个有效的输入包含在同一笔交易中，因此在输入之间发生交互的问题，这在OP_CLTV中出现，不会影响OP_CSV。

使用OP_CSV的相对时间锁定在创建并签名但未传播的多个（链式）交易时特别有用，即它们被保存在区块链之外（离线）。在父交易传播、被挖掘并按照相对时间锁定中指定的时间经过一段时间后，子交易才能被使用。这种用例的一个应用显示在“支付通道和状态通道”中以及“路由支付通道（闪电网络）”中。

OP_CSV在BIP112，CHECKSEQUENCEVERIFY中有详细定义。

带控制流的脚本（条件子句）

比特币脚本中的一个更强大的特性是流程控制，也称为条件子句。你可能熟悉使用IF...THEN...ELSE构造的各种编程语言中的流程控制。比特币的条件子句看起来有些不同，但实质上是相同的构造。

在基本层面上，比特币的条件操作码允许我们构造一个脚本，根据逻辑条件的TRUE/FALSE结果有两种解锁方式。例如，如果x为TRUE，则执行代码路径为A，否则执行代码路径为B。

此外，比特币的条件表达式可以“嵌套”无限，意味着一个条件子句可以包含另一个条件子句，后者又包含另一个条件子句，以此类推。比特币脚本流程控制可用于构造具有数百种可能执行路径的非常复杂的脚本。嵌套没有限制，但共识规则对脚本的最大字节大小施加了限制。

比特币使用OP_IF、OP_ELSE、OP_ENDIF和OP_NOTIF操作码实现流程控制。此外，条件表达式可以包含布尔运算符，如OP_BOOLAND、OP_BOOLOR和OP_NOT。

乍一看，你可能会觉得比特币的流程控制脚本很令人困惑。这是因为比特币脚本是一种栈语言。就像1 + 1被表达为1 1 OP_ADD一样“向后”，在比特币中的流程控制子句也看起来“向后”。

在大多数传统（过程式）编程语言中，流程控制看起来像这样：

```
if (condition):
 code to run when condition is true
else:
 code to run when condition is false
endif
code to run in either case
```

在像比特币脚本这样的基于栈的语言中，逻辑条件位于IF之前，这使得它看起来“向后”：

```
condition
IF
 code to run when condition is true
OP_ELSE
 code to run when condition is false
OP_ENDIF
code to run in either case
```

在阅读比特币脚本时，请记住正在评估的条件出现在IF操作码之前。

带有VERIFY操作码的条件子句

另一种比特币脚本中的条件形式是以VERIFY结尾的任何操作码。VERIFY后缀意味着如果评估的条件不为TRUE，则脚本的执行立即终止，并将事务视为无效。

与提供替代执行路径的IF子句不同，VERIFY后缀充当了一个守卫子句，只有在满足前提条件时才继续执行。

例如，以下脚本要求Bob的签名和产生特定哈希的预图像（秘密）。只有满足这两个条件才能解锁：

OP_HASH160 \ OP_EQUALVERIFY \<Bob's Pubkey> OP_CHECKSIG

为了花费这笔交易，Bob必须提供一个有效的预影像和一个签名：

\<Bob's Sig> \\

没有提供预影像，Bob无法进入检查他签名的脚本部分。

这个脚本也可以用OP_IF来编写：

```
OP_HASH160 <expected hash> OP_EQUAL
OP_IF
 <Bob's Pubkey> OP_CHECKSIG
OP_ENDIF
```

Bob的身份验证数据是相同的：

\ \<Bob's Sig> \

使用OP_IF的脚本与使用带有VERIFY后缀的操作码执行相同的操作；它们都作为守卫子句运行。然而，VERIFY的构造更高效，使用了两个较少的操作码。

那么，我们什么时候使用VERIFY，什么时候使用OP_IF呢？如果我们只是想要附加一个前提条件（守卫子句），那么VERIFY更好。然而，如果我们想要有多个执行路径（流程控制），那么我们需要一个OP_IF…OP_ELSE的流程控制子句。

在脚本中使用流程控制

在比特币脚本中使用流程控制的一个非常常见的用法是构建一个脚本，提供多个执行路径，每个路径都是赎回UTXO的不同方式。

让我们看一个简单的例子，我们有两个签名者，Alice 和 Bob，任何一个都能赎回。使用多重签名，这可以表示为一个 1-of-2 多重签名脚本。为了演示起见，我们将使用一个 OP_IF 子句完成相同的操作：

```
OP_IF
 <Alice's Pubkey>
OP_ELSE
 <Bob's Pubkey>
OP_ENDIF
OP_CHECKSIG
```

看着这个赎回脚本，你可能会问：“条件在哪里？在IF子句之前没有任何东西！”

条件不是脚本的一部分。相反，条件将在花费时提供，允许Alice和Bob“选择”他们想要的执行路径：

\<Alice's Sig> OP_TRUE

OP_TRUE作为条件（TRUE），将使OP_IF子句执行第一个赎回路径。这个条件将Alice拥有签名的公钥放入堆栈中。OP_TRUE操作码，也称为OP_1，会将数字1推送到堆栈中。

要使Bob赎回此项，他必须选择OP_IF中的第二个执行路径，给出一个FALSE值。OP_FALSE操作码，也称为OP_0，将一个空字节数组推送到堆栈中：

\<Bob's Sig> OP_FALSE

Bob的输入脚本导致OP_IF子句执行第二个（OP_ELSE）脚本，该脚本需要Bob的签名。

由于OP_IF子句可以嵌套，我们可以创建一个执行路径的“迷宫”。输入脚本可以提供一个“地图”，选择实际执行的执行路径：

```
OP_IF
 subscript A
OP_ELSE
 OP_IF
 subscript B
 OP_ELSE
 subscript C
 OP_ENDIF
OP_ENDIF
```

在这种情况下，有三条执行路径（子脚本A、子脚本B和子脚本C）。输入脚本以一系列TRUE或FALSE值的形式提供路径。例如，要选择子脚本B，输入脚本必须以OP_1 OP_0（TRUE，FALSE）结尾。这些值将被推送到堆栈上，使得第二个值（FALSE）位于堆栈的顶部。外部OP_IF子句弹出FALSE值并执行第一个OP_ELSE子句。然后，TRUE值移到堆栈的顶部，并由内部（嵌套的）OP_IF评估，选择B执行路径。

使用这种结构，我们可以构建具有数十或数百个执行路径的赎回脚本，每条路径提供一种不同的赎回UTXO的方式。要花费，我们构建一个输入脚本，通过在每个流程控制点上将适当的TRUE和FALSE值放入堆栈来导航执行路径。

复杂脚本示例

在本节中，我们将本章中的许多概念结合到一个示例中。

穆罕默德是迪拜的一家公司的所有者，经营着一家进出口公司；他希望建立一个具有灵活规则的公司资本账户。他创建的方案根据时间锁定需要不同级别的授权。多签名方案的参与者是穆罕默德，他的两个合作伙伴赛义德和扎伊拉，以及他们的公司律师。这三个合作伙伴根据多数原则做出决定，因此三个中的两个必须同意。但是，如果他们的密钥出现问题，他们希望他们的律师能够在三个合作伙伴签名中的任何一个的情况下恢复资金。最后，如果所有合作伙伴在一段时间内不可用或无能为力，他们希望律师在获得对资本账户的交易记录的访问权限后能够直接管理该账户。

示例 7-1 是穆罕默德设计的赎回脚本，以实现这一目标（行号已添加前缀）。

示例 7-1. 可变多重签名与时间锁定

```
01 OP_IF
02 OP_IF
03 2
04 OP_ELSE
05 <30 days> OP_CHECKSEQUENCEVERIFY OP_DROP
06 <Lawyer's Pubkey> OP_CHECKSIGVERIFY
07 1
08 OP_ENDIF
09 <Mohammed's Pubkey> <Saeed's Pubkey> <Zaira's Pubkey> 3 OP_CHECKMULTISIG
10 OP_ELSE
11 <90 days> OP_CHECKSEQUENCEVERIFY OP_DROP
12 <Lawyer's Pubkey> OP_CHECKSIG
13 OP_ENDIF
```

Mohammed的脚本使用嵌套的OP_IF...OP_ELSE流程控制子句实现了三条执行路径。

在第一条执行路径中，该脚本作为一个简单的2-of-3多重签名与三个合作伙伴一起操作。这个执行路径包括第3行和第9行。第3行将多重签名的法定人数设置为2（2-of-3）。可以通过在输入脚本末尾放置OP_TRUE OP_TRUE来选择此执行路径：

OP_0 \ \ OP_TRUE OP_TRUE

> 这个输入脚本开头的OP_0是因为OP_CHECKMULTISIG存在一个奇怪的特性，它会从栈中弹出一个额外的值。虽然OP_CHECKMULTISIG会忽略这个额外的值，但它必须存在，否则脚本会失败。使用OP_0推送一个空字节数组是对这个奇怪特性的一种变通方法，详情请参见第152页的“CHECKMULTISIG执行中的一个奇特现象”。

第二个执行路径只能在从UTXO创建起经过30天后才能使用。在那时，它需要律师的签名和三个合作伙伴中的一个签名（即1-of-3的多重签名）。这通过第7行实现，该行将多重签名的法定人数设置为1。要选择这个执行路径，输入脚本的结尾应为OP_FALSE OP_TRUE：

OP_0 \ \ OP_FALSE OP_TRUE

> 为什么是 OP_FALSE OP_TRUE？这不是倒过来了吗？首先将 FALSE 推送到堆栈上，然后在其上方推送 TRUE。因此，第一个 OP_IF 操作码首先弹出 TRUE。

最后，第三个执行路径允许律师独自花费资金，但只能在90天后。要选择此执行路径，输入脚本必须以 OP_FALSE 结束：

\<Lawyer's Sig> OP_FALSE

尝试在纸上运行脚本，看看它在堆栈上的行为。

隔离见证输出和交易示例

让我们来看一些示例交易，并了解它们在隔离见证下的变化。首先，我们将看看如何将P2PKH支付作为隔离见证程序完成。然后，我们将研究P2SH脚本的隔离见证等效物。最后，我们将看看如何将前述隔离见证程序嵌入P2SH脚本中。

支付给见证公钥哈希（P2WPKH）

让我们首先看一下一个P2PKH输出脚本的示例：

OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7

OP_EQUALVERIFY OP_CHECKSIG

有了隔离见证，Alice会创建一个P2WPKH脚本。如果该脚本与相同的公钥进行了绑定，它将如下所示：

0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7

如您所见，P2WPKH输出脚本比P2PKH等价脚本简单得多。它由两个值组成，推送到脚本评估堆栈上。对于旧的（不支持隔离见证的）比特币客户端，这两次推送看起来像是任何人都可以使用的输出。对于较新的、支持隔离见证的客户端，第一个数字（0）被解释为版本号（见证版本），而第二部分（20字节）是一个见证程序。这20字节的见证程序简单地是公钥的哈希值，就像P2PKH脚本中一样。

现在，让我们看一下Bob用来花费这个输出的相应交易。对于原始脚本，花费交易必须在交易输入中包含一个签名：

```
[...]
"vin" : [
 "txid": "abcdef12345...",
 "vout": 0,
 "scriptSig": “<Bob’s scriptSig>”,
]
[...]
```

然而，要花费P2WPKH输出，交易在该输入上没有签名。相反，Bob的交易有一个空的输入脚本，并包含一个见证结构：

```
[...]
"vin" : [
 "txid": "abcdef12345...",
 "vout": 0,
 "scriptSig": “”,
]
[...]
“witness”: “<Bob’s witness structure>”
[...]
```

P2WPKH类型钱包构建

P2WPKH见证程序应该只由接收方创建，而不应由支出方从已知的公钥、P2PKH脚本或地址进行转换。支出方无法知道接收方的钱包是否具有构建隔离见证交易和花费P2WPKH输出的能力。

此外，P2WPKH输出必须由压缩公钥的哈希构建。未压缩的公钥在隔离见证中是非标准的，并且可能会在未来的软分叉中被明确禁用。如果P2WPKH中使用的哈希来自未压缩的公钥，则可能无法花费，并且您可能会丢失资金。P2WPKH输出应由收款方的钱包通过从其私钥派生压缩公钥来创建。

P2WPKH应该由接收方通过将压缩的公钥转换为P2WPKH哈希来构建。支出方或其他任何人都不应该将P2PKH脚本、比特币地址或未压缩的公钥转换为P2WPKH见证脚本。一般来说，支出方应该按照接收方指示的方式发送。

支付给见证脚本哈希 (P2WSH)

segwit v0见证程序的第二种类型对应于P2SH脚本。我们在“支付到脚本哈希”中看到了这种类型的脚本。在该示例中，P2SH被Mohammed的公司用来表示多签名脚本。向Mohammed的公司的付款被编码为以下脚本：

OP_HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e OP_EQUAL

这个P2SH脚本引用了一个赎回脚本的哈希，该赎回脚本定义了一个2-of-3多重签名的要求来花费资金。要花费这个输出，Mohammed的公司将呈现赎回脚本（其哈希与P2SH输出中的脚本哈希匹配）以及满足该赎回脚本所需的签名，全部都在交易输入中：

```
[...]
"vin" : [
 "txid": "abcdef12345...",
 "vout": 0,
 "scriptSig": “<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 OP_CHECKMULTISIG>”,
]
```

现在，让我们看看如何将整个示例升级为segwit v0。如果Mohammed的客户正在使用兼容segwit的钱包进行付款，他们将创建一个P2WSH输出，看起来像这样：

0 a9b7b38d972cabc7961dbfbcb841ad4508d133c47ba87457b4a0e8aae86dbb89

与P2WPKH示例一样，你可以看到隔离见证等效脚本要简单得多，减少了P2SH脚本中的模板开销。相反，隔离见证输出脚本由两个值推送到堆栈中：一个见证版本（0）和见证脚本的32字节SHA256哈希（见证程序）。

虽然P2SH使用了20字节的RIPEMD160(SHA256(script))哈希，但P2WSH见证程序使用了32字节的SHA256(script)哈希。这种哈希算法选择上的差异是有意为之的，以在某些用例中提供更强的安全性（P2WSH提供128位的安全性，而P2SH提供80位的安全性）。详情请参阅“P2SH碰撞攻击”（第73页）。

\ 穆罕默德的公司可以通过呈现正确的见证脚本和足够的签名来消费P2WSH输出。见证脚本和签名将作为见证结构的一部分包含在其中。不会将任何数据放入输入脚本，因为这是一个原生见证程序，不使用遗留的输入脚本字段：

```
[...]
"vin" : [
 "txid": "abcdef12345...",
 "vout": 0,
 "scriptSig": “”,
]
[...]
“witness”: “<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 OP_CHECKMULTISIG>”
[...]
```

P2WPH和P2WSH之间的差异

\ 在前两节中，我们展示了两种类型的见证程序：“支付给见证公钥哈希（P2WPKH）”和“支付给见证脚本哈希（P2WSH）”。这两种类型的见证程序都由相同的版本号后跟数据推送组成。它们看起来非常相似，但解释方式有很大不同：其中一种被解释为公钥哈希，需要用签名来满足，而另一种被解释为脚本哈希，需要用见证脚本来满足。它们之间的关键区别在于见证程序的长度：

- P2WPKH中的见证程序为20字节。
- P2WSH中的见证程序为32字节。

这是区分两种见证程序类型的唯一差异。通过查看哈希的长度，节点可以确定它是P2WPKH还是P2WSH类型的见证程序。

升级到隔离见证

从前面的例子中我们可以看到，升级到隔离见证是一个两步过程。首先，钱包必须创建隔离见证类型的输出。然后，这些输出可以被知道如何构建隔离见证交易的钱包所消费。在这些例子中，Alice的钱包能够创建支付隔离见证输出脚本的输出。Bob的钱包也支持隔离见证，并且能够消费这些输出。 隔离见证被实现为向后兼容的升级，旧版和新版客户端可以共存。钱包开发人员独立升级了钱包软件以添加隔离见证功能。遗留的P2PKH和P2SH继续为未升级的钱包工作。这留下了两个重要的场景，在下一节中会讨论到：

- 一个没有隔离见证功能的付款方钱包能够向能够处理隔离见证交易的接收方钱包进行支付。
- 一个有隔离见证功能的付款方钱包能够通过地址识别和区分能够处理隔离见证交易的接收方和不能处理的接收方。

嵌入隔离见证到 P2SH

\ 假设，例如，Alice 的钱包尚未升级为隔离见证，但 Bob 的钱包已经升级并可以处理隔离见证交易。Alice 和 Bob 可以使用传统的非隔离见证输出。但是 Bob 可能希望使用隔离见证来减少交易费用，利用见证结构的成本降低。

在这种情况下，Bob 的钱包可以构建一个包含隔离见证脚本的 P2SH 地址。Alice 的钱包可以向其进行支付，而无需了解隔离见证。然后 Bob 的钱包可以使用隔离见证交易来花费这笔支付，从而利用隔离见证并减少交易费用。

P2WPKH 和 P2WSH 这两种形式的见证脚本都可以嵌入到一个 P2SH 地址中。第一种被称为嵌套 P2WPKH，第二种被称为嵌套 P2WSH。

嵌套的支付见证公钥哈希

嵌套的支付见证公钥哈希（Nested P2WPKH）是我们将要检查的第一种输出脚本形式。这是一种支付给见证公钥哈希的见证程序，嵌入在支付给脚本哈希脚本中，以便不了解隔离见证的钱包可以支付输出脚本。

Bob 的钱包构建了一个 P2WPKH 见证程序，其中包含 Bob 的公钥。然后对该见证程序进行哈希处理，得到的哈希值被编码为一个 P2SH 脚本。这个 P2SH 脚本被转换为比特币地址，即以“3”开头的地址，就像我们在“支付给脚本哈希”中看到的一样。

Bob's wallet首先使用我们之前看到的P2WPKH见证版本和见证程序：

0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7

\ 数据包括见证版本和Bob的20字节公钥哈希。Bob的钱包然后对数据进行哈希，首先使用SHA256，然后使用RIPEMD-160，产生另一个20字节的哈希值。接下来，赎回脚本哈希被转换为比特币地址。最后，Alice的钱包可以向37Lx99uaGn5avKBxiW26HjedQE3LrDCZru支付款项，就像向任何其他比特币地址支付一样。

为了支付Bob，Alice的钱包会使用P2SH脚本锁定输出：

OP_HASH160 3e0547268b3b19288b3adef9719ec8659f4b2b0b OP_EQUAL

即使Alice的钱包不支持SegWit，它创建的支付也可以由Bob通过SegWit交易进行花费。

嵌套的支付见证脚本哈希

\ 同样，可以将P2WSH见证程序（用于多签名脚本或其他复杂脚本）嵌入到P2SH脚本和地址中，从而使任何钱包都能进行与SegWit兼容的支付。

正如我们在“支付见证脚本哈希（P2WSH）”中所看到的，Mohammed的公司正在使用分隔见证支付多重签名脚本。为了让任何客户都能向他的公司付款，无论他们的钱包是否升级为SegWit，Mohammed的钱包都可以将P2WSH见证程序嵌入到P2SH脚本中。

首先，Mohammed的钱包使用SHA256对见证脚本进行哈希（仅一次），生成哈希值：

9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73

接下来，将哈希的见证脚本转换为带有版本前缀的P2WSH见证程序：

0 9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73

然后，见证程序本身使用SHA256和RIPEMD-160进行哈希运算，生成一个新的20字节哈希：

86762607e8fe87c0c37740cddee880988b9455b2

接下来，钱包使用这个哈希构造一个P2SH比特币地址：

3Dwz1MXhM6EfFoJChHCxh1jWHb8GQqRenG

现在，即使没有支持隔离见证的客户也可以向这个地址进行付款。要向Mohammed发送付款，一个钱包会使用以下的P2SH脚本锁定输出

OP_HASH160 86762607e8fe87c0c37740cddee880988b9455b2 OP_EQUAL

然后，Mohammed的公司可以构建隔离见证交易来花费这些支付，利用隔离见证功能，包括更低的交易费用。

默克尔替代脚本树(MAST)

使用OP_IF，您可以授权多个不同的花费条件，但这种方法具有一些不可取的方面：

\ 权重（成本）

您添加的每个条件都会增加脚本的大小，从而增加交易的权重和需要支付的费用，以便花费受该脚本保护的比特币。

大小受限

即使您愿意支付额外的条件，脚本中可放置的最大数量也是有限的。例如，传统脚本受到10,000字节的限制，实际上最多只能限制您使用几百个条件分支。 即使您可以创建与整个区块大小相同的脚本，它仍然只能包含大约20,000个有用的分支。这对于简单的支付来说已经很多了，但与比特币的某些想象中的用途相比却微不足道。

缺乏隐私

您添加到脚本中的每个条件在花费受该脚本保护的比特币时都会成为公开信息。例如，当有人从示例7-1中的脚本中花费时，Mohammed的律师和业务伙伴将能够看到整个脚本。这意味着即使他们的律师不需要签名，他仍然可以追踪所有他们的交易。

然而，比特币已经使用了一种称为默克尔树的数据结构，它允许验证一个元素是否是集合的成员，而无需识别集合中的每个其他成员。

我们将在“默克尔树”（第252页）中更多了解默克尔树，但基本信息是，我们想要的数据集的成员（例如，任意长度的授权条件）可以传递到哈希函数中以创建一个短的承诺（称为默克尔树的叶子）。然后，每个叶子与另一个叶子配对，并再次进行哈希运算，创建对叶子的承诺，称为分支承诺。对分支的承诺可以以同样的方式创建。这一步对分支重复进行，直到只剩下一个标识符，称为默克尔根。

使用我们在示例7-1中的示例脚本，我们为图7-5中的三个授权条件构建了一个默克尔树。

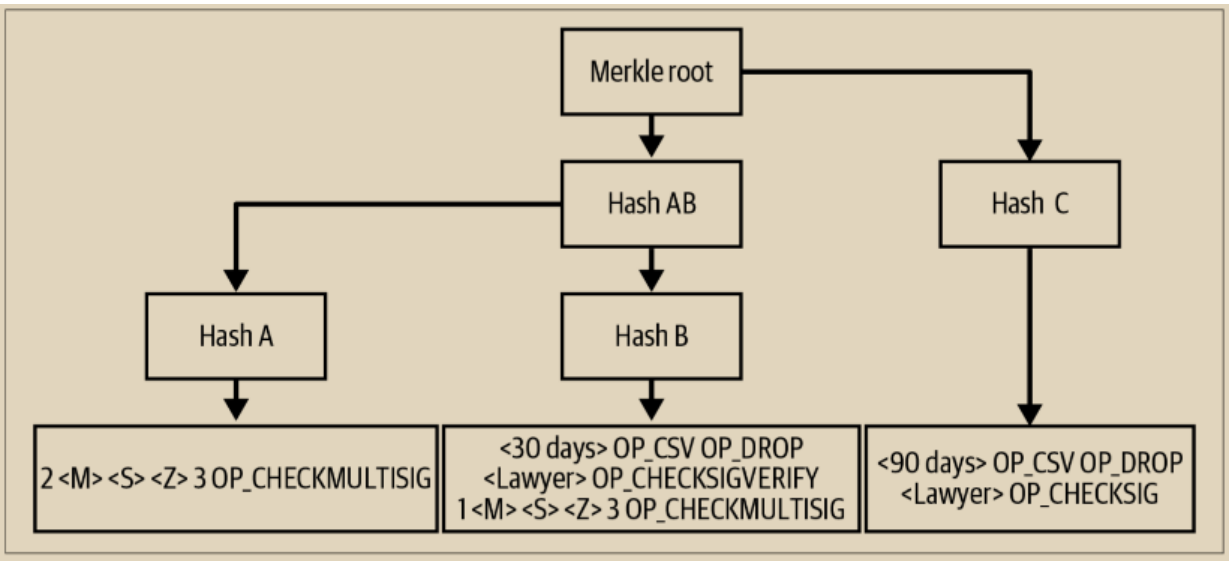


图 7-5. 一个包含三个子脚本的MAST

现在我们可以创建一个紧凑的成员证明，证明特定的授权条件是默克尔树的成员，而不透露有关默克尔树其他成员的任何细节。请参见图7-6，并注意阴影节点可以从用户提供的其他数据计算出来，因此在花费时间不需要指定它们。

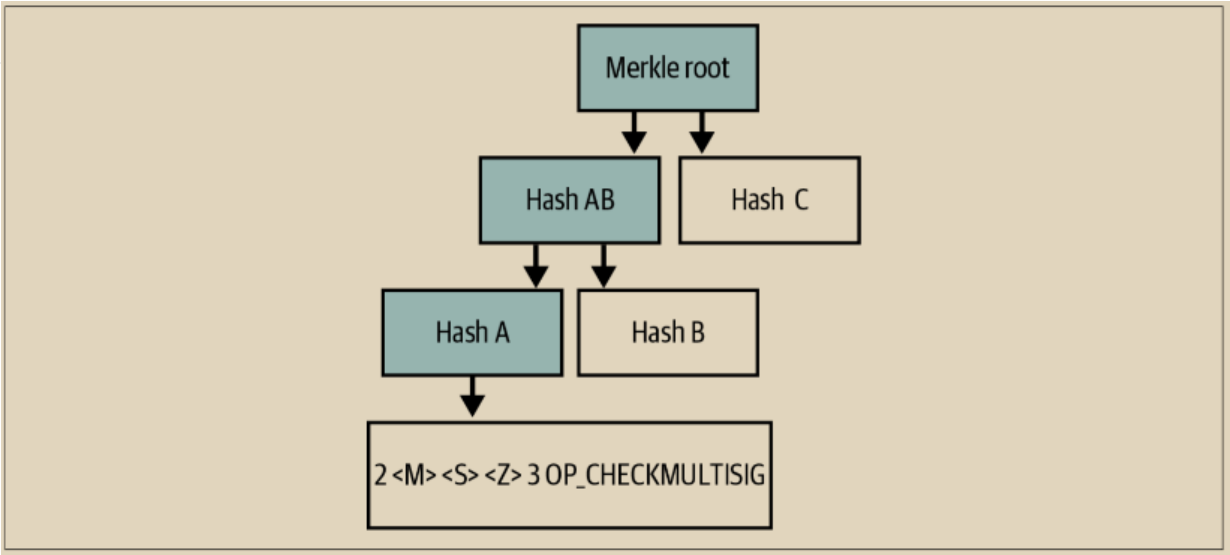


图 7-6. 一个子脚本的MAST成员证明

使用于创建承诺的哈希摘要每个都是32字节，因此证明图7-6的花费被授权（使用默克尔树和特定条件）和被认证（使用签名）需要383字节。相比之下，没有使用默克尔树的相同花费（即提供所有可能的授权条件）使用了412字节。

在这个例子中节省了29字节（7%）并不能完全体现潜在的节省。默克尔树的二叉树结构意味着每次将集合（在这种情况下是授权条件）的成员数量加倍时，您只需要一个额外的32字节承诺。在这种情况下，有三个条件，我们需要使用三个承诺（其中一个是默克尔根，需要包含在授权数据中）；我们也可以使用四个承诺来获得相同的成本。额外的承诺将给我们带来多达八个条件。使用16个承诺（512字节的承诺），我们可以拥有超过32,000个授权条件，远远超过了填充有OP_IF语句的整个交易块中的条件数量。使用128个承诺（4,096字节），我们理论上可以创建的条件数量远远超过了世界上所有计算机能够创建的条件数量。

通常情况下，并非每个授权条件被使用的可能性都一样。在我们的示例中，我们预计Mohammed和他的合作伙伴会频繁地花费他们的钱；延迟时间条件仅在出现问题时存在。我们可以根据这一了解重构我们的树结构，如图7-7所示。

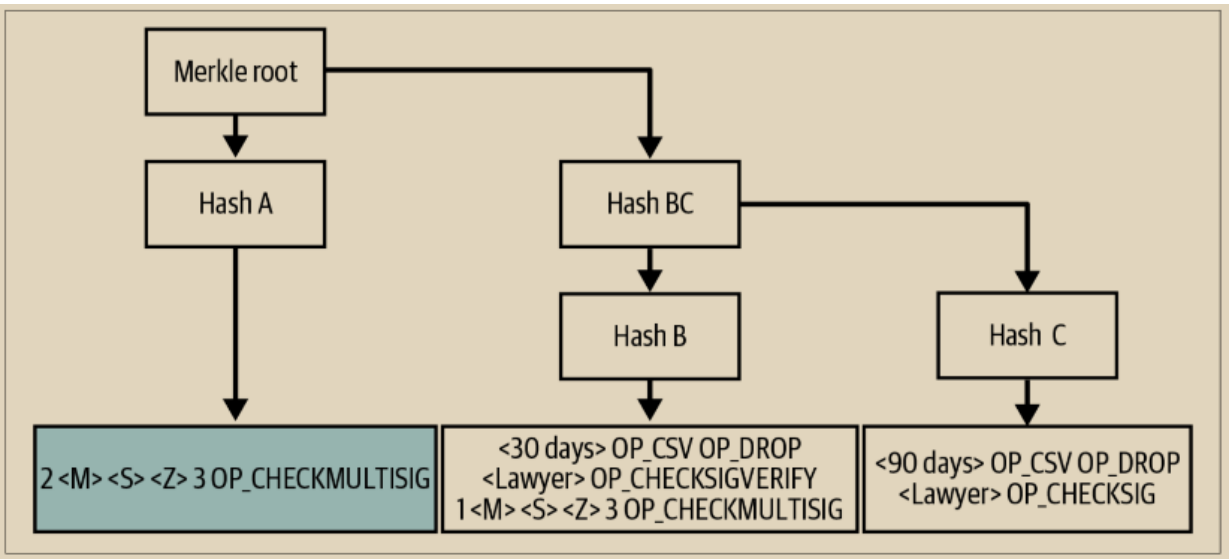


图 7-7. 一个带有最可能的脚本在最佳位置的MAST

\ 现在我们只需要为常见情况提供两个承诺（节省32字节），尽管对于不太常见的情况我们仍然需要三个承诺。如果你知道（或者能够猜测）使用不同授权条件的概率，你可以使用哈夫曼算法将它们放入一个最大效率的树中；详情请参阅BIP341。

无论树是如何构建的，我们可以从前面的例子中看到，我们只会公开实际使用的授权条件。其他条件保持私密。同样保持私密的是条件的数量：一棵树可能有一个条件，也可能有一万亿个条件——单看单个交易的链上数据，没有办法知道。

除了略微增加比特币的复杂性之外，MAST对比特币没有明显的不利影响，并且之前有两个坚实的提案，即BIP114和BIP116，然后发现了一种改进的方法，我们将在“Taproot”中看到。

> **MAST与MAST的比较**
>
> 最早关于比特币中现在所知的MAST的想法是merklized抽象语法树。在抽象语法树（AST）中，脚本中的每个条件都创建一个新的分支，如图7-8所示。

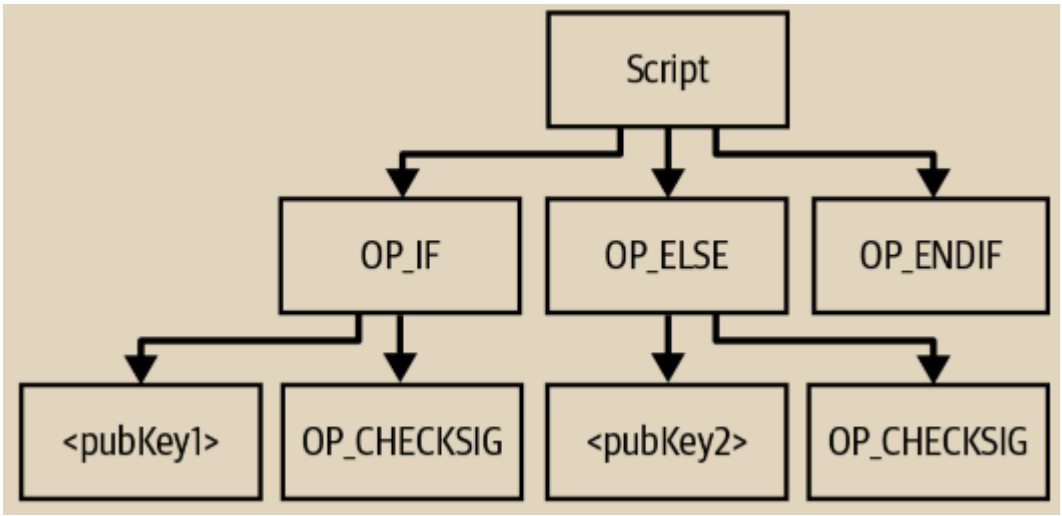


图 7-8. 一个脚本的抽象语法树（AST）

\ ASTs被广泛用于解析和优化其他程序的代码，比如编译器。一个merklized AST将承诺程序的每个部分，并启用172页中描述的功能，但它将需要为程序的每个单独部分至少透露一个32字节的摘要，这在大多数情况下对于区块链来说不是很空间高效的。

在比特币中，人们在大多数情况下称为MAST的是merklized alternative script trees，这是开发者Anthony Towns创造的一个缩写词。alternative script tree 是一个由脚本组成的集合，每个脚本都是完整的，但只能选择一个——它们彼此之间是替代的，如图7-9所示。

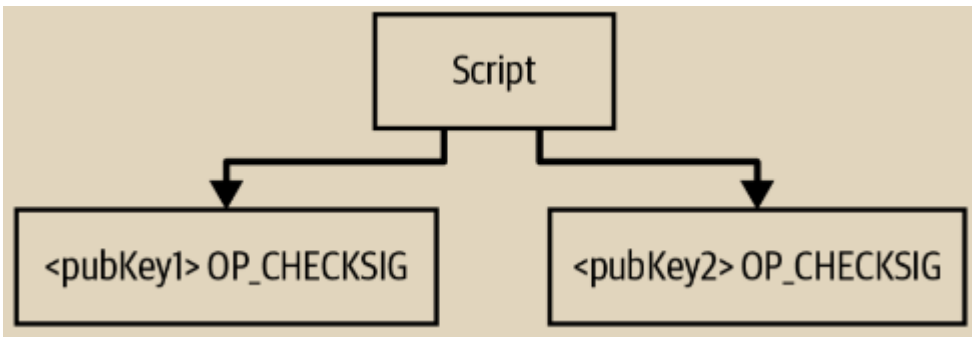


图 7-9. 一个备选脚本树

备选脚本树只需要在花费者选择的脚本和树根之间的每个层级深度上透露一个32字节的摘要。对于大多数脚本来说，这是对区块链空间更有效的利用。

支付到合约（P2C）

正如我们在“公共子密钥推导”中看到的那样，椭圆曲线密码学（ECC）的数学允许Alice使用私钥推导出她提供给Bob的公钥。他可以向该公钥添加任意值以创建派生的公钥。如果他将该任意值提供给Alice，她可以将其添加到她的私钥中，推导出相应的派生私钥。简而言之，Bob可以创建只有Alice可以创建相应私钥的子公钥。这对于BIP32样式的分层确定性（HD）钱包恢复很有用，但也可以用于其他用途。

让我们想象一下，Bob想从Alice那里购买东西，但他也想以后能够证明他支付了什么以防有任何争议。Alice和Bob就销售的物品或服务的名称达成一致（例如，“Alice的播客第123集”），并将该描述转换为数字，方法是将其哈希化并将哈希摘要解释为数字。Bob将该数字添加到Alice的公钥中并支付。该过程称为密钥调整，该数字称为调整值。

Alice可以通过使用相同的数字（调整值）调整她的私钥来支配资金。

稍后，Bob可以向任何人证明他支付了什么给Alice，方法是公开她的底层密钥和他们使用的描述。任何人都可以验证支付的公钥是否等于底层密钥加上对描述的哈希承诺。如果Alice承认那个密钥是她的，那么她就收到了付款。如果Alice花费了资金，这进一步证明她在签署支出交易时知道描述，因为只有在她知道调整值（描述）时，她才能为调整后的公钥创建有效的签名。

如果Alice和Bob都决定不公开他们使用的描述，它们之间的付款看起来就像任何其他付款一样。没有隐私损失。

由于P2C默认是私密的，我们无法知道它用于其原始目的的频率 - 在理论上，每笔付款都可能使用它，尽管我们认为这不太可能。然而，P2C今天在略有不同的形式中被广泛使用，我们将在“Taproot”中看到。

脚本化多重签名和门限签名

在“脚本化多重签名”中，我们讨论了需要多个密钥签名的脚本。然而，还有另一种需要多个密钥合作的方式，也被称为多重签名，这也让人感到困惑。为了区分这两种类型，在本节中，我们将称涉及OP_CHECKSIG类操作码的版本为脚本化多重签名，称另一版本为无脚本多重签名。

无脚本多重签名要求每个参与者都像创建私钥一样创建自己的秘密。我们将这个秘密称为部分私钥，尽管我们应该注意到它与常规完整私钥的长度相同。从部分私钥中，每个参与者使用我们在“公钥”中描述的与常规公钥相同的算法派生出部分公钥。每个参与者与所有其他参与者共享他们的部分公钥，然后将所有密钥组合在一起创建无脚本多重签名公钥。

这个组合的公钥看起来与任何其他比特币公钥都一样。第三方无法区分多方公钥和由单个用户生成的普通密钥。

要花费由无脚本多重签名公钥保护的比特币，每个参与者都生成一个部分签名。然后，将这些部分签名组合成一个常规完整签名。有许多已知的方法来创建和组合部分签名；我们将在第8章中更多地讨论这个主题。与无脚本多重签名的公钥类似，通过此过程生成的签名看起来与任何其他比特币签名都一样。第三方无法确定签名是由单个人创建还是由百万人相互合作创建的。

无脚本多重签名比脚本多重签名更小、更私密。 对于脚本多重签名，每个涉及的密钥和签名都会增加事务中的字节数。对于无脚本多重签名，大小是恒定的 —— 即使是一百万个参与者每人提供自己的部分密钥和部分签名，所放置在事务中的数据量也与使用单个密钥和签名的个体相同。隐私方面也是如此：因为每个新的密钥或签名都会向事务中添加数据，脚本多重签名会泄露有关使用了多少个密钥和签名的数据 —— 这可能会很容易地确定是哪些交易是由哪些参与者组创建的。然而，由于每个无脚本多重签名看起来都像其他无脚本多重签名和每个单个签名一样，没有泄露降低隐私的数据。

无脚本多重签名存在两个缺点。第一个是为了在比特币上创建它们，所有已知的安全算法都需要更多的互动轮次或更谨慎的状态管理，这可能在签名几乎无状态的硬件签名设备生成签名并且密钥在物理上分布时会有挑战。例如，如果你将硬件签名设备放在银行保险箱中，你可能需要访问该保险箱一次来创建脚本多重签名，但是对于无脚本多重签名可能需要两次或三次。

另一个缺点是阈值签名不会透露谁签名了。在脚本化阈值签名中，Alice、Bob 和 Carol 同意（例如）任意两人签名就足以花费资金。如果 Alice 和 Bob 签名，这就需要在链上放置每个人的签名，向任何知道他们密钥的人证明他们签名了，而 Carol 没有。在无脚本阈值签名中，来自 Alice 和 Bob 的签名与 Alice 和 Carol 或 Bob 和 Carol 之间的签名无法区分。这对隐私有益，但这意味着，即使 Carol 声称她没有签名，她也不能证明她没有签名，这可能对问责和审计不利。

对于许多用户和用例来说，多重签名的始终减小的大小和增加的隐私优于其创建和审计签名时的偶发挑战。

Taproot

人们选择使用比特币的一个原因是可以创建具有高度可预测结果的合约。由法院执行的法律合同在一定程度上依赖于案件中涉及的法官和陪审员的决定。相比之下，比特币合约通常要求参与者采取行动，但除此之外由成千上万个运行功能上相同代码的全节点来执行。在给定相同的合约和相同的输入时，每个全节点将始终产生相同的结果。任何偏差都意味着比特币出现故障。人类法官和陪审团比软件更灵活，但当不需要或不想要这种灵活性时，比特币合约的可预测性是一个重要资产。

如果合同中的所有参与者都意识到其结果已经完全可预测，实际上他们就没有继续使用合同的必要了。他们可以遵循合同规定的要求，然后终止合同。在社会中，这是大多数合同终止的方式：如果利益相关方满意，他们就不会将合同提交给法官或陪审团。在比特币中，这意味着任何需要大量区块空间来解决的合同也应该提供一个条款，允许通过相互满意来解决合同。

\ 在MAST和无脚本多重签名中，设计一个相互满意条款是很容易的。我们只需将脚本树的顶层之一设置为所有利益相关方之间的无脚本多重签名。我们已经在图7-7中看到了一个包含几方的复杂合同，其中包含了一个简单的相互满意条款。我们可以通过从脚本多重签名切换到无脚本多重签名来使其更加优化。

这相当有效且私密。如果使用了相互满意条款，我们只需要提供一个默克尔分支，而我们透露的只是有一个签名参与其中（可能来自一个人，也可能来自数千个不同的参与者）。但是，2018年的开发人员意识到，如果我们还使用了支付合同，我们可以做得更好。

在我们之前对支付合同的描述中，“支付合同（P2C）”中，我们通过调整公钥来承诺Alice和Bob之间的协议文本。我们可以改为通过承诺MAST的根来承诺合同的程序代码。我们调整的公钥是一个普通的比特币公钥，这意味着它可以要求来自单个人的签名，也可以要求来自多个人的签名（或者可以以特殊的方式创建，使其无法生成签名）。这意味着我们可以通过所有利益相关方的单个签名或者透露我们要使用的MAST分支来满足合同。包含公钥和MAST的承诺树如图7-10所示。

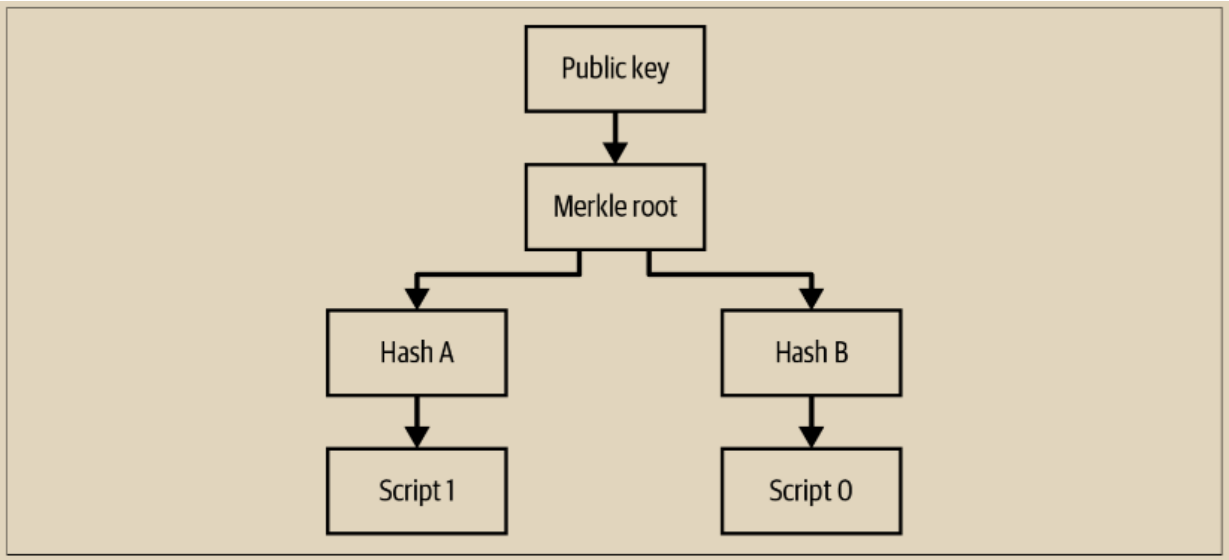


图 7-10. 一个公钥承诺到一个默克尔根的taproot

这使得使用多重签名的相互满意条款非常高效且非常私密。事实上，它的私密性甚至超出了表面上的表现，因为任何由单个用户创建的交易，只要该用户希望通过单个签名来满足它（或者通过多个不同钱包控制的多重签名），在链上看起来与相互满意的花费完全相同。在这种情况下，无论是由数百万用户参与的极其复杂的合同还是单个用户只是花费他们存储的比特币，链上都没有区别。

当可以使用密钥进行花费时，例如单个签名或无脚本多重签名，这被称为密钥路径（Keypath）花费。当使用脚本树时，这被称为脚本路径（Scriptpath）花费。对于密钥路径花费，放在链上的数据是公钥（在见证程序中）和签名（在见证堆栈上）。

对于脚本路径花费，链上的数据还包括公钥，它被放置在一个见证程序中，在这个上下文中称为taproot输出密钥。见证结构包括以下信息：

• 一个版本号。

• 底层密钥——在被默克尔根扭曲之前存在的密钥，用于生成taproot输出密钥。这个底层密钥被称为taproot内部密钥。

• 要执行的脚本，称为叶子脚本。

• 沿着连接叶子和默克尔根的路径上每个节点的一个32字节哈希值。

• 满足脚本所需的任何数据（例如签名或哈希前像）。

我们只知道一个重要的taproot缺点：希望使用MAST但不想要共同满意条款的合同参与者必须在区块链上包含一个taproot内部密钥，增加约33字节的开销。考虑到几乎所有合同都预计会受益于共同满意条款，或者其他使用顶级公钥的多签名条款，并且所有用户都受益于输出看起来相似的增加的匿名性集合，大多数参与taproot激活的用户认为这种罕见的开销并不重要。

对于taproot的支持是通过软分叉添加到比特币中的，并于2021年11月激活。

Tapscript

Taproot使得MAST成为可能，但只是使用了略有不同版本的比特币脚本语言，这个新版本被称为tapscript。其主要区别包括：

脚本化多重签名的变化

旧的OP_CHECKMULTISIG和OP_CHECKMULTISIGVERIFY操作码被移除了。这些操作码与Taproot软分叉中的另一项变化——使用Schnorr签名进行批量验证的能力——不太兼容（参见第187页的“Schnorr签名”）。相反，提供了一个新的OP_CHECKSIGADD操作码。当它成功验证一个签名时，这个新的操作码将一个计数器增加一，这样就可以方便地计算通过的签名数量，然后与所需的成功签名数量进行比较，以重新实现与OP_CHECKMULTISIG相同的行为。

所有签名的变化

Tapscript中的所有签名操作都使用了BIP340中定义的Schnorr签名算法。我们将在第8章更深入地探讨Schnorr签名。此外，任何预期不会成功的签名检查操作必须使用值OP_FALSE（也称为OP_0），而不是实际的签名。如果在失败的签名检查操作中提供任何其他内容，将导致整个脚本失败。这也有助于支持Schnorr签名的批量验证。

OP_SUCCESSx操作码

在以前版本的脚本中无法使用的操作码现在被重新定义为，如果使用它们，整个脚本将成功。这样做允许将来的软分叉在某些情况下重新定义它们为不成功，这是一种限制，因此可以在软分叉中实现。（相反，将一个不成功的操作定义为成功只能在硬分叉中完成，这是一种更具挑战性的升级路径。）

尽管我们在本章中深入研究了授权和认证，但我们跳过了比特币对花费者进行身份验证的一个非常重要的部分：它的签名。我们将在第8章中继续探讨这一点。

综合介绍

\ 目前比特币中使用了两种签名算法，分别是Schnorr签名算法和椭圆曲线数字签名算法（ECDSA）。这些算法基于椭圆曲线私钥/公钥对进行数字签名，如第56页的“椭圆曲线加密解释”中所描述的。它们用于花费SegWit v0 P2WPKH输出、SegWit v1 P2TR路径花费，以及脚本函数OP_CHECKSIG、OP_CHECKSIGVERIFY、OP_CHECKMULTISIG、OP_CHECKMULTISIGVERIFY和OP_CHECKSIGADD。每当其中一个被执行时，都必须提供一个签名。

在比特币中，数字签名有三个作用。首先，签名证明了私钥的控制者，也就是资金的所有者，已经授权使用这些资金。其次，授权的证据是不可否认的（不可否认性）。第三，被授权的交易不能被未经身份验证的第三方更改，其完整性得到保证。

> 每个交易输入及其可能包含的任何签名都与其他输入或签名完全独立。多方可以合作构建交易，每个人只需对一个输入进行签名。几种协议利用了这一点来创建用于隐私的多方交易。

在本章中，我们将介绍数字签名的工作原理，以及如何通过它们提供私钥控制的证明，而不暴露私钥。

数字签名的工作原理

数字签名由两部分组成。第一部分是使用私钥（签名密钥）为消息（交易）创建签名的算法。第二部分是一个算法，允许任何人在给定消息和相应的公钥的情况下验证签名。

生成数字签名

在比特币中使用的数字签名算法中，被签名的“消息”是交易，或更准确地说是交易数据的特定子集的哈希值，称为承诺哈希（参见“签名哈希类型（SIGHASH）”第185页）。签名密钥是用户的私钥。结果是签名：

$$ Sig = F*{sig}(F*{hash}(m),x)

$$

其中：

- x 是签名的私钥
- m 是要签名的消息，即承诺哈希（例如交易的部分）
- Fhash 是哈希函数
- Fsig 是签名算法
- Sig 是生成的签名

您可以在“Schnorr Signatures”第187页和“ECDSA Signatures”第197页找到有关schnorr和ECDSA签名数学的更多详细信息。

在schnorr和ECDSA签名中，函数Fsig 生成一个由两个值组成的签名 Sig。这两个值在不同算法中存在差异，我们稍后会详细探讨。计算出这两个值后，它们被序列化成一个字节流。对于ECDSA签名，编码使用称为Distinguished Encoding Rules或DER的国际标准编码方案。对于schnorr签名，使用了更简单的序列化格式。

验证签名

签名验证算法接受消息（通常是交易和相关数据的哈希）、签名者的公钥和签名，并且如果该签名对于此消息和公钥是有效的，则返回TRUE。 要验证签名，必须具备以下组件：

1. 签名本身。
2. 序列化的交易。
3. 有关要花费的输出的相关数据。
4. 与用于生成签名的私钥相对应的公钥。

总的来说，签名验证过程确认只有生成给定交易消息的公钥对应的私钥的个人或实体才能在给定交易上产生此签名。

签名哈希类型（SIGHASH）

数字签名适用于消息，在比特币中，这些消息就是交易本身。签名证明了签名者对特定交易数据的承诺。在最简单的形式中，签名适用于几乎整个交易，因此承诺了所有输入、输出和其他交易字段。然而，签名也可以仅适用于交易数据的子集，这对于许多情况都是有用的，正如我们将在本节中看到的那样。

比特币签名有一种方法来指示由私钥使用 SIGHASH 标志签名的交易数据的哪部分被包含在哈希中。SIGHASH 标志是附加到签名的单个字节。每个签名都有一个明确或隐含的 SIGHASH 标志，而且该标志可能从一个输入到另一个输入不同。一个包含三个已签名输入的交易可能有三个带有不同 SIGHASH 标志的签名，每个签名都签署（承诺）交易的不同部分。

请记住，每个输入可能包含一个或多个签名。因此，一个输入可能有具有不同 SIGHASH 标志的签名，这些签名承诺交易的不同部分。还要注意，比特币交易可能包含来自不同“所有者”的输入，这些所有者可能仅在部分构建的交易中签署一个输入，与其他人合作以收集所有必要的签名以使交易有效。 许多 SIGHASH 标志类型只有在考虑到多个参与者在比特币网络之外协作并更新部分签名的交易时才有意义。

SIGHASH 标志有三种类型：ALL、NONE 和 SINGLE，如表 8-1 所示。

表 8-1. SIGHASH 类型及其含义

SIGHASH标志	值	描述
ALL	0x01	签名适用于所有的输入和输出
NONE	0x02	签名适用于所有的输入，但不适用于任何输出
SINGLE	0x03	签名适用于所有输入，但仅适用于与签名输入具有相同索引号的一个输出

此外，还有一个修饰标志，SIGHASH_ANYONECANPAY，可以与前面的每个标志结合使用。当设置了ANYONECANPAY时，只有一个输入被签名，其余的（及其序列号）则保持开放以进行修改。ANYONECANPAY的值为0x80，通过按位或运算应用于组合标志，如表8-2所示。

表8-2. 带有修饰符的SIGHASH 类型及其含义

SIGHASH标志	值	描述	
ALL\	ANYONECANPAY	0x81	签名适用于一个输入和所有输出
NONE\	ANYONECANPAY	0x82	签名适用于一个输入，但不适用于任何输出
SINGLE\	ANYONECANPAY	0x83	签名适用于一个输入，以及具有相同索引号的输出

在签名和验证过程中应用SIGHASH标志的方式是，首先复制交易，然后将其中某些字段省略或截断（设置为空长度）。结果交易进行序列化。SIGHASH标志包含在序列化的交易数据中，然后对结果进行哈希。哈希摘要本身就是被签名的“消息”。根据使用的SIGHASH标志不同，将包括交易的不同部分。通过包含SIGHASH标志本身，签名也承诺了SIGHASH类型，因此它不能被更改（例如，由矿工更改）。

在“ECDSA签名的序列化（DER）”中，我们将看到DER编码签名的最后部分是01，这是ECDSA签名的SIGHASH_ALL标志。这将锁定交易数据，因此Alice的签名将承诺所有输入和输出的状态。这是最常见的签名形式。

现在让我们看看其他一些SIGHASH类型及其在实践中的使用方式：

\ ALL|ANYONECANPAY

这种类型的构建可用于创建“众筹”风格的交易。试图筹集资金的人可以构建一个只有一个输出的交易。单个输出支付“目标”金额给筹款人。这种交易显然是无效的，因为它没有输入。然而，其他人现在可以通过添加他们自己的输入作为捐款来修改它。他们使用ALL|ANYONECANPAY对自己的输入进行签名。除非收集到足够的输入达到输出值，否则该交易是无效的。每笔捐款都是一笔“承诺”，在筹集到整个目标金额之前，筹款人不能收取。不幸的是，该协议可以被筹款人添加他们自己的输入（或来自借给他们资金的人），即使他们没有达到指定的价值，也允许他们收取捐款。

NONE

这种类型的构建可用于创建特定金额的“持票支票”或“空白支票”。它承诺了所有的输入，但允许输出被更改。任何人都可以将自己的比特币地址写入输出脚本中。单独使用这种方式允许任何矿工更改输出目的地并将资金据为己有，但如果交易中的其他必要签名使用了SIGHASH_ALL或另一种承诺输出的类型，那么它允许这些花费者更改目的地，而不允许任何第三方（如矿工）修改输出。

