**第1章 介绍**

**1.1 什么是比特币**

比特币是一系列构成数字货币生态系统的概念和技术的组合。比特币的货币单位叫做“比特币”，用于存储和传递价值。比特币用户间的通讯主要通过比特币协议在因特网上进行，也可以在其他通讯网络中进行。比特币协议栈是开源的，可以在各种不同的计算设备上运行，包括笔记本电脑，智能手机等，用户可以很方便的接入比特币网络。

用户可以在网络中传递比特币，完成一切传统货币可以完成的事情，包括买卖商品，转账给特定的个人或者组织，发放贷款等等。比特币可以进行买卖，也可以在专业的货币交易所中，与其他货币进行兑换。比特币交易快速、安全并且没有边界，从某种程度上来说，是因特网上的一种完美的货币形式。

不像传统货币，比特币是一种彻底的虚拟货币，没有物理货币，甚至连电子货币本身都不存在。货币隐含在发送者与接受者进行价值交换的交易当中。比特币用户拥有自己的密钥，用以证明比特币网络中的交易所有权，从而实现交易消费，或将其传递给新的接收者。这些密钥通常存储于用户计算机的数字钱包中。拥有解锁交易的密钥是花费比特币的唯一要求，这也就把对比特币的控制权完完全全的交给了用户。

比特币是一个分布式的，点对点系统。网络中没有“中央”服务器，也没有控制点。比特币是由称之为“挖矿”的过程产生的，它是一种在验证比特币交易的过程中竞争解决一类数学问题的机制。任何比特币网络的参与者（运行完整的比特币协议栈的人）都可以成为挖矿者，他们可以使用自己的算机的处理能力去验证和记录交易。平均每隔10分钟，总有人能完成过去10分钟产生的交易的验证过程，并因此获得全新产生的比特币的奖励。本质上说，比特币的的挖矿机制使得央行的货币发行和清算机制得以去中心化，央行的功能被这种全局竞争机制替代了。

比特币协议内建的算法规范了全网挖矿的行为。矿工处理记录交易区块的难度可以动态调整，确保了不管网络上有多少矿工（CPU）同时在工作，最终都能维持大致10分钟挖到一个区块的速度。协议同样规定了每隔四年，新比特币的创建速度将减半，这将比特币的总量限制在了2100万的总量内。因此，比特币的发行量与预测的曲线可以尽量靠近，直到2140年达到2100万的总量。鉴于比特币发行量递减的原因，在长期内，比特币是维持通缩的。此外，比特币无法突破限制“印制”新钱导致通胀。

除了表现出来的货币属性，比特币也是协议的名称，它是一个网络，一种分布式计算的创新。比特币货币仅仅是基于这个发明的第一个真正的应用。作为一个开发者，我觉得比特币对货币的革新就如同当年互联网的出现，它是一种传播价值，通过分布式计算保护数字资产的拥有权的网络。比特币能做的比你第一眼看到的多得多。

在本章中，我们从解释主要的概念和名词开始，下载所需软件，并使用比特币进行简单交易。接下来的几章中，我们将一步步解释使比特币成为可能的技术细节，并深入了解比特币网络和协议的运行机制。

| **比特币之前的数字货币** |
| --- |
| 数字货币的出现与密码学的发展紧密相关。考虑到利用数字来表示货物或者服务的价值的根本挑战，数字货币的出现也就不足为奇了。任何接受数字货币的人都面对两个基本问题：  1. 我能相信钱是真实的而不是伪造的吗？  2. 我能确定没人会声明钱是他的而不是我的吗？（又叫“双重支付”问题）  纸币的发行者为了防止假钞，通过采用越来越复杂的纸张和更为先进的印刷技术来印制钞票。使用物理货币情况下，解决重复支付问题非常简单，因为同一张钞票不能同时在两个地方出现。当然，传统货币也经常以电子的方式进行存储和传递，在这种情况下，防伪和防止双重支付是通过中央机构对电子交易进行集中清算实现的，这个机构拥有货币流通的全局视角。对于数字货币而言，它无法依靠防伪油墨，全息安全线来提供安全保证，密码学提供了基础的保证，得以实现对用户合法价值的信任。特别的，加密数字签名算法使得用户可以对数字资产或数字资产的交易进行签名。利用合理的架构，数字签名也能用于解决双重支付问题。  上世纪80年代，当密码学越来越被人所熟知，并得到了越来越广泛的应用时，许多研究人员开始尝试使用密码学创建数字货币系统。这些早期的数字货币项目发行的货币通常由国家货币或者贵金属进行背书。  虽然这些数字货币系统也能运行，但是他们是中心化的，很容易被政府或者黑客攻击。早期数字货币与传统银行系统一样，利用中央清算机构定时去处理所有交易。不幸的是，这些数字货币系统大都都成了政府担忧的目标，最终因诉讼失败而消失了。也有因为母公司突然破产清算而悲壮倒闭的。为了应对反对者（不管是合法政府还是罪恶元素）的干扰，防止单点攻击，一个去中心化的数字货币系统成为必要。比特币就是这样一个系统，设计出来时就是完全的去中心化的，不需任何的中央集权机构，也不需要可被攻击且容易崩溃的单一控制节点。  比特币是几十年来密码学和分布式系统研究的巅峰之作，它汇集了四个方面的创新，形成了一个单一的强大组合。比特币系统由以下几个部分组成：   * 去中心化的点对点网络（比特币协议） * 公共交易账本（区块链） * 去中心化的基于数学的确定性的货币发行体系（分布式挖矿） * 去中心化的交易验证系统（交易脚本) |

**1.2 比特币历史**

2008年，一个化名为中本聪的人公开发表了一篇叫做《比特币：一个点对点数字货币系统》的论文，比特币从此出现在世人面前。中本聪结合之前发明的几种数字货币如b-money，HashCash等创建了一个完全去中心化的货币系统，它不依赖于任何中央机构进行货币发行或者交易结算、验证。其最主要的创新在于利用分布式计算系统（称之为“工作量证明”算法）来组织10分钟一次的全局“选举”，以使去中心化的网络达到对交易状态的共识。这个机制优雅的解决了双重支付问题，避免了货币能被多次消费的问题。之前，双重支付一直都是数字货币系统的弱点，以致不得不引入一个中央清算机构来完成交易清算。

比特币网络开始于2009年，基于中本发布的并被大量其他程序员修订过的参考实现发展而来。比特币发明以来，为比特币提供安全性和弹性保障的分布式计算已经实现了指数级的增长，现在其计算能力已超过了全世界最强大的超级计算机的处理能力。基于比特币与美元的汇率估算，比特币的全部市场容量介于50亿到100亿美元间。比特币网络迄今处理的最大的一笔交易额为1.5亿美元，瞬间就完成了传递和处理，并且产生没有任何费用。

中本聪自2011年4月起从公众视野中消失，将开发代码和建设网络的责任交给了一个活跃的志愿者小组。这个站在比特币后面的人（或者群体）的身份依然未知。但是，不管是中本聪还是任何其他人均无法对比特币系统进行控制，这个系统只依赖于完全透明的数学法则。发明本身是开创性的，并且已经在分布式计算、经济学、计量经济学的领域中产生了新的学科。

| **分布式计算问题的一个解决方案** |
| --- |
| 中本聪的发明也是对之前未能解决的分布式计算问题（拜赞庭将军问题）的一个实用方案。简单的说，问题在于如何在一个不可靠且存在潜在背叛风险的网络中交换信息并达成共识。中本聪的解决方案是在一个没有中央可信节点的情况下，利用工作量证明来达成共识，它标志着分布式计算科学的一个重大创新，它的适用性远远超过货币领域。比特币可以在去中心化的网络中达成共识，从而证明选举、彩票、资产注册、数字公证等等等等的公正性。 |

**1.3 比特币使用，用户，以及他们的故事**

比特币是一种技术，但它所代表的货币，本质上是人与人之间实现价值交换的语言。我们来看看使用比特币的人们，并通过他们的故事来了解几种常见的货币和协议使用场景。我们也将在本书中不断重复使用这些故事，以展示数字货币在现实生活中的用途，以及比特币中的各种技术是如何使这些用途成为可能的。

*北美低价值零售* 爱丽丝（Alice）住在北加州湾区，她从她的技术迷朋友中听说了比特币，也想开始使用它。我们将一路追随她的故事，从她了解到比特币，获得一些比特币，到花费比特币从帕洛阿尔托的鲍勃咖啡屋购买一杯咖啡。这个故事将从零售客户的角度介绍比特币软件，兑换，以及基本的交易过程。

*北美高价值零售* 卡洛（Carol）是旧金山一家画廊的老板，她以比特币计价出售昂贵的画作。这个故事将介绍高价值零售商面临的的51%共识攻击的风险。

*离岸合同服务* 鲍勃（Bob），帕洛阿尔托一家咖啡店的老板，正在建设一个新的网站。他与一个居住在印度班加罗尔的网站开发工程师高佩什（Gopesh）签订了合同。高佩什同意以比特币进行支付。这个故事将检验比特币在外包、合同服务以及国际汇款中的使用。

*慈善捐赠* 尤金妮娅（Eugenia）是菲律宾一家儿童慈善机构的负责人。近来，她听闻了比特币，希望通过比特币接触到一个全新的国内外捐献群体，以支持她的慈善事业。她研究了利用比特币将善款分发到需要的地方的方法。这个故事将展示利用比特币进行跨币种和国界的筹款过程以及利用开放账本实现慈善组织的透明化。

*进出口* 穆罕默德是（Mohammed）一个迪拜的电子产品进口商。他希望利用比特币从美国和中国进口电子产品到阿联酋，以加快进口的支付过程。这个故事将展示比特币是如何用于大型商业机构间的基于物理货品的国际支付的。

*比特币挖矿* 景（Jing）是上海的一个计算机工程专业的学生。他建了一个矿机在比特币网络上挖矿，利用他的专业特长增加收入。这个故事将检验比特币的“产业”基础：专业的设备用于保护比特币网络，并发行新的货币。

以上每个故事均基于那些正在利用比特币创造新的市场、新的产业，提出创新方案以解决全球经济问题的真实人物和真实产业。

**1.4 新手入门**

要加入比特币网络并开始使用这种货币，用户首先要下载一个软件或者利用一个web应用。由于比特币是一种标准，存在很多的比特币客户端的实现。当然，也有一个参考实现，称之为中本聪客户端，它作为开源项目由一个开发团队进行维护，是由中本聪开发的原型中继承而来的。

主要有三种形式的比特币客户端：

*完全客户端*

完全客户端，或者叫“完全节点”，是一个保存全部比特币交易历史（包含每个用户的每笔交易）的客户端，它管理用户的钱包，也可以直接在比特币网络上开始一笔交易。这种客户端类似于一个独立的email服务器，不需要依赖任何其他服务器或第三方服务就可以处理协议的方方面面。

*轻量级客户端*

轻量级客户端保存用户的钱包，它依赖于第三方服务器访问比特币交易和网络。轻量级客户端不保存完整的交易，所以，它必须信任第三方服务器以进行交易验证。就像email的客户端，连接到一个email服务器上访问邮箱，通过第三方与整个网络进行交互。

*web客户端*

web客户端通过浏览器访问，用户的钱包保存在一个第三方的服务器上。就像webmail，完全依赖第三方服务器。

| **移动客户端** |
| --- |
| 在手机上运行的移动客户端，比如那些基于安卓系统的客户端，可以是完全客户端，也可以是轻量级客户端或者web客户端。一些移动客户端与web或者桌面客户端同步，提供一个基于相同资金来源的跨设备的多平台钱包 |

客户端的选择基于客户对资金的控制意图。完全客户端提供了最高级别的独立控制，但是它也给用户带来了备份和安全的负担。另一方面，web客户端最容易安装和使用，但是由于安全和控制是与web服务提供方共享的，这就带来了交易风险。如果web钱包服务被盗用（实际上已经发生过多次），用户就会失去他们的所有资金。相反的，如果一个用户拥有完全客户端，却没有进行足够的备份，他们也可能会因为电脑故障而丢失自己的资金。

基于本书的目的，我们将从参考实现（中本聪客户端）到web钱包，演示各种可以下载的客户端。某些例子需要参考客户端，它不仅是一个完整的客户端，也向我们暴露了钱包、网络和交易服务的API接口。如果你准备研究比特币系统的编程接口，参考客户端是必须的。

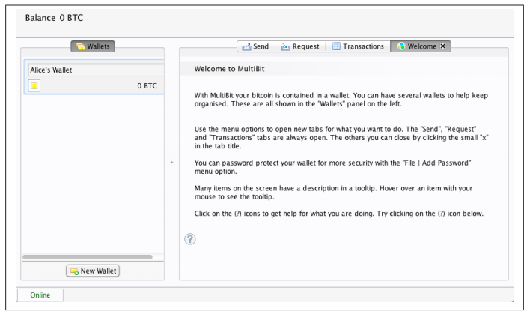
**1.4.1 快速入门**

爱丽丝，我们在第4页《比特币使用，用户和他们的故事》中介绍过的那位，不是一个技术宅，只是从朋友那听说了比特币。她通过访问<http://bitcoin.org>开始了她的比特币历程，在那儿，她看到了一个包含很多客户端列表的版单。根据<http://bitcoin.org>网站的建议，她选择了一个叫Mutibit的客户端。

爱丽丝根据<http://bitcoin.org>的下载链接，在自己的电脑上下载并安装了Multibit。Multibit提供了Windows，Mac OS和linux桌面等多个版本。

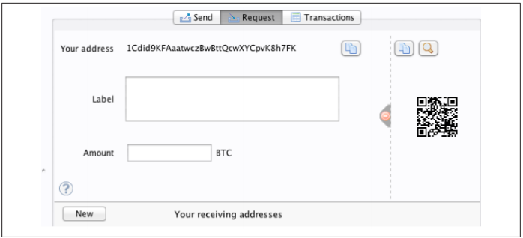
|  |  |
| --- | --- |
| bug | 比特币钱包必须由密码或口令进行保护。有很多坏蛋会尝试破解脆弱的密码，所以必须小心地选择一个无法轻易破解的密码。尽量使用混合了大小写字母、数字和其他符号的密码。避免使用生日、运动队名称等与个人相关的信息。也要避免很容易在字典中找到的任何语言的词语。如果可能，采用密码生成器生成一个完全随机的，至少12个字符长度的密码。记住：比特币是钱，而且可以在瞬间转移到世界上的任何地方。如果不好好进行保护，它是很容易被人窃取的。 |

当爱丽丝下载并安装了Multibit软件后，她点击运行，并出现了一个欢迎界面，如**图1-1**:



*图1-1 Multibit 比特币客户端欢迎界面*

通过点击Request选项卡，可以看到Multibit已自动为爱丽丝创建了一个钱包和比特币地址。如**图1-2**：



*图1-2 在Multibit客户端的Request选项卡下看到的爱丽丝的新比特币地址*

屏幕中最重要的部分是爱丽丝的比特币地址，就像一个email地址，爱丽丝可以将这个地址分享给别人，任何人也可以利用这个地址直接给爱丽丝转钱，转完后钱将直接进入爱丽丝的新钱包。屏幕上，地址看起来是一长串由字母和数字组成的字符串：1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK。挨着比特币地址的是一个二维码，它是另一种包含了相同信息的表现形式，可以被智能手机扫描识别。二维码就是窗口右方黑白图案构成的方框。爱丽丝可以通过点击比特币地址或者二维码边上的copy（拷贝）按钮将比特币地址或者二维码拷贝至剪贴板。如果单击二维码本身，它将放大显示以方便智能手机的摄像头扫描。

爱丽丝也可以将二维码打印下来，方便的交给其他人，免得他们输入那串冗长的字母和数字。

|  |  |
| --- | --- |
| notes | 比特币地址开始于1或3。就像email地址一样，可以在比特币用户间分享，别人可以通过这个地址给你的钱包直接发送比特币。不像email地址的地方是，只要你愿意，你可以随时创建新的比特币地址，所有这些地址都可以让收到的资金直接进入你的钱包。钱包就是简单的一堆地址和他们内含的解锁资金的密钥的集合。你可以通过每笔交易采用采用一个不同的地址以提高交易的隐私性。事实上，一个用户能创建的地址数量是没有限制的。 |

爱丽丝现在已经准备好可以使用她的新比特币钱包了。

**1.4.2 获取第一笔比特币**

现在还没有办法从银行或者外币兑换亭买到比特币。截至2014年，在大多数国家要购买到比特币都还是很困难的。你可以去一些专业的货币交易所，在那儿，你可以以一定的汇率利用当地货币买卖比特币。这些交易所的运营模式通常都是基于web的，主要包括：

*Bitstamp*

一个欧洲货币市场，它支持欧元(EUR)和美元(USD)等几种货币，通过电子转账的方式完成。

*Coinbase*

一个位于美国的比特币钱包和交易平台，在这个平台上，商家和客户可以通过比特币进行交易。Coinbase允许用户通过ACH系统将交易所账户与美元支票账户相连接，使得比特币的买卖变得非常简单。

这些加密货币交易所，是法币和加密货币的交汇点。它们同样需要符合国内国际的监管要求，并且经常被限定在一个国家或者经济区内，且指定使用该地区的法币。你指定的货币交易所只能使用你所在国家的货币，并且只能位于你的国家拥有司法管辖权的地方。与银行开户类似，你可能需要花上几天到几周的时间来开立使用这项服务的账户。他们要求你提供各种形式的身份证明以满足KYC（了解你的客户）和AML（反洗钱）等银行监管规定。一旦你拥有了一个比特币交易所账户，你就可以像使用代理账户买卖外币一样开始买卖比特币了。

Bitcoin Charts是一个提供价格行情以及大量货币交易所市场数据的网站，在这你可以找到更完整的列表。

以下是作为新用户获得比特币的四个其他渠道：

* 找到一个拥有比特币的用户，直接从他那买一些。这也是很多新用户最初获取比特币的方法。
* 使用类似localbitcoins.com这样的分类服务网站找到一个当地的卖家，并通过当面交易的方式用现金从他手里购买比特币。
* 以比特币计价的方式出售一项商品或服务。如果你是个程序员，可以出售你的编程技巧。
* 利用你所在城市的比特币ATM。利用CoinDesk的在线地图在你附近找到一个比特币ATM。

爱丽丝通过朋友介绍认识比特币的，因此当她在等待加州货币市场对她的账户进行验证和激活的过程中，也有很方便的途径可以获取她的第一个比特币。

**1.4.3 发送和接收比特币**

爱丽丝已经创建好了比特币钱包，现在她可以接收资金了。她的钱包软件随机生成了一个私钥（细节参看第63页的《私钥》）以及相应的比特币地址。这时候，她的比特币地址并没有让比特币网络知道，也没有注册到比特币系统的任何部分。她的比特币地址仅仅是一个数字，它与控制资金的密钥相关联。没有账户，也没有地址与账户间的关联关系。直到作为交易的价值接收方发布到比特币账本（区块链）前，该地址只是无数可能的比特币“有效”地址的一部分。一旦它和交易相关联，它就变成了网络中已知地址的一部分，而爱丽丝也就可以在公共账本上查看它的余额了。

爱丽丝在一个当地餐馆碰到她的朋友乔（Joe），就是他介绍爱丽丝进入了比特币世界，爱丽丝用美元跟乔交换了一些比特币。她带了一张打印好的比特币地址和二维码作为她钱包的标识。从安全的角度看，比特币地址无关敏感信息。它可以发到任何地方而不用担心对她的账户造成威胁。

爱丽丝希望换10美元的比特币，她不想在这项新技术上冒太多金钱上的风险。她给了乔10美元的现金以及打印出来的地址。然后乔就可以给他相应金额的比特币了。

接着，乔需要查出当前的汇率，从而把正确金额的比特币发给爱丽丝。有很多很多软件和网站提供当前的市场汇率。以下是几个比较流行的：

*Bitcoin Charts*

一个市场数据服务，提供世界范围内多个交易所的以本地货币标价的比特币市场汇率。

*Bitcoin Average*

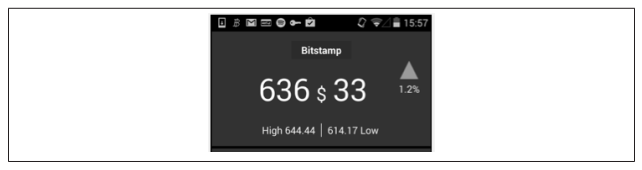
一个提供每种货币的“容量-权重-均值”的简单视图的网站。

*ZeroBlock*

一个免费的安卓和iOS应用，可以显示不同交易所的比特币价格。参看\*\*图1-3\*\*

*Bitcoin Wisdom*

另一个市场数据服务



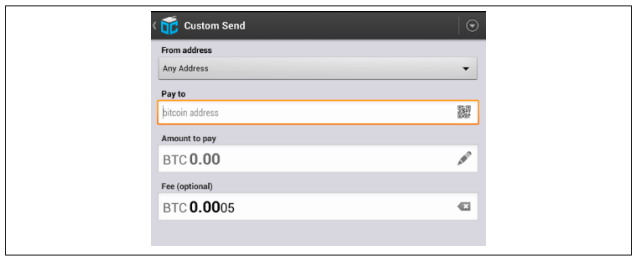
*图1-3 ZeroBlock， 一个比特币市场汇率应用（安卓、iOS）*

使用以上的某个网站或者应用，乔确定每个比特币的价格约为100美元。在这个汇率下，作为爱丽丝给他的10美元的对价，他需要给爱丽丝0.10个比特币，或者100毫币。

当乔确定了合理的交换价格后，他打开他的移动钱包应用，选择发送比特币。比方说，在安卓手机上选择“Blockchain（区块链）”移动钱包，他将看到屏幕上有两个必须的输入项，如图1-4。

* 交易目标地址
* 待发送比特币金额

在地址输入域中，有个类似二维码的小图标。它允许乔利用他的手机摄像头扫描二维码，这样就避免了手工输入爱丽丝的那串又长又难输的比特币地址（1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK）。乔点击二维码图标，激活手机摄像头，从爱丽丝给他的打印好的钱包上扫描到了二维码，并自动填充到比特币地址字段上。乔可以从中挑几个字符与爱丽丝打印出来的地址进行比对，以确认是否正确。



乔然后输入比特币交易金额，0.10比特币。他认真地进行了检查以确定输入了正确的金额，因为他准备做的是真正的金钱的传递，任何错误都会带来实打实的损失。最后他按下了发送按钮，将交易发送出去。这样，乔的手机比特币钱包构造了一个交易，这个交易将发送0.10比特币给爱丽丝提供的地址，资金的来源是乔的钱包，采用乔的私钥进行了签名。这告诉比特币网络，乔已经授权从他的地址转账到爱丽丝的新地址。随着交易通过点对点协议进行的传输，它很快就广播到了比特币网络中。不到一秒，大部分有良好连接的节点已经接收到了交易，并且第一次看到了爱丽丝的新地址。

如果爱丽丝随身带着智能手机或者笔记本电脑，她也能看到这笔交易。比特币账本（一种持续增长的，记录所有已发生的比特币交易的文件）是公开的，意味着她只需要关注她自己的地址，查看是否有资金发送到该地址。她可以在blockchain.info网站上输入她的地址，很轻易就能做到这点。网站会向她展示一个页面，这个页面包含了从这个地址发出或接收到的所有交易的信息。如果爱丽丝正在查看这个页面的话，在乔点击发送后，页面很快就会显示出最新的一笔乔转给她的0.10比特币的交易信息。

| **确认** |
| --- |
| 最初，爱丽丝的钱包会显示乔发出来的交易处于“未确认”状态。意思是交易已经广播到网络中，但是还没有被包含到比特币交易账本（称之为区块链）中。为了被包含进去，交易需要被矿工挑出，并打包进一个交易区块中。在差不多10分钟内，当一个新的交易区块被创建出来时，已包含入区块的交易被网络接受为“已确认”状态，这笔钱已经可以使用了。交易立即就可以看到，但是只有被包含进新挖出来的区块中，才能得到所有人的“信任”。 |

爱丽丝现在是光荣的比特币的所有者了，她拥有了0.10个可以使用的比特币。在下章，我们将看到她使用比特币的第一次购买行为，并会更细致地了解交易底层及交易传播技术。

**第2章 比特币是如何工作的**

**2.1交易，区块，挖矿以及区块链**

比特币系统不像传统银行系统或支付系统，它基于一种去中心化的信任。与传统中央权威信任机构相反，比特币的信任是系统的一种自然属性，源于比特币系统不同参与者间的交互。在本章中，我们将从较高的层面审视比特币系统，通过跟踪一笔交易，看它是如何在分布式共识机制下变为“被信任”和被接受，并最终被记录到区块链上（记录所有交易的分布式账本）。

文中的每个例子都是在比特币网络上真实执行过的交易，模拟了资金在用户间（乔，爱丽丝，鲍勃）转移的交互过程，即比特币从一个钱包进入另一个钱包的过程。为了在比特币网络和区块链中跟踪交易，我们可以利用区块链浏览器网站来可视化地展现每个步骤。区块链浏览器是一个web应用，起到了比特币搜索引擎的作用，通过它，可以搜索地址、交易、区块并查看他们间的关系和流程。

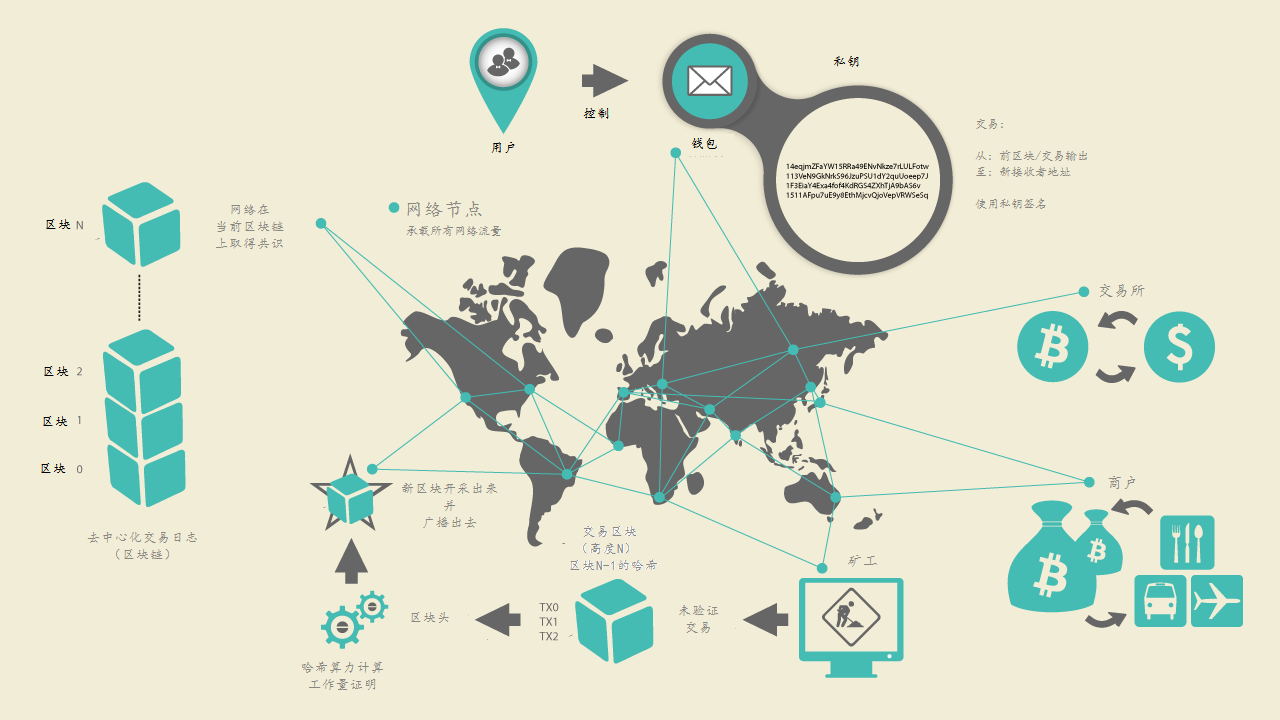
流行的区块链浏览器包括：

* Blockchain info
* Bitcoin Block Explorer
* insight
* blockr Block Reader

以上每个网站的区块链浏览器均带有搜索功能，可以依据地址，交易哈希，区块号码进行搜索，每个网站的搜索结果在比特币网络和区块链中都是等价的。在每个例子中，我们都会提供一个URL，让你直接找到相应的入口，以便对细节进行进一步研究。

**比特币概览**

在下述概览图（图2-1）中，我们可以看到，比特币系统由用户、交易和矿工组成，用户拥有钱包，钱包中管理着用户的密钥；交易在网络中传播；矿工通过竞争性计算来创建共识区块链，它是所有交易的权威账本。在本章中，我们将从较高的层面，跟踪一个交易在网络中的传播过程，并检视由此引起的比特币系统不同部分间的交互。接下来的章节中，我们将深入了解钱包、挖矿和商户系统后的更多技术细节。



*图2-1 比特币概览*

**购买一杯咖啡**

爱丽丝，这位我们之前已经介绍过的主角，是一个新用户，她刚刚获得了她的第一笔比特币。在第9页的《获取第一笔比特币》中，爱丽丝遇到了她的朋友乔，并用现金跟他交换了一笔比特币。那笔由乔创建的交易往爱丽丝的钱包中存进了0.10的比特币（BTC）。现在爱丽丝将进行她的第一笔基于比特币的零售交易---在鲍勃位于加州帕洛阿尔托的咖啡店中购买一杯咖啡。鲍勃的咖啡店最近刚开始接受比特币支付，他在销售系统上添加了比特币的选项。咖啡店的价格是以当地货币（美元）标注的，但是在收银处，客户可以选择以美元或者比特币进行支付。销售系统将自动依据主流市场的汇率，将美元价格转换为比特币价格，并同时显示两种价格，销售终端还会在屏幕上显示一个带有本笔支付请求的二维码：

合计:

$1.50 美元

0.015 比特币



*图2-2 支付请求二维码（提示：扫扫看！）*

支付二维码按照BIP0021的要求，将以下URL进行编码：

bitcoin:1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA?

