

加密币安全交易

实战

作者：王良明



目录

前言 8

1 加密币基础 1

1.1 加密币 1
1.2 私钥和公钥 4
1.3 数字签名 7
1.4 数字证书 15
1.5 哈希运算 19
1.6 数字钱包 22
1.5.1 传统钱包 22
1.5.2 HD 钱包 25
1.5.3 钱包地址 26
1.7 区块链 29
1.8 记账与账本 31
1.9 比特币简介 32

2 准备环境 36

2.1 准备浏览器 36
2.2 科学上网 42
2.2.1 梯子 42
2.2.2 Tor 网 47
2.3 安装 nginx 50
2.4 准备相关文件 51
2.4.1 方法 1：从网站下载 51
2.4.2 方法 2：自己编译 54
2.5 各种库介绍 60
2.5.1 bitcoinjs-lib 60
2.5.2 tiny-secp256k1 85
2.5.3 bip32 87
2.5.4 bip38 90
2.5.5 bip39 93
2.5.6 bip84 97

3 以太币 102

3.1 以太坊简介 102
3.1.1 以太币 102
3.1.2 智能合约 103
3.1.3 NFT 105

3.1.4 DeFi 105
3.1.5 以太坊地址 105
3.1.6 ENS 域名 107
3.1.7 交易费用 115
3.2 以太币开发 116
3.2.1 ethers.js 库 116
3.2.2 综合例子 124

4 理论进阶 127

4.1 椭圆曲线 127
4.2 签名算法 132
4.3 地址进阶 138
4.3.1 P2PKH 地址进阶 138
4.3.2 P2SH 地址进阶 139
4.3.3 P2WPKH 地址进阶 141
4.3.4 P2WSH 地址进阶 142
4.3.5 P2SH-P2WPKH 地址进阶 143
4.3.5 P2TR 地址进阶 144
4.4 HD 钱包进阶 148
4.5 助记词 151
4.5 比特币进阶 156
4.5.1 UTXO 156
4.5.2 解锁过程 158
4.5.3 交易剖析 168
4.5.4 脚本编程 179
4.5.5 P2TR 脚本路径 182

5 交易实操 196

5.1 钱包管理 196
5.1.1 步骤 1：从虚无到随机数 196
5.1.2 步骤 2：从随机数到助记词 197
5.1.3 步骤 3：从助记词到种子 198
5.1.4 步骤 4：从种子到根节点 198
5.1.5 步骤 5：从根节点到钱包节点 199
5.1.6 步骤 6：从钱包节点到钱包 200
5.1.7 综合：步骤 2~6 201
5.2 放币交易 203
5.2.1 填写交易中的输入 205
5.2.2 准备输入的输入脚本 208
5.2.3 填写交易中的输出 212
5.2.4 对输入签名 213
5.2.5 终结输入 214

5.2.6 提取事务 215
5.2.7 发布交易 215

6 初级操作环境 221

6.1 准备环境 221
6.2 比特币 224
6.2.1 钱包管理 224
6.2.2 放币交易 235
6.3 以太币 245
6.3.1 钱包管理 245
6.3.2 放币交易 247
6.4 其他功能 248
6.4.1 私钥保护 248
6.4.2 定制助记词 249
6.4.3 生成二维码 251
6.4.4 图片隐写 252
6.4.5 数字签名与验证 253
6.4.6 非对称加密与解密 255
6.4.7 对称加密与解密 257
6.4.8 UTF8 与 HEX 互换 257
6.5 购买加密币 258
6.5.1 主流 P2P 交易平台 258
6.5.2 去中心化 P2P 平台 258
6.6 安全增强（可选） 261

7 中级操作环境 262

7.1 准备虚拟机 262
7.2 准备有关文件 271
7.3 准备浏览器 271
7.3.1 Tor-browser 272
7.3.2 Firefox-esr 276
~~7.3.3 Google-Chrome~~ 276
~~7.4 安装 Bisq（不安装）~~ 277
~~7.5 安装 jami（不安装）~~ 281
7.6 给虚拟机瘦身 288
7.7 备份和恢复虚拟机 290
7.8 钱包与交易 291

8 高级操作环境 292

8.1 准备环境 292
8.1.1 安装 virtualbox 293
8.1.2 导入 Whonix 294

8.1.3 初始化 Tor 网关 298
8.1.4 准备在线工作站 309
8.1.5 配置离线签名机 319
8.1.6 备份和恢复虚拟机 322
8.2 钱包与交易 324

9 终极操作环境 326

9.1 准备工作 326
9.1.1 准备硬件 326
9.1.2 准备 QubesOS 安装 U 盘 327
9.1.3 qubes 规划 328
9.2 安装 Qubes OS 329
9.3 创建并配置各个 qubes 338
9.3.1 Debian-12-minimal 模板 338
9.3.2 准备 sys-proxy 340
9.3.3 配置 sys-whonix 342
9.3.4 配置 anon-whonix 342
9.3.5 准备 app-online 342
9.3.6 准备 app-vault 344
9.4 日常使用 346
9.4.1 拷贝和粘贴 346
9.4.2 使用 U 盘 347
9.4.3 数据持久化 348
9.4.4 qrexec 策略系统 350
9.4.5 常见命令 351
9.4.6 备份和恢复 352
9.5 钱包与交易 353

附录 A：手工生成原始比特币交易(P2PKH) 354

A.1 一进二出的交易 354
A.1.1 准备两个比特币测试地址 355
A.1.2 确定 32 个字节的前交易 ID：xxxxx……x 356
A.1.3 确定 4 个字节的输出索引：xxxxxxxx 356
A.1.4 确定第 1 个输出中的 8 个字节的发生额：xxxxxxxxxxxxxxxx 357
A.1.5 确定第 1 个输出中的输出脚本和长度：xx<输出脚本> 357
A.1.6 对交易签名 358
A.2 二进一出的交易 362

附录 B：手工生成原始比特币交易(SegWit) 372

B.1 准备四个地址 372
B.2 签名 375
B.2.1 对第一个输入签名（P2PKH） 375

B.2.2 对第二个输入签名（P2WPKH） 377
B.2.3 对第三个输入签名（P2SH-P2WPKH） 378
B.2.4 构造最终交易 380

附录 C：使用 MetaMask 钱包 382

附录 D：比特币测试地址 388

附录 E：莱特币 391

附录 F：索尔币 393

附录 G：狗狗币 395

前言

投资加密货币的基本原则就两个字：安全，这倒不是指投资盈亏的风险，而是指财富被丢、被盗或被抢的风险。“被丢”就是你忘记了私钥，那相当于存于此私钥的财富全被丢，谁也救不了你。“被盗”就是指你的私钥被他人获取，从而把存于此私钥的财富全部或部分转走，你几乎是不可能找到这个窃贼的。“被抢”就是强盗直接找到你本人，强迫你说出私钥，从而转走你的财富，而且报案还会泄露自己的身份，警察还不一定会受理。因为加密货币抢夺了政府发行货币的特权，世界上几乎很少存在公开支持加密货币的政府，所以投资加密货币几乎成了各国的灰色地带。

为了保证最大可能的安全性，我们需要从以下三方面入手：

1、不要相信他人开发的钱包，更不要把私钥托管给第三方。你的私钥只能是你自己亲手创建和保管，杜绝一切可能的泄密漏洞。

2、多创建几个私钥，其中一个私钥上储存你的大部分财富，其他私钥上储存少量的财富，万一你“被抢”，你只报出一个存有少量资金的私钥。

3、凡是在线买卖加密货币，或和他人沟通交易事宜，尽量走 Tor 网络。Tor 网络是目前唯一可以隐身的暗网，发明比特币的中本聪就是擦用 Tor 上网，至今发现不了他的任何蛛丝马迹。千万不要采用实名认证的社交账号和他人做加密货币交易。

假如安全等级分为 1 至 10 级，10 级最安全，那么可以这样说：采用公开的钱包和加密货币交易平台，这种操作环境的安全性为 3，初级操作环境的安全级别为 5，中级操作环境的安全性可达 7，高级操作环境的安全级别可达 8，终极操作环境的安全性可达到 9！

如果资金规模在十万以内，直接采用公开的钱包是可以接受的，资金规模几十万的，初级操作环境足矣，百万级的，中级操作环境足够，千万规模的，高级操作环境足以胜任，更大规模的资金，还是采用终极操作环境为妙。

本手册介绍如何纯手工创建钱包、如何购买加密币、如何在多个钱包之间转移加密币以及如何发布交易等，从而实现私钥完全自己控制，交易过程自己掌握，交易费用自己确定（远低于通过平台交易的费用）。本手册覆盖初级、中级、高级和终极操作环境，满足不通层次的需要。

如果只想直接使用别人开发的钱包，以太币建议采用 MetaMask 钱包，它不但能安装在手机上，还可作为电脑浏览器的一个插件。作为本手册的补充，请阅读附录 D。

如需要帮助，请联系我，发邮件（liangming.wang@gmail.com 或 183407563@qq.com）或通过微信和公众号联系。本书作者拥有书中文字、图片和代码全部版权，未经作者许可，他人不能商业发布。个人网站 https://www.veryopen.org，本人开发的开源离线钱包网站 https://www.walletoffline.com，代码托管在 github.com/veryopen/walletoffline 上。

特别声明：本文作者对你投资加密币的盈亏风险不承担任何责任。



我的微信：moodisk



我的公众号：农妇3拳

2025 年 7 月 9 日

1 加密币基础

计算机科学家、加密大师尼克·萨博在 1994 年写成了《智能合约》论文，是智能合约的开山之作。到了 1998 年，他又设计出了一种叫“比特黄金”的去中心化的数字货币机制，由于缺少广泛支持最后失败了，但是他却被公认为是比特币的最古老先驱。1976 年哈耶克完成《货币的非国家化》一书，从理论上奠定了数字货币的可行性。

1.1 加密币

一些伟人们喜欢把自己秃顶的油腻照片画在一张纸片上，并写上一串数字，然后大声嚷嚷“嗨，这是你们的工资，拿去购物!”人们愿意持有这张纸片的原因无非是：要么是伟人们手里有枪，你不敢不用，要么是他们的信用一贯很好，你愿意使用。这就是纸币的来历。纸币无一例外都在持续贬值，这是国家管理者之人性贪婪的直接后果。

除了金、银这种硬通货外，其他的货币其实都是虚拟货币，你兜里揣着一张百元人民币钞票与我脑袋里记着一串数字（加密币）其实没有本质区别，它们都是建立在发行机构的信用之上，一旦信用崩溃，一张纸和一串数字都一文不值。津巴布韦政府撒赖，信用崩溃，于是街上百万亿面额的津巴布韦元钞票满天飞，人们甚至都懒得弯腰去拾起满地撒落的纸币，这里的钱成了最不值钱的东西。示意图参见图 1-1。



图 1-1 钱成了最不值钱的东西

除了津巴布韦，索马利亚的先令、土库曼斯坦的马纳特、越南的盾、圣多美普林西比的多布拉、印度尼西亚的卢比、伊朗的里亚尔、巴拉圭瓜的拉尼等基本与废纸差不多。中国历史上的民国后期，金圆券与津巴布韦的辛巴威一个卵样。因此把加密货币称为虚拟货币其实很不贴切。

只要是人，就不能完全相信他，更何况隐藏在“人民政府”这块华丽遮丑布背后的“伟”人们！不断贬值和最终成为废纸，这是所有纸币的宿命，因为这深深地根植于幽暗的人性之中。

借助于电脑、加密技术和互联网，中本聪像一个伟大的接生婆，把比特币这种加密货币接生到了这个糟糕透顶的人世间来，正如当时无人意识到耶稣降生对人类产生的巨大影响那样，当今的人啦，依然没有意识到加密货币对人类再次产生的巨大影响！

加密货币不再建立在贪婪的人性之上，而是建立在冷冰冰的计算机技术之上。从此，强大得如美国政府，以及假如斯大林、希特勒等恶魔再生，他们都无法操控加密货币了，更何况芸芸众生乎？

别人给我的账户汇了钱，密码只掌握在我手里，因此只有我才能花掉账户里的钱，普通货币和加密货币的业务规则都是这样。但是经营普通货币的银行能重置我的密码，能冻结我的账户，能不经我同意转走我账户里的钱，甚至以国家安全为由把我的钱收归国有，最无赖的做法就是直接宣布货币作废，总之他们有一万种方法搞走我的钱。是的，只要他们登录后台数据库中，敲入几条指令，我一生靠啃泡菜苦下来的几个碎银就化为泡影了！

加密币就不是这样搞，它把账户数据存放在世界上千千万万台电脑上，这些电脑在哪个大洲上的哪个国家，电脑的主人是人是鬼，任何人都一概不知。只要你愿意，你也可以把你的那台烂电脑加入那千千万万的电脑行列当中去，挣点存储费，等哪天和女朋友吵架了，心里不爽，你就砸烂这台烂电脑出气，也不碍加密币什么卵事。由于这种分散存储，任何人就无法修改数据了，当然理论上你只要能在短时间内修改 51%的存储了加密币信息的电脑上的数据，你就能像传统银行那样搞走别人账户中的钱，但是千千万万的百分之五十一还是千千万万，你想在短时间内修改数据，可能吗？所以，现在真实的情况是，那些强盗们早已打消了这个歪心眼，都弃贱从良了——老老实实搬砖挖矿去了。没有诱惑，何来盗贼？圣人不死，大盗不止，这是同一个理。先透露一个秘密：加密币还额外使用了一种神秘的防篡改技术：哈希运算！

什么？密码被人猜出来了怎么办？你还以为是银行里设的 6 位数字密码吗？6 位数字的密码，用现在的电脑，写几行程序代码，一两分钟就可以猜到了。先给你透个底：加密币的密码是包含数字和字母 a、b、c、d、e、f 的字符串（字母不区分大小写），长度必须为 64 位，例如下面的密码：

```
41551bd051a6d5f2a1a3adc8c5b44e2f01809d5d50d530d62163a165524e1e31
```

像这样的密码，用现在最快的电脑去猜，至少也得花个千把年时间吧，尼玛，从中国的宋朝猜到现在，有可能被你猜中了，但就算你猜中了又怎么样呢？黄花菜早已凉了，弱弱地问一句：你的子孙还在吗？所以根本没有人动过猜测别人密码的念头，不信？你去随便告诉一个人说“地球上还有没被挖出来的金子！”（我敢肯定一定有），如果他听了你的话就随便找个山坡开挖，算你赢！

什么？又有问题？花钱时在网上输入密码，这不是容易被网络黑客盗取吗？哎呀，这不怪你啦！因为你才刚刚接触加密货币，你目前还不知加密货币为何物。我先把结果告诉你吧：你不用开户！所以不用设置密码！！花钱时也不用在网上输入密码！！！别瞪大眼睛看着我，只要你的好奇心上来了，我就好办了。下面我就用一个模拟场景来演示一下吧，该小明上场了。“小明，过来，轮到你上场了！”

打工十几年的小明手里有点闲钱，如果存在银行里，钱一定会持续贬值，划不来，于是小明想到了目前热门的比特币。他在纸上胡乱写出了一串长度为 64 的字符串（由字母 a-f 和数字组成），如下所示：

```
c9f9eb0c3b494f2899b052eb8533a2f9ea34956ce7eb8d6896f804d2ab20e699
```

小明接下来通过某种方法（后面会告诉你方法）从这一串字符转换出比特币的私钥和地址，如下所示：

```
私钥：L3zKswWoaw4frESMjY7PoEm1VtprvjqeeF5qrmRgsXVaarNQyRXp
地址：1PQ1E6Mpy7qivzjC8knvgQ8sQEQQFFDyWT
```

好了，转换出来的私钥就是小明要牢记的最终的比特币密码，一共 52 个字符，小明没有告诉任何人，当然能记住胡乱写出来的 64 个字符的字符串也行，因为小明可以随时转换出私钥和地址。转换出来的地址可以公开，小明告诉了李四，让他转 0.1 个比特币到这个地址，因为小明刚刚把购买比特币的六千美元通过银行转账给了李四。李四反馈说已经发布了转账交易，大概几分钟后到小明的账户。于是小明去网站 https://www.blockchain.com/btc/address/1PQ1E6Mpy7qivzjC8knvgQ8sQEQQFFDyWT 查询是否到账（其中红色下划线部分就是小明的比特币地址），参考图 1-2 所示。

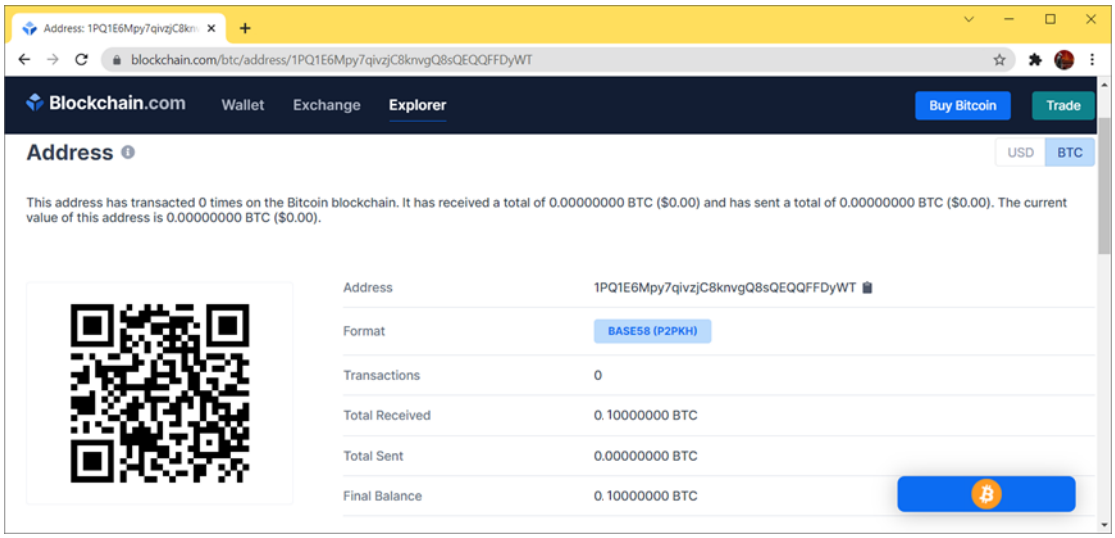


图 1-2 查询钱包

结果显示到账了。一年后小明想用比特币购买一部手机，比特币涨价了，一枚比特币值 7 万美元，因此小明的 0.1 比特币现在值 7000 美元，购买一部手机需要 1000 美元，换算为比特币就是 0.014 枚。手机店老板把他的比特币地址告诉小明，等比特币转账成功，小明就可以拿走手机了。小明怎么转账呢？他需要提供如下信息（称为交易数据）：

这次小明要把上次收到的那笔转账（0.1 个比特币）全部花掉，其中转给手机店老板 0.014 个比特币，愿意支付 0.00001 比特币（约 0.7 美元）的转账交易费用，剩下的 0.08599 比特币转回到小明的比特币地址（即 1PQ1E6Mpy7qivzjC8knvgQ8sQEQQFFDyWT）。

然后小明用密码（即上面的私钥）对以上的交易数据进行签名（签名时电脑不用连网），最后把签名后的数据发布出去。数分钟后手机店老板就能收到比特币了，小明也就可以拿走手机了。

小明花钱时，只有在发布已签名的交易数据时才要用到网络，其他步骤都可以在断网的情况下完成，即小明的私钥是不可能通过网络泄露的。

别人也不可能篡改已经签名的交易数据，因为一旦篡改了，其他存储数据的电脑（即矿工）就没法用小明的地址验证这笔交易，因此拒绝处理此交易，交易等于作废了！

到此，你脑瓜里应该存在这么几个疑惑吧：

1. 交易数据是如何保存在世界上千千万万台电脑上的？
2. 如何产生且安全地保管好 64 位密码？
3. 如何从 64 位密码转换出比特币或以太币的密钥和地址？
4. 如何对交易数据进行数字签名？
5. 签名后如何在网上发布交易？

我们先从公钥和私钥开始讲起，因为这是加密币最核心的技术，当然如果你不想了解加密货币的玄机，你只想投资加密币尽快“挣钱”，那你就跳过接下来的三个小节吧！不过，钻到钱眼里的人是很难挣到大钱的，只有眼里没有钱，你才能挣大钱！那些哭着喊着要为人民服务的人，在世界财富榜上，大部分都排在最前面，比尔·盖茨、马斯克、杰夫·贝索斯们在他们眼中算个毛，前者是万亿级，后者是百亿级至多千亿级，相差一、 两个数量级！

不管怎么说，你一定要有马云和马斯克的气概，马云说“我对钱没兴趣！”，马斯克说“Offer me money, offer me power, I

don’t care.”你瞧，不在乎钱，才能挣大钱！

1.2 私钥和公钥

告诉你，加密币使用的私钥就是由 0 和 1 组成的 256 位数——是的，就是 256 位二进制随机数（其他的应用场景可能会采用不同位数的私钥）。核心就是在 0~0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141 之间随机选择一个数，“随机”至关重要！类似如图 1-3 所示。

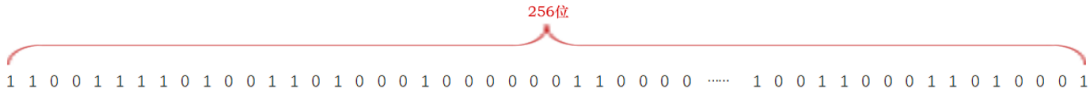


图 1-3 由二进制组成的私钥

256 位二进制数不好记，于是可以换算为 64 位的十六进制数（即 Hex 编码——由数字和字母 a,b,c,d,e,f 组成），类似下面的：

```
CF4D1030FECE1071B73246793D06D3D08EAD385481EE28CBF7B22859772798D1
```

尼玛，这还不好记啊，那进一步转换为 Base64 编码的 44 个字符吧，类似下面的：

```
z00QMP7OEHG3MkZ5PQbT0I6tOFSB7ijL97IoWXcnmNE=
```

最后一个字符一定是“=”。我靠，你玩我啊，这怎么记？！那算了，我只能帮你到这里了。我告诉你，256 位二进制、64 位的十六进制数和 44 个字符的 Base64 编码，你只要记住任何一个就行，因为它们可以互相转化。你记不住？那就写在纸上吧。

再次重申：加密币的私钥就是一个 256 位的二进制数。从最小的数（256 位全是 0）到最大的数（256 位全是 1），数的个数一共有 2^{256} 个（约等于 $1.15*10^{77}$，差不多等于宇宙中原子的个数），全 0 和全 1 不能作为私钥，还有一些保留的不能作为普通人的私钥，其他的都可以，只要不可预测且不可重复，具体怎么得到这个 256 位的随机数字并不重要。所以随机产生的一个私钥就相当于随机在宇宙中挑选一个原子，这样两个人随机产生的私钥，它们相同的概率完全可以忽略不计。因此一个好的私钥就体现在“随机”二字上——挑选时越随机越好。在我们实际的生活当中，很多看似随机的事件却是必然的，比如你抛一枚硬币，其实是可以根据硬币离开手时的状态（速度、自转等）、风速等因素算出最终的正反面。目前为止，人类发现的真正随机的事件是原子中的电子接下来出现的位置，这才是真随机，而其他的都称为伪随机。利用现有的技术手段在有限的时间内（比如 100 年）几乎不可能重复的事件，我们就可以用来产生需要的私钥。

这么神奇的私钥，竟然人人都可以随便做，而且不要钱，作为中国人，我感动得内牛满面，尼玛，在这里居然还有免费的东西！

手工产生私钥最简单的方法就是抛 256 次硬币（假如规定正面为 1，反面为 0），从而产生 256 位二进制数的私钥，见图 1-4 所示。

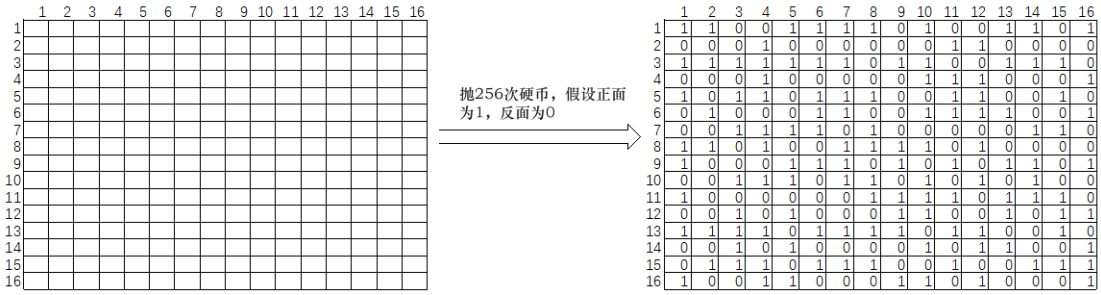


图 1-4 抛硬币产生私钥的过程

然后依次把各行中的二进制数拼接成一个 256 位的二进制数，就得到了一个很好的私钥，参见图 1-5 所示。

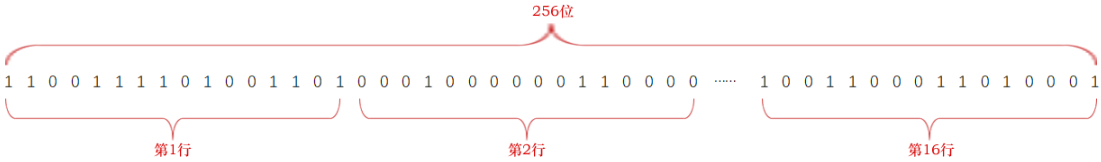


图 1-5 抛硬币产生的私钥

在本例子中，最后得到如下所示的 256 位的二进制私钥:

1100111101001101000100000011000011111110110011100001000001110001101101110011001001000110011110010011110100000110110100111101000010001110101011010011100000101010010000001111011100010100011001011111101111011001000101000010110010111011100100111100110000110100001。

不过比特币要求私钥小于 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141（即$2^{256}-2^{32}-2^9-2^8-2^7-2^6-2^4-1$）。256 位二进制数不好记，使用也不方便，一般要转换为其他编码形式，比如常见的 HEX（十六进制）或 Base64 等。对于 HEX，每 4 位二进制对应一位十六进制数，可参考表 1-1 作转换。

表 1-1 二进制与十进制对照表

二进制数	十进制数
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

因此上面抛硬币产生的随机数，通过纯手工方法转换，对应的 HEX 编码如下（一共 64 位）:

```
CF4D1030FECE1071B73246793D06D3D08EAD385481EE28CBF7B22859772798D1
```

参照上面的表格，换算的过程如图 1-6 所示。

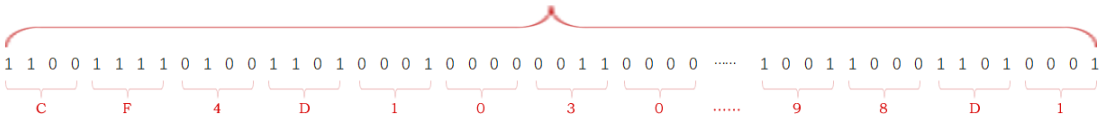


图 1-6 用十六进制表示的私钥

通过手工把二进制换算成十六进制数，很容易出错（不过出错关系也不大，反正你最终记住 HEX 编码的私钥就可以了）。

64 位的十六进制编码的私钥还是有点长，如果转换为 Base64 编码，长度会更短一些，Base64 编码机制是：以 24 位二进制数为一组，其中每 6 位二进制转换为 1 个字符，最后一组只有 16 位二进制（结果补 1 个“=”）。Base64 码表见表 1-2。

表 1-2 Base64 表

二进制	十进制	字符	二进制	十进制	字符	二进制	十进制	字符	二进制	十进制	字符
000000	0	A	010000	16	Q	100000	32	g	110000	48	w
000001	1	B	010001	17	R	100001	33	h	110001	49	x
000010	2	C	010010	18	S	100010	34	i	110010	50	y
000011	3	D	010011	19	T	100011	35	j	110011	51	z
000100	4	E	010100	20	U	100100	36	k	110100	52	0
000101	5	F	010101	21	V	100101	37	l	110101	53	1
000110	6	G	010110	22	W	100110	38	m	110110	54	2
000111	7	H	010111	23	X	100111	39	n	110111	55	3
001000	8	I	011000	24	Y	101000	40	o	111000	56	4
001001	9	J	011001	25	Z	101001	41	p	111001	57	5
001010	10	K	011010	26	a	101010	42	q	111010	58	6
001011	11	L	011011	27	b	101011	43	r	111011	59	7
001100	12	M	011100	28	c	101100	44	s	111100	60	8
001101	13	N	011101	29	d	101101	45	t	111101	61	9
001110	14	O	011110	30	e	101110	46	u	111110	62	+
001111	15	P	011111	31	f	101111	47	v	111111	63	/

参照此表，我们上面抛硬币产生的私钥编码成 base64 的过程如图 1-7 所示。

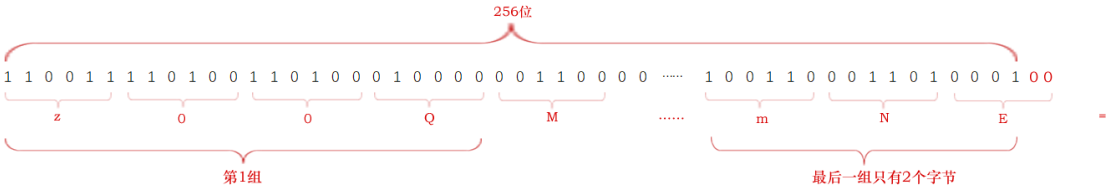


图 1-7 私钥编码成 base64 格式

最终的 Base64 编码为：

```
z00QMP7OEHG3MkZ5PQbT0I6tOFSB7ijL97IoWXcnmNE=
```

一共 44 个字符，相比于 HEX 编码少了 20 个。

从 256 位的私钥提取 512 位的公钥，存在多种方法，方法不同，提取出来的公钥也不同。提取公钥没法手工完成，需要用到程序代码，比如可使用 tiny-secp256k1 库的方法 xOnlyPointFromScalar(privateKey)或 pointFromScalar(privateKey))均可计算公钥，我在后续章节中会详细讲述。

一般很少人会采用纯手工方法制作私钥，大部分采用计算机指令完成，以后慢慢跟你讲，你先别急。

512 位二进制的公钥确实有点长，所以加密币早期的做法是进一步缩短公钥的长度，从而产生最终的钱包地址，如图 1-8 所示。不过后来发展到使用脚本的哈希充当收币地址，这一部分的内容后续章节会详细介绍。



图 1-8 私钥、公钥和地址的关系

1.3 数字签名

先设计一个问题：假如你在瑞士银行为远在美国的儿子开了一个账户，并存入一千万美元，你开户时设置支取款项的条件是：只要输入正确的用户名和密码，就可以通过网上银行转走资金。为了提高安全度，你特意设置了一个很长的密码。现在我问你：如何把用户名和密码告诉远在美国的儿子？

寄挂号信？打电话？发微信？发 qq？发邮件？托刚好要去美国的好友带口信？这些方法太危险啦！不过现在人傻钱多的人还真不少，都是祖上留下来的不义之财！

发 whatsapp？发 signal？在推特上发私信？这还不错，你还知道这些东西！不过一千万美元哦，可不是几百美元的事，丢了未免太可惜了。什么？你不在乎？我靠，你这钱来路不正啊，所以不珍惜，谁见过比尔·盖茨穿过一千美元的衣服？没有吧，因为他的钱全是一分一豪靠洒汗水苦下来的啊。

另外一种方法是伪装——把账户信息隐藏在诸如图片文件之中，然后把图片发过去，这样任何人都看到的是图片，儿子用 windows 的记事本打开图片就能看到银行账户信息。具体操作参考图 1-9，假定图中左侧的图片文件名为 earthrise.jpg，右上部的命令 echo buke... 就是把银行账号和密码追加到图片文件的末尾，右下角表示儿子收到图片文件 earthrise.jpg 后用记事本打开并翻到最后，即可看到需要的银行账户信息。

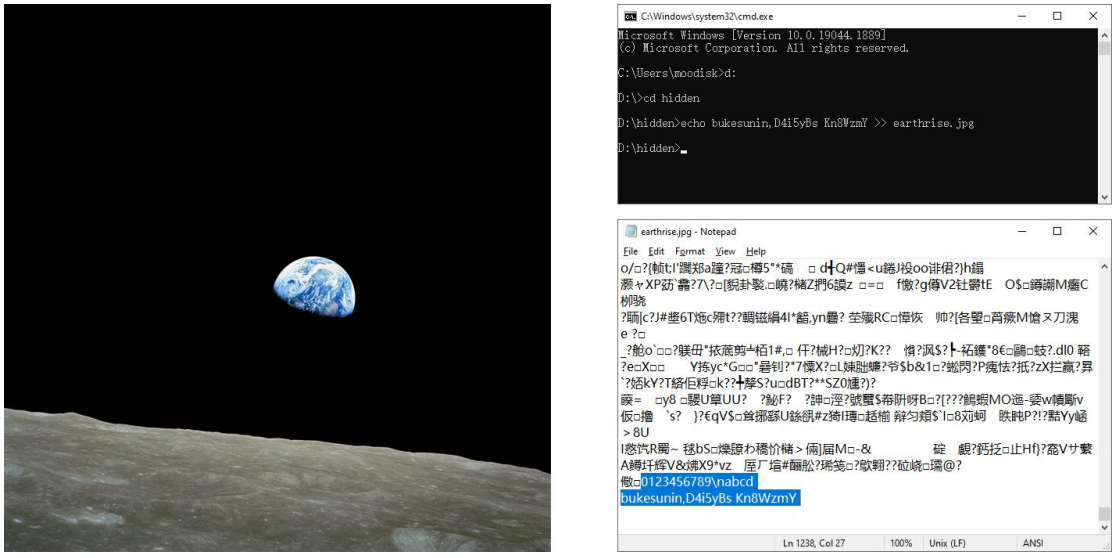


图 1-9 把信息隐藏在图片文件中

显然这有一个极大的风险，万一别人也采用诸如记事本这样的编辑器打开图片呢，岂不是泄露了密码。一个简单的补救措施就是账号和密码不要放在同一个图片之中，比如账号通过微信告知，而把密码放在图片中。这是最简单的图片隐写

术，高级的图片隐写术直接把信息拆散并保存到图片像素之中，采用同样的技术才能获取信息，比如 outGuess 就是一款流行的图片隐写工具，在 Linux 上安装 outguess，然后执行“outguess -k "Pass123" -d file.txt tupian.jpg secret.jpg”即可把 file.txt 文件中的内容隐写到 tupian.jpg 中，隐写后的图片为 secret.jpg，密码是 Pass123，执行命令“outguess -k "Pass123" -r secret.jpg 321.txt”从图片获取隐写内容到 321.txt 文件中。stego-js 是实现图片隐写的 js 库（可从 https://dimensiondev.github.io/Stego-JS/umd/dom.js 下载），另外 https://stego.js.org/实现在线隐写和隐读。

图片隐写术的原理很简单：用一个 32 位的整数保存图片的每一个像素颜色和透明度，即 RGBA 各占一个字节 (0~255)，把信息保存在字节的最低位，例如 R 值为 189，对应的二进制 10111101，把最低位改为 0 即可隐写‘0’，如此颜色分量 R 由 189 变成 188，人眼压根分辨不出图片的变化。一个像素最多可以隐写三个二进制，因为一般不修改 A 分量。所以一个分辨率为 x*y 的图片能隐写的数据量最多为$\left\lfloor\frac{(x\times y)\times 3}{8}\right\rfloor$ – 4字节，其中的“4”表示 4 各字节的整数，用于指定隐写的数据量。

其他的隐写技术还包括文本隐写术、音频隐写术、视频隐写术以及网络隐写术（协议隐写术），这里不再展开讨论。

把用户名和密码写在 word 文档中，然后加密保存，参见图 1-10 所示，最后把此 word 文档发邮件给儿子，并在微信里说：“儿子，打开文档的密码就是你姑姑的出生年月日。”这个方法不错，但是千万不要把文档和这句话一起通过微信发过去，老大的眼睛时时在盯着你的微信呢。

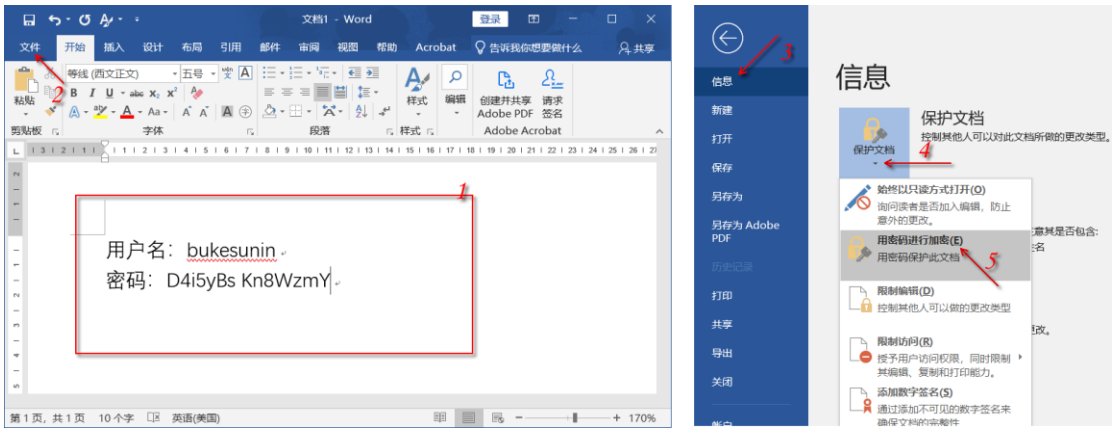


图 1-10 word 加密保存

如此，对这个 word 文档进行加/解密采用相同的密码——儿子的姑姑的出生年月日。这样的密码称为对称密码，也就是说加密和解密采用相同的密码。

每个人或一个家庭，建议仔细斟酌设计好几个密码，并且保证自己很难忘记而别人又不容易猜测，密码长度在 15 个字符以上，最好包含大小写字母、数字和其他符号，比如 1HblsqT& *就是一个好密码。对称密码在加密币行当中经常会使用，比如银行账户的密码，比如给虚拟机启动后设置的登录密码，再比如虚拟机磁盘的加密密码等。结合古诗词或《圣经》中的经文来设计对称密码是一个不错的注意，每个人都会有一些喜欢的诗词句子或经文，比如我就喜欢“飞流直下三千尺，日照香炉生紫烟。”从而设计出密码 Flzx3qC,RzxlszY.，根据原则：诗句的首尾字母大写，与数字同音的字就用数字代替。再比如《圣经》经文“申冤在我，我必报应。”可设计密码 SyzW&WbbY，不过里面缺少数字，呵呵。

除了对称密码，难道还有非对称密码？是的，那必须有——加密和解密的密码竟然不同！加密和解密的密码组成一对，其中一个密码称为私钥，另一个称为公钥，从私钥可以导出公钥，但从公钥很难算出私钥，因此最要紧的是要保管好私钥，而公钥是对外公开的。现在假设你儿子手上有一个私钥，他把导出来的公钥告诉你，你就可以用此公钥对上文提到的 word 文档加密，然后就可以放心大胆地把加密后的文档发给儿子，儿子用私钥解密文档即可看到明文了——瑞士银行的用户名和密码。

但是且慢！你儿子怎么就知道这个加密的文档就是你发的？因为儿子的公钥是公开的（很多人都有他的公钥），任何人都可以使用他的公钥加密文档然后发给他。也许你会说：没有这么傻的人吧，居然冒充父母给别人家的儿子发银行账户和密码？没错，还真没有这么傻的人，但我遇到过多次，当我走在街上时，突然有人假装无意丢一个钱包在我脚前，我才不去捡呢，因为我没这么傻！他们丢钱包只是诱饵，目的想钓出我兜里的血汗钱！天上掉下来的馅饼能砸在我这个矮个子的头上，不是诱饵又是什么？

那怎么办？简单啊，你签个名就是了，呵呵，但不是手写签名，是数字签名。即用你的私钥对信息（本例就是银行用户名和密码）签名，然后把信息和签名一起用儿子的公钥加密，最后发给儿子。儿子用他自己的私钥解密，再用你的公钥（儿子是知道你的公钥的）验证签名。整个过程可用示意图 1-11。

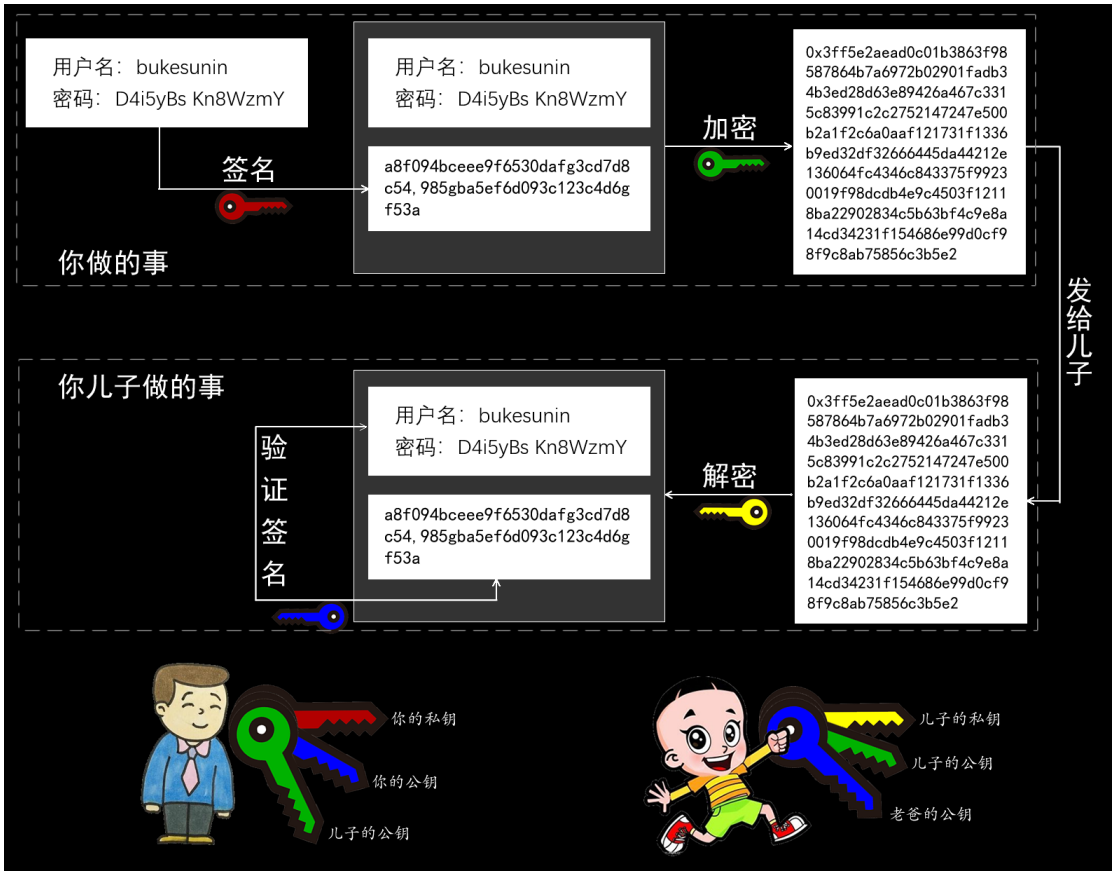


图 1-11 非对称加密与签名

前提是你拥有一对公/私钥，你儿子也拥有另外一对公/私钥，而且事先双方把自己的公钥告诉对方。注意图 1-11 中钥匙的颜色。

因为私钥只掌握在你手里，所以凡是用你私钥签名的任何文件都是你签发的，你不要撒赖啊。什么？你的私钥被别人盗用了？那也是你的问题，谁叫你保管不小心，一旦发现私钥泄露了，赶快做一个新的，同时把新的公钥公布于众。不过私钥还能设一个保护密码，这样就算别人盗取了你的私钥，但是如果不知道保护密码，他还是没法使用你的私钥的。私钥的保护密码是对称密码。

针对加密和解密有一些助记语：“公钥加密私钥解密”、“公加私解”、“公家死结”。

针对签名和验证也有一些助记语：“私钥签名公钥验证”、“私签公验”、“四千公燕”。

其实记住：公加私解。而签名/验证刚好反过来——私签公验。

不过在商业界，为了增加安全性和密钥对的安保，一般“加密/解密”和“签名/验证”采用不同的公/私钥对。加密币转账交易都需要交易发起方签名，而且还支持多签——类似企业的一份采购单，同时需要财务处经理和分管副总都签名才有效。

接下来我们通过一个实际的应用例子演示非对称密钥的使用：老爸把银行密码非常可靠安全地发给儿子。

双方从 https://files.gpg4win.org/gpg4win-latest.exe 下载加密/签名工具（gpg4win 是开源的，大家尽管放心使用，因为源代码都是公开的，所以作者不可能内置后门），并安装它，然后双击桌面上的“ 

”图标启动。先参考图 1-16 汉化它。

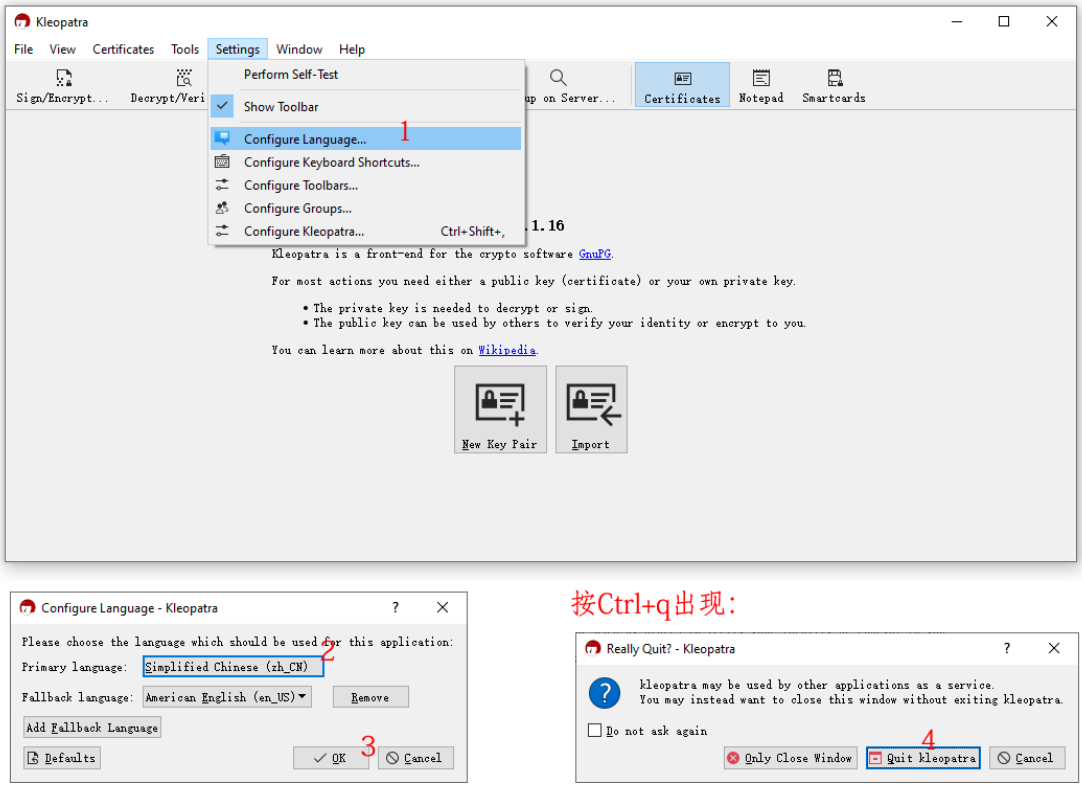


图 1-16 汉化 gpg4win

再次启动 gpg4win 就看到汉化了，然后点击“ 

”或者按 Ctrl+n 创建密钥对，出现如图 1-17 所示的画面。

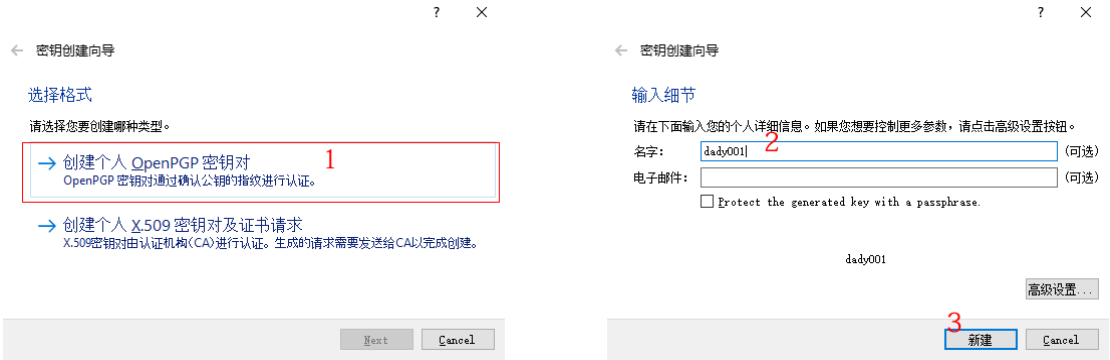


图 1-17 创建密钥对

爸爸创建密钥对时可以输入名字为 dady001 或者其他的都行，儿子那边创建时可以输入命令 son0001 什么的都可以。接

下来要把老爸的公钥导出来并发给儿子，参考图 1-18 操作。

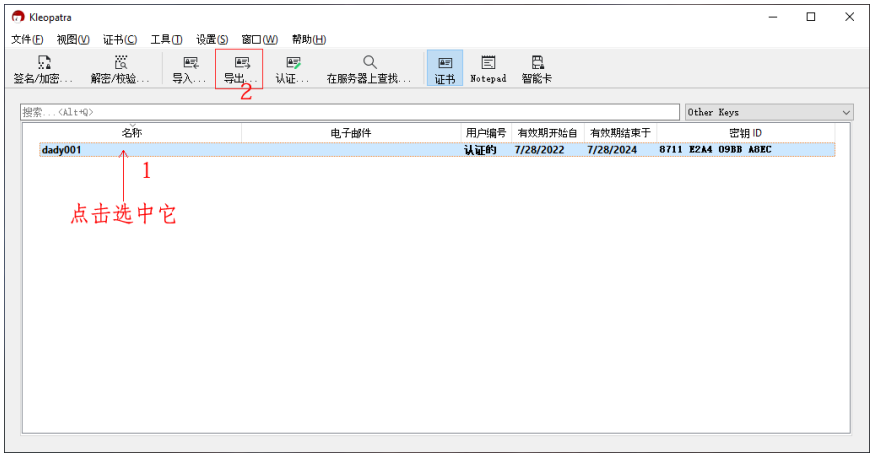


图 1-18 导出公钥

例如把公钥导出到桌面上的 dady001_0x09BBA8EC_public.asc 文件中，然后就可以把这个文件发给儿子，由于是公钥，不怕泄密，可以随便发。儿子也要把他的公钥导出并发给爸爸，比如发来的公钥文件名为 son0001_0xD552B2F8_public.asc，爸爸把儿子的公钥导入 gpg4win 中，参考图 1-19 所示操作。

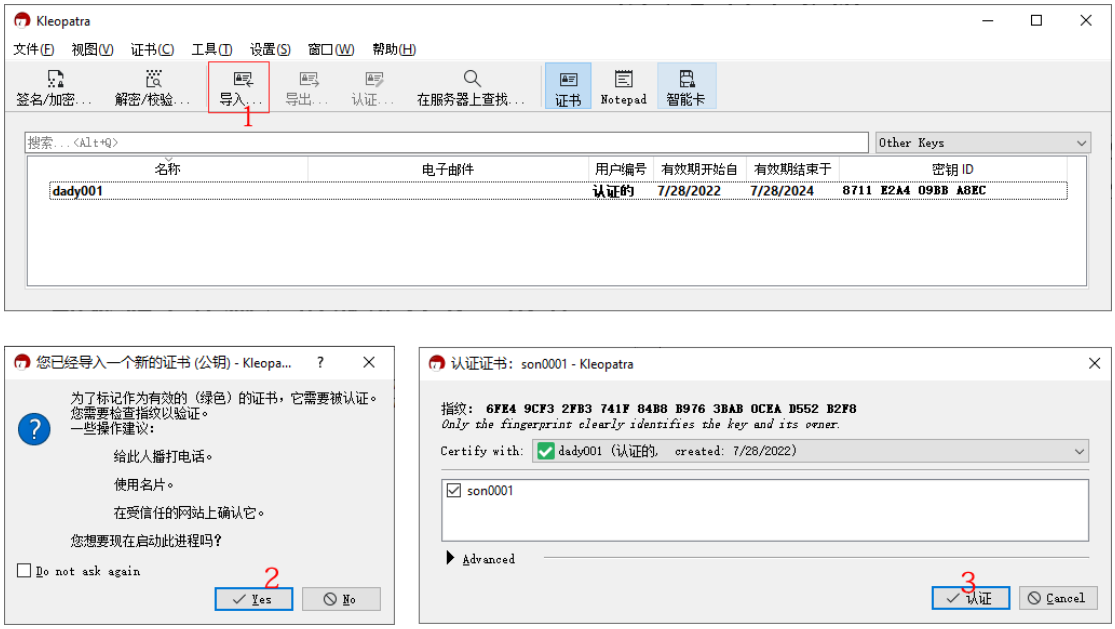


图 1-19 导入儿子的公钥

现在老爸有自己的密钥对和儿子的公钥，儿子也有他自己的密钥对和老爸的公钥，参考图 1-20 所示。

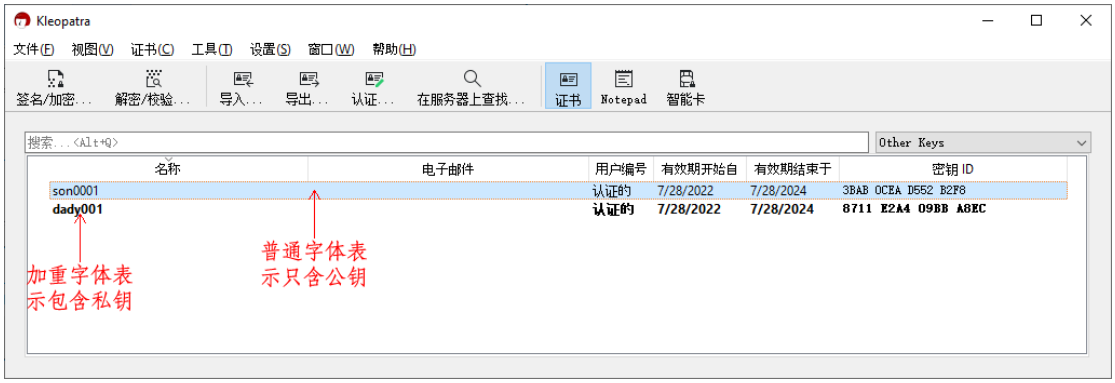


图 1-20 导入了儿子的公钥

现在开始对包含银行账号和密码的那个 word 文档签名和加密了。点击“签名/加密...”，然后选中那个包含银行账号信息的 word 文档（比如“银行账号.docx”），接下来出现图 1-21 所示的界面。

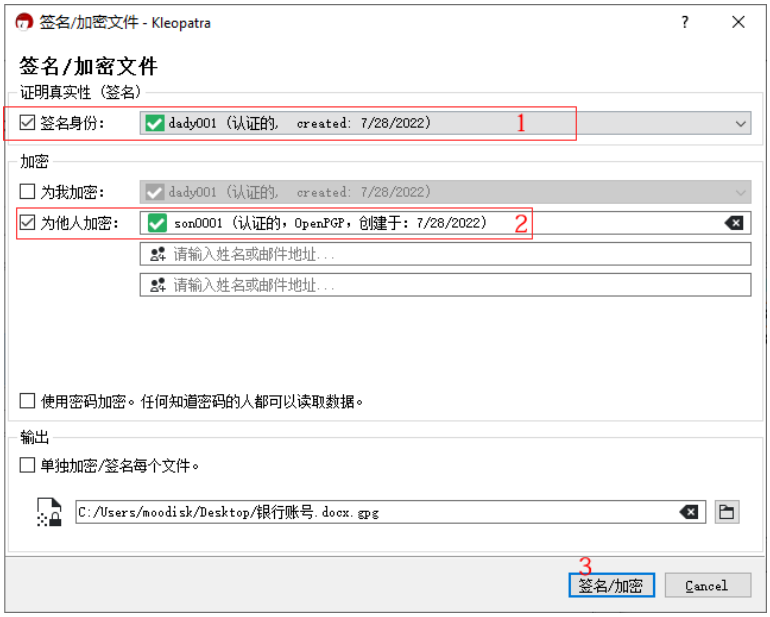


图 1-21 签名和加密

要记住：用老爸的私钥签名，用儿子的公钥加密！签名和加密后的文件就是“银行账号.docx.gpg”——在源文件名后添加了“.gpg”后缀。最后把文件“银行账号.docx.gpg”发给儿子即可（不管采用什么方式发都可以，因为就算泄露给其他人，他们也是没法解密的）。由于已经加了密，任何人都没法破解。儿子接到文件后点击“解密/校验...”进行解密并验证签名。儿子解密时采用自己的私钥解密，采用老爸的公钥验证签名。

现在遗留的问题是任何人只要能访问我的电脑，就能利用 gpg4win 程序冒充我的身份签名，或者干脆导出私钥并拷走。所以有必要给私钥加密——设一个保护密码，参考图 1-22 所示。

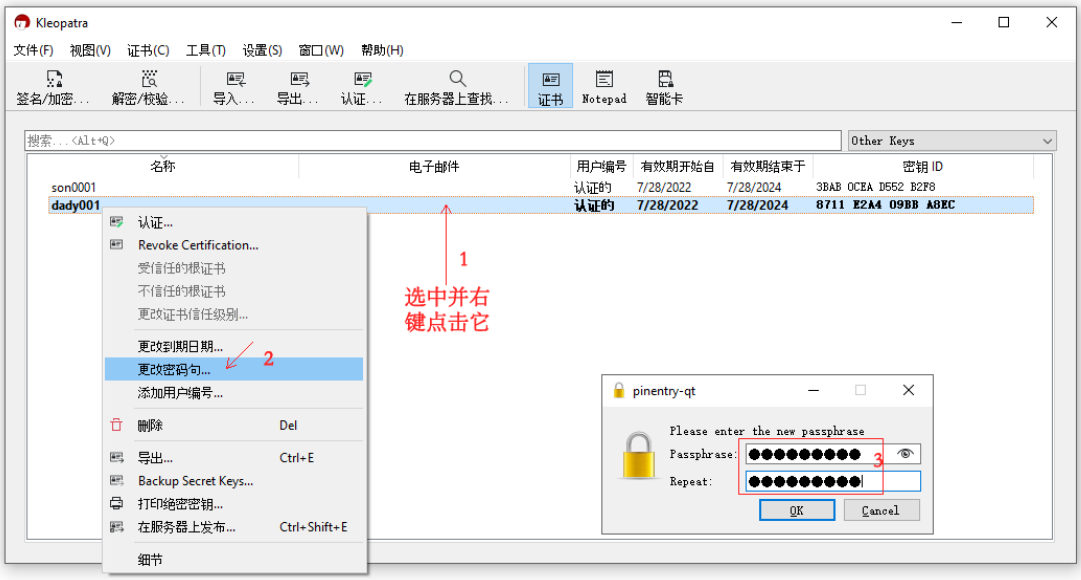


图 1-22 给私钥设置保护密码

这样以后导出私钥、用私钥签名时都要输入保护密码才行。导出并备份私钥可这样做：点击图 1-11 上的“Backup Secret Keys...”，然后取一个文件名（比如 dady001_0x09BBA8EC_SECRET.asc），最后输入保护密码即可，参见图 1-24。

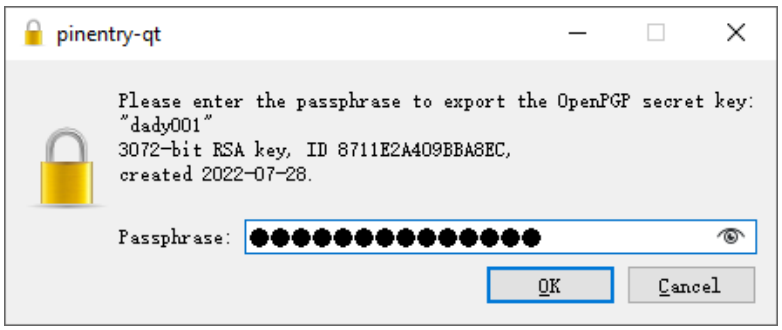
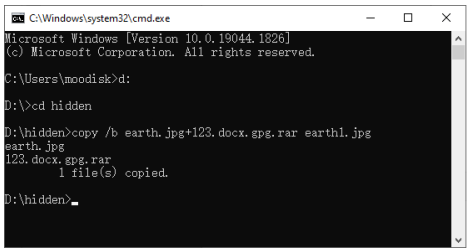


图 1-23 使用私钥时需输入保护密码

再把文件备份到 U 盘，也可以在其他电脑上导入此私钥。

把加密和签名后的文件公开传递给对方，还是有一定的风险：老大哥截获到这样的文件，他没法审查文件内容，于是有穿制服的人找上门来了，说“你发了什么东西？快说，不然跟我们走一趟!”所以我们还得进一步伪装——把加密文件隐藏到其他文件中，比如图片、视频之类的，然后把隐藏了加密文件的文件发给对方。操作很简单，先准备一个图片文件，如 earth.jpg，不妨假设包含了银行账户信息的加密文件为 123.docx.gpg，再安装 winrar 压缩工具（可从 https://www.win-rar.com 下载安装包），把 123.docx.gpg 压缩成 123.docx.gpg.rar，把 earth.jpg 和 123.docx.gpg.rar 放在同一个目录中（比如 d:\hidden），然后参照图 1-24 左侧部分的命令操作。



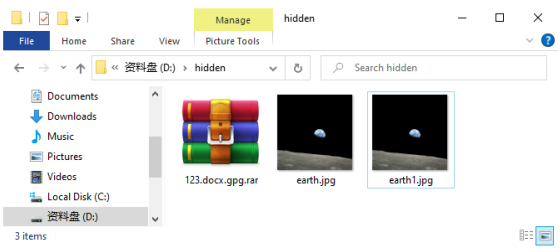


图 1-24 把信息隐藏到图片中

这样产生的新文件 earth1.jpg 还是图片文件，可以正常显示图片，但是里面已经隐藏了额外的信息，这样老大哥截获文件后看到只是一副普通图片而已，所以就 pass 了。对方收到 earth1.jpg 文件后采用 winrar 解压出 123.docx.gpg 文件，然后再去解密和验证签名即可。解压过程是这样：启动 winrar，然后点击菜单“File”、“Open archive”，参考图 1-25 操作。

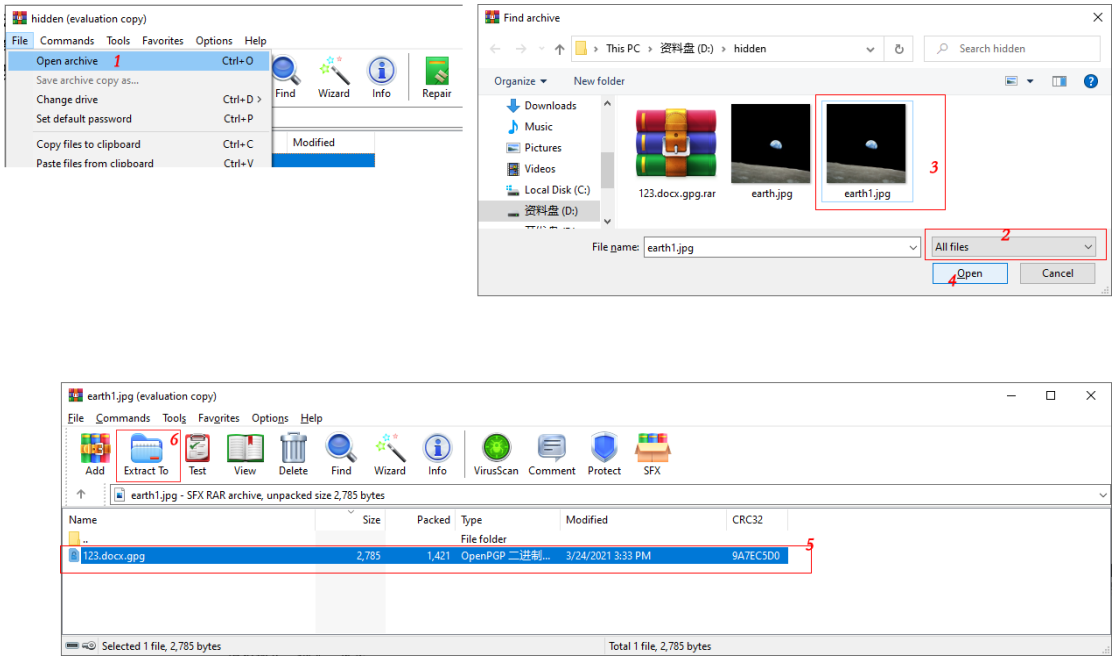


图 1-25 解压隐藏的文件

图片隐写术是另一种在图片中隐藏信息的方法，具体操作方法请参考§6.4.4 小节。

下面我们在 Linux 做签名的实验：

1. 产生密钥对。这个命令建议在电脑主控台上执行，前三个问题直接会取默认值：ECC（签名和加密）、Curve 25519、0 = 密钥永不过期，然后输入用户（如 Abraham II）和邮件地址（如 abrahamii@walletoffline.com），确认参数无误后输入私钥保护密码：

```
gpg --full-generate-key
.....
pub   ed25519 2025-06-14 [SC]
      0871A5938FE377B7D3CBA8E2AE8D057131094963 ____________________________________________________________#指纹
uid                 Abraham II <abrahamii@walletoffline.com>
sub   cv25519 2025-06-14 [E]
```

密钥指纹和公钥对外公布，指纹一般公布在公开网站上，并且以四个字符为一组以便于他人核对，如 0871 A593 8FE3 77B7 D3CB A8E2 AE8D 0571 3109 4963。

2. 管理密钥：

```
gpg -k                                                                                    #列出密钥对
.....
--------------------------------
pub   ed25519 2025-06-14 [SC]
      0871A5938FE377B7D3CBA8E2AE8D057131094963
uid           [ultimate] Abraham II <abrahamii@walletoffline.com>
sub   cv25519 2025-06-14 [E]
```

```
#导出公钥，gpg -a --export <邮件或者指纹> > public_key.asc。如：
gpg -a --export 0871A5938FE377B7D3CBA8E2AE8D057131094963 > public_key.asc

gpg --export-secret-keys --armor <邮件或者指纹> > privatekey.asc                    #导出私钥
gpg --import privatekey.asc                                                        #在其他电脑上导入密钥
gpg --delete-secret-and-public-key <邮件或者指纹>                                    #删除密钥对
gpg --import publicKey.asc                                                         #导入公钥
```

3. 签名（假如对 file.txt 文件签名）：

```
gpg --detach-sign -a -o file.txt.sig file.txt                                      #输入密码保护密码（如果有）
gpg --verify file.txt.sig file.txt                                                 #自己验证签名
```

4. 其他人验证签名：

```
gpg --import publicKey.asc
gpg --verify file.txt.sig file.txt
gpg: Signature made Sat Jun 14 15:50:25 2025 CST
gpg:                using EDDSA key 0871A5938FE377B7D3CBA8E2AE8D057131094963
gpg: Good signature from "Abraham II <abrahamii@walletoffline.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 0871 A593 8FE3 77B7 D3CB  A8E2 AE8D 0571 3109 4963
```

图 1-26 就是作者开发的离线钱包的下载页面。

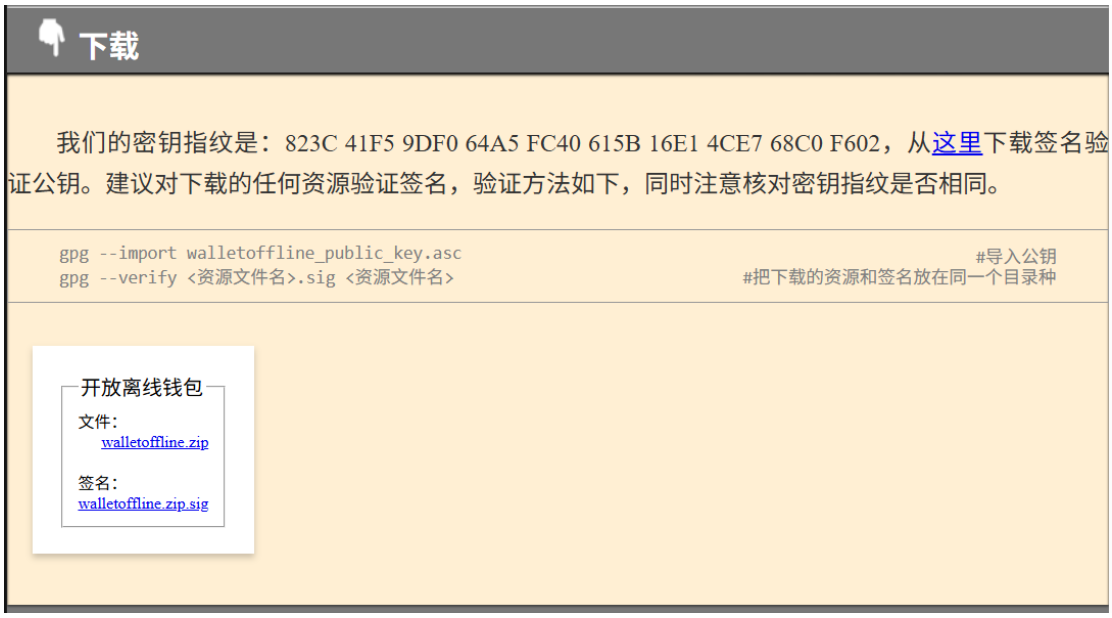


图 1-26 把公钥和密钥指纹公布在下载页面上

1.4 数字证书

前面讲过，公/私钥可用于信息的加密和解密、签名和验证签名，只要你把公钥发给对方，且对方愿意相信你并接受你的公钥。现在的问题是对方为何相信你并接受你的公钥？他是我的儿子啊，或者是我老爹啊，或者是我最好的朋友啊，是我的兄弟姐妹啊，……。但如果对方压根不认识你呢？ta 看到你发来的公钥就会惊讶地问“你谁呀？……操!”，得了，他根本不接受你的公钥。也许你会问，我和 ta 没有任何关系，没必要把公钥发给 ta 啊。你还别急，还真存在这样的情况：你要向陌生人证明自己的身份，比如你的网站，你是多么希望全世界的所有人都来访问你的网站对吧？但是对方凭

什么相信这就是你的网站而不是钓鱼网站？现在的网站地址都是以“https://”开头的，比如百度就是 https://www.baidu.com，京东 https://www.jd.com，中国银行 https://www.boc.cn，华为 https://www.huawei.com，……，就连作者自己的网站也是 https://www.veryopen.org。参见图 1-26。

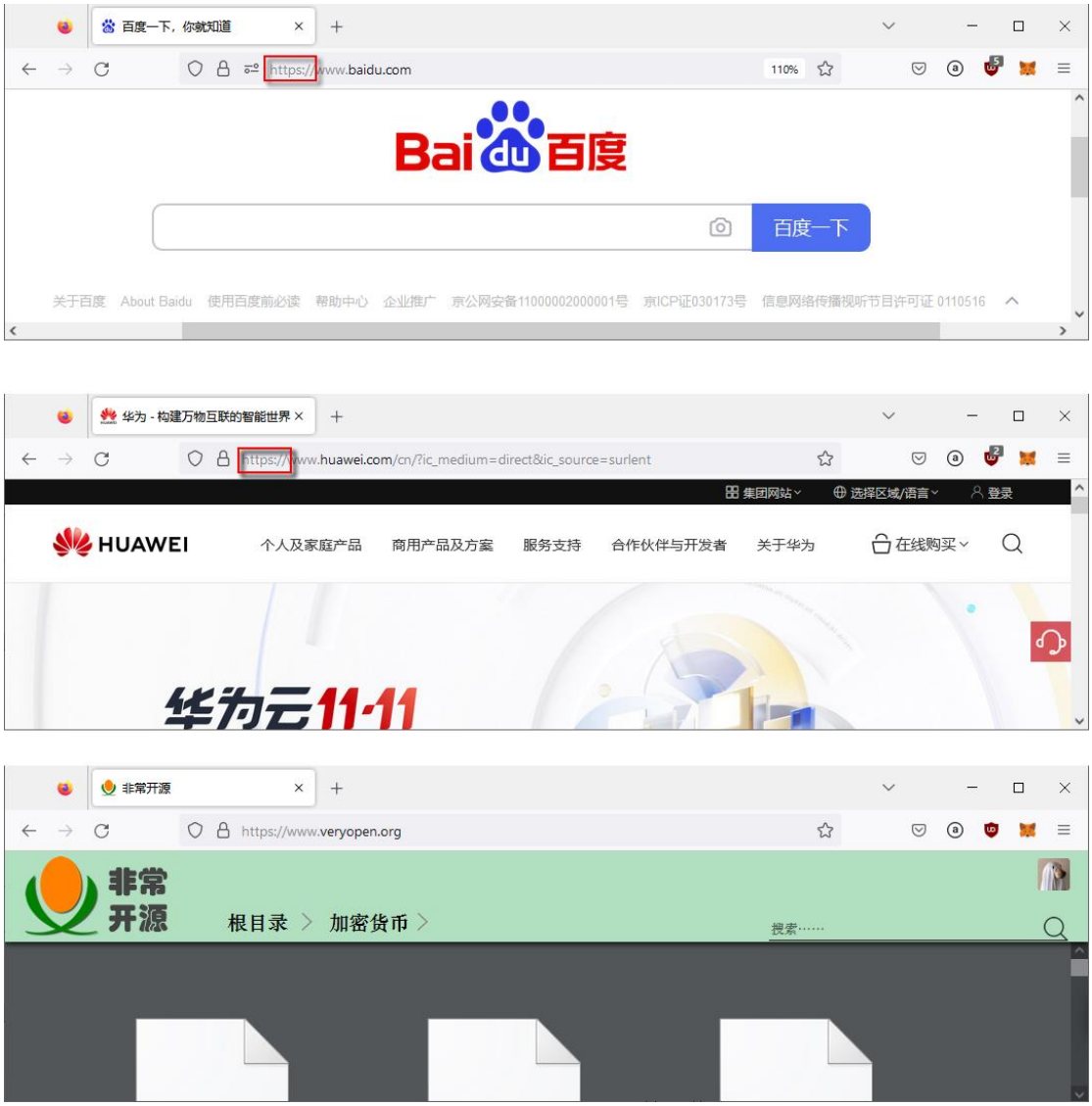


图 1-26 https 协议的网站

凡是以“https://”开头的网站都是向对方提供了公钥的，且对方通过权威的第三方认证机构，证明了此公钥确实是你的。这里的“对方”就是网页浏览器，比如 firefox 浏览器、chrome 浏览器、edge 浏览器等。例如你用 chrome 浏览器打开中国银行的网站，你的浏览器就接受了中国银行的网站服务器发来的公钥并通过第三方权威机构验证此公钥确实是中国银行的，所以你才会放心地输入银行账号和密码登录到网银。只不过接受公钥和认证都是由浏览器自动完成的。

其实网站服务器发给浏览器的除了公钥，还有网站主人的身份信息，如名称、网址、有效期、身份证或机构代码等，这些统称为证书内容，再加上第三方权威认证机构的签名，一起统称为数字证书，示意图参见图 1-27。

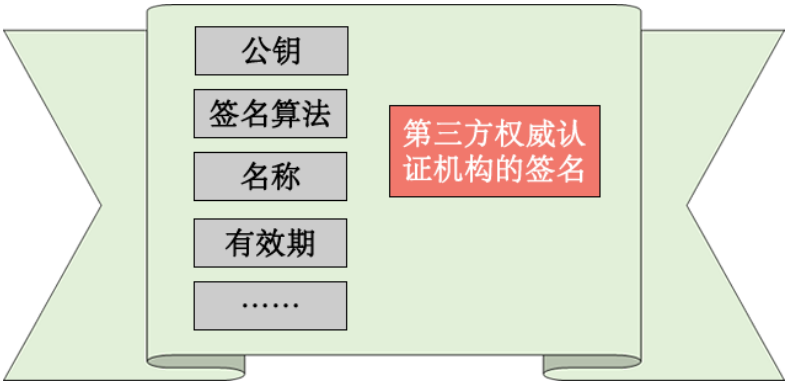


图 1-27 数字证书

第三方权威认证机构就是指政府的认证部门、或者有信誉的非政府组织、或者是非盈利性的民间组织，这些认证机构同样拥有自己的公/私钥，且把公钥授予了浏览器并且也得到浏览器的接受承认——即浏览器已经保存了认证机构的公钥。

下面以作者本人的网站的证书获取和认证为例讲解整个过程。

1. 我产生自己网站（即 www.veryopen.org）的公/私钥，然后把公钥和我的身份信息发给 DigiCert 公司。DigiCert 公司是一家权威的数字证书颁发机构。

2. DigiCert 公司收到我的申请之后，核实我的身份（采用各种线上和线下核实方法）。

3. 我支付数字证书费用（按年收费，有效期越长费用越高）。

4. DigiCert 公司产生数字证书并反馈给我。签名过程见下图所示，最后把证书内容和第三方权威机构的签名发回给我。

权威机构颁发证书的过程参见图 1-28。

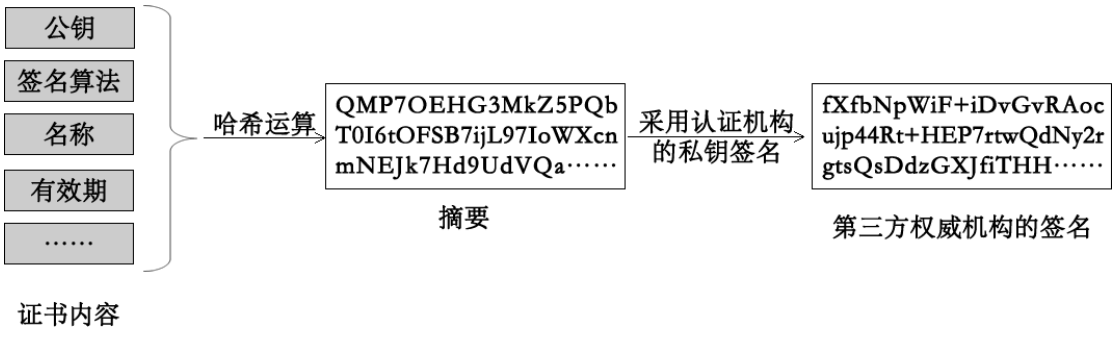


图 1-28 第三方权威认证机构的签名过程

5. 我收到数字证书后导入到网站服务器中。从此等待别人浏览我的网站。

当另一个人打开我的网站时，ta 的浏览器会收到我的网站服务器发来的数字证书，这时浏览器验证数字证书的过程是这样的：用第三方权威认证机构的公钥（比如 DigoCert 的公钥已经保存在浏览器中）验证数字证书中的签名（即第三方权威认证机构签名）——从签名算出摘要（不妨称为“摘要 1”），再对证书内容做同样的哈希运算，算出摘要（称为“摘要 2”），如果摘要 1 等于摘要 2 说明认证成功，否则就是认证失败。如果认证失败，浏览器会发出警告，并拒绝打开网站。如果认证成功，浏览器就用证书里的公钥加密一段随机产生的字符串（不妨称为“字符串 1”），然后把加密后的密文发给网站服务器，网站服务器用私钥解密后得到“字符串 1”，此后双方把“字符串 1”作为对称密码，用于加/解密此后交互的网页内容。因为对称加/解密运算效率要远远高于非对称加/解密运算。

图 1-29 是查看中国银行网站的数字证书。

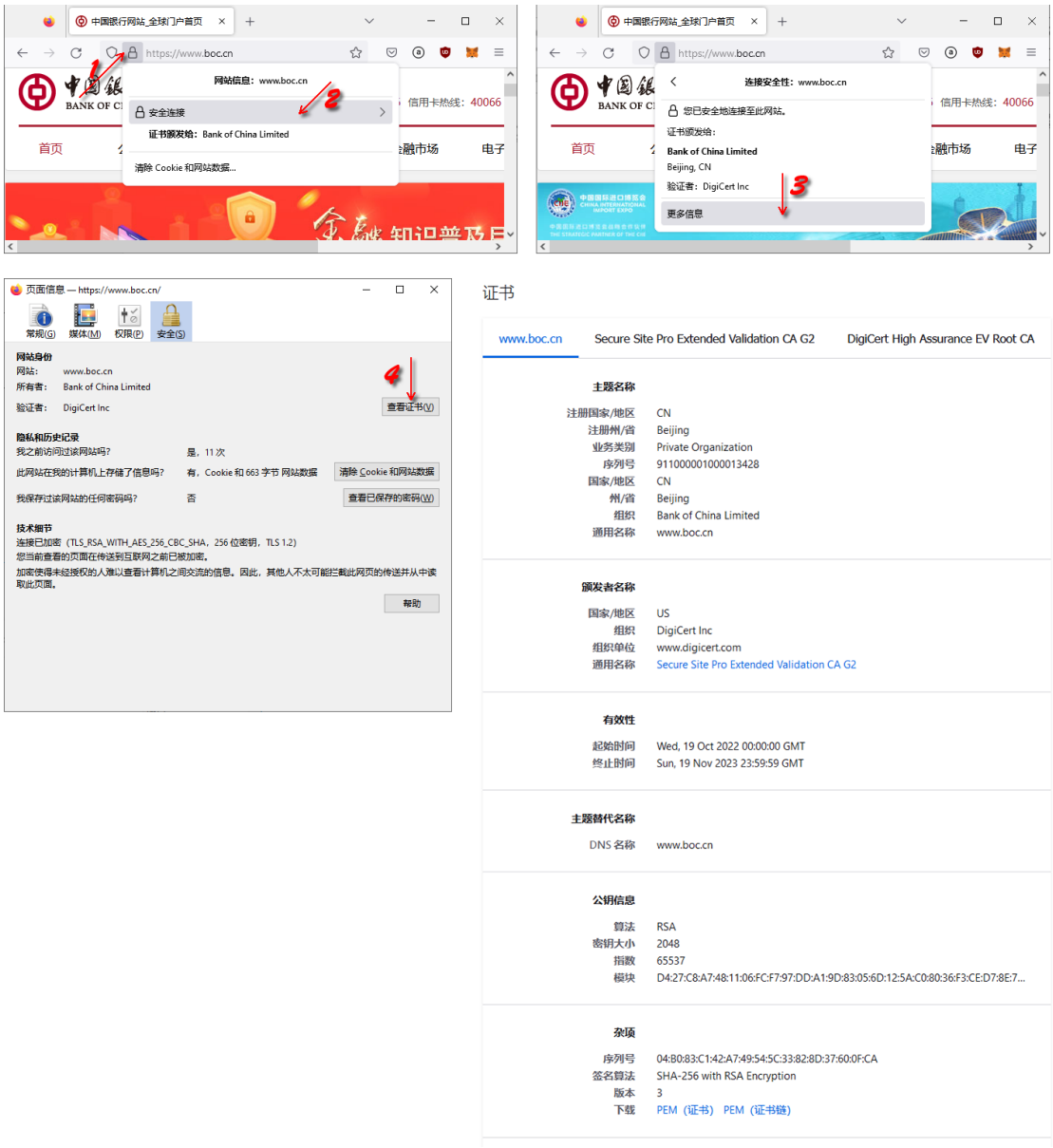


图 1-29 中行网站的证书

能颁发数字证书的权威机构称为 CA（Certificate Authority），也就是营运 PKI 的机构，PKI 是公钥基础设施(Public Key Infrastructure)的英文缩写，由于这个机构的公钥得到了普遍认可（比如浏览器都保存了此公钥），所以它能充当一个公证人的角色，即凡是用它的私钥签过名的公钥都被世人认为是可信的。被 CA 的私钥签过名的公钥就称为数字证书，以前有一个术语叫 SSL 证书，现在改名为 TLS 证书。用户如要获得数字证书，需要按年支付费用，在中国还不便宜，国外有人不高兴了，尼玛的不就是一串字符嘛，你每年都收我的钱，所以就搞出了开源数字证书的 CA：Let's Encrypt，不要钱但每三个月要刷新一下证书，可去其官网 https://letsencrypt.org/ 进一步了解如何使用。本书作者个人的网站（https://www.veryopen.org）就是采用这个开源数字证书。

如果是企业内部使用证书，比如 IT 系统的东西流量的数字证书（如 Kubernetes 集群中的通信双方大量采用数字证书），可以采用自签数字证书——企业内部自己搭建一个 CA 并产生一对 CA 公/私钥，然后用这个 CA 私钥签名其他公钥来产生数字证书，并把 CA 的公钥下发给各个需要验证数字证书的服务端和客户端。

1.5 哈希运算

哈希运算技术可用于防止电子数据资料被篡改。为了更容易理解哈希运算，我们先看下面一个场景。

A 公司今年挣了不少钱，年底想以网上抽奖的方式回馈客户，每个客户都可以登录公司网站抽奖，抽到的奖券中包含奖金数额（0 到 1 万之间的随机数），其实奖券就是一张名片大小的图片（图 1-30），客户把奖券发回给 A 公司领奖，公司收到奖券后按上面的金额给中奖的客户转钱。



图 1-30 A 公司的奖券

现在的问题是如何防止客户修改奖券上的金额，毕竟现在的 P 图技术这么强，改几个数字太容易了。你可能会说每张奖券在 A 公司都有留底啊，收到客户反馈的奖券，按编号找出留底的奖券，两者一比较就知道是否修改了。但此时客户说是公司修改了，公司说是客户修改了，法律到底相信谁？好吧，重新设计一下奖券，对图片虚线右侧部分做个哈希运算，并把运算结果写明在左侧下方，同时一旦抽中的奖券立即公布在公司网站上，参见图 1-31。



图 1-31 防止篡改的奖券

只要修改了图片右侧哪怕一个像素，再次作哈希运算，结果就完全不一样了。也许你可能会说：我修改金额之后再用新的哈希结果替换图片左侧旧的哈希值。关键是你不知道 A 公司采用哪种哈希算法（目前可用的哈希算法有很多，比如 md5、sha-1、RipeMD-160、sha-256 等）。

你可以把哈希运算想象成一台神秘的榨汁机器，你往里扔一根甘蔗，出来一杯饮料，扔一个西瓜也出来一杯饮料，扔一个地瓜还是出来一杯饮料，扔一条鱼依然出来一杯饮料，……。只是每次出来的饮料，它的颜色和味道各不相同罢了。假设刚才那个地瓜有点脏，你清洗后再扔进去，出来的饮料的颜色和味道却与没清洗时完全不同。

电脑中的哈希运算就是不管处理什么电子资料（如文本文件、图片、视频、word 文档、程序等），计算的结果都是固定长度的一串字符，比如都是 32 个字符。哈希运算有以下三个特征：

1. 相同的输入一定会得到相同的输出；
2. 不同的输入得到相同的输出是极小概率事件；
3. 即使对输入做极细微的改变，输出却完全不同。

第 1 个特征说明不管何时何地对没篡改的内容做哈希运算，永远会得到相同的结果，即验证是没有时效性和区域性限制的。第 2 个特征说明你最好打消这种侥幸心理：我修改了原内容，但碰巧得到相同的哈希值。输入不同而输出相同的现象称为哈希碰撞，当规定输出串足够长时（比如加密币是 32 个字符），碰撞的概率小到可以忽略不计，尽管理论上碰撞是一定存在的。第 3 个特征表明：即使你修改一个 word 文档（是一本上千页的书稿）中的一个标点符号，哈希运算的结果会完全不同，输出串中的每一个字符都变了，也就是说不可能从哈希值中寻找规律，从而反推算出原文是什么。

只告诉你哈希值和哈希算法，找出原文的唯一方法就是暴力破解：猜一个明文，做哈希运算，比较哈希值，如果哈希值相同那恭喜你，你猜到了，否则继续猜一个，如此循环下去。比如现在银行里只保存储户密码的哈希值（md5 算法，输出串长度为 16 个字符），当储户取钱时，输入六位数的密码，程序对输入的密码做 md5 哈希运算，然后与保存在数据库中的哈希值比较，两个哈希值相等说明这次输入的密码正确。作为银行机房的系统管理员，是很容易搞到所有客户的密码的哈希值的，然后对 000000 到 999999 之间的每一个数字（因为银行密码就是由六位数字组成）做 MD5 哈希运算，再与搞到手的哈希值比较，这样就能轻易猜出每个储户的密码。不过现在银行提高了暴力破解密码的难度，比如加盐，或采用输出长度更长的哈希运算，或采用慢哈希运算（如 bcrypt）等措施，这些内容超出了本书的范围，我就不再细讲了。

在加密币里，大量采用哈希运算来防止他人恶意篡改交易。

下面我们来做一些实际实验，计算一个 word 文档的哈希值，然后修改文档中的一个标点符号，再来验证是否修改过。继续采用 gpg4win（在“§1.2 数字签名”小节中已经安装），参考图 1-32。

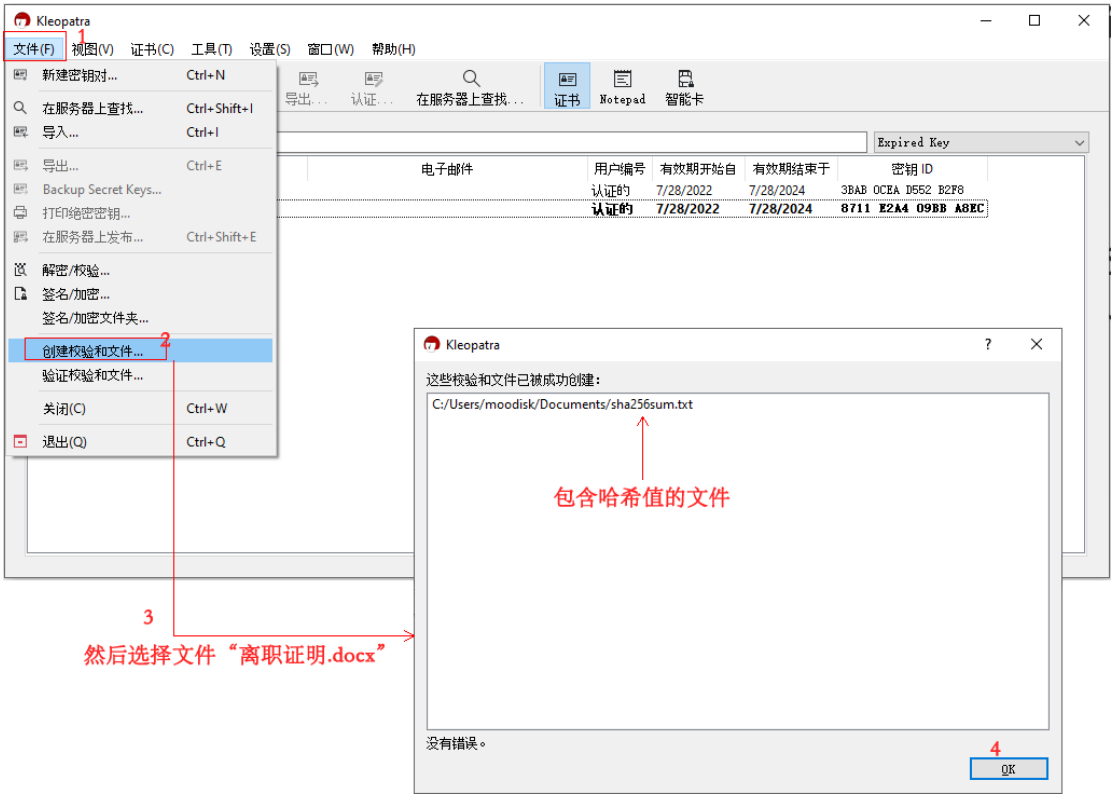


图 1-32 计算文件的哈希

产生的哈希值保存在文件 sha256sum.txt 文件中，这里文件就在“离职证明.docx”文件所在的目录中，打开此文件，发现里面的内容是：

```
bb77db0f14479ac460ca1b9ccf6d85443c7f636ff92932d05e576fabc2517f4d  C:/Users/moodisk/Documents/离职证明.docx
```

说明文件“C:/Users/moodisk/Documents/离职证明.docx”的哈希值为 bb77db0f14479ac460ca1b9ccf6d85443c7f636ff92932d05e576fabc2517f4d。

验证文件是否被修改过，只要把“离职证明.docx”和 sha256sum.txt 放在同一个目录中，然后点击图 1-28 中的“验证校验和文件”，接下来定位到 sha256sum.txt 文件即可进行验证。现在修改“离职证明.docx”（比如修改一个标点符号），再看验证的结果，如图 1-33 所示。

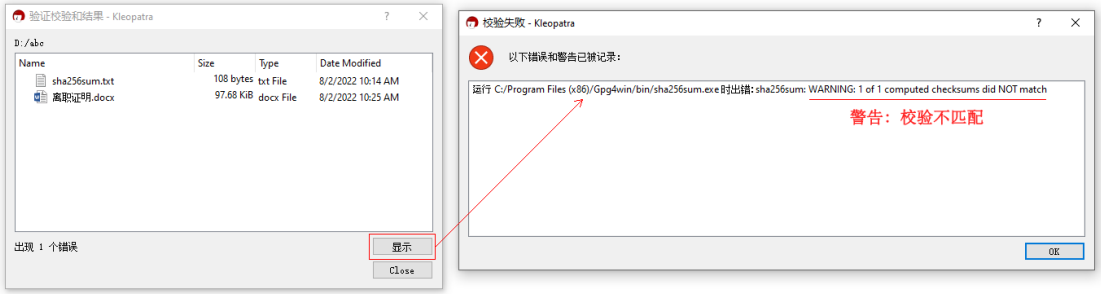


图 1-33 验证没通过

通过计算文档的哈希来确保文档没有被修改，在电子合同方面使用很广泛。只要哈希值被权威部门保管好，任何一方都无法修改电子合同。另一个应用广泛的场合是保证程序源代码不被恶意篡改——对源代码文件计算哈希值，然后公布在网上，别人下载源代码文件后再计算哈希值，然后与网上的哈希值比较，如果不同表示源代码被修改过。

注意：计算哈希值时可以一次性选择多个文件。

现在总结一下：公钥加密是为了防止泄密，私钥签名是为了防止抵赖，哈希计算是为了防止篡改。

因为稍微修改一下原文，哈希值就完全不同，而相同的内容哈希值总是一样。因为加密币的最佳实践中提到一点，那就是每一次收币采用一个新的钱包（即不同的公/私钥对），由此我们必须记住一大堆的私钥，这可不是一件容易做到的事儿！结合哈希计算的特点，有人发明了由一个随机数产生大量私钥的办法，而我们只需记住这个随机数即可，这个随机数称为种子。下面做个实验：

```
user@btceth:~$ dd if=/dev/random count=1 status=none | sha256sum
10d08831a5cf635d5fefd8ebb63d9c60671c04edc3d3af38da8c7695f3a44d85  -
user@btceth:~$ seed=10d08831a5cf635d5fefd8ebb63d9c60671c04edc3d3af38da8c7695f3a44d85            //定义种子
user@btceth:~$ for i in {1..5} ; do echo $seed$i | sha256sum ; done                          //产生五个私钥
cb0e22d4f444b9adc86f915c4cc77e8a455f5072296fee2655db7806a0327fba  -
9c84aa9b8c8552ddf3ba7dff8db93e7b57d3c443326dc4ab5c04f435740b1d5a  -
aab6fd7a1e0f1de07443902ee742f53c81cf2400126c79e24445f5780a1ab6d6  -
bcfe59d353d14465f2606f0d334dad038468524d97a3d2dd383e48767921b82b  -
302843c59f8b719cc14c4083baf8dd4254a54d952312d9de9dc6fc437b7489cf  -
```

由种子产生更多私钥的方法就是在种子末尾附加一个数（例子中就是 1~5 之间的一个数），然后作 sha256sum 计算，这样我们只要记住种子 10d08831a5cf635d5fefd8ebb63d9c60671c04edc3d3af38da8c7695f3a44d85 和产生密钥的方法即可。而且产生的私钥的个数几乎没有限制。不管采用什么方法，只要哪怕微弱修改一下种子，就可以产生一个完全不同的私钥。种子的长度可大可小，本例中种子的长度是 256 位二进制数，私钥也是 256 位二进制，所以 256 位的种子本身就可以作为私钥使用，而新产生的私钥也可以作为新的种子使用。这样我们就可以用一棵树表示由一个种子产生众多私钥的过程，参见图 1-34 所示。

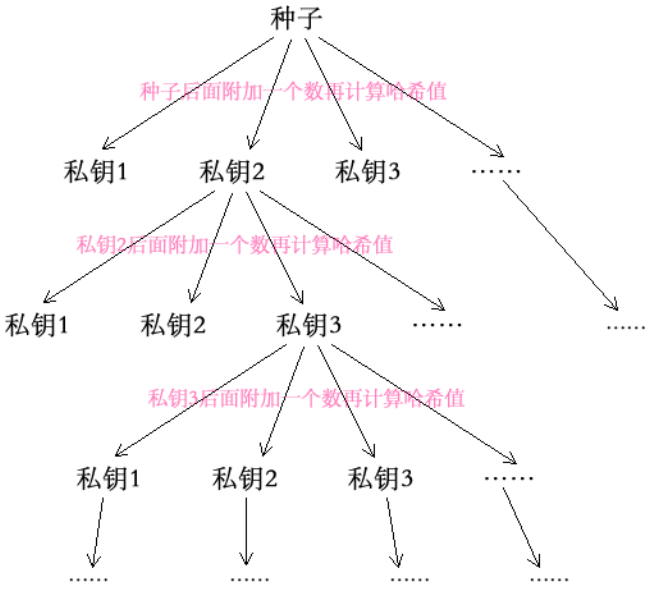


图 1-34 由种子产生私钥树

这就是 HD 钱包的最核心机制。关于 HD 钱包，请参考下一小节。

1.6 数字钱包

数字钱包本质上就是私钥（256 位的二进制数）！就是私钥!!就是私钥!!!重要的事情说三遍，你记住了一个私钥就拥有了一个数字钱包。有了私钥，你就有了一切：公钥、地址和加密货币、美女、房子、车子、……。别跟我谈思想，在中国这片土地上，谈思想显得很滑稽。

1.5.1 传统钱包

日常生活中的钱包通常表现为一个程序（如手机 app），此程序内嵌了私钥和地址，同时整合了操作，如买卖加密币等。

（钱包的）地址就相当于你银行的账号，把账号告诉别人，他就可以给你转账，而私钥相当于你的银行账户的密码，这个你绝对不要告诉任何人。但与银行账户不同的是：如果忘记了私钥，那么你的资金全部 game over 了，连同美女、房车、……都成了落花流水，消失不见了。所以安全的宗旨是你必须要好好保管私钥，只要有了私钥，公钥和地址就可以重新计算出来。因此创建钱包就是产生一个私钥，而加密币的私钥就是一个 256 位的二进制数。把私钥和地址写在一张纸片上，这就是所谓的纸钱包了，参考图 1-35。

私钥：
cf4d1030fece1071b73246793d06d3d08ead385481ee28cbf7b22859772798d1

地址：
0x3db989b56b83C5775e9a9F990EC80038F4160030

纸钱包

图 1-35 纸钱包

其实地址可以不写，因为可从私钥导出地址，但是为了方便别人转账给你，还是建议写出地址，不至于每次转账时总是要从私钥导一次地址。频繁使用私钥可不是一个好习惯，这样很容易暴露私钥！

什么？又怕纸片丢了？你就不知道把开头 5、6 个字符记在脑袋里，而把其余的字符抄在纸上吗？这样就算丢了，别人也不知道你的私钥了！不过这种方法还是存在风险：万一忘记了这 5、6 个字符就麻烦了！或者别人盗取了纸片并且发现私钥少了几个字符，是很容易暴力破解的。这时做个伪纸钱包是最后的办法了，做法如下：

用你终身不会忘记的字符串（比如常用的密码、姑姑的生日、小孩的姓名拼音等）替换私钥中某处（开头、结尾或中间）相同多的字符，从而得到一个新的私钥，再导出新私钥的地址，旧私钥和新地址组成一个伪纸钱包，参考图 1-36 所示。

私钥：
cf4d1030fece1071b73246793d06d3d08ead385481ee28cbf7b22859772798d1

地址：
0x5fd474551d2d49CE40bfAfAbB44d158F128E6bB1

伪纸钱包

图 1-36 伪纸钱包

实际的私钥是（假设用你小孩的生日）：

20180828fece1071b73246793d06d3d08ead385481ee28cbf7b22859772798d1

即用你小孩的出生年月日替换伪纸钱包上的私钥的前 8 个字符。不过建议最好不要用全数字，同时建议不要把地址写在纸钱包上，地址就保存在手机或电脑上，因为可能要随时发给别人，让对方转币。这样把地址告诉别人时采用拷贝粘贴的方式不容易出错，手工输入很容易出错，地址输错了，那加密币就被转到别人口袋里去了，是找不回来的。因此终极的伪纸钱包就成了图 1-37 这个样子了。

cf4d1030fece1071b73246793d06d3d08ead385481ee28cbf7b22859772798d1

图 1-37 终级的伪纸钱包

等等，万一纸片弄丢了，岂不是自己也忘记了私钥？！确实是个问题，不过我的做法是多写几张纸，七大姑八大姨兄弟姐妹那里都放过去，嘱咐他们要和贵重的细软放在一处。后面还会讲到以电子资料的形式加密保存，然后备份到多分 u 盘，存放在各处，这个称为磁钱包。

还有一种方法，就是采用 bip38 加密私钥——采用比较复杂的对称密码对私钥进行加密，加密后的私钥可以保存在电脑上，等以后要用时临时解密出来。

所以如果你投入加密币的资金比较多的话，私钥最好只掌控在你一个人的手里，当然，万一你遭遇不测（比如突然驾鹤西去），你的加密币总该留给家人吧，所以还得让最亲密的人也知道私钥。不要相信任何组织和个人，这是投资加密币的信条之一，信条之二是保管好了私钥就保管好了你的财富。私钥离你越“近”越好，安全程度从高到底排序依次是：脑钱包、伪纸钱包、纸钱包、磁钱包、硬（件）钱包、软（件）钱包、平台钱包。其中软钱包和平台钱包，私钥不是由你一个人控制，硬钱包如果连上互联网，存储在其中的私钥有可能也脱离了你的控制。

记住：钱包里只保存了私钥，可能还有地址，钱包里根本没有加密币！加密币只保存在区块链上（就像你的钱没有存放在银行卡上，而是存放在银行里一样），只要你记住了私钥，钱包丢了也不要紧（当然不能把私钥泄露给别人）的。只要保管好了私钥，你的资产一定安然无恙！相信我。

下面介绍如何加密存储磁钱包。

首先把私钥、公钥和地址保存在一个记事本文件中，假设此文件命名为 wallet.txt。然后参考“§1.3 数字签名”小节安装 gpg4win，最后参考图 1-38 加密 wallet.txt 文件，密码尽量长，而且最好包含大小写字母、数字和其他符号。加密后的文件名为 wallet.txt.gpg，放在和 wallet.txt 相同的目录中。

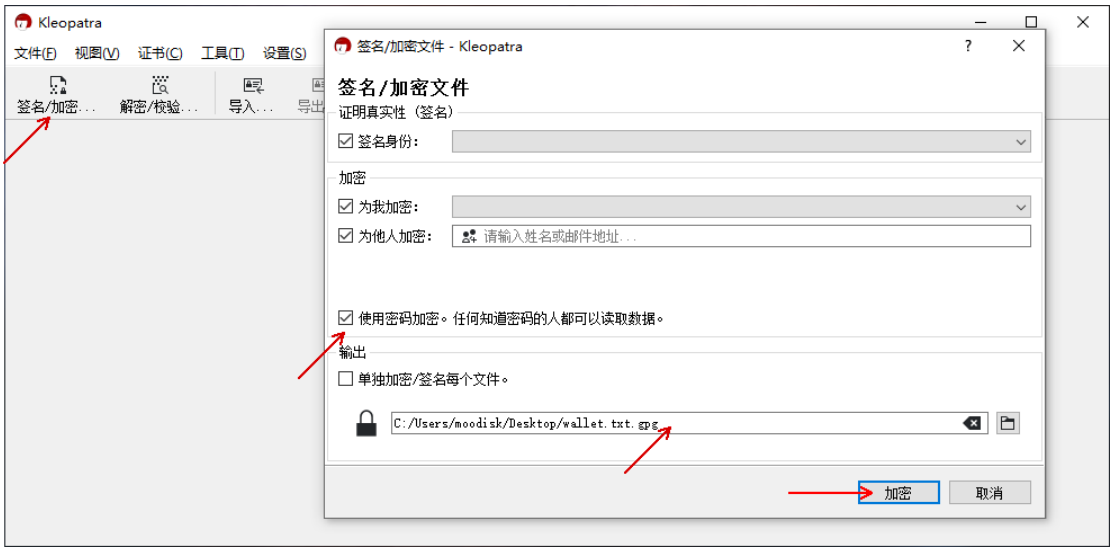


图 1-38 加密钱包

保管好 wallet.txt.gpg 文件，把 wallet.txt 删除（选中文件后按 Shift+Delete 键彻底删除文件，这样就不会放入垃圾桶）。以后要用到私钥时，临时解密出来，参考图 1-39 所示操作。

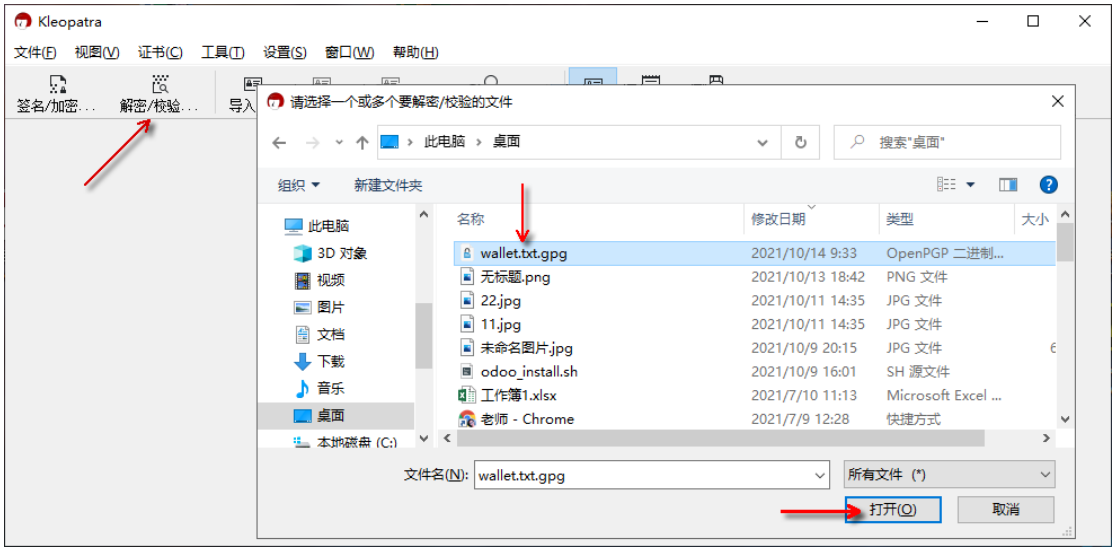


图 1-39 解密钱包

正确输完密码后在相同的目录中生成一个解密的文件 wallet.txt，切记：用完私钥后一定要彻底删除 wallet.txt 文件（鼠标点击选中 wallet.txt 文件，然后按 Shift+Delete 两个键即可彻底删除文件）。

不建议把加密后的文件 wallet.txt.gpg 保存在电脑上，建议你同时保存在三块相同的 U 盘里，然后拿烧烤用的锡箔纸或铝箔纸（见图 1-40）把 U 盘层层包裹起来，这样可以防磁防短暂的明火（但不能隔热），再密封在塑料袋中，这样就可以防水了，注意：密封时袋中的空气越少越好，氧气少，U 盘上的电子器件不容易氧化老化，我是用一个大的保鲜袋，像糖葫芦串一样连续打上三、四个结以确保封口严密。最后把 U 盘分开在异地存放，比如交给居住在不同地方的兄弟姐妹们保管。

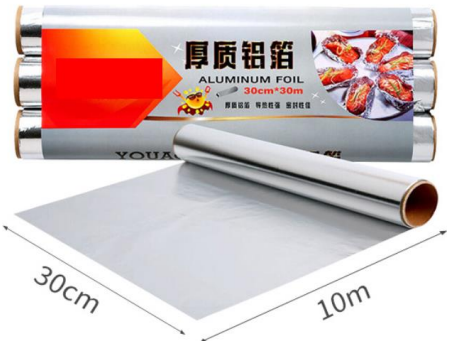


图 1-40 包裹 U 盘的锡箔纸

前面讲到脑钱包是最安全的（即记住私钥），但是现实生活当中又有几人能做到？而且根据中本聪的本意，一个地址只做一次交易，也就是说每一次交易都要更换私钥——一个钱包只交易一次。例如张三往李四的 A 钱包转一个比特币，以后李四把 A 钱包里的一个比特币转出去后，A 钱包就废弃，再也不用它了。这样就必须记住很多私钥，如果交易多，光记住私钥耗费的精力就够人喝一壶的，为此，有人搞出了 HD 钱包，具体参考下一小节。

1.5.2 HD 钱包

为了解决传统钱包难以记住一堆私钥的问题，人们提出了 bip32 规范——推出 HD 钱包，就是分层确定性（Hierarchical Deterministic）钱包，BIP32 规范文档在 https://en.bitcoin.it/wiki/BIP_0032。用通俗的话讲就是 bip32 规范能确定地从一个种子不可逆地算出无数多个私钥和公钥（钱包），别人从这些算出来的私钥和公钥中无法反推出种子。现在好了，你只要记住一个种子即可。但是种子也不好记啊，于是人们进一步制定了 BIP39 规范（https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki），采用 12 个或更多的单词组成一组助记词，比如“stove apology ancient exclude donate sail cheese strong decline first solution air license cruel agree”由 15 个单词组成的助记词。这些单词不能由自己随便定义，因为最后一个单词包含了校验码。https://iancoleman.io/bip39/是在线产生 BIP39 型的 HD 钱包的网站，你不妨上去试试。有了 bip32 和 bip39 两个规范之后，我们现在只要记住助记词即可，因为由助记词可算出种子，再由种子算出一堆的私钥。我靠，这么好的东东只能用于比特币，能否也用在诸如以太币等其他加密币呢？于是人们又推出了 bip44 规范，在扩展私钥生成树（下面会详细介绍）的节点编码中引入了币种。后来推出的 BIP84 规范可产生以 bc1 开头的原生隔离见证地址，这样的地址交易成本低且验证效率高的特点，被新钱包普遍采用。

可在网站 https://iancoleman.io/bip39/ 上申请 HD 钱包，参考图 1-41。

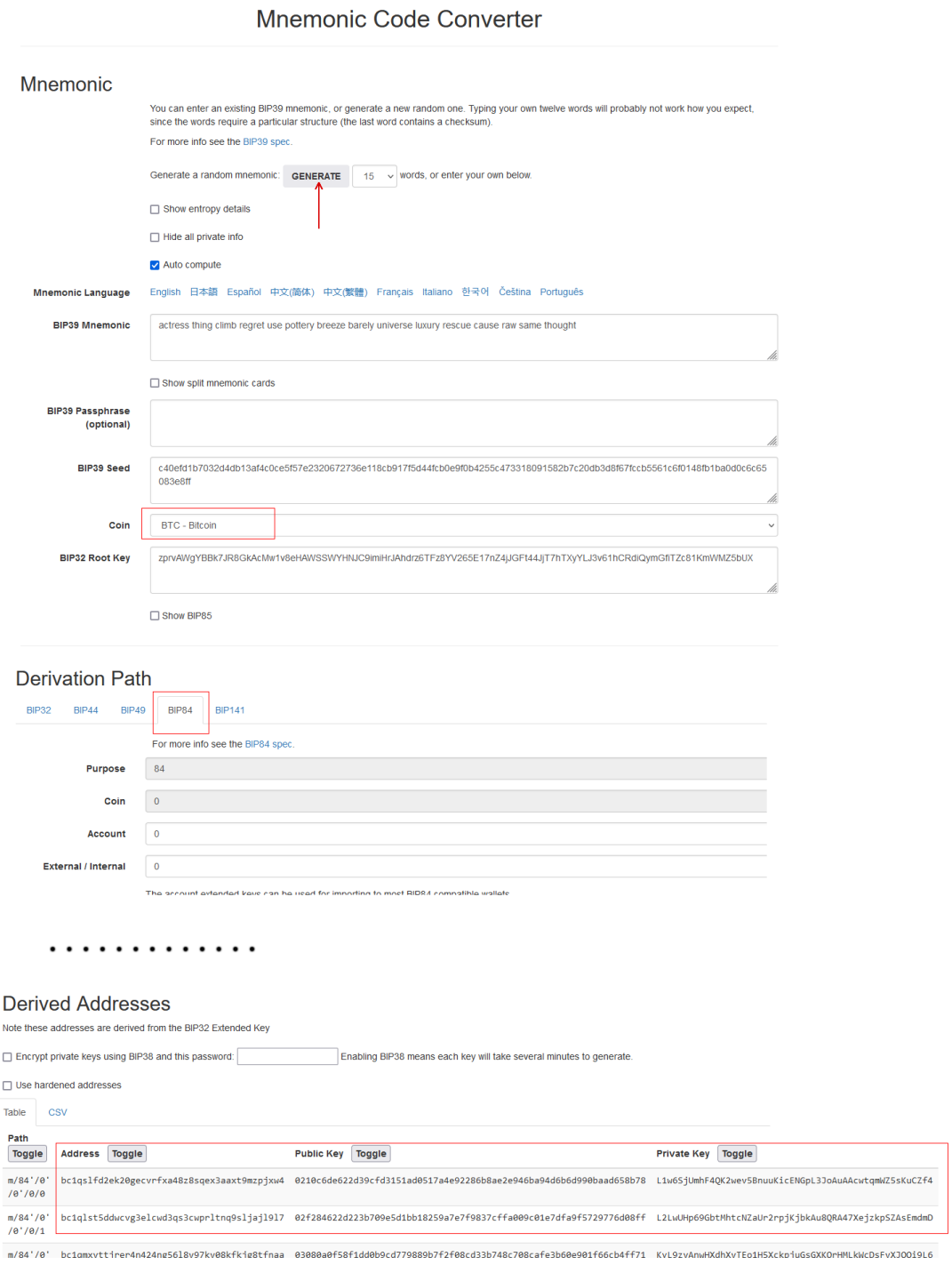


图 1-41 申请 HD 钱包

1.5.3 钱包地址

比特币的收款地址具有多种类型，每种类型都有各自的优点和特色，了解各种比特币地址类型关涉安全性、隐私性和更低的交易费用，综合比较请参考表 1-3

表 1-13 比特币各种地址类型汇总

地址类型	说明	地址格式	长度	交易费用	支持的功能
P2PK	Pay-to-Public-Key (最早期的地址，现在很少用了)	以 1 开头	66-130	很高	N/A
P2PKH	Pay-to-Public-Key-Hash (传统地址，交易费很高)	以 1 开头	26-35	高	N/A
P2SH	Pay-to-Script-Hash	以 3 开头	26-35	中	多签
P2WPKH	Pay-to-Witness-Public-Key-Hash (SegWit)	以 bc1 开头	42	很低	原生隔离见证
P2WSH	Pay-to-Witness-Script-Hash	以 bc1 开头	62	较低	多签/隔离见证
Nested SegWit	P2SH wrapping P2WPKH/P2WSH	以 3 开头	26-35	较低	多签/隔离见证
P2TR	Pay-to-Taproot	以 bc1p 开头	62	很低	多签/隔离见证/智能合约

1. P2PK 类型

是 Pay-to-Public-Key 的首字母缩写，这是最早使用的地址类型，即直接把公钥作为收币地址，用户必须使用此地址（即公钥）对应的私钥签名才能花钱。因为直接暴露公钥，所以降低了隐私，而且一笔交易的数据量比较大从而需要更多的交易费，所以现在这类地址几乎不再使用了。一个公钥为 512 位二进制数（压缩公钥为 256 位）。比如发生在 2010 年 4 约 10 日的交易 67b1a656c0b2c8cc8871cb6febb25da58dfb14f9878c2c79f2d4f0fd8acbfa37 给公钥 410443289d7517545327430c3e0c5547f087bf32669114cd1ec51c39e909ba657562338f2047442d063ba6e1c1ca522ca23b7035166c6947bf773485205da6431e23 转入 49.9 个比特币，这个公钥本身就是地址。

2. P2PKH 类型

是 Pay-to-Public-Key-Hash 的首字母缩写，就是把公钥的哈希值作为收币地址，这样就客服了 P2PK 地址存在的两个缺点：暴露隐私（公开了公钥）和交易数据量大。比特币主人花钱时需要提供公钥和签名，公钥做同样的哈希计算后验证是否与地址相等，其次用公钥对签名进行验证。P2PKH 在历史上使用最广，但是也存在一些缺陷：效率不高、安全性欠缺、功能弱（相对于后续出现的地址类型来说）。比如“1EKXAAiJDntbEK3WSVW3vTcwYJ5ciJ5kSL”就是一个 P2PKH 地址。

2010 年 5 约 22 日发生的一笔交易 a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d，出/入钱包的地址都是 P2PKH，这是美国人 Laszlo Hanyecz 耗费钱包的 20 个 UTXO，凑齐一万个比特币购买了一块比萨饼，因此这日也被打上“比特币的披萨日”的烙印，就是图 1-42 中的那个家伙。



图 1-42 Laszlo Hanyecz 花一万比特币购买比萨饼

按今天（2025 年 1 月 19 日）比特币兑美元计算，一个比特币值 104675 美元，一万个比特币约等于 10.5 亿美元，不过有人采访他，他说不后悔。

3. P2SH 类型

是 Pay-to-Script-Hash 的首字母缩写，意为支付到脚本的哈希——计算一个脚本的哈希并把它当作钱包的地址，这个脚本称为赎回脚本，即 P2SH 地址就是赎回脚本的哈希值并作 base58check 编码得到的值，即地址为 RIPEMD160(SHA256(赎回脚本))，它允许在满足指定脚本（脚本执行结果为真）或一组条件的情况下使用比特币，而不仅仅是证明公钥的所有权。发送者在交易输出中包含花费资金所需的赎回脚本的哈希值，而无需知道赎回脚本本身的详细信息。以后，接收者可以构建自定义输入脚本（例如多重签名要求或其他复杂条件），并通过“提供”与哈希匹配的赎回脚本、必要的签名或数据来花费资金。P2SH 地址支持多重签名钱包等高级应用，同时保护了隐私，因为脚本的细节是隐藏的。P2SH 还支持向后兼容性，因为它可用于以软分叉兼容的方式部署 SegWit 和其他协议升级。比如“38bPsA6ZXfRuxFD7efVXTkQd69422uzD4B”就是一个 P2SH 地址。

P2SH 的一个典型应用案例是多签，即允许多个人签名才能花钱。

3. P2WPH 类型

是 Pay-to-Witness-Public-Key-Hash 的首字母缩写，隔离见证（SegWit）分叉升级后引入的目前使用最广泛的地址类型。与传统的 P2PKH 类似，也是把公钥的哈希值作为收币地址。但是 P2WPKH 引入了 SegWit 的隔离见证数据结构，将解锁签名脚本移动到单独的见证字段。SegWit 隔离见证有效降低了交易数据量和交易费用，同时解决了交易延展性等关键问题。P2WPKH 地址还通过使用 Bech32 编码、增强的错误检查以及支持 Lightning Network 和 Taproot 等高级协议来提供更好的隐私。由于 P2WPKH 具备更高的安全性、效率和向前兼容性优势，目前使用最广泛。

P2WPKH 采用 Bech32 编码方式。Bech32 是 BIP 0173 和 BIP 0350 规范引入的隔离见证地址格式——以“bc1”开头的地址，在节约区块尺寸方面更具效率。

4. P2WSH 类型

是 Pay-to-Witness-Script-Hash 的首字母缩写，是在 P2SH 的基础上增加隔离见证，从而实现了隔离见证的多签这种极具吸引力的功能，一方面提高了效率、降低交易费用，另一方面还增强了比特币的智能合约功能（借鉴以太坊）。P2WSH 虽然比常规 P2WPKH 更复杂，但为更高级的可扩展性和可编程性铺平了道路，相信会有越来越多人的使用它。

5. 嵌套隔离见证地址类型

即 P2SH-P2WPKH 和 P2SH-P2WSH，由于一些老钱包不支持隔离见证地址类型，所以需要把隔离见证包裹在 P2SH 内部来实现向后兼容性。P2SH-P2WPKH 和 P2SH-P2WSH 是过渡性地址，会慢慢丢弃不用的。

P2SH-P2WPKH 地址与 P2SH 一样也是以 3 开头（测试网以 2 开头），都是支付到脚本的哈希（HASH160 运算的结果是 20 个字节），因此只从地址外观不能区分 P2SH 和 P2SH-P2WPKH 类型的地址。关键的区别是这个“脚本”（或称赎回脚本）：对于多签 2-of-3，P2SH 的“赎回脚本”类似“2 <公钥 1> <公钥 2> <公钥 3> 3 OP_CHECKMULTISIG”，而 P2SH-P2WPKH 的“赎回脚本”为“OP_HASH160 HASH160(见证脚本) OP_EQUAL”，其中见证脚本为“OP_0 HASH160(公钥)”。对于如何花费 P2SH-P2WPKH 类型的 UTXO，请参考“§4.5.2 解锁过程”小节。

6. P2TR 类型

P2TR（Pay-to-Taproot）是专门针对比特币的一种最新发展起来的地址类型，由 BIP86 规范引入，Taproot 是 Bitcoin 的一次大升级，在区块 709632 处（2021 年 11 月 12 日）被激活。它结合了 P2PK 和 P2SH 的优点，采用施诺尔（Schnorr）签名算法（BIP340 规范引入）而不再使用过去的 ECDSA 算法，从提供更强的隐私性和灵活性，采用了 MAST（Merkel 抽象语法树，由 BIP341 规范引入）和 Tapscript（由 BIP342 规范引入），在保持简化结构的同时支持复杂交易。P2TR 地址通过增强隐私、提高效率和灵活性，为比特币网络带来更强的扩展性和隐私保护。MAST 树的高度不能超过 128。

P2TR 类型的输出脚本为 Schnorr 公钥的哈希值加上 OP_1 前缀组成，这一点与 P2PKH 一样，只不过后者的公钥是普通的公钥。Schnorr 公钥也称为聚合公钥，详见后续介绍内容。

P2TR 地址生成方式不同于前述方式，而且采用了 Bech32m 编码，Bech32m 是 Bech32 的改进，因为后者存在一个缺陷：只要地址的末尾字符为 p，那么在它之前不管插入多少个 q，都不会影响地址的校验和。

P2TR 类型的输出存在两种花费方法：第一种方法是提供聚合签名和聚合公钥，第二种方法是提供 MAST 树上包含的任一输出脚本对应的输入脚本（类似 P2SH）以及路径。矿工判断一笔交易具体采用何种花费方式：如果对应的隔离见证的条款只有一项，说明就是第一种花费方法，如果对应的隔离见证的条款有多项，说明就是第二种花费方法。

1.7 区块链

任何一种加密币，每天都会发生大量的转账交易（不好意思，没有存/取现业务和贷款业务），这些转账交易被某个矿工保存在数据块中，然后这些新块被传播给其他的矿工保存，数据块的内部结构以及块与块之间的关系大致如图 1-43 所示（不同的加密币会做不大的改动）：

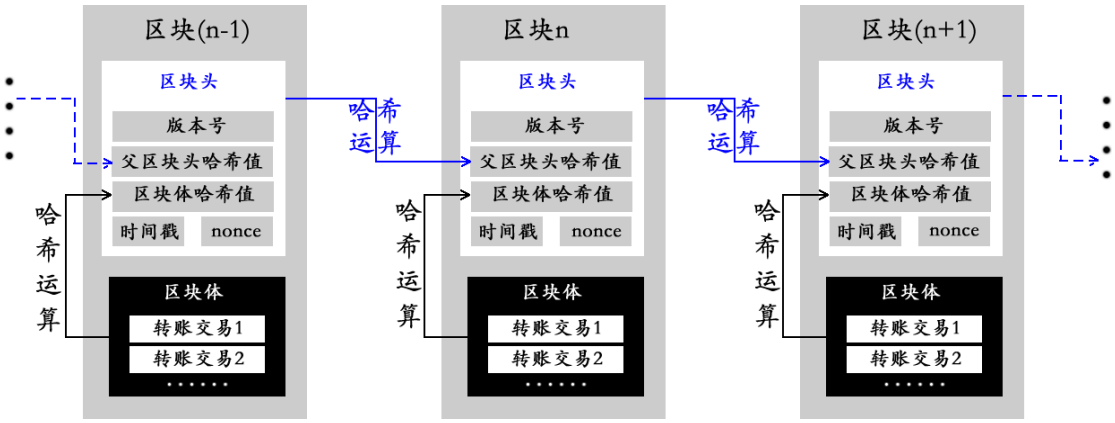


图 1-43 区块链

每个块的头部保存了父区块（前一个区块）头部的哈希值，这样各个块就组成了一条链——区块链，随着时间的推移，区块链会越来越长。同时头部又保存了本区块体的哈希值，这样就确保了一个区块的任何部分都不可能被修改。你想想，假如修改了区块 n 的转账交易 1，你还得修改本区快头部中的“区块体哈希值”，然后又得修改区块(n+1)中的“父区块头部哈希值”，还没完，区块(n+2)中的“父区块头部哈希值”也得修改，……。这还只是你自己电脑上的区块链，那保存在其他千千万万矿工的电脑上的区块也得修改，这样还改个屁啊，因为是根本不可能完成的事！

创造新区块的过程称为挖矿，率先创造一个新区块的矿工会奖励一定数量的加密币。矿工就是拼电脑的运算能力，这是真正的“能力之外的资本等于零”。找出符合如下条件的随机数 nonce：