NONE|ANYONECANPAY

这种类型的构建可用于构建一个“尘埃收集器”。在他们的钱包中有微小UTXO的用户，无法在费用超过UTXO价值时进行支出；请参阅“经济不划算的输出和不允许的尘埃”页。有了这种类型的签名，经济不划算的UTXO可以捐赠给任何人在他们想要的时候进行聚合和支出。

\ 有一些提议修改或扩展SIGHASH系统。截至目前，最广泛讨论的提议是BIP118，该提议提议添加两个新的sighash标志。使用SIGHASH_ANYPREVOUT的签名不会承诺输入的outpoint字段，从而允许其用于花费特定witness程序的任何先前输出。例如，如果Alice收到了两个相同金额的输出到相同的witness程序（例如，需要她钱包的单个签名），则用于花费其中一个输出的SIGHASH_ANYPREVOUT签名可以被复制并用于花费另一个输出到相同的目标。

使用SIGHASH_ANYPREVOUTANYSCRIPT的签名不会承诺输出点、金额、见证程序或taproot merkle树（脚本树）中的特定叶子，因此可以花费任何之前的输出，只要签名可以满足条件。例如，如果Alice收到了两个不同金额和不同见证程序的输出（例如，一个需要单个签名，另一个需要她的签名加上一些其他数据），则用于花费其中一个输出的SIGHASH_ANYPREVOUTANYSCRIPT签名可以被复制并用于花费另一个输出到相同的目标（假设第二个输出的额外数据已知）。

两个SIGHASH_ANYPREVOUT操作码的主要预期用途是改进的支付通道，例如闪电网络（LN）中使用的通道，尽管还描述了几种其他用途。

> 在用户的钱包应用程序中，很少会看到SIGHASH标志作为选项呈现。简单的钱包应用程序使用SIGHASH_ALL标志进行签名。更复杂的应用程序，例如LN节点，可能使用替代的SIGHASH标志，但它们使用已经广泛审查的协议来理解替代标志的影响。

施乃尔(Schnorr)签名

1989年，克劳斯·施乃尔发表了一篇论文，描述了他的同名签名算法。该算法并不特定于椭圆曲线密码学（ECC），尽管它如今可能与ECC联系最为密切，ECC是比特币和许多其他应用程序使用的密码学。施乃尔签名具有许多良好的特性：

\ ***可证明安全性***

对施乃尔签名安全性的数学证明仅依赖于解决离散对数问题（DLP）的难度，特别是对于比特币所使用的椭圆曲线（EC），以及哈希函数（比特币中使用的SHA256函数）产生不可预测的值的能力，称为随机预言模型（ROM）。其他签名算法可能具有额外的依赖项，或者需要更大的公钥或签名才能获得与ECC-Schnorr相同的安全性（当威胁定义为经典计算机时；其他算法可能提供更有效的安全性，针对量子计算机）。

线性性

施乃尔签名具有数学家称为线性性的属性，该属性适用于具有两个特定属性的函数。第一个属性是将两个或多个变量相加，然后对该总和运行函数将产生与分别对每个变量运行函数然后将结果相加相同的值，例如，f(x + y + z) == f(x) + f(y) + f(z)；这种属性称为可加性。第二个属性是将一个变量乘以一个数，然后对该乘积运行函数将产生与对变量运行函数然后乘以相同量的值相同的值，例如，f(a × x) == a × f(x)；这种属性称为一次齐次性。

在密码学操作中，一些函数可能是私有的（例如涉及私钥或秘密随机数的函数），因此能够在函数内外执行操作获得相同结果，使得多个方可协调合作而无需分享其秘密。我们将在“基于施乃尔的无脚本多重签名”（第193页）和“基于施乃尔的无脚本阈值签名”（第195页）中看到线性性的具体好处。

批量验证

当以某种方式使用（比特币即如此）时，施乃尔的线性性的一个后果是，相对而言，可以比独立验证每个签名所需的时间更简单地同时验证多个施乃尔签名。在一个批次中验证的签名越多，加速度越大。对于一个块中的典型签名数量，可以在大约一半的时间内批量验证它们，而这将花费验证每个签名的时间。

在本章的后面，我们将精确描述施乃尔签名算法，就像它在比特币中使用的那样，但我们将从一个简化版本开始，并逐步逼近实际的协议。

Alice首先选择一个大的随机数（x），我们称之为她的私钥。她还知道比特币椭圆曲线上的一个公共点称为生成器（G）（参见“公钥”）。Alice使用椭圆曲线乘法将G乘以她的私钥x，其中x被称为标量，因为它对G进行了缩放。结果是xG，我们称之为Alice的公钥。Alice将她的公钥提供给Bob。尽管Bob也知道G，但离散对数问题阻止Bob能够通过G将xG除以G来推导出Alice的私钥。

在以后的某个时间，Bob希望Alice通过证明她知道用于前面接收的公钥（xG）的标量x来证明自己的身份。Alice不能直接给Bob x，因为那样会使他能够向其他人证明自己是她，因此她需要在不向Bob透露x的情况下证明她对x的知识，这称为零知识证明。为此，我们开始施乃尔身份验证过程：

1. Alice选择另一个大的随机数（k），我们称之为私密nonce。她再次将其用作标量，将其与G相乘以产生kG，我们称之为公共nonce。她将公共nonce提供给Bob。
2. Bob选择自己的一个大随机数e，我们称之为挑战标量。我们称之为“挑战”，因为它用于挑战Alice证明她知道用于前面向Bob提供的公钥（xG）的私钥x；我们称之为“标量”，因为它稍后将用于乘以一个椭圆曲线点。
3. Alice现在拥有数（标量）x、k和e。她将它们组合在一起，使用公式s = k + ex来生成最终的标量s。她将s提供给Bob。
4. Bob现在知道标量s和e，但不知道x或k。但是，Bob知道xG和kG，并且他可以自己计算sG和exG。这意味着他可以检查Alice执行的操作的放大版本的相等性：sG == kG + exG。如果相等，那么Bob可以确信Alice在生成s时知道x。

> **Schnorr身份验证协议中的整数替代椭圆曲线点**
>
> 如果我们可以通过使用简单的整数代替椭圆曲线上的点来创建一个不安全的过度简化版本，可能更容易理解，交互式Schnorr身份验证协议。例如，我们将使用从3开始的质数： 设置：Alice选择x = 3作为她的私钥。她将其与生成器G = 5相乘，以获得她的公钥xG = 15。她将15提供给Bob。
>
> 1. Alice选择私密nonce k = 7，并生成公共nonce kG = 35。她将35提供给Bob。
> 2. Bob选择e = 11，并将其提供给Alice。
> 3. Alice生成 s = 40 = 7 + 11 × 3。她把40给了Bob。
> 4. Bob推导出 sG = 200 = 40 × 5 和 exG = 165 = 11 × 15。然后他验证 200 == 35 + 165。注意，这是Alice执行的相同操作，但所有值都被放大了5倍（G的值）。
>
> 当然，这是一个过度简化的例子。当使用简单整数时，我们可以通过生成器G来除以产品，以获取基础标量，但这并不安全。这就是比特币中使用的椭圆曲线加密的一个关键特性，即乘法容易，但除以曲线上的点是不切实际的。此外，对于这么小的数字，通过穷举法找到基础值（或有效的替代值）是很容易的；比特币中使用的数字要大得多。

让我们讨论一下交互式Schnorr身份验证协议的一些使其安全的特性：

\ 随机数（k）：

在第1步中，爱丽丝选择一个鲍勃不知道且无法猜测的数字，并将该数字的缩放形式kG交给他。此时，鲍勃已经拥有她的公钥（xG），这是她的私钥x的缩放形式。这意味着当鲍勃在处理最终方程式（sG = kG + exG）时，有两个他不知道的独立变量（x和k）。使用简单的代数可以解决一个未知变量的方程，但不能解决两个独立的未知变量，因此爱丽丝的nonce的存在防止了鲍勃能够推导出她的私钥。值得注意的是，此保护取决于nonce在任何方面都不可预测。如果爱丽丝的nonce有可预测的任何东西，鲍勃可能会利用这一点来推断出爱丽丝的私钥。有关更多详细信息，请参阅“签名中随机性的重要性”（第200页）。

挑战标量（e）：

鲍勃等待收到爱丽丝的公共nonce，然后在第2步中继续给她一个数字（挑战标量），爱丽丝之前不知道并且无法猜测。重要的是，鲍勃只有在她承诺了她的公共nonce之后才会给她挑战标量。想象一下，如果一个不知道x的人想要冒充爱丽丝，而鲍勃在他们告诉他公共nonce kG之前意外地给了他们挑战标量e，会发生什么情况。这使得冒充者可以更改鲍勃将用于验证的方程两侧的参数，sG == kG + exG；具体来说，他们可以同时更改sG和kG。考虑到方程式的简化形式：x = y + a。如果你可以同时更改x和y，你可以使用x' = (x – a) + a来消除a。现在，您选择的任何x值都将满足方程式。对于冒充者而言，实际方程式很简单，他们只需为s选择一个随机数，生成sG，然后使用EC减法选择一个等于kG的kG = sG – exG。他们将给出他们计算出的kG以及稍后的随机sG，而鲍勃会认为这是有效的，因为sG == （sG – exG) + exG。这解释了协议中操作顺序的重要性：鲍勃必须在爱丽丝承诺她的公共nonce之后才能给她挑战标量。

\ 在这里描述的交互式身份验证协议与克劳斯·施诺尔的原始描述部分匹配，但缺少了我们在去中心化比特币网络中需要的两个关键特性。其中之一是它依赖于鲍勃等待爱丽丝承诺她的公共nonce，然后鲍勃给她一个随机的挑战标量。在比特币中，每个交易的支出者都需要由数千个比特币全节点进行身份验证，包括尚未启动但其运营商将来会希望确保他们收到的比特币来自每个交易都有效的传输链的未来节点。无论今天还是将来，任何无法与爱丽丝通信的比特币节点都将无法对其进行身份验证，并且将与对其进行身份验证的每个其他节点不一致。对于像比特币这样的共识系统来说，这是不可接受的。为了让比特币运作，我们需要一个不需要爱丽丝与每个想要对她进行身份验证的节点进行交互的协议。

一种简单的技术，称为Fiat-Shamir变换，可以将Schnorr交互式身份验证协议转换为非交互式数字签名方案。回想一下步骤1和2的重要性，包括它们按顺序执行。爱丽丝必须承诺一个不可预测的nonce；鲍勃必须在收到她的承诺后才给爱丽丝一个不可预测的挑战标量。还记得我们在本书其他部分使用的安全加密哈希函数的属性：当给定相同的输入时，它将始终产生相同的输出，但当给定不同的输入时，它将产生与随机数据无法区分的值。这使得爱丽丝可以选择她的私有nonce，推导出她的公共nonce，然后对公共nonce进行哈希以获得挑战标量。因为爱丽丝无法预测哈希函数的输出（挑战），并且对于相同的输入（nonce），它总是相同的，这确保了爱丽丝即使选择了nonce并对其进行哈

希也能得到一个随机的挑战。我们不再需要鲍勃的互动。她只需发布她的公共nonce kG和标量s，每个数千个全节点（过去和未来）都可以对kG进行哈希以产生e，使用它来产生exG，然后验证sG == kG + exG。明确写出，验证方程变为sG == kG + hash(kG) × xG。

我们需要另一个东西来完成将交互式Schnorr身份验证协议转换为对比特币有用的数字签名协议。我们不仅希望爱丽丝证明她知道自己的私钥，还希望她能够承诺一个消息。具体来说，我们希望她承诺与她想要发送的比特币交易相关的数据。有了Fiat-Shamir变换，我们已经有了一种承诺，因此我们可以简单地使其额外承诺消息。现在，我们不仅使用hash(kG)，还使用hash(kG || m)来承诺消息，其中||表示串联。

现在我们已经定义了schnorr签名协议的一个版本，但还有一件事情我们需要做来解决比特币特定的问题。在BIP32密钥派生中，如在第92页的“公共子密钥派生”中描述的，非硬化派生的算法将一个公钥和一个非秘密值相加，以产生一个派生的公钥。这意味着也可以将那个非秘密值添加到一个密钥的有效签名中，以产生一个相关密钥的签名。这个相关签名是有效的，但它不是由持有私钥的人授权的，这是一个重大的安全失败。为了保护BIP32非硬化派生，并且支持人们希望在schnorr签名之上构建的几个协议，比特币的schnorr签名版本，称为secp256k1的BIP340 schnorr签名，还承诺要使用的公钥，除了公共nonce和消息之外。这使得完整的承诺哈希(kG || xG || m)。 现在我们已经描述了BIP340 schnorr签名算法的每个部分，并解释了它为我们做了什么，我们可以定义该协议了。整数的乘法是模p执行的，表示操作的结果被数字p除（如在secp256k1标准中定义的）并且余数被使用。数字p非常大，但如果它是3并且操作的结果是5，则我们实际使用的数字是2（即，5除以3的余数为2）。 设置：爱丽丝选择一个大的随机数(x)作为她的私钥（直接选择或使用BIP32等协议从大的随机种子值确定性地生成私钥）。她使用secp256k1中定义的参数（参见第56页的“椭圆曲线密码学解释”）将生成器G乘以她的标量x，产生xG（她的公钥）。她将她的公钥提供给以后将验证她的比特币交易的所有人（例如，通过在交易输出中包含xG）。当她准备花费时，她开始生成她的签名：

- Alice选择一个大的随机私用nonce k 并派生公共nonce kG。
- 她选择她的消息 m（例如，交易数据）并生成挑战标量 e = hash(kG || xG || m)。
- 她生成标量 s = k + ex。两个值 kG 和 s 是她的签名。她将这个签名提供给所有想要验证该签名的人；她还需要确保每个人都收到她的消息 m。在比特币中，这是通过将她的签名包含在她的支付交易的见证结构中，然后将该交易中继到完整节点来完成的。
- 验证者（例如完整节点）使用 s 派生 sG，然后验证 sG == kG + hash(kG || xG || m) × xG。如果等式成立，则 Alice证明了她知道自己的私钥 x（而没有泄露它），并承诺了消息 m（包含交易数据）。

Schnorr签名序列化

Schnorr签名的序列化包括两个值，kG 和 s。值 kG 是比特币椭圆曲线（称为 secp256k1）上的一个点，通常由两个32字节的坐标表示，例如 (x, y)。然而，只需要 x 坐标，因此只包含该值。当您在比特币的Schnorr签名中看到 kG 时，请注意它只是该点的 x 坐标。

值 s 是一个标量（意思是用来乘以其他数字的数字）。对于比特币的 secp256k1 曲线，它的长度永远不会超过32字节。

虽然 kG 和 s 可能有时是可以用较少于32字节表示的值，但它们很少会比32字节小得多，因此它们被序列化为两个32字节的值（即小于32字节的值具有前导零）。它们按照 kG 然后是 s 的顺序进行序列化，生成确切的64字节。

Taproot软分叉，也称为v1隔离见证，将Schnorr签名引入了比特币，并且截至目前，这是比特币中使用Schnorr签名的唯一方式。当与taproot keypath或scriptpath spending一起使用时，64字节的Schnorr签名被认为使用默认的签名哈希（sighash），即SIGHASH_ALL。如果使用了替代的sighash，或者如果花费者想要浪费空间来显式指定SIGHASH_ALL，那么将在签名后附加一个额外的字节，指定签名哈希，使签名成为65字节。 正

如我们将看到的那样，无论是64字节还是65字节，都比描述在“ECDSA签名序列化（DER）”中的序列化效率要高得多。

基于Schnorr的无脚本多重签名

在“Schnorr签名”页上描述的单签名Schnorr协议中，Alice使用签名（kG，s）公开证明了她对私钥的知识，这种情况下我们将其称为y。想象一下，如果Bob也有一个私钥（z），并且他愿意与Alice合作，以证明他们共同知道x = y + z，而不是他们中的任何一个向对方或其他人透露他们的私钥。让我们再次通过BIP340 schnorr签名协议进行说明。

> 我们即将描述的简单协议是不安全的，原因我们将很快解释。我们只是使用它来演示Schnorr多签名的机制，然后再描述与之相关的被认为是安全的协议。

Alice和Bob需要推导出x的公钥，即xG。由于可以使用椭圆曲线运算将两个EC点相加，因此他们从Alice派生yG和Bob派生zG开始。然后他们将它们相加在一起，创建xG = yG + zG。

该协议中的点xG是它们的聚合公钥。为了创建签名，他们开始简单的多签名协议：

1. 他们各自选择一个大的随机私有nonce，对于Alice是a，对于Bob是b。他们还各自派生相应的公共nonce aG 和 bG。他们一起生成聚合公共nonce kG = aG + bG。
2. 他们就要签名的消息m达成一致（例如，一笔交易），每个人都生成一个挑战标量的副本：e = hash(kG || xG || m)。
3. Alice生成标量 q = a + ey。Bob生成标量 r = b + ez。他们将这些标量相加得到 s = q + r。他们的签名是两个值 kG 和 s。
4. 验证者使用正常的方程检查他们的公钥和签名：sG == kG + hash(kG || xG || m) × xG。

Alice和Bob已经证明他们知道他们私钥的总和，而不让其中任何一方向另一方或其他任何人透露他们的私钥。该协议可以扩展到任意数量的参与者（例如，百万人可以证明他们知道他们百万个不同密钥的总和）。

前述的协议存在几个安全问题。最值得注意的是，一方可能在承诺自己的公钥之前学习到其他方的公钥。例如，Alice诚实地生成了她的公钥yG，并与Bob分享了它。Bob使用zG - yG生成了他的公钥。当他们的两个密钥结合在一起时（yG + zG - yG），正负yG项会相互抵消，因此公钥只代表z的私钥（即Bob的私钥）。现在Bob可以在没有任何帮助的情况下创建有效的签名。这称为密钥取消攻击。

有各种方法可以解决密钥取消攻击。最简单的方案是要求每个参与者在与其他所有参与者分享有关该密钥的任何内容之前，都要承诺其公钥的一部分。例如，Alice和Bob各自对其公钥进行哈希处理，并将其摘要与彼此分享。当他们都拥有对方的摘要时，他们可以分享他们的密钥。他们各自检查对方的密钥是否哈希为先前提供的摘要，然后正常进行协议。这可以防止他们中的任何一个选择会抵消其他参与者密钥的公钥。然而，要正确实施此方案是很容易失败的，例如，将其用于未硬化的BIP32公钥派生。此外，这增加了参与者之间的通信额外步骤，在许多情况下可能不理想。已经提出了更复杂的方案来解决这些缺陷。

除了密钥取消攻击之外，还存在许多可能针对随机数的攻击。请回忆一下，随机数的目的是防止任何人利用对签名验证方程中其他值的了解来解出您的私钥，从而确定其值。为了有效实现这一点，您必须每次签署不同消息或更改其他签名参数时都使用不同的随机数。这些不同的随机数不能以任何方式相关联。对于多重签名，每个参与者都必须遵循这些规则，否则可能会危及其他参与者的安全性。此外，还需要防止取消和其他攻击。实现这些目标的不同协议会做出不同的权衡，因此没有一种单一的多重签名协议适用于所有情况。相反，我们将注意到 MuSig 协议族中的三个协议：

MuSig

也称为 MuSig1，此协议在签署过程中需要三轮通信，类似于我们刚刚描述的过程。MuSig1 的最大优势在于其简单性。

MuSig2

此协议只需要两轮通信，有时可以将其中一轮与密钥交换合并。这可以显着加快某些协议的签名速度，例如计划在 LN 中使用无脚本多重签名。MuSig2 在 BIP327 中有规定（截至本文撰写时，这是唯一一个具有 BIP 的无脚本多重签名协议）。

MuSig-DN

DN 代表确定性随机数，消除了一个称为重复会话攻击的问题。它无法与密钥交换结合使用，且实现起来比 MuSig 或 MuSig2 要复杂得多。

对于大多数应用程序来说，MuSig2 是撰写本文时可用的最佳多重签名协议。

基于Schnorr的无脚本门限签名

无脚本多重签名协议仅适用于 k-of-k 签名。每个具有成为聚合公钥一部分的部分公钥的人都必须为最终签名贡献部分签名和部分随机数。然而，有时候，参与者希望允许其中的子集进行签名，例如 t-of-k，其中门限（t）数量的参与者可以为由 k 个参与者构建的密钥进行签名。这种类型的签名称为门限签名。

我们在“脚本多重签名”第150页看到了基于脚本的门限签名。 但就像无脚本多重签名相比脚本多重签名节省空间并增加隐私一样，无脚本门限签名相比脚本门限签名也节省空间并增加隐私。对于不参与签名的任何人来说，无脚本门限签名看起来就像任何其他可能由单一签名用户或通过无脚本多重签名协议创建的签名一样。

\ 已知有各种方法可用于生成无脚本门限签名，其中最简单的是对之前创建的无脚本多重签名稍作修改。此协议还依赖于可验证秘密共享（其本身依赖于安全秘密共享）。

基本的秘密共享可以通过简单的分割来实现。Alice有一个秘密数字，她将其分成三个长度相等的部分并与Bob、Carol和Dan分享。这三个人可以按正确顺序组合他们收到的部分数字（称为份额），以重构Alice的秘密。更复杂的方案涉及Alice向每个份额添加一些附加信息，称为修正码，允许其中任意两个人恢复该数字。这种方案并不安全，因为每个份额都使其持有者对Alice的秘密具有部分了解，使得参与者比没有份额的非参与者更容易猜测Alice的秘密。

安全的秘密共享方案阻止参与者在组合最低门限数量的份额之前了解有关秘密的任何信息。例如，如果Alice希望Bob、Carol和Dan中的任意两人能够重构她的秘密，她可以选择门限值为2。已知的最佳安全秘密共享算法是Shamir的秘密共享方案，通常缩写为SSSS，以其发现者的名字命名，他也是我们在“Schnorr Signatures”第187页看到的Fiat-Shamir变换的发现者之一。

在一些密码协议中，例如我们正在努力实现的无脚本门限签名方案，Bob、Carol和Dan知道Alice是否正确遵循了协议的一面至关重要。他们需要知道她创建的所有份额都来自同一个秘密，她使用了她声称的门限值，并且她给了他们每个人一个不同的份额。一个可以实现所有这些目标，并且仍然是一个安全的秘密共享方案的协议是可验证秘密共享方案。

要了解多重签名和可验证秘密共享如何对Alice、Bob和Carol起作用，想象一下他们每个人都希望接收可以由其中任意两人支配的资金。他们按照“基于 Schnorr 的无脚本多重签名”第193页所描述的方式进行合作，以生成接受资金的常规多重签名公钥（k-of-k）。然后，每个参与者从他们的私钥派生出两个秘密份额，一个用于其他两个参与者中的每一个。这些份额允许他们中的任意两个人重构多重签名的原始部分私钥。每个参与者将他们的一个秘密份额分发给其他两个参与者，导致每个参与者存储自己的部分私钥和其他每个参与者的一个份额。随后，每个参与者验证他们收到的份额的真实性和唯一性，与其他参与者给出的份额进行比较。

之后，当（例如）Alice和Bob希望在没有Carol参与的情况下生成一个无脚本门限签名时，他们交换他们拥有的两个份额给Carol。这使他们能够重构出Carol的部分私钥。Alice和Bob也有他们的私钥，使他们能够创建具有所有三个必要密钥的无脚本多重签名。

换句话说，刚刚描述的无脚本门限签名方案与无脚本多重签名方案相同，只是门限数量的参与者有能力重构任何其他无法或不愿签名的参与者的部分私钥。 在考虑无脚本门限签名协议时，需要注意几点：\ \ ***没有问责制***

由于Alice和Bob重构了Carol的部分私钥，因此通过涉及Carol和没有涉及Carol的过程生成的无脚本多重签名之间基本上没有区别。即使Alice、Bob或Carol声称他们没有签名，他们也没有确凿的方法证明他们没有帮助生成签名。如果重要的是知道组中的哪些成员签署了，那么您将需要使用脚本。

操纵攻击

想象一下，Bob告诉Alice说Carol不可用，所以他们一起重构Carol的部分私钥。然后Bob告诉Carol说Alice不可用，所以他们一起重构了Alice的部分私钥。现在Bob拥有自己的部分私钥以及Alice和Carol的私钥，使他能够在没有他们参与的情况下花费资金。如果所有参与者都同意只使用一种方案进行通信，该方案允许其中任何一个人看到其他所有消息（例如，如果Bob告诉Alice说Carol不可用，那么Carol在开始与Bob合作之前能够看到该消息），那么可以解决此攻击。在撰写本书时，针对此问题的其他解决方案，可能是更健壮的解决方案，正在进行研究。

尽管多位比特币贡献者已经对该主题进行了大量研究，并且我们预计在本书出版后将会有经过同行评审的解决方案问世，但尚未提出任何作为BIP的无脚本门限签名协议。

ECDSA签名

很不幸对于比特币和许多其他应用的未来发展，克劳斯·施纳尔申请了他发现的算法的专利，并阻止了它在开放标准和开源软件中的使用近20年。在1990年代初被禁止使用施纳尔签名方案的密码学家们开发了一种另一种构造，称为数字签名算法（DSA），其中包括一种适用于椭圆曲线的版本，称为ECDSA。

\ ECDSA方案以及建议曲线的标准参数在比特币开发于2007年之前已被广泛实现于加密库中。这几乎可以肯定是为什么ECDSA是比特币从首个发布版本直到2021年的taproot软分叉激活之前唯一支持的数字签名协议。ECDSA至今仍然支持所有非taproot交易。与施纳尔签名相比，一些不同之处包括：

更复杂

正如我们将看到的，ECDSA需要更多的操作来创建或验证签名，而不只是施纳尔签名协议。从实现的角度来看，它并没有显著的更复杂性，但这种额外的复杂性使ECDSA的灵活性更差、性能更差，而且更难以证明其安全性。

安全性难以证明

交互式施纳尔签名识别协议仅依赖于椭圆曲线离散对数问题（ECDLP）的强度。比特币中使用的非交互式身份验证协议还依赖于随机预言模型（ROM）。然而，ECDSA的额外复杂性阻碍了对其安全性的完整证明的发布（据我们所知）。我们不是加密算法的证明专家，但在30年之后，ECDSA可能只需要与施纳尔相同的两个假设来证明其安全性似乎不太可能。

非线性

ECDSA签名不能轻松地组合成无脚本多重签名，也不能用于相关的高级应用，比如多方签名适配器。虽然有针对这个问题的解决方法，但它们涉及额外的复杂性，会显著减慢操作速度，并且在某些情况下可能导致软件意外泄露私钥。

ECDSA算法

让我们来看看ECDSA的数学原理。签名是通过一个数学函数Fsig生成的，它产生由两个值组成的签名。在ECDSA中，这两个值是R和s。

签名算法首先生成一个私密nonce（k），并从中派生一个公共nonce（K）。数字签名的R值然后是nonce K的x坐标。

然后，算法计算签名的s值。就像我们在schnorr签名中所做的那样，涉及整数的操作都是模p进行的：

$$ s = k^{-1}(Hash(m)+x*R)

$$

其中：

- k 是私密nonce
- R 是公共nonce的x坐标
- x 是Alice的私钥
- m 是消息（交易数据）

验证是签名生成函数的逆函数，使用R、s值和公钥计算一个值K，它是椭圆曲线上的一个点（在签名创建中使用的公共nonce）：

$$ K = s^{-1} *Hash(m)* G + s^{-1} *R* X

$$

其中：

- R 和 s 是签名值
- X 是 Alice 的公钥
- m 是消息（被签名的交易数据）
- G 是椭圆曲线的生成点

如果计算得到的点 K 的 x 坐标等于 R，则验证者可以得出签名是有效的结论。

> ECDSA 的数学过程相当复杂；本书的范围之外无法完整解释。有许多在线优秀指南会逐步带领您了解它：搜索“ECDSA 解释”。

ECDSA签名序列化(DER)

让我们来看下面的 DER 编码签名：

3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204
b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301

该签名是由签名者生成的 R 和 s 值的序列化字节流，用于证明对用于花费输出的私钥的控制权。序列化格式包括以下九个元素：

- 0x30，表示 DER 序列的开始
- 0x45，序列的长度（69 字节）
- 0x02，后跟一个整数值
- 0x21，整数的长度（33 字节）
- R，00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb
- 0x02，后跟另一个整数
- 0x20，整数的长度（32 字节）
- S，4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
- 一个后缀（0x01），表示所使用的哈希类型（SIGHASH_ALL）

在签名中随机性的重要性

正如我们在“Schnorr Signatures”（第187页）和“ECDSA Signatures”（第197页）中所看到的，签名生成算法使用一个随机数k作为私有/公有nonce对的基础。k的值并不重要，只要它是随机的。如果来自同一个私钥的签名使用了私有nonce k并且对不同的消息（交易）进行签名，那么签名私钥可以被任何人计算出来。在签名算法中重复使用相同的值k会导致私钥曝光！

> 如果在两笔不同交易上的签名算法中使用相同的值k，那么私钥就可以被计算并暴露给世界！

这不仅仅是一种理论可能性。我们曾经看到这个问题导致比特币中几种不同实现的交易签名算法暴露私钥。由于意外重用k值，人们的资金被盗。重用k值的最常见原因是随机数生成器未正确初始化。

为了避免这种漏洞，业界最佳实践是不仅使用熵进行种子初始化的随机数生成器生成k，而是使用部分由交易数据本身加上正在使用的私钥进行种子初始化的过程。这确保了每个交易产生不同的k。确定k的行业标准算法是在RFC6979中定义的，由互联网工程任务组发布。对于schnorr签名，BIP340建议了一个默认的签名算法。

\ BIP340和RFC6979可以完全确定性地生成k，这意味着相同的交易数据将始终产生相同的k。许多钱包采用这种方法，因为这样可以轻松编写测试来验证其关键安全签名代码是否正确生成k值。BIP340和RFC6979都允许在计算中包含附加数据。如果该数据是熵，则即使签署了完全相同的交易数据，也会产生不同的k。这可以增加对侧信道和故障注入攻击的防护。

如果您正在实现比特币交易签名算法，您必须使用BIP340、RFC6979或类似的算法，以确保为每个交易生成不同的k。

\

隔离见证的新签名算法

\ 比特币交易中的签名是应用于承诺哈希上的，该哈希是从交易数据计算而来的，锁定了指示签署者对这些值的承诺的特定部分数据。例如，在简单的SIGHASH_ALL类型签名中，承诺哈希包括所有输入和输出。

不幸的是，遗留承诺哈希的计算方式引入了一种可能性，即验证签名的节点可能被迫执行大量的哈希计算。具体而言，哈希操作与交易中的输入数量呈大致二次方增长。因此，攻击者可以创建一个包含大量签名操作的交易，导致整个比特币网络必须执行数百甚至数千次的哈希操作来验证该交易。

隔离见证提供了解决这个问题的机会，方法是改变承诺哈希的计算方式。对于隔离见证版本0的见证程序，签名验证使用了BIP143规定的改进的承诺哈希算法。

新算法允许哈希操作的数量以更渐进的O(n)方式增加到签名操作的数量，减少了使用过于复杂交易创建拒绝服务攻击的机会。

在本章中，我们了解了比特币的Schnorr和ECDSA签名。这解释了完整节点如何验证交易，以确保只有控制比特币接收的密钥的人才能花费这些比特币。我们还研究了签名的几个高级应用，例如可以用于提高比特币效率和隐私的无脚本多签名和无脚本阈值签名。在过去的几章中，我们学会了如何创建交易，如何使用授权和认证来保护它们，并学会了如何对其进行签名。接下来，我们将学习如何通过向我们创建的交易添加费用来鼓励矿工确认它们。

综合介绍

我们在第8章中看到的Alice创建的数字签名只证明她知道自己的私钥，并且她承诺了一笔向Bob支付的交易。她可以创建另一个签名，该签名承诺了一笔向Carol支付的交易——即使用相同的输出（比特币）来支付Bob的交易。现在，这两笔交易是冲突的交易，因为只有一笔交易可以花费特定的输出，才能包含在具有最多工作量证明的有效区块链中——全节点用于确定哪些密钥控制哪些比特币的区块链。

为了保护自己免受冲突交易的影响，对Bob来说，他最好在Alice的交易被确认到足够深度的区块链中之前等待一段时间，然后再考虑他收到的钱是否可以花费（请参阅第14页的“确认”）。要将Alice的交易包含在区块链中，它必须被包含在一个交易块中。在一定时间内会产生有限数量的区块，并且每个区块只有有限的空间。只有创建该区块的矿工才能选择包含哪些交易。矿工可以按照任何他们想要的标准选择交易，包括拒绝包含任何交易。

> 在本章中，当我们提到“交易”时，我们指的是除了第一个交易之外的每个交易。在一个区块中的第一个交易是一个coinbase交易，它在第139页的“Coinbase交易”中有描述，允许该区块的矿工收取他们为产生该区块而获得的奖励。与其他交易不同，coinbase交易不花费前一笔交易的输出，并且还是适用于其他交易的几项规则的例外情况。Coinbase交易不支付交易费用，不需要调整费用，不受交易固定的影响，并且在讨论费用的后续讨论中主要没有意义，因此在本章中我们将忽略它们。

\ 几乎所有矿工选择交易包含在其区块中的标准是最大化他们的收入。比特币专门设计了一个机制来实现这一点，允许一笔交易向将该交易包含在区块中的矿工支付费用。我们称这种机制为交易费用，尽管它并不是通常意义上的费用。它不是由协议或任何特定的矿工设定的金额，而更像是拍卖中的出价。所购买的商品是区块中交易将消耗的有限空间的部分。矿工选择那些出价将使他们获得最大收入的交易集合。

在本章中，我们将探讨这些出价——交易费用的各个方面，以及它们如何影响比特币交易的创建和管理。

谁支付交易费用?

大多数支付系统都涉及一定的交易费用，但通常这种费用对于普通购买者来说是隐藏的。例如，一个商家可能会以相同的价格宣传同一件商品，无论您使用现金还是信用卡支付，尽管他们的支付处理器可能会对信用交易收取更高的费用，而他们的银行对现金存款收取的费用则较低。

在比特币中，每笔比特币的支出都必须经过身份验证（通常是通过签名），因此没有支付者的许可，交易是无法支付费用的。接收交易的人可以在不同的交易中支付费用，我们将在后面看到这种用法，但如果我们希望单个交易支付自己的费用，那么费用需要由支付者协商同意。它不能被隐藏。

比特币交易的设计是，支付者承诺支付的费用不需要在交易中额外占用空间。这意味着，尽管可以在不同的交易中支付费用，但在单个交易中支付费用是最有效（也是最便宜）的。

在比特币中，费用是一个竞价，支付的金额有助于确定交易需要多长时间才能确认。支付者和接收者通常都希望支付迅速确认，因此通常只允许支付者选择费用有时可能会成为问题；我们将在“父级支付子级费用（CPFP）提高费率”中解决这个问题。然而，在许多常见的支付流程中，希望交易迅速确认的最强烈的一方，即最愿意支付更高费用的一方，通常是支付者。

\ 出于技术和实际的原因，比特币中的惯例是由支付者支付交易费用。当然也有例外情况，比如接受未确认交易的商家，以及在签名后不立即广播交易的协议（这样做会阻止支付者选择适当的市场费用）。我们稍后会探讨这些例外情况。

手续费和费率

每个交易只支付一笔费用，不管交易的大小如何。然而，随着交易规模的增大，矿工能够容纳的交易数量就越少。因此，矿工评估交易的方式与您在市场上比较几种相同的物品的方式类似：他们将价格除以数量。

就像您可能会将几种不同袋装大米的成本除以每袋的重量来找到最低价格/重量（最佳交易）一样，矿工将交易的费用除以其大小（也称为其重量）来找到每个单位重量的最高费用（最大收益）。在比特币中，我们使用费率来表示交易的大小除以重量。

由于比特币多年来的变化，费率可以用不同的单位表示：

- BTC/字节（一个很少再使用的传统单位）
- BTC/千字节（一个很少再使用的传统单位）
- BTC/V字节（很少使用）
- BTC/千V字节（主要在比特币核心中使用）
- Satoshi/V字节（今天最常用的单位）
- Satoshi/权重（今天也经常使用）

在接受费率输入时要小心。如果用户将一个以一种分母打印的费率复制并粘贴到使用另一种分母的字段中，他们可能会支付过高的费用1000倍。如果他们改变分子，理论上可能会支付过高100,000,000倍。钱包应该让用户很难支付过高的费率，并可能希望提示用户确认任何不是由钱包本身使用可信数据源生成的费率。

过高的费用，也称为荒谬的费用，是任何远高于费率估算器目前认为必要的费用来确保交易在下一个区块中得到确认的费率。请注意，钱包不应完全阻止用户选择过高的费率 - 它们只应该让意外选择这种费率变得困难。用户在偶尔情况下过度支付费用存在合理的原因。

确定合适的费率估算

我们已经确定，如果你愿意等待更长的时间来确认你的交易，你可以支付较低的费率，但要注意支付过低的费率可能导致你的交易永远无法确认。由于费率是对区块空间的一个开放拍卖的出价，所以无法完全预测你需要支付多少费率才能在特定时间内确认你的交易。然而，我们可以根据最近其他交易所支付的费率来生成一个粗略的估算。

一个完整的节点可以记录每个交易的三个信息：它首次收到该交易的时间（区块高度）、该交易被确认的区块高度以及该交易支付的费率。通过将到达时间相似、确认时间相似且支付费率相似的交易分组在一起，我们可以计算确认支付某一费率的交易需要多少个区块。然后，我们可以假设现在支付类似费率的交易将需要相似数量的区块来确认。Bitcoin Core包含一个使用这些原理的费率估算器，可以使用 `estimatesmartfee` RPC调用，参数指定在交易高度高度可能确认之前你愿意等待多少个区块（例如，144个区块约为1天）：

```
$ bitcoin-cli -named estimatesmartfee conf_target=144
{
 "feerate": 0.00006570,
 "blocks": 144
}
```

\ 许多基于网络的服务也提供作为API的费率估算。当前列表请参见https://oreil.ly/TB6IN。

如前所述，费率估算永远不可能完美无缺。一个常见的问题是基本需求可能会发生变化，调整均衡点，将价格（费用）提高到新的高度，或者将其降低到最低水平。如果费率下降，那么以前支付正常费率的交易现在可能支付较高的费率，并且将比预期更早地得到确认。对于您已经发送的交易，无法降低费率，因此您将被困在支付更高费率的情况下。但是，当费率上升时，有必要采取方法来增加这些交易的费率，这称为***费率提升***。比特币中常用的两种费率提升方法是替换交易（RBF）和子付费（CPFP）。

替换交易（RBF）费率提升

使用RBF(Replace by fee)费率提升增加交易费率的方法是创建一个支付更高费用的冲突版本的交易。如果两个或更多交易之间存在冲突，那么它们被认为是冲突的交易，因为只有其中一个可以包含在有效的区块链中，迫使矿工只能选择其中一个。冲突发生在两个或更多交易尝试花费相同UTXO的情况下，即它们各自包含具有相同输出点的输入（引用先前交易的输出）。

为了防止某人通过创建无限数量的冲突交易并将它们通过中继全节点网络发送来消耗大量带宽，比特币核心和其他支持交易替换的全节点要求每个替换交易支付的费率都比被替换的交易更高。比特币核心目前还要求替换交易支付的总费用高于原始交易，但这个要求产生了不良的副作用，开发人员正在寻找在撰写本文时删除它的方法。

比特币核心目前支持两种RBF的变体：

选择性RBF

一个未经确认的交易可以向矿工和全节点发出信号，表明交易的创建者希望允许它被更高费率的版本替换。这个信号和使用它的规则在BIP125中指定。截至目前，这已经在比特币核心中默认启用了数年。

完全RBF

任何未经确认的交易都可以被更高费率的版本替换。截至目前，此功能可以选择在比特币核心中启用（但默认情况下是禁用的）。

> **为什么有两种不同的RBF**
>
> 有两种不同版本的RBF的原因是，完全RBF一直存在争议。比特币的早期版本允许交易替换，但这种行为在几个版本中被禁用。在那段时间里，使用现在称为比特币核心的软件的矿工或全节点不会用任何不同版本替换他们收到的未确认交易的第一个版本。一些商家开始期望这种行为：他们假设任何支付适当费率的有效未确认交易最终都会成为已确认的交易，因此他们在收到这样的未确认交易后不久提供他们的商品或服务。
>
> 然而，比特币协议无法保证任何未确认的交易最终都会被确认。正如本章前面提到的，每个矿工都可以自行选择要尝试确认的交易，包括这些交易的哪个版本。比特币核心是开源软件，因此任何拥有其源代码副本的人都可以添加（或删除）交易替换功能。即使比特币核心不是开源的，比特币也是一个开放的协议，可以由足够有能力的程序员从零开始重新实现，从而允许重新实现者包含或不包含交易替换功能。
>
> 交易替换打破了一些商家的假设，即每个合理的未确认交易最终都会被确认。交易的另一个版本可以支付与原始版本相同的输出，但不需要支付任何输出。如果未确认交易的第一个版本支付给了商家，而第二个版本可能没有支付给他们。如果商家基于第一个版本提供了商品或服务，但第二个版本得到了确认，那么商家将无法收回成本。
>
> 一些商家和支持他们的人请求不要重新启用比特币核心中的交易替换功能。其他人指出，交易替换提供了一些好处，包括能够对最初支付过低费率的交易进行费用增加。
>
> 最终，致力于比特币核心开发的开发人员实施了一项妥协：不是允许每个未确认的交易都被替换（全面RBF），而是只编程比特币核心允许那些表示希望允许替换的交易被替换（Opt-in RBF）。商家可以检查他们收到的交易是否有Opt-in信号，并将这些交易与没有信号的交易区别对待。
>
> 这并不改变基本的担忧：任何人仍然可以更改他们的比特币核心副本，或创建一个重新实现，以允许全面RBF，并且一些开发人员甚至做到了这一点，但似乎很少有人使用他们的软件。
>
> 几年后，致力于比特币核心的开发人员稍微改变了妥协。除了默认保留Opt-in RBF外，他们添加了一个选项，允许用户启用全面RBF。如果足够多的挖矿算力和传播全节点启用了此选项，则任何未确认的交易最终都可以被支付更高费率的版本替换。截至目前，尚不清楚是否已经发生了这种情况。

作为用户，如果您计划使用RBF费率提升，您首先需要选择一个支持该功能的钱包，比如列在https://oreil.ly/IhMzx上具有“发送支持”的钱包之一。

作为开发者，如果您计划实现RBF费率提升，您首先需要决定是执行选择加入RBF还是全面RBF。在撰写本文时，选择加入RBF是确保有效的唯一方法。即使全面RBF变得可靠，也可能有几年时间，加入RBF交易的替代版本得到的确认速度会略快于全面RBF替代版本。如果选择加入RBF，您的钱包需要实现BIP125中指定的信号，这是对交易中任意一个序列字段的简单修改。如果选择全面RBF，您无需在交易中包含任何信号。除此之外，与RBF相关的其他事项对这两种方法都是相同的。

当您需要提升费率时，您只需创建一个新交易，该交易至少花费与您要替换的原始交易相同的一个或多个UTXO。您可能希望保留支付接收者（或接收者）的相同输出。您可以通过减少找零输出的价值或向交易中添加额外的输入来支付增加的费用。开发者应该为用户提供一个费率提升界面，让用户只需提供（或建议）需要增加的费率即可完成所有工作。

> 在创建同一交易的多个替代版本时要非常小心。您必须确保所有版本的交易都相互冲突。如果它们不是全部冲突，那么可能会有多个独立的交易得到确认，导致您向接收方支付过多费用。例如：
>
> - 交易版本0包括输入A。
> - 交易版本1包括输入A和B（例如，您必须添加输入B以支付额外的费用）。
> - 交易版本2包括输入B和C（例如，您必须添加输入C以支付额外的费用，但C足够大，以至于您不再需要输入A）。
>
> 在这种情况下，任何保存了交易版本0的矿工都可以确认它和交易版本2。如果两个版本都支付给了相同的接收方，他们将被支付两次（矿工将从两个独立的交易中获得交易费用）。
>
> 避免这个问题的简单方法是确保替代交易始终包括与之前版本的交易相同的所有输入。

RBF费率调整与其他类型的费率调整相比的优势在于它可以非常有效地利用区块空间。通常，替代交易与其替代的交易大小相同。即使它更大，它通常也与用户在第一次创建交易时如果按照增加的费率支付所创建的交易大小相同。

\ RBF费率调整的根本劣势在于，通常只能由交易的创建者执行——即为交易提供签名或其他认证数据的人或团队。一个例外是设计允许通过使用签名哈希标志（参见“签名哈希类型（SIGHASH）”页）添加额外输入的交易，但这也带来了自己的挑战。一般来说，如果您是未确认交易的接收者，想要使其更快确认（或者根本确认），则不能使用RBF费率调整；您需要其他的方法。

RBF还存在其他问题，我们将在“交易固定”中探讨。

子支付父（Child Pays for Parent，CPFP）费率调整

任何收到未确认交易输出的人都可以通过消费该输出来激励矿工确认该交易。您想要确认的交易称为父交易。花费父交易输出的交易称为子交易。

正如我们在“输出点（Outpoint）”中学到的那样，确认的每个输入都必须引用区块链中较早的交易的未使用输出（无论是同一个区块中还是之前的区块）。这意味着希望确认子交易的矿工必须确保其父交易也被确认。如果父交易尚未被确认，但子交易支付的费用足够高，矿工可以考虑在同一个区块中确认它们是否会有利可图。

为了评估挖掘父交易和子交易的盈利性，矿工将它们视为一个包含有总大小和总费用的交易包，从中可以将费用除以大小以计算包费率。然后，矿工可以按照费率对他们知道的所有单个交易和交易包进行排序，并将收益最高的交易包含在他们尝试挖掘的区块中，直到达到允许包含在一个区块中的最大大小（权重）。为了找到更多可能有利可图的包，矿工可以评估跨多个代的包（例如，将未确认的父交易与其子交易和孙子交易相结合）。这称为祖先费率挖掘。

Bitcoin Core多年来一直实现了祖先费率挖掘，据信，目前几乎所有矿工都在使用该功能。这意味着钱包可以使用此功能通过使用子交易支付其父交易（CPFP）来提高收到的交易的费率。

CPFP相对于RBF有几个优势。任何收到交易输出的人都可以使用CPFP，包括支付接收者和支付者（如果支付者包含了找零输出）。此外，它也不需要替换原始交易，这使得它对某些商家的影响较小。

与RBF相比，CPFP的主要劣势是通常使用更多的区块空间。在RBF中，费率提升交易通常与其替代交易大小相同。在CPFP中，费率提升会增加一个完整的单独交易。使用额外的区块空间需要支付超出费率提升成本之外的额外费用。

CPFP存在一些挑战，其中一些我们将在“交易固定”中探讨。我们特别需要提到的另一个问题是最小中继费率问题，这个问题通过包传递来解决。

交易包中继

比特币核心的早期版本没有对它们存储在内存池中的未确认交易数量设置任何限制（请参阅第244页的“内存池和孤块池”）。当然，无论是内存（RAM）还是磁盘空间，计算机都有物理限制——一个完整的节点无法存储无限数量的未确认交易。比特币核心的后续版本将内存池的大小限制在大约一天的交易量，仅存储具有最高费率的交易或交易包。

这对大多数情况都非常有效，但它会产生依赖性问题。为了计算交易包的费率，我们需要父交易和子交易——但如果父交易的费率不够高，它就不会保留在节点的内存池中。如果一个节点收到一个子交易而没有访问其父交易，它就无法对该交易进行任何操作。

解决这个问题的方法是将交易作为一个整体进行中继，称为***交易包中继***，这样接收节点就可以在对任何单个交易进行操作之前评估整个包的费率。截至目前，致力于比特币核心的开发人员在实施事务包中继方面取得了显著进展，而且在本书出版时可能会提供一个有限的早期版本。

交易包中继对于基于时间敏感的预签名交易协议特别重要，例如闪电网络（LN）。在非合作情况下，一些预签名交易无法使用RBF进行费率提升，因此它们必须依赖CPFP。在这些协议中，一些交易可能也会在需要广播之前很长时间被创建，从而有效地无法估计适当的费率。如果一个预签名交易的费率低于进入节点内存池所需的金额，那么就无法通过子交易对其进行费率提升。如果这阻止了交易及时确认，一个诚实的用户可能会损失金钱。交易包中继是解决这个关键问题的解决方案。

交易固定

虽然我们描述的基本情况下RBF和CPFP费率提升都能正常工作，但与这两种方法相关的规则旨在防止对矿工和中继完整节点的拒绝服务攻击。这些规则的一个不幸副作用是它们有时会阻止某人能够使用费率提升。使得无法或难以对交易进行费率提升被称为***交易固定***。

一个主要的拒绝服务问题围绕着交易关系的影响。每当交易的输出被花费时，该交易的标识符（txid）将被子交易引用。然而，当一个交易被替换时，替换交易具有不同的txid。如果替换交易得到确认，则其后代不能包含在同一区块链中。重新创建和重新签名后代交易是可能的，但这并不保证会发生。这对RBF和CPFP有相关但不同的影响：

- 在RBF的情况下，当比特币核心接受替换交易时，通过忘记原始交易及所有依赖于该原始交易的后代交易，简化了事务处理。为了确保矿工接受替换交易更加有利可图，比特币核心只有在替换交易支付的费用高于将被遗忘的所有交易的费用时才接受替换交易。这种方法的缺点是，Alice可以创建一个支付给Bob的小交易。然后Bob可以使用他的输出来创建一个大的子交易。如果Alice想要替换她的原始交易，她需要支付的费用要比她和Bob最初支付的费用都要高。例如，如果Alice的原始交易大约为100个vbytes，Bob的交易大约为100,000个vbytes，并且他们都使用相同的费率，那么Alice现在需要支付比她最初支付的费用多1000多倍才能通过RBF提升她的交易费率。
- 在CPFP的情况下，每当节点考虑将一个事务包含在一个区块中时，它必须从它想要考虑的任何其他事务包中移除该事务包中的交易。例如，如果一个子交易支付给25个祖先，并且每个祖先有25个其他子代，那么在区块中包含该事务包需要更新大约625个事务包（25的平方）。类似地，如果从节点的内存池中移除一个具有25个后代的交易（例如被包含在一个区块中），并且每个后代都有25个其他祖先，则需要更新另外625个事务包。每次我们将参数加倍（例如，从25增加到50），我们的节点需要执行的工作量就增加了四倍。

此外，如果某个交易及其所有后代的替代版本被挖掘出来，将这个交易及其后代长期保留在内存池中是没有用的——除非发生罕见的区块链重新组织，否则这些交易都无法被确认。比特币核心将从其内存池中删除每个无法在当前区块链上确认的交易。在最坏的情况下，这可能会浪费大量节点的带宽，并可能被用来阻止交易正确传播。

为了防止这些问题以及其他相关问题，比特币核心限制了父交易在其内存池中最多具有25个 祖先或后代，并将所有这些交易的总大小限制为100,000 vbytes。这种方法的缺点是，如果一个交易已经有太多的后代（或者它及其后代太大），用户将无法创建CPFP费率提升。

\ 交易固定可能是意外发生的，但它也代表了诸如LN等多方时间敏感协议的严重漏洞。如果您的交易对手可以通过截止日期阻止您的某个交易确认，他们可能会从您那里窃取钱财。

协议开发人员多年来一直致力于减轻交易固定问题。其中一个部分解决方案在第213页的“CPFP Carve Out and Anchor Outputs”中描述。还提出了几种其他解决方案，截至目前，至少有一个解决方案正在积极开发中——短暂锚点。

CPFP削减和锚定输出

在2018年，LN的开发人员遇到了一个问题。他们的协议使用需要两个不同方的签名的交易。双方都不愿意信任对方，因此他们在协议中不需要信任的点上签署交易，允许他们两者之一在以后的某个时间广播其中一个交易，而另一方可能不想要（或无法）履行其义务。这种方法的问题在于，这些交易可能需要在未来的某个不确定的时间广播，超出了任何合理的能力来估算这些交易的适当费率。

理论上，开发人员可以设计他们的交易以允许使用RBF（使用特殊的sighash标志）或CPFP进行费率提升，但这两种协议都容易受到交易固定的影响。鉴于所涉及的交易是时间敏感的，允许交易对手使用交易固定来延迟交易确认很容易导致一种可重复利用的利用方法，恶意方可能会用来从诚实方那里窃取钱财。

LN开发人员Matt Corallo提出了一个解决方案：给予CPFP费率提升规则一个特殊的例外，称为CPFP削减。CPFP的正常规则禁止在以下情况下包含额外的后代：如果这将导致一个父事务有26个或更多的后代，或者如果这将导致一个父事务及其所有后代总共超过100,000个vbytes。根据CPFP削减的规则，即使这将超出其他限制，也可以将大小不超过1,000个vbytes的单个额外交易添加到一个包中，只要它是一个没有未确认祖先的未确认交易的直接后代。

例如，Bob和Mallory共同签署了一个具有两个输出的交易，一个输出分别给他们两个人。Mallory广播了该交易，并使用她的输出附加了25个子交易或等于100,000个vbytes大小的任何较小数量的子交易。如果没有削减，Bob将无法为他的输出附加另一个子交易以进行CPFP费率提升。有了削减，只要他的子交易的大小不超过1,000个vbytes（这应该是足够的空间），他就可以花费交易中的两个输出之一，即属于他的输出。

不允许多次使用CPFP削减，因此它只适用于双方协议。已经有提议将其扩展到涉及更多参与者的协议，但对此并没有太多的需求，开发人员正在专注于构建更通用的解决方案来对抗交易固定攻击。

截至目前，大多数流行的LN实现使用一种称为锚定输出的交易模板，该模板设计用于与CPFP削减一起使用。

将费用添加到交易中

交易的数据结构没有专门用于费用的字段。相反，费用被暗示为输入总和与输出总和之间的差异。在所有输出从所有输入中扣除后剩余的任何额外金额都是由矿工收取的费用：

Fees = Sum(Inputs) - Sum(Outputs)