amount=0.015&

label=Bob%27s%20Cafe&

message=Purchase%20at%20Bob%27s%20Cafe

URL的组成元素

比特币地址：1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA

支付金额：“0.015”

接收人地址的标签："鲍勃咖啡店”

支付描述：“在鲍勃咖啡店买东西”

不像一个只包含目标地址的二维码，支付请求二维码是一个二维码编码的URL，包含了目标地址，支付金额，通用的交易描述，如“鲍勃咖啡店”。比特币钱包软件可以使用这些信息预填充用于发送支付指令的信息，同时也可以人工可读的方式显示给用户。你可以使用一个钱包软件扫描二维码，看看爱丽丝看到了什么。

鲍勃说：“共1.5美元或者15毫比特币。”

爱丽丝用她的智能手机在屏幕上扫了一下二维码，手机显示需要支付0.0150比特币给鲍勃咖啡店，她点了发送，授权了这笔支付交易。几秒钟后（与刷信用卡授权的时间差不多），鲍勃在收银机上看到了这笔交易，交易完成了。

在接下来的章节中，我们将剖析这笔交易的细节，了解爱丽丝的钱包是如何构建这笔交易，如何将它广播到网络中，交易如何获得确认，最后鲍勃是怎么才能在后续交易中使用这笔钱的。

比特币网络支持将比特币拆分成小额进行交易，从千分之一比特币(一毫比特币）到一亿分之一比特币（通常称作聪）都可以。在本书中，我们用“比特币”指代任何数量的比特币货币，从最小的单位（1聪）到全部可被挖出的数量（2100万）。

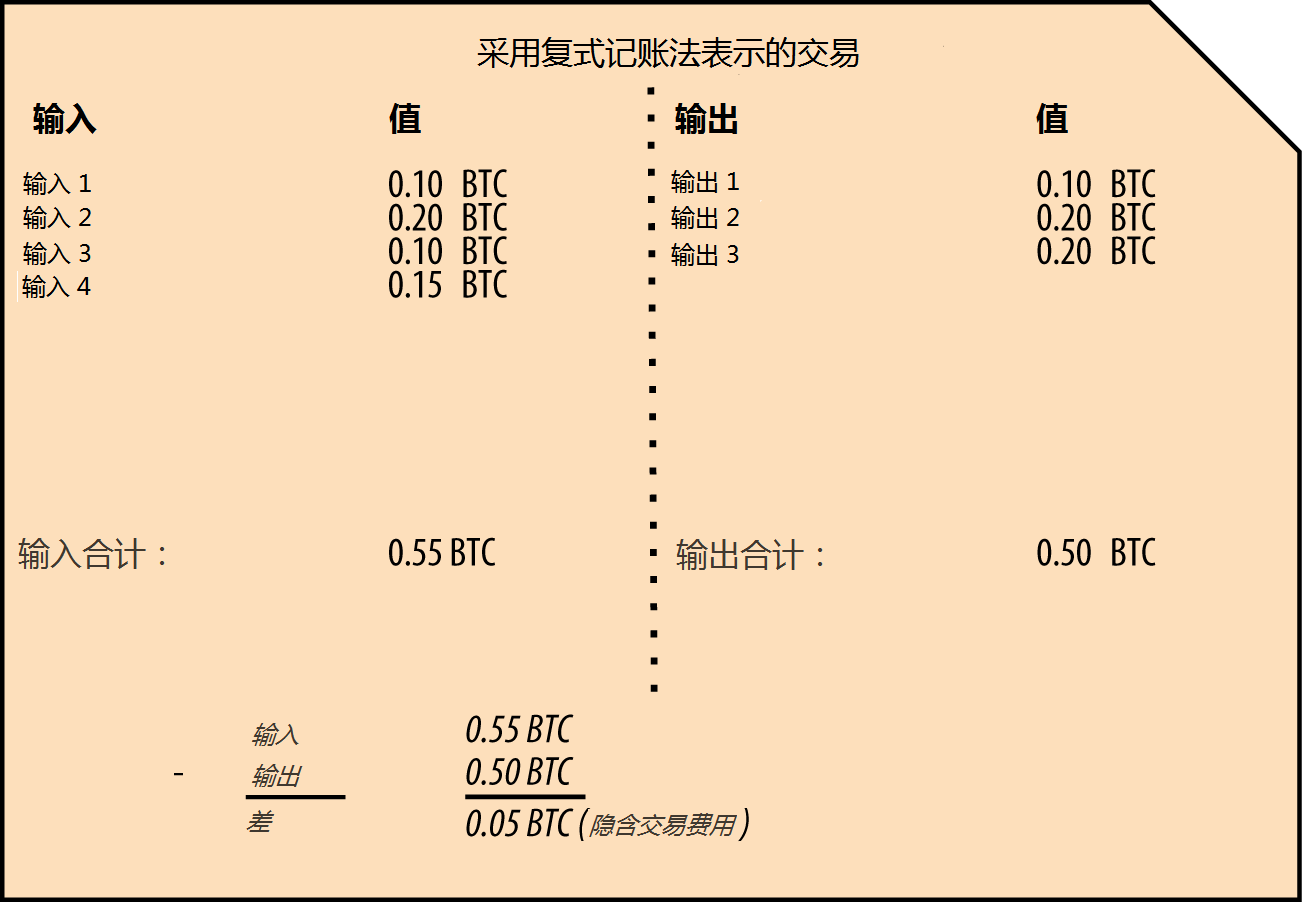
**比特币交易**

简单的说，一个比特币交易，就是告诉网络，某个拥有一定数量比特币的用户已经授权将这笔比特币转让给另一位用户。新的所有者可以通过另外一笔授权转让交易来使用这些比特币，以此类推，形成一个所有者转换的链条。

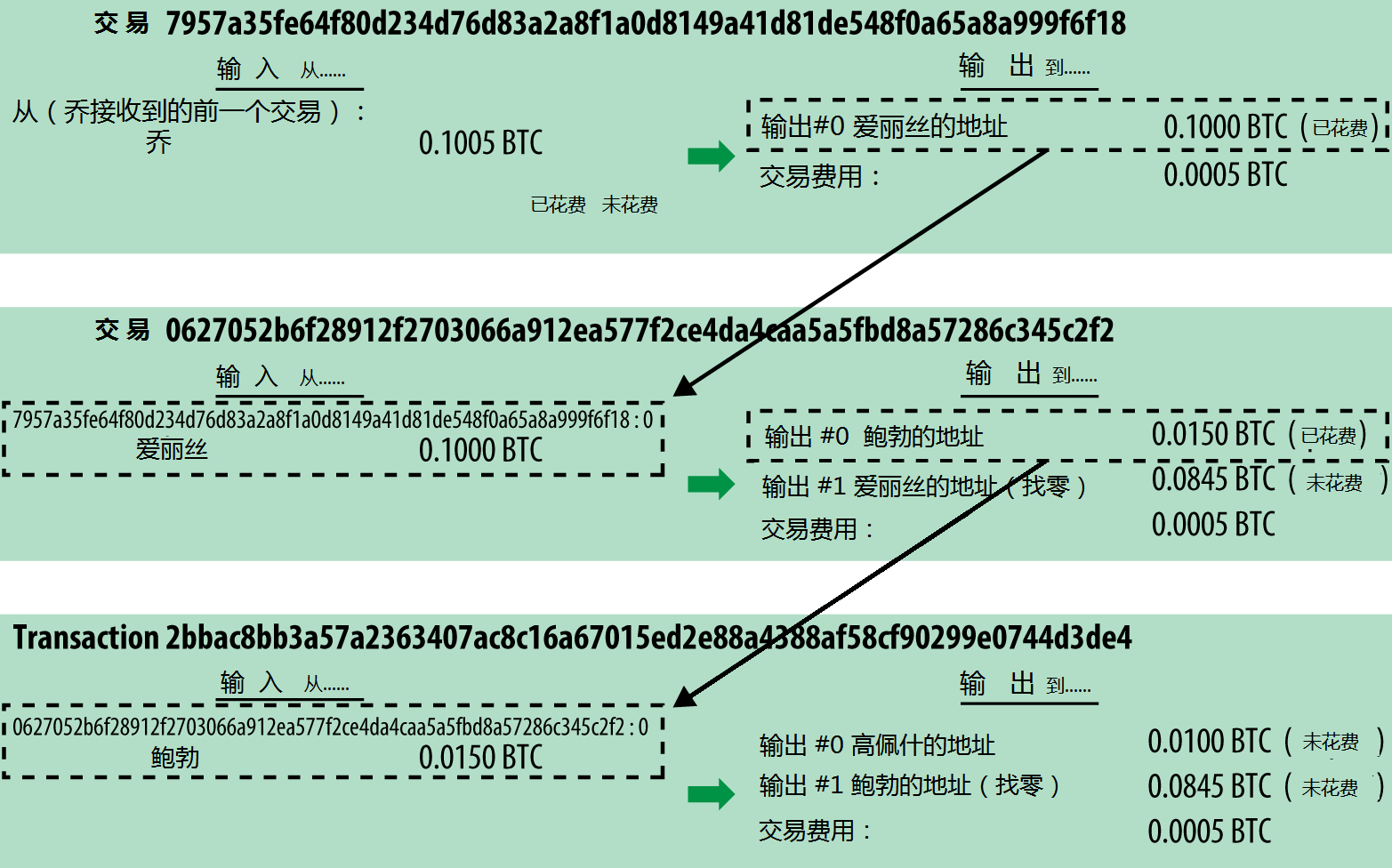
交易就像复式账本的一笔笔记录，每个交易均包含一到多条的“输入”---这是比特币账户的借方。另一方面，每笔交易也包含了一到多条的“输出”---这是比特币账户的贷方。输入和输出（借和贷）加起来不要求相等，实际上，输出加起来的和应稍小于输入的和，这个差额就是隐含的“交易费用”，这笔小额费用归将交易归集到账本的矿工所有。以复式账本表示的比特币交易如图2-3。

交易同样包含每笔待转让比特币（交易输入）的所有权证明，以所有者数字签名的方式来表示，数字签名可以被任何人独立验证。在比特币的术语中，“消费”就是签署一笔交易，将所有者从前序交易中获得的价值权益转让给以比特币地址代表的新所有者。

交易将价值从交易输入移动到交易输出。交易输入是价值的来源，通常是上一笔交易的输出。交易输出将一笔与私钥关联的价值赋给一个新用户。目标密钥称为*安全锁*。在未来的交易中，需要通过签名来获取这笔资金。一笔交易的输出作为新交易的输入，这样，随着价值不断从一个地址转移到另一个地址，形成了一条所有权的链条（见**图2-4**）。



*图2-3 采用复式记账法表示的交易*

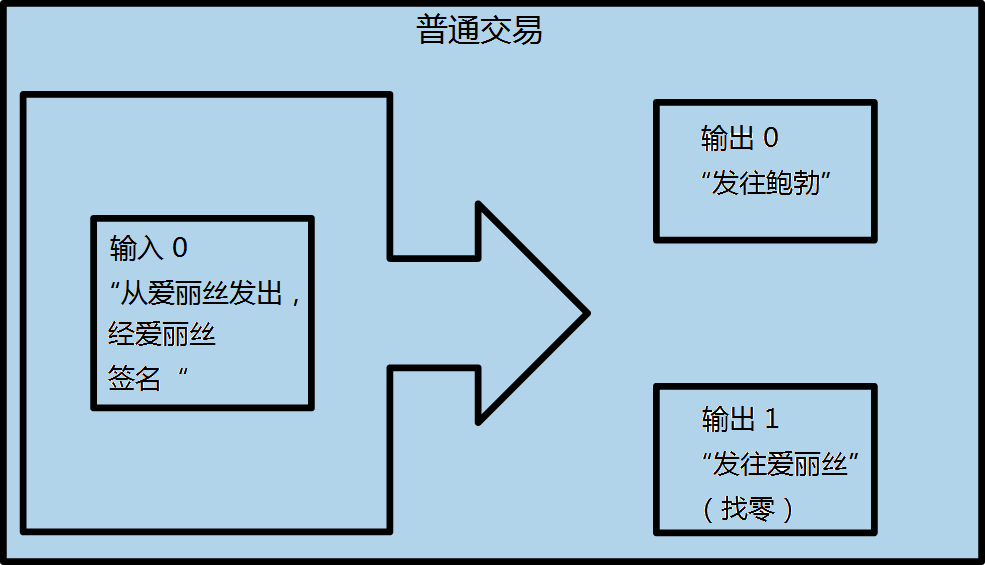


*图2-4 一个交易的链条，在链上一个交易的输出成为下一笔交易的输入。*

爱丽丝支付给鲍勃咖啡店的交易使用了上一笔交易的输出作为这笔交易的输入。在上一章中，爱丽丝用现金从她的朋友乔那里换到了一笔比特币。那个交易的一笔资金被爱丽丝的密钥锁定（阻碍）。她向鲍勃咖啡店支付咖啡费用的新交易中，引用了上笔交易的输出作为本笔交易的输入，输出则包含两部分，一部分是支付咖啡费用，另一部分用于找零。交易形成了一个链条，最新交易的输入对应上一笔交易的输出。爱丽丝的密钥提供的签名解锁了前笔交易的输出，向比特币网络证明了她对这些资金的所有权。她在交易中附上鲍勃的地址，形成一个“阻碍”，限制鲍勃必须使用签名才能花费这笔资金。这个过程展示了价值在爱丽丝和鲍勃中转移的过程。这个从乔到爱丽丝在到鲍勃的交易链见图2-4。

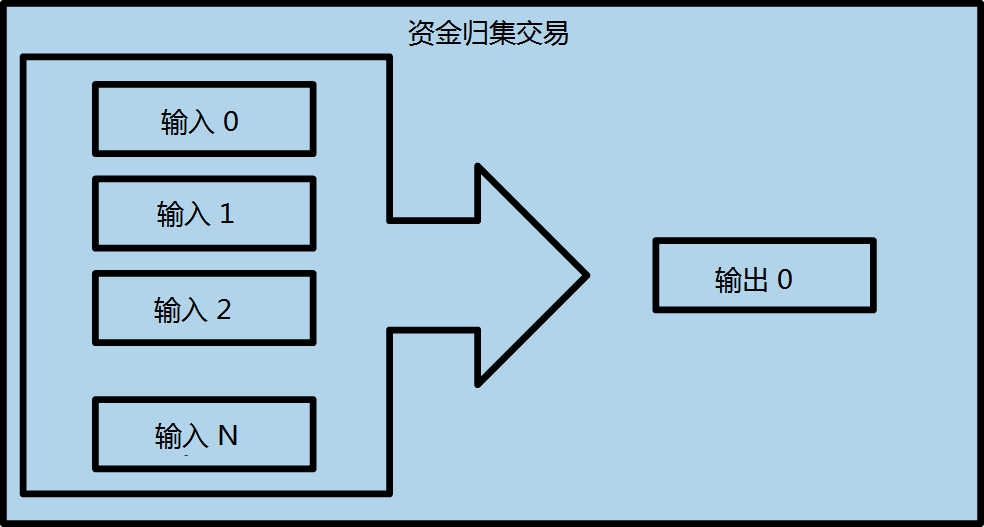
**常见交易形式**

最常见的交易就是从一个地址到另一个地址的支付，它通常还包含需要返还给初始所有者的零钱。这种类型的交易包含一个输入和两个输出，参见图2-5.



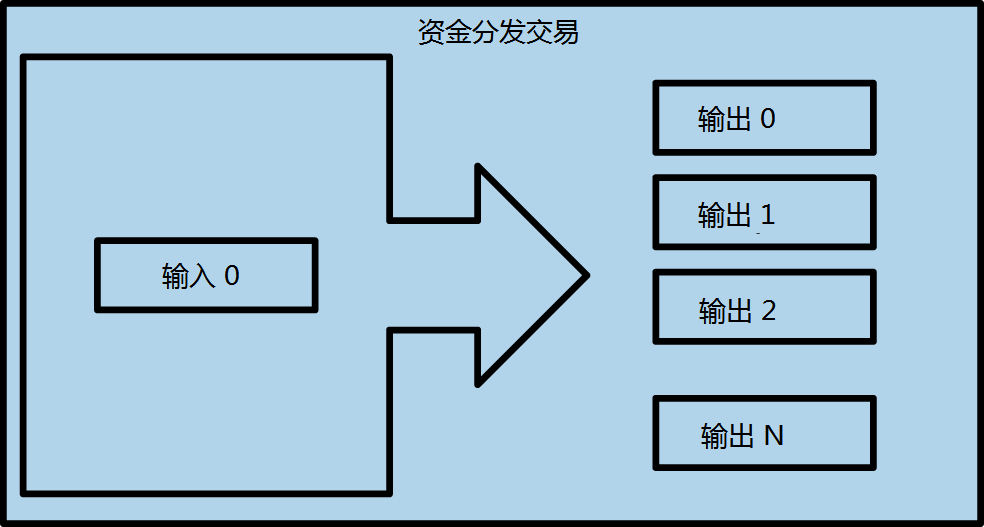
*图2-5 最常见普通交易*

另外一种常见交易形式是绑定几个输入，形成一个输出（见图2-6）。这与现实生活中将零钱换为大额钞票的场景类似。这种交易通常是钱包软件对交易中找回的一堆零钱进行清理的情况。



*图2-6 资金归集交易*

最后，还有一种在比特币账本中常见的交易形式，将一个交易输入分配给多个输出，每个输出代表不同接收者（见图2-7）。这种类型的交易有时是商业实体用于分发资金，比如给员工发工资。



\*图2-7 资金分发交易 \*

**创建交易**

钱包软件知道如何选择合适的输入和输出以创建符合爱丽丝要求的交易。爱丽丝要做的只是指定一个接收人和交易金额，不用操心具体的细节，钱包软件会自动完成剩下的工作。很重要的一点，钱包软件即使在离线的情况下也可以创建交易。就像在家里写一张支票，然后将其装进信封寄给银行，交易不需要在连接到比特币网络的情况下进行创建和签名。只有在执行交易时才需要将交易发送到网络。

**获得正确的输入**

爱丽丝的钱包软件首先必须找到足以支付鲍勃资金的交易输入。大部分钱包软件会保留一个“未花费的交易输出”的小型数据库，由钱包所有者的私钥锁定（阻碍）。因此，爱丽丝的钱包含有那笔她用现金跟乔交换比特币的交易输出的副本（参见第9页《获取第一笔比特币》）。一个运行在完全模式客户端下的钱包软件，实际上包含了网络上所有交易的“未花费的输出”。这使得钱包软件不仅能快速构建交易输入，也能验证一个新来的交易，判断其输入是否有效。由于完全客户端需要耗费非常多的存储空间，实际上大部分用户的钱包只运行一个轻量级客户端，只能用于跟踪用户自己的未花费输出。

如果一个钱包软件没有维护完整的“未花费的输出”，它可以使用不同供应商提供的API接口向比特币网络询问这些信息，或者也可以使用JSON RPC接口向一个完全节点询问相关信息。例2-1展示了一个RESTful API请求，它以HTTP GET命令的方式构造一个请求发往特定的URL地址。这个URL将根据请求中的地址信息返回所有“未花费的输出”信息，供任何需要使用这些信息构造交易输入的应用使用。在这里我们使用简单的命令行模式的HTTP客户端cURL来发送请求并获取应答。

*例2-1 获取爱丽丝比特币地址下所有未花费的输出*

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

*例2-2 请求应答*

{

"unspent\_outputs":[

{

"tx\_hash":"186f9f998a5...2836dd734d2804fe65fa35779",

"tx\_index":104810202,

"tx\_output\_n": 0,

"script":"76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",

"value": 10000000,

"value\_hex": "00989680",

"confirmations":0

}

]

}

例2-2的应答显示，只有一个归属于爱丽丝的地址1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK下的“未花费的输出”（尚未兑现）。应答还包含了产生这笔输出的交易引用（乔发起的支付交易），以单位聪表示的交易金额是1000万聪，也就是0.10比特币。利用这些信息，爱丽丝的钱包软件就可以构建出一个新交易向新地址发送资金了。

[点击http://bit.ly/1tAeeGr 查看乔到爱丽丝的交易](http://bit.ly/1tAeeGr)

就像你看到的，爱丽丝钱包中唯一的未花费输出已足够支付一杯咖啡的费用。如果这个条件不能满足，钱包软件就不得不去翻找所有小额未花费输出凑够这笔交易输入，就像从手提袋中翻找足够的硬币来凑够一杯咖啡的钱。在以上两种情况下，钱包软件都可能需要在交易输出中找回一些零钱，我们将在下节讨论。

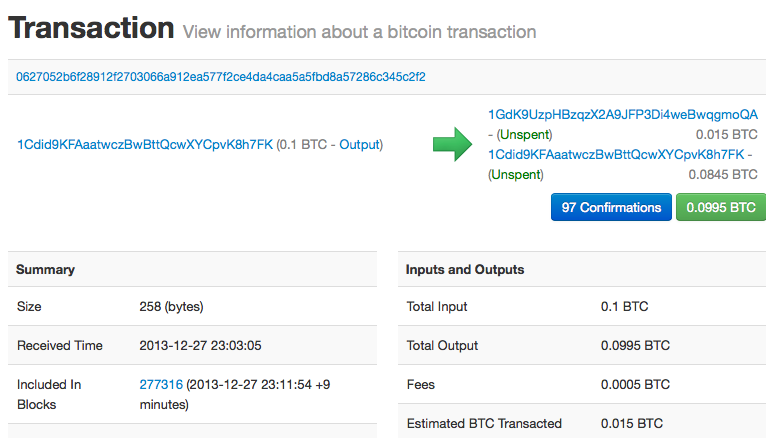
**创建输出**

交易输出以一种脚本的形式创建一个针对特定价值的阻碍，只能通过解决脚本问题来解除阻碍，从而兑现这笔输出。简单的说，爱丽丝的交易输出包含一个脚本，脚本大意是这样的：“这个输出将支付给那个能提供与鲍勃的公开地址匹配的签名的人”。由于只有鲍勃拥有与其地址匹配的密钥，所以只有鲍勃的钱包可以提供这样一个签名来兑现这笔输出。爱丽丝就以要求提供签名的方式，锁定了一笔输出价值。

这笔交易还包含另一部分，因为爱丽丝的资金是以0.10比特币表示的一个输出，超过了一杯咖啡0.015比特币的金额。爱丽丝需要拿回0.085比特币的找零。爱丽丝的找零操作是钱包在生成给鲍勃的支付交易时一并产生的。本质上看，爱丽丝的钱包将她的资金分成两笔支付，一笔给鲍勃，剩下的交还她自己。她可以在下一笔交易中使用这个找零输出。

最后，为了这笔交易尽快被网络执行，爱丽丝的钱包应用将添加一小笔的交易费用。这个金额不是显式的，它隐含于交易输入输出的差值中。假如交易中找零金额不是填0.085，而是用0.0845作为交易第二个输出（找零输出），这样就有0.005比特币（半毫比特）的剩余。输入的0.10比特没有被两个输出完全花费完，因为它们加起来不到0.10。输入输出的差额就构成了交易费用，矿工们将这笔交易加入到区块，并放入区块链账本的过程中会收集这些交易费用作为它们挖矿的报酬。

交易结果可以通过一种叫做区块链浏览器的web应用来查看，用普通浏览器访问：http://bit.ly/1tAeeGr ，结果如图2-8：



*图2-8 爱丽丝给鲍勃咖啡店付款的交易*

[点击http://bit.ly/1tAeeGr 查看爱丽丝给鲍勃咖啡店付款的交易](http://bit.ly/1u0FIGs)

**将交易添加到账本上**

爱丽丝的钱包软件创建的交易共258字节长，它包含用以证明她的资金所有权的信息以及资金接收者的信息。现在，交易必须发送到比特币网络上并使之成为分布式账本（区块链）的一部分。在接下来的一节中，我们将看到一个交易如何变成新区块的一部分，以及区块是如何被“挖出来”的。最后，我们还将看到这个新区块被加入到区块链后，是如何随着区块数量的增长而变得越来越可信的。

**发送交易**

因为交易已包含所有用于后续处理的信息，所以不要关心它是如何或者从哪里传入比特币网络的。比特币网络是一个点对点的网络，每个比特币客户端通过与不同的客户端连接，成为网络的参与者。比特币网络的目标就是把交易和区块广播给所有的参与者。

**如何传播**

爱丽丝的钱包软件可以将新交易发送到所有通过互联网与它相连的客户端，不管它是通过有线、WiFi，还是通过移动网络相连。她的比特币钱包不一定非要与鲍勃的钱包直接相连，她也不必非要使用咖啡店提供的互联网接入，虽然这两种方式也没什么不可以。任何比特币网络节点（其他客户端）接收到之前未见过的有效交易时将立即将其转发给与它相连的其他客户端。这样，交易就很快在这个点对点网络中传播开来，在短短几秒内即可到达大部分节点。

**鲍勃的视角**

如果鲍勃的钱包软件与爱丽丝的直接相连，鲍勃的钱包软件将是第一个接收到交易的节点。不过，即使爱丽丝的钱包通过别的节点发送，交易也会在短短几秒内到达鲍勃的钱包。鲍勃的钱包会立即将爱丽丝的交易识别为一笔消费支付交易，因为交易含有需要鲍勃的密钥进行解锁的输出。鲍勃的钱包软件也能独立确认这笔交易是有效封装的，使用了之前未花费的输出，并且附带了足够的交易费用使其能够被下个区块包含。基于此，鲍勃可以认定这个交易会很快被确认并添加到区块中，被骗的风险很小。

关于比特币的交易，有个常见认识误区，认为交易必须等待10分钟被新区块包含，或者等待一小时得到6个确认后才是有效的。虽然确认可以确保交易被全网接受，但是这种延迟对于小额支付，比如一杯咖啡，其实是没有必要的。商户对这种小额交易可以直接接受，其风险不会比一笔没有使用身份证或签名的信用卡交易风险更大，而现在商户一般都接受这种信用卡支付方式。

**比特币挖矿**

交易现在被广播到了网络中，在没有被确认并通过挖矿包含到区块前，它还没有成为共享账本（区块链）的一部分。关于挖矿，在第8章中将有详细解释。

比特币系统的信任是建立在计算的基础上的。交易打包进区块需要巨大的计算量来证明，但是验证这个证明只需很少的计算量。挖矿过程在比特币系统中有以下两个目的：

* 挖矿过程在每个新区块中创建新的比特币，就像中央银行发行货币。每个区块创建的新比特币是固定的，随着时间推移，这个数量会逐渐减少。
* 挖矿过程创造信任，需要确保只有足够运力投入到包含这些交易的区块后交易才能被确认。更多的区块意味着更多的计算量投入，也意味着更多的信任。

挖矿的过程就像一个巨大的竞争性数字拼图游戏，当有人找到一个解决方案时，游戏就重新开始，而游戏的难度也会自动进行调整，使得每找一个解决方案的时间大致维持在10分钟左右。想象一个巨大的数字拼图，高度有几千行，宽度有几千列。如果我给你看已经填充好的拼图，你可以很快的验证有没有错误。但是，如果只填了一部分，剩下的都是空白的，那就需要花费大量的时间才能解决。数字拼图游戏的难度可以通过调整尺寸（增减行列数）来调节，但是不管尺寸大小，其确认过程都很简单。比特币中用的“拼图”是建立在密码学哈希算法之上的，它展现了与拼图类似的特性：它也是不对称的，很难解决却很容易验证，而且它的难度也可以调整。

在第4页的《比特币的使用，用户和他们的故事》中，我们介绍过景，一个上海的计算机工程专业学生。景是以矿工的身份参与到比特币网络中的。每10分钟左右，景与成千上万的矿工一起展开一场查找一个区块解决方案的全球竞赛中。为了找到一个解决方案（称之为工作量证明），全网每秒要进行几亿亿次的哈希计算。比特币中的工作量证明算法是采用SHA256加密哈希函数不断地对区块头和一个随机数进行哈希计算，直到找到一个与预设的模式匹配的方案。第一个找到这个解决方案的矿工将赢得这一回合的竞争，随即将这个区块发布到区块链当中。

景从2010年开始挖矿，他使用了一台非常快的桌面电脑来查找新区块的工作量证明。随着越来越多的矿工加入到比特币网络中，挖矿的难度急速增长。很快的，景和其他矿工将他们的电脑升级到了更专业的硬件，比如那些在游戏电脑或终端中使用的高端专用图形处理单元（GPU）。截止到写这本书的时候，难度已经非常非常高，为了保证有利可图，只能使用特定用途集成电路芯片（application-specific integrated circuits， ASIC）来进行挖矿，这些ASIC芯片将几百个挖矿算法集成到硬件中，并使他们在一个芯片内并行计算。景加入了一个“矿池”，就像彩票池，允许多个参与者参与其中，共同工作并共享收益。景现在使用两个USB连接的ASIC机器每天24小时挖矿，他通过出售挖矿获得的比特币来支付购买硬件的费用，并获取一定收入。他的电脑运行着一个bitcond，比特币客户端的参考实现，作为他的专用挖矿软件的后端。

**交易区块挖矿**

一个在网络上传播的交易直到成为全局分布式账本（区块链）的一部分才真正得到确认。平均每隔10分钟，矿工会创建一个包含上个区块以来产生的所有交易的区块。新交易不停的从用户钱包或者其他应用中流入网络。当这些交易被其他节点捕获时，就会被加入一个各自维护的临时未验证交易池中。矿工创建新区块时，他们将未验证交易池中的交易取出，并入新建区块，然后尝试解决一个极为困难的问题（即工作量证明）来证明这个区块的有效性。挖矿的过程我们将在173页《介绍》中详细说明。