新区块的头部作为哈希运算的输入，哈希运算输出满足前 n 个字符都为“0”。

通过调整 n 的大小来改变计算的难度，现在的矿工都购置了大量的专门的挖矿机，所以目前的 n 比较大了。挖矿的过程是这样：从交易池中挑选一些转账交易打包到区块体中，然后构造区块头部，再计算头部的哈希值，如果得到的哈希值前面不是 n 个零，就改变 nonce 后再计算哈希值，直到计算出来的哈希值满足前 n 个字符为 0，或者被其他矿工先找到这样的 nonce。如果是你先找到这样的 nonce，那恭喜，挖矿成功！然后就向其他矿工广播新块，其他矿工会验证收到的新块是否合法，如果合法就终止本次挖矿，并把新块接入区块链的头部。最后大家继续挖下一个区块。参见图 1-44 所示。



图 1-44 挖矿示意图

但要注意的是：每个矿工正在挖的那个区块头部不同，因为区块体中的转账交易很可能不同，所以存在区块头部的区块体的哈希值就不同。每个矿工的电脑上存在一个交易池，存放了他收到的全部还没有打包入块的交易，矿工的交易池中的交易不一定相同。当矿工挖矿时，会从交易池中挑选符合个人偏好（比如交易费越高越优先）的交易打包入块，一个交易一旦出现在区块链上的某个区块中，交易就算完成，完成的交易是没法撤销的。

每一种加密币都会控制新块的产生速度，比特币每十分钟出一个新块，如果挖矿的人多且采用了大量矿机，就调大挖矿难度，比如前 16 个字符都是零，否则就调小难度，比如前 12 个字符是零，总之要基本保持较固定的出块时间。

区块链、矿工、钱包的关系参见图 1-45 所示。

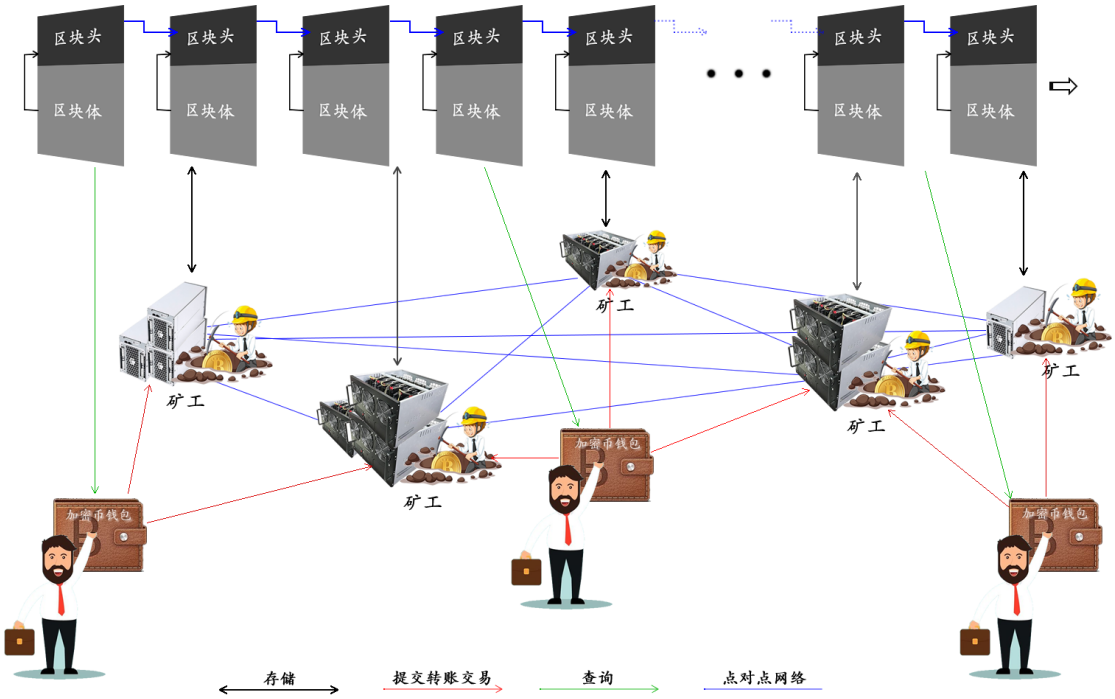


图 1-45 区块链、矿工、钱包的关系

区块链存储在矿工的电脑中，分布于全球的矿工的电脑组成一个点对点的网络，矿工的工作就是不断构建新区块（打包转账交易，并寻找满足条件的 nonce）以获取奖励和挣取交易费用；更广大的加密币钱包持有者日常操作就是查询，或者向附近的矿工提交已签名的转账交易；加密币钱包主要有四个功能：查询、存储私钥和地址、签名、发布交易。

1.8 记账与账本

对于家庭生活，如果你有记账的习惯，你可能会在账本上写着：

“今天收到 5000 元工资。”
“今天用 50 元购买了两袋卷纸。”
“张三还给我 1000 元。”
“交水电费 200 元。”
……

上面的每一句话就是一笔账，你再仔细看看，就会发现每笔账都会引起多个财产的变动。比如“今天用 50 元购买了两袋卷纸。”这笔账会导致家里的现金减少 50 元而卷纸增加两袋，再比如“张三还给我 1000 元。”会导致我的债权减少 1000 元而现金增加 1000 元，同样“交水电费 200 元。”会导致现金减少 200 元而债务（欠自来水和电力公司的钱）也减少 200 元。“今天收到 5000 元工资。”会导致债权（相当于公司欠你的钱）减少 5000 元而家里现金增加 5000。

一个家庭的财产主要是资产和负债，资产就是你的东西和别人欠你的东西，而负债就是你欠别人的东西。现在世界上通行的“借/贷”复式记账法规定：资产增加记贷方，减少记借方；负债增加记借方，而减少记贷方；有借必有贷，借贷必相等。会计也不再用自然语言记账，而是采用会计科目和会计分录。用会计分录对上面的账本记账如图 1-46 所示。



图 1-46 会计分录

一个会计分录中允许出现多个借方和多个贷方，比如我今天去超市花了 300.68 元，买回来一袋米、两斤猪肉、一箱啤酒，则记账如下：

借：现金 300.68 元
　贷：食物--大米 120.00 元
　　食物--猪肉 80.00 元
　　食物--啤酒 100.68 元

由于加密币不能取款、存款和贷款，所以只存在转账这一种交易类型，这样的话记账就简单多了，比如对于“我转给张三 0.001 个比特币。”就是从我的地址转给张三的地址 0.001 个比特币，即：

FROM	TO	VALUE
……	……	……
0x3db989b56b83C5775e9a9F990EC80038F4160030	0x5fd474551d2d49CE40bfAfAbB44d158F128E6bB1	100000
……	……	……

1 比特币=10^8 聪，交易额以“聪”为单位，所以 0.001 比特币等于 100000 聪。你只要使用某种工具（比如加密币钱包）把已经签名的转账交易数据发给附近的若干矿工就可以了，然后由矿工把这笔转账交易打包进一个新区块中，并开始挖矿（找到满足要求的 nonce），如果挖矿成功，新区快有效，表明你的转账交易顺利完成。否则等待下一个新块。

1.9 比特币简介

比特币的“样子”被人们化成如图 1-47 所示那样。



图 1-47 比特币的“样子”

中本聪结合计算机密码学（非对称加密、数字签名、哈希运算等）和建立在计算机分布式存储理论基础上的区块链技术，构建出一个划时代的产品——比特币！比特币是一种加密货币，是上帝用来摧垮人类巴别塔的工具。时至今日，人类构建了大大小小各种各样的巴别塔：国家、政府、金融机构、跨国公司、投行组织、联合国、共济会等等。按照各自的权势和财富，人类被置于巴别塔上不同层次，底层人的血汗不断被高层吮吸，也被高层奴役。而比特币不受任何个人或组织控制，它属于全体人类的，是跨越所有巴别塔的。

比特币到 2140 开采完毕，到时开采出来的比特币总量是 2100 万枚，所有的比特币均来自创建新区块时对矿工的奖励这条途径来发行的，规定第一个创世区块的奖励设定为 50 个比特币，此后每新增 210,000 个区块，奖励减半，直到 2140 年奖励为 0，2140 年之后矿工只能挣交易费了。大概十分钟左右产生一个新的比特币区块，产生区块的难度可以随时调整。

比特币算是一种通缩货币，属于稀缺资源，类似黄金。中本聪本人最初开采的 100 万枚一直没有用过，加上不断有人忘记私钥而永久丢失比特币，真正流动的比特币会少很多，所以比特币对美元具有很大的升值空间。

比特币单位有两个：比特币和聪，1 比特币=10^8 聪，在转账交易时采用单位“聪”，比如我转给张三 10000 聪，也就是 0.0001 个比特币。

投资比特币，你不用开户，你自己按照本手册前面介绍的方法制作一个钱包，然后直接让别人把比特币转到钱包的地址，你再通过其他途径把等值的美元或者人民币转给对方即可。比特币转账的记录会被矿工打包进新出的区块中，并最终接到比特币区块链上。你可随时按钱包地址查询转账记录，不过没有余额的概念，别人转给你的且你又没有花掉（转出）的全部转入的金额之和就是钱包的余额。

见表 1-4，假设我的钱包地址为 0xc2bd47b2895df22cd79a5c56129a6a906346b261，一共有三笔交易给我转账，但我花掉了一笔，因此我手上的总的比特币数量为 5000000+300000=5300000 聪=0.053 比特币。

表 1-4 资金去向链

FROM	TO	VALUE
0xe8dca8212dd0637ea0024a78d8a4e9bc52e0f84b	0xc2bd47b2895df22cd79a5c56129a6a906346b261	40000
0x8de47f5363aaa1a491716674332b31df53d73304	0xaa0ac10e198788d1e2bd6d6d2f3f540fc4640e48	1209000

……	……	……
0x3b1e73126e6d435631824cd4752623d9c1d45ca7	0xc2bd47b2895df22cd79a5c56129a6a906346b261	5000000 ②
……	……	……
0x9c34eff6d85d09e31e9aacdcc58d1c27750829cb	0xc2bd47b2895df22cd79a5c56129a6a906346b261	300000 ③
0xc2bd47b2895df22cd79a5c56129a6a906346b261	0xe1dff5c79fc320a97a3c9ffe0f431efdc5345383	39000
……	……	……

表中，有一笔转账（交易额为 40000 聪）被我花掉了（即转给别人 39000 聪，交易费 1000 聪），所以不能记在余额中。

假设现在我要给张三转 5200000 聪，如何操作呢？由于没有余额的概念，所以不能简单从总数中减去 5200000 聪。我需要花掉之前别人转给我的 5000000 聪和 300000 聪，假设交易费 1000 聪，因此本次交易的贷方有两个：给张三的地址转 5200000 聪，给我自己的地址转 99000 聪（5000000+300000-5200000-1000），而本次交易的借方也是两个：

上图中交易②的哈希值，交易②中的贷方序号
上图中交易③的哈希值，交易③中的贷方序号

一笔交易的哈希值是唯一的，通过查询哈希值可以唯一定位到一笔转账交易，前面已经提到过，一笔交易的贷方允许出现多个。比如李四发起了一笔转账交易，转给张三 50000 聪，转给我 800000 聪，转给张三的贷方序号不妨设为 0，转给我的贷方序号可能就是 1。所以“交易的哈希值”结合“贷方序号”能唯一确定我的一笔入账。

在网站 www.chainblock.com 上搜索钱包地址，可以查到与我钱包地址有关的所有交易，其中包括交易的 hash（哈希值）和以我地址为贷方的序号，参见图 1-48 所示。

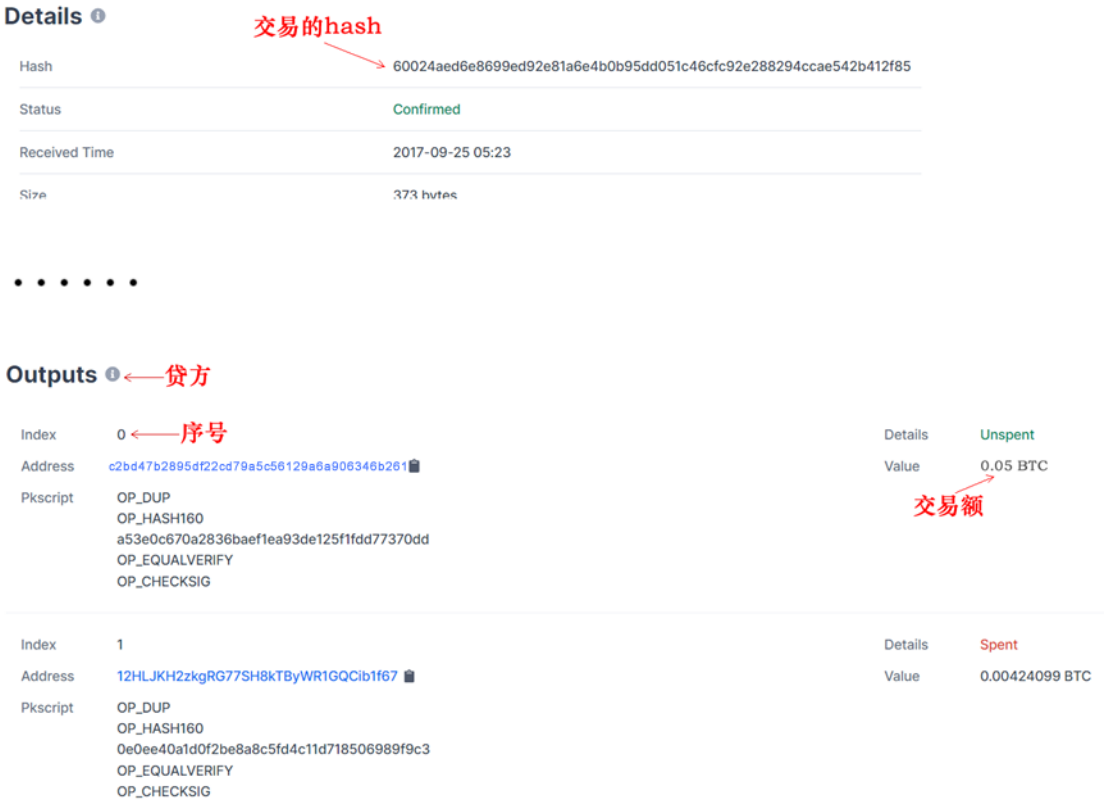


图 1-48 查询钱包的交易

一笔交易中的借方允许包含不同的钱包地址，贷方也是如此，但是必须满足等式：

一笔交易中的借方之和 = 贷方之和 + 交易费

可浏览 https://blockchain.info/unspent?active=<钱包地址>，获取某个地址的全部的未花掉的入账。

至于交易费的多寡，这跟一笔交易占用的字节数和成交的快慢有关，交易占用的字节数越多（包含更多的输入和输出），想更快成交，那么交易费就越高。一个一进一出的交易大概占用 180 个字节，其中“一进”140 字节，“一出”40 个字节，一天内完成交易，1 美元的交易费足矣。

可在网站 https://mempool.space/zh/查看实时的交易情况。

2 准备环境

建议采用电脑或笔记本电脑，不要采用手机和平板之类的移动设备，因为移动设备是最不安全的。另外建议从境外（含港澳台）购置品牌电脑设备（如苹果电脑、惠普电脑、德尔电脑等），而且尽量采用从微软官网下载的原版 windows 操作系统（https://www.microsoft.com/zh-cn/software-download/）或者苹果原版操作系统或者 Linux 操作系统，电脑上尽量不要安装国产软件（比如 qq、微信、搜狗拼音输入法等）。国内的软/硬件都不能保证没有后门。

为了操作方便和提高安全性，本书都是采用 Firefox 浏览器来交易加密货币，因为此浏览器是开源的，所以不太可能存在后门，而且可非常方便在“离线”、“在线”模式之间切换。

2.1 准备浏览器

强烈建议采用 firefox 浏览器，这是开源的浏览器，安全性很高。如果没有安装，请浏览 https://www.mozilla.org/en-US/firefox/enterprise/#download 网站下载企业版 firefox-esr，参考图 2-1 所示下载并安装。切记：千万不要从诸如 http://www.firefox.com.cn/这样的第三方网站下载，可能被植入后门。

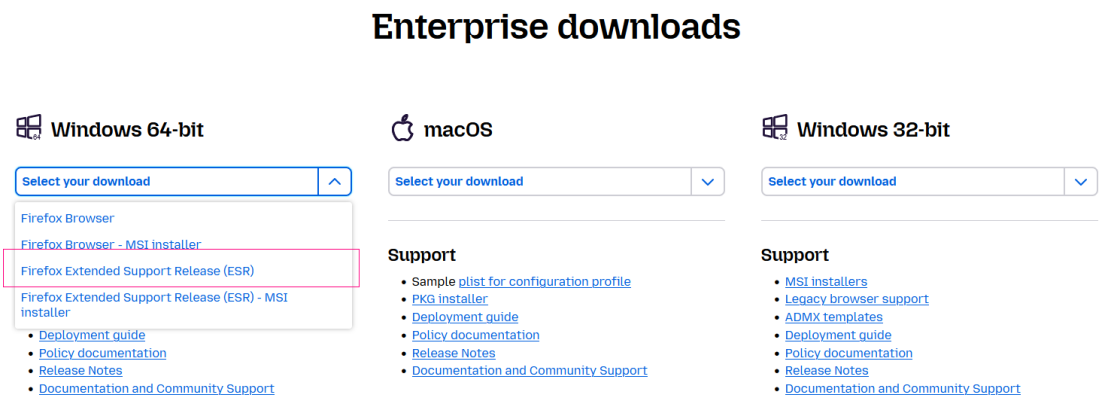


图 2-1 下载 firefox 企业版安装包

现在的电脑一般都是 64 位的，所以选择 Windows 64-bit，如果是老式机器，可能是 32 位的。然后双击下载的文件启动安装，按照提示操作即可。最后鼠标右键点击桌面上的“ 

”图标并选择“属性”，参考图 2-2 所示在目标后面添加 --private-window D:/www/index.html，再更改图标。这样以后每次启动浏览器就自动打开隐私窗口，隐私窗口不会记录任何信息，关闭浏览器后，一切上网痕迹都被清除了。

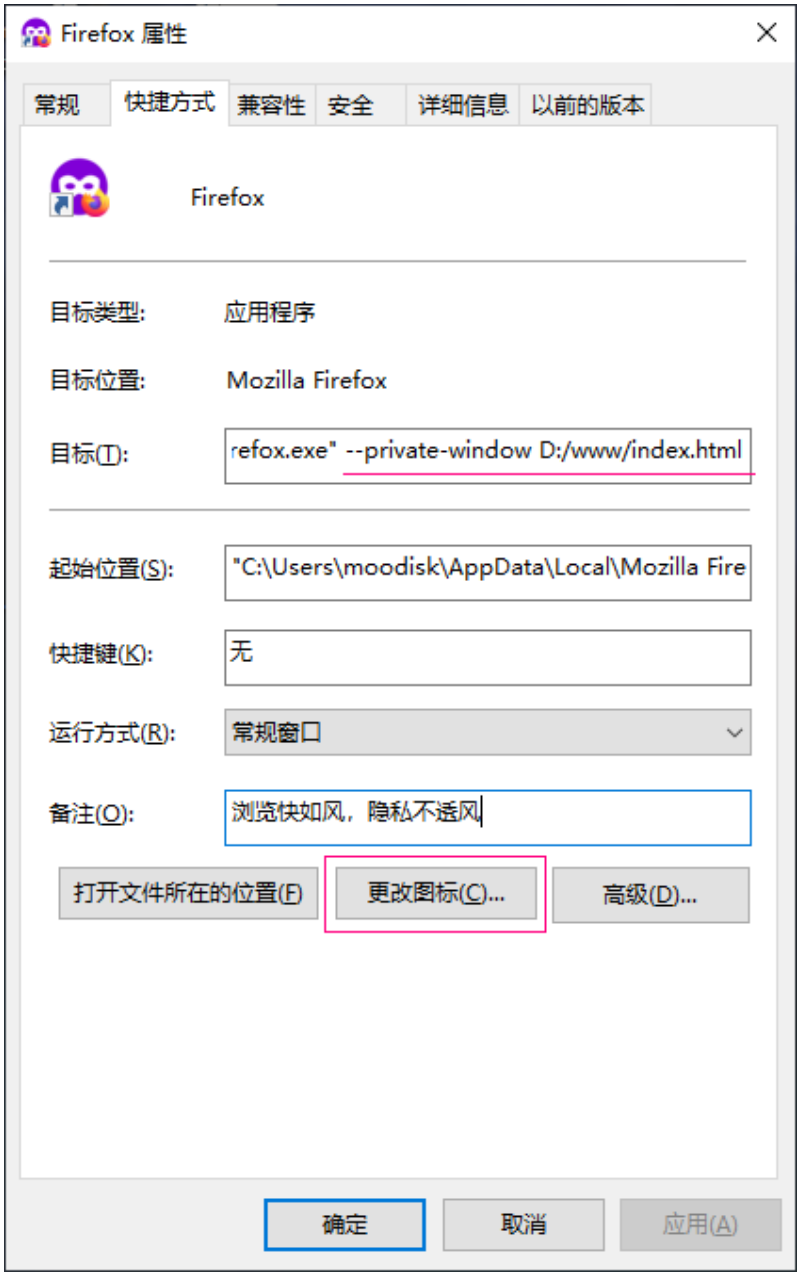


图 2-2 设置 firefox 隐私窗口

当然也可以启动浏览器后按 Ctrl+Shift+p 让浏览器进入隐私模式，或者参考图 2-3 所示操作。

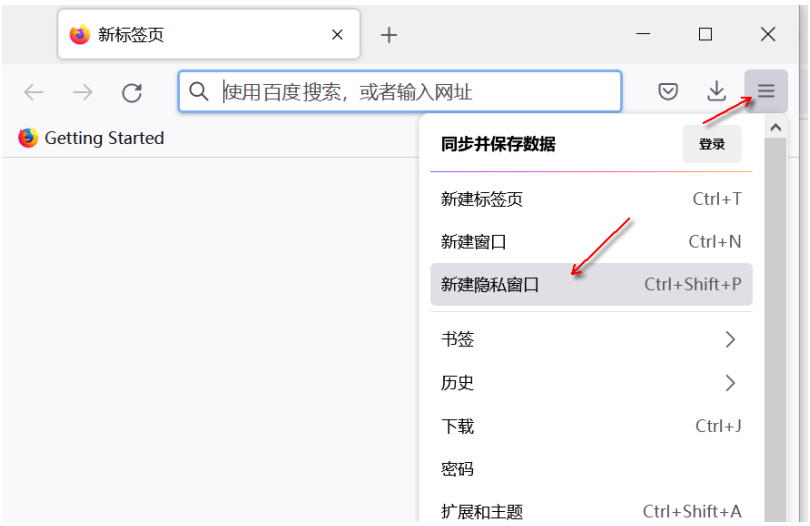


图 2-3 浏览器进入隐私模式

在隐私模式下，浏览器不会记录任何信息，当关闭浏览器后，一切上网痕迹不复存在！这对于交易加密货币至关重要，否则私钥就不是十分安全——有可能记录在缓存中！

让浏览器脱机工作，也就是让浏览器断网（切换到离线模式）。按 Alt 键显示菜单，点击左上角的“文件”中的“脱机工作”，确保“脱机工作”前方有一个“√”，这样浏览器工作在离线模式——相当于断网，从而保证你在接下来的操作过程中不会向网上泄露任何信息了，尽管电脑还是连同因特网。操作过程参考图 2-4 所示。

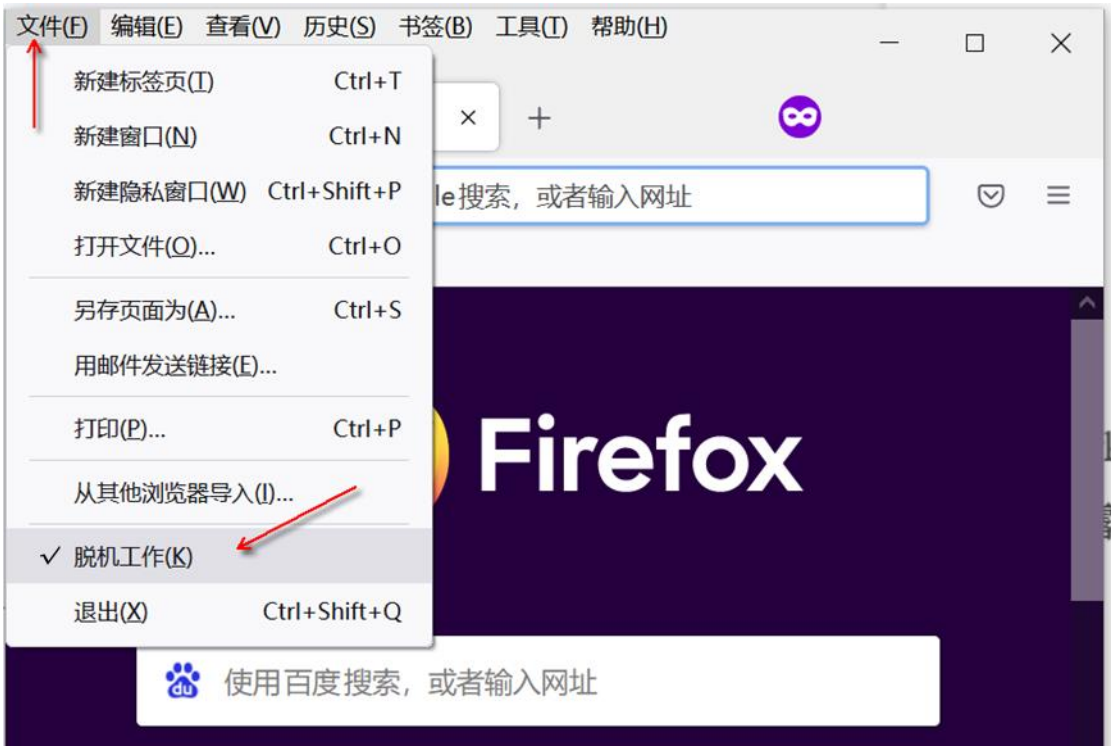


图 2-4 浏览器切换到离线模式

“脱机工作”前有一个“√”表示已经处于断网状态，此时浏览器不能打开因特网上的网站，这样签名交易时用到的私钥就不可能通过网络泄露出去。

按 F12 打开浏览器的开发者工具，并让界面停靠于底部，参考图 2-5 所示。



图 2-5 打开开发者工具

看到的浏览器界面如图 2-6 所示。

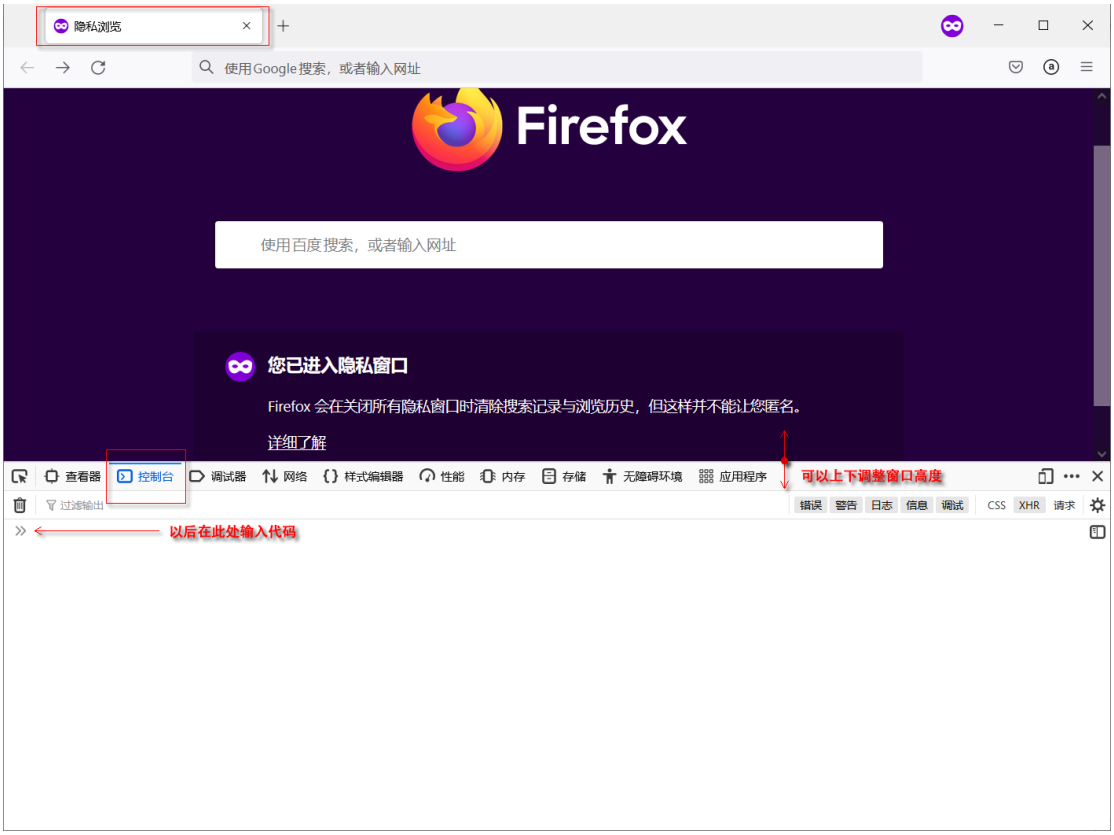


图 2-6 浏览器准备基本完成

不过入门操作环境不会用到开发者工具。

另外最好准备一个在线浏览器，专门用于上网，强烈建议采用谷歌浏览器 Chrome，科学上网后从 https://www.google.com/intl/en/chrome/?standalone=1 下载离线安装版本，或者直接在我的个人网站上有安装包供大家下载，

参见图 2-7。

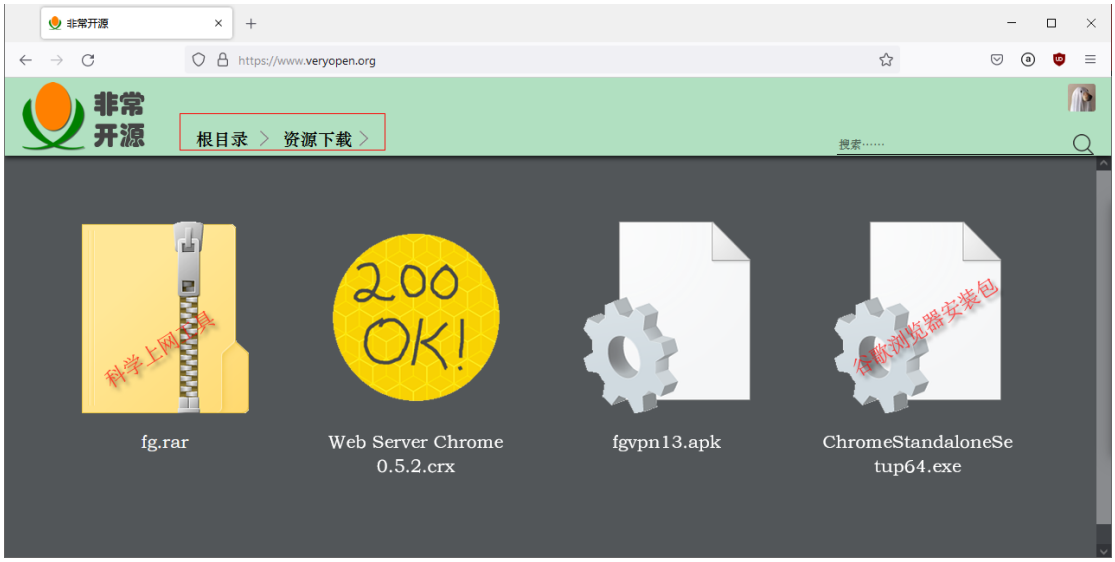


图 2-7 下载谷歌浏览器安装包

安装完毕后点击桌面上的 

图标启动 Chrome 浏览器，看到的效果图如图 2-8 所示。



图 2-8 谷歌浏览器

以后就在 firefox 浏览器中对加密币交易进行签名，在 Chrome 浏览器中发布交易。

1. 比特币网络

(1). 生产主网：

blockID=000000000000000000025f512a57921cea60fdb4e52eb8926da4bd67864b515c

0000000000000bae09a7a393a8acded75aa67e46cb81f7acaa5ad94f9eacd103

txid= c6e81c4a315d7eed40cb32b2558f4b142d37959f7e8eb3eb5c43fdf2931cca42 、4ac541802679866935a19d4f40728bb89204d0cac90d85f3a51a19278fe33aeb 、e6264ea63c267f1d54ae0125020745356eb054fdec09c51fcc403d9f0ce95ac2

address= 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa、bc1qd9k6d3jzaqsc3rvm039cae6y2v8sm2hlg6ucv6

发布交易：https://blockchair.com/broadcast 、https://live.blockcypher.com/btc/pushtx/ 、https://www.blockchain.com/btc/pushtx

解码原始交易：https://live.blockcypher.com/btc/decodetx/

获取交易的 hex 编码：https://blockstream.info/api/tx/{txid}/hex。直通。

获取交易：https://api.blockcypher.com/v1/btc/main/txs/{txid}。直通

获取 UTXO：https://api.blockcypher.com/v1/btc/main/addrs/{address}?unspentOnly=true 、https://blockstream.info/api/address/{address}/utxo、https://sochain.com/api/v2/get_tx_unspent/BTC/{address}。前两个直通。

获取地址余额：https://api.blockcypher.com/v1/btc/main/addrs/${address}/balance

获取区块原始数据：https://api.blockchair.com/bitcoin/raw/block/{blockID}（其中 raw_block 字段）、https://blockchain.info/rawblock/{blockID}?format=hex，两个不用梯子直通。

(2). 测试网：

txid= ffb32fbe3b0e260e38e82025d7dcf72a24feb3da455445015aaf8b8f9c9da68f

address= tb1qng6f35spexs5nr80enz3a76kuz5s9m20um22xx

获取钱包 UTXO：https://blockstream.info/testnet/api/address/{address}/utxo 、https://sochain.com/api/v2/get_tx_unspent/BTCTEST/{address} 、https://api.blockcypher.com/v1/btc/test3/addrs/{address}?unspentOnly=true

发布交易：https://mempool.space/testnet4/tx/push

解码原始交易：https://live.blockcypher.com/btc-testnet/decodetx/

获取交易的 hex 编码：https://blockstream.info/testnet/api/tx/{txid}/hex 、https://api.blockcypher.com/v1/btc/test3/txs/{txid}?includeHex=true、https://sochain.com/api/v2/get_tx_hex/BTCTEST/{txid}。前两个直通，第三个无效。

查看你钱包：https://live.blockcypher.com/btc-testnet/address/{adress}。无效。

2. 以太币网络

address=0xDeb5BF379E49300c78276167fBb8E3F18EA6C946、

(1). 生产主网：

获得 apiKey：https://developer.metamask.io/， 我的 apiKey: 5b34a68e44cf47c3afcd16c4a9b74341，(我的 MetaMask: lesson animal poverty pencil surround rough edge reveal exotic olympic hero doctor|2*Y)；https://etherscan.io/（moodisk/Hu*，token: NCGAQ33ESD34Y343Z4HHZHMT9D4DBFA9HK）。

Endpoint：https://mainnet.infura.io/v3/{apiKey}

获取钱包余额：https://api.etherscan.io/api?module=account&action=balance&address={address}&tag=latest&apikey={token}

(2). 测试网：

充值：https://cloud.google.com/application/web3/faucet/ethereum/sepolia

查询：https://sepolia.etherscan.io/

获得 apiKey：https://developer.metamask.io/

Endpoint：https://sepolia.infura.io/v3/{apiKey}。

2.2 科学上网

国内网络环境严酷，访问 p2p 加密币网络不稳定甚至根本没法访问，需要准备一把好梯子爬墙，也方便走暗网 Tor。

2.2.1 梯子

一、自搭梯子

建议租赁一台国外的云主机，我租的是谷歌云主机，最低配置加上网络流量，一个月费用不到 20 元人民币，谷歌云网站 https://cloud.google.com/，不过先要翻墙才能浏览此网站，糟糕，陷入先有鸡还是先有蛋的问题！需要先办理一张支持美元支付的信用卡，中行或农行的信用卡都行。

有了云主机后，在你本地电脑上运行命令“ssh -C -N -D 0.0.0.0:8080 用户@x.x.x.x”就可以建立 SOCKS5 类型的 VPN。

具体操作如下：

在国内搞 IT，经常要使用谷歌搜索最新的 IT 技术，这是可以自己搭建 ssh 隧道，前提是需要一台公网上的云主机（即 VPS），在云主机上安装 openssh-server 安全登录服务，它监听 22/tcp 号端口，一般此端口是可以直接从国内访问的，因为经常要远程登录云上主机。然后在家里的电脑上使用 ssh 命令开启动态端口，此端口叠加在 22 号端口之中——即“打洞”，请参考图 2-9。

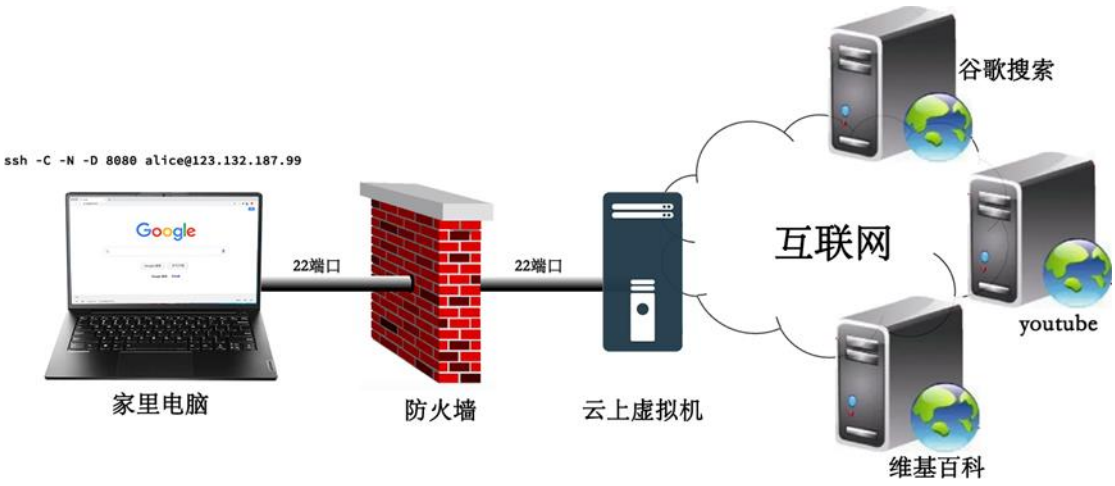


图 2-9 ssh 隧道示意图

假设家里的电脑是 windows 10，按 Win+R 键，在打开的窗口中输入 cmd 并回车打开一个黑底白字的命令窗口，然后输入命令：ssh -C -N -D 8080 alice@123.132.187.99，见图 2-10 所示。

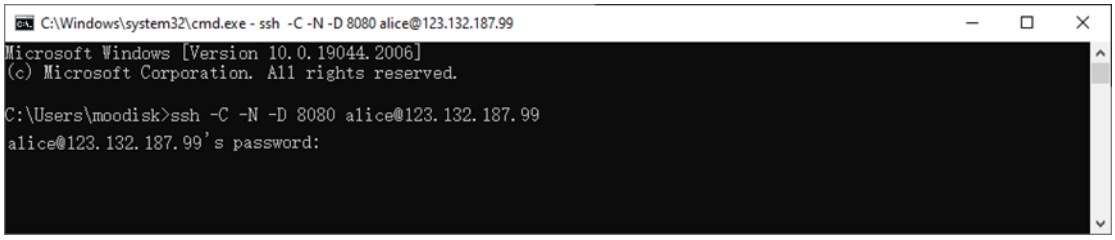


图 2-10 建立 ssh 隧道

假设云主机的 ip 为 123.132.187.99，已经创建了用户 alice，并设置了密码。这样就在本地启用了一个监听 8080 端口的进程，它就是一个类型为 socks5 的代理。如果要让 windows 成为以太 vpn 转发服务器，请使用命令 ssh -C -N -D 192.168.0.102:8080 alice@123.132.187.99，并且让 8080/tcp 端口通过 windows 的防火墙。为了免得每次竖起梯子都要输入密码，并增强安全性，建议采用公私钥对登陆，操作如下：

```
ssh-keygen -q -b 4096 -C "Socks5 VPN" -t rsa -f rsa_4096                    //产生一对公私钥
ssh-copy-id -i rsa_4096.pub alice@xx.yy.zzz.xyz                             //把公钥上传到 SSHD 服务器
```

以后就可以执行命令 ssh -C -N -D 0.0.0.0:8080 -i rsa_4096 alice@xx.yy.zzz.xyz 就可以竖起梯子了。

1. 针对 firefox 浏览器，设置代理上网的方法参见图 2-12、2-13 和 2-14。



图 2-12 firefox 设置代理（一）

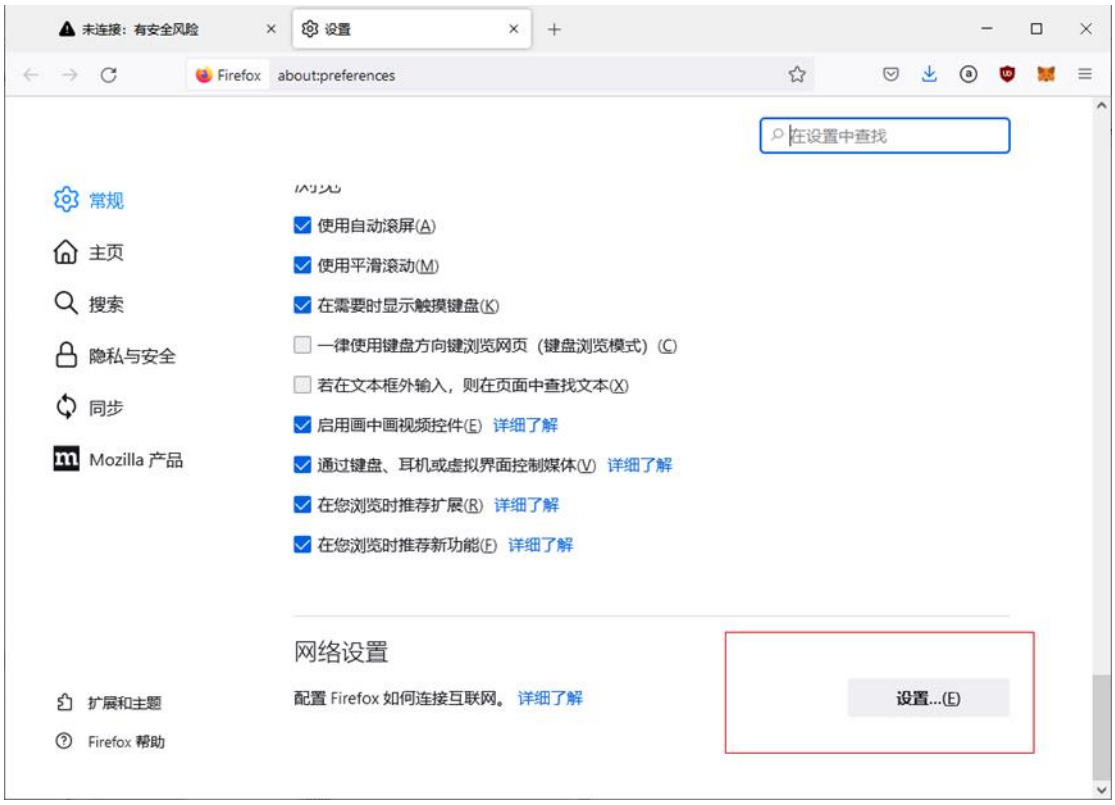


图 2-13 firefox 设置代理（二）



图 2-14 firefox 设置代理（三）

点击“确定”之后就可以打开谷歌搜索网站了。

2. 针对 Chrome 浏览器，设置 Socket5 代理，可这样启动：chrome.exe --user-data-dir="C:/Temp" --proxy-server="socks5://127.0.0.1:8080"。

3. 针对 Safari 浏览器：点击左上角菜单 “Safari 浏览器”»“系统偏好设置”»“网络”»“高级”»“代理”»“SOCKS 代理”：输入主机 127.0.0.1、端口 8080 » 点击“好”，完成设置。

二、使用自由门

从 https://dongtaiwang.com/loc/software/fg/801/fg801p.zip 下载自由门，然后解压出一个文件 fg801p.exe，双击它启动，做如图 2-15 所示的配置。

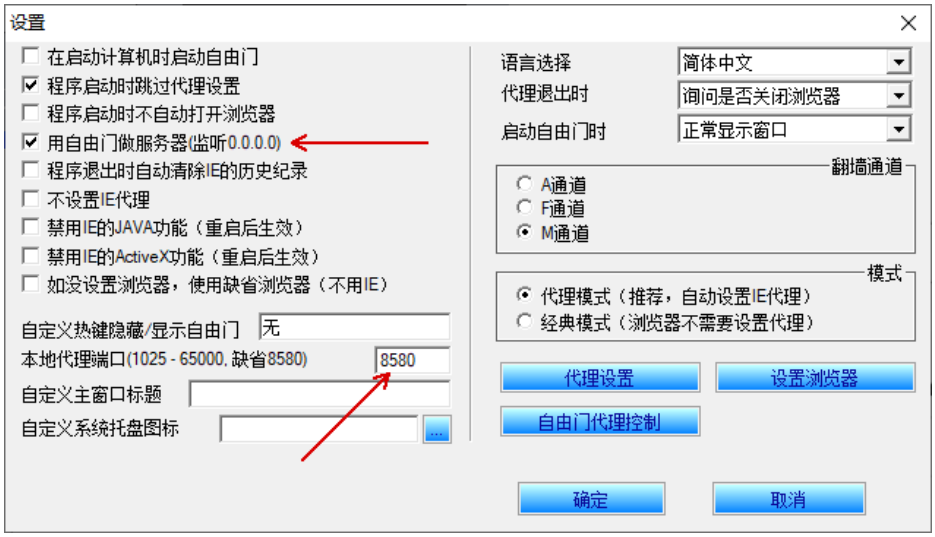


图 2-15 设置自由门本地代理

要让 8580/tcp 端口通过 windows 防火墙。

2.2.2 Tor 网



上世纪 90 年代，由美国军方资助开发了 Tor（The Onion Router，洋葱路由），目的是避免情报人员的在线活动被敌对国家监控，本世纪初开源，从此 Tor 成为了一个被军民广泛应用的匿名通信工具。Tor 是一种通过多层加密和若干匿名节点转发来实现网络匿名通信的技术，其核心设计目标是隐藏用户的身份和网络活动。

用户电脑上安装 Tor 客户端，客户端通过 Tor 目录服务器（Directory Server）随机获取三个节点（或更多），分别称为入口节点、中间节点（或多个）和出口节点，不妨分别假设为 A、B 和 C 节点。客户端先与 A 协商两个对称密钥 KeyA-f 和 KeyA-b，从而建立加密电路“客户端←→A”，接下来客户端与 B 协商（通过 A 转发）两个对称密钥 KeyB-f 和 KeyB-b，这样建立了加密电路“客户端←→A←→B”，最后客户端与 C 协商（通过 A、B 转发）两个对称密钥 KeyC-f 和 KeyC-b，这样建立了完整的加密电路“客户端←→A←→B←→C”，而客户端拥有三对对称密钥：(KeyA-f、KeyA-b)、(KeyB-f、KeyB-b)、(KeyC-f、KeyC-b)，用户要访问外网时（比如油管），客户端把发出去的每一个数据包进行三层加密，即 KeyA-f(KeyB-f(KeyC-f(data)))，然后发给 A，A 收到后解密外层得到 KeyB-f(KeyC-f(data))和 B 的 ip，并把数据发给 B，B 再解密外层得到 KeyC-f(data)和 C 的 ip，B 再发给 C，C 再解密外层得到 data 和目的服务器 ip，最后把 data 发给目的服务器。一层层解密像剥洋葱一样，这就是 Tor 这个名字的由来。目的服务器反馈的数据到达 C 后，C 加密成 KeyC-b(response)并发给 B，B 再加密一层为 Keyb-b(KeyC-b(response))并发给 A，A 也加密一层，即 KeyA-b(KeyB-b(KeyC-b(response)))最后发给客户端，客户端全部解密得到目的服务器反馈的数据 response。

所以出口节点 C 知道目的服务器和 B，也掌握 data 和 response，但不知道客户端和 A，如果 C 被“招降”，危险还是挺大的，所以我们即使走 Tor，也要加密 data 和 response，比如浏览 https 网站而不浏览 http 网站，使用 SSH 协议而不是 telnet 登陆远程电脑，使用 sftp 而不是 ftp 上传下载文件，客户端走 VPN 出去。每个节点只知道邻居节点而不掌握完整的路径，为了防止节点互相串谋，Tor 客户端每过十分钟就从新选择三个节点从而更改网络路径。Tor 由全球志愿者运营，无中心控制点，完全是去中心化的。

Tor 的原理请参考图 2-16 所示。

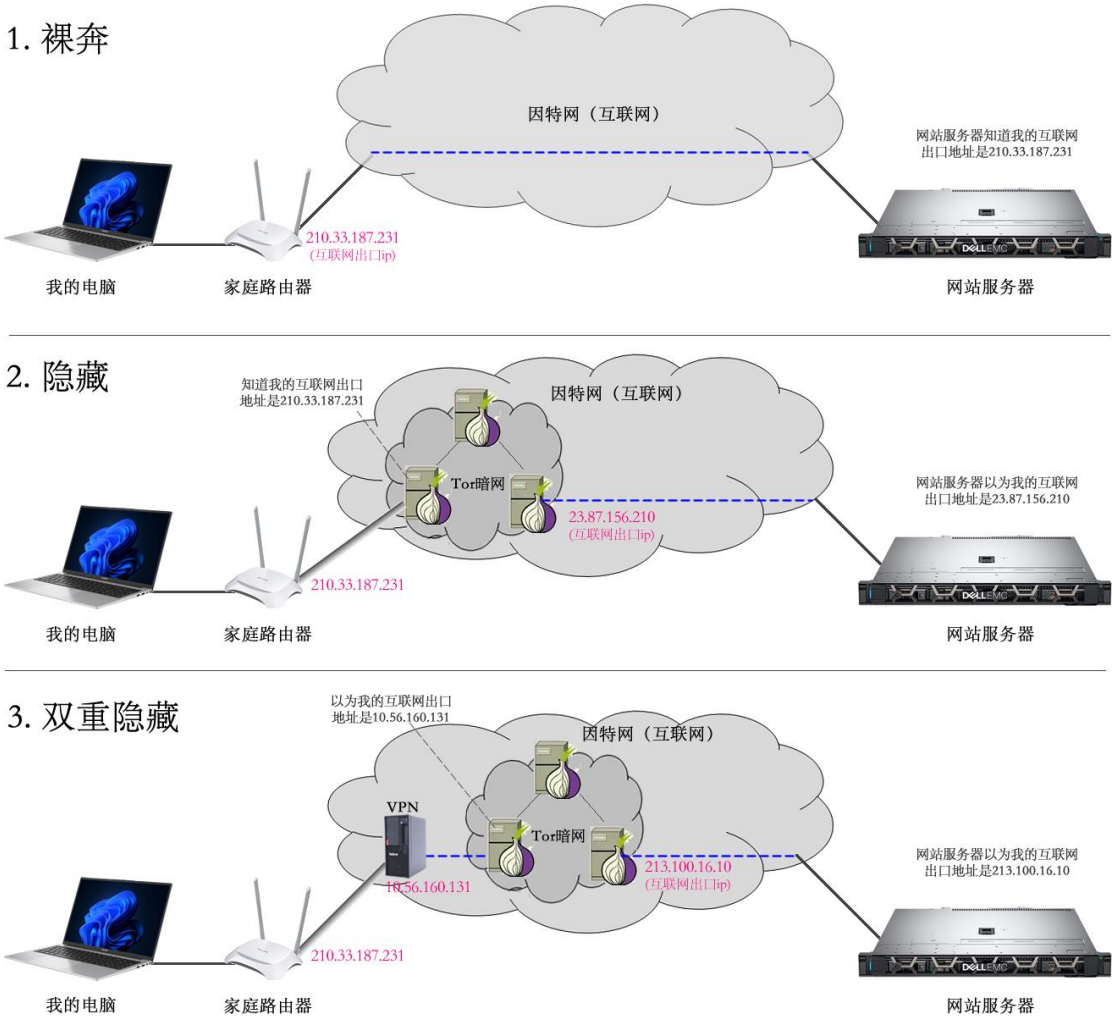


图 2-16 三种上网方式

图 2-16 中的第一种上网方式等于裸奔，家庭路由器的广域网口 ip 地址被电信部门登记在册，所以你上过什么网站政府全知道；第二种方式在中国行不通，不能直接接入 Tor，唯有第三种方式才可行，即先爬梯出去后再接入暗网，此时我的互联网出口 ip 地址不再是家里的路由器，而是分布在世界各地的 Tor 主机的 ip，而且我每次上网“通过”的 Tor 主机都不同，有时在德国，有时在美国，有时又在澳大利亚。中本聪就是通过 Tor 网使用因特网的，直到现在都不知道中本聪是人是鬼是男是女哪个国家的哪个人。

上面的描述只是隐藏了客户端（用户），如果连服务端都隐藏起来，那就叫暗网——双方的真实 Ip 和物理位置被完全隐藏起来，谁也不知道对方是谁。

暗网服务提供者首先生成非对称密钥对，即公私钥，然后将公钥哈希成 256 位二进制数并以 Base32 格式编码成 56 个字符组成的字符串(如 wasabiukrxmkdgve5kynjztuovbg43uxcbcxn6y2okcrsg7gb6jdmbad)，并组成网站域名(如 https://

wasabiukrxmkdgve5kynjztuovbg43uxcbcxn6y2okcrsg7gb6jdmbad.onion 或者 https://www.wasabiukrxmkdgve5kynjztuovbg43uxcbcxn6y2okcrsg7gb6jdmbad.onion)，最后通过 Tor 网络发布服务描述符（包含公钥和接入点信息）到分布式哈希表（DHT）。对于早期的版本，哈希后只取低位 16 个字符做域名，由于存在安全隐患，现在改用 54 个字符了。

客户端访问时通过“<公钥的哈希值>.onion”域名查询 DHT，获取服务描述符，然后开始访问服务。用户通过线下线上各种方式可获得某个服务的“公钥的哈希值”，比如我用谷歌搜索“Tor 暗网上的知名的服务”，结果有暗网邮箱：地址 https://protonmailrmez3lotccipshtkleegetolb73fuirgj7r4o4vfu7ozyd.onion，用途是加密电子邮件服务，以保护记者、活动人士通信隐私；电子书库：地址:http://xfmro77i3lixucja.onion，用途是免费下载数百万本电子书（类似影子图书馆）；混币服务： 地址 http://wasabiukrxmkdgve5kynjztuovbg43uxcbcxn6y2okcrsg7gb6jdmbad.onion，用途是比特币隐私混币工具（合法但可能被滥用）；Tor 的洋葱官网 http://2gzyxa5ihm7nsggfxnu52rck2vv4rvmdlkiu3zzui5du4xyclen53wid.onion。

知道域名后只要在 Tor 浏览器中打开网站即可，普通浏览器没法浏览暗网网站。至于如何搭建自己的暗网网站，请参考 https://dev.to/98lenvi/how-to-host-a-website-on-the-dark-web-56ah 网站。

Tor 官网给出的几个建议：（1）不要大流量使用 Tor，（2）不要在 Tor 中使用任何浏览器插件，（3）只浏览 HTTPS 协议的网站，（4）不要在线打开通过 Tor 下载的任何文档，（5）尽可能使用 Tor 桥。

Tor 的官方网站是 https://www.torproject.org/，可以爬梯打开此网站下载安装 Tor 浏览器或者 Tor 客户端。Tor 就是 socks5 代理，默认监听 127.0.0.1:9050，但是公开的 Tor 中继一般都被墙住了，所以需要再走网桥或者前置 vpn 出去。给 frontdesk@torproject.org 发一封空邮件，标题就是 private bridge cn，大概一天后会收到最新的网桥，类似：

```
obfs4 95.216.209.185:44535 11159547D80313031E0AEE719875C047282D1792
cert=qLgE6SMDD8ogWQRWhvxcTU0jAKuHNZPImc80fuFHFZiPqftGUytlMedD17UXwlZmI9oeMQ iat-mode=0
```

显然这是 obfs4 网桥（最常用的类型），它使用混淆技术，让流量看起来像普通 HTTPS 流量，避免被检测，完整格式是 obfs4 <IP>:<PORT> <FINGERPRINT> cert=<CERT> iat-mode=0

然后把它拷贝到 tor 浏览器的网桥配置里（打开 Tor Browser → 点击“配置连接” → 选择“使用网桥” → 输入网桥地址）。

对于 Tor 客户端，修改/etc/tor/torrc 文件：

```
……
#0-使用前置代理，1-使用网桥：
UseBridges 1

#前置 socks5 代理：
Socks5Proxy 127.0.0.1:8080

#网桥：
Bridge obfs4 95.216.209.185:44535 11159547D80313031E0AEE719875C047282D1792
cert=qLgE6SMDD8ogWQRWhvxcTU0jAKuHNZPImc80fuFHFZiPqftGUytlMedD17UXwlZmI9oeMQ iat-mode=0
……
```

然后重启 tor：systemctl restart tor.service

其他的类似技术还有 hyphanet（由 FreeNet 改名而来）、I2P 和 Zeronet。

2.3 安装 nginx

在本地启用一个 web 服务器，限制浏览器能访问的文件，是一个不错的注意。而 nginx 是小巧精悍的 web 服务器，对于 windows，请从这里 https://nginx.org/download/nginx-1.28.0.zip 下载，然后解压到 d 盘，参见图 2-17。

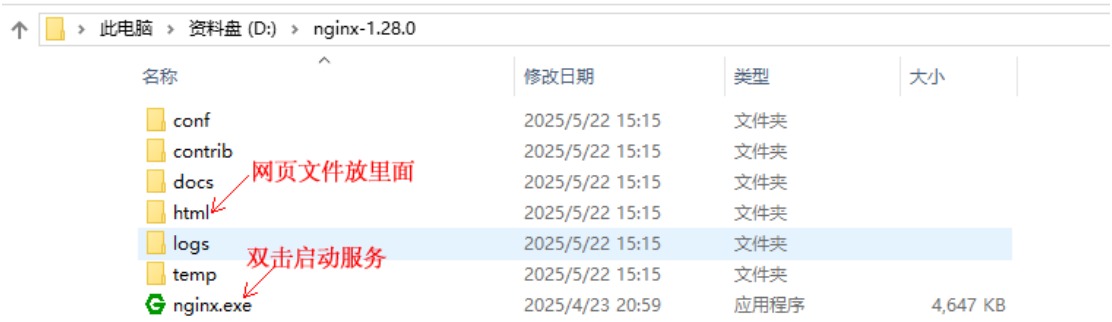


图 2-17 nginx 服务器目录结构

对于 Linux，直接安装 nginx 即可，不过页面文件目录在/var/www/html。

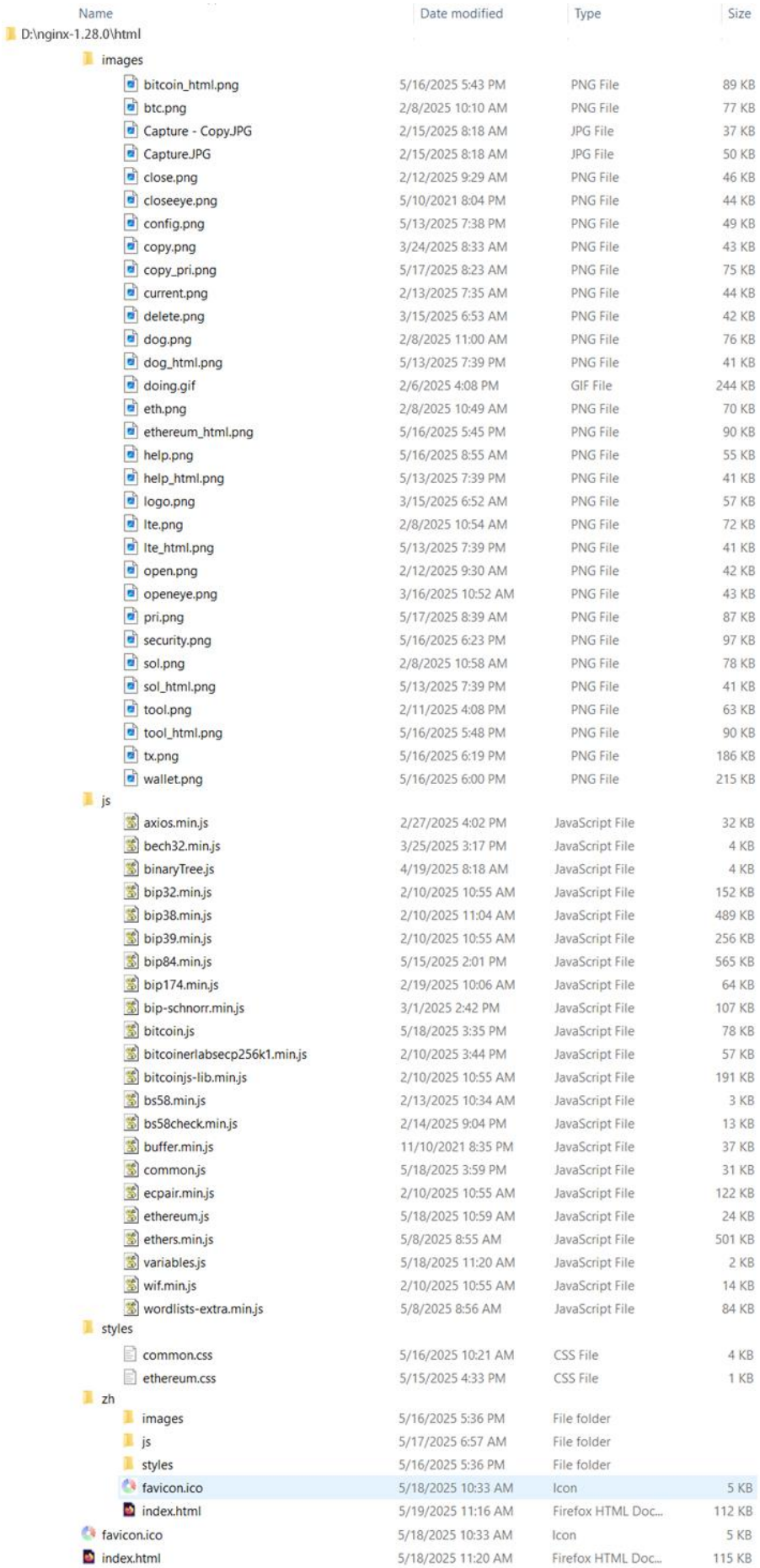
2.4 准备相关文件

需要用到几个 JavaScript 库，其中比特币需要 bitcore-lib 和 bitcoinjs-lib，以太币使用 web3 和 ether.js，另外还用到了 buffer 库。这些库都是开源的，可从网上下载，有的要自己编译。我已经把全部的库打包放在笔者的个人网站上，大家可以直接从那里下载使用。

下面分别讲讲这两种方法。

2.4.1 方法 1：从网站下载

从 https://www.veryopen.org 或者 https://www.walletoffline.com 网站上下载“加密币离线包”文件，然后解压缩后把 www 目录中的文件剪贴到 D:\nginx-1.28.0\html 目录里，最终看到的文件如图 2-18 所示。



Name	Date modified	Type	Size
D:\nginx-1.28.0\html			
images			
bitcoin_html.png	5/16/2025 5:43 PM	PNG File	89 KB
btc.png	2/8/2025 10:10 AM	PNG File	77 KB
Capture - Copy.JPG	2/15/2025 8:18 AM	JPG File	37 KB
Capture.JPG	2/15/2025 8:18 AM	JPG File	50 KB
close.png	2/12/2025 9:29 AM	PNG File	46 KB
closeeye.png	5/10/2021 8:04 PM	PNG File	44 KB
config.png	5/13/2025 7:38 PM	PNG File	49 KB
copy.png	3/24/2025 8:33 AM	PNG File	43 KB
copy_pri.png	5/17/2025 8:23 AM	PNG File	75 KB
current.png	2/13/2025 7:35 AM	PNG File	44 KB
delete.png	3/15/2025 6:53 AM	PNG File	42 KB
dog.png	2/8/2025 11:00 AM	PNG File	76 KB
dog_html.png	5/13/2025 7:39 PM	PNG File	41 KB
doing.gif	2/6/2025 4:08 PM	GIF File	244 KB
eth.png	2/8/2025 10:49 AM	PNG File	70 KB
ethereum_html.png	5/16/2025 5:45 PM	PNG File	90 KB
help.png	5/16/2025 8:55 AM	PNG File	55 KB
help_html.png	5/13/2025 7:39 PM	PNG File	41 KB
logo.png	3/15/2025 6:52 AM	PNG File	57 KB
lte.png	2/8/2025 10:54 AM	PNG File	72 KB
lte_html.png	5/13/2025 7:39 PM	PNG File	41 KB
open.png	2/12/2025 9:30 AM	PNG File	42 KB
openeye.png	3/16/2025 10:52 AM	PNG File	43 KB
pri.png	5/17/2025 8:39 AM	PNG File	87 KB
security.png	5/16/2025 6:23 PM	PNG File	97 KB
sol.png	2/8/2025 10:58 AM	PNG File	78 KB
sol_html.png	5/13/2025 7:39 PM	PNG File	41 KB
tool.png	2/11/2025 4:08 PM	PNG File	63 KB
tool_html.png	5/16/2025 5:48 PM	PNG File	90 KB
tx.png	5/16/2025 6:19 PM	PNG File	186 KB
wallet.png	5/16/2025 6:00 PM	PNG File	215 KB
js			
axios.min.js	2/27/2025 4:02 PM	JavaScript File	32 KB
bech32.min.js	3/25/2025 3:17 PM	JavaScript File	4 KB
binaryTree.js	4/19/2025 8:18 AM	JavaScript File	4 KB
bip32.min.js	2/10/2025 10:55 AM	JavaScript File	152 KB
bip38.min.js	2/10/2025 11:04 AM	JavaScript File	489 KB
bip39.min.js	2/10/2025 10:55 AM	JavaScript File	256 KB
bip84.min.js	5/15/2025 2:01 PM	JavaScript File	565 KB
bip174.min.js	2/19/2025 10:06 AM	JavaScript File	64 KB
bip-schnorr.min.js	3/1/2025 2:42 PM	JavaScript File	107 KB
bitcoin.js	5/18/2025 3:35 PM	JavaScript File	78 KB
bitcoinerlabsecp256k1.min.js	2/10/2025 3:44 PM	JavaScript File	57 KB
bitcoinjs-lib.min.js	2/10/2025 10:55 AM	JavaScript File	191 KB
bs58.min.js	2/13/2025 10:34 AM	JavaScript File	3 KB
bs58check.min.js	2/14/2025 9:04 PM	JavaScript File	13 KB
buffer.min.js	11/10/2021 8:35 PM	JavaScript File	37 KB
common.js	5/18/2025 3:59 PM	JavaScript File	31 KB
ecpair.min.js	2/10/2025 10:55 AM	JavaScript File	122 KB
ethereum.js	5/18/2025 10:59 AM	JavaScript File	24 KB
ethers.min.js	5/8/2025 8:55 AM	JavaScript File	501 KB
variables.js	5/18/2025 11:20 AM	JavaScript File	2 KB
wif.min.js	2/10/2025 10:55 AM	JavaScript File	14 KB
wordlists-extra.min.js	5/8/2025 8:56 AM	JavaScript File	84 KB
styles			
common.css	5/16/2025 10:21 AM	CSS File	4 KB
ethereum.css	5/15/2025 4:33 PM	CSS File	1 KB
zh			
images	5/16/2025 5:36 PM	File folder	
js	5/17/2025 6:57 AM	File folder	
styles	5/16/2025 5:36 PM	File folder	
favicon.ico	5/18/2025 10:33 AM	Icon	5 KB
index.html	5/19/2025 11:16 AM	Firefox HTML Doc...	112 KB
favicon.ico	5/18/2025 10:33 AM	Icon	5 KB
index.html	5/18/2025 11:20 AM	Firefox HTML Doc...	115 KB

图 2-18 准备的全部文件

其中 ethers.umd.min.js 和 wordlists-extra.umd.min.js 是处理以太坊的 js 库，可去 https://github.com/ethers-io/ethers.js/tree/main/dist 下载最新版本。zh 目录的简体中文页面，文件类型和个数与英文版一样。

然后直接用浏览器打开网址 localhost，最后看到的结果如图 2-19 所示。

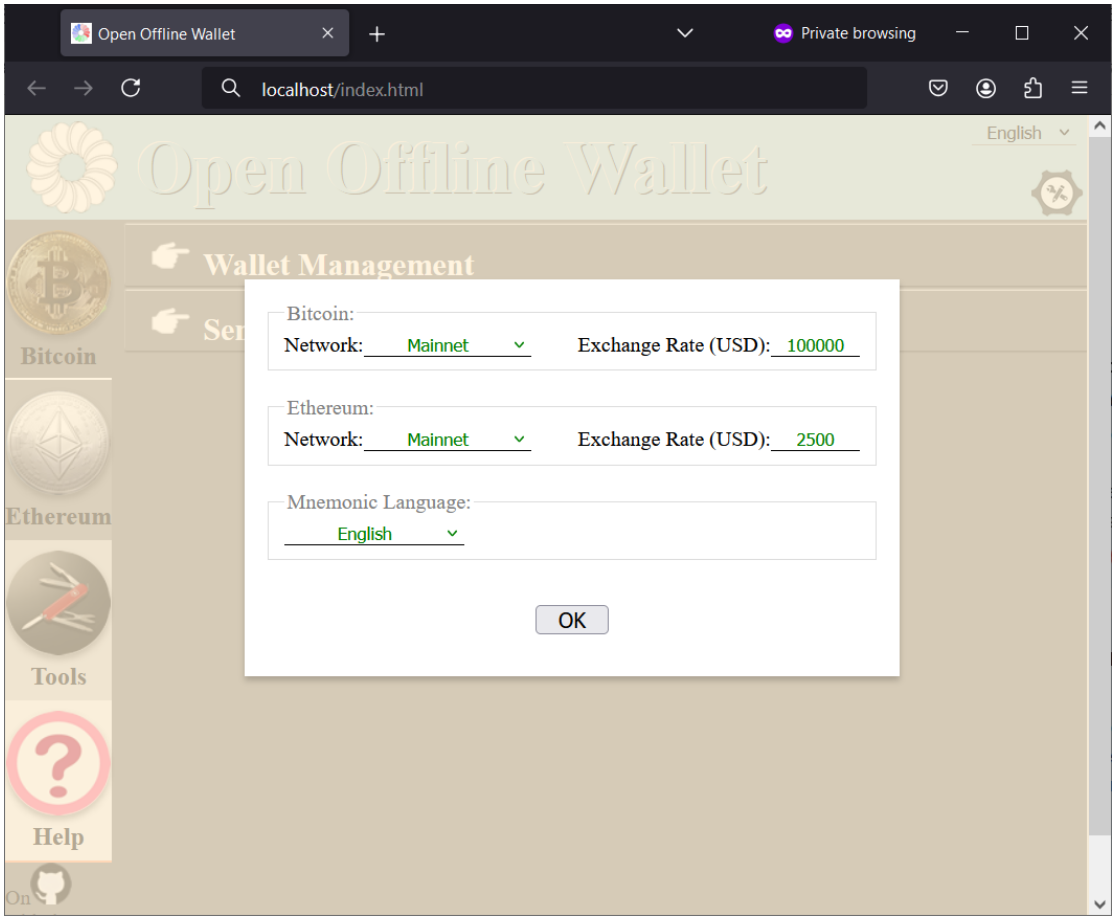


图 2-19 交易实战主页

上图中选择正确的比特币网络，并输入当时的比特币汇率，比如 2025 年 5 月 18 日星期五 1 比特币可兑换 100000 美元。

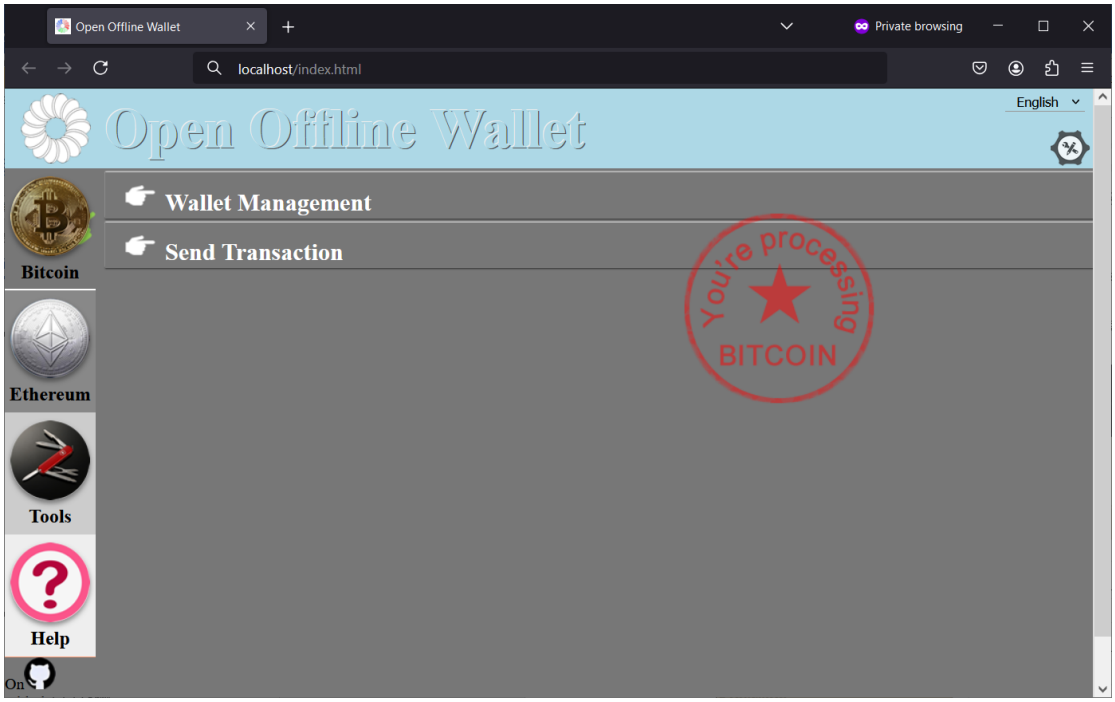


图 2-20 比特币实战主页面

2.4.2 方法 2：自己编译

如果你不太相信我个人网站上的 js 库，你也可以自己从其他地方下载或者编译。接下来介绍具体的过程，在 windows 或 linux 上都可以编译，准备工作你只要选择一个即可，即要么 windows 要么 linux。最后把你自己编译号的 js 库替换上一小节中的 js 库即可。

1. 准备工作

针对 Windows 的准备工作

从 https://nodejs.org/dist/v22.14.0/node-v22.14.0-x64.msi 下载 nodejs 的安装包，然后点击它安装，一路点击“下一步”即可正常安装。

然后按 win+r 后输入命令 cmd 回车打开一个黑白字符界面的命令窗口，参考图 2-21 所示。



图 2-21 windows 的命令窗口

然后输入如下命令：

```
md bitcoinjs-lib
cd bitcoinjs-lib
npm init -y
npm install bitcoinjs-lib tronweb
notepad package.json
{
  "dependencies": {
    "@bitcoinerlab/secp256k1": "^1.2.0",
    "bitcoinjs-lib": "^6.1.7",
    "bech32": "^2.0.0",
    "bip174": "^2.1.1",
    "bip32": "^4.0.0",
    "bip38": "^3.1.1",
    "bip39": "^3.1.0",
    "bip44": "^0.0.1",
    "bip84": "^0.2.9",
    "buffer": "^6.0.3",
    "ecpair": "^2.1.0",
    "tiny-secp256k1": "^2.2.3",
    "tronweb": "^6.0.4",                                                    //波场
    "wif": "^5.0.0",
    "bip-schnorr": "^0.6.7",
    "browserify": "^17.0.1",
    "uglify-js": "^3.19.3"
  }
}
```

针对 Linux 的准备工作

从 https://nodejs.org/dist/v22.13.1/node-v22.13.1-linux-x64.tar.xz 下载，然后解压到指定目录即可：

```
user@btceth:~$ wget https://nodejs.org/dist/v22.13.1/node-v22.13.1-linux-x64.tar.xz
user@btceth:~$ sudo tar -xf node-v22.13.1-linux-x64.tar.xz -C /usr/local/
user@btceth:~$ echo 'export PATH=$PATH:/usr/local/node-v22.13.1-linux-x64/bin/' >> ~/.bashrc
user@btceth:~$ . ~/.bashrc
user@btceth:~$ node -v
v22.13.0
```

然后输入如下命令：

```
user@btceth:~$ mkdir bitcoinjs-lib
user@btceth:~$ cd bitcoinjs-lib
user@btceth:~/bitcoinjs-lib$ npm init -y
user@btceth:~/bitcoinjs-lib$ npm install bitcoinjs-lib
user@btceth:~/bitcoinjs-lib$ vim package.json                              #2025 年 2 月 10 日的版本
{
  "dependencies": {
    "@bitcoinerlab/secp256k1": "^1.2.0",
    "bitcoinjs-lib": "^6.1.7",
```

```
    "bitcoinjs-message@2.2.0",
    "bech32": "^2.0.0",
    "bip174": "^2.1.1",
    "bip32": "^4.0.0",
    "bip38": "^3.1.1",
    "bip39": "^3.1.0",
    "bip44": "^0.0.1",
    "bip84": "^0.2.9",
    "buffer": "^6.0.3",
    "ecpair": "^2.1.0",
    "tiny-secp256k1": "^2.2.3",
    "tronweb": "^6.0.4",
    "wif": "^5.0.0",
    "bip-schnorr": "^0.6.7",
    "browserify": "^17.0.1",
    "uglify-js": "^3.19.3"
  }
}
```

文件 package.json 种列举了各种依赖模块和相应的版本号，其中 browserify 模块用来编译 js 库，uglify-js 用来压缩 js 库。

2. 安装这些依赖模块

```
npm install                                                    #安装依赖包
npm ls                                                         #查看已经安装的包和版本
bitcoinjs-lib@ C:\Users\moodisk\bitcoinjs-lib
+-- @bitcoinerlab/secp256k1@1.2.0
+-- bech32@2.0.0
+-- bip-schnorr@0.6.7
+-- bip174@2.1.1
+-- bip32@4.0.0
+-- bip38@3.1.1
+-- bip39@3.1.0
+-- bip44@0.0.1
+-- bip84@0.2.9
+-- bitcoinjs-lib@6.1.7
+-- bitcoinjs-message@2.2.0
+-- browserify@17.0.1
+-- buffer@6.0.3
+-- ecpair@2.1.0
+-- tiny-secp256k1@2.2.3
+-- tronweb@6.0.4
+-- uglify-js@3.19.3
`-- wif@5.0.0
```

如果要单独安装某个模块（如 xyz）的特定版本（如 2.3.4），请使用命令 npm install xyz@2.3.4，要查看一个模块（如 xyz）的全部版本，可使用命令 npm view xyz versions，或者在 https://www.npmjs.com/网站上搜索查看各个版本。安装模块时如果省略版本，那么表示安装最新版本，但是最新版本可能是开发板，开发板的 bug 比较多，建议不要用。

接下来就可以在 nodejs 中使用各个库了，例如从一个私钥导出比特币各种类型的地址：

```
user@btceth:~/bitcoinjs-lib$ node
Welcome to Node.js v22.12.0.
Type ".help" for more information.
> const bitcoin = require('bitcoinjs-lib');
undefined
> const ecc = require('tiny-secp256k1');
undefined
> const { ECPairFactory } = require('ecpair');
undefined
> const ECPair = ECPairFactory(ecc);
undefined
> const network = bitcoin.networks.bitcoin;
undefined
> const wif = 'L4AgDeoTPbdRd4S7X9UhXaNtyxZhDuj7EoUxPUndDYkvfmHjR8Cd';
undefined
> const keyPair = ECPair.fromWIF(wif);
undefined
> const { address: address_p2wpkh } = bitcoin.payments.p2wpkh({ pubkey: keyPair.publicKey, network });
undefined
> const { address: address_p2pkh } = bitcoin.payments.p2pkh({ pubkey: keyPair.publicKey, network });
```

```
undefined
> const { address: address_p2sh } = bitcoin.payments.p2sh({redeem: bitcoin.payments.p2wpkh({ pubkey:
keyPair.publicKey }), network });
undefined
> const toXOnly = pubKey => (pubKey.length === 32 ? pubKey : pubKey.slice(1, 33));
undefined
> bitcoin.initEccLib(ecc);
undefined
> const { address: address_p2tr } = bitcoin.payments.p2tr({ internalPubkey:
Buffer.from(toXOnly(keyPair.publicKey)), network: network });
undefined
> console.log(`从私钥${wif}导出的各类比特币地址: \nP2PKH 地址: ${address_p2pkh}\nP2SH 地址:
${address_p2sh}\nP2WPKH 地址: ${address_p2wpkh}\nP2TR 地址: ${address_p2tr}`);
从私钥 L4AgDeoTPbdRd4S7X9UhXaNtyxZhDuj7EoUxPUndDYkvfmHjR8Cd 导出的各类比特币地址:
P2PKH 地址: 1QKaUAoU2h9HUFyBpxFwrSZ4X4n4X2Xukp
P2SH 地址: 35SbZDs5biwdSFFUTR5GTQSH2fkDX2GqQ8
P2WPKH 地址: bc1qll89rqfmzllnyd9cnzy8vex9jueqv86y6587wg
P2TR 地址: bc1pgp2szey33f8eaqxd5hlulhmhd6dh8fz90zdgftr0var5jcs4vypsx5u8f6
undefined
>^d                                                                    #按 Ctrl+d 推出 nodejs
```

3. 编译成 js 库

下面我们把各种库编译成浏览器能使用的 js 库。

（1）编译 bitcoinjs-lib 库：

```
notepad main.js                                                              //Windows 上执行
vim main.js                                                    //Linux 上执行。输入如下内容
const bitcoin = require('bitcoinjs-lib');
window.bitcoin = bitcoin;                                    // 使其在全局可用
global.Buffer = require('buffer').Buffer;
```

（2）编译其他库：

bip84 的方法 fromMnemonic()默认不支持多中语言的助记词，需要修改：

```
vim node_modules/bip84/src/index.js
……
function fromMnemonic(mnemonic, password, isTestnet, coinType, pubTypes, network, wordlist) {
  if (!bip39.validateMnemonic(mnemonic, wordlist)) throw new Error('could not validate mnemonic words')
……
```

然后编译：

```
npx browserify -r ecpair -s ecpair > ecpair.js
npx browserify -r bip32 -s bip32 > bip32.js
npx browserify -r bip39 -s bip39 > bip39.js
npx browserify -r bip38 -s bip38 > bip38.js
npx browserify -r bip84 -s bip84 > bip84.js
npx browserify -r tiny-secp256k1 -s tiny-secp256k1 > tiny-secp256k1.js
npx browserify -r wif -s wif > wif.js
npx browserify -r bs58check -s bs58check > bs58check.js
npx browserify -r bitcoinjs-message -s bitcoinjs-message > bitcoinjs-message.js
npx browserify -r bs58 -s bs58 > bs58.js
npx browserify -r bip-schnorr -s bip-schnorr > bip-schnorr.js
```

波场 js 库不用单独编译，直接拷贝 node_modules/tronweb/dist/TronWeb.js 即可。

bip39 为助记词 js 库，里面包含了很多国家的语言，一般我们只用英语单词作为助记词，因此编译时可排除不用的语言，这样编译出来的 js 库就要小得多，参考下面的命令：

```
browserify -r bip39 -s bip39 \
 --exclude=./wordlists/japanese.json \
 --exclude=./wordlists/spanish.json \
 --exclude=./wordlists/italian.json \
```

```
 --exclude=./wordlists/french.json \
 --exclude=./wordlists/korean.json \
 --exclude=./wordlists/czech.json \
 --exclude=./wordlists/portuguese.json \
 --exclude=./wordlists/chinese_simplified.json \
 --exclude=./wordlists/chinese_traditional.json \
  > bip39.js
```

非压缩 js 库太大，需要压缩才会加快浏览器加载速度：

```
npm i uglify-js                                                    #安装压缩打包工具
npx uglify-js bip32.js -m -o bip32.min.js
npx uglify-js bip38.js -m -o bip38.min.js
npx uglify-js bip39.js -m -o bip39.min.js
npx uglify-js bip84.js -m -o bip84.min.js
npx uglify-js bitcoinjs-lib.js -m -o bitcoinjs-lib.min.js
npx uglify-js bitcoinjs-message.js -m -o bitcoinjs-message.min.js
npx uglify-js ecpair.js -m -o ecpair.min.js
npx uglify-js tiny-secp256k1.js -m -o tiny-secp256k1.min.js
npx uglify-js wif.js -m -o wif.min.js
npx uglify-js bs58.js -m -o bs58.min.js
npx uglify-js bs58check.js -m -o bs58check.min.js
npx uglify-js bip-schnorr.js -m -o bip-schnorr.min.js
ll *.min.js                          //Linux 上执行
dir *.min.js                         //Windows 上执行
-rw-r--r-- 1 wordpress wordpress 155503  2 月 10 日 10:49 bip32.min.js
-rw-r--r-- 1 wordpress wordpress 499911  2 月 10 日 11:02 bip38.min.js
-rw-r--r-- 1 wordpress wordpress 262115  2 月 10 日 10:48 bip39.min.js
-rw-r--r-- 1 wordpress wordpress 577394  2 月 10 日 10:48 bip84.min.js
-rw-r--r-- 1 wordpress wordpress 195207  2 月 10 日 10:37 bitcoinjs-lib.min.js
-rw-r--r-- 1 wordpress wordpress  13179  2 月 14 日 20:54 bs58check.min.js
-rw-r--r-- 1 wordpress wordpress   2915  2 月 13 日 10:33 bs58.min.js
-rw-r--r-- 1 wordpress wordpress 124281  2 月 10 日 10:39 ecpair.min.js
-rw-r--r-- 1 wordpress wordpress 444633  2 月 10 日 10:49 tiny-secp256k1.min.js
-rw-r--r-- 1 wordpress wordpress  14127  2 月 10 日 10:26 wif.min.js
```

是不是小了很多 。接下来把压缩后的 js 库拷贝到 www/js/目录下。不过在浏览器中测试时发现 tiny-secp256k1 存在 bug，于是编译另一个替代模块@bitcoinerlab/secp256k1 就没问题：

```
npm i @bitcoinerlab/secp256k1
npx browserify -r @bitcoinerlab/secp256k1 -s @bitcoinerlab/secp256k1 > bitcoinerlabsecp256k1.js
npx uglify-js bitcoinerlabsecp256k1.js -m -o bitcoinerlabsecp256k1.min.js
```

tiny-secp256k1 保留，在 nodejs 中建议还是使用它，等以后的新版本看看会不会修复这个 bug，如果要卸掉它可采用命令 npm uninstall tiny-secp256k1。

@bitauth/libauth 库功能强大，我们暂时用到它的比特币脚本虚拟机：

```
npm install @bitauth/libauth
```

然后在 nodejs 中导入：

```
const libauth = await import("@bitauth/libauth");
//或者:
const { createVirtualMachine } = await import("@bitauth/libauth");
```

最终的 package.json 文件内容如下：

```
wordpress@wordpress:~/bitcoinjs-lib$ cat package.json
{
  "dependencies": {
    "@bitauth/libauth": "^3.0.0",
    "@bitcoinerlab/secp256k1": "^1.2.0",
    "@noble/secp256k1": "^2.2.3",
    "axios": "^1.8.1",
    "bech32": "^2.0.0",
    "bip-schnorr": "^0.6.7",
    "bip174": "^2.1.1",
```

```
    "bip32": "^4.0.0",
    "bip38": "^3.1.1",
    "bip39": "^3.1.0",
    "bip44": "^0.0.1",
    "bip68": "^1.0.4",
    "bip84": "^0.2.9",
    "bitcoinjs-lib": "^6.1.7",
    "bitcoinjs-message": "^2.2.0",
    "browserify": "^17.0.1",
    "bs58check": "^4.0.0",
    "ecpair": "^2.1.0",
    "ethers": "^6.13.7",
    "merkle-lib": "^2.0.10",
    "secp256k1": "^5.0.1",
    "tiny-secp256k1": "^2.2.3",
    "tronweb": "^6.0.4",
    "uglify-js": "^3.19.3",
    "wif": "^5.0.0"
  }
}
```

去 https://www.cdnpkg.com/axios 下载最新版的 axios。截止到 2025 年 5 月 19 日星期六，最新版本为 1.4，可直接从 https://unpkg.com/axios@1.4.0/dist/axios.min.js 下载。

4. 在浏览器中使用

在 html 页面文件中使用这些 js 库，例如 index.html 文件内容如下：

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>从一个 wif 产生比特币各类地址</title>

    <script src="js/bitcoinjs-lib.min.js"></script><!--导入 bitcoin 对象-->
    <script src="js/bip32.min.js"></script><!--导入 bip32 对象-->
    <script src="js/bip38.min.js"></script><!--导入 bip38 对象-->
    <script src="js/bip39.min.js"></script><!--导入 bip39 对象-->
    <script src="js/bip84.min.js"></script><!--导入 bip84 对象-->
    <script src="js/wif.min.js"></script><!--导入 wif 对象-->
    <script src="js/ecpair.min.js"></script><!--导入 ecpair 对象-->
    <script src="js/bitcoinerlabsecp256k1.min.js"></script><!--导入 bitcoinerlabsecp256k1 对象-->
    <script src="js/bs58.min.js"></script><!--导入 bs58 对象-->
    <script src="js/bs58check.min.js"></script><!--导入 bs58check 对象-->
    <script src="js/axios.min.js"></script><!--导入 axios 对象，用于网络资源请求。-->
    <script src="js/bip-schnorr.min.js"></script><!--导入 bipSchnorr 对象，用于 Schnorr 签名。-->
    <script src="js/buffer.min.js"></script><!--Import Buffer object-->
    <script src="js/openpgp.min.js"></script><!--导入 openpgp 对象-->
    <script src="js/bitcoinjs-message.min.js"></script><!--导入 bitcoinjsMessage 对象-->
</head>

<body>
    <div style="line-height: 2rem;"></div>
    <script>
        const wif = 'L4AgDeoTPbdRd4S7X9UhXaNtyxZhDuj7EoUxPUndDYkvfmHjR8Cd';
        const network = bitcoin.networks.bitcoin;
        const ECPair = ecpair.ECPairFactory(bitcoinerlabsecp256k1);
        const keyPair = ECPair.fromWIF(wif);
        const { address: address_p2pkh } = bitcoin.payments.p2pkh({ pubkey: keyPair.publicKey });
        const { address: address_p2wpkh } = bitcoin.payments.p2wpkh({ pubkey: keyPair.publicKey });
        const { address: address_p2sh } = bitcoin.payments.p2sh({ redeem: bitcoin.payments.p2wpkh({ pubkey:
keyPair.publicKey })});

        const toXOnly = pubKey => (pubKey.length === 32 ? pubKey : pubKey.slice(1, 33));
        bitcoin.initEccLib(bitcoinerlabsecp256k1);
```

```
      const { address: address_p2tr } = bitcoin.payments.p2tr({ internalPubkey:
Buffer.Buffer.from(toXOnly(keyPair.publicKey)), network: network });

      document.getElementsByTagName('div')[0].innerHTML = `
      从私钥${wif}导出的各类比特币地址：<br>    P2PKH 地址：${address_p2pkh}<br>    P2SH 地址：${address_p2sh}<br>
         P2WPKH 地址：${address_p2wpkh}<br>    P2TR 地址：${address_p2tr}`;
   </script>
</body>

</html>
```

在浏览器中看到的效果如图 2.22 所示。

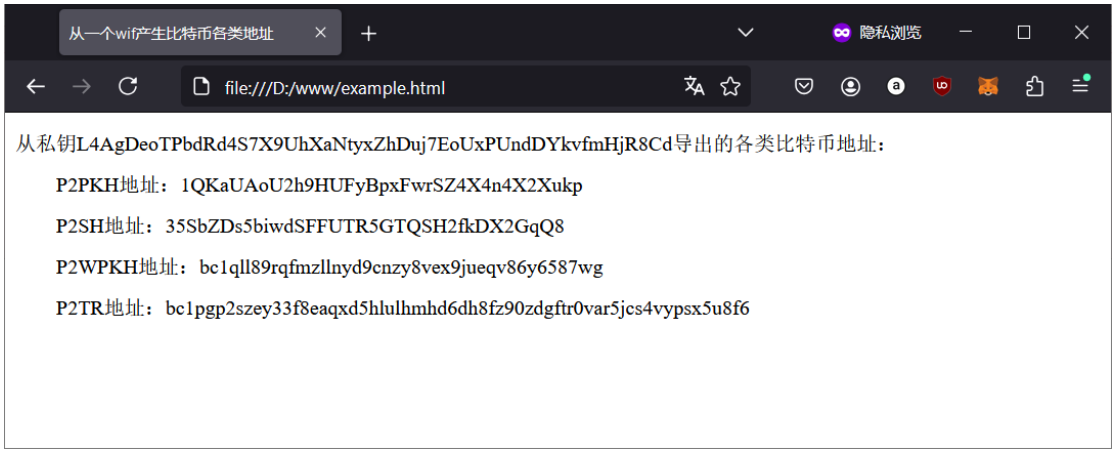


图 2-22 测试页面

2.5 各种库介绍

比特币的各种规范 https://github.com/bitcoin/bips/tree/master

2.5.1 bitcoinjs-lib

买卖加密币要用到另一个 JS 库 bitcoinjs-lib，此库有几个模块组成，我们可以导入需要的模块。以下是 bitcoinjs-lib 的模块结构（具体参考 https://bitcoinjs.github.io/bitcoinjs-lib/index.html）。

```
{
  address: {
    fromBase58Check(address: string):{version, hash},              //从 base58 编码的地址解码出：地址哈希和地址版本
    fromBech32(address: string):{version, prefix, data},      //从 bech32 编码的地址解码出：地址版本、前缀和地址本身
    toBase58Check(hash: Buffer, version: number):string,                              //从公钥推导 base58 格式的地址
    toBech32(data: Buffer, version: number, prefix: string):string,                   //从公钥推导 bech32 格式的地址
    fromOutputScript(output: Buffer, network?: Network):string,                             //从输出脚本中解码出地址
```

```
    toOutputScript(address: string, network?: Network):Buffer,                    //把地址编码成输出脚本
  },
  crypto: {
    ripemd160(buffer: Buffer):Buffer,                                             //做 ripemd160 哈希计算
    sha1(buffer: Buffer):Buffer,                                                  //做 sha1 哈希计算
    sha256(buffer: Buffer):Buffer,                                                //做 sha256 哈希计算
    hash160(buffer: Buffer):Buffer,                                               //做 hash160 哈希计算
    hash256(buffer: Buffer):Buffer,                                               //做 hash256 哈希计算
    TAGS: [
      'BIP0340/challenge',
      'BIP0340/aux',
      'BIP0340/nonce',
      'TapLeaf',
      'TapBranch',
      'TapSighash',
      'TapTweak',
      'KeyAgg list',
      'KeyAgg coefficient'
    ],
    TAGGED_HASH_PREFIXES: {                                                       //附加前缀然后再做哈希计算
      'BIP0340/challenge': <Buffer 7b b5 2d 7a 9f ef 58 32 3e b1 bf 7a 40 7d b3 82 d2 f3 f2 d8 1b b1 22 4f
49 fe 51 8f 6d 48 d3 7c 7b b5 2d 7a 9f ef 58 32 3e b1 bf 7a 40 7d b3 82 d2 f3 ... 14 more bytes>,
      'BIP0340/aux': <Buffer f1 ef 4e 5e c0 63 ca da 6d 94 ca fa 9d 98 7e a0 69 26 58 39 ec c1 1f 97 2d 77
a5 2e d8 c1 cc 90 f1 ef 4e 5e c0 63 ca da 6d 94 ca fa 9d 98 7e a0 69 26 ... 14 more bytes>,
      'BIP0340/nonce': <Buffer 07 49 77 34 a7 9b cb 35 5b 9b 8c 7d 03 4f 12 1c f4 34 d7 3e f7 2d da 19 87
00 61 fb 52 bf eb 2f 07 49 77 34 a7 9b cb 35 5b 9b 8c 7d 03 4f 12 1c f4 34 ... 14 more bytes>,
      TapLeaf: <Buffer ae ea 8f dc 42 08 98 31 05 73 4b 58 08 1d 1e 26 38 d3 5f 1c b5 40 08 d4 d3 57 ca 03
be 78 e9 ee ae ea 8f dc 42 08 98 31 05 73 4b 58 08 1d 1e 26 38 d3 ... 14 more bytes>,
      TapBranch: <Buffer 19 41 a1 f2 e5 6e b9 5f a2 a9 f1 94 be 5c 01 f7 21 6f 33 ed 82 b0 91 46 34 90 d0
5b f5 16 a0 15 19 41 a1 f2 e5 6e b9 5f a2 a9 f1 94 be 5c 01 f7 21 6f ... 14 more bytes>,
      TapSighash: <Buffer f4 0a 48 df 4b 2a 70 c8 b4 92 4b f2 65 46 61 ed 3d 95 fd 66 a3 13 eb 87 23 75 97
c6 28 e4 a0 31 f4 0a 48 df 4b 2a 70 c8 b4 92 4b f2 65 46 61 ed 3d 95 ... 14 more bytes>,
      TapTweak: <Buffer e8 0f e1 63 9c 9c a0 50 e3 af 1b 39 c1 43 c6 3e 42 9c bc eb 15 d9 40 fb b5 c5 a1 f4
af 57 c5 e9 e8 0f e1 63 9c 9c a0 50 e3 af 1b 39 c1 43 c6 3e 42 9c ... 14 more bytes>,
      'KeyAgg list': <Buffer 48 1c 97 1c 3c 0b 46 d7 f0 b2 75 ae 59 8d 4e 2c 7e d7 31 9c 59 4a 5c 6e c7 9e
a0 d4 99 02 94 f0 48 1c 97 1c 3c 0b 46 d7 f0 b2 75 ae 59 8d 4e 2c 7e d7 ... 14 more bytes>,
      'KeyAgg coefficient': <Buffer bf c9 04 03 4d 1c 88 e8 c8 0e 22 e5 3d 24 56 6d 64 82 4e d6 42 72 81 c0
91 00 f9 4d cd 52 c9 81 bf c9 04 03 4d 1c 88 e8 c8 0e 22 e5 3d 24 56 6d 64 82 ... 14 more bytes>
    },
    taggedHash(prefix: TaggedHashPrefix, data: Buffer):Buffer,                    //给 data 加上前缀后做 sha256 哈希计算
  },
  networks: {
    bitcoin: {                                                                    //比特币主网
      messagePrefix: '\x18Bitcoin Signed Message:\n',
      bech32: 'bc',
      bip32: [Object],
      pubKeyHash: 0,
      scriptHash: 5,
      wif: 128
    },
    regtest: {                                                                    //比特币本地测试网
      messagePrefix: '\x18Bitcoin Signed Message:\n',
      bech32: 'bcrt',
      bip32: [Object],
      pubKeyHash: 111,
      scriptHash: 196,
      wif: 239
    },
    testnet: {                                                                    //比特币测试网
      messagePrefix: '\x18Bitcoin Signed Message:\n',
      bech32: 'tb',
      bip32: [Object],
      pubKeyHash: 111,
      scriptHash: 196,
      wif: 239
    }
  },
  payments: {
    embed(a: Payment, opts?: PaymentOpts):Payment,                                //把数据嵌入到 Payment 对象
    p2ms(a: Payment, opts?: PaymentOpts):Payment,                                 //创建一个 p2ms 类型的 Payment 对象
    p2pk(a: Payment, opts?: PaymentOpts):Payment,                                 //创建一个 p2pk 类型的 Payment 对象
    p2pkh(a: Payment, opts?: PaymentOpts):Payment,                                //创建一个 p2pkh 类型的 Payment 对象
```

```
    p2sh(a: Payment, opts?: PaymentOpts):Payment,                          //创建一个 p2sh 类型的 Payment 对象
    p2wpkh(a: Payment, opts?: PaymentOpts):Payment,                        //创建一个 p2wpkh 类型的 Payment 对象
    p2wsh(a: Payment, opts?: PaymentOpts):Payment,                         //创建一个 p2wsh 类型的 Payment 对象
    p2tr(a: Payment, opts?: PaymentOpts):Payment,                          //创建一个 p2tr 类型的 Payment 对象
  },
  script: {                                                                //本模块操纵脚本语言(锁定和输入脚本要用到)
    OPS={ OP_FALSE: 0, OP_0: 0, OP_PUSHDATA1: 76, …},                                  //包含各种操作码的对象
    number: { decode: [Function: decode], encode: [Function: encode] },
    signature: { decode: [Function: decode], encode: [Function: encode] },
    isPushOnly(value: Stack): boolean,                                                 //判断堆栈中是否只存在压栈操作码
    compile(chunks: Uint8Array | Stack): Uint8Array,                                   //将脚本块数组编译为 Uint8Array
    countNonPushOnlyOPs(value: Stack): number,                             //统计堆栈中除了压栈操作码之外的操作码数量
    decompile(buffer): (number | Uint8Array)[] | null,                                 //将脚本缓冲区反编译为块数组
    fromASM(asm: string): Uint8Array,                                                                  //汇编
    isCanonicalPubKey(buffer: Uint8Array): boolean,                                                    //是否是规范的公钥
    isDefinedHashType(hashType: number): boolean,                                                      //哈希类型是否存在
    isCanonicalScriptSignature(buffer: Uint8Array): boolean,                                           //是否是规范的签名脚本
    toASM(chunks: Uint8Array | (number | Uint8Array)[]): string,                                                  //反汇编
    toStack(chunks: Uint8Array | (number | Uint8Array)[]): Uint8Array[]
  },
  Block 类,
  Psbt 类,                                                                                                     //遵循 bip174 的类
  opcodes 对象,                                                          //包含脚本的操作码。bitcoinjs-lib 7.0 之后的版本没有了
  Transaction 类,
  initEccLib(eccLib: undefined|TinySecp256k1Interface, opts?: {DANGER_DO_NOT_VERIFY_ECCLIB: boolean;}):
void
}
```

方法 initEccLib()初始化 ECC 库，如果参数 eccLib 空，则清除 ECC 库，有的方法如 bitcoinjs-lib.payments.p2tr()只有初始化了 ECC 库之后才能使用，eccLib 参数为 TinySecp256k1 兼容的实例即可，如：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');                 //或者 const ecc = require('@bitcoinerlab/secp256k1')

bitcoin.initEccLib(ecc);
```

2.5.1.1 Psbt 类

Psbt 类用于构建和操作部分签名的比特币交易（Partially Signed Bitcoin Transaction），它允许交易的不同部分在不同时间由不同的实体签名，这种格式对于多方签名和硬件钱包特别有用，因为它允许在交易被最终广播到比特币网络之前，安全地传递交易数据以进行签名。Psbt 类可以解析和生成基于 BIP174 的 PSBT 二进制文件，此类定义如下：

```
export declare class Psbt {
    readonly data: PsbtBase;
    static fromBase64(data: string, opts?: PsbtOptsOptional): Psbt;
    static fromHex(data: string, opts?: PsbtOptsOptional): Psbt;
    static fromBuffer(buffer: Buffer, opts?: PsbtOptsOptional): Psbt;
    private __CACHE;
    private opts;
    constructor(opts?: PsbtOptsOptional, data?: PsbtBase);
    get inputCount(): number;
    get version(): number;
    set version(version: number);
    get locktime(): number;
    set locktime(locktime: number);
    get txInputs(): PsbtTxInput[];
    get txOutputs(): PsbtTxOutput[];
    combine(...those: Psbt[]): this;
    clone(): Psbt;
```

```
    setMaximumFeeRate(satoshiPerByte: number): void;
    setVersion(version: number): this;
    setLocktime(locktime: number): this;
    setInputSequence(inputIndex: number, sequence: number): this;
    addInputs(inputDatas: PsbtInputExtended[]): this;
    addInput(inputData: PsbtInputExtended): this;
    addOutputs(outputDatas: PsbtOutputExtended[]): this;
    addOutput(outputData: PsbtOutputExtended): this;
    extractTransaction(disableFeeCheck?: boolean): Transaction;
    getFeeRate(): number;
    getFee(): number;
    finalizeAllInputs(): this;
    finalizeInput(inputIndex: number, finalScriptsFunc?: FinalScriptsFunc | FinalTaprootScriptsFunc): this;
    finalizeTaprootInput(inputIndex: number, tapLeafHashToFinalize?: Buffer, finalScriptsFunc?:
FinalTaprootScriptsFunc): this;
    private _finalizeInput;
    private _finalizeTaprootInput;
    getInputType(inputIndex: number): AllScriptType;
    inputHasPubkey(inputIndex: number, pubkey: Buffer): boolean;
    inputHasHDKey(inputIndex: number, root: HDSigner): boolean;
    outputHasPubkey(outputIndex: number, pubkey: Buffer): boolean;
    outputHasHDKey(outputIndex: number, root: HDSigner): boolean;
    validateSignaturesOfAllInputs(validator: ValidateSigFunction): boolean;
    validateSignaturesOfInput(inputIndex: number, validator: ValidateSigFunction, pubkey?: Buffer):
boolean;
    private _validateSignaturesOfInput;
    private validateSignaturesOfTaprootInput;
    signAllInputsHD(hdKeyPair: HDSigner, sighashTypes?: number[]): this;
    signAllInputsHDAsync(hdKeyPair: HDSigner | HDSignerAsync, sighashTypes?: number[]): Promise<void>;
    signAllInputs(keyPair: Signer, sighashTypes?: number[]): this;
    signAllInputsAsync(keyPair: Signer | SignerAsync, sighashTypes?: number[]): Promise<void>;
    signInput(inputIndex: number, keyPair: Signer, sighashTypes?: number[]): this;
    signTaprootInput(inputIndex: number, keyPair: Signer, tapLeafHashToSign?: Buffer, sighashTypes?:
number[]): this;
    private _signInput;
    private _signTaprootInput;
    signInputAsync(inputIndex: number, keyPair: Signer | SignerAsync, sighashTypes?: number[]):
Promise<void>;
    signTaprootInputAsync(inputIndex: number, keyPair: Signer | SignerAsync, tapLeafHash?: Buffer,
sighashTypes?: number[]): Promise<void>;
    private _signInputAsync;
    private _signTaprootInputAsync;
    private checkTaprootHashesForSig;
    toBuffer(): Buffer;
    toHex(): string;
    toBase64(): string;
    updateGlobal(updateData: PsbtGlobalUpdate): this;
    updateInput(inputIndex: number, updateData: PsbtInputUpdate): this;
    updateOutput(outputIndex: number, updateData: PsbtOutputUpdate): this;
    addUnknownKeyValToGlobal(keyVal: KeyValue): this;
    addUnknownKeyValToInput(inputIndex: number, keyVal: KeyValue): this;
    addUnknownKeyValToOutput(outputIndex: number, keyVal: KeyValue): this;
    clearFinalizedInput(inputIndex: number): this;
}
interface PsbtOptsOptional {
    network?: Network;
    maximumFeeRate?: number;
}
interface PsbtInputExtended extends PsbtInput, TransactionInput {
}
type PsbtOutputExtended = PsbtOutputExtendedAddress | PsbtOutputExtendedScript;
interface PsbtOutputExtendedAddress extends PsbtOutput {
    address: string;
    value: number;
}
interface PsbtOutputExtendedScript extends PsbtOutput {
    script: Buffer;
    value: number;
}
interface HDSignerBase {
    /**
     * DER format compressed publicKey buffer
     */
    publicKey: Buffer;
```

```
    /**
     * The first 4 bytes of the sha256-ripemd160 of the publicKey
     */
    fingerprint: Buffer;
}
export interface HDSigner extends HDSignerBase {
    /**
     * The path string must match /^m(\/\d+'?)+$/
     * ex. m/44'/0'/0'/1/23 levels with ' must be hard derivations
     */
    derivePath(path: string): HDSigner;
    /**
     * Input hash (the "message digest") for the signature algorithm
     * Return a 64 byte signature (32 byte r and 32 byte s in that order)
     */
    sign(hash: Buffer): Buffer;
}
/**
 * Same as above but with async sign method
 */
export interface HDSignerAsync extends HDSignerBase {
    derivePath(path: string): HDSignerAsync;
    sign(hash: Buffer): Promise<Buffer>;
}
export interface Signer {
    publicKey: Buffer;
    network?: any;
    sign(hash: Buffer, lowR?: boolean): Buffer;
    signSchnorr?(hash: Buffer): Buffer;
    getPublicKey?(): Buffer;
}
export interface SignerAsync {
    publicKey: Buffer;
    network?: any;
    sign(hash: Buffer, lowR?: boolean): Promise<Buffer>;
    signSchnorr?(hash: Buffer): Promise<Buffer>;
    getPublicKey?(): Buffer;
}

/**
 * This function must do two things:
 * 1. Check if the `input` can be finalized. If it can not be finalized, throw.
 *   ie. `Can not finalize input #${inputIndex}`
 * 2. Create the finalScriptSig and finalScriptWitness Buffers.
 */
type FinalScriptsFunc = (
      inputIndex: number,                                                    // Which input is it?
      input: PsbtInput,                                                   // The PSBT input contents
      script: Buffer,               // The "meaningful" locking script Buffer (redeemScript for P2SH etc.)
      isSegwit: boolean,                                                        // Is it segwit?
      isP2SH: boolean,                                                            // Is it P2SH?
      isP2WSH: boolean
) => {
      finalScriptSig: Buffer | undefined;
      finalScriptWitness: Buffer | undefined;
};

type FinalTaprootScriptsFunc = (
      inputIndex: number, // Which input is it?
      input: PsbtInput, // The PSBT input contents
      tapLeafHashToFinalize?: Buffer
) => {
      finalScriptWitness: Buffer | undefined;
};
type AllScriptType = 'witnesspubkeyhash' | 'pubkeyhash' | 'multisig' | 'pubkey' | 'nonstandard' | 'p2sh-
witnesspubkeyhash' | 'p2sh-pubkeyhash' | 'p2sh-multisig' | 'p2sh-pubkey' | 'p2sh-nonstandard' | 'p2wsh-
pubkeyhash' | 'p2wsh-multisig' | 'p2wsh-pubkey' | 'p2wsh-nonstandard' | 'p2sh-p2wsh-pubkeyhash' | 'p2sh-
p2wsh-multisig' | 'p2sh-p2wsh-pubkey' | 'p2sh-p2wsh-nonstandard';
```

此类能履行六个角色。

角色 1：创建者

存在四种实例化 Psbt 类的方法：

```
const psbt = new Psbt(opts?: PsbtOptsOptional = {}, data?: Psbt = ...): Psbt
const psbt = fromBase64(data: string, opts?: PsbtOptsOptional = {}): Psbt
const psbt = fromBuffer(buffer: Uint8Array, opts?: PsbtOptsOptional = {}): Psbt
const psbt = fromHex(data: string, opts?: PsbtOptsOptional = {}): Psbt
```

fromBase64()方法中的参数 data 就是 psbt 实例的方法 toBase64()输出的结果，另外 fromBuffer()和 toBuffer()方法互逆，fromHex()与 toHex()互逆。

其中接口 PsbtOptsOptional 定义如下：

```
interface PsbtOptsOptional {
   network?: Network;                                          //默认是比特币公网：bitcoin.networks.bitcoin
   maximumFeeRate?: number;                                    //最大费率，即每个字节最多几聪，默认是 5000
}
```

如果以后要添加输出，就会用到 network 参数。maximumFeeRate 定义的最大费率不是最终的交易费率，只是在发布交易时检查费用有没有超限，如果超限就会报错而拒绝发布。经常有人误解：花费一个大额 UTXO，但只给对方放少量币，以为多余是钱还在钱包里，实际上作为交易费都给了矿工，因为交易费等于全部输入中的金额之和减去全部输出中的金额之和。费率以后也可以调用 setMaximumFeeRate()方法修改。

例如实例化一个 Psbt 对象，采用测试网，每字节最多交易手续费 100 聪：

```
const bitcoin = require('bitcoinjs-lib');
const psbt = new bitcoin.Psbt({ network: bitcoin.networks.testnet, maximumFeeRate: 100 });
```

角色 2：修改者

给 Psbt 的实例增加一个输入：psbt.addInput(inputData: PsbtInputExtended): this。

给 Psbt 的实例增加多个输入：psbt.addInputs(inputDatas: PsbtInputExtended[]): this

其中 PsbtInputExtended 定义如下：

```
PsbtInputExtended = {
   partialSig?: PartialSig[];                //interface PartialSig { pubkey: Buffer, signature: Buffer }
   nonWitnessUtxo?: Buffer;
   witnessUtxo?: WitnessUtxo;                  //interface WitnessUtxo { script: Buffer, value: number; }
   sighashType?: Number;                  //指明签名的哈希类型，默认 SIGHASH_ALL，bitcoin.Transaction 中定义这些宏
   redeemScript?: Buffer;                                                  //赎回脚本（针对 P2SH 和 P2WSH 地址类型）
   witnessScript?: Buffer;                                                 //P2WSH 或 P2SH-P2WSH 输入的输入脚本
   bip32Derivation?: Bip32Derivation[];
                 //interface Bip32Derivation { masterFingerprint: Buffer, pubkey: Buffer, path: string }
   finalScriptSig?: Buffer;                            //默认由 finalizeInput()或 finalizeAllInputs()方法产生
   finalScriptWitness?: Buffer;                        //默认由 finalizeInput()或 finalizeAllInputs()方法产生
   porCommitment?: string;

   tapKeySig?: Buffer;
   tapScriptSig?: TapScriptSig[];          //interface TapScriptSig extends PartialSig {leafHash: Buffer}
   tapLeafScript?: TapLeafScript[];
                //interface TapLeafScript { leafVersion: number, script: Buffer, controlBlock: Buffer }
   tapBip32Derivation?: TapBip32Derivation[];
                          //interface TapBip32Derivation extends Bip32Derivation {leafHashes: Buffer[]}
   tapInternalKey?: Buffer;                                                //内部公钥（被 Tweak 的那个公钥）
   tapMerkleRoot?: Buffer;                                                 //MAST 树根

   hash: string | Buffer;                                                  //UTXO 的交易 id
   index: number;                                                          //UTXO 的输出顺序号
   sequence?: number;                                                      //顺序号，默认值为 0xffffffff
```

```
}
```

PsbtInputExtended 对象的成员 hash 和 index 确定一个要花费的 UTXO，其他可选字段（如 witnessUtxo、nonWitnessUtxo、bip32Derivation 等）用于提供额外的信息，以满足不同场景的需求（如隔离见证、多签名、HD 钱包等）。

1. 可选参数 witnessUtxo 的类型为对象{ script: Buffer, value: number }，script 为被花费的那个 UTXO 的输出脚本，value 为 UTXO 的金额（单位聪）。本参数针对隔离见证的 UTXO（如 P2WPKH、P2WSH、P2TR 地址类型），是必填项。例如：

```
witnessUtxo: {
  script: Buffer.from('0014abcdef1234567890abcdef1234567890abcdef12', 'hex'),
  value: 100000,
},
```

之所以要提供此参数，是为了签名和验证输入脚本。

2. 可选参数 nonWitnessUtxo 的类型为 Buffer，用于非隔离见证（如 P2PK、P2PKH、P2SH 地址类型）输入的完整交易数据。例如对于 UTXO (TxHASH=8ab9dd72d22337458af8285c70e7e65916893f8691daf6fcfb9dde70a39fdf10, output_index=0) 完整的交易数据可浏览 https://blockchain.info/rawtx/8ab9dd72d22337458af8285c70e7e65916893f8691daf6fcfb9dde70a39fdf10?format=hex，结果是：

```
nonWitnessUtxo:Buffer.from('010000000218c6b1df33f91cbe279fe04519ff20f5020504d5c41e5ee7facd2cc825fb3944000000006a4730440220169630d7c5b4b4faa8ea7606ebbc34fe9c6ef54c26608472f174e6e7b28d7cb3022075faa1d0a5e94a8daec63d8ea74cb970e01bebb78979583f90dbc73f268c637c0121021b059b0796099792f913787aaa04cbfbfab020c1e7e2b0ff0b209e0030c68272ffffffffa8cc9514f5eeafbf99f49388c68f50eea83a7550f8bc1eafba5e93dd1d539f6f000000006a4730440220298564c2048ee1510050b0a909e3447cd54a8d2404e8f7d444c67b012ef4a8610220111025b711fcccbcdc55d33d48fb8ebeccd2463aa6c3c13603a0880f5c700fcd0121038643e4640d1ae48a65f1fb4c2c8b948e62a4ee4d47a66815aa4be533d178f2dffffffffff01e8030000000000001976a914e06f1b3b23fe35386c31e5c6dc43b4fc4c025c3988ac00000000', 'hex');
```

3. 可选参数 redeemScript 指定赎回脚本，针对 P2SH 和 P2WSH 地址类型的 UTXO，关于赎回脚本请参考“1.9 比特币简介”小节。

4. 可选参数 witnessScript 指定 P2WSH 或 P2SH-P2WSH 类型的输入的输入脚本。仅当输入是 P2WSH 或 P2SH-P2WSH 时必填。看代码：

```
// 添加 P2WSH 输入
psbt.addInput({
   hash: 'a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6', //UTXO 的交易 ID
   index: 0, //UTXO 的输出索引
   witnessUtxo: {
     script: Buffer.from('0020abcdef1234567890abcdef1234567890abcdef1234567890abcdef12345678', 'hex'), //P2WSH 输出脚本
     value: 100000, //金额（以聪为单位）
   },
   witnessScript:
Buffer.from('522103abcdef1234567890abcdef1234567890abcdef1234567890abcdef1234567890210fedcba9876543210fedcba9876543210fedcba9876543210fedcba987654321052ae', 'hex'), //多签名输入脚本
});
```

5. 可选参数 bip32Derivation 的类型为对象{ masterFingerprint: Buffer; pubkey: Buffer; path: string; }，用于 HD 钱包或多签名场景，指定公钥的派生路径。如 path: "m/44'/0'/0'/0/1"。

6. 可选参数 finalScriptSig 指定最终的输入脚本，是专门针对非隔离见证地址的，默认由 finalizeInput()或 finalizeAllInputs()方法产生。也可以通过本参数认为定义。另一个可选参数 finalScriptWitness 功能类似，只不过针对隔离见证地址。

7. 可选参数 partialSig 的类型为对象{ pubkey: Buffer, signature: Buffer }的数组，用于多签场景，它包含多个部分签名，一个部分签名就是一对公钥和签名。看例子：

```
psbt.addInput({
   hash: '8ab9dd72d22337458af8285c70e7e65916893f8691daf6fcfb9dde70a39fdf10', //引用的 UTXO 的交易 ID
   index: 0, //引用的 UTXO 的输出索引
   witnessUtxo: {
```

```
      script: Buffer.from('0020abcdef1234567890abcdef1234567890abcdef1234567890abcdef12345678', 'hex'),
// P2WSH 锁定脚本
      value: 100000, // 金额（以 satoshi 为单位）
   },
   witnessScript:
Buffer.from('522103abcdef1234567890abcdef1234567890abcdef1234567890abcdef1234567890210fedcba9876543210fedc
ba9876543210fedcba9876543210fedcba987654321052ae', 'hex'),                                    //多签名见证脚本
   partialSig: [
   {
      pubkey: Buffer.from('03578ad3da7fd074d3a41ea41b6f910ed8a197f6ddc7c210d998d75d87e783d4e7', 'hex'),
// 公钥 1
      signature:
Buffer.from('304402206eabc8529418d3cba0a8deda8de3b1a0b38582f930ba2527390fba379ad7d87b022077de5573d30cb0e10
81da95f1f55fb30d7dc62ee569eea8fd893f8a72d75b77301', 'hex'),                                    //签名 1
   },
   {
      pubkey: Buffer.from('03974bf8842548de2191411f3ea8724a09ff7131fb7c43fd935208ca9328285126', 'hex'),
// 公钥 2
      signature:
Buffer.from('304402205aaaa23d7ba69eb6d233cd31ab242d095aaaa23d7ba69eb6d233cd31ab242d0902204eec8e7669241249b
4f597d9dfe34a9a25cc61f0f55f1c63b8f22df47371836d01', 'hex'),                                    // 签名 2
   },
   ],
});
```

8. 可选参数 tapLeafScript 指定叶子脚本，这是专门针对 P2TR 类型的输入，它的值为对象{ leafVersion: number, script: Buffer, controlBlock: Buffer }的数组，leafVersion 目前固定为 c0，controlBlock 为控制脚本，script 为脚本，详细介绍请回看“1.9.2 解锁过程”小节。

9. 可选参数 tapBip32Derivation 指定 Taproot 输入的 BIP32 派生信息，值为对象{ masterFingerprint: Buffer, pubkey: Buffer, path: string, leafHashes: Buffer[] }的数组。

```
tapBip32Derivation: [
   {
      masterFingerprint: Buffer.from('abcd1234', 'hex'),
      pubkey: Buffer.from('...', 'hex'),
      path: "m/86'/0'/0'/0/0",
      leafHashes: [Buffer.from('...', 'hex')],
   },
],
```

10. 可选参数 tapMerkleRoot 用于 Taproot 输入的默克尔树根。

修改一个输入：updateInput(inputIndex: number, updateData: PsbtInputUpdate): this

inputIndex 指明修改第几个输入，新的数据放在 updateData 中，PsbtInputUpdate 就是上面 PsbtInputExtended 中除了红色字体部分的剩余部分。

添加一个输出：psbt.addOutput(outputData: PsbtOutputExtended): this

添加多个输出：psbt.addOutputs(outputDatas: PsbtOutputExtended[]): this

其中 PsbtOutputExtended 定义如下：

```
PsbtOutputExtended = {
   redeemScript?: Buffer;
   witnessScript?: Buffer;
   bip32Derivation?: Bip32Derivation[]; //interface Bip32Derivation { masterFingerprint: Buffer, pubkey:
Buffer, path: string }
   tapBip32Derivation?: TapBip32Derivation[];      //interface TapBip32Derivation extends Bip32Derivation
{leafHashes: Buffer[]}
```

```
    tapTree?: TapTree;  //interface TapTree {leaves: TapLeaf[];}//interface TapLeaf {leafVersion: number,
script: Buffer, depth: number}, depth 叶子所在的层数，默克尔根为 0 层
    tapInternalKey?: Buffer;
    unknownKeyVals?: KeyValue[];
    address: string;
    script: Buffer;

    value: number;

};                                                                          //address 和 script 只能选择一个
```

修改一个输出：updateOutput(outputIndex: number, updateData: PsbtOutputUpdate): this

用 updateData 里的数据修改第 outputIndex 个输出，PsbtOutputUpdate 就是上面 PsbtOutputExtended 中除了红色字体的那部分。

updateGlobal(updateData: PsbtGlobalUpdate): this

PsbtGlobalUpdate 定义如下：

```
interface PsbtGlobalUpdate {

    globalXpub?: GlobalXpub[];

}

interface GlobalXpub {
    extendedPubkey: Buffer;
    masterFingerprint: Buffer;
    path: string;

}
```

角色 3：签名者

对所有输入签名（同步）：signAllInputs(keyPair: Signer, sighashTypes?: number[]): this

其中 Signer 定义如下：

```
interface Signer {
    network?: any;
    publicKey: Uint8Array;
    getPublicKey?(): Uint8Array;
    sign(hash: Uint8Array, lowR?: boolean): Uint8Array;
    signSchnorr?(hash: Uint8Array): Uint8Array;
}
```

对所有输入签名（异步）：signAllInputsAsync(keyPair: Signer | SignerAsync, sighashTypes? : number[]): Promise<void>

其中 SignerAsync 定义如下：

```
interface SignerAsync {
    network?: any;
    publicKey: Uint8Array;
    getPublicKey?(): Uint8Array;
    sign(hash: Uint8Array, lowR?: boolean): Promise<Uint8Array>;
    signSchnorr?(hash: Uint8Array): Promise<Uint8Array>;
}
```

签名一个输入（同步）：signInput(inputIndex: number, keyPair: Signer, sighashTypes?: number[]): this

签名一个输入（异步）：signInputAsync(inputIndex: number, keyPair: Signer|SignerAsync, sighashTypes?: number[]): this

signInputHD(inputIndex: number, hdKeyPair: HDSigner, sighashTypes?: number[] = ...): this，

signInputHDAsync(inputIndex: number, hdKeyPair: HDSigner, sighashTypes?: number[] = ...): Promise<void>

signAllInputsHD(hdKeyPair: HDSigner, sighashTypes?: number[] = ...): this，其中 HDSigner 定义如下：

```
interface HDSigner {
   fingerprint: Uint8Array;
   publicKey: Uint8Array;
   derivePath(path: string): HDSigner;
   sign(hash: Uint8Array): Uint8Array;
}
```

属性 fingerprint 为公钥做 sha256-ripemd160 运算后的前四个字节，publickey 为 DER 格式的压缩公钥。方法 derivePath() 从路径导出 HDSigner 接口，路径符合/^m(/\d+'?)+$/，如 m/84'/0'/0'/1/2，方法 sign()对 hash（通常为文本的哈希值）签名，返回 64 字节的数组。

signAllInputsHDAsync(hdKeyPair: HDSigner | HDSignerAsync, sighashTypes?: number[] = ...): Promise<void>，其中 HDSignerAsync 定义如下：

```
interface HDSignerAsync {
   fingerprint: Uint8Array;
   publicKey: Uint8Array;
   derivePath(path: string): HDSignerAsync;
   sign(hash: Uint8Array): Promise<Uint8Array>;
}
```

签名一个 Taproot 输入（同步）：signTaprootInput(inputIndex: number, keyPair: Signer, tapLeafHashToSign?: Uint8Array, sighashTypes?: number[]): this。