这是交易中一个有些令人困惑的元素，也是一个重要的要点要理解，因为如果您正在构建自己的交易，您必须确保不会因为未花费的输入而意外地包含一个非常大的费用。这意味着您必须考虑所有的输入，必要时通过创建找零来补足，否则您最终会给矿工一个非常大的小费！

例如，如果您花费一个20比特币的UTXO来进行1比特币的支付，您必须包括一个19比特币的找零输出返回到您的钱包。否则，这19比特币的“剩余”将被计为交易费用，并将由挖掘您交易的矿工在区块中收取。尽管您将获得优先处理并使矿工非常高兴，但这可能不是您的初衷。

> 如果您在手动构建的交易中忘记添加找零输出，您将支付找零作为交易费用。“找零请收下！”可能不是您的初衷。

\

时间锁对抗费用抢夺

费用抢夺是一种理论上的攻击场景，其中矿工试图重写过去的区块，从未来的区块“抢夺”高费用的交易，以最大化他们的盈利能力。

例如，假设现存最高的区块是第100,000号区块。如果一些矿工不是尝试挖掘第100,001号区块以延长链条，而是试图重新挖掘第100,000号区块。这些矿工可以选择在他们的候选区块#100,000中包括任何尚未被挖掘的有效交易。他们不必重新挖掘包含相同交易的区块。事实上，他们有动机选择在他们的区块中包含最赚钱的（每千字节最高费用）交易。他们可以包括任何在“旧”区块#100,000中的交易，以及当前内存池中的任何交易。实际上，他们在重新创建区块#100,000时可以将交易从“现在”拉到“过去”。

今天，这种攻击并不是非常有利可图，因为区块补贴远高于每个区块的总费用。但在将来的某个时候，交易费用将成为奖励的主要部分（甚至全部）。到那时，这种情况将不可避免地发生。

一些钱包通过创建具有锁定时间的交易来阻止费用抢夺，限制这些交易只能被包含在下一个区块或任何之后的区块中。在我们的情景中，我们的钱包会在它创建的任何交易上将锁定时间设置为100,001。在正常情况下，这个锁定时间没有任何效果——交易无论如何只能被包含在第100,001号区块中；它是下一个区块。

但在重新组织攻击下，矿工将无法从内存池中拉取高费用的交易，因为所有这些交易都将被时间锁定到第100,001号区块。他们只能重新挖掘第100,000号区块，使用当时有效的任何交易，实际上没有获得新的费用。

这并不能完全阻止费用抢夺，但它确实在某些情况下降低了其盈利能力，并有助于在区块补贴减少时保持比特币网络的稳定性。我们建议所有钱包在不影响锁定时间字段的其他用途时，实施反费用抢夺功能。

随着比特币不断成熟，以及补贴持续下降，费用对比特币用户变得越来越重要，无论是在他们的日常使用中快速确认交易，还是为矿工提供继续用新的工作量证明保护比特币交易的激励。

综合介绍

\ 比特币是建立在互联网之上的对等网络架构。 对等网络或P2P的术语意味着参与网络的全部节点彼此对等，它们都能执行相同的功能，并且没有“特殊”的节点。网络节点以“平面”拓扑连接成网状网络。在网络内部没有服务器、没有中心化的服务，也没有层次结构。P2P网络中的节点同时提供和消耗服务。P2P网络具有固有的韧性、去中心化和开放性。一个著名的P2P网络架构示例是早期的互联网本身，在IP网络中的节点是平等的。今天的互联网架构更加层次化，但是互联网协议仍保留着其平面拓扑的本质。除了比特币和互联网之外，P2P技术最大和最成功的应用是文件共享，Napster是其先驱，而BitTorrent则是这种架构的最新演变。

比特币的P2P网络架构远不止是一种拓扑选择。比特币是一个经设计的P2P数字现金系统，网络架构既是这一核心特性的反映，也是其基础。对控制的去中心化是一个核心设计原则，只有通过平面和去中心化的P2P共识网络才能实现和维护。

术语“比特币网络”指的是运行比特币P2P协议的节点集合。除了比特币P2P协议之外，还有其他用于挖矿和轻量级钱包的协议。这些附加协议由网关路由服务器提供，这些服务器使用比特币P2P协议访问比特币网络，然后将该网络扩展到运行其他协议的节点。例如，Stratum服务器通过Stratum协议连接Stratum挖矿节点到主比特币网络，并将Stratum协议桥接到比特币P2P协议。除了基本的比特币P2P协议之外，我们将在本章中描述一些最常用的这些协议。

节点类型和任务

尽管比特币P2P网络中的全节点（对等节点）彼此平等，但它们可能根据它们支持的功能而承担不同的角色。比特币全节点验证区块并可能包含其他功能，如路由、挖矿和钱包服务。

一些节点，称为***归档全节点***，还维护着完整且最新的区块链副本。这些节点可以为仅存储区块链子集并使用***简化支付验证***（SPV）方法部分验证交易的客户端提供数据。这些客户端称为轻量级客户端。

矿工通过运行专用硬件来解决工作量证明算法来竞争创建新的区块。一些矿工运行全节点，验证区块链上的每个区块，而其他一些矿工则是参与池挖矿的客户端，依赖于池服务器提供工作。

用户钱包可能会连接到用户自己的全节点，这在桌面比特币客户端中有时是这样的情况，但许多用户钱包，尤其是在资源受限的设备上运行的钱包，如智能手机上的钱包，是轻量级节点。

除了比特币P2P协议中的主要节点类型外，还有运行其他协议的服务器和节点，如专用的挖矿池协议和轻量级客户端访问协议。

网络

截至目前，主比特币网络使用比特币P2P协议运行，约有10,000个监听节点运行着各种版本的比特币核心（Bitcoin Core），还有几百个节点运行着各种其他实现的比特币P2P协议，如BitcoinJ、btcd和bcoin。在比特币P2P网络中，只有很小一部分节点也是挖矿节点。各种个人和公司通过运行归档完整节点与比特币网络接口，这些节点拥有区块链的完整副本和网络节点，但没有挖矿或钱包功能。这些节点充当网络边缘路由器，允许在其上构建各种其他服务（如交易所、钱包、区块浏览器、商家支付处理等）。

紧凑块传输

当矿工发现一个新的区块时，他们会向比特币网络（包括其他矿工）宣布。发现该区块的矿工可以立即在其上构建；尚未获知该区块的其他矿工将继续在前一个区块上构建，直到他们获知为止。

如果在他们获知新区块之前，其他矿工之一创建了一个区块，他们的区块将与第一个矿工的新区块竞争。只有一个区块会被包含在所有完整节点使用的区块链中，并且矿工只有在被广泛接受的区块中才能获得收益。

先有第二个区块构建在其上的任何一个区块获胜（除非有另一个接近平局的情况），这称为区块发现竞赛，在图10-1中进行了说明。区块发现竞赛使得最大的矿工获得了优势，因此它们与比特币的基本去中心化原则相矛盾。为了防止区块发现竞赛，并允许任何规模的矿工平等参与比特币挖矿的抽奖，极其有用的是尽量减少一个矿工宣布新区块与其他矿工接收到该区块之间的时间。

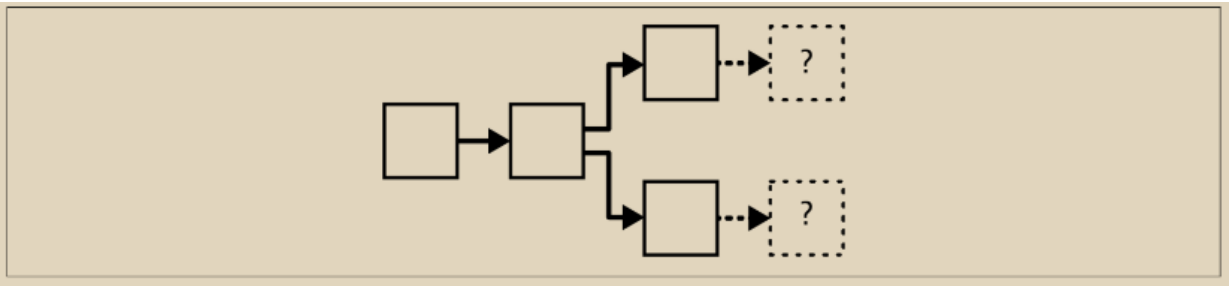


图 10-1. 需要挖矿竞赛的区块链分叉

\ 2015年，比特币核心的一个新版本添加了一项名为紧凑块传输（在BIP152中指定）的功能，可以更快地传输新区块并减少带宽使用。

作为背景，中继未确认交易的完整节点也会将许多这些交易存储在其内存池中（参见第244页的“内存池和孤立池”）。当其中一些交易在新的区块中得到确认时，节点不需要再收到这些交易的副本。

紧凑块传输不再接收冗余的未确认交易，而是允许节点将每个交易发送一个短的6字节标识符。当您的节点收到一个或多个标识符的紧凑块时，它会检查其内存池中是否存在这些交易，并在找到时使用它们。对于在本地节点的内存池中找不到的任何交易，您的节点可以向对等方发送请求以获取副本。

相反，如果远程节点认为您的节点的内存池中没有某些在区块中出现的交易，它可以在紧凑块中包含这些交易的副本。例如，比特币核心始终发送一个区块的 Coinbase 交易。

如果远程节点猜对了您的节点在其内存池中有哪些交易，以及哪些交易没有，它将以几乎理论上可能的效率发送一个区块（对于典型的区块，其效率将在97%到99%之间）。

> 紧凑块传输并不会减小区块的大小。它只是防止了节点已经拥有的信息的冗余传输。当一个节点之前没有关于一个区块的信息，例如当一个节点首次启动时，它必须接收每个区块的完整副本。

比特币核心当前实现了两种发送紧凑块的模式，如图10-2所示：

\ 低带宽模式

当您的节点请求对等方使用低带宽模式时（默认情况下），该对等方将告知您的节点一个新区块的32字节标识符（头哈希），但不会向您的节点发送任何有关该区块的详细信息。如果您的节点首先从其他来源获取到该区块，这将避免浪费更多的带宽来获取该区块的冗余副本。如果您的节点确实需要该区块，它将请求一个紧凑块。

高带宽模式

当您的节点请求对等方使用高带宽模式时，该对等方将在完全验证该区块的有效性之前，向您的节点发送一个新区块的紧凑块。该对等方进行的唯一验证是确保区块的头包含正确数量的工作证明。由于工作证明的生成成本昂贵（根据当前情况约为150,000美元），矿工不太可能伪造它只是为了浪费中继节点的带宽。在传输之前跳过验证允许新区块

在网络中的每个跳点之间以最小的延迟传播。

高带宽模式的缺点是，您的节点很可能会从每个选择的高带宽对等方接收到冗余信息。截至目前，比特币核心目前只要求三个对等方使用高带宽模式（并尝试选择已经快速宣布区块历史的对等方）。

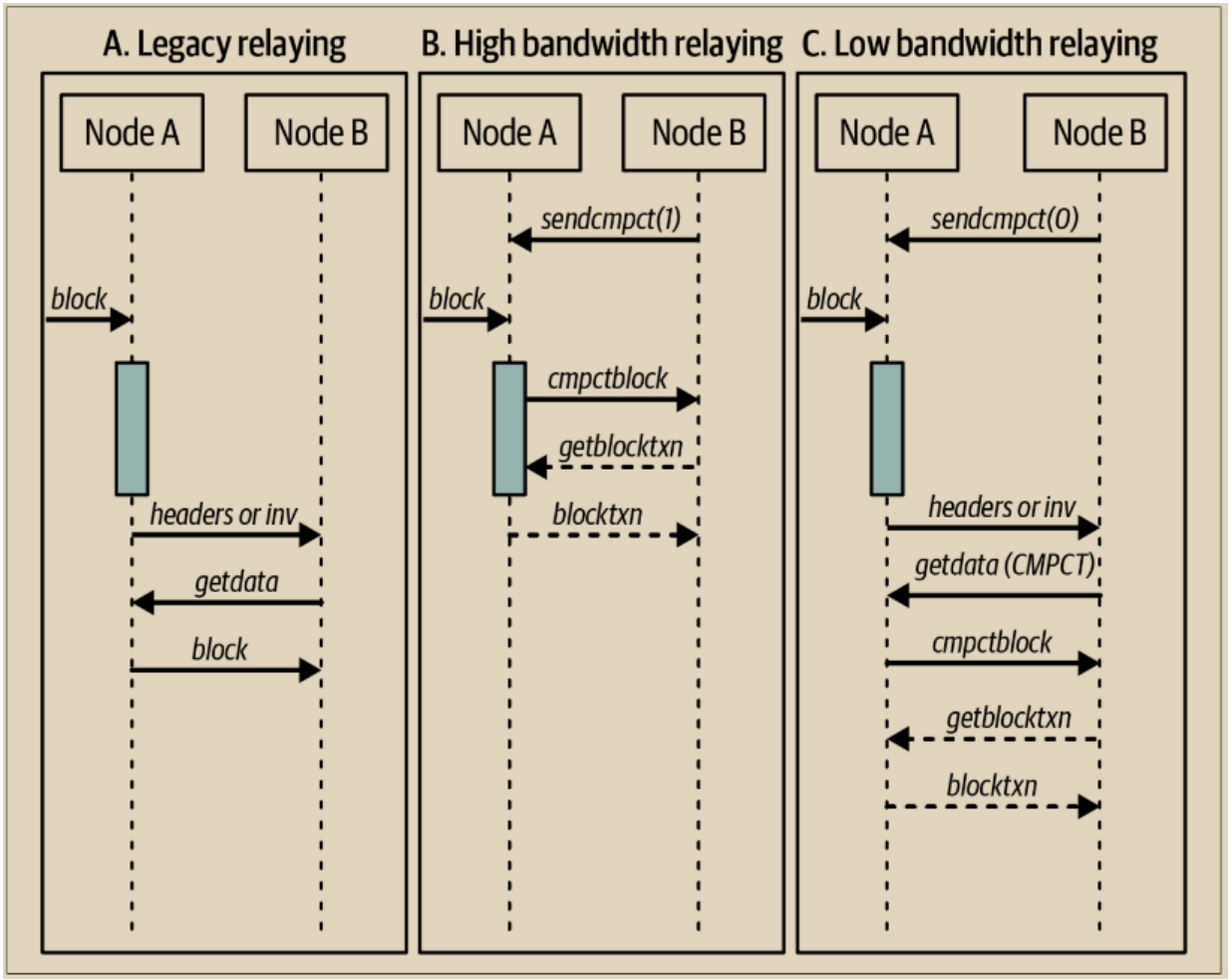


图 10-2. BIP152模式比较（来自BIP152）。阴影条表示节点验证区块所需的时间。

这两种方法的名称（取自BIP152）可能有些令人困惑。低带宽模式通过在大多数情况下不发送区块来节省带宽。高带宽模式比低带宽模式使用更多的带宽，但在大多数情况下，比实施紧凑块之前的区块传输使用的带宽要少得多。

私有区块传输网络

尽管紧凑块在最大程度上缩短了区块在网络中传播所需的时间，但仍然有可能进一步减少延迟。然而，与紧凑块不同，其他解决方案涉及到的权衡使它们无法在公共P2P中继网络中使用或不适合。因此，人们已经开始尝试为区块建立私有中继网络。

一种简单的技术是预先选择终点之间的路由。例如，一个中继网络，其中的服务器运行在靠近主要跨洋光纤线的数据中心中，可能能够比等待区块到达距离光纤线几公里之远的家庭用户节点更快地转发新的区块。

另一种更复杂的技术是前向纠错编码（FEC）。这允许将紧凑块消息分成几部分，并附加额外的数据到每个部分。如果有任何部分没有被接收到，那么这部分可以从已接收到的部分重建。根据设置，如果部分丢失，最多可以重建几部分。

FEC避免了由于底层网络连接的问题而导致紧凑块（或其某些部分）未到达的问题。这些问题经常发生，但我们通常不会注意到，因为我们大多使用自动重新请求丢失数据的协议。然而，请求丢失的数据会使接收数据的时间延长三倍。例如：

1. Alice向Bob发送一些数据。
2. Bob未收到数据（或数据损坏）。Bob重新向Alice请求数据。
3. Alice再次发送数据。

第三种技术是假设接收数据的所有节点几乎都具有相同的内存池中的大部分交易，因此它们都可以接受相同的紧凑块。这不仅节省了在每个跳点计算紧凑块的时间，而且意味着每个跳点可以在验证它们之前简单地将FEC数据包中继到下一个跳点。

前述每种方法的权衡是，它们与中心化结构很好地配合，但在无法相互信任的分散网络中效果不佳。位于数据中心的服务器需要花费金钱，并且通常可以被数据中心的运营者访问，这使它们比安全的家用计算机不太可靠。在验证之前传输数据很容易浪费带宽，因此它只能在私有网络中合理地使用，在这种网络中各方之间存在一定程度的信任和责任。

原始的比特币中继网络是由开发者Matt Corallo于2015年创建的，旨在实现矿工之间的快速区块同步，延迟极低。该网络由托管在全球基础设施上的几个虚拟专用服务器（VPSes）组成，用于连接大多数矿工和矿池。

2016年，随着由开发者Matt Corallo创建的*快速互联网比特币中继引擎*（FIBRE）的推出，原始的比特币中继网络被替换。FIBRE是一种软件，允许在节点网络中运行基于UDP的中继网络，用于中继区块。FIBRE实现了FEC和紧凑块优化，进一步减少了传输数据量和网络延迟。

网络发现

当新节点启动时，它必须发现网络上的其他比特币节点才能参与其中。要启动此过程，新节点必须至少发现网络上的一个现有节点并连接到它。其他节点的地理位置无关紧要；比特币网络拓扑结构不是地理上定义的。因此，可以随机选择任何现有的比特币节点。

要连接到已知的对等方，节点会建立一个TCP连接，通常连接到端口8333（通常被称为比特币使用的端口），或者如果提供了其他端口，则连接到替代端口。建立连接后，节点将通过发送版本消息（见图10-3）开始“握手”，该消息包含基本的识别信息，包括：

版本（Version）：

客户端“发言”的比特币P2P协议版本（例如，70002）

本地服务（nLocalServices）：

节点支持的本地服务列表

时间（nTime）：

当前时间

addrYou：

远程节点的IP地址，从本节点看到的

addrMe：

本地节点的IP地址，由本地节点发现的

subver：

显示在此节点上运行的软件类型的子版本（例如，/Satoshi:0.9.2.1/）

最佳高度（BestHeight）：

此节点区块链的块高度

fRelay：

由BIP37添加的字段，用于请求不接收未确认的交易

版本消息始终是任何对等方向另一个对等方发送的第一条消息。接收版本消息的本地对等方将检查远程对等方报告的版本，并决定远程对等方是否兼容。如果远程对等方是兼容的，则本地对等方将确认版本消息并通过发送verack建立连接。

新节点如何找到对等节点？第一种方法是使用一些DNS种子进行DNS查询，这些DNS种子是提供比特币节点IP地址列表的DNS服务器。其中一些DNS种子提供稳定的比特币监听节点的IP地址静态列表。一些DNS种子是BIND（Berkeley Internet Name Daemon）的自定义实现，它们从由网络爬虫或长时间运行的比特币节点收集的IP地址列表中返回随机子集。比特币核心客户端包含几个不同DNS种子的名称。不同DNS种子的所有权和实现的多样性为初始引导过程提供了高可靠性水平。在比特币核心客户端中，使用DNS种子的选项由选项开关-dnsseed控制（默认设置为1，以使用DNS种子）。

另外，一个对网络一无所知的引导节点必须至少给出一个比特币节点的IP地址，之后它可以通过进一步的介绍建立连接。命令行参数-seednode可用于仅使用它作为种子进行介绍的连接到一个节点。在使用初始种子节点进行介绍后，客户端将与其断开连接并使用新发现的对等节点。

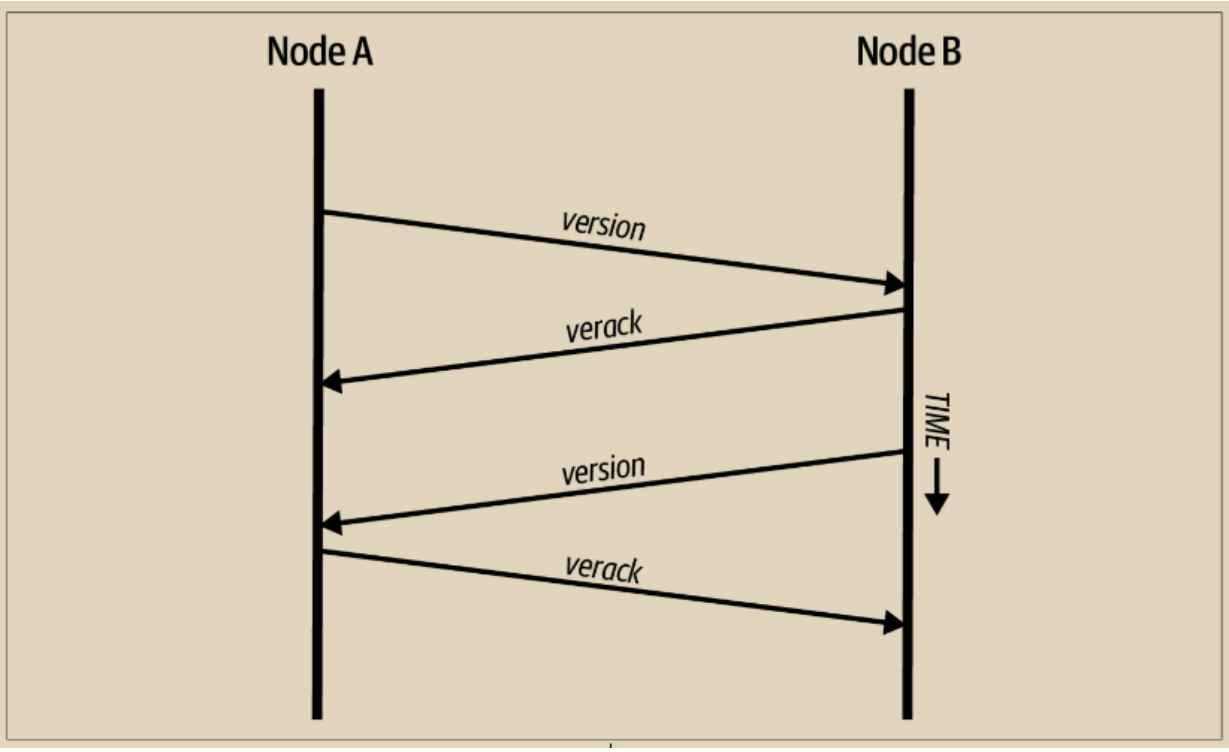


图 10-3. 对等方之间的初始握手

一旦建立了一个或多个连接，新节点将向其邻居发送一个包含自己 IP 地址的 addr 消息。邻居们将转发 addr 消息给它们的邻居，确保新连接的节点变得广为人知并且连接更加稳固。此外，新连接的节点可以向其邻居发送 getaddr，请求它们返回其他节点的 IP 地址列表。这样，一个节点就可以找到要连接的节点，并在网络上广告自己的存在，以便其他节点找到它。下图展示了地址发现协议。

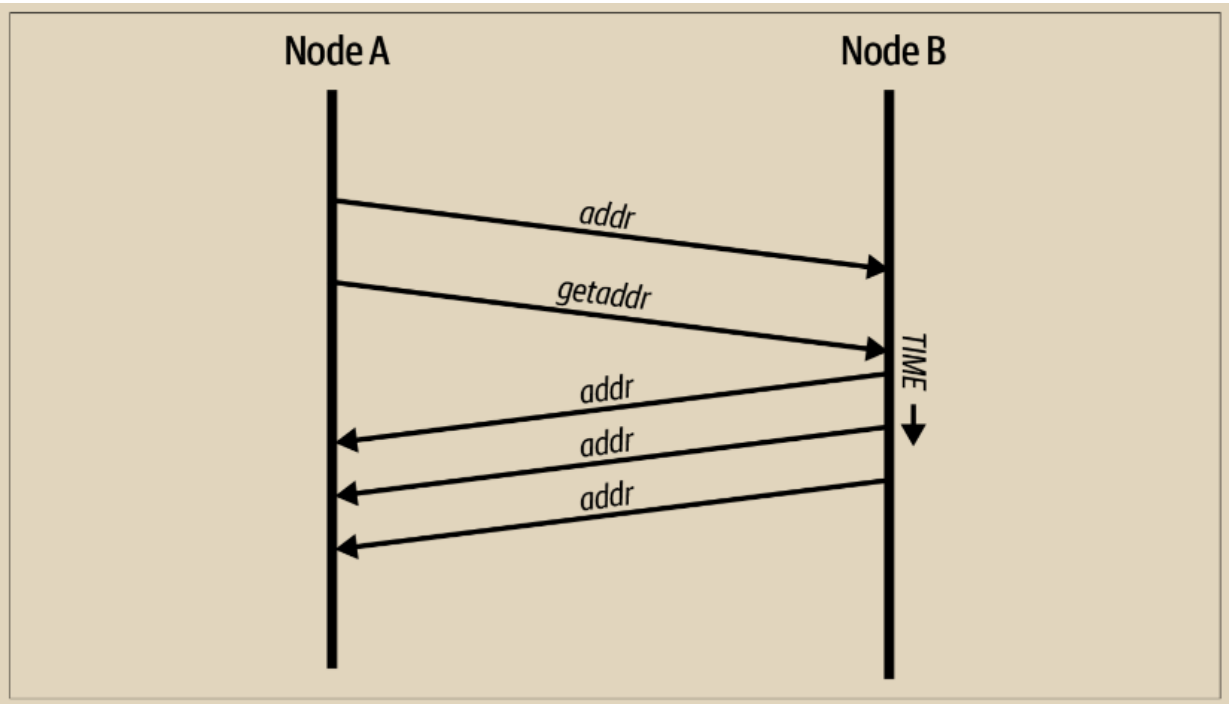


图 10-4. 地址传播和发现

\ 节点必须连接到几个不同的对等节点才能建立进入比特币网络的多样化路径。路径并不可靠 - 节点会不断加入和离开网络 - 因此节点必须在失去旧连接时继续发现新节点，并在引导其他节点时提供帮助。引导只需要连接到一个节点，因为第一个节点可以向其对等节点提供介绍，而这些对等节点可以提供进一步的介绍。连接到多个节点是不必要

的，也是对网络资源的浪费。引导完成后，节点将记住其最近成功的对等节点连接，因此如果重新启动，它可以快速重新与其以前的对等网络建立连接。如果没有任何以前的对等节点响应其连接请求，则节点可以再次使用种子节点进行引导。

在运行比特币核心客户端的节点上，您可以使用命令 getpeerinfo 列出对等连接：

```
$ bitcoin - cli getpeerinfo
[{
    "id": 0,
    "addr": "82.64.116.5:8333",
    "addrbind": "192.168.0.133:50564",
    "addrlocal": "72.253.6.11:50564",
    "network": "ipv4",
    "services": "0000000000000409",
    "servicesnames": [
        "NETWORK",
        "WITNESS",
        "NETWORK_LIMITED"
    ],
    "lastsend": 1683829947,
    "lastrecv": 1683829989,
    "last_transaction": 0,
    "last_block": 1683829989,
    "bytessent": 3558504,
    "bytesrecv": 6016081,
    "conntime": 1683647841,
    "timeoffset": 0,
    "pingtime": 0.204744,
    "minping": 0.20337,
    "version": 70016,
    "subver": "/Satoshi:24.0.1/",
    "inbound": false,
    "bip152_hb_to": true,
    "bip152_hb_from": false,
    "startingheight": 788954,
    "presynced_headers": -1,
    "synced_headers": 789281,
    "synced_blocks": 789281,
    "inflight": [],
    "relaytxes": false,
    "minfeefilter": 0.00000000,
    "addr_relay_enabled": false,
    "addr_processed": 0,
    "addr_rate_limited": 0,
    "permissions": [],
    "bytessent_per_msg": {
        ...
    },
    "bytesrecv_per_msg": {
        ...
    },
    "connection_type": "block-relay-only"
}, ]
```

要覆盖对等节点的自动管理并指定 IP 地址列表，用户可以提供选项 -connect= 并指定一个或多个 IP 地址。如果使用了此选项，则节点将只连接到选定的 IP 地址，而不会自动发现和维护对等连接。如果连接上没有流量，节点将定期发送消息以维持连接。

如果节点在某个连接上长时间没有通信，就会被认为已断开连接，并寻找一个新的对等节点。因此，网络会动态调整以适应短暂的节点和网络问题，并在没有任何中央控制的情况下根据需要有机地增长和收缩。

全节点

\ 全节点是在具有最多工作量证明的有效区块链上验证每个区块中的每笔交易的节点。

全节点独立处理每个区块，从第一个区块（创世区块）开始，一直构建到网络中已知的最新区块。全节点可以独立且具有权威性地验证任何交易。全节点依赖于网络来接收关于新交易区块的更新信息，然后验证并将其合并到本地视图中，以确定哪些脚本控制着哪些比特币，即未花费交易输出（UTXO）的集合。

运行全节点可以让您获得纯粹的比特币体验：独立验证所有交易，无需依赖或信任其他系统。

有一些替代的全节点实现，使用了不同的编程语言和软件架构，或者做出了不同的设计决策。然而，最常见的实现是比特币核心（Bitcoin Core）。超过 95% 的比特币网络上的全节点运行着不同版本的比特币核心。它在版本消息中发送的子版本字符串中被标识为“Satoshi”，如我们之前所见的 getpeerinfo 命令所示；例如，/Satoshi:24.0.1/。

交换“存货”

在连接到对等节点后，全节点将尝试构建完整的区块头链。如果它是一个全新的节点，并且没有任何区块链，那么它只知道一个区块，即创世区块，该区块静态地嵌入在客户端软件中。从区块 #0（创世区块）开始，新节点将需要下载数十万个区块才能与网络同步并重新建立完整的区块链。

同步区块链的过程始于版本消息，因为其中包含 BestHeight，即节点当前的区块链高度（区块数量）。一个节点将会收到来自其对等节点的版本消息，了解它们各自拥有的区块数量，并将其与自己的区块链进行比较。对等节点将交换一个包含其本地区块链顶部区块哈希的 getheaders 消息。其中一个对等节点将能够识别接收到的哈希属于一个不在顶部的区块，而是属于一个旧的区块，从而推断出自己的本地区块链比远程节点的区块链更长。

拥有更长区块链的对等节点比其他节点拥有更多区块，并且可以确定其他节点需要哪些区块头来“赶上”。它将识别出要共享的前 2,000 个区块头，并使用一个 headers 消息进行分享。节点会持续请求额外的区块头，直到收到远程对等节点声称拥有的每个区块头为止。

同时，节点将开始使用一个 getdata 消息为先前收到的每个区块头请求区块。节点将从每个已选对等节点请求不同的区块，这使得它可以断开与明显比平均速度慢的对等节点的连接，以寻找更新（可能更快）的对等节点。

例如，假设一个节点只有创世区块。然后，它将从对等节点接收到一个包含链中接下来的 2,000 个区块头的 headers 消息。它将开始向所有连接的对等节点请求区块，并保持一个最多包含 1,024 个区块的队列。区块需要按顺序进行验证，因此，如果队列中最旧的区块（节点下一个需要验证的区块）尚未收到，则节点会断开与应该提供该区块的对等节点的连接。然后，它会找到一个新的对等节点，该对等节点可能能够在节点的所有其他对等节点能够提供 1,023 个区块之前提供一个区块。

每个区块到达后，它都会被添加到区块链中，正如我们将在第 11 章中看到的那样。随着本地区块链逐渐建立起来，将请求和接收更多的区块，这个过程会一直持续，直到节点赶上网络的其余部分。

每当节点长时间离线时，比较本地区块链与对等节点并检索任何缺失区块的过程都会发生。

轻量级客户端

\ 许多比特币客户端都设计用于空间和功耗受限的设备，比如智能手机、平板电脑或嵌入式系统。对于这些设备，使用一种简化的支付验证（SPV）方法，使它们能够在不验证完整区块链的情况下运行。这种类型的客户端称为轻量级客户端。

轻量级客户端仅下载区块头，不下载每个区块中包含的交易。没有交易的区块头链约比完整区块链小约 10,000 倍。轻量级客户端无法构建所有可用于支出的 UTXO 的完整图片，因为它们不知道网络上的所有交易。相反，它们使用略有不同的方法验证交易，依赖对等节点根据需求提供区块链相关部分的部分视图。

作为类比，全节点就像是身处陌生城市的游客，装备有每条街道和每个地址的详细地图。相比之下，轻量级客户端就像是身处陌生城市的游客，向随机的陌生人寻求逐步指示，而只知道一个主要大道。虽然两位游客都可以通过访问街道来验证其存在，但没有地图的游客不知道侧街上有什么，也不知道还有哪些其他街道。站在 23 Church Street 前，没有地图的游客无法知道城市中是否有十几个其他的“23 Church Street”地址，以及这是否是正确的地址。没有地图的游客最好的机会就是询问足够多的人，并希望其中一些人不是试图抢劫他。

轻量级客户端通过参考交易在区块链中的深度来验证交易。全节点将构建一个经过完全验证的数千个区块和数百万笔交易的链，一直向下（时间上）延伸到创世区块。轻量级客户端将验证所有区块的工作量证明（但不验证区块及其所有交易的有效性），并将该链与感兴趣的交易链接起来。

例如，当查看第 800,000 个区块中的交易时，全节点将验证自创世区块以来所有 800,000 个区块，并构建一个完整的 UTXO 数据库，通过确认交易存在且其输出仍未花费来验证交易的有效性。轻量级客户端只能验证交易是否存在。客户端使用 Merkle 路径将交易与包含它的区块建立关联。然后，轻量级客户端等待直到看到块 800,001 到 800,006 堆叠在包含该交易的块的上方，然后通过建立其位于块 800,006 到 800,001 下面的深度来验证它。网络中的其他节点接受区块 800,000 并且矿工们在其之上产生了六个以上的区块，通过代理证明交易确实存在。

轻量级客户端通常无法被说服相信某个交易确实存在于一个区块中，而事实上该交易并不存在。轻量级客户端通过请求 Merkle 路径证明并验证区块链中的工作量证明来确认一个交易在一个区块中的存在。然而，一个交易的存在可以对轻量级客户端“隐藏”。轻量级客户端确实可以验证一个交易存在，但不能验证某个交易，比如同一 UTXO 的双重花费，不存在，因为它没有所有交易的记录。这种漏洞可以被用于对轻量级客户端进行拒绝服务攻击或双重支付攻击。为了防御这一点，轻量级客户端需要随机连接到几个客户端，以增加与至少一个诚实节点接触的概率。这种随机连接的需要意味着轻量级客户端也容易受到网络分割攻击或 Sybil 攻击的影响，其中它们连接到虚假节点或虚假网络，并且无法访问诚实节点或真实的比特币网络。

对于许多实际目的来说，连接良好的轻量级客户端已经足够安全，实现了资源需求、实用性和安全性之间的平衡。然而，要实现绝对的安全性，运行一个全节点是无可替代的。

> 全节点通过检查下面的成千上万个区块的整个链条来验证交易，以确保UTXO存在且未被花费，而轻量级客户端仅证明交易存在，并检查包含该交易的区块是否被其上方的少数区块所覆盖。

为了获取验证交易是否属于链的块头，轻量级客户端使用getheaders消息。响应的对等节点将使用单个headers消息发送最多2,000个块头。请参阅图10-5中的插图。

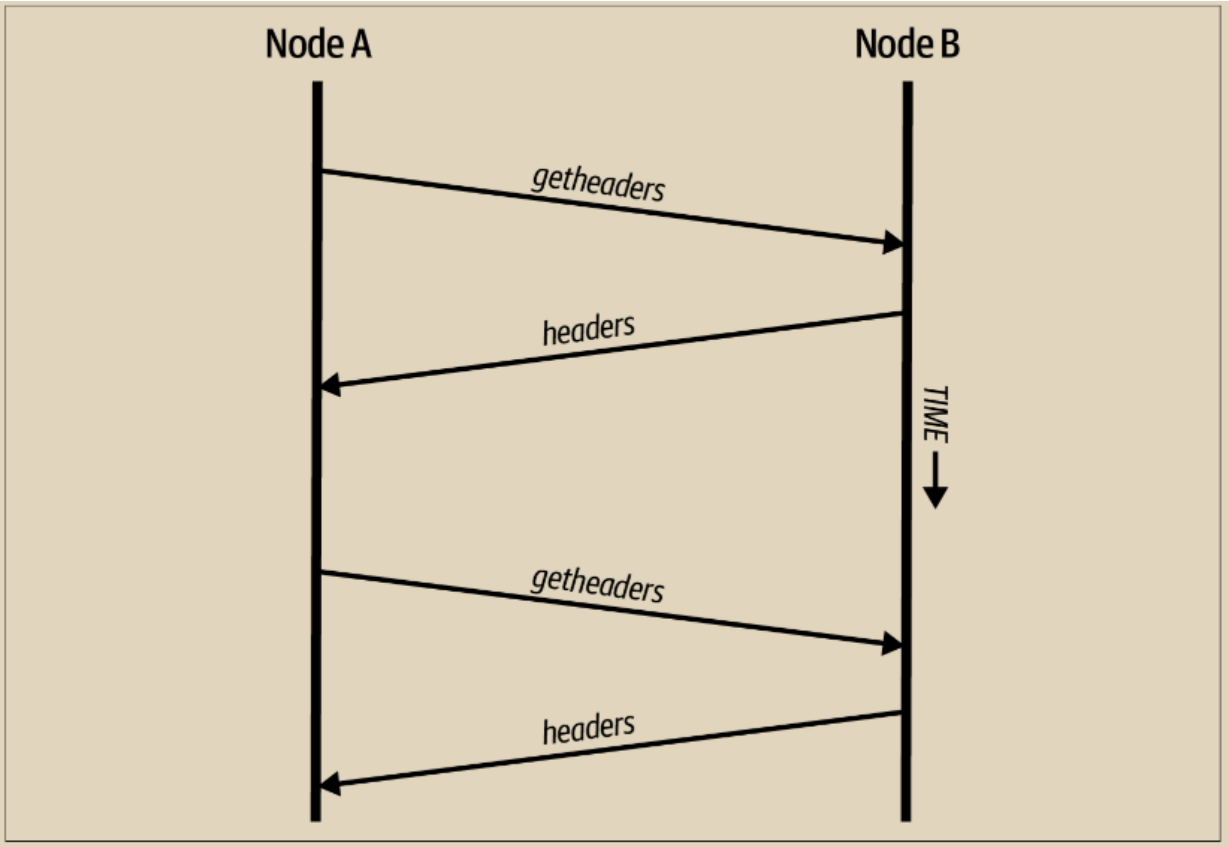


图 10-5. 轻量级客户端同步区块头

区块头允许轻量级客户端验证任何单个区块是否属于具有最多工作量证明的区块链，但它们并不告诉客户端哪些区块包含对其钱包有兴趣的交易。客户端可以下载每个区块并进行检查，但这将使用大量资源，相当于运行完整节点所需的资源的一部分，因此开发人员寻找其他解决方案。

\ 在轻量级客户端引入不久后，比特币开发人员添加了一项名为布隆过滤器的功能，旨在减少轻量级客户端需要使用的带宽，以了解其传入和传出交易。布隆过滤器允许轻量级客户端接收交易的子集，而不直接透露它们感兴趣的确切地址，通过使用概率而不是固定模式的过滤机制。

布隆过滤器

布隆过滤器是一种概率性搜索过滤器，用于描述所需的模式而不精确指定。布隆过滤器提供了一种有效的方式来表达搜索模式，同时保护隐私。轻量级客户端使用布隆过滤器向其对等节点请求与特定模式匹配的交易，而不会准确透露它们正在搜索的确切地址、密钥或交易。

在我们之前的类比中，一个没有地图的游客正在寻找一个特定的地址，“23 Church St.”。如果他们向陌生人询问这条街的方向，他们无意中就透露了自己的目的地。布隆过滤器就像是在询问：“这个街区是否有任何以 R-C-H 结尾的街道？”这样的问题相对于询问“23 Church St.”，透露的信息稍微少一些。通过调整搜索的精度，游客可以更详细地指定所需的地址，比如“以 U-R-C-H 结尾”，或者更不详细，比如“以 H 结尾”。通过改变搜索的精度，游客可以在获取更具体结果或更好的隐私之间进行权衡。如果他们要求一个不太具体的模式，他们会得到更多可能的地址和更好的隐私，但很多结果是不相关的。如果他们要求一个非常具体的模式，他们会得到更少的结果，但会失去隐私。

布隆过滤器通过允许轻量级客户端指定一个搜索模式来实现这一功能，该模式可以调整到精确度或隐私方面。一个更具体的布隆过滤器会产生准确的结果，但会透露轻量级客户端感兴趣的模式，从而透露用户钱包的地址。一个不太具体的布隆过滤器会产生更多关于更多交易的数据，其中许多与客户端无关，但会使客户端能够保持更好的隐私。

布隆过滤器工作原理

布隆过滤器被实现为一个可变大小的包含 N 个二进制数字（一个位域）和可变数量的 M 个哈希函数的数组。这些哈希函数被设计成总是产生一个介于 1 和 N 之间的输出，对应于二进制数字数组。哈希函数是确定性生成的，因此任何实现布隆过滤器的客户端都将始终使用相同的哈希函数，并为特定输入获得相同的结果。通过选择不同长度（N）的布隆过滤器和不同数量（M）的哈希函数，可以调整布隆过滤器，从而变化精度水平，进而影响隐私保护程度。

在图 10-6 中，我们使用一个非常小的 16 位数组和一组三个哈希函数来演示布隆过滤器的工作原理。

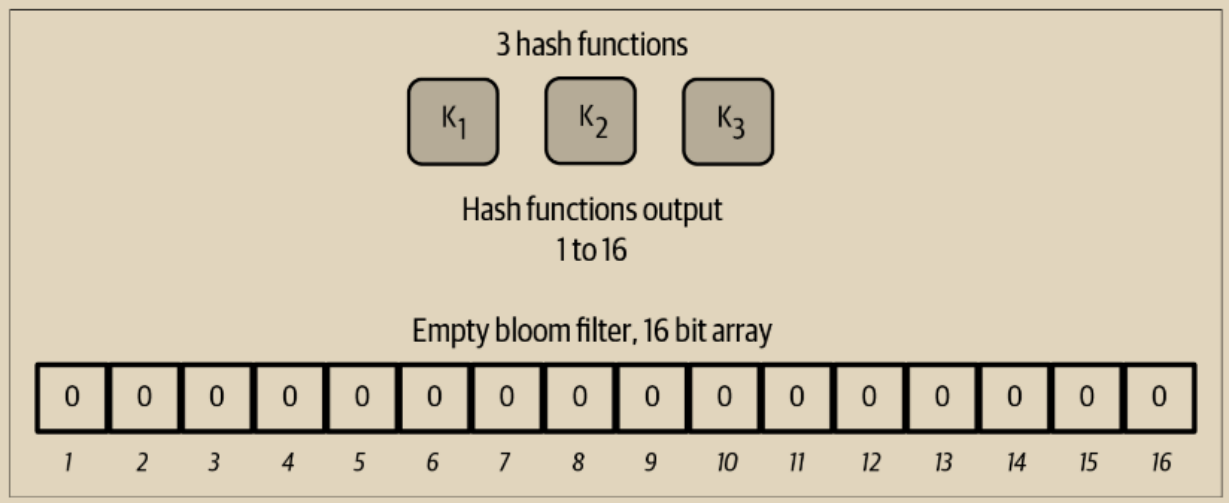


图 10-6. 一个简单的布隆过滤器示例，使用16位字段和三个哈希函数

布隆过滤器的初始化是将位数组全部设置为零。要向布隆过滤器添加一个模式，首先依次对该模式进行每个哈希函数的哈希计算。将第一个哈希函数应用于输入，得到一个介于1和N之间的数字。找到数组中相应的位（从1到N进行索引），并将其设置为1，从而记录哈希函数的输出。然后，使用下一个哈希函数来设置另一个位，依此类推。一旦所有M个哈希函数都被应用，搜索模式就会被“记录”在布隆过滤器中，作为M个从0变为1的位。

图10-7是将一个模式“A”添加到图10-6中显示的简单布隆过滤器的示例。

添加第二个模式与重复此过程一样简单。模式依次通过每个哈希函数进行哈希计算，然后通过将位设置为1来记录结果。需要注意的是，随着布隆过滤器填充更多模式，哈希函数的结果可能与已经设置为1的位重叠，此时该位不会更改。实质上，随着更多模式记录在重叠的位上，布隆过滤器开始饱和，其中更多的位被设置为1，过滤器的准确性降低。这就是为什么该过滤器是一种概率性数据结构的原因——随着添加更多模式，其准确性会降低。准确性取决于添加的模式数量与位数组（N）和哈希函数数量（M）的大小。更大的位数组和更多的哈希函数可以以更高的准确性记录更多的模式。较小的位数组或较少的哈希函数将记录较少的模式，并产生较低的准确性。

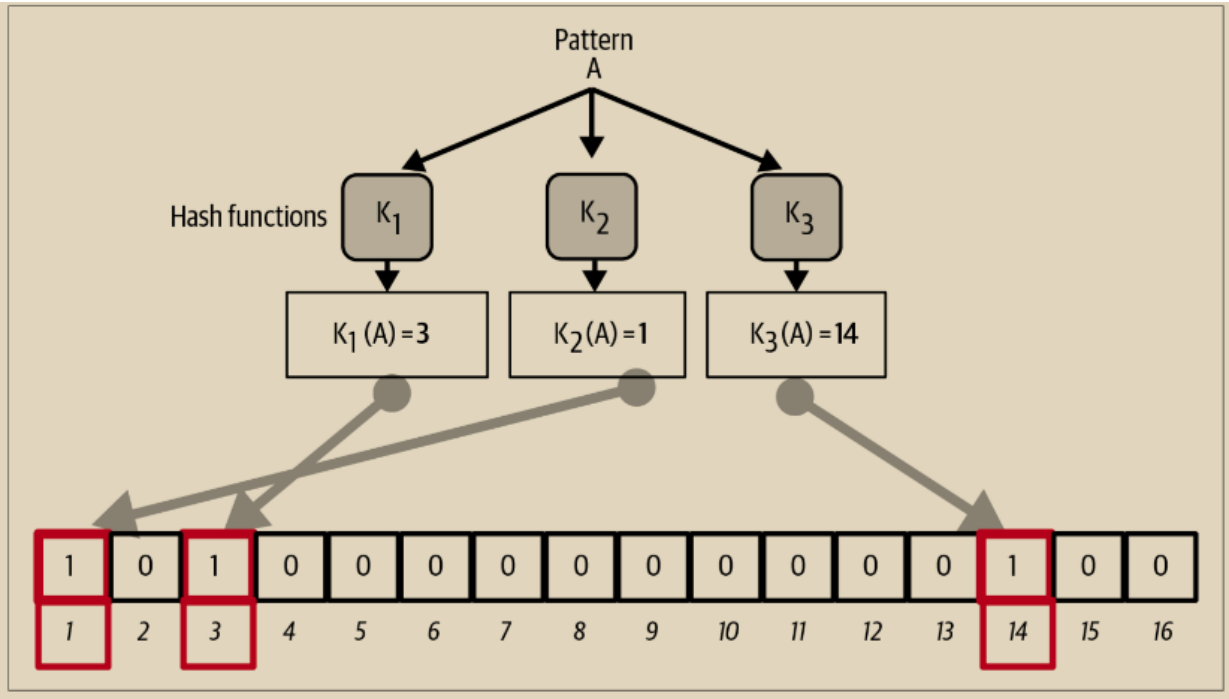


图 10-7. 将模式“A”添加到我们的简单布隆过滤器中

图10-8是将第二个模式“B”添加到简单布隆过滤器中的示例。

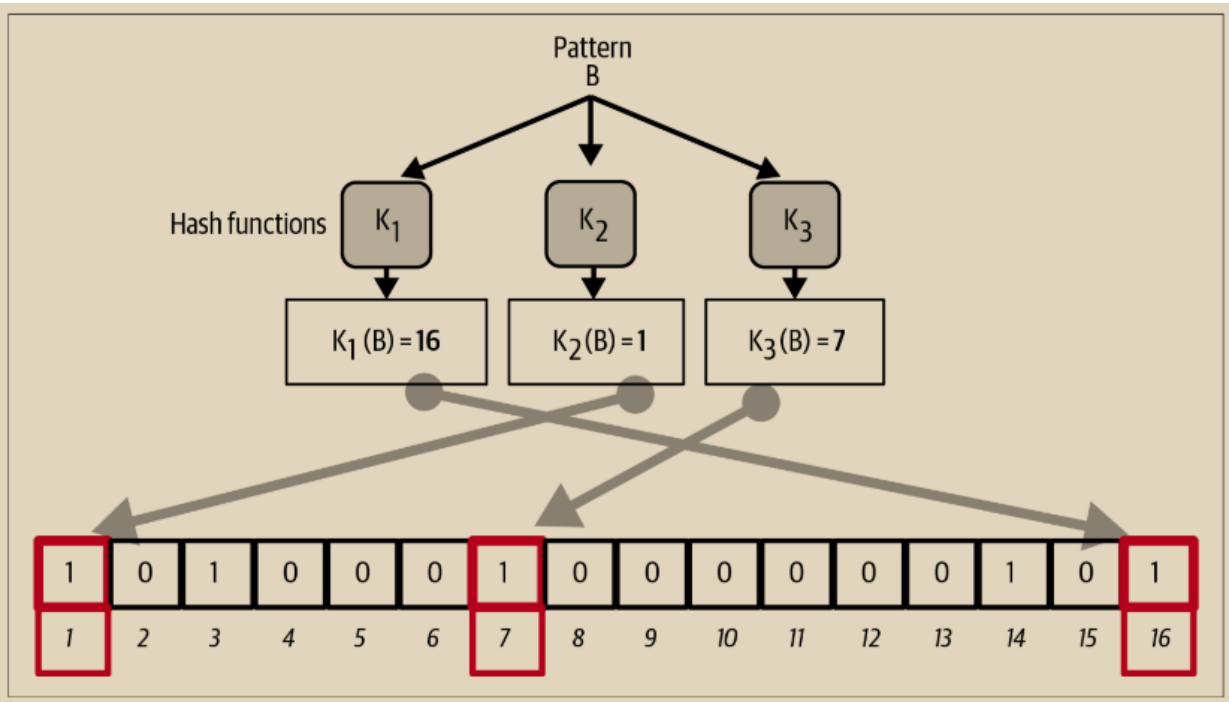


图 10-8. 将模式“B”添加到我们的简单布隆过滤器中。

\ 要测试一个模式是否存在于布隆过滤器中，需要对该模式应用每个哈希函数，并将结果与位数组进行比较。如果由哈希函数索引的所有位都被设置为1，那么该模式很可能被记录在布隆过滤器中。然而，由于这些位可能是由于多个模式的重叠而设置的，因此结果并不确定，而是概率性的。简单来说，布隆过滤器的正匹配是一个“也许，是”的情况。

图10-9是在简单布隆过滤器中测试模式“X”存在性的示例。由于相应的位被设置为1，所以该模式很可能匹配。

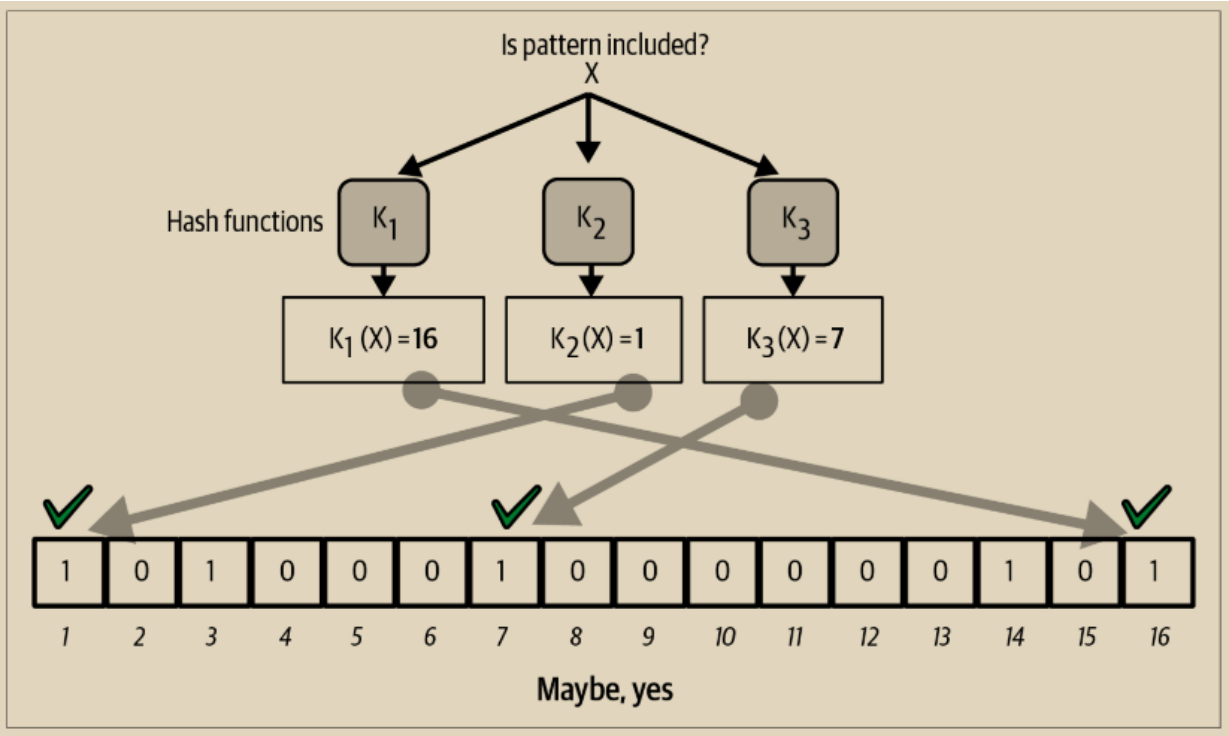


图 10-9. 在布隆过滤器中测试模式“X”的存在性。结果是一个概率性的正匹配，意味着“可能”