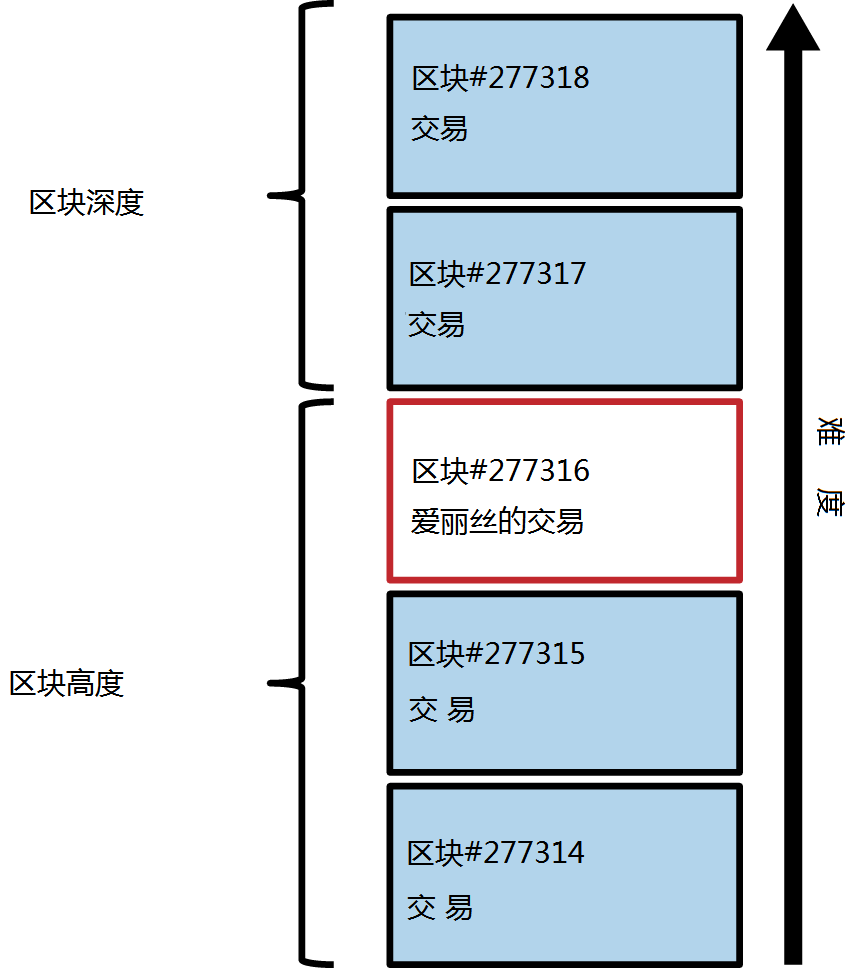
交易根据费用优先原则及其他一些规则被顺序加入到新区块中。当矿工从网络中接收到上一个区块时，他意识到已经在上一轮竞争中失败了，所以立即开始新区块的挖矿过程。他首先创建一个新区块，填上交易以及上个区块的指纹，然后开始计算这个新区块的工作量证明。矿工还会在区块中包含一个特殊的交易，这个交易向他自己的比特币地址发送一个新创建的比特币作为奖励（当前每区块25比特币。译者：此为作者写本书时的金额，现在是12.5比特币）。如果他找到一个工作量证明使得区块有效，他即赢得了这个奖励，因为他的成功挖出的区块被加入到全局区块链中后，他加入的奖励交易也变得可用了。景，由于加入了矿池，他把挖矿软件的新区块奖励地址设置为矿池的地址。在矿池中，一旦在上一轮挖矿竞争中胜出，它将把上一轮的奖励按照贡献工作量的大小分配给矿工。

爱丽丝的交易被网络节点提取并放进未验证交易池。因为交易包含了足够的费用，它会被放进景所在矿池的新建区块当中。交易从爱丽丝的钱包提交后大概5分钟，景的ASIC矿机找到了这个区块的工作量证明，并将其发布为277,316号区块，这个区块还包含419个其他交易。随着新区块在网络中的发布，其他矿工将立即对其进行验证并开始新一轮的挖矿竞争以生成下一个区块。

你可以通过这个地址：https://blockchain.info/block-height/277316 查看包含了爱丽丝交易的区块。

几分钟后，第277,317号区块又被别的矿工挖出。由于这个新区块是基于上一个包含了爱丽丝交易的区块 (277,316号区块)的，它在原有区块的基础上进行了更多的计算，因此，能进一步强化对那些交易的信任。包含爱丽丝交易的区块被认为是对该笔交易的一个确认。基于这个区块每产生一个新区块，就是是对交易的一次额外确认。由于新区块一个个的叠加在原有区块之上，这使得推翻原有交易的难度呈指数级增长，这样就保证了交易可信程度越来越高。

在**图2-9**中，我们可以看到包含爱丽丝交易的277,316号区块。在它之下有277,316个区块（包括区块#0），这些区块互相连接，直到0号区块---称之为创世区块，形成一个区块的链表（区块链）。随着时间的推移，区块的“高度”不断增长，每个区块以及整个链的计算难度都不断增加。由于在这个越来越长的链表上，不断叠加新的计算，包含爱丽丝交易的区块之后挖出的区块成了该笔交易的额外保证。按照惯例，任何经过6次确认后的区块被认为是不可撤销的，因为要撤销并重新计算6个区块需要极大的计算量。我们将在第8章审查挖矿的过程以及它创建信任的机制。



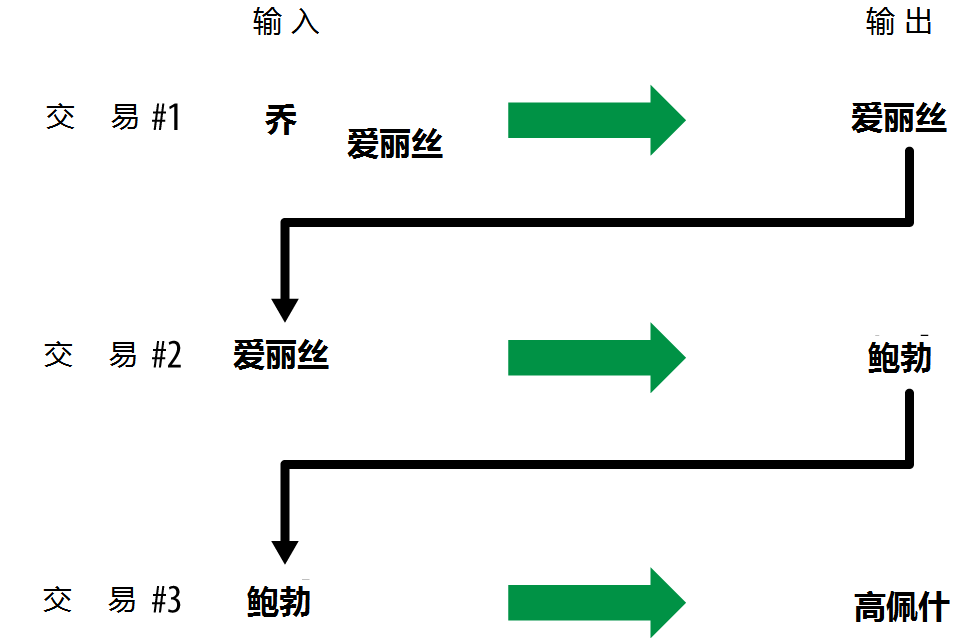
*图2-9 爱丽丝的交易包含在第277,316号区块中*

**花费交易**

既然爱丽丝的交易已经作为区块的一部分被包含进区块链中，它就成了比特币分布式账本的一部分，对所有比特币应用可见。每个比特币客户端都可以独立验证这笔交易，确认它是有效且可花费的。完全节点客户端可以从比特币在区块中创建的时刻起跟踪这笔资金的流转过程，直到他们到达鲍勃的地址。轻量级客户端只能确认交易是否在区块链中，有几个区块在它之后被挖出，从而获知网络已确认它的有效性并接受它，轻量级客户端的这种操作被称之为简化支付验证（参看第147页《简化支付验证节点（SPV）》

鲍勃现在可以引用这笔及其他笔交易的输出作为交易输入，生成他自己的新交易，并将资金转给新的所有者。举例来说，鲍勃可以用爱丽丝买咖啡的钱来支付承包商或供应商的费用。大多数情况下，鲍勃的比特币软件需要归集多个小的支付交易才能完成一笔较大的支付，或许是将一天的比特币收入集中到一个交易中。这将把不同的支付归集到一个作为咖啡店通用“经常”账户使用的地址下。参看图2-6了解整合交易。

当鲍勃将从爱丽丝及其他客户收到的款项花费出去的时候，他扩展了交易链条，结果是，新的交易加入到全局区块链账本，所有人都能看到和信任。我们假定鲍勃向他的网站设计师，班加罗尔的高佩什支付新网页的设计费用。现在交易链条看起来像**图2-10**这样。



*图2-10 爱丽丝的交易成为从乔到高配什的交易链条的一部分*

**第3章 比特币客户端**

**比特币核心：参考实现**

你可以从<http://bitcoin.org>下载比特币的参考客户端*比特币核心*，也叫“中本聪客户端”。这个参考客户端实现了比特币系统的所有功能，包括钱包，一个交易验证引擎（用于对全部交易账本，即区块链的全量副本进行交易验证），以及一个用于接入点对点比特币网络的完全网络节点。

在[bitcoin.org](http://bitcoin.org)网站的“选择钱包”页面，点击“比特币核心（Bitcoin Core）”下载参考客户端。基于你的操作系统，你可以下载到不同的安装包可执行文件。对于Windows系统，有zip压缩包和.exe可执行程序可供下载。对于MacOS，则是一个.dmg磁盘映像。对于Linux版本，包括Ubuntu系统的PPA包，或者tar.gz档案。[bitcoin.org](http://bitcoin.org)网页中列出的建议客户端见**图3-1**。



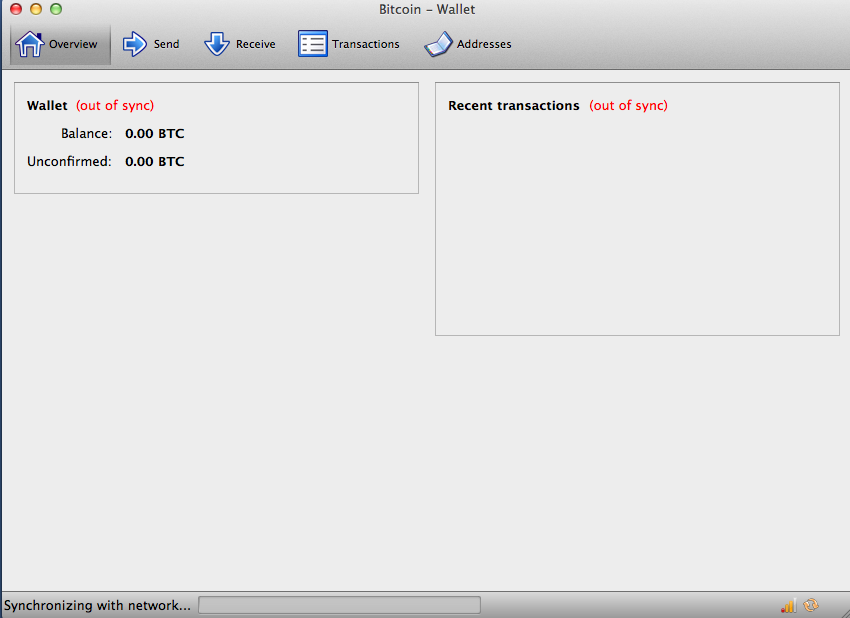
\*图3-1 从bitcoin.org选择下载合适的比特币客户端\*

*注：所选截图为当前最新客户端列表*

**首次运行比特币核心**

如果你已经下载了安装包，比如.exe，.dmg或者PPA，你可以像安装其他任何软件一样，在你的操作系统上安装比特币核心。对于Windows，运行.exe，并根据提示一步步进行安装。对于MacOS，先运行.dmg，完成后将Bitcoin-QT图标拉到应用程序文件夹即可。对于Ubuntu，在文件浏览器中双击PPA，系统将会打开软件包管理器进行软件包的安装。一旦安装完成，你将在应用列表中看到一个新的叫做Bitcoin-Qt的软件。双击图标可以启动比特币客户端。

首次运行比特币核心，它将开始下载区块链，这个过程可能需要持续几天（参看**图3-2**）。让它在后台运行，直到显示“同步完成”，并且余额旁不再显示“未同步”。



\*图3-2 比特币核心在区块链初始化时的屏幕显示\*

比特币核心本地保存一份交易账本（区块链）的全量副本，包含了自2009年比特币创立以来在比特币网络上发生过的所有交易。数据集大小约有几十G字节（在2013年末大概是16GB），它会以增量的形式，在几天内逐步下载完成。在区块链数据集下载完成之前，客户端都没法执行交易或者更新账户余额。这段时间内，客户端会在账户余额边上显示“未同步”，下方状态栏则会显示“正在同步”。为了完成初始化同步，请确保你有充足的硬盘空间、网络带宽和足够的时间。

**从源代码编译比特币核心**

对于开发者来说，也可以选择下载全量的源码（ZIP压缩包）或者从GitHub的官方源码库中克隆代码。在Github的bitcoin页面，从边栏选择下载ZIP包。或者，使用git命令行创建一个本地代码库，并从github下载副本。在下面的例子中，我们使用类Unix系统（linux，MacOS等）的命令行从github上克隆代码。

$ git clone https://github.com/bitcoin/bitcoin.git

Cloning into 'bitcoin'...

remote: Counting objects: 31864, done.

remote: Compressing objects: 100% (12007/12007), done.

remote: Total 31864 (delta 24480), reused 26530 (delta 19621)

Receiving objects: 100% (31864/31864), 18.47 MiB | 119 KiB/s, done.

Resolving deltas: 100% (24480/24480), done.

$

终端上的提示和输出结果可能会因为版本不同而有所不同。只要按照代码中所带的文档执行，即使实际输出结果与例子中显示的有轻微差异，也是正常的。

当git克隆操作完成后，在bitcoin目录中就拥有了一份代码库的完整副本。在提示符下键入命令“cd bitcoin",进入该目录：

$ cd bitcoin

不加参数的情况下，检出的本地副本与最新代码保持同步，这可能是比特币的一个不稳定版或beta版。因此在编译代码前，应通过加版本标签的形式来检出某个特定版本。这将让本地副本与版本库上某个特定版本的快照进行同步。这些版本标签是利用tag关键词进行标记的，它是开发者使用版本号对特定代码版本进行标记的一种技术。首先，为了找出所有可用标签，我们使用*git tag*命令：

$ git tag

v0.1.5

v0.1.6test1

v0.2.0

v0.2.10

v0.2.11

v0.2.12

[... many more tags ...]

v0.8.4rc2

v0.8.5

v0.8.6

v0.8.6rc1

v0.9.0rc1

这个标签列表列出了所有比特币的发行版本。按照惯例，*release candidates（候选发行版）*用于测试目的，带有“rc”后缀。稳定发行版则没有后缀，可以在生产系统上运行。从上述列表中，选择最高版本号的发行版，在写本书时，这个版本是v0.9.0rc1。为了让本地代码与这个版本同步，使用*git checkout*命令：

$ git checkout v0.9.0rc1

Note: checking out 'v0.9.0rc1'.

HEAD is now at 15ec451... Merge pull request #3605

$

源码中包含了文档，可以在几个文件中找到。键入*more README.md*, 查阅在bitcoin目录中的README.md主文档，根据提示，使用空格键来引导文档翻到下页。在本章中，我们将构建命令行比特币客户端，在linux上又名bitcoind。键入*more doc/build-unix.md*可以查阅在你的平台上编译bitcoind命令行客户端的指南。其他平台，如Mac OS X或者Windows的编译指南也可以在doc目录下找到，相应的文件为build-osx.md或者build-msw.md。

仔细研究构建的前置条件，在构建文档的前面部分有描述。这是一些必须在编译bitcoind之前已经在系统安装好的库文件。如果前置条件缺失，构建过程会失败，并显示错误信息。如果编译过程发现缺少了某些必须的库文件，你可以在安装好这些库文件后，重新执行编译程序，它将从刚才中断的地方开始继续进构建。假设所有前置要求都已经满足，你可以开始利用*autogen.sh*生成一系列构建脚本，开始构建过程。

比特币核心构建过程从0.9版开始变为采用autogen/configure/make系统。早期版本采用一个简单的Makefile文件，工作过程与下述例子有些微区别。请按照选定版本的操作指南来操作。0.9版引入的autogen/configure/make构建系统很可能成为所有后续版本的构建方法，也是下面例子演示的构建系统。

$ ./autogen.sh

configure.ac:12: installing `src/build-aux/config.guess'

configure.ac:12: installing `src/build-aux/config.sub'

configure.ac:37: installing `src/build-aux/install-sh'

configure.ac:37: installing `src/build-aux/missing'

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

$

*autogen.sh*脚本将创建一套自动化配置脚本，这些脚本通过检查你的系统以发现正确的设置，并确保你已经安装了编译代码所需的所有库文件。这些脚本里面最重要的是configure脚本，它提供了一系列不同选项，帮助你定制构建过程。输入*./configure --help*，可以查看所有选项：

$ ./configure --help

`configure' configures Bitcoin Core 0.9.0 to adapt to many kinds of systems.

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

To assign environment variables (e.g., CC, CFLAGS...), specify them as

VAR=VALUE. See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:

-h, --help display this help and exit

--help=short display options specific to this package

--help=recursive display the short help of all the included packages

-V, --version display version information and exit

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

Optional Features:

--disable-option-checking ignore unrecognized --enable/--with options

--disable-FEATURE do not include FEATURE (same as --enable-FEATURE=no)

--enable-FEATURE[=ARG] include FEATURE [ARG=yes]

[... more options ...]

Use these variables to override the choices made by `configure' or to help

it to find libraries and programs with nonstandard names/locations.

Report bugs to <info@bitcoin.org>.

$

configure脚本允许你通过--enable-FEATURE或--disable-FEATURE选项，来启用或禁用bitcoind某些功能。其中的FEATURE由具体功能名称代替，功能名称在上面的帮助信息中已经列出。在本章中，我们将构建的bitcoind客户端打开所有默认功能。我们不配置任何选项，但你最好还是好好研读一下帮助信息，了解客户端包含了哪些可选功能。接下来，运行configure脚本来自动发现所有必要的库，并为系统创建一个定制的构建脚本：

$ ./configure

checking build system type... x86\_64-unknown-linux-gnu

checking host system type... x86\_64-unknown-linux-gnu

checking for a BSD-compatible install... /usr/bin/install -c

checking whether build environment is sane... yes

checking for a thread-safe mkdir -p... /bin/mkdir -p

checking for gawk... no

checking for mawk... mawk

checking whether make sets $(MAKE)... yes

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

configure: creating ./config.status

config.status: creating Makefile

config.status: creating src/Makefile

config.status: creating src/test/Makefile

config.status: creating src/qt/Makefile

config.status: creating src/qt/test/Makefile

config.status: creating share/setup.nsi

config.status: creating share/qt/Info.plist

config.status: creating qa/pull-tester/run-bitcoind-for-test.sh

config.status: creating qa/pull-tester/build-tests.sh

config.status: creating src/bitcoin-config.h

config.status: executing depfiles commands

$

如果一切顺利，configure命令将成功完成定制化构建脚本的创建，这个脚本允许我们编译bitcoind。如果有缺失的库或者其他错误，configure命令将终止创建构建脚本，并输出错误。如果发生错误，很可能是因为缺失库或者库不兼容。再次查阅构建文档，确保已安装了所有先决要求。然后重新运行configure看看是否已修复了错误。接下来，你将编译源代码，这个过程可能会持续一个钟头。在编译的过程中，每隔几秒或几分钟就会输出一些信息，如果有什么问题，错误信息也会显示出来。编译如果被中断，你也可以随时恢复编译过程。键入*make*开始编译吧：

$ make

Making all in src

make[1]: Entering directory `/home/ubuntu/bitcoin/src'

make all-recursive

make[2]: Entering directory `/home/ubuntu/bitcoin/src'

Making all in .

make[3]: Entering directory `/home/ubuntu/bitcoin/src'

CXX addrman.o

CXX alert.o

CXX rpcserver.o

CXX bloom.o

CXX chainparams.o

[... many more compilation messages follow ...]

CXX test\_bitcoin-wallet\_tests.o

CXX test\_bitcoin-rpc\_wallet\_tests.o

CXXLD test\_bitcoin

make[4]: Leaving directory `/home/ubuntu/bitcoin/src/test'

make[3]: Leaving directory `/home/ubuntu/bitcoin/src/test'

make[2]: Leaving directory `/home/ubuntu/bitcoin/src'

make[1]: Leaving directory `/home/ubuntu/bitcoin/src'

make[1]: Entering directory `/home/ubuntu/bitcoin'

make[1]: Nothing to be done for `all-am'.

make[1]: Leaving directory `/home/ubuntu/bitcoin'

$

顺利的话，bitcoind就编译好了。最后的步骤是将bitcoind安装到系统路径中，仍然使用*make*命令：

$ sudo make install

Making install in src

Making install in .

/bin/mkdir -p '/usr/local/bin'

/usr/bin/install -c bitcoind bitcoin-cli '/usr/local/bin'

Making install in test

make install-am

/bin/mkdir -p '/usr/local/bin'

/usr/bin/install -c test\_bitcoin '/usr/local/bin'

$

你可以通过查看系统中以下两个可执行程序的位置来确认bitcoind是否已经正确安装：

$ which bitcoind

/usr/local/bin/bitcoind

$ which bitcoin-cli

/usr/local/bin/bitcoin-cli

默认安装时，bitcoind会被安装到*/usr/local/bin*目录下。当你第一次运行bitcoind时，它会提醒你创建一个配置文件，这个配置文件包含访问JSON-RPC接口的高强度密码。键入*bitcoind*在终端上运行bitcoind：

$ bitcoind

Error: To use the "-server" option, you must set a rpcpassword in the configuration file:

/home/ubuntu/.bitcoin/bitcoin.conf

It is recommended you use the following random password:

rpcuser=bitcoinrpc

rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK

(you do not need to remember this password)

The username and password MUST NOT be the same.

If the file does not exist, create it with owner-readable-only file permissions.

It is also recommended to set alertnotify so you are notified of problems;

for example: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com

在你喜欢的编辑器中编辑配置文件，设置参数，将密码替换为一个bitcoind建议的高强度密码。**不要**使用范例中使用的密码。在*.bitcoin*目录下创建一个命名为*.bitcoind/bitcoin.conf*的配置文件，输入用户名和密码：

rpcuser=bitcoinrpc

rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK

当你编辑这个配置文件时，你可能还希望设置其他几个选项，比如*txindex*（参见第47页《交易数据库索引和txindex选项）。若需要查看所有可用选项，请键入*bitcoind --help*。

现在，运行比特币核心客户端。第一次运行时，它会通过下载所有区块来重建比特币区块链。这是一个好几G的大文件，平均需要花费两天才能全量下载完成。你可以利用BitTorrent客户端从SourceForge下载部分区块链副本来缩短区块链初始化时间。

通过*-daemon*选项可以在后台运行bitcoind:

$ bitcoind -daemon

Bitcoin version v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)

Using OpenSSL version OpenSSL 1.0.1c 10 May 2012

Default data directory /home/bitcoin/.bitcoin

Using data directory /bitcoin/

Using at most 4 connections (1024 file descriptors available)

init message: Verifying wallet...

dbenv.open LogDir=/bitcoin/database ErrorFile=/bitcoin/db.log

Bound to [::]:8333

Bound to 0.0.0.0:8333

init message: Loading block index...

Opening LevelDB in /bitcoin/blocks/index

Opened LevelDB successfully

Opening LevelDB in /bitcoin/chainstate

Opened LevelDB successfully

[... more startup messages ...]

**通过命令行调用比特币核心的JSON-RPC接口**

比特币核心客户端实现了一个可供命令行助手*bitcoin-cli*调用的JSON-RPC接口。这使我们可以实验那些通常由程序通过API调用的功能。开始前，我们先调用help命令来看一下全部可用的RPC命令列表：

$ bitcoin-cli help

addmultisigaddress nrequired ["key",...] ( "account" )

addnode "node" "add|remove|onetry"

backupwallet "destination"

createmultisig nrequired ["key",...]

createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}

decoderawtransaction "hexstring"

decodescript "hex"

dumpprivkey "bitcoinaddress"

dumpwallet "filename"

getaccount "bitcoinaddress"

getaccountaddress "account"

getaddednodeinfo dns ( "node" )

getaddressesbyaccount "account"

getbalance ( "account" minconf )

getbestblockhash

getblock "hash" ( verbose )

getblockchaininfo

getblockcount

getblockhash index

getblocktemplate ( "jsonrequestobject" )

getconnectioncount

getdifficulty

getgenerate

gethashespersec

getinfo

getmininginfo

getnettotals

getnetworkhashps ( blocks height )

getnetworkinfo

getnewaddress ( "account" )

getpeerinfo

getrawchangeaddress

getrawmempool ( verbose )

getrawtransaction "txid" ( verbose )

getreceivedbyaccount "account" ( minconf )

getreceivedbyaddress "bitcoinaddress" ( minconf )

gettransaction "txid"

gettxout "txid" n ( includemempool )

gettxoutsetinfo

getunconfirmedbalance

getwalletinfo

getwork ( "data" )

help ( "command" )

importprivkey "bitcoinprivkey" ( "label" rescan )

importwallet "filename"

keypoolrefill ( newsize )

listaccounts ( minconf )

listaddressgroupings

listlockunspent

listreceivedbyaccount ( minconf includeempty )

listreceivedbyaddress ( minconf includeempty )

listsinceblock ( "blockhash" target-confirmations )

listtransactions ( "account" count from )

listunspent ( minconf maxconf ["address",...] )

lockunspent unlock [{"txid":"txid","vout":n},...]

move "fromaccount" "toaccount" amount ( minconf "comment" )

Using Bitcoin Core’s

ping

sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "commentto" )

sendmany "fromaccount" {"address":amount,...} ( minconf "comment" )

sendrawtransaction "hexstring" ( allowhighfees )

sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" )

setaccount "bitcoinaddress" "account"

setgenerate generate ( genproclimit )

settxfee amount

signmessage "bitcoinaddress" "message"

signrawtransaction "hexstring" ( [{"txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex"},...] ["privatekey1",...] sighashtype )

stop

submitblock "hexdata" ( "jsonparametersobject" )

validateaddress "bitcoinaddress"

verifychain ( checklevel numblocks )

verifymessage "bitcoinaddress" "signature" "message"

walletlock

walletpassphrase "passphrase" timeout

walletpassphrasechange "oldpassphrase" "newpassphrase"

**从比特币核心客户端的状态中获取消息**

命令：*getinfo*

比特币的*getinfo* RPC命令显示比特币网络节点、钱包、区块链数据库状态的基础信息，运行*bitcoin-cli getinfo*：

$ bitcoin-cli getinfo

{

"version" : 90000,

"protocolversion" : 70002,

"walletversion" : 60000,

"balance" : 0.00000000,

"blocks" : 286216,

"timeoffset" : -72,

"connections" : 4,

"proxy" : "",

"difficulty" : 2621404453.06461525,

"testnet" : false,

"keypoololdest" : 1374553827,

"keypoolsize" : 101,

"paytxfee" : 0.00000000,

"errors" : ""

}

数据以JavaScript对象符号（JSON）格式返回，这种格式不仅可以轻易被编程语言“消费”，也便于人工阅读。在数据中，我们可以看到比特币客户端的版本号（90000），协议版本号（70002），钱包版本号（60000）。也能看到钱包的余额，当前为0。还能看到区块高度，告诉我们客户端总共看到了多少区块（当前286,216）。另外，返回信息中还包含各种比特币网络相关的统计数据以及当前客户端相关的设置信息。我们将在本章剩余部分更详细的了解这些设置。

这将需要一些时间，也许超过一天，等待bitcoind客户端从其他比特币客户端下载区块，以“追赶上”当前的区块链高度。你可以通过getinfo查看已知的区块来检查同步进度。

**钱包设置和加密**

命令：*encryptwallet, walletpassphrase*

在开始创建密钥和其他命令前，你需要首先利用密码来给钱包加密。在这个例子中，我们使用*encryptwallet*命令和密码“foo”，显然用更加强壮和复杂的密码替代“foo”是必须的！

$ bitcoin-cli encryptwallet foo

wallet encrypted; Bitcoin server stopping, restart to run with encrypted wallet. The keypool has been flushed, you need to make a new backup.

$

你可以重新运行*getinfo*来检查钱包是否已经加密。这次，你会注意到有个新的条目叫*unlocked\_until*。这是一个计数器，显示钱包解密密码在内存中存储，保持钱包解锁状态的时间。最初，计数器会被设为0，代表钱包是被锁定的：

$ bitcoin-cli getinfo

{

"version" : 90000,

#[... other information...]

"unlocked\_until" : 0,

"errors" : ""

}

$

为了解锁钱包，发出*walletpassphrase*命令，它包含两个参数--密码、直到钱包重新锁定的超时秒数（时间计数器）：

$ bitcoin-cli walletpassphrase foo 360

$

你可以重新运行*getinfo*确认钱包是否已解锁以及超时时间：

$ bitcoin-cli getinfo

{

"version" : 90000,

#[... other information ...]

"unlocked\_until" : 1392580909,

"errors" : ""

}

**钱包备份，明文导出，恢复**

命令：*backupwallet, importwallet, dumpwallet*

接下来，我们练习创建钱包备份文件，然后从备份文件中恢复钱包。使用*backupwallet*命令来备份，提供文件名作为命令的参数。这里，我们将钱包备份到文件*wallet.backup*中：

$ bitcoin-cli backupwallet wallet.backup

$

现在，利用*importwallet*从备份文件中恢复钱包。如果钱包是锁定状态的，你需要先进行解锁（在上节可以查看*walletpassphrase*命令用法）。

$ bitcoin-cli importwallet wallet.backup

$

*dumpwallet*命令用于将钱包导出到人类可读的文本文件：

$ bitcoin-cli dumpwallet wallet.txt

$ more wallet.txt

# Wallet dump created by Bitcoin v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)

# \* Created on 2014-02- 8dT20:34:55Z

# \* Best block at time of backup was 286234 (0000000000000000f74f0bc9d3c186267bc45c7b91c49a0386538ac24c0d3a44),

# mined on 2014-02- 8dT20:24:01Z

KzTg2wn6Z8s7ai5NA9MVX4vstHRsqP26QKJCzLg4JvFrp6mMaGB9 2013-07- 4dT04:30:27Z change=1 # addr=16pJ6XkwSQv5ma5FSXMRPaXEYrENCEg47F

Kz3dVz7R6mUpXzdZy4gJEVZxXJwA15f198eVui4CUivXotzLBDKY 2013-07- 4dT04:30:27Z change=1 # addr=17oJds8kaN8LP8kuAkWTco6ZM7BGXFC3gk

[... many more keys ...]