签名一个 Taproot 输入（异步）：signTaprootInputAsync(inputIndex: number, keyPair: Signer | SignerAsync, tapLeafHashToSign?: Uint8Array, sighashTypes?: number[]): Promise<void>。

签名时会验证输入脚本能否解锁输出脚本。

角色 4：组合

组合多个 Psbt 对象：combine (...those: Psbt[]): this。主要合并各个 Psbt 中输入和输出，但要求全局参数（如版本、顺序号、锁定时间等）一致。

角色 5：输入终结

终结输入就是产生输入的输入脚本，即完成所有输入的最终化处理，包括验证签名、构建最终的输入脚本和清理中间数据以减少交易大小。

终结所有输入：finalizeAllInputs():this。

终结一个输入：finalizeInput(inputIndex, finalScriptsFunc?: FinalScriptsFunc | FinalTaprootScriptsFunc): this。

其中 finalScriptsFunc 和 FinalTaprootScriptsFunc 定义如下：

```
type FinalScriptsFunc = (
   inputIndex: number,                                                    //输入索引
   input: PsbtInput,                                                      //PSBT 的输入部分
   script: Buffer,                                                        //输出脚本(对 P2SH、P2WSH 来说就是赎回脚本)
   isSegwit: boolean,                                                     //是隔离见证吗?
   isP2SH: boolean,                                                       //是 P2SH 吗?
   isP2WSH: boolean                                                       //是 P2WSH 吗?
) => { finalScriptSig: Buffer | undefined; finalScriptWitness: Buffer | undefined; };

type FinalTaprootScriptsFunc = (
```

```
    inputIndex: number,                                                    //输入索引
    input: PsbtInput,                                                      //PSBT 的输入部分
    tapLeafHashToFinalize?: Buffer
) => { finalScriptWitness: Buffer | undefined; };
```

终结一个 Taproot 输入：finalizeTaprootInput(inputIndex: number, tapLeafHashToFinalize?: Uint8Array, finalScriptsFunc?: FinalTaprootScriptsFunc): this

清除一个输入的终结（即终结的逆操作）：clearFinalizedInput(inputIndex: number): this;

其中 FinalTaprootScriptsFunc 就是函数 tapScriptFinalizer(inputIndex: number, input: PsbtInput, tapLeafHashToFinalize?: Buffer)。

不过终结输入之前建议先调用方法 psbt.validateSignaturesOfAllInputs(validator: ValidateSigFunction) 检查签名或者 validateSignaturesOfInput(inputIndex: number, validator: ValidateSigFunction, pubkey?: Uint8Array): boolean ， 其 中 ValidateSigFunction 这样定义：type ValidateSigFunction = (pubkey: Buffer, msghash: Buffer, signature: Buffer) => boolean，比如用下面的函数做验证子：

```
//验证 ECDSA 签名:
const validator = ( pubkey, msghash, signature) => ECPair.fromPublicKey(pubkey).verify(msghash,
signature);

//验证 Schnorr 签名:
const validator = (pubkey, msghash, signature) => tiny-secp256k1.verifySchnorr(msghash, pubkey,
signature);
```

角色 6：提取事务者

从 Psbt 的实例提取事务：extractTransaction(disableFeeCheck?: boolean): Transaction，参数 disableFeeCheck 指明是否检查交易费用,true-不检查,false-检查,默认是 false。此方法返回 Transaction 对象,关于 Transaction 类请参考 Psbt.Transaction，最终调用 Transaction.toHex()得到用十六进制表示的交易串，这个交易串就可以发布给矿工了。

Psbt 类的其他方法

clone() - 克隆或者拷贝本 Psbt 对象，如 let psbt_back = psbt.clone();

getFee() - 获取交易费用，以聪为单位。

getFeeRate() - 获取费率。

getInputType(inputIndex: number): AllScriptType - 获得输入的类型。其中 AllScriptType 定义如下：

```
type AllScriptType = 'witnesspubkeyhash' | 'pubkeyhash' | 'multisig' | 'pubkey' | 'nonstandard' | 'p2sh-
witnesspubkeyhash' | 'p2sh-pubkeyhash' | 'p2sh-multisig' | 'p2sh-pubkey' | 'p2sh-nonstandard' | 'p2wsh-
pubkeyhash' | 'p2wsh-multisig' | 'p2wsh-pubkey' | 'p2wsh-nonstandard' | 'p2sh-p2wsh-pubkeyhash' | 'p2sh-
p2wsh-multisig' | 'p2sh-p2wsh-pubkey' | 'p2sh-p2wsh-nonstandard';
```

inputHasHDKey(inputIndex: number, root: HDSigner): boolean -

inputHasPubkey(inputIndex: number, pubkey: Uint8Array): boolean - 某个输入中含有这个公钥吗?

outputHasHDKey(inputIndex: number, root: HDSigner): boolean -

outputHasPubkey(inputIndex: number, pubkey: Uint8Array): boolean -

setInputSequence(inputIndex: number, sequence: number): this - 设置输入的顺序号，如 setInputSequence(0,0xfeffffff)。

setLocktime(locktime: number): this - 设置锁定时间。如 psbt.setLocktime(0)。

setMaximumFeeRate(satoshiPerByte: number): void - 设置最大费率。

setVersion(version): this - 设置版本。如 psbt.setVersion(2)。

toBase64(): string - 转换为 Base64 格式的字符串。

toBuffer(): Uint8Array - 转换为 Buffer。

toHex(): string - 转为 hex 编码。

下面利用 Psbt 类实现一个完整的例子，从 P2PKH 和 P2WPKH 两个 UTXO 放币到 P2TR 地址：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');
const { ECPairFactory } = require('ecpair');

//定义验证子（用于验证签名）:
const validator = ( pubkey, msghash, signature) => ECPair.fromPublicKey(pubkey).verify(msghash,
signature);

bitcoin.initEccLib(ecc);
const ECPair = ECPairFactory(ecc);
const network = bitcoin.networks.bitcoin;

const p2pkhPrivateKey = 'L3MLmwGxE9GuZiE9Gf17VSgYtBnMAMGuZ36yBRWR3WbooyJe4onD';
const p2wpkhPrivateKey = 'L37K5rF6VXD5XuBHRHxTvhPAvYAUvvMjFoJfFnypV2AMzHwBuNED';
const p2pkh_rawTX =
'010000000218c6b1df33f91cbe279fe04519ff20f5020504d5c41e5ee7facd2cc825fb3944000000006a4730440220169630d7c5b
4b4faa8ea7606ebbc34fe9c6ef54c26608472f174e6e7b28d7cb3022075faa1d0a5e94a8daec63d8ea74cb970e01bebb78979583f9
0dbc73f268c637c0121021b059b0796099792f913787aaa04cbfbfab020c1e7e2b0ff0b209e0030c68272ffffffffa8cc9514f5eea
fbf99f49388c68f50eea83a7550f8bc1eafba5e93dd1d539f6f000000006a4730440220298564c2048ee1510050b0a909e3447cd54
a8d2404e8f7d444c67b012ef4a8610220111025b711fcccbcdc55d33d48fb8ebeccd2463aa6c3c13603a0880f5c700fcd012103864
3e4640d1ae48a65f1fb4c2c8b948e62a4ee4d47a66815aa4be533d178f2dfffffffff01e803000000000001976a914e06f1b3b23f
e35386c31e5c6dc43b4fc4c025c3988ac00000000';                                    //完整的交易数据（需要从区块链获取）
const toAddress = 'bc1ptnpz3wsth2xpdx0lhv86f36k67t4l4e225j636j0vtsrlk42gndqhe5dx6';       //对方地址（P2TR）

//由私钥产生公私钥对（后面用于签名）:
const p2pkhKeyPair = ECPair.fromWIF(p2pkhPrivateKey, network);
const p2wpkhKeyPair = ECPair.fromWIF(p2wpkhPrivateKey, network);

//定义两个 UTXO:
const utxos = [
   {
      type: 'P2PKH',
      txId: '95056a5c14fbd6f9d9a5491af96a909fb2d9a44acfd8f910cfc4584463d43526',                  //交易 ID
      vout: 0,                                                                                  //输出索引
      scriptPubKey: '76a914c825a1ecf2a6830c4401620c3a16f1995057c2ab88ac',                       //输出脚本
      value: 1000000,                                                                  //金额（以 satoshi 为单位）
      keyPair: p2pkhKeyPair,                                                                    //私钥
   },
   {
      type: 'P2WPKH',
      txId: '466200308696215bbc949d5141a49a4138ecdfdfaa2a8029c1f9bcecd1f96177',                  //交易 ID
      vout: 1,                                                                                  //输出索引
      scriptPubKey: '00142566285e4349f21c0e07b08de695548628803ff3',                             //输出脚本
      value: 200000,                                                                   //金额（以 satoshi 为单位）
      keyPair: p2wpkhKeyPair,                                                                   //私钥
   },
];

const psbt = new bitcoin.Psbt({ network });                                          //实例化 Psbt 类

//添加输入:
for (const utxo of utxos) {
   if (utxo.type === 'P2PKH') {                                                      //P2PKH 类型的输入
      psbt.addInput({
```

```
            hash: utxo.txId,                                                    //要花费的那个 UTXO 的交易 ID
            index: utxo.vout,                                                   //要花费的那个 UTXO 的输出索引
            nonWitnessUtxo: Buffer.from(p2pkh_rawTX, 'hex'),
        });
    } else if (utxo.type === 'P2WPKH') {                                        //P2WPKH 类型的输入
        psbt.addInput({
            hash: utxo.txId,                                                    //引用的 UTXO 的交易 ID
            index: utxo.vout,                                                   //引用的 UTXO 的输出索引
            witnessUtxo: {
                script: Buffer.from(utxo.scriptPubKey, 'hex'),                  //输出脚本
                value: utxo.value,                                              //金额
            },
        });
    }
}

//添加输出:
psbt.addOutput({
    address: toAddress,                                                         //目标地址
    value: 1198000,                                                             //转账金额（扣除手续费）
});

//对输入输入:
for (let i = 0; i < utxos.length; i++) {
    psbt.signInput(i, utxos[i].keyPair);
}

//验证签名:
for (let i = 0; i < utxos.length; i++) {
    psbt.validateSignaturesOfInput(i, validator);
}

//终结输入，产生最终的输入脚本:
psbt.finalizeAllInputs();

//提取最终的交易:
const tx = psbt.extractTransaction();

//输出交易的 Hex 编码:
console.log('交易 Hex:', tx.toHex());
```

最后得到的 hex 格式的交易就可以发布了。

2.5.1.2 Transaction 类

此类型代表一个交易事务，其定义如下:

```
export interface Output {
    script: Buffer;
    value: number;
}
export interface Input {
    hash: Buffer;
    index: number;
    script: Buffer;
    sequence: number;
    witness: Buffer[];
}

class Transaction {
    version: number = 1;
    locktime: number = 0;
    ins: Input[] = [];
    outs: Output[] = [];

    DEFAULT_SEQUENCE = 0xffffffff;
    SIGHASH_DEFAULT = 0x00;
```

```
    SIGHASH_ALL = 0x01;
    SIGHASH_NONE = 0x02;
    SIGHASH_SINGLE = 0x03;
    SIGHASH_ANYONECANPAY = 0x80;
    SIGHASH_OUTPUT_MASK = 0x03;
    SIGHASH_INPUT_MASK = 0x80;
    ADVANCED_TRANSACTION_MARKER = 0x00;
    ADVANCED_TRANSACTION_FLAG = 0x01;

    //1.实例化方法有三个: new Transaction()、fromBuffer()和fromHex()。
    constructor() { this.version = 1; this.locktime = 0; this.ins = []; this.outs = []; }
    static fromBuffer(buffer: Uint8Array, _NO_STRICT?: boolean): Transaction;
    static fromHex(hex: string): Transaction;

    //2.操纵输入:
    addInput(hash: Uint8Array, index: number, sequence?: number, scriptSig?: Uint8Array): number;
    setInputScript(index: number, scriptSig?: Uint8Array): void;

    //3.操纵输出:
    addOutput(scriptPubKey: Uint8Array, value: bigint): number;

    //4.操纵见证:
    setWitness(index: number, witness: Uint8Array[]): void;
    stripWitnesses(): void;
    hasWitnesses(): boolean;
    hashForWitnessV1(inIndex: number, prevOutScripts: Uint8Array[], values: bigint[], hashType: number,
leafHash?: Uint8Array, annex?: Uint8Array): Uint8Array;
    hashForWitnessV0(inIndex: number, prevOutScript: Uint8Array, value: bigint, hashType: number):
Uint8Array;
    getHash(forWitness?: boolean): Uint8Array;

    //5. 综合:
    clone(): Transaction;
    static isCoinbaseHash(buffer: Uint8Array): boolean;
    isCoinbase(buffer: Uint8Array): boolean;
    weight(): number;
    virtualSize(): number;
    byteLength(_ALLOW_WITNESS?: boolean = true): number;
    hashForSignature(inIndex: number, prevOutScript: Uint8Array, hashType: number): Uint8Array;
    getId(): string;
    toBuffer(buffer?: Uint8Array, initialOffset?: number): Uint8Array;
    toHex(): string;
}
```

它有四个动态属性：version、locktime、ins[]和 outs[]，分别代表版本、锁定时间、输入和输出，另外有一些静态属性用于定义默认值。

方法 fromHex()从原始交易数据解构成 Transaction 对象的实例，它实际上还是调用了 fromBuffer()方法。比如查询交易 id 为 95056a5c14fbd6f9d9a5491af96a909fb2d9a44acfd8f910cfc4584463d43526 的原始 hex 数据，可浏览 https://api.blockchair.com/bitcoin/raw/transaction/95056a5c14fbd6f9d9a5491af96a909fb2d9a44acfd8f910cfc4584463d43526 ，返回的字段 raw_transaction 的值就是 hex 格式的原始交易数据：

```
01000000000010150e74aa67fbda81bb142e5fac01f117b48de2f6bfa114ca4ba29d1f5c59a90a50000000000ffffffff02fc130000
000000001976a914c825a1ecf2a6830c4401620c3a16f1995057c2ab88acfe4b0000000000001600142566285e4349f21c0e07b08d
e695548628803ff302473044022048e2502d86ad5f6121c7c1bab8d025405282f13fb271e2a8c26b7623a7c5370f022040faa136af
9a61f0d10b3a65bd046e662a472efcd6efde32027d0d1ebb4c063d012102cde523c7f00a7056efec43735ce6e9fbfdd5753517d47e
7845f023cc9fc7112800000000
```

然后通过 Transaction.fromHex() 静态方法解构出交易实例。也可以直接访问 https://blockchain.info/rawtx/95056a5c14fbd6f9d9a5491af96a909fb2d9a44acfd8f910cfc4584463d43526?format=hex 获取交易的原始数据。

比如例子：

```
const bitcoin = require('bitcoinjs-lib');

async function getRawTR(tr_id) {
  let url = `https://blockchain.info/rawtx/${tr_id}?format=hex`;
```

```
  //let url = `https://api.blockchair.com/bitcoin/raw/transaction/${tr_id}`;
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Request failed with status ${response.status}: ${response.statusText}`);
    }
    const data = await response.text();                                    // 获取纯文本数据
    return data;                                                           // 返回数据
  } catch (error) {
    throw error;                                                           // 抛出错误，以便调用者可以处理
  }
}

const rawTR_hex = await getRawTR('95056a5c14fbd6f9d9a5491af96a909fb2d9a44acfd8f910cfc4584463d43526');
//let data_json = JSON.parse(rawTR_hex);

let tr = bitcoin.Transaction.fromHex(rawTR_hex);
console.log(tr)
Transaction {
  version: 1,
  locktime: 0,
  ins: [
    {
      hash: <Buffer 50 e7 4a a6 7f bd a8 1b b1 42 e5 fa c0 1f 11 7b 48 de 2f 6b fa 11 4c a4 ba 29 d1 f5 c5
9a 90 a5>,
      index: 0,
      script: <Buffer >,
      sequence: 4294967295,
      witness: [Array]
    }
  ],
  outs: [
    {
      value: 5116,
      script: <Buffer 76 a9 14 c8 25 a1 ec f2 a6 83 0c 44 01 62 0c 3a 16 f1 99 50 57 c2 ab 88 ac>
    },
    {
      value: 19454,
      script: <Buffer 00 14 25 66 28 5e 43 49 f2 1c 0e 07 b0 8d e6 95 54 86 28 80 3f f3>
    }
  ]
}
```

进一步查看第一个输入：

```
console.log(tr.ins[0])                                                     //查看第一个输入
{
  hash: <Buffer 50 e7 4a a6 7f bd a8 1b b1 42 e5 fa c0 1f 11 7b 48 de 2f 6b fa 11 4c a4 ba 29 d1 f5 c5 9a
90 a5>,
  index: 0,
  script: <Buffer >,
  sequence: 4294967295,
  witness: [
    <Buffer 30 44 02 20 48 e2 50 2d 86 ad 5f 61 21 c7 c1 ba b8 d0 25 40 52 82 f1 3f b2 71 e2 a8 c2 6b 76 23
a7 c5 37 0f 02 20 40 fa a1 36 af 9a 61 f0 d1 0b 3a 65 ... 21 more bytes>,
    <Buffer 02 cd e5 23 c7 f0 0a 70 56 ef ec 43 73 5c e6 e9 fb fd d5 75 35 17 d4 7e 78 45 f0 23 cc 9f c7 11
28>
  ]
}
```

输入中的 hash 就是交易 id，把它转换为十六进制，并恢复字节序：

```
let txid = tr.ins[0].hash.toString('hex').replace(/(.{2})/g, "$1 ").split(" ").reverse().join("")
console.log(txid);
a5909ac5f5d129baa44c11fa6b2fde487b111fc0fae542b11ba8bd7fa64ae750
```

可进一步查询交易 a5909ac5f5d129baa44c11fa6b2fde487b111fc0fae542b11ba8bd7fa64ae750 中的第一个输出。反汇编两个输出的输出脚本：

```
let sc1 = bitcoin.script.toASM(tr.outs[0].script);
console.log(sc1);
OP_DUP OP_HASH160 c825a1ecf2a6830c4401620c3a16f1995057c2ab OP_EQUALVERIFY OP_CHECKSIG
```

```
let sc2 = bitcoin.script.toASM(tr.outs[1].script);
console.log(sc2);
OP_0 2566285e4349f21c0e07b08de695548628803ff3
```

一看就知道第一个输出的地址类型为 P2PKH，第二个输出的地址类型为 P2WPKH。

一般不直接使用 Transaction 类，而是用 Psbt 类，Psbt 类会使用 Transaction 类。

2.5.1.3 Block 类

直接操纵原始区块，bitcoinjs-lib 的 Block 的结构定义如下：

```
class Block {
   version: number;
   prevHash?: Buffer;
   merkleRoot?: Buffer;
   timestamp: number;
   witnessCommit?: Buffer;
   bits: number;                                                    //紧凑格式的目标难度值
   nonce: number;                                                   //挖矿时不断变化此值来修改区块哈希值
   transactions?: Transaction[];

   static fromBuffer(buffer: Buffer): Block;
   static fromHex(hex: string): Block;

   static calculateTarget(bits: number): Buffer;                    //将紧凑格式的难度目标转换为完整的 256 位目标值
   static calculateMerkleRoot(transactions: Transaction[], forWitness?: boolean): Buffer;
   getWitnessCommit(): Buffer | null;
   hasWitnessCommit(): boolean;
   hasWitness(): boolean;
   weight(): number;                                                //计算当权区块的权重
   byteLength(headersOnly?: boolean, allowWitness?: boolean): number;
   getHash(): Buffer;
   getId(): string;                                                 //返回区块哈希值（反字节顺序）
   getUTCDate(): Date;
   toBuffer(headersOnly?: boolean): Buffer;
   toHex(headersOnly?: boolean): string;
   checkTxRoots(): boolean;
   checkProofOfWork(): boolean;                                     //工作量有效证明：区块哈希小于目标难度值为有效
   private __checkMerkleRoot;
   private __checkWitnessCommit;
}
```

实例化 Block 类有三种方法：

```
const b = new bitcoin.Block()
const b = bitcoin.Block.fromHex(<整个区块的 hex 编码>);
const b = bitcoin.Block.fromBuffer(<整个区块的 Buffer>);
```

fromHex()实际上还是调用了 fromBuffer()方法。

比如查询比特币区块 0000000000000bae09a7a393a8acded75aa67e46cb81f7acaa5ad94f9eacd103 的原始 hex 数据，可浏览 https://api.blockchair.com/bitcoin/raw/block/0000000000000bae09a7a393a8acded75aa67e46cb81f7acaa5ad94f9eacd103，其中的键 raw_block 的值就是区块原始数据，或者浏览 https://blockchain.info/rawblock/0000000000000bae09a7a393a8acded75aa67e46cb81f7acaa5ad94f9eacd103?format=hex，直接返回原始数据（hex 格式）：

```
010000008a730974ac39042e95f82d719550e224c1a680a8dc9e8df9d007000000000000f50b20e8720a552dd36eb2ebdb7dceec95
69e0395c990c1eb8a4292eeda05a931e1fce4e9a110e1a7a58aeb016010000000100000000000000000000000000000000000000
00000000000000000000000000ffffffff5370736a049a110e1a04b099a417522cfabe6d6d4e6988c831bb48c551eea50f87b3c6461a
……
78cd42d8f9ee022100e7644bb652ac32abddd883f16525444221b27aad7db8f4db7c255a3c898e9b8a014104d868a086c096b7e99d
f70330cbb11c133ec52ff9dd0ef43734490b7929bcdeeba02274ef941aa37c2e15c50df9ad9645ad737b0b73f1b144d3e51431df22
```

```
f3affffffff0240420f00000000001976a9147bd51fae7a39fe2b417498b7dbe297a887cdefbd88ac700c6a00000000001976a914
ef48d8584b96d95992a664d524e52007b036754188ac00000000
```

然后通过 Block.fromHex()方法解码出结构。去掉 URL 中的“?format=hex”则返回 json 格式的数据。用 js 代码实现：

```
const bitcoin = require('bitcoinjs-lib');

async function getRawBlock(block_id) {
  let url = `https://blockchain.info/rawblock/${block_id}?format=hex`;
  //let url = `https://api.blockchair.com/bitcoin/raw/block/${block_id}`;
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Request failed with status ${response.status}: ${response.statusText}`);
    }
    const data = await response.text();                                    // 获取纯文本数据
    return data;                                                           // 返回数据
  } catch (error) {
    throw error;                                                           // 抛出错误，以便调用者可以处理
  }
}

let rawBlockdata = await getRawBlock('0000000000000bae09a7a393a8acded75aa67e46cb81f7acaa5ad94f9eacd103');
```

拿到了一个区块的元素数据，就可以通过 Block.fromHex()方法实例化 Block 类，代码如下：

```
var bl = bitcoin.Block.fromHex(rawBlockdata);
console.log(bl)
Block {
  version: 1,
  prevHash: <Buffer 8a 73 09 74 ac 39 04 2e 95 f8 2d 71 95 50 e2 24 c1 a6 80 a8 dc 9e 8d f9 d0 07 00 00 00
00 00 00>,
  merkleRoot: <Buffer f5 0b 20 e8 72 0a 55 2d d3 6e b2 eb db 7d ce ec 95 69 e0 39 5c 99 0c 1e b8 a4 29 2e
ed a0 5a 93>,
  timestamp: 1322131230,
  witnessCommit: undefined,
  bits: 437129626,
  nonce: 2964215930,
  transactions: [
    Transaction {
      version: 1,
      locktime: 0,
      ins: [Array],
      outs: [Array]
    },
    Transaction {
      version: 1,
      locktime: 0,
      ins: [Array],
      outs: [Array]
    },
    Transaction {
      version: 1,
      locktime: 0,
      ins: [Array],
      outs: [Array]
    },
    Transaction {
      version: 1,
      locktime: 0,
      ins: [Array],
      outs: [Array]
    },
    Transaction {
      version: 1,
      locktime: 0,
      ins: [Array],
      outs: [Array]
    },
    Transaction {
      version: 1,
      locktime: 0,
      ins: [Array],
      outs: [Array]
    },
    ......                                                                 //省略了 17 个事务
  ]
}
```

prevHash 就是上一个区块的 Id，转换为正常顺序的十六进制数：

```
bl.prevHash.toString('hex').replace(/(.{2})/g, "$1 ").split(" ").reverse().join("")
00000000000007d0f98d9edca880a6c124e25095712df8952e0439ac7409738a
```

区块中的一个交易就是一个 Transaction 实例，可进一步查看交易实例中的输入：

```
console.log(bl.transactions[0].ins[0]);
{
  hash: <Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00>,
  index: 4294967295,
  script: <Buffer 70 73 6a 04 9a 11 0e 1a 04 b0 99 a4 17 52 2c fa be 6d 6d 4e 69 88 c8 31 bb 48 c5 51 ee
a5 0f 87 b3 c6 46 1a de 47 6f e1 5c 98 be d7 c6 a5 74 ac a4 ff ... 33 more bytes>,
  sequence: 4294967295,
  witness: []
}
```

再反汇编第一个输出的输出脚本看看是什么样子：

```
let script_asm = bitcoin.script.toASM(bl.transactions[0].outs[0].script);
console.log(script_asm);
OP_DUP OP_HASH160 ca975b00a8c203b8692f5a18d92dc5c2d2ebc57b OP_EQUALVERIFY OP_CHECKSIG
```

显然转入地址就是 P2PKH 类型的地址（可参考“§4.5.2 解锁过程”）。

Block 类的静态方法 calculateTarget(bits: number): Buffer - 根据紧凑格式的难度目标 bits 转换为 256 位的真实难度目标，难度目标确定挖矿难度，难度目标值越小挖矿难度就越大——新区块的 hash 值必须小于等于难度目标才算挖矿成功。刚才解构出来的区块的 bits 值为 437129626，转化为十六进制就是 1a0e119a，它由四个字节组成，第一个字节为指数，剩下三个字节为系数，转换为真实难度目标的过程参考图 2-23。

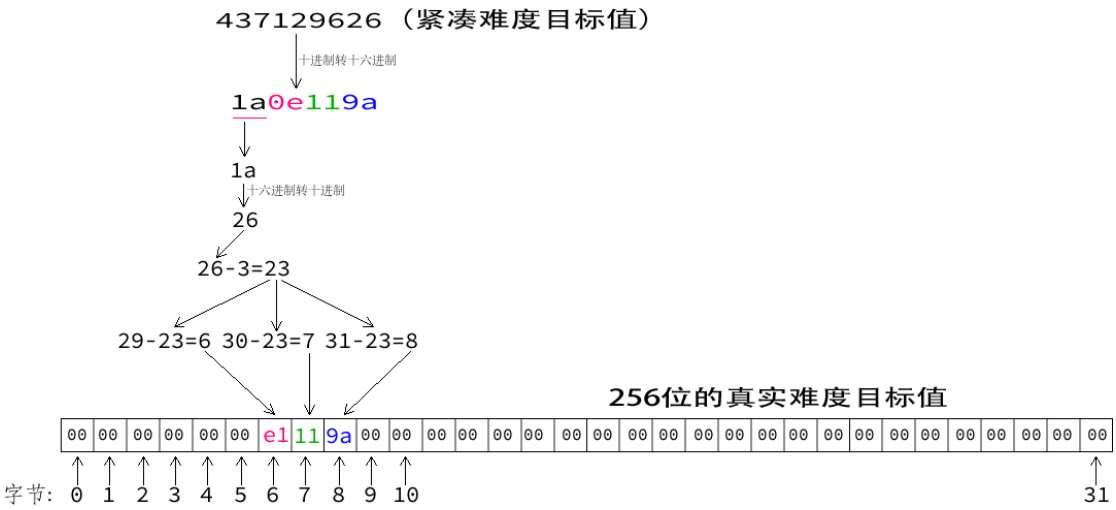


图 2-23 由紧凑难度目标转化为真实难度目标

看例子：

```
> bitcoin.Block.calculateTarget(0x1a558899)
<Buffer 00 00 00 00 00 00 00 55 88 99 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00>
> bitcoin.Block.calculateTarget(0x1a558899).toString('hex')
'0000000000005588990000000000000000000000000000000000000000000000'
```

方法 weight(): number - 计算区块的权重，计算公式为：Weight=(Non-witness bytes×4)+Witness bytes，即见证数据每个字节权重为 1，区块中的其他数据每个字节权重为 4。比特币网络的区块大小限制是基于权重而不是原始字节数的，目前比特币网络的区块权重上限为 4,000,000 权重单位，这大约相当于 1MB 的非见证数据和 3MB 的见证数据。

方法 byteLength(headersOnly?: boolean, allowWitness?: boolean): number - 方法用于计算比特币区块的原始字节长度，它

返回的是区块在序列化后的总字节数。

方法 getHash(): Buffer - 返回区块的哈希值。这个哈希值是通过对区块头进行两次 SHA-256 哈希计算得到的。

方法 getId(): String - 也返回区块的哈希值，但它的返回值是一个十六进制字符串，并且会反转字节顺序。其实由等式：getId() = getHash().toString('hex').replace(/(.{2})/g, "$1 ").split(" ").reverse().join("")。getId()的返回值作为区块的唯一 id。

方法 checkProofOfWork(): boolean - 验证区块的工作量证明是否有效。它会检查区块的哈希值是否小于或等于难度目标值，比如上面的例子，难度目标值为 0x0000000000000e119a0000000000000000000000000000000000000000000000，而区块的哈希值（getId()方法的返回值）为 0x0000000000000bae09a7a393a8acded75aa67e46cb81f7acaa5ad94f9eacd103，显然要小于难度目标值，所以 checkProofOfWork()方法返回 true。

2.5.1.4 script 对象

Script 对象提供诸如反编译、编译、与 ASM 相互转换、堆栈作和脚本验证功能等实用程序。定义如下：

```
script: {                                                                    //本模块操纵脚本语言(锁定和输入脚本要用到)
   OPS={ OP_FALSE: 0, OP_0: 0, OP_PUSHDATA1: 76, …},                                  //包含各种操作码的对象
   number: { decode(buffer: Buffer, maxLength?: number, minimal?: boolean): number, encode(_number:
number): Buffer },
   signature: { decode: [Function: decode], encode: [Function: encode] },
   isPushOnly(value: Stack): boolean,                                          //判断堆栈中是否只存在压栈操作码
   compile(chunks: Uint8Array | Stack): Uint8Array,                          //将脚本块数组编译为 Uint8Array
   countNonPushOnlyOPs(value: Stack): number,                          //统计堆栈中除了压栈操作码之外的操作码数量
   decompile(buffer): (number | Uint8Array)[] | null,                        //将脚本缓冲区反编译为块数组
   fromASM(asm: string): Uint8Array,                                                                  //汇编
   isCanonicalPubKey(buffer: Uint8Array): boolean,                                          //是否是规范的公钥
   isDefinedHashType(hashType: number): boolean,                                            //哈希类型是否存在
   isCanonicalScriptSignature(buffer: Uint8Array): boolean,                        //是否是规范的签名脚本
   toASM(chunks: Uint8Array | (number | Uint8Array)[]): string,                                       //反汇编
   toStack(chunks: Uint8Array | (number | Uint8Array)[]): Uint8Array[]
}
```

常量对象 OPS - 包含了各种脚本操作码，如下所示：

```
bitcoin.script.OPS = {
  OP_FALSE: 0,
  OP_0: 0,
  OP_PUSHDATA1: 76,
  OP_PUSHDATA2: 77,
  OP_PUSHDATA4: 78,
  OP_1NEGATE: 79,
  OP_RESERVED: 80,
  OP_TRUE: 81,
  OP_1: 81,
  OP_2: 82,
  OP_3: 83,
  OP_4: 84,
  OP_5: 85,
  OP_6: 86,
  OP_7: 87,
  OP_8: 88,
  OP_9: 89,
  OP_10: 90,
  OP_11: 91,
  OP_12: 92,
  OP_13: 93,
  OP_14: 94,
  OP_15: 95,
  OP_16: 96,
  OP_NOP: 97,
```

```
OP_VER: 98,
OP_IF: 99,
OP_NOTIF: 100,
OP_VERIF: 101,
OP_VERNOTIF: 102,
OP_ELSE: 103,
OP_ENDIF: 104,
OP_VERIFY: 105,
OP_RETURN: 106,
OP_TOALTSTACK: 107,
OP_FROMALTSTACK: 108,
OP_2DROP: 109,
OP_2DUP: 110,
OP_3DUP: 111,
OP_2OVER: 112,
OP_2ROT: 113,
OP_2SWAP: 114,
OP_IFDUP: 115,
OP_DEPTH: 116,
OP_DROP: 117,
OP_DUP: 118,
OP_NIP: 119,
OP_OVER: 120,
OP_PICK: 121,
OP_ROLL: 122,
OP_ROT: 123,
OP_SWAP: 124,
OP_TUCK: 125,
OP_CAT: 126,
OP_SUBSTR: 127,
OP_LEFT: 128,
OP_RIGHT: 129,
OP_SIZE: 130,
OP_INVERT: 131,
OP_AND: 132,
OP_OR: 133,
OP_XOR: 134,
OP_EQUAL: 135,
OP_EQUALVERIFY: 136,
OP_RESERVED1: 137,
OP_RESERVED2: 138,
OP_1ADD: 139,
OP_1SUB: 140,
OP_2MUL: 141,
OP_2DIV: 142,
OP_NEGATE: 143,
OP_ABS: 144,
OP_NOT: 145,
OP_0NOTEQUAL: 146,
OP_ADD: 147,
OP_SUB: 148,
OP_MUL: 149,
OP_DIV: 150,
OP_MOD: 151,
OP_LSHIFT: 152,
OP_RSHIFT: 153,
OP_BOOLAND: 154,
OP_BOOLOR: 155,
OP_NUMEQUAL: 156,
OP_NUMEQUALVERIFY: 157,
OP_NUMNOTEQUAL: 158,
OP_LESSTHAN: 159,
OP_GREATERTHAN: 160,
OP_LESSTHANOREQUAL: 161,
OP_GREATERTHANOREQUAL: 162,
OP_MIN: 163,
OP_MAX: 164,
OP_WITHIN: 165,
OP_RIPEMD160: 166,
OP_SHA1: 167,
OP_SHA256: 168,
OP_HASH160: 169,
OP_HASH256: 170,
OP_CODESEPARATOR: 171,
```

```
  OP_CHECKSIG: 172,
  OP_CHECKSIGVERIFY: 173,
  OP_CHECKMULTISIG: 174,
  OP_CHECKMULTISIGVERIFY: 175,
  OP_NOP1: 176,
  OP_NOP2: 177,
  OP_CHECKLOCKTIMEVERIFY: 177,
  OP_NOP3: 178,
  OP_CHECKSEQUENCEVERIFY: 178,
  OP_NOP4: 179,
  OP_NOP5: 180,
  OP_NOP6: 181,
  OP_NOP7: 182,
  OP_NOP8: 183,
  OP_NOP9: 184,
  OP_NOP10: 185,
  OP_CHECKSIGADD: 186,
  OP_PUBKEYHASH: 253,
  OP_PUBKEY: 254,
  OP_INVALIDOPCODE: 255
}
```

静态方法 number.encode(_number: number): Buffer 将一个整数编码为比特币脚本中的字节序列（小端存储），静态方法 number.decode(buffer: Buffer, maxLength?: number, minimal?: boolean): number 将比特币脚本中的字节序列解码为一个整数，参数 buffer 为小端存储的字节序列，maxLength 最大允许的字节长度，默认为 4 字节，参数 minimal 是否要求字节序列是最小表示形式（无前导零），默认为 true。如：

```
const bitcoin = require('bitcoinjs-lib');
const script = bitcoin.script;
const encoded = script.number.encode(123456);
const decoded1 = script.number.decode(encoded);
const i = Buffer.from('342100', 'hex');
const decoded2 = script.number.decode(i, null,false);
console.log(`${encoded.toString('hex')}--${decode1}--${decoded2}`);
40e201--123456--8500
```

方法 fromASM(asm: string): Uint8Array 编译汇编脚本，如下面的例子：

```
const bitcoin = require('bitcoinjs-lib');

const publicKey1 = '03cd8090fe7dea68fc8f50f8df45b354d3734cc6a753c0699812da43353bc865a2';
const publicKey2 = '03b1fbff0ee0f3aa5b2623ca8edc251e2aa740c53a56706fe274a13ef217ac039c';
let script_hex = bitcoin.script.fromASM(
    `
     OP_IF
        ${bitcoin.script.number.encode(5000).toString('hex')}
        OP_CHECKSEQUENCEVERIFY
        OP_DROP
     OP_ELSE
        ${publicKey1}
        OP_CHECKSIGVERIFY
     OP_ENDIF
     ${publicKey2}
     OP_CHECKSIG
    `.trim().replace(/\s+/g, ' '),
);
console.log(script_hex.toString('hex'));
63028813b275672103cd8090fe7dea68fc8f50f8df45b354d3734cc6a753c0699812da43353bc865a2ad682103b1fbff0ee0f3aa5b
2623ca8edc251e2aa740c53a56706fe274a13ef217ac039cac

const script_asm = bitcoin.script.toASM(Buffer.from(script_hex,'hex'));                    //反汇编
console.log(script_asm);
OP_IF 8813 OP_CHECKSEQUENCEVERIFY OP_DROP OP_ELSE
03cd8090fe7dea68fc8f50f8df45b354d3734cc6a753c0699812da43353bc865a2 OP_CHECKSIGVERIFY OP_ENDIF
03b1fbff0ee0f3aa5b2623ca8edc251e2aa740c53a56706fe274a13ef217ac039c OP_CHECKSIG
```

2.5.1.5 address 对象

bitcoinjs-lib.address 对象完成比特币地址的编码与译码、以及从地址生成输出脚本、或从输出脚本算出地址等任何，此对象定义如下：

```
/** base58check 解码结果（专门解码 P2PKH 和 P2SH 类型的地址）*/
export interface Base58CheckResult {
  /** 地址的哈希值*/
  hash: Uint8Array;
  /** 地址的版本：0x00 表示 P2PKH，0x05 表示 P2SH */
  version: number;
}

/** bech32 解码结果（专门解码 P2WPKH、P2WSH 和 P2TR 类型的地址）*/
export interface Bech32Result {
  /** 地址版本：0x00 表示 P2WPKH、P2WSH，0x01 表示 P2TR*/
  version: number;
  /** 地址前缀：bc 表示 P2WPKH、P2WSH、P2TR ，tb 为测试网*/
  prefix: string;
  /** 地址的哈希值(PWPKH、P2WSH)、Taproot 公钥（P2TR）：20 个字节表示 P2WPKH，32 个字节表示 P2WSH、P2TR */
  data: Uint8Array;
}

fromBase58Check(address: string): Base58CheckResult;                    //解码 Base58 编码的地址：P2PKH、P2SH
fromBech32(address: string): Bech32Result;                          //解码 Bech32 编码的地址：P2WPKH、P2WSH、P2TR
fromOutputScript(output: Uint8Array, network?: Network): string;                          //从输出脚本导出地址
toBase58Check(hash: Uint8Array, version: number): string;                  //把 hash 编码成 Base58，并附加版本
toBech32(data: Uint8Array,  version: number,  prefix: string): string;      //编码成 Bech32（版本=0）或 Bech32m
toOutputScript(address: string, network?: Network): Uint8Array;                           //把地址转换成输出脚本
```

发起交易时，对于输出，其中一项就是根据地址类型填写输出脚本，这可以采用方法 bitcoinjs-lib.address.toOutputScript() 完成，它会根据地址类型产生输出脚本，如：

```
const outputScript = bitcoin.address.toOutputScript('1PoH4n8uigob9UYR1Sjv8Eikymxmb6f5Mg');
const script_hex = outputScript.reduce((str, byte) => str + byte.toString(16).padStart(2, '0'), '');
console.log(script_hex)
76a914fa134fde9848d655806666e31f99eb1470cf579888ac
```

我们可运行 bitcoinjs-lib.script.toASM()方法反汇编刚刚产生的输出脚本：

```
let asm = bitcoin.script.toASM(outputScript);
console.log(asm);
OP_DUP OP_HASH160 fa134fde9848d655806666e31f99eb1470cf5798 OP_EQUALVERIFY OP_CHECKSIG
```

当然也可以编译汇编脚本，如：

```
bitcoin.script.fromASM('OP_0 fa134fde9848d655806666e31f99eb1470cf5798');
<Buffer 00 14 fa 13 4f de 98 48 d6 55 80 66 66 e3 1f 99 eb 14 70 cf 57 98>
```

2.5.1.6 payments 对象

创建和管理比特币的支付对象，支持的地址类型有 P2PKH (Pay-to-PubKey-Hash)、P2SH (Pay-to-Script-Hash)、P2WPKH (Pay-to-Witness-PubKey-Hash)、P2WSH (Pay-to-Witness-Script-Hash)和 P2TR (Taproot)。

Payments 对象的定义如下：

```
interface Payment {
   name?: string;
   network?: Network;
   output?: Buffer;                                                                                    //输出脚本
   data?: Buffer[];
```

```
    m?: number;
    n?: number;
    pubkeys?: Buffer[];
    input?: Buffer;
    signatures?: Buffer[];
    internalPubkey?: Buffer;
    pubkey?: Buffer;
    signature?: Buffer;
    address?: string;
    hash?: Buffer;
    redeem?: Payment;
    redeemVersion?: number;
    scriptTree?: Taptree;
    witness?: Buffer[];
}
type PaymentCreator = (a: Payment, opts?: PaymentOpts) => Payment;

type PaymentFunction = () => Payment;

interface PaymentOpts {
  validate?: boolean;
  allowIncomplete?: boolean;
}

type StackElement = Uint8Array | number;
type Stack = StackElement[];
type StackFunction = () => Stack;

type Taptree = [Taptree | Tapleaf, Taptree | Tapleaf] | Tapleaf;
interface Tapleaf {
    output: Buffer;
    version?: number;                                                    //默认为 192
 (0xc0)
}

//创建一个 pay-to-witness-public-key-hash (p2wpkh) 支付对象。a.address、a.hash、a.output、a.pubkey 和 a.witness
至少提供一个:
p2wpkh(a: Payment, opts?: PaymentOpts): Payment;

//创建一个 Pay-to-Public-Key-Hash (P2PKH)支付对象。a.address、a.hash、a.output、a.pubkey 和 a.input 至少提供一个:
p2pkh(a: Payment, opts?: PaymentOpts): Payment;

//创建一个 Pay-to-Taproot (P2TR) 支付对象。a.address、a.hash、a.output、a.pubkey、a.internalPubkey 和 a.witness
至少提供一个:
p2tr(a: Payment, opts?: PaymentOpts): Payment;

//创建一个 Pay-to-Script-Hash (P2SH) 支付对象。a.address、a.hash、a.output、a.redeem 和 a.input 至少提供一个:
p2sh(a: Payment, opts?: PaymentOpts): Payment;

//创建一个 Pay-to-Witness-Script-Hash (P2WSH) 支付对象。a.address、a.hash、a.output、a.redeem 和 a.witness 至少提
供一个:
p2wsh(a: Payment, opts?: PaymentOpts): Payment;

//创建一个 Pay-to-Multisig (P2MS)支付对象。要求(!a.input && !a.output &&!(a.pubkeys && a.m !== undefined) &&
!a.signatures)条件为假:
p2ms(a: Payment, opts?: PaymentOpts): Payment;

//创建一个 pay-to-public-key (P2PK)支付对象。要求(!a.input && !a.output && !a.pubkey && !a.signature)条件为假:
p2pk(a: Payment, opts?: PaymentOpts): Payment;

//在支付对象中嵌入数据。要求(!a.data && !a.output)条件为假:
embed(a: Payment, opts?: PaymentOpts): Payment;
```

2.5.1.7 crypto 对象

本对象包含一些哈希计算函数，如 hash160()、hash256()等。crypto 对象定义如下:

```
const TAGS = [
```

```
  'BIP0340/challenge',
  'BIP0340/aux',
  'BIP0340/nonce',
  'TapLeaf',
  'TapBranch',
  'TapSighash',
  'TapTweak',
  'KeyAgg list',
  'KeyAgg coefficient',
];

const TAGGED_HASH_PREFIXES: TaggedHashPrefixes = {
  'BIP0340/challenge': Uint8Array.from([
    123, 181, 45, 122, 159, 239, 88, 50, 62, 177, 191, 122, 64, 125, 179, 130,
    210, 243, 242, 216, 27, 177, 34, 79, 73, 254, 81, 143, 109, 72, 211, 124,
    123, 181, 45, 122, 159, 239, 88, 50, 62, 177, 191, 122, 64, 125, 179, 130,
    210, 243, 242, 216, 27, 177, 34, 79, 73, 254, 81, 143, 109, 72, 211, 124,
  ]),
  'BIP0340/aux': Uint8Array.from([
    241, 239, 78, 94, 192, 99, 202, 218, 109, 148, 202, 250, 157, 152, 126, 160,
    105, 38, 88, 57, 236, 193, 31, 151, 45, 119, 165, 46, 216, 193, 204, 144,
    241, 239, 78, 94, 192, 99, 202, 218, 109, 148, 202, 250, 157, 152, 126, 160,
    105, 38, 88, 57, 236, 193, 31, 151, 45, 119, 165, 46, 216, 193, 204, 144,
  ]),
  'BIP0340/nonce': Uint8Array.from([
    7, 73, 119, 52, 167, 155, 203, 53, 91, 155, 140, 125, 3, 79, 18, 28, 244,
    52, 215, 62, 247, 45, 218, 25, 135, 0, 97, 251, 82, 191, 235, 47, 7, 73,
    119, 52, 167, 155, 203, 53, 91, 155, 140, 125, 3, 79, 18, 28, 244, 52, 215,
    62, 247, 45, 218, 25, 135, 0, 97, 251, 82, 191, 235, 47,
  ]),
  TapLeaf: Uint8Array.from([
    174, 234, 143, 220, 66, 8, 152, 49, 5, 115, 75, 88, 8, 29, 30, 38, 56, 211,
    95, 28, 181, 64, 8, 212, 211, 87, 202, 3, 190, 120, 233, 238, 174, 234, 143,
    220, 66, 8, 152, 49, 5, 115, 75, 88, 8, 29, 30, 38, 56, 211, 95, 28, 181,
    64, 8, 212, 211, 87, 202, 3, 190, 120, 233, 238,
  ]),
  TapBranch: Uint8Array.from([
    25, 65, 161, 242, 229, 110, 185, 95, 162, 169, 241, 148, 190, 92, 1, 247,
    33, 111, 51, 237, 130, 176, 145, 70, 52, 144, 208, 91, 245, 22, 160, 21, 25,
    65, 161, 242, 229, 110, 185, 95, 162, 169, 241, 148, 190, 92, 1, 247, 33,
    111, 51, 237, 130, 176, 145, 70, 52, 144, 208, 91, 245, 22, 160, 21,
  ]),
  TapSighash: Uint8Array.from([
    244, 10, 72, 223, 75, 42, 112, 200, 180, 146, 75, 242, 101, 70, 97, 237, 61,
    149, 253, 102, 163, 19, 235, 135, 35, 117, 151, 198, 40, 228, 160, 49, 244,
    10, 72, 223, 75, 42, 112, 200, 180, 146, 75, 242, 101, 70, 97, 237, 61, 149,
    253, 102, 163, 19, 235, 135, 35, 117, 151, 198, 40, 228, 160, 49,
  ]),
  TapTweak: Uint8Array.from([
    232, 15, 225, 99, 156, 156, 160, 80, 227, 175, 27, 57, 193, 67, 198, 62, 66,
    156, 188, 235, 21, 217, 64, 251, 181, 197, 161, 244, 175, 87, 197, 233, 232,
    15, 225, 99, 156, 156, 160, 80, 227, 175, 27, 57, 193, 67, 198, 62, 66, 156,
    188, 235, 21, 217, 64, 251, 181, 197, 161, 244, 175, 87, 197, 233,
  ]),
  'KeyAgg list': Uint8Array.from([
    72, 28, 151, 28, 60, 11, 70, 215, 240, 178, 117, 174, 89, 141, 78, 44, 126,
    215, 49, 156, 89, 74, 92, 110, 199, 158, 160, 212, 153, 2, 148, 240, 72, 28,
    151, 28, 60, 11, 70, 215, 240, 178, 117, 174, 89, 141, 78, 44, 126, 215, 49,
    156, 89, 74, 92, 110, 199, 158, 160, 212, 153, 2, 148, 240,
  ]),
  'KeyAgg coefficient': Uint8Array.from([
    191, 201, 4, 3, 77, 28, 136, 232, 200, 14, 34, 229, 61, 36, 86, 109, 100,
    130, 78, 214, 66, 114, 129, 192, 145, 0, 249, 77, 205, 82, 201, 129, 191,
    201, 4, 3, 77, 28, 136, 232, 200, 14, 34, 229, 61, 36, 86, 109, 100, 130,
    78, 214, 66, 114, 129, 192, 145, 0, 249, 77, 205, 82, 201, 129,
  ]),
};

ripemd160(buffer: Buffer): Buffer;
sha1(buffer: Buffer): Buffer;
sha256(buffer: Buffer): Buffer;
hash160(buffer: Uint8Array): Uint8Array { return ripemd160(sha256(buffer)); }
hash256(buffer: Uint8Array): Uint8Array { return sha256(sha256(buffer)); }
taggedHash(prefix: TaggedHashPrefix, data: Uint8Array): Uint8Array {
```

```
  return sha256(Buffer.concat([TAGGED_HASH_PREFIXES[prefix], data]));
}
```

看例子:

```
const bitcoin = require('bitcoinjs-lib');
let s = Buffer.from('L12haRaMNv49bvSgMUgM1LG6G6VjCt3fXDd6VqbH6ijra5H4dNh9');
let e = bitcoin.crypto.hash160(s);
let h = bitcoin.crypto.taggedHash('TapLeaf',s);
```

2.5.2 tiny-secp256k1

tiny-secp256k1 是一个轻量级的 C 库，专门用于实现比特币和其他加密货币中使用的 secp256k1 椭圆曲线的密码学操作。它是 secp256k1 的一个精简版本，支持椭圆曲线数字签名（ECDSA）和 施诺尔（Schnorr）签名。实现类似功能的库还有 secp256k1、@bitcoinerlab/secp256k1、libsecp256k1 和@noble/secp256k1 等。

准备工作：

安装 nodejs 后再：

```
npm install tiny-secp256k1
node
> const secp256k1 = require('tiny-secp256k1');
```

一、公钥操作

1. isPoint(A:Buffer): boolean- 判断 A 是不是一个合法公钥。合法的公钥要求：以 0x02、0x03 或 0x04 开头，x 和 y 坐标取值范围为[1, p-1]，p 是椭圆曲线的阶（0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141）。

```
secp256k1.isPoint(Buffer.from('02c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d','hex'))
;
true
```

2. isPointCompressed (A: Buffer): boolean- 判断 A 是不是一个压缩公钥。压缩公钥以 0x02 或 0x03 开头，总长度为 33 个字节。

3. isXOnlyPoint(A: Buffer): boolean - 如果 A 是一个只有 x 坐标的公钥，则返回 true，否则返回 false。

4. pointCompress(A: Buffer, compressed?: boolean): Uint8Array - 如果 compressed 为 true，则把非压缩公钥 A 转换为压缩公钥，compressed 参数可省略，默认为 false。

```
const public =
'04bddbe897446200485b18be4ecfe4b9d4749a269ebef434460d4b4d058414e6a43f6a9d0f9c12b0a74e772f676035c5e99d6005a
f6134100619d75c9e7506b37a';
const publicCompress = secp256k1.pointCompress(Buffer.from(public,'hex'),true);
console.log(Buffer.from(publicCompress).toString('hex'));
02bddbe897446200485b18be4ecfe4b9d4749a269ebef434460d4b4d058414e6a4
```

5. pointAdd(A:Buffer, B:Buffer, compressed?: boolean): Uint8Array - 两个公钥相加，参数 compress 可以省略，默认值为 true。

```
const A = Buffer.from('02770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1','hex');
const B = Buffer.from('02c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d','hex');
const C = secp256k1.pointAdd(A, B, true);
console.log(Buffer.from(C).toString('hex'))
02b7277afca4430567d388793561e4b4e40fbe4887c7f85c5c5beb75af2c013e91
```

6. xOnlyPointFromPoint(A: Buffer): Uint8Array - 获取公钥的 x 坐标，参数 A 为一个合法的公钥。

7. xOnlyPointFromScalar(d: Buffer): Uint8Array - 从私钥 d 推导出公钥，这个公钥只有 x 坐标。d 必须 32 个字节长，且取值范围是[1, p-1]。

```
const d = Buffer.from('6268f766e57405a96c837625c1926045aaabd0f0afd7130635534ab7d561d371','hex');
const public = secp256k1.xOnlyPointFromScalar(d);
console.log(Buffer.from(public).toString('hex'))
bddbe897446200485b18be4ecfe4b9d4749a269ebef434460d4b4d058414e6a4
```

8. pointFromScalar(d: Buffer , compressed?: boolean): Uint8Array - 从私钥 d 推导出压缩公钥（compressed=true）或者非压缩公钥（compressed=false），compressed 默认为 true。

9. pointAddScalar(A: Uint8Array, tweak: Uint8Array, compressed?: boolean): Uint8Array - 公钥 A 加一个标量 tweak，tweak

取值范围是[0,p-1]，compressed 默认值为 true——默认返回压缩公钥。

10. pointMultiply(A: Uint8Array, tweak: Uint8Array, compressed?: boolean): Uint8Array - 公钥 A 乘以标量 tweak，返回压缩公钥（compressed=true）或非压缩公钥（compressed=false）。如果省略 compressed 参数，默认为 true。

11. xOnlyPointAddTweak(A: Uint8Array, tweak: Uint8Array): XOnlyPointAddTweakResult - 只有 x 坐标的公钥 A 加上标量 tweak，返回接口对象 XonlyPointAddTweakResult，此接口对象定义为 XOnlyPointAddTweakResult { parity: 1 | 0; xOnlyPubkey: Uint8Array;}，如果公钥的 y 坐标为奇数，那么 parity 等于 1，否则等于 0。

```
const A = Buffer.from('02770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1','hex');
const tweak = Buffer.from('6268f766e57405a96c837625c1926045aaabd0f0afd7130635534ab7d561d371','hex');
const result = secp256k1.xOnlyPointAddTweak(secp256k1.xOnlyPointFromPoint(A),tweak);
//result 的值为:
{
  parity: 1,
  xOnlyPubkey: Uint8Array(32) [
    252, 121, 172,  87, 255, 235,  67,   0,
    165, 237,  64,  71, 227, 123, 113, 120,
    198, 167, 233, 106,  44, 145,  52, 102,
    112, 205, 245, 178, 226,  84,   0, 168
  ]
}
```

12. xOnlyPointAddTweakCheck(A: Uint8Array, tweak: Uint8Array, B: Uint8Array, tweakParity?: TweakParity): boolean - 验证 B 是不是经 A 加上 tweak 而得到的，且满足 tweakParity 指定的奇偶校验要求（如果参数 tweakParity 没省略）。TweakParity 取值 1 或 0。

二、私钥操作

1. isPrivate(d: Buffer): boolean - 判断 d 是不是一个合法私钥：32 个字节长，且落在[1,p-1]区间。

2. privateNegate(d: Buffer): Uint8Array - 返回私钥 d 的模（即 p-d）。私钥和它的模都是合法的私钥，且它们的公钥具有相等的 x 坐标，但是 y 坐标具有不同的奇偶性（一个为奇数，另一个为偶数）。看例子：

```
const d = Buffer.from('6268f766e57405a96c837625c1926045aaabd0f0afd7130635534ab7d561d371','hex');
Buffer.from(secp256k1.pointFromScalar(d)).toString('hex');
//返回'02bddbe897446200485b18be4ecfe4b9d4749a269ebef434460d4b4d058414e6a4'
Buffer.from(secp256k1.pointFromScalar(secp256k1.privateNegate(d))).toString('hex');
//返回'03bddbe897446200485b18be4ecfe4b9d4749a269ebef434460d4b4d058414e6a4'
```

3. privateAdd(d: Uint8Array, tweak: Uint8Array): Uint8Array - 私钥 d 加上标量 tweak。参数 d 为私钥，tweak 落在[0,p-1]区间的标量。

4. privateSub(d: Uint8Array, tweak: Uint8Array): Uint8Array|null - 私钥 d 减去标量 tweak，返回私钥或 null（差为 0 时）。

三、签名操作

1. signSchnorr(h: Uint8Array, d: Uint8Array, e?: Uint8Array): Uint8Array - 用私钥 d 对 h 进行斯诺尔签名，并返回签名。h 参数为 32 个字节长度的标量，可选参数 e 为 32 个字节长度的数，作为签名的商。看例子：

```
const bitcoin = require('bitcoinjs-lib');

const msg = '她嫣然含笑，春光无限，娇美之状不可方物。';
const h = bitcoin.crypto.sha256(msg);
const d = Buffer.from('6268f766e57405a96c837625c1926045aaabd0f0afd7130635534ab7d561d371','hex');
const signature = secp256k1.signSchnorr(h, d);                                    //返回的签名如下
Uint8Array(64) [
   87, 224,  65, 218, 197, 101, 223,  98, 129, 102, 220,
  150, 105,  65, 215, 104, 213,  57, 147, 219,  33, 254,
  153,  76,  77, 143, 198,   8, 252, 177,  67,  50, 179,
  232, 132, 175,  41, 202,  98,   8, 230, 228, 254, 134,
   52,   2, 174, 248, 189, 202, 170, 194, 164, 147,  89,
   89,  51, 246, 166,  70,  31,  34, 182,  91
```

```
]
```

2. verifySchnorr(h: Uint8Array, Q: Uint8Array, signature: Uint8Array): boolean - 用公钥 Q 验证对 h 的签名 signature。注意，Q 只有 x 坐标，长度为 32 个字节。看例子：

```
//继续上面的代码
const Q = secp256k1.xOnlyPointFromScalar(d);
secp256k1.verifySchnorr(h, Q, signature);                                    //返回 true
```

3. sign(h: Uint8Array, d: Uint8Array, e?: Uint8Array): Uint8Array - 用私钥 d 对 h 进行 ECDSA 签名，参数介绍参考 signSchnorr()方法。

4. verify(h: Uint8Array, Q: Uint8Array, signature: Uint8Array, strict?: boolean): boolean - 用公钥 Q 验证对 h 的签名 signature。可选参数 strict 默认为 false，如果为 true，那么签名中的(r,s)必须小于 p/2。注意 Q 不能为只有 x 坐标的公钥，即采用 const Q = secp256k1.pointFromScalar(d)。

5. signRecoverable(h: Uint8Array, d: Uint8Array, e?: Uint8Array): RecoverableSignature - 用私钥 d 对 h 进行 ECDSA 签名，并返回 RecoverableSignature 接口对象。定义 RecoverableSignature { signature: Uint8Array; recoveryId: RecoveryIdType;}，RecoveryIdType 取值 0、1、2 或 3。看例子：

```
const e = new Uint8Array(32);
crypto.getRandomValues(e);
const ret = secp256k1.signRecoverable(h,d,e);                                //返回下面的值
{
  signature: Uint8Array(64) [
    225, 168,  34, 134, 177,  69, 151, 247, 244, 216,   4,
     57, 233, 219,  59, 157,  60,  75, 248,  83, 227, 240,
     12,   3, 163,  46, 139, 248, 212, 233, 144, 151,  23,
     19, 169, 219, 203,  37, 229, 155, 128,  34,   0, 237,
     16, 113, 138, 163, 157, 200,  81, 167,  12, 125, 149,
    242, 192, 245,  91,  89,   1, 174,  82, 181
  ],
  recoveryId: 1
}
```

6. recover(h: Uint8Array, signature: Uint8Array, recoveryId: RecoveryIdType, compressed?: boolean): Uint8Array | null - 从签名中恢复公钥，compressed 参数默认为 false（即返回非压缩公钥）。

```
secp256k1.recover(h, ret.signature, 1, true);                                //返回下面的值
Uint8Array(33) [
    2, 189, 219, 232, 151,  68,  98,   0,
   72,  91,  24, 190,  78, 207, 228, 185,
  212, 116, 154,  38, 158, 190, 244,  52,
   70,  13,  75,  77,   5, 132,  20, 230,
  164
]
```

2.5.3 bip32

bip32 是一个用于实现 BIP-32 层级确定性钱包(HD Wallet)的 JavaScript 库，它可以生成和管理比特币及其衍生币种的密钥对。bip32 的四个核心工厂方法 fromBase58()、fromPrivateKey()、fromPublicKey()、fromSeed()都是产生一个 BIP32 类的实例，这些方法的声明如下：

```
function fromBase58(                    //从扩展私钥（xprv）或公钥（xpub）实例化 BIP32，导入钱包（包含完整派生路径）
    key: string,                                                    // Base58 编码的 xpub/xprv
    network?: Network                                               // 网络类型（可选，默认比特币主网）
): BIP32Interface;
```

```
function fromPrivateKey(                                    //从私钥和链码实例化 BIP32，全功能钱包节点（观察，签名）
   privateKey: Buffer,                                                                  // 32 字节私钥
   chainCode: Buffer,                                                                   // 32 字节链码
   network?: Network                                                                  // 网络类型（可选）
): BIP32Interface;

function fromPublicKey(                                      //从公钥和链码实例化 BIP32，创建观察钱包
   publicKey: Buffer,                                                                // 33 或 65 字节公钥
   chainCode: Buffer,                                                                   // 32 字节链码
   network?: Network                                                                  // 网络类型（可选）
): BIP32Interface;

function fromSeed(                                          //从种子实例化 BIP32，全功能主节点（观察，签名）
   seed: Buffer,                                                                        // 64 字节种子
   network?: Network                                                                  // 网络类型（可选）
): BIP32Interface;
```

看例子：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');
const { BIP32Factory } = require('bip32');
const bip32 = BIP32Factory(ecc);

const xprv =
'xprv9zf1wBXpMpUwo1oDEW2VSA6xvn8r1iUypPmevmsorjGfpxFFeyMVB7qAFvXtahk86vQPrrmHAVXUG6wqdFUHcmgWhFEeKstT2pM93
iM3b5U';
const node0 = bip32.fromBase58(xprv);                                                   //BIP32 的实例

const customNetwork = {
   ...bitcoin.networks.bitcoin,                                                        // 继承主网配置
   bip32: {
      public: 0x04b24746,                                                           // zpub 的版本字节
      private: 0x04b2430c,                                                          // zprv 的版本字节
   },
};
const priv_ext =
'zprvAWgYBBk7JR8GjmDKs1WtqZsk8qALxy5M9uJSQuK3BAbuUHPf22PdbYQBnMQL9qna198PeXBa4JrwWvwWHitmh5LkkzFXxq84fqept
ns75EV';
let node1 = bip32Factory.fromBase58(priv_ext, customNetwork);

const node2 = bip32.fromPrivateKey(
   Buffer.from('294b0e8909824f11387032ea22e7ee8678a89f66cd25e52808af667440c795c7', 'hex'),          //私钥
   Buffer.from('e24e0d450129a5b6f3b21bd9e4fcffcabd3338468da1c688b547fa9b5551e34c', 'hex')           //链码
);

const node3 =
bip32.fromSeed(Buffer.from('4b04e49456d2abf518badd4034be04b7f0bae645a0a94e10e641c2e237f4e51a2be5c8c0ec61d6
2ca9e91483f1b237d4d07530028176de207649bfdaf4cfdee9', 'hex'));
```

BIP32 的实例包含如下属性：

```
BIP32 {
  __D: <Buffer b0 7d cb c9 34 54 0c c6 4b 6d 6d 0e bd 4b de a9 05 ce 67 4d be 0d 00 98 37 e6 d9 c1 53 9e
84 42>,                                                                                 //私钥
  __Q: undefined,                                                                       //公钥
  lowR: false,
  chainCode: <Buffer 50 ca a6 a0 c1 ee a8 1e 6c 63 d7 6d dd 23 bc 73 05 59 f6 c0 ff 87 02 f7 6d 75 c0 5a
46 55 84 c4>,                                                                           //链码
  network: {                                                                          //比特币网络
    messagePrefix: '\x18Bitcoin Signed Message:\n',
    bech32: 'bc',
    bip32: { public: 76067358, private: 76066276 },
    pubKeyHash: 0,
    scriptHash: 5,
    wif: 128
  },
  __DEPTH: 4,                                                                         //HD 节点树的深度
  __INDEX: 1,                                                                           //索引
```

```
  __PARENT_FINGERPRINT: 254402320                                    //父节点的指纹
}
```

可通过属性名 publicKey、privateKey、chainCode、network、depth、index、parentFingerprint、identifier、fingerprint、compressed 获取相关属性的值。

BIP32 的实例包含如下方法。

1. 派生子节点的方法：

派生子节点：derive(index: number, hardened?: boolean): BIP32 - 参数 index 确定子节点序号，从 0 开始，hardened 是否派生强化子节点，默认是 false。derive(i, true)等价于 derive(i')。

按路径派生：derivePath(path: string): BIP32 - 参数 path 的格式为 m / purpose' / coin_type' / account' / change / address_index，具体参考 1.5.2 小节，如"m/84'/0'/0'/0/0"

2. 语密钥有关的方法：

将节点序列化为 Base58 字符串(xpub/xprv)：toBase58(): string - 当节点包含私钥时返回 xprv 开头的扩展私钥，如果没有包含私钥，则返回 xpub 开头的扩展公钥。

获取公钥：Buffer：toPublicKey(): Buffer

获取私钥 Buffer (仅当节点包含私钥)：toPrivateKey(): Buffer

输出 WIF 格式的私钥：toWIF(): string。

3. 用于验证的方法：

检查节点是否被"阉割"(无私钥)：isNeutered(): boolean - 节点没包含私钥则返回 true。

生成 ECDSA 签名 (仅当节点包含私钥)：sign(msgHash: Buffer): Buffer。

验证 ECDSA 签名：verify(msgHash: Buffer, signature: Buffer): boolean - msgHash 为 256 位的哈希，看例子：

```
const msgHash = bitcoin.crypto.sha256('hello world');
const signature = node.sign(msgHash);
node.verify(msgHash, signature);                                    //返回 true
```

生成斯诺尔签名：signSchnorr(msgHash: Buffer, extraEntropy?: Buffer): Buffer - 可选参数 extraEntropy 提供额外的熵以增强安全性。

验证斯诺尔签名：verifySchnorr(msgHash: Buffer, signature: Buffer): boolean

4. 用来调整节点的方法：

移除私钥信息：neutered(): BIP32 - 创建当前节点的只读副本，移除私钥信息，从而转为观察钱包。

对私钥节点进行 Tweak 调整：tweakFromPrivateKey(tweak: Buffer): Bip32Signer - 私钥加上 tweak，常用于 Taproot 构造。

对公钥节点进行 Tweak 调整：tweakFromPublicKey(tweak: Buffer): Bip32Signer - 公钥加上 tweak，用于观察钱包的派生。

通用 Tweak 方法：tweak(tweak: Buffer): Bip32Signer - 根据节点类型自动选择公私钥调整——如果节点有私钥就调用 tweakFromPrivateKey()方法，否则调用 tweakFromPublicKey()方法。

Bip32Signer 类包含三个属性：__D（私钥，可通过 Bip32Signer.privateKey 获取）、__Q（公钥，可通过 Bip32Signer.publicKey 获取）、lowR，方法有：sign()、verify()、signSchnorr()、verifySchnorr()，前两个方法是 ECDSA 签名与验证，后两者是 Schnorr 签名与验证。没包含私钥的 Bip32Signer 不能用于签名。

进行公/私钥 tweak 调整只在处理 P2TR 类型的交易时才会采用。

看例子：

```
const bitcoin = require('bitcoinjs-lib');
const { randomBytes, } = require('node:crypto');
const ecc = require('tiny-secp256k1');
const { BIP32Factory } = require('bip32');
const bip32 = BIP32Factory(ecc);

const xprv =
'xprv9zf1wBXpMpUwo1oDEW2VSA6xvn8r1iUypPmevmsorjGfpxFFeyMVB7qAFvXtahk86vQPrrmHAVXUG6wqdFUHcmgWhFEeKstT2pM93
iM3b5U';
const node0 = bip32.fromBase58(xprv);
console.log(node0.isNeutered());
false                                                                    //表明 node0 包含了私钥
const node1 = node0.neutered();
console.log(node1.isNeutered());
true                                                                     //表明 node1 没有包含私钥
console.log(node0.toBase58());                                           //节点包含私钥，就输出扩展私钥
xprv9zf1wBXpMpUwo1oDEW2VSA6xvn8r1iUypPmevmsorjGfpxFFeyMVB7qAFvXtahk86vQPrrmHAVXUG6wqdFUHcmgWhFEeKstT2pM93i
M3b5U
console.log(node1.toBase58());                                           //节点没有包含私钥，就输出扩展公钥
xpub6DeNLh4iCC3F1VsgLXZVoJ3hUoyLRBCqBchFjAHRR4oehkaQCWfjiv9e7CKLLm4YmJiYoHfV5dGZHjf2rQ8PDpRXyEgJunr8vDYVUn
2co4U

const tweak = randomBytes(32);                                           //产生一个 32 字节的随机数
const bip32Signer0 = node0.tweakFromPrivateKey(tweak);
const bip32Signer1 = node1.tweakFromPublicKey(tweak);

const msgHash = bitcoin.crypto.sha256('Hello Bip32!');
const sig0 = bip32Signer0.sign(msgHash);
bip32Signer0.verify(msgHash,sig0);                                       //返回 true

bip32Signer1.signSchnorr(msgHash);                                       //报错：缺少私钥
```

2.5.4 bip38

bip38 对私钥加密和解密做了详细规范，并提供了相应的方法。官网 https://github.com/cryptocoinjs/bip38，规范文件在 https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki。

bip38 库使用了这些技术：AES256Encrypt（AES 加密）、AES256Decrypt（AES 解密）、SHA256（哈希算法）、Scrypt、ECMultiply（secp256k1 椭圆曲线的点乘法）、G（secp256k1 椭圆曲线的基点）、N（secp256k1 椭圆曲线的阶）和 Base58Check。

Scrypt 是一种众所周知的密钥派生算法，它采用以下参数： password（字符串类型的密码）、salt（字符串类型的盐）、n（整数）、r（整数）、p（整数）、length（整数），并确定性地生成长度等于 length 参数的字节数组。在 bip38 库中，取 SHA256(SHA256(私钥推导的比特币地址))前四字节作为盐，length 固定为 64，所以对私钥进行加解密时要指定 password、n、r 和 p 参数的值，n 的默认值为 16384，r 和 p 的默认值都是 8。参数 n 直接影响 CPU/内存消耗，值为 2 的 n 次方，值越大消耗的 cpu 和内存越多，密文私钥就越难以暴力破解，参数 r 定义块大小，影响内存消耗，值越大消耗的内存就越多；参数 p 定义并行数，影响计算复杂度。一次加密时间控制在 1 分钟以上比较合适，对于现代的电脑，n= 32768, r=16, p=12 较合适。

密文私钥以“6P”开头。

一、加密私钥

1. 函数 encrypt()的声明如下：

```
function encrypt(
    privateKey: Buffer,                                                    // 原始私钥的 Buffer 形式（32 字节）
    compressed: boolean,                                                   // 是否生成对应压缩地址的加密私钥
    password: string,                                                                          // 加密密码
    progressCallback?: (progress: number) => void,                                  // 进度回调（可选）
    scryptParams?: {                                                      // 自定义 scrypt 参数（可选）
        N?: number,                                          // CPU/内存成本参数（默认：16384），必须是 2 的 x 次方
        r?: number,                                                                // 块大小参数（默认：8）
        p?: number                                                                 // 并行化参数（默认：8）
    }
): string;
```

compressed 参数决定了生成的加密私钥对应的比特币地址类型，具体控制公钥的压缩格式：compressed: true：加密后的私钥将对应压缩公钥地址（以 1 或 3 开头的主流地址），compressed: false：加密后的私钥将对应非压缩公钥地址（以 1 开头的旧格式地址）。相同的私钥用不同压缩标志加密会得到不同的 BIP38 密文，解密时需要提供相同的压缩标志才能正确恢复原始私钥。

scryptParams 参数的值不同，加密的结果也不同，解密时一定要用相同的值。

返回加密后的私钥。看例子：

```
const bip38 = require('bip38');
const privateKey = Buffer.from('ef15bb70f682203a5c21f2220409d749fd4b2c7c1eac12fed81747aef04d7936','hex');
const encryptedKey = bip38.encrypt(privateKey, true, "password123!");             //返回如下的已经加了密的私钥
'6PYLKsDnsWVgtpTa1vUfyEL9E4gFDF3GLYWRScbtzCAFyV7ZHv5PmMhNkz'

bip38.encrypt(privateKey, false, "A1b2C3&Y4yhl9T", (progress)=>{console.log('加密进度：',progress);});
加密进度： { current: 1000, total: 262144, percent: 0.3814697265625 }
加密进度： { current: 2000, total: 262144, percent: 0.762939453125 }
加密进度： { current: 3000, total: 262144, percent: 1.1444091796875 }
……
加密进度： { current: 262000, total: 262144, percent: 99.945068359375 }
'6PRV4esXgpJmRNVAz247LrC35nFqs9z8JEXBCtWgaDSLh86w1v2JFPwbMM'

bip38.encrypt(privateKey, true, "password123!",null, { N: 8192, r: 8, p: 8 });                        //返回
6PYLKsDnt4LoR7pcWoEQHMi3wjKPr1t79q8VDWBWBPzcmHZFiNgNGmGnju
bip38.encrypt(privateKey, true, "password123!",null, { N: 32768, r: 16, p: 8 });                       //返回
'6PYLKsDnrisnkWm4RxZFWzQAQ7xHesfbokCcNC72EduNA6cR4aogi292gd'
```

2. 异步加密函数 encryptAsync()的声明如下：

```
function encryptAsync(
    privateKey: Buffer,
    compressed: boolean,
    password: string,
    progressCallback?: (progress: number) => void,
    scryptParams?: {
        N?: number;
        r?: number;
        p?: number;
    },
    promiseInterval?: number                                    //控制进度回调的触发间隔时间（毫秒），默认为 100 毫秒
): Promise<string>;
```

看例子：

```
const encryptedKey = await bip38.encryptAsync(
    privateKey,
    true,
    'password123',
```

```
    (progress) => { console.log(progress); },
    { N: 16384, r: 8, p: 8 },                                                    // scryptParams
    500                                                                  // promiseInterval (毫秒)
);
```

3. 底层加密函数 encryotRaw()的声明如下：

```
function encryptRaw(
    privateKey: Buffer,                                          // 原始私钥的 Buffer 形式（32 字节）
    compressed: boolean,                                                    // 是否生成压缩
    password: string,                                                       // 加密密码
    progressCallback?: (progress: number) => void,                          // 进度回调（可选）
    scryptParams?: {                                                 // 自定义 scrypt 参数（可选）
        N?: number,                                                  // CPU/内存成本参数（默认：16384）
        r?: number,                                                         // 块大小参数（默认：8）
        p?: number                                                          // 并行化参数（默认：8）
    }
): Buffer;
```

此函数直接处理原始参数（不包含 BIP38 的标准前缀和校验码），返回标识+盐+原始密文 Buffer （格式：0x01 | 0x42 | flagByte | salt (4) | 密文(32)共 39 个字节的 Buffer)。看例子：

```
const enc = bip38.encryptRaw(privateKey, true, 'a12345678!');
console.log(enc.toString('hex'));
0142e01b99bfebc3ae34000300563492e2289ce8257ecde4fcb1d8e21b2c4c3ce01afb4b12405d
```

结果是：01 42 e0 1b99bfeb(盐) c3ae34000300563492e2289ce8257ecde4fcb1d8e21b2c4c3ce01afb4b12405d(加密后的私钥)。

encryptRawAsync()是异步实现，参数与 encryptAsync()函数相同，这里不再赘述。

encrypt(privateKey, compressed, passphrase, progressCallback, scryptParams) 方法等价于 bs58check.encode(encryptRaw(privateKey, compressed, passphrase, progressCallback, scryptParams))。

4. 函数 verify(encryptedKey: string)验证一个字符串是不是一个合法的加密私钥 ，看例子：

```
bip38.verify('6PYLKsDnt4LoR7pcWoEQHMi3wjKPr1t79q8VDWBWBPzcmHZFiNgNGmGnju');                    //返回
true
```

二、解密私钥

1. 函数 decrypt()的声明如下：

```
function decrypt(
    encryptedKey: string,                                        // 原始私钥的 Buffer 形式（32 字节）
    passphrase: string,                                                     // 加密密码
    progressCallback?: (progress: number) => void,                          // 进度回调（可选）
    scryptParams?: {                                                 // 自定义 scrypt 参数（可选）
        N?: number,                                                  // CPU/内存成本参数（默认：16384）
        r?: number,                                                         // 块大小参数（默认：8）
        p?: number                                                          // 并行化参数（默认：8）
    }
): {
    privateKey: Buffer;
    compressed: boolean
};
```

看例子：

```
bip38.decrypt(encryptedKey,'password123!');                                                  //返回
{
```

```
  privateKey: <Buffer ef 15 bb 70 f6 82 20 3a 5c 21 f2 22 04 09 d7 49 fd 4b 2c 7c 1e ac 12 fe d8 17 47 ae
f0 4d 79 36>,
  compressed: true
}
```

2. 异步解密函数 decryptAsync()的声明如下：

```
function decryptAsync(
   encryptedKey: Buffer,
   passphrase: string,
   progressCallback?: (progress: number) => void,
   scryptParams?: {
      N?: number;
      r?: number;
      p?: number;
   },
   promiseInterval?: number                                    //控制进度回调的触发间隔时间（毫秒），默认为 100 毫秒
): Promise<{ privateKey: Buffer; compressed: boolean }>;
```

看例子：

```
bip38.decryptAsync(encrypted, 'password123!').then(ret => {
   console.log(ret.privateKey.toString('hex'))
});
ef15bb70f682203a5c21f2220409d749fd4b2c7c1eac12fed81747aef04d7936
```

3. 底层解密函数 decryptRaw()的声明如下：

```
function decryptRaw(
   encryptedKey: Buffer,                                                    // 密文私钥的 Buffer 形式（32 字节）
   passphrase: string,                                                                  // 加密密码
   progressCallback?: (progress: number) => void,                                  // 进度回调（可选）
   scryptParams?: {                                                          // 自定义 scrypt 参数（可选）
      N?: number,                                                        // CPU/内存成本参数（默认：16384）
      r?: number,                                                                 // 块大小参数（默认：8）
      p?: number                                                                  // 并行化参数（默认：8）
   }
): {
   privateKey: privateKey,
   compressed: compressed
 };
```

看例子：

```
const bs58check = require('bs58check');
const encryptedBuffer = bs58check.default.decode(encryptedKey);
bip38.decryptRaw(encryptedBuffer, 'password123!');
```

decrypt(encryptedKey, passphrase, progressCallback, scryptParams) 函数等价于 decryptRaw(bs58check.default.decode(encryptedKey), passphrase, progressCallback, scryptParams)。

2.5.5 bip39

bip39 库就是 BIP39 规范的 JavaScript 实现——处理 HD 钱包的助记词、种子和密钥等。

1. bip39.wordlists 对象

此对象保存了多种语言的常用词数组，数组长度都是 2048，比如 bip39.wordlists.english 保存了 2048 个常见的英文单词，即['abandon', 'ability', 'able', 'about', 'above', 'absent', 'absorb', 'abstract', 'absurd', 'abuse', 'access', 'accident', 'account',…]。

其他的语言还支持：chinese_simplified（简体中文）、chinese_traditional（繁体中文）、japanese（日语）、korean（韩语）、

french（法语）、italian（意大利语）、spanish（西班牙语）、portuguese（葡萄牙语）和 czech（捷克语），其中 bip39.wordlists.EN 为英语的别称，等于 bip39.wordlists.english，bip39.wordlists.JP 是日语的别称。

```
bip39.wordlists.chinese_simplified
[
  '的', '一', '是', '在', '不', '了', '有', '和', '人', '这',
  '中', '大', '为', '上', '个', '国', '我', '以', '要', '他',
  '时', '来', '用', '们', '生', '到', '作', '地', '于', '出',
  '就', '分', '对', '成', '会', '可', '主', '发', '年', '动',
  '同', '工', '也', '能', '下', '过', '子', '说', '产', '种',
  '面', '而', '方', '后', '多', '定', '行', '学', '法', '所',
  '民', '得', '经', '十', '三', '之', '进', '着', '等', '部',
  '度', '家', '电', '力', '里', '如', '水', '化', '高', '自',
  '二', '理', '起', '小', '物', '现', '实', '加', '量', '都',
  '两', '体', '制', '机', '当', '使', '点', '从', '业', '本',
  ... 1948 more items
]
[
  'abandon', 'ability', 'able',    'about',   'above',   'absent',
  'absorb',  'abstract', 'absurd',  'abuse',   'access',  'accident',
  'account', 'accuse',   'achieve', 'acid',    'acoustic', 'acquire',
  'across',  'act',      'action',  'actor',   'actress', 'actual',
  'adapt',   'add',      'addict',  'address', 'adjust',  'admit',
  'adult',   'advance',  'advice',  'aerobic', 'affair',  'afford',
  'afraid',  'again',    'age',     'agent',   'agree',   'ahead',
  'aim',     'air',      'airport', 'aisle',   'alarm',   'album',
  'alcohol', 'alert',    'alien',   'all',     'alley',   'allow',
  'almost',  'alone',    'alpha',   'already', 'also',    'alter',
  'always',  'amateur',  'amazing', 'among',   'amount',  'amused',
  'analyst', 'anchor',   'ancient', 'anger',   'angle',   'angry',
  'animal',  'ankle',    'announce', 'annual', 'another', 'answer',
  'antenna', 'antique',  'anxiety', 'any',     'apart',   'apology',
  'appear',  'apple',    'approve', 'april',   'arch',    'arctic',
  'area',    'arena',    'argue',   'arm',     'armed',   'armor',
  'army',    'around',   'arrange', 'arrest',
  ... 1948 more items
]
bip39.wordlists.chinese_traditional
[
  '的', '一', '是', '在', '不', '了', '有', '和', '人', '這',
  '中', '大', '為', '上', '個', '國', '我', '以', '要', '他',
  '時', '來', '用', '們', '生', '到', '作', '地', '於', '出',
  '就', '分', '對', '成', '會', '可', '主', '發', '年', '動',
  '同', '工', '也', '能', '下', '過', '子', '說', '產', '種',
  '面', '而', '方', '後', '多', '定', '行', '學', '法', '所',
  '民', '得', '經', '十', '三', '之', '進', '著', '等', '部',
  '度', '家', '電', '力', '裡', '如', '水', '化', '高', '自',
  '二', '理', '起', '小', '物', '現', '實', '加', '量', '都',
  '兩', '體', '制', '機', '當', '使', '點', '從', '業', '本',
  ... 1948 more items
]
bip39.wordlists.japanese
[
  'あいこくしん', 'あいさつ',  'あいた゛',   'あおそら',  'あかちゃん',
  'あきる',      'あけかた',  'あける',     'あこがれる', 'あさい',
  'あさひ',      'あしあと',  'あしわう',   'あすがる',  'あすぎ',
  'あそふ゛',     'あたえる',  'あたためる', 'あたりまえ', 'あたる',
  'あつい',      'あつかう',  'あっしゅく', 'あつまり',  'あつめる',
  'あてな',      'あてはまる', 'あひる',     'あふら',    'あふる',
  'あふれる',    'あまい',    'あまと゛',   'あまやかす', 'あまり',
  'あみもの',    'あめりか',  'あやまる',   'あゆむ',    'あらいくま',
  'あらし',      'あらすし゛', 'あらためる', 'あらゆる',  'あらわす',
  'ありかとう',  'あわせる',  'あわてる',   'あんい',    'あんかい',
  'あんこ',      'あんせん',  'あんてい',   'あんない',  'あんまり',
  'いいたす',    'いおん',    'いかい',     'いかく',    'いきおい',
```

```
  'いきなり',    'いきもの',  'いきる',    'いくし',    'いくふぐ',
  'いけはな',    'いけん',    'いこう',    'いこく',    'いこつ',
  'いさましい',  'いさん',    'いしき',    'いしゅう',  'いしょう',
  'いしわる',    'いすみ',    'いすれ',    'いせい',    'いせえひ',
  'いせかい',    'いせき',    'いせん',    'いそうろう','いそかじい',
  'いたい',      'いたく',    'いたすら',  'いたみ',    'いたりあ',
  'いちおう',    'いちし',    'いちと',    'いちは',    'いちふ',
  'いちりゅう',  'いつか',    'いっしゅん','いっせい',  'いっそう',
  ... 1948 more items
]
……
```

2. setDefaultWordlist()/getDefaultWordlist()方法

默认语言为英语，可用方法 bip39.getDefaultWordlist()查看当前的默认语言，还可用 setDefaultWordlist(language)设置默认语言，如默认语言设为简体中文：

```
bip39.setDefaultWordlist('chinese_simplified');
bip39.getDefaultWordlist();
'chinese_simplified'
```

再比如设为日语：

```
bip39.setDefaultWordlist('japanese');
```

3. generateMnemonic()方法

声明：function generateMnemonic(strength?: number, rng?: (size: number) => Buffer, wordlist?: string[]): string

generateMnemonic()方法首先调用 rng 方法产生 strength 位的随机数，然后在 wordlist 语言中产生相应的助记词，一组助记词中包含单词的个数可为 12、15、18、21 或 24。方法的三个参数均可选，strength 取值范围[128,256]且必须是 32 的倍数，默认为 128，假设产生的助记词单词个数为 n，那么存在等式 n*352/33=strength，因此 strength 只能取值 128、160、192、224 或 256。参数 rng 指明产生随机数的函数，默认是 crypto.randomBytes()，如果要指明第三个参数，那么 rng 需要一个值、null 或者"。例如：

```
bip39.generateMnemonic();
'meat abstract latin color city galaxy buddy hunt crumble stuff always marble'

bip39.generateMnemonic(32*5);
'girl action obscure moon filter bundle scan derive drama kitten peasant radio girl camera cliff'

bip39.generateMnemonic(192,'',bip39.wordlists.chinese_simplified);
'无 盘 痕 蒸 建 干 越 词 乡 削 曹 观 从 涨 际 怀 盗 晚'
```

即使实参相同，每次调用 generateMnemonic()方法返回的结果是不同的。

4. mnemonicToEntropy()/validateMnemonic()方法

声明：　　function mnemonicToEntropy(mnemonic: string, wordlist?: string[]): string;

function entropyToMnemonic(entropy: Buffer | string, wordlist?: string[]): string;

function validateMnemonic(mnemonic: string, wordlist?: string[]): boolean;

mnemonicToEntropy()方法从助记词可反算出随机数（以十六进制格式的字符串保存在 str_hex 变量中），

entropyToMnemonic()方法从随机数推导助记词，看例子：

```
bip39.setDefaultWordlist('english');
const mnemonic = bip39.generateMnemonic(192);
console.log(mnemonic);
person adapt rebuild marble before veteran mutual hurt siren window olive glory air dog milk exhaust name
document
const str_hex = bip39.mnemonicToEntropy(mnemonic);
console.log(str_hex);
a34062cd43d143e6648b7ec97f6e68b1c0568123227b92c8
const mnemonic1 = bip39.entropyToMnemonic(str_hex);                                    //从随机数恢复助记词
console.log(mnemonic1);
person adapt rebuild marble before veteran mutual hurt siren window olive glory air dog milk exhaust name
document
```

mnemonicToEntropy 方法是 entropyToMnemonic 方法的逆操作。

方法 bip39.validateMnemonic()验证助记词是否合法，如：

```
bip39.validateMnemonic('typical yellow spare faculty hotel slush matter online husband child season
leaf');
true                                                                                    ←在默认语言中合法
bip39.validateMnemonic('typical yellow spare faculty hotel slush matter online husband child season
leaf',bip39.wordlists.french);
false                                                                                   ←同一组助记词在法语中不合法
bip39.validateMnemonic('love yellow spare faculty hotel slush matter online husband child season leaf');
false                                                                                   ←修改了一个单词后的助记词不在合法
```

5. mnemonicToSeedSync()/mnemonicToSeed()方法

声明：　function mnemonicToSeedSync(mnemonic: string, password?: string): Buffer;

　　function mnemonicToSeed(mnemonic: string, password?: string): Promise<Buffer>;

两个方法都是完成从助记词到种子的任务，前者是同步方法，而后者为异步方法。这一步允许添加密码，这样即使泄露了助记词，只要密码没泄露，比特币也是安全的。看例子：

```
const mnemonic = 'shove visit join pull green game amount web first spoil patient crumble';
const seed1 = bip39.mnemonicToSeedSync(mnemonic);                                       //无密码产生种子
console.log(seed1.toString('hex'));
454a6d1cd3a1d2075bfa9ea00415ad688756554e164cfd1ec22e06e08abbf91efe1fe682c34692ce3c7666b1575303d510639ef6c4
f3b951c1b052e76ec966e2
const seed2 = bip39.mnemonicToSeedSync(mnemonic, 'a123456!');                           //用密码保护产生的种子
console.log(seed2.toString('hex'));
8eb6f3ffa861dc148c45ab4449540ef4324eaeba90f4edb5fa6d5d8e5c4a3fc23d04295625d87a50094344eb32ef6d1e610aa2cc34
e24670df6f753e27d958e1
```

可见同一个助记词，有无密码产生的种子是不同的。以后会用种子产生 HD 钱包。mnemonicToSeed()是异步方法，看例子：

```
bip39.mnemonicToSeed(mnemonic, 'a123456!').then(s=>console.log(s.toString('hex')));
Promise {
  <pending>,
  [Symbol(async_id_symbol)]: 401,
  [Symbol(trigger_async_id_symbol)]: 400
}
>
8eb6f3ffa861dc148c45ab4449540ef4324eaeba90f4edb5fa6d5d8e5c4a3fc23d04295625d87a50094344eb32ef6d1e610aa2cc34
e24670df6f753e27d958e1
```

种子是 512 位的数，公 64 个字节。由种子无法推算出助记词。

2.5.6 bip84

bip84 是用于实现 BIP-84 标准的 JavaScript 库，专门用于生成原生 SegWit (P2WPKH) 比特币钱包，这类地址以 bc1q 开头，提供更低的交易手续费和更好的隐私性。

接下来对 bip84 库逐一解释。

1. entropyToMnemonic()方法

从随机数产生助记词需要用到 bip84.entropyToMnemonic()方法，词方法就是调用 bip39 库的 entropyToMnemonic(entropy[, wordlist])方法，方法中的参数 entropy 为字符串（比如第一步产生的 rand_hex）或者 Buffer、ArrayBuffer、Array 以及类似数组对象的实例；方法的第二个参数指定助记词的语言，目前支持的语言有简体中文 chinese_simplified、繁体中文 chinese_traditional、日语 japanese、法语 french、葡萄牙语 portuguese、西班牙语 spanish、韩语 korean、意大利语 italian 以及捷克语 czech，具体可查看 bip39.wordlists 对象。

例如从第一步中产生的随机数（在 rand_hex 中）生成英语助记词：

```
const mnemonic = bip84.entropyToMnemonic(rand_hex);
console.log(mnemonic);
weasel prefer behave flat blanket mixture ensure napkin convince write cart perfect cabin make undo copper
adapt vessel either must piece senior patient famous
```

同样的随机数生成简体中文助记词：

```
const mnemonic_cn = bip84.entropyToMnemonic(rand_hex, bip39.wordlists.chinese_simplified);
console.log(mnemonic_cn);
'透 敏 雷 朵 辖 征 糊 家 洪 横 妇 饿 趋 固 扰 尤 腹 仰 甚 请 谈 晚 熟 范'
```

bip84.validateMnemonic()方法验证助记词是否合法，它直接调用了 bip39.validateMnemonic(mnemonic[, wordlist])，如：

```
bip84.validateMnemonic('typical yellow spare faculty hotel slush matter online husband child season
leaf');
true                                                                                    ←在默认语言中合法
bip84.validateMnemonic('typical yellow spare faculty hotel slush matter online husband child season
leaf',bip39.wordlists.french);
false                                                                                   ←同一组助记词在法语中不合法
bip84.validateMnemonic('love yellow spare faculty hotel slush matter online husband child season leaf');
false                                                                                   ←修改了一个单词后的助记词不在合法
```

到此我们得到了一个助记词并以字符串格式保存在 mnemonic 变量中。

2. generateMnemonic()方法

bip84.generateMnemonic()方法直接调用了 bip39.generateMnemonic([strength[, rng, wordlist]])方法，此方法首先调用 rng 方法产生 strength 位的随机数，然后在 wordlist 语言中产生相应的助记词，三个参数均可选，strength 取值范围[128,256]且必须是 32 的倍数，默认位 128。参数 rng 指明产生随机数的函数，默认是 crypto.randomBytes()，如果要指明第三个参数，那么 rng 需要一个值、null 或者''。例如：

```
bip84.generateMnemonic()
'meat abstract latin color city galaxy buddy hunt crumble stuff always marble'

bip84.generateMnemonic(32*5);
'girl action obscure moon filter bundle scan derive drama kitten peasant radio girl camera cliff'

bip84.generateMnemonic(192,'',bip39.wordlists.chinese_simplified);
'无 盘 痕 蒸 建 干 越 词 乡 削 曹 观 从 涨 际 怀 盗 晚'
```

generateMnemonic()方法一次性完成第一步（产生随机数）和第二部（产生助记词）

3. bip84. fromMnemonic()方法

此方法的原型为 fromMnemonic(mnemonic[, password, isTestnet, coinType, pubTypes, network, wordlist])，它是 fromMnemonic 对象的构造器，所以在实战时采用诸如 var root = new bip84.fromMnemonic(…)这样的语句。参数 mnemonic 是必须的，指定了助记词；password 定义密码，此密码加上助记词产生种子（内部调用 bip39.mnemonicToSeedSync(mnemonic, password ? password : '')），默认没有密码，种子保存在 fromMnemonic 对象的成员变量 seed 中；isTestnet 取值 true 或者 false，是否测试网，默认为 true（测试网）；coinType 指定币种，如果 isTestnet=true，那么 coinType 固定为 1，0-比特币（默认值），60-以太币，其他币种参见 SLIP-44；pubType 定义公钥类型，默认为比特币公钥类型，即{ mainnet: { zprv: '04b2430c', zpub: '04b24746'}, testnet: { vprv: '045f18bc', vpub: '045f1cf6'} }；参数 network 指定区块链网络类型，默认为比特币网络，即{ mainnet: networks.bitcoin, testnet: networks.testnet }， wordlist 指定助记词所在的语言。看例子：

```
const bip84 = require('bip84');
const mnemonic = bip84.generateMnemonic(32*5);
var root = new bip84.fromMnemonic(mnemonic, '!Password123&', false);
```

或者

```
rootNode = new bip84.fromMnemonic(wallets.mnemonic, wallets.seed_password, isTestNet_bitcoin, null, null,
null, bip39.wordlists.chinese_simplified);
```

对象 fromMnemonic 的实例化保存在 root 变量中，root 的数据成员如下：

```
fromMnemonic {
  seed: <Buffer 75 a4 21 b6 c5 84 a4 d6 75 6f 9b 44 cf 63 3f d9 60 d1 3c c6 59 20 11 6f d2 7f 76 96 50 e3
3d 30 30 8f d0 07 72 85 d3 a9 67 82 dc 2f 87 b1 39 2e 8c fa ... 14 more bytes>,
  isTestnet: false,
  coinType: 0,
  pubTypes: {
    mainnet: { zprv: '04b2430c', zpub: '04b24746' },
    testnet: { vprv: '045f18bc', vpub: '045f1cf6' }
  },
  network: {
    messagePrefix: '\x18Bitcoin Signed Message:\n',
    bech32: 'bc',
    bip32: { public: 76067358, private: 76066276 },
    pubKeyHash: 0,
    scriptHash: 5,
    wif: 128
  }
}
```

512 位二进制数的种子：

```
console.log(root.seed.toString('hex'));
75a421b6c584a4d6756f9b44cf633fd960d13cc65920116fd27f769650e33d30308fd0077285d3a96782dc2f87b1392e8cfafc99f932c05b468a349ff0880f48
```

fromMnemonic 对象的方法有：

getRootPrivateKey()：方法没有参数，从种子生成扩展根私钥（即图 2-17 中的 m），内部调用了 bip32.fromSeed(this.seed, this.network).toBase58()方法。

getRootPublicKey()：从种子获取扩展根公钥，内部调用了 bip32.fromSeed(this.seed, this.network).neutered().toBase58()。

deriveAccount(number, changePurpose)：从种子导出账户的扩展私钥，即导出 m/purpose'/coin_type'/account'节点的扩展私钥。参数 number 对应 account，可取值 0,1,2,3,…；changePurpose 对应 purpose，可取值 44,49,84，默认为 84，具体参考图 1-42。deriveAccount()方法内部调用了 bip32.fromSeed(this.seed, this.network).derivePath("m/" + purpose + "'/" + this.coinType + "'/" + number + "'").toBase58()。例如：

```
var child0 = root.deriveAccount(0);                                    //0 号账户（m/84'/0'/0'）的扩展私钥
```

4. bip84. fromZPrv (zprv, pubTypes, networks)

从扩展私钥实例化 fromZPrv 对象(对应一个账户),参数 zprv 指定扩展私钥,其他两个参数的解释请参考 fromMnemonic()方法的同名参数。例如:

```
var account0 = new bip84.fromZPrv(child0);                                    //0 号账户（即
m/84'/0'/0'）
```

对象的实例化在 account0 变量中，其数据成员如下:

```
fromZPrv {
  pubTypes: {
    mainnet: { zprv: '04b2430c', zpub: '04b24746' },
    testnet: { vprv: '045f18bc', vpub: '045f1cf6' }
  },
  networks: {
    mainnet: {
      messagePrefix: '\x18Bitcoin Signed Message:\n',
      bech32: 'bc',
      bip32: [Object],
      pubKeyHash: 0,
      scriptHash: 5,
      wif: 128
    },
    testnet: {
      messagePrefix: '\x18Bitcoin Signed Message:\n',
      bech32: 'tb',
      bip32: [Object],
      pubKeyHash: 111,
      scriptHash: 196,
      wif: 239
    }
  },
  network: {
    messagePrefix: '\x18Bitcoin Signed Message:\n',
    bech32: 'bc',
    bip32: { public: 76067358, private: 76066276 },
    pubKeyHash: 0,
    scriptHash: 5,
    wif: 128
  },
  isTestnet: false,
  zprv: 'xprv9zKHt6NaBTPmH2WndE2rrBrgVNiySbjW73W1h2wCrLCDRkML8MxoqBTdStNv8sZRPUXdGnxiPkrFzsQ63Dx2xYUEjYEVG3ZtuweVo8qdYKw'
}
```

一个账户就是一个 fromZPrv 对象的实例化，一个账户可拥有 2^{32} 个钱包，钱包编号从 0 开始，直到 2^{32}-1。对象 fromZPrv 的方法有:

getAccountPrivateKey()：获取账户的主扩展私钥。

getAccountPublicKey()：获取账户的主扩展公钥。

getPrivateKey(index[, isChange])：获取账户的第 index 个钱包的私钥，isChange 取值 true（内部找零钱包)、false（外部收币钱包)，默认为 false。

getPublicKey(index[, isChange])：获取账户的第 index 个钱包的公钥。参数解释同上。

getAddress(index[, isChange, purpose])：获取账户的第 index 个钱包的地址。参数 purpose 可取值 44,49,84，分别对应比特币的三类地址，默认为 84，具体参考图 1-42。可见同一个私钥可以导出三类地址。

看例子:

```
console.log("0 号账户的第一个对外收币钱包 = m/84'/0'/0'/0/0");
0 号账户的第一个对外收币钱包 = m/84'/0'/0'/0/0
console.log('钱包的私钥: ', account0.getPrivateKey(0));
```

```
钱包的私钥: L4vV81LsZ4FPqVPo6HyT8nNyE7tTBB5DaU1MvW89YdBJ356C8cG1
console.log('钱包的公钥: ', account0.getPublicKey(0));
钱包的公钥: 024e03ea1be927f288aafcdbc48666471316df63b9c28104c1b3648767b4c341f3
console.log('钱包的地址: ', account0.getAddress(0));
钱包的地址: bc1qjweyrtxrn9h2khxunj3v0d6mxqmx8dz2xcx08p
```

getKeypair(index[, isChange])：获取账户的第 index 个钱包的扩展密钥对，返回 BIP32 对象的实例化。如：

```
var walletKeypair = account0.getKeypair(0);                                    //账户第一个钱包的扩展密钥对
```

walletKeypair 的数据成员如下：

```
BIP32 {
  __D: <Buffer e5 d6 7e 5a 8a 81 a7 d4 39 ae 06 06 d5 b2 45 c9 dd b1 a6 f3 ce 9a e4 da 1e 7f 1a 2a fd 90 c8 8a>,
  __Q: undefined,
  lowR: false,
  chainCode: <Buffer 0b 6b 63 3d b5 46 85 e0 d6 d3 46 e2 37 27 68 4a 7b d4 a0 d2 07 87 1f ab ce 78 f3 48 07 3b 6a
4a>,
  network: {
    messagePrefix: '\x18Bitcoin Signed Message:\n',
    bech32: 'bc',
    bip32: { public: 76067358, private: 76066276 },
    pubKeyHash: 0,
    scriptHash: 5,
    wif: 128
  },
  __DEPTH: 5,
  __INDEX: 0,
  __PARENT_FINGERPRINT: 312722836
}
```

5. bip84.fromZPub (zpub, pubTypes, networks)

从一个扩展公钥实例化 fromZPub 对象，即观察账户。看例子：

```
const zpub = account0.getAccountPublicKey();
//const zpub='zpub6ryAtwFJKC32C6yVPy97dVzRPLrMjJSMJVU44D8bAgUxQkJwBDcBd76PLbND9FvvhaCboQsJvNLawVEyBX1LTNGXEwnS4ADjLNxFhQeJoDG';
var accountOnlyView = new bip84.fromZPub(zpub);
```

accountOnlyView 的数据成员如下：

```
fromZPub {
  pubTypes: {
    mainnet: { zprv: '04b2430c', zpub: '04b24746' },
    testnet: { vprv: '045f18bc', vpub: '045f1cf6' }
  },
  networks: {
    mainnet: {
      messagePrefix: '\x18Bitcoin Signed Message:\n',
      bech32: 'bc',
      bip32: [Object],
      pubKeyHash: 0,
      scriptHash: 5,
      wif: 128
    },
    testnet: {
      messagePrefix: '\x18Bitcoin Signed Message:\n',
      bech32: 'tb',
      bip32: [Object],
      pubKeyHash: 111,
      scriptHash: 196,
      wif: 239
    }
  },
  network: {
    messagePrefix: '\x18Bitcoin Signed Message:\n',
    bech32: 'bc',
    bip32: { public: 76067358, private: 76066276 },
    pubKeyHash: 0,
```

```
    scriptHash: 5,
    wif: 128
  },
  isTestnet: false,
  zpub: 'xpub6DJeHbuU1px4VWbFjFZsDKoR3QZTr4TMUGRcVRLpQfjCJYgUfuH4Nyn7JBT39Sd5tHxzJTgC13dVAv1qk8BJrtuKWGPatLaknvpxvFrFZPa'
}
```

fromZPub 对象的方法有：

getAccountPublicKey()：获取观察账户的主扩展公钥。

toNode(zpub)：

getPublicKey(index[, isChange])：获取观察账户的第 index 个钱包的公钥。参数解释参考上面 fromZPrv 对象的同名方法。

getAddress(index[, isChange, purpose])：获取观察账户第 index 个钱包的地址。

getPayment (index, isChange)：获得 p2wpkh 类型的支付交易，一个对象的实例化。例如：

```
var p = accountOnlyView.getPayment(0);
```

p 的数据成员：

```
{
  name: 'p2wpkh',
  network: {
    messagePrefix: '\x18Bitcoin Signed Message:\n',
    bech32: 'bc',
    bip32: { public: 76067358, private: 76066276 },
    pubKeyHash: 0,
    scriptHash: 5,
    wif: 128
  },
  address: [Getter/Setter],
  hash: [Getter/Setter],
  output: [Getter/Setter],
  pubkey: <Buffer 02 4e 03 ea 1b e9 27 f2 88 aa fc db c4 86 66 47 13 16 df 63 b9 c2 81 04 c1 b3 64 87 67 b4 c3 41
f3>,
  signature: [Getter/Setter],
  input: [Getter/Setter],
  witness: [Getter/Setter]
}
```

3 以太币

3.1 以太坊简介

3.1.1 以太币

以太币的“样子”如图 3-1 那样。

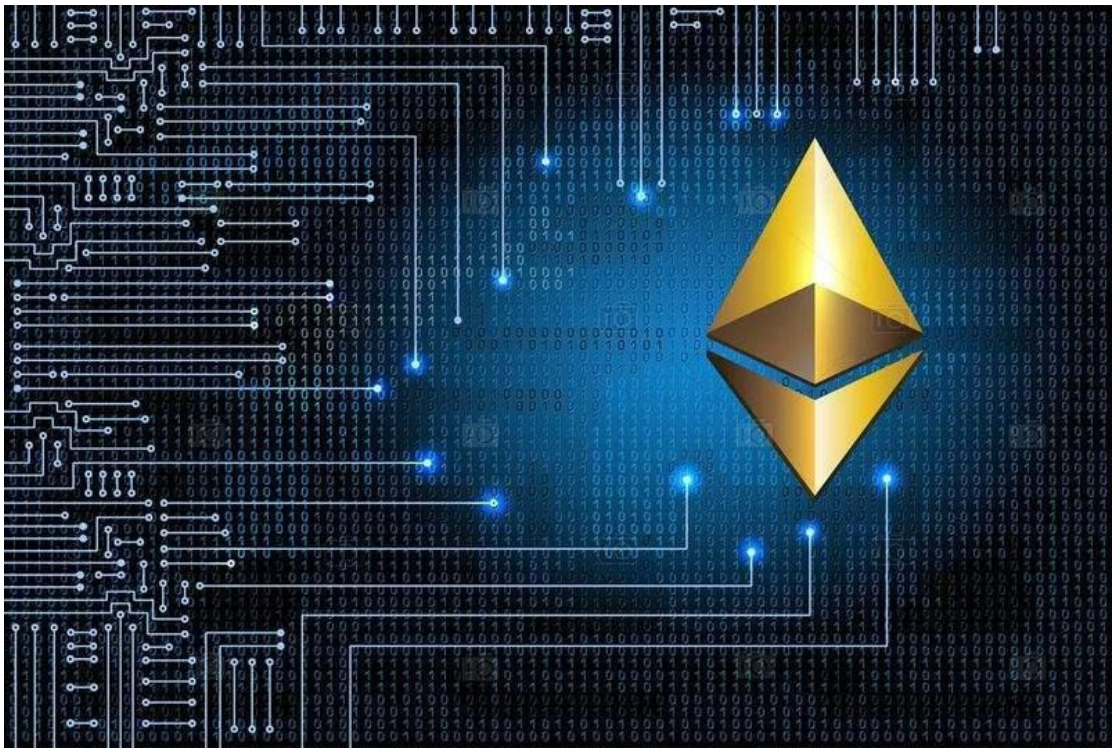


图 3-1 以太币的“样子”

如果说比特币奠定了中本聪在加密货币领域的先驱者地位，那么受比特币技术启发而开发出以太坊的维塔利克·布特林绝对是后起之秀，大有青出于蓝而胜于蓝之势，别忘了，他只是一个九零后。如果说比特币就是一个去中心化的分布式数据库，那么以太坊就是一台去中心化的世界计算机（或称为以太坊虚拟机，简称为 EVM）——由众多的互联在一起的电脑组成，这些电脑只能各自运行单独的程序（如智能合约这种小程序），因此不能完成大规模的并行计算任务。面对这样一台公正无私的永不停机的世界电脑，程序开发人员尽可以脑洞大开，里面充斥着大量的应用场景，不但能记账，还能部署去中心化的应用程序，其中以太币就是以太坊上的一个主要应用之一，其他的热门应用还有智能合约，建构于智能合约和以太币基础上的 DeFi、NFT、游戏等。DeFi（**De**centralized **Fi**nance）是去中心化的金融系统，借助于智能合约，每个人都可以向他人放款和借款，这个算是对传统银行信贷业务的颠覆，发展前景不可限量，对 DeFi 的详细介绍超出了本书的范畴，大家可以从其他渠道去了解。

2014 年，以太币初始发行 7200 万枚，此后规定每年大概发行 1800 万枚，发行是作为矿工的奖励这条途径的，具体奖励的以太币数量会经常变化，以前挖出一个区块曾奖励 2 枚以太币，最终的发行量没有上限，所以说以太币是一种通胀货

币，而比特币是通缩货币（因为遗失等原因，发行完毕后流通数量会减少）。以太币大概每 14 秒出一个新块，这较比特币每十分钟出一块速度要快得多。

以太币钱包具有余额的概念，所以转账交易相比比特币更容易理解一些，类似传统的银行账户。每笔交易中还可以附加高达 30KB 的备注，大约相当于 7000 个汉字，因此有人把一些重要的具有历史意义的内容放到以太币区块链上，这样就永不能修改和删除了，如果是记录坏事，真的是遗臭万年，如果是记录好事，那就流芳千古了。不过转账的交易费与这笔交易所占的字节数成正比，比如我要放一篇 5000 字的文章上去，交易费可能要十多美元了，给少了，矿工一般不会处理这样的交易。因为一个区块的大小是有上限的，所以交易越小，能放进一个区块中的交易数就越多，矿工挣取的交易费也就越多。你倒好，把一篇长文放进交易中，然后只想出几毛钱的交易费，鬼才理你呢！

为了更好地计算交易费用，以太坊引入了一个汽油(gas)的概念，每一笔交易费用固定 21000 升汽油，但是每升汽油的价格（单位为以太币）由交易发起者自己指定，一般是根据愿意出的美元来反推算每升汽油的价格，比如我愿意支付 3 美元的交易费用，假设现在一枚以太币值 4200 美元，因此根据下面的公司可算出每升汽油的价格：

$$交易费 = gas \times 21000 \times 以太币汇率$$

算出的 gas 约等于 3.4*10^{-8} 个以太币。在以太币交易过程中，会涉及很多单位，比如 Ether、Finney、Szabo、Gwei、Mwei、Kwei、wei 等，它们的之间的关系如下：

$$1Ether(以太币)=10^{3}Finney=10^{6}Szabo=10^{9}Gwei=10^{12}Mwei=10^{15}Kwei=10^{18}wei=10^{-3}Kether=10^{-6}Mether=10^{-9}Gether=10^{-12}Tether$$

最常用的单位有 wei（10^{-18} 以太币）、Gwei（10^{-9} 以太币）和 Ether（一个以太币）。

https://ethstats.net/网站展示了以太币的当前状态，包括已出的块数、最近出块的秒数、矿池数量、汽油价格等，如图 3-2。

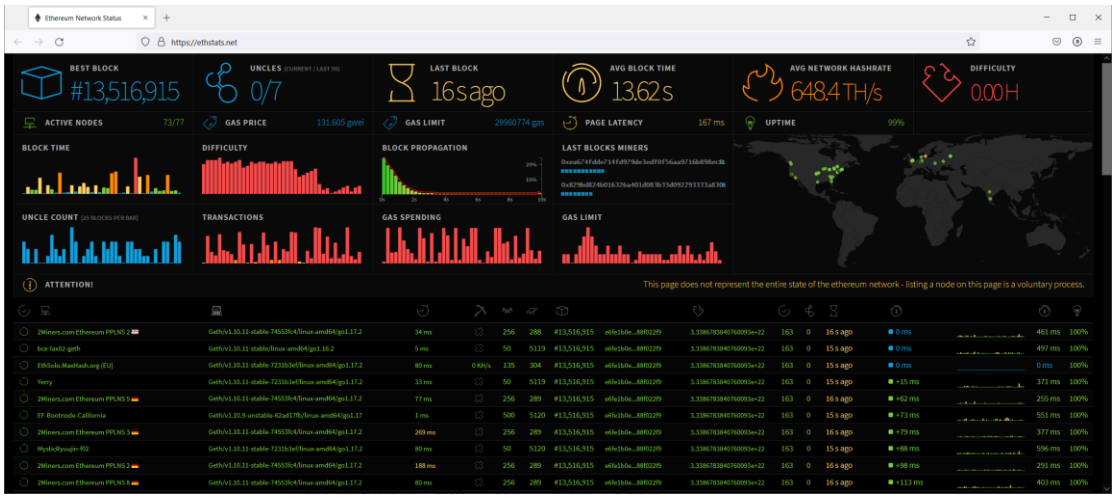


图 3-2 查看以太币的当前状态

3.1.2 智能合约

智能合约是一段可执行代码，当发生某些事情时或者被其他智能合约调用时就由以太坊虚拟机（EVM）执行它——由事件触发执行（如果把调用也当着事件的话）。这段可执行代码储存在以太坊区块链上（被当成交易额为 0 的以太币交易），任何人都无法修改代码，也不能删除它，更不能阻止它执行，只要发生了相应的事件，它就会被执行。储存在区块链上

的每个智能合约都有一个唯一的地址（类似钱包地址）。智能合约中可定义变量，执行过程中允许修改变量的值，因此变量就决定了智能合约的状态。“合约”被转换为“条件”或者说“事件”，并采用计算机编程语言写成可执行代码。就像自动售货机：你投入 5 元钱并按下 A1（这就是触发事件），自动售货机就会执行一系列的操作（相当于执行代码）：抓取相应的物品，投放到出口通道，储物槽内的物品往前推一格。

我们以一份简单的买卖合同为例进一步解析智能合约，合同见图 3-3 所示。

甲方：xxx
乙方：yyy

甲方先预付十万元定金，乙方收到定金后生产，半年内交货，甲方收到货物再支付款项五十万元整。
……

甲方签名盖章：　　乙方签名盖章：

图 3-3 买卖合同

显然存在三个条件（或者说预设了三个事件），即预付十万元、交货、支付尾款五十万元，参见图 3-4 所示。

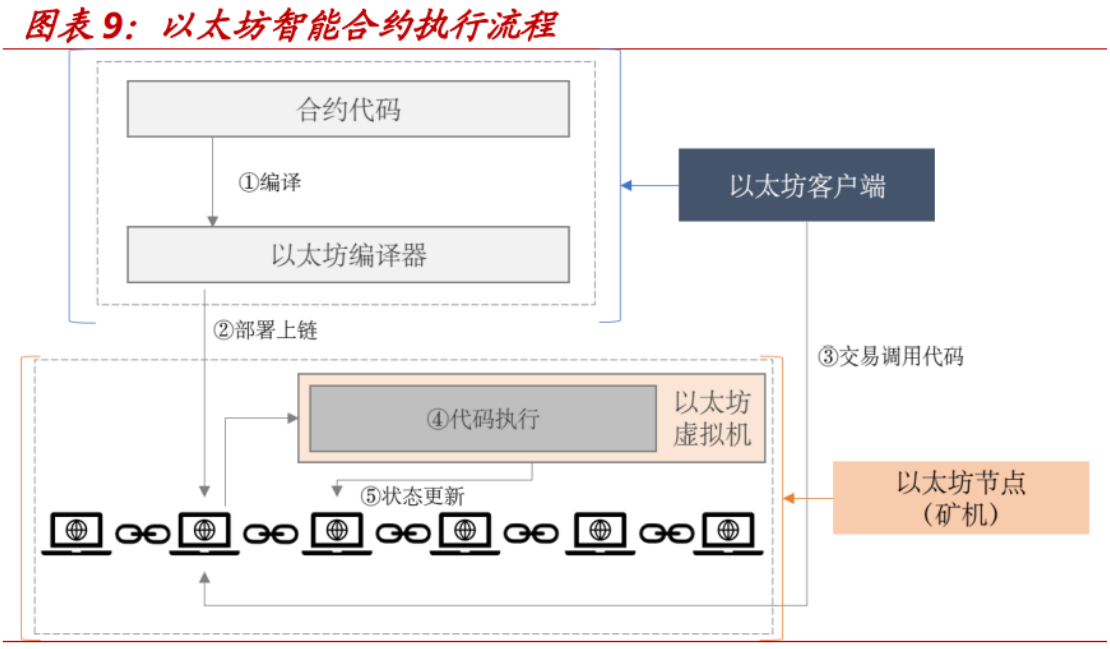


图 3-4 智能合约执行过程

开发智能合约的流行库 OpenZepplin，在线开发智能合约的网站 https://remix.ethereum.org。采用 Solidity 语言编写智能合约，大致的步骤是：编写智能合约源代码文件（solidity 语言）→ 把源代码文件编译成字节码 → 发布交易（需要普通以太钱包支付交易费）→ … → 调用或事件触发执行智能合约代码 → …。

下面实现一个智能合约：迷你银行。

首先从 https://github.com/argotorg/solidity/releases/download/v0.8.34/soljson.js 下载在浏览器中使用的 solidity 语言编译器。

3.1.3 NFT

nft 可以代表版权、加密币、股份（入股）、期权、选票、虚拟地产或者虚拟的稀缺资源（如 ens 域名）等等。在技术层面上，NFT 就是用智能合约实现的唯一资产，而智能合约就是一段部署在区块链上的程序代码。这段代码由以后的事件触发执行或者被直接调用执行。

例子：无聊猿 NFT。开发方将智能合约代码部署至以太坊，NFT 交易平台 OpenSea 收录并展示，当有人在 OpenSea 平台发起购买此无聊猿 NFT 操作，OpenSea 调用 RPC 接口访问以太坊主网发送交易请求，矿工打包交易找到智能合约执行，将链上状态更新完成交易。

3.1.4 DeFi

3.1.5 以太坊地址

以太坊地址有两类：一类是普通账户，另一类为合约账户，前者关联私钥（能签名交易），后者关联合约代码，合约代码只能由普通账户部署和激活执行。

以太币和比特币的公/私钥遵循相同的机制：即私钥为 256 位随机二进制数，通过符合 spec256k1 标准的椭圆曲线函数产生 512 位的公钥，但是由公钥产生钱包地址的方法不同，本小节介绍如何从公钥产生以太币钱包的普通账户地址。合约地址产生机制有所不同，我们在以太坊上创建一个合约时，新生成的合约的地址是根据发送者的地址（普通账户地址）和其已生成的事务数(nonce)确定的，经过 RLP 编码后再经 Keccak-256 哈希运算得出的。

以太币钱包的地址（160 位的二进制数，即 20 个字节）就是对公钥做进一步的变化处理而得到的，变化过程参见图 3-5 所示。



图 3-5 产生地址的过程

其中对 64 个字节的公钥做 keccak256 运算，也可以在 https://emn178.github.io/online-tools/网站在线完成，参见图 3-6 所示。

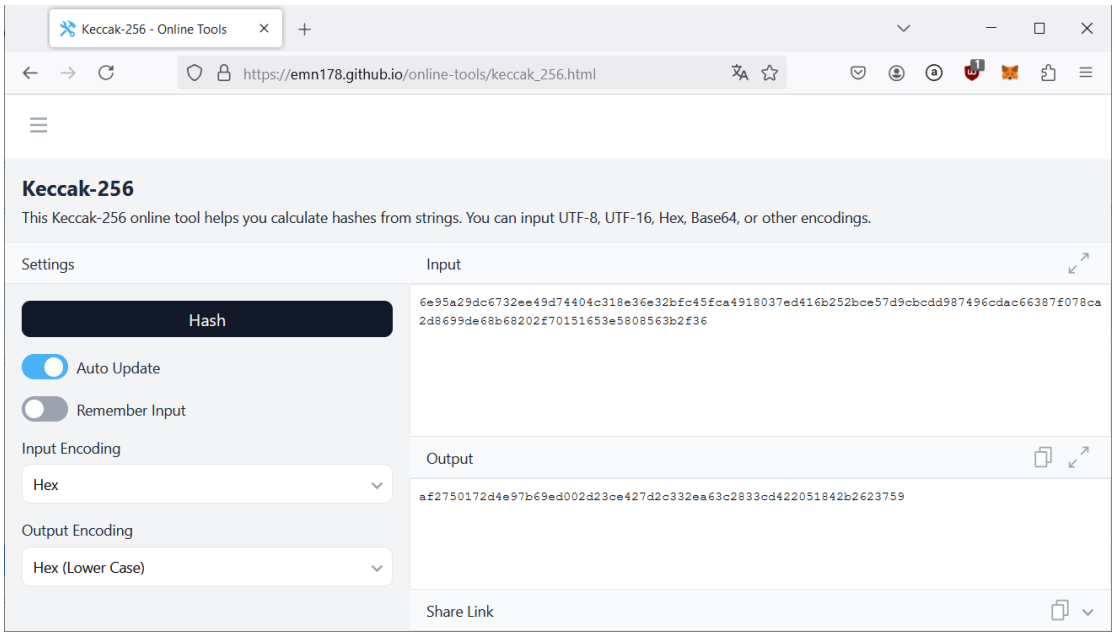


图 3-6 在线 keccak256 计算

以太坊也支持 HD 钱包，路径格式形如“m/44'/60'/i'/0/j”，表示第 i+1 个账户里的第 j+1 个钱包，i,j=0、1、2、3、…、2^{31}-1，当长度允许更长，比如 m/44'/60'/1'/0/2/0/1、m/44'/60'/1'/0/2/0/1/10 都是合法的。但是实际应用时不要太长，否则很容易搞错或忘记。

3.1.6 ENS 域名

以太币作为一种通胀货币，最适合于日常生活当中的支付，但是以太币地址太长，如果通过语言告诉他人那简直就是一场噩梦，所以以太坊推出了 ENS 域名项目（智能合约），把以太币地址解释成便于记忆的字符串——即形如 123.eth 这样 ENS 域名，类似 DNS 把 ip 解析到域名。大家可以去 https://app.ens.domains/注册 ENS 域名，域名至少 3 个字符，域名越短年租金越高，3 个字符的域名每年租金大约 640 美元，4 个字符一年大约 160，5 个字符以上的域名一年约 5 美元。比如我申请的 623456.eth 五年租金 25.26 美元。注册时一般是通过以太币钱包支付以太币，所以在以太币汇率高时注册比较划算。

每一个.eth 域名本质上都是一个符合 ERC-721 标准的 NFT。采用 ethers.js 库注册 ens 域名的代码如下：

```
// 1. 合约配置（主网）
const CONTROLLER_ADDRESS = "0x253553366Da8546fC250F225fe3d25d0C782303b";
const CONTROLLER_ABI = [
    "function makeCommitment(string name, address owner, uint256 duration, bytes32 secret, address
resolver, bytes[] data, bool reverseRecord, uint16 ownerAnyCanReset) public pure returns (bytes32)",
    "function commit(bytes32 commitment) public",
    "function register(string name, address owner, uint256 duration, bytes32 secret, address resolver,
bytes[] data, bool reverseRecord, uint16 ownerAnyCanReset) public payable",
    "function rentPrice(string name, uint256 duration) public view returns (tuple(uint256 base, uint256
premium))",
    "function commitments(bytes32 commitment) public view returns (uint256)"
];

/*
*  参数 domainName - 待注册的 ens 域名，不包含“.eth”后缀。
*  参数 years - 注册年数。
*/
async function ensRegistrationFlow(domainName, years) {
    // 初始化 Provider 和 Signer（以 MetaMask 钱包为例）
    const provider = new ethers.BrowserProvider(window.ethereum);
    const signer = await provider.getSigner();
    const userAddress = await signer.getAddress();

    const contract = new ethers.Contract(CONTROLLER_ADDRESS, CONTROLLER_ABI, signer);

    // 参数设置
    const name = domainName.replace(".eth", ""); // 仅传入前缀
    const duration = years * 31536000; // 1 年等于 31536000 秒（即 365*24*60*60）
    const secret = ethers.randomBytes(32); // 关键：必须保存，建议存入 localStorage
    const resolver = "0x231b0c8410e0e1135b39919038304b5f84f2759f"; // 公共解析器地址

    try {
        // --- 第一步：生成并提交承诺（Commit） ---
        console.log("正在生成承诺...");
        const commitment = await contract.makeCommitment(
            name, userAddress, duration, secret, resolver, [], true, 0
        );

        const commitTx = await contract.commit(commitment);//会消耗 Gas
        console.log("Commit 交易已发出:", commitTx.hash);
        await commitTx.wait();

        // --- 第二步：保存状态 ---
        // 非常重要！如果页面刷新，你需要这些数据来完成第二步
        const sessionData = { name, secret: ethers.hexlify(secret), duration };
        localStorage.setItem(`ens_pending_${name}`, JSON.stringify(sessionData));

        console.log("等待 60 秒（ENS 强制安全期）...");
        await new Promise(resolve => setTimeout(resolve, 61000));

        // --- 第三步：获取价格并注册 ---
        console.log("正在查询价格...");
        const priceResult = await contract.rentPrice(name, duration);
```

```
        // 价格包含 base 和 premium，增加 10% 缓冲区应对波动
        const totalValue = (priceResult.base + priceResult.premium) * 110n / 100n;

        console.log("正在执行最终注册...");
        const registerTx = await contract.register(
            name,
            userAddress,
            duration,
            secret,
            resolver,
            [],
            true, // 设置为 True 会自动同步反向解析
            0,
            { value: totalValue }
        );//会消耗 Gas

        const receipt = await registerTx.wait();
        console.log("注册成功！交易哈希:", receipt.hash);
        localStorage.removeItem(`ens_pending_${name}`);

    } catch (error) {
        console.error("注册流程出错:", error);
    }
}
```

下面以注册 623456.eth 为例说明操作全过程。前提是浏览器要安装好 MetaMask 插件（具体参考附录 C），并且要给 MetaMask 钱包充入一定数量的以太币，虽然使用信用卡也可以支付费用，但是注册过程超麻烦。