相反，如果对模式进行布隆过滤器测试，并且任何一个位设置为0，则证明该模式未记录在布隆过滤器中。负结果不是概率，而是确定性。简单来说，布隆过滤器的负匹配是“绝对不是！”

图10-10是对简单布隆过滤器中测试模式“Y”的存在性的示例。其中一个相应的位设置为0，因此该模式绝对不是匹配项。

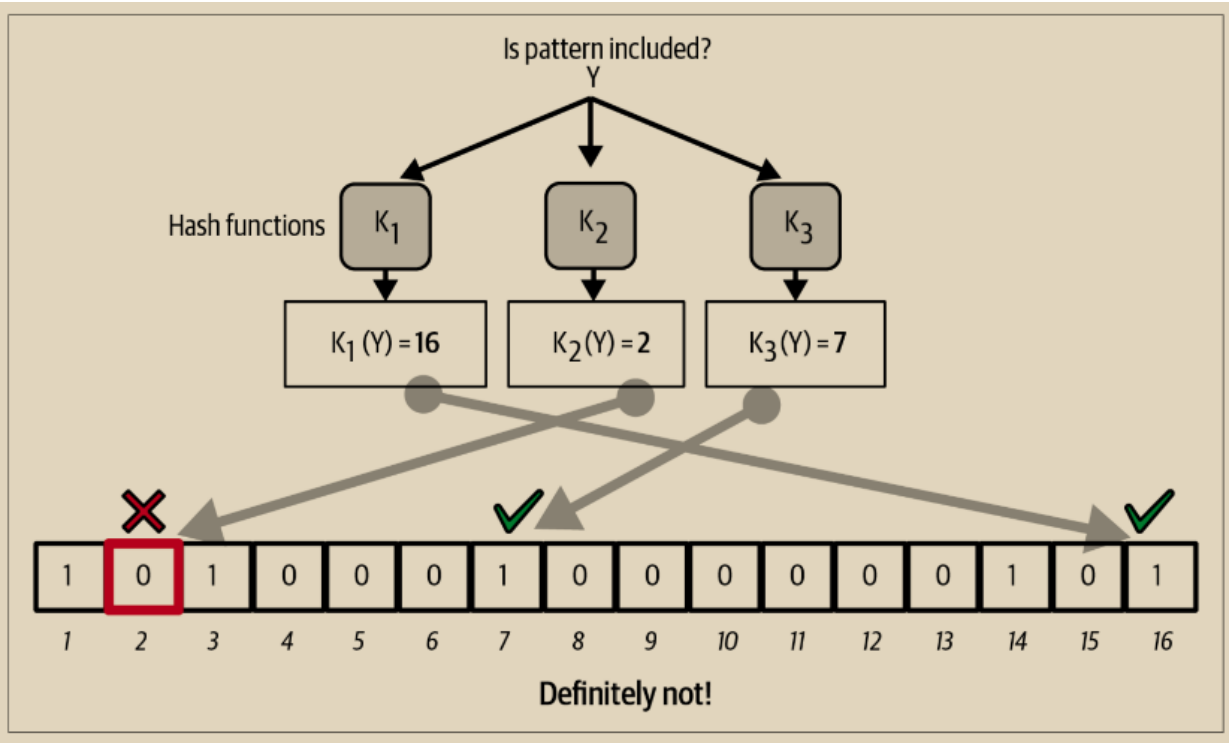


图 10-10. 测试在布隆过滤器中模式“Y”的存在性。结果是明确的负匹配，意味着“绝对不是！”

轻量级客户端如何使用布隆过滤器

\ 轻量级客户端使用布隆过滤器来过滤从其对等节点接收的交易（以及包含这些交易的区块），仅选择对轻量级客户端感兴趣的交易，而不会准确地揭示感兴趣的地址或密钥。

轻量级客户端将初始化一个布隆过滤器为“空”的状态；在这种状态下，布隆过滤器不会匹配任何模式。然后，轻量级客户端会列出其感兴趣的所有地址、密钥和哈希值。它会通过从其钱包控制的任何未花费交易输出（UTXO）中提取公钥哈希、脚本哈希和交易ID来执行此操作。然后，轻量级客户端将每个值添加到布隆过滤器中，以便如果这些模式存在于交易中，则布隆过滤器会“匹配”，而不会揭示模式本身。

然后，轻量级客户端将向对等节点发送一个filterload消息，其中包含用于连接的布隆过滤器。在对等节点上，布隆过滤器与每个传入交易进行匹配。完整节点会针对所有这些组件检查布隆过滤器，以寻找匹配，包括：

- 交易ID
- 每个交易输出脚本中的数据组件（脚本中的每个密钥和哈希）
- 每个交易输入
- 每个输入签名数据组件（或见证脚本）

通过检查所有这些组件，布隆过滤器可以用于匹配公钥哈希、脚本、OP_RETURN值、签名中的公钥，或者智能合约或复杂脚本的任何未来组件。

建立了过滤器后，对等节点将对每个交易输出进行布隆过滤器测试。只有与过滤器匹配的交易才会发送给客户端。

作为对客户端的getdata消息的响应，对等节点将发送一个merkleblock消息，其中包含与过滤器匹配的块头以及每个匹配交易的merkle路径（见“默克尔树”第252页）。然后，对等节点还会发送包含由过滤器匹配的交易的tx消息。

当完整节点发送交易给轻量级客户端时，轻量级客户端会丢弃任何误报，并使用正确匹配的交易来更新其UTXO集合和钱包余额。当它更新其自己的UTXO集合视图时，它还会修改布隆过滤器，以匹配任何引用它刚刚找到的UTXO的未来交易。然后，完整节点使用新的布隆过滤器来匹配新的交易，整个过程重复进行。

客户端设置布隆过滤器可以通过发送filteradd消息来交互式地添加模式到过滤器中。要清除布隆过滤器，客户端可以发送filterclear消息。因为不可能从布隆过滤器中删除模式，所以如果不再需要某个模式，则客户端必须清除并重新发送新的布隆过滤器。

轻量级客户端使用的网络协议和布隆过滤器机制在BIP37中定义。

不幸的是，在部署了布隆过滤器之后，很明显它们并没有提供太多的隐私。从对等节点接收到布隆过滤器的完整节点可以将该过滤器应用于整个区块链，以找到所有客户端的交易（以及误报）。然后，它可以查找交易之间的模式和关系。随机选择的误报交易不太可能具有从输出到输入的父子关系，但是来自用户钱包的交易很可能具有该关系。如果所有相关交易具有某些特征，例如至少一个P2PKH输出，则不具备该特征的交易可以假定不属于该钱包。

还发现，特殊构造的过滤器可能会迫使处理它们的完整节点执行大量工作，这可能导致拒绝服务攻击。

由于这两个原因，Bitcoin Core最终限制了对布隆过滤器的支持，仅允许节点运营者明确允许的IP地址上的客户端使用。

这意味着需要一种替代方法来帮助轻量级客户端找到它们的交易。

紧凑块过滤器

\ 2016年，一位匿名开发者在Bitcoin-Dev邮件列表上提出了一个逆向布隆过滤器过程的想法。根据BIP37布隆过滤器，每个客户端将它们的地址哈希为一个布隆过滤器，而节点则对每个交易的部分进行哈希以尝试匹配该过滤器。在新提议中，节点将每个区块中的交易部分进行哈希以创建一个布隆过滤器，而客户端则对它们的地址进行哈希以尝试匹配该过滤器。如果客户端找到匹配项，它们将下载整个区块。

> 尽管名称相似，但BIP152紧凑块和BIP157/158紧凑块过滤器是无关的。

这使得节点能够为每个区块创建一个单一的过滤器，并将其保存到磁盘上并重复提供，从而消除了BIP37的拒绝服务漏洞。客户端不向完整节点提供有关其过去或未来地址的任何信息。它们只下载可能包含数千个并非由客户端创建的交易的区块。它们甚至可以从不同的对等体下载每个匹配的区块，从而使完整节点更难以连接属于单个客户端的多个区块中的交易。

这种服务器生成的过滤器并不提供完美的隐私保护；它仍然给完整节点带来一些成本（并且需要轻量级客户端为区块下载使用更多带宽），而且这些过滤器只能用于已确认的交易（而非未确认的交易）。然而，它比BIP37客户端请求的布隆过滤器更加隐私和可靠。

在描述了基于布隆过滤器的原始想法之后，开发者们意识到了一种更好的服务器生成的过滤器的数据结构，称为Golomb-Rice编码集（GCS）。

Golomb-Rice编码集（GCS）

假设Alice想要将一组数字发送给Bob。简单的方法是直接将整个数字列表发送给他：

849

653

476

900

379

但是有一种更有效率的方法。首先，Alice将列表按数字顺序排列：

\ 379

476

653

849

900

然后，Alice发送第一个数字。对于剩余的数字，她发送该数字与前一个数字的差异。例如，对于第二个数字，她发送97（476 - 379）；对于第三个数字，她发送 177（653 - 476）；依此类推：

379

97

177

196

51

我们可以看到，有序列表中两个数字之间的差异产生的数字比原始数字要短。收到这个列表后，Bob可以通过将每个数字与其前一个数字相加来重建原始列表。这意味着我们节省了空间而没有丢失任何信息，这称为无损编码。 如果我们在固定值范围内随机选择数字，那么我们选择的数字越多，平均差异（均值）的大小就越小。这意味着我们需要传输的数据量并不像列表长度增加得那么快（直到某个点）。 更有用的是，列表中随机选择的数字的长度自然倾向于较小的长度。考虑从1到6选择两个随机数；这相当于掷两个骰子。两个骰子有36个不同的组合：

1 1 1 2 1 3 1 4 1 5 1 6

2 1 2 2 2 3 2 4 2 5 2 6

3 1 3 2 3 3 3 4 3 5 3 6

4 1 4 2 4 3 4 4 4 5 4 6

5 1 5 2 5 3 5 4 5 5 5 6

6 1 6 2 6 3 6 4 6 5 6 6

让我们找到两个数字中较大的数字与较小的数字之间的差异：

0 1 2 3 4 5

1 0 1 2 3 4

2 1 0 1 2 3

3 2 1 0 1 2

4 3 2 1 0 1

5 4 3 2 1 0

如果我们统计每个差异发生的频率，我们会发现较小的差异比较大的差异更有可能发生：

差异	频率
0	6
1	10
2	8
3	6
4	4
5	2

如果我们知道可能需要存储大数（因为即使它们很少，大的差异也可能发生），但通常需要存储小数，我们可以使用一种系统对每个数字进行编码，对小数使用较少的空间，并为大数提供额外的空间。从平均意义上讲，这种系统的性能会比对每个数字使用相同数量的空间好。

Golomb 编码提供了这种功能。Rice 编码是 Golomb 编码的一个子集，在某些情况下更方便使用，包括 Bitcoin 块过滤器的应用。

要包含在区块过滤器中的数据

\ 我们的主要目标是允许钱包确定一个区块是否包含影响该钱包的交易。为了使钱包更有效，它需要了解两种类型的信息：

1. 当它收到钱时

具体来说，当一个交易输出包含钱包控制的脚本（例如通过控制授权的私钥）时。

1. 当它花费钱时

具体来说，当一个交易输入引用钱包控制的先前交易输出时。

在设计紧凑型区块过滤器时的一个次要目标是允许接收过滤器的钱包验证它是否从对等方接收到准确的过滤器。例如，如果钱包下载了生成过滤器的区块，那么它可以生成自己的过滤器。然后，它可以将自己的过滤器与对等方的过滤器进行比较，并验证它们是否相同，从而证明对等方生成了准确的过滤器。

为了实现主要目标和次要目标，过滤器需要引用两种类型的信息：

- 每个区块中每个交易输出的脚本。
- 每个区块中每个交易输入的引用点。

紧凑型区块过滤器的早期设计包含了这两种信息，但后来意识到，如果我们牺牲次要目标，就可以更有效地实现主要目标。在新设计中，区块过滤器仍然引用两种类型的信息，但它们会更加相关：

- 与之前相同，每个区块中每个交易输出的脚本。
- 在更改中，它还将引用每个区块中每个交易输入引用的输出脚本。换句话说，正在花费的输出脚本。

这有几个优点。首先，这意味着钱包不需要跟踪引用点；它们可以只扫描它们期望收到资金的输出脚本。其次，任何时候一个区块中的后续交易花费了同一个区块中较早交易的输出，它们都将引用相同的输出脚本。在紧凑型区块过滤器中，对同一个输出脚本的多次引用是多余的，因此可以删除多余的副本，从而缩小过滤器的大小。

当全节点验证一个区块时，它们需要访问该区块中当前交易输出和被引用的先前区块中的交易输出的输出脚本，以便能够在这个简化模型中构建紧凑型区块过滤器。但是，一个区块本身不包含之前区块中的交易输出的输出脚本，因此没有方便的方式让客户端验证区块过滤器是否正确构建。然而，有一种替代方法可以帮助客户端检测对等方是否在说谎：从多个对等方获取相同的过滤器

从多个对等方下载区块过滤器

一个对等方可以向钱包提供一个不准确的过滤器。有两种方式可以创建不准确的过滤器。对等方可以创建一个引用实际上不存在于相关区块中的交易的过滤器（假阳性）。或者，对等方可以创建一个不引用实际上出现在相关区块中的交易的过滤器（假阴性）。

对不准确过滤器的第一种防护措施是让客户端从多个对等方获取过滤器。BIP157协议允许客户端仅下载一个短短的32字节的对过滤器的承诺，以确定每个对等方是否正在广告与客户端的所有其他对等方相同的过滤器。这样，客户端只需消耗少量带宽来查询许多不同对等方的过滤器，如果所有这些对等方都同意的话。

如果两个或更多不同的对等方对同一区块有不同的过滤器，那么客户端可以下载所有这些过滤器。然后，客户端还可以下载相关的区块。如果区块包含与钱包相关的任何交易，但不包含在其中的任何一个过滤器中，那么钱包就可以确定创建该过滤器的对等方是不准确的——Golomb-Rice编码集始终会包含一个潜在匹配。

或者，如果区块不包含过滤器中可能与钱包匹配的交易，这并不能证明该过滤器是不准确的。为了最小化GCS的大小，我们允许一定数量的假阳性。钱包可以继续从对等方下载附加的过滤器，无论是随机下载还是在它们指示匹配时，然后跟踪客户端的假阳性率。如果它与过滤器设计时所使用的假阳性率明显不同，那么钱包可以停止使用该对等方。在大多数情况下，不准确的过滤器的唯一后果就是钱包使用的带宽比预期更多。

使用有损编码来减少带宽

\ 我们希望在一个区块中传输的交易数据是输出脚本。输出脚本的长度各不相同，并且遵循不同的模式，这意味着它们之间的差异并不会像我们期望的那样均匀分布。然而，在本书的许多地方已经提到，我们可以使用哈希函数来创建对某些数据的承诺，并且还可以生成类似于随机选择的数字。在本书的其他地方，我们已经使用了一个具有密码学安全性的哈希函数，它提供了关于其承诺强度和其输出与随机性之间的不可区分性的保证。然而，还有一些更快速和更可配置的非密码学哈希函数，例如我们将用于紧凑块过滤器的SipHash函数。

所使用算法的详细信息在BIP158中有描述，但其要点是使用SipHash和一些算术运算将每个输出脚本简化为一个64位的承诺。您可以将其视为将一组大数字截断为较短数字的过程，这个过程会丢失数据（因此称为有损编码）。通过丢失一些信息，我们就不需要在后续存储那么多信息，从而节省空间。在这种情况下，我们从典型的长度为160位或更长的输出脚本减少到仅仅64位。

使用紧凑块过滤器

\ 每个区块中对输出脚本的每个承诺的64位值被排序，重复的条目被移除，并且通过查找每个条目之间的差值（增量）来构建GCS。然后，对等方将该紧凑块过滤器分发给它们的客户（如钱包）。

客户端使用这些增量来重建原始的承诺。客户端，比如钱包，还会对其监视的所有输出脚本进行相同方式的承诺生成，就像BIP158那样。它检查它生成的任何承诺是否与过滤器中的承诺匹配。

回想一下，紧凑块过滤器失真类似于截断数字的情况。想象一个客户端正在寻找一个包含数字123456的区块，而一个准确（但失真）的紧凑块过滤器包含数字1234。当一个客户端看到1234时，它将下载相关的区块。准确的过滤器包含1234的情况下，客户端学习到包含123456的区块的概率是100%，称为真正的正匹配。然而，该区块可能还包含123400、123401或近百个其他不符合客户端期望的条目（在这个例子中），称为假正匹配。

一个100%的真正匹配率是很好的。这意味着一个钱包可以依赖紧凑块过滤器来找到影响该钱包的每个交易。非零的假正匹配率意味着钱包将下载一些对其无兴趣的区块。这的主要后果是客户端会使用额外的带宽，这不是一个很大的问题。BIP158紧凑块过滤器的实际假正匹配率非常低，因此这不是一个主要问题。假正匹配率也可以帮助提高客户端的隐私，就像布隆过滤器一样，尽管任何想要最佳隐私的人仍然应该使用自己的完整节点。

长期来看，一些开发者主张让区块对该区块的过滤器进行承诺，最可能的方案是每个Coinbase交易对该区块的过滤器进行承诺。完整节点将为每个区块计算过滤器，并仅在包含准确承诺的情况下接受该区块。这将允许轻量级客户端下载一个80字节的区块头、一个（通常很小的）Coinbase交易和该区块的过滤器，以获得强有力的证据证明该过滤器是准确的。

轻量级客户端与隐私

\ 轻量级客户端的隐私性不如完整节点。完整节点会下载所有的交易，因此不会透露有关是否在其钱包中使用某些地址的信息。轻量级客户端只会下载与其钱包相关的交易。

布隆过滤器和紧凑块过滤器是减少隐私损失的方法。没有它们，轻量级客户端将不得不明确列出其感兴趣的地址，从而造成严重的隐私泄露。然而，即使使用了过滤器，一个监视轻量级客户端流量的对手或直接连接到其作为P2P网络中的节点，也可能随着时间的推移收集到足够的信息来了解轻量级客户端的钱包中的地址。

加密和认证连接

\ 大多数新用户认为比特币节点的网络通信是加密的。事实上，比特币的原始实现完全以明文通信，截至撰写本文时，现代的比特币核心实现也是如此。

为了增加比特币P2P网络的隐私和安全性，有一种解决方案提供了通信加密：Tor传输。

Tor是一项软件项目和网络，通过随机的网络路径提供数据的加密和封装，从而实现匿名性、不可追踪性和隐私性。

比特币核心提供了几种配置选项，允许您在Tor网络上传输比特币节点的流量。此外，比特币核心还可以提供Tor隐藏服务，允许其他Tor节点通过Tor直接连接到您的节点。

从比特币核心版本0.12开始，如果节点能够连接到本地Tor服务，它将自动提供一个隐藏的Tor服务。如果您已经安装了Tor，并且比特币核心进程以具有足够权限访问Tor身份验证cookie的用户身份运行，它应该会自动工作。使用调试标志来启用比特币核心的Tor服务调试，像这样：

```
$ bitcoind --daemon --debug=tor
```

您应该在日志中看到“tor: ADD_ONION successful”，表示比特币核心已将一个隐藏服务添加到Tor网络中。

您可以在比特币核心文档（docs/tor.md）和各种在线教程中找到有关将比特币核心作为Tor隐藏服务运行的更多说明。

\

内存池和孤立池

\ 几乎每个比特币网络上的节点都维护着一个临时的未确认交易列表，称为内存池（mempool）。节点使用这个池来跟踪已知于网络但尚未包含在区块链中的交易，称为未确认交易。

随着未确认交易的接收和验证，它们被添加到内存池，并被中继到相邻节点以在网络上传播。

一些节点实现还维护一个单独的孤立交易池。如果交易的输入引用了尚未知晓的交易，比如缺少的父交易，那么孤立交易将临时存储在孤立池中，直到父交易到来。

当交易被添加到内存池时，会检查孤立池中是否有任何引用该交易输出（其子交易）的孤立交易。任何匹配的孤立交易都将被验证。如果有效，它们将从孤立池中移除，并添加到内存池中，完成从父交易开始的链条。随着新添加的交易不再是孤立交易，这个过程会递归地重复，查找任何更进一步的后代，直到不再找到后代。通过这个过程，父交易的到来触发了整个链条的相互依赖交易的级联重构，将孤立交易与其父交易一直重组到链条的底部。

一些比特币实现还维护一个未花费交易输出（UTXO）数据库，这是区块链上所有未花费输出的集合。这代表了与内存池不同的一组数据。与内存池和孤立池不同，UTXO数据库包含了数百万条未花费交易输出的条目，从创世区块一直到现在所有尚未花费的输出。UTXO数据库以表格形式存储在持久性存储中。

虽然内存池和孤立池代表了单个节点的局部视角，可能会因节点的启动或重新启动时间的不同而变化，但UTXO数据库代表了网络的累积共识，因此在节点之间通常不会变化。

现在我们对节点和客户端用于在比特币网络上传输数据的许多数据类型和结构有了一定的了解，现在是时候来看看负责保持网络安全和运行的软件了。

综合介绍

\ 区块链是每一笔已确认比特币交易的历史记录。它使得每个完整节点能够独立确定哪些密钥和脚本控制着哪些比特币。在本章中，我们将探讨区块链的结构，以及它如何利用密码学承诺和其他巧妙的技巧使得每个部分都易于完整节点（有时甚至是轻量级客户端）进行验证。

区块链数据结构是一个有序的、反向链接的交易块列表。区块链可以存储为一个平面文件或简单的数据库。区块通过“反向链接”方式相互关联，每个区块都指向链中的前一个区块。区块链通常被视为一个垂直堆栈，每个区块都叠加在其他区块之上，而第一个区块则作为堆栈的基础。区块叠加在一起的可视化结果导致了使用诸如“高度”来指代距离第一个区块的距离，以及“顶部”或“尖端”来指代最近添加的区块等术语。

区块链中的每个区块都由哈希标识，使用 SHA256 密码哈希算法对区块的标头生成。每个区块还通过区块标头中的“上一个区块哈希”字段提交给前一个区块，即父区块。将每个区块与其父区块链接起来的哈希序列创建了一条链，一直回溯到创世区块。

尽管一个区块只有一个父区块，但它可以有多个子区块。每个子区块都提交给同一个父区块。在区块链“分叉”时会出现多个子区块，这是由于不同的矿工几乎同时发现了不同的区块。最终，只有一个子区块会成为被所有完整节点接受的区块链的一部分，从而解决了“分叉”。

“上一个区块哈希”字段位于区块标头内部，从而影响当前区块的哈希。对父区块的任何更改都需要子区块的哈希发生变化，这又需要更改孙区块的指针，进而改变孙区块，依此类推。这个序列确保了一旦一个区块有了许多后代，它就不能在不强制重新计算所有后续区块的情况下进行更改。由于这样的重新计算需要大量的计算（因此需要大量能源消耗），因此存在着一个长链条的区块链深层历史不容易更改的特性，这是比特币安全的一个关键特征。

一个理解区块链的方式类似于地质形成中的地层或冰川核心样品。表层可能会随着季节变化，甚至在沉淀之前就被吹走。但是一旦你深入到几英寸深度，地层就会变得越来越稳定。当你看到数百英尺深时，你看到的是一个保持不变数百万年的过去的快照。在区块链中，最近的几个区块可能会根据分叉而进行修改。前六个区块就像几英寸的表土一样。但是一旦你深入到区块链的更深处，超过六个区块，区块就越来越不可能改变。在回溯到 100 个区块之后，稳定性已经非常高，以至于 coinbase 交易（包含创建新区块的比特币奖励的交易）可以被花费。尽管协议始终允许通过更长的链来撤消链，并且任何区块被撤消的可能性始终存在，但是随着时间的推移，这种事件的概率会减小，直到变得微不足道。

区块的结构

一个区块是一个容器数据结构，用于聚合交易以便包含在区块链中。区块由一个包含元数据的头部组成，后面跟着一个包含大部分数据的交易列表。区块头部通常为 80 字节，而一个区块中所有交易的总大小可以达到约 4,000,000 字节。因此，一个完整的区块，包含所有交易，几乎可以比区块头部大约 50,000 倍。表 11-1 描述了比特币核心如何存储一个区块的结构。

表 11-1. 区块的结构

大小	字段	描述
4 字节	区块大小(Block Size)	该字段后面跟着区块的大小，以字节为单位
80字节	区块头(Block Header)	几个字段形成了区块头部
1-3字节(compactSize)	交易计数器(Transaction Counter)	接下来有多少个交易
可变字节	交易(Transactions)	记录在此区块中的交易

区块头

\ 区块头部由如下表11-2所示的区块元数据组成。

Table 11-2. 区块头部的结构

大小	字段	描述
4字节	版本(Version)	最初是一个版本字段；随着时间的推移，它的用途已经发展
32字节	之前的块哈希(Previous Block Hash)	链中前一个（父）块的哈希
32字节	默克尔根(Merkle Root)	该块交易的默克尔树根哈希
4字节	时间戳(Timestamp)	该区块的大致创建时间（Unix纪元时间）
4字节	目标(Target)	该区块的工作量证明目标的紧凑编码
4字节	只用一次的随机数 (Nonce)	工作证明算法使用的任意数据

nonce（随机数）、目标值和时间戳在挖矿过程中使用，并将在第12章中进行更详细的讨论。

区块标识符：区块头哈希和区块高度

一个区块的主要标识符是其加密哈希，这是通过SHA256算法对区块头进行两次哈希计算得到的。生成的32字节哈希称为区块哈希，但更准确地说是区块头哈希，因为只有区块头用于计算它。例如，比特币区块链上的第一个区块的区块哈希是000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f。区块哈希唯一而明确地标识一个区块，并且可以由任何节点独立地通过对区块头进行哈希计算来派生出来。

请注意，区块哈希实际上并未包含在区块的数据结构中。相反，每个节点在从网络接收到区块时计算区块哈希。为了便于索引和更快地从磁盘检索区块，区块哈希可能会存储在单独的数据库表中作为区块的元数据的一部分。

第二种识别区块的方式是通过其在区块链中的位置，称为区块高度。创世区块位于区块高度0（零），之前被以下区块哈希引用：000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f。因此，一个区块可以通过引用区块哈希或引用区块高度来进行识别。每个添加到该创世区块“之上”的后续区块都在区块链中处于一个“更高”的位置，就像箱子一样一层叠一层堆叠。在撰写本书时，即2023年中期，区块高度已经达到了800,000，这意味着在2009年1月创建的第一个区块之上堆叠了800,000个区块。

与区块哈希不同，区块高度不是唯一的标识符。虽然单个区块始终具有特定且不变的区块高度，但反之并非如此——区块高度并不总是唯一地标识单个区块。两个或多个区块可能具有相同的区块高度，竞争同一位置在区块链中。这种情况在第282页的“组装和选择区块链”部分中有详细讨论。在早期的区块中，区块高度也不是区块的数据结构的一部分；它未存储在区块内。每个节点在从比特币网络接收到区块时会动态地确定区块在区块链中的位置（高度）。后来的协议更改（BIP34）开始将区块高度包含在coinbase交易中，尽管其目的是确保每个区块具有不同的coinbase交易。节点仍然需要动态地确定区块的高度以验证coinbase字段。区块高度也可能作为元数据存储在索引数据库表中，以便更快地检索。

> 一个区块的区块哈希始终唯一地标识一个区块。一个区块也总是具有特定的区块高度。然而，并不总是情况下特定的区块高度标识单个区块。相反，两个或多个区块可能竞争区块链中的一个位置。

创世区块

\ 区块链中的第一个区块称为创世区块，创建于2009年。它是区块链中所有区块的共同祖先，这意味着如果你从任何一个区块开始，沿着时间的倒序链向后跟踪，最终会到达创世区块。

每个节点始终以至少一个区块的区块链开始，因为创世区块被静态编码在比特币核心中，因此无法更改。每个节点始终“知道”创世区块的哈希和结构，以及创建的固定时间，甚至包括其中的单个交易。因此，每个节点都有一个区块链的起点，一个安全的“根”，用于构建一个可信的区块链。

在Bitcoin Core客户端的chainparams.cpp中，可以看到静态编码的创世区块。

以下标识哈希属于创世区块：

000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

您可以在几乎任何区块浏览器网站上搜索该区块哈希，比如blockstream.info，您会找到一个页面描述该区块的内容，其中包含该哈希的URL链接。

https://blockstream.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

或者，您可以在命令行上使用Bitcoin Core获取该区块：

```
$ bitcoin-cli getblock \
 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
{
"hash": "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
"confirmations": 790496,
"height": 0,
"version": 1,
"versionHex": "00000001",
"merkleroot": "4a5e1e4baab89f3a32518a88c3[...]76673e2cc77ab2127b7afdeda33b",
"time": 1231006505,
"mediantime": 1231006505,
"nonce": 2083236893,
"bits": "1d00ffff",
"difficulty": 1,
"chainwork": "[...]0000000000000000000000000000000000000000000000100010001",
"nTx": 1,
"nextblockhash": "00000000839a8e6886ab5951d7[...]fc90947ee320161bbf18eb6048",
"strippedsize": 285,
"size": 285,
"weight": 1140,
"tx": [
"4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
] }
```

创世区块中包含一条信息。Coinbase 交易输入包含文本：“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks.” 这条信息旨在通过引用英国报纸《泰晤士报》的头条，提供该区块可能被创建的最早日期的证明。它还作为一种戏谑的提醒，强调了独立货币系统的重要性，因为比特币的推出恰逢前所未有的全球金融危机。这条信息是比特币的创始人中本聪嵌入到第一个区块中的。

区块链中的区块链接

图比特币全节点会验证从创世区块之后的每个区块。它们对区块链的本地视图会随着新的区块被发现并用于扩展链条而不断更新。当节点从网络中接收到新的区块时，它会验证这些区块，然后将它们链接到现有区块链的视图中。为了建立链接，节点会检查接收到的区块头，并查找“上一个区块哈希”。

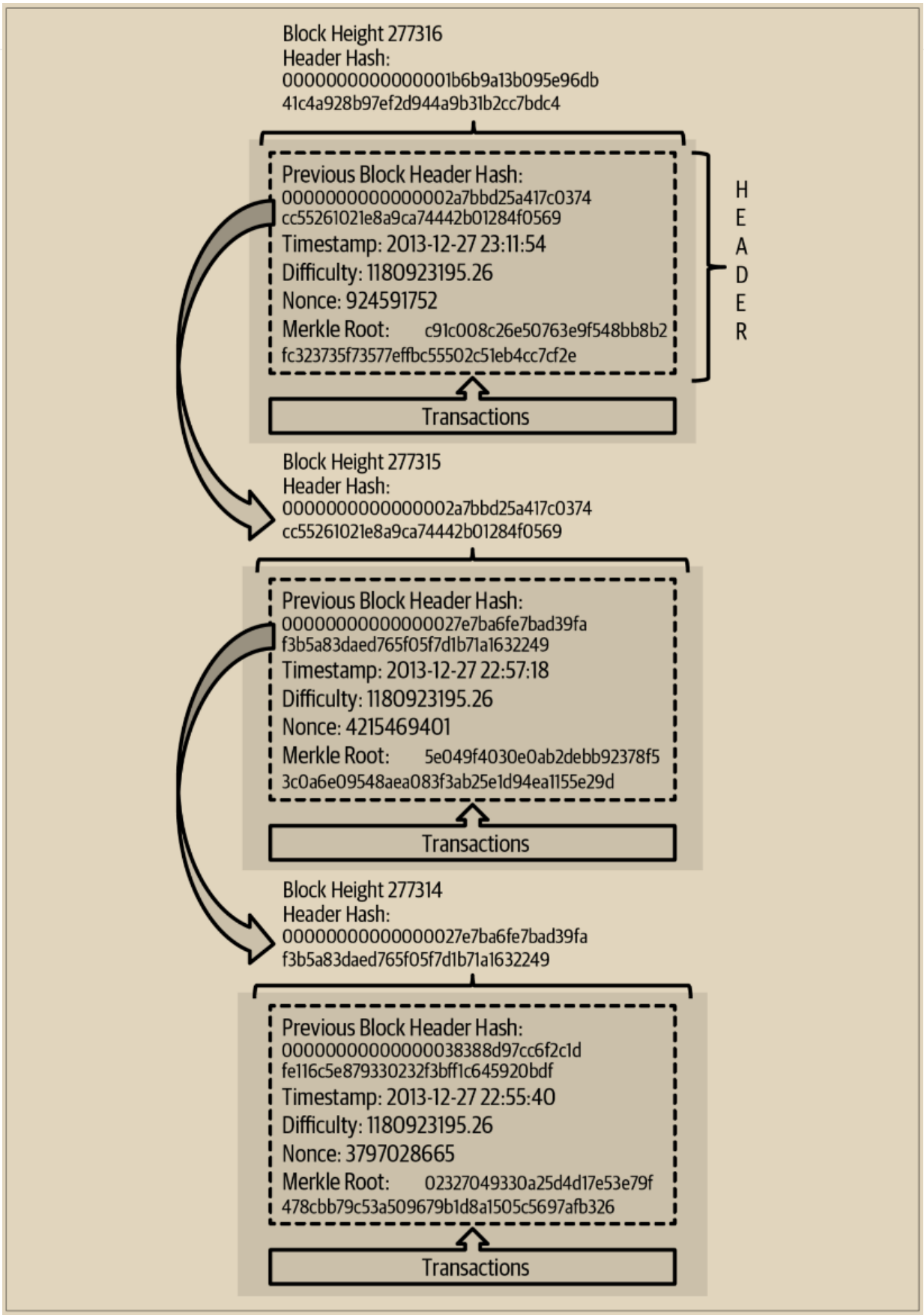
例如，假设一个节点在本地拷贝的区块链中有277,314个区块。节点所知道的最后一个区块是第277,314块，其区块头哈希为：

00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249

然后比特币节点从网络中接收到一个新的区块，它解析如下：

```
{
"size" : 43560,
"version" : 2,
"previousblockhash" :
"00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
"merkleroot" :
"5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
"time" : 1388185038,
"difficulty" : 1180923195.25802612,
"nonce" : 4215469401,
"tx" : [
"257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",
"[... many more transactions omitted ...]",
"05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
] }
```

查看这个新区块，节点发现了 previousblockhash 字段，其中包含其父区块的哈希值。这个哈希值是节点已知的，即链上高度为 277,314 的最后一个区块的哈希值。因此，这个新区块是链上最后一个区块的子区块，并扩展了现有的区块链。节点将这个新区块添加到链的末尾，使得区块链变得更长，高度为 277,315。图 11-1 显示了由 previousblockhash 字段中的引用链接起来的三个区块的链条。



11-1. 区块通过每个引用前一个区块头哈希值来链接成链

默克尔树

比特币区块链中的每个区块都包含了对该区块中所有交易的摘要，使用了默克尔树。

默克尔树，也称为二叉哈希树，是一种用于高效总结和验证大型数据集完整性的数据结构。默克尔树是包含了加密哈希的二叉树。计算机科学中的“树”一词用于描述分支数据结构，但这些树通常在图表中是倒置的，顶部是“根”，底部是“叶子”，正如你将在后面的示例中看到的那样。

比特币中使用默克尔树来总结一个区块中的所有交易，产生对整个交易集合的整体承诺，并允许非常有效地验证一个交易是否包含在区块中。默克尔树是通过递归地对元素成对进行哈希处理构建的，直到只剩下一个哈希，称为根，或默克尔根。比特币中用于默克尔树的加密哈希算法是SHA256应用两次，也称为双SHA256。

当N个数据元素被哈希并在默克尔树中进行总结时，你可以通过大约log2(N)次计算来检查其中是否包含任何一个数据元素，使得这成为一种非常高效的数据结构。

默克尔树是自底向上构建的。在下面的例子中，我们从四个交易A、B、C和D开始，它们形成了默克尔树的叶子节点，如图11-2所示。交易并不存储在默克尔树中；相反，它们的数据被哈希处理，产生的哈希结果存储在每个叶子节点中，分别为HA、HB、HC和HD：

\ HA = SHA256(SHA256(Transaction A))

然后，将相邻的叶子节点成对合并到一个父节点中，方法是将这两个哈希连接在一起，然后将它们一起进行哈希处理。例如，要构建父节点HAB，将两个子节点的32字节哈希连接起来，形成一个64字节的字符串。然后对该字符串进行双重哈希处理，以生成父节点的哈希值：

HAB = SHA256(SHA256(HA || HB))

这个过程持续进行，直到顶部只剩下一个节点，即被称为 Merkle 根的节点。这个 32 字节的哈希值存储在区块头中，总结了所有四笔交易中的所有数据。图 11-2 展示了通过节点的成对哈希计算根节点的过程。

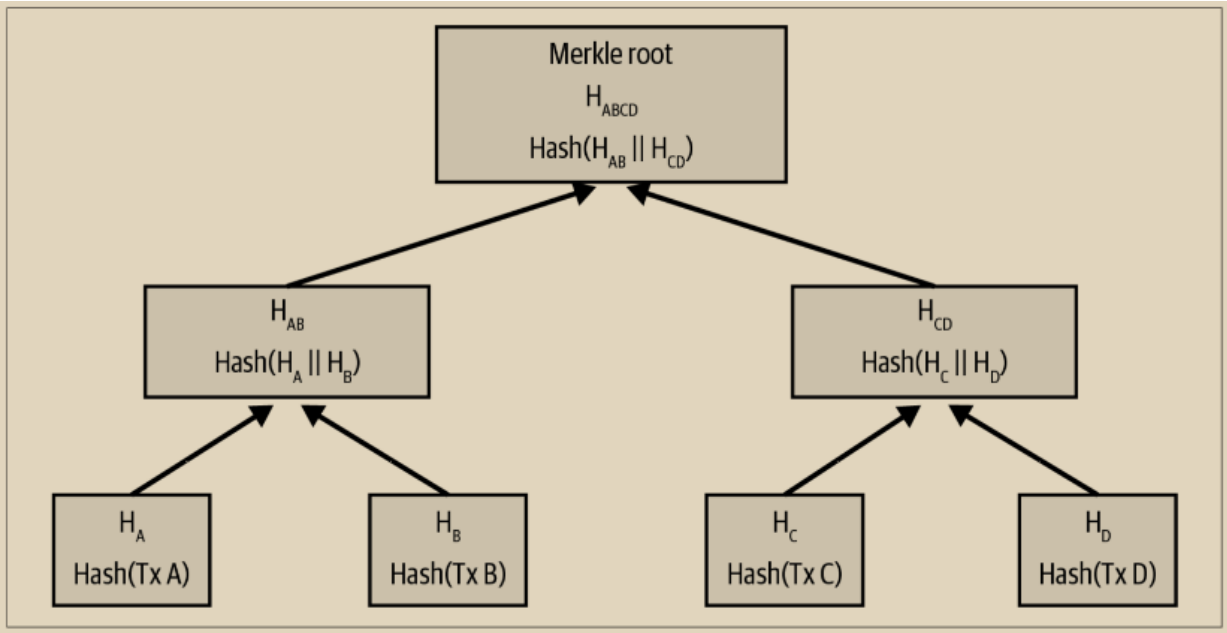


图 11-2. 计算 Merkle 树中的节点

因为 Merkle 树是一个二叉树，所以需要有偶数个叶子节点。如果要总结的交易数量为奇数，则最后一个交易的哈希将被复制，以创建一个偶数个叶子节点，也称为平衡树。如图 11-3 所示，交易 C 被复制。同样地，如果在任何层级中要处理的哈希数量为奇数，则最后一个哈希将被复制。

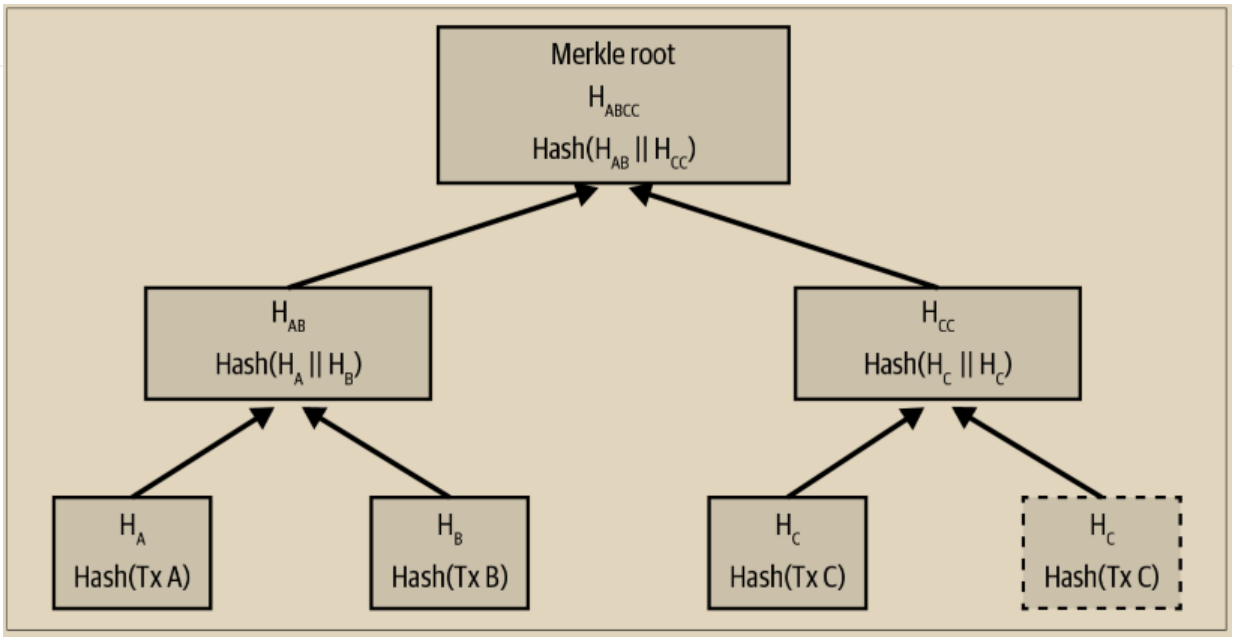


图 11-3. 复制一个数据元素可以实现有偶数个数据元素

> **比特币的默克尔树设计中存在一个缺陷**
>
> 比特币核心代码中的一段扩展注释，稍作修改后再次呈现，描述了比特币的默克尔树中一个重要问题：
>
> 警告！如果你正在阅读这篇文章，因为你正在学习加密货币和/或设计一个将使用默克尔树的新系统，请记住以下默克尔树算法存在一个严重的缺陷，与重复的交易ID相关，导致一个漏洞（CVE-2012-2459）。
>
> 原因是，如果在给定级别的列表中哈希的数量是奇数，那么在计算下一个级别之前，最后一个会被复制一次（这在默克尔树中是不寻常的）。 这会导致某些交易序列导致相同的默克尔根。例如，图11-4中的两棵树：

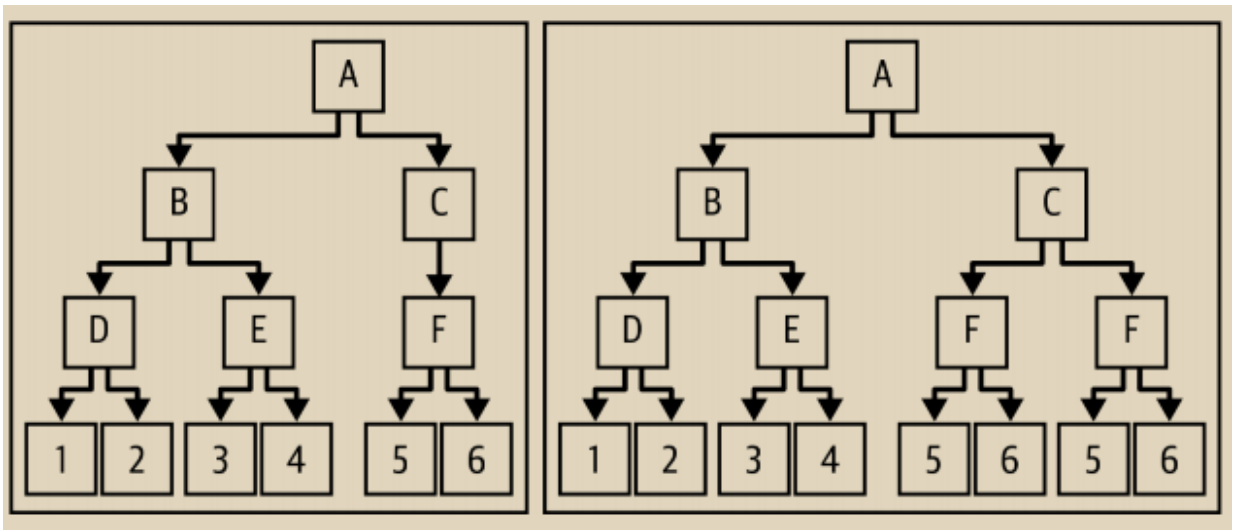


图 11-4. 两棵类比特币的默克尔树，根相同但叶节点数量不同

> 交易列表[1,2,3,4,5,6]和[1,2,3,4,5,6,5,6]（其中5和6重复出现）导致相同的根哈希A（因为(F)和(F,F)的哈希值均为C）。 这个漏洞的原因在于可以发送一个具有相同默克尔根和与原始交易列表不重复的块，导致验证失败。如果接收节点继续将该块标记为永久无效，那么它将无法接受进一步的未修改（因此可能有效）的相同块的版本。我们通过检测在列表末尾将两个相同的哈希值进行哈希处理的情况来防范这种情况，并将其视为块具有无效的默克尔根。假设没有双SHA256碰撞，这将检测到所有已知的在不影响默克尔根的情况下更改交易的方法。
>
> ——Bitcoin Core *src/consensus/merkle.cpp*

构建树的相同方法可以推广到构建任何大小的树。在比特币中，一个区块中常常有数千个交易，它们以完全相同的方式进行汇总，产生的只是 32 字节的数据作为单一的 Merkle 根。在图 11-5 中，您将看到一个由 16 个交易构建的树。请注意，尽管在图中根节点看起来比叶子节点大，但它们的大小是完全相同的，都是 32 字节。无论在区块中有一个交易还是一万个交易，Merkle 根始终将它们汇总为 32 字节。

为了证明特定交易包含在一个区块中，一个节点只需要产生大约 log2(N) 个 32 字节的哈希，构成了一个连接特定交易与树根的认证路径或 Merkle 路径。随着交易数量的增加，这一点尤其重要，因为交易数量的底数对数增长得非常缓慢。这使得比特币节点能够高效地产生由 10 到 12 个哈希（320–384 字节）组成的路径，这可以提供对多兆字节区块中一千多个交易的单个交易的证明。

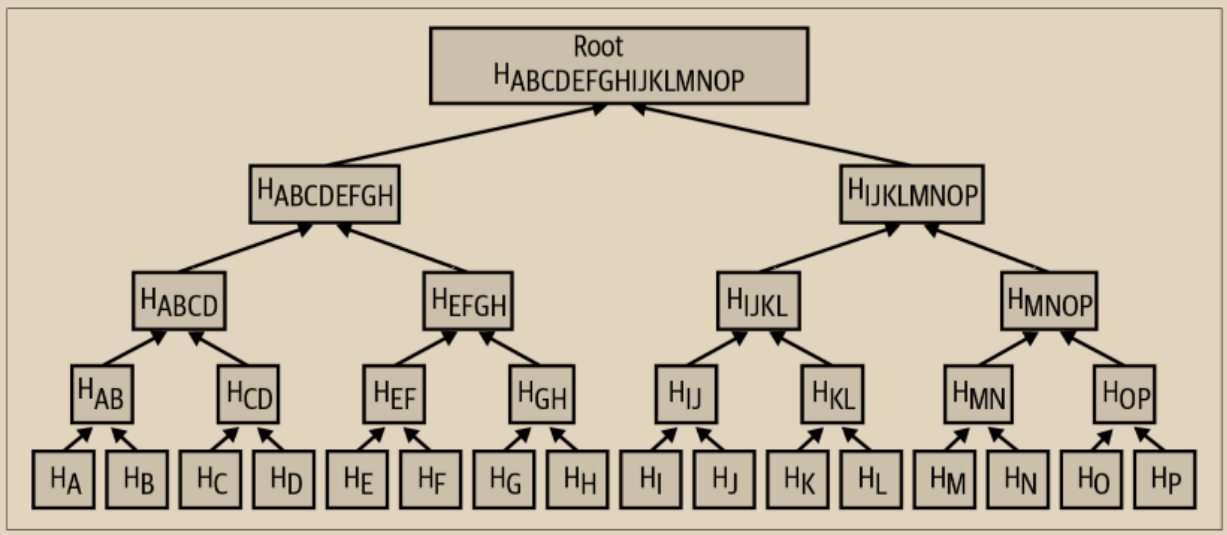


图 11-5. 总结许多数据元素的 Merkle 树

在图 11-6 中，节点可以通过生成一个只有四个 32 字节哈希值长的 Merkle 路径（总共 128 字节）来证明交易 K 包含在区块中。该路径由四个哈希值组成（带有阴影背景显示）HL，HIJ，HMNOP 和 HABCDEFGH。通过提供这四个哈希作为认证路径，任何节点都可以证明 HK（在图的底部具有黑色背景）包含在 Merkle 根中，方法是计算四个额外的逐对哈希值 HKL，HIJKL，HIJKLMNOP 和 Merkle 树根（在图中用虚线轮廓显示）。

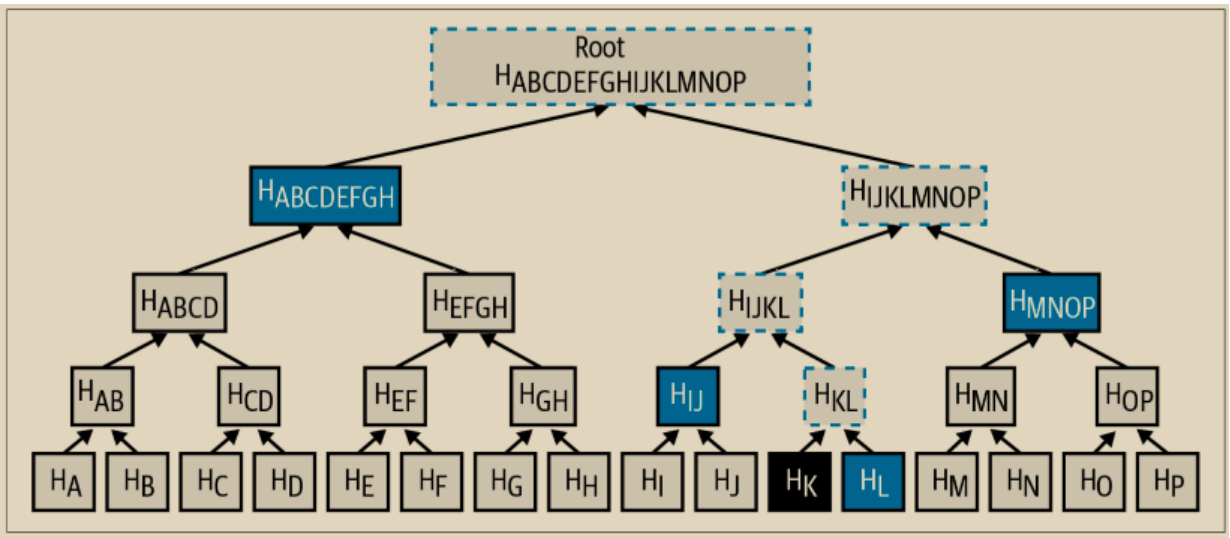


图 11-6. 用于证明数据元素包含在内的 Merkle 路径

当规模增加时，Merkle 树的效率变得明显。最大可能的区块可以容纳近 16,000 笔交易，占用 4,000,000 字节，但证明其中的任何一笔交易是该区块的一部分只需要交易的副本、80 字节区块头的副本和 448 字节的 Merkle 证明。这使得最大可能的证明几乎比最大可能的比特币区块小了 10,000 倍。

Merkle 树与轻量级客户端

默克尔树在轻量级客户端中被广泛使用。轻量级客户端不具备所有交易数据，也不下载完整的区块，仅下载区块头。为了验证某笔交易是否包含在一个区块中，而不必下载该区块中的所有交易数据，它们使用了默克尔路径。

举个例子，考虑一个对其钱包中包含的地址的收款感兴趣的轻量级客户端。轻量级客户端将在与对等节点的连接上建立一个布隆过滤器（参见“布隆过滤器”第231页），以限制接收的交易仅包含感兴趣的地址。当对等节点看到与布隆过滤器匹配的交易时，它将使用 merkleblock 消息发送该区块。merkleblock 消息包含区块头以及将感兴趣的交易链接到区块中的默克尔根的默克尔路径。 轻量级客户端可以使用这个默克尔路径将交易连接到区块头，并验证该交易是否包含在区块中。轻量级客户端还使用区块头将区块链接到区块链的其余部分。这两个链接的组合，交易和区块之间的链接以及区块和区块链之间的链接，证明了该交易已记录在区块链中。总的来说，轻量级客户端仅需要接收区块头和默克尔路径的不到1千字节的数据量，这比完整区块的数据量要小上千倍（目前约为2 MB）。

比特币的测试区块链

您可能会感到惊讶，原来比特币使用的不止是一个区块链。由中本聪于2009年1月3日创建的“主”比特币区块链，也就是我们在本章中研究的具有创世区块的那个，被称为主网（mainnet）。除了主网外，还有其他用于测试目的的比特币区块链：目前有测试网（testnet）、签名测试网（signet）和回归测试网（regtest）。我们逐一来看一下每一个。

测试网：比特币的测试场地

测试网是用于测试目的的测试区块链、网络和货币的名称。测试网是一个完整的实时P2P网络，具有钱包、测试比特币（测试网币）、挖矿以及主网的所有其他功能。最重要的区别是测试网币是没有价值的。

任何打算在比特币主网上投入生产使用的软件开发都可以先在测试网上使用测试币进行测试。这既可以保护开发人员免受由于错误而造成的货币损失，也可以保护网络免受由于错误而导致的意外行为。

当前的测试网被称为testnet3，这是测试网的第三个版本，于2011年2月重新启动，以重置先前测试网的难度。Testnet3是一个庞大的区块链，在2023年超过30 GB。它需要一段时间才能完全同步，并消耗您计算机上的资源。虽然不像主网那样，但也不完全是“轻量级”的