$

**钱包地址和接收交易**

命令：*getnewaddress, getreceivedbyaddress, listtransactions, getaddressesbyaccount, getbalance*

比特币参考客户端维护着一个地址池，地址池的大小在命令*getinfo*的输出项*keypoolsize*中展示。这些地址是自动生成，可用于公开接收地址或者找零地址。使用*getnewaddress*命令，可以生成一个新地址：

$ bitcoin-cli getnewaddress

1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL

现在，我们可以从外部钱包（假设你在交易所、网络钱包或者在其他地方运行的bitcoind钱包中已经拥有一些比特币），通过这个地址发送一笔小额的比特币到我们的bitcoind钱包。在这个例子中，我们将发送50毫比特（0.050比特币）到这个地址。

我们可以询问bitcoind客户端此地址已经接收到的比特币金额，查询时，需要指定确认次数，即多少次确认一笔资金才计入余额中，在这里，我们指定0次确认。从另一个钱包发送比特币几秒后，我们就可以看到，交易已经在钱包的余额中体现出来了。现在，我们使用*getreceivedbyaddress*命令并结合地址以及0确认次数来查看：

$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL 0

0.050000

如果省略掉命令最后一个0，我们将只能看到经过至少*minconf*次确认的金额，*minconf*是最少确认次数的设置值，未达到这个确认次数，交易不会计入余额。*minconf*的值是在bitcoind配置文件中设置的。因为发送这个比特币的交易仅仅发生在几秒前，它尚未被确认，所以我们看到它只列出了一个0余额：

$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL

0.00000000

钱包接收到的所有交易也可以利用*listtransactions*命令列出来：

$ bitcoin-cli listtransactions

[

{

"account" : "",

"address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",

"category" : "receive",

"amount" : 0.05000000,

"confirmations" : 0,

"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

"time" : 1392660908,

"timereceived" : 1392660908

}

]

我们还可以利用*getaddressesbyaccount*命令列出钱包中的所有地址：

$ bitcoin-cli getaddressesbyaccount ""

[

"1LQoTPYy1TyERbNV4zZbhEmgyfAipC6eqL",

"17vrg8uwMQUibkvS2ECRX4zpcVJ78iFaZS",

"1FvRHWhHBBZA8cGRRsGiAeqEzUmjJkJQWR",

"1NVJK3JsL41BF1KyxrUyJW5XHjunjfp2jz",

"14MZqqzCxjc99M5ipsQSRfieT7qPZcM7Df",

"1BhrGvtKFjTAhGdPGbrEwP3xvFjkJBuFCa",

"15nem8CX91XtQE8B1Hdv97jE8X44H3DQMT",

"1Q3q6taTsUiv3mMemEuQQJ9sGLEGaSjo81",

"1HoSiTg8sb16oE6SrmazQEwcGEv8obv9ns",

"13fE8BGhBvnoy68yZKuWJ2hheYKovSDjqM",

"1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",

"1KHUmVfCJteJ21LmRXHSpPoe23rXKifAb2",

"1LqJZz1D9yHxG4cLkdujnqG5jNNGmPeAMD"

]

最后，命令*getbalance*可以显示钱包的全部余额，它将合并所有经过至少*minconf*次确认的交易的金额。

$ bitcoin-cli getbalance

0.05000000

如果交易尚未确认，getbalance命令返回的余额是0。“*minconf*”选项决定了需要经过几次确认，交易金额才会体现到余额上。

**探索及解码交易**

命令：*gettransaction、getrawtransaction、decoderawtransaction*

现在，我们来看看前面利用*gettransaction*命令列出的传入交易。我们可以通过交易哈希提取到一笔交易，交易哈希就是前面我们看到的*txid*，提取交易的命令为：*gettransaction*。

$ bitcoin-cli gettransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3

{

"amount" : 0.05000000,

"confirmations" : 0,

"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

"time" : 1392660908,

"timereceived" : 1392660908,

"details" : [

{

"account" : "",

"address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",

"category" : "receive",

"amount" : 0.05000000

}

]

}

在未确认前，交易ID不具有权威性。区块链中交易哈希缺失不意味着交易未被执行。这被称之为“交易可锻性”，因为交易哈希可以在区块确认前被修改。一旦确认，*txid*就是不变的，权威的。

通过命令*gettransaction*显示的交易形式是一种简化的形式。为了获取完整交易代码并解码它，我们需要利用两个命令：*getrawtransaction*和*decoderawtransaction*。首先，使用命令*getrawtransaction*，并以交易哈希（*txid*）作为参数，将返回一个“原始”的十六进制字符串，就像它在比特币网络上的样子。

$ bitcoin-cli getrawtransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66

c309acbae2c14ae3

0100000001d717279515f88e2f56ce4e8a31e2ae3e9f00ba1d0add648e80c480ea22e0c7d3000000

008b483045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e61a2df0dd0758e227383b3

02203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b9b2ac1bd193dfba2014104793ac8

a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65e4c4b884ac5b5b6dede05ba84727e34

c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5ceaffffffff02404b4c00000000001976a91407bd

b518fa2e6089fd810235cf1100c9c13d1fd288ac1f312906000000001976a914107b7086b3151893

5c8d28703d66d09b3623134388ac00000000

为了解码这个十六进制字符串，需要使用*decoderawtransaction*命令。拷贝粘贴这些十六进制作为命令*decoderawtransaction*的第一个参数，就得到了完整的以JSON数据格式表示的内容。（出于格式化需要，十六进制字符串在以下例子中被截短了）：

$ bitcoin-cli decoderawtransaction 0100000001d717...388ac00000000

{

"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

"version" : 1,

"locktime" : 0,

"vin" : [

{

"txid" : "d3c7e022ea80c4808e64dd0a1dba009f3eaee2318a4ece562f8ef815952717d7",

"vout" : 0,

"scriptSig" : {

"asm" : "3045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e61a2df0dd0758e227383b302203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b9b2ac1bd193dfba20104793ac8a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65e4c4b884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5cea",

"hex": "483045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e61a2df0dd0758e227383b302203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b9b2ac1bd193dfba2014104793ac8a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65e4c4b884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5cea"

},

"sequence" : 4294967295

}

],

"vout" : [

{

"value" : 0.05000000,

"n" : 0,

"scriptPubKey" : {

"asm" : "OP\_DUP OP\_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP\_EQUALVERIFY OP\_CHECKSIG",

"hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",

"reqSigs" : 1,

"type" : "pubkeyhash",

"addresses" : [

"1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"

]

}

},

{

"value" : 1.03362847,

"n" : 1,

"scriptPubKey" : {

"asm" : "OP\_DUP OP\_HASH160 107b7086b31518935c8d28703d66d09b36231343 OP\_EQUALVERIFY OP\_CHECKSIG",

"hex" : "76a914107b7086b31518935c8d28703d66d09b3623134388ac",

"reqSigs" : 1,

"type" : "pubkeyhash",

"addresses" : [

"12W9goQ3P7Waw5JH8fRVs1e2rVAKoGnvoy"

]

}

}

]

}

这个JSON格式的输出展示了一个交易的所有组成元素，包括交易输入和输出。在这个例子中，我们看到交易使用了一个输入生成了两个输出，往我们的新地址记入了50毫比特。这个交易的输入是上一笔已确认交易的输出（显示为*vin*下*d3c7*开始的*txid*）。两个输出一个是50毫比特的入账，另一笔是交易找零。

我们可以使用同样的命令（*gettransaction*），通过查看txid引用的交易进一步对区块链进行探索。如此一级一级循着交易链条，我们可以看到资金一次又一次的从一个所有者地址转移到另一个所有者地址。

如果接收到交易已经被确认，*gettransaction*命令还将额外返回交易所在区块的*区块哈希（标识符）*：

$ bitcoin-cli gettransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66↵

c309acbae2c14ae3

{

"amount" : 0.05000000,

"confirmations" : 1,

"blockhash" : "000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c↵

2a717b",

"blockindex" : 18,

"blocktime" : 1392660808,

"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

"time" : 1392660908,

"timereceived" : 1392660908,

"details" : [

{

"account" : "",

"address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",

"category" : "receive",

"amount" : 0.05000000

}

]

}

在这儿，我们看到输出项中多了一个blockhash条目（区块哈希，交易所在区块的哈希值）和值为18的blockindex条目（区块索引，指出此交易是区块内的第18个交易）

| **交易数据库索引和txindex选项** |
| --- |
| 默认情况下，比特币核心创建一个仅包含与用户钱包相关的交易数据库。如果你希望访问所有类似gettransaction等命令能查看的交易，你需要配置比特币核心客户端，使其创建一个完整的交易索引。这可以通过设置txindex选项来实现。在比特币核心的配置文件（该文件通常位于你的Home目录下的*.bitcoin/bitcoin.conf*）中将txindex设置为1.一旦你修改了参数，你需要重启bitcoind并等待其完成索引创建 |

**探索区块**

命令：*getblock, getblockhash*

现在我们已经知道交易位于哪个区块内，我们可以查询该区块。使用*getblock*命令并以区块哈希作为参数：

$ bitcoin-cli getblock 000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c2a717b true

{

"hash" : "000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c2a717b",

"confirmations" : 2,

"size" : 248758,

"height" : 286384,

"version" : 2,

"merkleroot" : "9891747e37903016c3b77c7a0ef10acf467c530de52d84735bd55538719f9916",

"tx" : [

"46e130ab3c67d31d2b2c7f8fbc1ca71604a72e6bc504c8a35f777286c6d89bf0",

"2d5625725b66d6c1da88b80b41e8c07dc5179ae2553361c96b14bcf1ce2c3868",

"923392fc41904894f32d7c127059bed27dbb3cfd550d87b9a2dc03824f249c80",

"f983739510a0f75837a82bfd9c96cd72090b15fa3928efb9cce95f6884203214",

"190e1b010d5a53161aa0733b953eb29ef1074070658aaa656f933ded1a177952",

"ee791ec8161440262f6e9144d5702f0057cef7e5767bc043879b7c2ff3ff5277",

"4c45449ff56582664abfadeb1907756d9bc90601d32387d9cfd4f1ef813b46be",

"3b031ed886c6d5220b3e3a28e3261727f3b4f0b29de5f93bc2de3e97938a8a53",

"14b533283751e34a8065952fd1cd2c954e3d37aaa69d4b183ac6483481e5497d",

"57b28365adaff61aaf60462e917a7cc9931904258127685c18f136eeaebd5d35",

"8c0cc19fff6b66980f90af39bee20294bc745baf32cd83199aa83a1f0cd6ca51",

"1b408640d54a1409d66ddaf3915a9dc2e8a6227439e8d91d2f74e704ba1cdae2",

"0568f4fad1fdeff4dc70b106b0f0ec7827642c05fe5d2295b9deba4f5c5f5168",

"9194bfe5756c7ec04743341a3605da285752685b9c7eebb594c6ed9ec9145f86",

"765038fc1d444c5d5db9163ba1cc74bba2b4f87dd87985342813bd24021b6faf",

"bff1caa9c20fa4eef33877765ee0a7d599fd1962417871ca63a2486476637136",

"d76aa89083f56fcce4d5bf7fcf20c0406abdac0375a2d3c62007f64aa80bed74",

"e57a4c70f91c8d9ba0ff0a55987ea578affb92daaa59c76820125f31a9584dfc",

"9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

#[... many more transactions ...]

],

"time" : 1392660808,

"nonce" : 3888130470,

"bits" : "19015f53",

"difficulty" : 3129573174.52228737,

"chainwork" : "000000000000000000000000000000000000000000001931d1658fc04879e466",

"previousblockhash" : "0000000000000000177e61d5f6ba6b9450e0dade9f39c257b4d48b4941ac77e7",

"nextblockhash" : "0000000000000001239d2c3bf7f4c68a4ca673e434702a57da8fe0d829a92eb6"

区块包含367个交易，你可以看到，第18个交易（9ca8f9....）是往我们的地址上发送50毫比特的交易ID(txid)。输出项高度（*height*）告诉我们这是区块链中的第286384个区块。

我们也可以利用*getblockhash*命令，根据区块的高度来获取区块信息。这种情况下，我们使用区块高度作为参数，返回区块哈希：

$ bitcoin-cli getblockhash 0

000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

这里，我们取得了创世区块的哈希，它是中本聪挖到的第一个区块，其高度为0。进一步获取区块的详细信息如下：

$ bitcoin-cli getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

{

"hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",

"confirmations" : 286388,

"size" : 285,

"height" : 0,

"version" : 1,

"merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",

"tx" : [

"4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"

],

"time" : 1231006505,

"nonce" : 2083236893,

"bits" : "1d00ffff",

"difficulty" : 1.00000000,

"chainwork" : "0000000000000000000000000000000000000000000000000000000100010001",

"nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"

}

*getblock,getblockhash*以及*gettransaction*命令也可用于程序中查询区块链数据库。

**基于未花费的交易输出创建、签名、提交交易**

命令：*listunspent, gettxout, createrawtransaction, decoderawtransaction, signrawtransaction, sendrawtransaction*

比特币的交易是基于花费“输出”的概念，“输出”是前序交易的结果，由此创建了一个在所有者地址间的价值传递的交易链。我们的钱包现在接收到了一个交易，它将一个输出分配给我们的地址。一旦这个交易被确认，我们就能花费这个输出。

首先：我们利用*listunspent*命令来显示我们钱包中的未花费*已确认*的交易输出:

$ bitcoin-cli listunspent

[

{

"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

"vout" : 0,

"address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",

"account" : "",

"scriptPubKey" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",

"amount" : 0.05000000,

"confirmations" : 7

}

]

我们看到交易9ca8f9...创建了一个输出(vout索引号为0)，为地址1hvzSo...存入50毫比特，这个时刻，该交易已经经过7次确认。交易使用前序交易创建的输出作为它的输入，通过引用前序交易的txid和vout索引形成与前序交易的连接。我们现在可以创建一个新的交易来花费txid为9ca8f9...的第0个输出（vout=0），在这个交易中，我们将把上述交易的输出作为新交易的输入，并将它发给一个新的地址。

首先，我们更详细的查看这个交易的输出。我们使用*gettxout*来获取这个未花费输出。交易输出总是通过txid和vout被引用，这也是我们传给*gettxout*的参数：

$ bitcoin-cli gettxout 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3 0

{

"bestblock" : "0000000000000001405ce69bd4ceebcdfdb537749cebe89d371eb37e13899fd9",

"confirmations" : 7,

"value" : 0.05000000,

"scriptPubKey" : {

"asm" : "OP\_DUP OP\_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2

OP\_EQUALVERIFY OP\_CHECKSIG",

"hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",

"reqSigs" : 1,

"type" : "pubkeyhash",

"addresses" : [

"1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"

]

},

"version" : 1,

"coinbase" : false

}

我们在这儿看到的是给我们的地址1hvz...发送50毫比特的交易输出。为了花费这个输出，我们将创建一笔新交易。首先，我们生成一个新的地址用于发送资金：

$ bitcoin-cli getnewaddress

1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb

我们将发送25毫比特到这个新创建的地址（1LnfTn...)。在新交易中，我们将花费50毫比特的输出，并且发送25毫比特到这个新地址。由于我们必须花费前序交易的*整个*输出，所以交易还会产生一部分找零。我们将找零发送回原地址（1hvz...)。最后，我们还需要发送一些交易费用。为了发送费用，我们把找零的数额减少0.5毫比特，仅找回24.5毫比特。交易输出的总额为（25mBTC+24.5mBTC=49.5mBTC）与输入（50mBTC)的差额将会作为交易费用被矿工收集走。

我们使用*createrawtransaction*来创建这个交易。作为参数，我们提供交易输入（已确认交易的50毫比特未花费输出），以及2个交易输出（发往新地址的资金和返回发送者自身地址的交易找零）：

$ bitcoin-cli createrawtransaction '[{"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3", "vout" : 0}]' '{"1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb": 0.025, "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL": 0.0245}'

0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c0000000000ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000

*createrawtransaction*命令产生一串原始十六进制字符串，将我们提供的交易细节进行了编码。接下来，我们通过解码这串字符串确认一下所有事情是不是已正确完成，使用*decoderawtransaction*命令：

$ bitcoin-cli decoderawtransaction 0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c0000000000ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000

{

"txid" : "0793299cb26246a8d24e468ec285a9520a1c30fcb5b6125a102e3fc05d4f3cba",

"version" : 1,

"locktime" : 0,

"vin" : [

{

"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

"vout" : 0,

"scriptSig" : {

"asm" : "",

"hex" : ""

},

"sequence" : 4294967295

}

],

"vout" : [

{

"value" : 0.02500000,

"n" : 0,

"scriptPubKey" : {

"asm" : "OP\_DUP OP\_HASH160 d90d36e98f62968d2bc9bbd68107564a156a9bcf OP\_EQUALVERIFY OP\_CHECKSIG",

"hex" : "76a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac",

"reqSigs" : 1,

"type" : "pubkeyhash",

"addresses" : [

"1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb"

]

}

},

{

"value" : 0.02450000,

"n" : 1,

"scriptPubKey" : {

"asm" : "OP\_DUP OP\_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP\_EQUALVERIFY OP\_CHECKSIG",

"hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",

"reqSigs" : 1,

"type" : "pubkeyhash",

"addresses" : [

"1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"

]

}

}

]

}

看起来像是正确的！我们的新交易“花费”了我们的已确认交易中的未花费输出，并将其拆分为两个输出，一个是25毫比特，发到我们新地址，另一个24.5毫比特作为交易找零返回原地址。0.5毫比特的差额作为交易费用，将发给找到找到包含本交易区块的矿工。

就像你可能注意到的那样，交易包含一个空的*scriptSig*(脚本签名），因为我们尚未对其签名。没有签名，一笔交易是毫无意义的；因为我们尚未证明我们*拥有*未花费输出的地址。通过签名，我们移除了输出的阻碍，证明了我们确实拥有这笔输出，并且能够使用它。我们使用*signrawtransaction*命令来签名交易。它以原始交易的十六进制字符串作为参数。

一个加密的钱包必须先进行解锁才能对交易进行签名，因为签名需要访问钱包中的密钥。

$ bitcoin-cli walletpassphrase foo 360

$ bitcoin-cli signrawtransaction 0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c0000000000ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000

{

"hex" : "0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5127ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000",

"complete" : true

}

*signrawtransaction*命令返回另外一个十六进制编码的原始交易。我们解码看看有什么变化：

{

"txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",

"version" : 1,

"locktime" : 0,

"vin" : [

{

"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

"vout" : 0,

"scriptSig" : {

"asm" : "304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af52051a0601 03c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5127",

"hex" : "47304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5127"

},

"sequence" : 4294967295

}

],

"vout" : [

{

"value" : 0.02500000,

"n" : 0,

"scriptPubKey" : {

"asm" : "OP\_DUP OP\_HASH160 d90d36e98f62968d2bc9bbd68107564a156a9bcf OP\_EQUALVERIFY OP\_CHECKSIG",

"hex" : "76a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac",

"reqSigs" : 1,

"type" : "pubkeyhash",

"addresses" : [

"1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb"

]

}

},

{

"value" : 0.02450000,

"n" : 1,

"scriptPubKey" : {

"asm" : "OP\_DUP OP\_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP\_EQUALVERIFY OP\_CHECKSIG",

"hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",

"reqSigs" : 1,

"type" : "pubkeyhash",

"addresses" : [

"1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"

]

}

}

]

}

现在，交易中的输入包含了*scriptSig*，这是一个地址（1hvz...）拥有者提供的数字签名，解除了原交易输出上的阻碍，输出得以使用。签名使得交易可以被比特币网络上的任何节点进行确认。

是时候将这个新交易提交到网络了。我们使用*sendrawtransaction*命令，该命令以上述已签名交易的原始十六进制字符串作为参数，也就是刚刚我们解码的字符串。

$ bitcoin-cli sendrawtransaction0100000001e34ac1e2baac09c366fce1c2245536bda8f↵

7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bacfbc0c80↵

0c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3↵

897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d↵

3f00fd44ea6c71dc5127ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9bbd↵

68107564a156a9bcf88ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100c9↵

c13d1fd288ac00000000ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584a↵

c17b346

当交易提交到网络后，命令*sendrawtransaction*返回一个交易哈希(txid)。我们可以利用*gettransaction*命令来查询这个交易ID：

{

"amount" : 0.00000000,

"fee" : -0.00050000,

"confirmations" : 0,

"txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",

"time" : 1392666702,

"timereceived" : 1392666702,

"details" : [

{

"account" : "",

"address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",

"category" : "send",

"amount" : -0.02500000,

"fee" : -0.00050000

},

{

"account" : "",

"address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",

"category" : "send",

"amount" : -0.02450000,

"fee" : -0.00050000

},

{

"account" : "",

"address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",

"category" : "receive",

"amount" : 0.02500000

},

{

"account" : "",

"address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",

"category" : "receive",

"amount" : 0.02450000

}

]

}

就像之前看到的，我们依然可以使用*getrawtransaction*和*decodetransaction*命令查看更详细信息。它们返回的信息与交易提交网络前看到的是一样的。

**替代客户端，库，工具集**

除了参考客户端（bitcoind），其它客户端和库也能用来与比特币网络及其数据结构进行交互。这里是一些通过不同语言实现的客户端，提供了相应语言的本地化接口。

*libbitcoin和sx工具集*

一个C++多线程完全节点客户端和库，包含命令行工具

*bitcoinj*

一个Java完全节点客户端库

*btcd*

一个Go语言完全节点比特币客户端

*Bits of Proof(BOP)*

一个Java实现的企业级比特币客户端

*picocoin*

一个轻量级比特币客户端库的C语言实现

*pybitcointools*

一个Python实现的比特币库

*pycoin*

另一个Python比特币库

还有很多其他各种语言实现的库，同时还有更多的库正在开发中。

**libbitcoin和sx工具集**

libbitcoin库是一个C++的可扩展多线程、模块化的实现，它支持完全节点客户端，它还带一个命令行工具集，叫做sx，这个工具集提供很多与我们在本章中展示过的bitcoind客户端命令行一样的功能。sx工具集还提供一些bitcoind未提供的密钥管理和维护工具，包括type-2确定性密钥和密钥助记符功能。

**安装sx**

为了安装sx以及其支持库libbitcoin，在Linux系统上下载并安装在线安装包：

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

$ sudo bash ./install-sx.sh

现在，sx工具集已经安装好了，键入不带参数的sx命令打印帮助文档，这会列出所有可用命令（参看附录D）

sx工具集提供了许多实用命令来对地址进行编码或解码，也可以将它们的格式或者表现方式互相转换。可以利用它们来探索各种不同的格式，比如Base58，Base58Check，十六进制，等等。

**pycoin**

Python库pycoin，最早由理查德.吉斯（Richard Kiss）开发和维护，是一个基于python的库，它支持操作比特币密钥和交易，甚至也支持使用脚本语言来正确处理非标准交易。

pycoin库支持Python 2（2.7.x）和Python 3（3.3之后版本），同时还随带了一些好用的命令行工具，*ku*和*tx*。在python 3的虚拟环境（venv）安装pycoin 0.42的步骤如下：

$ python3 -m venv /tmp/pycoin

$ . /tmp/pycoin/bin/activate

$ pip install pycoin==0.42

Downloading/unpacking pycoin==0.42

Downloading pycoin-0.42.tar.gz (66kB): 66kB downloaded

Running setup.py (path:/tmp/pycoin/build/pycoin/setup.py) egg\_info for package pycoin

Installing collected packages: pycoin

Running setup.py install for pycoin

Installing tx script to /tmp/pycoin/bin

Installing cache\_tx script to /tmp/pycoin/bin

Installing bu script to /tmp/pycoin/bin

Installing fetch\_unspent script to /tmp/pycoin/bin

Installing block script to /tmp/pycoin/bin

Installing spend script to /tmp/pycoin/bin

Installing ku script to /tmp/pycoin/bin

Installing genwallet script to /tmp/pycoin/bin

Successfully installed pycoin

Cleaning up...

$

这里是一个利用pycoin库来获取并花费比特币的python脚本：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | #!/usr/bin/env python  from pycoin.key import Key  from pycoin.key.validate import is\_address\_valid, is\_wif\_valid  from pycoin.services import spendables\_for\_address  from pycoin.tx.tx\_utils import create\_signed\_tx  def get\_address(which):  while 1:  print("enter the %s address=> " % which, end='')  address = input()  is\_valid = is\_address\_valid(address)  if is\_valid:  return address  print("invalid address, please try again")  src\_address = get\_address("source")  spendables = spendables\_for\_address(src\_address)  print(spendables)  while 1:  print("enter the WIF for %s=> " % src\_address, end='')  wif = input()  is\_valid = is\_wif\_valid(wif)  if is\_valid:  break  print("invalid wif, please try again")  key = Key.from\_text(wif)  if src\_address not in (key.address(use\_uncompressed=False), key.address(use\_uncompressed=True)):  print("\*\* WIF doesn't correspond to %s" % src\_address)  print("The secret exponent is %d" % key.secret\_exponent())  dst\_address = get\_address("destination")  tx = create\_signed\_tx(spendables, payables=[dst\_address], wifs=[wif])  print("here is the signed output transaction")  print(tx.as\_hex()) |

命令行工具*ku*和*tx*的例子参看附录B。

**btcd**

btcd是一个Go语言开发的完全节点比特币工具。目前，它的下载、验证、服务区块链的规则与参考实现bitcoind接受区块的规则完全一致（甚至bug也一致）。它同样正确中转新挖出的区块，维护一个交易池，转发尚未进入区块的交易。它确保获准进入交易池的交易严格遵守通用规则以及大多数矿工提出的更为严格的过滤原则（“标准”交易）。

btcd和bitcoind间最主要的一个区别在于，btcd没有钱包功能，这是btcd的一个故意的设计选择。这意味着，你无法直接使用btcd发送或接收支付款项。钱包功能由btcwallet和btcgui提供，这两个软件都处于积极开发中。其他比较显著的区别还包括：btcd同时支持HTTP POST请求（bitcoind也支持）和默认的Websocket连接，实际上，btcd的RPC连接是默认启用TLS的（译者：TLS，安全传输层协议，用于在两个通信应用程序之间提供保密性和数据完整性）。

**安装btcd**

要在Windows下安装btcd，需要从GitHub（https://github.com/conformal/btcd/releases）上下载msi安装包，并进行安装；如果是Linux系统，假设你已安装了Go语言：

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

若需要更新btcd版本，直接执行：

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

**控制btcd**

btcd有一系列配置选项，你可以通过以下命令查看：

$ btcd --help

btcd安装时随带了一些好东西，比如btctl，这是一个命令行工具，可以用于通过RPC控制或查询btcd。btcd默认没有启用其RPC服务，要启动该服务，你至少需要在文件*btcd.conf*和btctl.conf\*中配置好RPC的用户名和密码：

* *btcd.conf*

[Application Options]

rpcuser=myuser

rpcpass=SomeDecentp4ssw0rd

* *btcctl.conf*

[Application Options]

rpcuser=myuser

rpcpass=SomeDecentp4ssw0rd

或者你也可以通过命令来覆盖配置文件，命令如下：

$ btcd -u myuser -P SomeDecentp4ssw0rd

$ btcctl -u myuser -P SomeDecentp4ssw0rd

要获取可用选项的列表，可以运行以下命令：

$ btcctl --help

**第4章 密钥，地址，钱包**

**4.1简介**

比特币的所有权是建立在*数字密钥*，*比特币地址*以及*数字签名*的基础上的。数字密钥实际上并不存储于网络上，而是由用户创建并以文件或者简单数据库的形式由用户自行保存， 叫做钱包。用户钱包中的数字密钥是完全独立于比特币协议的，它可以由钱包软件创建、管理，而无需与区块链关联或者访问互联网。密钥的存在使比特币的很多有意思的特性得以实现，包括去中心化的信任和控制，持有证明，加密安全模型等等。

为了加入到区块链中，每个比特币交易都需提供一个有效签名，这个签名只能由有效的数字密钥产生；因而，任何持有密钥的人都拥有这个账户中比特币的控制权。 密钥是成对出现的，包含一个私钥（需要保密）和一个公钥。我们可以把公钥想象成银行的账户代号，私钥则是控制这个账户的密码，或者支票上的用户签名。这些数字密钥极少被比特币用户看到。实际上，大多数时候，他们存储在钱包文件中，并且由钱包软件进行管理。

在比特币的支付环节，接收者的公钥是由其数字指纹所替代的，称为*比特币地址*，它就像支票上的接收人名称（“收款人”）。大多数情况下，比特币地址是从公钥产生并与之关联的。但是比特币地址也不全代表一个公钥，它也可以是其它“受益人”，比如稍后我们将在本章中将见到的“脚本”。这种将资金接收方抽象为一个比特币地址的方式，使得交易目标更为灵活，类似于纸质支票：一笔支付指令既可用于向个人账户付款，也可以用于向公司账户付款，既可以账单支付，也可以现金支付。比特币地址是用户可见的唯一表现形式，因为它是需要于公众共享的部分。

在本章中，我们将介绍钱包，它管理着用户的密钥。我们将了解密钥是如何产生，存储和管理的。我们也将回顾各种用于公私钥、比特币地址、脚本地址的编码格式。最后，我们将了解一下密钥的特殊使用场景，如消息签名、所有权证明、创建“炫耀”地址、纸钱包等。

**4.1.1 公钥密码学和加密货币**

公钥密码学诞生于上世纪70年代，是计算机和信息安全的数学基础。

自公钥密码学发明以来，素数幂运算、椭圆曲线乘法等合适的数学函数陆续被发现。这些函数都是不可逆的运算，也就是说，我们很容易从一个方向进行计算得出结果，但是想从结果倒推却是不可行的。基于这些数学函数，密码学使得数字密码和不可伪造数字签名的创建成为可能。比特币使用椭圆曲线乘法作为其公钥密码学的基础。