打开网址 https://app.ens.domains/，输入 623456 查询是可以注册，见图 3-7 所示。

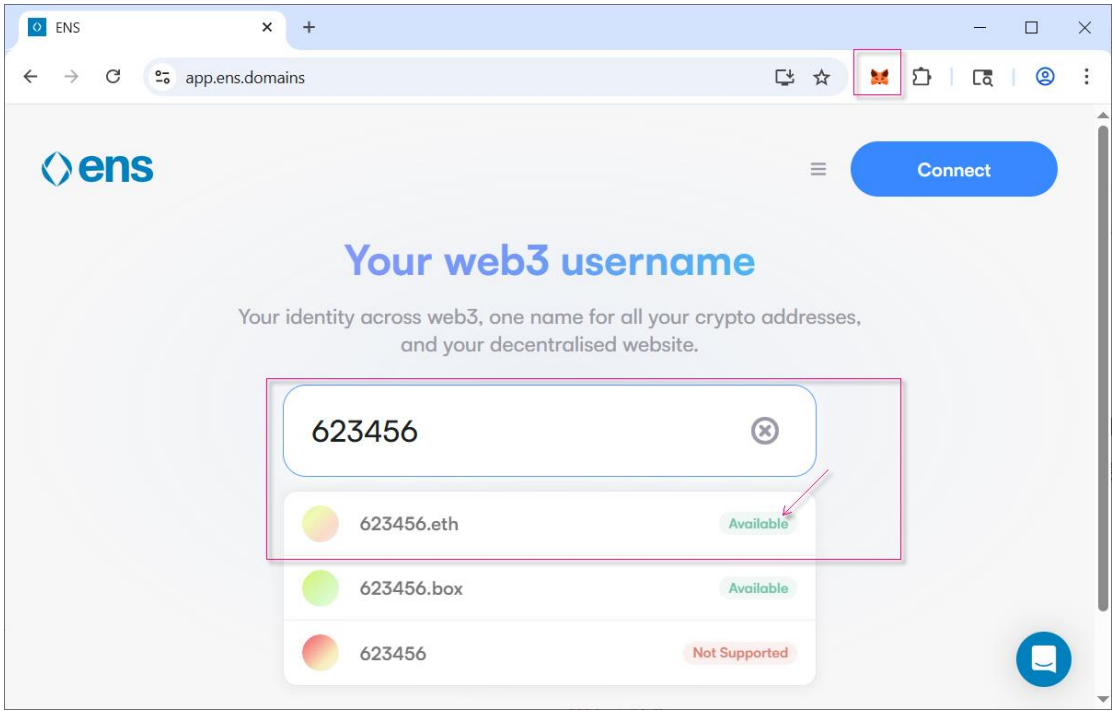


图 3-7 注册 623456.eth 域名

点击图中的“Connect”按钮连接到 MetaMask 钱包，并在钱包中确认。然后点击“Available”打开如图 3-8 所示的界面。

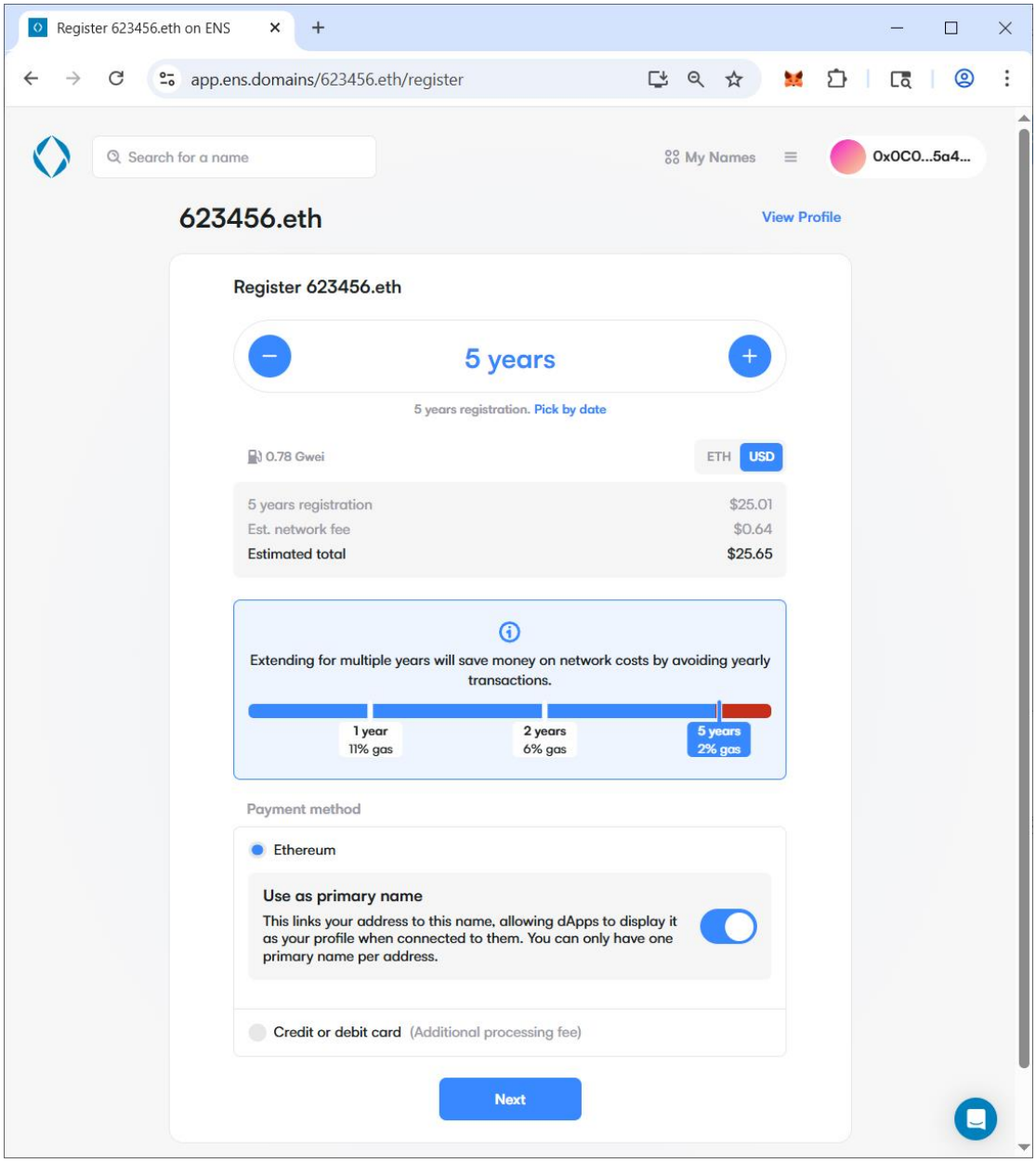


图 3-8 五年租金

点击“Next”按钮出现如图 3-9 所示的画面。

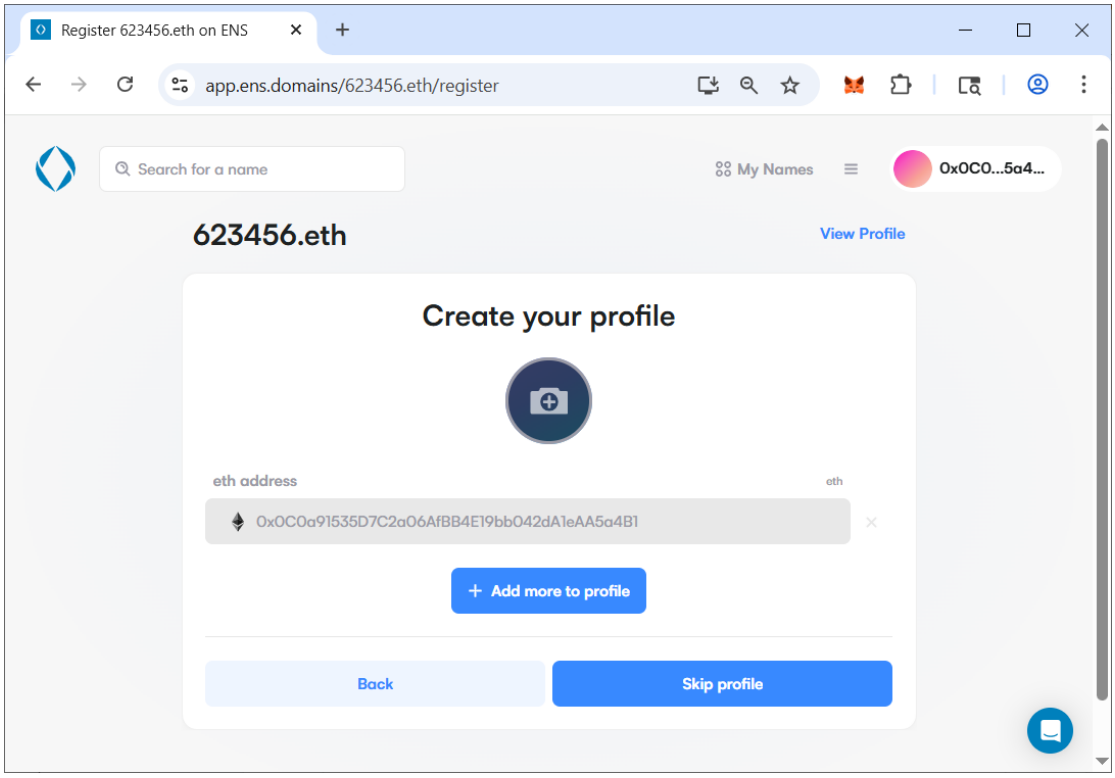


图 3-9 确认个性资料

图中的地址就是 MetaMask 钱包的地址，这里无法改变，等注册完之后再来修改。直接点击“Skip profile”按钮出现图 3-10。

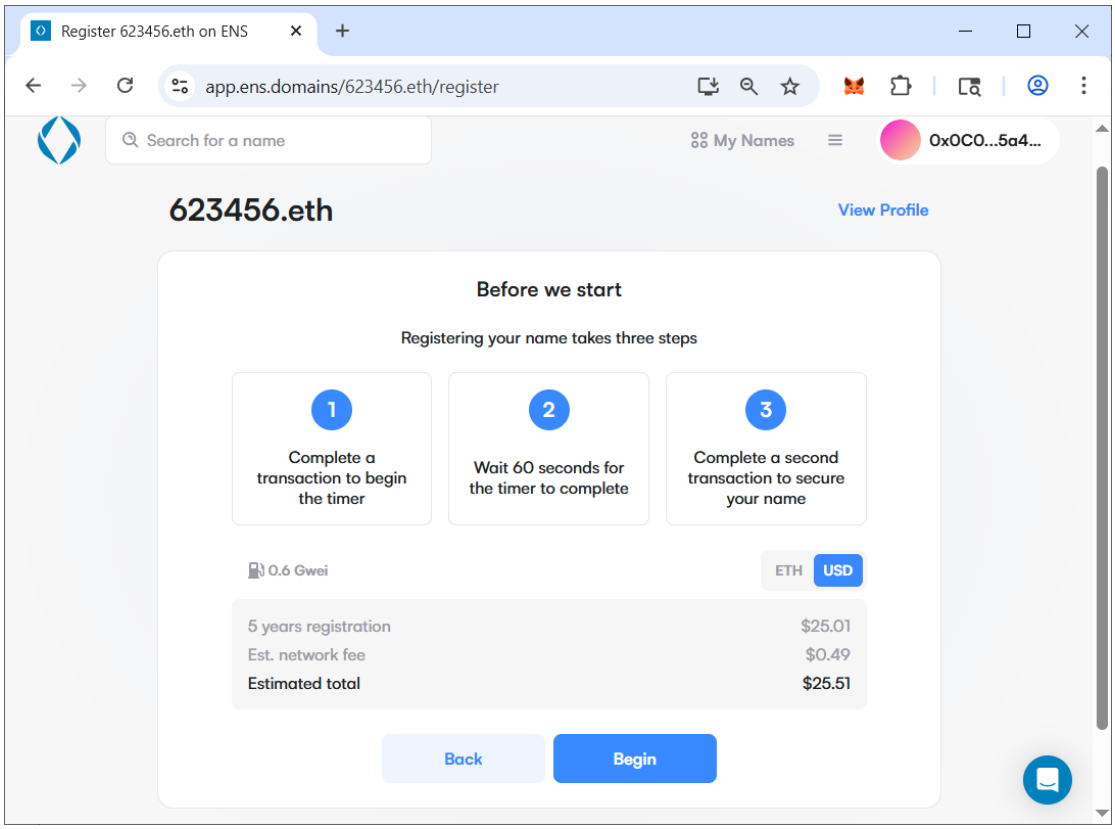


图 3-10 注册概览

说明整个注册过程要经过三步，提交两个交易事务。直接点击“Begin”按钮出现如图 3-11 所示的画面。

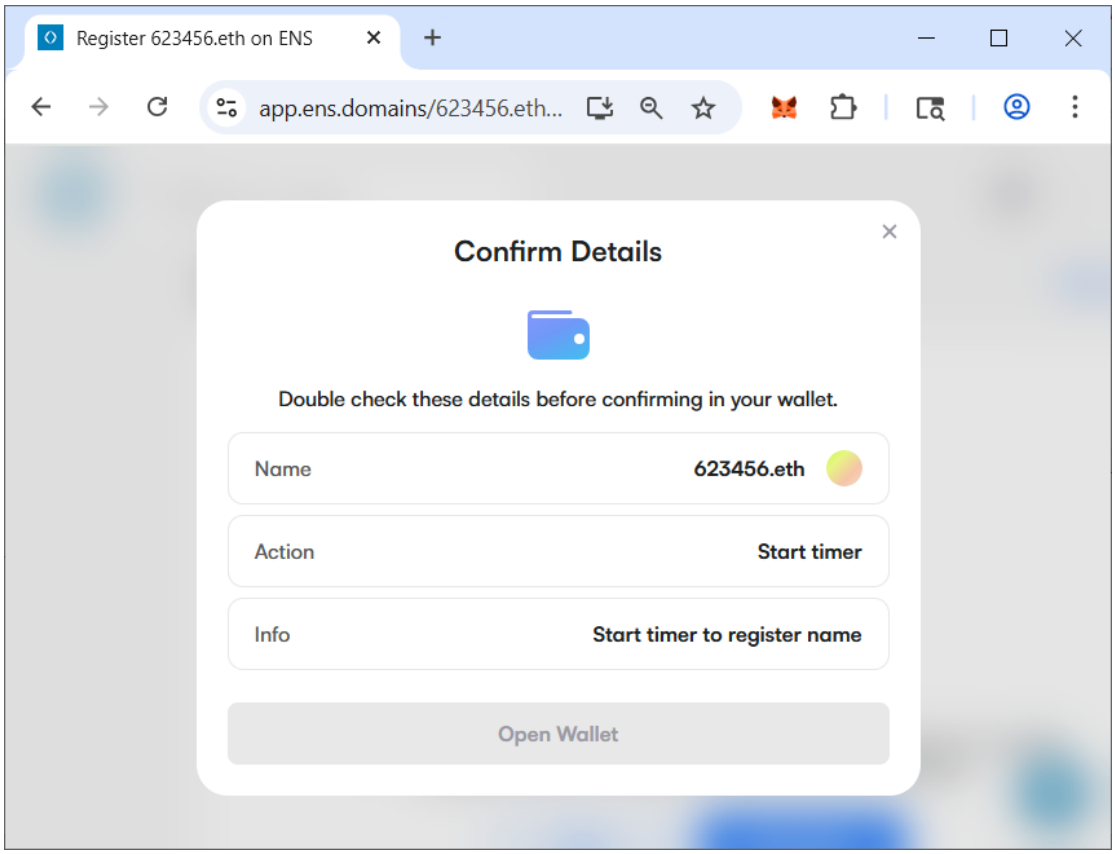


Figure 3-11 在链接钱包前再三确认

点击“Open Wallet”呼叫 MetaMask 钱包，弹出钱包页面让我们确认交易，见图 3-12 所示。

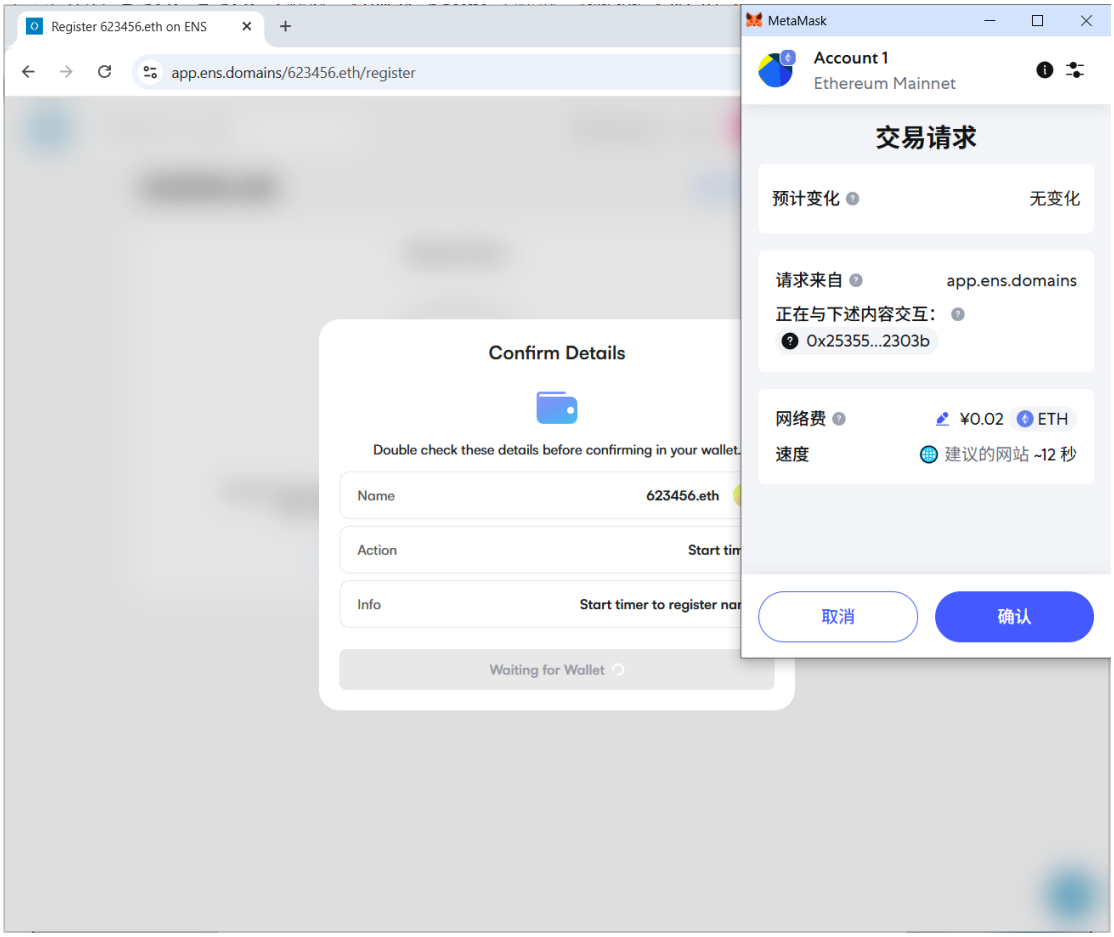


图 3-12 在 MetaMask 中确认

图中显示的网络费需要 0.02 个以太币（约 50 美元），这个不准确。在钱包中确认后，注册页面一直在等交易完成，直到出现图 3-13 所示的界面才表示交易完成。

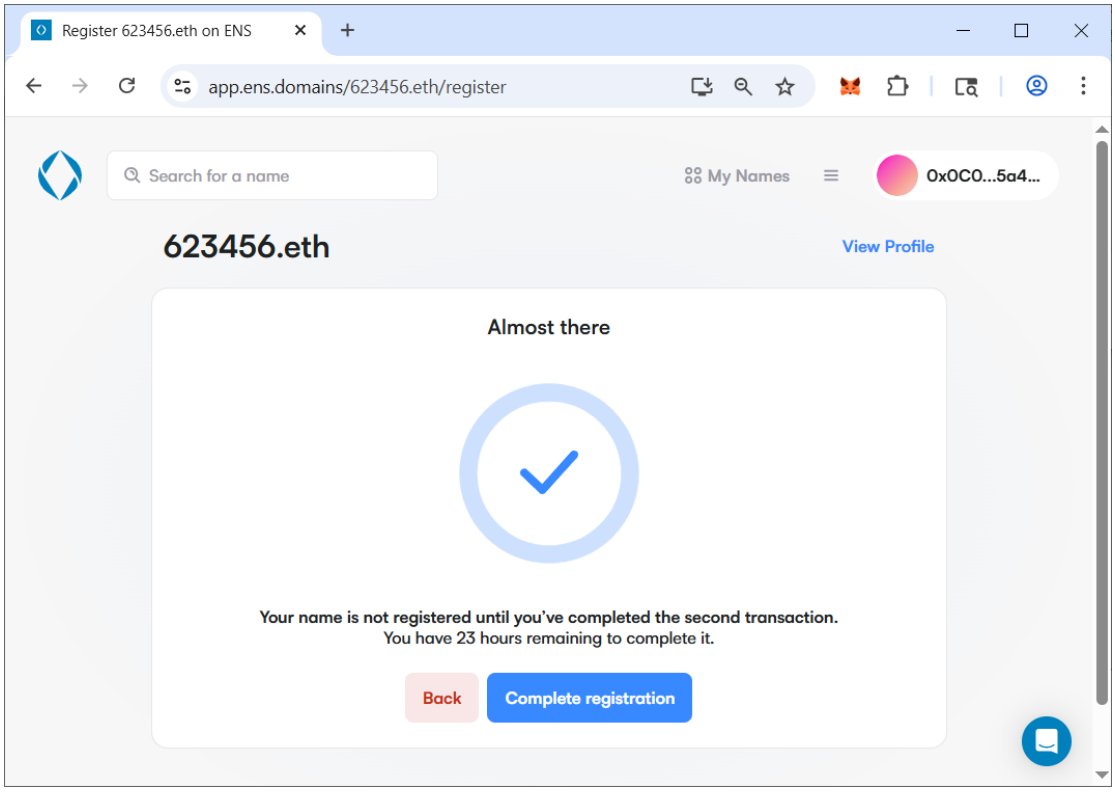


图 3-13 第一个交易完成

告诉我们必须在 23 小时内完成第二个交易，否则本次注册失败，等于白交了第一个交易的手续费。

点击“Complete registration”按钮继续完成第二个交易。然后呼叫钱包，并在钱包中确认。最后出现图 3-14 的画面。

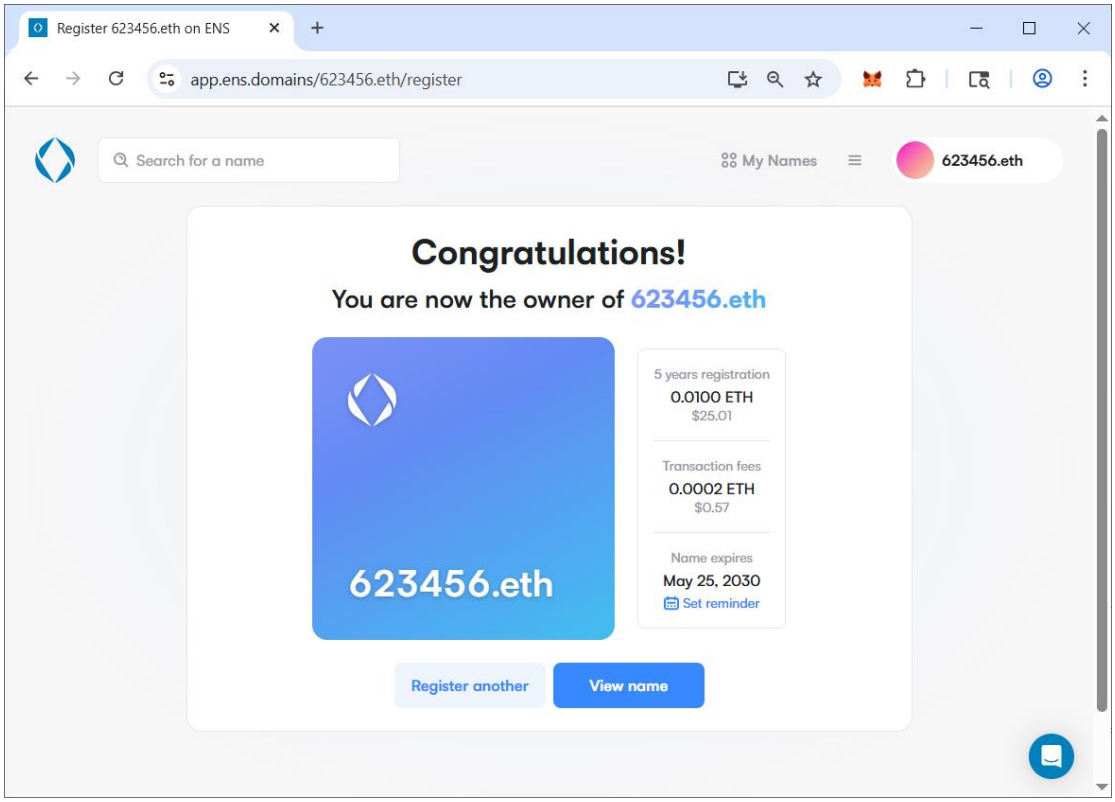


图 3-14 注册完成

点击“View name”按钮，进入管理 ENS 域名界面，见图 3-15 所示。

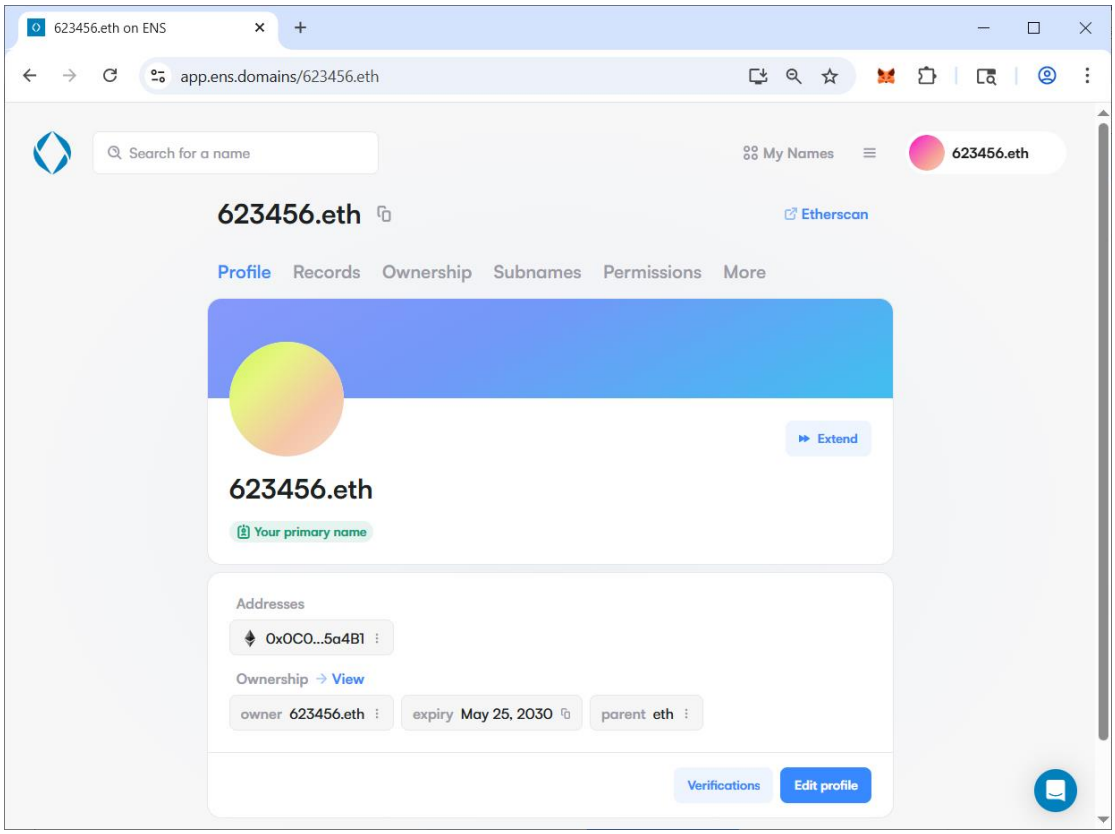


图 3-15 管理 ENS 域名

可在“Records”菜单中修改域名对应的钱包地址，同一个域名还可以增加其他加密币的地址（比如比特币），这有点意思。可在“Subnames”菜单中可添加二级域名，比如 1.623456.eth、234.623456.eth 等，且这些二级域名可以解析到不同的钱包地址。

以后只要打开网址 https://app.ens.domains/623456.eth 即可进入 ENS 域名管理页面，不过在连接到 MetaMask 钱包之前，只能看不能修改，见图 3-16。

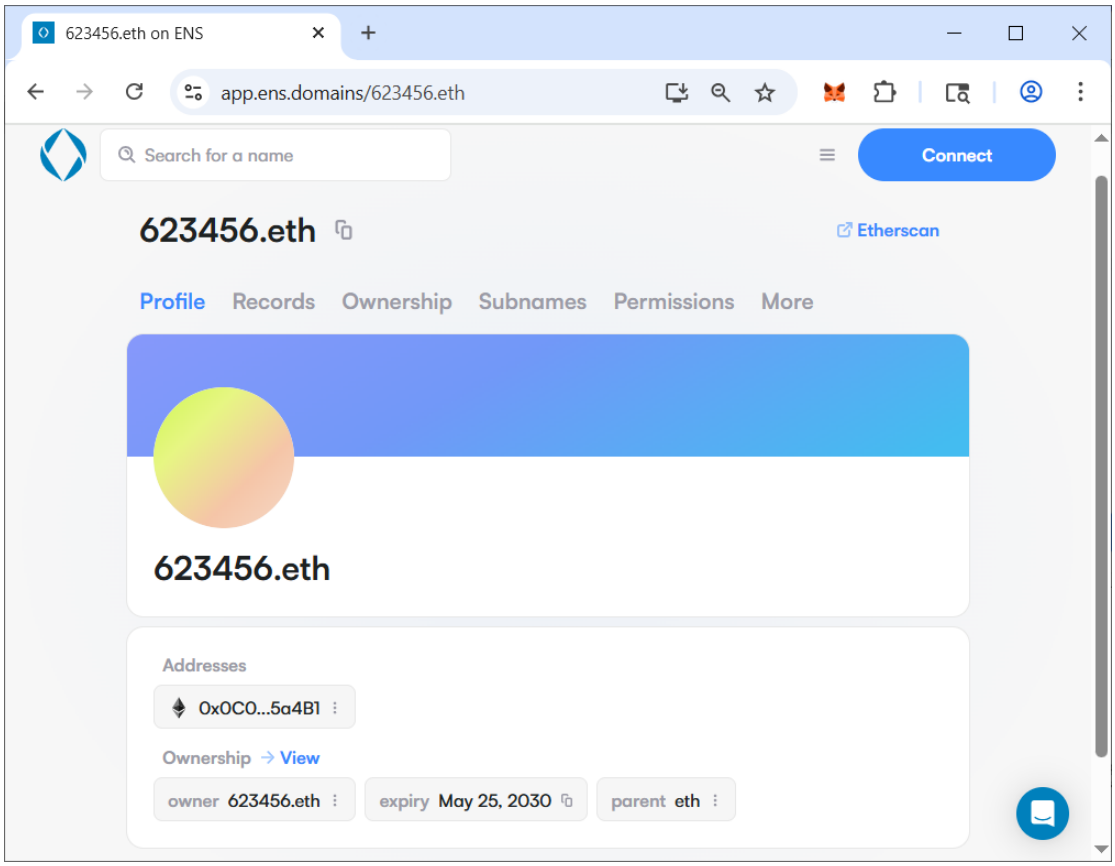


图 3-16 进入 ENS 域名管理

接下来我又注册了一个 2 年期的域名，最终两个域名总的费用是 38 美元不到。

一些建议：

1. 为了方便通过语言（如电话等）告诉别人 ENS 域名，强烈建议域名采用数字，而且避免使用 0 和 1，数字 0 与字母 0 容易混淆，数字 1 与字母 l 容易混淆。联排数字其实也不好，比如 66666、555833，乍一看似乎好记，但容易输入多了或者少了数字。加密币这玩意儿，一旦 ENS 域名或地址搞错，那财富就永远消失了。另外域名越短越好，长度 5 以内的太贵，每年租金数百美元，5、6 位数字已经足够好了。

2. ENS 域名除了添加地址或者二级域名外，不要增加任何诸如电话、姓名、邮件等隐私内容。

3.1.7 交易费用

以太币交易费用的计算多变且复杂，这是最让人诟病的地方。交易费用与交易类型紧密相关，具体请看表 3-1 所示。

表 3-1 交易类型与计费方式

交易类型	类型编号	引入提案	费用计算公式	矿工/验证者收益	销毁部分	适用场景	ethers.js 指定方式
传统交易	0	-	gasPrice × gasUsed	全部费用	无	兼容旧节点	{ type: 0, gasPrice }
EIP-1559 交易	2	EIP-1559	(baseFee + priorityFee) × gasUsed	priorityFee × gasUsed	baseFee × gasUsed	常规转账/合	{ type: 2, maxFeePerGas, maxPriorityFeePerGas }

						约调用	
访问列表交易	1	EIP-2930	gasPrice × (gasUsed + 访问列表节省 Gas)	全部费用	无	复杂合约交互	{ type: 1, accessList, gasPrice }
Blob 交易	3	EIP-4844	(baseFee + priorityFee) × gasUsed + blobFee × blobCount	priorityFee × gasUsed	baseFee × gasUsed + blobFee	Layer2 数据存储	{ type: 3, blobs }

表中 gasUsed 是实际消耗的油量，在发起交易时可以指定愿意承担的最大耗油量（gasLimit，默认是 21000）。现在“EIP-1559 交易”类型使用最广泛，它是 2021 年引入的交易类型。

使用 2 号类型（即 EIP-1559 交易）的交易必须明确指定交易类型（type=2）、每升油的最多小费（maxPriorityFeePerGas）以及最大单价（maxFeePerGas），矿工在打包出块时会自动计算基础单价（baseFee）、小费单价（priorityFee）和实际油耗（gasUsed），必须满足不等式：gasUsed≤gasLimit、priorityFee≤ maxPriorityFeePerGas、(baseFee + priorityFee)≤ maxFeePerGas，因此用户最终支付的交易费小于或等于 gasLimit×maxFeePerGas。矿工获得的收益是 priorityFee×gasUsed，所以我们指定的 maxPriorityFeePerGas 越大就越快被矿工处理，销毁的以太币（baseFee×gasUsed）被转到地址 0x0000000000000000000000000000000000000000，这是一个无法消费的地址，这相当于减少了货币流动量，甚至在交易高发时段还会导致通缩。可用下面的代码获取当前的价格：

```
provider.getFeeData().then((feeData)=>{console.log(`gasPrice=${feeData.gasPrice},maxFeePerGas=${feeData.maxFeePerGas},maxPriorityFeePerGas=${feeData.maxPriorityFeePerGas}`)});
gasPrice=1000027,maxFeePerGas=500500054,maxPriorityFeePerGas=500500000
```

gasPrice 变化不大，maxFeePerGas 和 maxPriorityFeePerGas 变化较大，但是 maxFeePerGas 减去 maxPriorityFeePerGas 一般小于 100（通常在 50~80 之间波动）。

3.2 以太币开发

以太坊的库 web3.js 于 2025 年 3 月停止更新，建议开发者迁移到 ethers.js 或 viem.js。

3.2.1 ethers.js 库

截止到 2025 年 5 月的最新版本 6.14 可从 https://github.com/ethers-io/ethers.js/tree/main/dist 下载 ethers.umd.js（或 ethers.umd.min.js）和 wordlists-extra.min.js，其中 wordlists-extra.min.js 是除英语外的多种语言的助记词库，而英语助记词库包含在 ethers.umd.js 中。官方没有提供 wordlists-extra 的 umd 版本，需要我们自己转换：

```
npm install rollup @rollup/plugin-commonjs --save-dev
vim rollup.config.js                                                    #输入以下内容
import commonjs from '@rollup/plugin-commonjs';
export default {
  input: 'wordlists-extra.min.js',
  output: {
    file: 'wordlists-extra.umd.min.js',
    format: 'umd',
    name: 'wordlistsExtra'
  },
  plugins: [commonjs()]
};

npx rollup -c                                                           #正式转换
```

转换后的文件名是 wordlists-extra.umd.min.js。

在 html 页面文件中导入：

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>以太币开发</title>

    <script src="js/ethers.umd.min.js"></script><!--导入 ethers 对象，专门用于处理以太币-->
    <!--
      如果需要用到其他语言的助记词，就导入它，比如我要用汉字就导入 LangZH，具体使用时 LangZH.wordlist("cn")简体中文，
LangZH.wordlist("tw")繁体中文。
      LangCz, LangEs, LangFr(法语), LangIt(意大利语), LangJa(日语), LangKo(韩语), LangPt(葡萄牙语), LangZh(中文)
    -->
    <script src="js/wordlists-extra.umd.min.js"></script><!--导入 wordlistsExtra 对象 -->

</head>

<body>

    <script>
       window.addEventListener('load', (evt) => {
           console.log(ethers);
           console.log(LangZh);
       });
    </script>

</body>

</html>
```

umd 的通用版本，允许浏览器以 file://协议打开本地文件，file://协议不能导入模块，否则会报如下的错误：

```
Access to script at 'file:///D:/www/js/ethers.min.js' from origin 'null' has been blocked by CORS policy:
Cross origin requests are only supported for protocol schemes: chrome, chrome-extension, chrome-untrusted,
data, http, https, isolated-app.
```

Ethers.js 文档页面请浏览 https://docs.ethers.org/v6/。

ethers 库涉及的几个重要的概念：

Provider（供体）—— 与区块链建立只读连接，只允许查询区块链状态，如账户、区块、交易详情以及事件日志等。

Signer（签名者）—— 提供了操纵账户的全部方法，一个账户对应一个私钥。

Transaction（交易）—— 封装了交易有关的方法和属性。一笔交易本质上就是改变以太坊区块链的状态：放币、部署合约以及修改合约的状态等。

Contract（合约）—— 合约就是部署在区块链的程序，包含代码和分配的存储，可以读取和写入存储。

Receipt（收据）—— 交易提交到区块链后，会被放入内存池（mempool），直到验证者决定打包。交易的变更仅在被打包到区块链后生效，此时会生成收据，包含交易详情，如所在区块、实际支付的费用、使用的 gas、发出的所有事件以及是否成功 / 回滚。

以太坊开发的一般步骤如下：

1. 连接到以太坊以获取 Provider 和 Signer

最常见的方法是采用浏览器的 MetaMask 插件（具体安装请参考附录 C），注册用户并成功登陆后会向浏览器注入全局变量 window.ethereum，此后采用下面的代码连接区块链：

```
let signer = null;
```

```
let provider;
if (window.ethereum == null) {
    console.log("没有安装 MetaMask 插件，采用只读的默认供体");
    provider = ethers.getDefaultProvider();
} else {
    provider = new ethers.BrowserProvider(window.ethereum);
    signer = await provider.getSigner();
}
```

获得了 Provider 和 Signer 两个对象。但是很不幸 MetaMask 插件已经被墙了，只好采用 https://etherscan.io/提供的 api 了上面代码 ethers.getDefaultProvider()获取一个默认供体，很可能失败，这时可指定使用自己的 token，比如使用我在 etherscan.io 上申请的 token 创建供体：

```
provider = ethers.getDefaultProvider("homestead", { etherscan: "NCGAQ33ESD34Y343Z4HHZHMT9D4DBFA9HK" });
```

通过非 MetaMask 获取的 Provider 没有 getSinger()方法，所以没法获得 Signer 对象。

2. 获得 Contract（合约）对象

```
const contract = new ethers.Contract(
    contractAddress,                                                    // 部署的地址
    contractABI,                                                        // 编译合约时生成的 API
    signer                                                              // 上面得到的 signer
);

const tx = await contract.addProposal(args);                           // 通过 contract.funName() 来调用合约中的方法
await tx.wait();                                                        // 等待合约交易成功
```

3. 获得 Wallet（钱包）和收据对象（需要发起交易）

```
const wallet = new ethers.Wallet(PRIVATEKEY, provider);
const value = Number(this.curETH) - 0;

const tx = {                                                            // 创建交易请求
    to: contractAddress,
    value: ethers.parseEther(value.toString()),
};

const txRes = await wallet.sendTransaction(tx);                        // 发送交易，获得收据
const receipt = await txRes.wait();                                     // 等待链上确认交易
console.log(receipt);                                                   // 打印交易的收据
```

4. 监听事件

```
//不采用 contract.on()方法，而改用 filters 监听事件，这能稳定触发，GetProposals()为合约里的方法
const myEventFilter = contract.filters.GetProposals();
// 订阅事件
const myEventListener = contract.on(myEventFilter, (event) => {
    console.log(
        "MyEvent fired!", //事件返回的参数
        event.args[0],   //from
        event.args[1],   //proposal_name
        event.args[2],   //voteCount
        event.args[3]    //id
    );
```

3.2.1.1 Provider

获取区块数据的抽象类，定义了一系列的方法，这些方法大致归为下面几类。

一、 账户类

1. provider.getBalance (address [, blockTag = latest]) ⇒ Promise< BigNumber >：获取地址的余额（单位 wei），地址可为以太坊域名，如 abraham.eth。例如：

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>以太币开发</title>

    <script type="module">
        import { ethers } from "./js/ethers.min.js";
        window.ethers = ethers;
    </script>

    <!--如果需要用到其他语言的助记词，就导入它，比如我要用汉字
      LangCz, LangEs, LangFr(法语), LangIt(意大利语), LangJa(日语), LangKo(韩语), LangPt(葡萄牙语), LangZh(中文)
    -->
    <script type="module">
        import { LangZh } from "./js/wordlists-extra.min.js";
        window.LangZh = LangZh;
    </script>

</head>

<body>

    <script>
        var signer = null;
        var provider;
        window.addEventListener('load', (evt) => {
            allstuff();
            provider.getBalance("abraham.eth").then((balance) => { alert(balance) });
        });
        async function allstuff() {
            if (window.ethereum == null) {
                console.log("没有安装 MetaMask 插件，采用只读的默认供体");
                //provider = ethers.getDefaultProvider();
                provider = ethers.getDefaultProvider("homestead", {
                    etherscan: "NCGAQ33ESD34Y343Z4HHZHMT9D4DBFA9HK"
                });
            } else {
                provider = new ethers.BrowserProvider(window.ethereum);
                signer = await provider.getSigner();
            }
        }
    </script>

</body>

</html>
```

2. provider.getCode(address [, blockTag = latest]) ⇒ Promise< string< DataHexString >：获取地址的合约代码，返回 0x 前缀的十六进制传，0x0 表示没有合约代码。

3. broadcastTransaction(signedTx: string)⇒ Promise< TransactionResponse >：发布已经签名的交易（HEX 格式），最终解析成 TransactionResponse 对象，此对象主要的属性有 blockNumber（交易加入的区块号）、hash（交易 id 号）、to（入账地址）、value（转账额度）。

3.2.1.2 Signer

3.2.1.3 Mnemonic

实现了 BIP-39 规范的随机数与助记词互相转换的类。

1. 属性

entropy - 只读属性，随机数；password - 只读属性，助记词保护密码；phrase - 只读属性，助记词；wordlist - 只读属性，助记词所在的语言。

2. 实例化 Mnemonic

fromEntropy(entropy: BytesLike, password?: null | string, wordlist?: null | Wordlist)⇒ Mnemonic - 从随机数实例化 Mnemonic 类。参数 entropy 是 Uint8Array 类型的随机数，产生的助记词包含的单词个数 12、15、18、21 或者 24，具体由随机数所占的字节数决定，即分别对应随机数的字节数 16、20、24、28、32；参数 wordlist 指定语言，如 LangZH.wordlist('tw')，默认是英语。如：

```
let mnemonic = ethers.Mnemonic.fromEntropy(ethers.randomBytes(24),'passwd123');
console.log(mnemonic.phrase);
idle trap abandon prison plastic seminar finish hour witness trouble human sleep cheese rain inhale ask
drift piece

mnemonic = ethers.Mnemonic.fromEntropy(ethers.randomBytes(24),'passwd123', LangZh.wordlist('cn'));
console.log(mnemonic.phrase);
富 累 龙 剥 王 灵 胶 找 流 分 迟 甚 愈 矩 季 愤 迈 床
```

fromPhrase(phrase: string, password?: null | string, wordlist?: null | Wordlist)⇒ Mnemonic - 从助记词实例化 Mnemonic 类。

3. 方法

computeSeed()⇒ string - 计算种子。

entropyToPhrase(entropy: BytesLike, wordlist?: null | Wordlist)⇒ string - 静态方法，直接从随机数到指定语言的助记词。

isValidMnemonic(phrase: string, wordlist?: null | Wordlist)⇒ boolean - 静态方法，判断助记词是否合法。

phraseToEntropy(phrase: string, wordlist?: null | Wordlist)⇒ string - 静态方法，从助记词到随机数。

3.2.1.4 Wallet

Wallet 类继承 BaseWallet 和 AbstractSigner，专门管理单个私钥（也就是一个钱包）。

1. 实例化钱包对象的四种方法：

new Wallet(key: string | SigningKey, provider?: null | Provider) - 从私钥 key 实例化。

createRandom(provider?: null | Provider)⇒ HDNodeWallet | Wallet - 新建 HD 钱包对象，路径是

"m/44'/60'/0'/0/0"。

fromEncryptedJsonSync(json: string, password: Uint8Array | string)⇒ HDNodeWallet | Wallet - 通过解密 json 格式的钱包实例化。方法 wallet.encryptSync(password)可产生加密 json 格式的钱包。

fromPhrase(phrase: string, provider?: Provider)⇒ HDNodeWallet - 从助记词产生钱包对象。

2. 属性：

address - 钱包地址；mnemonic - 助记词对象 Mnemonic 的实例；publicKey - 公钥；privateKey - 私钥；path - HD 钱包路径，默认是 m/44'/60'/0'/0/0；index - 索引，默认为 0；fingerprint - 指纹；chainCode - 链码；depth - HD 层数，对于 m/44'/60'/0'/0/0 就是 5 层，m/44'/60'/0'/0/0/3 就是 6 层。

3. 方法

sendTransaction(tx: TransactionRequest)⇒ Promise< TransactionResponse > - 发布交易（会自动签名）。接口 TransactionRequest 代表一个交易，其定义如下：

to⇒ null | AddressLike ：对方地址或 ENS 域名；value⇒ null | BigNumberish ：转账金额（单位 wei）；gasPrice⇒ null | BigNumberish - 汽油单价（单位 wei），每笔交易耗油 21000 升，等于说交易费用是 21000*gasPrice，按照今天的汇率一枚以太币兑换 1800 美元算，如果我愿意出交易费 1 美元，推算汽油单价为 gasPrice = 10^{18}/(21000*1800) ≈ 26000000000=26Gwei；from⇒ null | AddressLike - 我方地址；gasLimit⇒ null | BigNumberish - 耗费最多汽油数；maxFeePerBlobGas⇒ null | BigNumberish - 愿为每升汽油支付的小费上限；maxFeePerGas⇒ null | BigNumberish - 每升汽油的交易费上限；data⇒ null | string - 备注内容，可用于把信息永久保存在以太坊上，最多可达 7000 汉字，字数越多，交易费就越贵。

其中 maxFeePerBlobGas 和 maxFeePerGas 用于目前使用最广泛的 2 号交易类型，详细介绍请参考“§3.1.6 交易费用”小节。

备注转换为 HEX 格式后保存在 data 属性上，可以使用 ethers.js 的几个方法实现 utf8 与 hex 的相互转换，看例子：

```
const comment = "吾本草民，蹢躅尘埃，苟活于世，生既无益，死亦何损？秉笔流涕，心与世辞。123abc";
const data = ethers.hexlify(ethers.toUtf8Bytes(comment));                    //UTF8 字符串->HEX
console.log(data);
0xe590bee69cace88d89e6b091efbc8ce8b8afe8ba85e5b098e59f83efbc8ce88b9fe6b4bbe4ba8ee4b896efbc8ce7949fe697a2e6
97a0e79b8aefbc8ce6adbbe4baa6e4bd95e68d9fefbc9fe7a789e7ac94e6b581e6b695efbc8ce5bf83e4b88ee4b896e8be9ee38082

const origin = ethers.toUtf8String(data);                                    //HEX->UTF8 字符串
console.log(origin);
吾本草民，蹢躅尘埃，苟活于世，生既无益，死亦何损？秉笔流涕，心与世辞。
```

也可去 https://onlinetools.com/hex/convert-hex-to-utf8 网站完成 utf8 与 hex 的相互转换。

signMessageSync(message: string | Uint8Array)⇒ string - 对一段文本进行签名，返回的签名是带前缀 0x 的十六进制字符串。参数 message 就是被签名的字符串文本。验证签名可以采用下面的方法：

ethers.verifyMessage(message: Uint8Array | string, sig: SignatureLike)⇒ string - 验证一段文本的签名。参数 sig 就是签名（带 0x 前缀）。方法返回一个以太币地址，然后比较从公钥导出的地址是否相等来间接验证签名是否成功。看例子：

```
const privateKey = '0xb5561cd1984c9dd40513c6361f80f0ece77b44d32da43ae050580de99b593341';
const msg = '吹龙笛，击鼍鼓；皓齿歌，细腰舞。';

const wallet = new ethers.Wallet(privateKey);
const signature = wallet.signMessageSync(msg);
console.log(signature);
0x2eaa7f1c2c6dd3f91bd2354f29b534b6b291f5a930967274366d89663f547a8d1c41d993ed712f7d624cbee3a945a0042005d99ac16e895364cb0eb9e3bcdffc1b

const publicKey = '0x03e826f71fb384b8ccacff108696cbfc9c869666e4d0b7a888724c79ba45950aa4';
const address1 = ethers.computeAddress(publicKey);
const address2 = ethers.verifyMessage(msg, signature);
const isValid = address1.toLowerCase() === address2.toLowerCase() ? true : false;
if(isValid){
     console.log('验证成功! ');
}else{
     console.log('验证失败! ');
}
```

3.2.1.5 HDNodeWallet

HDNodeWallet 类继承 BaseWallet 和 AbstractSigner，专门管理 HD 钱包，属性与 Wallet 差不多，只不过多了一个扩展私钥属性 extendedKey。

1. 实例化

createRandom(password?: string, path?: string, wordlist?: Wordlist)⇒ HDNodeWallet - 新建一个 HD 钱包。参数 password 为助记词保护密码；path 钱包路径，如 m/44'/60'/0'/0/1；wordlist 为助记词所在的语言，如简体中文 LangZh.wordlist('cn')。例如从繁体中文种选择助记词，并产生 HD 钱包：

```
let hd = ethers.HDNodeWallet.createRandom("a12345678!", "m/44'/60'/0'/0/1", LangZh.wordlist("tw"));
console.log(hd.mnemonic.phrase);
肥 既 間 西 鄰 圍 店 遲 朋 賴 車 艦
```

fromMnemonic(mnemonic: Mnemonic, path?: string)⇒ HDNodeWallet - 从助记词对象恢复 HD 钱包，参数 mnemonic 为 Mnemonic 对象，参数 path 为 HD 钱包路径，默认是"m/44'/60'/0'/0/0"。例如：

```
const mnemonic = "idle trap abandon prison plastic seminar finish hour witness trouble human sleep cheese
rain inhale ask drift piece";
const Mnem = ethers.Mnemonic.fromPhrase(mnemonic, "Pass123");
const Hd_wallet = ethers.HDNodeWallet.fromMnemonic(Mnem);
```

fromPhrase(phrase: string, password?: string, path?: string, wordlist?: Wordlist)⇒ HDNodeWallet - 从助记词恢复 HD 钱包。参数 phrase 为助记词，password 为助记词保护密码，path 为 HD 钱包路径，wordlist 为助记词所在是语言。看例子：

```
const mnem = 'arm female elevator raccoon solid identify number arena arrive detail enlist fury';
const hd = ethers.HDNodeWallet.fromPhrase(mnem,'Pass321!');
console.log(hd.mnemonic.computeSeed());                                                    //种子
```

```
0xa95cc57248b20bb397e43563ef172136703f3c027bd5669fa4ac1a7b6f26dce830f1e7773dfe51e53d5e2f0bb5f02f882cc196f19f2e9510de67bd945d217600
console.log(hd.extendedKey);                                                                                    //扩展私钥
xprvA3wFEdzfuAsXs99TEAJYcXibJTAgpAwU5X4tJHiPgmHtDssX6jmCrLTEL2UPK7a8EBWWp49uEiE9FCRxHEHxuii2q8eaJEtTdn1dMGbTQuj
```

fromSeed(seed: BytesLike)⇒ HDNodeWallet - 从种子恢复 HD 钱包。参数 seed 为 64 个字节的种子（随机数），可以用 HDNodeWallet.mnemonic.computeSeed()方法导出 HD 钱包（适合于包含助记词的钱包）的种子。看例子：

```
let seed = ethers.randomBytes(64);
const hdWallet = ethers.HDNodeWallet.fromSeed(seed);
```

fromExtendedKey(extendedKey: string)⇒ HDNodeWallet | HDNodeVoidWallet - 从根扩展私钥恢复 HD 钱包。

2. 方法

deriveChild(index: Numeric)⇒ HDNodeWallet - 产生下一代中的第 index 个 HD 钱包，index=0、1、2、…。例如当前 HD 钱包的路径是 m/44'/60'/0'/0/0，那么 deriveChild(2)的路径就是 m/44'/60'/0'/0/0/2。

derivePath(path: string)⇒ HDNodeWallet - 产生若干辈后代的 HD 钱包，参数 path 是相对本钱包而言的路径，如"1/4"表示我的第 2 个儿子的第 5 个儿子（因为 0 为第 1 个）对应的 HD 钱包，即绝对路径就是"m/44'/60'/0'/0/0/1/4"（假如本钱包的路径是"m/44'/60'/0'/0/0"）。deriveChild(5)等价于 derivePath("5")。

encrypt(password: Uint8Array | string, progressCallback?: ProgressCallback)⇒ Promise< string > - 把 HD 钱包加密成 JSON 钱包。如果没省略 progressCallback 参数，那么会周期调用此函数以显示进度。例子：

```
Hd_wallet.encrypt("A1b2C3!").then(js => console.log(js));
{
   "address": "58dc96af1ad1d72706899195ebc612cd42814415",
   "id": "e0eac7e3-41fc-4ef9-815f-2d8db74c050c",
   "version": 3,
   "Crypto": {
      "cipher": "aes-128-ctr",
      "cipherparams": {
         "iv": "8c157ff72bb652785d9871e0ca464315"
      },
      "ciphertext": "2a08d3b573df41dae14f4cd28e1302c488d495b4df00eee04404decf1596c489",
      "kdf": "scrypt",
      "kdfparams": {
         "salt": "685dc6103013fe3dbdfb3ae013ec2758a7a5b6d4efc986283fb787c993ad2f7d",
         "n": 131072,
         "dklen": 32,
         "p": 1,
         "r": 8
      },
      "mac": "50c781fca9b5acb72770e4abab0ba6a347cf65da585a417e626dd896016e70a5"
   }
}
```

encryptSync(password: Uint8Array | string)⇒ string 为同步加密函数。ethers.Wallet.fromEncryptedJsonSync()方法能解密 JSON 钱包。

neuter()⇒ HDNodeVoidWallet - 转换为观察 HD 钱包（移除了私钥）。

3.2.1.6 Transaction

代表一次对区块链的修改操作，比如一笔交易。有两种方法实例化，一是“new Transaction()”全部采用默认值实例化，

二是“Transaction.from(tx?: string | TransactionLike < string >)⇒ Transaction”从序列化的 hex 串实例化。例如：

```
const txHex =
'0x01f87083aa36a7808504d20cf53382526c94100f5ad4a17684f9bbe3670fd6bd0068e50c95708711c37937e0800080c001a0528
47dfbde95517bd90f7da51aa9196a7cff33fe11edaa41975dac8b6804000ca06045b6de4b796c7985e199c66eebd0277e5922f7418
17d61288b9f52f60fc608';
const tx = ethers.Transaction.from(txHex);
```

1. 常用属性

chainId⇒ bigint - 区块链 id，主链 1，Sepolia 测试链 11155111；data⇒ string - 备注；from⇒ null | string - 转出地址；fromPublicKey⇒ null | string - 签名者的公钥（已经签名的交易）；gasLimit⇒ bigint - 耗油上限；gasPrice⇒ null | bigint - 油价；hash⇒ null | string - 交易 id；maxFeePerGas⇒ null | bigint - 最高油价；maxPriorityFeePerGas⇒ null | bigint - 给矿工的小费（每升汽油多少 wei）以便激励矿工愿意优先处理本交易；nonce⇒ number - 流水号；serialized⇒ string - 对已经签名的交易序列化（产生 Hex 字符串）；signature⇒ null | Signature - 签名；type⇒ null | number - 交易类型的代号；typeName⇒ null | string - 交易类型的名称；unsignedSerialized⇒ string - 交易对象中需要被签名保护的那些字段序列化，后续做法是对它计算哈希之后签名；value⇒ bigint - 转账额度。还有一些不常用的属性。

2. 常用的方法

isSigned()⇒ boolean - 是否已经签名。

toJSON()⇒ any - 返回 json 对象。

clone()⇒ Transaction - 克隆

3.2.2 综合例子

为了测试，请先准备几个以太坊钱包，以太坊的地址不分测试和正式地址，例如下面是我准备的两个钱包：

用助记词 educate boost author base own issue area cheap palace century common cup 在网站 https://iancoleman.io/bip39/产生的钱包：

1. 钱包路径 m/44'/60'/0'/0/0：

私钥：0xa6df52d32548bc18c1441a9e33b658c06e4ca117e59bf2dab20cbc088f3c6291

公钥：0x026eec7ccf29c60930e3927d2512e166f511c833b903f5001f3c808f13dae37dcb

地址：0xDeb5BF379E49300c78276167fBb8E3F18EA6C946

2. 钱包路径 m/44'/60'/0'/0/1：

私钥：0x6b714fb46b3558192193712dd9dacabd0e29f75b02c81e21d4fc675f9d50719a

公钥：0x02c225dcebb72cf0c08ba75fa94e1597154e2efd35b912e653ddb7c5cd8a39c653

地址：0xB8309ea73A7C61599bF33C73085D24511E042b06

3. 钱包路径 m/44'/60'/0'/0/2：

私钥：0xdf3ac918c6fec262c51d6c43f6654332b5046be7587dc77caf0cbad9ea1926c8

公钥：0x03309e2771d15d1bd988f94b5899d39388a3c40e9c08426a6db208e55baa59749c

地址：0x100F5ad4a17684f9BBE3670fD6bd0068E50C9570

从 https://cloud.google.com/application/web3/faucet/ethereum/sepolia 往第一个钱包充值 0.05 个以太币，每个谷歌账户每天只能充值一次。可去 https://sepolia.etherscan.io/查询测试用的钱包余额。

去获得 https://developer.metamask.io/（siguo*|Hua#ei*）获得 apiKey: 5b34a68e44cf47c3afcd16c4a9b74341，然后构建测试网端点：https://sepolia.infura.io/v3/{YOUR_API_KEY}和生产网端点：https://mainnet.infura.io/v3/{YOUR_API_KEY}。

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>以太币开发</title>

    <script type="module">
        import { ethers } from "./js/ethers.min.js";
        window.ethers = ethers;
    </script>

<!--如果需要用到其他语言的助记词，就导入它，比如我要用汉字
     LangCz, LangEs, LangFr(法语), LangIt(意大利语), LangJa(日语), LangKo(韩语), LangPt(葡萄牙语), LangZh(中文)
    <script type="module">
        import { LangZh } from "./js/wordlists-extra.min.js";
        window.LangZh = LangZh;
    </script>
-->

</head>

<body>

    <script>
        var signer = null;
        var provider;
        window.addEventListener('load', (evt) => {
            allstuff(true);
            if (!provider) {
                alert('初始化 provider 失败! ');
                return;
            }

/* 产生钱包的 4 种方法 */
           //方法 1 - 从私钥产生钱包:
           const privateKey = '0xa6df52d32548bc18c1441a9e33b658c06e4ca117e59bf2dab20cbc088f3c6291';
           const wallet = new ethers.Wallet(privateKey, provider);

           //方法 2 - 从助记词产生钱包:
           //const mnemonic = "educate boost author base own issue area cheap palace century common cup";
           //const wallet = ethers.Wallet.fromPhrase(mnemonic, provider);
```

```
          //方法 3 - 新建钱包:
          //const wallet = ethers.Wallet.createRandom();

          //方法 4 - 解密钱包:
          //const password = 'A1b2C3';
          //const wallet = ethers.Wallet.fromEncryptedJsonSync(encryptedJson, password);            //解密
          ////const encryptedJson  = wallet.encryptSync(password);                                  //加密
/*   The End   */

          const toAddress = '0xB8309ea73A7C61599bF33C73085D24511E042b06';//也可以是 ENS 域名，如 abraham.eth。
          const tx = {
             to: toAddress,
             value: ethers.parseEther('0.01'), // 转换为 wei
             // gasLimit: 21000, // 可选，标准转账的 gas limit
             // gasPrice: ethers.parseUnits("20", "gwei") // 可选，自定义 gas 价格
          };
          wallet.sendTransaction(tx).then((transaction) => {
             console.log(`交易已发送，交易 ID: ${transaction.hash}`);
             transaction.wait().then((receipt) => {
                console.log(`交易已确认，区块号: ${receipt.blockNumber}`);
                provider.getBalance(toAddress).then((balance) => {
                   alert(`对方钱包余额: ${ethers.formatEther(balance)}`);
                });
                provider.getBalance(wallet.address).then((balance) => {
                   alert(`我方钱包余额: ${ethers.formatEther(balance)}`);
                });
             });
          });
       });

       async function allstuff(isTestNet) {
          if (window.ethereum == null) {
             console.log("没有安装 MetaMask 插件，采用只读的默认供体");
             const network = isTestNet ? "sepolia" : "homestead";

/* 产生供体的三种方法 */
             //方法 1:
             //provider = ethers.getDefaultProvider(network);

             //方法 2:
             // provider = ethers.getDefaultProvider(network, {
             //    etherscan: "NCGAQ33ESD34Y343Z4HHZHMT9D4DBFA9HK"
             // });

             //方法 3:
             const apiKey = '5b34a68e44cf47c3afcd16c4a9b74341';
             const rpcURL = `https://${isTestNet ? "sepolia" : "mainnet"}.infura.io/v3/${apiKey}`;
             const provider = new ethers.JsonRpcProvider(rpcURL);
/*  The End  */

          } else {
             provider = new ethers.BrowserProvider(window.ethereum);
             signer = await provider.getSigner();
          }
       }
    </script>

</body>

</html>
```

4 理论进阶

4.1 椭圆曲线

从私钥提取公钥加密币采用椭圆曲线。

椭圆曲线的一般式为$y^2 = x^3 + ax + b$，而比特币公钥采用 spec256k1 标准的特殊椭圆曲线从私钥“计算”出来公钥，此椭圆函数为 $y^2 = (x^3 + 7)\ mod\ p$ ，其中 $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 = 115792089237316195423570985008687907853269984665640564039457584007908834671663$，也就是把 $y^2 = (x^3 + 7)$函数的坐标（即 x,y）限制在[0,p)之内，此函数的曲线图参考图 4-1，可见椭圆曲线并不是椭圆，而是关于 x 轴对称的曲线。

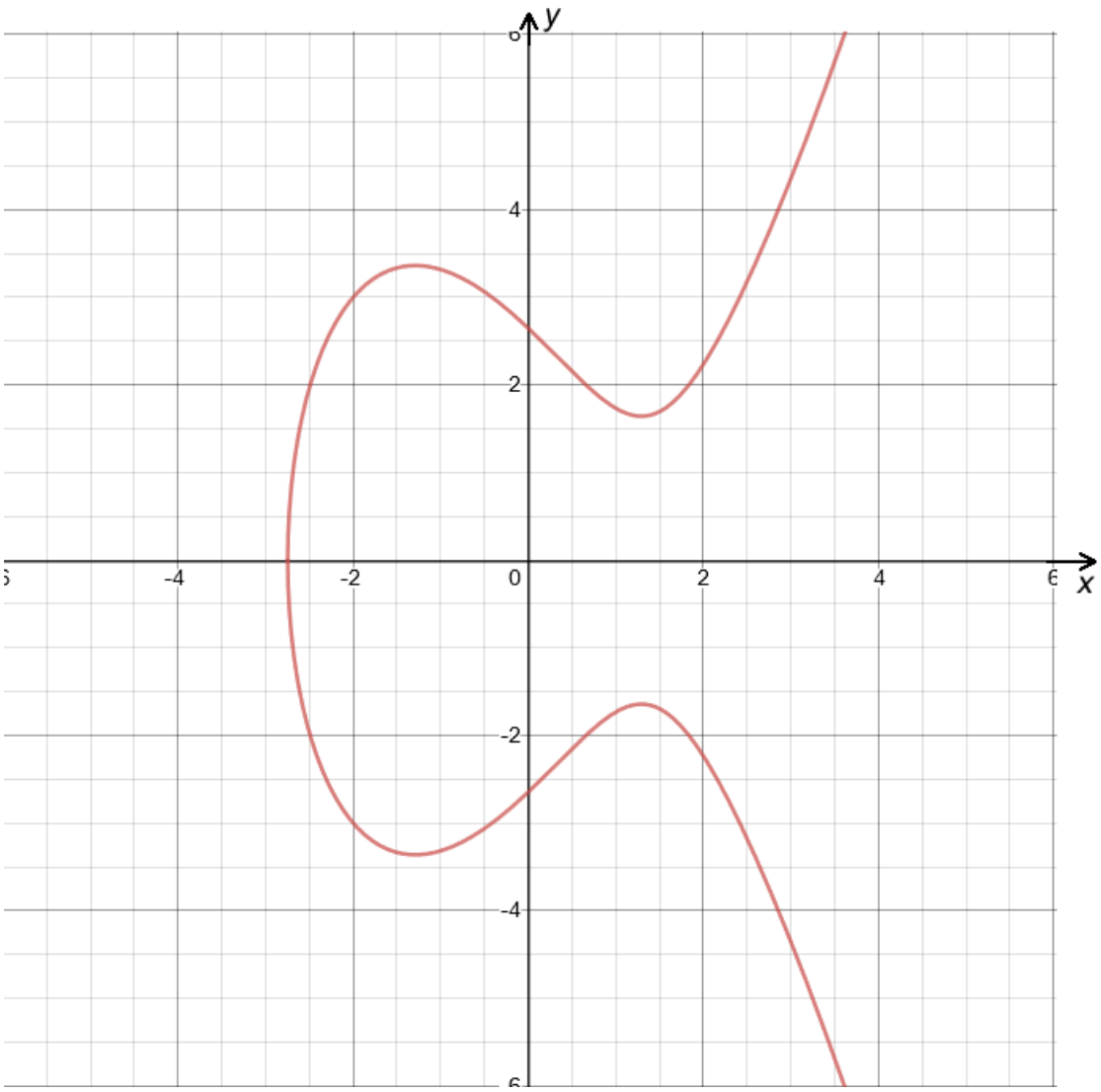


图 4-1 spec256k1 标准的椭圆曲线

从曲线上取两点，比如 A 和 B 点，作一根直线，此直线与曲线相交于另一点 C'，C'关于 x 轴的对称点为 C，如此从 A 和 B“计算”出 C 定义为椭圆曲线上的加法，表示为 A+B=C，参考图 4-2 所示。

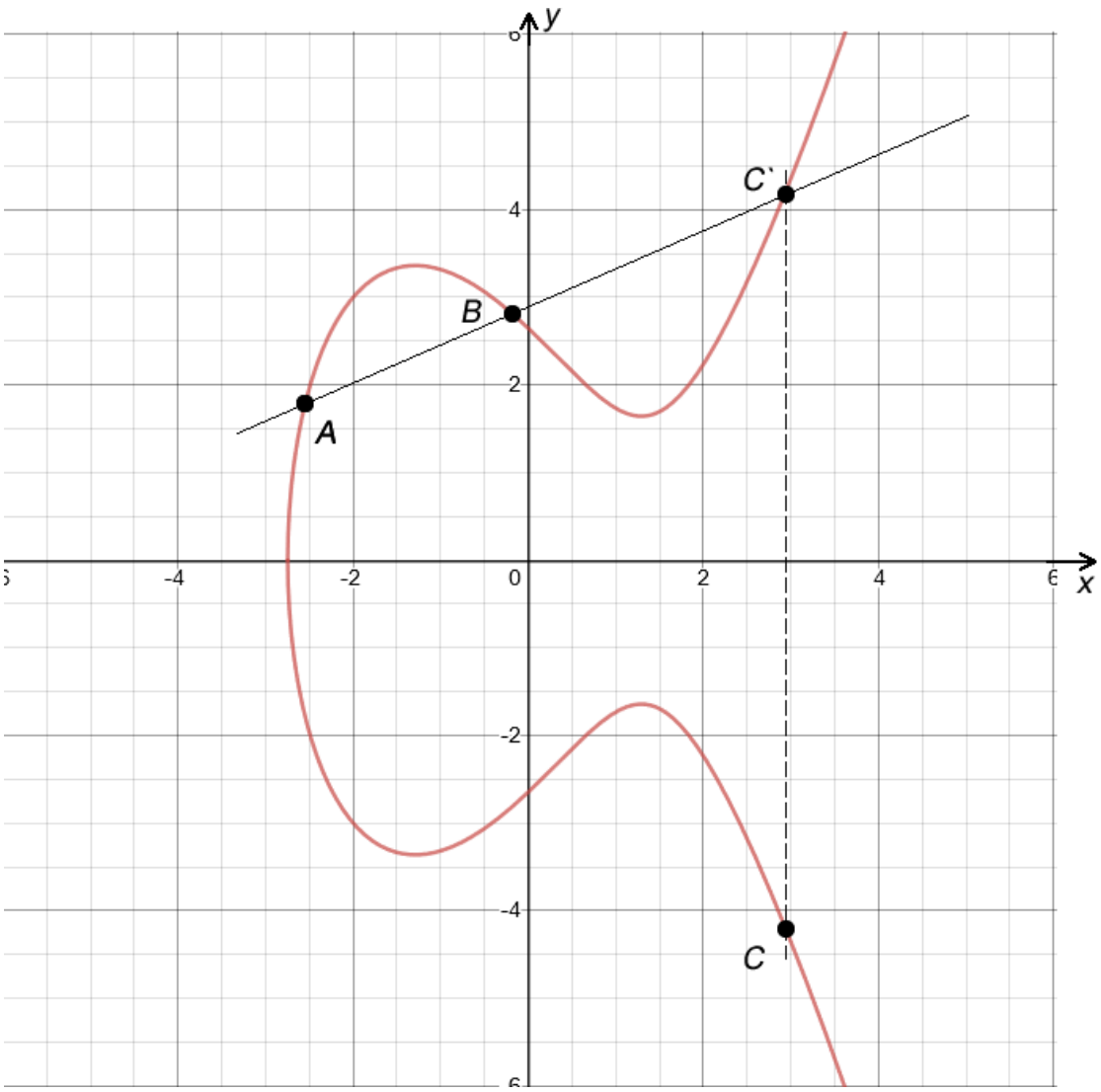


图 4-2 椭圆曲线上的加法（A+B=C）

再次 C+A=D 的示意图参考图 4-3 所示。

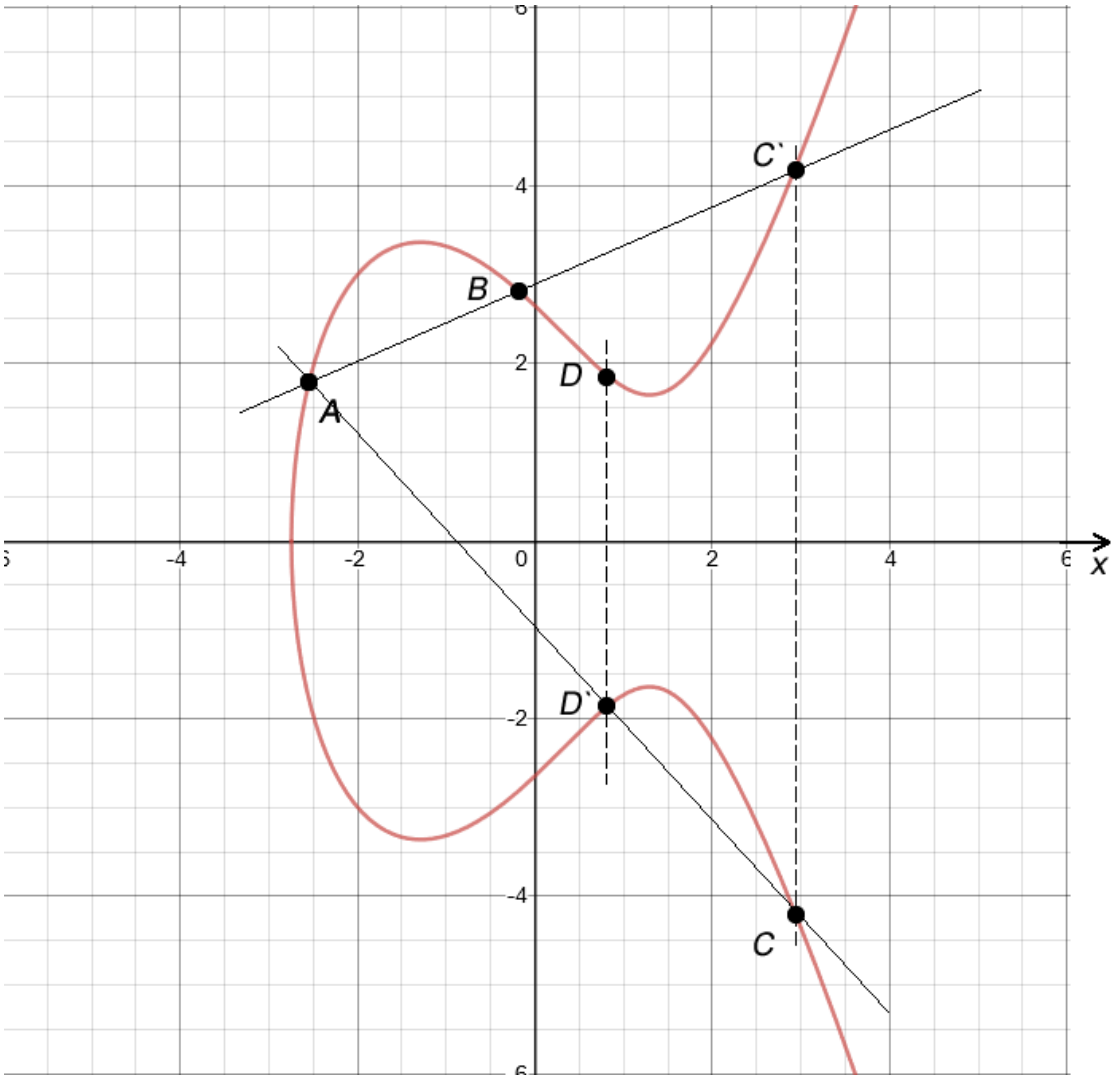


图 4-3 椭圆曲线上的加法（C+A=D）

依次类推，可以继续加下去，……。A 和 B 点称为基点，显然从基点开始经过 *n* 次加法后的点是确定的。在加密币中，规定 A 和 B 点为同一点，这时的直线就是经过 A 点的切线，因此 A+A=C、C+A=D 的示意图见 4-4 所示，A 点称为生成元，一般表示为 G。

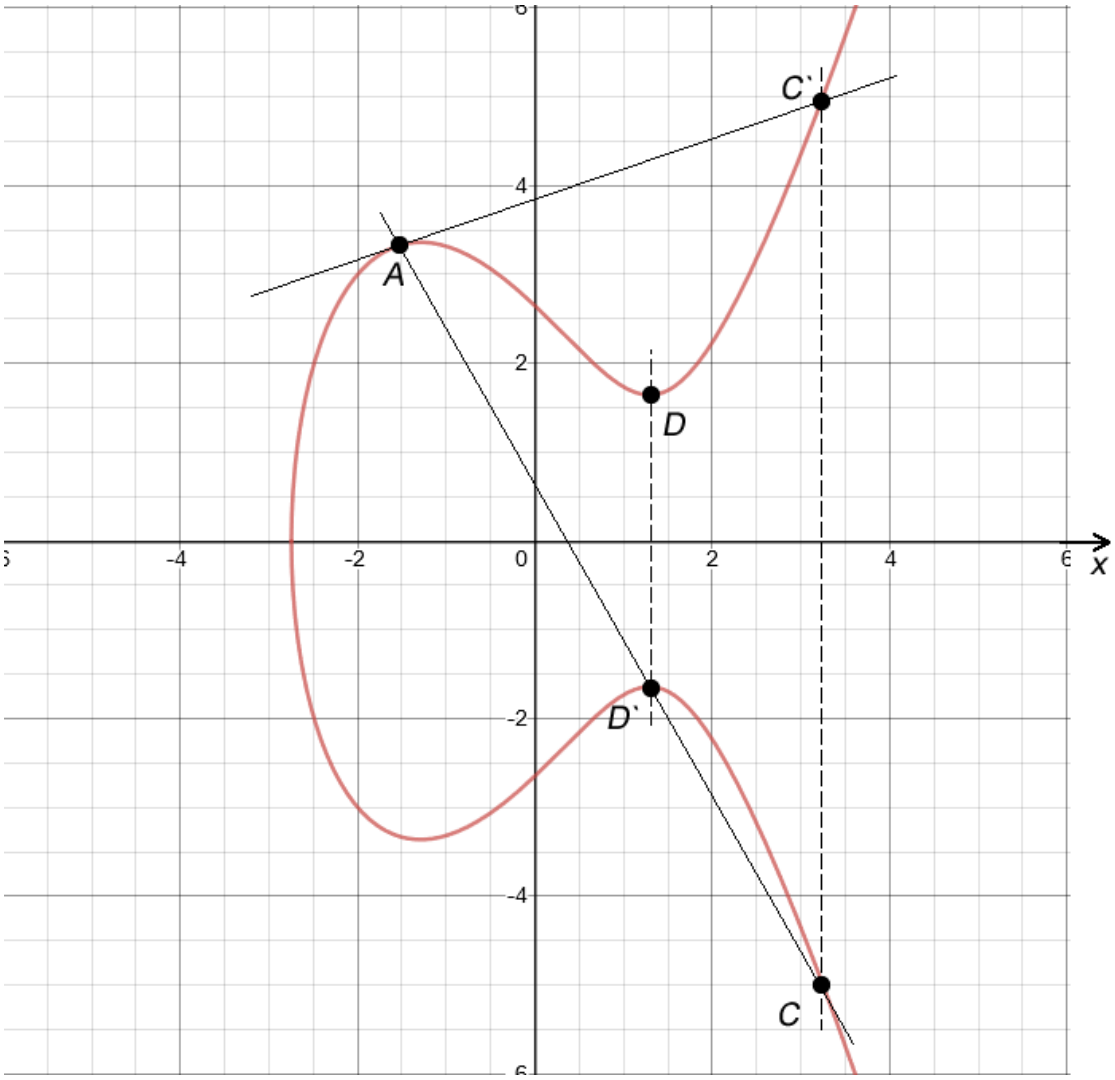


图 4-4 spec256k1 标准的椭圆曲线上的加法（A+A=C）

因此可推出：

$$\begin{cases} A + A = C \\ C + A = D \end{cases} => \ (A + A) + A = D => A + A + A = D$$

spec256k1 标准的椭圆曲线上的减法定义为：A - B = A + (-B)，其中-B 称为 B 的逆元，B 与-B 关于 x 轴对称。

spec256k1 标准的椭圆曲线上的乘法定义为：

$$n \times A = \overbrace{A + A + \cdots + A}^{n个A} = Z_n$$

其中 n 就是私钥（256 位随机二进制数），不过 n 只能是标量，Z_n 点的坐标拼接在一起就是 512 位二进制的公钥（即 xy）。众所周知，只要给出 Z_n 点的 x 坐标，就可以根据椭圆函数算出 Z_n 点的 y 坐标，所以 x 就可以作为公钥使用，这时公钥只有 256 位二进制加上一个字节的前缀（0x02 或 0x03，前者表示计算出来的 y 坐标取正数，后者表示 y 坐标为负数，因为对于一个 x，算出的 y 有正负两个值），因此只包含 x 的公钥称为压缩公钥，如：

0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798

比特币规定生成元 G 点的坐标为：

```
(55066263022277343669578718895168534326250603453777594175500187360389116729240,
 32670510020758816978083085130507043184471273380659243275938904335757337482424)
```

在计算过程中，如果坐标值大于 p，就对 p 求模，这样保证坐标值永远不会超过 p，p 称为阶。

显然知道 n（私钥）很容易求出公钥，但是从公钥几乎不可能求出私钥，目前唯一可能从公钥（假如为 p_1）破解私钥的办法就是暴力破解，即先猜一个私钥，然后求出公钥（假如为 p_2），并比较 p_1 和 p_2 是否相等，如果相等表示刚才的私钥猜测对了，但是这种暴力破解法需要消耗大量的计算资源，就算目前最快的电脑，也要耗费上百年才可能破解成功。

椭圆曲线上的点除法运算定义为 A / n = A * (1/n)，所以除法就是乘法，不过先要计算标量的倒数，这很消耗计算资源。

公钥和私钥的一个最重要应用之一就是非对称加解密，即用公钥给内容加密成密文，对方用私钥解密还原出明文。基于椭圆曲线的公私钥加解密的过程如下（假设明文为 T、私钥 k、公钥 P、椭圆曲线的基点 G，因此存在 $P=kG$）：

加密过程：

1) 将明文 T 编码到椭圆曲线上的一个点 M；

2) 产生一个随机数 r；

3) 计算椭圆曲线上的两个点：$C1 = M + r \times P$和$C2 = r \times G$；（加密用到公钥 P）

4) 把 $C1$ 和 $C2$ 传递给对方。

解密过程：

1) 计算$C1 - k \times C2$得到 M；（解密用到私钥 k）

2) 把 M 反编码到 T，即得到明文；

因为有：$C1 - k \times C2 = M + r \times P - k \times (r \times G) = M + r \times P - r \times (k \times G) = M + r \times P - r \times P = M$

公钥和私钥的另一个重要的应用就是数字签名，加密币严重依赖数字签名。

4.2 签名算法

签名与验证签名的过程有点复杂，ECDSA 签名（椭圆曲线数字签名）和验证算法大致过程是这样的（假设父亲的私钥为 k、公钥为 P、被签名文本为 M、椭圆曲线的阶为 n、椭圆曲线的基点为 G）：

ECDSA 签名的过程：

1) 计算 M 的哈希值 z；

2) 产生随机数 e，计算$e \times G = Q$，取 Q 点的 x 坐标存入 r 中；

3) 计算 $r = r \bmod n$，如果 r 等于 0 则回到 2)；

4) 计算 $s = (z + k \times r)/e \bmod n$，如果 s 等于 0 则返回 2)；（签名用到了父亲的私钥 k）

5) 把 M、z、r 和 s 传递给对方。（如果不需保证被签名文档的完整性，只需转递 r 和 s 即可）

由于每次签名使用了随机数 e，因此，即使父亲对同一个文本多次签名，每次签名的结果也会不同（即 r 和 s 不同）。

ECDSA 验证签名的过程：

1) 计算 M 的哈希值，并与 z 比较，如果不相等，那表明 M 被篡改过，停止进一步验证；

2) 计算 $\frac{z \times G}{s} + \frac{r \times P}{s} = R$，$R$ 为椭圆曲线上的一个点；（验证签名用到了父亲的公钥 P）

3) 如果 R 的 x 坐标等于 r，那么签名验证成功，否则验证失败。

数学推导过程：$R = \frac{z \times G}{s} + \frac{r \times P}{s} = \frac{z \times G + r \times P}{s} = \frac{z \times G + r \times k \times G}{s} = \frac{(z + r \times k) \times G}{s} = \frac{(z + r \times k) \times G}{\frac{z + r \times k}{e}} = \frac{(z + r \times k) \times G \times e}{z + r \times k} = e \times G = Q$，如果 R 和 Q 是椭圆曲线上的同一个点，只要比较横坐标是否相等即可。关于椭圆曲线上加法和乘法，请参考 § 1.2 小节中的相关内容。

由于验证签名涉及标量乘法和除法，众所周知，乘除计算会消耗电脑大量的计算资源，所以 ECDSA 验证签名效率不高，另一个缺点是 ECDSA 签名占用较多的区块链空间。为此 2021 年比特币进行了 Taproot 软分叉升级，其中的一大改变是引入了施诺尔（Schnorr）签名算法。施诺尔签名算法专利于 2008 年到期，而同年中本聪发布了比特币白皮书，在 2008 年之前的开发阶段，施诺尔签名算法受到专利保护，所以当时中本聪选择了开源的 ECDSA 签名算法。

施诺尔签名算法的签名和验证签名的大致过程如下（假设父亲的私钥为 k、公钥为 P、被签名文本为 m、椭圆曲线的生成元为 G）：

施诺尔签名生成过程：

1) 生成一个随机值 e；

2) 计算 $R = e \times G$；

3) 计算哈希值 $z = H(R, m)$，其中 H 是哈希函数；

4) 计算 $s = e + z \times k$；

5) (R, s) 就是签名。

施诺尔签名的验证过程：

1) 计算哈希值 z = H(R, m)；

2) 检查 $s \times G$ 是否等于 $R + z \times P$；（$s \times G$ 和 $R + z \times P$都是椭圆曲线上的点，因此只需比较 x 坐标即可）

3) 如果相等，则签名有效，否则签名无效。

施诺尔签名与 ECDSA 签名相比具有两大优势：一是更小更快，二是施诺尔签名是线性的，即两个签名可以相加得到聚合密钥的有效签名，从而实现了多签功能，这是一个非常令人激动的特征！对于加密币交易的签名，m 就是交易中某些特定部分的哈希值。

基于 Schnorr 多签协议（MuSig2）的工作原理如下：

MuSig2 的核心是将多个签名者的公钥聚合为一个单一公钥，并通过协作生成一个单一的 Schnorr 签名。它分为两个主要阶段：密钥聚合和签名生成。

1. 公钥聚合

输入：多个签名者的公钥集合 $P_1,P_2,...,P_n$，假设从小到大排好了序，例如比特币中的 secp256k1 公钥。

步骤：

(1) 计算一个公共的“绑定值” $L=H(P_1 \| P_2 \| ... \| P_n)$，其中 H 是哈希函数（如 SHA256），‖ 表示拼接。
(2) 为每个公钥 Pi 计算挑战因子 $a_i=H(L \| P_i)$。
(3) 聚合公钥 $X=a_1 \cdot P_1+a_2 \cdot P_2+...+a_n \cdot P_n$，其中· 是椭圆曲线上的标量乘法，+ 是点加法。

输出：一个单一的聚合公钥 X。

这种聚合方式确保即使某些签名者恶意选择公钥，也无法推导出其他参与者的私钥。

2．签名生成阶段

MuSig2 的签名过程需要两轮交互，相比 MuSig1 的三轮更高效。

输入：

签名者的私钥 $x_1,x_2,...,x_n$（分别对应公钥 $P_1,P_2,...,P_n$）。

待签名的消息 m（如比特币交易的哈希）。

步骤：

第一轮：Nonce 生成和交换。
(1) 每个签名者 i 生成两个 32 各比特的随机数（nonce）$r_{i,1}$ 和 $r_{i,2}$（称为“双 nonce”设计）。
(2) 计算对应的公钥点 $R_{i,1}=r_{i,1} \cdot G$ 和 $R_{i,2}=r_{i,2} \cdot G$，其中 G 是 secp256k1 的生成元。
(3) 如果 $R_{i,1}[0]$等于 3，则 $r_{i,1}=n- r_{i,1}$、$R_{i,1}[0]=2$；如果 $R_{i,2}[0]$等于 3，则 $r_{i,2}=n- r_{i,2}$、$R_{i,2}[0]=2$，其中 n 为椭圆曲线的阶。
(3) 将 $R_{i,1},R_{i,2}$ 广播给其他签名者。
(4) 所有签名者聚合这些 nonce：$R_1=\sum R_{i,1}$，$R_2=\sum R_{i,2}$。

第二轮：部分签名和聚合
(1) 计算一个绑定因子 $b=H(R_1 \| R_2 \| m)$。
(2) 计算最终的聚合 nonce $R=R_1+b \cdot R_2$。
(3) 每个签名者 i 计算部分签名：
e=H(x(R) ‖ X ‖ m)（挑战值）。（x(R)取公钥的 x 坐标，共 32 个字节）
$s_i=r_{i,1}+b \cdot r_{i,2}+e \cdot a_i \cdot x_i$（其中 a_i 是公钥聚合时的挑战因子）。
(4) 将 s_i 广播给其他签名者。
(5) 聚合签名 $s=\sum s_i$。

输出：一个 Schnorr 签名 <x(R), s> ，长度为 64 字节（x(R)和 s 各为 32 字节）。

3. 验证

验证者使用聚合公钥 X 和签名 <R, s>：

(1) 计算 e=H(R ‖ X ‖ m)。

(2) 验证 s· G=[02]||R+e· X。(R 为 32 个字节的 X 坐标，需要添加 02 前缀转化为压缩公钥)

如果等式成立，则签名有效。

下面我们举一个 MulSig2 协议的例子：由三人签名一条信息，其他人可以验证。假设三人的私钥分别是 'cUfGmsyHgFiueRjMtxAb3XDeFvdmJuqpqMPUnjYcqSzesk27ye8f' 、'cQvuoR1w4JcaLmFZDde2JbfLqb38ETHo8Td3vQGAmEDw7vN2XmcJ' 和 'cU5FNwCsgVVGz7HxtipJyxNhxhTqvzow4PbmkgSvSYTubji41zua'，格式都是 WIF，被签名的内容为“Hello MulSig2!”。

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');
const { ECPairFactory } = require('ecpair');
bitcoin.initEccLib(ecc);

const ECPair = ECPairFactory(ecc);

const network = bitcoin.networks.testnet;
const toXOnly = pubKey => (pubKey.length === 32 ? pubKey : pubKey.slice(1, 33));

// 椭圆曲线的阶:
const secp256k1_order = BigInt('0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141');

// 提供的 WIF 私钥(在实际的生活当中，每个人只知道自己的私钥和所有人的公钥，不会知道其他人的私钥的):
const wifKey = 'cUfGmsyHgFiueRjMtxAb3XDeFvdmJuqpqMPUnjYcqSzesk27ye8f';                                  //我的私钥
const wifKey1 = 'cQvuoR1w4JcaLmFZDde2JbfLqb38ETHo8Td3vQGAmEDw7vN2XmcJ';                                 //你的私钥
const wifKey2 = 'cU5FNwCsgVVGz7HxtipJyxNhxhTqvzow4PbmkgSvSYTubji41zua';                                 //他的私钥

const message = "Hello MulSig2!";                                                                     //被签名的内容

// 从 WIF 解析私钥和公钥对，并按公钥由小到大排序:
//我的公私钥:
let keyPair = ECPair.fromWIF(wifKey, network);
const privateKey = keyPair.privateKey;                                                                  //原始格式的私钥
const publicKey = keyPair.publicKey;                                                                  //把这个公钥广播出去

//你的公私钥:
keyPair = ECPair.fromWIF(wifKey1, network);
const privateKey1 = keyPair.privateKey;                                                                 //原始格式的私钥
const publicKey1 = keyPair.publicKey;                                                                 //把这个公钥广播出去

//他的公私钥:
keyPair = ECPair.fromWIF(wifKey2, network);
const privateKey2 = keyPair.privateKey;                                                                 //原始格式的私钥
const publicKey2 = keyPair.publicKey;                                                                 //把这个公钥广播出去

//每个人收齐了所有人的公钥后整理成一个排序好了的数组:
const publicKeys = [publicKey, publicKey1, publicKey2].sort((a, b) => Buffer.compare(a, b));
//规定公钥从小到大排序能保证参与签名的所有人计算的聚合公钥相等，当然也可以规定从大到小排序，只要大家遵循相同的排序规则即可。

// 1. 聚合公钥（MuSig2）——每个人算出来的聚合公钥都相等:
// 参数 publicKeys 包含所有人的公钥，且从小到大排好了序
function aggregatePublicKeys(publicKeys) {
   const keyCombo = Buffer.concat(publicKeys);
   const L = bitcoin.crypto.sha256(keyCombo);
   let aggregatedPubKey = null;
   for (let pubkey of publicKeys) {
      const challenge = bitcoin.crypto.sha256(Buffer.concat([L, pubkey]));
      const scaledPubKey = ecc.pointMultiply(pubkey, challenge, true);
      aggregatedPubKey = aggregatedPubKey ? ecc.pointAdd(aggregatedPubKey, scaledPubKey, true) :
scaledPubKey;
   }

   return aggregatedPubKey;
}

const aggregatedPubKey = aggregatePublicKeys(publicKeys);
```

```
console.log('聚合公钥(hex):', Buffer.from(aggregatedPubKey).toString('hex'));
聚合公钥(hex): 029f1e992ea71d35959b7e93fceedc471d5ff92fc6a39350bec06349200679d088

// 2. 生成和聚合 Nonce
/*
    如果私钥 k 的公钥为 P(x,y)，那么 secp256k1_order-k 的公钥就是 P(y,-y)，且 y 和-y 一定是一个是奇数一个是偶数。
    其中 secp256k1_order 为椭圆曲线的阶。
    Schnorr 签名要求 Y 坐标为偶数的那个公钥。
*/
function generateNoncePair() {
    let r1 = new Uint8Array(32);
    crypto.getRandomValues(r1);
    let r2 = new Uint8Array(32);
    crypto.getRandomValues(r2);
    let R1 = ecc.pointFromScalar(r1);
    if (R1[0] === 3) {                                                          //公钥的 Y 坐标为奇数
        r1 = ecc.privateNegate(r1);                                           //计算 secp256k1_order-r1
        R1[0] = 2;                                                              //使用 P(x,-y)这个公钥
    }
    let R2 = ecc.pointFromScalar(r2);
    if (R2[0] === 3) {
        r2 = ecc.privateNegate(r2);
        R2[0] = 2;
    }
    return { r1, R1, r2, R2 };
}

//实操时每个人只生成一个 NoncePair，并广播 NoncePair.R1 和 NoncePair.R2。
const noncePair = generateNoncePair();                                                  //我
const noncePair1 = generateNoncePair();                                                 //你
const noncePair2 = generateNoncePair();                                                 //他

//每个人把收到的 R1 和 R2 聚合在一起:
const noncePairs = [
    {
        R1: noncePair.R1,
        R2: noncePair.R2
    },
    {
        R1: noncePair1.R1,
        R2: noncePair1.R2
    },
    {
        R1: noncePair2.R1,
        R2: noncePair2.R2
    }
];
const aggregatedNonce = noncePairs.reduce(
    (acc, nonce) => ({
        R1: acc.R1 ? ecc.pointAdd(acc.R1, nonce.R1, true) : nonce.R1,
        R2: acc.R2 ? ecc.pointAdd(acc.R2, nonce.R2, true) : nonce.R2
    }),
    {}
);

// 3. MuSig2 签名
const messageBuffer = Buffer.from(message, 'utf8');

// 使用 bitcoinjs-lib 的 taggedHash
const msgHash = bitcoin.crypto.taggedHash('BIP0340/challenge', messageBuffer);
console.log('被签名内容的哈希(hex):', msgHash.toString('hex'));
被签名内容的哈希(hex): dceee5cc9aca12b3408c52bb1fdd24926603300cf2b418d0d5efafa9253fd309

// 手动实现 privateMul，参数 a 和 b 是两个私钥，且都是 Buffer 类型:
function privateMul(a, b) {
    const bnA = BigInt('0x' + a.toString('hex'));
    const bnB = BigInt('0x' + b.toString('hex'));
    const result = (bnA * bnB) % secp256k1_order;
    return Buffer.from(result.toString(16).padStart(64, '0'), 'hex');
}
```

```
/*
    MulSig2 协议的签名函数。
    参数 privKey - 签名私钥。
    参数 publicKeys - 包含全部公钥的数组，且已经从小到大排好了序。
    参数 noncePair - 我的 NoncePair={r1:…, R2:…, r2:…, R2:…}.
    参数 aggregateNonce - 聚合 Nonce={R1:…, R2:…}。
    参数 msgHash - 被签名内容的哈希。
*/
function signMuSig2(privKey, publicKeys, noncePair, aggregatedNonce, msgHash) {
    const L = bitcoin.crypto.sha256(Buffer.concat(publicKeys));
    const pubKey = ecc.pointFromScalar(privKey, true);
    const challenge = bitcoin.crypto.sha256(Buffer.concat([L, pubKey]));
    const b = bitcoin.crypto.sha256(Buffer.concat([aggregatedNonce.R1, aggregatedNonce.R2, msgHash]));
    const R = ecc.pointAdd(aggregatedNonce.R1, ecc.pointMultiply(aggregatedNonce.R2, b, true), true);
    const Rx = toXOnly(R);
    const e = bitcoin.crypto.sha256(Buffer.concat([Rx, aggregatedPubKey, msgHash]));

    const br2 = privateMul(b, Buffer.from(noncePair.r2));
    const ea = privateMul(e, challenge);
    const eax = privateMul(ea, privKey);
    const s = ecc.privateAdd(noncePair.r1, ecc.privateAdd(br2, eax));

    return { R: Rx, s };
}

// 每个人生成自己的部分签名，并广播出去:
const partialSignature = signMuSig2(privateKey, publicKeys, noncePair, aggregatedNonce, msgHash);  //我的
const partialSignature1 = signMuSig2(privateKey1, publicKeys, noncePair1, aggregatedNonce, msgHash);//你的
const partialSignature2 = signMuSig2(privateKey2, publicKeys, noncePair2, aggregatedNonce, msgHash);//他的

// 每个人收齐所有人的部分签名后聚合在一起:
const partialSignatures = [partialSignature, partialSignature1, partialSignature2];
const aggregatedSignature = {
    R: partialSignature.R,
    s: partialSignatures.reduce((acc, sig) => acc ? ecc.privateAdd(acc, sig.s) : sig.s, null)
};

const finalSignature = Buffer.concat([aggregatedSignature.R, aggregatedSignature.s]);
console.log(`聚合签名(hex): ${finalSignature.toString('hex')}\n签名长度: ${finalSignature.length}字节`);
聚合签名(hex):
dbab5938ab16e80b7f24e6be41ad55d6d3e0c742766d8cfaefff5d1ff3592c3d4aa277f6f1eb55511c47d90ff7598a4482d280a06e
3431e2f7090db2a23c863c
签名长度: 64 字节

// 4. 验证签名
function verifyMuSig2(aggregatedPubKey, msgHash, signature) {
    const Rx = signature.slice(0, 32);
    const s = signature.slice(32);
    const e = bitcoin.crypto.sha256(Buffer.concat([Rx, aggregatedPubKey, msgHash]));

    const left = ecc.pointFromScalar(s, true);
    const R = new Uint8Array(Buffer.concat([Buffer.from([0x02]),Rx]));
    const right = ecc.pointAdd(R, ecc.pointMultiply(aggregatedPubKey, e, true), true);
    const isValid = Buffer.compare(left, right) === 0;
    return isValid;
}

const isValid = verifyMuSig2(aggregatedPubKey, msgHash, finalSignature);
console.log('签名有效吗? :', isValid);
签名有效吗? : true
```

4.3 地址进阶

4.3.1 P2PKH 地址进阶

从公钥推导地址的过程请参考图 4-5 所示。

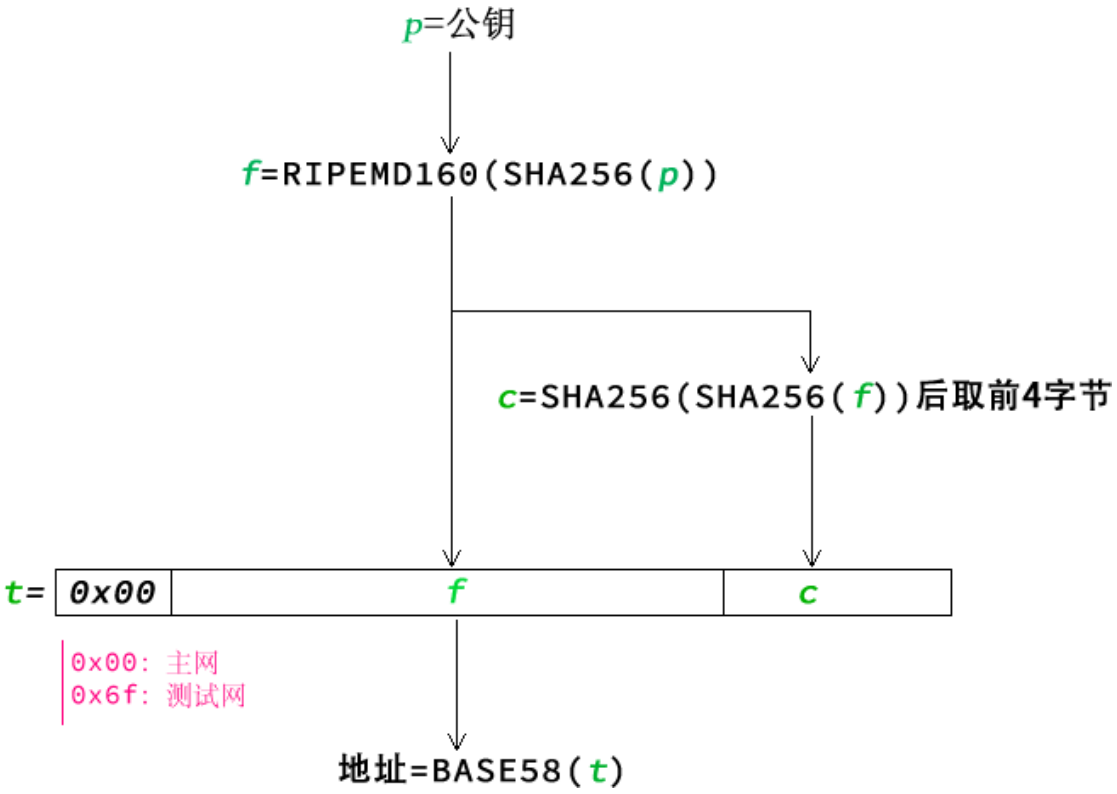


图 4-5 P2PKH 地址的推导过程

实战：

```
const bitcoin = require('bitcoinjs-lib');
const bs58check = require('bs58check');

let publicKey = '0299dc20b67d59d7c9bfe7500994a5399948933675cd82a54f84dcd14a0210e98c';            //压缩公钥
let p = Uint8Array.from(pubKey);
let f = bitcoin.crypto.ripemd160(bitcoin.crypto.sha256(p));
let c = bitcoin.crypto.sha256(bitcoin.crypto.sha256(f)).slice(0,4);
let t = Buffer.concat([Buffer.from('00', 'hex'), f, c]);
let address = bs58check.default.encode(t);
console.log(address);
1rp9mWLQLABvPj4iJHkcjMMwCjyCP3RdAYCnaor                                                    //最终地址
```

4.3.2 P2SH 地址进阶

赎回脚本的哈希值作为钱包地址，对于 t-of-k 多签，赎回脚本的格式就是：

```
t <pubKey1> <pubKey2> ··· <pubKeyk> K OP_CHECKMULTISIG
```

注意：赎回脚本中的公钥必须按字典序从小到大排序。

其地址的产生过程参见图 4-6 所示。

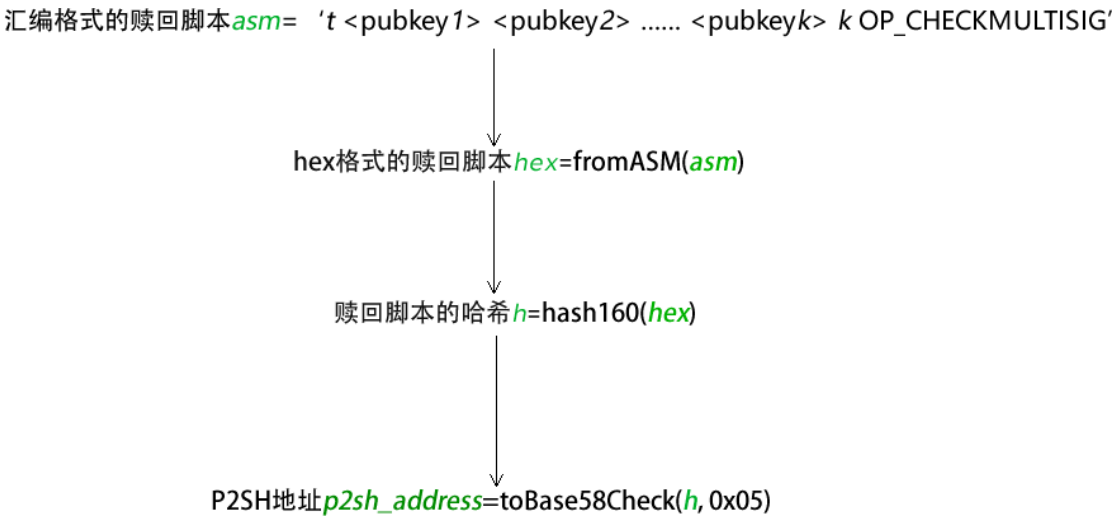


图 4-5 t-of-k 多签 P2SH 地址生成过程

t-of-k 多签的意思是花钱的时候，提供的签名个数至少 t 个，最多 k 个，且规定了 1≤t≤k≤15，也就是说 P2SH 支持最多 15 个签名。这特别适合企业开支——需要总经理、分管财务的副总经理、财务主任和经办员一起签字才能支出一笔大额款项。

对于 t-of-k 多签，默认的赎回脚本为“*t* <*公钥*$_1$> <*公钥*$_2$> ··· <*公钥*$_k$> *k OP_CHECKMULTISIG*”，当然你也可以自定义赎回脚本，比如包含时间锁、哈希锁等复杂的支付条件，但是一定要记得花费时需要用到私钥才能解锁，否则资金不安全。

下面做个实验，手工生成 2-of-3 的多签赎回脚本、输出脚本和 P2SH 地址：

```
const bitcoin = require('bitcoinjs-lib');
const publicKeys = [
    '03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6',                    //公钥 1
    '035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7',                    //公钥 2
    '03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4'                     //公钥 3
].sort((a, b) => Buffer.from(a, 'hex').compare(Buffer.from(b, 'hex')));                  //必须从小到大排序

const redeem_ASM = `OP_2 ${publicKeys.join(' ')} OP_3 OP_CHECKMULTISIG`; //赎回脚本（汇编）
console.log(redeem_ASM);
OP_2 035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7
03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4
03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6 OP_3 OP_CHECKMULTISIG

const redeem = bitcoin.script.fromASM(redeem_ASM);                                       //赎回脚本（Buffer）
console.log(redeem.toString('hex'));
5221035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f72103b87af9c1261086bcaf20134ea90d3170
895f2d4cd4ea9e2d20f52d2a67f393a42103f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f653ae

const redeem_hash = bitcoin.crypto.hash160(redeem);                                      //赎回脚本的哈希值（Buffer）
```

```
console.log(redeem_hash.toString('hex'));
a37c569d0430a09b7dd6fa4c8dce4be93d989ac2

const p2sh_address = bitcoin.address.toBase58Check(redeem_hash, 0x05);                    //p2sh 地址
console.log(p2sh_address);
3GbT1zRssitxEqv5BqArn9W9DcsVmkXNbM

console.log(output_ASM);
const output_ASM = `OP_HASH160 ${redeem_hash.toString('hex')} OP_EQUAL`;                  //输出脚本（汇编）
console.log(output_ASM);
OP_HASH160 a37c569d0430a09b7dd6fa4c8dce4be93d989ac2 OP_EQUAL

const output = bitcoin.script.fromASM(output_ASM);                                        //输出脚本（Buffer）
console.log(output.toString('hex'));
a914a37c569d0430a09b7dd6fa4c8dce4be93d989ac287
```

到此，我们得到了赎回脚本：

```
//Hex 格式：
5221035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f72103b87af9c1261086bcaf20134ea90d3170
895f2d4cd4ea9e2d20f52d2a67f393a42103f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f653ae

//汇编格式：
OP_2 035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7
03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4
03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6 OP_3 OP_CHECKMULTISIG
```

赎回脚本的哈希值（hash160(redeem) = ripemd160(sha256(redeem))）：

```
a37c569d0430a09b7dd6fa4c8dce4be93d989ac2
```

P2SH 地址：

```
3GbT1zRssitxEqv5BqArn9W9DcsVmkXNbM
```

以及输出脚本：

```
a914a37c569d0430a09b7dd6fa4c8dce4be93d989ac287                                            //Hex 格式
OP_HASH160 a37c569d0430a09b7dd6fa4c8dce4be93d989ac2 OP_EQUAL                              //汇编格式
```

再举一个例子：半年后三个子女中两人签名即可花钱。

```
const bitcoin = require('bitcoinjs-lib');

const HALF_YEAR_BLOCKS = 26208;              //即 0x6660。半年 182 天，182*24*60/10=26208，每 10 分钟产生一个区块
const publicKeys = [                                                                      //三个子女的公钥
    Buffer.from('03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6', 'hex'),
    Buffer.from('035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7', 'hex'),
    Buffer.from('03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4', 'hex')
].sort((a, b) => a.compare(b));                                                           //必须从小到大排序

const mulSign = bitcoin.script.compile([
    bitcoin.script.number.encode(2),                                                      // 需要 2 个签名
    ...publicKeys,                                                                        // 三个公钥
    bitcoin.script.number.encode(3),                                                      // 总共 3 个公钥
    bitcoin.opcodes.OP_CHECKMULTISIG
]);
console.log(mulSign.toString('hex'));
5221035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f72103b87af9c1261086bcaf20134ea90d3170
895f2d4cd4ea9e2d20f52d2a67f393a42103f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f653ae

const redeem_script = bitcoin.script.compile([
    bitcoin.script.number.encode(HALF_YEAR_BLOCKS),
    bitcoin.opcodes.OP_CHECKSEQUENCEVERIFY,
    bitcoin.opcodes.OP_DROP,
    ...bitcoin.script.decompile(mulSign)
]);
console.log(redeem_script.toString('hex'));
```

```
026066b2755221035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f72103b87af9c1261086bcaf2013
4ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a42103f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f3
93f653ae

console.log(bitcoin.script.toASM(redeem_script));                                    //赎回脚本（汇编语言）
6066 OP_CHECKSEQUENCEVERIFY OP_DROP OP_2
035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7
03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4
03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6 OP_3 OP_CHECKMULTISIG

const redeem_hash = bitcoin.crypto.hash160(redeem_script);                    //赎回脚本的哈希值（Buffer）
console.log(redeem_hash.toString('hex'));
822dfea19c7d99a71907f629a19a631ce84f75d6

const p2sh_address = bitcoin.address.toBase58Check(redeem_hash, 0x05);
console.log(p2sh_address);
3DZLsFEfevh6G56h9mu6TkBkiRNtbWRi7z ----------------------------------------------------------------------//p2sh 地址

const output_ASM = `OP_HASH160 ${redeem_hash.toString('hex')} OP_EQUAL`;              //输出脚本（汇编）
console.log(output_ASM);
OP_HASH160 822dfea19c7d99a71907f629a19a631ce84f75d6 OP_EQUAL

const output = bitcoin.script.fromASM(output_ASM);                                   //输出脚本（Buffer）
console.log(output.toString('hex'));
a914822dfea19c7d99a71907f629a19a631ce84f75d687
```

4.3.3 P2WPKH 地址进阶

P2WPKH 采用 Bech32 编码方式。Bech32 是 BIP 0173 和 BIP 0350 规范引入的隔离见证地址格式——以“bc1”开头的地址，在节约区块尺寸方面更具效率。Bech32 地址产生的过程如下：

(1) 获取压缩公钥（x 坐标加上 0x02 或 0x03 前缀）p。如十六进制格式的压缩公钥：p=0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798；

(2) 对 p 做 SHA-256 哈希运算得到 h_1，显然 h_1 为 256 位二进制的一个数。如：h_1=0f715baf5d4c2ed329785cef29e562f73488c8a2bb9dbc5700b361d54b9b0554；

(3) 对 h_1 做 RIPEMD-160 哈希运算得到 h_2。，h_2 为 160 位二进制的一个数。如十六进制格式为：h_2=751e76e8199196d454941c45d1b3a323f1433bd6，对应的二进制数就是：h_2=*0111010100011110011101101110100000011001100100011001011011010100010101001001010000011100010001 0111010001101100111010001100100011111100010100001100111011110101 10*；

(4) 对 h_2 从最低位开始每隔 5 个二进制拆分成一段，每段前面补三个 0 转换成一个字节，得到 h_3。例如：h_2=*0111010100011110011101101110100000011001100100011001011011010100010101001001010000011100010001 0111010001101100111010001100100011111100010100001100111011110101 10* → *01110 10100 01111 00111 01101 11010 00000 11001 10010 00110 01011 01101 01000 10101 00100 10100 00011 10001 00010 11101 00011 01100 11101 00011 00100 01111 11000 10100 00110 01110 11110 10110* →000*01110* 000*10100* 000*01111* 000*00111* 000*01101* 000*11010* 000*00000* 000*11001* 000*10010* 000*00110* 000*01011* 000*01101* 000*01000* 000*10101* 000*00100* 000*10100* 000*00011* 000*10001* 000*00010* 000*11101* 000*00011* 000*01100* 000*11101* 000*00011* 000*00100* 000*01111* 000*11000* 000*10100* 000*00110* 000*01110* 000*11110* 000*10110* →000*01110*000*10100*000*01111*000*00111*000*01101*000*10100*000*00000*000*11001*000*10010*000*00110*000*01011*000*01101*000*01000*000*10101*000*00100*000*10100*000*00011*000*10001*000*00010*000*11101*000*00011*000*01100*000*11101*000*00011*000*00100*000*01111*000*11000*000*10100*000*00110*000*01110*000*11110*000*10110=h_3，表示为十六进制就是

h_3=*0e140f070d1a001912060b0d081504140311021d030c1d03040f1814060e1e16*；

(5) 在 h_3 的前面加上一个字节的见证版本（目前固定为 0）得到 h_4。如：h_4=00*0e140f070d1a001912060b0d081504140311021d030c1d03040f1814060e1e16*；

(6) 计算 h_4 和 H.R.P (bc-MainNet、tb-TestNet)的校验和，得到 c。如：*c=0c0709110b15*；

(7) 把 c 附加到 h_4 的末尾得到 h_5。如：h_5=000e140f070d1a001912060b0d081504140311021d030c1d03040f1814060e1e160c0709110b15；

(8) h_5 的每个字节转换为 Bech32 字符，得到 h_6。如：h_6=*qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4*。十进制数 0~31 对应的 Bech32 字符参见表 4-1。

表 4-1 Bech32 字符编码

	0	1	2	3	4	5	6	7
+0	q	p	z	r	y	9	x	8
+8	g	f	2	t	v	d	w	0
+16	s	3	j	n	5	4	k	h
+24	c	e	6	m	u	a	7	l

h_5 的每个字节用十进制表示不会大于 31，因为这些字节是从 h_2 分割 5 位二进制再在高位补 000 得到的。结合表 4-1，由 h_5 转换位 Bech32 编码的过程参见图 4-7 所示，比如末尾一个字节十进制数是 21，它等于 5+16，因此对应表中的（+16, 5）元素，即 4，所以 4 就是数字 21 的 Bech32 码。

```
十六进制--> 00 0e 14 0f 07 0d 1a 00 19 12 06 0b 0d 08 15 04 14 03 11 02 1d 03 0c 1d 03 04 0f 18 14 06 0e 1e 16 0c 07 09 11 0b 15
  十进制--> 0  14 20 15 7  13 26 0  25 18 6  11 13 8  21 4  20 3  17 2  29 3  12 29 3  4  15 24 20 6  14 30 22 12 7  9  17 11 21
Bech32字符--> q  w  5  0  8  d  e  q  e  j  x  t  d  g  4  y  5  r  3  z  a  r  v  a  r  y  0  c  5  x  w  7  k  v  8  f  3  t  4
Bech32编码--> qw508deqejxtdg4y5r3zarvary0c5xw7kv8f3t4
```

图 4-7 Bech32 编码过程

(9) 最后给 h_6 加上前缀 bc（比特币主网）或者 tb（比特币测试网）和分隔符 1 得到最终的地址 a。如：a=bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4；

4.3.4 P2WSH 地址进阶

输出脚本格式为“OP_0 <对赎回脚本进行 sha256 哈希运算（32 个字节）>”，对输出脚本进行 Bech32 编码即得到地址，可见地址和输出脚本很容易相互转换。看下面的代码：

```
const bitcoin = require('bitcoinjs-lib');

const HALF_YEAR_BLOCKS = 26208;                  //即 0x6660。半年 182 天，182*24*60/10=26208，每 10 分钟产生一个区块
const publicKeys = [                                                                    //三个子女的公钥
    Buffer.from('03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6', 'hex'),
    Buffer.from('035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7', 'hex'),
    Buffer.from('03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4', 'hex')
```

```
].sort((a, b) => a.compare(b));                                                    //必须从小到大排序

const mulSign = bitcoin.script.compile([
    bitcoin.script.number.encode(2),                                               // 需要 2 个签名
    ...publicKeys,                                                                 // 三个公钥
    bitcoin.script.number.encode(3),                                               // 总共 3 个公钥
    bitcoin.opcodes.OP_CHECKMULTISIG
]);
console.log(mulSign.toString('hex'));
5221035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f72103b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a42103f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f653ae

const redeem_script = bitcoin.script.compile([
    bitcoin.script.number.encode(HALF_YEAR_BLOCKS),
    bitcoin.opcodes.OP_CHECKSEQUENCEVERIFY,
    bitcoin.opcodes.OP_DROP,
    ...bitcoin.script.decompile(mulSign)
]);
console.log(redeem_script.toString('hex'));
026066b2755221035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f72103b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a42103f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f653ae

console.log(bitcoin.script.toASM(redeem_script));                                  //赎回脚本（汇编语言）
6066 OP_CHECKSEQUENCEVERIFY OP_DROP OP_2
035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7
03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4
03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6 OP_3 OP_CHECKMULTISIG

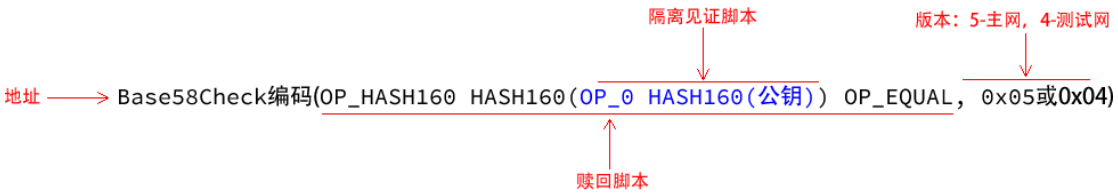
const redeem_hash = bitcoin.crypto.sha256(redeem_script);
const output = bitcoin.script.fromASM(`OP_0 ${redeem_hash.toString('hex')}`);      //输出脚本
console.log(output.toString('hex'));
0020388744432f5ec48170bd2a612196b3d25adb741e58033892788a13e9504cf31e

const p2wsh_address = bitcoin.address.toBech32(redeem_hash, 0, 'bc');              //p2wsh 地址
console.log(p2wsh_address);
bc1q8zr5gse0tmzgzu9a9fsjr94n6fddkaq7tqpn3ync3gf7j5zv7v0qvykg5c

//另一种产生输出脚本和地址的方法:
const p2wsh = bitcoin.payments.p2wsh({
    redeem: { output: redeem_script },
    network: bitcoin.networks.bitcoin
});
console.log('P2WSH 输出脚本:', p2wsh.output.toString('hex'));
P2WSH 输出脚本: 0020388744432f5ec48170bd2a612196b3d25adb741e58033892788a13e9504cf31e
console.log('Bech32 地址:', p2wsh.address);
Bech32 地址: bc1q8zr5gse0tmzgzu9a9fsjr94n6fddkaq7tqpn3ync3gf7j5zv7v0qvykg5c
```

4.3.5 P2SH-P2WPKH 地址进阶

即支付到隔离见证脚本哈希，而 P2SH 就是支付到赎回脚本哈希，“隔离见证脚本哈希”和“赎回脚本哈希”都是 20 个字节的数据，因而只从地址外观无法分别以 3 开头的地址到底是 P2SH 类型还是 P2SH-P2WPKH 类型。从私钥或者公钥计算 P2SH-P2WPKH 地址的大致过程如下：



具体方法参考下面代码。

```
const hexPrivateKey = 'b4bcb509ef3362e1dc3e0a58a8073520f723b00fbd3cb24186de89cbe9c69e13';
// 1. 从十六进制私钥创建密钥对
const keyPair = ECPair.fromPrivateKey(Buffer.Buffer.from(hexPrivateKey, 'hex'), {
    network,
    compressed: true // 必须使用压缩公钥
});

// 2. 生成公钥哈希 :
const pubKeyHash = bitcoin.crypto.hash160(keyPair.publicKey);

// 3. 创建 P2WPKH 隔离见证脚本 (0014<20-byte-pubkey-hash>)
const witnessScript = bitcoin.script.compile([
    bitcoin.opcodes.OP_0,
    pubKeyHash
]);

// 4. 创建 P2SH 赎回脚本 (将隔离见证脚本包裹在 P2SH 中)
//--//方法 1:
//--const redeemScript = bitcoin.script.compile([
//--    bitcoin.opcodes.OP_HASH160,
//--    bitcoin.crypto.hash160(witnessScript),
//--    bitcoin.opcodes.OP_EQUAL
//--]);
//方法 2:
const scriptHash = bitcoin.crypto.hash160(witnessScript),

// 5. 生成 P2SH 地址
//--//方法 1:
//--const p2shAddress = bitcoin.address.fromOutputScript(redeemScript, network);
//方法 2:
const address = bitcoin.address.toBase58Check(
    scriptHash,
    network.scriptHash // 主网 0x05, 测试网 0xc4
);
```

4.3.5 P2TR 地址进阶

P2TR 的 Taproot 公钥（tweaked public key）记为 *Q*，存在公式 *Q=lift_x(P)+t*G*，其中 *P* 为内部公钥（某个普通公钥或者多签参与者提供的多个公钥的聚合公钥），*G* 为椭圆曲线的生成元，*t*G* 为 MAST 树产生的公钥，那 *t* 就是 MAST 树根的哈希值。那 *t* 是怎么来的呢？这还得从 MAST 树说起（见后续内容），lift_x(P) 的功能是返回椭圆曲线上一个点，这个点的 x 坐标就是 P 的 x 坐标，y 坐标是偶数。在椭圆曲线上，每一个 x 坐标对应的 y 坐标有两个，其中一个一定是奇数，另一个一定是偶数，而 Schnorr 签名的公钥只用 x 坐标，所以严格规定必须采用偶数的 y 坐标。

最终的 P2TR 类型的输出脚本就是“OP_1 20<*Q* 的 *x* 坐标>”，OP_1 为版本号，对应 0x51，Q 的 x 坐标占据 32 个字节。

假如<*Q* 的 *x* 坐标>为 6e95a29dc6732ee49d74404c318e36e32bfc45fca4918037ed416b252bce57d9，那么用 hex 表示的输出脚本就是 51206e95a29dc6732ee49d74404c318e36e32bfc45fca4918037ed416b252bce57d9，其中 20 表示 Q 的 x 坐标的长度。

内部公钥 P 的计算方法

早期就是把参与多签的全部公钥简单相加，但是后来出现了伪造证书来读取加密信息的攻击(即 Key Reinstallation Attacks，简称 KRACK)，因此计算方法被修改，具体参见图 4-6 所示，假设有三人参与多签。

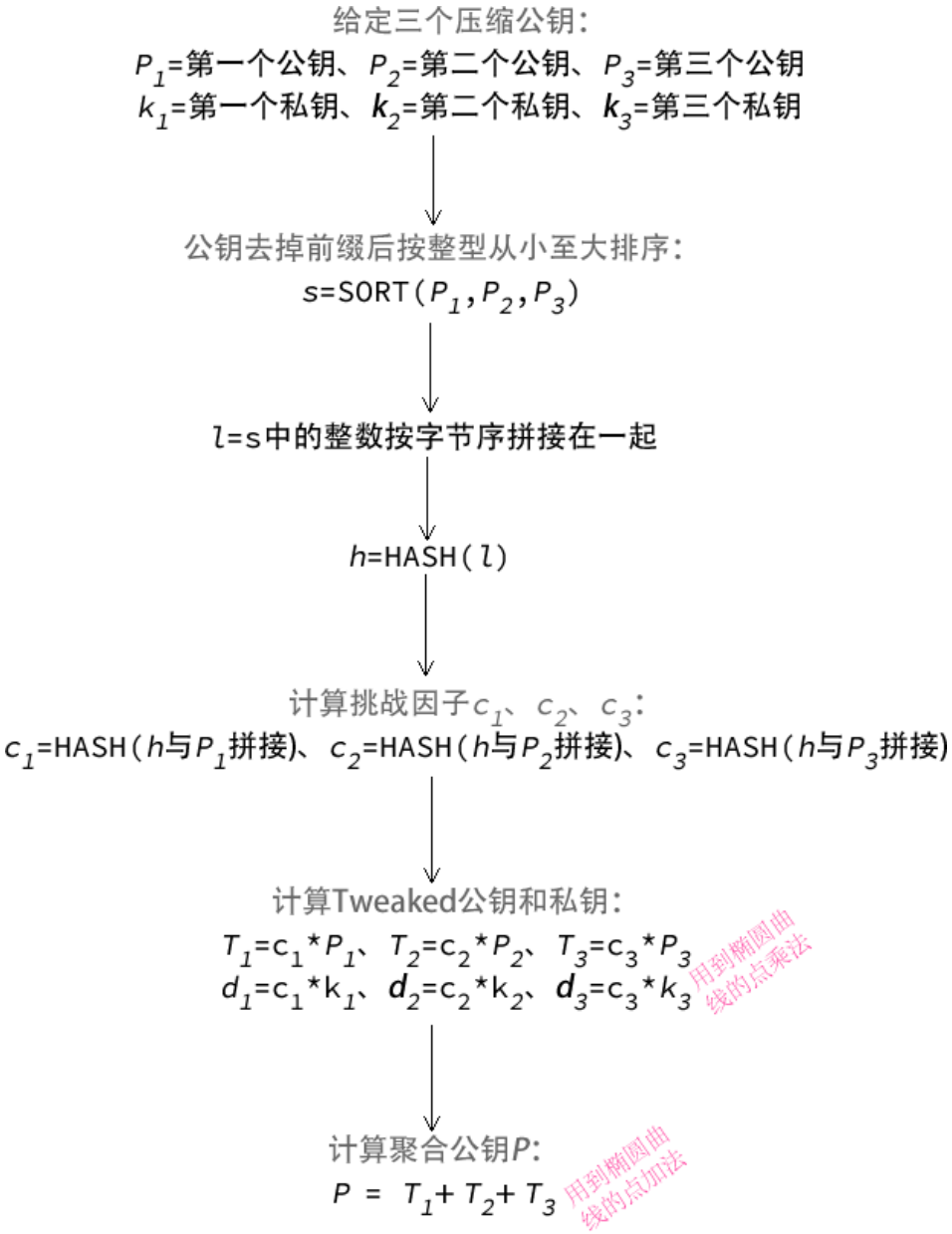


图 4-6 计算聚合公钥

其中（d_i,T_i）为一对经过调整（Tweaked）的公/私钥对，私钥 d_i，公钥 T_i，发起 P2TR 交易时，就是采用这种调整后的公私钥。下面举例说明如何计算聚合公钥：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');

const privateKeys = [
    Buffer.from("fdabf38a53f4a43b67c02d07db681ce0e3a6777bad7f8cc8b20d63aa26fb57e7", "hex"),            //K₁
    Buffer.from("e877816b39b65874fd7e96b979c22f7746936ba1c0c1187005cd8666c2ad99d3", "hex"),            //K₂
    Buffer.from("d816e35425c3c09d40ba29a7142b9629be38559648b054ae5821698bb911df52", "hex"),            //K₃
];
```