> 本书中描述的测试网和其他测试区块链不使用与主网地址相同的地址前缀，以防止某人意外地向测试地址发送真正的比特币。主网地址以1、3或bc1开头。本书提到的测试网络的地址以m、n或tb1开头。其他测试网络或在测试网络上开发的新协议可能会使用其他地址前缀或进行修改。

在tesetnet上使用Bitcoin Core

像许多其他比特币程序一样，Bitcoin Core 完全支持在 testnet 上的操作，作为替代主网。Bitcoin Core 的所有功能都可以在 testnet 上使用，包括钱包、挖掘 testnet 币和同步完整的 testnet 节点。 要在 testnet 上启动 Bitcoin Core 而不是主网，您可以使用 testnet 开关：

```
$ bitcoind -testnet
```

在日志中，您应该看到 bitcoind 正在默认的 bitcoind 目录的 testnet3 子目录中构建一个新的区块链：

```
bitcoind: Using data directory /home/username/.bitcoin/testnet3
```

要连接到 bitcoind，您可以使用 bitcoin-cli 命令行工具，但您还必须将其切换到 testnet 模式：

```
$ bitcoin-cli -testnet getblockchaininfo
{
 "chain": "test",
 "blocks": 1088,
 "headers": 139999,
 "bestblockhash": "0000000063d29909d475a1c[...]368e56cce5d925097bf3a2084370128",
 "difficulty": 1,
 "mediantime": 1337966158,
 "verificationprogress": 0.001644065914099759,
 "chainwork": "[...]0000000000000000000000000000000000000000000044104410441",
 "pruned": false,
 "softforks": [
 [...]
```

\ 您也可以在其他完整节点实现上运行 testnet3，例如用 Go 编写的 btcd 和用 JavaScript 编写的 bcoin，以在其他编程语言和框架中进行实验和学习。

Testnet3 支持 mainnet 的所有功能，包括隔离见证 v0 和 v1（参见“隔离见证”第 137 页和“Taproot”第 178 页）。因此，testnet3 也可以用于测试隔离见证功能。

测试网络存在一些问题

\ testnet 不仅使用与比特币几乎完全相同的数据结构，而且它几乎使用与比特币完全相同的工作量证明安全机制。testnet 的显着差异在于，它的最低难度是比特币的一半，而且允许在最低难度下包含一个块，如果该块的时间戳比上一个块晚超过 20 分钟。

不幸的是，比特币的 PoW 安全机制是设计依赖于经济激励，而在禁止具有价值的测试区块链中不存在这种激励。在主网上，矿工被激励将用户交易包含在他们的区块中，因为这些交易支付费用。在测试网上，交易仍然包含一种称为费用的东西，但是这些费用没有任何经济价值。这意味着测试网矿工包含交易的唯一动机是因为他们希望帮助用户和开发人员测试他们的软件。

可悲的是，喜欢破坏系统的人通常会感受到更强烈的激励，至少在短期内是如此。由于PoW挖矿旨在无需许可地进行，任何人都可以进行挖矿，无论他们的意图是好是坏。这意味着恶意的挖矿者可以在测试网络上连续创建许多区块而不包含任何用户交易。当发生这些攻击时，测试网络对用户和开发者来说就变得无法使用。

Signet：权威测试网络

没有已知的方法可以在没有引入经济激励的情况下提供一个高度可用的基于无许可 PoW 的系统，因此比特币协议开发者开始考虑替代方案。主要目标是尽可能保留比特币的结构，以便软件可以在测试网络上运行，而不需要进行太多的更改，但同时也要提供一个有用的环境。次要目标是制定一个可重复使用的设计，以便新软件的开发人员可以轻松创建自己的测试网络。

比特币核心和其他软件中实现的解决方案被称为“signet”，由 BIP325 定义。signet 是一个测试网络，每个区块都必须包含证据（例如签名），证明该区块的创建是由可信任的权威机构批准的。

在比特币上进行挖矿是无许可的——任何人都可以做——而在 signet 上进行挖矿是完全许可的。只有获得许可的人才能这样做。这对于比特币的主网来说是完全不能接受的改变——没有人会使用那种软件——但对于一个测试网络来说是合理的，其中的币没有价值，唯一的目的是测试软件和系统。

BIP325 的 signet 设计旨在非常容易创建自己的 signet。如果你不同意别人如何运行他们的 signet，你可以启动自己的 signet 并连接你的软件到它上面。

默认的signet 和自定义 signet

\ 比特币核心支持一个默认的 signet，我们相信这是目前撰写时最广泛使用的 signet。它目前由该项目的两名贡献者运营。如果你启动比特币核心并使用 signet 参数而没有其他与 signet 相关的参数，那么你将使用这个 signet。

截至撰写本文时，默认的 signet 有大约 150,000 个区块，大小约为一千兆字节。它支持与比特币主网相同的所有功能，并且还用于通过比特币审判项目测试提议的升级，该项目是比特币核心的软件分支，仅设计用于在 signet 上运行。

如果你想使用不同的 signet，称为自定义 signet，你需要了解确定区块何时被授权的脚本，称为挑战脚本。这是一个标准的比特币脚本，因此它可以使用多重签名等功能，允许多人授权区块。你可能还需要连接到一个种子节点，该节点将为你提供自定义 signet 上对等节点的地址。例如：

bitcoind -signet -signetchallenge=0123...cdef -signetseednode=example.com:1234

\ 截至撰写本文时，我们通常建议将挖矿软件的公开测试放在 testnet3 上进行，将所有其他比特币软件的公开测试放在默认 signet 上进行。

要与你选择的 signet 进行交互，你可以使用 bitcoin-cli 的 -signet 参数，类似于你使用 testnet 的方式。例如：

```
$ bitcoin-cli -signet getblockchaininfo
{
 "chain": "signet",
 "blocks": 143619,
 "headers": 143619,
 "bestblockhash": "000000c46cb3505ddd296537[...]ad1c5768e2908439382447572a93",
 "difficulty": 0.003020638517858618,
 "time": 1684530244,
 "mediantime": 1684526116,
 "verificationprogress": 0.999997961940662,
 "initialblockdownload": false,
 "chainwork": "[...]0000000000000000000000000000000000000000000019ab37d2194",
 "size_on_disk": 769525915,
 "pruned": false,
 "warnings": ""
}// Some code
```

\

Regtest：本地区块链

\ Regtest，全称“回归测试”，是比特币核心功能之一，允许你创建一个用于测试目的的本地区块链。与signet和testnet3不同，它们是公共的、共享的测试区块链，regtest区块链旨在作为本地测试的封闭系统运行。你可以从头开始启动一个regtest区块链。你可以向网络添加其他节点，也可以只运行一个节点，用于测试比特币核心软件。

要启动比特币核心进入regtest模式，你可以使用regtest标志：

```
$ bitcoind -regtest
```

与testnet一样，比特币核心将在bitcoind默认目录的regtest子目录下初始化一个新的区块链：

```
bitcoind: Using data directory /home/username/.bitcoin/regtest
```

要使用命令行工具，您也需要指定regtest标志。让我们尝试使用getblockchaininfo命令来检查regtest区块链：

```
$ bitcoin-cli -regtest getblockchaininfo
{
 "chain": "regtest",
 "blocks": 0,
 "headers": 0,
 "bestblockhash": "0f9188f13cb7b2c71f2a335e3[...]b436012afca590b1a11466e2206",
 "difficulty": 4.656542373906925e-10,
 "mediantime": 1296688602,
 "verificationprogress": 1,
 "chainwork": "[...]0000000000000000000000000000000000000000000000000000000000000002",
 "pruned": false,
 [...]
```

您可以看到，目前还没有区块。让我们创建一个默认钱包，获取一个地址，然后挖一些（500个区块）以获取奖励：

```
$ bitcoin-cli -regtest createwallet ""
$ bitcoin-cli -regtest getnewaddress
bcrt1qwvfhw8pf79kw6tvpmtxyxwcfnd2t4e8v6qfv4a
$ bitcoin-cli -regtest generatetoaddress 500 \
 bcrt1qwvfhw8pf79kw6tvpmtxyxwcfnd2t4e8v6qfv4a
[
 "3153518205e4630d2800a4cb65b9d2691ac68eea99afa7fd36289cb266b9c2c0",
 "621330dd5bdabcc03582b0e49993702a8d4c41df60f729cc81d94b6e3a5b1556",
 "32d3d83538ba128be3ba7f9dbb8d1ef03e1b536f65e8701893f70dcc1fe2dbf2",
 ...,
 "32d55180d010ffebabf1c3231e1666e9eeed02c905195f2568c987c2751623c7"
]
```

挖掘这些区块只需要几秒钟时间，这确实很容易进行测试。如果您检查钱包余额，您会看到您获得了前400个区块的奖励（coinbase 奖励必须在您能够支配之前有100个区块的深度）：

```
$ bitcoin-cli -regtest getbalance
12462.50000000
```

使用测试区块链进行开发

比特币的各种区块链（regtest、signet、testnet3、mainnet）为比特币开发提供了一系列测试环境。不论你是为比特币核心开发或其他完整节点共识客户端开发；为钱包、交易所、电子商务网站等应用程序开发；甚至是开发新颖的智能合约和复杂脚本，都可以使用测试区块链。

\ 你可以利用测试区块链建立一个开发流程。在开发过程中，在本地的 regtest 上测试你的代码。一旦准备好在公共网络上尝试，切换到 signet 或 testnet，将你的代码暴露在一个更加动态、具有更多代码和应用程序多样性的环境中。最后，一旦你对代码的功能有信心，切换到 mainnet 在生产环境中部署。在进行更改、改进、修复错误等操作时，重新启动流水线，先在 regtest 上部署每个变更，然后在 signet 或 testnet 上部署，最后再部署到生产环境中。

现在我们了解了区块链包含的数据以及加密承诺如何安全地将各个部分紧密地联系在一起，接下来我们将看看提供计算安全性并确保没有一个区块可以在不使所有其他区块失效的情况下进行更改的特殊承诺：比特币的挖矿功能。

综合介绍

“挖矿”这个词有点误导性。通过唤起对贵金属开采的联想，它将我们的注意力集中在挖矿的奖励上，即每个区块中生成的新比特币。尽管挖矿是通过这种奖励来激励的，但挖矿的主要目的并不是奖励或生成新比特币。如果你将挖矿仅视为生成比特币的过程，那么你误把手段（激励）当作了过程的目标。挖矿是支撑去中心化清算所的机制，用于验证和清算交易。挖矿是使比特币特殊的发明之一，是基于P2P数字现金的去中心化共识机制的基础之一。

挖矿保障了比特币系统的安全性，并使得网络范围内的共识得以出现，而无需中央权威。新铸造的比特币和交易费用的奖励是一种激励机制，将矿工的行为与网络的安全性对齐，同时实施货币供应。

> 挖矿是比特币的共识安全性分散化的机制之一。

矿工将新交易记录在全球区块链上。每隔约10分钟就会挖出一个新的区块，其中包含自上一个区块以来发生的交易，从而将这些交易添加到区块链中。成为区块一部分并添加到区块链上的交易被视为已确认，这使得比特币的新所有者知道，他们收到的比特币已经受到了不可撤销的保障。

此外，区块链中的交易具有由它们在区块链中的位置定义的拓扑顺序。如果一个交易出现在较早的区块中，或者在同一个区块中较早出现，那么它就比另一个交易更早。在比特币协议中，仅当交易花费了出现在区块链中较早的交易的输出时（无论它们是在同一个区块中还是在较早的区块中），该交易才有效，且仅当没有任何先前的交易花费了这些相同的输出时才有效。在一条区块链中，拓扑顺序的执行确保没有两个有效的交易可以花费相同的输出，从而消除了双重支付的问题。在一些建立在比特币之上的协议中，比特币交易的拓扑顺序也被用来建立事件序列；我们将在“单次使用密封”一节中进一步讨论这个想法。

矿工因提供挖矿所带来的安全性而获得两种类型的奖励：每个新区块中创建的新比特币（称为补贴），以及包含在区块中的所有交易的交易费用。为了获得这个奖励，矿工们竞争以满足基于加密哈希算法的挑战。解决这个问题的解决方案，称为工作证明，包含在新区块中，并作为证明矿工投入了大量的计算工作量。解决工作证明算法以获得奖励和记录交易到区块链的权利的竞争，是比特币安全模型的基础。

比特币的货币供应是通过类似于中央银行通过印刷纸币发行新货币的过程创建的。矿工每增加一个区块就可以添加的新比特币的最大数量大约每四年减少一半（或者确切地说是每增加210,000个区块）。它从2009年1月的每个区块50个比特币开始，于2012年11月减半为每个区块25个比特币。在2016年7月再次减半为每个区块12.5个比特币，然后在2020年5月减半为6.25个比特币。根据这个公式，挖矿奖励将以指数方式递减，直到大约在2140年，所有的比特币都将被发行完毕。在2140年之后，将不会再发行新的比特币。

比特币矿工还从交易中获得费用。每个交易都可能包含一个交易费，形式为交易的输入和输出之间的比特币的余额。赢得比特币的矿工可以“留下找零”来处理包含在获胜区块中的交易。如今，交易费通常仅占矿工收入的一小部分，绝大部分收入来自新铸造的比特币。然而，随着奖励随时间减少和每个区块中的交易数量增加，越来越大比例的挖矿收入将来自交易费。逐渐地，挖矿奖励将被交易费主导，这将成为矿工的主要激励。在2140年之后，每个区块中的新比特币数量将降至零，挖矿将仅由交易费来激励。

在本章中，我们首先将挖矿作为货币供应机制进行讨论，然后再看看挖矿的最重要功能：支撑比特币安全的去中心化共识机制。

为了理解挖矿和共识，我们将追踪Alice的交易，看它是如何被Jing的挖矿设备接收并添加到一个区块中的。然后，我们将跟踪这个区块是如何被挖出、添加到区块链中，并通过紧急共识的过程被比特币网络接受的。\

比特币经济学与货币创造

\ 比特币在每个区块创建时以固定且递减的速率铸造。每个区块平均每10分钟产生一次，其中包含全新的比特币，从无到有地创造出来。每经过210,000个区块，或大约每四年，货币发行速度就会减少50%。在网络运行的头四年中，每个区块包含50个新的比特币。

第一次减半发生在第210,000个区块。在本书出版后的下一次减半预期将在第840,000个区块发生，这可能会在2024年4月或5月产生。新比特币的发行速度经过32次这样的减半呈指数递减，直到第6,720,000个区块（大约在2137年开采），当时它将达到最小的货币单位1个聪。最终，在大约2140年之后，几乎将发行20,999,999,997,690,000个聪，或几乎21百万比特币。此后，区块将不再包含新的比特币，矿工将仅通过交易费获得奖励。图12-1显示了随着货币发行减少，随时间推移在流通中的总比特币数量。

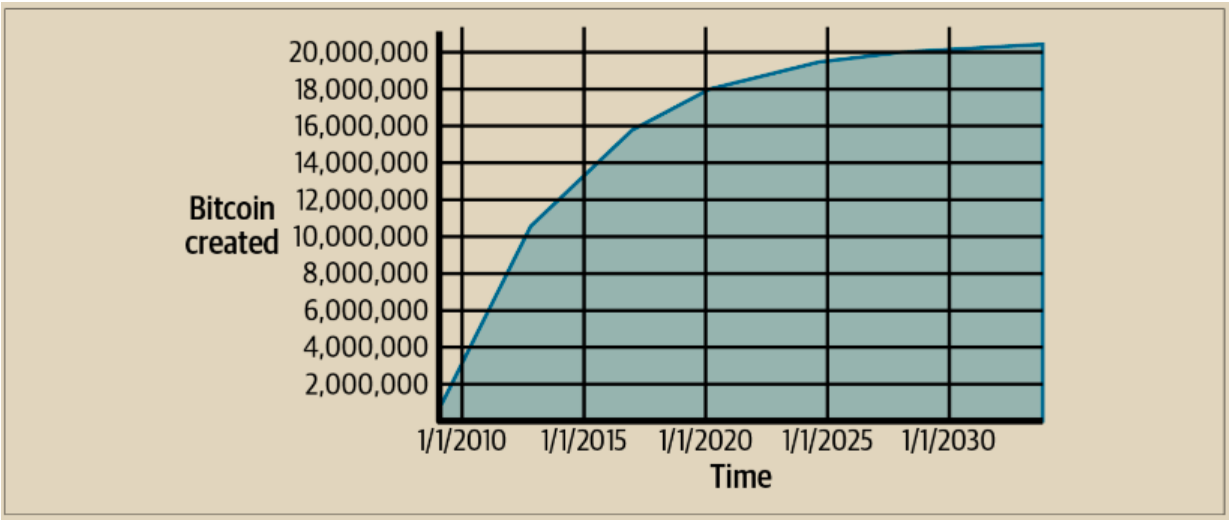


图 12-1. 随时间推移，比特币货币供应根据几何递减的发行率而变化

> 比特币的最大挖掘数量是比特币可能的挖掘奖励的上限。实际上，矿工可能会有意地挖出一个奖励不足的区块。这样的区块已经被挖出来了，未来可能还会被挖出更多，导致货币的总发行量降低。

在示例 12-1 中的代码中，我们计算将发行的比特币总量。

示例 12-1. 用于计算总比特币发行量的脚本

```
# 定义常量
initial_reward = 50  # 每个区块的初始奖励，单位为比特币
halving_interval = 210000  # 每次减半之间的区块数量
total_blocks = 6930000  # 预计直到最后一次减半的总区块数量

# 计算总的比特币发行量
total_bitcoin_issued = 0
reward = initial_reward
halvings = 0

# 循环遍历减半次数，直到达到最后一次减半
while halvings < total_blocks // halving_interval:
    total_bitcoin_issued += halving_interval * reward
    reward /= 2  # 将奖励减半
    halvings += 1

# 计算最后一次减半后剩余区块的发行量
remaining_blocks = total_blocks % halving_interval
total_bitcoin_issued += remaining_blocks * reward

# 输出总的比特币发行量
print("总比特币发行量:", total_bitcoin_issued, "比特币")
```

\ 示例 12-2 展示了运行该脚本所产生的输出，所以比特币发行总量大概为2100万个。

示例 12-2. 运行 max_money.py 脚本

```
$ python max_money.py
总比特币发行量: 20999999.99755528 比特币
```

有限且递减的发行量创造了一个固定的货币供应，抵制了通货膨胀。与中央银行可以无限制地印刷法定货币不同，没有任何个人或实体有能力膨胀比特币的供应。

> **通货紧缩的货币**
>
> 固定且递减的货币发行最重要且备受争议的后果是货币往往具有固有的通货紧缩性。通货紧缩是由于供需不平衡导致货币价值上升（和汇率上升）的现象。价格通货紧缩是通货膨胀的反面，意味着货币随时间具有更多的购买力。
>
> 许多经济学家认为通货紧缩的经济是一场灾难，应该尽一切努力避免。这是因为在通货紧缩迅速的时期，人们往往会囤积货币而不是花费它，希望价格会下跌。这种现象在日本的“失落的十年”期间出现过，当时需求完全崩溃，推动该货币陷入通货紧缩螺旋。
>
> 比特币专家认为，通货紧缩本身并不是坏事。相反，通货紧缩与需求崩溃相关，因为这是我们研究通货紧缩的最明显的例子。在可能无限印刷货币的法定货币中，要进入通货紧缩螺旋是非常困难的，除非需求完全崩溃且不愿意印刷货币。比特币中的通货紧缩不是由于需求崩溃引起的，而是由于可预测的供应受限造成的。 当然，通货紧缩的积极方面是它是通货膨胀的反面。通货膨胀会导致货币逐渐贬值，从而导致一种形式的隐性税收，惩罚储户以援助债务人（包括最大的债务人，即政府本身）。受政府控制的货币存在易于发行债务的道德风险，这些债务后来可以通过贬值以牺牲储户的利益而被抹去。
>
> 尚待观察的是，当通货紧缩不是由经济迅速收缩驱动时，货币的通货紧缩特征是否是一个问题，还是一个优势，因为对通货膨胀和贬值的保护超过了通货紧缩的风险。

去中心化共识

\ 在前一章中，我们讨论了区块链，即所有交易的全局列表，每个比特币网络中的参与者都接受它作为所有权转移的权威记录。

但是，如何让网络中的每个人在不必信任任何人的情况下就谁拥有什么达成单一的普遍“真相”呢？所有传统的支付系统都依赖于一个信任模型，其中有一个中央机构提供清算服务，基本上验证和清算所有交易。比特币没有中央机构，但一些全节点却拥有一个完整的公共区块链副本，可以将其视为权威记录，而无需信任任何人。区块链不是由中央机构创建的，而是由网络中的每个节点独立组装而成的。不知何故，网络中的每个节点，在通过不安全的网络连接传输的信息的作用下，都能得出相同的结论，并组装出与其他人相同的区块链副本。本章将探讨比特币网络在没有中央机构的情况下如何实现全局共识的过程。

中本聪的一个发明是分布式的紧急共识机制。紧急共识是指共识不是明确地实现的——没有选举或固定的共识达成时刻。相反，共识是成千上万个独立节点异步交互的结果，所有节点都遵循简单的规则。比特币的所有属性，包括货币、交易、支付以及不依赖中央机构或信任的安全模型，都源于这一发明。

\ 比特币的分散式共识是由网络中各个节点上独立进行的四个过程相互作用而产生的：

- 每个完整节点对每个交易进行独立验证，基于一套全面的标准清单
- 挖矿节点将这些交易独立聚合到新的区块中，并通过工作证明算法进行计算证明
- 每个节点对新区块进行独立验证，并将其组装成链
- 每个节点通过工作证明选择具有最大累积计算量的链

在接下来的几节中，我们将探讨这些过程以及它们如何相互作用，以创造出网络范围共识的 emergent 性质，使得任何比特币节点都能组装出自己的权威、可信的、公共的、全球区块链副本。

交易的独立验证

\ 在第6章中，我们看到钱包软件通过收集UTXO（未使用的交易输出）、提供适当的身份验证数据，然后构建分配给新所有者的新输出来创建交易。然后，生成的交易被发送到比特币网络中的相邻节点，以便在整个比特币网络中传播。

然而，在将交易转发到其邻居之前，每个接收到交易的比特币节点都会首先验证该交易。这确保只有有效的交易才会在网络中传播，而无效的交易则会在首个遇到它们的节点处被丢弃。

每个节点都会根据长长的检查清单验证每个交易：

- 交易的语法和数据结构必须正确。
- 输入和输出列表都不为空。
- 交易的重量足够小，以便使其适合在一个区块中。
- 每个输出值以及总值必须在允许的值范围内（大于等于零，但不超过2100万比特币）。
- 锁定时间等于INT_MAX，或者锁定时间和序列值符合锁定时间和BIP68规则。
- 交易中包含的签名操作（SIGOPS）的数量少于签名操作限制
- 花费的输出与内存池中的输出或主分支中未花费的输出匹配。
- 对于每个输入，如果引用的输出交易是coinbase输出，则必须至少有COINBASE_MATURITY（100）次确认。任何绝对或相对锁定时间也必须满足。节点可能在它们成熟之前转发交易到一个区块，因为如果包含在下一个区块中，它们就会成熟。
- 如果输入值的总和小于输出值的总和，则拒绝。
- 每个输入的脚本必须针对相应的输出脚本进行验证。

请注意，这些条件随着时间的推移而改变，以添加新功能或解决新类型的拒绝服务攻击。

通过在收到每个交易并在传播之前进行独立验证，每个节点构建了一个有效但尚未确认的交易池，称为内存池或mempool。

挖矿节点

一些节点在比特币网络中是专门的节点，称为矿工节点。小静是一个比特币矿工；他通过运行“挖矿设备”来赚取比特币，这是一个专门设计用于挖矿比特币的计算机硬件系统。小静的专用挖矿硬件连接到运行完整节点的服务器。与其他完整节点一样，小静的节点接收并传播比特币网络上的未确认交易。然而，小静的节点还将这些交易聚合到新的区块中。

让我们跟随在Alice从Bob购买商品时（见“从在线商店购买”第16页）创建的区块。为了演示本章的概念，让我们假设包含Alice交易的区块是由小静的挖矿系统挖出的，并跟踪Alice的交易如何成为这个新区块的一部分。

小静的挖矿节点维护着区块链的本地副本。在Alice购买商品时，小静的节点已经跟上了具有最多工作证明的区块链。小静的节点正在监听交易，尝试挖掘新的区块，并且还在监听其他节点发现的区块。当小静的节点正在挖掘时，它通过比特币网络接收到了一个新的区块。这个区块的到来标志着对该区块的搜索结束，以及对下一个区块的搜索开始。

在前几分钟里，小静的节点在寻找上一个区块的解决方案的同时，还在准备下一个区块的交易。到现在为止，它已经在内存池中收集了几千个交易。在接收到新的区块并验证通过后，小静的节点还将把它与内存池中的所有交易进行比较，并删除已经包含在该区块中的交易。留在内存池中的交易是未确认的，正在等待记录在新的区块中。

小静的节点立即构建一个新的部分区块，作为下一个区块的候选区块。这个区块称为候选区块，因为它还不是一个有效的区块，因为它不包含有效的工作证明。只有当矿工成功找到符合工作证明算法的解决方案时，区块才变得有效。

当小静的节点聚合来自内存池的所有交易时，新的候选区块将包含数千个每笔交易都支付的交易费，他将尝试领取这些费用。

Coinbase交易

\ 在任何区块中，第一个交易都是一笔特殊的交易，称为coinbase交易。这笔交易由Jing的节点构建，并支付他挖矿所获得的奖励。

Jing的节点将coinbase交易创建为对自己钱包的支付。Jing收集的挖矿奖励总额是区块补贴（2023年为6.25个新比特币）和包含在区块中的所有交易的交易费用之和。

与常规交易不同，coinbase交易不消耗（花费）UTXO作为输入。相反，它只有一个输入，称为coinbase输入，隐含包含区块奖励。coinbase交易必须至少有一个输出，并且可以有尽可能多的输出，以适应区块。在2023年的coinbase交易中，常见的是有两个输出：其中一个是使用OP_RETURN的零值输出，用于承诺区块中所有隔离见证（segwit）交易的所有见证。另一个输出支付矿工的奖励。

\

Coinbase奖励和交易费

为构建coinbase交易，Jing的节点首先计算交易费的总额：

```
Total Fees = Sum(Inputs) – Sum(Outputs)
```

接下来，Jing的节点计算新区块的正确奖励。奖励是根据区块高度计算的，从每个区块的50比特币开始，每210,000个区块减半一次。

可以在比特币核心客户端中的GetBlockSubsidy函数中看到计算方式，如示例12-3所示。

\ 示例 12-3. 计算区块奖励 —— 函数 GetBlockSubsidy，比特币核心客户端，main.cpp

```
CAmount GetBlockSubsidy(int nHeight, const Consensus::Params& consensusParams)
{
    int halvings = nHeight / consensusParams.nSubsidyHalvingInterval;
    // Force block reward to zero when right shift is undefined.
    if (halvings >= 64)
        return 0;
    CAmount nSubsidy = 50 * COIN;
    // Subsidy is cut in half every 210,000 blocks.
    nSubsidy >>= halvings;
    return nSubsidy;
}
```

\ 初始补贴以 satoshi 为单位计算，将 50 乘以 COIN 常量（100,000,000 satoshi）。这将初始奖励（nSubsidy）设置为 50 亿 satoshi。

接下来，该函数通过当前区块高度除以减半间隔（SubsidyHalvingInterval）来计算已发生的减半次数。

然后，函数使用二进制右移操作符，在每次减半中将奖励（nSubsidy）除以二。对于区块 277,316，这将使 50 亿 satoshi 的奖励右移一次（一次减半），结果为 25 亿 satoshi，或 25 个比特币。经过第 33 次减半后，奖励将被舍入为零。使用二进制右移操作符是因为它比多次重复的除法更有效。为了避免潜在的错误，当进行了 63 次减半后，将跳过移位操作，并将补贴设置为 0。

最后，将 coinbase 奖励（nSubsidy）与交易费用（nFees）相加，并返回总和。

> 如果 Jing 的挖矿节点编写 coinbase 交易，是什么阻止了 Jing “奖励”自己 100 或 1,000 个比特币？答案是，夸大的奖励将导致其他所有人都认为该区块无效，从而浪费了 Jing 用于 PoW 的电力。只有在所有人接受该区块时，Jing 才能花费奖励。

Coinbase 交易的结构

通过这些计算，Jing 的节点随后构建 coinbase 交易，以支付他自己的区块奖励。

coinbase 交易具有特殊的格式。它不是像普通交易输入一样指定要花费的前一个 UTXO，而是有一个“coinbase”输入。我们在“Inputs”（第123页）中研究了交易输入。让我们将普通交易输入与 coinbase 交易输入进行比较。表12-1显示了普通交易的结构，而表12-2显示了coinbase交易输入的结构。

表12-1. “正常”交易输入的结构

大小	字段	描述
32字节	交易哈希(Transaction Hash)	指向要被消费的 UTXO 所在的交易
4字节	输出索引(Output Index)	要被消费的 UTXO 的索引号，第一个是 0
1-9字节（紧凑编码）	脚本长度(Script Size)	脚本长度（以字节为单位），后跟着
可变长度	输入脚本(Input Script)	一个满足UTXO输出脚本条件的脚本
4字节	序列号(Sequence Number)	用于BIP68时间锁定和交易替换信号的多用途字段

表12-2. Coinbase交易输入的结构

大小	字段	描述
32字节	交易哈希(Transaction Hash)	所有位都是零：不是交易哈希引用
4字节	输出索引(Output Index)	所有位都是一：0xFFFFFFFF
1字节	Coinbase 数据大小(Coinbase Data Size)	Coinbase 数据的长度，从 2 到 100 字节
可变长度	Coinbase 数据(Coinbase Data)	任意数据，用于额外的 nonce 和挖矿标签；在 v2 块中，必须以区块高度开头
4字节	序列号(Sequence Number)	设置为0xFFFFFFFF

在 coinbase 交易中，前两个字段的值不代表 UTXO 引用。第一个字段不是“交易哈希”，而是填充了 32 字节，全部设置为零。而“输出索引”填充了 4 字节，全部设置为 0xFF（十进制 255）。输入脚本被 coinbase 数据替代，这是矿工使用的数据字段，我们将在接下来看到。

Coinbase数据

\ Coinbase 交易没有输入脚本字段。相反，这个字段被 coinbase 数据替换，其长度必须在 2 到 100 字节之间。除了前几个字节外，coinbase 数据的其余部分可以由矿工任意使用；它是任意的数据。

例如，在创世区块中，中本聪在 coinbase 数据中添加了文本“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks”，将其作为这个区块可能创建的最早日期的证明，并传递了一条信息。目前，矿工经常使用 coinbase 数据来包含额外的 nonce 值和标识矿池的字符串。

coinbase 的前几个字节曾经是任意的，但现在不再是这样了。根据 BIP34，版本 2 的区块（版本字段设置为 2 或更高）必须在 coinbase 字段的开头作为脚本“push”操作包含区块高度。

构建区块头

构建区块头，挖矿节点需要填写六个字段，如表12-3所列。

表 12-3. 区块头结构

大小	字段	描述
4字节	版本(Version)	一个多功能的位字段
32字节	前一区块哈希(Previous Block Hash)	指向链中前一（父）区块的哈希的引用
32字节	默克尔根(Merkle Root)	这个区块交易的默克尔树的根节点哈希
4字节	时间戳(Timesteamp)	此区块的大致创建时间（自Unix纪元起的秒数）
4字节	目标(Target)	此区块的工作量证明算法目标
4字节	唯一随机数(Nonce)	一个用于工作量证明算法的计数器

\ 版本字段最初是一个整数字段，并在比特币网络的三次升级中使用，这些升级定义在BIPs 34、66和65中。每次升级时，版本号都会递增。后来的升级将版本字段定义为位字段，称为versionbits，允许同时进行最多29个升级；详情请参阅“BIP9：信号和激活”第298页。更晚一些，矿工开始使用一些versionbits作为辅助随机数字段。

> BIPs 34、66和65中定义的协议升级是按照这个顺序依次发生的，BIP66（严格的DER）在BIP65（OP_CHECKTIMELOCKVERIFY）之前发生，因此比特币开发人员通常按照这个顺序而不是按照数字顺序列出它们。

今天，versionbits字段没有意义，除非正在进行升级共识协议的尝试，在这种情况下，您将需要阅读其文档，以确定它如何使用versionbits。

接下来，挖矿节点需要添加“上一个区块哈希”（也称为prevhash）。这是来自网络的上一个块的块头哈希，Jing的节点已经接受并选择为他的候选块的父块。

> 通过选择候选块头中的上一个区块哈希字段所指示的特定父区块，Jing将他的挖矿算力承诺到扩展以该特定区块结尾的链上。

下一步是使用默克尔树来承诺所有交易。每个交易都按拓扑顺序列出，使用其见证交易标识符（wtxid），其中32个0x00字节代表第一个交易（coinbase）的wtxid。正如我们在“Merkle Trees”中看到的，如果有奇数个wtxid，则最后一个wtxid将与自身哈希，创建每个包含一个交易哈希的节点。然后，交易哈希按成对组合，创建树的每个级别，直到所有交易被总结到树的“根”节点中。Merkle树的根将所有交易总结为一个单一的32字节值，即见证根哈希。

见证根哈希被添加到coinbase交易的输出中。如果块中的交易不需要包含见证结构，则可以跳过此步骤。然后，每个交易（包括coinbase交易）都使用其交易标识符（txid）进行列出，并用于构建第二个Merkle树，其根成为Merkle根，对其进行承诺到块头中。

Jing的挖矿节点然后添加一个4字节的时间戳，编码为Unix“纪元”时间戳，它是基于从1970年1月1日UTC/GMT午夜开始经过的秒数。

然后，Jing的节点填写nBits目标，它必须设置为使其成为有效块所需的PoW的紧凑表示。目标以“目标位”度量存储在块中，它是目标的尾数-指数编码。编码有一个字节的指数，后跟一个3字节的尾数（系数）。例如，在块277,316中，目标位值为0x1903a30c。第一部分0x19是十六进制指数，而下一部分0x03a30c是系数。目标的概念在“Retargeting to Adjust Difficulty”中解释，“目标位”表示在“Target Representation”中解释。

最后一个字段是随机数，初始化为零。

填写了所有其他字段后，候选块的头部现在已经完成，挖矿过程可以开始。目标现在是找到一个导致哈希小于目标的头部。挖矿节点将需要测试数十亿或数万亿个头部的变化，直到找到满足要求的版本。

挖掘区块

现在，由 Jing 的节点构建了一个候选块，是时候让 Jing 的硬件挖矿设备来“挖”这个块，以找到使该块有效的工作证明算法的解决方案。在本书中，我们已经研究了密码哈希函数在比特币系统各个方面的应用。SHA256 哈希函数是比特币挖矿过程中使用的函数。

简单来说，挖矿是反复对候选块头部进行哈希，不断更改一个参数，直到生成的哈希值匹配特定目标。哈希函数的结果无法提前确定，也无法创建一个能够产生特定哈希值的模式。哈希函数的这一特性意味着，要想产生与特定目标匹配的哈希结果，唯一的方法就是不断尝试，修改输入，直到所需的哈希结果出现为止。

工作量证明算法(PoW)

哈希算法接受任意长度的数据输入，并产生一个固定长度的确定性结果，称为摘要。摘要是对输入的数字承诺。对于任何特定的输入，生成的摘要将始终相同，并且可以轻松地由实施相同哈希算法的任何人进行计算和验证。密码哈希算法的一个关键特征是，找到产生相同摘要的两个不同输入是计算上不可行的（称为碰撞）。作为推论，选择一种输入以产生所需的摘要也几乎是不可能的，除非尝试随机输入。

使用SHA256，输出始终是256位长，而不考虑输入的大小。例如，我们将计算短语“Hello, World!”的SHA256哈希值：

```
$ echo "Hello, world!" | sha256sum
d9014c4624844aa5bac314773d6b689ad467fa4e1d1a50a1b8a99d5a95f72ff5 -
```

这个256位的输出（用十六进制表示）是该短语的哈希或摘要，并且依赖于短语的每个部分。添加一个字母、标点符号或任何其他字符将产生一个不同的哈希。

在这种情况下使用的一个变量被称为 nonce。Nonce 用于改变密码函数的输出，这里用于改变 SHA256 对该短语的摘要输出。

为了从这个算法中提出一个挑战，让我们设定一个目标：找到一个短语，它产生一个十六进制哈希，以零开头。幸运的是，这并不困难，如示例 12-4 所示。

示例 12-4. 简单的工作量证明实现

```
$ for nonce in $( seq 100 ) ; do echo "Hello, world! $nonce" | sha256sum ; done
3194835d60e85bf7f728f3e3f4e4e1f5c752398cbcc5c45e048e4dbcae6be782 -
bfa474bbe2d9626f578d7d8c3acc1b604ec4a7052b188453565a3c77df41b79e -
[...]
f75a100821c34c84395403afd1a8135f685ca69ccf4168e61a90e50f47552f61 -
09cb91f8250df04a3db8bd98f47c7cecb712c99835f4123e8ea51460ccbec314 -
```

短语“Hello, World! 32”产生了以下哈希，符合我们的标准：09cb91f8250df04a3db8bd98f47c7cecb712c99835f4123e8ea51460ccbec314。找到它花费了32次尝试。从概率上讲，如果哈希函数的输出是均匀分布的，我们预计每16次哈希（从0到F的16个十六进制数字中的一个）就会找到一个以0开头的结果。用数字表示，这意味着找到的哈希值小于0x1000000000000000000000000000000000000000000000000000000000000000。我们将这个阈值称为目标，目标是找到一个数字上小于目标的哈希。如果我们降低目标，找到一个小于目标的哈希的任务就会变得越来越困难。

为了给出一个简单的类比，想象一场游戏，玩家反复投掷一对骰子，试图得到小于指定目标的点数。在第一轮中，目标是12。除非你掷出双6，否则你就赢了。在下一轮中，目标是11。玩家必须掷出10或更少的点才能赢，这又是一个容易的任务。假设几轮后，目标降到了5。现在，超过一半的骰子投掷将超过目标，因此无效。要赢得的次数越低，掷骰子的次数就越多。最终，当目标是3（可能的最小值）时，每36次投掷中只有一次，约占3%，会产生一个成功的结果。

从知道骰子游戏目标是3的观察者的角度来看，如果有人成功地进行了一次投掷，可以假设他们平均尝试了36次。换句话说，可以从目标所施加的困难来估计成功所需的工作量。当算法基于SHA256等确定性函数时，输入本身就构成了证明，证明了为了产生低于目标的结果所做的一定量工作。因此，工作量证明。

> 尽管每次尝试都会产生随机结果，但任何可能结果的概率都可以事先计算。因此，指定难度的结果构成了特定工作量的证明。

在示例12-4中，获胜的“nonce”是32，这个结果可以由任何人独立确认。任何人都可以将数字32添加为短语“Hello, world!”的后缀并计算哈希，以验证其是否小于目标：

```
$ echo "Hello, world! 32" | sha256sum
09cb91f8250df04a3db8bd98f47c7cecb712c99835f4123e8ea51460ccbec314 -
```

虽然验证只需一次哈希计算，但我们需要32次哈希计算才能找到有效的 nonce。如果我们有一个更低的目标（更高的难度），那么需要进行更多的哈希计算才能找到合适的 nonce，但任何人只需要一次哈希计算来验证。通过知道目标，任何人都可以利用统计数据估计难度，从而大致了解需要多少工作量才能找到这样的 nonce。

> 工作量证明必须产生一个小于目标的哈希。更高的目标意味着更容易找到小于目标的哈希。更低的目标意味着更难找到小于目标的哈希。目标和难度是反相关的。

比特币的工作量证明与示例12-4中显示的挑战非常相似。矿工构建一个填充了交易的候选块。然后，矿工计算此块头的哈希，并查看其是否小于当前目标。如果哈希不小于目标，则矿工将修改 nonce（通常只是递增一个）并重试。在比特币网络的当前难度下，矿工必须尝试大量次数才能找到一个导致块头哈希低到足够的 nonce。

目标表示

区块头中包含的目标值使用一种称为“目标位”或简称“位”（bits）的记法表示，在第277,316个区块中的值为0x1903a30c。这种记法将工作量证明目标表达为一个系数/指数格式，其中前两个十六进制数字表示指数，接下来的六个十六进制数字表示系数。因此，在这个区块中，指数为0x19，系数为0x03a30c。

从这种表示中计算难度目标的公式是：

$$ target = coefficient *2^{(8* (exponent -3))}

$$

使用该公式，以及难度位值为0x1903a30c，我们得到：

$$ target = 0x03a30c *2^{(0x08* (0x19 - 0x03))}

$$

结果为：

22,829,202,948,393,929,850,749,706,076,701,368,331,072,452,018,388,575,715,328

或者，用十六进制表示：

0x0000000000000003A30C000000000000000000000000000000000000000000000000

这意味着对于高度为277,316的有效区块，其区块头哈希必须小于目标值。在二进制中，该数字必须有超过60个前导零位。在这个难度水平下，每秒处理1万亿次哈希（1 TH/sec）的单个矿工平均每8,496个区块或平均每59天才能找到一个解决方案。

调整难度的重新定位

正如我们所见，目标确定了难度，因此影响了寻找解决方案的时间。这引出了一个显而易见的问题：为什么需要调整难度，谁来进行调整，以及如何进行调整呢?

比特币的区块平均每 10 分钟产生一个。这是比特币的节拍，支撑着货币发行频率和交易结算速度。它不仅在短期内必须保持恒定，而且在数十年的时间跨度内也必须保持稳定。随着时间的推移，预计计算机算力将继续快速增长。此外，参与挖矿的参与者数量和他们使用的计算机也将不断变化。为了保持区块生成时间在 10 分钟左右，挖矿的难度必须进行调整，以适应这些变化。 实际上，工作证明目标是一个动态参数，定期调整以满足每 10 分钟一个区块的目标。简单来说，目标被设置为当前挖矿算力会产生 10 分钟一个区块的时间间隔。

那么，在完全去中心化的网络中如何进行这样的调整呢？重新调整难度在每个节点上都会自动进行。每产生 2,016 个区块，所有节点都会重新调整工作证明。计算实际时间跨度与期望的每个区块 10 分钟时间跨度之间的比例，并对目标进行相应的调整（增加或减少）。简单来说：如果网络发现区块生成速度比每 10 分钟快，难度会增加（目标减少）。如果区块的发现速度比预期的慢，难度会减少（目标增加）。

该方程可以总结为：

新目标 = 旧目标 *（20,160 分钟 / 上一个 2,015 个区块的实际时间）

> 尽管每产生 2,016 个区块就进行一次目标校准，但由于比特币软件中的一个偏移错误，实际上是基于前 2,015 个区块的总时间（而不是应该的 2,016 个区块），导致难度朝更高方向偏差 0.05%。

示例 12-5 展示了比特币核心客户端使用的代码。

示例 12-5. 调整工作证明目标：pow.cpp 中的 CalculateNextWorkRequired()

```
// Limit adjustment step
int64_t nActualTimespan = pindexLast->GetBlockTime() - nFirstBlockTime;
LogPrintf(" nActualTimespan = %d before bounds\n", nActualTimespan);
if (nActualTimespan < params.nPowTargetTimespan/4)
    nActualTimespan = params.nPowTargetTimespan/4;
if (nActualTimespan > params.nPowTargetTimespan*4)
    nActualTimespan = params.nPowTargetTimespan*4;
// Retarget
const arith_uint256 bnPowLimit = UintToArith256(params.powLimit);
arith_uint256 bnNew;
arith_uint256 bnOld;
bnNew.SetCompact(pindexLast->nBits);
bnOld = bnNew;
bnNew *= nActualTimespan;
bnNew /= params.nPowTargetTimespan;

if (bnNew > bnPowLimit)
    bnNew = bnPowLimit;
```

\ 为了避免难度的极端波动，每个周期内的重新调整幅度必须小于四倍（4）。如果所需的目标调整超过四倍，它将被调整为四倍，不会更多。任何进一步的调整将在下一个重新调整周期内完成，因为不平衡将持续到接下来的2,016个区块。因此，哈希功率和难度之间的巨大差异可能需要多个2,016个区块周期来平衡。

请注意，目标与交易数量或交易价值无关。这意味着用于保护比特币的哈希功率和因此消耗的电力完全独立于交易数量。比特币可以扩展并保持安全性，而无需从当前水平增加哈希功率。哈希功率的增加代表了市场力量，因为新的矿工进入市场。只要足够多的哈希功率由诚实的矿工控制以追求奖励，就足以防止“接管”攻击，因此足以保护比特币。

挖矿的难度与电力成本和比特币与用于支付电力的货币的汇率密切相关。高性能的挖矿系统在当前一代硅制造技术下尽可能高效，以最高速率将电力转换为哈希计算。对挖矿市场的主要影响是一千瓦时电力的价格与比特币之间的汇率，因为这决定了挖矿的盈利能力，从而影响了进入或退出挖矿市场的激励。

中位时间过去（Median-Time-Past, MPT）

\ 在比特币中，墙上时间和共识时间之间存在微妙但非常重要的区别。比特币是一个去中心化的网络，这意味着每个参与者都有自己的时间观念。网络上的事件不会在所有地方即时发生。必须考虑网络延迟，以适应每个节点的时间观点。最终，一切都会同步，形成一个共同的区块链。 比特币大约每10分钟就会就过去的区块链状态达成共识。

区块头中设置的时间戳由矿工设置。共识规则允许一定程度的灵活性，以解决去中心化节点之间的时钟精度差异。然而，这给矿工造成了一个不利的激励，即在区块中虚报时间。例如，如果矿工将时间设定在未来，他们可以降低难度，从而可以挖掘更多区块并声称一部分留给未来矿工的区块补贴。如果他们可以将某些区块的时间设定在过去，那么他们可以将当前时间用于其他区块，从而再次使区块之间的时间看起来很长，以操纵难度。

为了防止操纵，比特币有两个共识规则。第一个是，没有节点将接受任何时间超过两小时的区块。第二个是，没有节点将接受一个时间小于或等于最近的11个区块的中位数时间的区块，称为过去中位数时间（MTP）。

作为BIP68相对时间锁定激活的一部分，对交易中的时间（绝对和相对）计算方式也进行了更改。以前，矿工可以在区块中包含任何具有与该区块相等或更早的时间锁定的交易。这促使矿工使用他们认为可能的最新时间（接近两小时的未来）以便更多的交易符合他们的区块。

为了消除虚报的激励并加强时间锁的安全性，BIP113被提出，并在相对时间锁的BIP同时激活。MTP成为所有时间锁计算中使用的共识时间。通过从大约过去两小时开始计算中点，减少了任何一个区块时间戳的影响。通过整合11个区块，没有单个矿工可以影响时间戳，以获得尚未成熟的具有时间锁的交易的费用。

MTP改变了锁定时间、CLTV、序列和CSV的时间计算实现。由MTP计算的共识时间通常比墙上时钟时间晚约一小时。如果您创建时间锁定交易，请在估算要编码的锁定时间、序列、CLTV和CSV的期望值时予以考虑。

\

成功挖掘区块

如前所述，Jing的节点已经构建了一个候选区块并准备好进行挖矿。Jing拥有多个硬件挖矿设备，其中包括专用的应用特定集成电路，数十万个集成电路以惊人的速度并行运行比特币的双 SHA256 算法。许多这些专用机器通过USB或局域网连接到他的挖矿节点。接下来，运行在Jing桌面上的挖矿节点将区块头传输到他的挖矿硬件上，开始每秒测试数万亿种不同的区块头变化。由于nonce只有32位，挖矿硬件在耗尽所有nonce可能性（约40亿个）后，会更改区块头（调整coinbase额外的nonce空间、版本位或时间戳），并重置nonce计数器，测试新的组合。

大约在开始挖掘特定区块后的11分钟内，其中一台挖矿机找到了一个解决方案，并将其发送回挖矿节点。

立即，Jing的挖矿节点将该区块传输给所有的对等节点。它们接收、验证，然后传播新的区块。随着区块在网络中扩散，每个节点都将其添加到自己的区块链副本中，将其扩展到新的高度。当挖矿节点接收并验证区块时，它们放弃了在相同高度寻找区块的努力，并立即开始计算链中的下一个区块，使用Jing的区块作为“父区块”。通过在Jing新发现的区块之上构建，其他矿工实际上是在利用他们的挖矿算力来背书Jing的区块和其延伸的链。

在下一节中，我们将看到每个节点用于验证区块并选择最难链的过程，从而形成构成去中心化区块链的共识。

验证新区块

比特币共识机制的第三步是网络中每个节点对每个新区块进行独立验证。随着新解决的区块在网络中传播，每个节点都会执行一系列测试来验证它。独立验证还确保只有遵循共识规则的区块才会被纳入区块链中，从而使其矿工获得奖励。违反规则的区块将被拒绝，不仅会使其矿工失去奖励，还会浪费寻找工作证明解决方案的努力，因此给这些矿工带来了创建区块的所有成本，但却没有给予他们任何奖励。

当一个节点收到一个新区块时，它将通过检查一个必须满足所有条件的长列表来验证该区块；否则，该区块将被拒绝。这些条件可以在比特币核心客户端的 CheckBlock 和 CheckBlockHeader 函数中看到，其中包括：

• 区块数据结构在语法上是有效的。

• 区块头哈希小于目标值（实施工作证明）。

• 区块时间戳位于 MTP 和未来两小时之间（允许时间误差）。

• 区块权重在可接受范围内。

• 第一个交易（仅第一个）是 coinbase 交易。

• 区块中的所有交易都使用“交易独立验证”部分讨论的交易清单进行验证。

网络中每个节点对每个新区块的独立验证确保了矿工无法作弊。在前面的部分中，我们看到矿工可以编写一个交易，以奖励他们在区块内创建的新比特币并获取交易费。为什么矿工不会为自己编写一个价值一千个比特币的交易，而是正确的奖励？因为每个节点都根据相同的规则验证区块。无效的 coinbase 交易会使整个区块无效，导致该区块被拒绝，因此该交易永远不会成为区块链的一部分。矿工必须根据所有节点遵循的共享规则构建一个区块，并使用正确的 PoW 解决方案对其进行挖掘。为此，他们在挖掘过程中耗费了大量的电力，如果他们作弊，所有的电力和努力都将被浪费。这就是为什么独立验证是分散式共识的关键组成部分。

组装和选择区块链

比特币的去中心化共识机制的最后一部分是将区块组装成链，并选择具有最多工作量证明的链。 最佳的区块链是具有最多累积工作量证明的所有有效区块链。

*最佳链*也可能有与最佳链上的区块“同级”的分支。这些区块是有效的，但不是最佳链的一部分。它们被保留供将来参考，以防这些次要链中的任何一个后来成为主要链。当出现同级区块时，通常是由于在相同高度几乎同时挖掘不同区块造成的。

当收到一个新区块时，节点将尝试将其添加到现有的区块链上。节点将查看区块的“上一个区块哈希”字段，该字段是指向区块父级的引用。然后，节点将尝试在现有的区块链中找到该父级。大多数情况下，父级将是最佳链的“尖端”，这意味着这个新区块扩展了最佳链。

有时，新区块并不会扩展最佳链。在这种情况下，节点将新区块的头部连接到一个次要链上，然后将次要链的工作与先前的最佳链进行比较。如果次要链现在是最佳链，节点将相应地重新组织其已确认交易和可用UTXO的视图。如果节点是矿工，它现在将构建一个候选区块，扩展这个新的，具有更多工作量证明的链。

通过选择具有最大累积工作量的有效链，所有节点最终达成网络范围内的共识。随着更多的工作被添加，最终解决了链之间的临时差异，从而扩展了可能的链。

> 本节中描述的区块链分叉是由全球网络中的传输延迟自然发生的结果。我们也将在本章后面看到故意引发的分叉

\ 分叉几乎总是在一个区块内解决的。如果两个区块几乎同时被位于前一个分叉的相反“方向”的矿工发现，那么意外的分叉可能会延伸到两个区块。然而，这种情况发生的几率很低。

比特币的区块间隔为10分钟，是快速确认时间和分叉概率之间的设计妥协。更快的区块时间会使交易似乎更快得到确认，但会导致区块链分叉更频繁，而更慢的区块时间会减少分叉的数量，但使结算速度看起来更慢。

> 哪种更安全：一个交易包含在平均每10分钟一个区块的区块中，还是一个交易包含在已建立在其上的九个区块的区块中，平均每个区块之间的时间是一分钟？答案是它们同样安全。想要进行双重支付的恶意矿工需要做的工作量与总网络算力的10分钟相等，以创建具有相同工作证明的链。
>
> 区块之间时间缩短并不意味着更早的结算。它的唯一好处是为那些愿意接受这些保证的人提供更弱的保证。例如，如果你愿意接受矿工在三分钟内就同意了最佳区块链作为足够的安全性，那么你会更喜欢一个每分钟一个区块的系统，你可以等待三个区块，而不是一个每10分钟一个区块的系统。区块之间的时间越短，越容易因意外分叉而浪费矿工的工作量（除其他问题外），因此许多人更喜欢比特币的10分钟区块而不是更短的区块间隔。

挖矿和哈希彩票

\ 比特币挖矿是一个极具竞争性的行业。比特币的哈希算力每年呈指数增长。有些年份的增长反映了技术的完全变革，比如2010年和2011年，许多矿工从使用CPU挖矿转向了GPU挖矿和可编程门阵列（FPGA）挖矿。2013年ASIC挖矿的引入又使挖矿能力实现了另一个巨大的飞跃，通过将双SHA256函数直接置于专门用于挖矿目的的硅片上。最早的这种芯片在一个单独的盒子中就能提供比2010年整个比特币网络的挖矿能力还要多。

截至目前为止，人们认为比特币挖矿设备已经没有更大的飞跃了，因为该行业已经达到了摩尔定律的前沿，摩尔定律规定计算密度大约每18个月翻一番。尽管如此，网络的挖矿能力仍在快速增长。