在比特币中，我们利用公钥学创建密钥对来控制对比特币的访问权。密钥对由私钥和公钥（由私钥派生而来）组成，公钥用于接收比特币，私钥用于对支付比特币的交易进行签名。

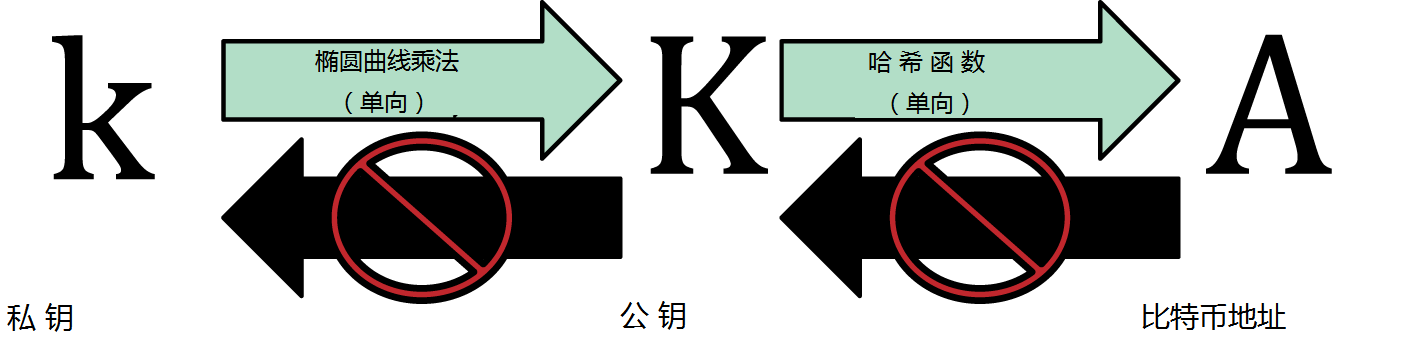
公钥与私钥在数学上的关系使得私钥可用于生成消息签名。在不泄露私钥的情况下，公钥可以对该签名的有效性进行验证。

花费比特币时，当前比特币的持有人需要随同交易同时提供公钥和签名（签名每次不同，但都由相同的私钥创建）。通过附带的公钥和签名，比特币网络中的所有参与都可以验证交易的有效性，并接受该笔交易，即确认该笔交易的资金在交易的发起时点确实由发起人所有。

|  |  |
| --- | --- |
| notes | 在大部分钱包的实现中，出于便利性，私钥和公钥是以*密钥对*的形式存储在一起的。但是由于公钥可以由私钥计算得来，因此只存储私钥也是可行的。 |

**4.1.2 私钥和公钥**

一个比特币钱包通常包含一个密钥对的集合，每个密钥对包含一个私钥和一个公钥。私钥(k)，是个数字，通常随机获取。从私钥出发，利用椭圆曲线乘法（一种单向加密函数），可以计算得出一个公钥（K）。从公钥出发，利用单向加密哈希函数可以生成比特币地址（A）。在本节中，我们从生成私钥开始，接着研究生成公钥的椭圆曲线函数，最后从公钥生成一个比特币地址。私钥、公钥、比特币地址的关系见下图：



*图4-1 私钥，公钥和比特币地址*

**4.1.3 私钥**

一个私钥就是一串随机提取的数字，拥有和控制私钥是用户控制与比特币地址关联的所有资金的根本。用户交易时需要证明使用的资金是他自己的，这就需要使用其私钥对交易进行签名。私钥在任何时候均必须保证其私密性，将私钥透露给第三方等同于把由这个私钥保护的比特币的控制权交给了第三方。私钥同样要进行备份，保护，防止意外丢失，如果私钥丢失，将是不可恢复的，受它保护的资金也就彻底丢失了。

|  |  |
| --- | --- |
| notes | 比特币私钥只是一串数字。你可以利用硬币、铅笔和纸张的来随机获取：抛256次硬币，你就获得了一个256位的二进制数字，这个数字可用作比特币钱包的私钥。生成私钥后，相应的公钥就可以利用私钥计算得出 |

**4.1.3.1 从一个随机数生成私钥**

要生成私钥，第一步也是最关键的一步是找到一个安全的熵源或者随机源。创建比特币密钥本质上就是“取得一个0到$2^{256}$之间的数字”。如果能保证随机数获取的方式是不可预测、不可重复的，则实际采用哪种方法无关紧要。比特币软件利用操作系统底层的随机数生成器来生成256比特的熵（随机数），通常操作系统的随机数是利用一种人工随机源进行初始化的，这也是为什么生成的过程中要你随机晃动鼠标几秒钟。对于真正的偏执狂，投256次骰子，并且用铅笔和纸张记录下来还是更靠谱的。

更准确的说，私钥是从1到n-1之间的任意数字，其中n是一个常量（$n=1.158\*10^{77}$,比 $2^{256}$ 略小），在比特币中这个常量是作为椭圆曲线的幂来定义的（参见第65页，《椭圆曲线密码学解释》）。为了生成这样一个密钥，我们随机取一个256比特长度的数字，并验证其是否小于n-1。以程序的术语，这通常是从一个密码学安全的随机源中抽取一长段字符串，并通过SHA256哈希算法进行计算，这样就可以很方便的生成一个256比特长度的数字。如果上述步骤结果小于n-1，我们就得到了一个合适的私钥。否则，我们需要重复以上步骤，直到最终得到一个合适的私钥。

|  |  |
| --- | --- |
| notes | 不要试图自己写代码生成密钥或者采用编程语言提供的“简单”的随机数生成器。采用密码学安全的伪随机数生成器（CSPRNG），并且从具有足够信息熵的来源中提取随机数种子。仔细研究你的随机数生成库的文档，以确定你选择的随机数生成器是密码学安全的。正确选择的CSPRNG算法对于密钥的生成至关重要。 |

以下是以十六进制形式表示的随机生成的私钥(k)（256比特的二进制数字用十六进制表示共有64位，每位代表4比特）： 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD

|  |  |
| --- | --- |
| notes | 比特币私钥的可选范围是0-$2^{256}$，$2^{256}$ 是一个难以想象的大数，以十进制表示，它大概是$10^{77}$， 而可见的宇宙大概是由$$10^{80}$个原子构成的 |

为了使用比特币核心客户端（参见第3章）创建一个新的密钥，可以使用getnewaddress命令。出于安全考虑，命令输出只显示公钥，私钥不显示。要让bitcoind进程暴露私钥，使用dumpprivkey命令。dumpprivkey命令以"Base58校验和"(Base58 checksum-encoded)编码格式表示密钥，叫做钱包导入格式（Wallet Import Format, WIF)，我们将在第76页的《私钥格式》中进一步详细介绍。以下例子展示了利用上述两个命令生成和显示私钥的步骤：

$ bitcoind getnewaddress

1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

$ bitcoind dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

*dumpprivkey*命令打开钱包并且解压出由*getnewaddress*生成的私钥。bitcoind是无法通过公钥知道私钥的，除非它们同时存储于钱包中。

|  |  |
| --- | --- |
| notes | *dumpprivkey命令不是从公钥生成一个私钥，因为这根本不可能。这个命令只是简单的从“钱包”中取出由getnewaddress生成的私钥。* |

你也可以使用命令行工具sx（参见第56页，《libbitcoin和sx工具集》）来生成和展示私钥；相应的sx命令是*newkey*：

$ sx newkey

5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

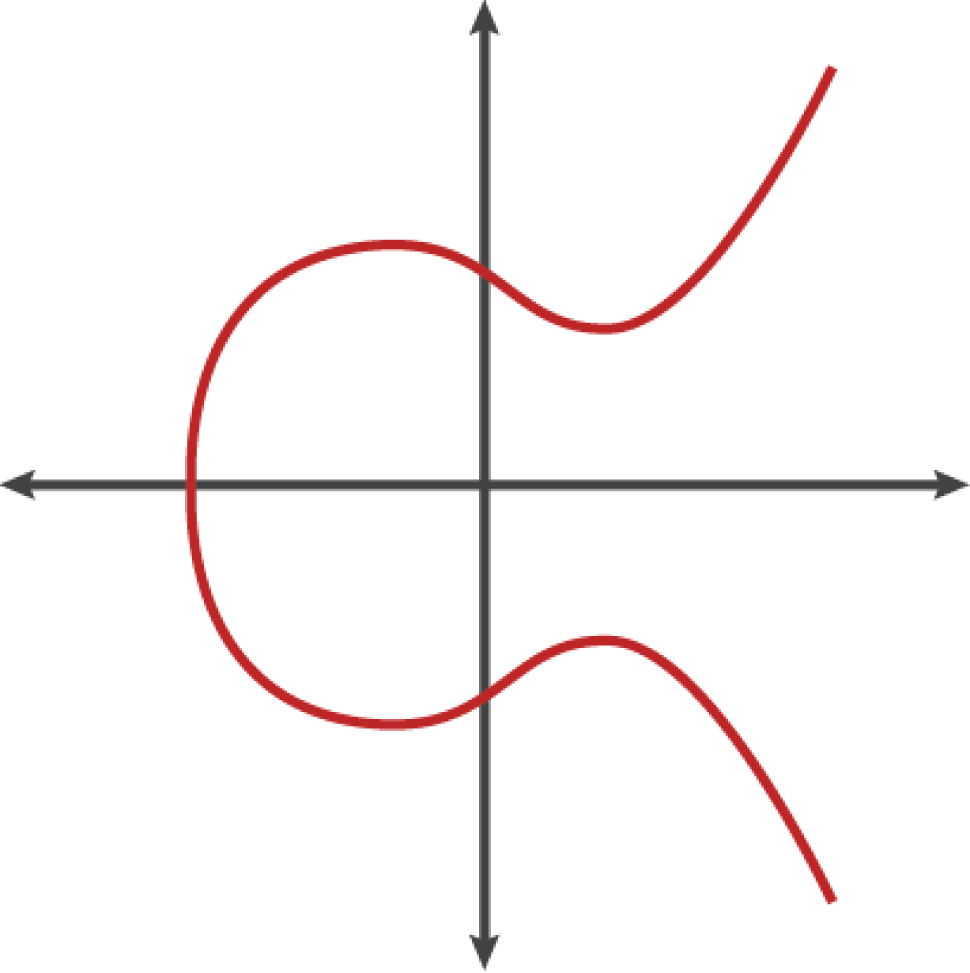
**4.1.4 公钥**

公钥是使用椭圆曲线乘法从私钥计算得来的，这是一个不可逆的过程:$ K=k\\*G $，其中，*k*是私钥，*G*是一个常数点（称之为*生成点*）,*K*是计算结果，即公钥。其反向操作被称为“查找离散对数”---已知*K*，求*k*，其难度与尝试所有k可能值的难度差不多，也就是暴力搜索。在我们演示如何从私钥生成公钥之前，我们先仔细看一下椭圆曲线加密算法。

**4.1.5 椭圆曲线加密算法解释**

椭圆曲线加密算法是一种非对称加密算法，或者叫公钥加密算法，它的基础是表示为椭圆曲线上点的加法或乘法运算的离散对数问题。

**图4-2**是一个椭圆曲线的例子，与比特币中使用的类似。



*图4-2 一条椭圆曲线*

比特币使用一个特定的椭圆曲线和一系列数学常量，这些内容在美国国家标准技术研究所（NIST）的secp256k1标准中进行了定义。secp256k1曲线是由以下函数定义的，是一条椭圆曲线：

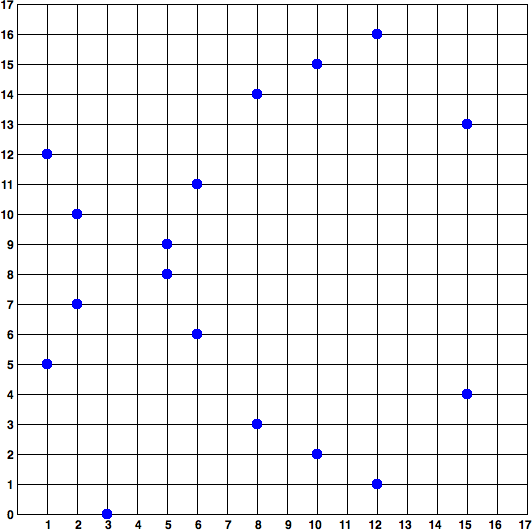
$$ y^2=(x^3+7)over(\mathbb{F}\_p) $$

或者

$$ y^2 \text{mod} ~p=(x^3+7)~\text{mod}~p $$

*mod p*(对素数p取模）表明，这个曲线是在素数幂p的有限域上，也写成$\mathbb{F}\_p$，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。

由于这条曲线是基于素数幂而不是基于实数有限域定义的，它的图像看起来像一堆散乱在两个象限上的点，很难画图表示。但是它在数学原理上与基于实数的椭圆曲线是一样的。作为一个例子，**图4-3**显示了一个基于素数幂17（远小于实际值）的有限域上的椭圆曲线，可以看到一系列点散布在网格上。而*secp256k1*比特币椭圆曲线可以想作一个在巨大网格上的更为复杂的散列点。



\*图4-3 椭圆曲线密码学：一个在F(p)上的椭圆曲线的图像，其中p=17\*

作为例子，点P，其坐标为(x,y)，是一个secpk1曲线上的一个点。可以用Python来进行检验：

|  |
| --- |
| P = (55066263022277343669578718895168534326250603453777594175500187360389116729240, 32670510020758816978083085130507043184471273380659243275938904335757337482424)  Python 3.4.0 (default, Mar 30 2014, 19:23:13)  [GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin  Type "help", "copyright", "credits" or "license" for more information.  >>> p = 115792089237316195423570985008687907853269984665640564039457584007908834671663  >>> x = 55066263022277343669578718895168534326250603453777594175500187360389116729240  >>> y = 32670510020758816978083085130507043184471273380659243275938904335757337482424  >>> (x \*\* 3 + 7 - y\*\*2) % p  0 |

在椭圆曲线数学中，有个点叫做"无穷远点"，它与0在加法中扮演的角色大致相同。在计算机中，它有时也表示为x=y=0（虽然不满足椭圆曲线方程，但它是一个简单可校验的独立案例）。

还有个“+”号运算符，叫做“加法”，它有点类似于小时候学过的传统实数加法。给定在椭圆曲线上的两个点$ P\_1 $, $P\_2$，存在第三个点$ P\_3=P\_1+P\_2$，也在椭圆曲线上。

在几何学上，这个点$P\_3$是通过在$P\_1$和$P\_2$间绘制一条直线来计算的。这条直线将与椭圆曲线相交于一点$ P\_3’=(x,y) $，通过x轴映射，得到$P\_3=(x,-y)$。

有很多特殊案例解释了“无穷远点”存在的必要性。

如果$P\_1$和$P\_2$是同一点，那$P\_1$，$P\_2$的连接线必然与曲线在$P\_1$点相切，曲线有且仅有一个新的点与直线相交。可以使用微积分技术来计算切线的斜率。虽然我们感兴趣的限制到曲线上两个坐标均为整数的点，这些技术仍然能奇怪的满足要求！

在某些情况下（比如：$ P\_1 $和$ P\_2 $的x值相同，但y值不同），切线将是垂直的，则$ P\_3 $=“无穷远点”。

如果P1是无穷远点，那么$ P\_1+P\_2=P\_2$。相应的，如果P2是无穷远点，那么$ P\_1+P\_2=P\_1$。这显示了其与0一样的性质。

事实证明，这里的“+”号符合联合率，也就是说$(A+B)C=A(B+C)$。这意味着，我们可以不带括号的写$A+B+C$，而不会有任何歧义。

至此，我们已经定义了加法，我们也可以按照标准的方式通过扩展加法的方式来定义乘法。对于椭圆曲线上的点P，如果k是个整数，那么$kP=P+P+P+...+P$ (k次）。需要注意的是，在这种情况下，k有时也叫做“指数”。

**4.1.6 生成一个公钥**

从一个密钥（形式上是一个随机生成的数字k）出发，我们将它与曲线上预定义的点相乘，可以生成曲线上的另一个点，这就是相应的公钥K，而这个预定义的点叫做生成点G。生成点是作为secp256k1标准的一部分定义的，对于比特币而言，其所有密钥均使用相同G点。

$$ K=k\*G $$

这里，k是密钥，G是生成点，K是生成的公钥，椭圆曲线上的一个点。由于生成点对所有比特币用户来说都是一样的，一个密钥k与G相乘后总能得到相同的公钥K。k与K将的关系是固定的，但是只能从k到K单向计算。这也是比特币地址（从K衍生而来）可以与任何人共享，却不会暴露用户私钥（k）的原因。

|  |  |
| --- | --- |
| notes | 私钥可以转换为公钥，但是公钥不能转换回私钥，因为数学上，这种计算是单向的。 |

实现椭圆曲线乘法，我们使用之前生成的私钥k，与生成点G相乘，以获得公钥K：

K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD \* G

公钥K定义成一个点$ K=(x,y)$:

k = (x,y)

其中，

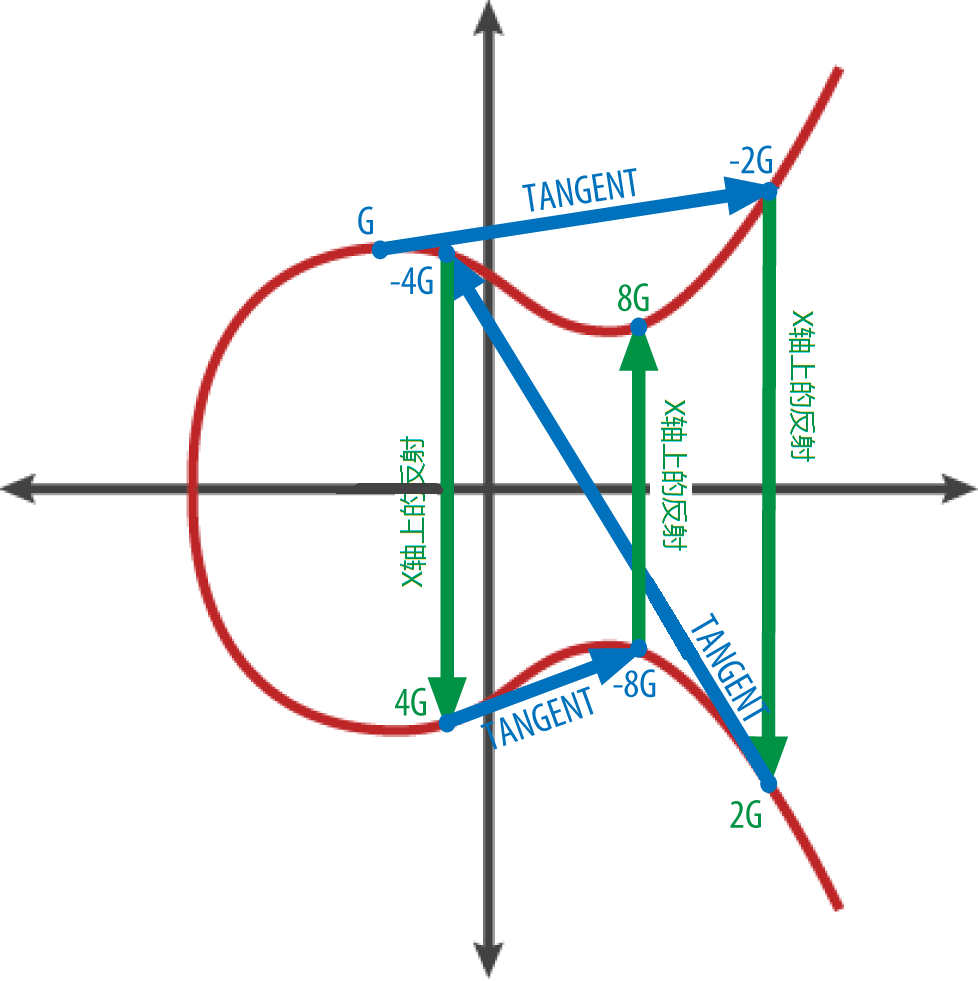
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

为了形象化演示一个点与整数的相乘，我们使用一个简单的基于实数的椭圆曲线--记住，实数与整数在数学上都是一样的。我们的目标是找到生成点G的倍数kG。也就是G相加k次。在椭圆曲线中，一个点与其自身相加等同于在这个点上画一条切线，找到切斜与曲线相交的点，相交点相对x轴对称的点就是我们要找的点。

**图4-4**显示了如何利用几何学操作在曲线上获得G，2G，4G。

|  |  |
| --- | --- |
| notes | 大多数比特币实现都利用OpenSSL加密库来完成椭圆曲线算法的计算。比如，为了获得公钥，就会用到EC\_POINT\_mul()函数。 |



*图4-4 椭圆曲线密码学：演示G在椭圆曲线上被整数k相乘*

**4.2 比特币地址**

比特币地址是一串由数字和字母构成的字符串，可以分享给任何想给你转钱的人。地址是从公钥转换而来的，包含数字和字母，其第一个字符是1（数字）。下面就是一个比特币地址的例子：

1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

比特币地址在交易中通常以资金“接收者”的形式出现。如果我们将比特币交易与纸质支票做个类比，那么比特币地址就相当于支票上的受益人，也就是我们写入支票“受款人”后的内容。在纸质支票上，受益人可以是一个银行账户持有者的姓名，也可以是公司，机构，甚至现金。由于支票不需要指定账号，只是使用一个抽象的名字作为资金接收人，因此支票作为支付工具非常灵活。比特币交易采用了类似的抽象即比特币地址，使其同样极具灵活性。比特币地址可以指代一个密钥对的拥有者，也可以指代其他一些东西，比如支付脚本，我们将在第132页《支付到脚本哈希（P2SH）》中讲到。现在，我们通过一个简单例子，了解比特币地址是如何由公钥产生，并可以代表公钥的。

比特币地址是通过一个单向加密哈希算法从公钥推导出来的。“哈希算法”是一种单向函数，它可以对任意长度的输入进行计算，生成输入信息的指纹或者“哈希”。加密哈希函数在比特币中应用广泛，包括比特币地址，脚本地址，以及挖矿中的工作量证明算法等。用于从公钥创建比特币地址的算法是安全哈希算法（SHA）和RACE完整性原语求值信息摘要算法(RIPEMD)，在应用中用的是其中的两个特定算法，SHA256和RIPEMD160。

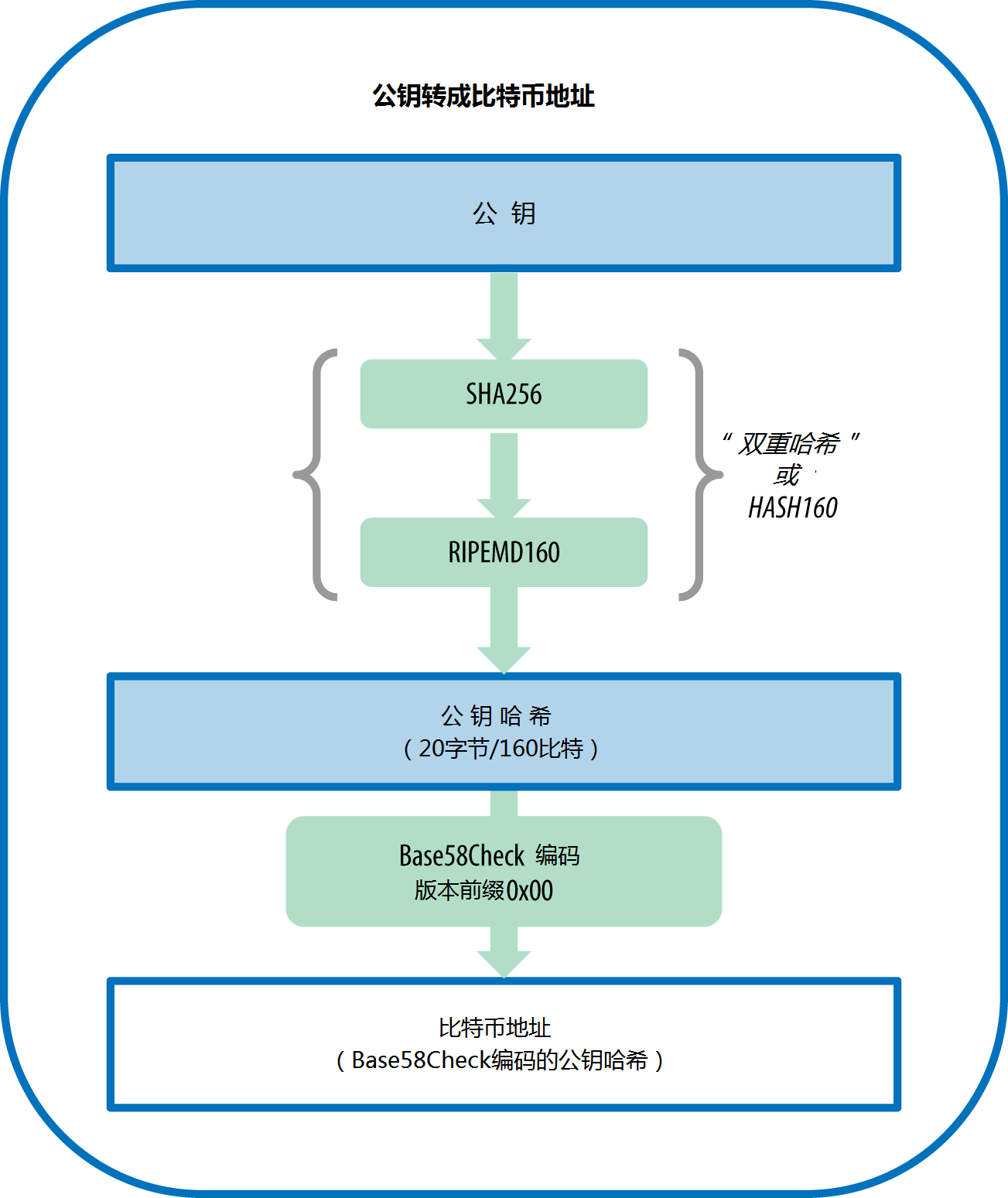
我们从公钥K开始，计算它的SHA256哈希值，然后从其结果中计算RIPEMD160的哈希值，这样我们就创建了一个160比特（20字节）长度的数字：

A = RIPEMD160(SHA256(K))

其中K是公钥，A是计算结果，即比特币地址

|  |  |
| --- | --- |
| notes | 比特币地址与公钥不一样，比特币地址是利用单向哈希函数从公钥衍生得来的。 |

比特币地址基本上是以Base58Check编码形式（参见第72页，《Base58和Base58Check编码》）展现给用户的，它使用58个字符（一个Base58数字系统）和一个校验码以让人易于阅读，避免歧义，防止地址转录及输入时犯错。Base58Check在比特币中还有很多其他应用，比如比特币地址、用户私钥、加密的密钥、脚本哈希等。在下一节，我们将看到Base58Check的编码和解码机制，以及他们的结果展示。**图4-5**阐述了从公钥到比特币地址的转换过程。



*图 4-5 从公钥到比特币地址：将公钥转换为比特币地址*

**4.2.1 Base58和Base58Check编码**

为了以更少的符号，更紧凑的方式表示一个很大的数，很多计算机系统采用超过十的进制（底数），并混合使用字母和数字的表示法。举例来说，传统十进制系统使用10个从0到9的数字字符，十六进制系统使用16个字符，包括10个数字字符和A到F 6个字母。一个数采用十六进制表示的话就会比用十进制表示来得短。Base-64编码采用26个小写字母，26个大小字母，10个数字字符，以及两个额外的字符，比如“+”，“/”将二进制数据转换为基于文本的数据，比如email。Base-64主要用于email中二进制附件的编码。Base58也是一个基于文本的二进制编码格式，用于比特币及很多其他密码货币系统中。它在紧凑表示、可读性和错误检测及预防方面提供了一种平衡。Base58是Base64的一个子集，使用大小写字母和数字，但是省略了一些容易混淆的字符。具体来说，Base58是Base64中去掉0(数字零)、O(字母o的大写)、l(小写的L）、I(大写的i)、字符“+”和字符“/”。或者更简单的说，它是个去掉四个字符（0,O,l,I)的大小写字母与数字的集合。

*例4-1, 比特币的Base58字符表*

123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

为了增加额外安全性，防止打字和转录错误，常用于比特币的Base58Check编码在Base58编码格式的基础上增加了内置的错误检测码。校验码为4个字节长，添加到需要编码的数据后面。校验码从需要编码数据的哈希值得到，从而可以检测和避免转录和输入错误。取得一个Base58Check编码的数据后，解码软件可以计算原始数据的校验码，并与数据中的校验码进行比对。如果两者不一致，就说明原始数据有误，Base58Check数据无效。举例来说，这可以避免钱包将输错的比特币地址当做一个有效目标，防止了资金丢失风险。

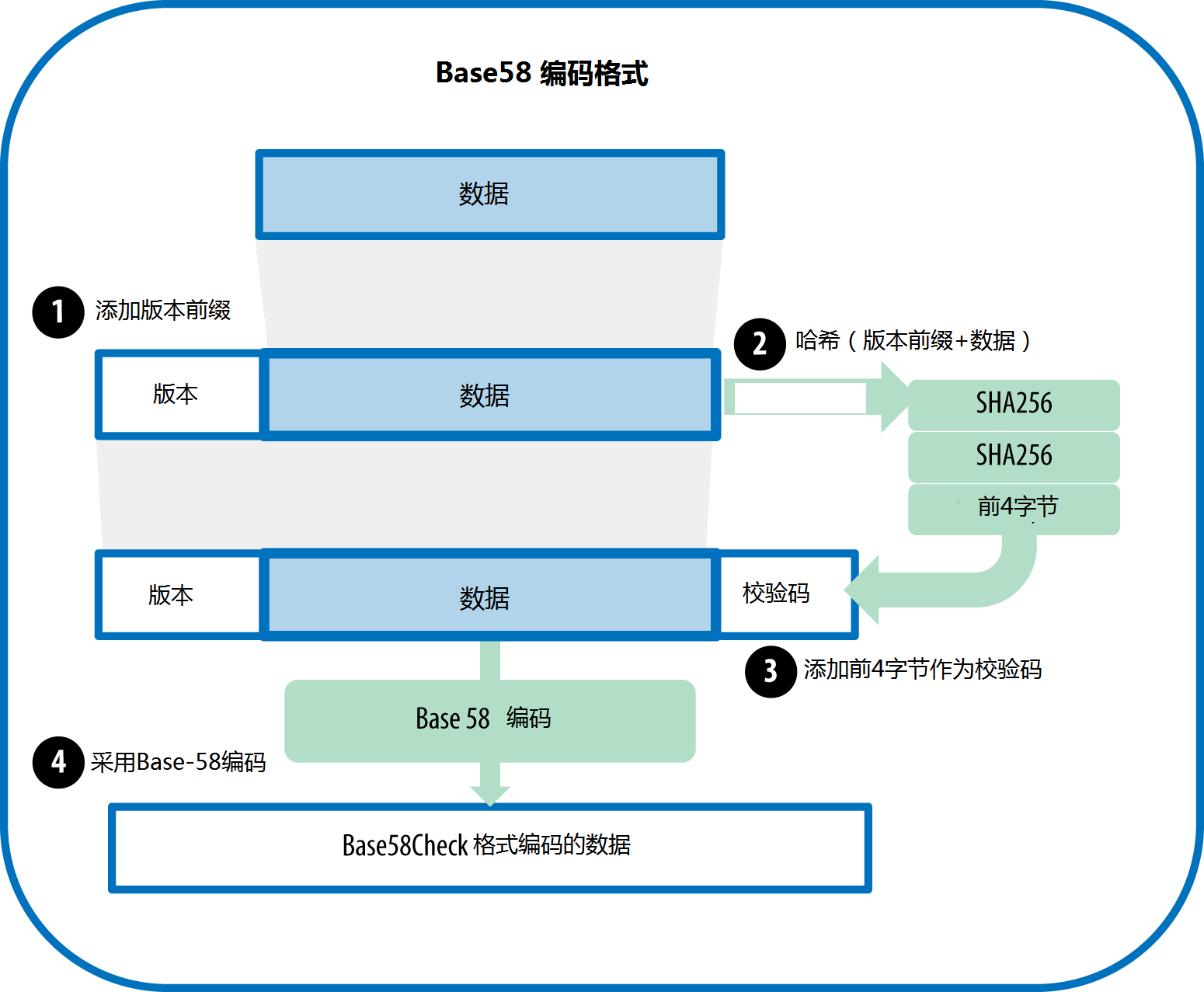
为了将数据（一个数）转换为Base58Check编码格式，我们首先添加一个前缀到数据中，称之为“版本字节”，这可以让我们更容易判断数据的类型。例如，比特币地址的前缀是0（十六进制表示为0x00），而私钥的前缀为128（十六进制表示为0x80）。常用版本前缀请参看表4-1。

接下来，我们计算“双重SHA”校验码，即对前面得到的数据（前缀+数据）进行两次SHA256哈希计算：

checksum = SHA256(SHA256(prefix+data))

从结果的32字节哈希值（哈希的哈希），我们取前面四个字节。这四个字节作为错误检查码，或者校验码附加到数据的最后。

这样，就形成了由三部分（前缀，原始数据，校验码）组成的数据。现在可以利用前面介绍的Base58字符表，对结果数据进行编码。图4-6描述了Base58Check的编码过程。



*图4-6 Base58Check编码：一种Base58的，版本化的，可校验的比特币数据无歧义编码格式*

在比特币中，大多数需要向用户展示的数据均以Base58Check格式进行编码，这使得数据紧凑，易读，易于检查错误。Base58Check编码的版本前缀用于创建易于辨别的格式，这意味着，当以Base58编码时，使用Base58Check编码后的数据头部包含了特定的字符。这个字符使用户很容易判断出数据类型，以及如何去使用它。比如，Base58Check编码的比特币地址以1打头，而Base58Check编码的WIF格式私钥以5打头。一些版本前缀的例子，及其编码后的Base58字符见表4-1.