```
//椭圆曲线的阶:
const SECP256K1_N = BigInt("0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141");

const publicKeys = privateKeys.forEach(priv => ecc.pointFromScalar(priv));                    //从私钥产生公钥
publicKeys.forEach(e => {console.log(Buffer.from(e).toString('hex'))});
02a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956
02661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41
027def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d5

const s = publicKeys.sort();                                                                  //上图中的 s
s.forEach(e => {console.log(Buffer.from(e).toString('hex'))});
02661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41
027def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d5
02a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956

const l = Buffer.concat(s);
console.log(l.toString('hex'));
02661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41027def35da02605b57c9a66fd5813df877950f6f
d924291d881d1754c0fc6a93d502a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956

// 2. 计算聚合系数 h = SHA256(pk1 || pk2 || ...)
const h = bitcoin.crypto.sha256(l);
console.log(h.toString('hex'));
6510cf0480be50b587e15fc0c5f5a58b2ff09efccbb8ff5b54cbbf36b367bf53

// 3. 计算每个公钥的挑战因子: c_i = SHA256(L || pk_i) mod n
const coefficients = publicKeys.forEach(pk => {
   const hash = bitcoin.crypto.sha256(Buffer.concat([h, pk]));
   return BigInt("0x" + hash.toString("hex")) % SECP256K1_N;
});                                          //c₁=coefficients[0]、c₁=coefficients[1]、c₁=coefficients[2]
coefficients.forEach(e=>console.log(e.toString(16)));
a3e0431200ea6516329d961a6c29a4032da1ef439d994db2e5c6f6f0c68919b0
e00b5ac42f1d6681b923b4b5d406bbb90b2172a45a930ee970da73dc89e53436
e1ff6533cbd241592151964fcc4ce1e791c606ab37eb8d3bd56da02bfc35d52b

// 4. 计算聚合公钥: P = c₁*P₁ + c₂*P₂ + c₃*P₃
let P = null;
let Ts = [];                                                                                  //tₙ,n=1,2,3
let Ds = [];                                                                                  //dₙ,n=1,2,3
publicKeys.forEach((pk, i) => {
   const tweakBuffer = Buffer.from(coefficients[i].toString(16).padStart(64, '0'), 'hex');
   const t = ecc.pointMultiply(pk, tweakBuffer);
   Ts.push(Buffer.from(t));
   P = P ? ecc.pointAdd(P, t) : t;
   const d = coefficients[i] * BigInt("0x" + privateKeys[i].toString('hex')) % SECP256K1_N;
   Ds.push(Buffer.from(d.toString(16),'hex'));
});
console.log(Buffer.from(P).toString('hex'));
02f70508807c9dc8d7c3f461bd94adfca2860f1bba93e7f41884e37ba6e863d1ab                             //即上图中的 P
Ts.forEach(e => {console.log(Buffer.from(e).toString('hex'))});
03676767571c885d52a5d070a2cde3ba60e7f9e27c8551688a07e72455f075747d                                  //T₁
02e8c2d17eeb7023ebd3eda36d4aee73756d0fd267a0bdc939cb5d5d44ee25f8c0                                  //T₂
021e2fcbaeb5512a88bd321c6931e34421c4387ea2de0524ab69d85a4e3cccfa7d                                  //T₃
Ds.forEach(e => {console.log(Buffer.from(e).toString('hex'))});
ff462a79026c585f74213c9f0b780e70bdafe15ac382535ac1bf2ac27a12e55c                                   //d₁
d64cf2fa10fbf3f7dab35a183901da7054ef3d92adcf26d4e8d5fa97f17fa4eb                                   //d₂
e670a6f6cb7b065a4c98c5eb7f9f8509f8032f90c3bd993713dd92e252de5e42                                   //d₃
```

关于椭圆曲线上的点乘法和点加法请参考“§1.2 私钥和公钥”，另外图 4-6 涉及的操作“两个整数按字节序拼接”用 python 语句实现就是:

```
a = 0x10101
b = 0x20202
L = a.to_bytes(4, 'big') + b.to_bytes(4, 'big')
print(L)
b'\x00\x01\x01\x01\x00\x02\x02\x02'
```

以后签名只能用 d_1、d_2 和 d_3。

计算 t

之前讲过，t 就是 MAST 树根的哈希值。假设有三个人 A、B、C 参与多签，形成的一个 MAST 树如图 4-7 所示。

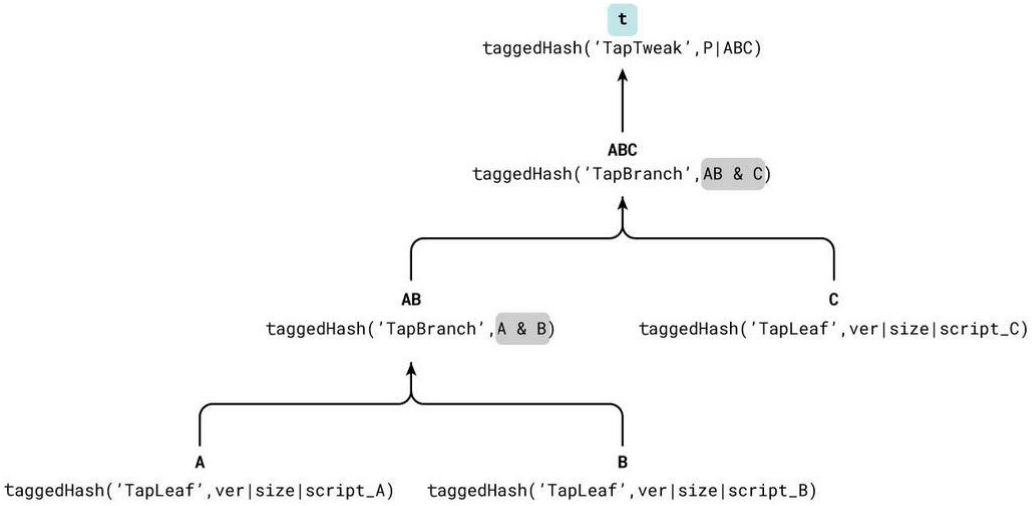


图 4-7 三人参与多签的 MAST 树

其中 script_A、script_B 和 script_C 分别是各人的输出脚本，ver 表示 Tapscript 版本号，当前默认值为 0xc0，size 表示输出脚本 scirpt_*的字节数，A&B 表示 A、B 从小到大排序后按字节拼接（A|B 表示 B 拼接在 A 的后面），ver|size|script_*表示 Buffer 类型的数据拼接在一起，类似这样的代码 Buffer.concat([Buffer.from('1234','hex'), Buffer.from('5678','hex')])，结果就是<Buffer 12 34 56 78>。taggedHash 表示带标签的 hash 函数，固定长度 32 字节，计算方式是：

```
taggedHash(tag, x) = sha256(Buffer.concat([TAGGED_HASH_PREFIXES[tag], x]));
TAGGED_HASH_PREFIXES = {
  ......
  TapLeaf: Buffer.from([
    174, 234, 143, 220, 66, 8, 152, 49, 5, 115, 75, 88, 8, 29, 30, 38, 56, 211,
    95, 28, 181, 64, 8, 212, 211, 87, 202, 3, 190, 120, 233, 238, 174, 234, 143,
    220, 66, 8, 152, 49, 5, 115, 75, 88, 8, 29, 30, 38, 56, 211, 95, 28, 181,
    64, 8, 212, 211, 87, 202, 3, 190, 120, 233, 238,
  ]),
  TapBranch: Buffer.from([
    25, 65, 161, 242, 229, 110, 185, 95, 162, 169, 241, 148, 190, 92, 1, 247,
    33, 111, 51, 237, 130, 176, 145, 70, 52, 144, 208, 91, 245, 22, 160, 21, 25,
    65, 161, 242, 229, 110, 185, 95, 162, 169, 241, 148, 190, 92, 1, 247, 33,
    111, 51, 237, 130, 176, 145, 70, 52, 144, 208, 91, 245, 22, 160, 21,
  ]),
  TapSighash: Buffer.from([
    244, 10, 72, 223, 75, 42, 112, 200, 180, 146, 75, 242, 101, 70, 97, 237, 61,
    149, 253, 102, 163, 19, 235, 135, 35, 117, 151, 198, 40, 228, 160, 49, 244,
    10, 72, 223, 75, 42, 112, 200, 180, 146, 75, 242, 101, 70, 97, 237, 61, 149,
    253, 102, 163, 19, 235, 135, 35, 117, 151, 198, 40, 228, 160, 49,
  ]),
  TapTweak: Buffer.from([
    232, 15, 225, 99, 156, 156, 160, 80, 227, 175, 27, 57, 193, 67, 198, 62, 66,
    156, 188, 235, 21, 217, 64, 251, 181, 197, 161, 244, 175, 87, 197, 233, 232,
    15, 225, 99, 156, 156, 160, 80, 227, 175, 27, 57, 193, 67, 198, 62, 66, 156,
    188, 235, 21, 217, 64, 251, 181, 197, 161, 244, 175, 87, 197, 233,
  ]),
  ......
}
```

有了 Taproot 根公钥 Q 后，就可以变形为收币地址了(对 Q 的 X 坐标进行 Bech32m 编码就可以得到 Taproot 地址)，Taproot 地址以“bc1p”开头。Q 的 x 坐标值（32 个字节）加上 0x51 前缀就是 P2TR 的输出脚本，即 0x5120< Q 的 x 坐标值>。看下面的例子——由 Taproot 公钥推导 P2TR 地址、再从地址推导输出脚本：

```
const bitcoin = require('bitcoinjs-lib');
const secp256k1= require('tiny-secp256k1');
```

```
bitcoin.initEccLib(secp256k1);

let taproot_publickey = '225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796';   //Taproot 公钥
let p2tr_address = bitcoin.address.toBech32(Buffer.from(taproot_publickey, 'hex'), 1, 'bc');//从公钥推导地址
console.log(p2tr_address);
bc1pyfw56zu5vsq0ulu9kytasgw4xwnm3eysll6tfdz8d9gtht97k7tqxsz78n                    //P2TR 类型的地址
let output_script = bitcoin.address.toOutputScript(p2tr_address);
console.log(output_script.toString('hex'));
'5120225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796'            //hex 格式的输出脚本
console.log(bitcoin.script.toASM(output_script));
OP_1 225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796              //反汇编输出脚
本
```

当然也可以从 P2TR 类型的地址导出 Taproot 公钥：

```
let p2tr_address = 'bc1pyfw56zu5vsq0ulu9kytasgw4xwnm3eysll6tfdz8d9gtht97k7tqxsz78n';
let taproot_publickey = bitcoin.address.fromBech32(p2tr_address).data.toString('hex');
console.log(taproot_publickey);
225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796
```

不过要注意的是 Taproot 公钥是你拥有的那个私钥所对应的公钥经调整而来的，前者是 Tweaked 公钥，后者称为内部公钥。

4.4 HD 钱包进阶

HD 钱包的核心是扩展私钥生成树（见图 4-8 所示），根私钥和私钥 k_n 都是 512 位的扩展私钥，扩展私钥以 xprv 为前缀，每一个扩展私钥都能导出最多 2^{32} 个（约 43 亿）扩展私钥，所以扩展私钥生成树的高度没有限制。父扩展私钥能确定地导出所有子扩展私钥，但是子扩展私钥不能反推导出父扩展私钥。512 位的扩展私钥的左边 256 位作为私钥，右边 256 位作为串码，串码、公钥和序号作为 HMAC-SHA512 函数的参数导出子扩展公钥，而串码、私钥和序号作为 HMAC-SHA512 函数的参数推导子扩展私钥——硬化推导。

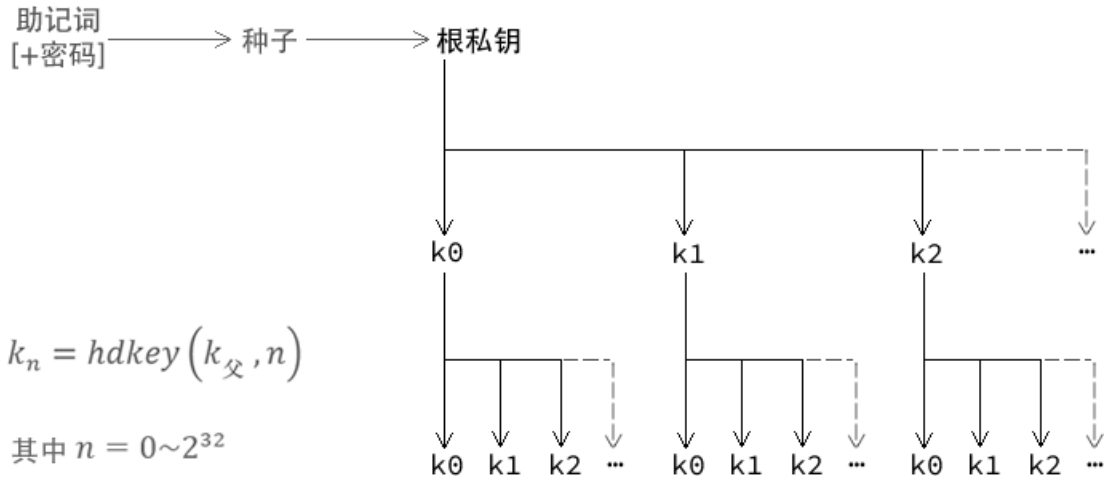


图 4-8 扩展私钥生成树

每一个子扩展私钥能计算出一个 512 位的扩展公钥，扩展公钥会添加 xpub 前缀。扩展公钥有一个令人惊奇的属性，那就是能直接导出下一层级的子扩展公钥，从而形成一棵扩展公钥生成树，类似图 4-8 那样。如此，一个扩展公钥可用两

种方法得到：一是由私钥算出，二是从父扩展公钥导出。扩展公钥生成树的一个很好的应用就是观察钱包——只监管此扩展公钥生成树上的所有地址上的资金变动情况，但是不能使用这些资金。扩张私钥生成树和扩展公钥生成树的关系参考图 4-9 所示。

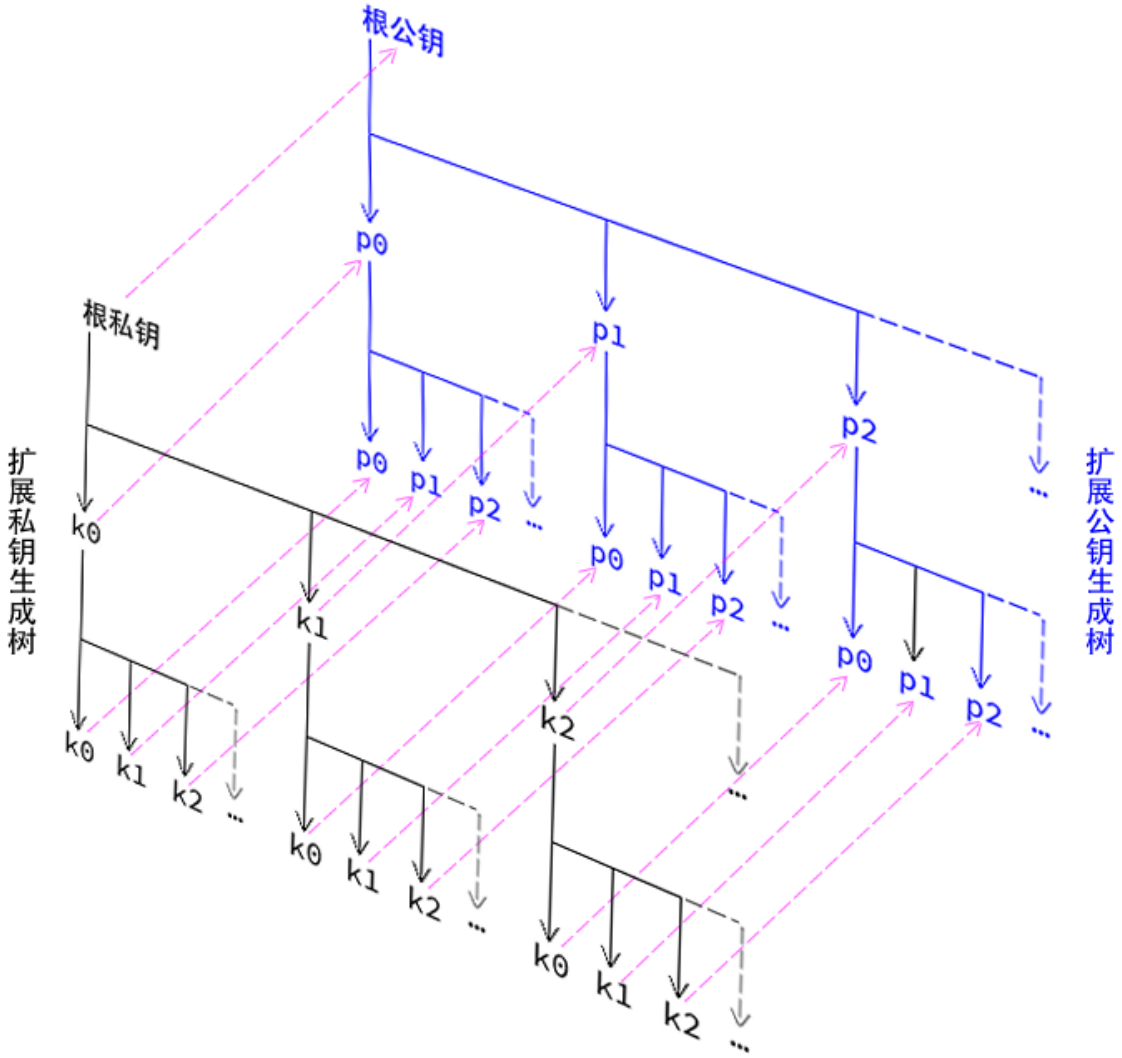


图 4-9 扩展公私要生成树的关系

由扩展私钥和扩展公钥可计算出真正用于加密币的私钥和公钥。

由扩展公钥直接导出下一层级的扩展公钥的理论依据是：K=k*G → K+i*G=k*G+i*G → K+i*G=(k+i)*G，比如 Bob 知道私钥 k 和公钥 K，然后把公钥 K 告诉 Alice，这样 Alice 可以直接算出第 i 个子公钥，即 K+i*G，Bob 也能算出第 i 个子私钥，即 k+i。这里的 i 称为钥匙扣（Key Tweak）。

通常把根私钥记作 m，子私钥按层级记作 m/x/y/z/…等，其中 x 为第一层上的索引、y 为第二层上的索引，以此类推，示意图参见图 4-10。

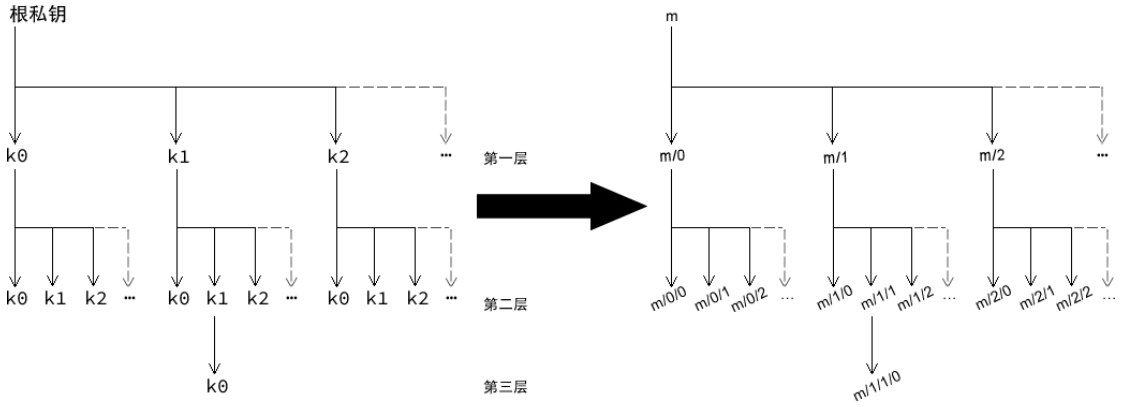


图 4-10 标记扩展私钥生成树

以后都是按照标记来产生对应的扩展私钥的，比如 m/2/0/3/8/5 标记在扩展私钥生成树上的路径：第一层上的索引 2、第二层索引 0、第三层索引 3、第四层索引 8、第五层索引 5。

通常把根公钥记作 M，子公钥按层级记作 M/x/y/z/…。

不过这样的私钥生成树存在一个严重的安全问题，就是能从子扩展私钥反向推导出父扩展私钥，进而推导出整个扩展私钥生成树！为了避免这个问题，人们引入了一种叫硬化的推导算法（Hardened Derivation），它“切断”了从子到父的反向推导路径，此算法把索引$[0, 2^{31})$作为普通索引，而索引$[2^{31}, 2^{32})$作为硬化索引，硬化索引通常记作 0'、1'、2'……，即索引 0'=2^{31}，1'=2^{31}+1，2'=2^{31}+2，以此类推。不过硬化索引同时破坏了扩展公钥生成树，即观察钱包无效了。

HD 钱包适用于大多数加密币，为了统一标准，人们制订了 BIP-44 规范——定义了一种如何派生私钥的标准，它的标记格式参见图 4-11 所示。

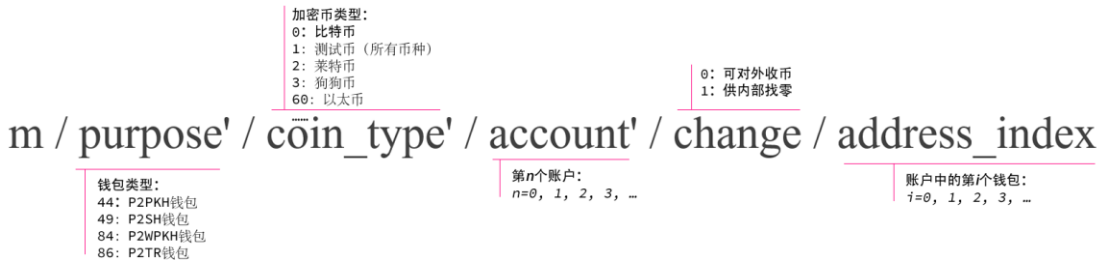


图 4-11 扩展私钥生成树节点编码格式

其中，purpose 可取值 44（P2PKH 传统地址）、49（P2SH 多签/隔离见证地址）、84（P2WPKH 原生隔离见证地址）和 86（P2TR 地址），因此 44'=2^{31}+44=2147483692；coin_type 在 SLIP-44 中定义加密币类型，比如比特币=0、所有加密币的测试链=1、莱特币=2、狗狗币=3、以太币=60 等；account 表示用户的某个“账户”，由用户自定义；change=0 表示外部交易（钱包外部可见，用于接受别人转来的币），change=1 表示内部交易（钱包内部使用，用于交易找零，内部钱包不用对外啊公部，具备更好的隐私）；address_index 则是真正派生的索引为 0～2^{31}-1 的地址。如 m/44'/0'/0'/0/1 表示比特币的第一个账户用于外部交易的索引为 1 的私钥。不过以太坊地址的编码格式首部固定为 m/44'/60'/i'/0/j，即第 i+1 个账户里的第 j+1 个钱包，当然长度允许延展，例如 m/44'/60'/1'/0/0/1/10，不过建议不要太长太复杂，免得被搞混遗忘。

从图 1-42 可知，第 1 至 3 层（即 purpose'/coin_type'/account'）都加了“'”，表明都是硬化的，第 4 层区分内外部钱包，真正的能收币的钱包在第 5 层，因此整个生成树的架构参见图 4-12。

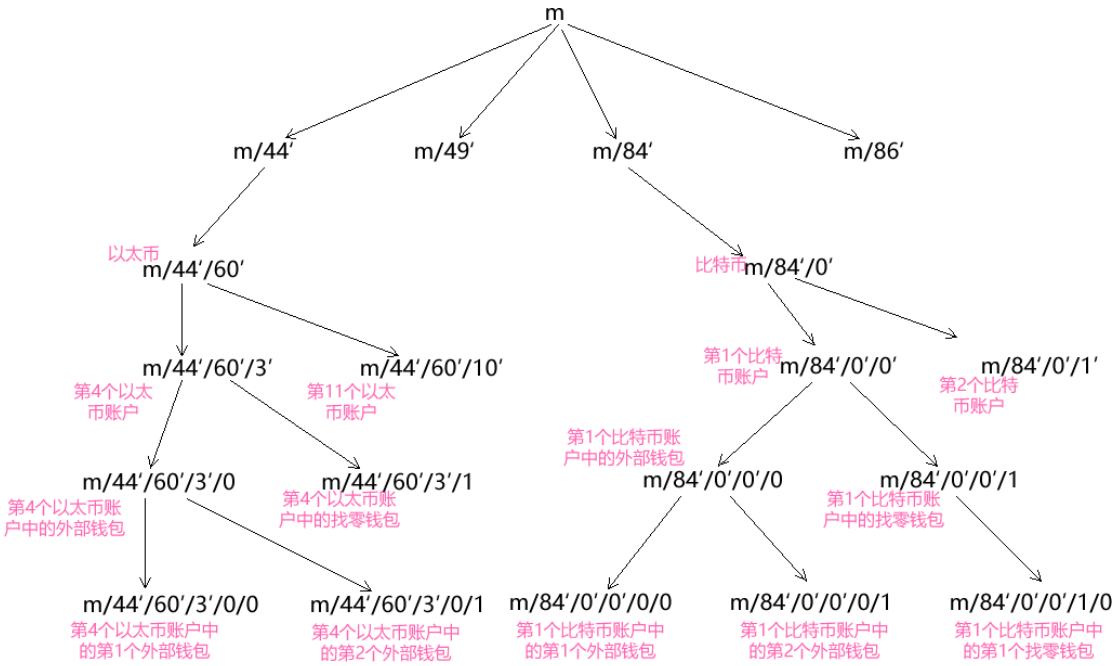


图 4-12 生成树的划分

对于一颗私钥生成树，我们只要记住了种子或者根私钥（参见图 4-8 所示）就能算出全部的子私钥，由于种子和根私钥都是 256 位二进制数，不好记住，为此，人们制订了 BIP-39 规范——一种通过助记词来推算种子的算法，具体介绍请垂阅下一节。

加密币钱包地址先后经历了四个阶段，最早是传统地址（Legacy），此后依次历经多签&隔离见证地址（Nested Segwit）、原生隔离见证地址（Native Segwit）以及主根地址（Taproot）。其中 Taproot 地址是最新发展起来的（由 BIP86 规范引入）。同一个私钥可以导出四种类型的地址。例如下面的是从 https://iancoleman.io/bip39/网站产生的一个私钥、公钥和比特币地址（适合于比特币交易）：

公钥：03bcca0d4632d16e18bb38b61469a59bfc4cfa6e20df4b90632dcc0055829cbb05

私钥：Ky7VKW3D6fLkCN4V2RxRPV9vYc5hEHUtf5z8MHrHhYY3W2gjvV7r

原生隔离见证地址：bc1q4dmtj07jay3dzeku2hs5tevtfkwkq3a8ru2f6m

下面是以太坊的私钥、公钥和地址（path = m/44'/60'/0'/0/0）：

私钥：0x819c0f76f560c359d80385ff87b4e9b447ae96055606f35c2e6bceea3425756f

公钥：0x03a6f084dd04a9ba005069eaa1173f04d0fcfe56864003a2a707b0a4401ce79822

地址：0x4A135285115520633C72c485c08c04956d5dA7Fc

尽量使用以 24 个助记词+密码产生的 HD 硬化钱包。

4.5 助记词

以英文单词为例，首先挑选 2048 个常用的英文单词，构造一个单词数组：const words = ['abandon', 'ability', 'able', ..., 'zoo']，

然后生成一个 128~256 位二进制的随机数，注意随机数的总位数必须是 32 的倍数，不妨假设 ENT 表示随机数的二进制位数。例如，生成的 256 位随机数以十六进制表示为 179e5af5ef66e5da5049cd3de0258c5339a722094e0fdbbbe0e96f148ae80924，此时 ENT=256，然后在随机数末尾加上校验码，校验码取随机数的 SHA256 哈希值的前 ENT/32 位二进制数，最后分割成长度为 11 的片段， 每个片段的取值都落在[0,2047]内，片段作为单词数组的下标，把这些下标对应的单词取出来并以空格作为分隔符组合在一起就是助记词。表 1-1 列举了随机数长度对应的助记单词个数的关系。

表 4-2 随机数长度与助记单词个数关系表

随机数长度（ENT）	哈希值前几位（CS=ENT/32）	合成数长度（ENT+CS）	助记单词个数(MS=(ENT+CS)/11)
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

诸如简体中文等其他语言也可以作助记词，具体参考 https://github.com/bitcoin/bips/blob/master/bip-0039/bip-0039-wordlists.md。图 4-30 列举了全部 2048 个简体中文助记词（保存在 bip39.wordlists.chinese_simplified 数组中），每行左侧的数字加上每列上面的数字就是数组下标，比如 200+14=214，bip39.wordlists.chinese_simplified[214]就是“资”，图 4-13 列出了全部的助记词集合。

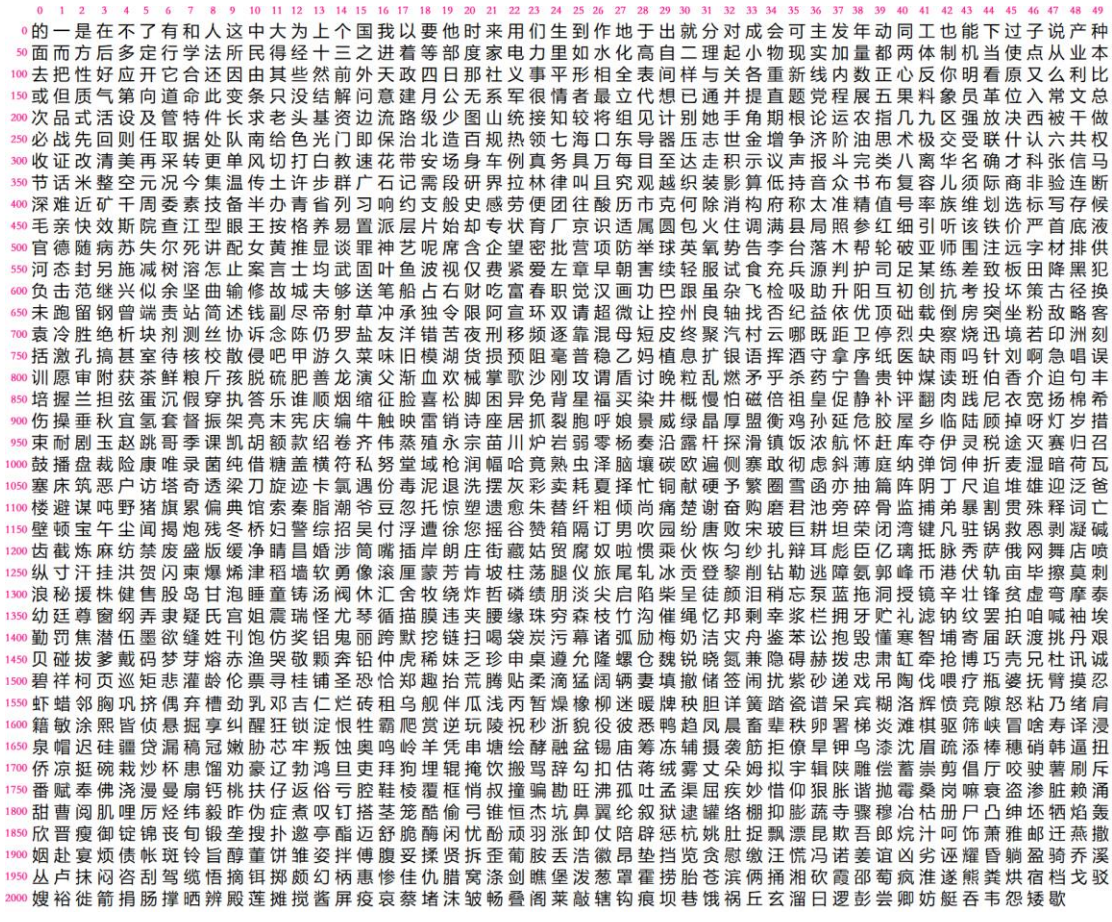


图 4-13 简体中文助记词集合

bip39.wordlists.english 保存了 2048 个英语单词，这些单词列举在图 4-14 中。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	abandon	ability	able	about	above	absent	absorb	abstract	absurd	abuse	access	accident	account	accuse	achieve	acid	acoustic	acquire
18	across	act	action	actor	actress	actual	adapt	add	addict	address	adjust	admit	adult	advance	advice	aerobic	affair	afford
36	afraid	again	age	agent	agree	ahead	aim	air	airport	aisle	alarm	album	alcohol	alert	alien	all	alley	allow
54	almost	alone	alpha	already	also	alter	always	amateur	amazing	among	amount	amused	analyst	anchor	ancient	anger	angle	angry
72	animal	ankle	announce	annual	another	answer	antenna	antique	anxiety	any	apart	apology	appear	apple	approve	april	arch	arctic
90	area	arena	argue	arm	armed	armor	army	around	arrange	arrest	arrive	arrow	art	artefact	artist	artwork	ask	aspect
108	assault	asset	assist	assume	asthma	athlete	atom	attack	attend	attitude	attract	auction	audit	august	aunt	author	auto	autumn
126	average	avocado	avoid	awake	aware	away	awesome	awful	awkward	axis	baby	bachelor	bacon	badge	bag	balance	balcony	ball
144	bamboo	banana	banner	bar	barely	bargain	barrel	base	basic	basket	battle	beach	bean	beauty	because	become	beef	before
162	begin	behave	behind	believe	below	belt	bench	benefit	best	betray	better	between	beyond	bicycle	bid	bike	bind	biology
180	bird	birth	bitter	black	blade	blame	blanket	blast	bleak	bless	blind	blood	blossom	blouse	blue	blur	blush	board
198	boat	body	boil	bomb	bone	bonus	book	boost	border	boring	borrow	boss	bottom	bounce	box	boy	bracket	brain
216	brand	brass	brave	bread	breeze	brick	bridge	brief	bright	bring	brisk	broccoli	broken	bronze	broom	brother	brown	brush
234	bubble	buddy	budget	buffalo	build	bulb	bulk	bullet	bundle	bunker	burden	burger	burst	bus	business	busy	butter	buyer
252	buzz	cabbage	cabin	cable	cactus	cage	cake	call	calm	camera	camp	can	canal	cancel	candy	cannon	canoe	canvas
270	canyon	capable	capital	captain	car	carbon	card	cargo	carpet	carry	cart	case	cash	casino	castle	casual	cat	catalog
288	catch	category	cattle	caught	cause	caution	cave	ceiling	celery	cement	census	century	cereal	certain	chair	chalk	champion	change
306	chaos	chapter	charge	chase	chat	cheap	check	cheese	chef	cherry	chest	chicken	chief	child	chimney	choice	choose	chronic
324	chuckle	chunk	churn	cigar	cinnamon	circle	citizen	city	civil	claim	clap	clarify	claw	clay	clean	clerk	clever	click
342	client	cliff	climb	clinic	clip	clock	clog	close	cloth	cloud	clown	club	clump	cluster	clutch	coach	coast	coconut
360	code	coffee	coil	coin	collect	color	column	combine	come	comfort	comic	common	company	concert	conduct	confirm	congress	connect
378	consider	control	convince	cook	cool	copper	copy	coral	core	corn	correct	cost	cotton	couch	country	couple	course	cousin
396	cover	coyote	crack	cradle	craft	cram	crane	crash	crater	crawl	crazy	cream	credit	creek	crew	cricket	crime	crisp
414	critic	crop	cross	crouch	crowd	crucial	cruel	cruise	crumble	crunch	crush	cry	crystal	cube	culture	cup	cupboard	curious
432	current	curtain	curve	cushion	custom	cute	cycle	dad	damage	damp	dance	danger	daring	dash	daughter	dawn	day	deal
450	debate	debris	decade	december	decide	decline	decorate	decrease	deer	defense	define	defy	degree	delay	deliver	demand	demise	denial
468	dentist	deny	depart	depend	deposit	depth	deputy	derive	describe	desert	design	desk	despair	destroy	detail	detect	develop	device
486	devote	diagram	dial	diamond	diary	dice	diesel	diet	differ	digital	dignity	dilemma	dinner	dinosaur	direct	dirt	disagree	discover
504	disease	dish	dismiss	disorder	display	distance	divert	divide	divorce	dizzy	doctor	document	dog	doll	dolphin	domain	donate	donkey
522	donor	door	dose	double	dove	draft	dragon	drama	drastic	draw	dream	dress	drift	drill	drink	drip	drive	drop
540	drum	dry	duck	dumb	dune	during	dust	dutch	duty	dwarf	dynamic	eager	eagle	early	earn	earth	easily	east
558	easy	echo	ecology	economy	edge	edit	educate	effort	egg	eight	either	elbow	elder	electric	elegant	element	elephant	elevator
576	elite	else	embark	embody	embrace	emerge	emotion	employ	empower	empty	enable	enact	end	endless	endorse	enemy	energy	enforce
594	engage	engine	enhance	enjoy	enlist	enough	enrich	enroll	ensure	enter	entire	entry	envelope	episode	equal	equip	era	erase
612	erode	erosion	error	erupt	escape	essay	essence	estate	eternal	ethics	evidence	evil	evoke	evolve	exact	example	excess	exchange
630	excite	exclude	excuse	execute	exercise	exhaust	exhibit	exile	exist	exit	exotic	expand	expect	expire	explain	expose	express	extend
648	extra	eye	eyebrow	fabric	face	faculty	fade	faint	faith	fall	FALSE	fame	family	famous	fan	fancy	fantasy	farm
666	fashion	fat	fatal	father	fatigue	fault	favorite	feature	february	federal	fee	feed	feel	female	fence	festival	fetch	fever
684	few	fiber	fiction	field	figure	file	film	filter	final	find	fine	finger	finish	fire	firm	first	fiscal	fish
702	fit	fitness	fix	flag	flame	flash	flat	flavor	flee	flight	flip	float	flock	floor	flower	fluid	flush	fly
720	foam	focus	fog	foil	fold	follow	food	foot	force	forest	forget	fork	fortune	forum	forward	fossil	foster	found
738	fox	fragile	frame	frequent	fresh	friend	fringe	frog	front	frost	frown	frozen	fruit	fuel	fun	funny	furnace	fury
756	future	gadget	gain	galaxy	gallery	game	gap	garage	garbage	garden	garlic	garment	gas	gasp	gate	gather	gauge	gaze
774	general	genius	genre	gentle	genuine	gesture	ghost	giant	gift	giggle	ginger	giraffe	girl	give	glad	glance	glare	glass
792	glide	glimpse	globe	gloom	glory	glove	glow	glue	goat	goddess	gold	good	goose	gorilla	gospel	gossip	govern	gown
810	grab	grace	grain	grant	grape	grass	gravity	great	green	grid	grief	grit	grocery	group	grow	grunt	guard	guess
828	guide	guilt	guitar	gun	gym	habit	hair	half	hammer	hamster	hand	happy	harbor	hard	harsh	harvest	hat	have
846	hawk	hazard	head	health	heart	heavy	hedgehog	height	hello	helmet	help	hen	hero	hidden	high	hill	hint	hip
864	hire	history	hobby	hockey	hold	hole	holiday	hollow	home	honey	hood	hope	horn	horror	horse	hospital	host	hotel
882	hour	hover	hub	huge	human	humble	humor	hundred	hungry	hunt	hurdle	hurry	hurt	husband	hybrid	ice	icon	idea
900	identify	idle	ignore	ill	illegal	illness	image	imitate	immense	immune	impact	impose	improve	impulse	inch	include	income	increase
918	index	indicate	indoor	industry	infant	inflict	inform	inhale	inherit	initial	inject	injury	inmate	inner	innocent	input	inquiry	insane
936	insect	inside	inspire	install	intact	interest	into	invest	invite	involve	iron	island	isolate	issue	item	ivory	jacket	jaguar
954	jar	jazz	jealous	jeans	jelly	jewel	job	join	joke	journey	joy	judge	juice	jump	jungle	junior	junk	just
972	kangaroo	keen	keep	ketchup	key	kick	kid	kidney	kind	kingdom	kiss	kit	kitchen	kite	kitten	kiwi	knee	knife
990	knock	know	lab	label	labor	ladder	lady	lake	lamp	language	laptop	large	later	latin	laugh	laundry	lava	law
1008	lawn	lawsuit	layer	lazy	leader	leaf	learn	leave	lecture	left	leg	legal	legend	leisure	lemon	lend	length	lens
1026	leopard	lesson	letter	level	liar	liberty	library	license	life	lift	light	like	limb	limit	link	lion	liquid	list
1044	little	live	lizard	load	loan	lobster	local	lock	logic	lonely	long	loop	lottery	loud	lounge	love	loyal	lucky
1062	luggage	lumber	lunar	lunch	luxury	lyrics	machine	mad	magic	magnet	maid	mail	main	major	make	mammal	man	manage
1080	mandate	mango	mansion	manual	maple	marble	march	margin	marine	market	marriage	mask	mass	master	match	material	math	matrix
1098	matter	maximum	maze	meadow	mean	measure	meat	mechanic	medal	media	melody	melt	member	memory	mention	menu	mercy	merge
1116	merit	merry	mesh	message	metal	method	middle	midnight	milk	million	mimic	mind	minimum	minor	minute	miracle	mirror	misery
1134	miss	mistake	mix	mixed	mixture	mobile	model	modify	mom	moment	monitor	monkey	monster	month	moon	moral	more	morning
1152	mosquito	mother	motion	motor	mountain	mouse	move	movie	much	muffin	mule	multiply	muscle	museum	mushroom	music	must	mutual
1170	myself	mystery	myth	naive	name	napkin	narrow	nasty	nation	nature	near	neck	need	negative	neglect	neither	nephew	nerve
1188	nest	net	network	neutral	never	news	next	nice	night	noble	noise	nominee	noodle	normal	north	nose	notable	note
1206	nothing	notice	novel	now	nuclear	number	nurse	nut	oak	obey	object	oblige	obscure	observe	obtain	obvious	occur	ocean
1224	october	odor	off	offer	office	often	oil	okay	old	olive	olympic	omit	once	one	onion	online	only	open
1242	opera	opinion	oppose	option	orange	orbit	orchard	order	ordinary	organ	orient	original	orphan	ostrich	other	outdoor	outer	output
1260	outside	oval	oven	over	own	owner	oxygen	oyster	ozone	pact	paddle	page	pair	palace	palm	panda	panel	panic
1278	panther	paper	parade	parent	park	parrot	party	pass	patch	path	patient	patrol	pattern	pause	pave	payment	peace	peanut
1296	pear	peasant	pelican	pen	penalty	pencil	people	pepper	perfect	permit	person	pet	phone	photo	phrase	physical	piano	picnic
1314	picture	piece	pig	pigeon	pill	pilot	pink	pioneer	pipe	pistol	pitch	pizza	place	planet	plastic	plate	play	please
1332	pledge	pluck	plug	plunge	poem	poet	point	polar	pole	police	pond	pony	pool	popular	portion	position	possible	post
1350	potato	pottery	poverty	powder	power	practice	praise	predict	prefer	prepare	present	pretty	prevent	price	pride	primary	print	priority
1368	prison	private	prize	problem	process	produce	profit	program	project	promote	proof	property	prosper	protect	proud	provide	public	pudding
1386	pull	pulp	pulse	pumpkin	punch	pupil	puppy	purchase	purity	purpose	purse	push	put	puzzle	pyramid	quality	quantum	quarter
1404	question	quick	quit	quiz	quote	rabbit	raccoon	race	rack	radar	radio	rail	rain	raise	rally	ramp	ranch	random
1422	range	rapid	rare	rate	rather	raven	raw	razor	ready	real	reason	rebel	rebuild	recall	receive	recipe	record	recycle
1440	reduce	reflect	reform	refuse	region	regret	regular	reject	relax	release	relief	rely	remain	remember	remind	remove	render	renew
1458	rent	reopen	repair	repeat	replace	report	require	rescue	resemble	resist	resource	response	result	retire	retreat	return	reunion	reveal
1476	review	reward	rhythm	rib	ribbon	rice	rich	ride	ridge	rifle	right	rigid	ring	riot	ripple	risk	ritual	rival
1494	river	road	roast	robot	robust	rocket	romance	roof	rookie	room	rose	rotate	rough	round	route	royal	rubber	rude
1512	rug	rule	run	runway	rural	sad	saddle	sadness	safe	sail	salad	salmon	salon	salt	salute	same	sample	sand
1530	satisfy	satoshi	sauce	sausage	save	say	scale	scan	scare	scatter	scene	scheme	school	science	scissors	scorpion	scout	scrap
1548	screen	script	scrub	sea	search	season	seat	second	secret	section	security	seed	seek	segment	select	sell	seminar	senior
1566	sense	sentence	series	service	session	settle	setup	seven	shadow	shaft	shallow	share	shed	shell	sheriff	shield	shift	shine
1584	ship	shiver	shock	shoe	shoot	shop	short	shoulder	shove	shrimp	shrug	shuffle	shy	sibling	sick	side	siege	sight
1602	sign	silent	silk	silly	silver	similar	simple	since	sing	siren	sister	situate	six	size	skate	sketch	ski	skill
1620	skin	skirt	skull	slab	slam	sleep	slender	slice	slide	slight	slim	slogan	slot	slow	slush	small	smart	smile
1638	smoke	smooth	snack	snake	snap	sniff	snow	soap	soccer	social	sock	soda	soft	solar	soldier	solid	solution	solve
1656	someone	song	soon	sorry	sort	soul	sound	soup	source	south	space	spare	spatial	spawn	speak	special	speed	spell
1674	spend	sphere	spice	spider	spike	spin	spirit	split	spoil	sponsor	spoon	sport	spot	spray	spread	spring	spy	square
1692	squeeze	squirrel	stable	stadium	staff	stage	stairs	stamp	stand	start	state	stay	steak	steel	stem	step	stereo	stick
1710	still	sting	stock	stomach	stone	stool	story	stove	strategy	street	strike	strong	struggle	student	stuff	stumble	style	subject
1728	submit	subway	success	such	sudden	suffer	sugar	suggest	suit	summer	sun	sunny	sunset	super	supply	supreme	sure	surface
1746	surge	surprise	surround	survey	suspect	sustain	swallow	swamp	swap	swarm	swear	sweet	swift	swim	swing	switch	sword	symbol
1764	symptom	syrup	system	table	tackle	tag	tail	talent	talk	tank	tape	target	task	taste	tattoo	taxi	teach	team
1782	tell	ten	tenant	tennis	tent	term	test	text	thank	that	theme	then	theory	there	they	thing	this	thought
1800	three	thrive	throw	thumb	thunder	ticket	tide	tiger	tilt	timber	time	tiny	tip	tired	tissue	title	toast	tobacco
1818	today	toddler	toe	together	toilet	token	tomato	tomorrow	tone	tongue	tonight	tool	tooth	top	topic	topple	torch	tornado
1836	tortoise	toss	total	tourist	toward	tower	town	toy	track	trade	traffic	tragic	train	transfer	trap	trash	travel	tray
1854	treat	tree	trend	trial	tribe	trick	trigger	trim	trip	trophy	trouble	truck	TRUE	truly	trumpet	trust	truth	try
1872	tube	tuition	tumble	tuna	tunnel	turkey	turn	turtle	twelve	twenty	twice	twin	twist	two	type	typical	ugly	umbrella
1890	unable	unaware	uncle	uncover	under	undo	unfair	unfold	unhappy	uniform	unique	unit	universe	unknown	unlock	until	unusual	unveil
1908	update	upgrade	uphold	upon	upper	upset	urban	urge	usage	use	used	useful	useless	usual	utility	vacant	vacuum	vague
1926	valid	valley	valve	van	vanish	vapor	various	vast	vault	vehicle	velvet	vendor	venture	venue	verb	verify	version	very
1944	vessel	veteran	viable	vibrant	vicious	victory	video	view	village	vintage	violin	virtual	virus	visa	visit	visual	vital	vivid
1962	vocal	voice	void	volcano	volume	vote	voyage	wage	wagon	wait	walk	wall	walnut	want	warfare	warm	warrior	wash
1980	wasp	waste	water	wave	way	wealth	weapon	wear	weasel	weather	web	wedding	weekend	weird	welcome	west	wet	whale
1998	what	wheat	wheel	when	where	whip	whisper	wide	width	wife	wild	will	win	window	wine	wing	wink	winner
2016	winter	wire	wisdom	wise	wish	witness	wolf	woman	wonder	wood	wool	word	work	world	worry	worth	wrap	wreck
2034	wrestle	wrist	write	wrong	yard	year	yellow	you	young	youth	zebra	zero	zone	zoo				

图 4-14 英语助记词集合

例如下面是一组由 24 个单词组成的助记词：

bleak version runway tell hour unfold donkey defy digital abuse glide please omit much cement sea sweet tenant demise taste emerge inject cause link

生成助记词的过程是计算机随机产生的，用户只要记住这些助记词，就可以根据助记词推算出 HD 钱包的种子，不过强烈建议由助记词推算种子时加上密码，这样即使助记词被泄露，在别人不知道密码的情况下，算出的种子是不同的，因此最终推导的扩展私钥生成树也不同，这样就相当于又加了一把锁。

使用 2048 轮 HMAC-SHA512 算法计算 512 位比特的种子：

```
种子 = PBKDF2( PRF = HMAC-SHA512，密码 = 助记词(UTF-8)，盐 = "mnemonic"+[可选密码](UTF-8)，迭代次数 = 2048，输出长度 = 64 字节 )
```

可见助记词作为 PBKDF2 函数的密码，而助记词保护密码作为 PBKDF2 函数的盐

SLIP39 在 BIP39 的基础上做了改进，即由多组助记词中的少量组就能恢复种子！各组助记词可由不同的人保管，比如夫妻双方和小孩各保管一组助记词，然后规定以后只要用两组即可恢复种子，或者总经理、分管财务的副总和财务主管各保管一组。SLIP39 官网 https://slip39.com/，代码托管在 https://github.com/iancoleman/slip39，实战网站 https://iancoleman.io/slip39/，参考图 4-15 所示。

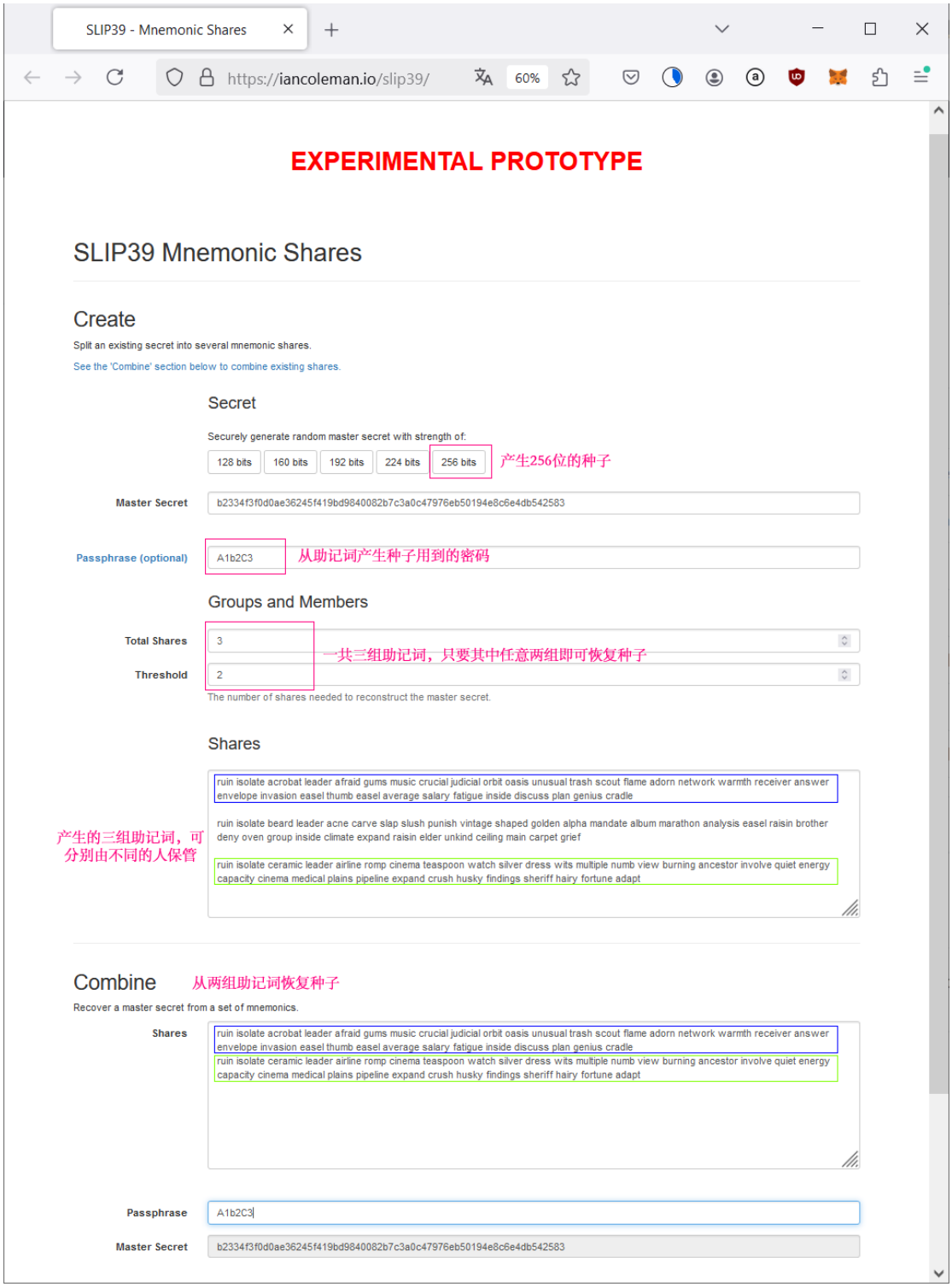


图 4-15 SLIP39 规范实战

显然由一个随机数推导出来的下标是确定的，那么由这组下标从各种语言中都可取出一组助记词，由于在计算种子时把助记词作为密码使用，所以即使由同一个随机数产生的不同语言中的助记词，推导出来的种子完全不同。

下面我们手工来产生助记词：

```
//下面开始从熵 entropy 产生助记词:
const entropy = new Uint8Array(32);                                    //16,20,24,28 或者 32
crypto.getRandomValues(entropy);
```

```
const hash = ethers.sha256(entropy);
const hashBits = ethers.toBeArray(hash).reduce(
   (acc, byte) => acc + byte.toString(2).padStart(8, '0'), ''
);

const checksumLength = entropy.length / 4;
const checksum = hashBits.slice(0, checksumLength);

const entropyBits = entropy.reduce(
   (acc, byte) => acc + byte.toString(2).padStart(8, '0'), ''
);

const bits = entropyBits + checksum;

const chunks = bits.match(/.{1,11}/g) || [];                                    //按 11 位长度拆分成数组
const indices = chunks.map(binary => parseInt(binary, 2));
const mnemonic_en = indices.reduce(
   (acc, index) => acc + ' ' + bip39.wordlists.english[index], ''
);

const mnemonic_cn = indices.reduce(
   (acc, index) => acc + ' ' + bip39.wordlists.chinese_simplified[index], ''
);

console.log(`英语助记词: ${mnemonic_en.trim()}\n 简体中文助记词: ${mnemonic_cn.trim()}`);
英语助记词: hover ordinary picnic shoulder recycle bomb jungle tiger habit uniform nature honey wood carbon
fluid laptop menu bulb atom dentist image penalty science belt
简体中文助记词: 磁 纵 汤 辉 懂 品 殖 纬 矛 撒 园 免 敲 海 洋 鼓 脂 指 前 片 套 浪 疗 建
```

用下面的语句把英语助记词转换为简体中文助记词：

```
const entropy = ethers.toBeArray(ethers.Mnemonic.phraseToEntropy(mnemonic_en.trim()));
const mnemonic = ethers.Mnemonic.entropyToPhrase(entropy, wordlistsExtra.LangZh.wordlist('cn'));
```

4.5 比特币进阶

4.5.1 UTXO

还没有被下一个交易花费的 Output 被称为 UTXO——Unspent TX Output，即未花费的交易输出。一个地址的全部 UTXO 的金额加起来就是钱包的总余额（假设一个钱包对应一个地址的话）。

对于目前最流行的隔离见证交易，一笔交易包含这些主要数据：交易 id 号、若干个输入、若干个输出、见证数据等，交易 id 号是全局唯一的，通过它能查询到那笔交易，比如你可以在 https://blockchair.com/zh/bitcoin 上查询 113a340329a6e782b2a5485f2adfd8fae2d31ba389db6f1ae64cf638ef3e5d19 这个交易 id 号；而每个输入的组成包括：另外一笔交易（假设为 X）的 id 号、X 的输出序号等，“交易 id+输出序号”能唯一引用一个输出；每个输出的组成：对方地址、输出序号（从 0 开始）、发生额和输出脚本等。一个交易的输入指向另一些交易的输出，具体参考图 4-16。

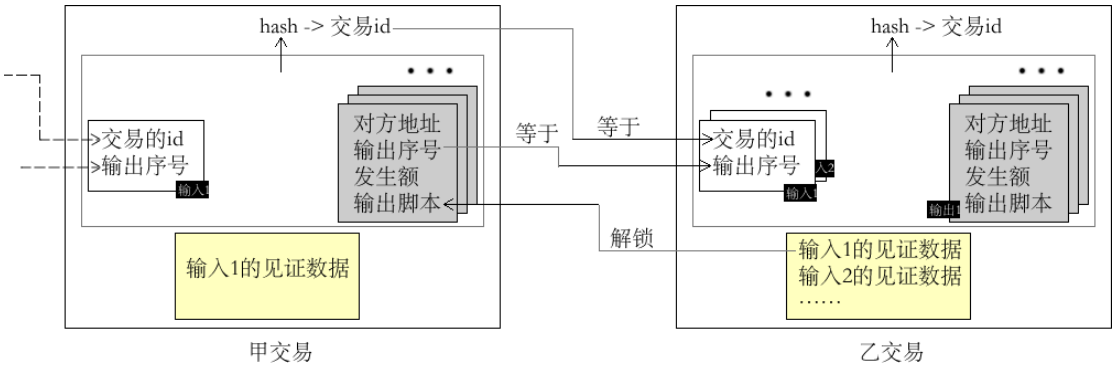


图 4-16 交易组成及输入引用

一个输入一定引用了另一笔交易的输出（当然除了挖矿奖励之外），此“引用”就是花钱（花掉一个 UTXO）的意思，一旦一个 UTXO 被花掉，里面的比特币就被全部花掉，不存在只花掉部分比特币的问题，因此一个输出只有两种状态：被花掉或没花掉，没被花掉的输出称为 UTXO。交易是保存在区块里面的（参考图 1-44），而区块一旦上了链，区块里面的内容就再也不能修改，这意味着区块中的交易永远不会被修改。那么又是怎么来判断一个输出是不是被花掉了呢？很简单，就是查询此输出是不是被另一个交易的一个输入引用，如果引用了表明已经花掉，否则就没有花掉。

每个输出中有一个输出脚本（你可以理解为一把锁），因为一个输入必须引用一个输出，前面讲过“引用”就是“花钱”，所以一个输入能否合法花掉一个 UTXO，关键就看输入提供的输入脚本（你可以理解为钥匙）能不能解锁输出中的输出脚本，能解锁就能花钱，否则不能花钱（矿工会拒绝不能花钱的交易）。在《精通比特币（第三版）》中，输出脚本英文名叫 Output Script，由于输入脚本（Input Script）发展到后来占据比较大的空间，因此单独把输入脚本从交易的每个输入中独立出来存放，独立存放的输入脚本称为隔离见证。总之见证数据（或输入脚本）与输出脚本的关系就是钥匙和锁的关系。

例如“OP_DUP OP_HASH160 0aa35c213b8e3de1bad9325b7696dde7f443f2ca OP_EQUALVERIFY OP_CHECKSIG”就是一个最早期的 P2PKH 类型的输出脚本（汇编格式），相应的此输出脚本的输出脚本必须是“<签名><公钥>”（即正确的签名和公钥），矿工在构造区块时要验证能否“解锁”。解锁过程具体是这么做的：首先把输出脚本附加在输入脚本的右边，然后采用堆栈引擎从左至右开始执行，还是拿刚才的例子说明，合并后如下：

```
<签名><公钥> OP_DUP OP_HASH160 0aa35c213b8e3de1bad9325b7696dde7f443f2ca OP_EQUALVERIFY OP_CHECKSIG
```

首先遇到“<签名>”，直接压栈，然后遇到“<公钥>”，也压栈，再遇到“OP_DUP”，就把栈顶的“<公钥>”复制一份并压栈，接下来遇到“O_HASH160”就弹出栈顶的“<公钥>”并对它做 hash160 运算，把运算结果压栈，遇到输出脚本中的常数“0aa35c213b8e3de1bad9325b7696dde7f443f2ca”就直接压栈，遇到“OP_EQUALVERIFY”就比较堆栈最上面的两个元素，如果相等就弹出它们，否则表明交易非法，矿工直接丢弃此交易。现在堆栈中还有两个元素，即“<公钥><签名>”，接下来遇到输出脚本中的“OP_CHECKSIG”，就用堆栈里的“<公钥>”验证堆栈里的“<签名>”即可，如果签名验证失败就丢弃这个交易。解锁成功的条件是处理过程中没有出错且堆栈最后为空。

正如生活当中的门锁由早期的挂锁发展到后来的球锁、密码锁、指纹锁、遥控锁那样，输出脚本也发展了很多代。早期的输入脚本直接放在每个输入中，而且被打包进区块，如此每个区块能容纳的交易数量就很少，因为区块大小受到严格限制（不大于 4M），最近发展到把每个输入中的输入脚本单独保存（即隔离见证），而且见证数据不参与生成交易的 id——交易 id 是通过对交易的哈希运算得到的。

脚本中诸如 OP_DUP、OP_HASH160、OP_EQUALVERIFY、OP_CHECKSIG 等操作符的详细解释请参考 https://en.bitcoin.it/wiki/Script 网站。由此网站查询可知 OP_DUP 对应的操作码 0x76，OP_HASH160 为 0xa9、OP_EQUALVERIFY 为 0x88，OP_CHECKSIG 为 0xac，“0aa35c213b8e3de1bad9325b7696dde7f443f2ca”的长度为 0x14，所以汇编格式的输出脚本“OP_DUP OP_HASH160 0aa35c213b8e3de1bad9325b7696dde7f443f2ca OP_EQUALVERIFY

OP_CHECKSIG”编译成十六进制格式的输出脚本就是 76a9140aa35c213b8e3de1bad9325b7696dde7f443f2ca88ac，最终保存在交易中的输出脚本就是这个格式。当然也可以采用下面的语句编译：

```
const bitcoin = require('bitcoinjs-lib');
const asm = 'OP_DUP OP_HASH160 0aa35c213b8e3de1bad9325b7696dde7f443f2ca OP_EQUALVERIFY OP_CHECKSIG';
console.log(bitcoin.script.fromASM(asm).toString('hex'));
76a9140aa35c213b8e3de1bad9325b7696dde7f443f2ca88ac
```

下一小节详细介绍输出脚本和见证数据的格式以及验证过程。

4.5.2 解锁过程

输出脚本类似方程，输入脚本类似方程的解，把这个解代入方程，如果方程成立说明解锁成功，所以后来把输入脚本称为见证数据，这似乎更贴切。比如 x+2=4 为输出脚本，2 就是见证数据，因为把 2 代入方程 2+2=4 成立，任何能使输出脚本为“真”的见证数据都是合法的，例如 8-6 也是刚才这个例子的见证数据。当然在加密币交易中，见证数据一般只有私钥拥有者才知道，需要用到私钥，比如签名。见证数据的核心就是证明谁是加密币的拥有者，比如早期的输出脚本就是一个“能用这个公钥验证的签名”，输入脚本就是一个“签名”，即能提供正确的签名就证明谁就是加密币的主人，请回顾“§1.3 数字签名”小节——私钥签名，公钥验证签名。

输出脚本、输入脚本、赎回脚本、见证数据以及具体的解锁过程可能是理解加密币最难的地方。钱包的地址类型不同，那么输出脚本和见证数据的格式也不尽相同。关于脚本的规范，请浏览 https://en.bitcoin.it/wiki/Script。

脚本在堆栈引擎中被执行。早期执行的过程是这样的：把输出脚本附加在见证数据或输入脚本的末尾，然后从左至右开始执行，遇到常量就压栈，遇到操作符就执行相应的动作，直到出错或者执行完毕，最后堆栈中剩下 TRUE 表示解锁成功，否则解锁失败。比如下面的例子。

```
输出脚本：3 OP_ADD 5 OP_EQUAL
见证数据（或称为输入脚本）：2。
```

把输出脚本附加在见证数据（或称为输入脚本）的末尾得到：

```
2 3 OP_ADD 5 OP_EQUAL
```

然后从左至右执行：遇到 2 压栈，遇到 3 压栈，遇到操作符 OP_ADD，弹出栈顶两个数后相加得到 5，再把 5 压栈，接下来遇到 5 压栈，最后遇到 OP_EQUAL 操作符，弹出栈顶两个元素（都是 5），然后比较它们是否相等，相等就为真，把 TRUE 压栈，即解锁成功。

显然对于一个确定的输出脚本来说，存在很多的输入脚本，如上面的例子，只要输入脚本的执行结果为 2 即可，比如 10 8 OP_SUB 也是合法的见证数据。

不过输出脚本和输入脚本合并后再执行的机制存在一个 BUG——见证数据 1 OP_RETURN 能解锁任何输出脚本！因为操作符 OP_RETURN 终止继续执行脚本，且此时栈内只有一个元素 1（即 TRUE），表示解锁成功。为了解决这个 BUG，新规要求：1）输入脚本和输出脚本分开执行——先执行输入脚本，然后执行输出脚本；2）输入脚本中的指令执行出错，或者全部执行完毕；3）保持堆栈内的数据不变，然后执行输出脚本。

比特币地址的类型不同，输出脚本就不同，那么显然输入脚本也不同了。接下来我们逐一介绍。

1. P2PK 类型的地址

P2PK 是比特币刚刚诞生时采用的地址类型，直接把公钥作为收币地址，其输出脚本和输入脚本最简单，如下：

```
<pubKey> OP_CHECKSIG                                                    //输出脚本
<sig>                                                                   //输入脚本
```

验证过程就是用公钥“<pubKey>”验证签名“<sig>”即可，而“<sig>”就是用“<pubKey>”对应的私钥对本交易的哈希值签名（采用 ECDSA 签名算法）。不过新钱包一般不再使用 P2PK 地址，因为交易费用过高。

例如：

```
输出脚本：03d260c2cefb9ebecb0c77473209e2d5a59ffaaa7f0c0ecc4ec7c134491aa27bb3ac
输入脚本：483045022100b8ba06b873826dcc4e6db5cff360c0f670ee527a1422f77489a27ea443
```

绿色部分就是压缩公钥，红色部分就是 OP_CHECKSIG 操作码，蓝色部分就是签名。

2. P2PKH 类型的地址

计算公钥的哈希值并作为收币地址，其输出脚本和输入脚本如下：

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG               //输出脚本（汇编格式）
<sig> <pubKey>                                                          //输入脚本（汇编格式）
```

验证过程是对“<pubKey>”作 hash160 哈希运算后与“<pubKeyHash>”比较，如果相等就进一步用“<pubKey>”验证签名“<sig>”。看例子：

```
输出脚本：76a914c825a1ecf2a6830c4401620c3a16f1995057c2ab88ac
输入脚本：
47304402204244bf2735c3ba76eecf7b3145abf12d08ffbbbebd50dc32c349ce4aed265f9d022073d257e6a2ea611f19cc3637529d
7776c63a7a1a86d51531d765135ab051a67c014104e7e8eeb1d060e2ae611e2e6253b7948d7c02b7ea7c2dce09d939225a5c39d865
53b07acf5073b1dc1e432cdf574d587b2800e0c3d4e43fcb3a1f84bfdf602a1e
```

14 为十进制的 20，表示公钥哈希的长度，在 https://en.bitcoin.it/wiki/Script 上查询得知：OP_DUP=0x76，OP_HASH160=0xa9，OP_EQUALVERIFY=0x88，OP_CHECKSIG=0xac。输出脚本中的 47（十进制 71）为签名长度，41 为非压缩公钥长度，不过从 2012 年开始，输入脚本中的公钥必须为压缩公钥。

3. P2SH 类型的地址

2012 年引入 P2SH。之前讲过，P2SH 的钱包地址就是赎回脚本的哈希值经 Base58Check 编码后得到的，地址以 3 开头，如“38bPsA6ZXfRuxFD7efVXTkQd69422uzD4B”，P2SH 支持多签是一个令人激动的特征之一。

对于一个 *t-of-k* 多签的 P2SH 交易，赎回脚本（Redeem Script）的一般形式是（也可以自定义）：

```
t <pubKey1> <pubKey2> ... <pubKeyk> k OP_CHECKMULTISIG
```

其中：t≤签名个数≤k，且 1≤t, f≤16，也就是说 P2SH 最多支持 16 个签名，即需要至少 t 个签名才能花钱，这特别适合企业开支——需要总经理、分管财务的副总经理、财务主任和经办员一起签字才能支出一笔大额款项。

输出脚本的一般形式是：

```
OP_HASH160 <赎回脚本哈希值> OP_EQUAL
```

<赎回脚本哈希值>= RIPEMD160(SHA256(赎回脚本))，一共 20 个字节。

输入脚本的一般形式是：

```
<sig₁> <sig₂> ... <sigₜ> <赎回脚本>
```

假设是一个 *2-of-3* 的 P2SH 多签交易，假设参与者为 A、B、C，那么上面的三个脚本就具体成了下面的格式。

赎回脚本（Redeem Script）：

```
2 <pubKeyA> <pubKeyB> <pubKeyC> 3 OP_CHECKMULTISIG
```

输出脚本：

```
OP_HASH160 <20 个字节的赎回脚本的哈希值> OP_EQUAL
```

输入脚本（假如由 B 和 C 参与签名）：

```
<sigB> <sigC> <赎回脚本>
```

解锁时分两步处理，首先计算赎回脚本的哈希值，并与输出脚本中的哈希值比较，是否相等：

```
<2 pubKeyA pubKeyB pubKeyC 3 OP_CHECKMULTISIG> OP_HASH160 <20 个字节的赎回脚本的哈希值> OP_EQUAL
```

然后验证多签：

```
<sigB> <sigC> 2 <pubKeyA> <pubKeyB> <pubKeyC> 3 OP_CHECKMULTISIG
```

处理过程：首先入栈<sigB>、<sigC>、2、<pubKeyA>、<pubKeyB>、<pubKeyC> 、3，栈顶元素是 3，接下来遇到 OP_CHECKMULTISIG 操作符，即多签验证，这时要弹出堆栈顶部 7 个元素，三个公钥中只要存在两个能验证签名的公钥，解锁就算成功。

产生赎回脚本、输出脚本的实验请参考“4.3.2 比特币地址”小节。

4. P2WPKH 类型的地址

通行的输出脚本如下（允许定制）：

```
OP_0 <pubKeyHash>                                                    //汇编格式
0014318e36e32bfc45fca4918037ed416b252bce57d9                         //hex 格式
```

00 就是版本号 0，14 就是十进制的 20，表示公钥哈希值的长度，蓝色字体部分就是举例说明的脚本哈希值了。OP_0 为见证版本 0（即 segwit v0），<pubKeyHash>就是公钥的哈希值，即由 hash160(<压缩公钥>)计算得到，共 20 个字节。

通行的输入脚本如下（隔离见证数据，也可以定制）：

```
<sig> <pubKey>                                                       //汇编格式
02483045022100f1622d4147f1856cadc312df066e5a19b382b20a45d6a16efbf912b94eac5ebb022036cfe4f3a1e2342c185aa0168dbe0e388c3199d6d0d74ded414bc3787e41669f012102cdcfe7b2facd6a068fa0fb0ac5aa02fee40995ef6a7bd06a54d6c758988ad8fa
                                                                     //hex 格式（举例说明）
```

02 表示本隔离见证包含两个条款，绿色 48 表示第一个条款的长度，即签名长度为 72 个字节，绿色的 21 表示第二个条款的长度，即压缩公钥的长度为 33 个字节。

解锁过程：

（1）. 首先从输出脚本中抽出<pubKeyHash>并转换为 P2PKH 的输出脚本格式：

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

最后把转换后的输出脚本附加到见证数据的末尾，形如：

```
<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

（2）．把<sig> <pubKey>压栈，接下来遇到 OP_DUP，把栈顶元素复制一份并压栈，这时栈内有：

```
<sig> <pubKey> <pubKey>
```

（3）．遇到 OP_HASH160 操作符，弹出栈顶元素并计算其 hash160 哈希值，再压栈，这时栈内元素为：

```
<sig> <pubKey> <pubKeyHash>
```

（4）．把<pubKeyHash>压栈，接下来遇到 OP_EQUALVERIFY 操作符，弹出堆栈顶部两个元素并比较它们是否相等，如果相等就继续。这时堆栈内容如下：

```
<sig> <pubKey>
```

（5）．最后遇到 OP_CHECKSIG 操作符，那么弹出栈内最后两个元素，然后用公钥<pubKey>验证签名<sig>。

5. P2WSH 类型的地址

输出脚本：

```
OP_0 <32 个字节的 SHA256(赎回脚本)>                                    //汇编格式
00206e95a29dc6732ee49d74404c318e36e32bfc45fca4918037ed416b252bce57d9    //hex 格式
```

00 就是版本号 0，20 就是十进制的 32，表示脚本哈希值的长度，蓝色字体部分就是举例说明的脚本哈希值了。可见输出脚本的格式与 P2WPKH 完全相同，只是第二部分的长度不同，P2WPKH 只有 20 个字节。

输入脚本为空，而见证数据为（标准格式）：

```
<sig1> <sig2> ... <sigN> <赎回脚本>                                    //汇编格式
040048304502210...
044022004aaa8c83cf0a3ae69145bcf5ed596f191a2fa8affd78091220b187bf404e64e022060957cb42da6df3fbde0d7780d8e9ee67644b1760433424d6362cc86a8bc04e501695211 03b1d7
b531a4a9ca4a701cae7b1260bbe47b0b063ca40de3a3015f073d64ce614f21020166885dc4c7f6989e377eaff4bc87e0b4491efb61e28a4175f4c967a0d0917421023d60b07d04326b1277cf8
4a4a17f6683de8f798e0efb570be76223bbb63faef053ae                        //hex 格式（2-of-3 例子）
```

赎回脚本与 P2SH 相同，对于 2-of-3 都是 OP_2 <pubKey1> <pubKey2> <pubKey3> OP_3 OP_CHECKMULTISIG。开头的 0400 表示本见证有 4 个条款，第一个条款长度为 0（空条款），接下来的 48 为第二个条款的长度为 72 个字节（即第一个签名的长度），再接下来的 47 表示第三个条款长度为 71 个字节（即第二个签名的长度），69 表示整个赎回脚本的长度为 105 个字节，52 就是 OP_2，后面三个黄色 21 表示三个公钥的长度都为 33 个字节，53 对应 OP_3，ae 表示 OP_CHECKMULTISIG。

赎回脚本作为见证数据的最后一个条目存在，在它之前的条目作为赎回脚本的参数。赎回脚本允许自定义，总长度不能超过 3600 个字节。

解锁过程是这样的：

1)、计算赎回脚本的 SHA-256 值，并与输出脚本中的“<32 个字节的 SHA256(赎回脚本)>”比较，如果相等继续；

2)、见证数据中除了最后一个条目，其他的条目全部入栈；

3)、然后执行赎回脚本，如果最后结果为 true，表示解锁成功。

比如除了签名外，再添加一个猜谜（参见图 4-17）：

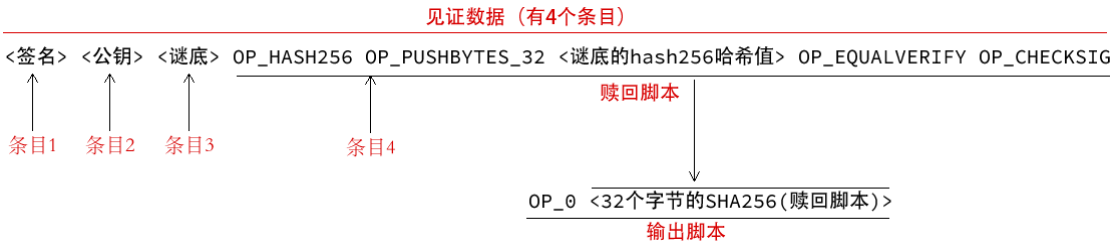


图 4-17 自定义赎回脚本

解锁过程如下：

1)、入栈“<签名>”，然后入栈“<公钥>”，再入栈“<谜底>”；

2)、遇到操作符“OP_HASH256”，则出栈“<谜底>”，计算它的 hash256 后，把计算结果入栈；

3)、遇到操作符“OP_PUSHBYTES_32”，把接下来的“<谜底的 hash256 哈希值>”入栈；

4)、遇到操作符“OP_EQUALVERIFY”，弹出栈顶两个元素并比较它们是否相等，相等则继续，否则解锁失败；

5)、此时站内还有签名和公钥，最后遇到操作符“OP_CHECKSIG”，则弹出栈顶两个元素并验证签名。

注意 OP_EQUAL 和 OP_EQUALVERIFY 两个操作符的区别：前者还会把比较结果（true 或者 false）入栈，而且还需要 OP_VERIFY 配合，后者弹出栈顶两个元素并比较，如果不等直接以失败结束，相等也不入栈 true，所以执行效率更高。

6. P2SH-P2WPKH 类型的地址

P2SH 和 P2SH-P2WPKH 的比较参见表 4-2。

表 4-2 P2SH 和 P2SH-P2WPKH 的比较

组成部分	P2SH多签	P2SH-P2WPKH
赎回脚本	`t pubKeys... k OP_CHECKMULTISIG`	`OP_0 <pubKeyHash>`
输出脚本	`OP_HASH160 <hash> OP_EQUAL`	`OP_HASH160 <hash> OP_EQUAL`
输入脚本	`sigs... <赎回脚本>`	`OP_PUSHBYTES_22 <赎回脚本>`
见证数据	无	`<sig> <pubKey>`
签名位置	在ScriptSig中	在Witness中
典型地址	3... (主网)	3... (主网)

下面是一个花费 P2SH-P2WPKH 类型 UTXO 的实际例子：

```
输出脚本：OP_HASH160 OP_PUSHBYTES_20 be662dcac3e5d09e4cbe13deb11f39a8b2ef5344 OP_EQUAL（汇编格式）

赎回脚本：OP_0 OP_PUSHBYTES_20 b23450d31f49a43dcb05ab26bd1f2348fd533e47（汇编格式）
         0014b23450d31f49a43dcb05ab26bd1f2348fd533e47
输入脚本：OP_PUSHBYTES_22 0014b23450d31f49a43dcb05ab26bd1f2348fd533e47（汇编格式）
         160014b23450d31f49a43dcb05ab26bd1f2348fd533e47
隔离见证（格式<signature> <pubKey>）：
3045022100adfaa30759f2bebf945e87ff475218e7427c0c69dce1972a95548172e50c7b430220371f1aa7a24c88afe662a72b38a4fa64548ad
9c894882e43fc767ce268caaee701 03f3ac2a5ba8a7d78f094d17106764a6737ea69803b38a960ed15cdc55e484738e
```

解锁时必须提供正确的赎回脚本、输入脚本、和隔离见证。解锁过程分成如下几步：

步骤 1：验证输出脚本

把输出脚本附加在输入脚本末尾，然后从左往右在堆栈中执行：

```
OP_PUSHBYTES_22 0014b23450d31f49a43dcb05ab26bd1f2348fd533e47 OP_HASH160 OP_PUSHBYTES_20 be662dcac3e5d09e4cbe13deb11f39a8b2ef5344 OP_EQUAL
```

首先遇到操作符“OP_PUSHBYTES_22”，则入栈后续的 22 个字节“0014b23450d31f49a43dcb05ab26bd1f2348fd533e47”，接下来遇到“OP_HASH160”，则出栈“0014b23450d31f49a43dcb05ab26bd1f2348fd533e47”，并计算它的哈希值（参考下面的语句），再入栈得到的哈希值。

```
bitcoin.crypto.hash160(Buffer.from('0014b23450d31f49a43dcb05ab26bd1f2348fd533e47', 'hex')).toString('hex');
//应得到哈希值 be662dcac3e5d09e4cbe13deb11f39a8b2ef5344
```

接下来遇到“OP_PUSHBYTES_20”，表示直接入栈后面的 20 个字节“be662dcac3e5d09e4cbe13deb11f39a8b2ef5344”，接下来执行到操作符“OP_EQUAL”，就弹出栈顶两个元素并比较它们是否相等。本例子是相等的。

步骤 2：验证隔离见证

从隔离见证中分离公钥“03f3ac2a5ba8a7d78f094d17106764a6737ea69803b38a960ed15cdc55e484738e”，并计算公钥的哈希（参考下面的语句），最后用得到的公钥哈希值与赎回脚本中的<pubKeyHash>（即 b23450d31f49a43dcb05ab26bd1f2348fd533e47）比较是否相等。

```
bitcoin.crypto.hash160(Buffer.from('03f3ac2a5ba8a7d78f094d17106764a6737ea69803b38a960ed15cdc55e484738e', 'hex')).toString('hex');
//得到的哈希为 b23450d31f49a43dcb05ab26bd1f2348fd533e47
```

最后用公钥验证隔离见证中的签名。

7. P2TR 类型的地址

输出脚本很简单，只有一个 Schnorr 公钥加上版本 1 做前缀：

```
OP_1 <x-only-pubKey>                                                    //汇编格式
51206e95a29dc6732ee49d74404c318e36e32bfc45fca4918037ed416b252bce57d9    //hex 格式
```

51 就是版本号“1”，20 就是十进制的 32，表示 Schorr 公钥的长度，蓝色字体部分就是公钥本身了，当采用密钥路径花钱时，就采用此公钥验证签名。

P2TR 类型的 UTXO 有两种花费方法：第一种方法是提供 Schnorr 签名（用 Schnorr 公钥对应的私钥签名）（称为密钥路径花费，类似 P2PKH），第二种方法是提供控制块和脚本（称为脚本路径花费，类似 P2SH）。

下面是密钥路径花费的一个例子：

```
51207bf5d836308dcebe5793f64c65bffd732f09376e039c4691bcd575217d1fd8d4                    //hex 格式的输出脚本
0141e6e1fe41524e65e3040bc3d080a136345c2c806eb7f336dd6a7a79e9054b0d1fc6a8d836667ef6e9f2188cd1270ab28e5e0eb6
42eac89f2ec50a32ca54aaf9d601                                                            //hex 格式的隔离见证
```

01 表示见证条款数 1 个，41（十六进制）表示 Schnorr 签名长度为 65（十进制）个字节，最后的 01 为 sig_hash 值：SIGHASH_ALL (0x01)，如果取 SIGHASH_DEFAULT（0x00）就可省略它，这时签名长度为 64，比如交易 37777defed8717c581b4c0509329550e344bdc14ac38f71fc050096887e535c8 的首个 Input 的签名只占 64 字节。不过要注意 Schnorr 签名的私钥为 k+t，其中 k 称为内部私钥，就是我们平时拥有的私钥，关于 t 请参考“§4.3.5 P2TR 地址进阶”小节中的 P2TR 部分。因为(k+t, Q)才是一对合法的公私钥对。

当见证条款数目大于 1 时就是脚本路径花费了，最后一个条款是控制块（Control Block），倒数第二个条款是脚本（位于 MAST 树的叶子上），其他条款就是脚本的参数。下面看一个具体的脚本路径花费的例子（摘自交易 905ecdf95a84804b192f4dc221cfed4d77959b81ed66013a7e41a6e61e7ed530，钱包地址 bc1p9l9dw3cz09jje30c3wysseuddax40t6kyuvrkqlusszvknskmzys00k7zm，花费的 UTXO=（txHash: 5b79f5d6039c188613342eb13961dd7d1e1a0f90023d3eaed25fc85a29201bb4, output-index:0)）：

```
51202fcad7470279652cc5f88b8908678d6f4d57af5627183b03fc8404cb4e16d889                    //hex 格式的输出脚本
0441 23b1d4ff27b16af4b0fcb9672df671701a1a7f5a6bb7352b051f461edbc614aa6068b3e5313a174f90f3d95dc4e06f69bebd9c
f5a3098fde034b01e69e8e788901400fd4a0d3f36a1f1074cb15838a48f572dc18d412d0f0f0fc1eeda9fa4820c942abb77e4d1a3c
2b99ccf4ad29d9189e6e04a017fe611748464449f681bc38cf39440febe583fa77e49089f89b78fa8c116710715d6e40cc5f5a075
ef1681550dd3c4ad20d0fa46cb883e940ac3dc5421f05b03859972639f51ed2eccbf3dc5a62e2e1b15ac41c02e44c9e47eaeb4bb31
3adecd11012dfad435cd72ce71f525329f24d75c5b9432774e148e9209baf3f1656a46986d5f38ddf4e20912c6ac28f48d6bf74746
9fb1                                                                                   //hex 格式的隔离见证
```

04 表示包含 4 个条款，后面绿色的 41、40、44 和 41 分别表示各个条款的长度（字节数）。最后一个条款为控制块，即：

```
c02e44c9e47eaeb4bb313adecd11012dfad435cd72ce71f525329f24d75c5b9432774e148e9209baf3f1656a46986d5f38ddf4e209
12c6ac28f48d6bf747469fb1
```

其中 c0（有时为 c1）为 MAST 树叶版本（c?&0xfe）和校验位(c?&1 = y(Q)%2)，紫色字体部分就是内部公钥 P（Internal Key P），也称聚合公钥，剩余部分就是兄弟节点的哈希（从叶子到根的路径如果比较长，那么路径上的每一个兄弟都要提供）。倒数第二个见证条款是一个脚本（对应默克尔树上的一片叶子），即：

```
20febe583fa77e49089f89b78fa8c116710715d6e40cc5f5a075ef1681550dd3c4ad20d0fa46cb883e940ac3dc5421f05b03859972
639f51ed2eccbf3dc5a62e2e1b15ac
```

对应的汇编格式：

```
febe583fa77e49089f89b78fa8c116710715d6e40cc5f5a075ef1681550dd3c4 OP_CHECKSIGVERIFY
d0fa46cb883e940ac3dc5421
f05b03859972639f51ed2eccbf3dc5a62e2e1b15 OP_CHECKSIG
```

这是两个 Schnorr 公钥加上两个操作符，显然需要提供两个 Schnorr 签名才能解锁。

倒数第二个条款之前的所有条款为上面那个脚本的参数，本例中是两个 Schnorr 签名，因为本 UTXO 是 2-of-2 多签。这里的脚本格式为“<P1> OP_CHECKSIGVERIFY <P2> OP_CHECKSIG”，输入参数是“sig(P2) sig(P1)”，执行时把脚本附加在参数后面，然后开始在堆栈引擎中执行，执行结果为 true 表示通过。操作符 OP_CHECKSIGVERIFY 和 OP_CHECKSIG 都是弹出堆栈顶部的两个元素（其中最上面的那个元素是公钥，另一个是签名）并验证签名，只不过 OP_CHECKSIGVERIFY 操作符验证签名成功后继续执行脚本，而 OP_CHECKSIG 不管验证成功与否都不再继续执行脚本。但是最终能否成功花钱还得先验证脚本（即倒数第二个条款）是不是在 MAST 树上，下面我们介绍到底如何验证这一步。假设 MAST 树如图 4-17 所示。

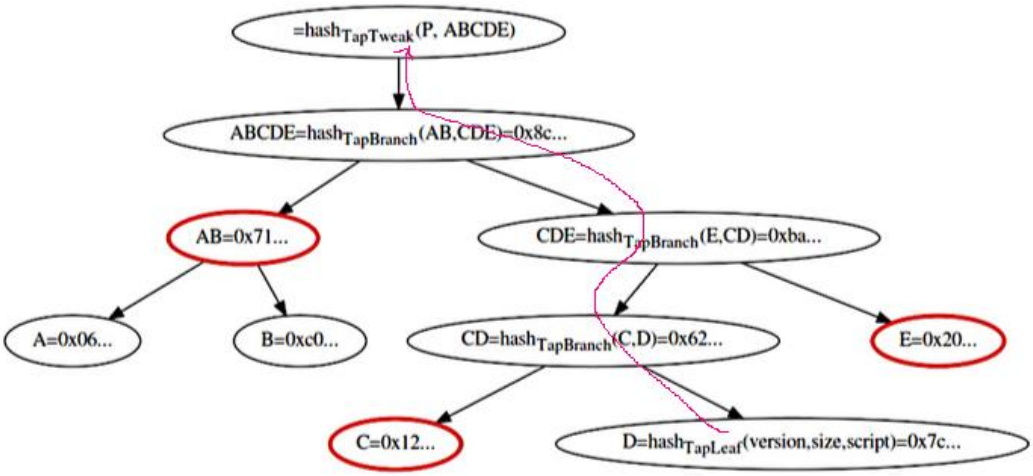


图 4-17 验证脚本在 MAST 上

有五个叶子节点，代表五个脚本，等于说存在六种花钱的方法（密钥路径一种、脚本路径五种）。假设现在想使用脚本 D 来花钱，那么除公开脚本 D 源码外，只需再提供 C、E 和 AB 三个哈希值就可以证明脚本 D 是不是在 默克尔树上，并不需要公开其它脚本（即 A、B、C、E）的源码。因此完整控制块就是：

```
<Control byte with leaf version and parity bit> <Internal key P> <C> <E> <AB>
```

其中一个字节的<Control byte with leaf version and parity bit>和 32 个字节的<Internal key P>是必须要的，其他的都是 32 个字节长度，是要根据路径长度来选择，因此控制块的总长度为 33+32*m，其中 m 表示路径沿线上的兄弟个数，取值范围是[0,128]，比如在图 1-60 中，m 为 3。<Control byte with leaf version and parity bit>等于 0xc0|y(Q)%2，即它的最低位如果等于 0，那么 Q 对应的 y 坐标是偶数，否则就是奇数，因此<Control byte with leaf version and parity bit>要么等于 0xc0，要么等于 0xc1。

回到图 4-17 这个例子，叶子 D 中的脚本（script）为倒数第二个见证条款，然后从叶子 D 开始一路往根计算哈希，最后得到 t，再计算 Taproot 公钥 Q（具体计算过程请参考“§4.3.5 P2TR 地址”小节），如果 Q 等于输出脚本中的公钥，那说明见证中的脚本确实在默克尔树上。

上面举的一个那个实现脚本路径花费的例子，控制块的组成和对应的 MAST 树参见图 4-18。

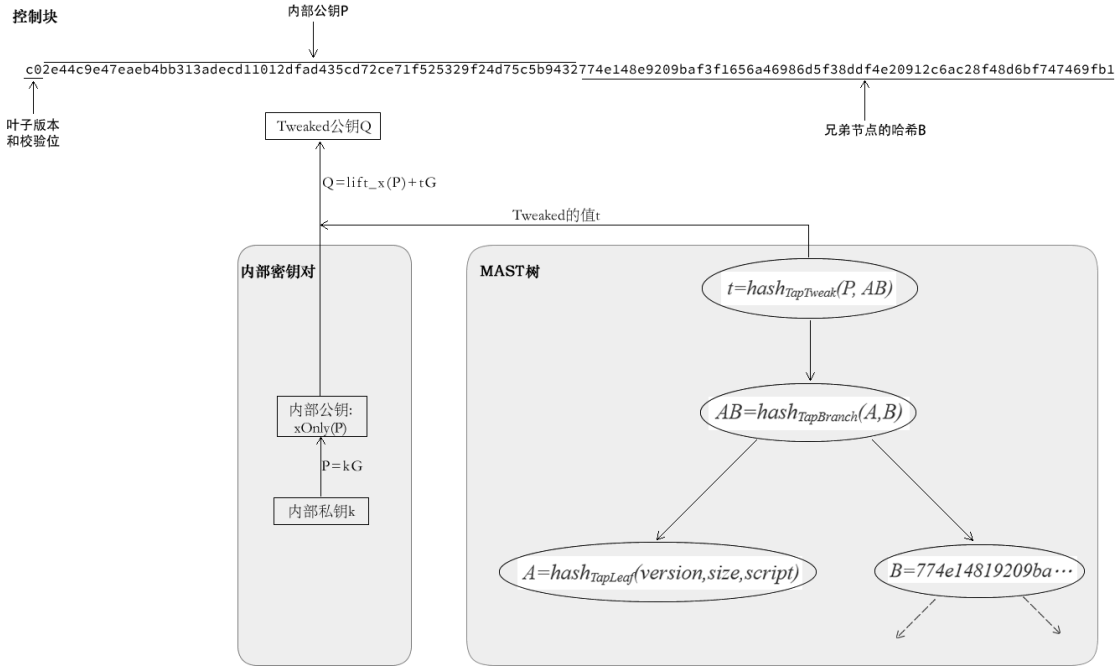


图 4-18 控制块与对应的 MAST 树

下面我们来计算 t、Q 和 P2TR 地址:

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');
bitcoin.initEccLib(ecc);

const output = '51202fcad7470279652cc5f88b8908678d6f4d57af5627183b03fc8404cb4e16d889';     //UTXO 的输出脚本
const version = 'c0';
const script =
'20febe583fa77e49089f89b78fa8c116710715d6e40cc5f5a075ef1681550dd3c4ad20d0fa46cb883e940ac3dc5421f05b0385997
2639f51ed2eccbf3dc5a62e2e1b15ac';
//对应的汇编'febe583fa77e49089f89b78fa8c116710715d6e40cc5f5a075ef1681550dd3c4 OP_CHECKSIGVERIFY
d0fa46cb883e940ac3dc5421f05b03859972639f51ed2eccbf3dc5a62e2e1b15 OP_CHECKSIG'

const size = '44';                                                                    //script 的长度（字节数）
const B = '774e148e9209baf3f1656a46986d5f38ddf4e20912c6ac28f48d6bf747469fb1';          //MAST 树上的兄弟哈希
const P = '2e44c9e47eaeb4bb313adecd11012dfad435cd72ce71f525329f24d75c5b9432';          //内部公钥

const utxoScript = Buffer.from(output, 'hex');
const utxoPubKey = utxoScript.slice(2);                                               //输出中的公钥
const internalPubKey = Buffer.from(P, 'hex');
const siblingHash = Buffer.from(B, 'hex');

const s = version + size + script;
let data = Buffer.from(s, 'hex');
console.log(s);
c04420febe583fa77e49089f89b78fa8c116710715d6e40cc5f5a075ef1681550dd3c4ad20d0fa46cb883e940ac3dc5421f05b0385
9972639f51ed2eccbf3dc5a62e2e1b15ac

let A = bitcoin.crypto.taggedHash('TapLeaf', data);

let AB = bitcoin.crypto.taggedHash(
     'TapBranch',
     Buffer.concat([ A < siblingHash ? A: siblingHash, A < siblingHash ? siblingHash: A])
);

let t = bitcoin.crypto.taggedHash('TapTweak', Buffer.concat([internalPubKey, AB]));          //内部公钥在前面
console.log(t.toString('hex'));
4cb37e7aa8fe3b7a1a6a93096f4ac2f72bc8b47e31703df82728d4493e1d52c1                       //t
```

```
const taprootPubKey = ecc.xOnlyPointAddTweak(internalPubKey, t).xOnlyPubkey;
const Q = Buffer.from(taprootPubKey);
console.log(Q.toString('hex'));
2fcad7470279652cc5f88b8908678d6f4d57af5627183b03fc8404cb4e16d889 ----------//Q
console.log(utxoPubKey.equals(Q));
true                                                    //两个公钥相等，表明脚本在 MAST 书上

const p2trAddress = bitcoin.address.fromOutputScript(
  bitcoin.script.fromASM(`OP_1 ${Q.toString('hex')}`),
  bitcoin.networks.bitcoin
);
console.log(p2trAddress);
bc1p9l9dw3cz09jje30c3wysseuddax40t6kyuvrkqlusszvknskmzys00k7zm ----------//P2TR 地址也对了
```

最后总结为表 4-3。

表 4-3 地址类型、输入/输出脚本关系

地址类型	输入脚本	输出脚本	隔离见证
P2PK	<pubKey> OP_CHECKSIG	<sig>	
P2PKH	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	<sig> <pubKey>	
P2MS	OP_<M> <pubKey>...<pubKey> OP_<N> OP_CHECKMULTISIG	OP_0 <sig> <sig> ...	
P2SH	OP_HASH160 <scriptHash> OP_EQUAL	<scriptParam>... <redeemScript>	
P2SH (2-of-3签名)	OP_HASH160 <multisig_scriptHash> OP_EQUAL	0 <sig1> <sig2> <OP_3 <pubKey1> <pubKey2> <pubKey3> OP_2 OP_CHECKMULTISIG>	
P2SH-P2WPKH	OP_HASH160 <scriptHash> OP_EQUAL	0 <pubKeyHash>	<sig> <pubKey>
P2SH-P2WSH	OP_HASH160 <scriptHash> OP_EQUAL	0 <redeemScriptHash>	<scriptParam>... <redeemScript>
P2SH-P2WSH (2-of-3签名)	OP_HASH160 <scriptHash> OP_EQUAL	0 <multisig_scriptHash>	0 <sig1> <sig2> <OP_3 <pubKey1> <pubKey2> <pubKey3> OP_2 OP_CHECKMULTISIG>
Native P2WPKH	OP_0 <pubKeyHash>		<sig> <pubKey>
Native P2WSH	OP_0 <scriptHash>		<scriptParam>... <redeemScript>
Native P2WSH (2-of-3签名)	OP_0 <multisig_scriptHash>		0 <sig1> <sig2> <OP_3 <pubKey1> <pubKey2> <pubKey3> OP_2 OP_CHECKMULTISIG>
P2TR (密钥路径消费)	OP_1 <pubKey>		<schnorr-sig>
P2TR (脚本路径消费)	OP_1 <pubKey>		<script> <controlBlock>

表中<sig>为采用 ECDSA 签名算法的签名，<schnorr-sig>是采用 Schnorr 签名算法的签名。

8. OP_RETURN 操作符

在输出脚本中使用 OP_RETURN 操作符，允许在比特币区块链上存储至多 80 个字节的任意数据，此操作符直接返回无效，所以这样的“UTXO”永远不能被花掉，所以金额为 0。输出脚本的格式如下：

```
OP_RETURN <至多 80 个字节的数据>
```

参考例子：txHash=6dfb16dd580698242bcfd8e433d557ed8c642272a368894de27292a8844a4e75，index=2

```
const bitcoin = require('bitcoinjs-lib');
const data = Buffer.from('Hello Blockchain!', 'utf8');
const psbt = new bitcoin.Psbt({ network: bitcoin.networks.testnet });

//添加输入
```

```
// 添加 OP_RETURN 输出
psbt.addOutput({
  script: bitcoin.script.compile([
    bitcoin.opcodes.OP_RETURN,
    data
  ]),
  value: 0 // OP_RETURN 输出必须为 0 值
});

//添加找零输出
```

4.5.3 交易剖析

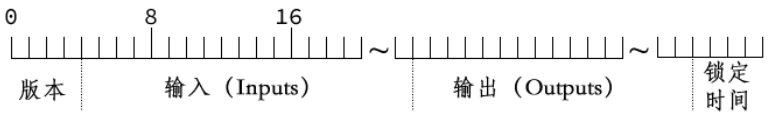
3.5.3.1 交易结构

发布后的交易就是已经序列化的一串字符，比如交易 ID 为 466200308696215bbc949d5141a49a4138ecdfdfaa2a8029c1f9bcecd1f96177 的交易，其序列化后的字符串是：

```
01000000000101eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da135698679268041c54a0100000000ffffffff02204e0000
000000002251203b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068f824010000000000160014752c1
65ea7be772b2c0acb7f4d6047ae6f4768e0141cf5efe2d8ef13ed0af21d4f4cb82422d6252d70324f6f4576b727b7d918e521c00b5
1be739df2f899c49dc267c0ad280aca6dab0d2fa2b42a45182fc83e817130100000000
```

交易的结构请参考图 4-19 所示。

传统交易：



隔离见证交易：

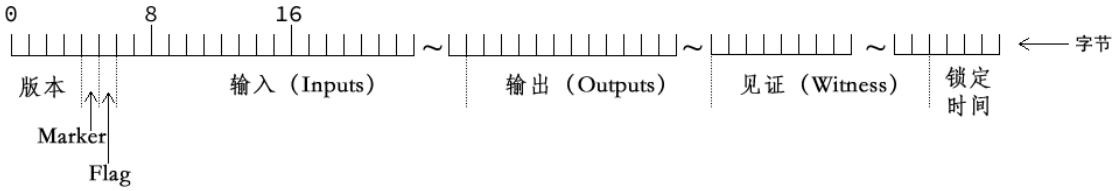


图 4-19 交易的结构

如果输入的地址类型为 P2PK、P2PKH 和 P2SH，那么就是传统交易；如果输入的地址类型为 P2WPKH、P2WSH 和 P2TR，那么就是隔离见证交易。如果一个交易中的多个输入中既包含传统地址（P2PK、P2PKH、P2SH）的输入，又包含隔离地址（P2WPKH、P2WSH、P2TR）的输入，那么此交易就是隔离见证交易，每个输入都要占据一个见证，只不过传统地址的输入对应的隔离见证为空值而输入脚本不能为空，而隔离见证地址的输入对应的隔离见证不能为空而输入脚本必须为空。输出中的地址不管是什么类型，输出脚本（或称为输出脚本）直接包含在各个输出中，即输出脚本不单独存放。

1. 版本（Version）字段：

占四个字节，且小端存储，第 1 版本就是数字 1，转换成十六进制的小端存储格式就是 0x01000000。遵循 BIP69 规范的软分叉时引入了第 2 版，即 0x02000000。BIP68 规范对输入部分中的顺序号字段增加了额外的约束条件，但这些约束条件仅适用于 2 或更高版本的交易，遵循版本 1 的交易不受影响。规范 BIP112 作为 BIP68 的补充——升级了一个操作码 OP_CHECKSEQUENCEVERIFY，在版本 1 中，此操作码执行时会失败。现在升级到版本 3 的呼声越来越大。

2. 标记（Marker）和标志（Flag）字段：

针对隔离见证，所以引了这两个字段，各占一个字节。如果交易中包含了隔离见证，那么 Marker 字段必须为 0（即 0x00），而 Flag 字段不能为 0，在当前的 P2P 协议里，固定为 1（即 0x01），其他值保留用于以后的协议升级。如果交易中没有包含隔离见证，那么 Marker 和 Flag 两个字段不存在。

3. 输入（Inputs）字段：

输入字段又包含更多的字段，请参考图 4-20 所示。

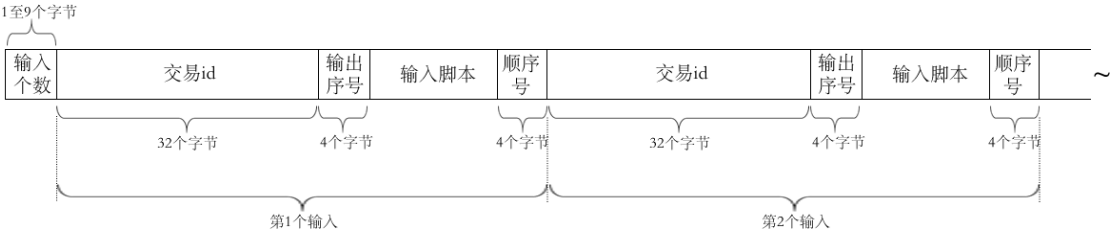


图 4-20 交易中输入的结构

一个交易中可包含多个输入（即花费多个 UTXO），因此要首先定义输入个数（count）。输入个数采用可变整型（varuint），定义见表 4-4。

表 4-4 可变整型

值范围	占用字节	格式
≥ 0 && ≤ 252 (0xfc)	1	uint8_t
≥ 253 && ≤ 0xffff	3	0xfd后面跟一个uint16_t类型的整数
≥ 0x10000 && ≤ 0xffffffff	5	0xfe后面跟一个uint32_t类型的整数
≥ 0x100000000 && ≤ 0xffffffffffffffff	9	0xff后面跟一个uint64_t类型的整数

一般输入个数很少超过 252 个以上的，这时等价于类型 uint8_t，只占用一个字节，比如 2 个输入，就是 0x02。当超过 252 且小于 0x10000 时占用三个字节，第一个字节为 0xfd，接下来是两个字节的 uint16_t 整形，比如有 1000 个输入，那么可变整数就是 0xfd03e8，小端存储就是 0xfde803。

对于每个具体的输入，其包含的字段为：

交易 id：共 32 个字节，如 466200308696215bbc949d5141a49a4138ecdfdfaa2a8029c1f9bcecd1f96177，每笔交易都有一个全

局唯一 id 号。

输出序号：每个交易可包含多个输出，每个输出有一个序号，从 0 开始连续编号，下一次转账时，这些输出就是以后其他交易的输入了，所以<交易 id+输出序号>唯一定位一个 UTXO。输出序号占用四个字节。

输入脚本：又包含两部分，首先是可变整数，指明后续那个真正输入脚本的长度；然后才是真正的输入脚本。如果本输入是见证隔离地址，那么输入脚本固定为 0x00。如：

```
6b483045022100a6cc4e8cd0847951a71fad3bc9b14f24d44ba59d19094e0a8cfa2580bb664b020220366060ea8203d766722ed0a02d1599b99d3c95b97dab8e41d3e4d3fe33a5706201210369e03e2c91f0badec46c9c903d9e9edae67c167b9ef9b550356ee791c9a40896
```

绿色字体部分就是输入脚本本身，长度 107 个字节，即 0x6b。

顺序号：固定占四个字节，这个字段的用途和含义随着时间的推移而发生了变化。一般固定为 0xffffffff。顺序号各种取值的意义见表 4-5。

表 4-5 顺序号的取值说明

顺序号取值	Relative Locktime	Replace-by-Fee	Locktime
0x00000001 <= Sequence <= 0x7fffffff	启用	显式启用	启用
0x80000000 <= Sequence <= 0xfffffffd	不启用	显式启用	启用
Sequence = 0xfffffffe	不启用	不显式启用	启用
Sequence = 0xffffffff	不启用	不显式启用	不启用

只有当全部输入的顺序号为 0xffffffff 时，全局的 Locktime 才被忽略。关于相对锁定时间（Relative Locktime）请参考 https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki，关于 RBF（Replace-bt-Fee）表示：当因为费用过低而使得交易久久未被矿工处理时，允许交易发起者重新发布类似交易以提高交易手续费，更详细的解锁请参考 https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki。

4. 输出（Output）字段：

其中输出个数字段请参考输入个数字段，见图 4-21 所示。

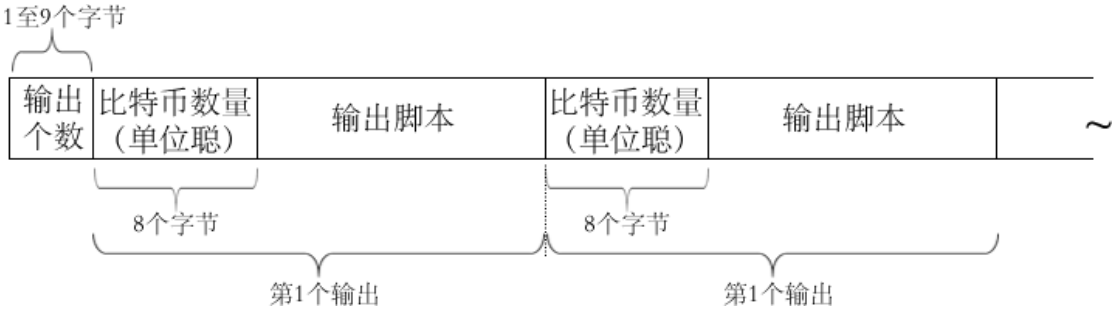


图 4-21 输出字段说明

比特币数量：以聪为单位，一个比特币等于 1 亿聪，本字段定义转账额度，可为 0。

输出脚本：输出脚本由两部分组成：<脚本长度><输出脚本本身>，解释类似上面的输入脚本，这里不再赘述。

5. 见证（Witness）字段：

每一个输入都对应一个见证，一个见证包含若干条款，因此见证的数量等于输入的个数。对于非见证隔离的地址来说，对应的见证固定为 0x00。具体参考图 4-22 所示。



图 4-22 见证结构

“条款数量”和“条款长度”都是可变整型。

5. 锁定时间（LockTime）字段：

交易体的最后一个字段就是锁定时间，占据 4 个字节。从区块高度 31,000 开始，禁止在区块中包含一个交易，除非满足以下规则之一：

(1) 交易指示应该有资格包含在任何块中，将其锁定时间设置为 0，即让交易尽快打包入链。

(2) 如果 LockTime<500,000,000，那么 LockTime 定义的是第几个区块，即本交易只能打包进第 LockTime 个及之后的区块中。例如 LockTime=123,456，那么本交易达能打包进 123,456 或任何后续的区块中。

(3) 如果 LockTime≥500,000,000，那么 LockTime 就是自 1970 年 01 月 01 日 T00:00 UTC 以来的秒数，即自 1970 年 01 月 01 日 T00:00 UTC 以来到挖矿时的秒数大于等于 LockTime 时，本交易才被打包入链。

例如自 1970 年 1 月 1 日 0 时 0 分 0 秒至 2025 年 3 月 3 日 10 点 11 分 11 秒流逝的秒数为 1,740,967,942，并让 LockTime 等于 1740967942，这样只有过了 2025 年 3 月 3 日 10 点 11 分 11 秒之后，矿工才会把交易打包入链。实操时一般指定 LockTime 为 0x00000000 即可，就是尽快打包入链让交易完成。因为被锁定时间影响的交易还没有打包入链，所以这些交易中的输入对应的 UTXO 还可以被另外的交易花费。

注意：当交易中的所有输入的顺序号均为 0xffffffff 时，锁定时间不起作用。

另外一种规定资金在某个时间点之后才能花费的方法就是通过输出脚本搞定——即在输出脚本中增加前缀“{生效时间} OP_CHECKLOCKTIMEVERIFY OP_DROP ”，其中{生效时间}为小端 Hex 存储格式。看下面的 js 代码：

```
//计算自 1970 年 1 月 1 日 0 时 0 分 0 秒到现在的秒数:
let s = Math.floor(Date.now()/1000);
console.log(s);
1740970399

//计算自 1970 年 1 月 1 日 0 时 0 分 0 秒至 2026 年 10 月 11 日 0 时 0 分 0 秒之间的秒数:
const t = '2026-10-11T00:00:00Z';
let m = Math.floor((new Date(t)).getTime()/1000);
console.log(m);
1791676800

//换算成小端 Hex 格式:
const bitcoin = require('bitcoinjs-lib');
let l = bitcoin.script.number.encode(m).toString('hex');                    //换成小端存储
//let h = m.toString(16).padStart(8, '0');                                  //转换为八位十六进制数
//let l = h.replace(/(.{2})/g, "$1 ").split(" ").reverse().join("");        //换成小端存储
```

```
console.log(`${m}(十进制) -> ${h}(大端十六进制) -> ${l}(小端十六进制)`);
1791676800(十进制) -> 6acad180(大端十六进制) -> 80d1ca6a(小端十六进制)

//构建 P2PKH 类型的输出脚本:
const bitcoin = require('bitcoinjs-lib');
const pubHash = '371c20fb2e9899338ce5e99908e64fd30b789313';                    //公钥哈希
const asm = `${l} OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 ${pubHash} OP_EQUALVERIFY OP_CHECKSIG`;
let script_hex = bitcoin.script.fromASM(asm).toString('hex');
console.log(`输出脚本: \n  汇编格式 - ${asm}\n   Hex 格式 - ${script_hex}`);
输出脚本:
  汇编格式 - 80d1ca6a OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 371c20fb2e9899338ce5e99908e64fd30b789313 OP_EQUALVERIFY OP_CHECKSIG
  Hex 格式 - 0480d1ca6ab17576a914371c20fb2e9899338ce5e99908e64fd30b78931388ac
```

即 0480d1ca6ab17576a914371c20fb2e9899338ce5e99908e64fd30b78931388ac 解构成如图 4-23 所示。

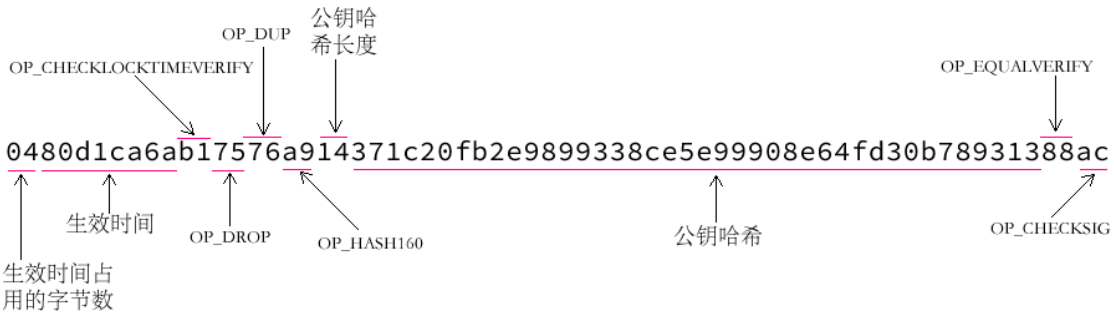


图 4-23 解构输出脚本

带有 OP_CHECKLOCKTIMEVERIFY 操作符的输出脚本所对应的 UTXO 显然已经在区块链上，前面讲过被 LockTime 影响的交易还没有进入区块链。在以后发起的交易中如果要成功花费此 UTXO（它的输出脚本包含 OP_CHECKLOCKTIMEVERIFY 操作符），那么必须同时满足两个条件：(1).本输入的顺序号小于 0xffffffff，通常为 0xfffffffe；(2). 0 ≤ {生效时间} ≤ LockTime < 500,000,000 或者 500,000,000 ≤ {生效时间} ≤ LockTime。OP_CHECKLOCKTIMEVERIFY 主要比较{生效时间}和 LockTime。

还有一种叫着“相对时间锁”的也可以规定 UTXO 在相对于现在过了多长时间之后才允许花费，即采用带有 OP_CHECKSEQUENCEVERIFY 操作符的输出脚本——在输出脚本中增加前缀“{生效时间} OP_CHECKSEQUENCEVERIFY OP_DROP”，其中{生效时间}为小端 Hex 存储格式。它主要比较{生效时间}和输入的{顺序号}，输入的顺序号是一个 32 位二进制的整数，各位定义如图 4-24 所示。

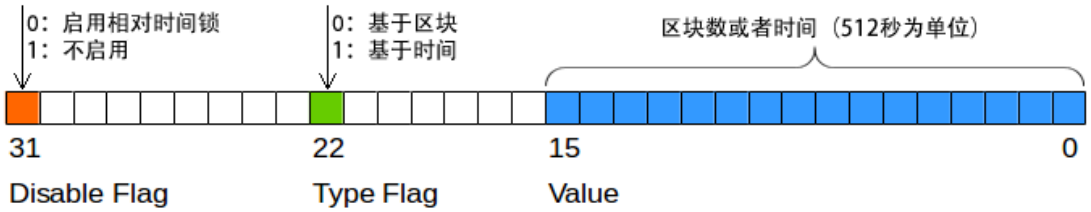


图 14-24 顺序号各位含义

如果最高位为 1，那么相对时间锁不起作用。最低两个字节存储具体的相对“时间”值 value，如果：(1) 第 23 位是 0，那么 value 就是区块数，即从本 UTXO 所在的区块开始，要过至少 value 个区块后才允许花费此 UTXO；(2) 第 23 位是 1，那么 value 就是时间，即从本 UTXO 所在的区块开始，要过至少 value*512 秒后才允许花费此 UTXO。同时还要满足条件：value 大于等于{生效时间}的最低两个字节且顺序号和{生效时间}的第 23 位的值相同。见表 4-6。

表 4-6 时间锁说明

类型	说明
Timelock, Sequence (>= 0x80000000)	限制 Tx 打包上链的绝对时间锁
Timelock, Sequence (<= 0x7fffffff)	限制 Tx 打包上链的相对时间锁
OP_CHECKLOCKTIMEVERIFY Sequence	限制 UTXO 消费的绝对时间锁
OP_CHECKSEQUENCEVERIFY Sequence	限制 UTXO 消费的相对时间锁

知道交易结构之后，我们现在尝试手工来解构下面的交易：

```
010000000001 01eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da135698679268041c54a0100000000ffffffff02204e0000
000000002251203b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068f8240100000000001600147752c1
65ea7be772b2c0acb7f4d6047ae6f4768e0141cf5efe2d8ef13ed0af21d4f4cb82422d6252d70324f6f4576b727b7d918e521c00b5
1be739df2f899c49dc267c0ad280aca6dab0d2fa2b42a45182fc83e817130100000000
```

首先是四个字节的版本，即 01000000，恢复正常顺序就是 0x00000001（版本 1）。接下来是一个字节的 Marker，即 0x00，表明交易中包含隔离见证。下面就是一个字节的 Flag，即 0x01。接下来就是可变整型的输入个数，这里为 0x01，即只有一个输入。然后是 32 个字节的交易 id，即 eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da135698679268041c54a，恢复正常顺序就是 0x4ac541802679866935a19d4f40728bb89204d0cac90d85f3a51a19278fe33aeb，查询网址 https://api.blockcypher.com/v1/btc/main/txs/4ac541802679866935a19d4f40728bb89204d0cac90d85f3a51a19278fe33aeb 可得到第二个输出的输出脚本为 5120225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796，采用下面的命令反编译它：

```
const bitcoin = require('bitcoinjs-lib');
let output_hex = '5120225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796';
let output_asm = bitcoin.script.toASM(Buffer.from(output_hex, 'hex'));                    //反汇编
let taproot_publickey = '225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796';
let p2tr_address = bitcoin.address.toBech32(Buffer.from(taproot_publickey, 'hex'), 1, 'bc'); //从公钥推导地址
console.log(`${output_asm}\n${p2tr_address}`);
'OP_1 225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796'
'bc1pyfw56zu5vsq0ulu9kytasgw4xwnm3eysll6tfdz8d9gtht97k7tqxsz78n'
```

OP_1 (0x51) 就是版本，显然后面的 225d4d0b946400fe7f85b117d821d533a7b8e490fff4b4b4476950bbacbeb796 就是 Taproot 公钥，从此公钥推导的钱包地址为 bc1pyfw56zu5vsq0ulu9kytasgw4xwnm3eysll6tfdz8d9gtht97k7tqxsz78n，这是个 P2TR 类型的地址。后面的隔离见证如果走的是密钥路径验证，那么必须提供本公钥所对应的那个私钥的签名才能解锁。

接下来就是四个字节的输出序号，即 01000000（0x00000001）。 接下来为 00，表明本输入为隔离见证，输入脚本放在隔离见证结构中。接下来为四个字节的顺序号，即 ffffffff（0xffffffff）。

在接下来就是输出部分了。输出个数 02 (0x02)，表明本交易有两个输出。首先是第一个输出的比特币数量（八个字节），即 204e000000000000，转换成十六进制就是 0x0000000000004e20，即 20000 聪（十进制）。接下来为输出脚本的长度 22，即 0x22，等于十进制的 34。显然接下来 34 个字节就是输出脚本了，即 51203b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068（反编译为 OP_1 3b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e0706，又是一个 Taproot 公钥）。然后就是第 2 个输出的

发生额 f824010000000000，即 0x0000000000124f8，等价于十进制的 75000 聪。接下来的 16 就是输出脚本的长度，即输出脚本的长度为 22 个字节。接下来 22 个字节的输出脚本 00147752c165ea7be772b2c0acb7f4d6047ae6f4768e（反编译为 OP_0 7752c165ea7be772b2c0acb7f4d6047ae6f4768e，版本为 OP_0，因此本输出为 P2WPKH 类型的钱包地址）。这样两个输出就解构完了。

再来解构隔离见证部分。由于只有一个输入，所以也只有一个隔离见证。首先条款数量 01，由此可知是走 Taproot 的密钥路径验证，条款长度 41（十六进制），即 65 个字节，因此接下来的 cf5efe2d8ef13ed0af21d4f4cb82422d6252d70324f6f4576b727b7d918e521c00b51be739df2f899c49dc267c0ad280aca6dab0d2fa2b42a45182fc83e8171301 就是见证条款（即 Schnorr 签名）。最后的 00000000 就是锁定时间。解构结果见表 4-7。

表 4-7 解构交易

版本				01000000 00000001	4个字节
Marker				00	1个字节
Flag				01	1个字节
输入个数				01	可变整型（1/3/5/9个字节）
第一个输入	交易id			eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da135698679268041c54a 4ac54180267986935a19d4f40728bb89204d0cac90d85f3a51a19278fe33aeb（正常序）	32个字节
	输出序号			01000000 00000001	4个字节
	输入脚本长度			00	可变整型（1/3/5/9个字节）
	顺序号			ffffffff	4个字节
输出个数				02	可变整型（1/3/5/9个字节）
第一个输出	金额			204e000000000000 0000000000004e20（20000聪）	8个字节
	输出脚本长度			22 （十进制34）	可变整型（1/3/5/9个字节）
	输出脚本			51203b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068 OP_1 3b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068（OP_1 <x-only-pubKey>）	N/A
第二个输出	金额			f824010000000000 0000000000124f8（75000聪）	8个字节
	输出脚本长度			16 （十进制22）	可变整型（1/3/5/9个字节）
	输出脚本			00147752c165ea7be772b2c0acb7f4d6047ae6f4768e OP_0 7752c165ea7be772b2c0acb7f4d6047ae6f4768e（OP_0 <pubKeyHash>）	N/A
见证	见证1	条款数量		01	可变整型（1/3/5/9个字节）
		条款1	条款长度	41 （十进制65）	可变整型（1/3/5/9个字节）
			条款内容	cf5efe2d8ef13ed0af21d4f4cb82422d6252d70324f6f4576b727b7d918e521c00b51be 739df2f899c49dc267c0ad280aca6dab0d2fa2b42a45182fc83e8171301 （Schnorr签名）	N/A
锁住时间				00000000	4个字节

再来解构一个输入包含传统地址类型和原生隔离地址类型的交易。

以后补上

3.5.3.2 交易 id

TxId（或者称 TxHash）由 [nVersion][txins][txouts][nLockTime] 这 4 部分内容拼接在一起，然后进行 两次 SHA256 运算后得到。引入隔离见证后，TxId 的生成规则没有变化。对于表 4-7 所示的交易，可用下面的代码计算交易 id：

```
const bitcoin = require('bitcoinjs-lib');

const nVersion = '01000000';
const txins = '01eb3ae38f27191aa5f3850dc9cad00492b88b72404f9da135698679268041c54a0100000000ffffffff';
const txouts  =
'02204e0000000000002251203b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e07068f8240100000000001600147752
c165ea7be772b2c0acb7f4d6047ae6f4768e'
const nLockTime = '00000000';

const content = Buffer.from(nVersion+txins+txouts+nLockTime,'hex');
const id_tmp = bitcoin.crypto.sha256(bitcoin.crypto.sha256(content)).toString('hex');
```

```
const TxId = id_tmp.replace(/(.{2})/g, "$1 ").split(" ").reverse().join("");                    //转换为小端存储
console.log(TxI);
466200308696215bbc949d5141a49a4138ecdfdfaa2a8029c1f9bcecd1f96177                                 //得到的 TxHash
```

3.5.3.3 交易签名

比特币交易中大量采用数字签名，P2TR 地址类型的见证数据中的签名采用 Schnorr 签名算法，其他地址类型统一采用 ECDSA 签名算法。

在一笔交易中，允许各个输入的签名来自不同的私钥，同一个输入允许多个采用不同 SIGHASH 标志的签名，关于 SIGHASH 标志的解释请看下文。

ECDSA 签名

先从 https://blockchain.info/rawtx/c6e81c4a315d7eed40cb32b2558f4b142d37959f7e8eb3eb5c43fdf2931cca42?format=hex 下载一个原始的交易：

```
010000000110df9fa370de9dfbfcf6da91863f891659e6e7705c28f88a453723d272ddb98a000000008a47304402205aaaa23d7ba69eb6d233cd31ab242d09b6ab1a325b8e406e55d9e5575832734102204eec8e7669241249b4f597d9dfe34a9a25cc61f0f55f1c63b8f22df47371836d014104b6b6509ee40cb6947af267d3cd7da306dbb4fb2691fc30c1e76ede0b3e533980c87e8a4d382f80b2338fcb15460af964428662aa144e02e52047637e7900b18cfeffffff012802000000000000001976a914884377972c0df175fc1ef878a490d701c0fd195e88ac18090800
```