额外的Nonce解决方案

\ 自2012年以来，挖矿已经发展出一种解决区块头结构中的一个基本限制的方法。在比特币的早期阶段，挖矿者可以通过迭代Nonce直到得到的哈希值低于目标值来找到一个区块。随着难度的增加，挖矿者经常会循环遍历所有40亿个Nonce值，但仍然找不到一个区块。然而，这个问题很容易通过更新区块时间戳以考虑经过的时间来解决。因为时间戳是头部的一部分，改变时间戳会允许挖矿者再次迭代Nonce值，从而产生不同的结果。然而，一旦挖矿硬件超过4GH/s，这种方法就变得越来越困难，因为Nonce值在不到一秒钟的时间内就会被耗尽。随着ASIC挖矿设备开始超过TH/s的哈希率，挖矿软件需要更多的Nonce值空间才能找到有效的区块。时间戳可以被延长一点，但如果将其移动得太远，将会导致区块无效。区块头需要一种新的变化来源。

一个被广泛采用的解决方案是使用coinbase交易作为额外Nonce值的来源。因为coinbase脚本可以存储2到100字节的数据，挖矿者开始使用这个空间作为额外Nonce空间，从而使他们能够探索更大范围的区块头值以找到有效的区块。coinbase交易包含在默克尔树中，这意味着coinbase脚本的任何更改都会导致默克尔根的变化。8字节的额外Nonce加上4字节的“标准”Nonce，使得挖矿者每秒可以探索总共296（8后跟28个零）种可能性，而无需修改时间戳。

今天广泛使用的另一种解决方案是利用区块头版本位字段的最多16位用于挖矿，这就是BIP320中描述的方法。如果每台挖矿设备都有自己的coinbase交易，那么通过仅对区块头进行更改，这使得个别挖矿设备可以达到高达281 TH/s的哈希速率。这比每40亿个哈希值增加一次coinbase交易中的额外Nonce更简单，后者需要重新计算默克尔树的整个左侧直到根节点。

挖矿池

矿工们在这个竞争激烈的环境中，单独工作的个体矿工（也称为独立矿工）几乎没有机会。他们找到一个足以抵消电费和硬件成本的区块的可能性非常低，这就像是在赌博，就像是在买彩票一样。即使是最快的消费者 ASIC 挖矿系统也无法与商业运营相比，商业运营可以将数以万计的这些系统堆叠在靠近发电站的巨大仓库中。许多矿工现在合作组成矿池，将他们的算力汇集在一起，分享奖励给成千上万的参与者。通过参与矿池，矿工能够获得总奖励的一个较小份额，但通常每天都会获得奖励，减少了不确定性。

让我们看一个具体的例子。假设一个矿工购买了一批挖矿硬件，总算力相当于当前总网络算力的 0.0001%。如果协议难度永远不变，那么该矿工将大约需要 20 年的时间才能找到一个新的区块。这可能是一个漫长的等待时间来获得报酬。然而，如果该矿工与其他矿工一起在一个矿池中合作，这些矿工的总算力相当于总网络算力的 1%，他们平均每天将会挖到超过一个区块。该矿工将只会收到他们在奖励中的一部分份额（减去矿池收取的任何费用），因此他们每天只会收到一小部分奖励。如果他们每天挖矿 20 年，他们将获得与自己独立挖到平均区块相同的金额（不考虑矿池费用）。唯一的基本区别在于他们收到支付的频率。

挖矿池通过专门的矿池挖矿协议协调着数百甚至数千名矿工。矿工们将他们的挖矿设备配置为连接到矿池服务器，之前需要在矿池创建一个帐户。他们的挖矿硬件在挖矿时保持连接到矿池服务器，与其他矿工同步他们的努力。因此，矿池矿工共同努力挖掘一个区块，然后分享奖励。

成功的区块将奖励支付给矿池的比特币地址，而不是给个体矿工。一旦矿工的奖励份额达到了一定的阈值，矿池服务器将定期向矿工的比特币地址进行支付。通常，矿池服务器会收取一定比例的奖励作为提供矿池挖矿服务的费用。

\ 参与矿池的矿工分担搜索候选区块解决方案的工作，根据其挖矿贡献获得“份额”。挖矿池为获得份额设置了更高的目标（更低的难度），通常比比特币网络的目标容易多达1000倍。当矿池中的某个人成功挖掘了一个区块时，奖励由矿池获得，然后按照各个矿工贡献的份额比例与所有矿工分享。

许多矿池对任何矿工都是开放的，无论其规模大小、专业性或业余性。因此，矿池的参与者可能有些人只有一台小型的挖矿机器，而其他人可能拥有一间装满高端挖矿硬件的车库。有些人可能只使用几十千瓦的电力进行挖矿，而另一些人可能运行着消耗兆瓦电力的数据中心。矿池如何衡量个人的贡献，以公平地分配奖励，又不会出现作弊的可能性呢？答案是利用比特币的工作证明算法来衡量每个矿池矿工的贡献，但设置较低的难度，以便即使最小的矿池矿工也能够频繁地赢得份额，从而有利于向矿池做出贡献。通过为赢得份额设置较低的难度，矿池衡量了每个矿工的工作量。每当矿池矿工找到一个区块头散列小于矿池目标的情况，他们就证明了他们已经完成了寻找该结果的散列工作。该区块头最终会提交给coinbase交易，并可用于证明该矿工使用了一个将区块奖励支付给矿池的coinbase交易。每个矿池矿工都被分配一个略有不同的coinbase交易模板，以便每个人都散列不同的候选区块头，防止工作的重复。

寻找份额的工作以一种在统计上可衡量的方式，有助于寻找一个低于比特币网络目标的散列的整体工作。数千个矿工试图寻找低价值的散列，最终将会找到一个足够低以满足比特币网络的目标。

让我们回到掷骰子游戏的类比。如果骰子玩家的目标是投掷小于四（总体网络难度）的骰子，那么矿池将设置一个更容易的目标，计算矿池玩家成功投掷小于八的次数。当矿池玩家投掷小于八（矿池份额目标）时，他们赢得份额，但他们并没有赢得游戏，因为他们没有达到游戏目标（小于四）。矿池玩家往往会更频繁地实现更容易的矿池目标，即使他们没有实现赢得游戏的更难的目标。偶尔，矿池玩家中的一名将会掷出一个小于四的组合点数，从而矿池获胜。然后，可以根据他们赢得的份额将收益分配给矿池玩家。

同样，一个挖矿池将设置一个（更高、更容易的）矿池目标，以确保个人矿池矿工通过找到区块头散列小于矿池目标而经常赢得份额。偶尔，这些尝试中的一个将产生一个区块头散列小于比特币网络目标的情况，使其成为一个有效的区块，整个矿池都赢得了胜利。

托管矿池

\ 大多数矿池都是“托管”的，这意味着有一个公司或个人在运行一个矿池服务器。矿池服务器的所有者被称为矿池操作员，他们会向矿工收取一定比例的费用作为矿池收益。

矿池服务器运行着专门的软件和矿池挖矿协议，协调着矿池矿工的活动。矿池服务器还连接到一个或多个完整的比特币节点。这使得矿池服务器能够代表矿工验证区块和交易，减轻了他们运行完整节点的负担。对于一些矿工来说，能够在不运行完整节点的情况下进行挖矿是加入托管矿池的另一个好处。

矿池矿工使用诸如Stratum（版本1或版本2）之类的挖矿协议连接到矿池服务器。Stratum v1会创建包含候选区块头的区块模板。矿池服务器通过聚合交易、添加一个coinbase交易（带有额外的nonce空间）、计算默克尔根，并链接到上一个区块的哈希来构建候选区块。然后，候选区块的头部作为模板发送给每个矿池矿工。每个矿池矿工使用区块模板进行挖矿，目标难度比比特币网络的目标难度更高（更容易），并将任何成功的结果发送回矿池服务器以获取份额。

Stratum v2还可以选择允许矿池中的个别矿工选择哪些交易出现在他们自己的区块中，他们可以使用自己的完整节点来选择。

点对点挖矿矿池（P2Pool）

\ 管理型矿池使用 Stratum v1 存在欺诈的可能性，因为矿池操作者可能会指导矿池努力进行双花交易或使块无效（见第288页的“算力攻击”）。此外，集中式矿池服务器代表了单点故障。如果矿池服务器宕机或受到拒绝服务攻击的影响，矿池矿工将无法进行挖矿。为了解决这些集中化问题，2011年提出并实施了一种新的矿池挖矿方法：点对点挖矿矿池（P2Pool），即没有中央操作者的点对点挖矿矿池。

P2Pool 通过分散矿池服务器的功能，实施了一个称为共享链（share chain）的并行类似区块链的系统。共享链是以比比特币区块链更低的难度运行的区块链。共享链允许矿池矿工通过在共享链上每30秒挖掘一个共享块来协作于一个去中心化矿池中。共享链上的每个块记录了为贡献工作的矿池矿工分配的比例份额奖励，并从上一个共享块中延续这些份额。当共享链中的一个共享块也达到比特币网络目标时，它会被传播并包含在比特币区块链上，奖励为所有贡献到所有先前的共享块的矿池矿工。基本上，共享链允许所有矿池矿工使用类似比特币区块链的去中心化共识机制来跟踪所有份额。

P2Pool 挖矿比矿池挖矿更复杂，因为它要求矿池矿工运行一台具有足够磁盘空间、内存和互联网带宽以支持比特币全节点和 P2Pool 节点软件的专用计算机。P2Pool 矿工将他们的挖矿硬件连接到本地的 P2Pool 节点，该节点通过向挖矿硬件发送区块模板来模拟矿池服务器的功能。在 P2Pool 上，个别矿池矿工构建自己的候选块，聚合交易，就像独立挖矿者一样，然后在共享链上协作挖矿。

P2Pool 是一种混合方法，具有比独立挖矿更加精细化的支付优势，但又不像管理型矿池那样给予矿池操作者太多的控制权。

尽管 P2Pool 减少了矿池运营商的权力集中，但它可能会受到针对共享链本身的 51% 攻击的威胁。P2Pool 的更广泛采用并不解决比特币本身的 51% 攻击问题。相反，P2Pool 使比特币作为一个多样化的挖矿生态系统更加健壮。截至撰写本文时，P2Pool 已经逐渐失去了使用，但新的协议，例如 Stratum v2，可以允许个别矿工选择他们在区块中包含的交易。

哈希攻击

\ 比特币的共识机制理论上容易受到矿工（或矿池）的攻击，他们试图利用自己的算力进行不诚实或破坏性的行为。正如我们所见，共识机制依赖于大多数矿工出于自身利益而诚实行事。然而，如果一个矿工或一组矿工能够获得相当大的算力份额，他们就可以攻击共识机制，破坏比特币网络的安全性和可用性。

需要注意的是，算力攻击对未来共识影响最大。随着时间的推移，最佳区块链上的确认交易变得越来越不可变。虽然理论上，可以在任何深度达到分叉，但在实践中，迫使一个非常深的分叉所需的计算能力是巨大的，使得旧区块变得非常难以更改。算力攻击也不会影响私钥和签名算法的安全性。

共识机制面临的一种攻击场景被称为多数攻击或 51% 攻击。在这种情况下，一组矿工，控制了总网络算力的大部分（例如 51%），串通一气攻击比特币。拥有挖掘大多数区块的能力，攻击者可以导致区块链中故意的“分叉”，双重支付交易，或对特定交易或地址执行拒绝服务攻击。分叉/双花攻击是指攻击者通过在之前的确认区块下方分叉并重新汇聚到备用链上，导致先前确认的区块无效。拥有足够的算力，攻击者可以使六个或更多的区块无效，导致原本被视为不可变的交易（六次确认）被取消。需要注意的是，双花只能对攻击者自己的交易进行，因为攻击者可以生成有效的签名。如果使一笔交易无效使攻击者可以获得不可逆的交易支付或产品而无需付款，那么双花自己的交易可能是有利可图的。

\ 让我们来看一个 51% 攻击的实际例子。在第一章中，我们看到了 Alice 和 Bob 之间的一笔交易。作为卖家，Bob 愿意在不等待确认（在区块中挖矿）的情况下接受付款，因为相比较迅速的客户服务而言，小额商品的双花风险较低。这类似于一些咖啡店接受小于 25 美元的信用卡支付，无需签名，因为信用卡追回的风险较低，而为了获取签名而延迟交易的成本相对较高。相比之下，以比特币出售更昂贵的商品存在双花攻击的风险，即买方广播一笔竞争交易，使用相同的输入（UTXOs），从而取消向商家的支付。51% 攻击允许攻击者在新链中双花自己的交易，从而撤销旧链中相应的交易。

在我们的例子中，恶意攻击者 Mallory 前往 Carol 的画廊，购买了一套描绘 Satoshi Nakamoto 为普罗米修斯的精美画作。Carol 以 25 万美元的比特币将画作出售给了 Mallory。Carol 在仅一次确认后，便将画作包装交给了 Mallory。Mallory 与一个名为 Paul 的共犯合作，Paul 经营着一个庞大的矿池，共犯一旦 Mallory 的交易被纳入一个区块中，就会发起攻击。Paul 指示矿池重新挖掘与包含 Mallory 交易的区块相同高度的区块，用一笔双花交易取代 Mallory 向 Carol 的支付。双花交易消耗相同的 UTXO 并将其支付给 Mallory 的钱包，而不是支付给 Carol，本质上允许 Mallory 保留比特币。然后，Paul 指示矿池挖掘额外的区块，使包含双花交易的链比原始链更长（导致在包含 Mallory 交易的区块下面出现分叉）。当区块链分叉有利于新（更长）链时，双花交易将取代对 Carol 的原始支付。Carol 现在失去了三幅画，并且没有收到支付。在所有这些活动中，Paul 的矿池参与者可能对双花尝试毫无察觉，因为他们使用自动化矿工进行挖矿，无法监控每一笔交易或区块。

为了防范这种攻击，销售价值较高的商品的商家必须在将产品交付给买家之前等待至少六次确认。等待超过六次确认有时可能是有必要的。或者，商家应该使用托管的多签名账户，在资金到位后再等待几次确认。确认次数越多，通过重新组织区块链来使交易无效的难度就越大。对于价值较高的商品，即使买家必须等待 24 小时才能交付，比特币支付仍然会很方便和高效，这大约对应着 144 次确认。

\ 除了双花攻击之外，共识攻击的另一个场景是拒绝为特定参与者（特定比特币地址）提供服务。拥有大多数挖矿算力的攻击者可以审查交易。如果它们被包含在另一个矿工挖掘的区块中，攻击者可以故意分叉并重新挖掘该区块，再次排除特定的交易。这种类型的攻击可能会导致针对特定地址或一组地址的持续拒绝服务，只要攻击者控制了大多数挖矿算力。

尽管其名称是 51% 攻击，但实际上并不需要 51% 的哈希算力。事实上，这样的攻击可以尝试用更小的哈希算力。51% 的阈值只是这种攻击几乎肯定会成功的水平。哈希算力攻击本质上是下一个区块的拉锯战，而“更强大”的一方更有可能获胜。哈希算力较少时，成功的可能性会降低，因为其他矿工通过他们的“诚实”挖矿算力生成了一些区块。从一个角度来看，攻击者拥有的哈希算力越多，他可以故意创建的分叉越长，他可以使过去更多的区块无效，或者他可以控制更多未来的区块。安全研究团队使用统计建模声称，各种类型的哈希算力攻击可能只需要 30% 的哈希算力就可以实现。

由于挖矿池引入的控制中心化，导致了挖矿池运营商进行以盈利为目的的攻击的风险。托管挖矿池中的池运营商控制候选区块的构建，并控制包含哪些交易。这使得挖矿池运营商有权排除某些交易或引入双花交易。如果这种权力滥用以一种有限且微妙的方式进行，挖矿池运营商理论上可以从哈希算力攻击中获利而不被注意到。

然而，并非所有攻击者都会受到利润的驱使。一个潜在的攻击场景是，攻击者打算在没有从这种破坏中获利的情况下破坏比特币网络。对比特币进行恶意攻击可能需要巨大的投资和隐蔽的规划，但可能会由一个资金充裕的、最有可能是由国家赞助的攻击者发起。另外，一个资金充裕的攻击者可以通过同时积累挖矿硬件、妥协挖矿池运营商，并对其他挖矿池进行拒绝服务攻击来攻击比特币。所有这些场景在理论上都是可能的。

毫无疑问，严重的哈希算力攻击会在短期内侵蚀人们对比特币的信心，可能导致价格大幅下跌。然而，比特币网络和软件不断发展，因此比特币社区将采取措施应对攻击。

改变共识规则

共识规则确定了交易和区块的有效性。这些规则是所有比特币节点之间合作的基础，并负责将所有本地视角汇聚成整个网络中的一个一致的区块链。

虽然共识规则在短期内是不变的，并且必须在所有节点之间保持一致，但在长期内它们并不是不变的。为了使比特币系统能够发展和演进，规则可以不时地发生变化，以适应新功能、改进或错误修复。然而，与传统软件开发不同，共识系统的升级要困难得多，并且需要所有参与者之间的协调。

硬分叉

在第282页的“组装和选择区块链”的部分，我们看到比特币网络可能会短暂地分歧，网络的两个部分会在短时间内跟随区块链的两个不同分支。我们看到了这个过程是如何自然地发生的，作为网络正常运行的一部分，以及在挖出一个或多个区块后，网络如何汇聚到一个共同的区块链上。

还有另一种情况会导致网络分歧并跟随两个链的情况：共识规则的变化。这种类型的分叉被称为硬分叉，因为在分叉之后，网络可能无法汇聚到一个单一的链上。相反，这两个链可以独立发展。硬分叉发生在网络的一部分根据与其余网络不同的一组共识规则运行时。这可能是因为存在错误，也可能是因为故意改变了共识规则的实现方式。

硬分叉可以用来改变共识规则，但它们需要系统中所有参与者之间的协调。任何没有升级到新共识规则的节点都无法参与共识机制，并在硬分叉时被迫进入另一个链。因此，硬分叉引入的变化可以被认为是不“向前兼容”的，因为未升级的系统由于新的共识规则而无法处理区块。

让我们通过一个具体的例子来看一下硬分叉的机制。

图12-2显示了一个具有两个分叉的区块链。在第4个区块高度处发生了一个区块的分叉。这是我们在第282页“组装和选择区块链”中看到的一种自发性分叉。随着第5个区块的挖掘，网络汇聚到一个链上，分叉被解决了。

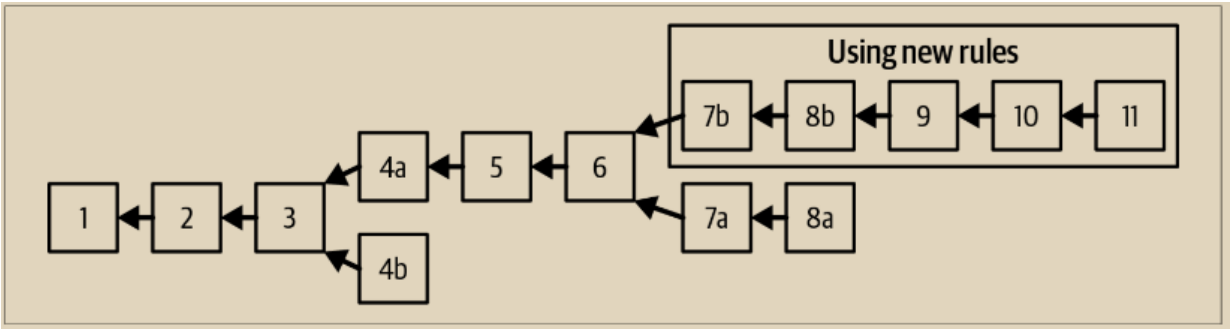


图 12-2. 一个带有分叉的区块链

然而，到了区块高度 6，一个新版本的客户端实现发布了，其中包含了对共识规则的更改。从区块高度 7 开始，运行此新版本实现的矿工将接受一种新类型的比特币；我们将其称为“foocoin”。就在这之后，运行新实现的节点创建了一个包含 foocoin 的交易，而一个使用更新软件的矿工挖出了包含此交易的 7b 区块。

任何未升级软件以验证 foocoin 的节点或矿工现在都无法处理 7b 区块。从他们的角度来看，既包含 foocoin 的交易，也包含该交易的 7b 区块都是无效的，因为他们是基于旧的共识规则进行评估的。这些节点将拒绝该交易和该区块，并且不会传播它们。使用旧规则的任何矿工都不会接受区块 7b，并将继续挖掘一个父块为区块 6 的候选区块。实际上，如果他们所连接的所有节点也遵循旧规则并因此不传播该区块，他们甚至可能都收不到区块 7b。最终，他们将能够挖掘区块 7a，这个区块在旧规则下是有效的，且不包含任何 foocoin 交易。

从这一点开始，两条链将继续分歧。在“b”链上的矿工将继续接受和挖掘包含 foocoin 的交易，而在“a”链上的矿工将继续忽略这些交易。即使区块 8b 不包含任何 foocoin 交易，使用“a”链的矿工也无法处理它。对他们来说，它似乎是一个无效的区块，因为其父区块“7b”未被识别为有效区块。

硬分叉：软件、网络、挖矿和链

对于软件开发者来说，“分叉”一词还有另一种含义，这给“硬分叉”一词增添了困惑。在开源软件中，当一组开发者选择遵循不同的软件路线图并开始竞争性地实现一个开源项目时，就会发生分叉。我们已经讨论了导致硬分叉的两种情况：共识规则中的错误和对共识规则的有意修改。在有意修改共识规则的情况下，软件分叉将先于硬分叉发生。然而，要发生这种类型的硬分叉，必须开发、采纳和启动一种新的共识规则的软件实现。

试图改变共识规则的软件分叉的例子包括比特币 XT 和比特币 Classic。然而，这两个程序都没有导致硬分叉。虽然软件分叉是一个必要的先决条件，但它本身并不足以导致硬分叉发生。要发生硬分叉，必须采纳竞争性的实现并激活新的规则，由矿工、钱包和中间节点完成。相反，比特币核心有许多替代实现，甚至软件分叉，它们不改变共识规

则，除非存在错误，否则可以在网络上共存并互操作而不会导致硬分叉。

共识规则在验证交易或区块时可能以明显和明确的方式存在差异。这些规则也可能以更微妙的方式存在差异，比如在应用于比特币脚本或数字签名等密码原语的共识规则的实现上。最后，由于系统限制或实现细节所施加的隐含共识约束，共识规则可能以意想不到的方式存在差异。一个例子是在将比特币核心 0.7 升级到 0.8 时发生的未预期的硬分叉，这是由于用于存储区块的 Berkeley DB 实现的限制引起的。

在概念上，我们可以将硬分叉分为四个阶段：软件分叉、网络分叉、挖矿分叉和链分叉。这个过程始于开发者创建了一个具有修改后共识规则的替代客户端的实现。

当这个分叉实现在网络中部署时，一定比例的矿工、钱包用户和中间节点可能会采用并运行这个实现。首先，网络将会发生分叉。基于原始共识规则实现的节点将拒绝任何根据新规则创建的交易和区块。此外，遵循原始共识规则的节点可能会与向它们发送这些无效交易和区块的任何节点断开连接。因此，网络可能会分裂为两个部分：旧节点只会保持与旧节点连接，而新节点只会与新节点连接。一个基于新规则的区块会在网络中传播并导致网络分裂为两个。新矿工可能会在新区块的基础上挖矿，而旧矿工将在旧规则的基础上挖掘另一条链。由于连接到两个独立网络，分区的网络将使得根据不同共识规则运作的矿工不太可能接收到对方的区块。

对于软件开发者来说，“分叉”一词还有另一种含义，这给“硬分叉”一词增添了困惑。在开源软件中，当一组开发者选择遵循不同的软件路线图并开始竞争性地实现一个开源项目时，就会发生分叉。我们已经讨论了导致硬分叉的两种情况：共识规则中的错误和对共识规则的有意修改。在有意修改共识规则的情况下，软件分叉将先于硬分叉发生。然而，要发生这种类型的硬分叉，必须开发、采纳和启动一种新的共识规则的软件实现。

试图改变共识规则的软件分叉的例子包括比特币 XT 和比特币 Classic。然而，这两个程序都没有导致硬分叉。虽然软件分叉是一个必要的先决条件，但它本身并不足以导致硬分叉发生。要发生硬分叉，必须采纳竞争性的实现并激活新的规则，由矿工、钱包和中间节点完成。相反，比特币核心有许多替代实现，甚至软件分叉，它们不改变共识规则，除非存在错误，否则可以在网络上共存并互操作而不会导致硬分叉。

共识规则在验证交易或区块时可能以明显和明确的方式存在差异。这些规则也可能以更微妙的方式存在差异，比如在应用于比特币脚本或数字签名等密码原语的共识规则的实现上。最后，由于系统限制或实现细节所施加的隐含共识约束，共识规则可能以意想不到的方式存在差异。一个例子是在将比特币核心 0.7 升级到 0.8 时发生的未预期的硬分叉，这是由于用于存储区块的 Berkeley DB 实现的限制引起的。

在概念上，我们可以将硬分叉分为四个阶段：软件分叉、网络分叉、挖矿分叉和链分叉。这个过程始于开发者创建了一个具有修改后共识规则的替代客户端的实现。

当这个分叉实现在网络中部署时，一定比例的矿工、钱包用户和中间节点可能会采用并运行这个实现。首先，网络将会发生分叉。基于原始共识规则实现的节点将拒绝任何根据新规则创建的交易和区块。此外，遵循原始共识规则的节点可能会与向它们发送这些无效交易和区块的任何节点断开连接。因此，网络可能会分裂为两个部分：旧节点只会保持与旧节点连接，而新节点只会与新节点连接。一个基于新规则的区块会在网络中传播并导致网络分裂为两个。

新矿工可能会在新区块的基础上挖矿，而旧矿工将在旧规则的基础上挖掘另一条链。由于连接到两个独立网络，分区的网络将使得根据不同共识规则运作的矿工不太可能接收到对方的区块。

矿工和难度的分歧

\ 随着矿工分散到挖掘两条不同的链上，算力被分配到了这两条链之间。挖矿算力可以以任意比例分配到这两条链上。新规则可能只被少数矿工遵循，也可能被绝大多数矿工遵循。

举个例子，假设出现了80% - 20%的分割，其中大部分的挖矿算力采用了新的共识规则。同时假设分叉发生在一个重新调整难度期之后。

这两条链将分别继承重新调整难度期的难度。新的共识规则将有80%的之前可用的挖矿算力加入其中。从这条链的角度来看，挖矿算力突然下降了20%，相对于之前的周期。区块平均每12.5分钟被发现一次，代表了用于扩展这条链的挖矿算力下降了20%。这种区块发行速率会持续（除非挖矿算力发生变化），直到挖出2,016个区块，这将需要大约

25,200分钟（每个区块12.5分钟），或者17.5天。之后，将进行一次重新调整，并根据这条链中挖矿算力的减少（减少了20%）来产生10分钟的区块。

少数链，使用旧规则挖矿，只拥有20%的挖矿算力，将面临更为困难的任务。在这条链上，区块的平均挖掘时间将增加到50分钟。在挖掘2,016个区块之前，难度将不会进行调整，这将需要100,800分钟，或者大约10周来挖掘完毕。假设每个区块的容量是固定的，这也将导致交易容量减少5倍，因为每小时可用于记录交易的区块数量减少了。

有争议的硬分叉

\ 这标志着去中心化共识软件开发的黎明。就像开发中的其他创新改变了软件的方法和产品，并在其后产生了新的方法、新的工具和新的社区一样，共识软件开发也代表着计算机科学的一个新领域。通过比特币开发的辩论、实验和磨难，我们将看到新的开发工具、实践、方法和社区涌现出来。

硬分叉被视为风险，因为它迫使少数人要么升级，要么留在少数链上。许多人认为将整个系统分裂为两个竞争系统的风险是不可接受的。因此，许多开发人员不愿意使用硬分叉机制来实现对共识规则的升级，除非整个网络几乎一致支持。任何没有几乎一致支持的硬分叉提案都被认为是太有争议的，不值得尝试，因为会有分裂系统的风险。

我们已经看到新的方法论出现来解决硬分叉的风险。在接下来的部分中，我们将看看软分叉以及共识修改的信号和激活方法

软分叉

\ 并非所有的共识规则更改都会引起硬分叉。只有那些不向前兼容的共识更改才会导致分叉。如果更改是以这样一种方式实现的，即未修改的客户端仍然将交易或区块视为在先前规则下有效，那么更改就可以在不进行分叉的情况下进行。

术语软分叉被引入以区分这种升级方法与“硬分叉”。实际上，软分叉根本不是一种分叉。软分叉是对共识规则的前向兼容更改，使得未升级的客户端可以继续与新规则保持一致。

软分叉的一个并非立即显而易见的方面是，软分叉升级只能用于约束共识规则，而不能用于扩展它们。为了实现向前兼容，根据新规则创建的交易和区块必须在旧规则下也是有效的，但反之则不然。新规则只能限制有效内容；否则，它们将在旧规则下被拒绝时触发硬分叉。

软分叉可以以多种方式实现——这个术语并未指定特定的方法，而是一组具有共同特点的方法：它们不要求所有节点都进行升级，也不会强制非升级节点退出共识。比特币已经实施了两个软分叉，基于对NOP操作码的重新解释。比特币脚本中保留了10个NOP操作码以备将来使用，从NOP1到NOP10。根据共识规则，脚本中存在这些操作码被解释为一个空操作符，意味着它们没有任何效果。执行在NOP操作码之后继续，就好像它不存在一样。

因此，软分叉可以修改NOP代码的语义，赋予它新的含义。例如，BIP65（CHECKLOCKTIMEVERIFY）重新解释了NOP2操作码。实现BIP65的客户端将NOP2解释为OP_CHECKLOCKTIMEVERIFY，并对包含此操作码的UTXO施加绝对锁定时间的共识规则。这个更改是一个软分叉，因为根据BIP65有效的交易也在不实现（不了解）BIP65的任何客户端上有效。对于旧的客户端来说，脚本包含一个NOP代码，而这个代码会被忽略。

软分叉的批评

\ 基于 NOP 操作码的软分叉相对来说是比较少有争议的。NOP 操作码被放置在比特币脚本中，其明确的目标是允许非破坏性的升级。

然而，许多开发者担心软分叉升级的其他方法会做出不可接受的折衷。软分叉变更的常见批评包括：

1. **技术债务**：由于软分叉比硬分叉升级更加技术复杂，它们引入了技术债务。技术债务是指由于过去的设计折衷而增加了未来代码维护成本的情况。代码复杂性又增加了出现错误和安全漏洞的可能性。
2. **验证放宽**：未修改的客户端将交易视为有效，而不评估修改后的共识规则。实际上，未修改的客户端不是使用完整范围的共识规则进行验证，因为它们对新规则视而不见。这适用于基于 NOP 的升级，以及其他软分叉升级。
3. **不可逆转的升级**：由于软分叉创建了具有额外共识约束的交易，它们在实践中变成了不可逆转的升级。如果软分叉升级在激活后被撤销，那么根据旧规则创建的任何交易都可能导致在旧规则下损失资金。例如，如果 CLTV 交易在旧规则下进行评估，则没有时间锁定约束，可以随时使用。因此，批评者认为，由于错误而必须撤销的软分叉几乎肯定会导致资金损失。

基于区块版本的软分叉信号

由于软分叉允许未修改的客户端继续在共识中运行，一种“激活”软分叉的机制是通过矿工表明他们已准备好并愿意执行新的共识规则。如果所有矿工都执行新规则，那么未升级的节点接受的块不会被升级的节点拒绝，因此不会有风险。这种机制由 BIP34 引入。

BIP34: 信号传递和激活

\ BIP34使用区块版本字段允许矿工表明他们对特定共识规则变更的准备就绪。在BIP34之前，按照共识规定，区块版本被设置为“1”，但并未被强制执行。

BIP34定义了一项共识规则变更，要求coinbase交易的coinbase字段（输入）包含区块高度。在BIP34之前，coinbase可以包含矿工选择包含的任意数据。激活BIP34后，有效的区块必须在coinbase的开头包含特定的区块高度，并且用大于或等于“2”的区块版本号进行标识。

为了表示他们准备好执行BIP34规则，矿工将区块版本设置为“2”，而不是“1”。这并不立即使版本“1”的区块无效。一旦激活，版本“1”的区块将变得无效，所有版本“2”的区块必须包含coinbase中的区块高度才能被视为有效。

BIP34定义了一个基于最近1,000个区块的滚动窗口的两步激活机制。矿工通过构造区块并将版本号设置为“2”来表示他们个人对BIP34的准备就绪。严格来说，这些区块还不必符合在coinbase交易中包含区块高度的新共识规则，因为共识规则尚未被激活。共识规则分两步激活：

- 如果最近1,000个区块中有75%（即750个）标记为版本“2”，则版本“2”区块必须在coinbase交易中包含区块高度，否则将被拒绝为无效。版本“1”区块仍然被网络接受，不需要包含区块高度。在此期间，新旧共识规则并存。
- 当95%（即最近1,000个区块中的950个）为版本“2”时，版本“1”区块不再被视为有效。只有包含coinbase中区块高度的版本“2”区块才有效（根据以前的阈值）。此后，所有区块必须符合新的共识规则，所有有效的区块必须在coinbase交易中包含区块高度。

在BIP34规则下成功进行信号传递和激活后，这一机制又被用于两次激活软分叉：

- BIP66 严格的DER签名编码通过使用区块版本“3”进行BIP34风格的信号传递和激活。
- BIP65 CHECKLOCKTIMEVERIFY通过使用区块版本“4”进行BIP34风格的信号传递和激活。

在激活BIP65之后，BIP34的信号传递和激活机制被废除，并被下文描述的BIP9信号传递机制所取代。

BIP9: 信号传递和激活

\ BIP34、BIP66和BIP65采用的机制成功激活了三次软分叉。然而，它被替换掉，因为它有几个局限性：

- 通过使用区块版本的整数值，一次只能激活一个软分叉，因此需要在软分叉提案之间进行协调，并就它们的优先级和顺序达成一致。
- 此外，由于区块版本是递增的，这种机制没有提供拒绝更改然后提出不同更改的简单方法。如果旧客户端仍在运行，它们可能会将新更改的信号误认为是先前被拒绝更改的信号。
- 每次新更改都会无法撤销地减少未来更改的可用区块版本。

为了克服这些挑战并提高未来更改的实施速度和便利性，提出了BIP9。

BIP9将区块版本解释为位字段而不是整数。由于区块版本最初是用作整数表示的，从版本1到版本4，因此只剩下29位可用于作为位字段。这意味着有29个位可用于独立且同时地对29个不同的提案进行准备就绪的信号传递。

BIP9还设置了信号传递和激活的最大时间。这样，矿工不需要永远进行信号传递。如果在超时期限（在提案中定义）内未激活提案，则将视为被拒绝。提案可以使用不同的位重新提交进行信号传递，以延长激活期限。

此外，在超时期限过去并且功能已被激活或拒绝后，信号位可以在不引起混淆的情况下用于其他功能。因此，最多可以同时传递29个更改。超时后，位可以“回收”以提出新的更改。

> 虽然信号位可以被重用或回收，只要投票期不重叠，但BIP9的作者建议只在必要时重用位；由于旧软件中的错误可能导致意外行为。简而言之，在所有29个位都被使用一次之前，我们不应该期望看到重用。

提出的更改由包含以下字段的数据结构标识：

\ 名称

用于区分提案之间的简短描述。通常是BIP描述提案，格式为“bipN”，其中N是BIP编号。

比特

0到28，矿工用于表示对此提案的批准意见的区块版本中的比特。

开始时间

信号开始的时间（基于MTP），在此之后，比特的值被解释为对提案的准备就绪进行信号传递。

结束时间

如果在这个时间（基于MTP）之后，变更尚未达到激活阈值，那么该变更将被视为被拒绝。

与BIP34不同，BIP9根据2,016个区块的难度调整周期中的整个时间间隔来计算激活信号。对于每个调整周期，如果对某个提案进行信号传递的区块总数超过95%（即2,016个中的1,916个），则该提案将在下一个调整周期后被激活。

BIP9提供了一个提案状态图，以图12-3所示的各种阶段和转换来说明提案的不同阶段和转换。

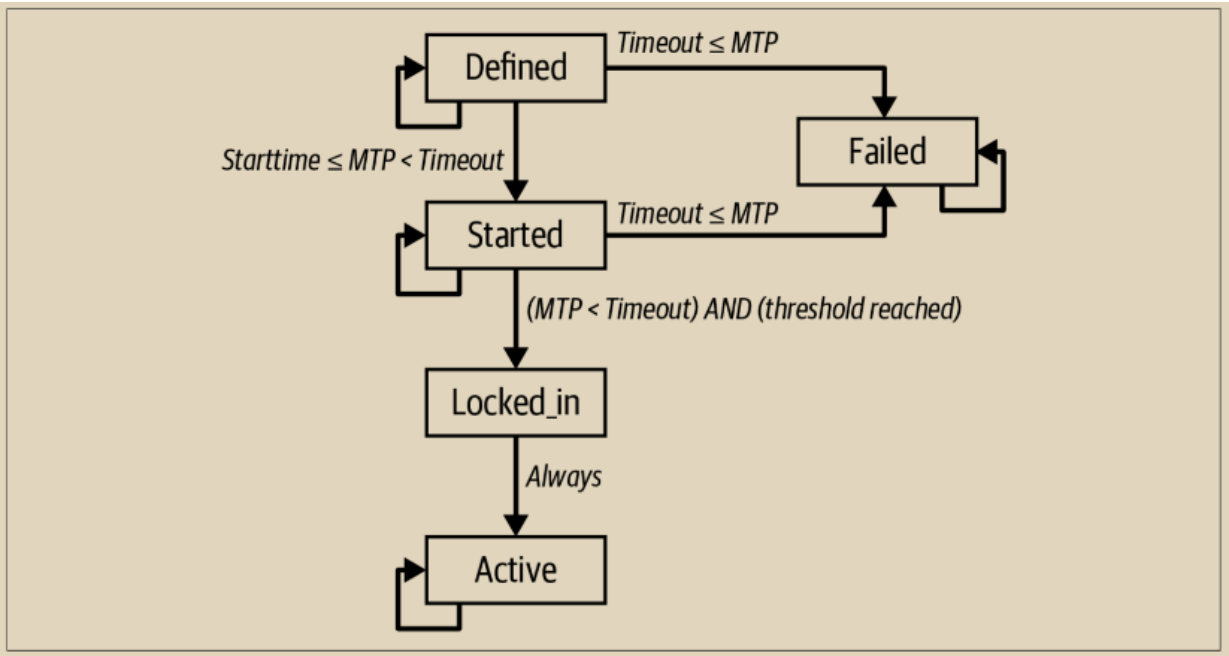


图 12-3. BIP9 状态转移图

\ 提案一旦在比特币软件中的参数已知（已定义），则会进入DEFINED状态。对于MTP在开始时间之后的区块，提案状态将转换为STARTED。如果在一个调整周期内超过了投票门槛，并且超时时间尚未到期，则提案状态将转换为LOCKED_IN。一个调整周期后，提案将变为ACTIVE。一旦提案达到该状态，将永久保持在ACTIVE状态。如果超时时间到期，而投票门槛尚未达到，则提案状态将变为FAILED，表示提案被拒绝。FAILED提案将永久保持在该状态。

BIP9首次用于激活CHECKSEQUENCEVERIFY及相关的BIPs（68、112、113）。名为“csv”的提案于2016年7月成功激活。

该标准在BIP9（带有超时和延迟的版本位）中定义。

BIP8: 强制性的提前激活期锁定

在BIP9成功用于与CSV相关的软分叉之后，下一个软分叉共识更改的实施也试图使用它进行矿工强制激活。然而，一些人反对该软分叉提案，称为segwit，很少有矿工在几个月内表明愿意执行segwit。

后来发现，一些矿工，尤其是与持不同意见者有关的矿工，可能使用了一种称为隐蔽ASICBoost的功能的硬件，使他们比使用隐蔽ASICBoost的其他矿工具有隐藏优势。无意中，segwit干扰了使用隐蔽ASICBoost的能力 - 如果激活segwit，那些使用它的矿工将失去他们的隐藏优势。

在社区发现这种利益冲突后，一些用户决定行使他们的权力，不接受矿工不遵循特定规则的区块。用户最终想要的规则是segwit添加的新规则，但用户想要通过利用计划执行segwit规则的大量节点来增加他们的努力，只要有足够的矿工表示愿意。一个化名开发者提出了BIP148，该提案要求实施它的任何节点从某个日期开始拒绝所有不为segwit进行信号的区块，并持续到segwit激活。

尽管只有少数用户实际运行了BIP148代码，但许多其他用户似乎同意这种情绪，并可能准备承诺BIP148。在BIP148即将生效几天前，几乎所有矿工开始表明他们准备好执行segwit规则。大约两周后，Segwit达到了其锁定阈值，并在两周后激活。

许多用户开始相信，BIP9的一个缺陷是矿工可以通过不发出信号一年来阻止激活尝试成功。他们希望有一种机制，可以确保软分叉在特定的区块高度激活，但也允许矿工提前发出信号以锁定它。

为此开发的方法是BIP8，它与BIP9类似，不同之处在于它定义了一个MUST_SIGNAL期间，在这个期间，矿工必须发出信号表示他们准备执行软分叉提案。

在2021年，已经发布了使用BIP8尝试激活taproot提案的软件，并且有证据表明至少有少数用户运行了该软件。其中一些用户还声称，他们愿意使用BIP8来强制矿工激活taproot是它最终激活的原因。他们声称，如果taproot没有迅速激活，其他用户也会开始运行BIP8。不幸的是，我们无法证明会发生什么，因此我们无法确定BIP8对taproot激活的贡献有多大。

速审试验(Speedy trial): 快速失败或最终成功

\ 尽管 BIP9 本身似乎没有导致 SegWit 的激活，尽管该提案得到了广泛支持，但对许多协议开发人员来说，BIP9 本身是否是失败并不清楚。正如前面提到的，矿工最初未能表达对 SegWit 的支持可能主要是由于一次性利益冲突导致的，这种情况在未来可能不再适用。对一些人来说，值得再次尝试 BIP9。另一些人则持不同意见，希望使用 BIP8。

经过数月的讨论，那些对特定激活方案最感兴趣的人之间达成了一个妥协，以激活 Taproot。提出了修改版的 BIP9，该版将只允许矿工在非常短的时间内表达他们打算执行 Taproot 规则的意图。如果信号不成功，可以使用其他激活机制（或者可能放弃这个想法）。如果信号成功，将在约六个月后的特定区块高度开始执行。这个机制由其中一位促进者命名为 “speedy trial”。

Speedy trial 激活尝试进行了，矿工迅速表示愿意执行 Taproot 规则，大约六个月后成功激活了 Taproot。对于 speedy trial 的支持者来说，这是一个明显的成功。其他人仍然对未使用 BIP8 表示失望。

目前尚不清楚 speedy trial 是否会再次用于未来尝试激活软分叉。

共识软件开发

共识软件继续演进，人们就改变共识规则的各种机制进行广泛讨论。由于比特币本身的特性，对于改变而言，它设定了极高的协调和共识标准。作为一个去中心化系统，它没有可以对网络参与者强加意愿的“权威”。权力分散在多个利益相关者之间，如矿工、协议开发者、钱包开发者、交易所、商家和最终用户。决策不能由这些利益相关者之一单方面做出。例如，虽然矿工可以通过简单多数（51%）对交易进行审查，但他们受到其他利益相关者的同意的限制。如果他们单方面行动，其他参与者可能会拒绝接受他们的区块，使经济活动继续在少数链上进行。没有经济活动（交易、商家、钱包、交易所），矿工将挖掘一种价值为空的货币，生成空块。权力的分散意味着所有参与者必须协调一致，否则就无法进行任何改变。在这个系统中，稳态是稳定的状态，只有在有非常大多数人的强烈共识下才可能进行一些少数的改变。软分叉的95%阈值反映了这一现实。

重要的是要认识到，对于共识开发来说，没有完美的解决方案。硬分叉和软分叉都涉及权衡。对于某些类型的改变，软分叉可能是更好的选择；对于其他类型的改变，硬分叉可能更合适。没有完美的选择；两者都带有风险。共识软件开发的一个不变特征是改变是困难的，共识迫使进行妥协。

一些人认为这是共识系统的弱点。随着时间的推移，你可能会认识到这是系统的最大优势。

在本书的这一部分，我们已经完成了对比特币系统本身的讨论。剩下的是建立在比特币之上的软件、工具和其他协议。

综合介绍

保护你的比特币是具有挑战性的，因为比特币不像银行账户中的余额。你的比特币非常类似于数字现金或黄金。你可能听过这样的说法：“拥有就是九分之十的法律。”嗯，在比特币中，拥有就是十分之十的法律。拥有花费某些比特币的密钥等同于拥有现金或一块贵重金属。你可以丢失它，放错地方，被盗，或者不小心把错误的金额给别人。在这些情况下，用户在协议内没有任何救济措施，就像他们在公共人行道上掉了钱一样。

然而，比特币系统具有现金、黄金和银行账户所不具备的功能。比特币钱包，包含你的密钥，可以像任何文件一样备份。它可以存储在多个副本中，甚至可以打印在纸上作为硬拷贝备份。你不能“备份”现金、黄金或银行账户。比特币与以往的任何东西都有足够的不同，我们也需要以一种新颖的方式来思考如何保护我们的比特币。

安全原则

比特币的核心原则是去中心化，这对安全性有重要影响。传统的银行或支付网络等集中化模型依赖于访问控制和审核来阻止不良行为者进入系统。相比之下，像比特币这样的去中心化系统将责任和控制权交给了用户。由于网络的安全性基于独立验证，因此比特币流量不需要加密（尽管加密仍然有用）。

在传统的支付网络（如信用卡系统）中，支付是开放式的，因为其中包含用户的私人标识符（信用卡号码）。在初始收费后，任何拥有该标识符的人都可以“拉取”资金并多次向所有者收费。因此，支付网络必须通过端到端加密进行安全保护，并确保在传输过程中或存储时（静态时）没有窃听者或中间人可以 compromise 支付流量。如果恶意行为者获得系统访问权，则他可以 compromise 当前的交易和可用于创建新交易的支付令牌。更糟糕的是，当客户数据遭到 compromise 时，客户会暴露于身份盗窃风险，并必须采取措施防止被盗账户被用于欺诈行为。

比特币与此截然不同。比特币交易仅授权特定金额给特定收款人，不能伪造。它不会透露任何私人信息，如交易各方的身份，并且不能用于授权额外的付款。因此，比特币支付网络不需要加密或受到窃听的保护。实际上，您可以通过开放的公共频道（如不安全的WiFi或蓝牙）广播比特币交易，而不会丧失安全性。

比特币的去中心化安全模型赋予用户很大的权力。随之而来的是保护他们密钥机密性的责任。对于大多数用户来说，这并不容易做到，特别是在像互联网连接的智能手机或笔记本电脑这样的通用计算设备上。尽管比特币的去中心化模型可以防止信用卡一样的大规模 compromise，但许多用户仍然无法充分保护好自己的密钥，并逐个遭受黑客攻击。

安全地开发比特币系统

比特币开发者的一个关键原则是去中心化。大多数开发者熟悉集中式安全模型，并可能会尝试将这些模型应用于他们的比特币应用程序中，结果可能会灾难性。

比特币的安全依赖于用户对密钥的去中心化控制以及用户对交易的独立验证。如果你想利用比特币的安全性，你需要确保自己在比特币的安全模型之内。简单来说：不要从用户手中夺走密钥的控制权，也不要外包验证。

例如，许多早期的比特币交易所将所有用户资金集中存放在一个单一的“热”钱包中，其密钥存储在单一服务器上。这样的设计剥夺了用户的控制权，并将密钥的控制权集中在一个系统中。许多此类系统已经遭受了黑客攻击，给他们的客户带来了灾难性的后果。

除非你准备在运营安全、多层访问控制和审计方面进行大量投资（就像传统银行那样），否则在将资金带出比特币的去中心化安全环境之前，你应该非常谨慎。即使你有足够的资金和纪律来实施强大的安全模型，这样的设计仍然只是复制了传统金融网络脆弱的模式，这些模式常常受到身份盗窃、腐败和贪污的困扰。要利用比特币独特的去中心化安全模型，你必须避免采用可能会感觉熟悉但最终会破坏比特币安全性的集中式架构的诱惑。

"信任之根"

信任之根是传统安全架构的一个概念，它是一个可信的核心，作为整个系统或应用安全的基础。安全架构围绕着信任之根构建，形成一系列同心圆，就像洋葱的层次一样，从中心向外扩展信任。每一层都基于更可信的内部层，使用访问控制、数字签名、加密和其他安全原语进行构建。随着软件系统变得越来越复杂，它们更容易包含漏洞，从而使它们容易受到安全威胁。因此，软件系统越复杂，保护它们就越困难。信任之根的概念确保了大部分信任被放置在系统中最不复杂、因此最不容易受攻击的部分，而更复杂的软件则围绕它进行分层。这种安全架构在不同的规模上重复出现，首先在单个系统的硬件中建立信任之根，然后通过操作系统将该信任之根扩展到更高级别的系统服务，最后通过许多服务器层层叠加在同心圆形的信任中。

比特币的安全架构则不同。在比特币中，共识系统创建了一个完全去中心化的受信任的区块链。一个正确验证的区块链使用创世区块作为信任之根，建立起到当前区块的信任链。比特币系统可以和应该使用区块链作为它们的信任之根。在设计一个由许多不同系统上的服务组成的复杂比特币应用时，您应该仔细检查安全架构，以确定信任被放置在哪里。最终，唯一明确可信的是一个完全验证过的区块链。如果您的应用明确或隐含地将信任置于除了区块链之外的任何东西，那应该是一个令人担忧的问题，因为这会引入漏洞。评估应用程序的安全架构的一个好方法是考虑每个单独的组件，并评估一个假设情景，即该组件完全被损害并受到恶意操作者控制。逐个考虑应用程序的每个组件，并评估如果该组件被损坏会对整体安全性产生什么影响。如果在组件被损坏时您的应用程序不再安全，那就表明您在这些组件中放置了信任。一个没有漏洞的比特币应用程序应该只对比特币共识机制的妥协具有漏洞，这意味着它的信任之根是建立在比特币安全架构的最强部分上。

大量比特币交易所被黑客攻击的例子强调了这一点，因为它们的安全架构和设计甚至在最简单的审查下也失败了。这些集中化的实施明确地将信任投资于比特币区块链之外的许多组件，例如热钱包、集中式数据库、易受攻击的加密密钥等。

用户安全最佳实践

人类几千年来一直使用物理安全控制。相比之下，我们对数字安全的经验不到50年。现代通用操作系统并不十分安全，也不特别适合存储数字货币。我们的计算机通过始终连接的互联网暴露于外部威胁。它们运行来自数百个作者的成千上万个软件组件，通常可以无限制地访问用户的文件。在你的计算机上安装的成千上万个软件中，只要有一个恶意软件，就可能 compromise 你的键盘和文件，窃取存储在钱包应用程序中的任何比特币。保持计算机免受病毒和木马的威胁所需的维护水平超出了除了极少数计算机用户以外的所有人的技能水平。

尽管经过数十年的研究和信息安全方面的进步，数字资产仍然容易受到决心坚定的对手的攻击。即使在金融服务公司、情报机构和国防承包商等最受保护和限制的系统中，也经常发生入侵事件。比特币创造了具有内在价值的数字资产，并可以即时且不可撤销地被盗和转移到新的所有者手中。这为黑客提供了巨大的激励。直到现在，黑客不得不在攻击后将身份信息或账户令牌（如信用卡和银行账户）转化为价值。尽管清洗和洗钱金融信息的难度很大，但我们看到了越来越严重的盗窃案件。比特币加剧了这个问题，因为它不需要被清洗或洗钱；比特币本身就很有价值。

比特币还创造了改进计算机安全的激励。以前，计算机受到威胁的风险模糊而间接，而比特币使这些风险变得清晰而明显。在计算机上持有比特币会引起用户对改进计算机安全的关注。由于比特币和其他数字货币的广泛传播和增加采用，我们看到了黑客技术和安全解决方案的升级。简单来说，黑客现在有了一个非常吸引人的目标，用户有明确的激励去保护自己。

在过去的三年里，直接由比特币的采用带来的是信息安全领域的巨大创新，包括硬件加密、密钥存储和硬件签名设备、多重签名技术和数字托管等。在接下来的部分中，我们将探讨实用用户安全的各种最佳实践。

比特币的物理存储

因为大多数用户对于物理安全比信息安全更为熟悉，所以保护比特币的一种非常有效的方法是将其转换为物理形式。比特币的私钥和用于生成它们的种子只是长数字。这意味着它们可以以物理形式存储，比如打印在纸上或刻在金属板上。然后，保护这些密钥就变得简单，只需物理安全地保管打印的密钥种子副本。打印在纸上的种子称为“纸质备份”，许多钱包都可以创建它们。将比特币保持离线称为冷存储，这是最有效的安全技术之一。冷存储系统是指在离线系统上生成密钥（从不连接到互联网的系统），并将其离线存储，可以是在纸上或数字媒体上，比如USB存储设备。

硬件签名设备

从长远来看，比特币安全可能越来越多地采用防篡改的硬件签名设备形式。与智能手机或台式电脑不同，比特币硬件签名设备只需要保存密钥并使用它们生成签名。由于没有通用软件可以受到威胁，且接口有限，硬件签名设备可以为非专家用户提供强大的安全性。硬件签名设备可能成为存储比特币的主要方法。

确保您的访问权限

尽管大多数用户正确地担心比特币被盗的风险，但有一个更大的风险。数据文件经常丢失。如果其中包含比特币密钥，损失就会更加痛苦。在努力保护他们的比特币钱包时，用户必须非常小心，不要走得太远，以至于最终丢失他们的比特币。2011年7月，一个著名的比特币意识和教育项目丢失了近7000个比特币。在防止盗窃的努力中，所有者实施了一系列复杂的加密备份。最终，他们意外丢失了加密密钥，使备份变得毫无价值，损失了一大笔财富。就像将钱藏在沙漠里一样，如果你把比特币保护得太好，可能再也找不到它们了。

要花费比特币，你可能需要备份的不仅仅是你的私钥或用于派生它们的BIP32种子。当使用多重签名或复杂的脚本时，情况尤其如此。大多数输出脚本都承诺了实际的条件，必须满足这些条件才能花费该输出中的比特币，而除非你的钱包软件能够向网络公开这些条件，否则无法满足该承诺。钱包恢复码必须包括这些信息。有关更多详情，请参阅第五章