*表4-1 Base58Check版本前缀及编码后结果的举例*

| **类型** | **版本前缀（十六进制）** | **Base58结果前缀** |
| --- | --- | --- |
| 比特币地址 | 0x00 | 1 |
| 脚本支付地址 | 0x05 | 3 |
| 比特币测试网地址 | 0x6F | m或n |
| 私钥钱包导入格式 | 0x80 | 5, K or L |
| BIP38加密私钥 | 0x0142 | 6P |
| BIP32扩展公钥 | 0x0488B21E | xpub |

我们来看一下完整的比特币地址生成过程，从私钥到公钥（椭圆曲线上的一个点），到双重哈希的地址，最后Base58Check编码。例4-2的C++代码一步步展示了从私钥一直到Base58Check编码的比特币地址的完整的过程。代码使用了libbitcoin库中的一些函数（第56页《其他替代客户端，库和工具集》中介绍过）

*例4-2 从私钥创建Base58Check编码的比特币地址*

|  |
| --- |
| #include <bitcoin/bitcoin.hpp>  int main()  {  // 私钥  bc::ec\_secret secret = bc::decode\_hash("038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");  // 获取公钥  bc::ec\_point public\_key = bc::secret\_to\_public\_key(secret);  std::cout << "Public key: " << bc::encode\_hex(public\_key) << std::endl;  // 创建比特币地址  // 通常使用一下语句:  // bc::payment\_address payaddr;  // bc::set\_public\_key(payaddr, public\_key);  // const std::string address = payaddr.encoded();  // 计算作为P2PKH地址的公钥哈希  const bc::short\_hash hash = bc::bitcoin\_short\_hash(public\_key);  bc::data\_chunk unencoded\_address;  // 保留25字节  // [ version:1 ]  // [ hash:20 ]  // [ checksum:4 ]  unencoded\_address.reserve(25);  // Version byte, 0 is normal BTC address (P2PKH).  unencoded\_address.push\_back(0);  // 哈希数据  bc::extend\_data(unencoded\_address, hash);  // Checksum is computed by hashing data, and adding 4 bytes from hash.  bc::append\_checksum(unencoded\_address);  // 最后，我们必须结果编码为比特币的base58格式  assert(unencoded\_address.size() == 25);  const std::string address = bc::encode\_base58(unencoded\_address);  std::cout << "Address: " << address << std::endl;  return 0;  } |

代码使用预定义好的私钥，使得每次运行都可以得到相同的比特币地址，就像**例4-3**展示的一样：

*例4-3 编译运行这段代码*

|  |
| --- |
| # 编译addr.cpp代码  $ g++ -o addr addr.cpp $(pkg-config --cflags --libs libbitcoin)  # 运行可执行文件addr  $ ./addr  Public key: 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa  Address: 1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK |

**4.2.2 密钥格式**

不管私钥还是公钥都可以表示为一系列不同的格式。这些表现形式均对同样数字进行编码，尽管他们看起来并不一样。这些格式主要作用在于方便用户阅读及密钥转录，避免引入错误。

**4.2.2.1 私钥格式**

私钥可以表示为几种不同格式，所有格式均代表与之对应的相同的256比特长的数字。**表4-2**显示三种用于表示私钥的常用格式。

*表4-2 私钥表示形式（编码格式）*

| **类型** | **前缀** | **说明** |
| --- | --- | --- |
| Hex | 无 | 64位十六进制数字 |
| WIF | 5 | Base58Check编码：带128前缀和32比特校验码的Base58编码 |
| WIF-compressed | K 或L | 与上面一样，但编码前加一个0x01后缀 |

**表4-3**通过这三种格式展示同一个私钥

*表4-3 例子：相同密钥，不同格式*

| **格式** | **私钥** |
| --- | --- |
| Hex | 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD |
| WIF | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn |
| WIF-compressed | KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ |

以上这些是用不同格式对相同密钥编码后的结果。虽然看起来不同，但是一种编码格式可以很容易的转换为另一种格式。

**4.2.2.2 从Base58Check到十六进制解码**

使用sx工具包（参看第56页《libbitcoin和sx工具集》）可以让我们很容易写出脚本或者命令行“管道”来操控比特币密钥，地址和交易。你可以使用sx工具集通过命令行来解码Base58Check格式。

我们使用*base58check-decode*命令：

$ sx base58check-decode 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd 128

结果是一个十六进制格式的密钥，跟着一个钱包导入格式（WIF）的版本前缀128。

**4.2.2.3 从十六进制到Base58Check编码**

为了编码为Base58Check格式（与前述命令刚好相反），我们提供十六进制的私钥，跟着一个钱包导入格式（WIF）的版本前缀128：

$ sx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd 128

5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

**4.2.2.4 从十六进制（压缩格式密钥）到Base58Check的编码**

为了将“压缩”格式的私钥（参见80页《压缩格式私钥》）编码到Base58Check，我们在十六进制密钥的最后加上后缀01，然后进行编码：

$ sx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 128

KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

生成的WIF-压缩格式结果以字母“K”开头。这表示私钥带有“01”后缀，并且只可用于生成压缩格式的公钥（参见第78页《压缩格式公钥》）。

**4.2.2.5 公钥格式**

公钥也同样采用不同的格式显示，最重要的是*压缩*和*非压缩*公钥格式。

就像我们前面看到的，公钥是一个在椭圆曲线上的点，包含一对坐标（x,y）。它通常的表现形式为：04前缀开头，紧跟两个256比特长度的数字，一个代表x坐标，另一个代表y坐标。04前缀用于区别非压缩公钥和压缩公钥，压缩公钥是以02或03开头的。

下面是之前我们通过私钥生成的公钥，以x,y坐标表示之：

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

下面是520比特（130个十六进制数字）的数字显示的相同的密钥，04开头，紧跟着x坐标和y坐标，即04 x y：

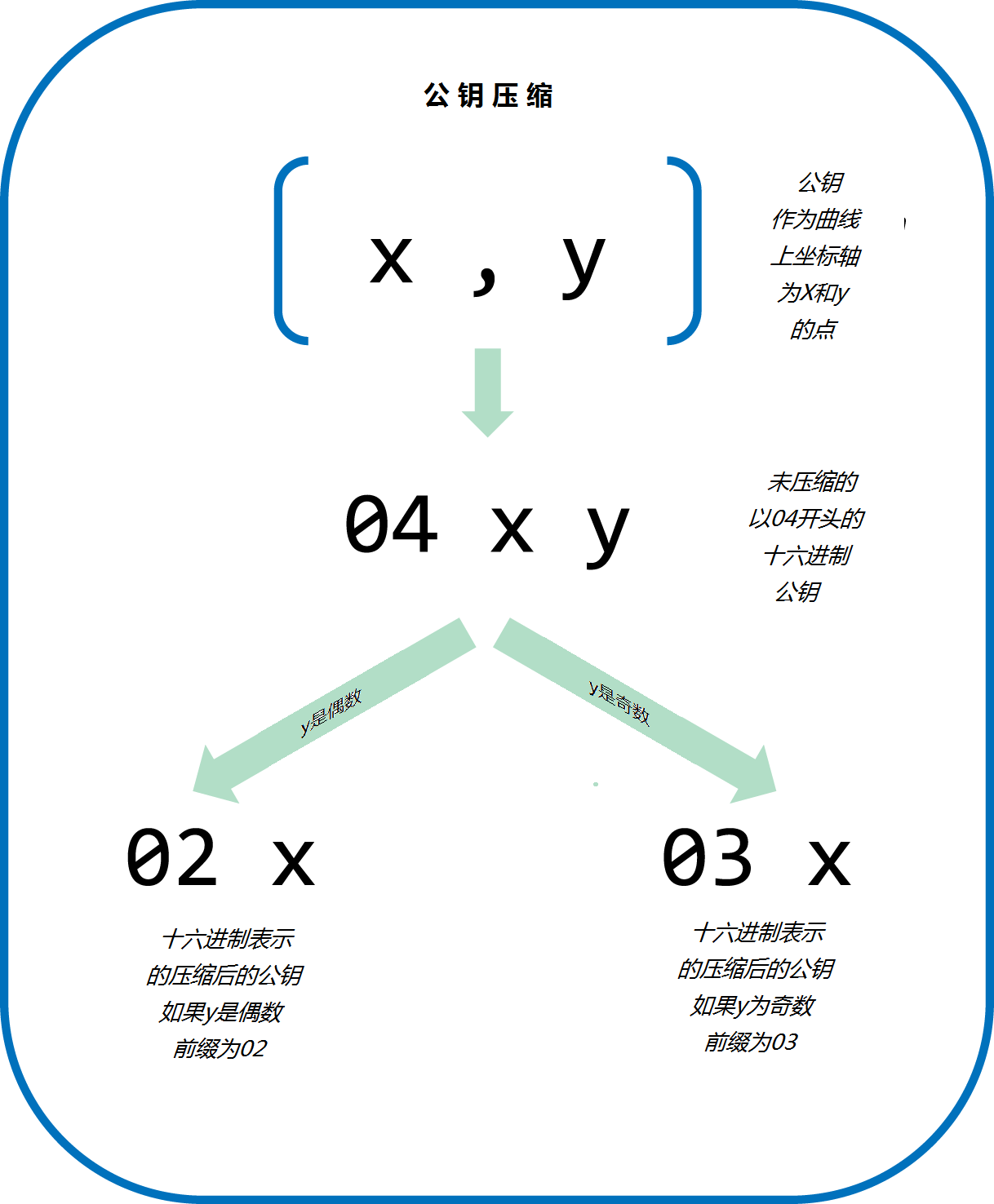
K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

**4.2.2.6 压缩公钥**

压缩公钥引入比特币的目的是为了降低交易大小，使得保存比特币区块链数据库的节点更加节省磁盘空间。大多数交易均包含公钥，用以验证所有者的身份并花费比特币。每个公钥要求520比特长度（前缀+x+y），每个区块由几百个交易组成，而每天都有成千上万个交易加入区块，这给区块链的存储带来了一定负担。

我们在第65页《公钥》中看到，公钥其实是椭圆曲线上的一个点。因为曲线代表了一个数学公式，曲线上的一个点代表了方程的一个解，那么，如果我们知道x坐标，y坐标也能通过求解方程得到：$ y^2 ~\text{mod}~p=(x^3+7)~\text{mod}~p $ 。这允许我们可以只存储公钥中的x坐标，而把y坐标省略掉，这样就将所需空间减少了256比特，几乎少了一半。长期来看，交易中节省的空间是相当可观的。

未压缩公钥带04前缀，而压缩公钥开始于02或者03。我们看看为什么它有两个不同的前缀：方程的左边是 $ y^2 $，这意味着y的解是个平方根，可以是正值或者负值。或者直观的说，结果中y坐标可以位于x轴之上，也能位于x轴之下。就如我们在**图4-2**看到的曲线图像，它是对x轴对称的。所以，当我们省略y坐标时，我们必须保留y的*符号*（正还是负），换言之，我们必须记住这个点是在x轴之上还是x轴之下，上下两个点代表了两个不同的公钥。当我们在素数幂P的有限域内以二进制算术计算椭圆曲线的时候，y坐标可能是偶数也可能是奇数，分别对应y坐标的正负符号。这样，为了区分两个可能的y值，我们在压缩公钥中用前缀02代表偶数，03代表奇数，允许软件从x坐标准确推断出y坐标，并将公钥解压成完整坐标的点。公钥压缩过程参见**图4-7**。



*图4-7 公钥压缩*

这是前面生成的，用压缩格式存储的264比特（66十六进制长）公钥，以03前缀，代表y坐标是奇数：

K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

压缩公钥对应的是相同的私钥，也就是从相同的私钥生成的。但是它看起来与非压缩公钥并不一样。更重要的是，如果我们将一个压缩公钥使用双重哈希函数（RIPEMD160(SHA256(K))）转换为一个比特币地址，那么将生成一个不同的地址。这很容易引起混淆，因为这意味着一个私钥可以生成两种不同表达方式的公钥（压缩、非压缩），进一步还生成了两个不同的比特币地址。但是对应这两个不同比特币地址的私钥却是相同的。

压缩公钥已逐渐成了比特币客户端的默认配置，这对降低交易大小，从而降低区块链大小有一定的正面作用。但是，并不是所有客户端都支持压缩公钥格式。支持压缩密钥的新客户端必须能兼容不支持压缩公钥的老客户端发来的交易。这对于从其他钱包应用中导入私钥时尤为重要，因为新钱包需要扫描区块链查找与这些引入的私钥相关的所有交易。比特币钱包到底该扫描哪种类型的地址呢？非压缩公钥生成的地址还是压缩公钥生成的地址？这两种地址都是有效的比特币地址，他们都可以被私钥签名，但他们的确是不同的两个地址！

为了解决这个麻烦，当私钥从钱包中导出时，用于表示私钥的*钱包导入格式*（WIF）在新比特币钱包中采用了不同的实现方式，它可以指示私钥已经被用于创建压缩公钥，并且也生成了*压缩*比特币地址。这就使得导入钱包可以辨别私钥是从旧的钱包来还是新的钱包来，从而根据非压缩还是压缩的比特币地址从区块链上搜索交易。我们将在下一节来看看这个过程的细节。

**4.2.2.7 压缩私钥**

名词“压缩私钥”有点讽刺意味，实际上私钥以WIF-压缩格式导出时比那些“未压缩”的还长了一字节。这是因为它加了一个01的后缀，这个后缀标注了它是从一个新钱包来的，只能用于生成压缩公钥。私钥既没有被压缩，也不可能被压缩。“压缩私钥”的真正含义是“只能用于生成压缩公钥的私钥”，同样，“非压缩私钥”就是“只能生成非压缩公钥的私钥”。为避免更多混淆，你最好将导出格式称之为“WIF-压缩格式”或者“WIF”，而不是将私钥称之为“压缩的”或者“非压缩的”。

记住，这些格式不是可交换的。在实现了压缩公钥的新钱包中，私钥只能被转换为WIF-压缩格式公钥（带K或L前缀）。如果钱包是一个较老的实现，还不能支持压缩公钥，这种情况，私钥只能导出为WIF（带5前缀）。这样做的目的是通知导入这些私钥的钱包，告诉它是以压缩公钥以及他们相应的比特币地址进行搜索，还是采用非压缩公钥及其地址进行搜索。

如果比特币钱包支持压缩公钥，它将在所有交易中使用压缩公钥。钱包中的私钥用于在曲线中生成公钥点，这将被进行压缩。压缩后公钥用于生成比特币地址，并用于交易当中。当在一个支持压缩公钥的钱包是导出私钥时，钱包导入格式将被改的，添加一个01的后缀到私钥中。经过Base58Check编码后的私钥称之为“压缩WIF”，开始于字母K或L，而不是像老钱包使用的WIF编码（未压缩）一样开始于数字“5”。

*表4-4显示同一个密钥的WIF和WIF压缩格式*

| **格式** | **私钥** |
| --- | --- |
| 十六进制 | 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD |
| WIF | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn |
| 十六进制压缩 | 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD\_01\_ |
| WIF-压缩 | KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ |

|  |  |
| --- | --- |
| notes | “压缩私钥”是一个误会！它们并没有被压缩；相反的，WIF-压缩格式只是指示它们只能用于生成压缩公钥以及他们相应的比特币地址。很讽刺的，一个“WIF-压缩”编码的私钥比“不压缩”的格式长一个字节，因为它被加上了一个“01”的后缀用于与“非加密”进行区分。 |

**4.3 在Python中实现密钥和地址**

最全的Python比特币库是维塔里克.布特林（Vitalik Buterin，就是那个后来发明了以太坊的大牛）发明的*pybitcointools*。在**例4-4**中，我们利用这个库（import时使用bitcoin）来生成密钥和地址，并以不同格式进行展示。

*例4-4 利用pybitcointools库生成并格式化密钥和地址*

|  |
| --- |
| import bitcoin  # 生成一个随机私钥  valid\_private\_key = False  while not valid\_private\_key:  private\_key = bitcoin.random\_key()  decoded\_private\_key = bitcoin.decode\_privkey(private\_key, 'hex')  valid\_private\_key = 0 < decoded\_private\_key < bitcoin.N  print "Private Key (hex) is: ", private\_key  print "Private Key (decimal) is: ", decoded\_private\_key  # 私钥转换为 WIF 格式  wif\_encoded\_private\_key = bitcoin.encode\_privkey(decoded\_private\_key, 'wif')  print "Private Key (WIF) is: ", wif\_encoded\_private\_key  # 添加后缀"01" 以提示这是个压缩的私钥  compressed\_private\_key = private\_key + '01'  print "Private Key Compressed (hex) is: ", compressed\_private\_key  # 从压缩私钥生成WIF格式(WIF-compressed)  wif\_compressed\_private\_key = bitcoin.encode\_privkey(  bitcoin.decode\_privkey(compressed\_private\_key, 'hex'), 'wif')  print "Private Key (WIF-Compressed) is: ", wif\_compressed\_private\_key  # 将椭圆曲线生成点与私钥相乘得到一个公钥点  public\_key = bitcoin.fast\_multiply(bitcoin.G, decoded\_private\_key)  print "Public Key (x,y) coordinates is:", public\_key  # 十六进制编码, 前缀 04  hex\_encoded\_public\_key = bitcoin.encode\_pubkey(public\_key,'hex')  print "Public Key (hex) is:", hex\_encoded\_public\_key  # 压缩公钥，依据y的奇偶数性质调整前缀  (public\_key\_x, public\_key\_y) = public\_key  if (public\_key\_y % 2) == 0:  compressed\_prefix = '02'  else:  compressed\_prefix = '03'  hex\_compressed\_public\_key = compressed\_prefix + \  bitcoin.encode(public\_key\_x, 16)  print "Compressed Public Key (hex) is:", hex\_compressed\_public\_key  # 从公钥生成比特币地址  print "Bitcoin Address (b58check) is:", \  bitcoin.pubkey\_to\_address(public\_key)  # 从压缩公钥生成压缩比特币地址  print "Compressed Bitcoin Address (b58check) is:", \  bitcoin.pubkey\_to\_address(hex\_compressed\_public\_key) |

例4-5显示这段代码的运行结果

*例 4-5. 运行 key-to-address-ecc-example.py*

|  |
| --- |
| $ python key-to-address-ecc-example.py  Private Key (hex) is:  3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa6  Private Key (decimal) is:  26563230048437957592232553826663696440606756685920117476832299673293013768870  Private Key (WIF) is:  5JG9hT3beGTJuUAmCQEmNaxAuMacCTfXuw1R3FCXig23RQHMr4K  Private Key Compressed (hex) is:  3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa601  Private Key (WIF-Compressed) is:  KyBsPXxTuVD82av65KZkrGrWi5qLMah5SdNq6uftawDbgKa2wv6S  Public Key (x,y) coordinates is:  (41637322786646325214887832269588396900663353932545912953362782457239403430124L,  16388935128781238405526710466724741593761085120864331449066658622400339362166L)  Public Key (hex) is:  045c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec243bcefdd4347074d44bd7356d6a53c495737dd96295e2a9374bf5f02ebfc176  Compressed Public Key (hex) is:  025c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec  Bitcoin Address (b58check) is:  1thMirt546nngXqyPEz532S8fLwbozud8  Compressed Bitcoin Address (b58check) is:  14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3 |

例4-6是另一个例子，使用python的ECDSA库来计算椭圆曲线，没有使用任何特定的比特币库

*例4-6. 演示比特币密钥中使用的椭圆曲线数学的脚本*

|  |
| --- |
| import ecdsa  import os  from ecdsa.util import string\_to\_number, number\_to\_string  # secp256k1, http://www.oid-info.com/get/1.3.132.0.10  \_p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2FL  \_r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141L  \_b = 0x0000000000000000000000000000000000000000000000000000000000000007L  \_a = 0x0000000000000000000000000000000000000000000000000000000000000000L  \_Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798L  \_Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8L  curve\_secp256k1 = ecdsa.ellipticcurve.CurveFp(\_p, \_a, \_b)  generator\_secp256k1 = ecdsa.ellipticcurve.Point(curve\_secp256k1, \_Gx, \_Gy, \_r)  oid\_secp256k1 = (1, 3, 132, 0, 10)  SECP256k1 = ecdsa.curves.Curve("SECP256k1", curve\_secp256k1, generator\_secp256k1, oid\_secp256k1)  ec\_order = \_r  curve = curve\_secp256k1  generator = generator\_secp256k1  def random\_secret():  convert\_to\_int = lambda array: int("".join(array).encode("hex"), 16)  # 从操作系统的密码学安全随机生成器收集256比特的随机数  byte\_array = os.urandom(32)  return convert\_to\_int(byte\_array)  def get\_point\_pubkey(point):  if point.y() & 1:  key = '03' + '%064x' % point.x()  else:  key = '02' + '%064x' % point.x()  return key.decode('hex')  def get\_point\_pubkey\_uncompressed(point):  key = '04' + \  '%064x' % point.x() + \  '%064x' % point.y()  return key.decode('hex')  # 生成一个新私钥  secret = random\_secret()  print "Secret: ", secret  # 获取公钥点  point = secret \* generator  print "EC point:", point  print "BTC public key:", get\_point\_pubkey(point).encode("hex")  # 给定点(x, y) 我们利用下面生成对象:  point1 = ecdsa.ellipticcurve.Point(curve, point.x(), point.y(), ec\_order)  assert point1 == point |

例4-7显示了这个例子的输出。

\*例4-7，安装python ECDSA库并运行ec\_math.py脚本

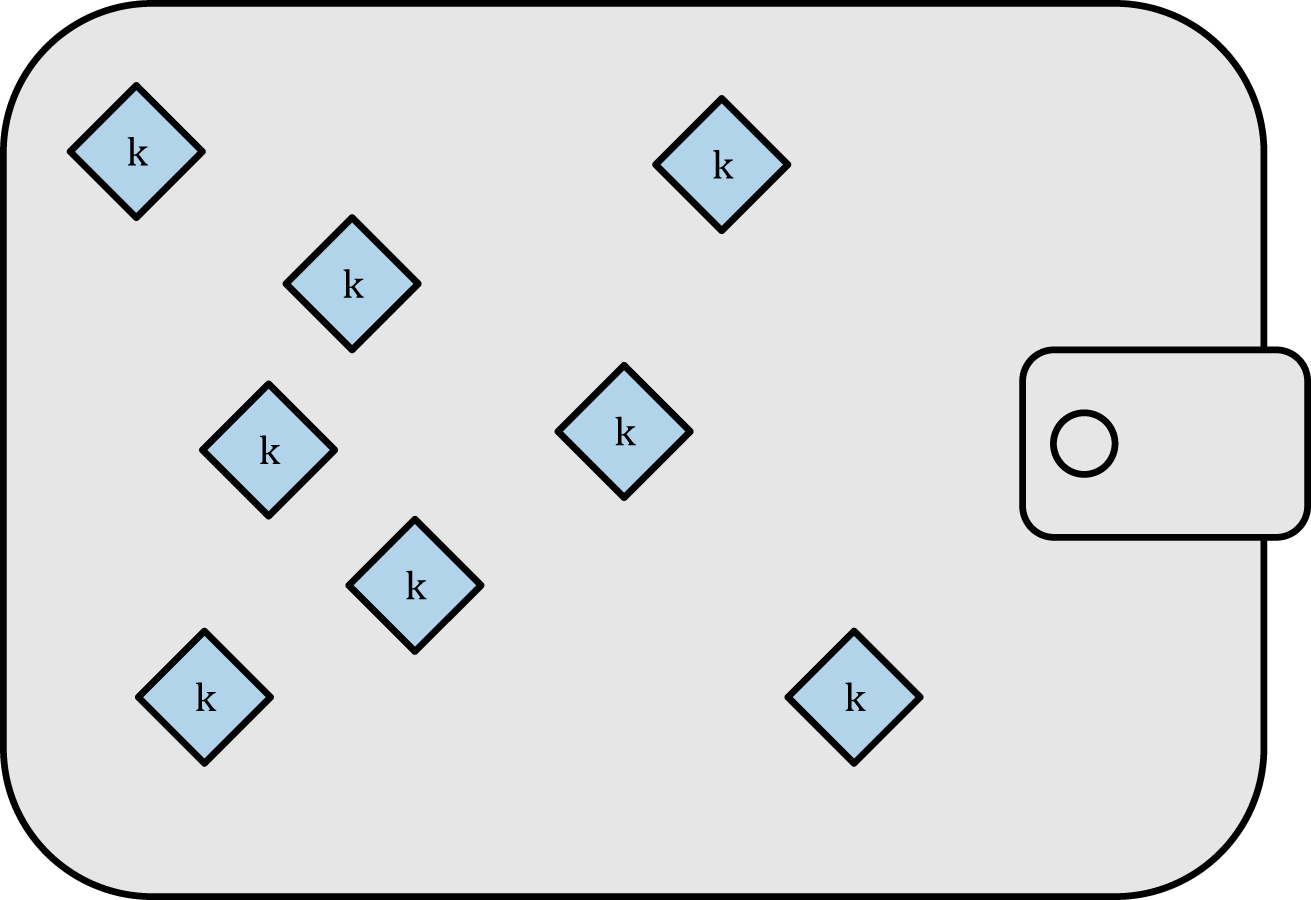
|  |
| --- |
| $ # Install Python PIP package manager  $ sudo apt-get install python-pip  $ # Install the Python ECDSA library  $ sudo pip install ecdsa  $ # Run the script  $ python ec-math.py  Secret:  38090835015954358862481132628887443905906204995912378278060168703580660294000  EC point:  (700488535318671794898577504976069662723825834713229354546245955400072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

**4.4 钱包**

钱包是保存私钥的容器，通常以结构化文件或者简单数据库的方式实现。另一生成密钥的方式是*确定性密钥生成*。这种情况下，可以通过单向哈希函数从上一个私钥中生成一个新的私钥，按顺序连接，形成一个链条。如果需要重建这个链，你只需生成第一个私钥（称之为种子或者主密钥）即可生成整个序列。在本节中，我们将检查不同的的密钥生成方式以及相应的钱包结构。

|  |  |
| --- | --- |
| notes | 比特币钱包包含的是密钥，不是比特币。每个用户拥有一个包含很多密钥的钱包。钱包实际上是一个密钥链，包含一对对的私/公钥（参看第63页《私钥和公钥》）。用户使用密钥对交易进行签名，以证明交易输出是他们的（他们的比特币）。比特币以交易输出的方式存储于区块链上（通常记为vout或者txout）。 |