解构后见表 4-8 所示。

表 4-8 交易解构

版本		01000000 00000001	4个字节
输入个数		01	可变整型（1/3/5/9个字节）
第一个输入	交易id	10df9fa370de9dfbfcf6da91863f891659e6e7705c28f88a453723d272ddb98a 8ab9dd72d22337458af8285c70e7e65916893f8691daf6fcfb9dde70a39fdf10（正常序）	32个字节
	输出序号	00000000 00000000	4个字节
	脚本长度	8a (十进制138)	可变整型（1/3/5/9个字节）
	输入脚本	47304402205aaaa23d7ba69eb6d233cd31ab242d09b6ab1a325b8e406e55d9e5575832734102204eec8e7669241249b4f597d9dfe34a9a25cc61f0f55f1c63b8f22df473718366d014104b6b6509ee40cb6947af267d3cd7da306dbb4fb2691fc30c1e76ede0b3e533980c87e8a4d382f80b2338fcb15460af964428662aa144e02e52047637e7900b18c	N/A
	顺序号	feffffff fffffffe（正常序）	4个字节
输出个数		01	可变整型（1/3/5/9个字节）
第一个输出	金额	2802000000000000 0000000000000228（正常序。等于十进制552聪）	8个字节
	脚本长度	19 （十进制25）	可变整型（1/3/5/9个字节）
	输出脚本	76a914884377972c0df175fc1ef878a490d701c0fd195e88ac OP_DUP OP_HASH160 884377972c0df175fc1ef878a490d701c0fd195e OP_EQUALVERIFY OP_CHECKSIG	N/A
锁住时间		18090800 00080918(正常序。等于十进制526616，小于500000000，因此交易只包含在大于526616的区块中)	4个字节

输入引用了 UTXO（交易 id: 8ab9dd72d22337458af8285c70e7e65916893f8691daf6fcfb9dde70a39fdf10）、输出序号: 0），浏览 https://api.blockcypher.com/v1/btc/main/txs/8ab9dd72d22337458af8285c70e7e65916893f8691daf6fcfb9dde70a39fdf10 可获得此 UTXO 的输出脚本为：

```
76a914e06f1b3b23fe35386c31e5c6dc43b4fc4c025c3988ac                                  //hex 格式
OP_DUP OP_HASH160 e06f1b3b23fe35386c31e5c6dc43b4fc4c025c39 OP_EQUALVERIFY OP_CHECKSIG  //汇编格式
```

由此可见这是个 P2PKH 类型的输出。因此输入脚本格式必须为<sig> <publickey>，表 4-8 中的输入脚本进一步解构成表

4-9。

表 4-9 解构输入脚本

签名长度		47 （十进制71）
签名(DER)	Sequence	30
	Length	44
	Integer	02
	Length	20
	R	5aaaa23d7ba69eb6d233cd31ab242d09b6ab1a325b8e406e55d9e55758327341
	Integer	02
	Length	20
	S	4eec8e7669241249b4f597d9dfe34a9a25cc61f0f55f1c63b8f22df47371836d
签名类型		01
公钥长度		41 （十进制65）
公钥	前缀	04 （04-非压缩公钥、02-y坐标为偶数的压缩公钥、03-y坐标为奇数的压缩公钥）
	X	b6b6509ee40cb6947af267d3cd7da306dbb4fb2691fc30c1e76ede0b3e533980
	Y	c87e8a4d382f80b2338fcb15460af964428662aa144e02e52047637e7900b18c

下面我们再来看看表中的签名是怎么来的。比特币数字签名过程：

$$\text{Sig} = F_{sig}\left(F_{hash}(m), x\right)$$

其中：

x：用来签名的私钥。

m：被签名的内容。对于比特币，为交易的部分字段。由 SIGHASH 标志规定 m 具体包含交易中的哪些字段。

F_{hash}：哈希函数，哈希值是 256 位二进制数。

F_{sig}：签名函数，对于比特币，P2TR 采用 Schnorr 签名算法，其他地址类型采用 ECDSA 签名算法。

Sig：签名的结果

SIGHASH 的值参考表 4-10。

表 4-10 SIGHASH 取值

签名类型	**值**	**说明**
SIGHASH_ALL	0x01	被签名的内容包含所有的输入和输出。这是默认值
SIGHASH_NONE	0x02	包含全部输入，不包含任何输出。由于输出没有被签名，所以允许他人修改输出，这适合于创建特定金额的“持票支票”或“空白支票”。
SIGHASH_SINGLE	0x03	包含全部输入和那个输出序号与本输入序号相同的输出
SIGHASH_ALL | SIGHASH_ANYONECANPAY	0x81	只包含一个输入和全部输出
SIGHASH_NONE | SIGHASH_ANYONECANPAY	0x82	只包含一个输入。这适合于将金额特别微小的 UTXO 捐赠给他人，在他们想要的时候进行聚合和支出。
SIGHASH_SINGLE | SIGHASH_ANYONECANPAY	0x83	包含一个输入和那个输出序号与本输入序号相同的输出
SIGHASH_ALL | SIGHASH_ANYPREVOUT	0x41	展望中.....
SIGHASH_NONE | SIGHASH_ANYPREVOUT	0x42	
SIGHASH_SINGLE | SIGHASH_ANYPREVOUT	0x43	

SIGHASH_ALL | SIGHASH_ANYPREVOUTANYSCRIPT	0xc1	
SIGHASH_NONE | SIGHASH_ANYPREVOUTANYSCRIPT	0xc2	
SIGHASH_SINGLE | SIGHASH_ANYPREVOUTANYSCRIPT	0xc3	

针对签名类型 SIGHASH_ALL，下面举例进一步说明：现在假如有一个包含两个输入和一个输出的交易，两个输入需要签名，签名过程参见图 4-25 所示。

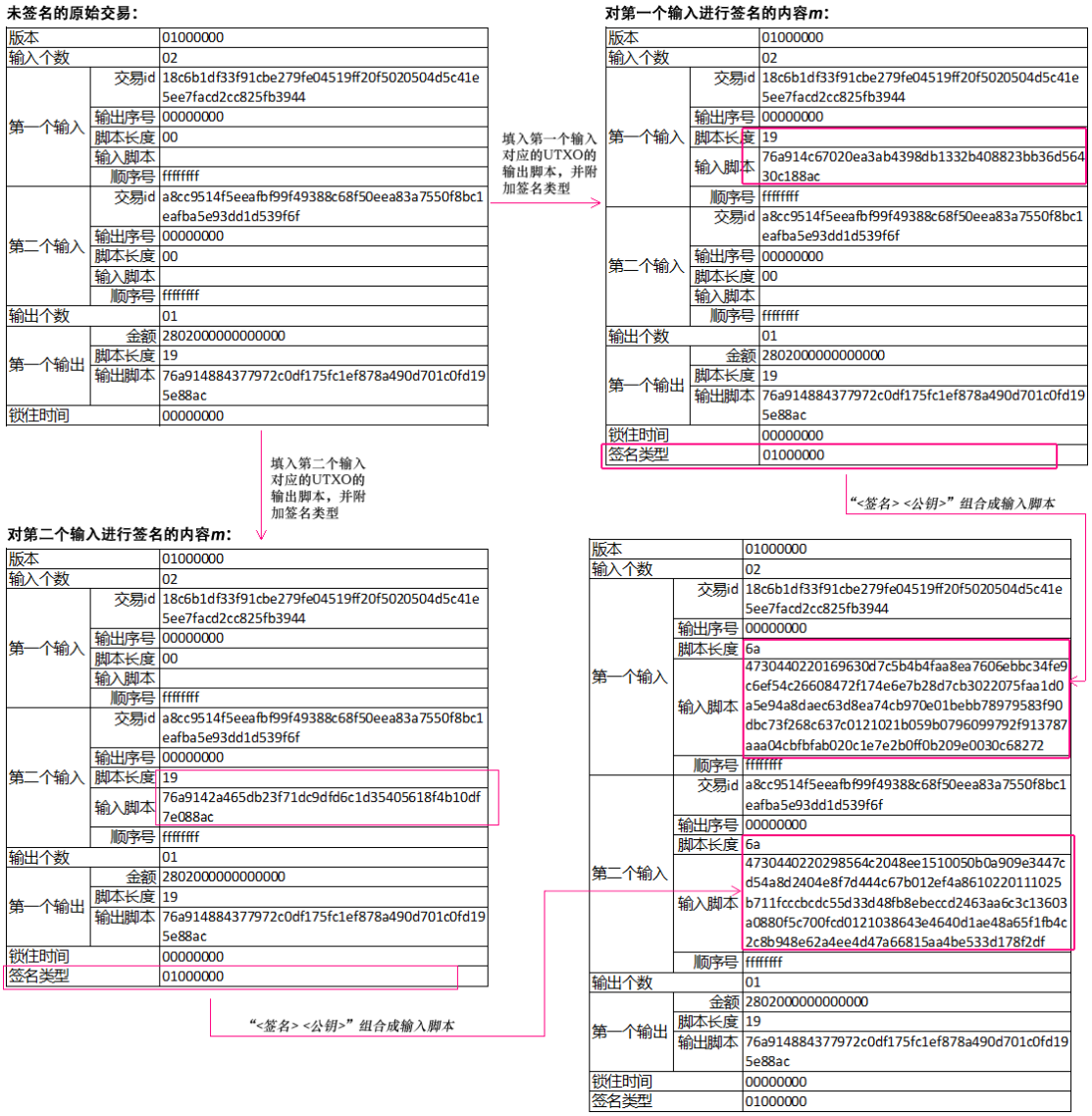


图 4-25 SIGHASH_ALL 类型的签名过程

图中未签名的原始交易为：

```
0100000002218c6b1df33f91cbe279fe04519ff20f5020504d5c41e5ee7facd2cc825fb394400000000ffffffffa8cc9514f5eeafbf99f49388c68f50eea83a7550f8bc1eafba5e93dd1d539f6f0000000000ffffffff0128020000000000001976a914884377972c0df175fc1ef878a490d701c0fd195e88ac00000000
```

第一个输入花费的 UTXO 的输出脚本为：

```
76a914c67020ea3ab4398db1332b408823bb36d56430c188ac
```

第二个输入花费的 UTXO 的输出脚本为：

```
76a9142a465db23f71dc9dfd6c1d35405618f4b10df7e088ac
```

对于第一个输入，待签名的内容 m 为：

```
010000000218c6b1df33f91cbe279fe04519ff20f5020504d5c41e5ee7facd2cc825fb39440000000000019
76a914c67020ea3ab4398db1332b408823bb36d56430c188acffffffffa8cc9514f5eeafbf99f49388c68f50
eea83a7550f8bc1eafba5e93dd1d539f6f0000000000ffffffff012802000000000000001976a914884377
972c0df175fc1ef878a490d701c0fd195e88ac0000000001000000
```

对于第二个输入，待签名的内容 m 为：

```
010000000218c6b1df33f91cbe279fe04519ff20f5020504d5c41e5ee7facd2cc825fb39440000000000000
ffffffffa8cc9514f5eeafbf99f49388c68f50eea83a7550f8bc1eafba5e93dd1d539f6f000000000019
76a9142a465db23f71dc9dfd6c1d35405618f4b10df7e088acffffffff012802000000000000001976a914
884377972c0df175fc1ef878a490d701c0fd195e88ac0000000001000000
```

下面我们用代码来计算被签内容 m 的哈希值、签名以及输入脚本：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');
const { ECPairFactory } = require('ecpair');
const ECPair = ECPairFactory(ecc);
const network = bitcoin.networks.bitcoin;
const sigHash_type = bitcoin.Transaction.SIGHASH_ALL;

const wif = 'L4AgDeoTPbdRd4S7X9UhXaNtyxZhDuj7EoUxPUndDYkvfmHjR8Cd';                    //用于签名的私
钥

const keyPair = ECPair.fromWIF(wif);
const input1_m =
'010000000218c6b1df33f91cbe279fe04519ff20f5020504d5c41e5ee7facd2cc825fb394400000000000197
6a914c67020ea3ab4398db1332b408823bb36d56430c188acffffffffa8cc9514f5eeafbf99f49388c68f50
eea83a7550f8bc1eafba5e93dd1d539f6f0000000000ffffffff012802000000000000001976a914884377
972c0df175fc1ef878a490d701c0fd195e88ac0000000001000000';

const hash1 = bitcoin.crypto.hash256(Buffer.from(input1_m, 'hex'));
const hash2 = bitcoin.crypto.hash256(hash1);

const signatrue1 = ecc.sign(hash2, keyPair.privateKey);                              //ECDSA 签
名
console.log("r:"+Buffer.from(signatrue1.slice(0,32)).toString('hex')+"\ns:"+Buffer.from(signatrue1.slice(3
2)).toString('hex'));
r:6eabc8529418d3cba0a8deda8de3b1a0b38582f930ba2527390fba379ad7d87b
s:77de5573d30cb0e1081da95f1f55fb30d7dc62ee569eea8fd893f8a72d75b773

//将 ECDSA 签名编码为 DER 格式：
const sigDER = bitcoin.script.signature.encode(Buffer.from(signatrue1), sigHash_type).toString('hex');
console.log(sigDER);
304402206eabc8529418d3cba0a8deda8de3b1a0b38582f930ba2527390fba379ad7d87b022077de5573d30cb0e1081da95f1f55fb
30d7dc62ee569eea8fd893f8a72d75b77301

//生成输入脚本：
let input_script = bitcoin.script.fromASM(`${sigDER}
${keyPair.publicKey.toString('hex')}`).toString('hex');
input_script = (input_script.length/2).toString(16)+ input_script;
console.log(input_script);
6a47304402206eabc8529418d3cba0a8deda8de3b1a0b38582f930ba2527390fba379ad7d87b022077de5573d30cb0e1081da95f1f
55fb30d7dc62ee569eea8fd893f8a72d75b7730121026e95a29dc6732ee49d74404c318e36e32bfc45fca4918037ed416b252bce57
d9
```

可采用类型的方法完成第二个输入的签名。

对于 P2WPKH 类型的输入，签名有不同，具体参见 https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki#Specification。

4.5.4 脚本编程

能成功转账比特币的条件是输入脚本能正确解锁输出脚本，也就是先执行输入脚本，然后再执行输出脚本，执行成功则表明完成解锁，允许转账（即花钱）。脚本是在堆栈引擎中被执行的。本小节介绍编写脚本的语言，脚本程序中包含数字和操作符，遇到数字入栈，遇到操作符则处理栈内的数字。比特币脚本语言的发展经历了三个阶段，分别是传统脚本语言、隔离见证脚本语言和 Tapscript（用于 Taproot 脚本路径花费）。

操作符参见下面各表。

表 4-11 常量

操作符	操作码	十六进制	输入	输出	描述
OP_0, OP_FALSE	0	0x00		(empty value)	入栈一个空的字节数组
N/A	1-75	0x01-0x4b	(special)	data	入栈后续的 n 个字节的数据,n=1-75
OP_PUSHDATA1	76	0x4c	(special)	data	入栈后续的 1 个字节并转化为整数 n，然后再入栈接下来的 n 个字节的内容。例如 OP_PUSHDATA1 9 e78e8be889afe6988e 之后栈顶内容为 e78e8be889afe6988e
OP_PUSHDATA2	77	0x4d	(special)	data	读取后续的 2 个字节并转换为整数 n，然后把接下来的 n 个字节的内容以小端格式压栈。如果入栈的内容长度小于 76，直接用 OP_PUSHBYTES_*n*（简写为 *n*）操作符，长度在[76,255]内，建议采用 OP_PUSHDATA1
OP_PUSHDATA4	78	0x4e	(special)	data	读取后续的 4 个字节并转换为整数 n，然后把接下来的 n 个字节的内容以小端格式压栈。实际上入栈的用户数据至多 520 个字节，所以本操作符从为使用过
OP_1NEGATE	79	0x4f		-1	入栈-1
OP_1, OP_TRUE	81	0x51		1	入栈 1
OP_2-OP_16	82-96	0x52-0x60		2-16	入栈数字 2-16

表 4-12 流控

操作符	操作码	十六进制	输入	输出	描述
OP_NOP	97	0x61			什么都不做
OP_IF	99	0x63	<expression> if [statements_1] [else [statements_2]]* endif		如果栈顶元素不是 0，就执行 statements_1。同时弹出栈顶元素。
OP_NOTIF	100	0x64	<expression> notif [statements_1] [else [statements_2]]* endif		如果栈顶元素为假，就执行 statements。同时弹出栈顶元素
OP_ELSE	103	0x67	<expression> if [statements_1] [else [statements_2]]* endif		否则执行 statements_2，要与 OP_IF 或 OP_NOTIF 配合使用。
OP_ENDIF	104	0x68	<expression> if [statements_1] [else [statements_2]]* endif		条件语句块的结尾，要与 OP_IF 或 OP_NOTIF 配合使用。
OP_VERIFY	105	0x69	True / false	Nothing / *fail*	如果栈顶元素非真，则交易无效。栈顶元素出栈。
OP_RETURN	106	0x6a	Nothing	fail	交易作废。常用于把存储数据到比特币链上

表 4-13 堆栈操作

操作符	操作码	十六进制	输入	输出	描述
OP_TOALTSTACK	107	0x6b	x1	(alt)x1	把主栈顶部元素移到副栈。副栈完全独立于主栈，可用于存储临时数据
OP_FROMALTSTACK	108	0x6c	(alt)x1	x1	把主栈顶部元素移到副栈

OP_IFDUP	115	0x73	x	x / x x	如果栈顶元素非 0，就复制它到栈顶
OP_DEPTH	116	0x74	Nothing	<Stack size>	计算站内元素个数，并入栈计算结果
OP_DROP	117	0x75	x	Nothing	弹出栈顶元素
OP_DUP	118	0x76	x	x x	复制栈顶元素并入栈
OP_NIP	119	0x77	x1 x2	x2	移除栈顶元素的下面那个元素
OP_OVER	120	0x78	x1 x2	x1 x2 x1	把栈顶元素的下面那个元素复制到栈顶
OP_PICK	121	0x79	xn ... x2 x1 x0 <n>	xn ... x2 x1 x0 xn	从栈顶元素往下数的第 n+1 个元素复制到栈顶
OP_ROLL	122	0x7a	xn ... x2 x1 x0 <n>	... x2 x1 x0 xn	从栈顶元素往下数的第 n+1 个元素移到栈顶
OP_ROT	123	0x7b	x1 x2 x3	x2 x3 x1	把第 3 个元素移到栈顶
OP_SWAP	124	0x7c	x1 x2	x2 x1	交换顶部两个元素
OP_TUCK	125	0x7d	x1 x2	x2 x1 x2	复制栈顶元素到第 2 个元素的下面
OP_2DROP	109	0x6d	x1 x2	Nothing	弹出顶部两个元素
OP_2DUP	110	0x6e	x1 x2	x1 x2 x1 x2	复制顶部两个元素到栈顶
OP_3DUP	111	0x6f	x1 x2 x3	x1 x2 x3 x1 x2 x3	复制顶部三个元素到栈顶
OP_2OVER	112	0x70	x1 x2 x3 x4	x1 x2 x3 x4 x1 x2	复制第 3、4 个元素到栈顶
OP_2ROT	113	0x71	x1 x2 x3 x4 x5 x6	x3 x4 x5 x6 x1 x2	移动第 5、6 个元素到栈顶
OP_2SWAP	114	0x72	x1 x2 x3 x4	x3 x4 x1 x2	交换顶部两对元素的位置

表 4-14 求长度

操作符	操作码	十六进制	输入	输出	描述
OP_SIZE	130	0x82	in	in size	计算栈顶元素的长度（字符串），并把计算出来的长度压栈。栈顶元素不弹出。

表 4-15 按位逻辑运算

操作符	操作码	十六进制	输入	输出	描述
OP_EQUAL	135	0x87	x1 x2	True / false	出栈两个元素并比较，如果相等则入栈 1，否则入栈 0
OP_EQUALVERIFY	136	0x88	x1 x2	Nothing / *fail*	等价于先执行 OP_EQUAL，再执行 OP_VERIFY

表 4-16 算数运算

操作符	操作码	十六进制	输入	输出	描述
OP_1ADD	139	0x8b	in	out	栈顶元素加 1
OP_1SUB	140	0x8c	in	out	栈顶元素减 1
OP_NEGATE	143	0x8f	in	out	栈顶元素取反
OP_ABS	144	0x90	in	out	栈顶元素取绝对值
OP_NOT	145	0x91	in	out	如果栈顶元素为 0 或 1，就取反，否则置为 0
OP_0NOTEQUAL	146	0x92	in	out	如果栈顶元素非 0 就置为 1
OP_ADD	147	0x93	a b	out	弹出顶部两个元素，相加后结果入栈
OP_SUB	148	0x94	a b	out	弹出顶部两个元素并相减（那个栈顶元素为减数），结果压栈
OP_BOOLAND	154	0x9a	a b	out	出栈两个元素，如果两个元素非 0 则入栈 1，否则入栈 0
OP_BOOLOR	155	0x9b	a b	out	出栈两个元素，如果两个元素为 0 则入栈 0，否则入栈 1
OP_NUMEQUAL	156	0x9c	a b	out	出栈两个元素（数字），如果相等则入栈 1，否则入栈 0
OP_NUMEQUALVERIFY	157	0x9d	a b	Nothing / *fail*	等价于先执行 OP_NUMEQUAL，再执行 OP_VERIFY
OP_NUMNOTEQUAL	158	0x9e	a b	out	出栈两个元素并比较，如果不等则入栈 1，否则入栈 0
OP_LESSTHAN	159	0x9f	a b	out	出栈两个元素并比较，如果下面的元素小于上面的，则入栈 1，否则入栈 0
OP_GREATERTHAN	160	0xa0	a b	out	出栈两个元素并比较，如果下面的元素大于上面的，则入栈 1，否则入栈 0
OP_LESSTHANOREQUAL	161	0xa1	a b	out	出栈两个元素并比较，如果下面的元素小于等于上面的，则入栈 1，否则入栈 0
OP_GREATERTHANOREQUAL	162	0xa2	a b	out	出栈两个元素并比较，如果下面的元素大于等于上面的，则入栈 1，否则入栈 0
OP_MIN	163	0xa3	a b	out	出栈两个元素，并入栈最小的那个元素
OP_MAX	164	0xa4	a b	out	出栈两个元素，并入栈最大的那个元素
OP_WITHIN	165	0xa5	x min max	out	出栈三个元素，如果 x 落在[min,max)内则入栈 1，否则入栈 0

表 4-17 签名验证操作

操作符	操作码	十六进制	输入	输出	描述
OP_RIPEMD160	166	0xa6	in	hash	对输入进行 RIPEMD-160 运算，并把结果入栈
OP_SHA1	167	0xa7	in	hash	对输入进行 SHA-1 运算，并把结果入栈
OP_SHA256	168	0xa8	in	hash	对输入进行 SHA-256 运算，并把结果入栈
OP_HASH160	169	0xa9	in	hash	先作 SHA-256 运算，然后再作 RIPEMD-160 运算，把结果入栈
OP_HASH256	170	0xaa	in	hash	进行两次 SHA-256 运算
OP_CODESEPARATOR	171	0xab	Nothing	Nothing	被签名的内容包含交易的大部分字段（如输入、输出等），输出脚本中的 OP_CODESEPARATOR 起到分隔脚本的作用，在执行后续的验证签名操作符时，OP_CODESEPARATOR 之前的部分被排除在被签名的内容之外。不过后来引入了 sighash 标志能更好地控制交易中的哪些字段参与签名
OP_CHECKSIG	172	0xac	sig pubkey	True / false	验证签名，如果验证成功则入栈 1，否则入栈 0
OP_CHECKSIGVERIFY	173	0xad	sig pubkey	Nothing / *fail*	相当于先执行 OP_CHECKSIG，再执行 OP_VERIFY
OP_CHECKMULTISIG	174	0xae	x sig1 sig2 ... <number of signatures> pub1 pub2 <number of public keys>	True / False	多签验证。 对于 N-of-M 多签，脚本为：*0 sig1 sig2 ... sigN N pub1 pub2 ... pubM M*。执行过程是这样的：从左至右找一个公钥 *pubi* 验证 *sig1*，再从第 *i+1* 开始找一个公钥 *pubj* 验证 *sig2*，依次类推，直到全部签名验证通过，则入栈 1，否则入栈 0。脚本前增加 0 是为了解决一个 bug。
OP_CHECKMULTISIGVERIFY	175	0xaf	x sig1 sig2 ... <number of signatures> pub1 pub2 ... <number of public keys>	Nothing / *fail*	相当于先执行 OP_CHECKMULTISIG，再执行 OP_VERIFY
OP_CHECKSIGADD	186	0xba	sig n pub	out	签名验证通过，则入栈 n+1，如果签名为 OP_0，则入栈 n，其他情况脚本执行失败。本操作符只是 tapscript 新增的，专门用于 P2TR 类型的脚本路径花费 UTXO，它取代了传统的 OP_CHECKMULTISIG 和 OP_CHECKMULTISIGVERIFY

表 4-18 时间锁操作符

操作符	操作码	十六进制	输入	输出	描述
OP_CHECKLOCKTIMEVERIFY (以前叫 OP_NOP2)	177	0xb1	x	x / *fail*	如果满足下面的条件之一，交易无效：1. 栈顶元素大于交易的 nLockTime，2. 空栈，3. 栈顶元素为负数，4. 栈顶元素小于 500000000 而 nLockTime 大于等于 500000000，5.输入的 nSequence 等于 0xffffffff。其他情况脚本继续执行，此时相当于 OP_NOP 操作符。不弹出栈顶元素。
OP_CHECKSEQUENCEVERIFY (以前叫 OP_NOP3)	178	0xb2	x	x / *fail*	如果相对锁时间（输入的 nSequence 字段）小于栈顶元素，则交易无效。更详细的介绍参考 BIP 0112。不弹出栈顶元素。

对于 OP_CHECKSIGADD 操作符，举一个实现 2-of-3 多签的例子：

```
输出脚本：key1 OP_CHECKSIG key2 OP_CHECKSIGADD key3 OP_CHECKSIGADD OP_2 OP_GREATERTHANOREQUAL
输入脚本：sig3 sig2 sig1
执行时先合并：
sig3 sig2 sig1 key1 OP_CHECKSIG key2 OP_CHECKSIGADD key3 OP_CHECKSIGADD OP_2 OP_GREATERTHANOREQUAL
```

执行过程大致如下：

首先入栈 sig3 sig2 sig1 key1 → 接下来遇到 OP_CHECKSIG 操作符，则弹出栈顶两个元素，用 key1 验证签名 sig1，如果验证成功则入栈 1，否则入栈 0 → 入栈 key2 后的站内元素（假设刚才签名验证成功）：sig3 sig2 1 key2 → 遇到操作符 OP_CHECKSIGADD，弹出栈顶三个元素，用 key2 验证 sig2，如果验证成功则弹出的第 2 个元素加 1 后再入栈，否则加 0 后入栈 → 入栈 key3 后的站内元素（假设刚才签名验证成功）：sig3 2 key3 → 遇到操作符 OP_CHECKSIGADD，弹出栈顶三个元素，用 key3 验证 sig3，如果验证成功则入栈 3，否则入栈 2 → 遇到 OP_2，则入栈 2 → 最后遇到 OP_GREATERTHANOREQUAL，弹出站内两个元素后比较它们的大小，如果第 2 个元素大于栈顶元素，则入栈 1，否则入栈 0。

如果站内最后剩下 1，表明脚本解锁成功，允许花掉 UTXO，否则失败。

4-19 保留字

操作符	操作码	十六进制	输入
OP_RESERVED	80	0x50	交易无效，除非出现在未被执行的 OP_IF 分支上
OP_VER	98	0x62	交易无效，除非出现在未被执行的 OP_IF 分支上
OP_VERIF	101	0x65	交易无效
OP_VERNOTIF	102	0x66	交易无效
OP_RESERVED1	137	0x89	交易无效，除非出现在未被执行的 OP_IF 分支上
OP_RESERVED2	138	0x8a	交易无效，除非出现在未被执行的 OP_IF 分支上
OP_NOP1，OP_NOP4-OP_NOP10	176, 179-185	0xb0, 0xb3-0xb9	这些操作符被忽略

看例子

例子 1：哈希碰撞——猜出两端不同的文字但哈希结果相同：

```
scriptPubKey: OP_2DUP OP_EQUAL OP_NOT OP_VERIFY OP_SHA1 OP_SWAP OP_SHA1 OP_EQUAL
scriptSig: <preimage1> <preimage2>
```

例子 2：猜谜

```
scriptPubKey: OP_HASH256 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000 OP_EQUAL
scriptSig:<data_hash>
```

例子 3：资金冻结

```
scriptPubKey: <expiry time> OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY
OP_CHECKSIG
scriptSig: <sig> <pubKey>
```

例子 4：

```
OP_0 OP_IF OP_2 OP_ELSE OP_3 OP_ENDIF                                                    //栈顶最后为 3
```

4.5.5 P2TR 脚本路径

之前讲过 P2TR 有两种花钱方法，一种是密钥路径，另一种是脚本路径。密钥路径类似 P2PKH，用私钥签名即可。而脚本路径更加灵活，它实际上包含了“密钥路径”的功能，但是还能构建更加复杂的花钱条件。本小节用一个实际的例子来说明这一点。

任务描述（比特币测试网）：

```
生成一个 P2TR 地址，让别人往此地址转比特币，允许以下几种方式花掉此地址上的 UTXO:
(1). 我可随时签名花钱;
(2). 我老婆可随时签名花钱;
(3). 半年后律师、公司副总和财务主管一起签名才能花钱;
(4). 二十年后我三个子女之中的任何两人签名也可以花钱。
公私钥对分配如下(要求公钥都是 02 开头):
(1). 我:
    WIF 私钥: cSYrtBbF1vEzoKtD1LGBqge71DXJ8TmVzajApwQBSZzdGJbL3q4Z
    压缩公钥: 024c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c
(2). 配偶:
    WIF 私钥: cTUwYps8n7GRTS789b8JH7GDdRjK6roY6RWJCjkSJubR7raxc4cg
    压缩公钥: 02b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853
(3). 子女 1:
    WIF 私钥: cUfGmsyHgFiueRjMtxAb3XDeFvdmJuqpqMPUnjYcqSzesk27ye8f
    压缩公钥: 02770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1
```

```
(4). 子女 2:
   WIF 私钥: cQvuoR1w4JcaLmFZDde2JbfLqb38ETHo8Td3vQGAmEDw7vN2XmcJ
   压缩公钥: 02c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d
(5). 子女 3:
   WIF 私钥: cU5FNwCsgVVGz7HxtipJyxNhxhTqvzow4PbmkgSvSYTubji41zua
   压缩公钥: 02563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6
(6). 律师:
   WIF 私钥: cUpkXKX3ST3gxC5J8tQrtur5RpCsp4hRfUxiro4vd1jdpxBSUeci
   压缩公钥: 027def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d5
(7). 副总:
   WIF 私钥: cW5oiUD8h6sTXiL23qDjhW6zEckug6STUq77BdkcC4MVRUemuiMR
   压缩公钥: 02a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956
(8). 财务主管:
   WIF 私钥: cVNaz7DkL6cxo5jepr6JPhfTfsDwC7GpqLXkrhYusoeZ8mquig2X
   压缩公钥: 02661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41
```

请采用 biecoinjs-lib 等 js 库完成这五个子任务：(1). 产生 P2TR 地址，(2). 我花钱，(3). 配偶花钱，(4). 公司团队花钱，(5). 三个子女花钱。

根据任务要求，我们绘出如图 4-26 所示的的默克尔树。

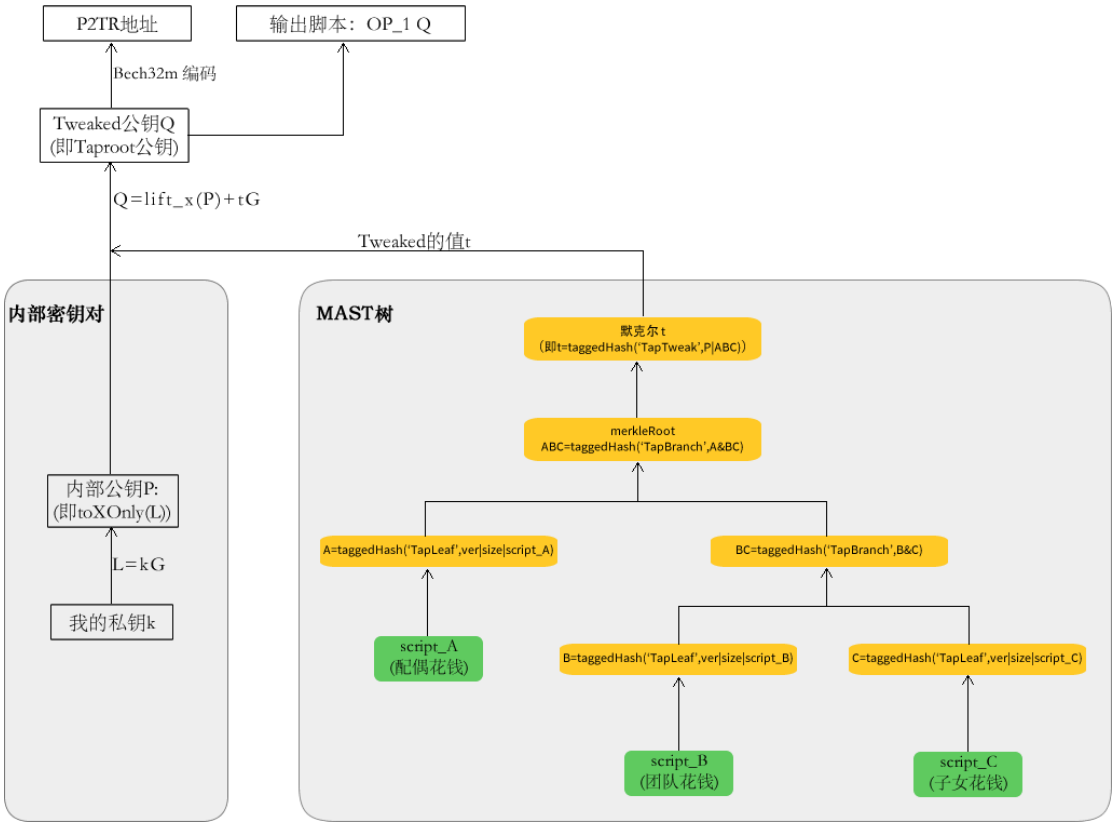


图 4-26 默克尔树

总接以下四种花钱方式:

(1). 使用 *k+t* 作为私钥【即 Tweaked Private Key，也可能是(secp256k1 的阶-k)+t】进行 Schnorr 签名——我花钱。

(2). 公布 script_A 和哈希 BC——配偶花钱，所以还需要把 BC 告诉她。

(3). 公布 script_B、哈希 A 和哈希 C——公司团队花钱，还需要把 A 和 C 告诉他们。

(4). 公布 script_C 和哈希 A 和哈希 B——子女花钱，还需要把 A 和 B 告诉他们。

首先需要如下三个函数:

```
const bitcoin = require('bitcoinjs-lib');
const toXOnly = pubKey => (pubKey.length === 32 ? pubKey : pubKey.slice(1, 33));

/*
  data: 类型 Buffer，字符串
  返回附加了长度作为前缀的字符串。用可变整型表示字符串长度。
*/
function varuintPre(data) {
   let len = data.length;
   if (len < 253) {
      return len.toString(16).padStart(2, '0') + data.toString('hex');
   } else {
      let tmp = '';
      if (len < 0x10000) {
         tmp = 'fd' + len.toString(16).padStart(4, '0');
      } else if (len < 100000000) {
         tmp = 'fe' + len.toString(16).padStart(8, '0');
      } else {
         tmp = 'ff' + len.toString(16).padStart(16, '0');
      }
      return tmp.replace(/(.{2})/g, "$1 ").split(" ").reverse().join("") + data.toString('hex');
   }
}

/*
  keyPair - 内部密钥对（由“图 4-26 默克尔树”中的“我的私钥 k”产生的）
  opts.network - 网络类型
  opts.merkleRoot - 默克尔根哈希（32 字节 Buffer）（即“图 4-26 默克尔树”中的“ABC”）
  返回由私钥 k+t 产生的密钥对，在我花钱时用来签名。
*/
function tweakSigner(keyPair, opts) {
   if(!keyPair || !opts){
      throw new Error('缺少参数! ');
   }
   let privateKey = keyPair?.privateKey;
   if (!privateKey) {
      throw new Error('Private key is required for tweaking signer!');
   }
   if (keyPair.publicKey[0] === 3) {                                          //奇数私钥转化为偶数私钥
      privateKey = ecc.privateNegate(privateKey);                          //secp256k1 的阶减去 privateKey
   }

   const pubKey = toXOnly(keyPair.publicKey);
   const tweak = bitcoin.crypto.taggedHash(                 //计算 tweak（即“图 4-26 默克尔树”中的“默克尔 t”）
      'TapTweak',
      Buffer.concat(opts.merkleRoot ? [pubKey, opts.merkleRoot] : [pubKey]),
   );

   let tweakedPrivateKey = ecc.privateAdd(privateKey, tweak);

   if (!tweakedPrivateKey) {
      throw new Error('Invalid tweaked private key!');
   }

   return ECPair.fromPrivateKey(Buffer.from(tweakedPrivateKey), {
      network: opts?.network || bitcoin.networks.testnet                                  //默认为比特币测试
网
   });
}

/*
  参数 provableNonce - 相当于默克尔树的 P|merkleRoot
  返回一个不能以密钥路径花费的内部公钥。
*/
function makeUnspendableInternalKey(provableNonce) {
   // 这是 secp256k1 的生成元 G，它的私钥就是 1:
   const G =
Buffer.from('0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798483ada7726a3c4655da4fbfc0e1
108a8fd17b448a68554199c47d08ffb10d4b8', 'hex');
   //32 字节的 Hx 也是椭圆曲线上的一个有效点的 x 坐标（压缩公钥），但是我们不知道它的私钥是什么，把它作为 P2TR 的内部公钥使用，因
此不能以密钥路径花费。
```

```
    const Hx = bitcoin.crypto.sha256(G);

    if (provableNonce) {
        if (provableNonce.length !== 32) {
            throw new Error(
                'provableNonce 是默克尔树叶（脚本）拥有者共享的 32 个字节的随机数。',
            );
        }
        // 使用一个共享的随机数，我们可创建不能以密钥花费的内部公钥
        // Q = Hx + int(hash_taptweak(provableNonce))*G
        // 因为我们不知道 Hx 的私钥，所以也不知道 Q 的私钥
        const tapHash = bitcoin.crypto.taggedHash('TapTweak', provableNonce);
        const ret = ecc.xOnlyPointAddTweak(Hx, tapHash);
        if (!ret) {
            throw new Error(
                'provableNonce 产生了一个无效的公钥',
            );
        }
        return Buffer.from(ret.xOnlyPubkey);
    } else {
        //直接把 Hx 作为内部私钥的缺点是别人都知道此 UTXO 不能以密钥路径花费，对于大部分普通人来说问题不大，但是对于诸如交易所那
样的机构不太愿意暴露这种隐私
        return Hx;
    }
}
```

任务 1. 产生一个 P2TR 地址：

```
const bitcoin = require('bitcoinjs-lib');
const { ECPairFactory } = require('ecpair');
const ecc = require('@bitcoinerlab/secp256k1');          // @bitcoinerlab/secp256k1 和 tiny-secp256k1 任选一个
//const ecc = require('tiny-secp256k1');
const wif = require('wif');

const toXOnly = pubKey => (pubKey.length === 32 ? pubKey : pubKey.slice(1, 33));

const ECPair = ECPairFactory(ecc);
bitcoin.initEccLib(ecc);

const network = bitcoin.networks.testnet;                                                    //使用比特币测试网
const publicKeys = {
    me: Buffer.from('4c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c', 'hex'),  // 我的公钥

    // 配偶的公钥
    spouse: Buffer.from('b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853', 'hex'),

    // 三个子女的公钥
    children: [
        Buffer.from('770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1', 'hex'),
        Buffer.from('c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d', 'hex'),
        Buffer.from('563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6', 'hex')
    ],

    // 管理团队的公钥
    management: [
        Buffer.from('7def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d5', 'hex'),      // 律师
        Buffer.from('a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956', 'hex'),      // 副总
        Buffer.from('661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41', 'hex')    // 财务主管
    ]
};

// 半年后的时间锁（可用相对时间，即基于区块高度）
const HALF_YEAR_BLOCKS = 26208;                          //半年 182 天，182*24*60/10=26208，每 10 分钟产生一个区块

// 20 年后的时间锁（绝对时间戳）
const TWENTY_YEARS = Math.floor(Date.now() / 1000) + 20 * 365 * 24 * 60 * 60;

// 配偶单独花钱的脚本（类似 P2PKH）——script_A:
const script_A = bitcoin.script.compile([
    bitcoin.opcodes.OP_DUP,
    bitcoin.opcodes.OP_HASH160,
```

```
    bitcoin.crypto.hash160(toXOnly(publicKeys.spouse)),              //是用publicKeys.spouse还是toXonly(…)?
    bitcoin.opcodes.OP_EQUALVERIFY,
    bitcoin.opcodes.OP_CHECKSIG
]);
console.log(`${script_A.toString('hex')}\n${bitcoin.script.toASM(script_A)}`);
76a914eb6fdc07374ab2ac9a7c07193066fc982520c42488ac
OP_DUP OP_HASH160 eb6fdc07374ab2ac9a7c07193066fc982520c424 OP_EQUALVERIFY OP_CHECKSIG
//script_A
const script_A = bitcoin.script.compile([
    toXOnly(publicKeys.spouse),
    bitcoin.opcodes.OP_CHECKSIG
]);
console.log(`${script_A.toString('hex')}\n${bitcoin.script.toASM(script_A)}`);
20b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853ac
b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853 OP_CHECKSIG

// 管理团队 3/3 多签脚本:
const managementScript = bitcoin.script.compile([
    bitcoin.script.number.encode(3),                                                    // 需要 3 个签名
    ...publicKeys.management,                                                           // 三个管理公钥
    bitcoin.script.number.encode(3),                                                    // 总共 3 个公钥
    bitcoin.opcodes.OP_CHECKMULTISIG
]);

// 半年后管理团队花费脚本——script_B:
const script_B = bitcoin.script.compile([
    bitcoin.script.number.encode(HALF_YEAR_BLOCKS),
    bitcoin.opcodes.OP_CHECKSEQUENCEVERIFY,
    bitcoin.opcodes.OP_DROP,
    ...bitcoin.script.decompile(managementScript)
]);
console.log(`${script_B.toString('hex')}\n${bitcoin.script.toASM(script_B)}`);
026066b27553207def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d520a45bf184a457694e4b0cc4ef79
d24447fb1b0de1897b96d652d8b4f73075595620661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c4153
ae
6066 OP_CHECKSEQUENCEVERIFY OP_DROP OP_3 7def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d5
a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956
661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41 OP_3 OP_CHECKMULTISIG          //script_B

// 子女 2/3 多签脚本:
const childrenScript = bitcoin.script.compile([
    toXOnly(publicKeys.children[0]),
    bitcoin.opcodes.OP_CHECKSIG,
    toXOnly(publicKeys.children[1]),
    bitcoin.opcodes.OP_CHECKSIGADD,
    toXOnly(publicKeys.children[2]),
    bitcoin.opcodes.OP_CHECKSIGADD,
    bitcoin.opcodes.OP_2,
    bitcoin.opcodes.OP_GREATERTHANOREQUAL
]);

// 20 年后子女花费脚本——script_C:
const script_C = bitcoin.script.compile([
    bitcoin.script.number.encode(TWENTY_YEARS),
    bitcoin.opcodes.OP_CHECKLOCKTIMEVERIFY,
    bitcoin.opcodes.OP_DROP,
    ...bitcoin.script.decompile(childrenScript)
]);
console.log(`${script_C.toString('hex')}\n${bitcoin.script.toASM(script_C)}`);
052596888d00b17520770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1ac20c5e1f3dc9d9a23650710
daa31cfc5faa055c4350fa94690936666a60c6a3e16dba20563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88
157da6ba52a2
2596888d00 OP_CHECKLOCKTIMEVERIFY OP_DROP 770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1
OP_CHECKSIG c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d OP_CHECKSIGADD
563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6 OP_CHECKSIGADD OP_2 OP_GREATERTHANOREQUAL
//script_C

//计算哈希 A:
let s = `c0${varuintPre(script_A)}`;                                    //version|size|script_A
const A = bitcoin.crypto.taggedHash('TapLeaf', Buffer.from(s,'hex'));
console.log(A.toString('hex'));
494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5fd23534b49                    //哈希 A
```

```
//计算哈希 B:
s = `c0${varuintPre(script_B)}`;
const B = bitcoin.crypto.taggedHash('TapLeaf', Buffer.from(s,'hex'));
console.log(B.toString('hex'));
b3a89c901850dc8010aae0e31acbcf2b193b643ffe4f880fa95e506fa39f27b9                    //哈希 B

//计算哈希 C:
s = `c0${varuintPre(script_C)}`;
const C = bitcoin.crypto.taggedHash('TapLeaf', Buffer.from(s,'hex'));
console.log(C.toString('hex'));
3fd18c4440c1e3da8c38833414b47afefac8942c4e7c7d432249707176bcbc2a                    //哈希 C

//计算哈希 BC:
const BC = bitcoin.crypto.taggedHash(
    'TapBranch',
    Buffer.concat([ B < C ? B: C, B < C ? C: B])
);
console.log(BC.toString('hex'));
adca7de3ce0750a80d80fbb5c5990bb5bf5aedc8a84495e698a1dcb436f3cd41                    //哈希 BC

//计算默克尔树根的哈希 ABC:
const ABC = bitcoin.crypto.taggedHash(
    'TapBranch',
    Buffer.concat([ A < BC ? A: BC, A < BC ? BC: A])
);
console.log(ABC.toString('hex'));
0b5088cdafe0c6b3686a87377252bf2f2c929294f9c10a3e54aee388b0256da8          //哈希 ABC（即默克尔树根）

//计算默克尔 t:
let data = Buffer.concat([toXOnly(publicKeys.me), ABC]);                    //内部公钥在前面
const t = bitcoin.crypto.taggedHash('TapTweak', data);
console.log(t.toString('hex'));
4614f83194bd3cbc367126871b3b76c0bdbcf184e22419b78b086d9f44abdec7                    //默克尔 t

//计算 Q:
const tapRoot = ecc.xOnlyPointAddTweak(toXOnly(publicKeys.me), t);
const parity = tapRoot.parity;                              //校验位，0 或 1，控制块第 1 个字节等于 0xc0|parity
console.log(parity);
1
const taprootPubKey = tapRoot.xOnlyPubkey;
const Q = Buffer.from(taprootPubKey);
console.log(Q.toString('hex'));
2960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0                    //Q

const p2trAddress = bitcoin.address.fromOutputScript(
    bitcoin.script.fromASM(`OP_1 ${Q.toString('hex')}`),
    network
);
console.log(p2trAddress);
tb1p99svx4dxnwte3095ccnwntv3700uqyf7mudexgkef8zd9teuwncqwa7x8p                    //P2TR 地址

//下面计算 Tweaked 私钥，即我的私钥加上 t，用于我签名
const myWif = 'cSYrtBbF1vEzoKtD1LGBqge71DXJ8TmVzajApwQBSZzdGJbL3q4Z'                    //我的私钥
const keyPair = ECPair.fromWIF(myWif, network);
let privateKey = keyPair.privateKey;
if (keyPair.publicKey[0] === 3) {                                   //奇数私钥转化为偶数私钥
    privateKey = ecc.privateNegate(privateKey);
}
const tweakedPrivateKey = ecc.privateAdd(privateKey, t);
const tweakedWIF = wif.encode({
    version: network == bitcoin.networks.bitcoin ? 128 : 239,
    privateKey:tweakedPrivateKey,
    compressed: true
});
console.log(tweakedWIF);
cUu6EApZKw9N8YLNFJGvBVxxHtvw9nn6PAFw2xUsXFDmtDsF3fDu                    //Tweaked 私钥

const tweakedKeyPair = ECPair.fromPrivateKey(Buffer.from(tweakedPrivateKey), {network: network});
console.log(tweakedKeyPair.publicKey.toString('hex'));
```

```
032960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0 //可见 toXOnly()后等于 Q。前缀 03 表示公钥的 y 坐标为奇数，对应前面的 parity 就是 1

//另一种推导 P2TR 地址的方法如下:
// 构建默克尔树:
const merkleTree = [
   {                                                                                    // 路径 1: 配偶可随时花钱
      output: script_A,
      version: 192                                                                      // Tapscript 版本 0xc0
   },
   [
      {                                                                                 // 路径 2: 二十年后子女可花钱
         output: script_C,
         version: 192
      },
      {                                                                                 // 路径 3: 半年后管理团队可花钱
         output: script_B,
         version: 0xc0
      }
   ]
];

const p2tr = bitcoin.payments.p2tr({
   internalPubkey: toXOnly(publicKeys.me),                                  // 使用您的公钥作为内部密钥(去掉压缩前缀)
   scriptTree: merkleTree,
   network
});

console.log(`address=${p2tr.address}\nmerkleRoot=${p2tr.hash.toString('hex')}\nQ=${p2tr.pubkey.toString('hex')}\noutput=${p2tr.output.toString('hex')}`);
address=tb1p99svx4dxnwte3095ccnwntv3700uqyf7mudexgkef8zd9teuwncqwa7x8p          //推导的地址与第一种方法相同
merkleRoot=0b5088cdafe0c6b3686a87377252bf2f2c929294f9c10a3e54aee388b0256da8                  //即 ABC
Q=2960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0
output=51202960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0          //格式 OP_1 <Q>
```

即最后产生的 P2TR 地址为 tb1p99svx4dxnwte3095ccnwntv3700uqyf7mudexgkef8zd9teuwncqwa7x8p，默克尔树是：

```
const merkleTree = [
   {                                                                                    //配偶可随时花钱
      output: bitcoin.script.fromASM('b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853 OP_CHECKSIG'),
      version: 192
   },
   [
      {                                                                                 //20 年后三个子女中的任何两人花钱脚本路径
         output: bitcoin.script.fromASM('2596888d00 OP_CHECKLOCKTIMEVERIFY OP_DROP
770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1 OP_CHECKSIG
c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d OP_CHECKSIGADD
563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6 OP_CHECKSIGADD OP_2
OP_GREATERTHANOREQUAL'),
         version: 192
      },
      {                                                                                 //半年后团队花钱脚本路径
         output: bitcoin.script.fromASM('6066 OP_CHECKSEQUENCEVERIFY OP_DROP OP_3
7def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d5
a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956
661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41 OP_3 OP_CHECKMULTISIG'),
         version: 192
      },
   ],
]
```

总结出表 4-20。

表 4-20 计算出的各种数据

内部私钥 k	cSYrtBbF1vEzoKtD1LGBqge71DXJ8TmVzajApwQBSZzdGJbL3q4Z	
内部公钥 P	4c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c	
默克尔树 t	4614f83194bd3cbc367126871b3b76c0bdbcf184e22419b78b086d9f44abdec7	
默克尔树根哈希	0b5088cdafe0c6b3686a87377252bf2f2c929294f9c10a3e54aee388b0256da8	即哈希 ABC

Tweaked 私钥 (k+t)	cUu6EApZKw9N8YLNFJGvBVxxHtvw9nn6PAFw2xUsXFDmtDsF3fDu	我花钱时签名的私钥
Q	2960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0	
script_A	20b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853ac	配偶花钱
script_B	026066b27553207def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d520a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f73075595620661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c4153ae	团队花钱
script_C	052596888d00b17520770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1ac20c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa946909366666a60c6a3e16dba20563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6ba52a2	子女花钱
哈希 A	494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5fd23534b49	
哈希 B	b3a89c901850dc8010aae0e31acbcf2b193b643ffe4f880fa95e506fa39f27b9	
哈希 C	3fd18c4440c1e3da8c38833414b47afefac8942c4e7c7d4322497071 76bcbc2a	
哈希 BC	adca7de3ce0750a80d80fbb5c5990bb5bf5aedc8a84495e698a1dcb436f3cd41	
哈希 ABC	0b5088cdafe0c6b3686a87377252bf2f2c929294f9c10a3e54aee388b0256da8	
P2TR 地址	tb1p99svx4dxnwte3095ccnwntv3700uqyf7mudexgkef8zd9teuwncqwa7x8p	
输出脚本	51202960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0	
校验位	1	即 parity
默克尔树	[{output: script_A}, [{output: script_B}, {output: script_C}]]	

在以脚本路径花钱时构造的控制块的第一个字节等于 0xc0|parity，我们现在的 parity=1，所以控制块的第 1 个字节就是 0xc1，即十进制的 193。

接下来就让别人往此地址 tb1p99svx4dxnwte3095ccnwntv3700uqyf7mudexgkef8zd9teuwncqwa7x8p 打款了。

任务 2. 密钥路径放币——我花钱，只需用 Tweaked 私钥签名即可。

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('@bitcoinerlab/secp256k1');
const { ECPairFactory } = require('ecpair');
const ECPair = ECPairFactory(ecc);

const toXOnly = pubKey => (pubKey.length === 32 ? pubKey : pubKey.slice(1, 33));
const validator = (pubkey, msghash, signature) => ecc.verifySchnorr(msghash, pubkey, signature);

bitcoin.initEccLib(ecc);

const network = bitcoin.networks.testnet;
const myWif = 'cSYrtBbF1vEzoKtD1LGBqge71DXJ8TmVzajApwQBSZzdGJbL3q4Z';
const ABC = '0b5088cdafe0c6b3686a87377252bf2f2c929294f9c10a3e54aee388b0256da8';                //默克尔根哈希

const keyPair = ECPair.fromWIF(myWif, network);

//由内部密钥对转化为 tweaked 密钥对:
const signPair = tweakSigner(keyPair, { network: network, merkleRoot: Buffer.from(ABC, 'hex') });
const output = `OP_1 ${toXOnly(signPair.publicKey).toString('hex')}`;                        //UTXO 的输出脚本

const psbt = new bitcoin.Psbt({network: network});
psbt.setVersion(2);                                                                  //设置版本，默认就是 2，所以本语句可省略
psbt.setLocktime(0);

psbt.addInput({
   hash: 'b105ea58168881e90f8d33a2b0a847a1127942c2451585b749fb8cce0b8d92fd',                    // 交易 ID
   index: 0,
   witnessUtxo: {
      script: bitcoin.script.fromASM(output),                                                  //UTXO 的输出脚本
      value: 10000,                                                                  // UTXO 金额（单位：satoshis）
   },
   tapInternalKey: toXOnly(keyPair.publicKey),                                                 // 内部公钥
   tapMerkleRoot: Buffer.from(ABC, 'hex')                                                     //默克尔树根哈希
});

psbt.addOutput({
  address: 'tb1qwv5wau4ye8w7p0z9ylcscy8cet9zs5d80aut9s',                                       // 接收方地址
  value: 9500,                                                                       // 转账金额（扣除手续费）
});

psbt.signInput(0,signPair);
psbt.validateSignaturesOfAllInputs(validator);
psbt.finalizeAllInputs();
```

```
const tx = psbt.extractTransaction();
const txHex = tx.toHex();
console.log(txHex);
02000000000101fd928d0bce8cfb49b7851545c2427912a147a8b0a2338d0fe981881658ea05b10000000000ffffffff011c250000
0000000001600147328eef2a4c9dde0bc4527f10c10f8caca2851a7014039949fc2cbb3eb94d0532b4754b1eae9c5f6af7904838b67
df08cc63ff6df50e7ed7cbbcdf2981ba723e30d0b385837d41fb72acad1af2b8efc0c48d280d846400000000
```

任务 3. 脚本路径放币——配偶花钱，需要公布 script_A 和哈希 BC。

增加一个函数 witnessStackToScriptWitness 把若干见证条款整合成一个见证：

```
const varuint = require('varuint-bitcoin');

/*
  参数 witness - 类型 Buffer[]，包含若干见证条款
  返回 Buffer 类型的见证序列
*/
function witnessStackToScriptWitness(witness) {
   let buffer = Buffer.allocUnsafe(0);

   function writeSlice(slice) {
      buffer = Buffer.concat([buffer, Buffer.from(slice)]);
   }

   function writeVarInt(i) {
      const currentLen = buffer.length;
      const varintLen = varuint.encodingLength(i);
      buffer = Buffer.concat([buffer, Buffer.allocUnsafe(varintLen)]);
      varuint.encode(i, buffer, currentLen);
   }

   function writeVarSlice(slice) {
      writeVarInt(slice.length);
      writeSlice(slice);
   }

   function writeVector(vectors) {
      writeVarInt(vectors.length);
      vectors.forEach(writeVarSlice);
   }

   writeVector(witness);
   return buffer;
}
```

配偶花费的实现代码：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('@bitcoinerlab/secp256k1');
const { ECPairFactory } = require('ecpair');
const ECPair = ECPairFactory(ecc);

const validator = (pubkey, msghash, signature) => ecc.verifySchnorr(msghash, pubkey, signature);
const network = bitcoin.networks.testnet;

const script_A = Buffer.from('20b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853ac',
'hex');
const BC = Buffer.from('adca7de3ce0750a80d80fbb5c5990bb5bf5aedc8a84495e698a1dcb436f3cd41', 'hex');
const P = Buffer.from('4c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c','hex');  //内部公钥
const leafVersion = 0xc0;
const parity = 1;                                                    //这个要根据实际情况而定。我们这里是 1，有时是 0

const A = bitcoin.crypto.taggedHash(
   'TapLeaf',
   Buffer.from(`${leafVersion.toString(16)}${varuintPre(script_A)}`, 'hex')
);

const controlBlock = Buffer.concat([ Buffer.from([leafVersion|parity]), P, BC ]);     //应该“|y(Q) mod 2”

const psbt = new bitcoin.Psbt({ network: network });

psbt.addInput({
```

```
    hash: 'b105ea58168881e90f8d33a2b0a847a1127942c2451585b749fb8cce0b8d92fd',      // 替换为实际 UTXO 的交易 ID
    index: 0,                                                                      // UTXO 的输出索引
    witnessUtxo: {
        script: Buffer.from('51202960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0', 'hex'),
// P2TR 输出脚本
        value: 10000                                                          // UTXO 金额（单位：satoshis）
    },
    tapLeafScript: [{
        script: script_A,
        controlBlock: controlBlock,
        leafVersion: leafVersion
    }]
});

psbt.addOutput({
  address: 'tb1qwv5wau4ye8w7p0z9ylcscy8cet9zs5d80aut9s',                                  // 接收方地址
  value: 9500,                                                                  // 转账金额（扣除手续费）
});
const spouseWIF = 'cTUwYps8n7GRTS789b8JH7GDdRjK6roY6RWJCjkSJubR7raxc4cg';
const keyPair = ECPair.fromWIF(spouseWIF, network);
psbt.signInput(0, keyPair);
psbt.validateSignaturesOfAllInputs(validator);

//定义我们自己的终止器:
const customFinalizer = (_inputIndex: number, input: any) => {
    const scriptSolution = [
        input.tapScriptSig[0].signature,
        toXOnly(keyPair.publicKey)
    ];
    const witness = scriptSolution.concat(script_A).concat(controlBlock);

    return {
        finalScriptWitness: witnessStackToScriptWitness(witness)
    }
}

psbt.finalizeInput(0, customFinalizer);
psbt.finalizeAllInputs()
const tx = psbt.extractTransaction();
const txHex = tx.toHex();
console.log(txHex);
02000000000101fd928d0bce8cfb49b7851545c2427912a147a8b0a2338d0fe981881658ea05b10000000000ffffffff011c250000
0000000001600147328eef2a4c9dde0bc4527f10c10f8caca2851a703407f546e8f1222dcf412a51ea2c994b9e99412412e4cb52cde
04e57161f14e139b707c7887035a2197357da7b3308337dc220b8f1a1e62faf1bf6d9c5c4aa751af2220b8eac037eea8aec3d26e3e
9ac4b698fe0aca713f19ea22aa673b75786ac65853ac41c14c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc58172
03391cadca7de3ce0750a80d80fbb5c5990bb5bf5aedc8a84495e698a1dcb436f3cd4100000000
```

任务 4. 脚本路径放币——公司团队花钱，公布 script_B、哈希 A 和哈希 C。

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('@bitcoinerlab/secp256k1');
const { ECPairFactory } = require('ecpair');
const wif = require('wif');
const ECPair = ECPairFactory(ecc);
bitcoin.initEccLib(ecc);

const validator = (pubkey, msghash, signature) => ecc.verifySchnorr(msghash, pubkey, signature);
const network = bitcoin.networks.testnet;
const leafVersion = 0xc0;
const P = Buffer.from('4c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c', 'hex');//内部公钥

//脚本路径上的兄弟节点哈希，顺序从叶子脚本哈希的兄弟 -> … -> 默克尔树根的儿子
const script_paths = [
    '3fd18c4440c1e3da8c38833414b47afefac8942c4e7c7d432249707176bcbc2a',              //script_B 的哈希的兄弟 C
    '494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5fd23534b49'                    //默克尔树根的儿子 A
];

const WIFs = [                                                                  //参与签名的三个私钥（WIF 格式）
    'cUpkXKX3ST3gxC5J8tQrtur5RpCsp4hRfUxiro4vd1jdpxBSUeci',
```

```
    'cW5oiUD8h6sTXiL23qDjhW6zEckug6STUq77BdkcC4MVRUemuiMR',
    'cVNaz7DkL6cxo5jepr6JPhfTfsDwC7GpqLXkrhYusoeZ8mquig2X'
];
const privateKeys = WIFs.map(w => Buffer.from(wif.decode(w, network == bitcoin.networks.bitcoin ? 128 :
239).privateKey));
const publicKeys = privateKeys.map(p => Buffer.from(ecc.xOnlyPointFromScalar(p)));                    //公钥
const keyPairs = WIFs.map(w => ECPair.fromWIF(w, network));                                          //密钥对，用于签名

// 叶子脚本 (3-of-3 多签 + 半年时间锁)
const HALF_YEAR_BLOCKS = 26208;                          //半年 182 天，182*24*60/10=26208，每 10 分钟产生一个区块
const script_B = bitcoin.script.compile([
    bitcoin.script.number.encode(HALF_YEAR_BLOCKS),
    bitcoin.opcodes.OP_CHECKSEQUENCEVERIFY,
    bitcoin.opcodes.OP_DROP,
    bitcoin.opcodes.OP_3,
    ...publicKeys,
    bitcoin.opcodes.OP_3,
    bitcoin.opcodes.OP_CHECKMULTISIG
]);

const parity = 1;
const controlBlock = Buffer.concat([                                                                  // 构造控制块
    Buffer.from([leafVersion | parity]),                                                             // 带内部公钥的版本号
    P,
    ...script_paths.map(h => Buffer.from(h, 'hex'))
]);

const psbt = new bitcoin.Psbt({ network: network });

psbt.addInput({
    hash: 'b105ea58168881e90f8d33a2b0a847a1127942c2451585b749fb8cce0b8d92fd',
    index: 0,
    witnessUtxo: {
        script: Buffer.from('51202960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0', 'hex'),
//UTXO 的 P2TR 输出脚本 (OP_1 <P>)
        value: 10000
    },
    tapLeafScript: [{
        script: script_B,
        controlBlock,
        leafVersion
    }],
    sequence: 26210,                                                                                 // 必须 ≥ 时间锁值 (26208)
});

psbt.addOutput({
    address: 'tb1qwv5wau4ye8w7p0z9ylcscy8cet9zs5d80aut9s',
    value: 9500
});

keyPairs.forEach(k => psbt.signInput(0, k));
psbt.validateSignaturesOfAllInputs(validator);

//const tx = psbt.__CACHE.__TX;
///const sighash = tx.hashForWitnessV1(
//    0,//输入索引
//    [script_B],
//    [10000],
//    bitcoin.Transaction.SIGHASH_DEFAULT
//);

// 每个参与者生成部分签名
//const sigs = privateKeys.map(p => Buffer.from(ecc.signSchnorr(sighash, p)));
//psbt.validateSignaturesOfAllInputs(validator);
// ========== 5. 构造见证数据 ==========
//const witness = Buffer.concat([
//    bitcoin.script.number.encode(witnessStack.length),
//    ...witnessStack.map(item => Buffer.concat([
//        bitcoin.script.number.encode(item.length),
//        item
//    ]))
//]);
```

```
//psbt.updateInput(0, {                                                    // 更新輸入的见证数据
//   finalScriptWitness: witness,
//});

psbt.finalizeAllInputs();
const finalTx = psbt.extractTransaction();
console.log('最终交易:', finalTx.toHex());
最终交易:
02000000000101fd928d0bce8cfb49b7851545c2427912a147a8b0a2338d0fe981881658ea05b1000000000062660000011c250000
00000001600147328eef2a4c9dde0bc4527f10c10f8caca2851a70540ffffa036f79e079b468f506c9de9f8077dda77d3135915ec
2368a464626bcbac2d91e97111b2943be89c95c047a822edc7ffa39577e32c332b38324f0c1983e340ab1ed84cc8ba4004fb2004c9
566afcdce9e548ab7a0c92e39ab2d50595165d804256d46f9e7783b30ebc81cd962720f059593f96660716ea82d06171c11dbfd640
d4bc5b9b538c3f8e444a531caaa3b6099d8d146907e3b80c55cfa2eaafe0c90f4d9cf2339f95ea8889a5d79c8cd8fdef8a6c085d18
e079c02c7ea5b8c775eb5d6b026066b27553207def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d520a4
5bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f73075595620661caba6b0396d1355273f6f9c4345826490e935f3
6206473c0cf1e694d49c4153ae61c14c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c3fd18c4440c1
e3da8c38833414b47afefac8942c4e7c7d432249707176bcbc2a494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5
fd23534b4900000000
console.log('交易 ID:', finalTx.getId());
交易 ID: f2a7aaa95dad3774977e04ae7d2dfddf0acb3cb2c2e40eda0a847f17df80040e
```

任务 5. 脚本路径放币——20 年之后子女花钱，公布 script_C、哈希 A 和哈希 B。

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('@bitcoinerlab/secp256k1');
const { ECPairFactory } = require('ecpair');
const wif = require('wif');
const ECPair = ECPairFactory(ecc);
bitcoin.initEccLib(ecc);

const validator = (pubkey, msghash, signature) => ecc.verifySchnorr(msghash, pubkey, signature);
const network = bitcoin.networks.testnet;
const leafVersion = 0xc0;
const P = Buffer.from('4c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c', 'hex');//内部公钥

//脚本路径上的兄弟节点哈希，顺序从叶子脚本哈希的兄弟 -> … -> 默克尔树根的儿子
const script_paths = [
    'b3a89c901850dc8010aae0e31acbcf2b193b643ffe4f880fa95e506fa39f27b9',              //script_C 的哈希的兄弟 B
    '494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5fd23534b49'                    //默克尔树根的儿子 A
];

const WIFs = [                                                                  //参与签名的三个私钥（WIF 格式）
    'cUfGmsyHgFiueRjMtxAb3XDeFvdmJuqpqMPUnjYcqSzesk27ye8f',
    'cQvuoR1w4JcaLmFZDde2JbfLqb38ETHo8Td3vQGAmEDw7vN2XmcJ',
    'cU5FNwCsgVVGz7HxtipJyxNhxhTqvzow4PbmkgSvSYTubji41zua'
];
const privateKeys = WIFs.map(w => Buffer.from(wif.decode(w, network == bitcoin.networks.bitcoin ? 128 :
239).privateKey));
const publicKeys = privateKeys.map(p => Buffer.from(ecc.xOnlyPointFromScalar(p)));                  //公钥
const keyPairs = WIFs.map(w => ECPair.fromWIF(w, network));                                 //密钥对，用于签名

// 叶子脚本 (2-of-3 多签 + 半年时间锁)
const leafScript = bitcoin.script.compile([
    publicKeys[0],
    bitcoin.opcodes.OP_CHECKSIG,
    publicKeys[1],
    bitcoin.opcodes.OP_CHECKSIGADD,
    publicKeys[2],
    bitcoin.opcodes.OP_CHECKSIGADD,
    bitcoin.opcodes.OP_2,
    bitcoin.opcodes.OP_GREATERTHANOREQUAL
]);
console.log(leafScript.toString('hex'));
20770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1ac20c5e1f3dc9d9a23650710daa31cfc5faa055c
4350fa94690936666a60c6a3e16dba20563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6ba52a2

const TWENTY_YEARS = Math.floor(Date.now() / 1000) + 20 * 365 * 24 * 60 * 60;
const script_C = bitcoin.script.compile([
```

```
    bitcoin.script.number.encode(TWENTY_YEARS),
    bitcoin.opcodes.OP_CHECKLOCKTIMEVERIFY,
    bitcoin.opcodes.OP_DROP,
    ...bitcoin.script.decompile(leafScript)
]);
console.log(script_C.toString('hex'))
05a726898d00b17520770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1ac20c5e1f3dc9d9a23650710
daa31cfc5faa055c4350fa94690936666a60c6a3e16dba20563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88
157da6ba52a2

const parity = 1;
const controlBlock = Buffer.concat([                                                    // 构造控制块
    Buffer.from([leafVersion | parity]),                                              // 带内部公钥的版本号
    P,
    ...script_paths.map(h => Buffer.from(h, 'hex'))
]);
console.log(controlBlock.toString('hex'))
c14c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391cb3a89c901850dc8010aae0e31acbcf2b193b643f
fe4f880fa95e506fa39f27b9494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5fd23534b49

const psbt = new bitcoin.Psbt({ network: network });

psbt.addInput({
    hash: 'b105ea58168881e90f8d33a2b0a847a1127942c2451585b749fb8cce0b8d92fd',
    index: 0,
    witnessUtxo: {
        script: Buffer.from('51202960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0', 'hex'),
//UTXO 的 P2TR 输出脚本 (OP_1 <P>)
        value: 10000
    },
    tapLeafScript: [{
        script: script_C,
        controlBlock,
        leafVersion
    }],
    sequence: 0xfffffffe,
    locktime: TWENTY_YEARS,                                                          // 必须 ≥ 时间锁值
});

psbt.addOutput({
    address: 'tb1qwv5wau4ye8w7p0z9ylcscy8cet9zs5d80aut9s',
    value: 9500
});

keyPairs.forEach(k => psbt.signInput(0, k));                                        //三个签名或者两个签名均可
//psbt.signInput(0,keyPairs[0]);
//psbt.signInput(0,keyPairs[2]);
psbt.validateSignaturesOfAllInputs(validator);

psbt.finalizeAllInputs();
const finalTx = psbt.extractTransaction();
console.log('最终交易:', finalTx.toHex());
最终交易:
02000000000101fd928d0bce8cfb49b7851545c2427912a147a8b0a2338d0fe981881658ea05b10000000000feffffff011c250000
000000001600147328eef2a4c9dde0bc4527f10c10f8caca2851a70440522590969dfb2d0f2f7364b8a173313dd8242e2ebe282c2e
47d25ce0fb974b7128976bdb497a49e9074db09019058d527cd4bfa724131c0f57eea85e845e07d9401591bfc0412de1e9445ff122
b4e370decee408694b52127577f7ce60b7c52114e9028f6d5ae4e39e689e28550d312fd9ba6906359f8b9c5b75e68ac781df319a70
05a726898d00b17520770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1ac20c5e1f3dc9d9a23650710
daa31cfc5faa055c4350fa94690936666a60c6a3e16dba20563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88
157da6ba52a261c14c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391cb3a89c901850dc8010aae0e31a
cbcf2b193b643ffe4f880fa95e506fa39f27b9494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5fd23534b490000
0000
console.log('交易 ID:', finalTx.getId());
交易 ID: e91e14947d50c053f982d6117932e8561eac1b77e2f9c7724d91edc5501874fe
```

总结:

1. 密钥路径花费时需要提供默克尔树根哈希、内部公钥，用内部公钥对应私钥签名即可。

2. 脚本路径花费时需要提供叶子脚本、从叶子到根的各个兄弟节点（哈希值）、内部公钥，然后根据叶子脚本（输出脚本）构造相应的输入脚本即可。

5 交易实操

加密币交易的流程大致如下：

产生一个随机数→产生一组助记词→生成种子→生成扩展根私钥→生成“标记扩展私钥生成树”上某个节点的扩展私钥→产生钱包（加密币的私钥和地址）→加密币交易，参考图 5-1 所示。

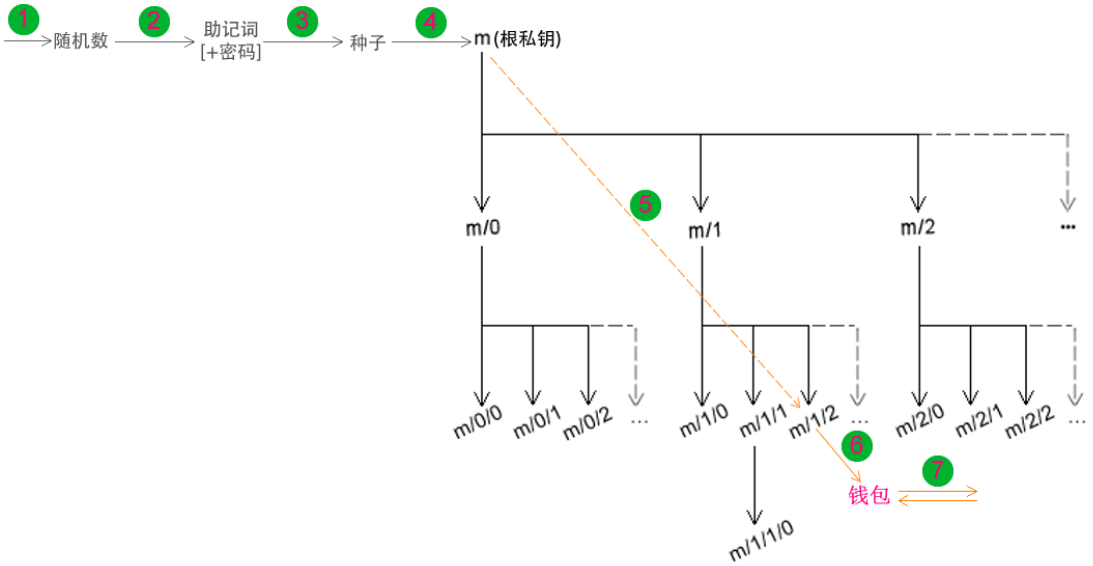


图 5-1 加密币交易流程

大致分为两阶段：钱包管理和放币交易。下面根据图 4-1 中列举的步骤逐次采用各种加密币的 JS 库来实现。

5.1 钱包管理

5.1.1 步骤 1：从虚无到随机数

随机数可采用浏览器的 window.crypto.getRandomValues 或者 nodejs 的 crypto.randomBytes 方法。例如产生 256 位二进制随机数并以 16 进制字符串格式保存在 rand_hex 变量中：

方法 1：在浏览器中使用

```
const array = new Uint8Array(32);
crypto.getRandomValues(array);
let rand_hex = array.toHex();
```

chrome 浏览器不支持数组的 toHex()方法，可用下面的方法（此方法全部浏览器通用）替代：

```
let rand_hex = array.reduce((str, byte) => str + byte.toString(16).padStart(2, '0'), '');
```

由 16 进制字符串转换成无符号字节数组（假设 str_hex 存放了 16 进制字符串）：

```
const aBytes = new Uint8Array(rand_hex.match(/.{1,2}/g).map(byte => parseInt(byte, 16)));
```

方法 2：在 nodejs 中使用

crypto.randomBytes(size[, callback])产生 size 个字节的随机数并以 Buffer 类型返回，如果省略参数 callback 就是同步，否则位异步，回调函数声明为 callback(err, buff)，如果出错，err 保存 Error 类型的错误信息，否则 err 为 null，返回的随机数在 buf 中。

```
//同步方法:
const { randomBytes, } = require('node:crypto');

const buf = randomBytes(32);
rand_hex = buf.toString('hex');
console.log(`${buf.length} 字节的随机数: ${rand_hex}`);
32 字节的随机数: cc168257d1749b3562586adfc3daaed4173ef8b120deef70dda826e6befec746

//异步方法:
const { randomBytes, } = require('node:crypto');

randomBytes(32, (err, buf) => {
   if (err) throw err;
   rand_hex = buf.toString('hex');
   console.log(`${buf.length} 字节的随机数: ${rand_hex}`);
});
32 字节的随机数: 8605c9afb94a8520a17fb9a1d45c692691c349f9b13f1a1603240960ddc22251
```

关于 nodejs 的 Crypto 库更多的方法请浏览 https://nodejs.org/api/crypto.html。

到此我们得到了一个随机数并以十六进制格式的字符串保存在 rand_hex 变量中。

5.1.2 步骤 2：从随机数到助记词

从随机数产生助记词，需要用到 bip39 库的 entropyToMnemonic()方法，此方法的具体介绍请参考 2.5.6 小节。

```
const bip39 = require('bip39');
const mnemonic_en =bip39.entropyToMnemonic(rand_hex);                                    //英文助记词
console.log(mnemonic_en);
slot reduce nominee pepper endless one maximum ask thank burst produce popular initial labor setup bridge
urge manage head evidence hip legal brown tail

const mnemonic_cn = bip39.entropyToMnemonic(rand_hex, bip39.wordlists.chinese_simplified);      //中文助记词
console.log(mnemonic_cn)
晨 寒 碱 株 司 彪 爸 它 霉 西 珠 贫 娘 税 丙 统 傅 铜 句 富 顺 枪 手 悄

const mnemonic_jp = bip39.entropyToMnemonic(rand_hex, bip39.wordlists.japanese);                //日文助记词
console.log(mnemonic_jp)
ひっこし　にんしき　ちまた　てんき　けんない　つのる　たこやき　いとう　ほんやく　おいつく　ないす　としょかん　せいよう　せんろ　はっちゅう　えしゃく　やすみ　たいまつはな　しゃま　こうりつ　しょもつ　そつう　えらぶ゛ほこる
```

只要随机数确定了，助记词就确定了。也可以使用 bip84 库的同名方法产生助记词。一般使用英文助记词，后面的步骤使用 mnemonic_en 变量。

5.1.3 步骤 3：从助记词到种子

从助记词产生种子需要用到 bip39 库的 mnemonicToSeedSync([mnemonic[, password]])方法，此方法返回 64 个字节的种子，可保存在长度为 64 的 Uint8Array 数组中。参数 mnemonic 为助记词，password 为密码，密码是为了保护助记词，如果别人知道了助记词但不知道密码，还是没法获取你的资金，同一个助记词，密码不同则产生的种子也不同，等于后面产生的整个扩展私钥生成树也不同。mnemonic 和 password 都可以省略，此时会自动产生助记词。但是强烈建议 mnemonic 不要省略，否则以后产生的钱包也不是你的了。

例如生成无密码保护的种子：

```
const seed = bip39.mnemonicToSeedSync(mnemonic_en);
console.log(seed.toString('hex'));
8095eae6530ab4d05d4b0743de7e769ed4198f1c7fa1a0c91cb476d52fa795d32d3ac22c99a5bf626906f98b712815a8f743d3d9a2
6b925e3bfb57861bbc26bb
```

或者用密码 password123!保护的种子：

```
const seed_pass = bip39.mnemonicToSeedSync(mnemonic_en, "password123!");
console.log(seed_pass.toString('hex'));
406f5cf5cc61c472b23505014c5b64a5a7b14df7759b9ef68b573f92da3e2cfbff77850afd88f66d2af45fceb0abed17eadb532be3
bb1527faf7f7f32fb95ce8
```

看到了吗？同一组助记词，有无密码生成的种子完全不同，还可以进一步验证，密码不同种子也不同！切记：密码（如果有密码的话）和助记词都不能忘记。

存在 mnemonicToSeedSync()方法的一个异步实现 mnemonicToSeed()，方法的声明一样。例如：

```
bip39.mnemonicToSeed(mnemonic_en, "A1b2C3&").then(
    seed => console.log(seed.toString('hex'))
);
```

只要助记词和密码一样， 异步和同步方法产生的种子也一样。

到此图 4-1 中的前三步已经完成，我们得到了种子——一个 512 位二进制的数字，并保存在类型为 Uint8Array 的数组变量 seed（无密码保护）和 seed_pass（有密码保护）中。

5.1.4 步骤 4：从种子到根节点

需要用到 bip32、tiny-secp256k1 和 bitcoinjs-lib 三个库，下面我们使用受密码保护的种子 seed_pass 来产生 HD 钱包的根节点。

从 Buffer 格式的种子产生根节点：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');
const { BIP32Factory } = require('bip32');
const bip32 = BIP32Factory(ecc);

console.log(seed_pass.toString('hex'));
4b04e49456d2abf518badd4034be04b7f0bae645a0a94e10e641c2e237f4e51a2be5c8c0ec61d62ca9e91483f1b237d4d075300281
76de207649bfdaf4cfdee9

const rootNode = bip32.fromSeed(seed_pass, bitcoin.networks.bitcoin);
```

从十六进制字符串格式的种子恢复根节点：

```
const bitcoin = require('bitcoinjs-lib');
const { BIP32Factory } = require('bip32');
const ecc = require('tiny-secp256k1');
const bs58check = require('bs58check');

const bip32 = BIP32Factory(ecc);
const network = bitcoin.networks.bitcoin;
const seed_hex =
'4b04e49456d2abf518badd4034be04b7f0bae645a0a94e10e641c2e237f4e51a2be5c8c0ec61d62ca9e91483f1b237d4d07530028
176de207649bfdaf4cfdee9';
const seed = Buffer.from(seed_hex,'hex');
var rootNode = bip32.fromSeed(seed, network);
```

从根扩展私钥恢复 HD 钱包的根节点：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');
const { BIP32Factory } = require('bip32');

const bip32Factory = BIP32Factory(ecc);
// 手动定义网络配置
const customNetwork1 = {
   ...bitcoin.networks.bitcoin, // 继承主网配置
   bip32: {
      public: 0x04b24746, // zpub 的版本字节
      private: 0x04b2430c, // zprv 的版本字节
   },
};
const priv_ext =
'zprvAWgYBBk7JR8GjmDKs1WtqZsk8qALxy5M9uJSQuK3BAbuUHPf22PdbYQBnMQL9qna198PeXBa4JrwWvwWHitmh5LkkzFXxq84fqept
ns75EV';
let rootNode = bip32Factory.fromBase58(priv_ext, customNetwork);
```

上面三种方法产生的根节点 rootNode 是相等的，根节点为一 BIP32 类型的对象，主要成员是 32 字节的私钥、32 字节的链码以及若干方法。

从根节点恢复 hd 钱包的根扩展私钥：

```
let rootExt_xprv = root.toBase58();                              //BIP44 的根扩展私钥只是 Base58 编码，以 xprv 开头
console.log(rootExt_prv);
xprv9s21ZrQH143K3Aq6CHweRPgjntsT5j6MKgFzr7XGR9r9N5mCWi4WMR5ujwVAA2UjBrtn9ZzT8z9qkMiNrL4k6byZ2Jrgo1V68PXY7f
1g4Nf
let zprv = '04b2430c';
let payload = bs58check.decode(rootExt_xprv);
let key = payload.slice(4);
let rootExt_zprv = bs58check.encode(Buffer.concat([Buffer.from(zprv,'hex'),key]));//BIP84 的根扩展私钥采用
Base58Check 编码，以 zprv 开头
console.log(rootExt_zprv);
zprvAWgYBBk7JR8GjmDKs1WtqZsk8qALxy5M9uJSQuK3BAbuUHPf22PdbYQBnMQL9qna198PeXBa4JrwWvwWHitmh5LkkzFXxq84fqeptn
s75EV
```

也就是说只要你有种子或者根扩展私钥，都可以恢复 HD 钱包的根节点，有了根节点，就能恢复所有的钱包。

5.1.5 步骤 5：从根节点到钱包节点

有了根节点后，我们可以产生各种加密币的各个账户中的众多钱包节点。上面的操作已经把根节点保存在 rootNode 变量中了。

之前讲过扩展私钥生成树节点编码为 m / purpose' / coin_type' / account' / change / address_index，具体参考 1.5.2 小节。

P2WPKH 钱包节点：

```
const accountPath = "m/84'/0'/0'";                                    //0 号账户路径（P2WPKH 钱包）
const account0 = rootNode.derivePath(accountPath);

const externalNode = account0.derive(0);                                  //外部节点 m/84'/0'/0'/0
const p2wpkhWalletNode0 = externalNode.derive(0);                      //0 号外部钱包节点 m/84'/0'/0'/0/0

const changeNode = account0.derive(1);                                    //内部节点 m/84'/0'/0'/1
const changeWalletNode0 = changeNode.derive(0);                     //0 号找零钱包节点 m/84'/0'/0'/1/0
```

当然找零钱包不一定需要，收币钱包也是可以充当找零钱包的。从 externalNode 可以推导出众多的钱包，即 externalNode.derive(i)，i=0、1、2、3、…，分别代表第 1、2、3、…个钱包。

P2TR 钱包节点（密钥路径花费）：

```
bitcoin.initEccLib(ecc);

const accountPath = "m/86'/0'/0'";                                    //0 号账户路径（P2TR 钱包）
const account0 = rootNode.derivePath(accountPath);
const externalNode = account0.derive(0);                                //收币地址链节点 m/86'/0'/0'/0
const p2trWalletNode0 = externalNode.derive(1);                     //1 号钱收币包节点 m/86'/0'/0'/0/1
```

当然也可以产生 P2PKH 类型的钱包，只不过 0 号账户的路径是 m/44'/0'/0'。

钱包节点为 BIP32 对象，主要的属性有私钥（__D）、公钥（__Q）、链码（chainCode）、深度（__DEPTH）和钱包索引（__INDEX）。比如对于上面的 p2wpkhWalletNode0 钱包节点，其深度为 5，钱包索引为 1。BIP32 对象的详细介绍请参考 2.5.3 小节。

5.1.6 步骤 6：从钱包节点到钱包

有了钱包节点（分别保存在 p2wpkhWalletNode0 和 p2trWalletNode0 变量中），很容易输出钱包（私钥、公钥和地址）。

P2WPKH 钱包：

```
const { address: p2wpkh_address } = bitcoin.payments.p2wpkh({                  //推导出 P2WPKH 类型的收币地址
    pubkey: p2wpkhWalletNode0.publicKey,
    network: bitcoin.networks.bitcoin
});

console.log(`收币钱包: \n  私钥: ${p2wpkhWalletNode0.toWIF()}\n  公钥:
${p2wpkhWalletNode0.publicKey.toString('hex')}\n  地址: ${p2wpkh_address}`);
收币钱包:
  私钥: KxQDMDkdkUocVEae5Leaqhc44NGsm4USMZu6ADjeLfjGpMeTaDRx
  公钥: 02044e84485fb86340bdf1319724cb9eb447e07f8d53369c6793d9182e22b7687c
  地址: bc1qchw6c5ughtmxwwql6tfj9lqg0av5sd28uvjayz

const { address: change_address } = bitcoin.payments.p2wpkh({                  //推导出 P2WPKH 类型的找零地址
    pubkey: changeWalletNode0.publicKey,
    network: bitcoin.networks.bitcoin
});

console.log(`找零钱包: \n  私钥: ${ changeWalletNode0.toWIF()}\n  公钥:
${changeWalletNode0.publicKey.toString('hex')}\n  地址: ${change_address}`);
找零钱包:
  私钥: L48S4eVQuNdPCgPxuwWBAwZnciwgXCpzZUrQLC2BK8VmXLymT44M
  公钥: 02f5f910d01f07a25bfcf200a86e3f6760eafed5cfb4b836d40a7d1fad247c2df1
  地址: bc1qgmp6lrkvwkqxedls5s3qplcrhkepv3mua904yu
```

P2TR 钱包（密钥路径花费）：

```
const { address: p2tr_address } = bitcoin.payments.p2tr({
   internalPubkey: Buffer.from(ecc.xOnlyPointFromPoint(p2trWalletNode0.publicKey)),
   network: bitcoin.networks.bitcoin
});
console.log(`收币钱包: \n  私钥: ${p2trWalletNode0.toWIF()}\n  公钥:
${p2trWalletNode0.publicKey.toString('hex')}\n  地址: ${p2tr_address}`);
收币钱包:
  私钥: KxSiojK75kFfdZXMvde4GzZ64CddURuovuHyxmLkDZPYYgNYbd1o
  公钥: 03bf7751649b411dc8cadcbd17fea01b80850e99795002c2e5b9b30e05aecd2532
  地址: bc1phtjx66w635dmewf2dz8g4uzn63gqq60z6ufjjtm25vnxw5muj0dqg79rva
```

5.1.7 综合：步骤 2~6

使用 bip84 库，我们可以一次性完成所有步骤。下面都是在 nodejs 中执行。

```
const bip84 = require('bip84');

const mnemonic = bip84.generateMnemonic(256);                                    //产生 24 个助记词
console.log('mnemonic:', mnemonic);
mnemonic: glove hybrid mention modify knife boring salmon shy paddle bean tape innocent scheme tool
jealous lamp catch divide polar six cricket bench draft october

//实例化 fromMnemonic 对象:第三个参数 false 表示比特币公网，如果为 true 表示测试网，第四个参数定义币种:
var root = new bip84.fromMnemonic(mnemonic, 'Password123!', false, 0);
//var root = new bip84.fromMnemonic(mnemonic, 'A1b2C3&', true, 0);                //产生比特币测试网钱包
console.log('rootpriv:', root.getRootPrivateKey());                              //显示扩展根私钥
rootpriv: zprvAWgYBBk7JR8GixFRv6YqmvcEFxsCacmxPpLV32Hbeb5uw94nVhbro5ZErCwXFA2c8EU7A6kgSNKkDs1ai3yoUhrhMi5wsWcqJ684vjGQnta
console.log('rootpub:', root.getRootPublicKey());                                //显示扩展根公钥
rootpub: zpub6jftahH18ngZwSKu285r94Yxozhgz5Vom3G5qQhDCvctowPw3Ev7LssihUFy1KBuCJUxPmxsKxoLFQmXSFYkGdfFHRYhAiRenmcQGfB6DBf
console.log('seed:', root.seed.toString('hex'));                                 //由助记词产生的种子
seed: a5dd2346835934b4da457c3ad3eccf6143c14011544e03cfbd5fb0365385c4b6caf7632cb7de9a6fd6a98d1eb9e9bab7f9e81499206eac0477368259572252c0

var child0 = root.deriveAccount(0);                                              //导出节点 m/84'/0'/0'的扩展私钥

var account0 = new bip84.fromZPrv(child0);                                       //从 m/84'/0'/0'的扩展私钥导出账户
console.log("Account 0, root = m/84'/0'/0'");
Account 0, root = m/84'/0'/0'
console.log('Account 0 xprv:', account0.getAccountPrivateKey());                 //显示账户的主私钥
Account 0 xprv: zprvAchcUbW2xpcohmU9r4S35wCrFsUcZcqJ5jaHPT6wqvh2GB1ELB57HPQfE6FB2m97srW2SpLU5SVebKpe57ysfpcQCEZyJMwxbPzwUZ2c8LZ
console.log('Account 0 xpub:', account0.getAccountPublicKey());                  //显示账户的主公钥
Account 0 xpub: zpub6qgxt72voCB6vFYcx5y3T59aouK6y5Z9SxVtBqWZQGE18yLNsiPMqBj95N32bJaJAMMDAixmfWXt4TsqgdP1DxgvkScegJ8j7pfcrDwxHDQ

console.log("Account 0, first receiving address = m/84'/0'/0'/0/0");           //第一个 P2WPKH 收款钱包
Account 0, first receiving address = m/84'/0'/0'/0/0
console.log('Prvkey:', account0.getPrivateKey(0));
Prvkey: L2oxiU55LhzEoy1niTKe2aiqDAAPTaZNWw7McLrwcN9QFVYBTiEa
console.log('Pubkey:', account0.getPublicKey(0));
Pubkey: 022090267b0b8c067505a33bc41bc6d028930ed85d46b64923f33a720e94d5d58e
console.log('Address:', account0.getAddress(0));
Address: bc1qu3adnrjk0x2lcgdg470mu6jv5mf4dff2zg2vpx

console.log("Account 0, second receiving address = m/84'/0'/0'/0/1");          //第二个 P2WPKH 收款钱包
Account 0, second receiving address = m/84'/0'/0'/0/1
console.log('Prvkey:', account0.getPrivateKey(1));
Prvkey: L3Vcp9fRWMM2zy9KCiwcojfdMAGvK42pbynLUNmNrnHPhXRDJpQ6
console.log('Pubkey:', account0.getPublicKey(1));
Pubkey: 02efff191eae15286bd07c13da53b06c24ff77641ffe57ec413b6847dbe95ed3f4
console.log('Address:', account0.getAddress(1));
```

```
Address: bc1qar66rr2zlntqx38hp2l6vlas9cw3ej0dkgy5ge

console.log("Account 0, first receiving address = m/44'/0'/0'/0/0");                    //第一个 P2PKH 收款钱包
Account 0, second receiving address = m/84'/0'/0'/0/1
console.log('Prvkey:', account0.getPrivateKey(0, false, 44));
Prvkey: L2oxiU55LhzEoy1niTKe2aiqDAAPTaZNWw7McLrwcN9QFVYBTiEa
console.log('Pubkey:', account0.getPublicKey(0, false, 44));
Pubkey: 022090267b0b8c067505a33bc41bc6d028930ed85d46b64923f33a720e94d5d58e
console.log('Address:', account0.getAddress(0, false, 44));
Address: 1Mq6EVbaYVAyjJVvGF4TGyzEKJbb8FjbGt

console.log("Account 0, first receiving address = m/49'/0'/0'/0/0");                    //第一个 P2SH 收款钱包
Account 0, second receiving address = m/84'/0'/0'/0/1
console.log('Prvkey:', account0.getPrivateKey(0, false, 49));
Prvkey: L2oxiU55LhzEoy1niTKe2aiqDAAPTaZNWw7McLrwcN9QFVYBTiEa
console.log('Pubkey:', account0.getPublicKey(0, false, 49));
Pubkey: 022090267b0b8c067505a33bc41bc6d028930ed85d46b64923f33a720e94d5d58e
console.log('Address:', account0.getAddress(0, false, 49));
Address: 3KvT6grUdiQU49SSpY9aVxZPshzywp4An1

console.log("Account 0, first receiving address = m/84'/0'/0'/0/0'");              //第一个 P2WPKH 硬化收款钱包
Account 0, first receiving address = m/84'/0'/0'/0/0'
console.log('Prvkey:', account0.getPrivateKey(2^31));
Prvkey: L3hSnjqt4djgEWvPCA5HpweanyM9Dx7VTwNKKqWPqrRxfgVMqWZT
console.log('Pubkey:', account0.getPublicKey(2^31));
Pubkey: 023be1899e00648d747f513f3728b27d9836142dc9a72d569b6f5dfc1227563685
console.log('Address:', account0.getAddress(2^31));
Address: bc1qkhmx82ed2l2ugly0patx9kxcsakf4u0z6ahm7e

console.log("Account 0, first change address = m/84'/0'/0'/1/0");                    //第一个找零钱包
Account 0, first change address = m/84'/0'/0'/1/0
console.log('Prvkey:', account0.getPrivateKey(0, true));
Prvkey: KymLDNhvm3bi7t6X5ggnWgrDTSyiiMCYCVwU4DW9m7UeoNe1ARmw
console.log('Pubkey:', account0.getPublicKey(0, true));
Pubkey: 02cba326e5bf10a94a399e456a2589aac4111eca58bac44b056aeae0486f2af483
console.log('Address:', account0.getAddress(0, true));
Address: bc1qlxs47fflgmu0z7cdvyhufkjl6s6dwdra7gtqy6
```

到此拿到一个账户 account0，以后可随时调用 account0.getPrivateKey()、account0.getPublickKey()和 account0.getAddress()方法获取某个钱包。

如果要产生 P2TR 地址，需要用到 bitcoinjs-lib 库，例如：

```
const bip39 = require('bip39');
const ecc = require('tiny-secp256k1');
const crypto = require('crypto');
const bitcoin = require('bitcoinjs-lib');
const {ECPairFactory} = require('ecpair');
const assert = require('assert');
const { BIP32Factory } = require('bip32');
const bip32 = BIP32Factory(ecc);
const toXOnly = require('bitcoinjs-lib/src/psbt/bip371').toXOnly;                    //从公钥中获取 x 坐标

bitcoin.initEccLib(ecc);
const network = bitcoin.networks.bitcoin;                                            //测试网
bitcoin.networks.testnet

const rand_hex = crypto.randomBytes(32).toString('hex');                    //1: 产生随机数
const mnemonic = bip39.entropyToMnemonic(rand_hex);                         //2:随机数->助记词
//const mnemonic = 'page sad mirror long travel trust zebra exact lucky fence display hope govern report muscle spring ship rival mesh breeze random dust situate absurd'
const seed = bip39.mnemonicToSeedSync(mnemonic, "A1b2C3&");                 //3: 助记词->种子
//seed = '9ef7b63641ee79d37fe27284aaa0fe36b6516de46699c617562f0dcb1a88b268';                    //直接从现有的种子开始
const root = bip32.fromSeed(seed, network);                                 //4: 种子->根节点

const path = "m/86'/0'/0'/0/0";                                             //第一个外部收币钱包的路径
const childNode = root.derivePath(path);                                    //5: 产生子节点 m/86'/0'/0'/0/0
const keyPair = ECPairFactory(ecc).fromPrivateKey(childNode.privateKey, {network});
const { address, output } = bitcoin.payments.p2tr({
    internalPubkey: Buffer.from(toXOnly(keyPair.publicKey)),
    network: network
});                                                                         //6: 产生类型为 P2TR 的地址
```

```
assert.ok(!!output);
const WIF = keyPair.toWIF();
console.log("Taproot 钱包: \n   私钥: ",WIF, "\n   地址: ", address);
Taproot 钱包:
   私钥:  L5LVJozY3AFfeLB8vCHpTqtNKi6r1ZBBnrJZbYzio3j5BkBJSF5s
   地址:  bc1p4afc9suptyj3d2z05krf9akt8aglh56x2gmn4hh35x5th5sxuqtqxez2gh
```

到此，我们有了一个账户 account0，有了账户等于就有了非常多的钱包，钱包编号从 0 开始，直到 2^{32}-1。下面只剩下第 7 步了——放币交易。

一个人可拥有多个账户，每个账户可包含多个钱包，一个钱包就是一个私钥，从私钥可以导出公钥和地址，公钥和地址对外公开，以便让别人往此地址放币。

5.2 放币交易

放币交易就是引用一些 UTXO（未花费的输出），再创造一些输出，全部引用的 UTXO 的资金之和等于全部输出的资金之和加上交易费。<交易 id, 输出序号>能全局确定一个 UTXO，要能成功“消费”一个 UTXO，还需要根据输入钱包的类型提供正确的输入脚本或者见证数据，新创造的输出也要根据对方地址类型构建合适的输出脚本。请参考图 5-2 作进一步理解。

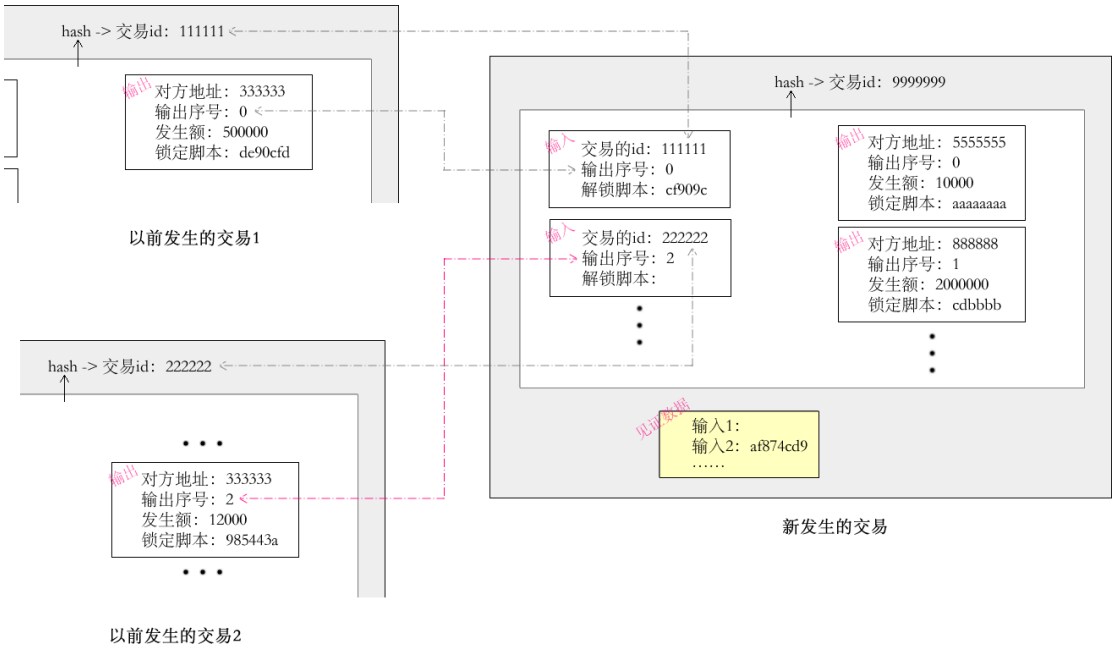


图 5-2 新发生交易的输入引用历史交易的输出

交易双方地址类型不同，构造交易的内容（输出脚本、输入脚本、赎回脚本和见证数据）也不同，请参考表 5-1。

https://www.veryopen.org

https://www.walletoffline.com

表5-1 交易的锁定脚本和解锁脚本

发币方 \ 收币方	P2PK	P2PKH	P2SH	P2WPKH	P2WSH	P2TR
P2PK	**输入：** 解锁脚本：<sig> **输出：** 锁定脚本：<pubKey> OP_CHECKSIG	**输入：** 解锁脚本：<sig> **输出：** 锁定脚本：OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	**输入：** 解锁脚本：<sig> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：<sig> **输出：** 锁定脚本：OP_0 <pubKeyHash>	**输入：** 解锁脚本：<sig> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：<sig> **输出：** 锁定脚本：<x-only-pubKey>
P2PKH	**输入：** 解锁脚本：<sig> <pubKey> **输出：** 锁定脚本：<pubKey> OP_CHECKSIG	**输入：** 解锁脚本：<sig> <pubKey> **输出：** 锁定脚本：OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	**输入：** 解锁脚本：<sig> <pubKey> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：<sig> <pubKey> **输出：** 锁定脚本：OP_0 <pubKeyHash>	**输入：** 解锁脚本：<sig> <pubKey> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：<sig> <pubKey> **输出：** 锁定脚本：<x-only-pubKey>
P2SH	**输入：** 解锁脚本：*OP_0 <sig1> <sig2> ... <sigM> <赎回脚本>* 赎回脚本：*OP_M <pubKey1> <pubKey2> ... <pubKeyN> OP_N OP_CHECKMULTISIG* **输出：** 锁定脚本：*<pubKey> OP_CHECKSIG*	**输入：** 解锁脚本：OP_0 <sig1> <sig2> ... <sigM> <赎回脚本> 赎回脚本：OP_M <pubKey1> <pubKey2> ... <pubKeyN> OP_N OP_CHECKMULTISIG **输出：** 锁定脚本：OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	**输入：** 解锁脚本：OP_0 <sig1> <sig2> ... <sigM> <赎回脚本> 赎回脚本：OP_M <pubKey1> <pubKey2> ... <pubKeyN> OP_N OP_CHECKMULTISIG **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：OP_0 <sig1> <sig2> ... <sigM> <赎回脚本> 赎回脚本：OP_M <pubKey1> <pubKey2> ... <pubKeyN> OP_N OP_CHECKMULTISIG **输出：** 锁定脚本：OP_0 <pubKeyHash>	**输入：** 解锁脚本：OP_0 <sig1> <sig2> ... <sigM> <赎回脚本> 赎回脚本：OP_M <pubKey1> <pubKey2> ... <pubKeyN> OP_N OP_CHECKMULTISIG **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：OP_0 <sig1> <sig2> ... <sigM> <赎回脚本> 赎回脚本：OP_M <pubKey1> <pubKey2> ... <pubKeyN> OP_N OP_CHECKMULTISIG **输出：** 锁定脚本：<x-only-pubKey>
P2WPKH	**输入：** 解锁脚本：'' 见证数据：<sig> <pubKey> **输出：** 锁定脚本：<pubKey> OP_CHECKSIG	**输入：** 解锁脚本：'' 见证数据：<sig> <pubKey> **输出：** 锁定脚本：OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	**输入：** 解锁脚本：'' 见证数据：<sig> <pubKey> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：'' 见证数据：<sig> <pubKey> **输出：** 锁定脚本：OP_0 <pubKeyHash>	**输入：** 解锁脚本：'' 见证数据：<sig> <pubKey> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：'' 见证数据：<sig> <pubKey> **输出：** 锁定脚本：<x-only-pubKey>
P2WSH	**输入：** 解锁脚本：'' 见证数据：<sig1> <sig2> ... <sigM> <赎回脚本> **输出：** 锁定脚本：<pubKey> OP_CHECKSIG	**输入：** 解锁脚本：'' 见证数据：<sig1> <sig2> ... <sigM> <赎回脚本> **输出：** 锁定脚本：OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	**输入：** 解锁脚本：'' 见证数据：<sig1> <sig2> ... <sigM> <赎回脚本> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：'' 见证数据：<sig1> <sig2> ... <sigM> <赎回脚本> **输出：** 锁定脚本：OP_0 <pubKeyHash>	**输入：** 解锁脚本：'' 见证数据：<sig1> <sig2> ... <sigM> <赎回脚本> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：'' 见证数据：<sig1> <sig2> ... <sigM> <赎回脚本> **输出：** 锁定脚本：<x-only-pubKey>
P2TR	**输入：** 解锁脚本：'' 见证数据：<control-block> <script-path (if needed)> <Schnorr签名> **输出：** 锁定脚本：<pubKey> OP_CHECKSIG	**输入：** 解锁脚本：'' 见证数据：<control-block> <script-path (if needed)> <Schnorr签名> **输出：** 锁定脚本：OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	**输入：** 解锁脚本：'' 见证数据：<control-block> <script-path (if needed)> <Schnorr签名> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：'' 见证数据：<control-block> <script-path (if needed)> <Schnorr签名> **输出：** 锁定脚本：OP_0 <pubKeyHash>	**输入：** 解锁脚本：'' 见证数据：<control-block> <script-path (if needed)> <Schnorr签名> **输出：** 锁定脚本：OP_HASH160 <赎回脚本哈希值> OP_EQUAL	**输入：** 解锁脚本：'' 见证数据：<control-block> <script-path (if needed)> <Schnorr签名> **输出：** 锁定脚本：<x-only-pubKey>

如果要新建钱包，建议不要采用较旧的地址类型，比如 P2PK、P2PKH 和 P2SH 尽量不要用，尽可能采用 P2TR、P2WSH 和 P2WPKH，当然收币时尽量采用双方钱包都支持的最新地址类型，即遵循这样的优先次序：P2TR > P2WSH > P2WPKH > P2SH > P2PKH > P2PK，而放币时尽量采用对方钱包支持的最新地址类型。

对于一个具体的加密币钱包来说，其实只有一个操作，就是放币——把钱包里的加密币转给别人。而放币的核心就是发起一个交易，对于比特币来说，就是在交易中明确“要花费哪些 UTXO 并转币到什么地址”这个问题。

从“图 4-19 交易的结构”可知，构造交易的核心是正确填写输入和输出，而输入是难点，因为首先要根据地址判断是否要隔离见证，然后要根据 UTXO 中的输出脚本给出正确的输入脚本或者见证数据，只要存在差错，就不能“花钱”。填写输出虽然相对容易，但是存在巨大风险：如果写错了地址，那么转出去的加密币就永远丢失了；如果搞错了输出脚本，那么转出去的币对方可能没法解锁而永远“花费”不了，也等于永久损失了。尽管填写输入较为复杂，但是至少没什么风险，因为搞错了大不了本次交易失败，只要修正错误再次发起交易即可。

放币主要采用 bitcoinjs-lib 库的 Psbt 类。

首先实例化 Psbt 类，并设置全局参数：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('@bitcoinerlab/secp256k1');
const { ECPairFactory } = require('ecpair');
const ECPair = ECPairFactory(ecc);
bitcoin.initEccLib(ecc);

const psbt = new bitcoin.Psbt({ network: bitcoin.networks.bitcoin });
psbt.setVersion(2);                                                    //设置版本，默认就是 2，所以本语句可省略
psbt.setLocktime(0);                                                   //设置锁定时间，默认为 0，所以本语句可省略
console.log(psbt);
Psbt {
  data: Psbt {
    inputs: [],
    outputs: [],
    globalMap: { unsignedTx: PsbtTransaction {} }
  }
}
```

接下来开始填写输入和输出，下面我们先从输入开始吧。

5.2.1 填写交易中的输入

填写输入需要用到 Psbt 类的方法 addInput()、addInputs()或者 updateInput()，对于这几个方法的介绍请参考“§2.5.1.1 Psbt 类”小节。前两个方法均有一个参数，其类型为 PsbtInputExtended 对象，此对象定义如下：

```
PsbtInputExtended = {
   partialSig?: PartialSig[];
   nonWitnessUtxo?: Buffer;
   witnessUtxo?: WitnessUtxo;
   sighashType?: number;
   redeemScript?: Buffer;
   witnessScript?: Buffer;
   bip32Derivation?: Bip32Derivation[];
   finalScriptSig?: Buffer;
   finalScriptWitness?: Buffer;
   porCommitment?: string;
   tapKeySig?: Buffer;
   tapScriptSig?: TapScriptSig[];
   tapLeafScript?: TapLeafScript[];
```

```
    tapBip32Derivation?: TapBip32Derivation[];
    tapInternalKey?: Buffer;
    tapMerkleRoot?: Buffer;
    hash: string | Buffer;
    index: number;
    sequence?: number;
}

PartialSig {
    pubkey: Buffer;
    signature: Buffer;
}

WitnessUtxo {
    script: Buffer;
    value: number;
}

TapScriptSig extends PartialSig {
    leafHash: Buffer;
}

TapLeafScript extends TapScript {
    controlBlock: Buffer;                                          //脚本路径花费时的控制块
}

TapScript {
    leafVersion: number;                                                    //版本，目前为 0xc0
    script: Buffer;                                                //叶子脚本（脚本路径花费时的锁定脚本）
}

TapBip32Derivation extends Bip32Derivation {
    leafHashes: Buffer[];
}

Bip32Derivation {
    masterFingerprint: Buffer;
    pubkey: Buffer;
    path: string;
}
```

由“图 4-19 交易中输入的结构”可知，PsbtInputExtended 对象的 hash（UTXO 的交易 id）和 index（UTXO 的输出序号）是必须的，其他可以省略（以后再填写或取默认值）。因此 psbt.addInput()方法至少是这样的：

```
psbt.addInput({
    hash: '<UTXO 的交易 id>',
    index: <UTXO 的输出序号>,
});
```

因为“<UTXO 的交易 id>，<UTXO 的输出序号>”唯一确定了一个 UTXO，所以接下来就要查询钱包的 UTXO 了。一个地址对应一个钱包，所以可根据地址查询即可，即 https://api.blockcypher.com/v1/btc/main/addrs/<地址>?unspentOnly=true，如查询地址 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa 的全部 UTXO，浏览 https://api.blockcypher.com/v1/btc/main/addrs/1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa?unspentOnly=true 得到如图 5-3 所示的结果。



图 5-3 查询钱包的 UTXO

可见有三个 UTXO，钱包余额是 6616 聪。或者使用如下代码查询钱包的 UTXO：

```
async function getUtxo(address) {
    let url1 = `https://api.blockcypher.com/v1/btc/main/addrs/${address}?unspentOnly=true`;
    if(address.substring(0, 2) === 'tb'){//测试地址
        url1 = `https://api.blockcypher.com/v1/btc/test3/addrs/${address}?unspentOnly=true`;
    }
//要查询余额 https://api.blockcypher.com/v1/btc/main/addrs/${address}/balance
    const res = await fetch(url1)
    return await res.json();
}

getUtxo('tb1qkdl3ecltxnd3nmgq32lmm56vtarasf94epeka0').then((res)=>console.log(res));          //查询测试网
getUtxo('1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa').then((res)=>console.log(res));                   //查询生产主网
//getUtxo('bc1qujepl0k5n0ga2e86yskvxa6auehpf6dlf84dx0').then((res)=>console.log(res));
```

假如现在要花掉图 3-50 中第二个 UTXO（即金额 1500 聪），于是添加输入的代码为：

```
psbt.addInput({
    hash: 'fb538c3f4893d0d9b3b2bc3bb85b2da86612af5958652b38f88adc3039fc79e3',
    index: 0,
    sequence: 0xffffffff,                                          //顺序号默认值就是 0xffffffff，所以语句可以省略
});
```

一个交易中允许出现多个输入，且各个输入的钱包地址可为不同的地址类型，可能有的是非隔离见证地址（如 P2PK、P2PKH、P2SH），有的是隔离见证地址（如 P2WPKH、P2WSH、P2TR）。只要出现一个隔离见证地址，那么交易中必须增加隔离见证字段（数组型），所有输入必须在隔离见证字段中各占一个数组元素，非隔离见证输入对应的隔离见证数据元素为空值，因为它的输入脚本仍然直接嵌在输入中，而隔离见证输入中的输入脚本字段为空，但是对应的隔离见证数组元素不能为空。

5.2.2 准备输入的输入脚本

输入的输入脚本一般与私钥有关，主要是用私钥签名，本小节先为此后的签名准备脚本。当然与私钥无关的输入脚本历史上也曾发生过，比如猜谜或者寻找一个哈希碰撞的字符串。

引用的 UTXO 所对应的地址类型不同，那么构造的输入脚本也不同，下面我们具体分析。

1. P2PK 类型的输入

属于非隔离见证，需要 UTXO 所在那个交易的完整原始 hex 串来构建输入脚本。采用 updateInput(inputIndex: number, updateData: PsbtInputUpdate)方法来修改刚才添加的那个输入中的输入脚本。

```
const prevTxHex = await getRawTR('fb538c3f4893d0d9b3b2bc3bb85b2da86612af5958652b38f88adc3039fc79e3');
const prevTx = bitcoin.Transaction.fromHex(prevTxHex);
psbt.updateInput(0, {nonWitnessUtxo: prevTx.toBuffer()});
console.log(psbt.txInputs);
[
  {
    hash: <Buffer e3 79 fc 39 30 dc 8a f8 38 2b 65 58 59 af 12 66 a8 2d 5b b8 3b bc b2 b3 d9 d0 93 48 3f 8c
53 fb>,
    index: 0,
    sequence: 4294967295
  }
]
```

函数 getRawTR()请参考“§2.5.1.2 Transaction 类”小节，此函数根据交易 id 获取原始的交易数据（hex 格式）。当然也可以在调用 addInput()方法时直接增加 nonWitnessUtxo 字段，如：

```
psbt.addInput({
    hash: 'fb538c3f4893d0d9b3b2bc3bb85b2da86612af5958652b38f88adc3039fc79e3',
    index: 0,
```

```
    nonWitnessUtxo: prevTx.toBuffer(),
    sequence: 0xffffffff
});
```

2. P2PKH 类型的输入

一种处理方法与 P2PK 完全相同，另一种方法是给 witnessUtxo 参数设置 UTXO 所在那个交易的输出部分的 script 和 value（假设为第一个输出）。看代码：

```
psbt.updateInput(0,
{
    witnessUtxo: {
        script: prevTx.outs[0].script,
        value: prevTx.outs[0].value
    }
});
```

下标 0 表示第一个输出。

3. P2SH 类型的输入

属于多签地址，因此涉及到赎回脚本。首先需要构建赎回脚本，下面的例子假如是 2-of-3 多签：

```
const pubKey1 = Buffer.from('03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6', 'hex');
const pubKey2 = Buffer.from('035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7', 'hex');
const pubKey3 = Buffer.from('03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4', 'hex');
const p2ms = bitcoin.payments.p2ms({                                                    //构建 2-of-3 多签脚本
    m: 2,                                                                                  //需要 2 个签名
    pubkeys: [pubKey1, pubKey2, pubKey3],                                                    //公钥列表
    network: bitcoin.networks.bitcoin,                                                     //比特币网络
});
```

到此我们获得了赎回脚本（Redeem Script）：p2ms.output。

接下来我们修改输入：

```
const prevTxHex = await getRawTR('fb538c3f4893d0d9b3b2bc3bb85b2da86612af5958652b38f88adc3039fc79e3');
const prevTx = bitcoin.Transaction.fromHex(rawTR_hex);
psbt.updateInput(0, {
    nonWitnessUtxo: prevTx.toBuffer(),
    redeemScript: p2ms.output,
});
```

函数 getRawRT()请参考“§2.5.1.2 Transaction 类”小节。

4. P2WPH 类型的输入

属于隔离见证地址。给参数 witnessUtxo 设置值即可：

```
const prevTxHex = await getRawTR('fb538c3f4893d0d9b3b2bc3bb85b2da86612af5958652b38f88adc3039fc79e3');
const prevTx = bitcoin.Transaction.fromHex(prevTxHex);
psbt.updateInput(0,{
    witnessUtxo: {
        script: prevTx.outs[0].script,
        value: prevTx.outs[0].value
    }
});
```

5. P2WSH 类型的输入

属于隔离见证多签地址。需要给 witnessUtxo 和 witnessScript 参数设置值，witnessUtxo 参数的设置与 P2WPKH 一样，而 witnessScript 的值就是赎回脚本。看代码：

```
const pubKey1 = Buffer.from('03f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f6', 'hex');
const pubKey2 = Buffer.from('035ceea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f7', 'hex');
const pubKey3 = Buffer.from('03b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a4', 'hex');
const p2ms = bitcoin.payments.p2ms({                                          //构建 2-of-3 多签脚本
   m: 2,                                                                      //需要 2 个签名
   pubkeys: [pubKey1, pubKey2, pubKey3],                                      //公钥列表
   network: bitcoin.networks.bitcoin,                                         //比特币网络
});
console.log(p2ms.output.toString('hex'));                                     //打印赎回脚本
522103f56ea6f1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393f621035ceea6f1261086bcaf20134ea90d3170
895f2d4cd4ea9e2d20f52d2a67f393f72103b87af9c1261086bcaf20134ea90d3170895f2d4cd4ea9e2d20f52d2a67f393a453ae
const witnessScript = p2ms.output;
const prevTxHex = await getRawTR('fb538c3f4893d0d9b3b2bc3bb85b2da86612af5958652b38f88adc3039fc79e3');
const prevTx = bitcoin.Transaction.fromHex(prevTxHex);
psbt.updateInput(0,{
   witnessUtxo: {
      script: prevTx.outs[0].script,                                          //UTXO 的输出脚本（P2WSH 类型）
      value: prevTx.outs[0].value
   },
   witnessScript: witnessScript,
});
```

6. P2TR 类型的输入

属于隔离见证地址，支持多签。对于 P2TR 类型的 UTXO，存在两种花费方式：一是密钥路径花费，二是脚本路径花费。需要获取前一笔交易的输出数据用来设置 witnessUtxo 参数，以及用 Taproot 的内部公钥来设置 tapInternalKey 参数，tapLeafScript 参数是可选的，它定义多签脚本的路径。

1. 密钥路径花费：

```
const toXOnly = require('bitcoinjs-lib/src/psbt/bip371').toXOnly;                    //从公钥中获取 x 坐标

const prevTxHex = await getRawTR('fb538c3f4893d0d9b3b2bc3bb85b2da86612af5958652b38f88adc3039fc79e3');
const prevTx = bitcoin.Transaction.fromHex(prevTxHex);
const publicKey = '0299dc20b67d59d7c9bfe7500994a53999948933675cd82a54f84dcd14a0210e98c';
const internalPublickey = ecc.xOnlyPointFromPoint(Buffer.from(publicKey,'hex'));          //获取公钥的 x 坐标
psbt.updateInput(0,{
   witnessUtxo: {
      script: prevTx.outs[0].script,                                          //UTXO 的输出脚本（P2TR 类型）
      value: prevTx.outs[0].value
   },
   tapInternalKey: internalPublickey                                          //内部公钥就是 x 坐标值
});
```

函数 getRawRT()请参考“§2.4.1.2 Transaction 类”小节。

2. 脚本路径花费：

不再提供 tapInternalKey 参数，而提供 tapLeafScript: TapLeafScript[]参数，接口 TapLeafScript 定义如下：

```
interface TapLeafScript {
   leafVersion: number,                              //bip341.LEAF_VERSION_TAPSCRIPT 宏，目前值为 192（0xc0）
   script: Buffer,                                                            //MAST 树上一个叶子的脚本
   controlBlock: Buffer                                                                    //控制块
}
```

控制块脚本格式<Control byte with leaf version and parity bit> <Internal key P> <从叶子到根路径上的兄弟>。请参考下面的例子。

```
const bip341 = require('bitcoinjs-lib/src/payments/bip341');

const prevTxHex = await getRawTR('fb538c3f4893d0d9b3b2bc3bb85b2da86612af5958652b38f88adc3039fc79e3');
const prevTx = bitcoin.Transaction.fromHex(prevTxHex);
const publicKey1 = '0299dc20b67d59d7c9bfe7500994a5399948933675cd82a54f84dcd14a0210e98c';
const publicKey2 = '031dd06c3b052e60c80062d20d15b7b0031412d61e0ccfa706a752c05a629bb0d7';
const internalPubkey = Buffer.from(publicKey1,'hex');
const leafPubkey = Buffer.from(publicKey2,'hex');
const leafScriptAsm = `${publicKey2} OP_CHECKSIG`;
const leafScript = bitcoin.script.fromASM(leafScriptAsm);

//构建默克尔抽象语法树（MAST）树:
const scriptTree = [
   [
      {
         output: bitcoin.script.fromASM(
            '50929b74c1a04954b78b4b6035e97a5e078a5a0f28ec96d547bfee9ace803ac0 OP_CHECKSIG',
         ),
      },
      [
         {
            output: bitcoin.script.fromASM(
               '50929b74c1a04954b78b4b6035e97a5e078a5a0f28ec96d547bfee9ace803ac1 OP_CHECKSIG',
            ),
         },
         {
            output: bitcoin.script.fromASM(
               '2258b1c3160be0864a541854eec9164a572f094f7562628281a8073bb89173a7 OP_CHECKSIG',
            ),
         },
      ],
   ],
   [
      {
         output: bitcoin.script.fromASM(
            '50929b74c1a04954b78b4b6035e97a5e078a5a0f28ec96d547bfee9ace803ac2 OP_CHECKSIG',
         ),
      },
      [
         {
            output: bitcoin.script.fromASM(
               '50929b74c1a04954b78b4b6035e97a5e078a5a0f28ec96d547bfee9ace803ac3 OP_CHECKSIG',
            ),
         },
         [
            {
               output: bitcoin.script.fromASM(
                  '50929b74c1a04954b78b4b6035e97a5e078a5a0f28ec96d547bfee9ace803ac4 OP_CHECKSIG',
               ),
            },
            {
               output: leafScript,
            },
         ],
      ],
   ],
];

// bip341.LEAF_VERSION_TAPSCRIPT=192 或者 0xc0
const redeem = { output: leafScript, redeemVersion: bip341.LEAF_VERSION_TAPSCRIPT,};

const { output, witness } = bitcoin.payments.p2tr({
   internalPubkey: ecc.xOnlyPointFromPoint(internalPubkey),
   scriptTree,
   redeem,
   network: bitcoin.networks.bitcoin,
});                                                                    //output 为输出脚本，witness 为见证数据

console.log(bitcoin.script.toASM(output));                                  //反汇编输出脚本看看长什么样子
OP_1 e7cb05ae80495226f39b83a25fd1a0c53bfd5b383bee9160b0beb909c159591a

psbt.updateInput(0,{
   witnessUtxo: {
```

```
      script: output, ------------------------------------------------------------ //UTXO 的输出脚本（P2TR 类型）
      value: prevTx.outs[0].value
   },
   tapLeafScript: [
      {
         leafVersion: redeem.redeemVersion,
         script: redeem.output,
         controlBlock: witness[witness.length - 1],
      },
   ],
});

console.log(psbt.data.inputs[0]);
{
  unknownKeyVals: [],
  witnessUtxo: {
    script: <Buffer 51 20 e7 cb 05 ae 80 49 52 26 f3 9b 83 a2 5f d1 a0 c5 3b fd 5b 38 3b ee 91 60 b0 be b9
09 c1 59 59 1a>,
    value: 1500
  },
  tapLeafScript: [
    {
      leafVersion: 192,
      script: <Buffer 21 03 1d d0 6c 3b 05 2e 60 c8 00 62 d2 0d 15 b7 b0 03 14 12 d6 1e 0c cf a7 06 a7 52
c0 5a 62 9b b0 d7 ac>,
      controlBlock: <Buffer c1 99 dc 20 b6 7d 59 d7 c9 bf e7 50 09 94 a5 39 99 48 93 36 75 cd 82 a5 4f 84
dc d1 4a 02 10 e9 8c b4 24 de a0 9f 84 0b 93 2a 00 37 3c dc db d2 56 50 ... 111 more bytes>
    }
  ]
}

console.log(bitcoin.script.toASM(psbt.data.inputs[0].tapLeafScript[0].script));                    //反汇编
'031dd06c3b052e60c80062d20d15b7b0031412d61e0ccfa706a752c05a629bb0d7 OP_CHECKSIG'
```

关于 P2TR 脚本路径花费更详细的介绍，请参考“§3.5.5 P2TR 脚本路径”小节。

5.2.3 填写交易中的输出

添加输出采用方法 psbt.addOutput(outputData: PsbtOutputExtended): this 和 psbt.addOutputs(outputDatas: PsbtOutputExtended[]): this。其中 PsbtOutputExtended 定义如下：

```
PsbtOutputExtended = {
   redeemScript?: RedeemScript;
   witnessScript?: WitnessScript;
   bip32Derivation?: Bip32Derivation[];
   tapBip32Derivation?: TapBip32Derivation[];
   tapTree?: TapTree;
   tapInternalKey?: TapInternalKey;
   unknownKeyVals?: KeyValue[];
   address: string;
   script: Buffer;
   value: number;                                                                    //比特数量（单位聪）
};                                                                                   //address 和 script 只能选择一个
```