分散风险

你会把自己的全部净值都放在钱包里的现金吗？大多数人会认为那样很鲁莽，然而比特币用户经常将所有比特币都存放在一个单一的钱包应用中。相反，用户应该将风险分散到多个不同的比特币应用程序中。审慎的用户只会将很小一部分，可能不到5%的比特币存放在在线或移动钱包中作为“零钱”。其余的应该分散在几种不同的存储机制中，例如桌面钱包和离线（冷存储）。

多签和治理

无论是公司还是个人存储大量比特币时，都应考虑使用多重签名比特币地址。多重签名地址通过要求超过一个签名来保护资金。签名密钥应该存储在多个不同的位置，并由不同的人控制。例如，在企业环境中，密钥应该是独立生成的，并由几位公司高管持有，以确保没有单个人可以危害资金。多重签名地址还可以提供冗余性，其中一个人持有存储在不同位置的多个密钥。

生存能力

\ 一个经常被忽视的重要安全考虑是可用性，特别是在密钥持有者无法使用或去世的情况下。比特币用户被告知要使用复杂的密码，并保持其密钥安全和私密，不与任何人分享。不幸的是，这种做法几乎让用户的家人无法恢复任何资金，如果用户无法使用时。实际上，在大多数情况下，比特币用户的家人可能完全不知道比特币资金的存在。

如果你拥有大量的比特币，你应该考虑与信任的亲戚或律师分享访问详细信息。更复杂的生存能力方案可以通过多重签名访问和通过专业的“数字资产执行者”律师进行财产规划来设置。

比特币是一种复杂的新技术，仍在被开发者探索中。随着时间的推移，我们将开发出更好的安全工具和实践，以便非专家使用。就目前而言，比特币用户可以使用本文讨论的许多技巧来享受安全、无忧的比特币体验。

综合介绍

让我们现在在我们对比特币主要系统（一层）的理解基础上，将其视为其他应用程序或二层的平台。在本章中，我们将探讨比特币作为应用平台提供的功能。我们将考虑应用程序构建的基本元素，这些元素构成了任何区块链应用程序的构建块。我们将探讨使用这些基本元素的几个重要应用，例如客户端验证、支付通道和路由支付通道（闪电网络）。

构建块（原语）

当比特币系统在长期运行并且正常操作时，它提供了一些保证，这些可以作为构建应用程序的基础。这些保证包括：

1. 无双重支付

比特币的去中心化共识算法的最基本保证确保在同一有效的区块链中，任何未花费交易输出（UTXO）不会被两次花费。

1. 不可变性

一旦交易记录在区块链中，并且随后的区块添加了足够的工作量，该交易的数据就变得几乎不可改变。不可变性是由能量支持的，因为重写区块链需要耗费能量来进行工作量证明（PoW）。所需的能量和因此不可变性的程度会随着在包含交易的区块之上进行的工作量的增加而增加。

1. 中立性

去中心化的比特币网络传播有效的交易，无论这些交易的来源是什么。这意味着任何人都可以创建一个有效的交易，并相信他们可以随时传输该交易并使其包含在区块链中。

1. 安全时间戳

共识规则拒绝任何时间戳过于未来的区块，并试图防止时间戳过于过去的区块。这确保了区块的时间戳在一定程度上是可信的。区块上的时间戳意味着所有包含交易的输入在未花费之前的参考。

1. 授权

在去中心化网络中验证的数字签名提供了授权保证。包含对数字签名的要求的脚本在没有脚本中隐含的私钥持有者的授权的情况下无法执行。

1. 可审计性

所有交易都是公开的，并且可以进行审计。所有交易和区块都可以追溯到创世区块，形成一个完整的链。

1. 会计

在任何交易（除了coinbase交易）中，输入的价值等于输出的价值加上费用。在交易中不能创建或销毁比特币价值。输出不能超过输入。

1. 不会过期

有效的交易不会过期。如果它今天有效，只要输入保持未花费且共识规则不变，它将在不久的将来仍然有效。

1. 完整性

使用SIGHASH_ALL签名的比特币交易的输出或由其他SIGHASH类型签名的交易的部分不能修改，否则将使签名无效，从而使交易本身无效。

1. 交易原子性

比特币交易是原子性的。它们要么有效和确认（已被挖掘），要么无效。部分交易不能被挖掘，并且没有交易的中间状态。在任何时间点，交易要么已被挖掘，要么未被挖掘。

1. 不可分割的价值单位

交易输出是离散且不可分割的价值单位。它们可以完全花费或未花费。它们不能被分割或部分花费。

1. 控制的法定人数

脚本中的多签名约束要求在多签名方案中预先定义的法定人数进行授权。该要求由共识规则执行。

1. 时锁/老化

任何包含相对或绝对时锁的脚本子句只能在其年龄超过指定时间后执行。

1. 复制

区块链的去中心化存储确保一旦交易被挖掘，在足够的确认之后，它将在网络中复制，并且变得耐用且对电源损失、数据损失等具有抵抗能力。

1. 防伪保护

交易只能花费现有的、经过验证的输出。不能创建或伪造价值。

1. 一致性

在没有矿工分区的情况下，记录在区块链中的区块会根据它们记录的深度以指数递减的可能性进行重组或不一致。一旦深度记录，改变所需的计算和能量使得改变实际上是不可行的。

1. 记录外部状态

交易可以通过OP_RETURN或支付给合约来提交数据值，表示外部状态机中的状态转换。

1. 可预测的发行

不到2100万比特币将以可预测的速度发行。

构建块的列表并不完整，每次引入新功能时都会添加更多的构建块。

基于构建块的应用程序

\ 基于比特币提供的构建块，可以用来构建应用程序的信任平台元素。以下是一些现有应用程序及其使用的构建块的示例：

存在证明（数字公证）

不变性 + 时间戳 + 持久性。区块链上的交易可以承诺一个数值，证明某个数据在记录时存在（时间戳）。该承诺无法事后修改（不变性），并且证据将被永久存储（持久性）。

众筹（Lighthouse）

一致性 + 原子性 + 完整性。如果您签署了一个筹款交易的一个输入和输出（完整性），其他人可以为筹款做出贡献，但在达到目标（输出金额）之前无法支出（原子性）（一致性）。

支付通道

控制多重签名 + 时间锁定 + 无双花 + 不过期 + 抗审查 + 授权。一个多重签名的2-of-2（控制多重签名），带有时间锁定的（时间锁定）作为支付通道的“结算”交易可以在任何时间由任一方（授权）持有（不过期）和支出（抗审查）。然后，两个当事人可以创建超过（无双花）较短时间锁定的结算的承诺交易（时间锁定）。

彩色币

\ 彩色代币是一组类似的技术，利用比特币交易记录除比特币以外的外部资产的创建、所有权和转移。这里所说的“外部资产”是指不直接存储在比特币区块链上的资产，与比特币本身相对，后者是区块链内在的资产。

彩色代币用于跟踪数字资产，以及由第三方持有并通过彩色代币关联的所有权证书进行交易的物理资产。数字资产的彩色代币可以代表无形资产，如股票证书、许可证、虚拟物品（游戏道具），或几乎任何形式的受许可知识产权（商标、版权等）。有形资产的彩色代币可以代表大宗商品（黄金、白银、石油）、土地所有权证书、汽车、船只、飞机等。

这个术语来源于将一个名义上的比特币金额（例如，一个 satoshi）“着色”或标记为代表比特币金额本身以外的东西。类比一下，想象一下在一张 1 美元纸币上盖章，写着“这是 ACME 公司的股票证书”或“这张纸币可以兑换 1 盎司银”并将这张 1 美元纸币作为另一种资产的所有权证书进行交易。彩色代币的第一个实现称为增强填充顺序彩色或 EPOBC，它将外部资产分配给 1 satoshi 输出。这样，它是一个真正的“彩色代币”，因为每个资产都被添加为单个 satoshi 的属性（颜色）。最新的彩色代币实现使用其他机制来附加元数据到交易中，与外部数据存储结合，将元数据关联到特定资产。截至本文写作时，主要使用的三种机制是单次使用密封、支付到合约和客户端验证

一次性密封

\ "Single-use seals" 源自物理安全领域。某人通过第三方寄送物品时，需要一种方式来检测是否被篡改，因此他们会使用一种特殊的机制来保护包裹，如果包裹被打开，这种机制就会被明显损坏。如果包裹到达时密封完好，发件人和收件人就可以放心包裹在运输过程中未被打开。

在彩硬币的背景下，一次性密封是指一种数据结构，只能与另一种数据结构关联一次。在比特币中，未花费的交易输出（UTXO）满足这种定义。在有效的区块链中，UTXO只能被花费一次，并且花费它们的过程将它们与花费交易中的数据关联起来。

这为现代彩色币的转移提供了部分基础。一个或多个彩色币被接收到一个UTXO中。当该UTXO被花费时，花费交易必须描述彩色币如何被使用。这就引出了“支付给合约”（Pay to Contract，P2C）。

“支付给合约”（Pay to Contract，P2C）

我们之前在第176页的“支付到合约（P2C）”中学习了有关P2C的知识，它成为比特币共识规则的Taproot升级的基础之一。简而言之，P2C允许支出者（Bob）和接收者（Alice）就某些数据达成一致，例如合约，然后调整Alice的公钥，使其承诺该合约。随时，Bob都可以揭示Alice的基础密钥和用于承诺合约的调整，证明她收到了资金。如果Alice花费了资金，这完全证明了她知道该合约，因为她只能通过知道调整（即合约）来花费收到的资金到P2C调整过的密钥。

P2C调整密钥的一个强大特性是，除了Alice和Bob之外的所有人看起来都像是其他公钥，除非他们选择透露用于调整密钥的合约。关于合约的任何信息都不会公开，甚至不会透露合约的存在。

P2C合约可以是任意长且详细的，条款可以用任何语言编写，可以引用参与者想要的任何内容，因为合约不会被全节点验证，只有包含承诺的公钥被发布到区块链。

在有色硬币的背景下，Bob可以通过花费相关的UTXO来打开含有他的有色硬币的一次性密封。在花费该UTXO的交易中，他可以承诺一个合约，指示下一个所有者（或所有者）必须遵守的条款，以便进一步花费硬币。新所有者不一定是Alice，即使Alice是Bob花费的UTXO的接收者，而Alice也已经根据合约条款调整了她的公钥。

因为全节点不会（也不能）验证合约是否正确执行，我们需要弄清楚谁负责验证。这就引出了客户端验证。

客户端验证

Bob拥有与一个UTXO相关联的彩色币。他以一种方式花费了该UTXO，这种方式承诺了一项合约，该合约指示了彩色币的下一个接收者（或接收者们）将如何证明他们对这些硬币的所有权以便进一步花费它们。

实际上，Bob的P2C合约可能简单地承诺了一个或多个唯一的标识符，用于决定何时再次花费这些彩色币的UTXO，作为单次使用密封。例如，Bob的合约可能表明，Alice接收到的UTXO现在控制他一半的彩色币，而他另一半的彩色币现在分配给了与Alice和Bob的交易无关的另一个UTXO。这在阻碍区块链监视方面提供了显著的隐私保护。

当Alice想要将她的彩色币花费给Dan时，她首先需要向Dan证明自己控制着这些彩色币。Alice可以通过向Dan揭示她的P2C基本公钥和Bob选择的P2C合约条款来实现这一点。Alice还向Dan揭示了Bob用作单次使用密封的UTXO以及Bob给她关于彩色币的前任所有者的任何信息。简而言之，Alice向Dan提供了有关每笔彩色币先前转移的完整历史记录，每个步骤都与比特币区块链（但不存储任何特殊数据在链上——只是常规公钥）相关联。这个历史记录很像我们称之为区块链的常规比特币交易的历史记录，但彩色历史对于区块链的其他用户完全不可见。

Dan使用他的软件进行此历史记录的验证，称为客户端验证。值得注意的是，Dan只需要接收和验证与他想要接收的彩色币相关的历史记录部分。他不需要了解其他人的彩色币发生了什么情况——例如，他永远不需要知道Bob的另一半硬币发生了什么，那些Bob没有转给Alice的硬币。这有助于增强彩色币协议的隐私性。

现在我们已经了解了单次使用密封、支付到合约和客户端验证，我们可以看一下截至目前正在使用它们的两个主要协议，即RGB和Taproot资产。

RGB

\ RGB协议的开发者开创了现代基于比特币的彩色币协议中使用的许多理念。RGB设计的主要要求之一是使协议与链下支付通道兼容（参见第318页的“支付通道和状态通道”），例如闪电网络（LN）中使用的通道。RGB协议的每一层都实现了这一点：

一次性密封 为了创建支付通道，Bob将他的彩色币分配给一个需要他和Alice双方签名才能花费的UTXO。他们对该UTXO的相互控制用作未来转移的一次性密封。

付款合约（P2C） Alice和Bob现在可以签署多个版本的P2C合约。底层支付通道的执行机制确保双方有动机只在链上发布最新版本的合约。

客户端验证 为了确保Alice和Bob都不需要相互信任，他们各自检查了所有彩色币的历史转移，直到其创建，以确保所有合约规则都被正确遵循。

RGB的开发人员已经描述了他们的协议的其他用途，例如创建可以定期更新以防止私钥泄露的身份令牌。

欲了解更多信息，请参阅RGB的文档。

Taproot资产

\ Taproot 资产原名 Taro，是一种受 RGB 强烈影响的彩色币协议。与 RGB 相比，Taproot 资产使用的一种 P2C 合约形式与 Taproot 用于启用 MAST 功能的版本非常相似（请参阅“Merklized Alternative Script Trees (MAST)”第172页）。Taproot 资产相比 RGB 的优势在于，它与广泛使用的 Taproot 协议的相似性使得钱包和其他软件更容易实现。但其缺点是，在实现非资产功能，如身份令牌时，可能没有 RGB 协议灵活。

> Taproot 是比特币协议的一部分，而 Taproot 资产却不是，尽管名字相似。RGB 和 Taproot 资产都是建立在比特币协议之上的协议。比特币原生支持的唯一资产是比特币。

与 RGB 相比，Taproot 资产更加注重与 LN 的兼容性。在 LN 上转发非比特币资产的一个挑战是，有两种不同的方式来进行发送，每种方式都有一套不同的权衡：

\ *本地转发*

在发送方和接收方之间的每一跳都必须知道特定资产（彩色币类型）并具有足够的余额来支持转发付款。

转换转发

发送方旁边的跳点和接收方旁边的跳点必须知道特定资产并具有足够的余额来支持向前转发付款，但是其他每一跳只需要支持比特币支付的转发。

本地转发在概念上更简单，但基本上需要为每种资产建立一个独立的类似 Lightning 的网络。转换转发允许利用比特币 LN 的规模经济，但可能会受到一个称为自由美式看涨期权的问题的影响，即接收方可能根据最近的汇率变化选择性地接受或拒绝某些支付，以从其旁边的跳点中汲取资金。尽管没有已知的完美解决方案来解决自由美式看涨期权问题，但可能存在限制其危害的实际解决方案。

Taproot 资产和 RGB 理论上都可以支持本地和转换转发。Taproot 资产专门设计用于转换转发，而 RGB 则已经提出了实现两种方式的建议。

有关更多信息，请参阅 Taproot 资产的文档。此外，Taproot 资产的开发者正在研究一些 BIP，这些 BIP 可能在本书印刷后可用。

\

支付通道和状态通道

支付通道是一种在比特币区块链之外交换比特币交易的无信任机制。这些交易，如果在比特币区块链上结算，是有效的，但实际上它们被保留在链外，等待最终的批量结算。由于这些交易没有被结算，它们可以进行交换而不受通常的结算延迟的影响，从而实现了极高的交易吞吐量、低延迟和细粒度的交易。

实际上，“通道”这个术语是一个隐喻。状态通道是在两个参与者之间交换状态的虚拟构造，位于区块链之外。实际上并没有“通道”，底层的数据传输机制也不是通道。我们使用“通道”这个术语来表示两个参与者之间的关系和共享状态，而这些状态是位于区块链之外的。

为了进一步解释这个概念，可以将其类比为 TCP 流。从更高层次的协议角度来看，它是连接互联网上两个应用程序的“套接字”。但是如果观察网络流量，TCP 流实际上只是 IP 数据包上的一条虚拟通道。每个 TCP 流的端点对 IP 数据包进行序列化和组装，以创建字节流的幻象。在底层，它实际上是由断开的数据包组成的。

类似地，支付通道只是一系列交易。如果这些交易被正确地排序和连接，它们将创建可信的可赎回义务，尽管你不信任通道的另一侧。

在本节中，我们将研究各种形式的支付通道。首先，我们将研究用于构建单向（单向）支付通道的机制，用于按流量计费的微支付服务，例如视频流。然后，我们将扩展这个机制，介绍双向支付通道。最后，我们将探讨如何将双向通道端对端连接，形成多跳通道在一个路由网络中，最初是在闪电网络的名称下提出的。

支付通道是状态通道概念的一部分，它表示在区块链之外通过最终结算在区块链上保证的状态的更改。支付通道是一个状态通道，其中被更改的状态是虚拟货币的余额。

状态通道 - 基本概念和术语

\ 状态通道是通过在区块链上锁定共享状态的交易来建立的，这称为资金交易。这笔单一交易必须被传输到网络并被挖掘以建立通道。在支付通道的例子中，锁定的状态是通道的初始余额（以货币计）。

然后，两个参与方交换已签名的交易，称为承诺交易，以修改初始状态。这些交易是有效的交易，因为它们可以由任一方提交进行结算，但是实际上被每个参与方持有，直到通道关闭。状态更新可以根据每个参与方创建、签名和传输交易的速度来创建，实际上这意味着可以交换数十个每秒。

在交换承诺交易时，两个参与方还会阻止使用先前的状态，以便始终使用最新的承诺交易进行赎回。这样做可以防止任一方通过单方面关闭通道并使用比当前状态更有利的先前状态来作弊。在本章的其余部分，我们将检查可以用来阻止发布先前状态的各种机制。

最后，通道可以通过合作方式关闭，通过提交最终的结算交易到区块链上，或者通过单方面关闭，通过任一方提交最后的承诺交易到区块链上。需要单方面关闭选项以防止其中一方意外断开连接。结算交易代表通道的最终状态，并在区块链上进行结算。

在通道的整个生命周期中，只需要提交两笔交易进行区块链上的挖掘：资金和结算交易。在这两个状态之间，两个参与方可以交换任意数量的承诺交易，这些交易对其他人不可见，也不会提交到区块链上。

图14-1说明了Bob和Alice之间的支付通道，显示了资金、承诺和结算交易。

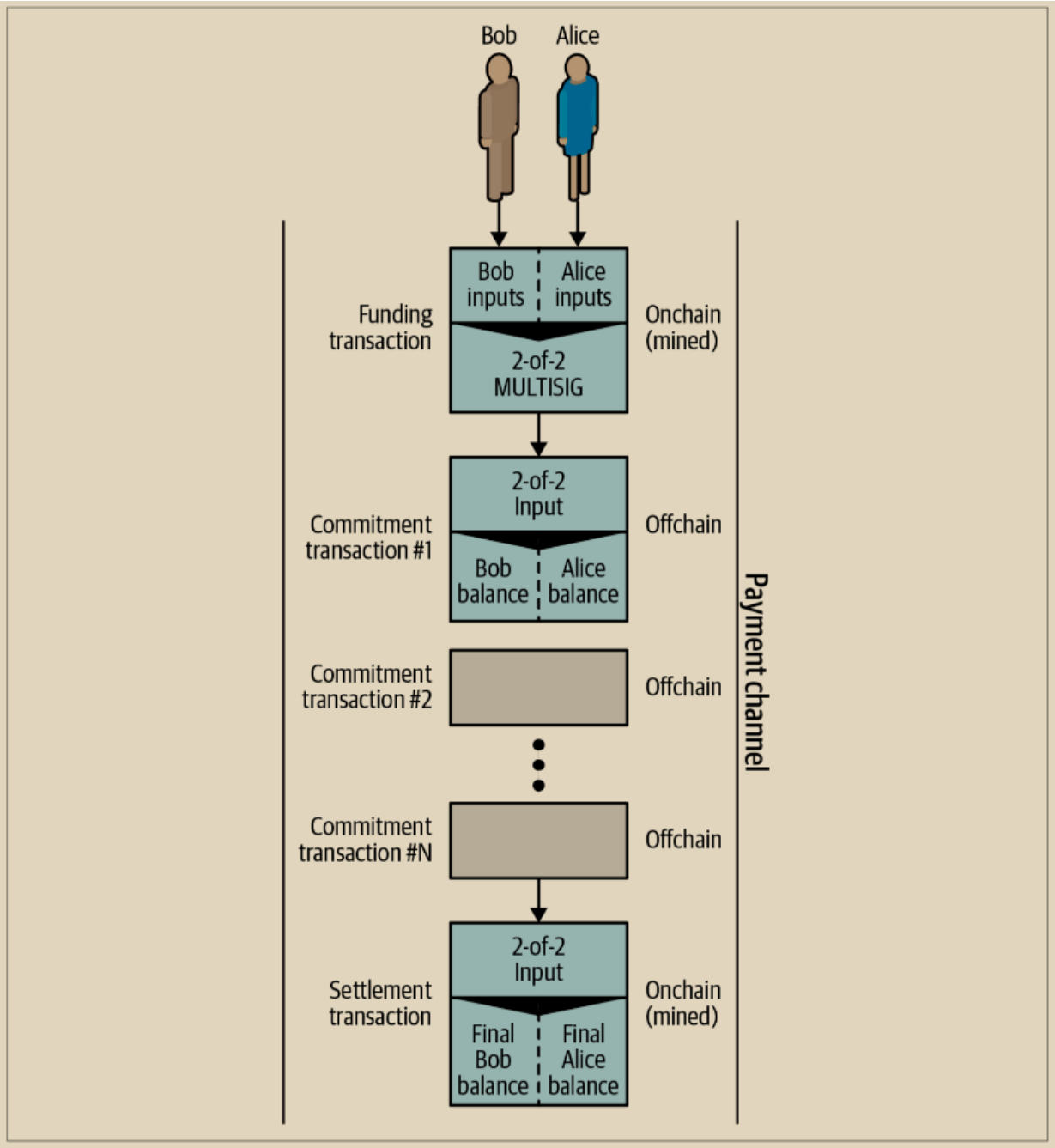


图 14-1. Bob 和 Alice 之间的支付通道，显示了资金、承诺和结算交易

简单支付通道示例

为了解释状态通道，我们从一个非常简单的例子开始。我们展示一个单向通道，意味着价值只在一个方向上流动。我们还将从一个天真的假设开始，即没有人试图作弊，以保持简单。一旦我们解释了基本的通道概念，我们将看看如何使它变得无需信任，以便双方都无法作弊，即使他们试图这样做。

在这个例子中，我们假设有两个参与者：Emma 和 Fabian。Fabian 提供按秒计费的视频流服务，使用微支付通道。Fabian 每秒视频收取 0.01 毫比特（0.00001 BTC），相当于每小时视频收取 36 毫比特（0.036 BTC）。Emma 是一位购买 Fabian 的视频流服务的用户。图 14-2 显示了 Emma 使用支付通道从 Fabian 购买视频流服务。

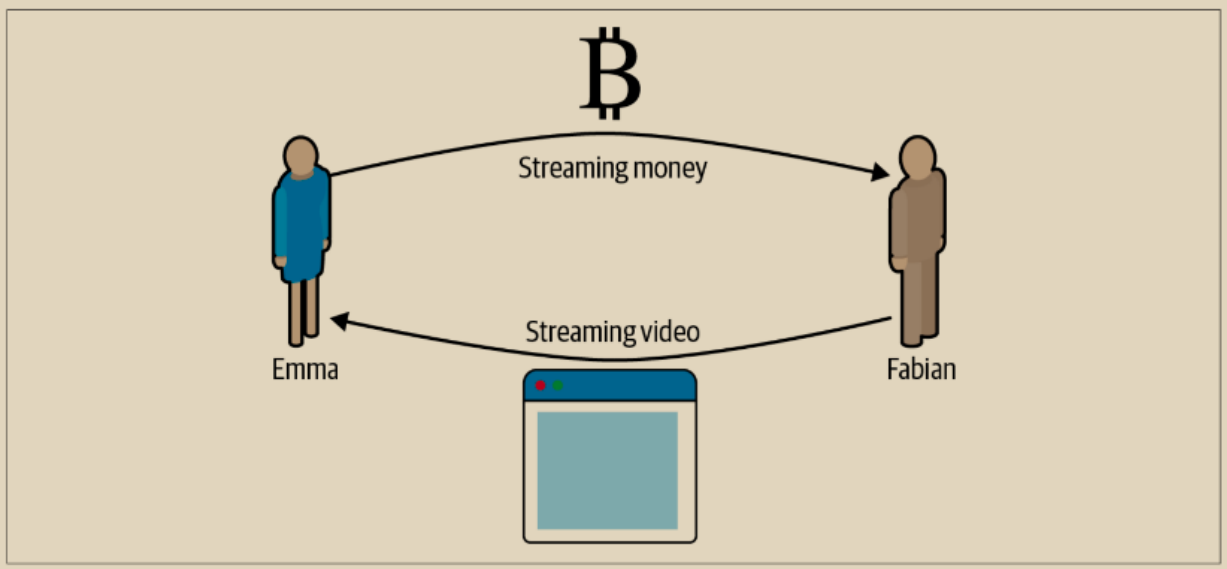


图 14-2. Emma 使用支付通道购买 Fabian 的视频流服务，按每秒的视频付费

\ 在这个例子中，Fabian 和 Emma 使用的是能够处理支付通道和视频流的特殊软件。Emma 在她的浏览器中运行该软件，而 Fabian 则在服务器上运行。这个软件包括基本的比特币钱包功能，可以创建和签署比特币交易。用户完全不知道“支付通道”这个概念和术语，他们看到的是按秒付费的视频。

为了建立支付通道，Emma 和 Fabian 建立了一个 2-of-2 的多签名地址，他们各自持有其中一个密钥。从 Emma 的角度来看，她浏览器中的软件会显示一个带有地址的二维码，并要求她提交支付金额，以便观看多长时间的视频。然后，Emma 把这个地址进行了资金注入。Emma 支付到多签名地址的交易是支付通道的资金或锚定交易。

在这个例子中，假设 Emma 用 36 毫比特（0.036 BTC）资助了该通道。这将允许 Emma 观看多达 1 小时的流媒体视频。在这种情况下，资金交易设置了该通道中可传输的最大金额，即通道容量。

资金交易从 Emma 的钱包中消耗一个或多个输入来获取资金。它创建一个输出，金额为 36 毫比特，支付到由 Emma 和 Fabian 共同控制的多签名 2-of-2 地址。它可能还有其他输出，作为找零返回给 Emma 的钱包。

资金交易确认到足够的深度后，Emma 可以开始观看视频。Emma 的软件创建并签署一个承诺交易，将通道余额更改为向 Fabian 的地址支付 0.01 毫比特，同时将 35.99 毫比特退还给 Emma。Emma 签署的交易消耗了资金交易创建的 36 毫比特输出，并创建了两个输出：一个是她的退款，另一个是 Fabian 的支付。这笔交易只部分签署 - 它需要两个签名（2-of-2），但只有 Emma 的签名。当 Fabian 的服务器接收到这笔交易时，它会添加第二个签名（对于 2-of-2 输入），然后将其与 1 秒钟的视频一起返回给 Emma。现在，双方都有了一个完全签署的承诺交易，可以兑现，代表了通道的正确最新余额。双方都不会将此交易广播到网络中。

在下一轮中，Emma 的软件创建并签署另一个承诺交易（承诺 #2），该交易使用来自资金交易的相同 2-of-2 输出。第二次承诺交易将一个输出的 0.02 毫比特分配给 Fabian 的地址，另一个输出的 35.98 毫比特返回给 Emma 的地址。这笔新交易是两秒视频的支付。Fabian 的软件签署并返回第二次承诺交易，再加上另一秒的视频。

通过这种方式，Emma 的软件继续向 Fabian 的服务器发送承诺交易，以换取视频流。通道的余额逐渐积累到 Fabian 的利益，因为 Emma 消耗了更多的视频秒数。假设 Emma 观看了 600 秒（10 分钟）的视频，创建并签署了 600 笔承诺交易。最后一笔承诺交易（#600）将两个输出，将通道的余额分成 6 毫比特给 Fabian，30 毫比特给 Emma。

最后，Emma 点击“停止”以停止观看视频。现在 Fabian 或 Emma 可以传输最终状态交易以进行结算。这最后一笔交易是结算交易，向 Fabian 支付 Emma 消耗的所有视频，将剩余的资金交易退还给 Emma。

图 14-3 展示了 Emma 和 Fabian 之间的通道以及更新通道余额的承诺交易。

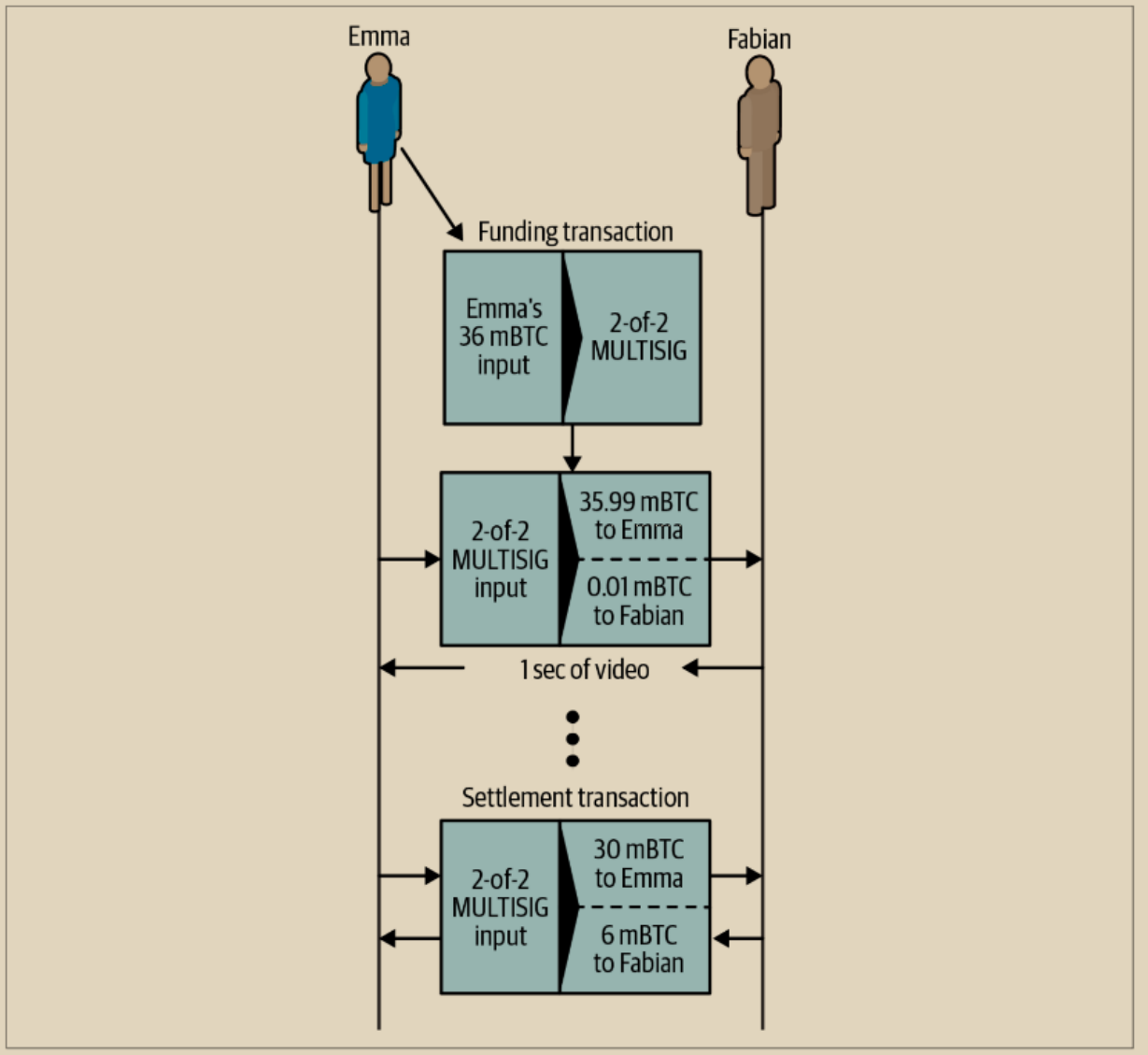


图 14-3. Emma与Fabian之间的支付通道，显示更新通道余额的承诺交易

创建无需信任的通道

刚刚描述的通道只有在双方合作、没有任何故障或试图欺骗的情况下才能运作。让我们看看一些破坏这一通道的情况，并看看如何修复：

• 一旦发生资金交易，Emma 需要 Fabian 的签名才能拿回任何资金。如果 Fabian 消失，Emma 的资金将被锁定在 2-of-2 中，并且实际上丢失了。如果在至少有一份双方签署的承诺交易之前，其中一方变得不可用，那么按照当前构建的通道，资金将会丢失。

• 在通道运行期间，Emma 可以取任何 Fabian 已经签署的承诺交易中的一份并将其传输到区块链。如果她可以传输承诺交易 # 1 并仅支付 1 秒的视频，为什么要支付 600 秒的视频呢？Emma 可以通过广播对她有利的先前承诺来欺骗，这导致通道失败。

这两个问题都可以通过时间锁解决——让我们看看如何使用交易级时间锁。

Emma 不能冒险进行 2-of-2 多重签名资金，除非她有一种保证的退款。为了解决这个问题，Emma 同时构建了资金和退款交易。她签署了资金交易，但不将其传输给任何人。Emma 仅将退款交易传输给 Fabian 并获得他的签名。

退款交易充当了第一笔承诺交易，其时间锁确定了通道的生命周期上限。在这种情况下，Emma 可以将锁定时间设置为 30 天或未来 4,320 个区块。在之后的所有承诺交易必须有一个更短的时间锁，以便在退款交易到期之前赎回。

现在，Emma 有了一个完全签名的退款交易，她可以放心地传输已签名的资金交易，知道即使 Fabian 消失，她也可以在时间锁到期后赎回退款交易。

在通道的生命周期内，双方交换的每笔承诺交易都将被时间锁定到未来。但是，对于每个承诺，延迟都会略微缩短，以便最新的承诺在使其无效的先前承诺之前可以被赎回。由于时间锁，没有一方能够成功传播任何承诺交易，直到其时间锁到期。如果一切顺利，他们将合作并以和解交易优雅地关闭通道，无需传输中间承诺交易。如果没有，最新的承诺交易可以传播以结算账户并使所有先前的承诺交易无效。

例如，如果承诺交易 # 1 被时间锁定到未来的 4,320 个区块，那么承诺交易 # 2 将被时间锁定到未来的 4,319 个区块。承诺交易 # 600 可以在承诺交易 # 1 生效之前的 600 个区块被消费。

图 14-4. 显示了每个承诺交易设置了一个较短的时间锁，使其可以在先前的承诺变为有效之前被消费

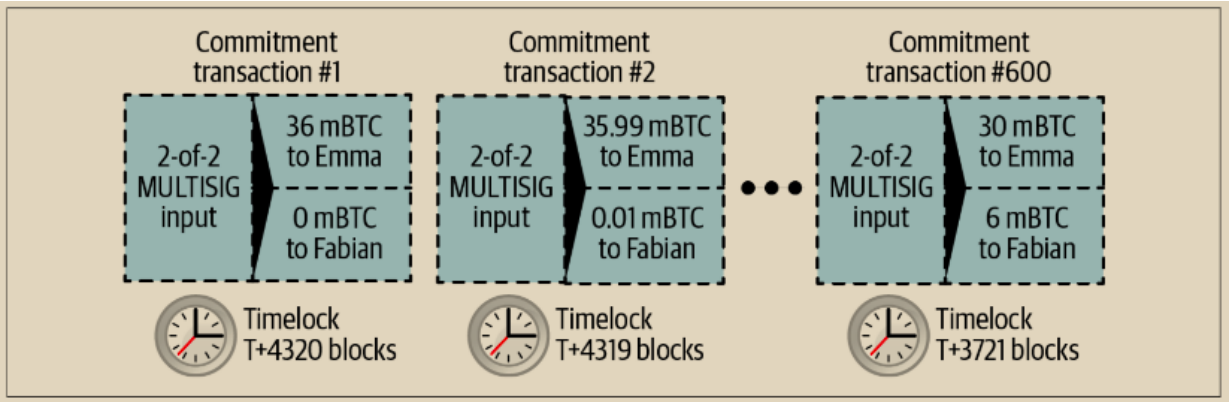


图 14-4 . 每个承诺交易都设置了较短的时间锁，使其可以在先前的承诺变为有效之前被消费

每个后续的承诺交易都必须设置更短的时间锁，以便在其前身和退款交易之前进行广播。提前广播承诺交易的能力确保它能够支出资金输出，并阻止其他任何承诺交易通过支出输出来赎回。比特币区块链提供的保证，防止双重支付并强制执行时间锁，有效地使每个承诺交易都能使其前身无效。

状态通道使用时间锁来跨时间维度执行智能合约。在本示例中，我们看到时间维度保证了最近的承诺交易在任何早期承诺之前生效。因此，最新的承诺交易可以被传输，支出输入并使先前的承诺交易无效。使用绝对时间锁来执行智能合约可以防止其中一方作弊。这种实现除了绝对事务级别的锁定时间外，不需要任何其他内容。接下来，我们将看到如何使用脚本级别的时间锁，CHECKLOCKTIMEVERIFY 和 CHECKSEQUENCEVERIFY，来构建更灵活、有用和复杂的状态通道。

时间锁不是使前期承诺交易无效的唯一方法。在接下来的章节中，我们将看到如何使用吊销密钥来实现相同的结果。时间锁是有效的，但它们有两个明显的缺点。首先，在通道首次打开时，通过建立最大时间锁，它们限制了通道的寿命。更糟糕的是，它们迫使通道实现在允许长期存在的通道和迫使其中一方在提前关闭时等待很长时间以获得退款之间取得平衡。例如，如果通过将退款时间锁设置为30天来允许通道保持开放30天，则如果其中一方立即消失，另一方必须等待30天才能获得退款。终点越远，退款的时间就越长。

第二个问题是，由于每个后续的承诺交易必须递减时间锁，因此对于可以在各方之间交换的承诺交易数量有明确的限制。例如，一个30天的通道，将时间锁设置为未来的4320个块，只能容纳4320个中间承诺交易，然后必须关闭。将承诺交易之间的时间锁定间隔设置为1个块存在风险。通过将承诺交易之间的时间锁定间隔设置为1个块，开发人员为通道参与者创造了一个非常高的负担，他们必须保持警惕，保持在线并监视，并准备在任何时候传输正确的承诺交易。

在之前描述的单向通道示例中，消除每次承诺的时间锁是很容易的。在Emma从Fabian那里收到定时退款交易的签名后，不会在承诺交易上设置时间锁。相反，Emma向Fabian发送每个承诺交易的签名，但Fabian不会向Emma发送任何承诺交易的签名。这意味着只有Fabian拥有承诺交易的两个签名，因此只有他可以广播其中之一。当Emma完成视频流时，Fabian总是倾向于广播支付他最多的交易，即最新的状态。这种构造称为Spillman风格的支付通道，它最早是在2013年首次描述和实现的，尽管它们只能与见证（segwit）交易一起使用，这些交易直到2017年才可用。

现在我们了解了如何使用时间锁来使前期的承诺无效，我们可以看到通过广播承诺交易来合作关闭通道和单方关闭通道之间的区别。在我们之前的示例中，所有的承诺交易都有时间锁，因此广播承诺交易总是涉及等待直到时间锁到期。但是，如果双方就最终余额达成一致，并知道他们都持有最终将实现该余额的承诺交易，他们可以构建一个不带时间锁的结算交易来代表相同的余额。在合作关闭中，任何一方都可以采用最新的承诺交易并构建一个完全相同的结算交易，只是省略了时间锁。双方都可以签署此结算交易，因为知道没有任何作弊的方法，并且无法获得更有利的余额。通过合作签署和传输结算交易，他们可以立即关闭通道并兑现余额。最糟糕的情况是，其中一方可能会小气，拒绝合作，并迫使另一方通过最新的承诺交易进行单方关闭。如果这样做，他们也必须等待他们的资金。

非对称撤销承诺

另一种处理先前承诺状态的方法是明确地撤销它们。然而，这并不容易实现。比特币的一个关键特性是，一旦一笔交易有效，它就保持有效，不会过期。取消交易的唯一方法是让一笔与之冲突的交易得到确认。这就是为什么在简单支付通道示例中我们使用了时间锁，以确保更新的承诺可以在旧的承诺变得有效之前花费。然而，在时间上排序承诺会产生许多约束，使得支付通道难以使用。

即使一笔交易无法取消，也可以通过构造一种使其不受欢迎的方式来实现。我们这样做的方式是给每一方提供一个撤销密钥，如果对方试图作弊，就可以用来惩罚对方。这种撤销先前承诺交易的机制首次作为闪电网络的一部分提出。

为了解释撤销密钥，我们将构建一个更复杂的支付通道，该通道由 Hitesh 和 Irene 经营的两家交易所之间创建。Hitesh 和 Irene 在印度和美国分别经营比特币交易所。Hitesh 的印度交易所的客户经常向 Irene 的美国交易所的客户发送付款，反之亦然。目前，这些交易发生在比特币区块链上，但这意味着支付费用并等待几个区块的确认。在交易所之间建立支付通道将大大降低成本并加快交易流程。 Hitesh 和 Irene 通过协作构建资金交易来启动通道，每个人用 5 比特币为通道提供资金。在签署资金交易之前，他们必须签署第一组承诺（称为退款），以分配给 Hitesh 和 Irene 初始余额各为 5 比特币。资金交易将通道状态锁定在一个 2-of-2 的多重签名中，就像简单通道示例中一样。

资金交易可能有来自 Hitesh 的一个或多个输入（总计达到 5 比特币或更多），以及来自 Irene 的一个或多个输入（总计达到 5 比特币或更多）。输入必须略微超过通道容量，以支付交易费用。交易有一个输出，将总共的 10 比特币锁定到由 Hitesh 和 Irene 共同控制的 2-of-2 多重签名地址。如果他们的输入超出了他们的预期通道贡献，资金交易可能还会有一个或多个输出将剩余资金找零返回给 Hitesh 和 Irene。这是一笔由两方提供和签署的单一交易。它必须在协作构建并由每一方签署之后才能传输。

现在，与其创建一笔由两方签署的单一承诺交易，Hitesh 和 Irene 创建了两个不同的非对称承诺交易。

Hitesh 拥有一笔承诺交易，其中包含两个输出。第一个输出**立即**支付给 Irene 她所欠的 5 比特币。第二个输出向 Hitesh 支付他所欠的 5 比特币，但必须经过 1,000 个区块的时间锁定才能生效。交易输出如下：

```
Input: 2-of-2 funding output, signed by Irene


Output 0 <5 bitcoins>:
     <Irene's Public Key> CHECKSIG
Output 1 <5 bitcoins>:
     <1000 blocks>
     CHECKSEQUENCEVERIFY
     DROP
     <Hitesh's Public Key> CHECKSIG
```

Irene 拥有一笔不同的承诺交易，其中包含两个输出。第一个输出立即向 Hitesh 支付他所欠的 5 比特币。第二个输出向 Irene 支付她所欠的 5 比特币，但必须经过 1,000 个区块的时间锁定才能生效。Irene 拥有的承诺交易（由 Hitesh 签名）如下：

```
Input: 2-of-2 funding output, signed by Hitesh

Output 0 <5 bitcoins>:
    <Hitesh's Public Key> CHECKSIG

Output 1 <5 bitcoins>:
    <1000 blocks>
    CHECKSEQUENCEVERIFY
    DROP
    <Irene's Public Key> CHECKSIG
```

这样，每个参与方都拥有一笔承诺交易，用于支配 2-of-2 的资金输出。该输入由另一方签名。任何时候，持有交易的一方也可以签名（完成 2-of-2）并广播交易。然而，如果他们广播承诺交易，它会立即支付给另一方，而他们必须等待时间锁定的到期。通过对一个输出的赎回施加延迟，我们在每个参与方选择单方面广播承诺交易时都让其稍微处于不利地位。但仅靠时间延迟还不足以促使公平行为。

图 14-5 显示了两个非对称的承诺交易，其中支付给承诺持有者的输出被延迟。

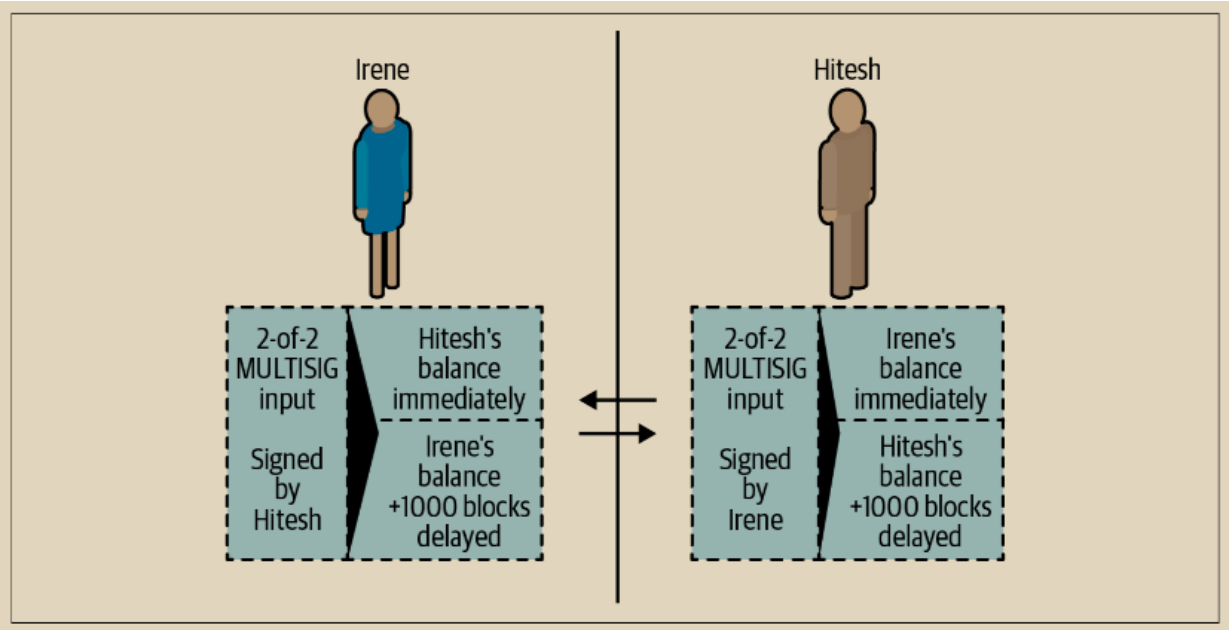


图 14-5. 两个非对称的承诺交易，其中支付给持有交易一方的款项被延迟

\ 现在，我们介绍这个方案的最后一个元素：一个撤销密钥，用于防止作弊者广播已过期的承诺。 撤销密钥允许受害方通过获取渠道的全部余额来惩罚作弊者。

撤销密钥由两个部分组成，每个渠道参与者独立生成一半。 它类似于2-of-2多重签名，但使用椭圆曲线算术构建，以便双方都知道撤销公钥，但每个参与方只知道撤销秘密的一半。

在每一轮中，双方都向对方揭示他们撤销秘密的一半，从而为对方（现在拥有两半）提供了要求惩罚输出的手段，如果此已撤销交易被广播。

每个承诺交易都有一个“延迟”输出。 该输出的赎回脚本允许一方在1,000个区块之后赎回它，或者另一方在拥有撤销密钥的情况下赎回它，以惩罚广播已撤销的承诺。

因此，当Hitesh为Irene创建一个承诺交易供其签名时，他将第二个输出设置为在1,000个区块之后支付给自己，或者支付给撤销公钥（他只知道一半的秘密）。 Hitesh构建了此交易。 他只会在准备转移到新的渠道状态并希望撤销此承诺时向Irene揭示其撤销秘密的一半。

第二个输出的脚本如下：

```
Output 0 <5 bitcoins>:
    <Irene's Public Key> CHECKSIG

Output 1 <5 bitcoins>:
IF
    # Revocation penalty output
    <Revocation Public Key>
ELSE
    <1000 blocks>
    CHECKSEQUENCEVERIFY
    DROP
    <Hitesh's Public Key>
ENDIF
CHECKSIG
```

\ Irene可以放心地签署此交易，因为如果传输，它将立即支付她所欠的款项。Hitesh持有该交易，但知道如果他在单方面关闭渠道时传输该交易，他将需要等待 1,000 个区块才能获得支付。在渠道推进到下一个状态之后，Hitesh必须撤销此承诺交易，然后Irene才会同意签署任何进一步的承诺交易。要做到这一点，他所要做的就是将他的撤销密钥的一半发送给Irene。一旦Irene拥有了这个承诺的撤销秘钥的两个部分，她就可以放心地签署未来的承诺。她知道如果Hitesh试图通过发布先前的承诺来欺骗，她可以使用撤销密钥来赎回Hitesh的延迟产出。如果Hitesh欺骗，Irene会获得双重产出。同时，Hitesh只拥有该撤销公钥的一半，因此无法在 1,000 个区块之前赎回产出。在 1,000 个区块结束之前，Irene将能够赎回产出并惩罚Hitesh。

撤销协议是双边的，意味着在每个轮次中，随着渠道状态的推进，两方交换新的承诺，交换前一次承诺的撤销密钥，并签署彼此的新的承诺交易。在他们接受新状态之后，他们通过彼此提供必要的撤销密钥来使先前的状态不可能被使用，以惩罚任何欺骗行为。

让我们看一个它是如何工作的例子。Irene 的一位客户想要将 2 个比特币发送给 Hitesh 的一位客户。为了在渠道之间传输 2 个比特币，Hitesh 和 Irene 必须推进渠道状态以反映新的余额。他们将承诺到一个新的状态（状态号 2），在这个状态下，渠道的 10 个比特币被分成 7 个比特币给 Hitesh 和 3 个比特币给 Irene。为了推进渠道的状态，他们将各自创建新的承诺交易，反映新的渠道余额。

与之前一样，这些承诺交易是非对称的，因此每个一方持有的承诺交易会迫使他们等待，如果他们要赎回它。至关重要的是，在签署新的承诺交易之前，他们必须首先交换撤销密钥，以使任何过时的承诺无效。在这种特定情况下，Hitesh 的利益与渠道的真实状态保持一致，因此他没有理由广播先前的状态。然而，对于 Irene 来说，状态号 1 让她的余额高于状态号 2。当 Irene 给 Hitesh 提供她之前承诺交易（状态号 1）的撤销密钥时，她实际上正在撤销她从将渠道退回到先前状态中获利的能力，因为有了撤销密钥，Hitesh 可以在没有延迟的情况下赎回先前承诺交易的所有产出。这意味着如果 Irene 广播先前的状态，Hitesh 可以行使权利并获得所有产出。重要的是，撤销不会自动发生。虽然 Hitesh 有能力惩罚 Irene 的欺骗行为，但他必须密切关注区块链的变化迹象。如果他看到先前的承诺交易被广播，他有 1,000 个区块的时间来采取行动并使用撤销密钥来阻止 Irene 的欺骗行为，并通过取走全部余额来惩罚她，总共 10 个比特币。

具有相对时间锁定（CSV）的非对称可撤销承诺是实施支付渠道的一种更好的方法，也是这项技术中的一个非常重要的创新。通过这种构造，渠道可以无限期保持开放，并且可以有数十亿个中间承诺交易。在 LN 的实现中，承诺状态由一个 48 位索引标识，允许在任何单个渠道中进行超过 281 万亿（2.8×10^{14}）次状态转换。

哈希时间锁合约（Hash Time Lock Contracts -HTLC）

\ 支付通道可以通过一种特殊的智能合约进一步扩展，该合约允许参与者将资金锁定到一个可赎回的秘密，并设置一个到期时间。这个特性被称为哈希时间锁合约，简称HTLC，它被用于双向和路由的支付通道。

首先解释一下HTLC中的“哈希”部分。要创建一个HTLC，支付的预期接收者首先会生成一个秘密R。然后，他们计算这个秘密的哈希H：

```
H = Hash(R)
```

\ 这产生了一个哈希H，可以包含在输出的脚本中。谁知道这个秘密，就可以用它来赎回输出。这个秘密R也被称为哈希函数的原像。原像只是作为哈希函数输入的数据。

HTLC的第二部分是“时间锁”组件。如果秘密没有被揭示，支付HTLC的一方可以在一段时间后获得“退款”。这是通过使用CHECKLOCKTIMEVERIFY实现的绝对时间锁。

实施HTLC的脚本可能如下所示：

```
IF
     # Payment if you have the secret R
     HASH160 <H> EQUALVERIFY
     <Receiver Public Key> CHECKSIG
ELSE
     # Refund after timeout.
     <lock time> CHECKLOCKTIMEVERIFY DROP
     <Payer Public Key> CHECKSIG
ENDIF
```

任何知道秘密R的人，当其哈希等于H时，都可以通过执行IF流程的第一条子句来赎回此输出。

如果秘密没有被揭示，并且在一定数量的区块之后HTLC被要求，支付方可以使用IF流程中的第二个子句来要求退款。

这是HTLC的一个基本实现。这种类型的HTLC可以被任何知道秘密R的人兑现。HTLC可以采取许多不同的形式，脚本略有变化。例如，通过在第一条子句中添加CHECKSIG运算符和一个公钥，可以将哈希的兑现限制为特定的接收方，后者也必须知道秘密R。

路由支付通道（闪电网络）

\ 闪电网络（LN）是一个提议的双向支付通道的路由网络，连接端到端。这样的网络可以允许任何参与者在不信任任何中间人的情况下将支付从一个通道路由到另一个通道。LN最早由Joseph Poon和Thadeus Dryja于2015年2月描述，建立在许多其他人提出和详细阐述的支付通道概念之上。

“闪电网络”指的是一个用于路由支付通道网络的特定设计，目前至少有五个不同的开源团队实现了该设计。这些独立的实现由一组在闪电技术基础（BOLT）存储库中描述的互操作性标准协调。

基本闪电网络示例

让我们看看这是如何工作的。 在这个示例中，我们有五个参与者：Alice、Bob、Carol、Diana和Eric。这五个参与者之间已经建立了支付通道，成对连接。Alice与Bob有一个支付通道，Bob与Carol连接，Carol与Diana连接，Diana与Eric连接。为简单起见，让我们假设每个通道由每个参与者提供2比特币的资金，因此每个通道的总容量为4比特币。

图14-6显示了闪电网络中的五个参与者，通过双向支付通道相连，可以将支付从Alice发送到Eric（请参见第332页的“路由支付通道（闪电网络）”）。

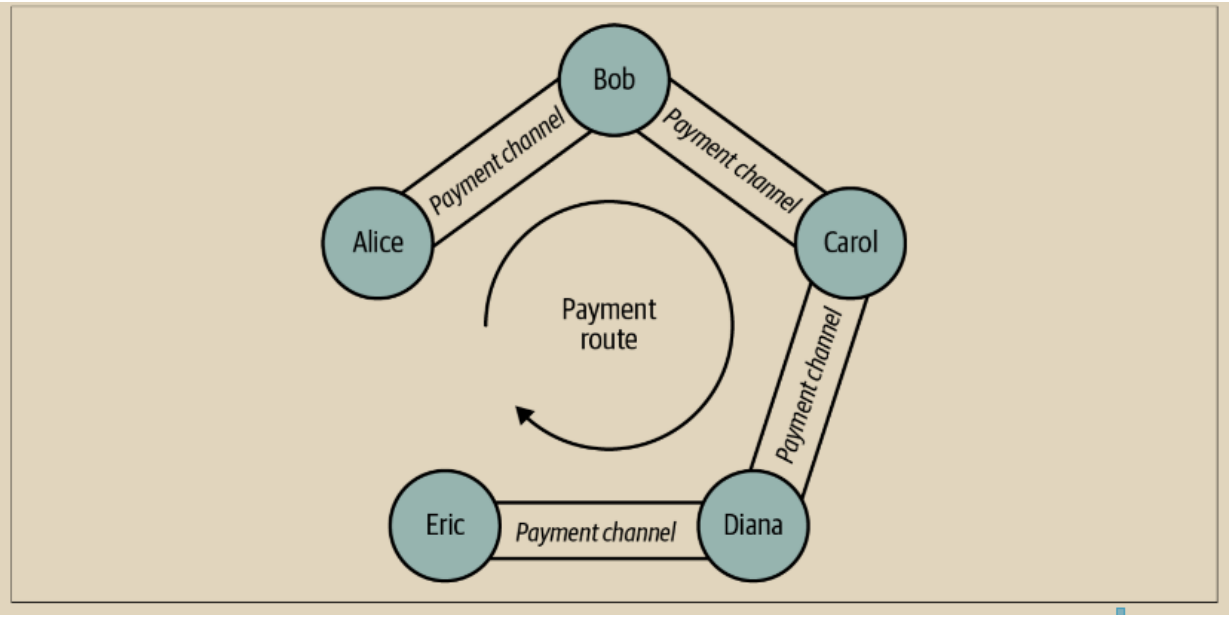


图 14-6. 一系列双向支付通道连接在一起形成一个闪电网络，可以将支付从Alice发送到Eric。

Alice想要支付给Eric 1比特币。然而，Alice与Eric之间没有通过支付通道连接。创建支付通道需要一笔资金交易，这笔交易必须被提交到比特币区块链上。Alice不想打开一个新的支付通道并承诺更多的资金。有没有一种间接支付给Eric的方法呢?