**4.4.1 非确定性（随机）钱包**

在早期的比特币客户端中，钱包是随机生成私钥的简单集合。这种类型的钱包称为Type-0非确定性钱包。举例来说，比特币核心客户端在第一次启动时预生成100个随机私钥，后面则根据需要继续产生，每个密钥只使用一次。这种类型钱包被戏称为“只是一堆密钥”，或者简写为“JBOK”，这种钱包已逐渐被确定性钱包所替代，因为它们使之难以管理，备份和导入。随机密钥的缺点在于如果你生成了太多的密钥，你必须经常对所有这些密钥进行备份，如果没有备份，一旦钱包无法访问，这些密钥控制的资金将彻底消失。每个地址仅在交易中使用一次也与避免地址重用的原则直接冲突。地址重用允许他人通过将多个交易和地址互相关联，减少了隐私保护。一个Type-0非确定型钱包是一个无奈的选择，特别是你想避免地址重用而不得不使用大量密钥时，这使得频繁备份成为必要。虽然比特币核心客户端包含一个Type-0钱包，但比特币核心的开发者却不建议使用这个钱包。图4-8展示了一个非确定型钱包，包含一些随机密钥的松散组合。

*图4-8 Type-0 非确定性（随机）钱包：随机生成密钥的集合*

**4.4.2 确定性（种子）钱包**

确定性或者叫“种子”钱包是另一种钱包类型，它包含的私钥是通过使用单向哈希函数，从一个共同的种子衍生而来的。种子是一个随机生成的数字，它与其他诸如索引号或者“链码”（参看第87页《层次化的确定性钱包（BIP0032/BIP0044）》）等组合并计算得出私钥。在确定性钱包中，只要使用种子就可恢复所有衍生密钥，也就是说只要在创建钱包时做个简单备份就够了。种子也一样可以在钱包的导入导出时使用，利用种子可以非常简单的将用户的所有密钥从一个钱包软件迁移到另一个钱包。

**4.4.3 助记码词汇表**

助记码是一些英文单词序列，用于代表（编码）一个随机数字，这个随机数字就是用于创建确定性钱包的种子。通过这个单词序列足以重建种子并根据种子重建钱包及所有衍生而来的密钥。一个实现了确定性钱包（带助记码功能）的应用，在首次启动时将会向用户展示一个12到24个单词的序列。这个单词序列就是钱包的一个备份，可以用于在相同的应用或者兼容的应用中恢复并重建所有密钥。助记码词汇表使得用户钱包备份变得极为简单，毕竟相对一串随机数字，这些词汇更容易阅读和转录。

助记码在比特币改进提案39（[BIP0039]）中首次定义，目前还属于草案状态。需要注意的是，BIP0039仍是一个草案，而不是标准。特别的，还有一个不同的标准，使用了一套不同的词汇，在以太坊钱包中使用，并且其定义早于BIP0039。BIP0039已被Trezor钱包和其他一些钱包使用，但是与以太坊的实现不兼容。

BIP0039按如下步骤定义助记码和种子：

1. 创建一个128位到256位随机序列（熵）
2. 创建随机序列的校验码，即随机序列SHA256哈希值的前几位
3. 将校验码附加到随机序列之后
4. 将序列拆成11位长的小段，使用这些小段与一个预定义的包含2048个单词的词典做对应（译者：2^11=2048，故每段均可以对应到一个单词）
5. 生成12到24个单词作为助记码

**表4-5** 显示熵的大小与助记码长度的关系

*表4-5 助记码：熵与词汇数量*

| **熵（位数）** | **校验码（位数）** | **熵+校验码** | **词汇数量** |
| --- | --- | --- | --- |
| 128 | 4 | 132 | 12 |
| 160 | 5 | 165 | 15 |
| 192 | 6 | 198 | 18 |
| 224 | 7 | 231 | 21 |
| 256 | 8 | 264 | 24 |

助记码代表了128到256位的数字，使用密钥扩展函数PBKDF2，可以产生更长的（512位）的种子。种子用于创建确定性钱包以及所有衍生的密钥。

**表4-6**和**图4-7**显示了一些助记码及其他们生成的种子的例子

*表4-6 128位助记码和结果种子*

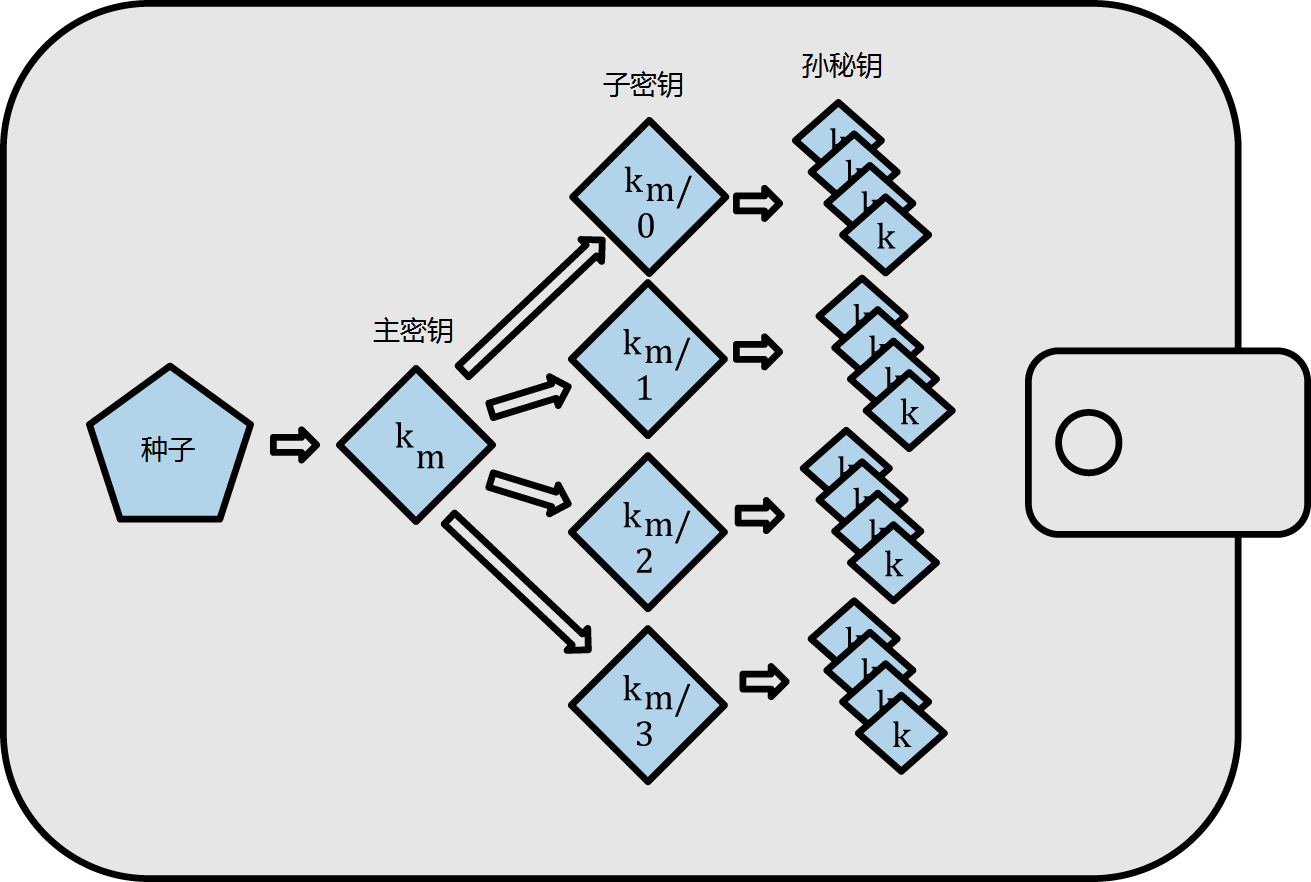
|  |  |
| --- | --- |
| **熵输入（128位）** | 0c1e24e5917779d297e14d45f14e1a1a |
| **助记码（12个单词）** | army van defense carry jealous true garbage claim echo media make crunch |
| **种子（512位）** | 3338a6d2ee71c7f28eb5b882159634cd46a898463e9d2d0980f8e80dfbba5b0fa0291e5fb888a599b44b93187be6ee3ab5fd3ead7dd646341b2cdb8d08d13bf7 |

*表4-7 256位助记码和结果种子*

|  |  |
| --- | --- |
| **熵输入（128位）** | 2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c |
| **助记码（12个单词）** | cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige |
| **种子（512位）** | 3972e432e99040f75ebe13a660110c3e29d131a2c808c7ee5f1631d0a977fcf473bee22fce540af281bf7cdeade0dd2c1c795bd02f1e4049e205a0158906c343 |

**4.4.4 层次化确定性钱包（BIP0032/BIP0044）**

确定性钱包是开发来用以实现从一个“种子”生成很多密钥。形式最先进的确定性钱包是*层次化确定性钱包*，或者叫HD钱包，在BIP0032中被定义为标准。层次化确定性钱包包含的密钥是一种树形结构，一个父密钥可以衍生出一系列的子密钥，每个子密钥又可以衍生出一系列孙密钥，以此类推，直到树的深度达到无穷大。树的结构如**图4-9**。



*图4-9 Type-2层次化确定性钱包：从种子生成一棵密钥树*

|  |  |
| --- | --- |
| notes | 如果你自己开发一个比特币钱包，它必须是一个HD钱包，遵循BIP0032和BIP0044标准 |

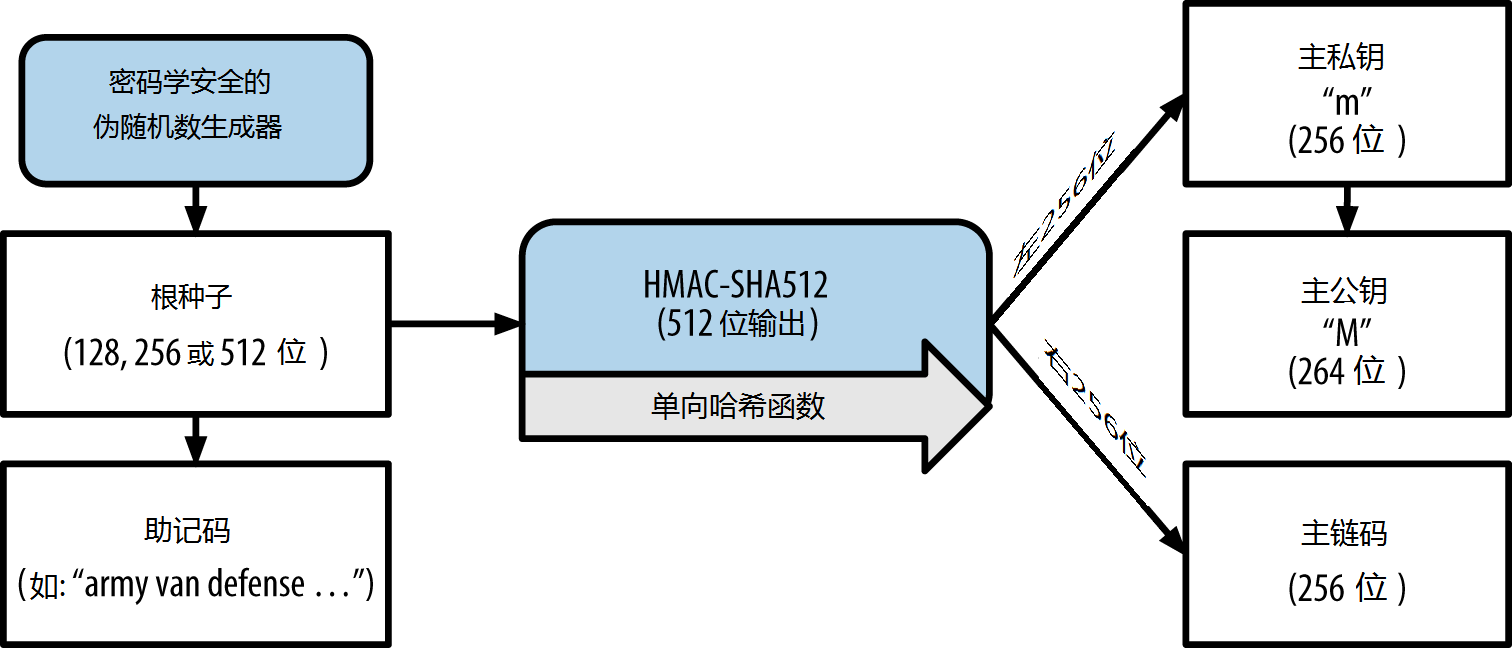
HD钱包相对随机（非确定性）密钥有两大优势。首先，树结构可以用于表达额外的组织含义，比如，一个特定分支的子密钥用于接收来款交易，而另一个分支的子密钥用于支付交易的找零。不同分支的密钥同样可以用于公司财务设置，分配不同的分支给不同的部门，子公司，用于特定用途或者会计账目。

HD钱包的第二个优势在于用户可以在不访问私钥的情况下创建一系列公钥。这使得HD钱包可以在不安全的服务器上使用，也可以针对每笔交易发行一个不同的公钥。公钥不需要提前预载或者衍生，服务器也不需保存用于花费资金的私钥。

**4.4.4.1 从种子生成HD钱包**

HD钱包从单一的根种子产生，它是一个128,256或者512位的随机数字。HD钱包中的其他一切东西均确定性的从这个根种子衍生而来，这使得从种子重建整个兼容HD钱包得以成为可能。同样的包含成千上万密钥的钱包也更加易于备份、恢复、导出和导入，所有要做的仅仅是传输一个根种子。根种子最常见表现形式为助记码单词序列，这使得根种子的转录和保存更易进行。在上一节《助记码词汇表》中我们已经详细描述过。

下图是HD钱包创建主密钥和主链码的过程：



*图4-10 从根种子创建主密钥和链码*

根种子是作为HMAC-SHA512算法的输入，生成的哈希值用于创建*主私钥*（m）和*主链码*。通过主私钥（m）相应生成一个主公钥（M），这过程使用我们之前见过的椭圆曲线乘法m\*G。链码用于引入熵，这在从父密钥创建子密钥的过程中需要用到，我们将在下节讲述。

**4.4.4.2 子私钥的衍生**

层次化确定性钱包使用一个*子密钥衍生*（CKD）函数从父密钥衍生出子密钥。

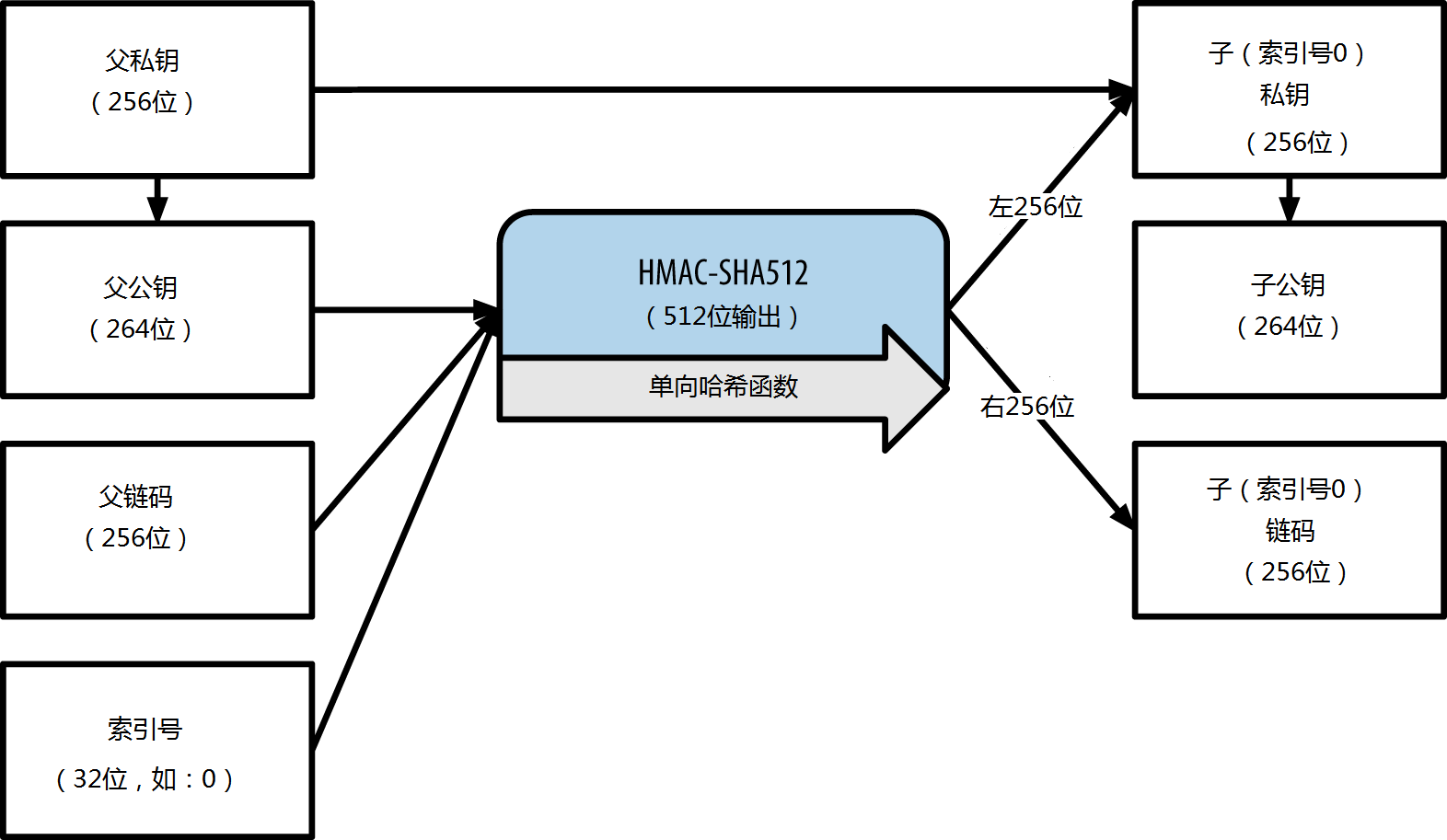
子密钥衍生函数基于单向哈希函数，它包括：

* 一个父私钥或公钥（ECDSA 非压缩密钥）
* 一个叫做链码的种子（256位）
* 一个索引号（32位）

链码用于引入看似随机的数据到这个过程中，使得单凭索引不足以衍生其他子密钥。因此，除非你也知道链码，只拥有一个子密钥是无法找到其同辈密钥的。初始链码种子（在树的根部）是由随机数生成的，但是子节点的链码是从父节点的链码派生而来。

这三个元素组合并哈希生成子密钥。如下所示。

父公钥，链码，索引号组合后，采用HMAC-SHA512算法进行哈希计算，生成512位的哈希。结果哈希分成两半。右边的256位哈希输出成为子节点的链码。左边的256位哈希与索引号被加入到父私钥从而形成子私钥。在图4-11中，我们可以看到从父密钥生成第0个子密钥（索引号为0）的过程。



*图4-11 扩展父私钥生成子私钥*

改变索引号，允许我们扩展父密钥，并顺序生成其他子密钥，比如子密钥0，子密钥1，子密钥2，等等。每个父密钥可以最多产生20亿个子密钥。

从树上下降一层，重复以上步骤，子密钥成了父密钥，可以创建它们自己的子密钥，以此类推，可以产生无穷多的后代。

**4.4.4.3 使用派生子密钥**

子私钥与非确定性（随机）密钥是不可区分的。因为衍生函数是一个单向函数，子密钥不能用于找到其父密钥。子密钥同样不能用于找到任何同辈密钥。如果你拥有第n个子密钥，你无法找到它的同辈，比如第n-1子密钥或者第n+1子密钥，或者其他任何序列中的子密钥。只有父密钥和链码可用于衍生所有子密钥。没有子链码，子密钥也无法用于衍生任何孙密钥。你需要同时拥有子私钥和子链码来开始一个新的分支并衍生孙密钥。

那么，子密钥本身可以拿来做什么呢？它可以用于产生一个公钥，一个比特币地址。然后你可以用于对发送到该地址上的未花费交易输出进行签名，从而解锁资金。

|  |  |
| --- | --- |
| notes | 子私钥，相应的公钥以及比特币地址，和随机生成的密钥、地址没有任何区别。事实上，它们构成的序列在生成它们的HD钱包外是不可见的。一旦创建，它们与“普通”密钥并没有任何区别。 |

**4.4.4.4 扩展密钥**

就像之前看到的，密钥衍生函数可以在树的任何层次基于三个要素：密钥、链码、索引号，创建新的子密钥。两个至关重要的要素是密钥和链码，它们的组合叫做*扩展密钥*。名词“扩展密钥”也可以理解为“可扩展的密钥”，因为这样一个密钥可用于衍生子密钥。

扩展密钥将一个256位密钥和256位链码连接成一个512位的序列进行存储和展现。有两种类型的扩展密钥：一个是扩展私钥，它是由私钥和链码的组合，用于产生子密钥（并由此产生子公钥）；另一个是扩展公钥，它是公钥和链码的组合，可用于创建子公钥，就如第68页《创建公钥》中描述的。

将扩展公钥想象为HD钱包树状结构中一个分支的根节点，从分支的根节点出发，可以派生出分支下的其他节点。扩展私钥可以创建完整的分支，而扩展公钥只能创建分支中的公钥。

|  |  |
| --- | --- |
| notes | 一个扩展密钥包括一个私钥或者公钥，链码。一个扩展密钥可以创建其子密钥，在树状结构中生成它自己的分支。分享一个扩展密钥可以授权对整个分支的访问权限。 |

扩展密钥采用Base58Check编码格式以便在不同的BIP0032兼容的钱包软件间进行导入和导出。扩展密钥的Base58Check编码采用一个特殊的版本号，其前缀为“xprv”和“xpub”，以便于识别。由于扩展密钥是512或513位，它比我们之前看到的其他Base58Check编码的字符串长很多。

下面是一个Base58Check编码的扩展私钥的例子：

xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6GoNMKUga5biW6Hx4tws2six3b9c

下面是其对应的扩展公钥，同样以Base58Check格式进行编码：

xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunSDMstweyLXhRgPxdp14sk9tJPW9

**4.4.4.5 子公钥派生**

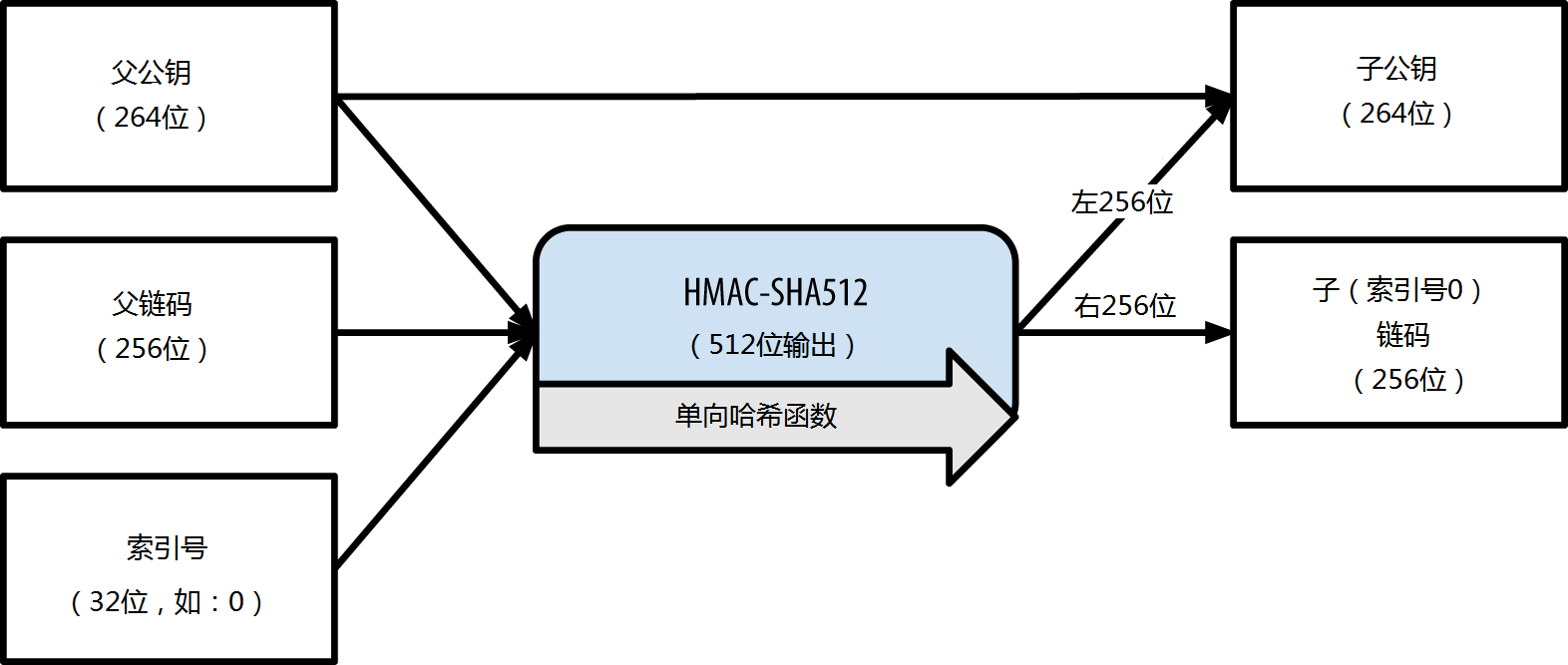
就如之前提到过的，层次化确定性钱包的一个很有用的特性就是可以从父公钥派生子公钥，而不需要其私钥。这使我们可以使用两种方式派生子公钥：或者从子私钥计算得出，或者直接从其父公钥派生。

如此，一个扩展公钥可以用于派生其HD钱包结构的分支下的所有子公钥（只有子公钥）。

当服务器或者应用程序只有一个扩展公钥的副本而没有任何私钥时，利用这种快捷的公钥派生方式，可以创建非常安全的仅限公钥的部署。这种部署方式可以创建无穷多的公钥和比特币地址，但是不能花费任何发送到这些地址上的资金。同时，在其他更安全的服务器上，扩展私钥可以派生出与扩展公钥相对应的私钥，用于签署交易并花费资金。

这种解决方案下的一个典型应用场景是在一台用于电子商务应用的web服务器上安装扩展公钥。web服务器可以利用公钥派生函数创建一个全新的比特币地址用于每个交易（比如，用于客户的购物车）。Web服务器没有任何私钥部署在其上，这样就避免了私钥丢失的风险。若没有HD钱包，唯一的解决途径就是在分开的安全服务器上创建成千上万个比特币地址，然后将它们预加载到电子商务服务器上。这种方式很麻烦，需要持续的维护以避免电子商务服务器用尽那些密钥。

另外一个场景是用于冷存储或硬件钱包。在这种情景下，扩展私钥可以存储在一个纸钱包或者硬件设备（比如Trezor硬件钱包）上，而扩展公钥可以在线保存。用户可以随时创建“接收”地址，而私钥却可以安全的存储在线下。为了花费资金，用户可以在离线签名比特币客户端上使用扩展私钥，或者使用硬件钱包设备（比如Trezor）来签署交易。图4-12显示扩展父公钥派生子公钥的机制。

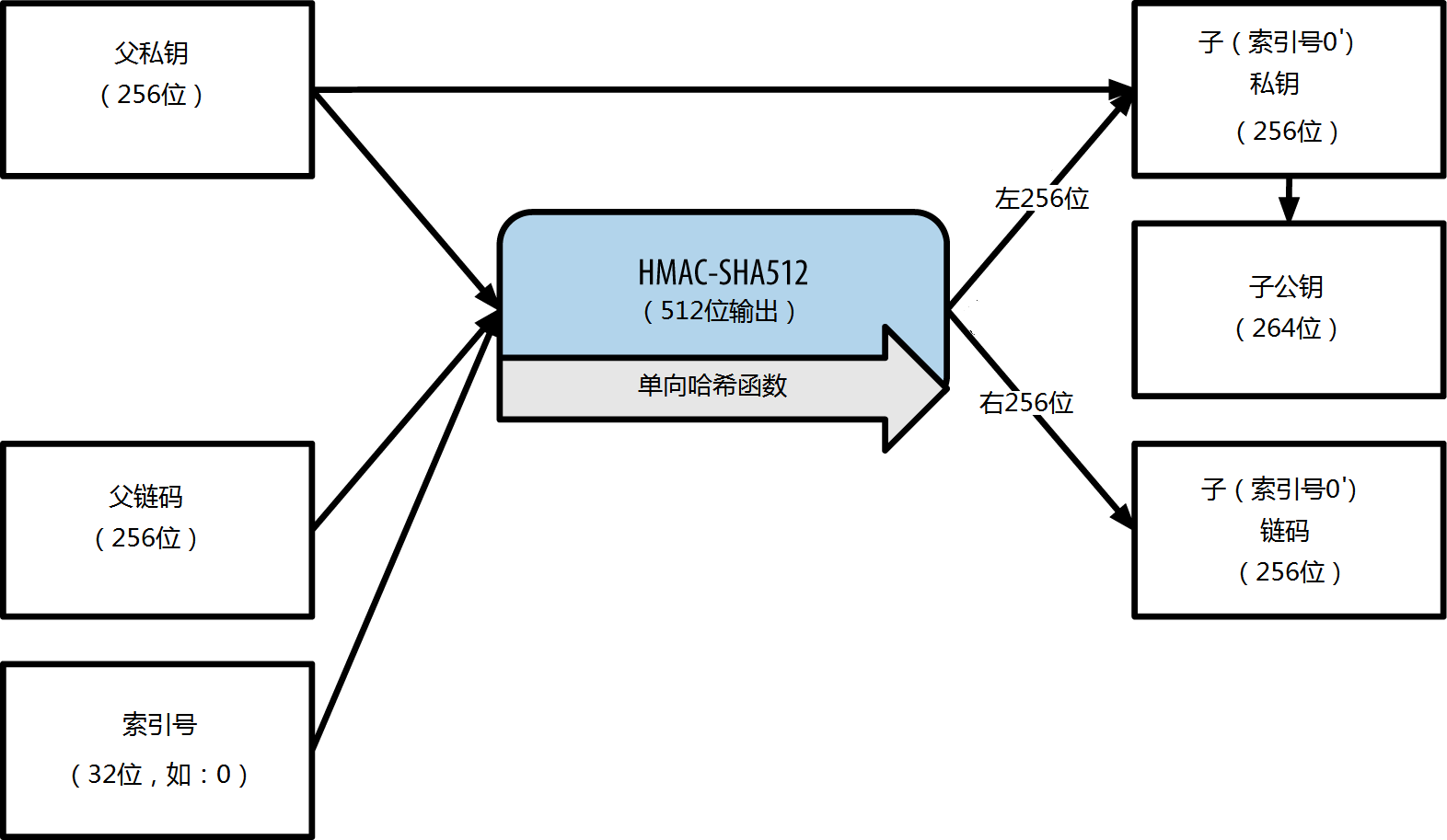


*图4-12 扩展一个父公钥以创建子公钥*

**4.4.4.6 强化的子密钥派生**

从扩展公钥派生出公钥分支的能力是非常有用的，但是它有个与身俱来的潜在威胁。访问一个扩展公钥不会给予访问子私钥的权限。但是，因为扩展公钥包含链码，如果子私钥已知，或者不小心泄露，那么就可以利用子私钥和链码派生出所有子私钥。一个泄露的子私钥，配合一个父链码，就将导致所有子节点的私钥的泄露。更糟糕的是，一个子私钥，连同父链码，可以推导出父私钥。