可见必须的项目是{address, value}或者{script, value}，前者使用更普遍，即：

```
psbt.addOuput({
   address: '<地址>',
   value: <比特币数量（单位聪），
});
```

如向地址 1LmKctEVmYwKc3h1BVgbgCYw8hgNGKRoqu 转 900000 聪（0.009 个比特）：

```
psbt.addOutput({
   address: '1LmKctEVmYwKc3h1BVgbgCYw8hgNGKRoqu',
```

```
    value: 900000,
});
```

不过地址千万不要搞错了，转错了地址资金就永远丢失了。金额也不要搞错。

5.2.4 对输入签名

各个输入允许用不同的私钥签名，且同一个输入允许多个签名。签名可用 Psbt 类的如下方法：

```
signInput(inputIndex: number, keyPair: Signer, sighashTypes?: number[]): this;
signInputAsync(inputIndex: number, keyPair: Signer | SignerAsync, sighashTypes?: number[]): Promise<void>;

signAllInputs(keyPair: Signer, sighashTypes?: number[]): this;
signAllInputsAsync(keyPair: Signer | SignerAsync, sighashTypes?: number[]): Promise<void>;

signAllInputsHD(hdKeyPair: HDSigner, sighashTypes?: number[]): this;
signAllInputsHDAsync(hdKeyPair: HDSigner | HDSignerAsync, sighashTypes?: number[]): Promise<void>;

signTaprootInput(inputIndex: number, keyPair: Signer, tapLeafHashToSign?: Buffer, sighashTypes?: number[]): this;
signTaprootInputAsync(inputIndex: number, keyPair: Signer | SignerAsync, tapLeafHash?: Buffer, sighashTypes?:
number[]): Promise<void>;
```

*Async()为异步方法。如果所有输入都是同一个钱包，可采用 signAllInputs()或 signAllInputsAsync()一次性完成签名，否则只能调用 signInput()或 signInputAsync()对每一个输入单独签名。参数 sighashTypes 指明被签名的内容包含交易的哪些字段，默认是 SIGHASH_ALL，具体参考“§3.5.3.3 交易签名”小节中的“表 3-33 SIGHASH 取值”。

其中接口 Signer 定义如下：

```
interface Signer {
    network?: any;
    publicKey: Uint8Array;
    getPublicKey?(): Uint8Array;
    sign(hash: Uint8Array, lowR?: boolean): Uint8Array;
    signSchnorr?(hash: Uint8Array): Uint8Array;
}
```

ECPair 对象实现了 Signer 接口，而 ECPair 对象可用下面的代码实例化：

```
const bitcoin = require('bitcoinjs-lib');
const ecc = require('tiny-secp256k1');
const { ECPairFactory } = require('ecpair');
const ECPair = ECPairFactory(ecc);

const privateKey = 'L3MLmwGxE9GuZiE9Gf17VSgYtBnMAMGuZ36yBRWR3WbooyJe4onD';                    //WIF 格式私钥
const keyPair = ECPair.fromWIF(privateKey, bitcoin.networks.bitcoin);          //keyPair 就是 ECPair 的实例化
//const privateKey = 'afbb9ce6e4e3766a751a23c78067ea180a852bfefe528ff7b93fa2c672d08d60';          //原始私钥
//const keyPair = ECPair.fromPrivateKey(Buffer.from(privateKey,'hex'), bitcoin.networks.bitcoin);
```

如果你的私钥是原始格式，就调用方法 ECPair.fromPrivateKey()实例化 ECPair 对象。

有了 keyPair 后，再来签名：

```
//接上面的代码……
psbt.signInput(0, keyPair);
```

或者采用异步方式：

```
sbt.signInputAsync(0, keyPair).then(() => {
    // 验证签名
    psbt.validateSignaturesOfInput(0);

    // 完成签名
    psbt.finalizeAllInputs();
```

```
    // 提取最终的交易
    const tx = psbt.extractTransaction();

    // 输出交易的 Hex 编码
    console.log('交易 Hex:', tx.toHex());
}).catch((error) => {
    console.error('签名失败:', error);
});
```

对于是 P2TR 的密钥路径花费，需要调整签名的密钥对：

```
const signPair = tweakSigner(keyPair, { network: network });
psbt.signInput(0, signPair);

function tweakSigner(keyPair, opts) {
    let privateKey = keyPair.privateKey;
    if (!privateKey) {
        throw new Error('Private key is required for tweaking signer!');
    }
    if (keyPair.publicKey[0] === 3) {                                                    //奇数私钥
        privateKey = ecc.privateNegate(privateKey);
    }

    let tweakedPrivateKey = ecc.privateAdd(
        privateKey,
        tapTweakHash(ecc.xOnlyPointFromPoint(keyPair.publicKey)),
    );

    if (!tweakedPrivateKey) {
        throw new Error('Invalid tweaked private key!');
    }

    return ECPair.fromPrivateKey(Buffer.Buffer.from(tweakedPrivateKey), {
        network: opts.network || network
    });
}

function tapTweakHash(pubKey, h) {
    return bitcoin.crypto.taggedHash(
        'TapTweak',
        Buffer.Buffer.concat(h ? [pubKey, h] : [pubKey]),
    );
}
```

5.2.5 终结输入

终结输入调用方法 finalizeAllInputs ()、finalizeInput()或者 finalizeTaprootInput()，不过 finalizeInput()会根据输入的类型判断要不要调用 finalizeTaprootInput()，所以不管输入是什么类型，都可以调用 finalizeInput()完成输入终结处理。方法 clearFinalizedInput()作逆终结处理。输入终结主要完成两个任务：(1)生成输入脚本或者见证数据；(2) 删除 PSBT 中不再需要的中间数据（如 partialSig、bip32Derivation 等)。

不过在作终结处理之前最好调用方法 validateSignaturesOfAllInputs(validator: ValidateSigFunction)验证所有输入的签名，其中参数 validator 为验证函数，可用下面代码定义：

```
const ecc = require('tiny-secp256k1');
const { ECPairFactory } = require('ecpair');
const ECPair = ECPairFactory(ecc);
//接上一小节的代码……
const validator_ecdsa = ( pubkey, msghash, signature) => ECPair.fromPublicKey(pubkey).verify(msghash,
signature);                                                                          //对于 ECDSA 签名
const validator_schnorr = (pubkey, msghash, signature) => tiny-secp256k1.verifySchnorr(msghash, pubkey,
signature);                                                                          //对于 Schnorr 签名
```

```
psbt.validateSignaturesOfAllInputs(validator_ecdsa);                    //验证签名
psbt.finalizeAllInputs();                                               //终结所有输入
```

5.2.6 提取事务

到此一个 PSBT 对象实例构造完成了，接下来就是从中提取事务，并生成可发布的 hex 格式交易。提取事务调用方法 extractTransaction(disableFeeCheck?: boolean): Transaction，返回 Transaction 类的实例，参数 disableFeeCheck 指明是否检查交易费用，true-不检查，false-检查，默认是 false。关于 Transaction 类的详细介绍请参考“§2.4.1.2 Transaction 类”小节。

```
//接上一小节的代码……
const tx = psbt.extractTransaction();
const txHex = tx.toHex();
console.log(txHex);
0100000001 10df9fa370de9dfbfcf6da91863f891659e6e7705c28f88a453723d272ddb98a000000008a47304402205aaaa23d7ba69eb6d233cd31ab242d09b6ab1a325b8e406e55d9e5575832734102204eec8e7669241249b4f597d9dfe34a9a25cc61f0f55f1c63b8f22df47371836d014104b6b6509ee40cb6947af267d3cd7da306dbb4fb2691fc30c1e76ede0b3e533980c87e8a4d382f80b2338fcb15460af964428662aa144e02e52047637e7900b18cfeffffff0128020000000000001976a914884377972c0df175fc1ef878a490d701c0fd195e88ac18090800
```

5.2.7 发布交易

正式发布交易前，最好把 Hex 格式的交易拷贝到 https://live.blockcypher.com/btc/decodetx/上解码，然后仔细检查交易内容，尤其要确保交易费用和输出中的地址、输出脚本、金额无误。交易费用控制在 1 美元上下，直接用公式计算即可：100000000 ÷ 比特币价格（美元） = xxxx（聪），例如现在一个比特币的价格是 84365 美元，那么 1 美元的交易费转化为“聪”就是100000000 ÷ 84365 ≈ 1185 聪。转账地址错了就直接把比特币转错了人，资金永远丢失了。输出脚本错了，那么对方永远无法解锁，等于资金被永久冻结了。图 5-4 为解码后的结果。

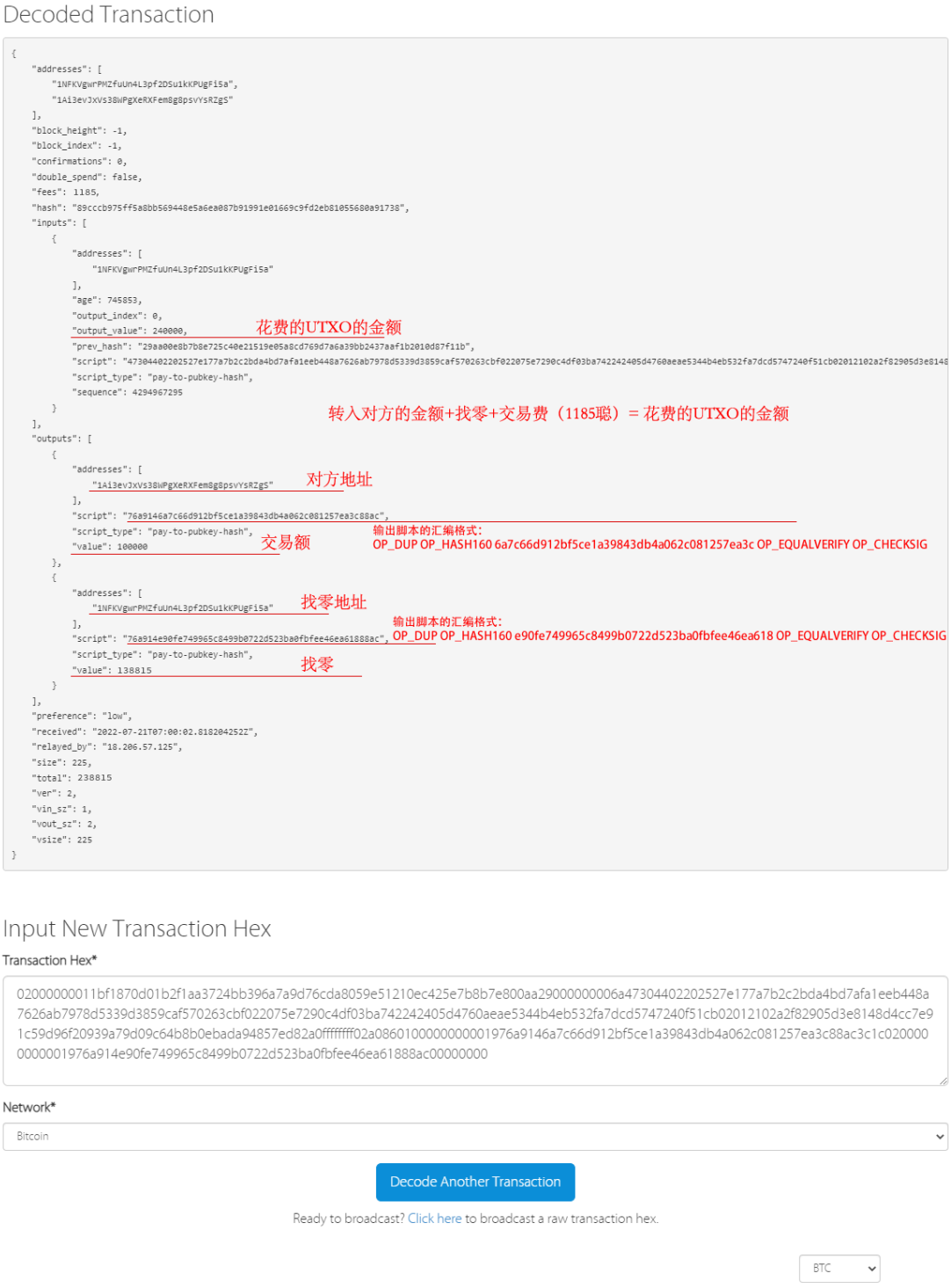


图 5-4 解码 Hex 格式的交易

检查交易无误后有两种方法发布交易。

方法 1：通过网站在线发布

把经过解码后再三确认的 Hex 格式交易拷贝到这些网站 https://www.walletoffline.com/crypto/ 、https://live.blockcypher.com/btc/pushtx/、https://www.blockchain.com/btc/pushtx 和 https://blockchair.com/broadcast 当中的任何一个网站上去发布，参见图 5-5。

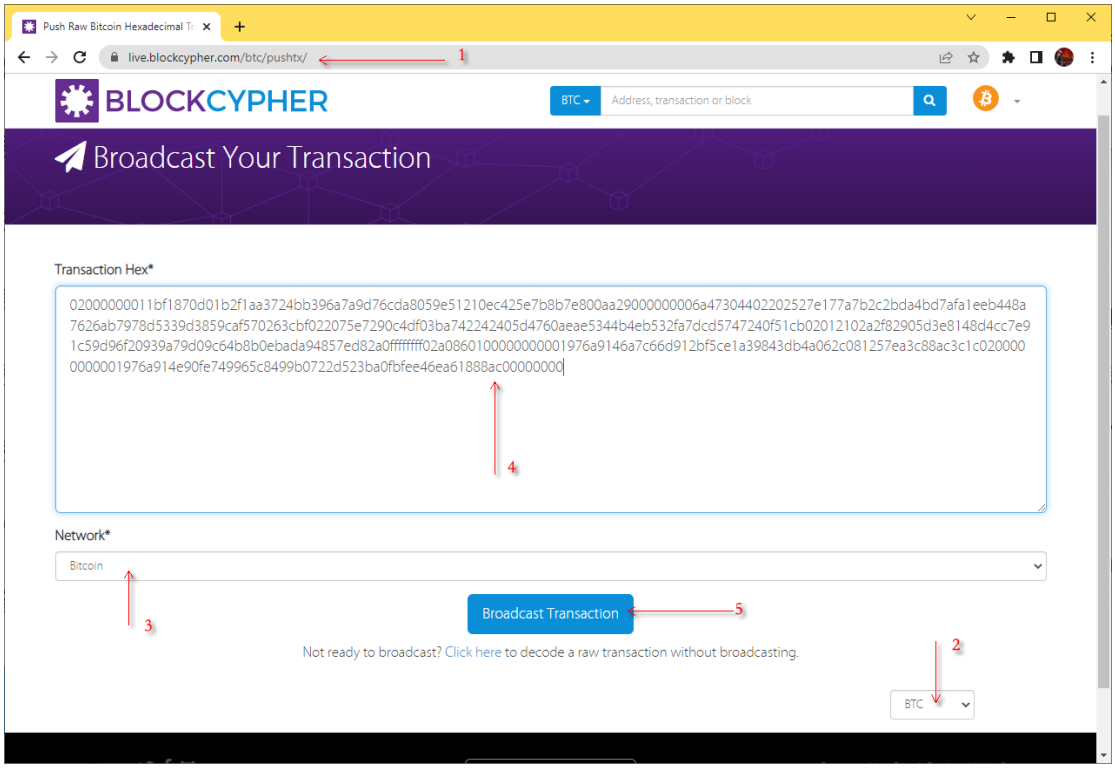


图 5-5 正式发布交易

发布交易后的页面如图 5-6 所示。

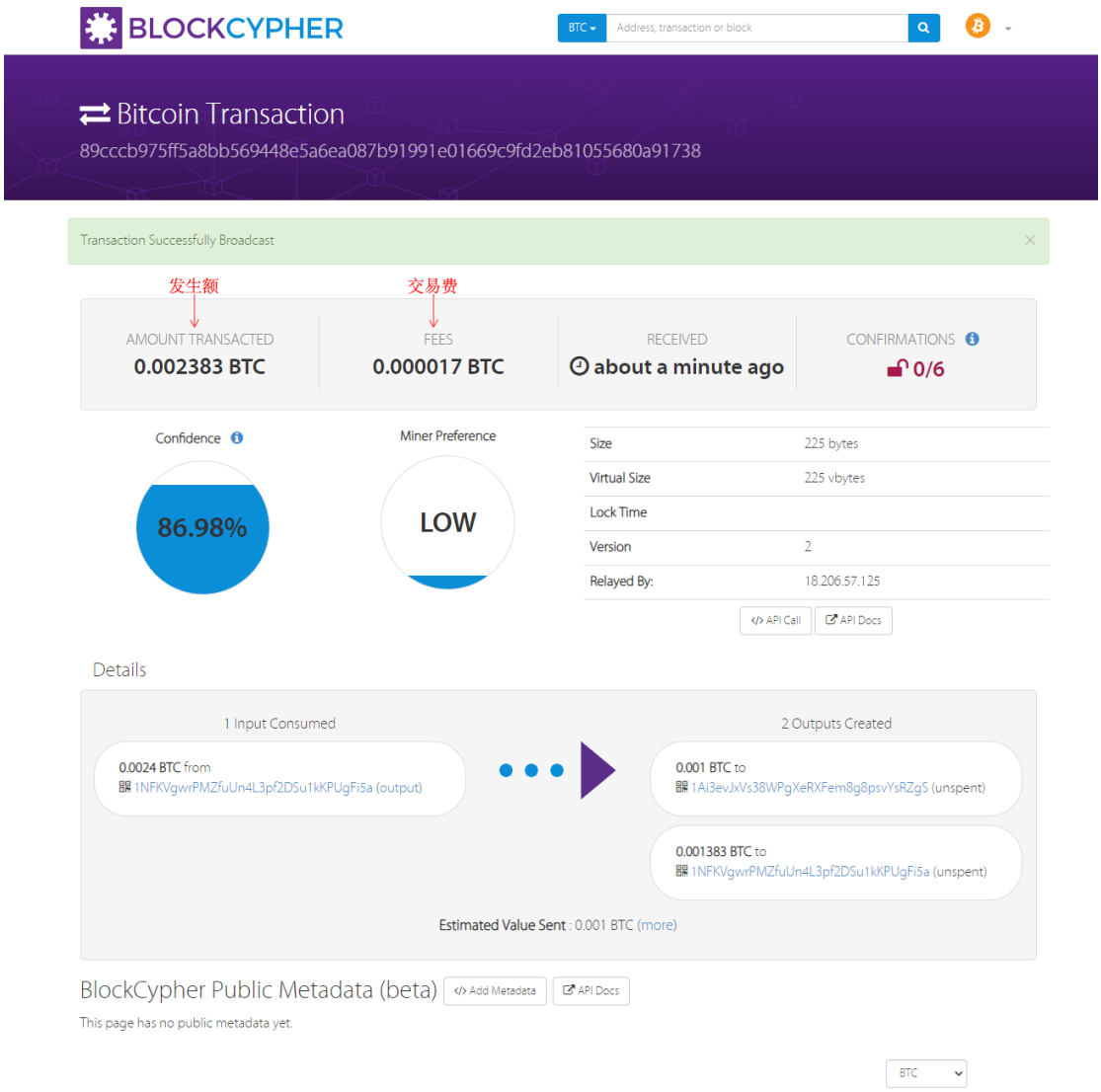


图 5-6 交易成功发布

交易成功发布并不代表交易已经完成，因为给的交易费比较低，交易完成的时间比较长，有时需要几个小时。

方法 2：通过代码发布

假如最后以十六进制表示的交易保存在 txHex 变量中，然后用下面的代码发布交易：

首先安装 axios:

```
npm install axios
```

然后在 nodejs 中执行以下代码发布交易：

```
const axios = require('axios');
const blockstream = new axios.Axios({baseURL: `https://blockstream.info/api`});
blockstream.post('/tx', txHex).then(res => {console.log(res)});
```

如要在浏览器中使用，请从 https://www.cdnpkg.com/axios 下载最新版的 axios。

或者执行下面的代码发布交易：

```
//tx: 已经签名的交易的 Hex 编码串，isTestNetwork: true-测试网络，否则生产网络。
async function broadcastTx(tx, isTestNetwork){
    const net = (isTestNetwork || false) ? 'test3': 'main';
    const res = await fetch(
        `https://api.blockcypher.com/v1/btc/${net}/txs/push`,
        {
            method: 'POST',
            body: JSON.stringify({ tx, }),
            headers: { 'Content-Type': 'application/json' }
        }
    );
    return await res.json();
}
broadcastTx(txHex, false);
```

本例子花费的 utxo 金额为 0.0024 比特，转给对方 0.001 比特，交易费为 0.000017 比特，所以找回零钱 0.001383 比特。

交易成功发布后，过几分钟，再去 https://www.blockchain.com 或 https://blockchair.com/zh 查询交易是否完成。图 5-7 查询对方到账情况。

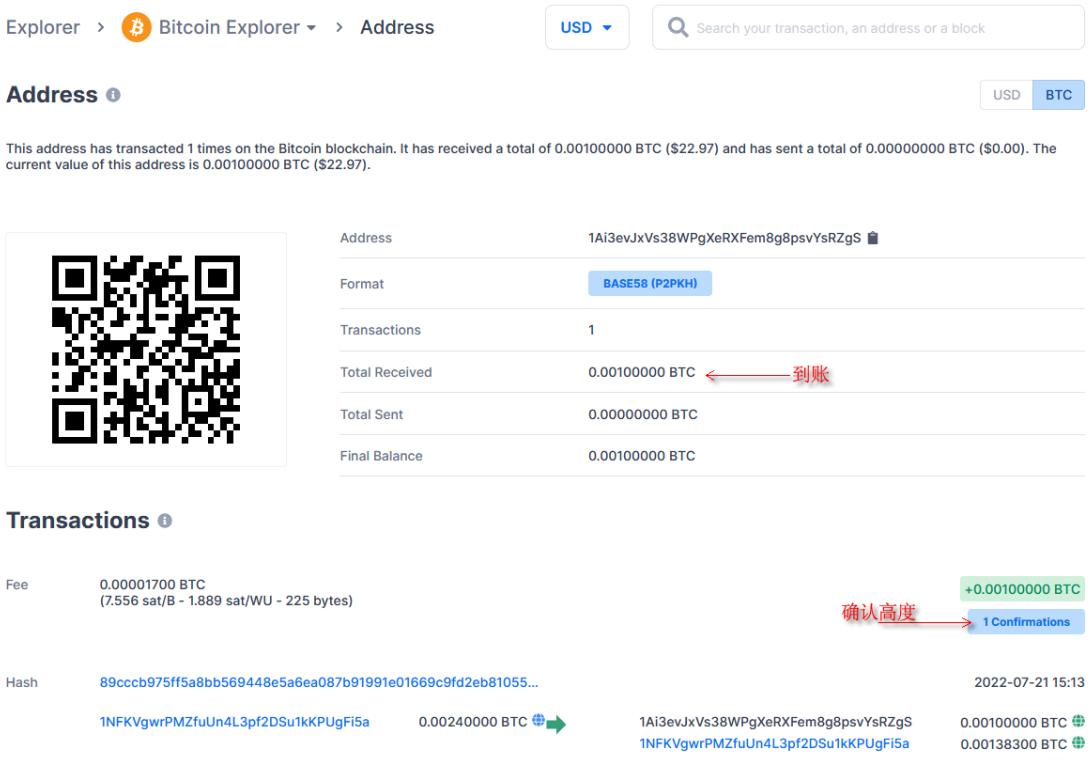


图 5-7 对方到账成功

再来查询找零是否完成，参见图 5-8。


Address
USD
BTC
This address has transacted 3 times on the Bitcoin blockchain. It has received a total of 0.24220492 BTC ($5,563.79) and has sent a total of 0.00240000 BTC ($55.13). The current value of this address is 0.23980492 BTC ($5,508.66).
Address 1NFKVgwrPMZfuUn4L3pf2DSu1kKPUgFi5a
Format BASE58 (P2PKH)
Transactions 3
Total Received 0.24220492 BTC
Total Sent 0.00240000 BTC
Final Balance 0.23980492 BTC
Transactions
Fee 0.00001700 BTC
(7.556 sat/B - 1.889 sat/WU - 225 bytes)
本钱包花费总计
-0.00101700 BTC
确认高度
1 Confirmations
Hash 89cccb975ff5a8bb569448e5a6ea087b91991e01669c9fd2eb81055...
2022-07-21 15:13
1NFKVgwrPMZfuUn4L3pf2DSu1kKPUgFi5a 0.00240000 BTC
1Ai3evJxVs38WPgXeRXFem8g8psvYsRZgS 0.00100000 BTC
1NFKVgwrPMZfuUn4L3pf2DSu1kKPUgFi5a 0.00138300 BTC
找零

图 5-8 找零完成

6 初级操作环境

如果投入的资金不多（比如 100 万以内），并且不太在乎隐藏自己的身份，而且掌握的计算机技术有限，建议采用初级操作环境。

初级操作环境的环境是采用脱机工作的 firefox 浏览器做交易签名，然后把已签名的交易发布到网上。

6.1 准备环境

请你首先确保第 2 章前四小节已经完成。在 firefox 浏览器的隐私窗口中打开 index.html 页面，首先看到如图 6-1 所示的页面。

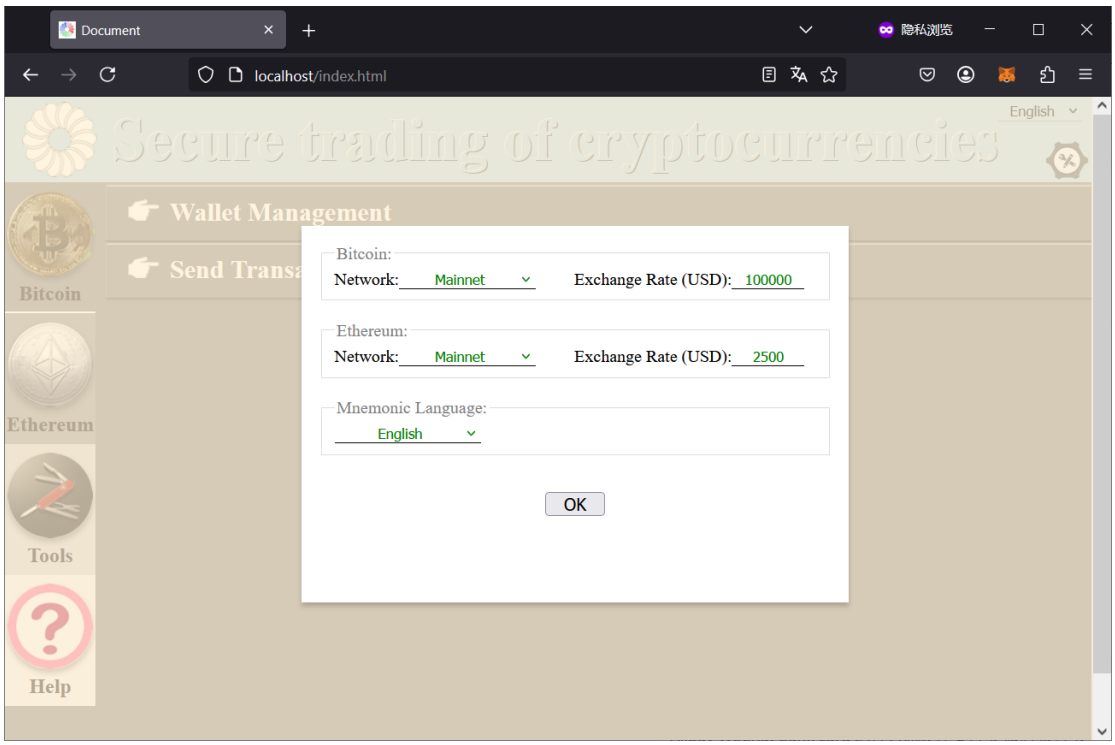


图 6-1 首页面

这里选择比特币和以太币网络，默认是比特币主网（生产网），其他选项还有：比特币公共测试网和比特币本地测试网，如果你只是想测试比特币交易，请选择后两者，如果你本地搭建了比特币的全功能节点，就选择比特币本地测试网，否则选择比特币公共测试网。比特币主网就是真实的比特币区块链，这里发生的交易都是真金白银，操作时要小心谨慎！

比特币汇率是以美元计价的，输入当下的真实汇率即可，这个是用来换算交易费用的，比如交易时，你愿意出 1 美元的交易费，系统会根据汇率自动换算成多少聪。

点击图 6-1 中的“确认”按钮后出现图 6-2 所示的画面。

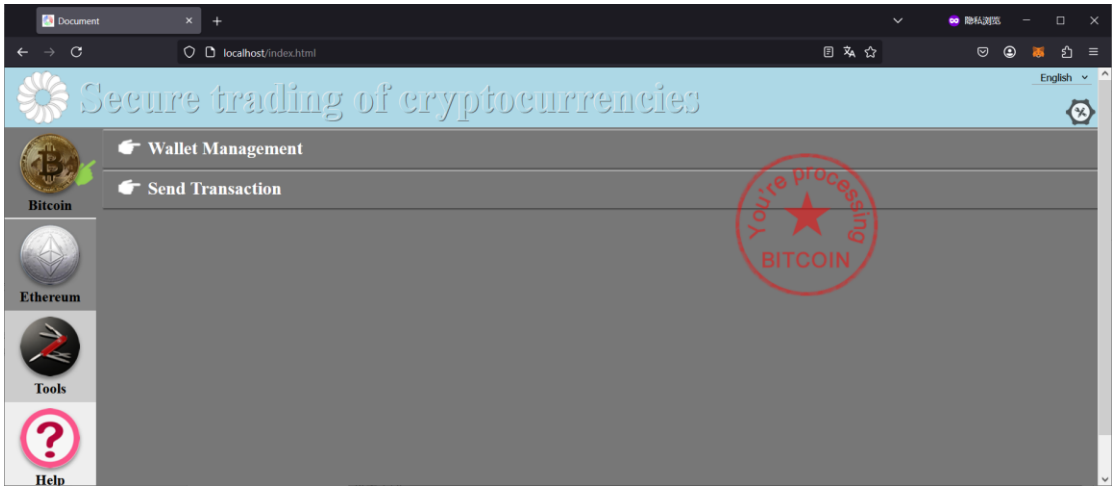


图 6-2 加密比安全交易主页

在右上角处可以修改比特币和以太币网络以及它们各自兑美元的汇率，默认是英语，点击右上角的语言选择，比如选择简体中文后的界面，参见图 6-3 所示操作。



图 6-3 修改比特币汇率

如果出现警告‘小心：浏览器处于“在线工作”状态。’，表示浏览器已经连上了互联网，在签名时一定要让浏览器处于脱机工作状态，具体操作是：按一下 Alt 键浏览器最上面会弹出菜单，然后点击“文件”中的“脱机工作”即可（前面出现√），参见图 6-4 所示。



图 6-4 浏览器切换到脱机工作模式

建议你不要马上交易，先花点时间认真看看“帮助”中的内容（点击图 6-3 中左侧的“帮助”），参见图 6-5。



图 6-5 仔细阅读帮助

6.2 比特币

6.2.1 钱包管理

点击左侧导航条上的“比特币”，然后点击“钱包管理”，打开如图 6-6 所示的页面。



图 6-6 钱包管理主页面

6.2.1.1 创建钱包/恢复钱包

点击图 6-6 中的“新建钱包 / 恢复钱包”，打开如图 6-7 所示的画面。



图 6-7 “新建钱包&恢复钱包”页面

蓝色箭头指到的地方是新建钱包时会涉及的，默认是英文的 24 个单词的助记词、无密码产生种子、0 号账户、0 号钱包。绿色箭头部分与恢复钱包有关，即恢复钱包有三种方法：(1) 从助记词恢复；(2) 从种子恢复；(3) 从根扩展私钥恢复。

1. 新建钱包

假如有这些要求：由 21 个单词组成的助记词、用密码 a12345678!保护助记词、1 号账户、8 号钱包。输入相应的内容后点击“新建钱包”按钮，会自动产生助记词、种子、根扩展私钥和钱包的详细信息，如图 6-8 所示画面。

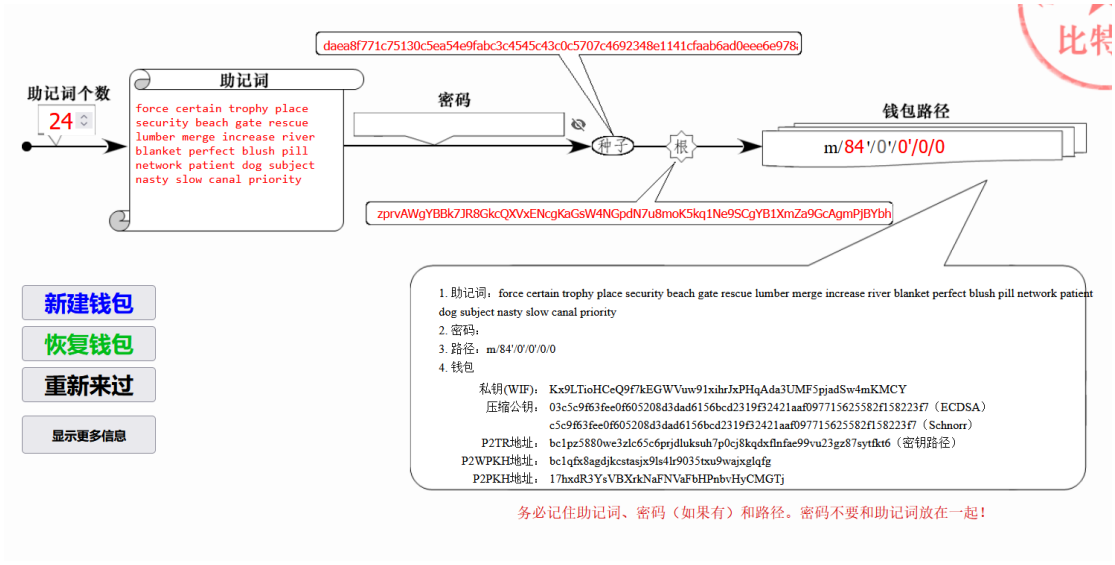


图 6-8 新建钱包

要牢记助记词、助记词保护密码（如果有）和钱包路径，有了这三者，以后可随时恢复钱包。图 6-8 中的钱包路径 m/84'/0'/1'/0/8，其中的红色“1”表示 1 号账户，蓝色“8”表示 8 号钱包。

如果资金较少，也不频繁交易，也可以只记住一个钱包即可，即只记住私钥

L2QKSu8sHME77FezRvYq6zVj16QiWB1JwHUfitgApMGqVWB7zoBW 就可以了，其他的诸如公钥、地址可以随时从私钥推导出来，不过为了收币方便，还是要记住地址，这样可随时告诉别人转账到此地址。P2WPKH 地址交易费用低，支持的钱包最多，我们优先使用它，而 P2TR 地址最近才出现的，目前支持的钱包不多，你如果把 P2TR 地址告诉别人，别人不一定转得了币，P2PKH 地址太老了，交易费用也高，会慢慢弃用的。

点击“显示更多信息”按钮，可以额外看到账户扩展公私钥（如图 6-9 所示），有的钱包 app 支持从账户扩展公私钥恢复钱包，如果你以后使用其他钱包 app 时导入钱包。

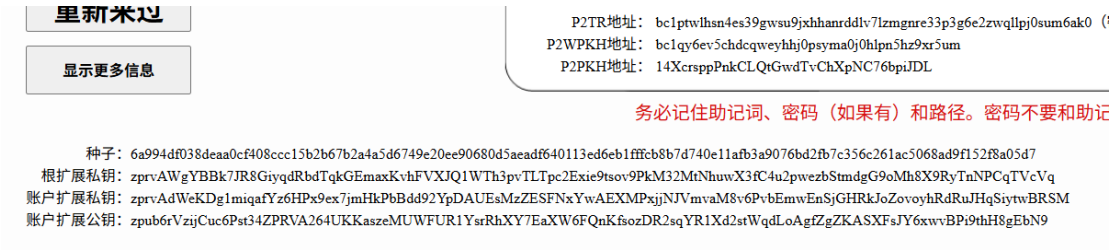


图 6-9 显示更多信息

当然了，你如果记住了种子或者根扩展私钥，也是可以恢复整个 HD 钱包的。

2. 恢复钱包

有三种方法恢复钱包。

方法一、助记词+助记词保护密码（如果有）：先点击“重新来过”清空全部数据，然后输入助记词、密码（没有就不用输）、账户号和钱包号，最后点击“恢复钱包”按钮即可。

方法二、种子：先点击“重新来过”清空全部数据，然后输入种子、账户号和钱包号，最后点击“恢复钱包”按钮即可。

方法三、根扩展私钥：先点击“重新来过”清空全部数据，然后输入根扩展私钥、账户号和钱包号，最后点击“恢复钱包”按钮即可。

6.2.1.2 从私钥导出公钥和钱包地址

点击“钱包管理”下的“从私钥导出公钥和钱包地址”，打开如图 6-10 所示的画面。



图 6-10 从私钥导出公钥和钱包地址

输入私钥即可导出公钥和各种类型的比特币地址。允许输入的私钥格式有三种：WIF 格式（含加密格式）、十六进制格式和二进制格式，十六进制格式不要输入前缀“0x”，二进制格式必须是 256 位。所以允许通过抛硬币的方式产生 256 位二进制数字，然后作为私钥导出公钥和地址。

小心：如果私钥对应的钱包已经有了比特币，请遵循这样的安全规范：（1）用浏览器的隐私窗口打开本页面，（2）让浏览器处于脱机工作模式，（3）分段拷贝私钥，配合手工输入若干字符，参考图 6-11 所示操作。



图 6-11 分段拷贝私钥

分段拷贝可以防止剪贴板间谍软件窃取私钥。图 6-12 是从 L2QKSu8sHME77FezRvYq6zVj16QiWB1JwHUfitgApMGqVWB7zoBW 私钥导出地址的例子。



图 6-12 从私钥导出地址的例子

各种格式的私钥可以互相转换，非压缩公钥包含 x 和 y 坐标，共 64 个字节。

6.2.1.3 自定义钱包地址

点击“钱包管理”下的“自定义钱包地址”，打开如图 6-13 所示的画面。



图 6-13 自定义钱包地址

在“多签地址”中：

P2SH、P2WSH 和 P2TR 类型的地址都支持多签——由多个人签名才能花钱。“*n-of-m* 多签”表示至少 n 人最多 m 人签名才能花钱。



图 6-13 生成多签地址

比如我要生成 P2SH 和 P2WSH 的 2-of-3 多签地址，假设三个公钥分别是 02770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1、02c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d 和 02563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6，结果参考图 6-14 所示。



图 6-14 P2SH 和 P2WSH 的 2-of-3 多签地址

以后可让别人往 bc1qnlz6jdehzz68z9czuz9levmf5czdpudahyp0ta0x6lf0wgxklfrs7apvx2 或 34umyEgUMxve5ru74PLWHJjKdKcJ8Exeow 地址转币，当我们要花费这些地址的 UTXO 时，需要至少两个人签名才行——三个公钥对应的两个或三个私钥签名。

下面我们再来举例说明生成 P2TR 钱包的脚本路径地址，假设内部公钥是 024c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c，默克尔树是：

```
[
   {
      b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853 OP_CHECKSIG
   },
   [
      {
           2596888d00 OP_CHECKLOCKTIMEVERIFY OP_DROP
770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1 OP_CHECKSIG
c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d OP_CHECKSIGADD
563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6 OP_CHECKSIGADD OP_2 OP_GREATERTHANOREQUAL
       },
      {
          6066 OP_CHECKSEQUENCEVERIFY OP_DROP OP_3
7def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d5
a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956
661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41 OP_3 OP_CHECKMULTISIG
      }
   ]
]
```

结果见图 6-15 所示。

P2TR密钥和路径花费地址

输入内部公钥：

024c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c

输入默克尔二叉树：

```
        b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853 OP_CHECKSIG
    },
    [
        {
                2596888d00 OP_CHECKLOCKTIMEVERIFY OP_DROP 770c8febd8c26aa31678f390fbd9df9840b26
40cfbf8d3048af58ead89d150d1 OP_CHECKSIG c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666
a60c6a3e16d OP_CHECKSIGADD 563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6 O
P_CHECKSIGADD OP_2 OP_GREATERTHANOREQUAL
        },
        {
            6066 OP_CHECKSEQUENCEVERIFY OP_DROP OP_3 7def35da02605b57c9a66fd5813df877950f6fd
924291d881d1754c0fc6a93d5 a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956 66
1caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41 OP_3 OP_CHECKMULTISIG
        }
    ]
]
```

确定 重置 帮助

P2TR地址：bc1p99svx4dxnwte3095ccnwntv3700uqyf7mudexgkef8zd9teuwncqe4gfaw

默克尔树根：0b5088cdafe0c6b3686a87377252bf2f2c929294f9c10a3e54aee388b0256da8

Taproot公钥：2960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0

输出脚本：51202960c355a69b9798bcb4c626e9ad91f3dfc0113edf1b9322d949c4d2af3c74f0

存在以下几种花钱方法

1. 密钥路径花钱：使用公钥024c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c对应的私钥签名。

2. 脚本路径花钱方法有3种：

(1). 脚本 = 'b8eac037eea8aec3d26e3e9ac4b698fe0aca713f19ea22aa673b75786ac65853 OP_CHECKSIG'

控制块 = 'c14c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391cadca7de3ce0750a80d80fbb5c5990bb5bf5aedc8a84495e698a1dcb436f3cd41'

(2). 脚本 = '2596888d00 OP_CHECKLOCKTIMEVERIFY OP_DROP 770c8febd8c26aa31678f390fbd9df9840b2640cfbf8d3048af58ead89d150d1 OP_CHECKSIG c5e1f3dc9d9a23650710daa31cfc5faa055c4350fa94690936666a60c6a3e16d OP_CHECKSIGADD 563ccbebd4747938ac6e6e6dfaea04dedac50f8db53f10de366bdc1a88157da6 OP_CHECKSIGADD OP_2 OP_GREATERTHANOREQUAL'

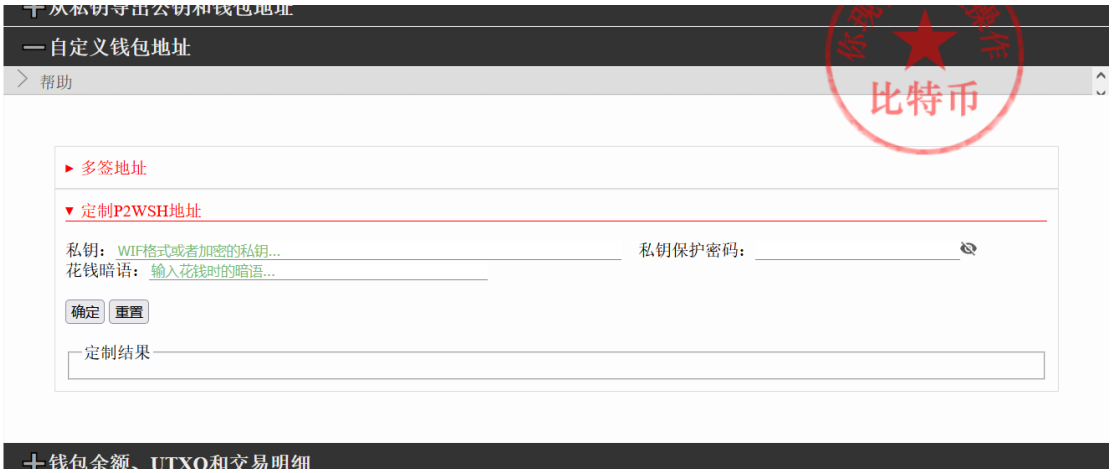
控制块 = 'c14c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391cb3a89c901850dc8010aae0e31acbcf2b193b643ffe4f880fa95e506fa39f27b9494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5fd23534b49'

(3). 脚本 = '6066 OP_CHECKSEQUENCEVERIFY OP_DROP OP_3 7def35da02605b57c9a66fd5813df877950f6fd924291d881d1754c0fc6a93d5 a45bf184a457694e4b0cc4ef79d24447fb1b0de1897b96d652d8b4f730755956 661caba6b0396d1355273f6f9c4345826490e935f36206473c0cf1e694d49c41 OP_3 OP_CHECKMULTISIG'

控制块 = 'c14c72d41da2289ac33d845f8cc2467185467468bc4035f0081b7cc5817203391c3fd18c4440c1e3da8c38833414b47afefac8942c4e7c7d432249707176bcbc2a494d60065b5a484058c16a65de49310c9db39f42cdd61b5062e4d5fd23534b49'

图 6-15 P2TR 脚本路径花费地址

点击“定制 P2WSH 地址”，打开如 6-16 所示的画面。



6-16 定制 P2WSH 地址

在 P2WSH 的赎回脚本中添加验证花钱暗语，在花费 UTXO 时，不但要求签名，还要提供花钱暗语，这样计算别人获取了你的私钥，如果不知道暗语，他还是无法转走你的比特币的。不过一个暗语只能使用一次，因为一旦使用了暗语等于泄露了暗语。比如我有一个添加了暗语的钱包地址，用这个地址不断接受比特币，等到花钱时，我必须一次性花掉此地址的所有 UTXO，比如转到另一个暗语所对应的地址。

6.2.1.4 钱包余额、UTXO 和交易明细

点击“钱包管理”下的“钱包余额、UTXO 和交易明细”，打开如图 6-16 所示的画面。



图 6-16 查询钱包余额

比如我查询地址 bc1qujepl0k5n0ga2e86yskvxa6auehpf6dlf84dx0 的余额，一共由 200 个 UTXO，余额为 132860398 聪（即 1.32860398 个比特币），见图 6-17 所示。



图 6-17 查询余额的结果

点击图 6-17 中的 709ef9d9188da43fa6a018ad998ebbe6794a686a79281f0d9ca4bdd776a817aa 可进一步查询这笔交易的详细内容，见图 6-18。



图 6-18 查询一笔交易

点击“交易原始数据”按钮，可以看到交易的 hex 格式原始数据，见图 6-19 所示。



图 6-19 交易的 hex 格式数据

6.2.1.5 签名/验证

用钱包私钥对一段文本签名，因为签名中包含了公钥，验证时导出签名中的公钥再推导比特币地址，与你发给他的那个地址相比较来证明地址确实是你的，从一些加密币交易平台（如 HashKey Exchange）提取加密币时经常以这样的方式对地址确权。参考图 6-20 所示。

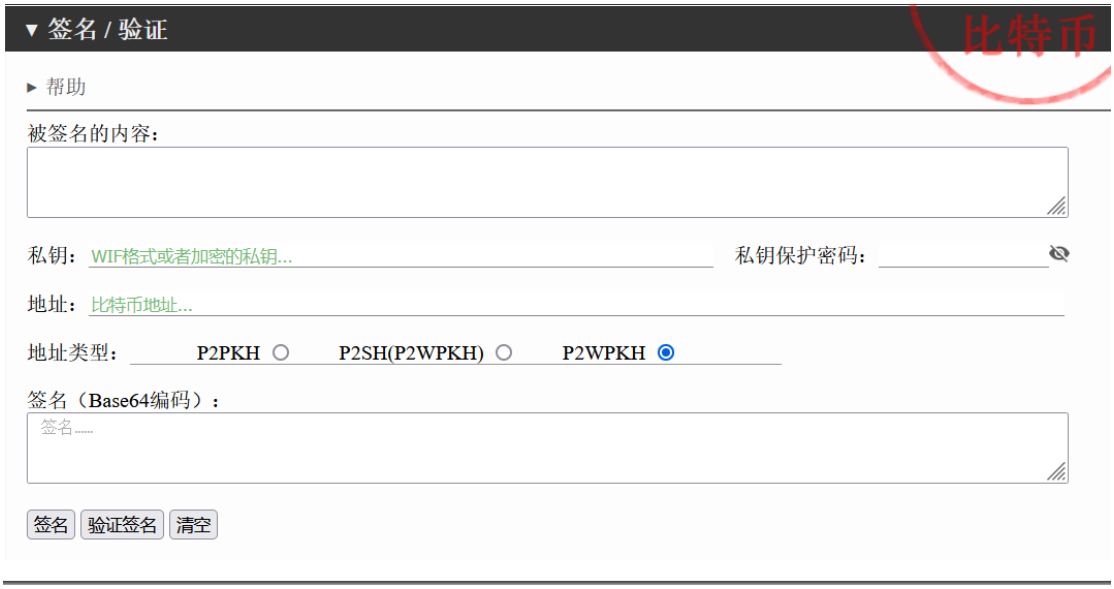


图 6-20 比特币签名与验证

签名时输入“被签名的内容”和私钥（或者已加密的私钥），验证时输入“被签名的内容”、地址和签名。

6.2.2 放币交易

点击图 6-19 中的“放币交易”，打开如图 6-20 所示的页面。

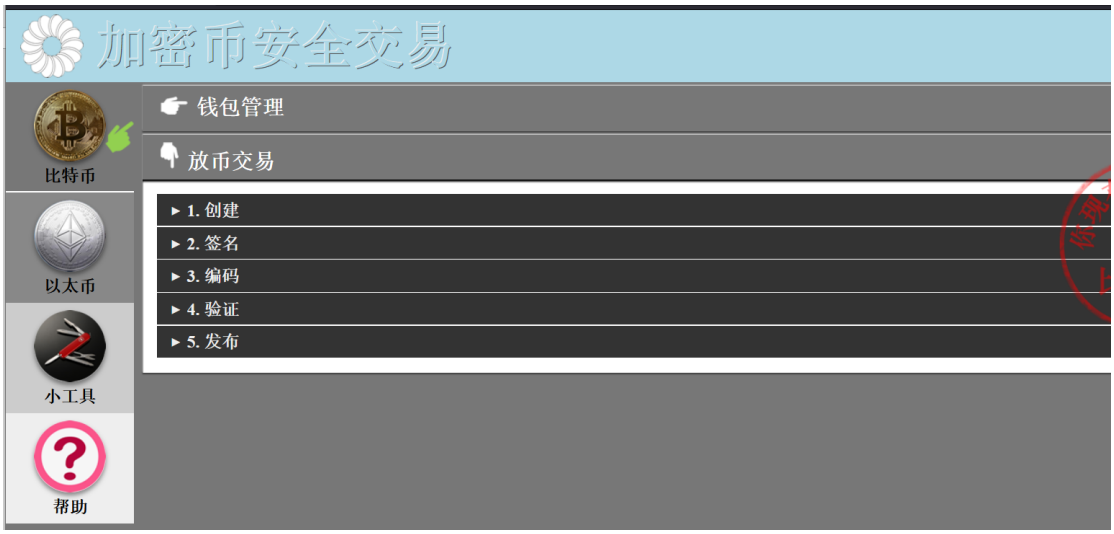


图 6-20 放币交易

放币交易的步骤包括：创建一笔放币交易、对交易签名、交易编码成 hex 字符串、解构交易的 hex 串以便检查交易无误、最后发布交易。

6.2.2.1 创建

点击“放币交易”下的“1. 创建”，打开如图 6-21 所示的画面。



图 6-21 创建交易

除非你知道版本、Marker、锁定时间和 Flag 的确实意义，并真的需要改变它们，否则不要轻易修改这些参数。你可以自己指定交易手续费，一般是 3 至 5 美元，不要太低，低于 1 美元的话，交易成功完成的时间可能比较长（数小时）。交易手续费越高，越快成交，手续费是给比特币矿工的，矿工会优先打包处理交易费高的交易。总金额减去对方入账金额的差就是交易手续费。

首先要添加输入，在图 6-21 的左下方“我的钱包地址”处输入 tb1qqxc0yef599p7r9gmkayf8pln5d7kunwc9qwd38，然后点击“查询 UTXO”按钮，结果见图 6-22。



图 6-22 查询钱包中的 UTXO

即我的钱包只有一个 80000 聪的 UTXO 可花费，点击“花费”按钮把此 UTXO 作为输入添加到交易体中，之后看到的画面见图 6-23 所示。



图 6-23 往交易体中添加一个输入

当然我们还可以把其他钱包中的 UTXO 添加到交易体中。

别紧张，因为还没有添加输出，所以此时的 80000 聪全部当作交易手续费了。接下来我们添加输出，在上图右下方输入地址 tb1p0980z73rdgfachyf9dq4tspzhpl35e9adutfyxeldapzec0afrjs5k8qrs 和入账金额 40000，然后点击“添加输出”按钮，再输入地址 tb1qwv5wau4ye8w7p0z9ylcscy8cet9zs5d80aut9s 和入账金额 39000，点击“添加输出”按钮添加另一个输出，之后的画面见图 6-24 所示。



图 6-24 添加了两个输出

现在的手续费剩下 1000 聪了，按 1 比特币兑换 80000 美元算，也就 0.8 美元了，尽管警告交易手续费过低，我们不管它了。

到此创建了一个交易体，它包含了一个输入、两个输出。已经添加的输入和输出允许删除。

6.2.2.2 签名

点击“放币交易”下的“2. 签名”，打开如图 6-25 所示的画面。



图 6-25 签名界面

签名要用到私钥，所以要特别红色字体的警告语，且分段拷贝私钥。签名的私钥就是输入所对应的那个 UTXO 所在钱包的私钥。签名类型默认为 SIGHASH_ALL，非必要不要轻易改变。如果使用密文私钥，请输入加密时用到的密码以及 N、r、p 参数，N、r、p 参数对应加密的“CPU/内存成本”、“块大小”、“并行化”参数。

首先点击“导入输入”按钮从交易体中导入全部输入，以便后续对它们签名，然后勾选一个输入，再分段拷贝私钥，最后点击“签名”按钮完成签名，参考图 6-26 操作。



图 6-26 签名步骤

当然你也可以采用密文私钥签名，只不过需要输入解密密码。

6.2.2.3 编码

点击“放币交易”下的“3. 编码”，打开如图 6-27 所示的画面。



图 6-27 编码界面

这一步比较简单，直接点击“输出 Hex 格式”按钮把交易体输出成 Hex 编码的字符串，见图 6-28 所示。

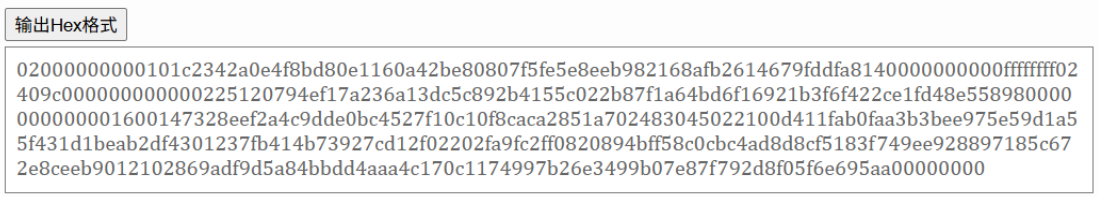


图 6-28 交易的 Hex 码

6.2.2.4 验证

点击“放币交易”下的“4. 验证”，打开如图 6-29 所示的画面。



图 6-29 验证界面之一

图 6-29 中的“Hex 格式交易串”自动导入上一步产生的 Hex 字符串，当然你也可以手工输入。点击“解构”按钮后的画面如图 6-29 所示。

交易解构如下：

交易Id：d878ef0efe2bc1f4f7a4cd4fb9ac43bf2b3fbc7a272f3c031a4fd6c018aaac0a 交易费用：1000聪（0.8美元）

版本				2 02000000
Marker				00
Flag				01
输入个数				1
第1个输入	交易id			14a8dffd794661b2af682198eb8e5efef50708e82ba460110ed88b4f0e2a34c2 c2342a0e4f8bd80e1160a42be80807f5fe5e8eeb982168afb2614679fddfa814（小端序）
	输出序号			0 00000000
	输入脚本长度			0
	顺序号			4294967295 ffffffff（16进制）
输出个数				2
第1个输出	地址			tb1p0980z73rdgfachyf9dq4tspzhpl35e9adutfyxeldapzec0afrjs5k8qrs
	金额			40000聪 00a0000000000000
	输出脚本长度			34 22（16进制）
	输出脚本			5120794ef17a236a13dc5c892b4155c022b87f1a64bd6f16921b3f6f422ce1fd48e5 OP_1 794ef17a236a13dc5c892b4155c022b87f1a64bd6f16921b3f6f422ce1fd48e5（汇编格式）
第2个输出	地址			tb1wv5wau4ye8w7p0z9ylcscy8cet9zs5d8tapcq6
	金额			39000聪 00a0000000000000
	输出脚本长度			22 16（16进制）
	输出脚本			00147328eef2a4c9dde0bc4527f10c10f8caca2851a7 OP_0 7328eef2a4c9dde0bc4527f10c10f8caca2851a7（汇编格式）
见证数据	见证1	条款数量		2
		条款1	长度	72
			内容	3045022100d411fab0faa3b3bee975e59d1a55f431d1beab2df4301237fb414b73927cd12f 02202fa9fc2ff0820894bff58c0cbc4ad8d8cf5183f749ee928897185c672e8ceeb901
		条款2	长度	33
			内容	02869adf9d5a84bbdd4aaa4c170c1174997b26e3499b07e87f792d8f05f6e695aa
锁住时间				0 00000000

关闭

图 6-29 解构交易 Hex 码

仔细检查交易的各个字段，重点检查对方地址、入账金额、输出脚本以及交易手续费。

当然也可以把交易的 Hex 码字符串拷贝到 https://live.blockcypher.com/btc/decodetx/（比特币主网）、https://live.blockcypher.com/btc-testnet/decodetx/（比特币测试网）网站去解构。

确认无误后就可以发布了。

6.2.2.5 发布

点击“放币交易”下的“5. 发布”，打开如图 6-30 所示的画面。

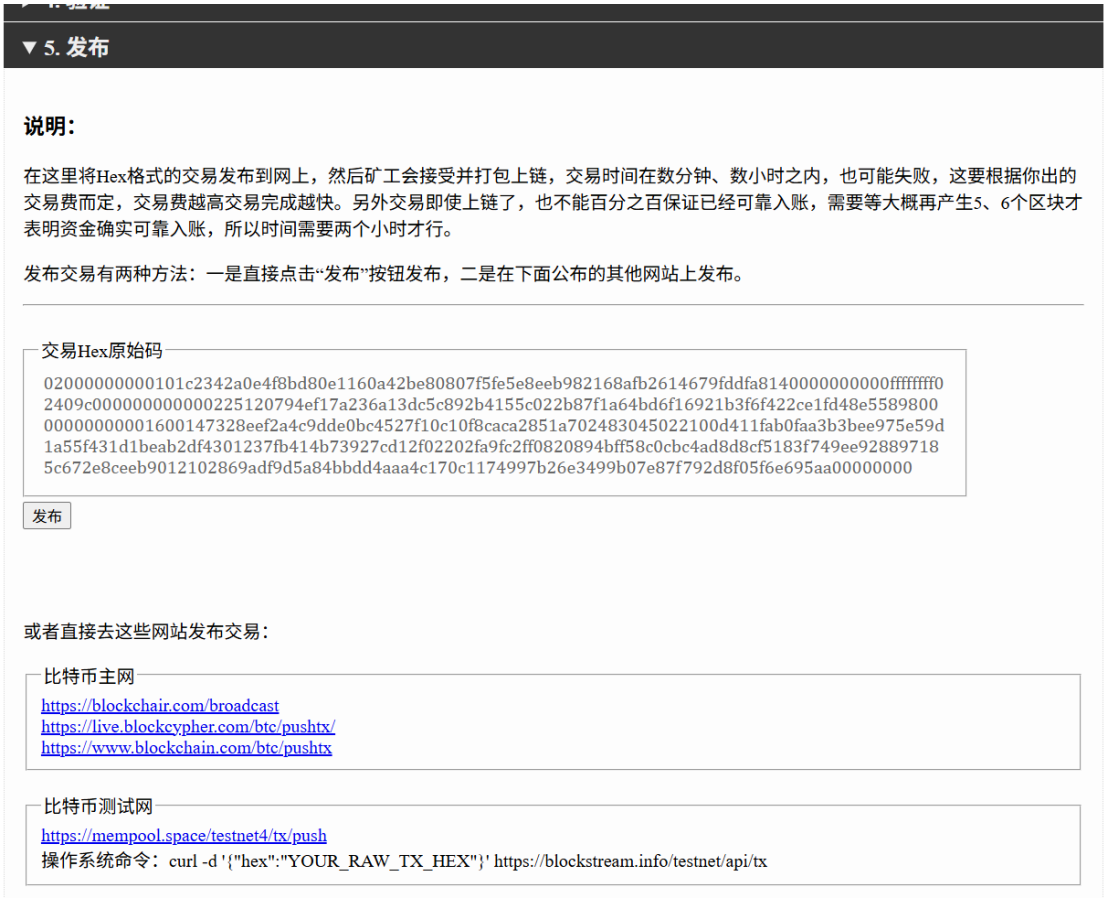


图 6-30 发布交易

点击“发布”按钮即可，不过你也可以拷贝到第三方网站去发布，这些网站有:

比特币主网: https://blockchair.com/broadcast、https://live.blockcypher.com/btc/pushtx/、https://www.blockchain.com/btc/pushtx。

比特币测试网：https://mempool.space/testnet4/tx/push

6.2.2.6 检查交易结果

过几分钟，再去 https://www.blockchain.com 或 https://blockchair.com/zh 查询交易是否完成。图 6-31 查询对方到账情况。

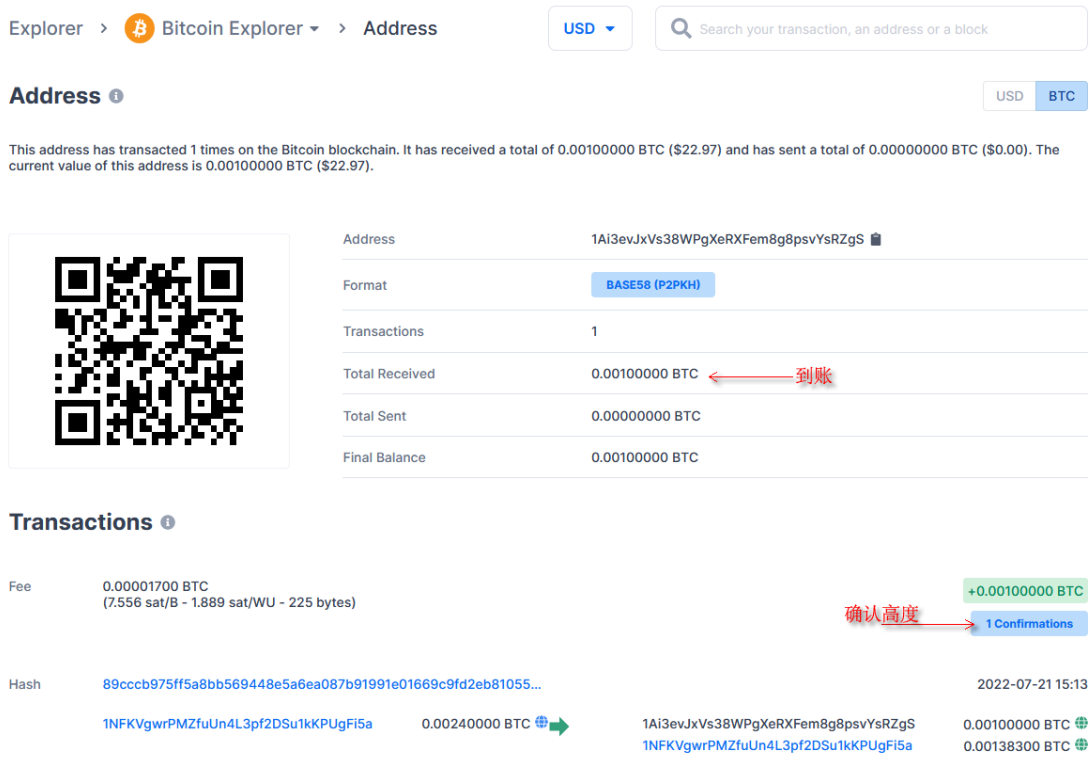


图 6-31 对方到账成功

再来查询找零是否完成，参见图 6-32。

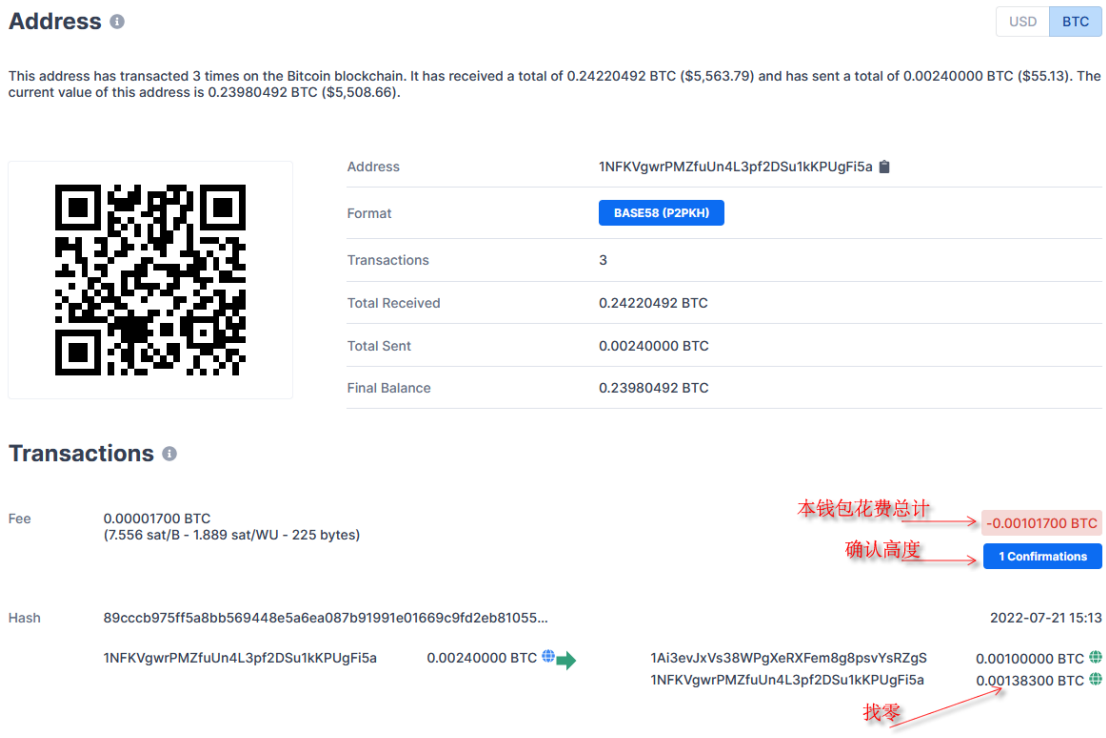


图 6-32 找零完成

“确认高度”是指本交易所在的区块之后新接上的区块数量（示意图见 6-33），比特币的区块长度大概每十分钟增加 1，如果你对区块不了解，请往前翻阅“§1.6 区块链”章节。“确认高度”的值越大，说明本交易越可靠（越难以作废），如

果是大额交易（比如转移成百上千个比特币），“确认高度”一般达到 5 以上就绝对没问题了（此后不会再显示确认高度）。

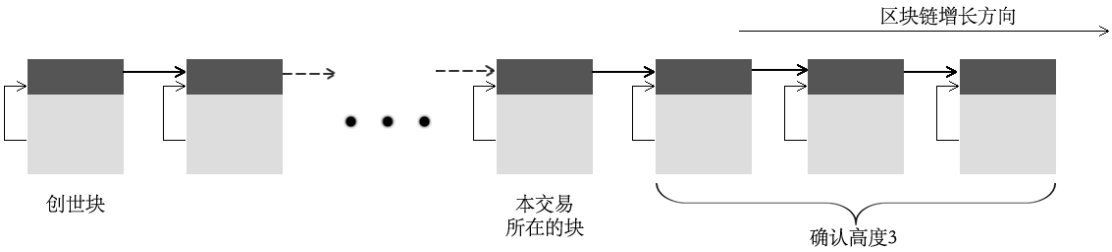


图 6-33 “确认高度”解释

至于交易费用，4 个小时内成交的费用大概 0.8 美元，也可以浏览估算交易费网站：https://www.buybitcoinworldwide.com/fee-calculator/

6.3 以太币

6.3.1 钱包管理

6.3.1.1 新建钱包 / 恢复钱包

点击左侧的“以太币”，然后点击“钱包管理”下的“新建钱包&恢复钱包”，出现如图 6-34 所示的界面。



图 6-34 以太币钱包管理

在这里管理以太币钱包。如果密码不同，相同的助记词会产生不同的以太币钱包。可以从助记词恢复 HD 钱包，也可以从已有的私钥（不管是被密码保护的私钥还是未保护的私钥）导出公钥和地址。

1. 如果要新建钱包，首先确定助记词个数（最少 12 个，最多 24 个，助记词越多越不容易被猜出）和助记词使用的语言（比如英语助记词、简体中文助记词等）。“助记词”+“助记词保护密码”可以导出大量的私钥（一个私钥对应一个钱包）。密码不同，即使助记词相同，那么导出的私钥也不同，所以建议使用密码来保护助记词。“钱包路径”为形如“m/44'/60'/i'/0/j…”的字符串，表示第 i+1 个账户里的第 j+1 个钱包， i 和 j 都是大于或等于 0 的整数，你只能改变 i 和 j，或者增加长度。例如默认路径“m/44'/60'/0'/0/0”表示第 1 个账户里的第 1 个钱包，“m/44'/60'/0'/0/5”表示第 1 个账户里的第 6 个钱包，“m/44'/60'/3'/0/2”表示第 4 个账户里的第 3 个钱包，“m/44'/60'/2'/0/1/3”表示第 3 个账户里的第 2 个节点下的第 4 个子钱包。不过建议钱包路径不要太长太复杂，否则容易搞错或忘记。

如果输入了私钥保护密码，那么最终的私钥被加密保护，以后签名时使用被密码保护的私钥时需要输入这个私钥保护密码，不要忘记了。

2. 如果要恢复钱包，有两种方法，一种方法是从“助记词语言”、“助记词”、“助记词保护密码”和“钱包路径”恢复，只要输入以前产生钱包时的语言、助记词、助记词保护密码和钱包路径即可恢复全部的私钥；第二种方法是从私钥恢复钱包地址，即输入一个私钥即可，如果私钥被加密保护，还需输入私钥保护密码。

切记：创建的钱包如果你打算正式使用，一定要牢记助记词、助记词保护密码（如果有的话）、钱包路径、私钥、地址，私钥建议进一步加密保护（在“钱包管理”»“私钥保护”页操作）。地址公布给其他人，以便他人往此地址放币，其他任何信息绝对不要外泄。

6.3.1.2 签名 / 验证

在以太币交易时，经常会遇到这种情况：对方需要向你确认地址的真实性，他会发一段文字给你，让你签名，然后把签名反馈给他，他收到签名后从签名算出公钥再推导出地址，最后比较两个地址是否相等。签名与验证参见图 6-35 所示。



图 6-35 签名与验证

对一段文本签名，这里直接调用 ethers.js 库的 signMessageSync()方法。对于 HashKey Exchange 加密币交易平台，当要提取以太币时，需要验证一段文本签名，以确保转入的钱包是自己的。

签名时需要输入被签名的文本、私钥（或加密私钥及密码），验证时需要输入被签名的文本、公钥和签名。

6.3.2 放币交易

依次点击“以太币”、“放币交易”、“1. 创建”，出现如图 6-35 所示的画面。



图 6-35 以太币放币交易

“转入钱包地址”就是以太币转入的钱包地址，形如“0xB8309ea73A7C61599bF33C73085D24511E042b06”。

“转账金额”就是转账多少个以太币，如 0.005 个以太币，也可以为 0 个以太币（这时只想把信息保存在备注中）。

“交易费用”就是本次转账你愿意承担多少美元的手续费，最终会从转出钱包中扣除等值的以太币。手续费与转账金额没关系，但与“备注”有关联，如果没有备注，一笔交易手续费一般是 5 美元上下，手续费越高越快完成交易，当然如果低于 1 美元，交易时间比较长甚至失败。

“钱包私钥”就是资金转出钱包的私钥，用来签名。明文私钥形如 0x6b714fb46b3558192193712dd9dacabd0e29f75b02c81e21d4fc675f9d50719a，加了密的私钥形如 6PYK8XSffZBgqqCMX2METG2VXUdAX5EfMDnCAV6D66B2dWQzdF4Za9Yspr。

“私钥密码”用来解密私钥的，如果你的私钥被加密，必须输入密码以便解密私钥，否则“私钥密码”不要输。

“流水号”就是转出钱包目前是第几次转出以太币，新钱包第 1 次转出以太币时，流水号为 0，第 2 次为 1，以此类推。如果网络通畅，流水号会自动获得，否则必须手工输入，而且不能输错。可以去其他第三方网站查询钱包的流水号。

“交易类型”默认是 2（EIP-1559），这是目前使用最广泛的，交易费用低，除非你彻底了解“交易类型”的真实含义且必须修改，否则不要轻易改动。

“油耗上限”默认是 21000，如果你增加了备注内容，必须上调油耗上限，否则矿工很可能会忽略此交易。

“给矿工的小费”是 2 号交易类型特有的，小费包含在“交易费用”之中，即你愿意从交易费用之中分出多少作为给矿工的小费。交易费用减去小费的差额会转入坟场地址（0x0000000000000000000000000000000000000000），这个地址的以太币是永远无法转出的，这样相当于销毁了以太币，从而减少流通量。 一般小费占到交易费用的九成以上。小费是给矿工的，所以小费越多越能激励矿工尽快处理你的交易。

“备注”最多可输入 7000 个汉字，如果你想在区块链上永久保存信息，就输入到备注里，字数越多，交易费用就越高，

几十上百元的交易费都有可能。

最后点击图 6-35 中的“转账”按钮正式发布交易，如果网络不通，发布就会失败，这时可把 hex 格式的交易串拷贝到其他地方去发布。

6.4 其他功能

其他功能放在左侧的“工具”菜单里。

6.4.1 私钥保护

私钥代表一个钱包，用私钥签名就能花钱，所以私钥特别重要，除了自己外不要告诉任何人，为了杜绝因泄露私钥而导致资金损失，建议对私钥进行加密保护。

点击左侧“工具”下的“私钥保护”，打开如图 6-36 所示的画面。

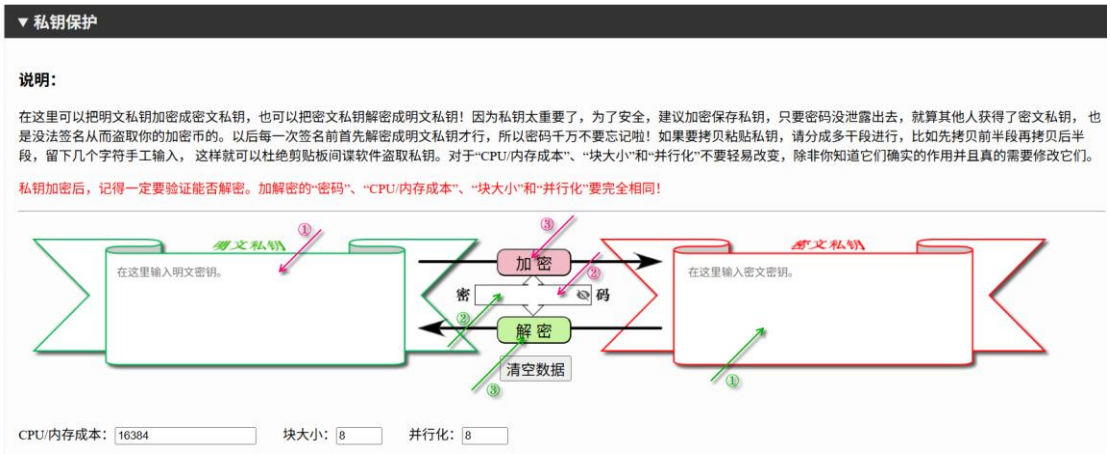


图 6-36 私钥保护页面

图 6-36 中的红色箭头及标注就是加密私钥的步骤，即输入私钥，输入密码，再点击“加密”按钮。至于参数“CPU/内存成本”、“块大小”、“并行化”主要涉及加解密的效率，这些参数的值不同会影响结果，如果你真的需要修改它们，那么同时请牢记修改后的值。“CPU/内存成本”必须是 2 的 n 次方，n=0、1、2、3、……。对于这些参数的进一步理解，请参考“§2.4.4 bip38”小节中的 n、r、p 参数的介绍。

密码不要太短，建议 10 个字符以上，且包含数字、大小写字母和各种标点符号。

比如对私钥 L3bprT1HbdD9yMhpsZUntXSDLSeBrg87qCP3BemSofcKREEZ1dfw 用密码 Pass123!Y4yhBm 保护，加密后的私钥为 6PYLJJ6wJ6c3jyVPyHZMWR5octU7XMHVPZqZjECMoy829vcQLtdWx2Kvcs，见图 6-37 所示。

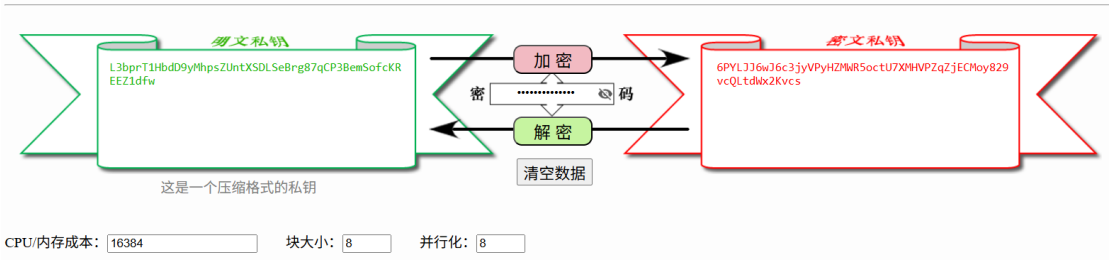


图 6-37 例子：加密保护私钥

以后用加密后的私钥签名时需要输入密码 Pass123!Y4yhBm 才行，密码记在脑袋里，绝对不要与加密后的私钥放在一起。

解密操作：先点击“清空数据”按钮，然后在右侧的“密文私钥”中输入加了密的私钥，在输入解密密码，最后点击“解密”按钮，参见图 6-38 所示。

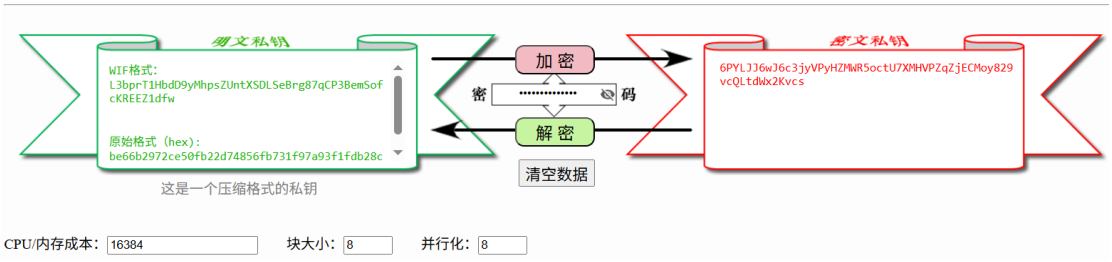


图 6-38 解密私钥

注意：解密时必须输入在加密时使用的密码、CPU/内存成本、块大小和并行化才行。

每一次对私钥加密后最好验证一下能否解密，而且没有加密的私钥应该冷保存，比如写在纸上放在家里，或者保存在 U 盘中，U 盘放在家里，平时使用加了密的私钥签名交易。

6.4.2 定制助记词

合法的助记词最后一个单词是其他单词的校验，所以自己不能随便选择所有的单词，而且单词数量只限于 12、15、18、21 或 24 个。

但你可以定制助记词（比如选择自己熟悉的便于记忆的单词），除了最后一个单词外的其他单词都可以自己指定，不过指定的单词越多安全性就越差，毕竟人为选择的单词随机性不佳，相对系统随机选择的助记词来说，更容易被他人猜出。所以遵循的原则就是自己指定的单词越少越好。

点击左侧“工具”下的“定制助记词”，打开如图 6-39 所示的画面。

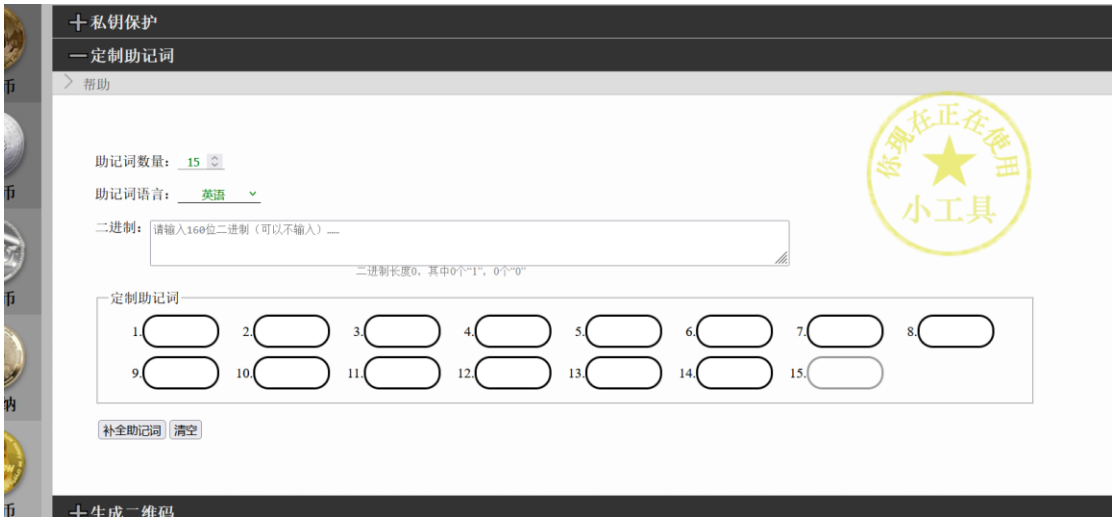


图 6-39 定制助记词

首先确定一个助记词包含的单词个数，数量越多越安全，其次选择好助记词所在的语言，默认是英语，你也可以选择简体中文、日语等其他语言，不过英语通用性强，不管在世界什么地方用什么钱包，英语助记词都是支持的，而其他语言的通用性就差一些，比如你下次使用某款钱包 app，可能就不支持简体中文的助记词。

在这里你可以用两种方法定制助记词：（1）直接输入二进制，比如通过抛硬币的方式产生随机二进制；（2）直接指定助记词单词，比如选择自己熟悉的便于记忆的单词，除了最后一个单词外的其他单词都可以自己指定，不过指定的单词越多安全性可能就越差，毕竟人为选择的单词随机性不佳，相对系统随机选择的助记词来说，更容易被他人猜出。所以遵循的原则就是自己指定的单词越少越好。

操作：首先确定一个助记词包含的单词个数，数量越多越安全，其次选择好助记词所在的语言，默认是英语，你也可以选择简体中文、日语等其他语言，不过英语通用性强，不管在世界什么地方用什么钱包，英语助记词都是支持的，而其他语言的通用性就差一些，比如你下次使用某款钱包 app，可能就不支持简体中文的助记词。 如果输入了的二进制就不要指定单词了（表示使用方法（1）），在没有输入二进制的前提下，允许指定某些单词，除了最后一个单词不能指定外，其他的都可以，比如指定前五个单词为 love you very much really，或者在简体中文中指定 1、3、5、7 四个汉字为“中”、“王”、“大”、“民”。最后点击“补全助记词”按钮即可产生完整的助记词。

然后指定某些单词，除了最后一个单词不能指定外，其他的都可以，比如指定前五个单词为 love you very much really，或者在简体中文中指定 1、3、5、6、7、8、11、12 八个汉字为“中”、“国”、“有”、“十”、“四”、“亿”、“人”、“口”，最后点击“补全助记词”按钮即可产生完整的助记词，参见图 6-40。



图 6-40 定制中文助记词

注意：你最后决定使用的助记词要去比特币或者以太币的钱包管理中验证，看看能否正确产生钱包！

6.4.3 生成二维码

把公钥、地址、ENS 域名等信息做成二维码，保存在手机上，可以方便发给他人，或者便于用移动设备扫描。

注意：可以把私钥或助记词做成二维码，但是千万不要保存在移动设备上（如手机、平板等），或用移动设备扫描，因为移动设备是最不安全的！当然你可以单独购买一部手机，不安装任何应用软件，也不联网，不插入手机卡，专门用于保存私钥和助记词。参见图 6-41 所示。



图 6-41 生成二维码

6.4.4 图片隐写

把一些敏感信息隐藏在一张图片中，然后再把图片传递给别人以达到交换敏感信息的目的。允许加密后再隐藏，如果别人获取到了图片但是没有密码，还是不能获取信息。不过现在存在一些软件工具能发现隐藏在图片中的信息，所以建议还是加密后再隐写。注意：图片最好选择层次感强烈的风景画，有蓝天、白云、草原、大海之类的。平均来说，8 个像素能保存一个汉字，三个像素能保存一个字符。 对于一些机密信息，最佳的做法是：机密信息保存在一个文件中，然后采用“对称加 / 解密”（见本页下面的菜单）加密文件，再把加密后的文件隐写到图片，可能的话隐写时再增加一个密

码。如图 6-42 所示。



图 6-42 图片隐写

6.4.5 数字签名与验证

数字签名用于确权文件的主人，类似于现实生活当中的手写签名。用私钥签名，用公钥验证签名。签名后的文件不可更改，否则签名验证就会失败。对方必须获得文件、公钥和签名后才能验证签名。加密币的密钥一般不用来对文件签名，因为需要公布私钥从而泄露了隐私，在日常生活当中，通常采用 pgp 兼容的密钥来签名和验证签名。所以这里提供了两种签名方法：第一中方法是采用加密币密钥。

第二种方法采用 pgp 兼容密钥。当采用第一种方法时要注意两点：私钥必须为十六进制表示的字符串，即 hex 编码格式，如 c6f53ee725f194b9bc8d06b61b91fa1d2fb6884d2a36fe037579bbb950f67ee2，公钥也是 hex 编码的六进制字符串，且必须去掉 0x 前缀，例如 02f7eb7c084a14c30a3a5e6e0dcc10a9881ebf90ba45039f366d1919559c92f3c4。签名算法目前流行的有 Schnorr、ECDSA、DSA 和 RSA，Schnorr 比较先进，但目前使用暂时没有 ECDSA 那么普遍。

签名时必须选择一个文件，并且输入私钥，然后把文件、公钥和签名结果发给对方，对方可以验证签名，参考图 6-43 所示的界面是第一种方法。



图 6-43 签名与验证签名

对文件签名，这里采用 Schnorr 算法。图 6-44 为第二种方法。



图 6-44 使用 PGP 兼容密钥

6.4.6 非对称加密与解密

用公钥对文件加密，用私钥解密文件。如果加/解密是为了安全地与其他人传递信息，不太合适使用加密币的公钥和私钥，因为这会泄露公钥，所以还是建议采用 PGP 密钥。图 6-45 是采用加密币的情景。



图 6-45 非对称加密与解密（加密币密钥）

而图 6-46 为 PGP 密钥。



图 6-46 非对称加密与解密（PGP 密钥）

6.4.7 对称加密与解密

这里实现了基于 OpenPGP 标准的对称加密和解密，兼容 GnuPG、gpg4win 等开源工具。见图 6-45。



图 6-45 对称加/解密

6.4.8 UTF8 与 HEX 互换

utf8 文本（例如简体中文）与 hex 编码相互转换，如果要想把信息保存在区块链上，必须转换成 hex 编码才行。见图 6-46。

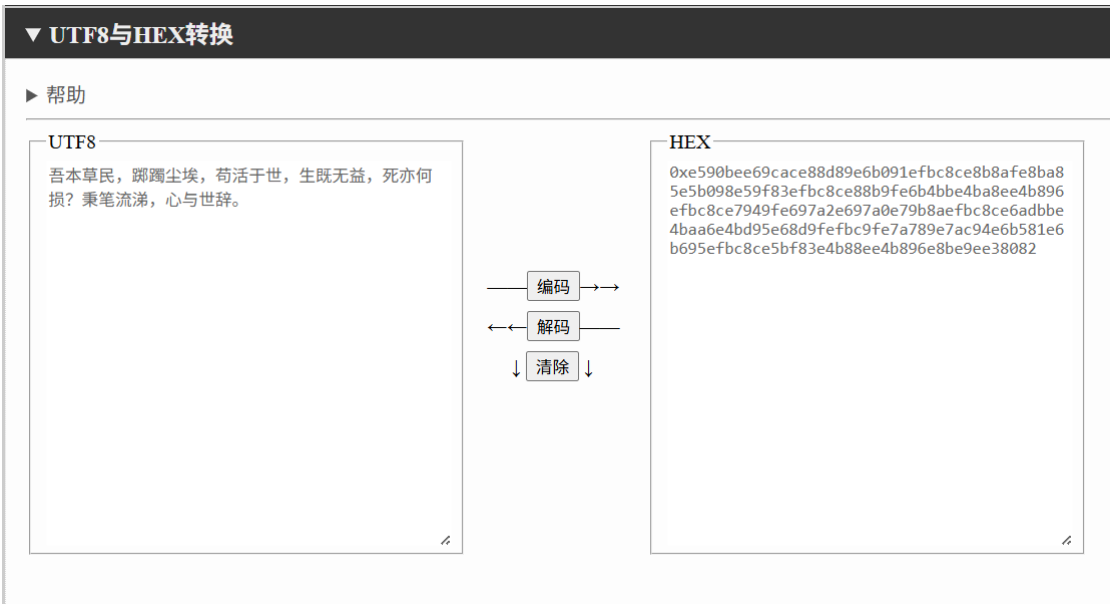


图 6-46 utf8 与 hex 互换

6.5 购买加密币

可以线下和线上购买比特币，线下就是直接去香港加密比窗口购买，或者从熟人那里购买，但是暴露了隐私，不适合大额度购买比特币，毕竟在很多国家持有加密币不合法。

所以很多人还是喜欢线上购买，虽然支付法币也暴露了隐私。

6.5.1 主流 P2P 交易平台

Binance P2P

网址: https://p2p.binance.com

优势：流动性高，支持多种法币（USD、EUR、RMB 等）和加密货币，内置争议解决机制。

注意：需完成 KYC（Know Your Customer，实名）认证，部分地区可能受限。存在中间币 USDT，与美元汇率 1。允许把平台上的加密比转出到自己的钱包。

Bybit P2P

网址: https://www.bybit.com/fiat/trade/p2p

类似 Binance，支持多币种交易，适合亚洲用户。

OKX P2P

网址: https://www.okx.com/p2p-markets

老牌交易所的 P2P 服务，支持多种支付方式。

6.5.2 去中心化 P2P 平台

Bisq

网址: https://bisq.network

完全去中心化，无需 KYC，支持比特币等主流币种。

注意：交易速度较慢，适合技术熟练用户。

HodlHodl

网址: https://hodlhodl.com

不托管用户资金，支持多币种，需使用多重签名托管。

LocalCoinSwap

网址: https://localcoinswap.com

支持多种加密货币和支付方式，部分用户需 KYC。si*/gmail*|2*Y#,w*g@hotmail.com

注册后购买参考图 6-47。

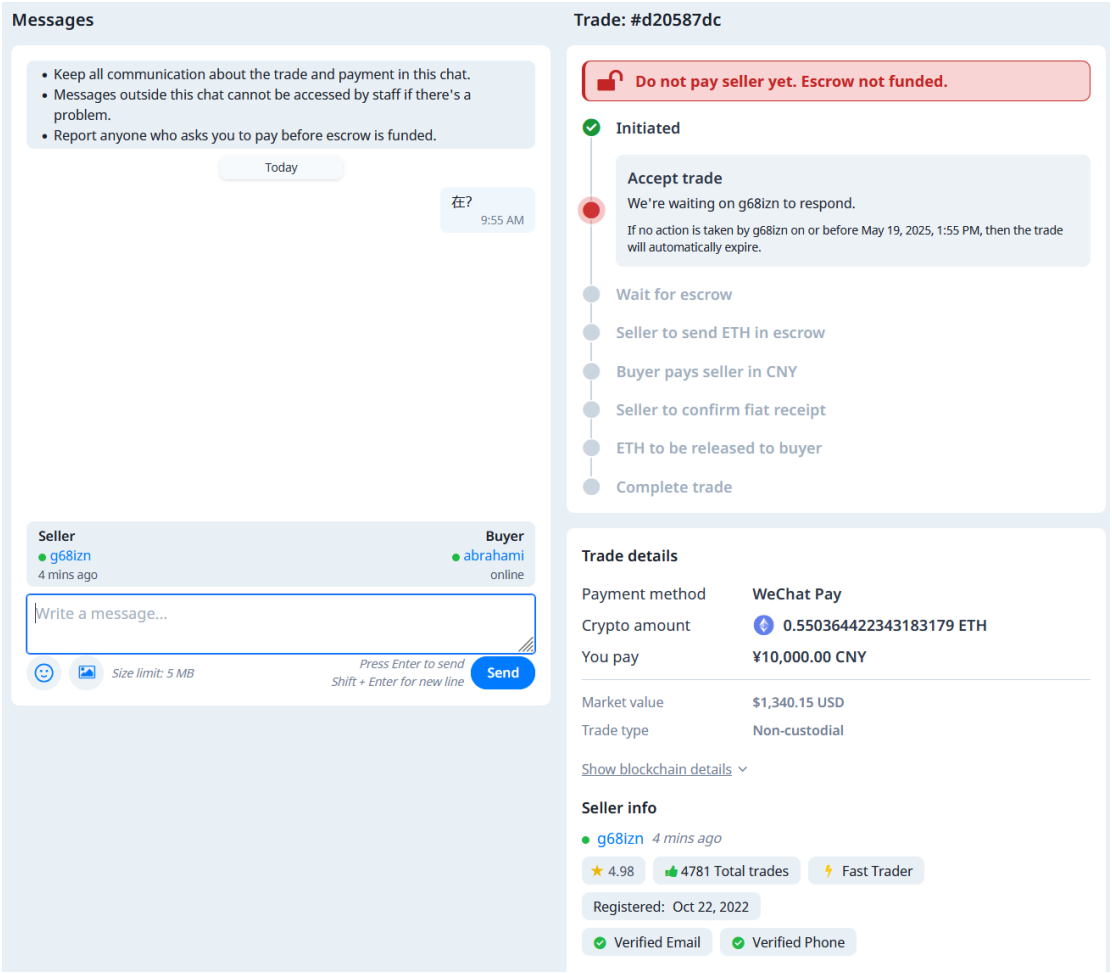


图 6-47 localcoinswap 购买步骤

去 https://www.blockchain.com/或 https://blockchair.com/zh 网站上查查是否真的到账，参见图 6-48 所示。

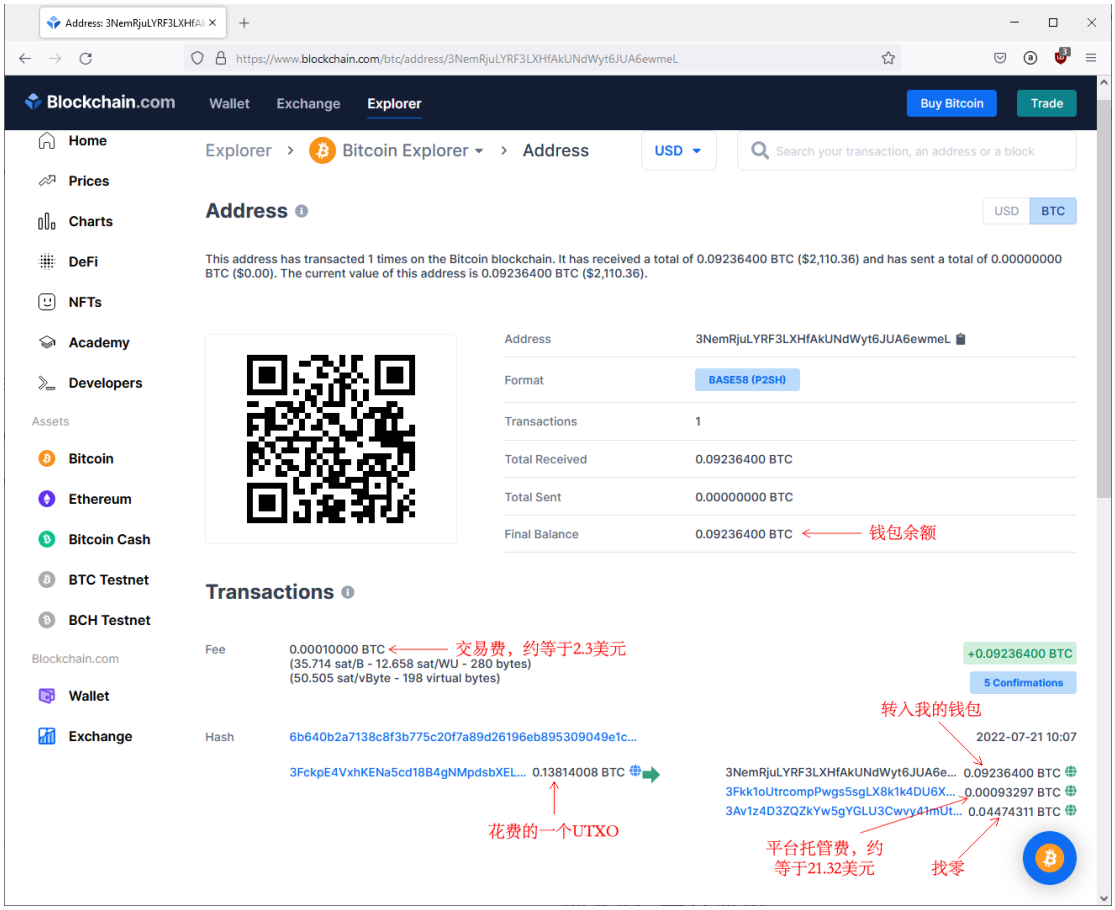


图 6-48 查看钱包余额

从图可以看出，给矿工的费用是 0.0001 比特，给 localcryptos.com 平台的佣金是 0.00093297 比特，卖方花费的一个 utxo，金额为 0.13814008 比特，转给我 0.092364 比特，所以找零 0.04474311 比特，有等式：

0.0001+0.00093297+0.04474311+0.092364=0.13814008

因此本次总的交易费用为 0.00103297 比特，约等于 168 元人民币，费用还是比较高的。费用全部是卖方出，不过卖方的出价一般高于市价几个百分点，国外的卖家会高十几个百分点。

从 localcoinswap.com 找回助记词的方法为：点击右上角的用户名下面的 Settings，然后再点击 security，再点击 Export Mnemonic，可以看到一个由 24 个单词组成的助记词和账户私钥。注意这里的账户私钥不是钱包私钥，HD 钱包私钥可从助记词恢复（可在 https://iancoleman.io/bip39/ 或者 https://www.walletoffline.com/btceth 完成），比特币钱包路径为 m/49'/0'/0'/0/0，对应的地址类型为 P2SH-P2WPKH，以太币钱包路径为 m/44'/60'/0'/0/0，莱特币钱包路径 m/84'/2'/0'/0/0。在 localcoinswap.comd 上登陆之后点击 Wallet 可查看各种加密币的钱包地址。从私钥或者公钥产生 P2SH-P2WPKH 地址的方法请参考“§4.3.5 P2SH-P2WPKH 地址进阶”小节。

6.6 安全增强（可选）

把网页文件封装在容器中是一种安全增强措施。在安装了 docker 环境的电脑上新建目录 walletoffline，然后把开放离线钱包的页面文件主目录拷贝到 walletoffline 中，再在此目录中新建文件 Dockerfile，此文件内容如下：

```
FROM nginx:alpine

RUN rm -rf /usr/share/nginx/html/*
COPY ./btceth /usr/share/nginx/html
```

最终的目录解构参见图 6-49 所示。

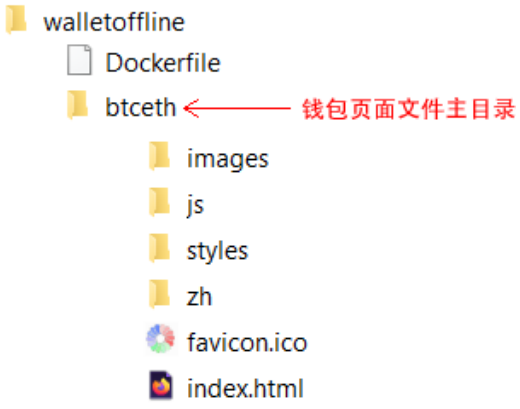


图 6-49 构建容器时的目录解构

然后进入目录 walletoffline 中，执行下面的命令。

```
docker build -t walletoffline ./

docker images
REPOSITORY      TAG      IMAGE ID      CREATED         SIZE
walletoffline   latest   1e421a193ec3  28 minutes ago  60.5MB

docker run -d -p 80:80 walletoffline
```

这样就可以用浏览器打开网址 http://localhost 了。或者采用下面的命令向其他人发布钱包容器：

```
docker image save -o ./walletoffline.tar walletoffline
```

然后把 walletoffline.tar(或先压缩它)放到网上让别人下载使用。其他人下载这个容器文件后，需要安装容器环境(windows 上可安装 docker desktop)，最后执行下面的命令导入容器并启动：

```
docker load -i walletoffline.tar
docker run -d -p 80:80 walletoffline
```

7 中级操作环境

中级篇就是在工作电脑上安装 virtualbox 虚拟机软件（或者 Vmware），然后创建一台虚拟机，虚拟机中安装 Debian linux 操作系统~~，且启用 Tor 暗网~~。在虚拟机里配置脱机工作的 firefox 浏览器，以后加密币的交易创建和签名就在 firefox 浏览器中进行。再安装 Chrome 浏览器，走 VPN 出去，专门用于上网。~~再安装 tor browser 浏览器专门用于发布交易以及与外界联系，~~在虚拟机的浏览器中操作就跟初级操作环境一样。

为了增加安全性，建议采用从中国大陆之外购置的电脑设备（如苹果电脑、惠普电脑、戴尔电脑等），安装 MAC OS 或 Linux 操作系统，如果实在要用 windows，那么尽量采用从微软官网下载原版 windows 操作系统 (https://www.microsoft.com/zh-cn/software-download/windows10/)，电脑上尽量不要安装中国大陆开发的软件（比如 qq、微信、搜狗拼音输入法等）。

当然也可以在虚拟机里安装 windows 操作系统，然后就参照“初级操作环境”去交易加密币。之所以建议采用 Linux 操作系统，那是因为 Linux 系统更安全，无病毒。

7.1 准备虚拟机

我们使用 VMware WorkStation 虚拟机软件（也可以使用 virtualBox），在虚拟机里安装 Debian 12 操作系统，单独创建一个 500MB 左右的加密分区来保存重要的私密资料（如私钥等），挂载时需要输入密码才行。走 Tor 网（tor-browser 浏览器），让 Tor 再走梯子出去。在 firefox 浏览器的离线工作状态下处理敏感数据（如产生钱包、签名等）。虚拟机相当于一台独立的电脑，专门在虚拟机中交易加密币，能起到很好的隔离效果。

VMware WorkStation 17.5 版本个人使用开始免费，但是是闭源的，virtualBox 尽管是开源软件，但是属于 Oracle 公司的，所以并不怎么“自由”。VMware WorkStation 更稳定，使用简单，且被得到广泛使用。

先去 https://support.broadcom.com/注册用户（wlm@ver*.org|Hua*3$），然后可从官网搜索并下载 vmware workstation 下载最新版，最新版本是 VMware-Workstation-Full-26H1u1-25688693.exe，下载后直接安装，然后重启电脑。启动 Vmware 后按 Ctrl+n 创建一台虚拟机，参见图 7-2 操作。

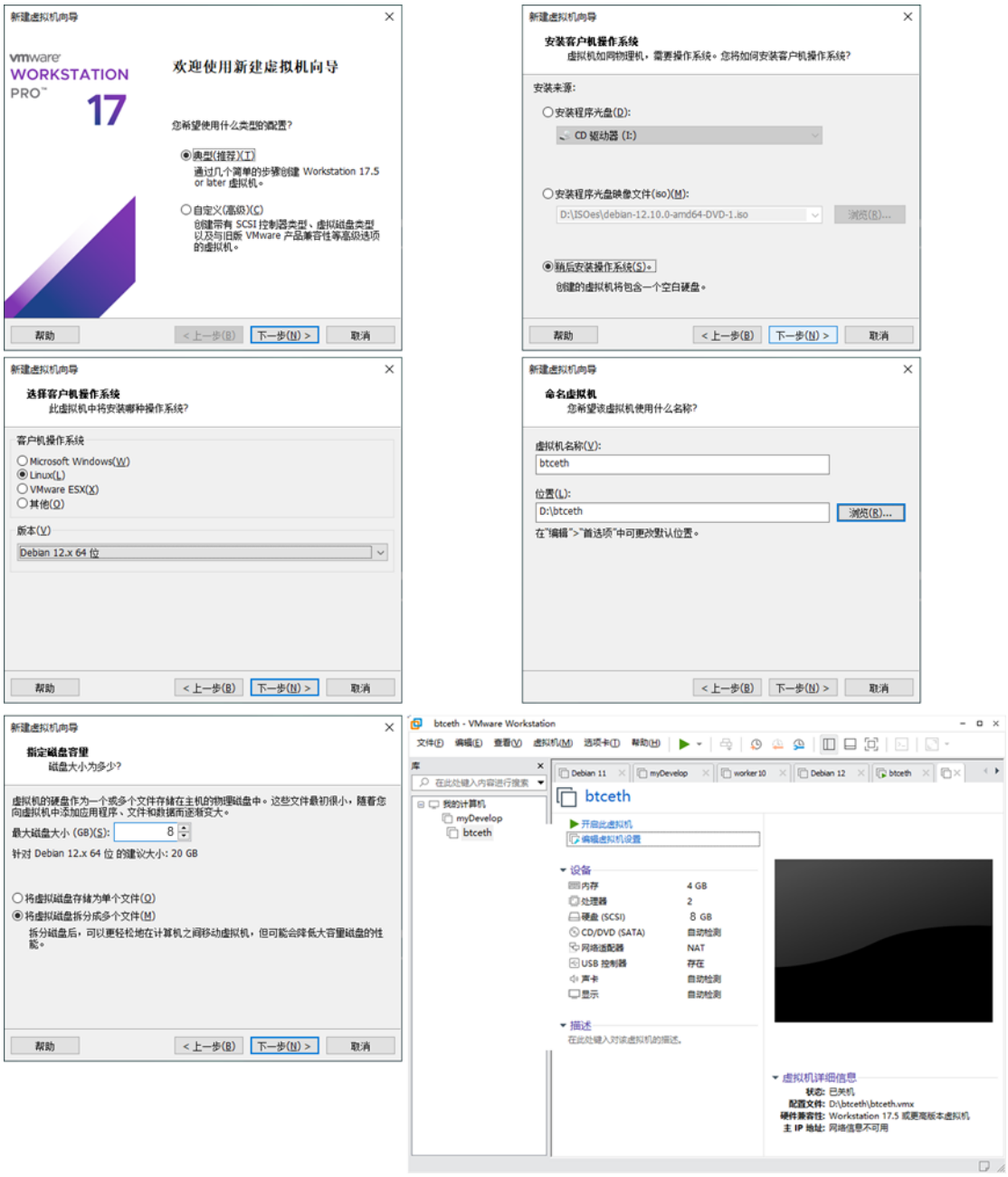


图 7-2 创建虚拟机

在虚拟机的光驱里放入下载号的 debian 安装镜像文件 debian-12.10.0-amd64-DVD-1.iso，最后启动虚拟机开始安装操作系统，参见图 7-3 所示。

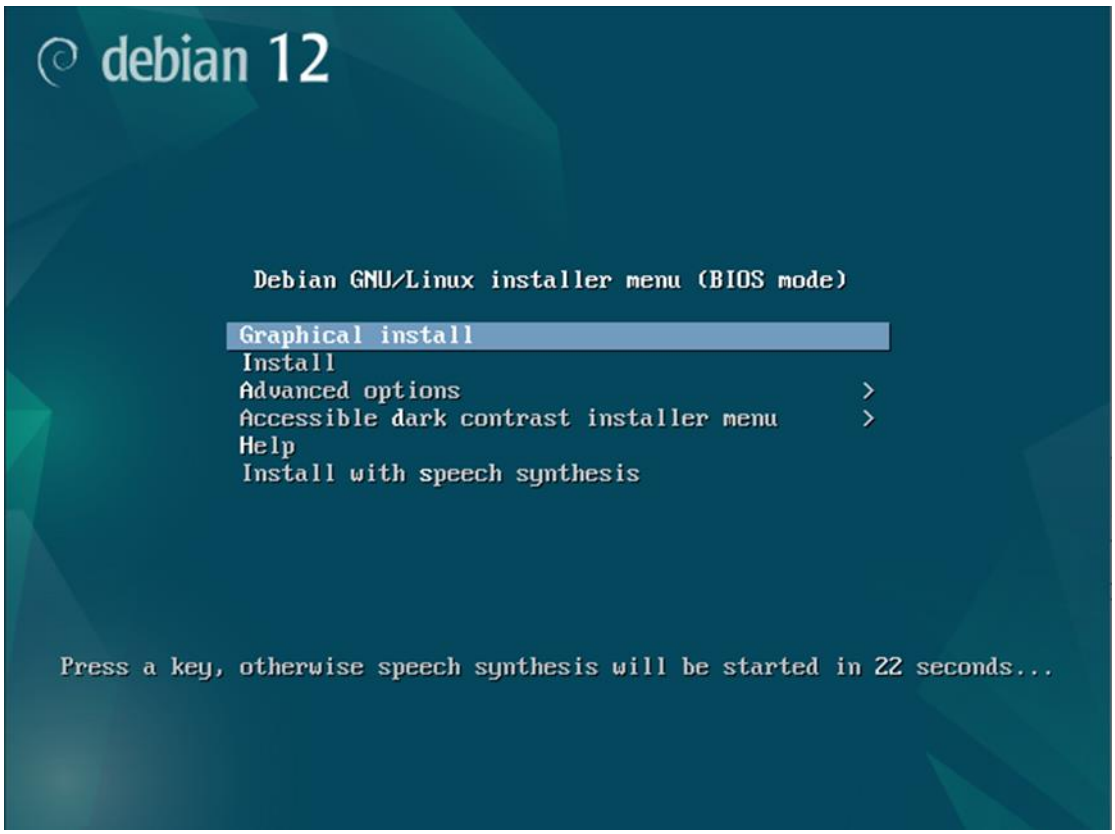


图 7-3 开始安装 Debian

鼠标点入上图的黑屏中，然后按回车开始安装 Linux 操作系统。注意如果要想从虚拟机中移出鼠标，按键盘右侧的 Ctrl 键。

然后都是下一步。设置主机名为 btceth，给 root 设置密码（2&Y），然后再创建普通用户 user，并设置一个密码（H&），user 的密码最好不要与 root 密码相同。直到如图 7-3 所示的画面，选择“Manual”（手工分区）。

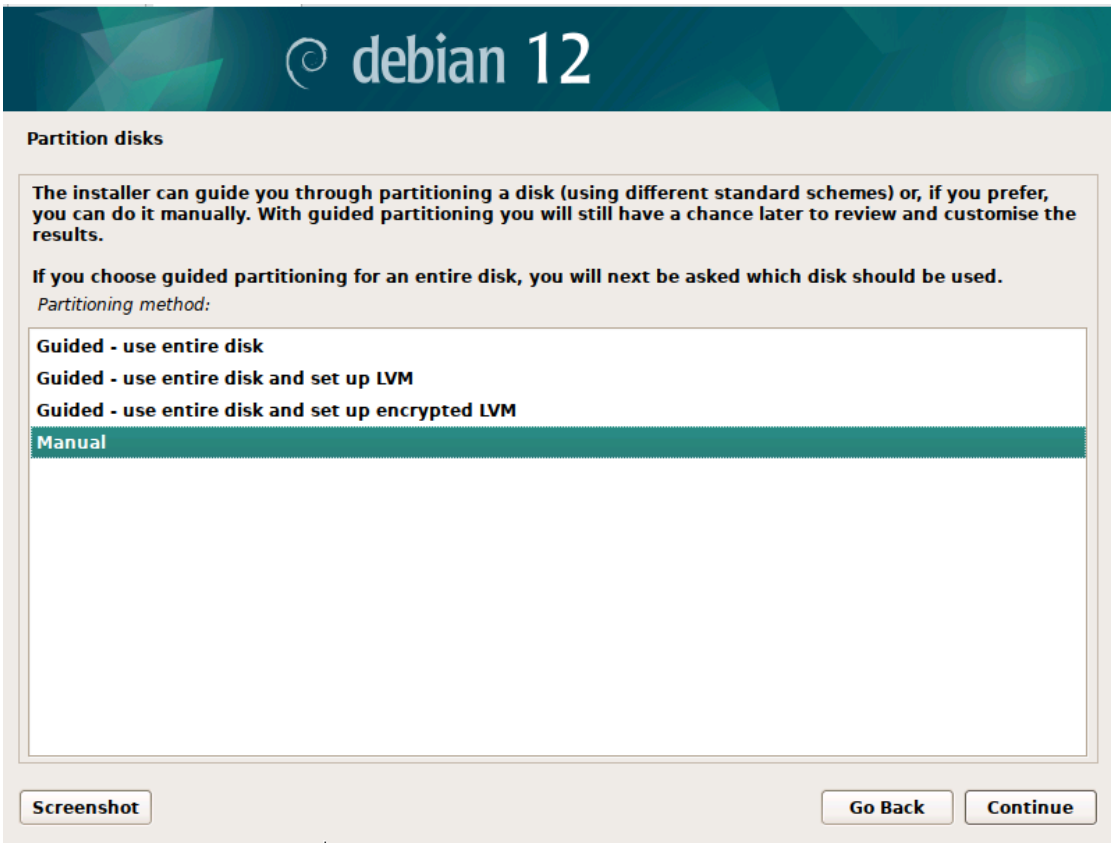


图 7-4 选择手工分区

创建一个 500MB 的小分区，作为加密磁盘，存放敏感数据，剩余空间全部给根分区，不要交换分区，见图 7-5 所示。

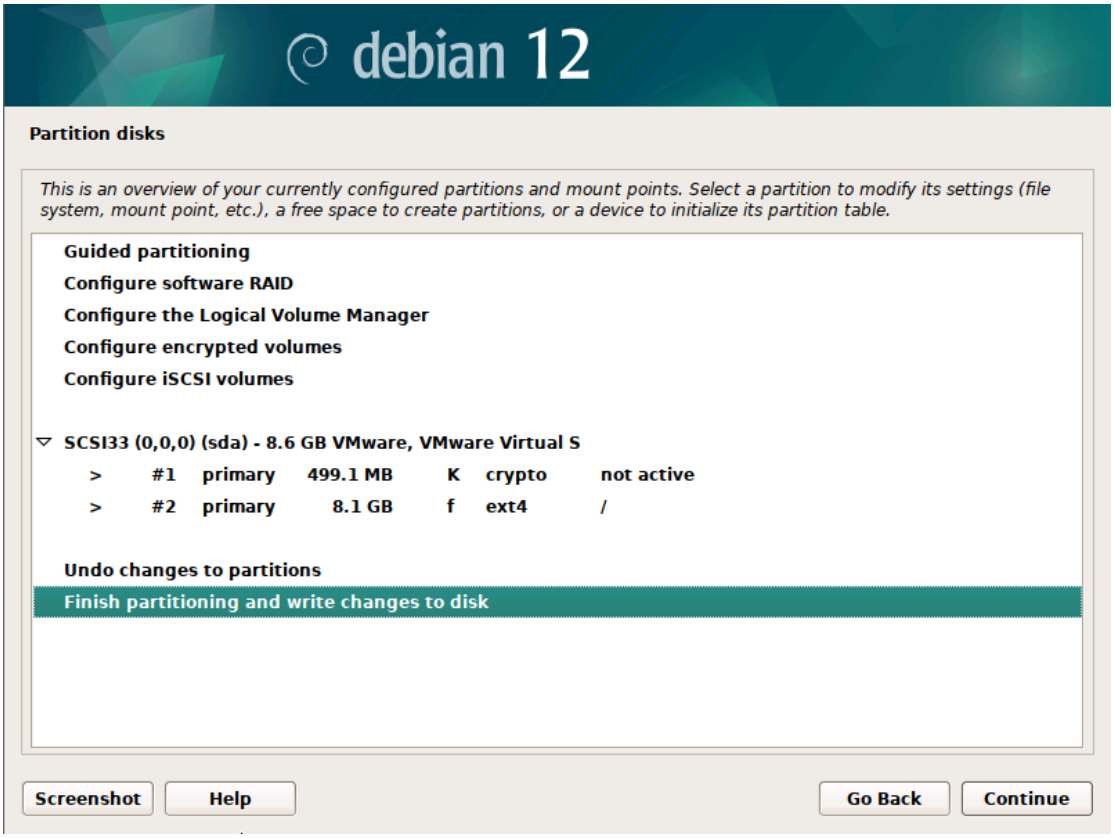


图 7-5 分两个区

接下来的画面选择“no”（不需要交换分区），在接下来选择“Yes”写入磁盘，在“Continue without a network mirror?”界面上选择“Yes”，此后开始正式安装。直到出现如 7-6 所示的画面。这里最小化安装——只选 ssh server 和标准化工具，以后再补上其他软件。

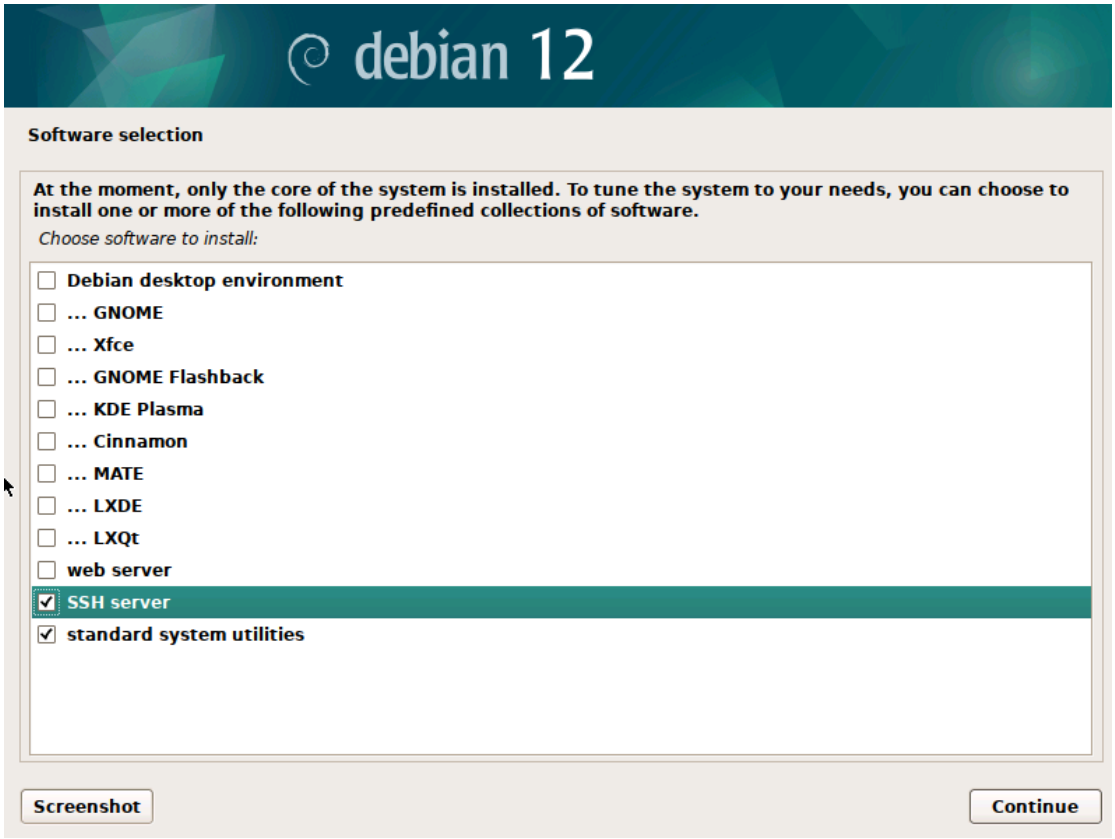


图 7-6 选择左面管理器

接下来一路“Conitue”，直到如图 7-7 所示的界面。

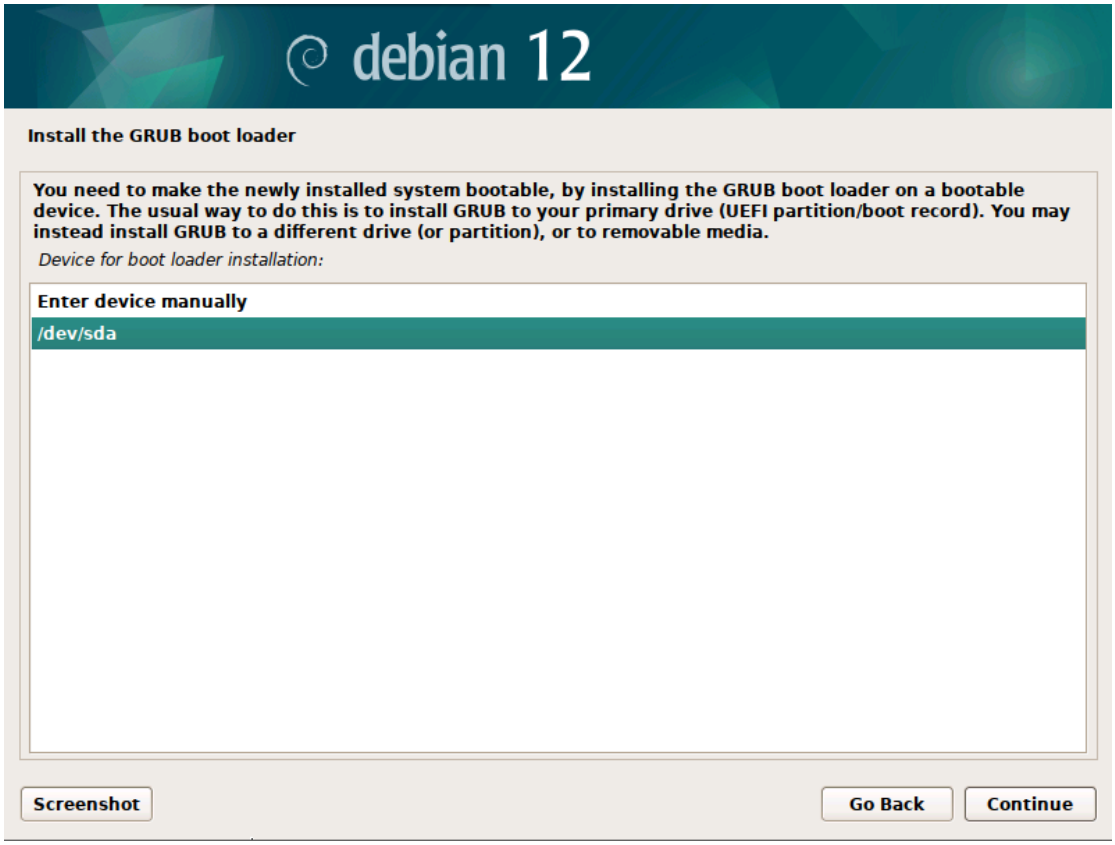


图 7-7 GRUB 安装到主扇区

安装完了重启虚拟机，然后以 root 用户登陆机器，如图 7-8 所示。

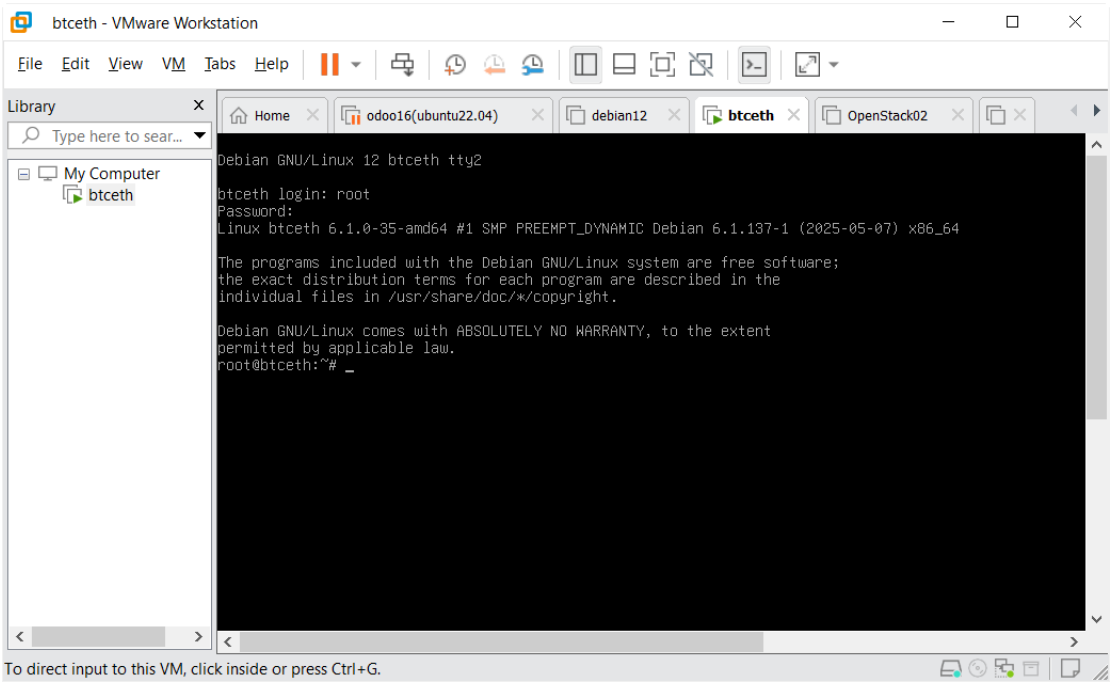


图 7-8 补上一些软件

输入下面的命令补上需要的软件：

```
cat >/etc/apt/sources.list <<EOF
deb http://deb.debian.org/debian bookworm main contrib non-free non-free-firmware
deb http://deb.debian.org/debian bookworm-updates main contrib non-free non-free-firmware
EOF
apt update -y
apt install -y xorg
apt install -y --no-install-recommends xfce4
apt install lightdm -y
apt install -y fonts-wqy-microhei fonts-wqy-zenhei xfce4-terminal cryptsetup vim mousepad nginx
apt install -y firefox-esr --no-install-recommends
shutdown -r now
```