图14-7显示了从Alice到Eric的支付通过一系列连接参与者的支付通道上的HTLC承诺的逐步过程。

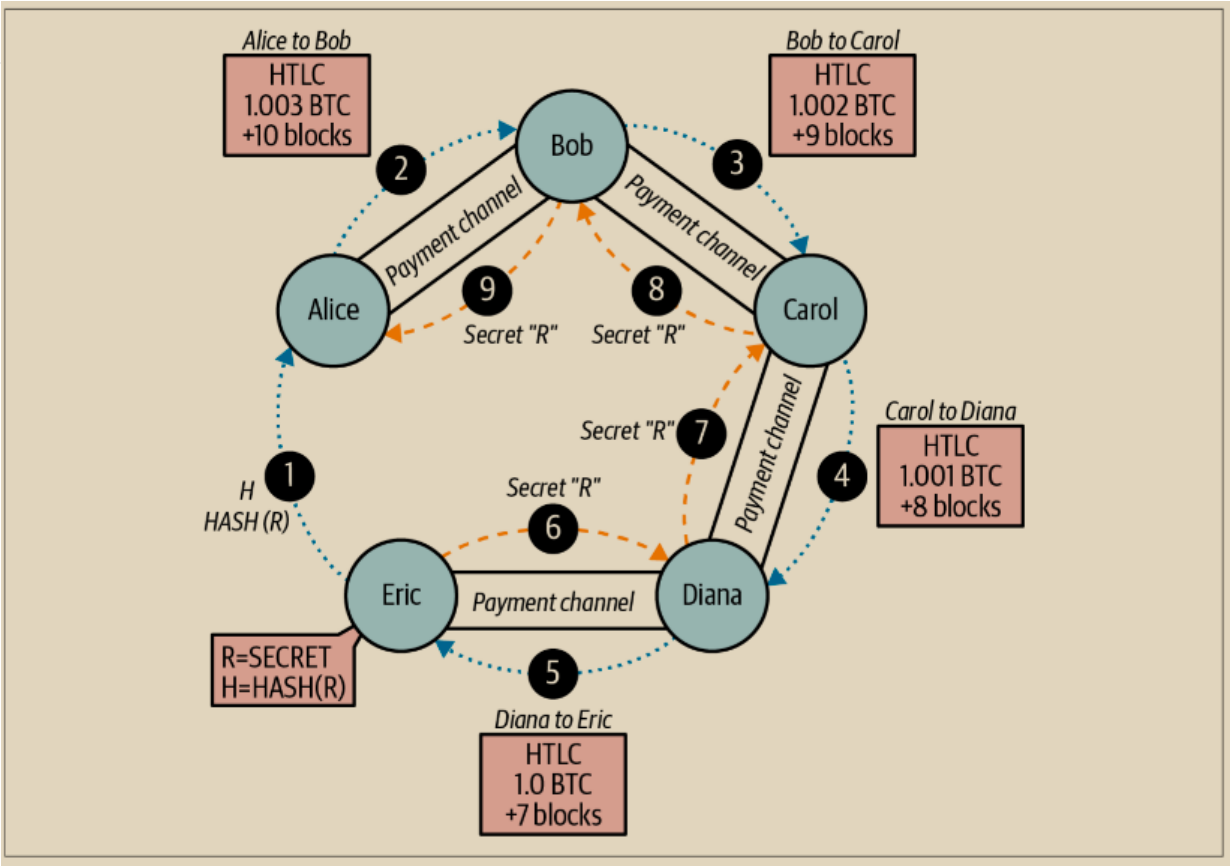


图 14-7. 使用LN 的支付路由步骤

Alice要向Eric支付1比特币。然而，Alice与Eric之间没有通过支付通道连接。创建支付通道需要一个资金交易，这必须提交到比特币区块链上。Alice不想开启一个新的支付通道并投入更多的资金。有没有一种方法可以间接地支付给Eric呢？ 图14-7展示了从Alice向Eric路由支付的逐步过程，通过连接连接参与者之间的HTLC承诺的一系列支付通道。

步骤1：Alice的LN节点跟踪着她与Bob的支付通道，并且能够发现不同通道之间的路由。此外，Alice的LN节点能够通过互联网连接到Eric的LN节点。Eric的LN节点使用随机数生成器生成一个秘密R。Eric的节点不会向任何人透露这个秘密。相反，Eric的节点计算了秘密R的哈希H，并以发票的形式将哈希传输给Alice的节点。

步骤2：现在Alice的LN节点在Alice的节点和Eric的节点之间构建了一条路由。Alice的节点使用路径查找算法找到了一个有效的路由。

然后，Alice的节点构建了一个HTLC，付给哈希H，设置了10个区块的退款超时（当前区块 + 10），金额为1.003比特币。Alice向Bob提供了这个HTLC，从她与Bob的通道余额中扣除了1.003比特币，并将其分配给了HTLC。 HTLC的含义是：“如果Bob知道这个秘密，Alice承诺将通道余额中的1.003比特币支付给Bob，如果10个区块后Alice将其回收。” Alice的余额通过与Bob的通道的承诺交易来表达，其中包含三个输出：2比特币给Bob，0.997比特币给Alice，1.003比特币被提交给了Alice的HTLC。 Alice的余额减少了HTLC提交的金额。

步骤3：Bob现在有了一项承诺，如果他能在接下来的10个区块内获得秘密R，他就可以获得Alice锁定的1.003比特币。有了这个承诺，Bob的节点在他与Carol的支付通道上构建了一个HTLC。Bob的HTLC承诺了1.002比特币给哈希H，持续9个区块，卡罗尔可以在其中拥有秘密R的情况下赎回。 Bob知道，如果Carol能够声明他的HTLC，她必须提供R。如果Bob在9个区块内获得R，他可以用它来获取Alice向他的HTLC。如果Carol无法获取他的HTLC并且他无法获取Alice的HTLC，一切将恢复到之前的通道余额，没有人会损失。 Bob和Carol之间的通道余额现在是：2给Carol，0.998给Bob，1.002由Bob提交给了HTLC。

步骤4：现在，Carol有了一项承诺，如果她在接下来的九个区块内获得秘密R，她就可以获得Bob锁定的1.002比特币。现在，她可以在她与Diana的通道上进行HTLC承诺。她对哈希H承诺了1.001比特币，持续8个区块，Diana可以在其中拥有秘密R的情况下赎回。从Carol的角度来看，如果这样做可以获得0.001比特币，并且如果失败，她不会损

失任何东西。她向Diana的HTLC只有在R被揭示后才有效，此时她可以从Bob那里获得HTLC。Carol和Diana之间的通道余额现在是：2给Diana，0.999给Carol，1.001由Carol提交给了HTLC。

步骤5：最后，Diana可以向Eric提供一个HTLC，承诺1比特币给哈希H，持续7个区块。Diana和Eric之间的通道余额现在是：2给Eric，1给Diana，1由Diana提交给了HTLC。

在这一步的路由中，Eric知道秘密R。因此，他可以要求Diana提供的HTLC。他将R发送给Diana并领取了1比特币，将其添加到他的通道余额中。

现在，Eric和Diana之间的通道余额是：1给Diana，3给Eric。 现在，Diana知道了秘密R。因此，她现在可以要求Carol提供的HTLC。Diana将R发送给Carol，并将1.001比特币添加到她的通道余额中。现在，Carol和Diana之间的通道余额是：0.999给Carol，3.001给Diana。 Diana参与了此支付路由，获得了0.001的收益。

通过路由返回，秘密R使每个参与者都能够要求未完成的HTLC。 Carol从Bob那里要求了1.002，将他们的通道余额设置为：0.998给Bob，3.002给Carol。最后，Bob从Alice那里要求了HTLC。 他们的通道余额更新为：0.997给Alice，3.003给Bob。

Alice向Eric支付了1比特币，而不需要向Eric开启一个支付通道。支付路由中的任何中间方都不必互相信任。通过在通道中短暂承诺他们的资金，他们能够赚取一小笔费用，唯一的风险是如果通道关闭或路由支付失败，会有一小段时间的退款延迟。

闪电网络传输和路径查找

\ 闪电网络节点之间的所有通信都是点对点加密的。此外，节点具有用作标识符和相互验证的长期公钥。

当节点希望向另一个节点发送付款时，首先必须通过连接具有足够容量的付款通道来构建网络路径。节点会广播路由信息，包括它们打开的通道、每个通道的容量以及它们收取的路由费用。路由信息可以通过各种方式共享，随着LN技术的发展，不同的路径查找协议也应运而生。当前实现的路径发现使用P2P模型，其中节点以“泛洪”模式将通道公告传播给它们的对等节点，类似于比特币传播交易的方式。

在我们之前的例子中，Alice的节点使用这些路径发现机制之一找到连接她的节点和Eric的节点的一个或多个路径。一旦Alice的节点构建了一条路径，她将通过将一系列加密的和嵌套的指令传播到相邻的每个支付通道来初始化该路径。

重要的是，此路径仅为Alice的节点所知。付款路由中的所有其他参与者只看到相邻的节点。从Carol的角度来看，这看起来像是Bob向Diana支付的付款。Carol不知道Bob实际上是在转发Alice的付款，她也不知道Diana将要向Eric转发一笔付款。

这是LN的一个关键特性，因为它确保了付款的隐私，并且使得很难对其进行监视、审查或黑名单。但是，Alice如何建立这个付款路径而不向中间节点透露任何信息呢?

LN实现了一种基于名为Sphinx的方案的洋葱路由协议。此路由协议确保付款发送方可以构建和传输一条通过LN的路径，以便：

- 中间节点可以验证和解密它们的路由信息的一部分，并找到下一个跳点。
- 除了前一个和下一个跳点之外，它们无法了解路径中的任何其他节点。
- 它们无法确定付款路径的长度或自己在该路径中的位置。
- 路径的每个部分都以一种方式加密，以使网络级攻击者无法将路径的不同部分的数据包相互关联。
- 与互联网上的洋葱路由匿名协议Tor不同，没有“出口节点”可以被监视。付款不需要传输到比特币区块链上；节点只需要更新通道余额。

使用这种洋葱路由协议，Alice将路径的每个元素都包装在一层加密中，从最后开始，逐层向后处理。她使用Eric的公钥对一条消息进行加密，以将消息发送给Eric。然后，将此消息包装在一个加密的消息中，发送给Diana，同时将Eric标识为下一个接收者。对Diana的消息再次被包装在一个加密的消息中，发送给Carol，同时将Diana标识为下一个接收者。对Carol的消息被加密到Bob的公钥。因此，Alice构建了这个加密的多层“洋葱”消息。她将其发送给Bob，后者只能解密并解开外层。在内部，Bob发现了一条地址给Carol的消息，他可以转发给Carol，但自己无法解读。随着路径的传递，消息被转发、解密、转发，以此类推，一直到Eric。每个参与者只知道前一个和下一个节点在每一跳中。

路径的每个元素包含有关必须传递给下一跳的HTLC、发送的金额、要包括的费用以及HTLC的CLTV锁定时间（以区块为单位）到期的信息。随着路由信息的传播，节点将HTLC承诺转发到下一个跳点。

这时，您可能会想知道节点如何不知道路径的长度和自己在该路径中的位置。毕竟，他们收到一条消息并将其转发给下一个跳点。这条消息不是变得更短了吗，以便他们可以推断出路径的大小和自己的位置？为了防止这种情况发生，数据包的大小是固定的，并填充了随机数据。每个节点只能看到下一个跳点和一个固定长度的加密消息以转发。只有最终的接收者看到没有下一个跳点。对于其他人来说，似乎总是还有更多的跳点要走。

闪电网络的好处

闪电网络是一种第二层的路由技术。它可以应用于支持一些基本功能的任何区块链，如多重签名交易、时间锁和基本智能合约。

闪电网络建立在比特币网络之上，为比特币增加了显著的容量、隐私、细粒度和速度，同时不损害无需中介进行无信任操作的原则：

隐私

闪电网络支付要比比特币区块链上的支付更加私密，因为它们不是公开的。虽然路由中的参与者可以看到沿着它们的通道传播的支付，但他们不知道发件人或收件人。

可替换性

闪电网络使对比特币进行监视和黑名单的难度大大增加，增强了货币的可替代性。

速度

使用闪电网络的比特币交易在毫秒内结算，而不是像使用HTLCs在交易提交到区块之前需要数分钟甚至数小时的时间。

细粒度

闪电网络可以实现至少与比特币的“尘埃”限制一样小的支付，甚至可能更小。

容量

闪电网络通过几个数量级增加了比特币系统的容量。可以通过闪电网络路由的每秒支付数量的上限仅取决于每个节点的容量和速度。

无信任操作

闪电网络使用各个节点之间作为对等方的比特币交易。因此，闪电网络保留了比特币系统的原则，同时显著扩展了其运行参数。

我们仅仅探讨了使用比特币区块链作为信任平台构建的一些新兴应用。这些应用将比特币的范围扩展到了支付之外。

现在您已经读到了本书的末尾，您会如何运用您所获得的知识呢？数百万甚至数十亿人知道“比特币”这个名字，但只有很少一部分人像您现在这样对比特币的工作原理了解得很多。这种知识是宝贵的。更宝贵的是像您这样对比特币充满兴趣，愿意花时间阅读几百页内容的人。

如果您还没有开始行动，请考虑以某种方式为比特币做出贡献。您可以运行一个全节点来验证您收到的比特币支付，构建可以让其他人更轻松使用比特币的应用程序，或者帮助教育其他人了解比特币及其潜力。您甚至可以采取罕见的步骤，为开源比特币基础设施软件做出贡献，比如比特币核心，与一小部分非常聪明的人一起精心构建工具，尽管没有人会为此付费，但可能有数十亿人会依赖这些工具。

无论您的比特币之旅是什么样子，我们都感谢您将《精通比特币》纳入其中。

比特币 - 一种点对点的电子现金系统

Satoshi Nakamoto

satoshin@gmx.com

www.bitcoin.org

摘要。一种纯粹的点对点版本的电子现金将允许在线支付直接从一方发送到另一方，而无需经过金融机构。数字签名提供了部分解决方案，但如果仍然需要信任的第三方来防止双重支付，则主要优势将会丧失。我们提出了使用点对点网络解决双重支付问题的方案。网络通过将交易的时间戳散列到基于哈希的工作量证明的持续链中来记录交易，形成一个不能在不重新执行工作量证明的情况下更改的记录。最长的链不仅作为事件序列的证明，而且作为它来自最大的CPU计算能力池的证明。只要大多数CPU计算能力被不合作攻击网络的节点所控制，它们就会生成最长的链并超越攻击者。网络本身需要最少的结构。消息以尽力而为的方式广播，节点可以随意离开和重新加入网络，接受最长的工作量证明链作为它们离线期间发生的事情的证明。

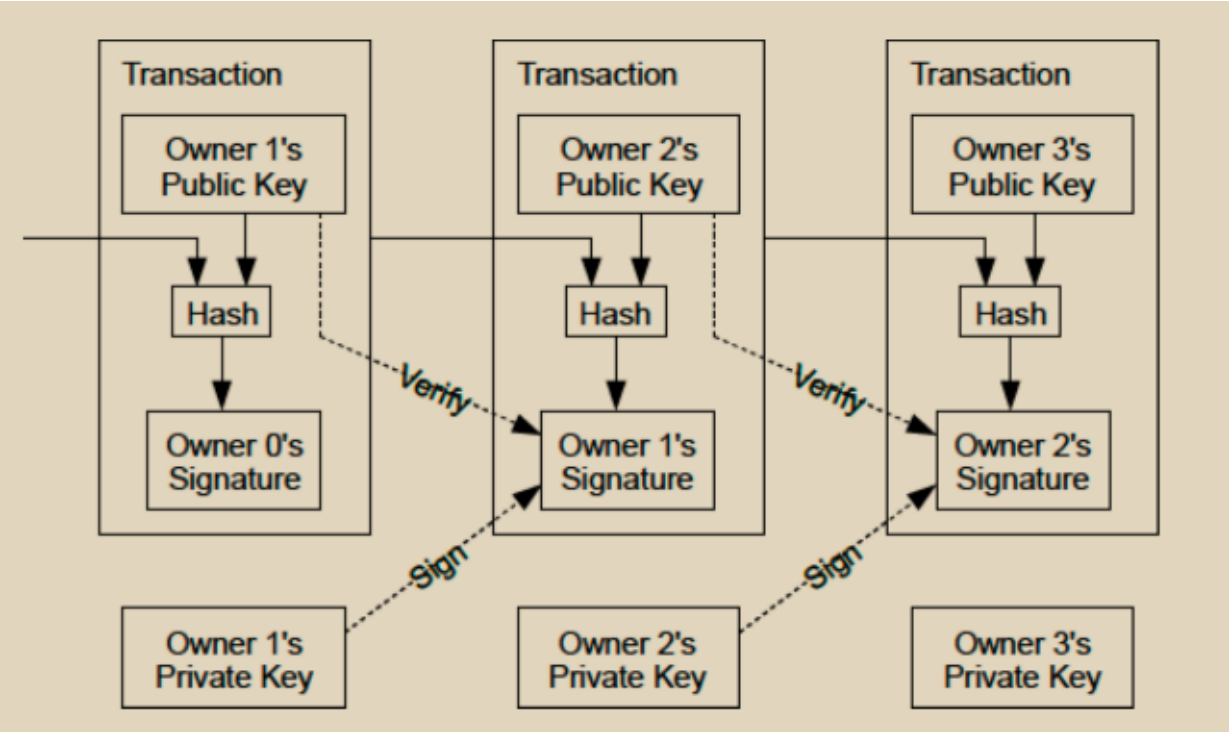
介绍

互联网上的商务几乎完全依赖于金融机构作为可信第三方来处理电子支付。虽然该系统对于大多数交易已经运作得足够好，但它仍然受到基于信任模型的固有弱点的影响。完全不可逆转的交易实际上是不可能的，因为金融机构无法避免调解纠纷。调解的成本增加了交易成本，限制了最小的实际交易规模，并且割断了小额休闲交易的可能性，且在无法进行非可逆支付的非可逆服务上也存在更广泛的成本。由于可能发生逆转，信任的需求扩散开来。商家必须警惕他们的客户，麻烦他们提供比他们本来需要的更多的信息。一定比例的欺诈被接受为不可避免的。这些成本和支付不确定性可以通过使用实物货币面对面进行避免，但是在没有可信第三方的通信渠道上进行支付没有任何机制存在。

所需要的是一种基于加密证明而不是信任的电子支付系统，允许任何两个愿意的当事人直接进行交易而无需信任第三方。计算上不可逆转的交易将保护卖家免受欺诈，而常规的第三方担保机制可以轻松实现以保护买家。在本文中，我们提出了使用点对点分布式时间戳服务器来生成交易时间顺序的计算证明来解决双重支付问题的方案。只要诚实的节点共同控制的CPU算力多于任何合作的攻击者节点组，系统就是安全的。

交易

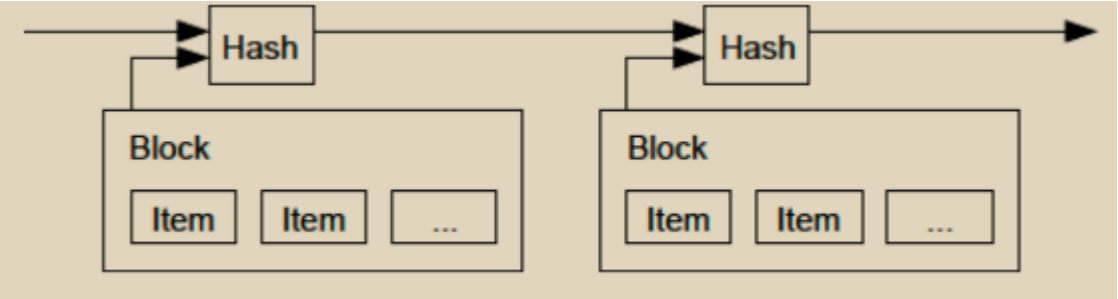
我们将电子硬币定义为一系列数字签名。每个所有者通过对上一笔交易的哈希和下一位所有者的公钥进行数字签名，并将它们添加到硬币的末尾来将硬币转移给下一个所有者。收款人可以验证这些签名以验证所有权链。



问题当然是收款人无法验证其中一个所有者是否曾经双重支付了该硬币。一个常见的解决方案是引入一个可信的中央机构或者铸币厂，来检查每一笔交易是否存在双重支付。在每笔交易之后，硬币必须被退回给铸币厂以发行新的硬币，而只有直接由铸币厂发行的硬币才被信任不会被双重支付。这种解决方案的问题在于整个货币系统的命运取决于运行铸币厂的公司，每笔交易都必须通过他们进行，就像银行一样。

我们需要一种方法让收款人知道前任所有者没有签署任何早期的交易。对于我们而言，最早的交易才是重要的，所以我们不关心后来的双重支付尝试。确认没有交易存在的唯一方法是了解所有交易。在基于铸币厂的模型中，铸币厂知道所有交易，并决定哪些先到达。为了在没有可信方的情况下实现这一点，交易必须公开宣布，而且我们需要一个参与者能够就它们被接收的顺序达成一致意见的系统。收款人需要证明，在每笔交易时，大多数节点都同意它是第一次接收的。

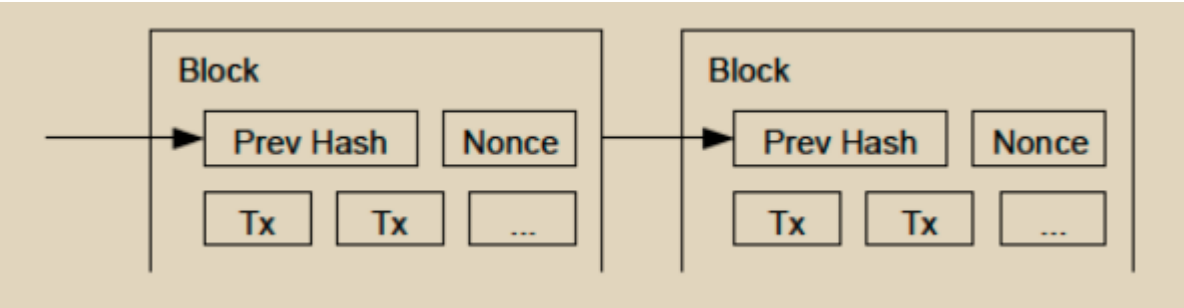
时间戳服务器



我们提出的解决方案始于一个时间戳服务器。时间戳服务器的工作方式是对要进行时间戳处理的数据块进行哈希，并广泛发布该哈希值，比如在报纸或Usenet帖子中[2-5]。时间戳证明了数据必须在某个时间点存在，显然，为了被包含在哈希中。每个时间戳在其哈希中包含了前一个时间戳，形成一个链条，每个额外的时间戳都加强了之前的时间戳。

工作量证明(PoW)

为了在点对点基础上实现分布式时间戳服务器，我们将需要使用类似于Adam Back的Hashcash的工作量证明系统，而不是使用报纸或Usenet帖子。工作量证明涉及扫描一个值，使其哈希（例如，使用SHA-256）以一定数量的零位开始。所需的平均工作量随所需的零位数量呈指数增长，并且可以通过执行单个哈希来验证。对于我们的时间戳网络，我们通过在区块中递增一个随机数（nonce）来实现工作量证明，直到找到一个值，使得区块的哈希具有所需的零位。一旦CPU投入了足够的工作量使其满足工作量证明，区块就无法更改，除非重新进行这项工作。随着后续区块的链接，要更改该区块需要重新做所有后续区块的工作。



工作量证明还解决了确定多数决策中的代表性的问题。如果多数决策基于一个IP地址一票，那么任何能够分配多个IP的人都可以篡改它。工作量证明本质上是一个CPU一票制。多数决策由最长的链表示，该链在其中投入了最大的工作量证明努力。如果多数CPU算力由诚实节点控制，那么诚实链将增长最快，并超过任何竞争链。要修改过去的区块，攻击者必须重新做该区块以及其后的所有区块的工作量证明，然后赶上并超过诚实节点的工作量。我们将在后面展示，随着后续区块的添加，较慢的攻击者追赶的概率呈指数级下降。

为了补偿硬件速度的增加和随时间变化的节点运行兴趣的差异，工作量证明的难度由一个移动平均数确定，以达到每小时平均区块数量的目标。如果生成得太快，难度就会增加。

网络

\ 运行网络的步骤如下：

1. 新的交易被广播到所有节点。
2. 每个节点将新的交易收集到一个区块中。
3. 每个节点努力为其区块找到一个难以证明的工作量。
4. 当一个节点找到一个工作量证明时，它将该区块广播到所有节点。
5. 节点只接受其中所有交易都是有效的且未被花费的区块。
6. 节点通过努力创建链中的下一个区块来表示它们对该区块的接受，使用接受区块的哈希作为上一个哈希。

节点始终认为最长的链是正确的，并将继续努力延伸它。如果两个节点同时广播了不同版本的下一个区块，一些节点可能会先收到其中一个。在这种情况下，它们会在收到的第一个上工作，但会保存另一个分支，以防它变得更长。当找到下一个工作量证明并且一个分支变得更长时，平局将被打破；之前在另一个分支上工作的节点将转移到较长的分支上。

新的交易广播不一定需要到达所有节点。只要它们到达许多节点，它们就会很快进入一个区块。区块广播也能容忍丢失的消息。如果一个节点没有收到一个区块，它将在收到下一个区块并意识到它错过了一个时请求它。

\

激励

根据惯例，区块中的第一笔交易是一笔特殊的交易，用于创建一个由该区块的创建者拥有的新币。这为节点支持网络提供了激励，并为最初将币分配到流通中提供了一种方式，因为没有中央机构来发行它们。持续不断地增加一定数量的新币相当于金矿工耗费资源将金添加到流通中。在我们的情况下，耗费的资源是CPU时间和电力。

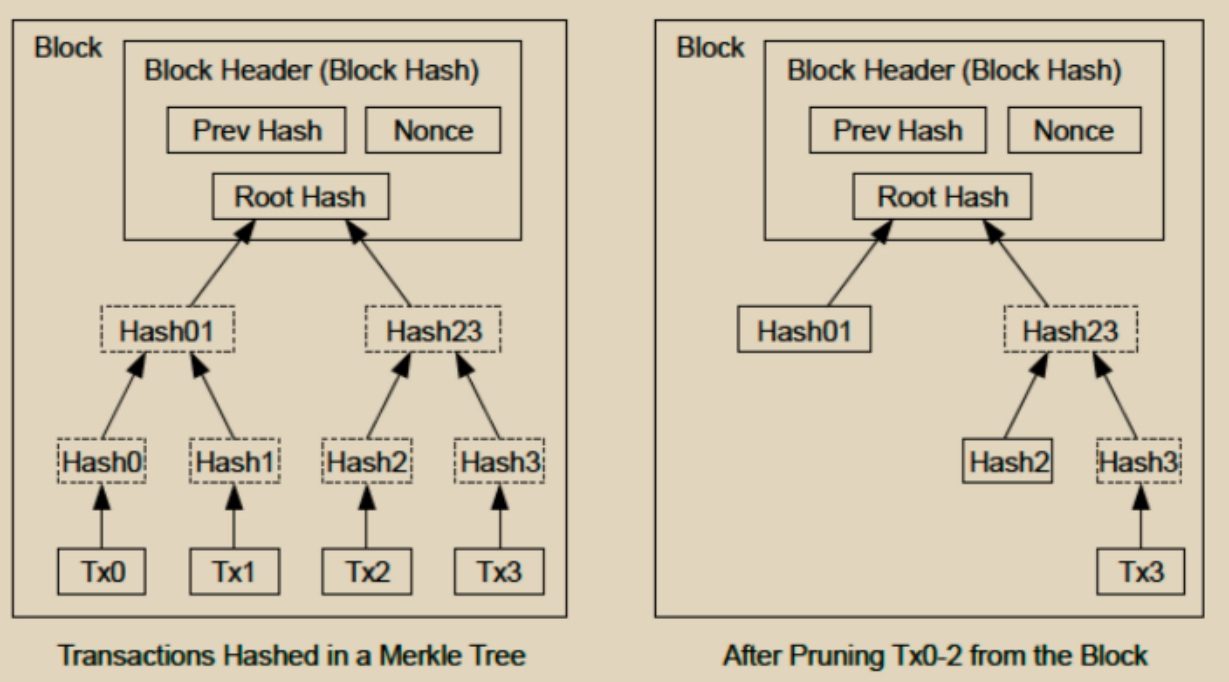
激励也可以通过交易费来资助。如果一笔交易的输出价值小于其输入价值，那么差额将作为交易费添加到包含该交易的区块的激励价值中。一旦预定数量的币进入流通，激励可以完全过渡到交易费，并且完全不受通货膨胀的影响。

激励可能有助于鼓励节点保持诚实。如果一个贪婪的攻击者能够聚集比所有诚实节点更多的CPU算力，他将不得不在利用它来欺骗他人通过窃取回他的支付，或者用它来生成新的币之间做出选择。他应该会发现，遵循规则更有利于他，这些规则给予他的新币比其他人加在一起的要多，而不是破坏系统和他自己财富的有效性。

\

回收磁盘空间

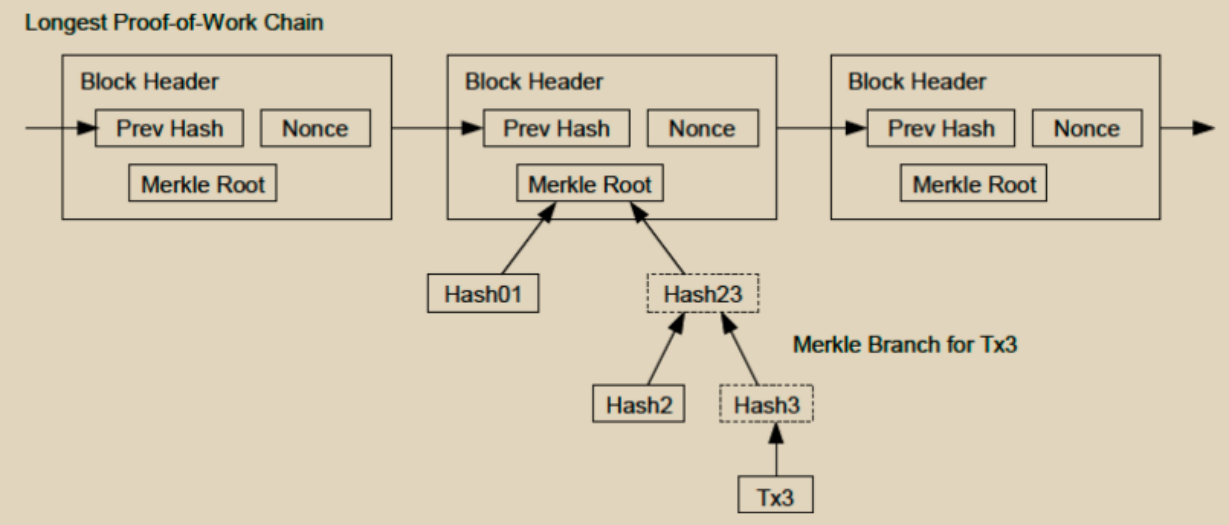
一旦一个币的最新交易被足够多的区块深埋之后，该币之前的已花费交易就可以被丢弃以节省磁盘空间。为了实现这一点而不破坏区块的哈希，交易被哈希成一个默克尔树，在区块的哈希中只包含根。旧的区块可以通过截断树的分支来压缩。内部哈希值不需要被存储。



一个没有交易的区块头大约是80字节。假设每10分钟生成一个区块，那么每年大约是80字节 * 6 * 24 * 365 = 4.2MB。截至2008年，计算机系统通常配备2GB的RAM，而摩尔定律预测当前每年增长1.2GB，即使区块头必须保留在内存中，存储也不应该是一个问题。

简化支付验证（Simplified Payment Verification，SPV）

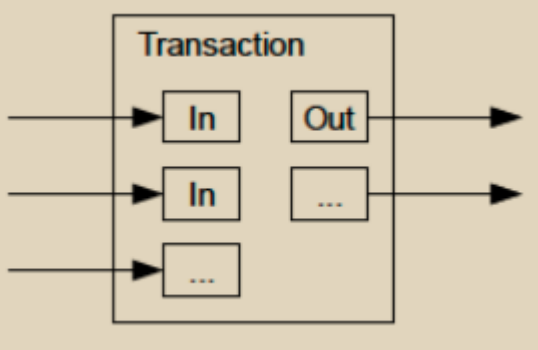
用户只需保留最长工作量证明链的区块头副本即可验证支付，他可以通过查询网络节点获取这些副本，直到确信自己拥有了最长的链，并获取将交易链接到其时间戳位置的默克尔分支。虽然用户无法自行验证交易，但通过将其与链中的某个位置关联起来，他可以看到网络节点已接受该交易，并且之后添加的区块进一步确认了网络已接受该交易。



因此，只要诚实的节点控制着网络，验证就是可靠的，但如果网络被攻击者压倒性地控制，则更容易受到攻击。虽然网络节点可以自行验证交易，但简化的方法可能会被攻击者伪造的交易欺骗，只要攻击者能够继续压倒网络。保护措施之一是接受来自网络节点的警报，当它们检测到无效区块时，提示用户的软件下载完整区块和警报交易以确认不一致之处。经常收到支付的企业可能仍然希望运行自己的节点，以获得更独立的安全性和更快的验证速度。

合并和拆分价值

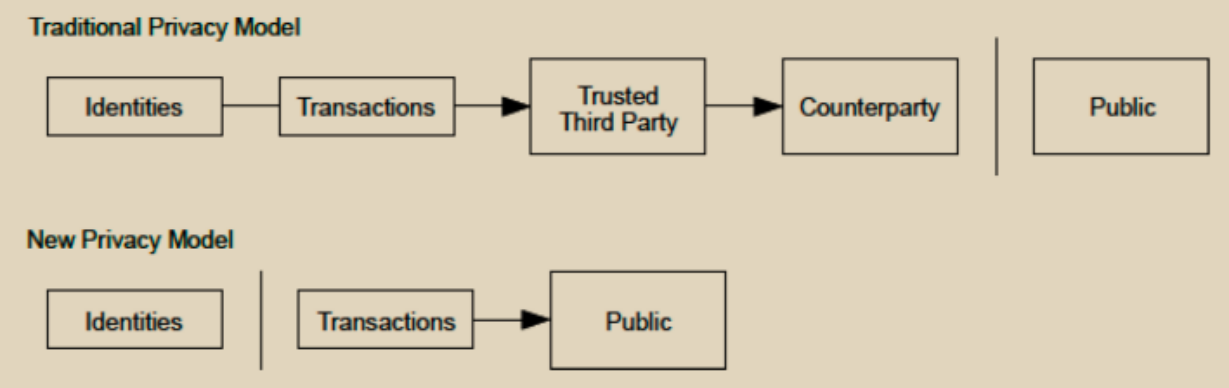
虽然可以单独处理硬币，但对于每一分钱都进行单独交易将会很不方便。为了允许价值的拆分和合并，交易包含多个输入和输出。通常情况下，会有一个较大的前一笔交易作为单一输入，或者多个输入合并较小的金额，最多有两个输出：一个用于支付，另一个将余额（如果有）退回给发送者。



应该注意的是，扇出（即一笔交易依赖于多笔交易，而这些交易又依赖于更多的交易）在这里并不是一个问题。这里从不需要提取一笔交易的完整独立历史记录。

隐私

传统银行模式通过限制信息访问权限仅限于参与方和受信任的第三方来实现一定程度的隐私。必须公开宣布所有交易的必要性排除了这种方法，但隐私仍然可以通过在另一个地方中断信息流来维护：通过保持公钥匿名。公众可以看到某人向另一个人发送了一笔金额，但没有信息将交易与任何人联系起来。这类似于股票交易所公布的信息水平，即单笔交易的时间和金额（“电子交易信息”）是公开的，但没有透露交易双方是谁。



作为额外的防火墙，应该为每个交易使用一个新的密钥对，以防止它们被链接到同一所有者。一些链接仍然是不可避免的，特别是对于多输入交易，它们必然会揭示它们的输入是由同一所有者拥有的。风险在于，如果密钥的所有者被揭示，链接可能会揭示其他属于同一所有者的交易。

计算

\ 我们考虑的情况是攻击者试图比诚实链更快地生成替代链。即使成功，也不会使系统对任意更改敞开大门，比如凭空创造价值或获取从未属于攻击者的资金。节点不会接受无效的交易作为支付，而诚实的节点永远不会接受包含这些交易的区块。攻击者只能尝试改变自己的一笔交易，以取回最近花费的资金。

诚实链和攻击者链之间的竞赛可以被描述为一个二项随机游走。成功事件是诚实链被延长一块，其领先优势增加 +1，而失败事件是攻击者链被延长一块，使差距减少 -1。

攻击者从给定的落后位置赶上的概率类似于《赌徒破产问题》。假设一个信用无限的赌徒从赤字开始，并进行可能无限次的尝试来达到盈亏平衡。我们可以计算他是否会达到盈亏平衡，或者攻击者是否会赶上诚实链的概率，如下所示：

p = 诚实节点找到下一个区块的概率

q = 攻击者找到下一个区块的概率

qz = 攻击者从 z 个区块落后赶上的概率

$$q_z = \begin{cases} 1 & \text{if } p \leq q \\ (q/p)^z & \text{if } p > q \end{cases}$$

鉴于我们假设 p > q，随着攻击者需要赶上的区块数量增加，概率呈指数下降。面对不利的情况，如果攻击者在早期没有幸运地迈出重要的一步，随着他落后的距离增加，他的机会将变得微乎其微。

现在我们考虑新交易的接收方需要等待多久才能足够确信发送方无法更改交易。我们假设发送方是一名攻击者，他希望让接收方相信他已经付款了一段时间，然后在一段时间后将其更改为支付给自己。当接收方发现时，他已经晚了，但发送方希望这会太晚。

接收方在签名前不久生成一个新的密钥对，并将公钥提供给发送方。这样可以防止发送方提前准备好一连串的区块，不断地工作直到他幸运地领先一段距离，然后在那一刻执行交易。一旦交易发送出去，不诚实的发送方会秘密地开始在一个平行链上工作，其中包含他交易的另一个版本。

接收方等待直到交易被添加到一个区块中，并且之后链接了 z 个区块。他不知道攻击者取得了多少进展，但是假设诚实的区块需要平均预期时间来生成，攻击者的潜在进展将服从泊松分布，其期望值为：

$$\lambda = z\frac{q}{p}$$

为了得到攻击者现在仍然有可能赶上的概率，我们将他可能已经取得的每一次进展的泊松密度与他从那一点上赶上的概率相乘：

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (q/p)^{(z-k)} & \text{if } k \leq z \\ 1 & \text{if } k > z \end{cases}$$

重新排列以避免对分布的无限尾部求和...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!}\left(1 - (q/p)^{(z-k)}\right)$$

转换为C代码...

```c
#include <math.h>
double AttackerSuccessProbability(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
            poisson *= lambda / i;
        sum -= poisson * (1 - pow(q / p, z - k));
    }
    return sum;
}
```

运行一些结果，我们可以看到概率随着 z 呈指数级下降。

```
q=0.1
z=0 P=1.0000000
z=1 P=0.2045873
z=2 P=0.0509779
z=3 P=0.0131722
z=4 P=0.0034552
z=5 P=0.0009137
z=6 P=0.0002428
z=7 P=0.0000647
z=8 P=0.0000173
z=9 P=0.0000046
z=10 P=0.0000012
q=0.3
z=0 P=1.0000000
z=5 P=0.1773523
z=10 P=0.0416605
z=15 P=0.0101008
z=20 P=0.0024804
z=25 P=0.0006132
z=30 P=0.0001522
z=35 P=0.0000379
z=40 P=0.0000095
z=45 P=0.0000024
z=50 P=0.0000006
```

解决 P 小于 0.1% 的情况...

```
P < 0.001
q=0.10 z=5
q=0.15 z=8
q=0.20 z=11
q=0.25 z=15
q=0.30 z=24
q=0.35 z=41
q=0.40 z=89
q=0.45 z=340
```

结论

我们提出了一种在不依赖信任的情况下进行电子交易的系统。我们从通常的由数字签名制成的硬币框架开始，这提供了对所有权的严格控制，但在没有防止双重支付的方式的情况下是不完整的。为了解决这个问题，我们提出了一种使用工作证明的点对点网络，用于记录交易的公共历史，如果诚实的节点控制了大多数 CPU 计算能力，那么这种记录将很快变得计算上不可行，以防止攻击者修改交易历史。这个网络以其非结构化的简单性而稳健。节点一起工作，几乎不需要协调。它们不需要被识别，因为消息不会被路由到任何特定的位置，只需要尽力而为地传递。节点可以随意离开和重新加入网络，接受工作证明链作为他们离开时发生的事情的证据。它们通过CPU计算能力来表达对有效区块的接受，通过继续扩展有效区块并拒绝在无效区块上工作来拒绝无效区块。任何必要的规则和激励措施都可以通过这种共识机制来执行。

引用

[1] W. Dai, “b-money,” http://www.weidai.com/bmoney.txt, 1998.

[2] H. Massias, X.S. Avila, and J.-J. Quisquater, “Design of a secure timestamping service with minimal trust requirements,” In 20th Symposium on Information Theory in the Benelux, May 1999.

[3] S. Haber, W.S. Stornetta, “How to time-stamp a digital document,” In Journal of Cryptology, vol 3, no 2, pages 99-111, 1991.

[4] D. Bayer, S. Haber, W.S. Stornetta, “Improving the efficiency and reliability of digital time-stamping,” In Sequences II: Methods in Communication, Security and Computer Science, pages 329-334, 1993.

[5] S. Haber, W.S. Stornetta, “Secure names for bit-strings,” In Proceedings of the 4th ACM Conference on Computer and Communications Security, pages 28-35, April 1997.

[6] A. Back, “Hashcash - a denial of service counter-measure,” http://www.hashcash.org/papers/hashcash.pdf, 2002.

[7] R.C. Merkle, “Protocols for public key cryptosystems,” In Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society, pages 122-133, April 1980.

[8] W. Feller, “An introduction to probability theory and its applications,” 1957.

综合介绍

比特币改进建议（Bitcoin Improvement Proposals，BIPs）是提供信息给比特币社区或描述比特币或其流程或环境中的新功能的设计文档。

根据 BIP1《BIP 目的和指南》，BIPs 有三种类型：

标准BIP

描述影响大多数或所有比特币实现的任何更改，例如对网络协议的更改、区块或交易有效性规则的更改，或者任何影响使用比特币的应用程序互操作性的更改或添加。

信息性BIP

描述比特币设计问题或向比特币社区提供一般性指南或信息，但不提出新功能。信息性BIP不一定代表比特币社区的共识或建议，因此用户和实施者可以忽略信息性BIP或遵循其建议。

流程BIP

描述比特币流程或提议对流程进行更改（或事件）。流程BIP类似于标准BIP，但适用于比特币协议之外的其他领域。它们可能提出一种实现方法，但不涉及比特币的代码库；它们通常需要社区共识。与信息性BIP不同，它们不仅仅是建议，用户通常不能忽略它们。示例包括程序、指南、决策过程的更改以及比特币开发中使用的工具或环境的更改。任何元-BIP也被视为流程BIP。

\ BIPs被记录在GitHub上的一个版本化存储库中。一份来自开源比特币核心项目的MIT许可的文件，在这里以编辑形式重现，描述了它实现了哪些BIP，包括列出每个BIP的拉取请求（PR）和比特币核心版本，在哪个版本中支持每个BIP被添加或显著更改。

已由比特币核心实现的BIPs:

\ !ChatGPT

• BIP9: 允许并行部署多个软分叉的更改自v0.12.1起已经实现（PR＃7575）。

• BIP11: 多重签名输出自v0.6.0起为标准（PR＃669）。

• BIP13: 用于P2SH地址的地址格式自v0.6.0起已经实现（PR＃669）。

• BIP14: 自v0.6.0起使用子版本字符串作为用户代理（PR＃669）。

• BIP16: 付款到脚本哈希评估规则自v0.6.0起已实现，并于2012年4月1日生效（PR＃748）。

• BIP21: 用于比特币支付的URI格式自v0.6.0起已实现（PR＃176）。

• BIP22: 用于挖矿的getblocktemplate（GBT）RPC协议自v0.7.0起已实现（PR＃936）。

• BIP23: 自v0.10.0rc1起，已实现了一些GBT的扩展，包括longpolling和块提议（PR＃1816）。

• BIP30: 禁止使用与之前的未完全花费交易相同的txid创建新交易的评估规则自v0.6.0起已实现，规则于2012年3月15日生效（PR＃915）。

• BIP31: pong协议消息（和协议版本提升到60001）自v0.6.1起已实现（PR＃1081）。

• BIP32: 自v0.13.0起实现了分层确定性钱包（PR＃8035）。

• BIP34: 要求区块包含它们的高度（编号）在coinbase输入中，以及引入版本2区块的规则自v0.7.0起已实现。规则自2013年3月5日（区块224413）起对版本2区块生效，并自2013年3月25日（区块227931）起不再允许版本1区块（PR＃1526）。

• BIP35: mempool协议消息（和协议版本提升到60002）自v0.7.0起已实现（PR＃1641）。自v0.13.0起，仅适用于NODE_BLOOM（BIP111）节点。

• BIP37: 自v0.8.0起实现了用于交易中继的布隆过滤，用于块的部分默克尔树，以及协议版本提升到70001（启用低带宽轻量级客户端）（PR＃1795）。自v0.19.0以来默认禁用，可以通过-peerbloomfilters选项启用。

• BIP42: 修复了在块13440000后补贴计划恢复的错误自v0.9.2（PR＃3842）。

• BIP43: 从v0.21.0开始引入的实验性描述符钱包默认使用BIP43提出的分层确定性钱包派生（PR＃16528）。

• BIP44: 从v0.21.0开始引入的实验性描述符钱包默认使用BIP44提出的分层确定性钱包派生（PR＃16528）。

• BIP49: 从v0.21.0开始引入的实验性描述符钱包默认使用BIP49提出的分层确定性钱包派生（PR＃16528）。

• BIP61: 拒绝协议消息（和协议版本提升到70002）已在v0.9.0中添加（PR＃3185）。从v0.17.0开始，是否发送拒绝消息可以通过-enablebip61选项配置，并且支持已被弃用（默认禁用），从v0.18.0开始。支持已在v0.20.0中删除（PR＃15437）。

• BIP65: CHECKLOCKTIMEVERIFY软分叉自v0.12.0起合并（PR＃6351），并回溯到v0.11.2和v0.10.4。仅在PR＃6124中添加了内存池限时付款。

• BIP66: 严格的DER规则和相关的版本3区块自v0.10.0起已实现（PR＃5713）。

• BIP68: 自v0.12.1起实现了序列锁（PR＃7184），并自v0.19.0起被埋藏（PR＃16060）。

• BIP70，71，72: 自v0.9.0以来，比特币核心GUI支持支付协议（PR＃5216）。支持可以选择在构建时禁用，自v0.18.0起（PR 14451），并且自v0.19.0起在构建时默认禁用（PR＃15584）。自v0.20.0起已删除（PR 17165）。

• BIP84: 自v0.21.0起，默认情况下使用BIP84提议的分层确定性钱包派生的实验性描述符钱包。(PR＃16528)

• BIP86: 自v23.0起，默认情况下使用BIP86提议的分层确定性钱包派生的描述符钱包。(PR＃22364)

• BIP90: 自v0.14.0起，触发BIPs 34、65和66激活的机制简化为区块高度检查。(PR＃8391)

• BIP111: 自v0.13.0起，为所有对等版本添加并强制执行NODE_BLOOM服务位。(PR＃6579和PR＃6641)

• BIP112: 自v0.12.1起，实现了CHECKSEQUENCEVERIFY操作码。(PR＃7524)，并自v0.19.0起被淘汰。(PR＃16060)

• BIP113: 自v0.12.1起，实现了过去中位时间锁定时间计算。(PR＃6566)，并自v0.19.0起被淘汰。(PR＃16060)

• BIP125: 选择性完全替代费用信号部分实施。

• BIP130: 自v0.12.0起，直接头部公告通过与对等版本≥70012协商 (PR 6494)。

• BIP133: 自v0.13.0起，对等版本≥70013的费率过滤消息受到尊重并发送(PR 7542)。

• BIP141: 隔离见证(共识层)自v0.13.0起 (PR 8149)，在v0.13.1中定义为主网 (PR 8937)，并自v0.19.0起被淘汰 (PR＃16060)。

• BIP143: 为版本0见证程序的交易签名验证自v0.13.0起 (PR 8149)，在v0.13.1中定义为主网 (PR 8937)，并自v0.19.0起被淘汰 (PR＃16060)。

• BIP144: 自0.13.0起的隔离见证(PR 8149)。

• BIP145: 隔离见证的getblocktemplate更新自v0.13.0起 (PR 8149)。

• BIP147: 自v0.13.1起的NULLDUMMY软分叉 (PR 8636和PR 8937)，自v0.19.0起被淘汰 (PR＃16060)。

• BIP152: 自v0.13.0起使用紧凑块传输和相关优化(PR 8068)。

• BIP155: 自v0.21.0起支持addrv2和sendaddrv2消息，可以中继Tor V3地址(和其他网络)。(PR 19954)

• BIP157 158: 轻量级客户端的紧凑块过滤器可以在v0.19.0上索引 (PR #14121)，并在v0.21.0上提供给P2P网络的对等体 (PR #16442)。

• BIP159: 自v0.16.0起，NODE_NETWORK_LIMITED服务位被标记，自v0.17.0起连接到这些节点 (PR 11740, PR 10387)。

• BIP173: 自v0.16.0起，原生隔离见证输出的Bech32地址得到支持 (PR 11167)，自v0.20.0起默认生成Bech32地址 (PR 16884)。

• BIP174: 自v0.17.0起存在用于部分签名比特币交易(PSBT)的RPC (PR 13557)。

• BIP176: Bits Denomination [QT only]自v0.16.0起得到支持 (PR 12035)。

• BIP325: 自v0.21.0起支持Signet测试网络 (PR 18267)。

• BIP339: 自v0.21.0起支持根据wtxid中继交易 (PR 18044)。

• BIP340 341 342: 自v0.21.0起实现了Taproot的验证规则(包括Schnorr签名和Tapscript叶节点)，自v0.21.1起在主网上激活(PR 21377, PR 21686)。

• BIP350: 自v22.0起，用于原生v1+隔离见证输出的地址使用bech32m而不是bech32(PR 20861)。

• BIP371: 自v24.0起，支持PSBT的Taproot字段 (PR 22558)。

• BIP380 381 382 383 384 385: 自v0.17.0起实现了输出脚本描述符和大部分脚本表达式 (PR 13697)。

• BIP386: 自v22.0起实现了tr()输出脚本描述符 (PR 22051)。