为防止这种威胁，HD钱包使用一个替代的派生函数叫做*强化派生(hardened derivation)*，它断开了父公钥与子链码间的关系。强化派生函数利用父私钥来派生子链码，而不是父公钥。这样就创建了一个父/子序列间的“防火墙”，链码无法用来推断父级私钥或同级私钥。派生函数看起来与普通子私钥派生一模一样，除了使用父私钥替代了父公钥。如图4-13：



*图4-13 子密钥的强化派生；省略了父公钥*

当使用强化的私钥派生函数时，产生的子私钥和链码是与普通的派生函数产生的结果完全不同的。其产生的密钥“分支”可以用于生成没有弱点的扩展公钥，因为它们包含的链码无法用于推导任何私钥。强化派生的引入，在扩展公钥所在层级与其上层级间形成了一个隔离。

简而言之，如果你想利用扩展公钥派生公钥分支的便利性，又不想使自己暴露在链码泄露的风险中，你需要从一个强化的父密钥，而不是普通父密钥来派生。作为最佳实践，主密钥的第一层子密钥总是通过一个强化派生函数派生而来，避免主密钥的推导。

**4.4.4.7 普通派生和强化派生的索引号**

派生函数中用到的索引号是一个32位的整数。为了易于区分从普通派生函数和强化派生函数派生的密钥，索引号被拆分成两个范围。介于0到2^{32}-1（0x0到0x7FFFFFFF）只用于普通派生，索引号介于2^{31}到2^{32}-1(0x80000000 to 0xFFFFFFFF)只用于强化派生。这样，如果索引号小于2^31，意味着这是个普通子密钥，而如果索引号大于等于2^31，子密钥就是强化的。

为了使索引号易于阅读和显示，强化子密钥的索引号从0开始显示，但是带一个上引号。第一个普通子密钥显示为0，但是第一个强化子密钥（索引号为0x80000000）显示为0'。相应的，第二个强化密钥的索引号为0x80000001，显示为1',以此类推。当你看到一个HD钱包的索引号i'，与2^31+i等价。

**4.4.4.8 HD钱包密钥标识符（路径）**

HD钱包中的密钥使用“路径”命名规则进行标识，树上的层与层之间采用一个“/”分隔符进行分隔。从主私钥派生而来的私钥开始于“m”。从主公钥派生而来的公钥开始于“M”。这样，主私钥的第一个子私钥表示为"m/0"。主公钥的第一个子公钥表示为“M/0"。第一个子私钥的第二个孙子私钥表示为“m/0/1”，以此类推。

一个密钥的“血缘关系”是从右往左读的，直到到达主密钥的位置。举例来说，标识符“m/x/y/z”描述一个密钥，它是密钥“m/x/y”的第z个子密钥，而“m/x/y”是“m/x”的第y个子密钥，“m/x”是m的第x个子密钥。

*表4-8 HD钱包的路径举例*

| **HD路径** | **密钥描述** |
| --- | --- |
| m/0 | 主私钥（m）的第一个（0）子私钥 |
| m/0/0 | 第一个子私钥的第一个孙私钥 |
| m/0'/0 | 第一个强化子私钥（m/0'）的第一个普通孙私钥 |
| m/1/0 | 第二个子私钥（m/1）的第一个孙私钥 |
| M/23/17/0/0 | 第24个子公钥的第18个孙公钥的第1个曾孙公钥的第1个曾曾孙公钥 |

**4.4.4.9 HD钱包树状结构导航**

HD钱包的树状结构提供了无比强大的弹性。每个父扩展密钥可以拥有40亿个子密钥：20亿普通子密钥和20亿强化子密钥。每个子密钥又拥有40亿个子密钥，以此类推。只要你愿意，这课树任意扩展到无限代。但是，正因为其弹性，要对这课无限树进行导航也变得极其困难。尤为困难的是在不同HD钱包间进行转移时，因为内部机构到分支及子分子的映射的可能性是无穷多的。

对这一复杂性，两个比特币改进建议（BIP）提供了一种解决方案，即对HD钱包的树状结构创建一些建议标准。BIP0043建议第一个强化子索引作为一个特别的标识符，用于表示树结构的“目的”。基于BIP0043，HD钱包只能使用一个层级-1的分支，通过定义索引号的目的，使用索引号表示树的剩余部分的结构和命名空间。举例来说，一个HD钱包只使用分支m/i'/是为了标识一个特定目的，而这个目的由索引号“i”进行指定。

BIP0044扩展了上述规范，提出一种多账号结构作为BIP0043下索引号44'的“目的”。所有遵循BIP0044的结构仅使用树的一个分支：m/44'/。

BIP0044规定包含五个预定义层级的树状结构：

m / purpose' / coin\_type' / account' / change / address\_index

第一层“目的”（purpose）总是设置为44'。第二层“币类别”(coin\_type)指定加密币类型，允许多币种HD钱包，每个币种在第二层下拥有自己的子树。目前已经定义了三种货币类型：比特币m/44'/0;比特币测试网络（Bitcoin Testnet）m/44'/1';莱特币（Litecoin）m/44'/2'。

第三层是“账户”（account），它允许用户将它们的钱包分成几个逻辑独立的子账户，用作会计或组织机构用途。比如，一个HD钱包可能包含两个比特币“账户”：m/44'/0'/0'，m/44'/0'/1'。每个账户都是它们子树的根。

在第四层，“找零”（changge），一个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。注意，不管上层是否使用强化派生，这层均使用普通派生。这是为了允许在这层树上可以导出扩展公钥用于不安全的环境中。可用的地址作为第四层的子密钥，是由HD钱包进行派生计算得来的，从而形成了第五层的“地址索引”（address\_index）。比如：主账户中的第三个比特币支付接收地址就是M/44'/0'/0'/0/2.**表4-9**列出了更多的例子：

*表4-9 BIP0044 HD钱包结构示例*

| **HD路径** | **密钥描述** |
| --- | --- |
| M/44'/0'/0'/0/2 | 主账户上的第3个收款公钥 |
| M/44'/0'/0'/1/14 | 第4个比特币账户上的第15个找零地址公钥 |
| m/44'/2'/0'/0/1 | 莱特币主账户的第二个私钥，用于交易签名 |

**4.4.4.10 使用sx工具集进行HD钱包试验**

使用第3章介绍的命令行工具sx，你可以生成和扩展BIP0032确定性密钥，并以不同格式展示它们：

|  |
| --- |
| $ sx hd-seed > m #从一个种子创建一个新的主私钥，并存储于文件“m”中  $ cat m #显示主扩展私钥  xprv9s21ZrQH143K38iQ9Y5p6qoB8C75TE71NfpyQPdfGvzghDt39DHPFpovvtWZaRgY5uPwV7RpEgHs7cvdgfiSjLjjbuGKGcjRyU7RGGSS8Xa  $ cat m | sx hd-pub 0 #生成M/0扩展公钥  xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunSDMstweyLXhRgPxdp14sk9tJPW9  $ cat m | sx hd-priv 0 #生成m/0扩展私钥  xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6GoNMKUga5biW6Hx4tws2six3b9c  $ cat m | sx hd-priv 0 | sx hd-to-wif #以WIF格式显示私钥m/0  L1pbvV86crAGoDzqmgY85xURkz3c435Z9nirMt52UbnGjYMzKBUN  $ cat m | sx hd-pub 0 | sx hd-to-address # show the bitcoin address of M/0  1CHCnCjgMNb6digimckNQ6TBVcTWBAmPHK  $ cat m | sx hd-priv 0 | sx hd-priv 12 --hard | sx hd-priv 4 #生成m/0/12'/4  xprv9yL8ndfdPVeDWJenF18oiHguRUj8jHmVrqqD97YQHeTcR3LCeh53q5PXPkLsy2kRaqgwoS6YZBLatRZRyUeAkRPe1kLR1P6Mn7jUrXFquUt |

**4.5高级密钥和地址**

在本节中，我们将观察密钥和地址的高级形式，比如加密私钥，脚本和多签名地址，炫耀地址以及纸钱包。

**4.5.1 加密私钥（BIP0038）**

私钥必须一直保持私密。私钥的保密性要求在实践中是很难实现的，因为它与同等重要的安全目标---可用性，形成了直接冲突。当你需要保存使用的备份以防丢失时，如何保证私钥的私密性更为困难。将私钥保存在钱包中，并通过密码加密会安全一些，但是这样钱包就需要进行备份。时不时的，用户需要将密钥从一个钱包转移到另一个钱包---比如为了升级或者更换钱包软件。私钥备份可能也需要保存在纸上（见第104页《纸钱包》），或者一个外置存储介质上，比如U盘。但是假如备份自己被盗或者丢失呢？这些冲突的安全目标，推动了一个私钥加密提议的诞生，它便携、方便，可以被不同的钱包和比特币客户端所理解，这个提议由比特币改进建议38号标准化，称之为BIP0038（见[BIP0038])。

BIP0038提出了使用密码加密私钥，并进行Base58Check编码的标准，该标准保证密钥可以在备份介质上安全存储、在不同钱包间安全转移、在任何可能导致密钥泄露的场合保存。这个加密标准使用**高级加密标准**（AES），AES标准是美国国家标准技术研究所（NIST）创立的，广泛应用于商业及军事领域的数据加密。

BIP0038加密方案以一个比特币私钥作为输入，私钥通常以钱包导入格式（WIF）编码，是一个使用Base58Check字符，并带前缀“5”的字符串。另外，BIP0038方案还需要一个长密码---由多个单词或者包含数字字母的复杂的字符串。BIP0038方案的结构是一个Base58Check编码的加密私钥，以前缀“6P”开头。如果你看到一个密钥以“6P”开头，意味着它是一个加密的密钥，需要密码才能将其转换（解密）回可在钱包中使用的WIF格式的私钥（前缀为“5”）。很多钱包软件现在已经支持编码BIO0038加密私钥，并会提示用户输入密码以解密和导入使用。第三方应用，比如难以置信的好用的基于浏览器的“**Bit Address**”（Wallet Details选项卡）就可用于解密BIP0038密钥。

BIP0038加密密钥最常用的案例是使用纸钱包，纸钱包用于在一张纸上备份私钥。一旦用户选择了足够强壮的密码，一个带BIP0038加密密钥的纸钱包将会极为安全，是比特币离线存储的极好选择（这也被称为“冷存储”）。

使用[bitaddress.org](http://bitaddress.org)测试表4-10中的加密密钥，看是如何通过输入密码得到解密密钥的。

*表4-10 BIP0038加密私钥示例*

|  |  |
| --- | --- |
| **私钥(WIF)** | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn |
| **密码** | MyTestPassphrase |
| **加密密钥（BIP0038）** | 6PRTHL6mWa48xSopbU1cKrVjpKbBZxcLRRCdctLJ3z5yxE87MobKoXdTsJ |

**4.5.2 支付给脚本的哈希值(P2SH)与多重签名地址**

就如我们了解的，传统比特币地址是以数字“1”开头的，从公钥派生而来，而公钥又是从私钥派生而来的。虽然任何人都可以往一个“1”开头的比特币地址发送比特币，但是只有能提供与公钥哈希相对应的私钥，并进行签名的人才能花费这笔比特币。

以“3”开头的比特币地址是pay-to-script hash(支付给脚本哈希值，P2SH)地址，有时也被错误的称作多重签名（multi-signature或multi-sig）地址。这种地址指定交易的受益人是一个脚本的哈希而不是公钥的所有人。这项功能于2012年1月以BIP0016建议的形式引进，由于它为比特币地址本身提供了一个拥有更多功能的机会，目前已被广泛采纳。不像发送资金到“1”开头比特币地址（也被称为“支付给公钥的哈希值”（P2PKH））的普通交易，发送资金到“3”开头地址，除了需要公钥哈希和作为所有权证明的私钥签名，还需要更多的东西。这些要求在地址创建时就在脚本中指定了，所有以这个地址作为目标的交易输入都会被这些要求所阻隔。一个发生到脚本哈希地址是从一个交易脚本中创建的，它定义了谁可以花费这个交易输出（更详细的说明，参见第132页《支付给脚本的哈希值(P2SH)》）。编码一个支付给脚本的哈希值地址需要使用与创建比特币地址相同的双重哈希函数，只是它作用于脚本上而不是公钥上：

script hash = RIPEMD160(SHA256(script))

结果的“脚本哈希”加上版本前缀“5”，并以Base58Check格式编码，得到一个3开头的已编码的地址。这是一个P2SH地址的范例：*32M8ednmuyZ2zVbes4puqe44NZumgG92sM*

|  |  |
| --- | --- |
| notes | P2Sh不必与多重签名标准交易相同。一个P2SH地址*常常*表示一个多重签名脚本，但它也可以表示其他类型的交易。 |

**4.5.3 多重签名地址和P2SH**

目前，P2SH实现的功能大多是多重签名地址脚本。就像名字暗示的，底层脚本要求超过一个签名来证明所有权并花费资金。比特币多重签名功能是设计来要求在N个密钥中，至少需要M个签名（称之为“阈值”）的，称之为M-N多重签名，这里M小于等于N。举例来说，咖啡店老板鲍勃（第一章介绍的）可以使用一个多重签名地址，要求进行1-2签名，其中一个密钥来自于鲍勃，另一个来自于他妻子，以确保他们中的人一个都可以签名花费被这个地址锁定的交易输出。这与传统银行提供的“联合账户”功能类似，联合账户允许夫妻中任何一人签名即可使用资金。或者高佩什，那个为鲍勃设计网站的设计师，可以为其业务创建一个2-3签名地址，确保至少两个业务合作伙伴对交易签名后才能花费此地址内的资金。

我们在第5章中将探索如何创建一个交易以花费从P2SH（以及多重签名）地址接收到的资金。

**4.5.4 虚荣地址**

虚荣地址是一种包含人工可读信息的有效比特币地址。比方说*1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33*就是一个有效地址，它包含了英文单词“Love”，并作为Base-58字符的前面4位。虚荣地址需要生产并测试数十亿的候选私钥，直到找到一个能生成指定模式的私钥。虽然有些生成虚荣地址的优化算法，但是归根结底，其过程都是随机选择一个私钥，生成公钥，并由此生成地址，最后检查地址是否满足虚荣需求，重复这个过程几十亿次直到找到匹配的。

一旦找到一个与期望模式匹配的虚荣地址，与之相应的私钥就可用于花费比特币，这与其他地址完全一样。虚荣地址与普通地址相比没有任何区别。它们基于相同的椭圆曲线密码学（ECC）和安全哈希算法（SHA）。想通过虚荣地址查找私钥其难度也与普通地址一样。

在第1章中，我们介绍过尤金妮亚，一个菲律宾儿童慈善机构的总监。假设尤金妮亚正举办一个比特币的募捐活动，她想使用公布虚荣地址作为募捐地址。尤金妮亚创建了一个以“1Kids”开头的比特币地址，用于募捐宣传。下面我们来看看这个虚荣地址是如何创建的，以及这个地址对于尤金妮亚的慈善机构又意味着什么。

**4.5.4.1 创建虚荣地址**

首先，我们必须认识到比特币地址只是一个简单的数字，以Base58字符编码表示。为了找到“1Kids”开头的地址，可以看做是从1Kids11111111111111111111111111111到1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz中随便找到一个。则总共约有58^29（约等于1.4\*10^{51}）个地址，都是以“1Kids”开头。表4-11显示了具有1Kids开头的地址范围：

*表4-11 以“1Kids”开头的虚荣地址范围*

|  |  |
| --- | --- |
| **开始** | 1Kids11111111111111111111111111111 |
| **截止** | 1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz |

我们把模式“1Kids”看作一个数字，看一下在比特币地址里找到这个模式的频率（参看表4-12）。一台不带特殊硬件的台式电脑，每秒大概可以进行100000次密钥搜索。

*表4-12 出现虚荣模式（“1KidsCharity”）的频率以及在台式电脑上平均消耗时间*

| **长度** | **模式** | **频率** | **平均搜索时间** |
| --- | --- | --- | --- |
| 1 | 1K | 1/58(58个密钥中1次) | <1毫秒 |
| 2 | 1Ki | 1/3,364 | 50毫秒 |
| 3 | 1Kid | 1/(195\*10^3) | <2秒 |
| 4 | 1Kids | 1/(11\*10^6) | 1分钟 |
| 5 | 1KidsC | 1/(656\*10^6) | 1小时 |
| 6 | 1KidsCh | 1/(38\*10^9) | 2天 |
| 7 | 1KidsCha | 1/(2.2\*10^{12}) | 3-4个月 |
| 8 | 1KidsChar | 1/(128\*10^12) | 13-18年 |
| 9 | 1KidsChari | 1/(7\*10^15) | 800年 |
| 10 | 1KidsCharit | 1/(400\*10^15) | 46,000年 |
| 11 | 1KidsCharity | 1/(23\*10^18) | 250万年 |

就像你看到的尤金妮亚不可能马上创建“1KidsCharity”这样的地址，即使她能同时访问几千台计算机。任何一个字符的增加，难度就会增加58倍。超过7个字符的模式一般只能通过特定的硬件找到，比如定制的带有多块图形处理单元（GPU）的电脑。这些电脑通常是比特币“矿机”再利用的产物，这些“矿机”用作挖矿已经无利可图，但是作为查找虚荣地址却是可用的。在GPU上查找虚荣地址要比通用CPU上查找快很多数量级。

查找虚荣地址的另外一种方式是外包给虚荣矿池，比如“Vanity Pool”矿池。矿池是那些拥有GPU硬件的人提供的一种服务，他们通过帮助别人查找虚荣地址还挣取比特币。通过支付一笔很小的资金（0.01比特币，在写本书时大概相当于5美元），尤金妮亚就可以通过外包搜索得到一个7位数的虚荣地址，这个过程仅需几个小时，而如果自己用CPU计算的话，需要花上几个月时间。

创建虚荣地址是一种暴力破解过程：尝试一个随机密钥，检查它的地址是否与目标模式匹配，不断重复直到成功。**例4-8**演示了一个“虚荣地址矿工”，一个用于查找虚荣地址的程序，基于C++。例子使用了我们在第56页《替代客户端、库和工具集》中介绍过的libbitcoin库。

*例4-8 虚荣地址矿工*

|  |
| --- |
| #include <bitcoin/bitcoin.hpp>  //待查找字符串  const std::string search = "1kid";  //创建一个32字节随机密钥.  bc::ec\_secret random\_secret(std::default\_random\_engine& engine);  //从EC密钥中解压出比特币地址.  std::string bitcoin\_address(const bc::ec\_secret& secret);  //与搜索字符串进行不区分大小写的比较  bool match\_found(const std::string& address);  int main()  {  std::random\_device random;  std::default\_random\_engine engine(random());  // Loop continuously...  while (true)  {  //创建一个随机密钥.  bc::ec\_secret secret = random\_secret(engine);  //获取地址.  std::string address = bitcoin\_address(secret);  //与我们的搜索字符串匹配吗? (1kid)  if (match\_found(address))  {  //成功!  std::cout << "Found vanity address! " << address << std::endl;  std::cout << "Secret: " << bc::encode\_hex(secret) << std::endl;  return 0;  }  }  // 永远无法到达这里!  return 0;  }  bc::ec\_secret random\_secret(std::default\_random\_engine& engine)  {  // 创建新密钥...  bc::ec\_secret secret;  //对每个设置随机值的字节进行迭代...  for (uint8\_t& byte: secret)  byte = engine() % std::numeric\_limits<uint8\_t>::max();  // 返回结果.  return secret;  }  std::string bitcoin\_address(const bc::ec\_secret& secret)  {  // 密钥转换为公钥...  bc::ec\_point pubkey = bc::secret\_to\_public\_key(secret);  // 最后创建地址.  bc::payment\_address payaddr;  bc::set\_public\_key(payaddr, pubkey);  // 返回编码后的地址.  return payaddr.encoded();  }  bool match\_found(const std::string& address)  {  auto addr\_it = address.begin();  //循环比较搜索字符串和小写的地址字符.  for (auto it = search.begin(); it != search.end(); ++it, ++addr\_it)  if (\*it != std::tolower(\*addr\_it))  return false;  //到达搜索字符串尾部，地址匹配与搜索模式匹配.  return true;  } |

示例代码必须使用C++编译器进行编译，并与libbitcoin库连接（库必须预先安装在系统上）。运行示例时，只要运行vanity-miner（假设编译时输出文件为vanity-miner）可执行程序，不需任何参数（参看例4-9），程序将尝试找到一个“1Kids”开头的地址。

*例4-9 编译并运行vanity-miner示例程序*

|  |
| --- |
| $ # 使用g++编译  $ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin)  $ # 运行例子  $ ./vanity-miner  Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT  Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f  $ # 重新运行以查找其他结果  $ ./vanity-miner  Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn  Secret: 7f65bbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623  $ # 使用"time"命令查看需要多长时间找到一个结果  $ time ./vanity-miner  Found vanity address! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM  Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349  real 0m8.868s  user 0m8.828s  sys 0m0.035s |

示例代码需要花费数秒钟找到一个匹配三个字符模式“Kid”的结果，就如我们刚刚通过Unix的*time*命令看到的执行时间。你可以在源码中修改搜索模式，看搜索四个甚至五个字符的模式需要多长时间。

**4.5.4.2 虚荣地址的安全**

虚荣地址是一把双刃剑，既可以用于增强也可以用于削弱安全措施。用于提高安全性方面，一个与众不同的地址使得入侵者难以使用他们自己的地址替换成你的地址，以欺骗你的客户向他们付款。不幸的是，虚荣地址使得任何人都可以创建一个类似于随机地址的地址，或者另一个虚荣地址，以此欺骗你的客户。

尤金妮亚可以向愿意捐款的人宣布一个随机生成地址（比如：1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy），或者她也可以生成一个以1Kids开头的易于识别的虚荣地址。

不管是哪种方式，均面临同样的威胁就，在使用固定地址（而不是每个捐献人一个独立的动态地址）的情况下，一旦有人侵入网站，并将这个地址替换为他自己的地址，捐献人的资金将被转入他的账户。如果你通过不同的渠道发布捐献地址，你的客户可以在确定支付前直观的检查这个地址，它与他们在你的网站、你的email或者你的传单上发布的是一样的。如果是一个随机地址，就像*1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy*这样，大部分客户可能会比对签名几个字符，比如*1J7mdg*，如果相同则认为地址是一样的。对于有意盗取资金的入侵者，可以使用一个虚荣地址生成器，生成一个前面几位相同的替代地址，看起来与这个地址很接近，就如表4-13所示的：

*表4-13 生成与随机地址类似的虚荣地址*

|  |  |
| --- | --- |
| 原始随机地址 | 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy |
| 虚荣地址（4个字符匹配） | 1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy |
| 虚荣地址（5个字符匹配） | 1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n |
| 虚荣地址（6个字符匹配） | 1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX |

那么虚荣地址能增强安全性吗？如果尤金妮亚生成一个虚荣地址*1Kids33q44erFfpeXrmDSz7zEqG2FesZEN*，客户很可能就会比对虚荣模式和其后的几个字符，比如“1Kids33”。这就使得攻击者必须至少生成6个字符（多两个字符），这将比尤金妮亚查找4位虚荣地址所花时间多3,364倍（58X58）。最重要的，尤金妮亚花费的努力（或者向虚荣地址矿池交付的资金）使得攻击者不得不生成更长的虚荣地址。如果尤金妮亚缴费让虚荣地址矿池生成一个8个字符的虚荣地址，攻击者将不得不创建10个字符长度的虚荣地址，这在普通电脑上已无法完成，而如果采用虚荣地址挖矿机或者虚荣地址矿池，也是极为昂贵的。尤金妮亚可承受的费用到了攻击者这边就变成难以承受的了，特别是如果潜在的回报无法覆盖生成虚荣地址需要花费的费用时。

**4.5.5 纸钱包**

纸钱包就是将比特币私钥银证在纸张上。通常，出于方便起见，纸钱包也包含相应的比特币地址，不过这不是必须的，因为地址可以从私钥派生而来。纸钱包是一个非常高效的创建备份或离线比特币存储的手段，也称之为“冷存储”。作为一个备份机制，纸钱包可以提供足够的安全手段防止因为计算机的灾难造成的私钥丢失，比如硬盘失效，被盗或者意外删除。作为一种“冷存储”机制，如果纸钱包的密钥是离线生成，且从未在计算机系统上存储过，那么它们在对抗黑客，键盘记录以及其他在线威胁方面更具安全性。

纸钱包可以有不同的形状、尺寸和外观设计，但是本质上，就是将密钥和地址打印在纸张上。表4-14显示了一个最简单形式的纸钱包。

*表4-14 最简单形式的纸钱包---打印出来的比特币地址和私钥*

| **公开地址** | **私钥（WIF格式）** |
| --- | --- |
| 1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn |

使用一些工具，比如bitaddress.org上客户侧的JavaScript生成器，可以简便的生成纸钱包。这个页面包含生成私钥和纸钱包必须的所有代码，即使与互联网完全断开，也不影响使用。为了使用这个工具，将HTML页面保持到本地硬盘或者外置U盘，然后断开与互联网的连接，在浏览器中打开刚才保存的文件。更安全的使用方法是，启动一个干净的操作系统，比如一个CDROM启动的干净Linux系统，来使用这个工具。这个工具在离线状态下生成的任何密钥均能通过USB线（不是无线）在本地打印机上打印出来。这样，创建的纸钱包的密钥只在本地纸张上存在，而不会在任何在线系统上存储。将纸钱包存在一个防火的保险箱内，发送比特币到它们对应的地址，一个降低但是高效的“冷存储”方案就实现了。图4-14显示一个通过bitaddress.org网站生成的纸钱包。



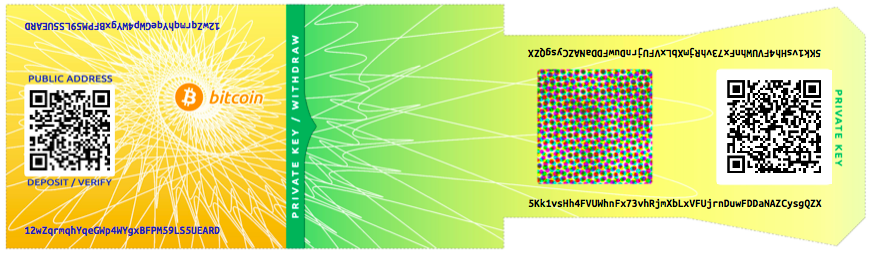
*图4-14 通过bitaddress.org生成的简单纸钱包的例子*

简单纸钱包系统的缺点是打印的密钥容易被贼盯上。一个能接触到纸钱包的贼既可以把那张纸偷走，也可以把私钥拍下来，从而控制被这些密钥控制的比特币。更复杂的纸钱包存储系统采用BIP0038加密私钥。打印在纸钱包上的密钥被主人脑袋记着的密码保护着。没有密码，加密密钥毫无用处。不管怎样，纸钱包仍然比密码保护的钱包更为安全，因为这些密钥从未在线，并且必须通过物理的方式从保险柜或者其他物理安全的存储设备上取得。图4-15显示一个在bitaddress.org上生成的使用加密密钥（BIP0038）的纸钱包。



*图4-15 通过bitaddress.org生成的加密钱包，密码是“test”*

|  |  |
| --- | --- |
| bug | 虽然你可以往一个纸钱包上多次存入资金，但是你必须一次性使用掉所有资金。这是因为在解锁和使用资金的过程中，你已经暴露了密钥，同样也因为当你花费的金额少于全部金额时。一些钱包可有不同的功能。有些设计来作为礼物送给别人，因此设计了季节主题，比如圣诞主题，元旦主题等；另外一些则存于银行金库或者保险柜中，私钥需要通过某些方式进行隐藏，比如覆盖不透明的贴纸，折叠并使用防篡改粘合箔密封等等。图4-16到4-18是一系列带安全保护和备份功能的不同的纸钱包。 以生成一个找零地址。解决这个问题的一种方案是，一次性取出纸钱包地址上的所有余额，将剩余部分发送到另外一个新的纸钱包中。 |

纸钱包具有不同的设计和尺寸，也拥 

*图4-16 bitcoinpaperwallet.com生成纸钱包例子，私钥打印在折叠翼片上*



*图4-17 bitcoinpaperwallet.com的纸钱包，私钥被密封隐藏*

另外一种设计，支持额外的私钥和地址的备份，形式上是一个附加的存根，类似票根，允许存储多个副本以达到防火防洪以及其他自然灾害的目的。



*图4-18 具有额外密钥副本的纸钱包，印制在备份“存根”上*