重启之后显示图形登陆画面，见图 7-9 所示。

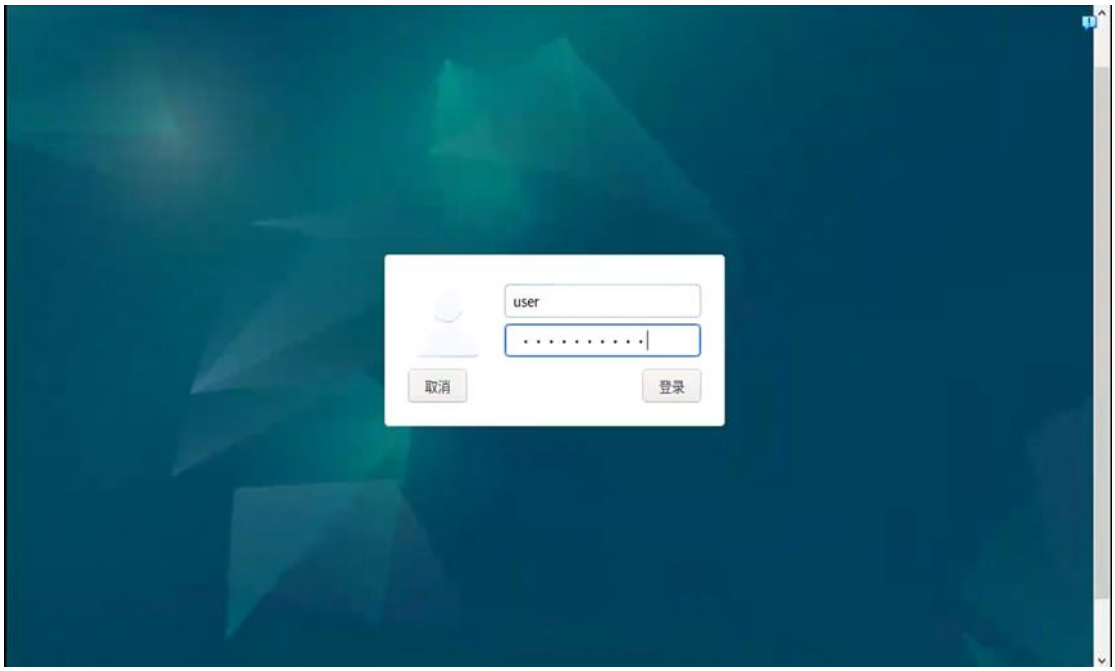


图 7-9 登录到 Debian

禁用虚拟机内外拷贝粘贴和拷贝，点击“编辑虚拟机设置”» “选项”» “客户机隔离”，具体请参考图 7-10 操作。

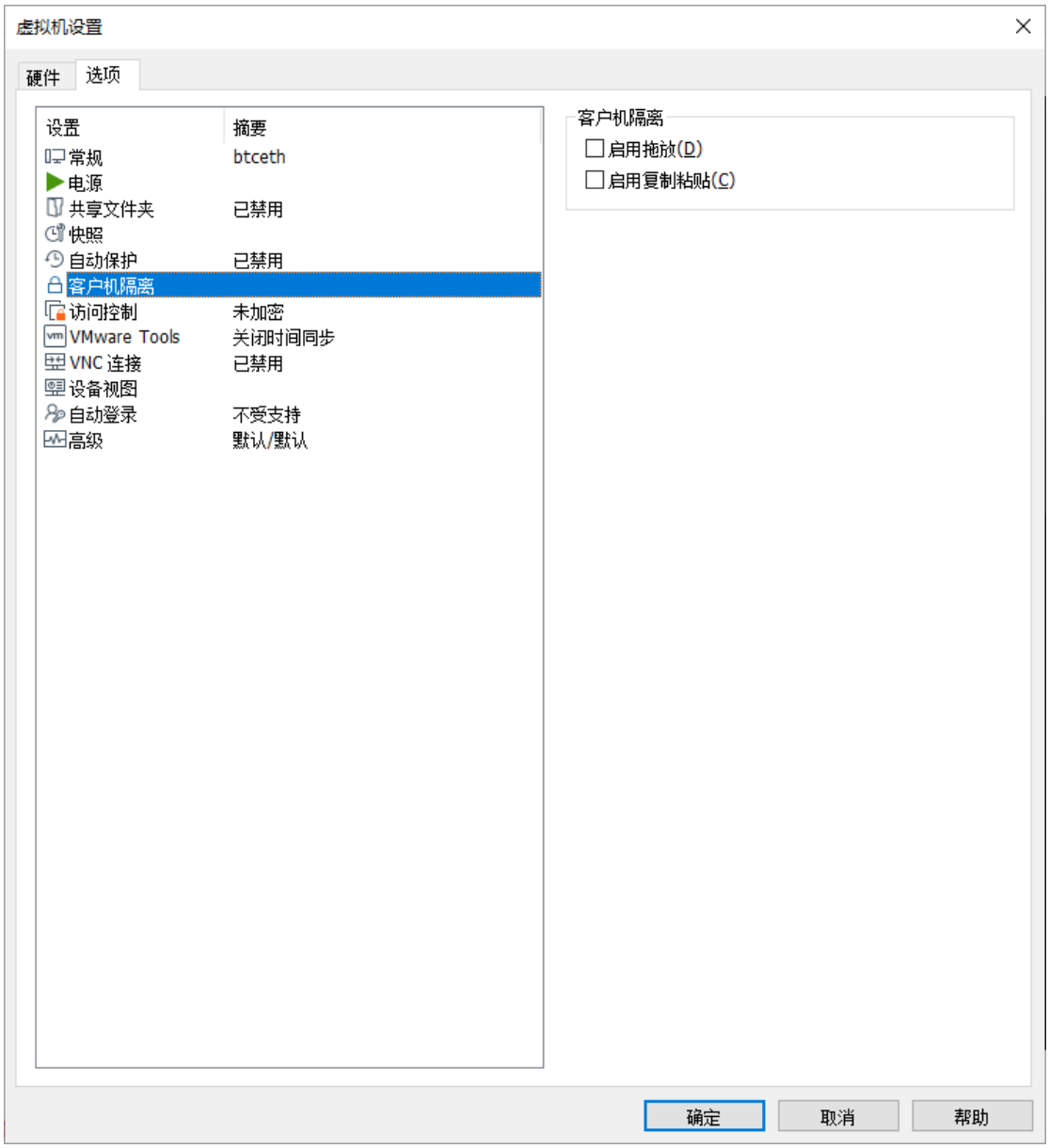


图 7-10 Vmware 禁用虚拟机内外粘贴拷贝

当在虚拟机中拷贝敏感数据时特别要注意，尽量禁止内外拷贝。内外文件拖放也可做类似控制，参考上图中的“拖放”菜单。

配置加密分区：

```
apt-get install cryptsetup
cryptsetup luksFormat /dev/sda1                                    #需要输入密码
cryptsetup open /dev/sda1 encdisk
mkfs.ext4 /dev/mapper/encdisk

//挂载:
mkdir /opt/encrypt
mount /dev/mapper/encdisk /opt/encrypt

//用完卸载:
umount /opt/encrypt
cryptsetup close encdisk
```

开机时自动挂载（不过不建议这样做，需要时手工挂载）：

```
echo 'encdisk /dev/sda1 none luks' >>/etc/crypttab
echo '/dev/mapper/encdisk /opt/encrypt ext4 defaults 0 2' >> /etc/fstab
```

以后重要的资料就保存在/opt/encrypt 目录中即可。此后日常使用：

```
//使用：
cryptsetup open /dev/sda1 encdisk                                    #需要输入密码
mount /dev/mapper/encdisk /opt/encrypt

//用完卸载：
umount /opt/encrypt
cryptsetup close encdisk
```

不记住历史命令：

```
echo "export HISTFILESIZE=0">>/etc/profile
echo "export HISTSIZE=0">>/etc/profile
history -c
```

修改 ssh 的默认端口 22 为 7208：

```
vim /etc/ssh/sshd_config
……
port 7208
……

systemctl restart ssh
```

开启防火墙：

```
apt install firewalld -y
systemctl --now enable firewalld
firewall-cmd --set-target=DROP --permanent                           #默认策略是拒绝
firewall-cmd --remove-service=ssh --permanent
firewall-cmd --add-port=7208/tcp --permanent
firewall-cmd --remove-forward --permanent                              #禁止转发
firewall-cmd --add-icmp-block-inversion --permanent                    #不能 ping
firewall-cmd --reload
```

禁用除 tty7（通常用于图形界面）外的所有 TTY：

```
systemctl mask getty@tty1.service
systemctl mask getty@tty2.service
systemctl mask getty@tty3.service
systemctl mask getty@tty4.service
systemctl mask getty@tty5.service
systemctl mask getty@tty6.service
vim /etc/systemd/logind.conf
……
[Login]
NAutoVTs=7
ReserveVT=7
……

systemctl daemon-reload
systemctl restart systemd-logind
```

卸载 appmor 模块：

```
systemctl stop apparmor
systemctl disable apparmor
apt remove --assume-yes --purge apparmor
sed -i 's/^GRUB_CMDLINE_LINUX=""/GRUB_CMDLINE_LINUX="apparmor=0"/' /etc/default/grub
update-grub
shutdown -r now
```

禁用磁盘日志（放在内存）：

```
vim /etc/systemd/journald.conf
......
[Journal]
Storage=volatile
......


systemctl stop systemd-journald
rm -rf /var/log/journal/*
systemctl restart systemd-journald
```

7.2 准备有关文件

首先参考“§2.4 准备相关文件”小节在 windows 上准备好 D:\nginx-1.28.0\html 目录，然后使用“scp -r D:\nginx-1.28.0\html user@192.168.0.128:./”命令把 html 目录上传到虚拟机 user 用户的主目录中，然后以 root 用户登陆后把 html 目录中的全部文件移到/var/www/html 中，见图 7-12 所示。

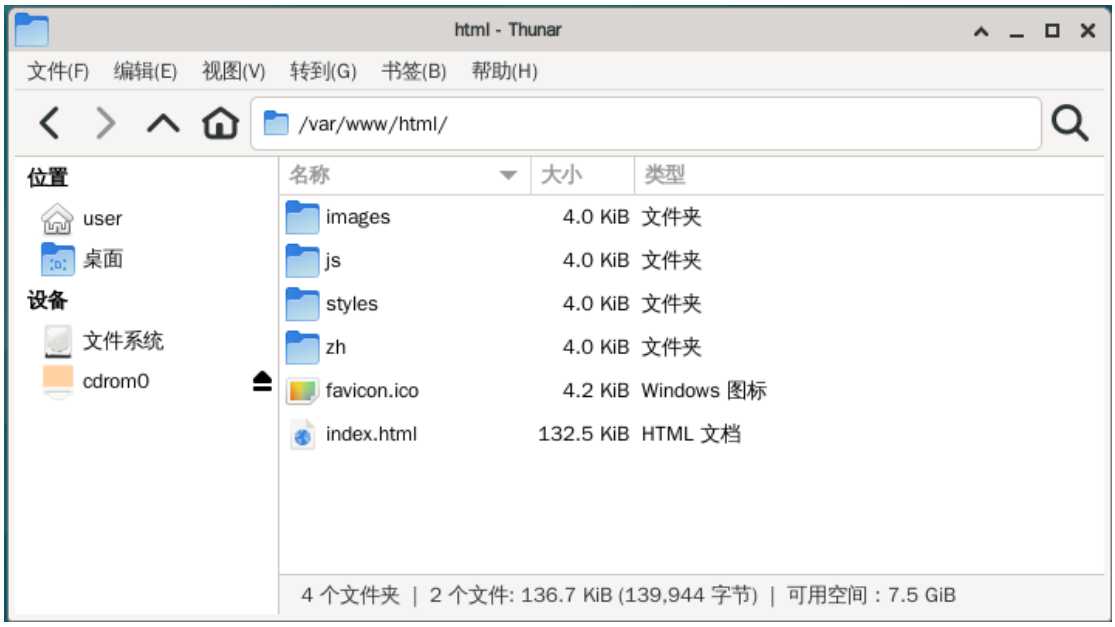


图 7-12 准备相关文件

7.3 准备浏览器

7.3.1 Tor-browser

如果不特别在乎自己的上网行踪，本浏览器可以不安装。

上网时为了隐匿身份，我们采用 Tor 来隐藏自己的互联网出口 ip，而且中国这边本身被长城防火墙阻隔了 Tor 暗网，所以只能先通过梯子爬出去，然后再走 Tor 暗网，这其实隐匿效果更好。

在 windows 这边下载 tor 浏览器 https://dist.torproject.org/torbrowser/14.5/tor-browser-linux-x86_64-14.5.tar.xz，从 windows 把 tor-browser-linux-x86_64-14.5.tar.xz 拖到虚拟机的桌面上，然后再移到 softwares 目录中（在主文件夹中创建目录 softwares）。最后采用下面的命令解压：

```
user@btceth:~$ cd softwares
user@btceth: ~/softwares$ tar -xf tor-browser-linux-x86_64-14.5.tar.xz
```

这样在 softwares 目录中产生一个目录 tor-browser，用鼠标双击此目录进入，如图 7-13。

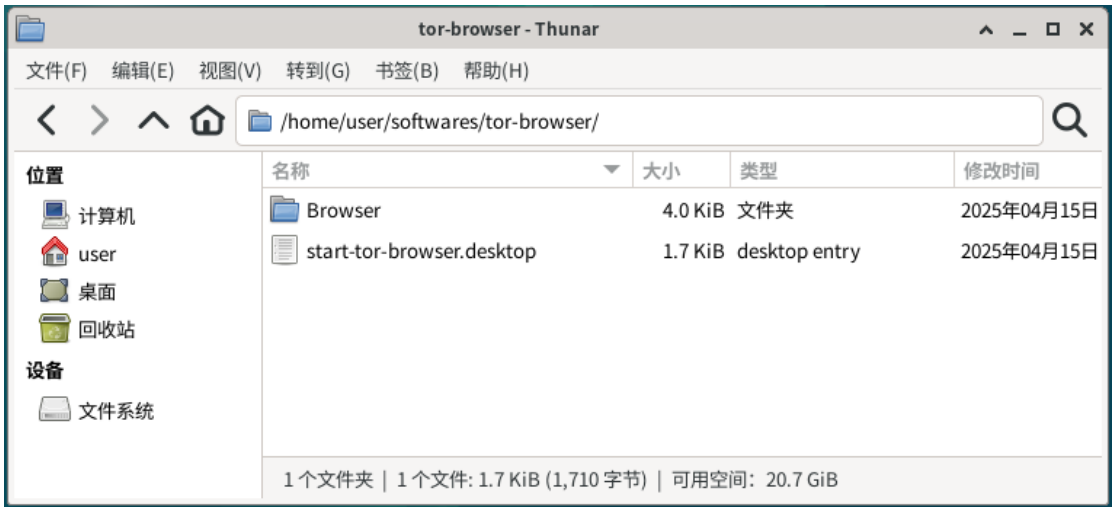


图 7-13 进入 tor-browser 目录

然后双击其中的 start-tor-browser.desktop 文件启动 tor-browser 浏览器，如图 7-14 所示。

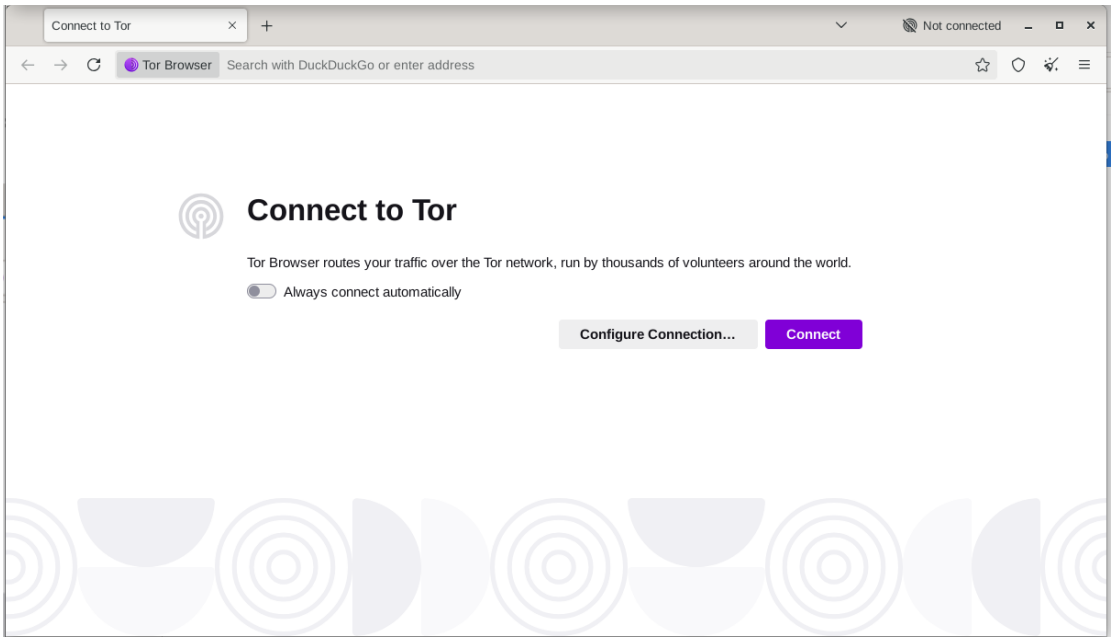


图 7-14 启动 tor 浏览器

点击“Configure Connection...”，参考图 7-15 所示的设置代理。

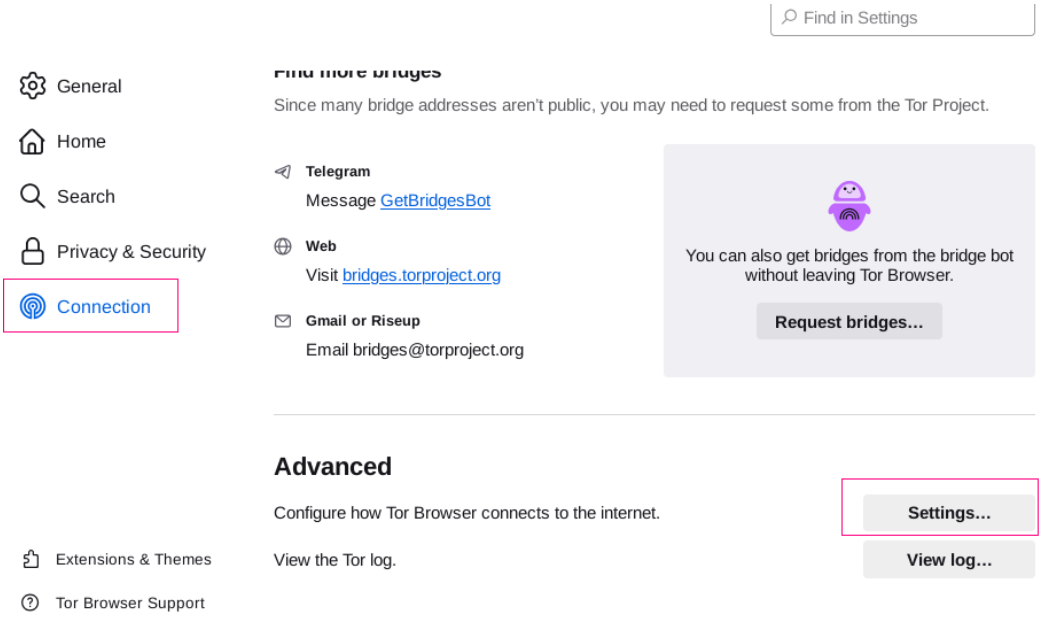


图 7-15 设置代理之一

方法 1：使用自由门做梯子

参考图 7-16 所示设置自由门梯子，假设运行自由门的电脑的 ip 地址为 192.168.0.102。

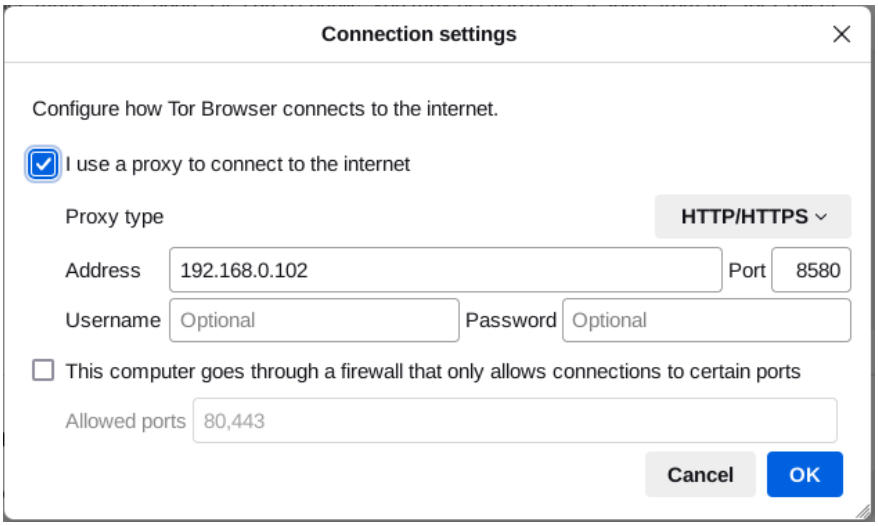


图 7-16 设置 Tor 浏览器的代理

----------方法 1 结束线----------

方法 2：使用 SOCKS5 梯子

在 76-19 中，选择代理类型 socks5，ip 和端口更具实际情况替换，具体参考“§2.2 科学上网”小节。

----------方法 2 结束线----------

最后点击 tor-browser 浏览器中的“Connect”，参考图 7-17 所示。

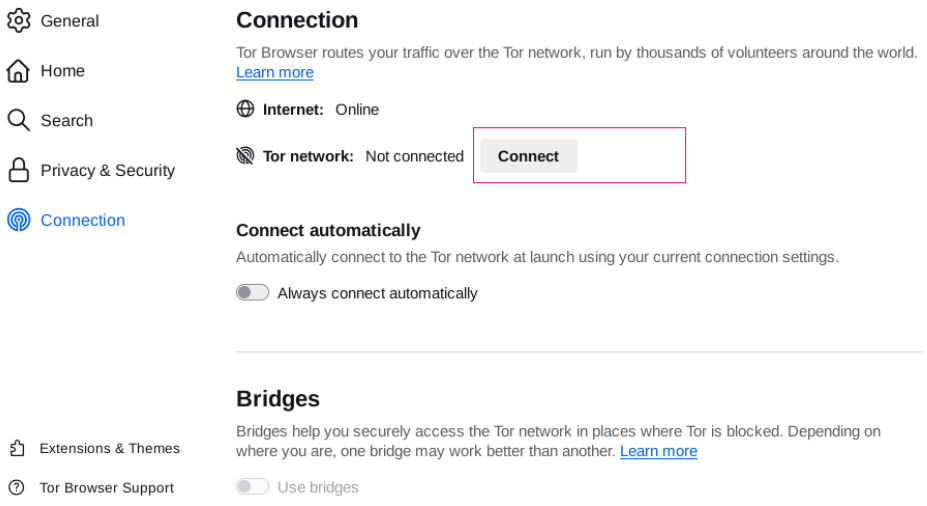


图 7-17 连通暗网

如果看到如图 7-18 所示的画面，表示 Tor 网配置好了。

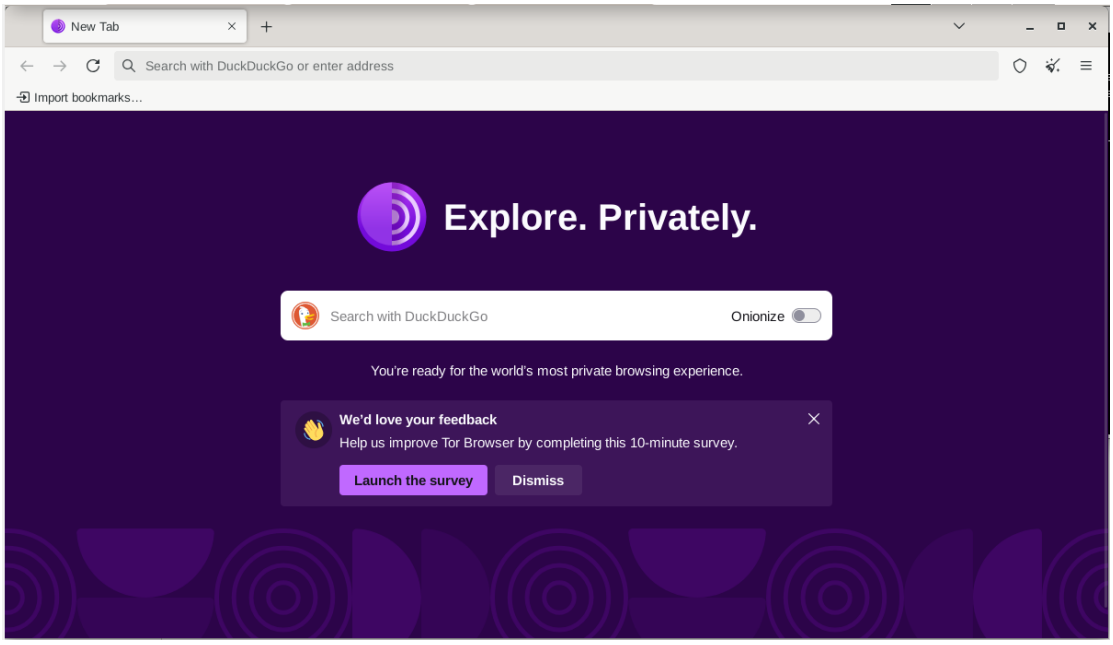


图 7-18 Tor 浏览器准备就绪

输入 www.google.com 看看能否打开。这样以后都用 tor-browser 走暗网到互联网。再浏览网站 https://dnsleaktest.com/检查 DNS 是否被泄露，如果看到的是中国大陆之外的 ip 地址，表明没有泄露。

Tor 的配置文件在/usr/local/etc/torrc.d/目录下，其中 40_tor_control_panel.conf 文件由 anon-connection-wizard 管理（核心参数“Socks5Proxy 10.3.230.36:8080”定义 socks5 的梯子），不要人为修改，用户自定义参数可放在 50_user.conf 文件中。

把左上角的“所有应用程序”»“互联网”»“Firefox ESR”拖到桌面上，双击启动它，设置永久隐私模式：点击右上角的“三”»“设置”，点击左侧的“隐私与安全”，然后参考图 7-19 所示的操作。

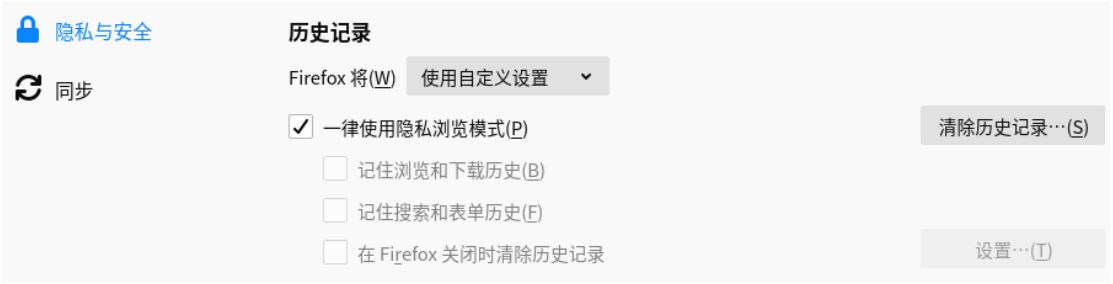


图 7-19 firefox 设置隐私模式

勾选“一律使用隐私浏览模式(P)”，会提示重启浏览器。至于 firefox 的其他设置，请参考“§2.1 准备好浏览器”。

有时需要特别指定 Tor 的出口地区，比如模仿一个美国的用户（出口 ip 落在美国境内），这时可修改 Tor 的配置文件 torrc（位于目录 tor-browser\Browser\TorBrowser\Data\Tor 中），确保存在如下一行：

```
ExitNodes {us} StrictNodes 1
```

类似 us 的国家代码请浏览网站 https://zh.wikipedia.org/zh-sg/%E5%9C%8B%E5%AE%B6%E5%9C%B0%E5%8D%80%E4%BB%A3%E7%A2%BC，其中“二位字母代码”这一列就是。如果要定义若干个出口地区，请用采用类似“ExitNodes {us},{hk} StrictNodes 1”的格式。常见的代码有 us（美国）、hk（香港）、ky（开曼群岛）、fr（法国）、de（德国）、in（印度）、it（意大利）、nz（新西兰）、tw（台湾）、gb（英国）

等。

最后在 tor 浏览器中添加 MetaMask 扩展插件，或者参考附录 C 把此插件装好。

7.3.2 Firefox-esr

firefox-esr 是企业稳定版，这里用来离线打开 localhost 网站，进入加密币交易页面。

```
apt install firefox-esr --no-install-recommends
```

在桌面上建立启动器，并总是以无痕模式启动，启动器的命令如下：

```
/usr/lib/firefox-esr/firefox-esr --private-window ~/www/index.html
```

一定要让浏览器处于离线工作模式下对交易签名。

~~7.3.3 Google-Chrome~~

从 chrome 浏览器经 VPN 上外网。

去 https://www.google.com/chrome/dr/download/ 下载最新版 google-chrome-stable_current_amd64.deb，然后安装：

```
dpkg -i google-chrome-stable_current_amd64.deb
apt -f install -y
```

在桌面上建立启动器，以无痕模式启动并且经过 socks5 代理出去，命令为：

```
/usr/bin/google-chrome-stable --proxy-server="socks5://127.0.0.1:8080" --incognito %U
```

建立 socks5 通道——在桌面上建立启动器 ssh，命令为：

```
/usr/bin/ssh -C -N -D 8080 -i ~/rsa_4096 alice@ xx.yy.zzz.xyz
```

创建登陆公私钥：

```
ssh-keygen -q -b 4096 -C "Socks5 VPN" -t rsa -f rsa_4096                    #产生一对公私钥
ssh-copy-id -i rsa_4096.pub alice@xx.yy.zzz.xyz
```

以后先要启动 ssh，然后才可用 chrome 上网。

下面的命令在 Linux 上配置 Chrome+Tor。

```
sudo apt install tor -y
sudo echo "Socks5Proxy 127.0.0.1:8080" >> /etc/tor/torrc #要提前建立 socks5 VPN
systemctl --now enable tor
cat >>~/.bashrc <<E"O"F
function set-tor-exit() {
   if [ -z "$1" ]; then
      echo "错误：请提供一个两字母的国家代码（例如：us, hk, de, jp）。"
      echo "使用 'reset' 来清除设置。"
      return 1
   fi
```

```
    if [ "$1" = "reset" ]; then
        sudo sed -i '/^ExitNodes/d' /etc/tor/torrc
        sudo sed -i '/^StrictNodes/d' /etc/tor/torrc
        echo "Tor 出口节点设置已清除，将随机选择出口。"
    else
        # 先删除旧的设置，避免重复
        sudo sed -i '/^ExitNodes/d' /etc/tor/torrc
        sudo sed -i '/^StrictNodes/d' /etc/tor/torrc

        # 添加新的设置
        echo "ExitNodes {$1}" | sudo tee -a /etc/tor/torrc
        echo "StrictNodes 1" | sudo tee -a /etc/tor/torrc
        echo "Tor 出口节点已设置为: $1"
    fi

    # 重启 Tor 服务以应用配置
    echo "正在重启 Tor 服务..."
    sudo systemctl restart tor
    echo "Tor 服务已重启。请等待约 30 秒以建立新的链路。"
}
EOF
. ~/.bashrc

set-tor-exit us  #出口切换到美国
set-tor-exit hk  #出口切换到香港
set-tor-exit de  #出口切换到德国
set-tor-exit reset #出口随机选择

google-chrome-stable --proxy-server="socks5://127.0.0.1:8080" #安装 Proxy SwitchyOmega 3 (ZeroOmega)插件
配置 Proxy SwitchyOmega:
安装后，点击浏览器右上角的扩展图标，找到 Proxy SwitchyOmega 并点击“选项”--》在左侧菜单中，点击 “新建情景模式...”--》情景模式名称：输入 Tor (或任何你喜欢的名字)--》类型：选择 “代理服务器”--》点击 “创建”--》在右侧的代理服务器设置中，按照如下填写：代理协议：选择 SOCKS5；代理服务器：填写 127.0.0.1；代理端口：填写 9050--》点击左侧的“应用选项”保存设置。
现在，当你需要通过 Tor 上网时，只需点击浏览器右上角的 Proxy SwitchyOmega 图标，然后选择 Tor 这个情景模式即可。所有流量将通过本地的 Tor 服务。选择“直接连接”则恢复正常上网。

浏览 https://check.torproject.org/，看看是不是走 tor
再浏览 https://dnsleaktest.com/，看看出口 ip 所在地。
```

~~7.4 安装 Bisq（不安装）~~

通过 https://bisq.network/交易真正实现了比特币的理想：去中心化、匿名和透明。但是必须在本地电脑上安装一个软件，而不能直接在网站上交易，这样意味着私钥只掌握你自己手里(此软件要在线使用，这就不能绝对保证私钥的安全性，所以建议在 bisq 中购买的加密币还是要通过手工方法转移到自己的伪纸质钱包中)。

先从 https://bisq.network/downloads/下载安装包和签名文件，参考图 7-20。

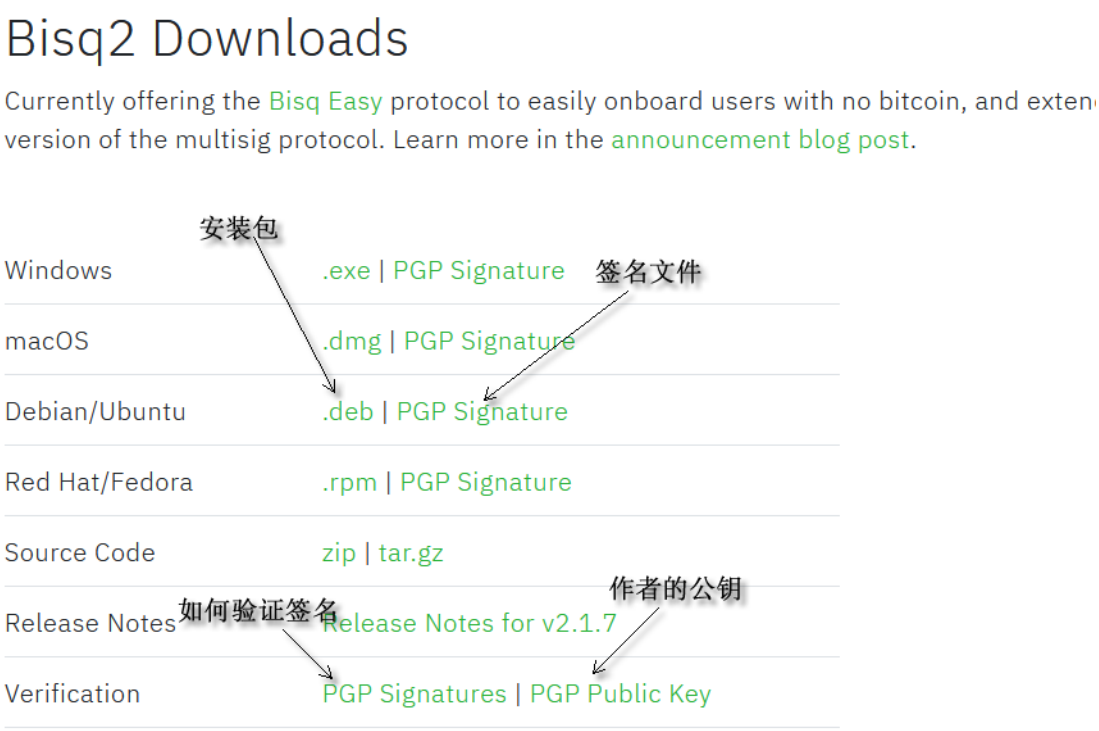


图 7-20 下载 bisq

下载作者的公钥 E222AA02.asc、安装包 Bisq-2.1.7.deb、签名文件 Bisq -2.1.7.deb.asc，三个文件都放在 softwares 目录中，这样原始数据（安装包）、签名（签名文件）和公钥都齐了，接下来就可以验证签名了。也可以用下面的命令下载：

```
scurl-download https://github.com/bisq-network/bisq2/releases/download/v2.1.7/Bisq-2.1.7.deb
scurl-download https://github.com/bisq-network/bisq2/releases/download/v2.1.7/Bisq-2.1.7.deb.asc
scurl-download https://github.com/bisq-network/bisq2/releases/download/v2.1.7/E222AA02.asc
```

执行命令 gpg --import E222AA02.asc 导入公钥，再执行命令 gpg --digest-algo SHA256 --verify Bisq-2.1.7.deb.asc 验证签名，作者把自己的公钥 ID 和指纹数据都公布在 https://bisq.wiki/Downloading_and_installing#Verify_installer_file 页面上，图 7-21 是部分页面截图。



图 7-21 作者的公钥和指纹数据

具体参考图 7-22 进行验证。

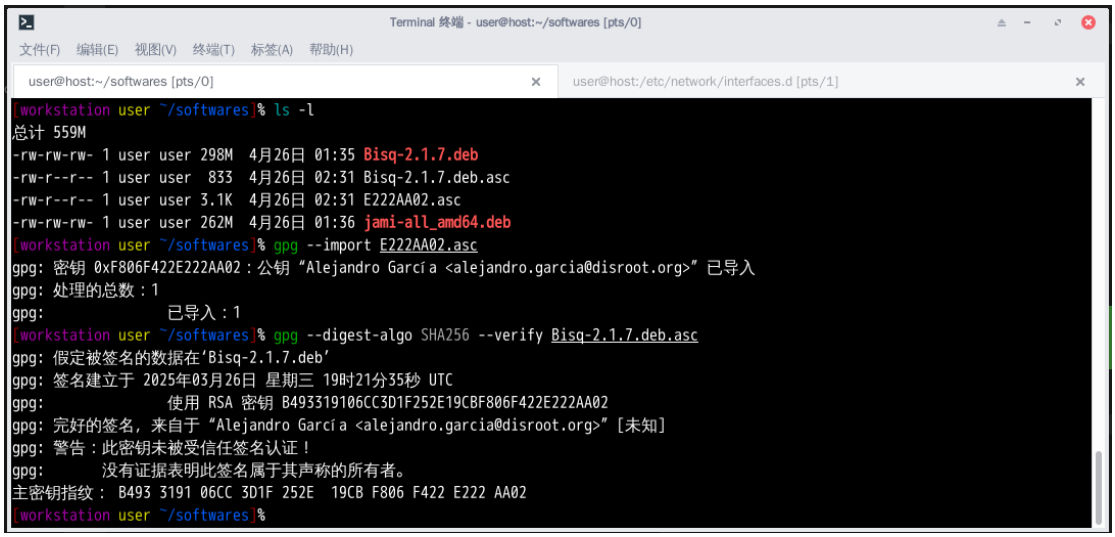


图 7-22 验证签名

注意：上图中的警告表示作者的私钥是自己产生的，不是由权威的第三方机构颁发的，我们可以忽略这些警告。

执行下面几个命令以避免 Tor 走 Tor 的循环圈子：

```
mkdir -p /home/user/.local/share/Bisq2/tor
sudo touch /home/user/.local/share/Bisq2/tor/tor
sudo chmod +x /home/user/.local/share/Bisq2/tor/tor
```

最后运行命令安装：

```
sudo apt install tor -y
sudo dpkg -i Bisq-2.1.7.deb

//配置 tor 走 socks5 前置代理（默认使用网桥）：
sudo vim /etc/tor/torrc
……
Socks5Proxy 127.0.0.1:8080
……
UseBridges 0
……
sudo systemctl restart tor
```

点击左上角的菜单“所有应用程序”下的“互联网”中的 Bisq2 启动它（它走 Tor 出去），如图 7-23。

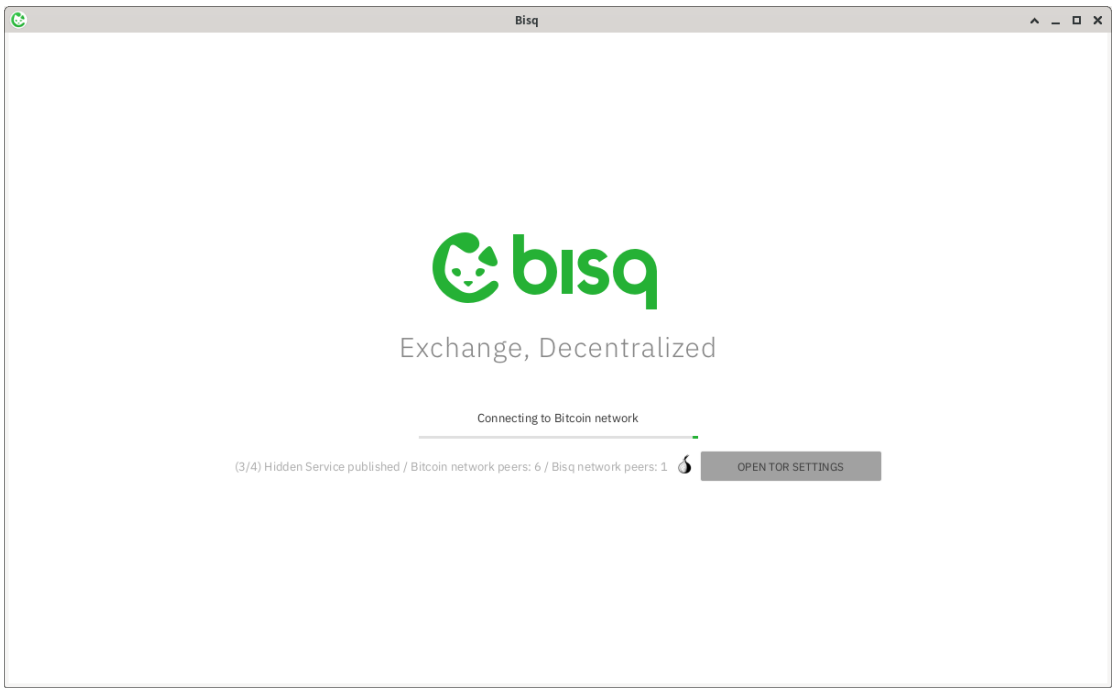


图 7-23 启动 Bisq

Bisq 初始化非常耗 CPU，初始化完毕后（大概半个小时内）就正常了，首先提示输入一个名字，然后进入主画面，见图 7-24 所示。

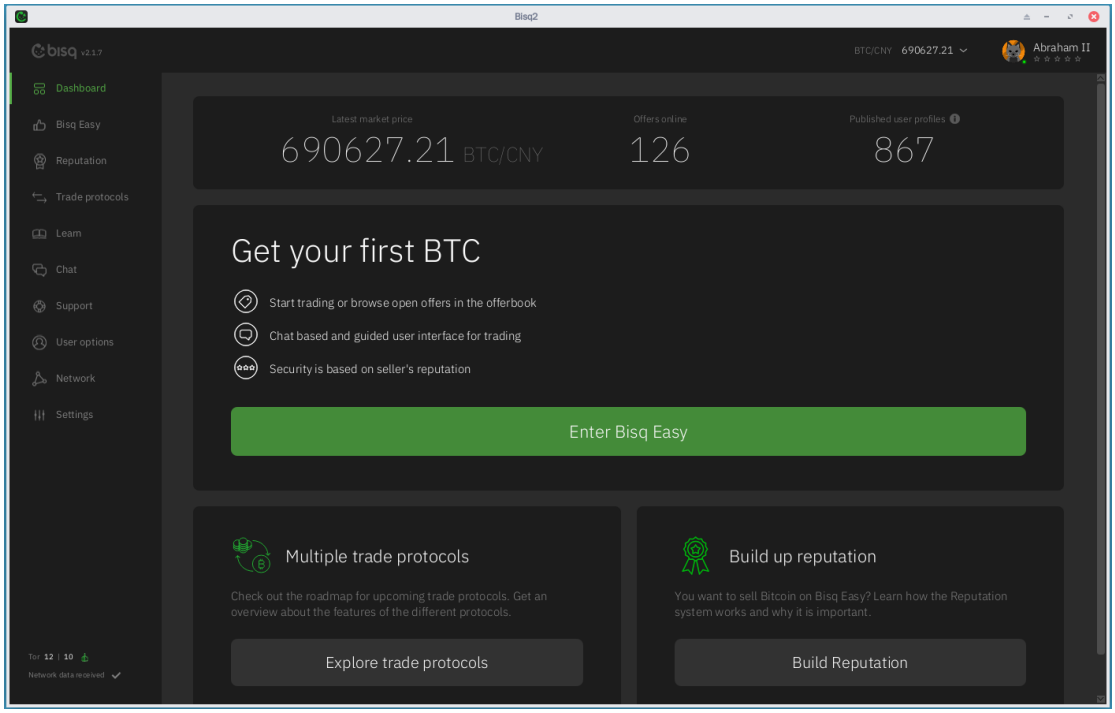


图 7-24 Bisq 启动后的界面

Bisq 的交易思路是：引入一个监督者（经过 Bisq 认证且存有 2 个比特以上的财富），通过一个多签钱包，购买者需要先缴纳一定比率的保证金，当交易完成后，保证金会退还。交易费比较高，由转账双方承担，交易费包括监督者的佣金、bisq 的中间费和矿工的挖矿费用，所以 bisq 适合金额比较大的交易。

~~7.5 安装 jami（不安装）~~

开源软件 Jami 是一个去中心化的点对点即时通信工具，可以用文字、语音和视频沟通，通信内容只有天知地知你知我知（我和你是通信的双方），放心，绝对没有老大哥盯着。打开网站 https://jami.net/download-jami-linux/，下载 Debian 操作系统的的版本，见图 7-25 所示。

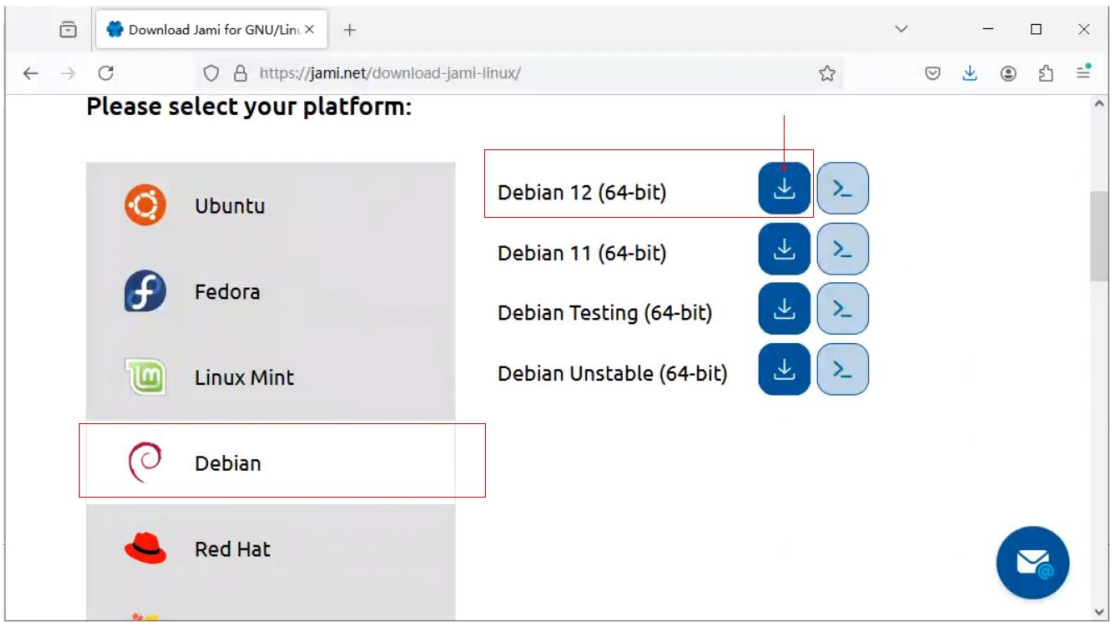


图 7-25 下载最新版 jami

我下载的是这个 https://dl.jami.net/ring-manual/debian_12/jami-all_amd64.deb。

采用下面的命令安装 jami：

```
sudo dpkg -i ./jami-all_amd64.deb
```

如果报错，不妨试试下面的命令来安装依赖的软件：

```
sudo apt install -f
```

然后点击菜单“所有应用程序”»“互联网”»“Jami”启动它，首先看到如图 7-26 所示的画面。



图 7-26 启动 Jami

如果你以前没有 Jami 账户，就点击“创建 Jami 账户”新建一个，参考图 7-27。



图 7-27 创建用户

创建用户实际上就是创建一对基于 Ed25519 的公私钥，都是 32 个字节长度，私钥保存在本地，公钥通过分布式网络（OpenDHT）广播出去，公钥的哈希值（160 位二进制数）作为用户的默认 id，如 f1b7a56f6ed69f25ddc281f194b8737466970baa，允许修改 id 为便于记忆的名字，如 xitler，不过名字必须全网唯一。每个用户在 OpenDHT（分布式哈希表）中保存一条记录“<id>:<公钥>:<互联网出口 ip>”，当用户 A 想要和用户 B 通讯时，在 OpenDHT 中查询用户 B 的 id（用户名或公钥的哈希），得到用户 B 的公钥和互联网出口 ip，然后与“<互联网出口 ip:4222/UDP>”建立 p2p 链接，如果不能建立 p2p 链接，只有通过 TURN 中继转发了。通讯双方每次对话会协商一个临时的对称密钥（也是 256 位二进制数），然后用此密钥对会话内容加解密。

显然用户的互联网出口 ip 暴露了地理位置隐私，让 Jami 走 Tor 暗网就可以了。

如果已有一个账户，就点击“从归档备份导入”按钮，然后定位到那个备份文件，如图 7-28 所示。



图 7-28 从备份中恢复账户

直接把我的 id 发给对方，这样双方可以开通信，如图 7-29 所示。

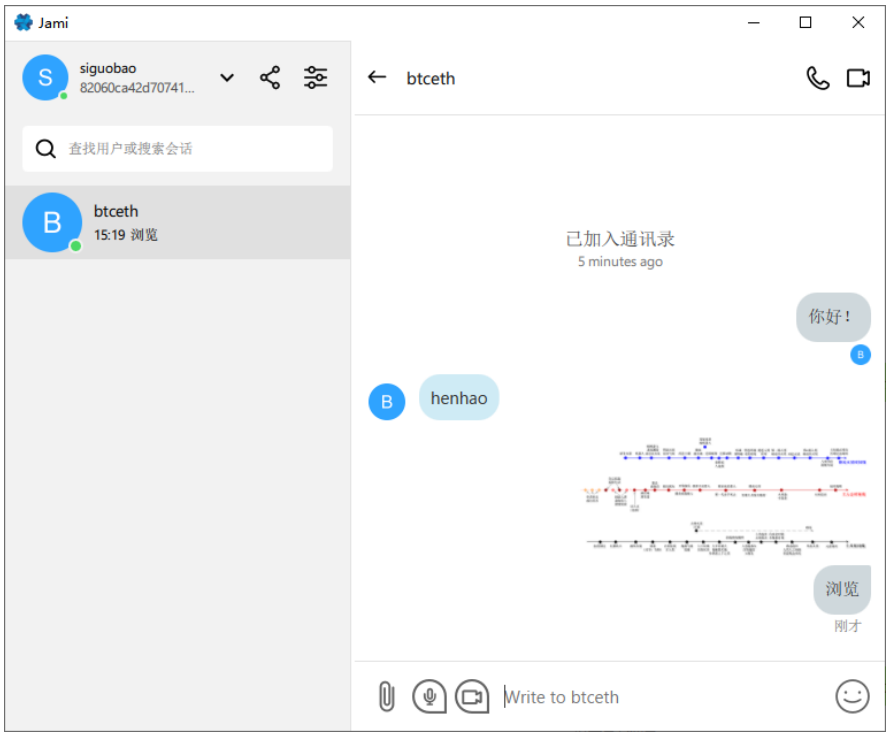


图 7-29 与别人聊天

因为账户信息值保存在本地，所以设备坏了就会永久丢失私钥（即账户），所以要做好备份。点击左侧栏右上角的齿轮图标，左侧边点击“账户”»“管理账户”，右侧滚到最底部，然后点击“备份”按钮，参考图 7-30 操作。



图 7-30 备份 Jami 账户

由于完全是 P2P 网络，当任一方无网络时是无法发“离线”消息的。由于同一个账户允许在多台设备上登陆，所以最佳的实践是：在一台平时不关机且连网的电脑上登陆 jami，同一个账户也在移动设备上登陆，这样即使移动设备无网络，电脑上依然可以收到信息。

同一本 Jami 允许登陆多个账户，点击右上角的账户，在弹出的界面上点击“添加其他账户”，参见图 7-31 操作。

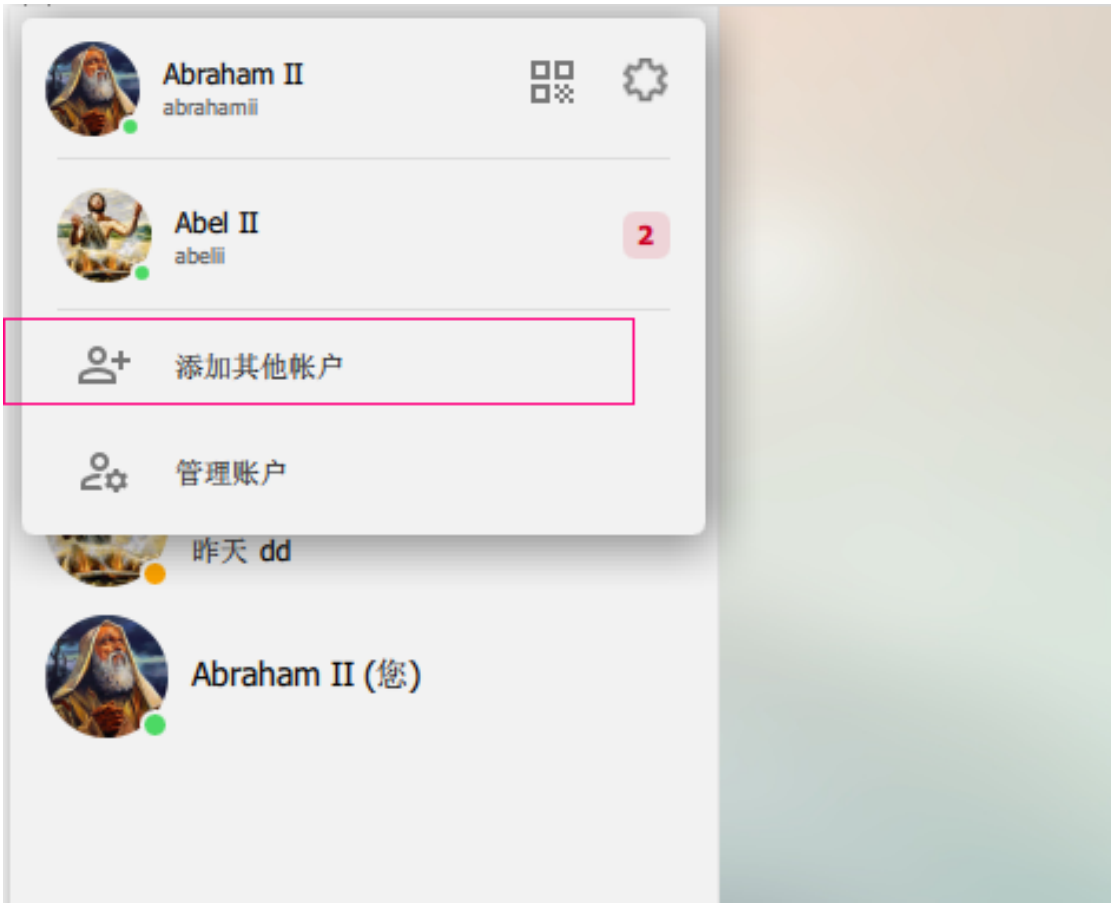


图 7-31 登陆多个账户

最后给虚拟机瘦身：

```
apt clean
apt autoremove --purge
journalctl --vacuum-time=1d
rm -rf /var/log/*.gz /var/log/*.old /var/log/*.log /var/log/apt/*
rm -rf /usr/share/doc/* /usr/share/man/* /usr/share/info/*
rm -rf /tmp/* /var/tmp/*
```

清零全部空余空间：

```
dd if=/dev/zero of=/file.zero
rm -rf /file.zero
shutdown -h now
```

整理磁盘碎片，参考图 7-32 所示，这样大概能瘦身 2GB。

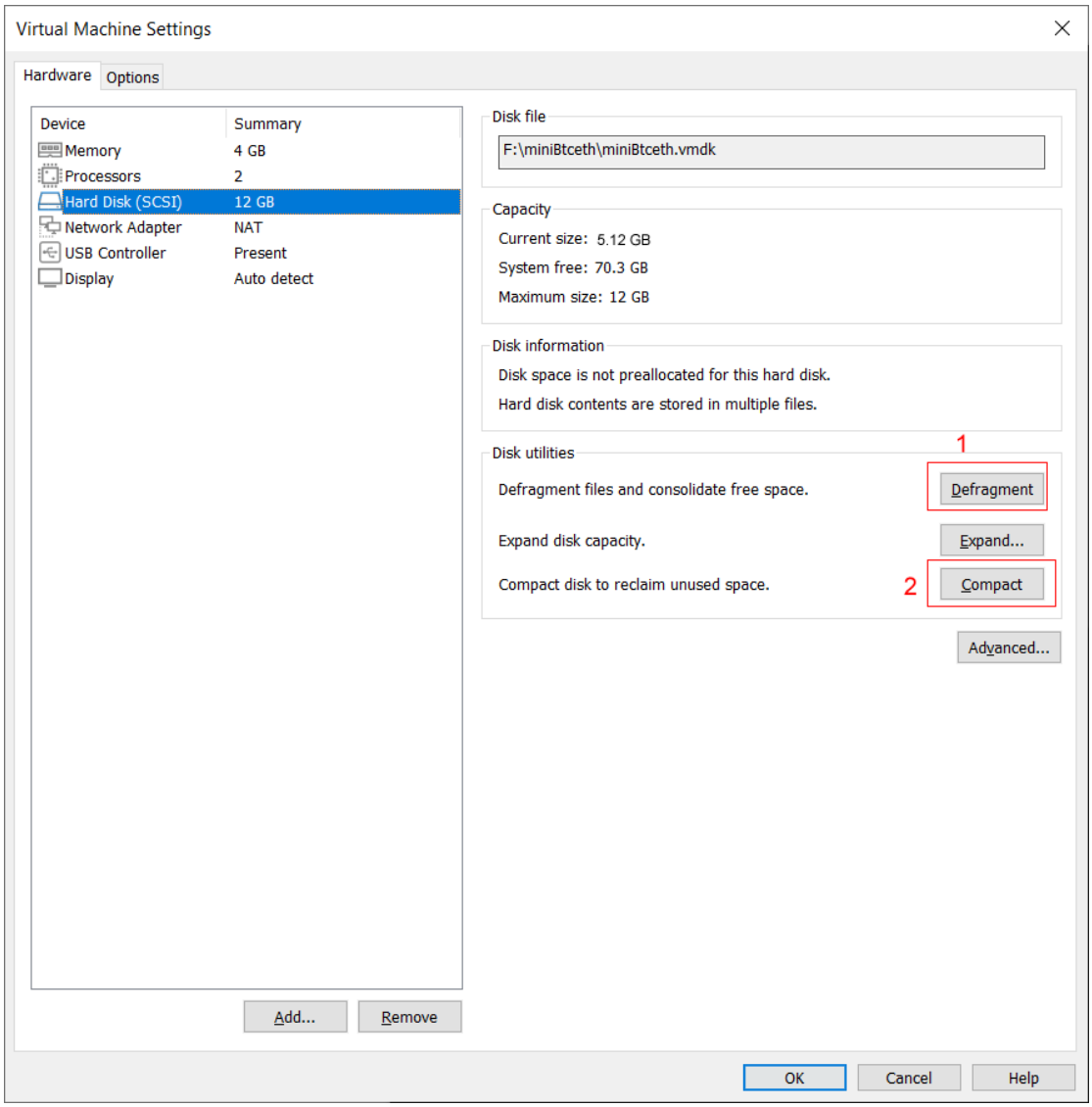


图 7-32 整理虚拟机磁盘

固化虚拟机磁盘，对于 VMWare 虚拟机软件，请参考图 7-33 所示操作。

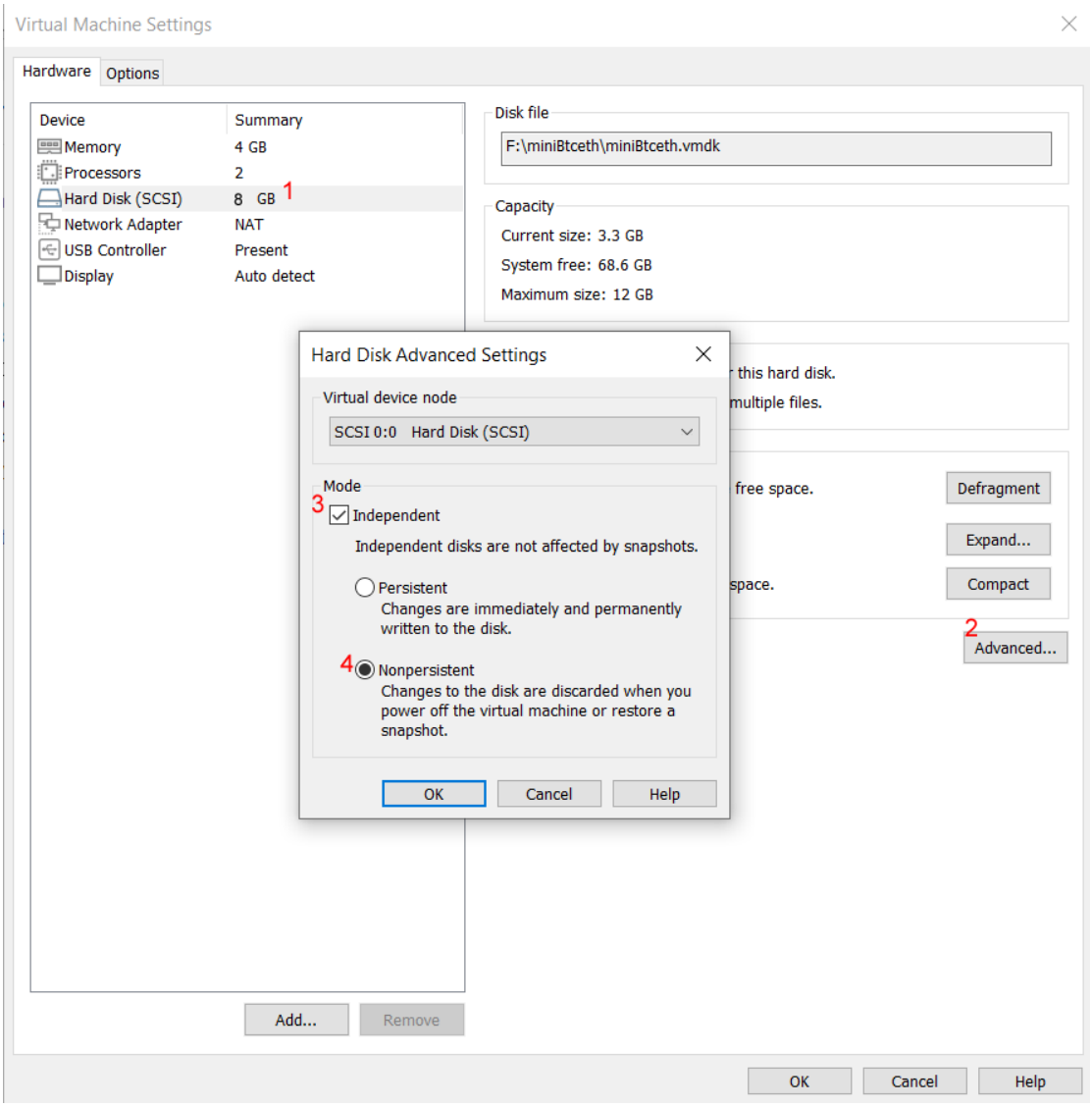


图 7-33 固化虚拟机磁盘

7.6 给虚拟机瘦身

一、删除用不上的文件和目录

```
vim /etc/fstab
UUID=14198b57-b022-45f0-afbb-325b4ff8ab78 /           ext4    strictatime,errors=remount-ro 0     1

shutdown -r now
```

2. 正常使用半个月后：

```
find /usr/share -type f -atime +15 -mtime +15 -exec mv {} {}.delete0724 \;
```

在/usr/lib/modules/下，把不要的旧版删掉

```
find /usr/lib/modules/6.1.0-32-amd64/kernel/drivers/ -type f -atime +15 -mtime +15 -exec mv {} {}.delete0724 \;
find /usr/lib/modules/6.1.0-32-amd64/kernel/sound/ -type f -atime +15 -mtime +15 -exec mv {} {}.delete0724 \;
find /usr/lib/modules/6.1.0-32-amd64/kernel/fs/ -type f -atime +15 -mtime +15 -exec mv {} {}.delete0724 \;
find /usr/lib/modules/6.1.0-32-amd64/kernel/net/ -type f -atime +15 -mtime +15 -exec mv {} {}.delete0724 \;
find /usr/lib/x86_64-linux-gnu/ -type f -atime +15 -mtime +15 -exec mv {} {}.delete0724 \;

find /usr/share/apt-listchanges/ -name "*.delete0724" -exec sh -c 'mv "$0" "${0%.delete0724}"' {} \;
mv /usr/lib/modules/6.1.0-32-amd64/kernel/drivers/md/{dm-mod.ko.delete0724,dm-mod.ko}
```

如果出现问题，用下面的命令恢复全部文件：

find /usr/share -type f -name "*.delete0724" -exec sh -c 'mv "$0" "${0%.delete0724}"' {} \;

3. 再正常使用一段时间后：

```
find /usr -type f -name "*.delete0724" -delete
dmesg -C
```

重启后，用 dmesg 命令查看缺少文件的报错信息，恢复这些文件。

把/etc/fstab 文件中的“strictatime,”删掉。

二、文件碎片整理

4. 给虚拟机增加一个 3G 的硬盘。

5. 从 https://www.system-rescue.org/Download/下载 systemrescue 光盘，用词光盘启动机器，然后：

```
mkfs.ext4 /dev/sdb
mount /dev/sdb /mnt
fsarchiver savefs /mnt/sda2.fsa /dev/sda2
dd if=/dev/zero of=/dev/sda2 bs=4M status=progress
fsarchive restfs /mnt/miniBtceth.fsa id=0,dest=/dev/sda2
```

6. 恢复分区：

```
dd if=/dev/zero of=/dev/sda2 bs=4M status=progress
fsarchiver restfs /mnt/sda2.fsa id=0,dest=/dev/sda2
```

三、收缩磁盘

6. VMware 虚拟机》Setting》Hard Disk》Defragment 和 Compact 按钮。

7. 用 7-zip 压缩虚拟机并发布。

7.7 备份和恢复虚拟机

给虚拟机磁盘加密，这样就算别人把你的虚拟机拷贝走了，他也是没法启动虚拟机，也无法直接查看虚拟机磁盘中的内容。关闭虚拟机，进入设置界面，然后参考图 7-33 所示操作。

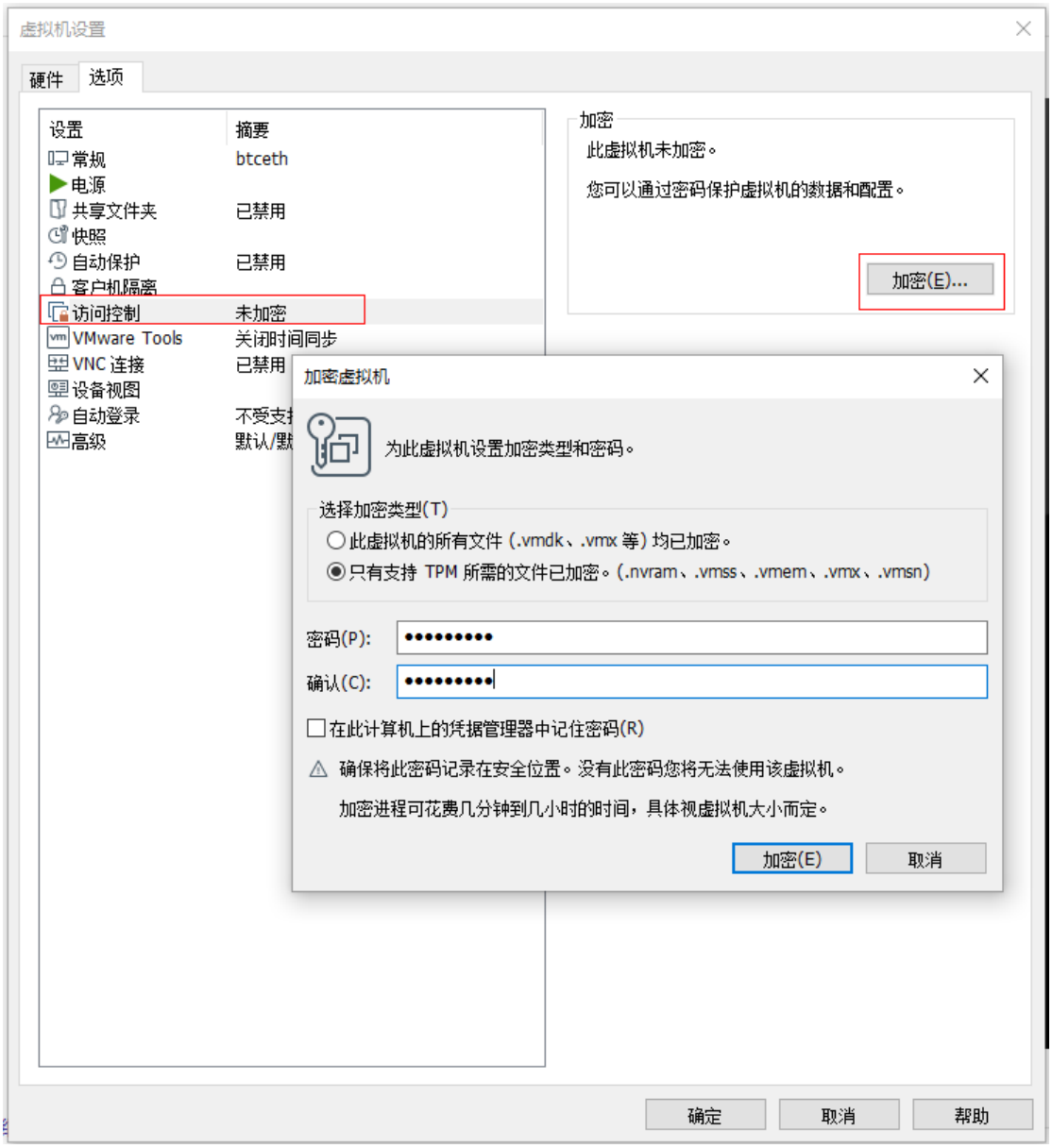


图 7-35 Vmware 虚拟机加密

备份虚拟机比较简单，直接压缩虚拟机在书宿主机的目录即可。

7.8 钱包与交易

地址、公钥以及被密码保护的私钥就放在虚拟机里的 wallet.txt 文件中，建议对此文件加密保护，使用时解密原文，使用如下的命令：

```
gpg --symmetric wallet.txt                                    //采用对称密码加密文件 wallet.txt
gpg -d wallet.txt.gpg                                         //解密后的内容直接显示在屏幕上
```

密码尽量复杂一些，加密后的文件名为 wallet.txt.gpg，放在和 wallet.txt 相同的目录中，删除源文件 wallet.txt。gpg 命令当然也支持非对称加密。

加密后的 wallet.txt.gpg 文件放在加密分区中。

切记：无密码保护的私钥不要放在虚拟机里，更不要放在 windows 的电脑上。

钱包管理和放币交易请参考第 6 章，这里就不再赘述了。

8 高级操作环境

高级操作环境采用全部的安全增强措施：在一台完全没有网络的电脑上签名，在另一台联网的电脑上发布已经签名的交易，发布交易和浏览加密币网站都走 Tor 网。在中国大陆，Tor 网必须走 VPN 出去。所以我们采用三台电脑，其中一台专门用作暗网网关，这里用三台虚拟机充当电脑，效果和安全性与物理机一样的，而且还更方便操作。三台虚拟机中全部安装 Debian linux 操作系统，布局图参考图 8-1 所示。

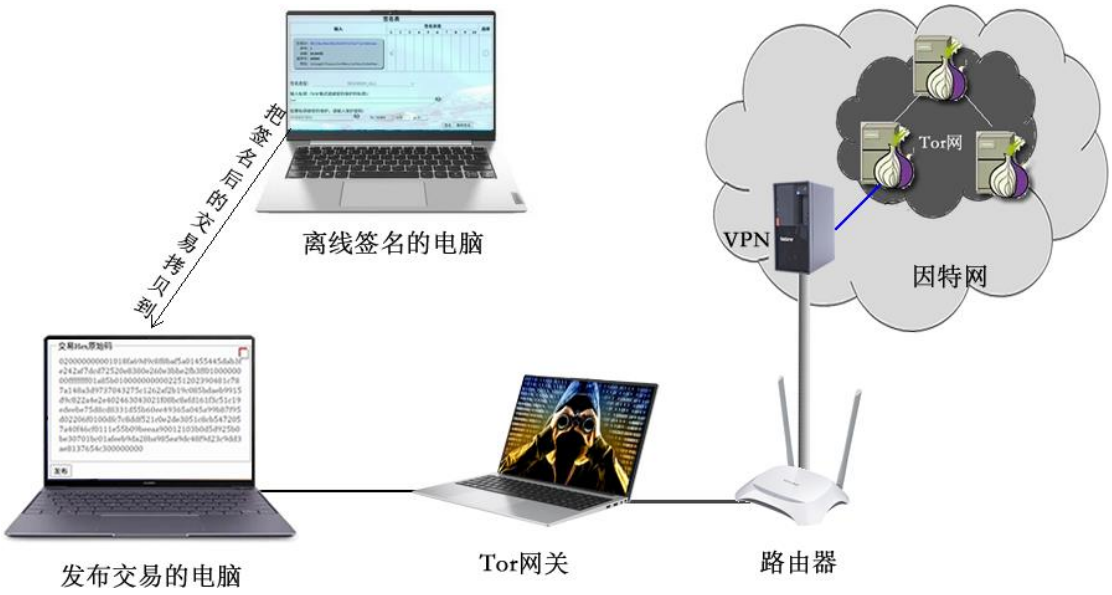


图 8-1 三台电脑分工图

在离线电脑上创建交易并签名，在发布电脑上查询钱包 UTXO、余额和发布签名后的交易，网络流量全经过 Tor 网关，Tor 网关与互联网上的一台 VPN 服务器建立加密隧道，Tor 网关上的 tor 客户端通过此 VPN 隧道接入 Tor 入口主机，Tor 入口主机链接 Tor 中间主机，Tor 中间主机再连接 Tor 出口主机到达因特网。这样我的因此受到双重隐匿。

8.1 准备环境

你必须熟悉 virtualbox 虚拟机软件，了解 linux 操作系统，能了解 nodejs 当然更好。虚拟机软件、操作系统、浏览器、DNS 域名服务、即时通信工具等全部采用“粉碎凌镜”网站（https://prism-break.org/en/all/）上的推荐的，即：virtualbox 虚拟机、debian（或 qubes）操作系统、tor-browser 浏览器、dnscrypt-proxy 域名服务、jami 点到点即时通信工具、bisq 去中心化加密币交易工具等。图 8-2 是此网站推荐的操作系统。

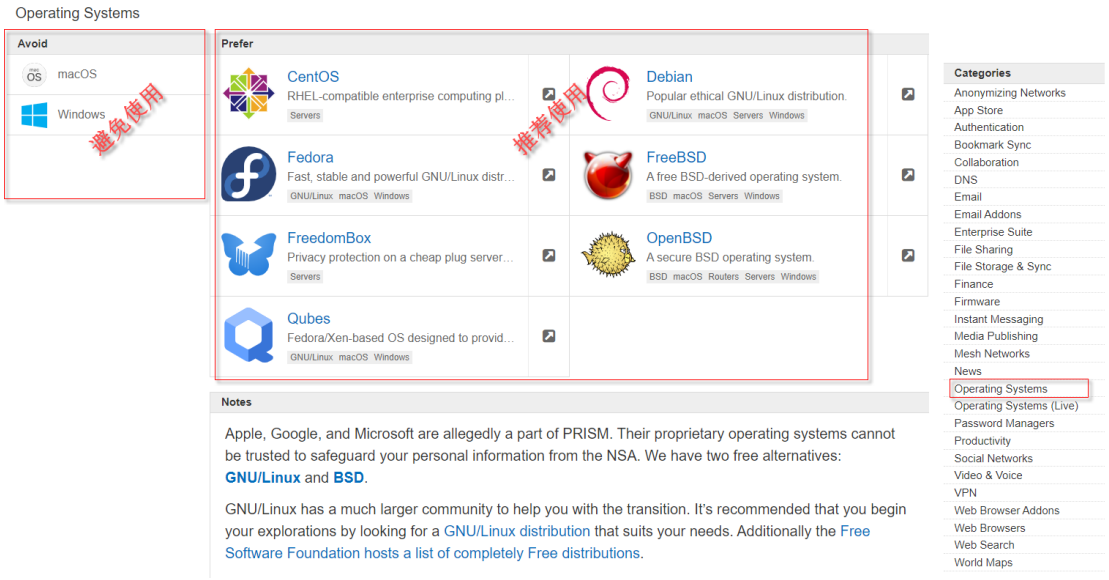


图 8-2 “粉碎凌镜”网站推荐的操作系统

8.1.1 安装 virtualbox

virtualBox 是开源软件，但是属于 Oracle 公司的，所以并不怎么“自由”。

先从 https://download.virtualbox.org/virtualbox/7.1.10/VirtualBox-7.1.10-169112-Win.exe 下载虚拟机软件，下载后安装它。再从 https://download.virtualbox.org/virtualbox/7.1.10/Oracle_VirtualBox_Extension_Pack-7.1.10-169112.vbox-extpack 下载扩展包，扩展包支持磁盘加密，双击下载的 Oracle_VirtualBox_Extension_Pack-7.1.10-169112.vbox-extpack 安装即可。安装完后启动 VirtualBox，见图 8-3 所示。

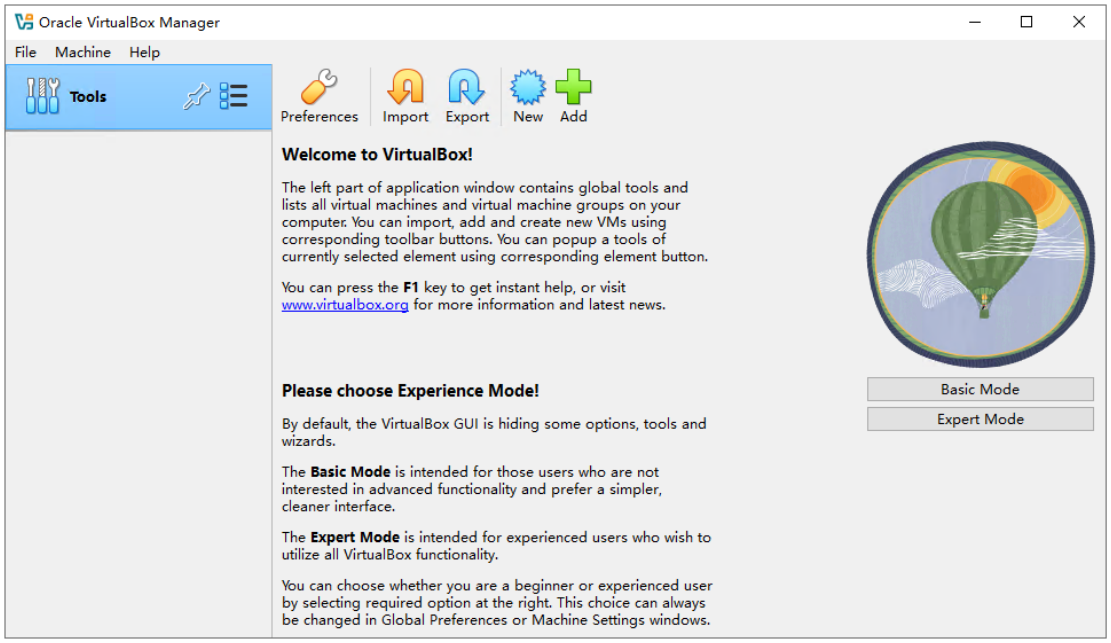


图 8-3 VirtualBox 的界面

8.1.2 导入 Whonix

我们采用安全性极高的 Whonix 操作系统，这是一款基于 Debian 12 定制的且已经配好 Tor 的安全操作系统，号称是世界上最无懈可击的隐私操作系统。打开其官网的下载页面 https://www.whonix.org/wiki/Download，找到适合于自己操作系统的安装包，见图 8-4。

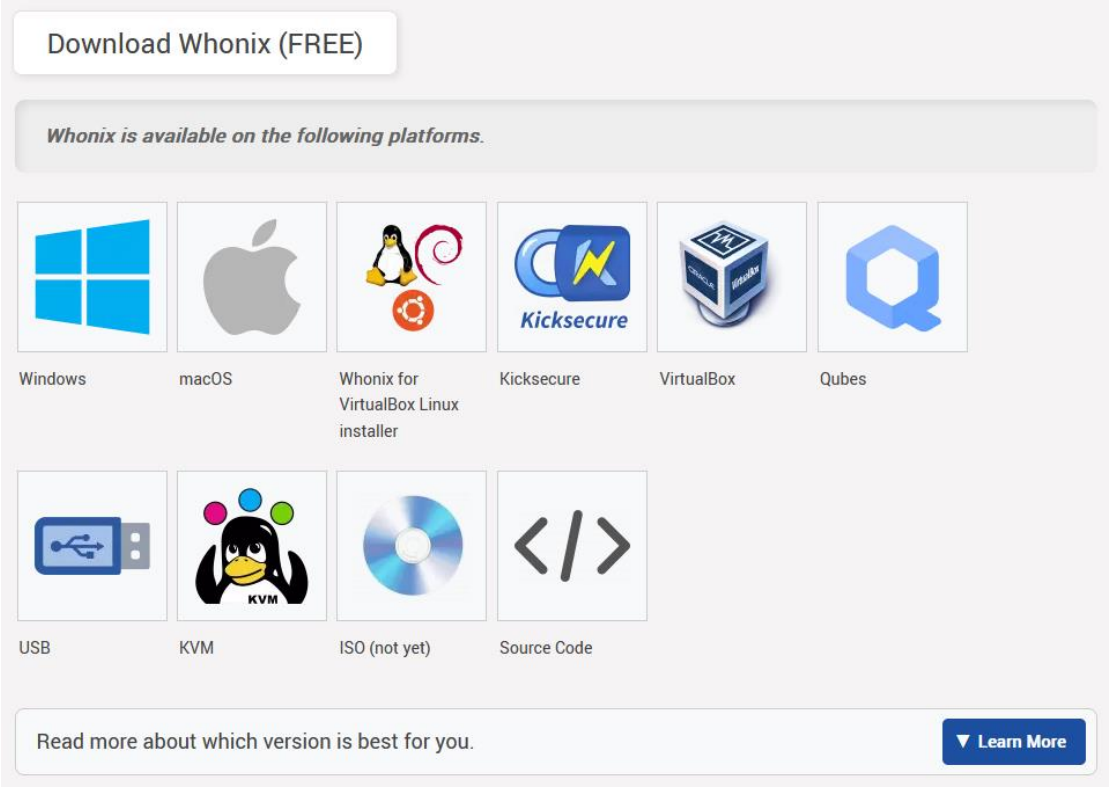


图 8-4 选择适合与自己操作系统的镜像包

图 8-4 告诉我们 Whonix 可以运行的平台，如果你的电脑是 windows 操作系统，请点击图中的 windows 图标，如果是 Linux 且想使用 virtualbox 虚拟机，请点击 Whonix for VirtualBox Linux installer 图标，如果是 Linux 且使用 Kvm 虚拟机，请点击 KVM 图标，Whonix 还可以驻留在 U 盘中，方便随身携带，那么请点击 USB 图标。如果你不知道选择，请点击“Learn More”进一步了解自己的选择，参见图 8-5 所示。

OS	Host	Knowledge	Recommendation	Virt	Status	Freedom
	Windows	Newcomer	Windows (Download)		Production	FREEDOM
	Linux	Newcomer	Whonix for VirtualBox Linux installer		Production	FREEDOM
	Intel-based MacBook	Newcomer	macOS (Download for Intel-based MacBook)		Production	FREEDOM
	Apple Silicon	Newcomer	macOS (Download for Apple Silicon)		Development	FREEDOM
	Non-Apple Hardware Hackintosh	Newcomer	macOS (Download for Non-Apple Hardware Hackintosh)		Experimental	FREEDOM
	Qubes	Advanced	Qubes-Whonix™ (Download)		Production	FREEDOM
	Linux	Advanced	KVM (Download)		Production	FREEDOM
	USB	Newcomer	USB		Production	FREEDOM
	ARM64 (Architecture)	Advanced	install Whonix on ARM64		Experimental	FREEDOM
	Raspberry Pi (Hardware)	Advanced	install Whonix on Raspberry Pi		Experimental	FREEDOM
	ppc64el (Architecture)	Advanced	install Whonix on ppc64el		Experimental	FREEDOM
	RISCV64 (Architecture)	Advanced	install Whonix on RISCV64		Experimental	FREEDOM
	QEMU (Tiny Code Generator)	Advanced	-		Experimental, Unsupported	FREEDOM
	Any	Advanced	-		Experimental, Unsupported	FREEDOM
	2 PCs, personal computer, notebook	Advanced	x86 compatible		Experimental	FREEDOM
	32-bit	Advanced	Whonix 32 bit information		Production	FREEDOM
	Source Code	Advanced	Build Documentation		Production	FREEDOM
	VMware	Advanced	VMware		Experimental, Declined Feature [1]	NON-FREEDOM
	Hyper-V	Advanced	Hyper-V		Experimental, Declined Feature [1]	NON-FREEDOM

图 8-5 选择适合自己的 Whonix

首选还在实验（Experumental）阶段的不能用，由于我的电脑安装的是 windows，所以点击图 8-5 中的第一行的 Windows(Download)进入下载页面，见图 8-6 所示。

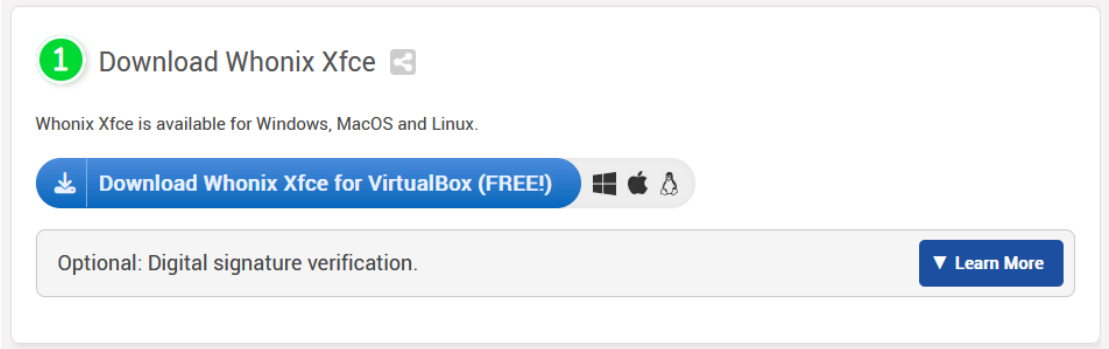


图 8-6 下载 Whonix Xfce for VirtualBox

是这样的，Whonix 镜像只能安装恢复到 VirtualBox 虚拟机里，就下载 windows 的 whonix，我是下载这个：https://www.whonix.org/download/ova/17.3.9.9/Whonix-Xfce-17.3.9.9.Intel_AMD64.ova。下载完毕后点击图 8-3 中的



“Import”按钮（或者直接按 Ctrl+i），把之前下载的 Whonix 导入进来，参见图 8-7。

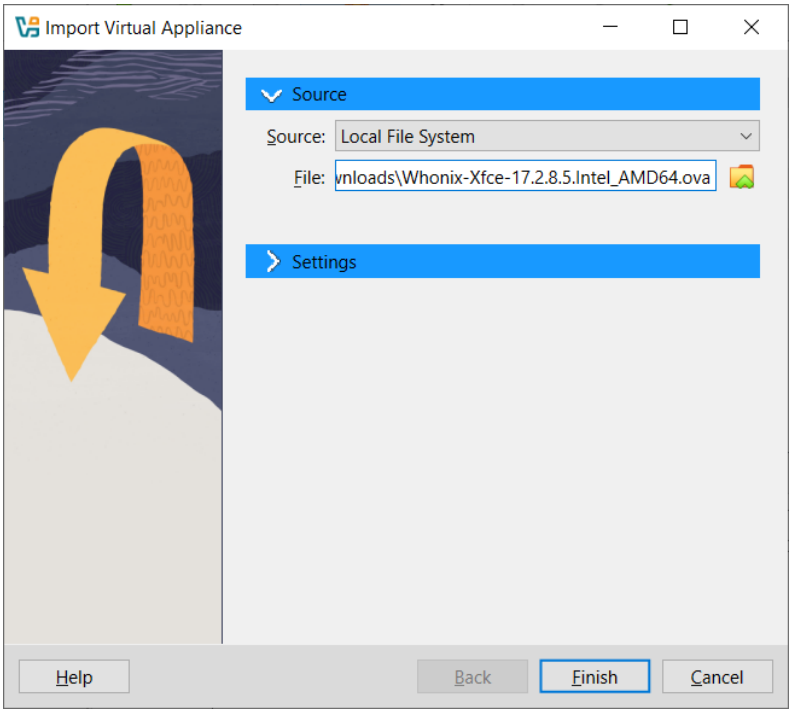


图 8-7 定位到之前下载的 Whonix 文件

点击“Settings”后如图 8-8 所示。

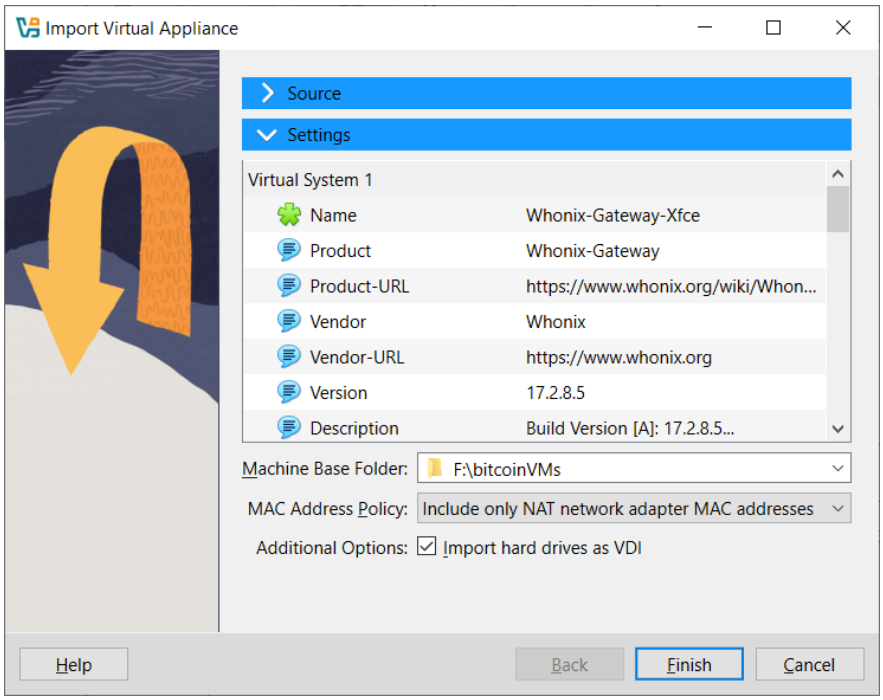


图 8-8 导入虚拟机总览

点击 Finish 按钮后同意 Tor 网关上的两个软件的协议，之后正式导入虚拟机，参见图 8-9。

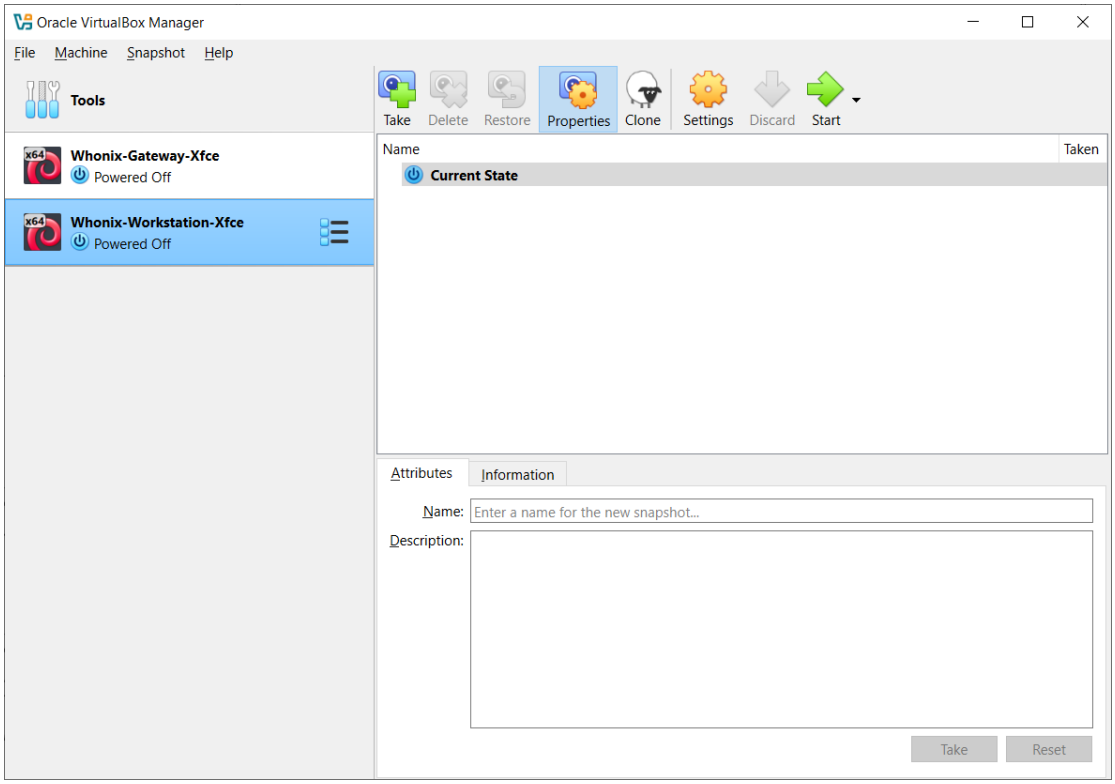


图 8-9 导入了两台虚拟机

其中虚拟机 Whonix-Gateway-XFCE 用作暗网网关，里面配置好了 Tor，平时不要在此虚拟机里操作，只要启动就行了。另一台虚拟机 Whonix-Workstation-XFCE 是工作机，它通过 Whonix-Gateway-XFCE 走暗网 Tor 到因特网，我们以后就在

此虚拟机中做在线的加密货币交易，比如发布已签名交易、查询钱包余额、与他人在线沟通等。

下面我们首先来初始化 Whonix-Gateway-XFCE 虚拟机。

8.1.3 初始化 Tor 网关

Tor 暗网用于隐匿自己的互联网出口 ip。互联网出口 ip 就是对方（一般是网站服务器）看到的你的 ip 地址。

关闭与宿主机的剪贴板共享和文件拖放功能【名字解释：宿主机是指安装和运行 VirtualBox 的电脑，我这里就是 Windows 10】。在图 8-9 左侧选中 Whonix-Gateway-XFCE 虚拟机，然后依次点击 Setting-->General-->Advanced: Shared Clipboard=disabled，Drag'n'Drop=disabled，见图 8-10 所示。

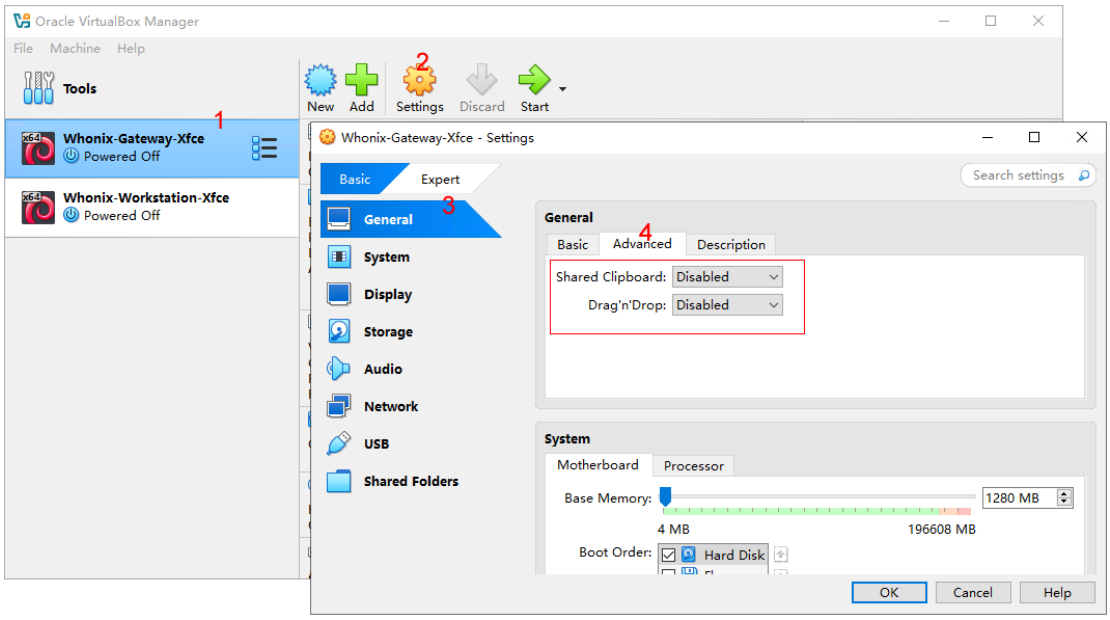


图 8-10 关闭内外剪贴板共享和拖放功能

参考图 8-11 所示禁用声卡。

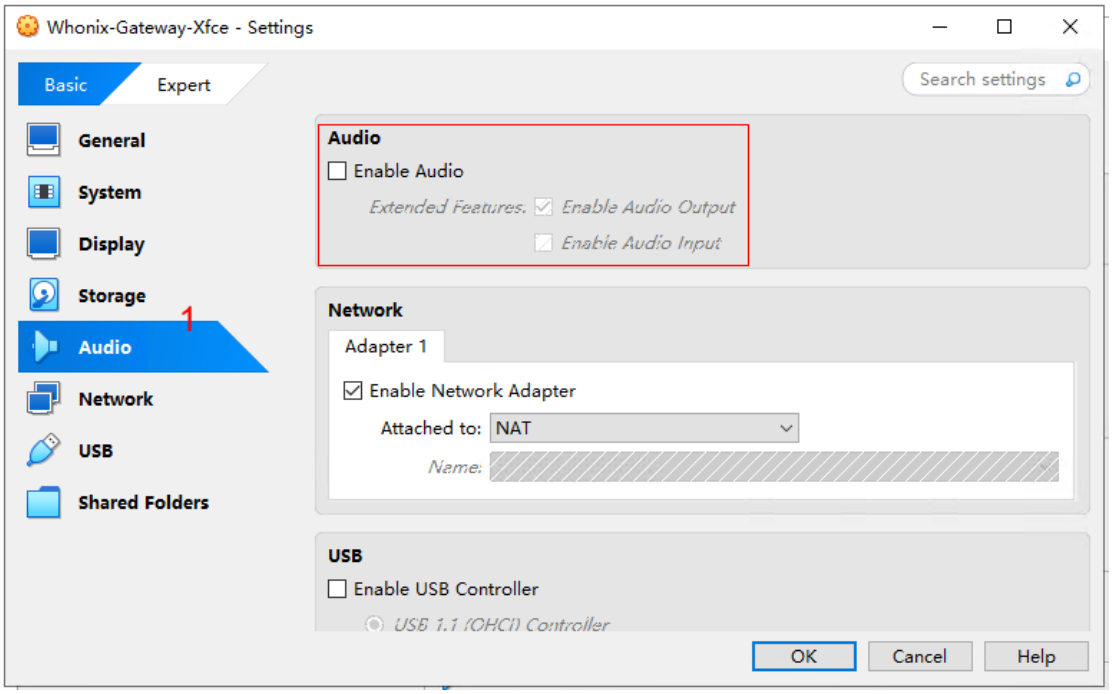


图 8-11 禁用声卡

点击 Start 图标启动 Whonix-Gateway-XFCE 虚拟机，并做初始化配置，见图 8-12。

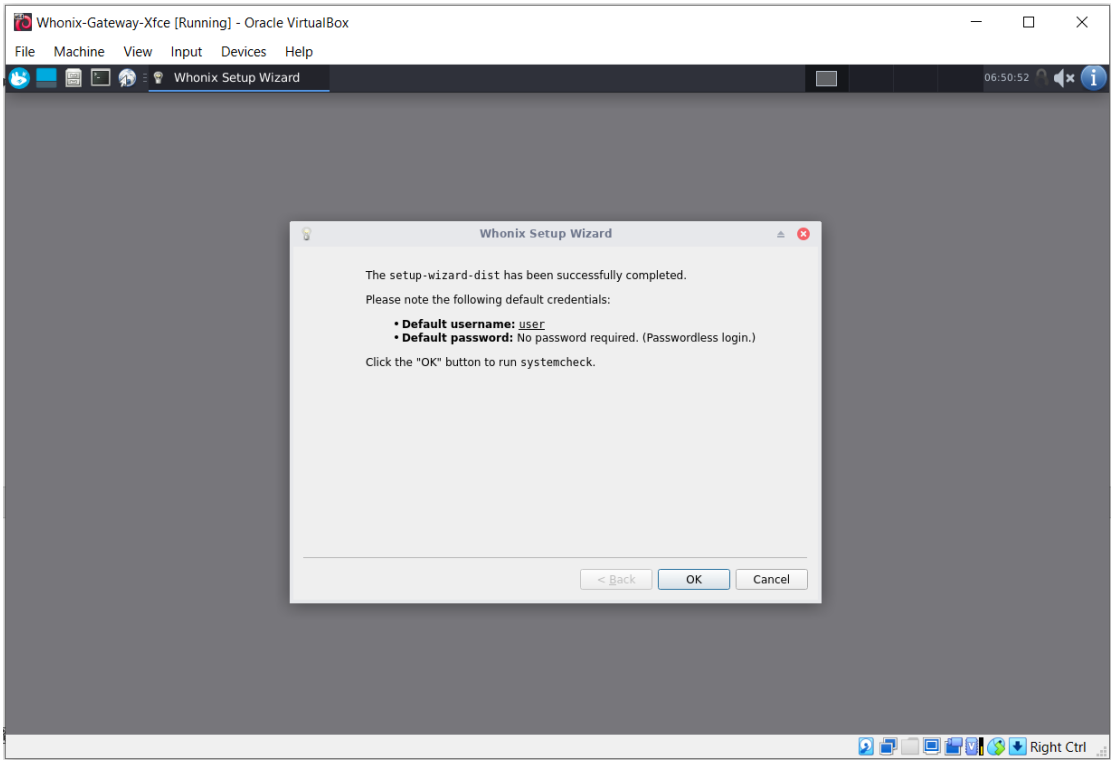


图 8-12 初始化配置 Tor 网关

登陆用户为 user，无密码。点击 OK 按钮后运行 systemcheck 工具，此工具首先开始测试 Tor 网，参考图 8-13 所示。



图 8-13 测试 Tor 网络

如果测试失败，即出现图 8-14 所示的错误提示。

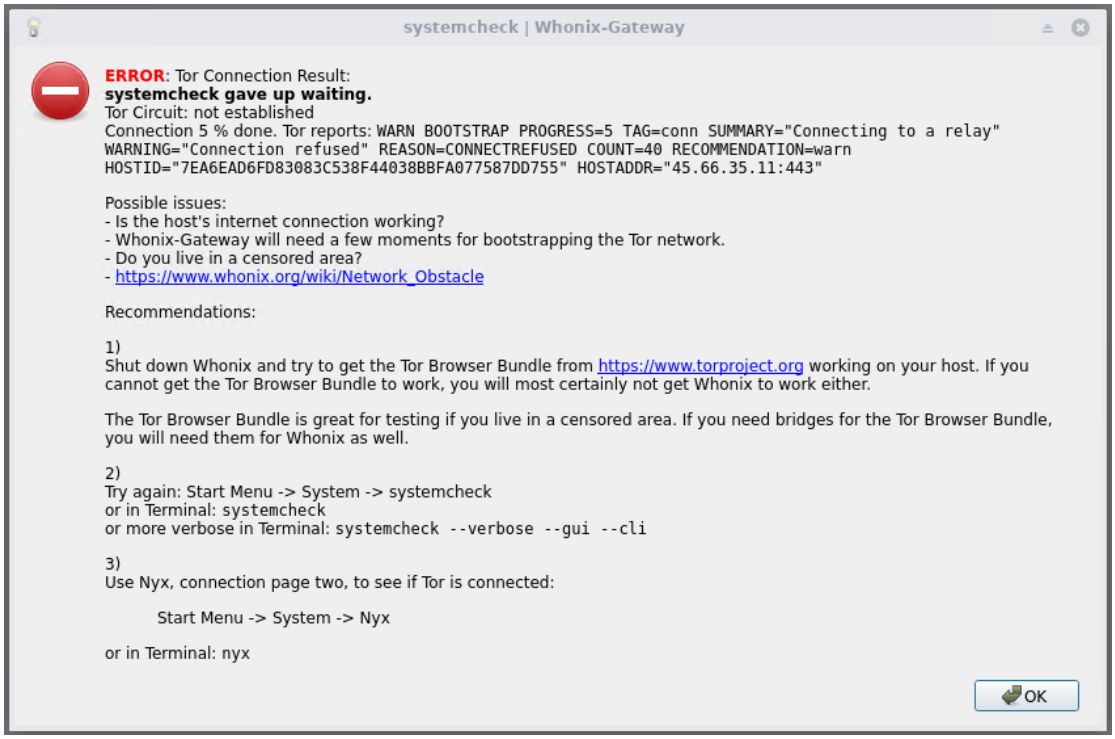


图 8-14 检测 Tor 网失败

在中国大陆失败是正常的，下面我们来配置 Tor 网，点击左上角的 Applications 图标，然后再依次点击 System、Anon Connection Wizard，之后会出现如图 8-15 所示的画面。

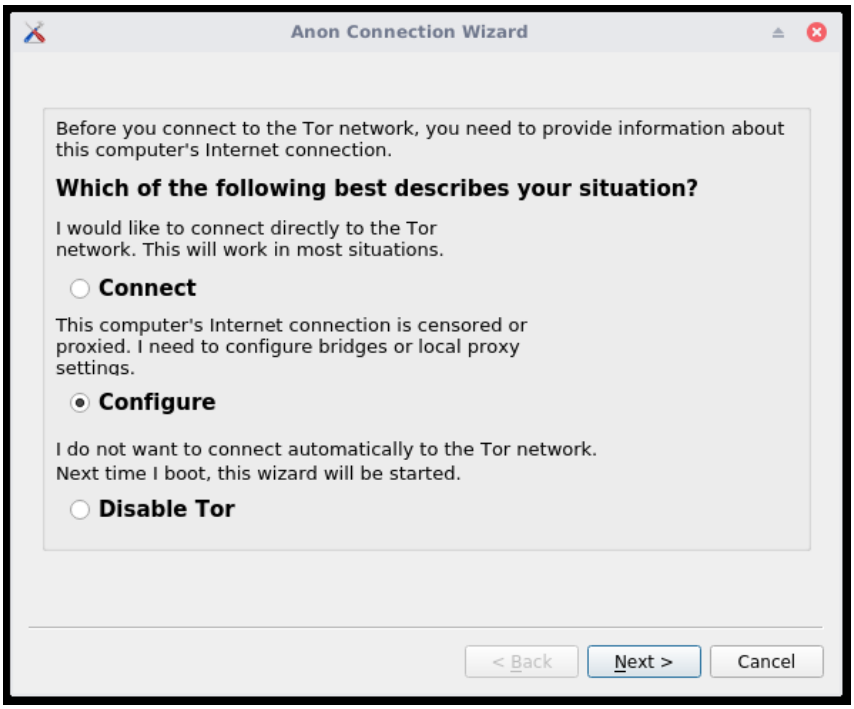


图 8-15 配置 Tor 网

在禁用 Tor 的国家（如中国），要点选“Configure”，然后点击“Next”后出现如图 8-16 所示的画面。

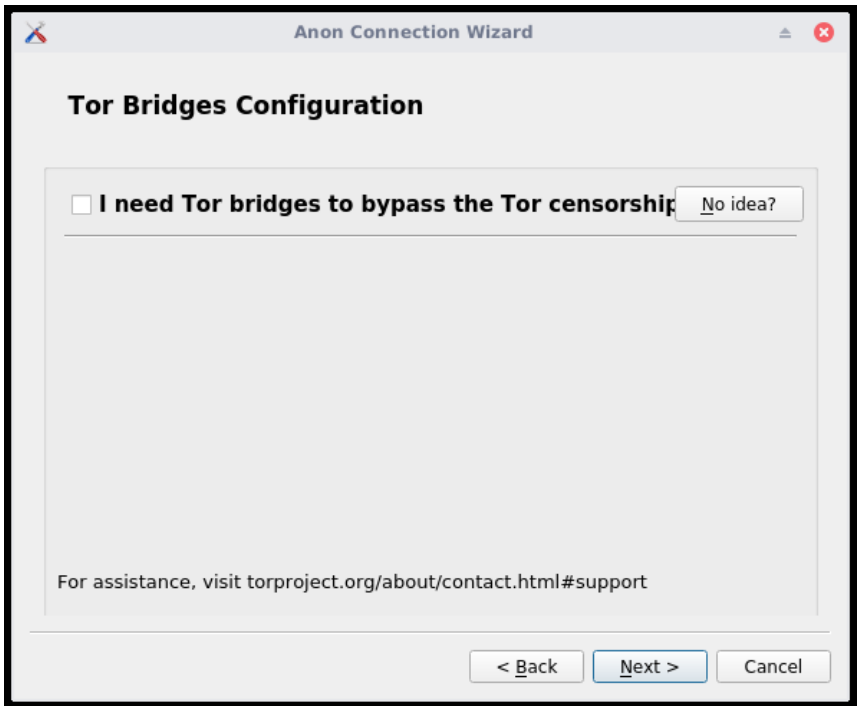


图 8-16 Tor 桥配置

在允许 Tor 的国家勾选“I need Tor bridges….”，但在中国不勾选。直接点击“Next”，之后如图 8-17 所示。

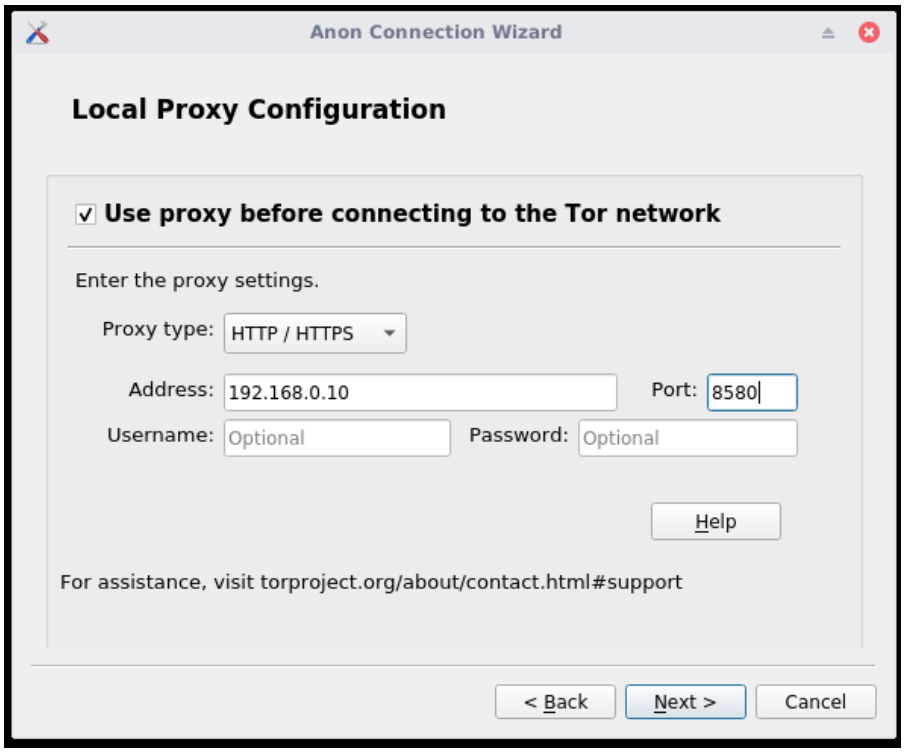


图 8-17 为 Tor 配置 VPN 代理

这里很关键，要勾选“Use proxy before connecting to the Tor network”，并且填入代理 vpn 的 ip 和端口。代理类型支持 http/https、SOCKS4 和 SOCKS5。

我在我的 windows 10 上安装了自游门（可从 http://dongtaiwang.com/下载，或从我的网站下载），并且配置成局域网代理具体安装和配置请参考“§2.2 科学上网”小节。把局域网内运行了自游门的电脑的 ip（比如我的 windows 10）填入上图中的 Address 后面，把自游门的代理端口 8580 填入上图中的 Port 后。

好了，最后显示如图 8-18 所示配置总览，如果无误就直接下一步了。

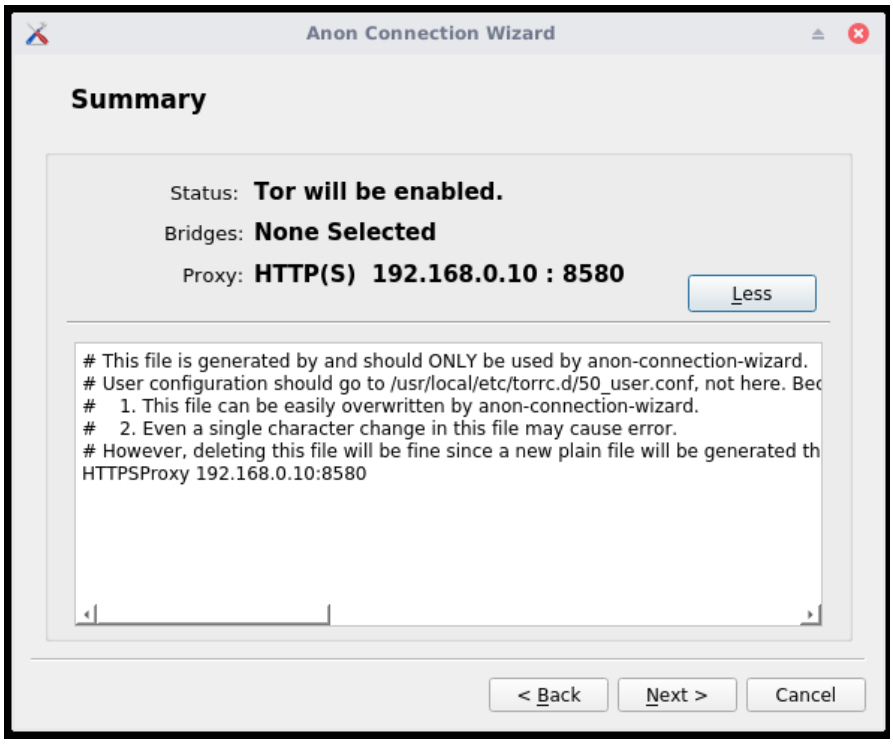


图 8-18 Tor 最终的配置总览

Tor 开始初始化，见图 8-19。

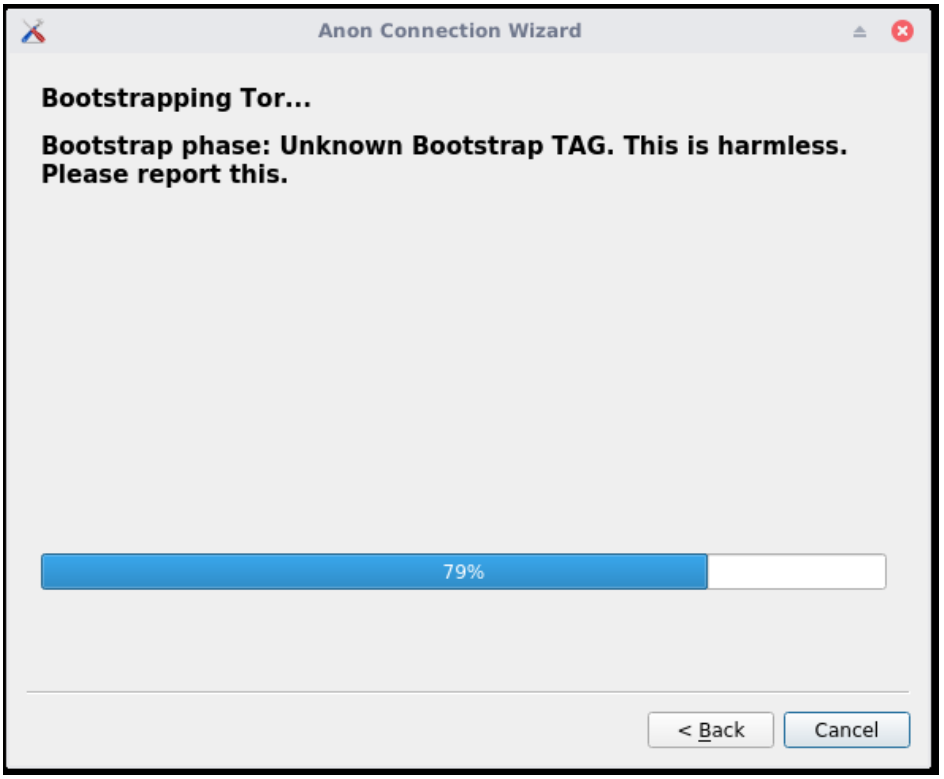


图 8-19 Tor 开始初始化

自由门不稳定，如果你有一台国外的云平台虚拟主机，且虚拟主机里运行了 ssh 服务，那么可以在你的宿主机上运行命

令“ssh -C -N -D 192.168.0.10:8085 user123@x.x.x.x”建立 SOCKS5 类型的 VPN，然后在图 8-17 中选择 SOCKS5 类型。命令中的 x.x.x.x 为你国外虚拟主机的 ip，user123 为国外虚拟主机上的一个用户。具体参考“§2.2 科学上网”小节。以后可随时打开 Tor 面板修改梯子参数（Applications→Sytem→Tor Control panel），参见图 8-20。

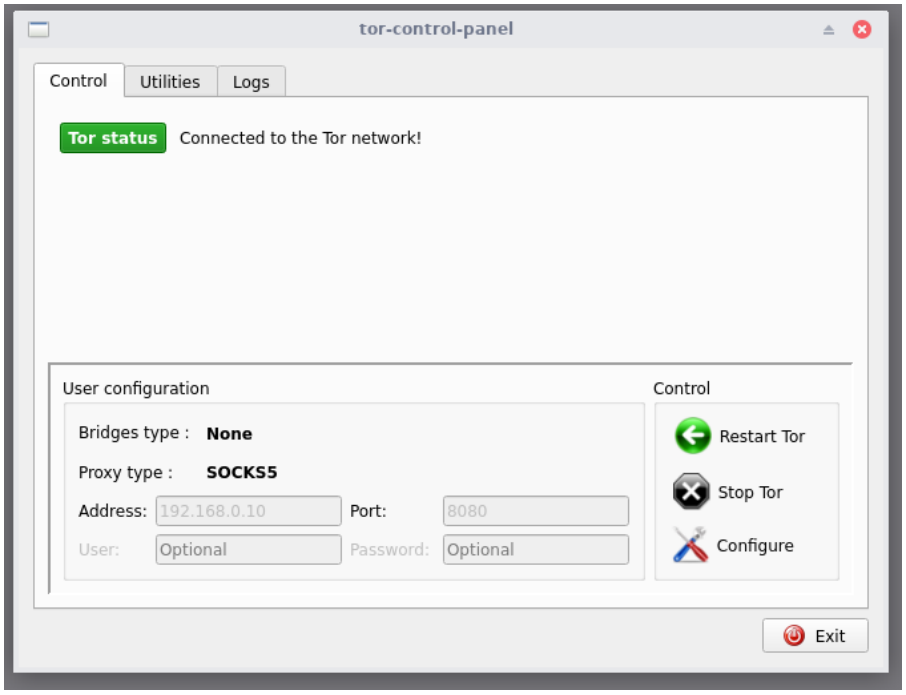


图 8-20 Tor 控制面板

在图 8-20 上的 Utilities 页上还可以浏览电路详情。

然后再次运行检查工具 systemcheck（参考图 8-21 操作）

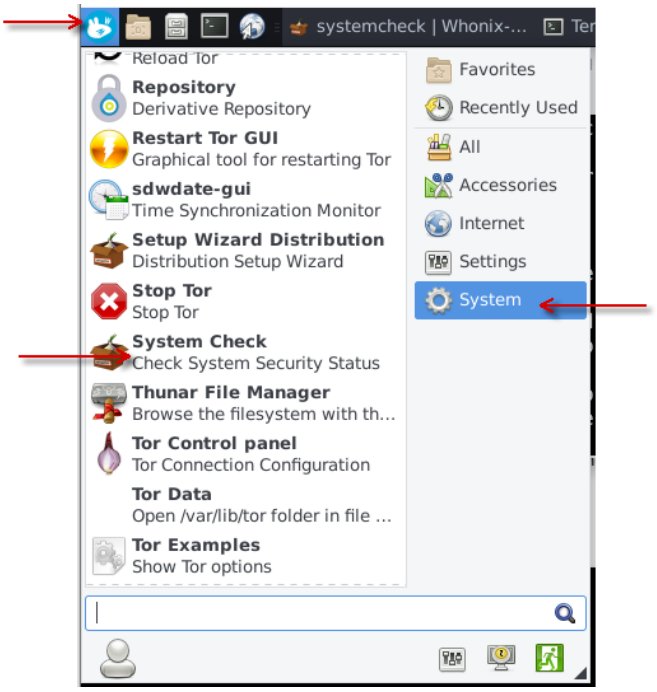


图 8-21 再次做系统检查

检查的结果参考图 8-22 所示。

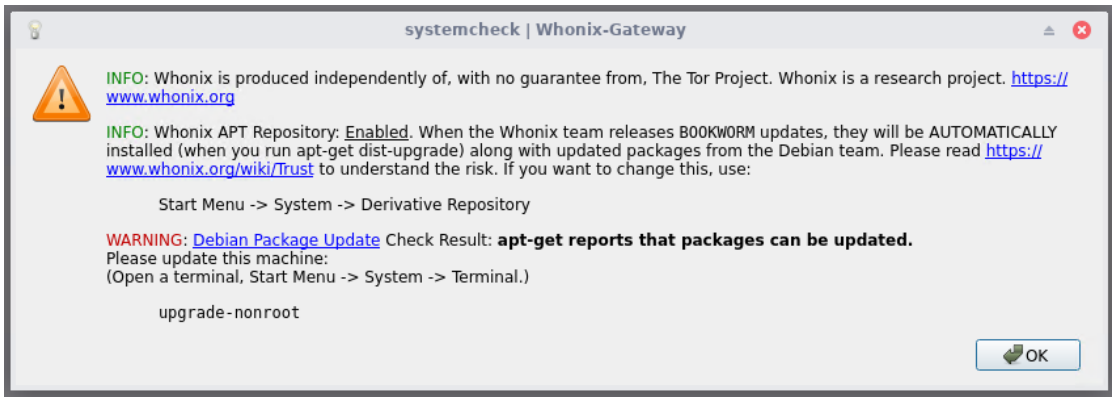


图 8-22 systemcheck 检测成功

初始化完毕后，我们首先修改 user 用户的密码，当前没有密码，新密码要长一点，尽量包含字母、数字和其他符号，参考图 8-23。

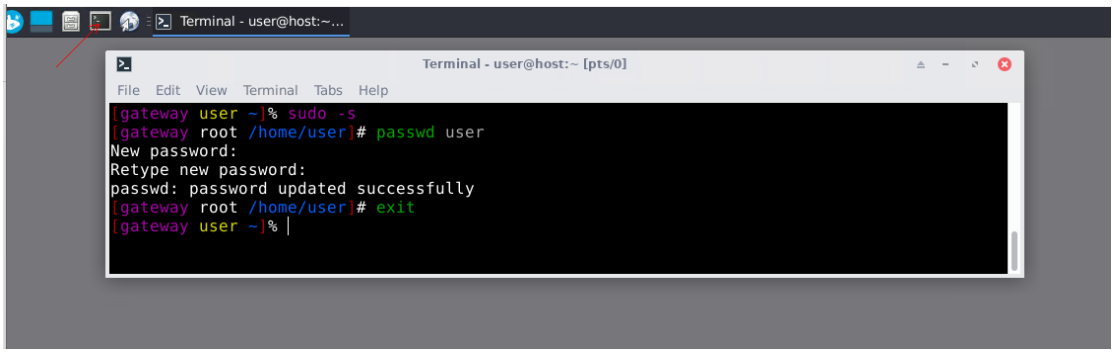


图 8-23 修改密码

同时打上最新的补丁，见图 8-24。

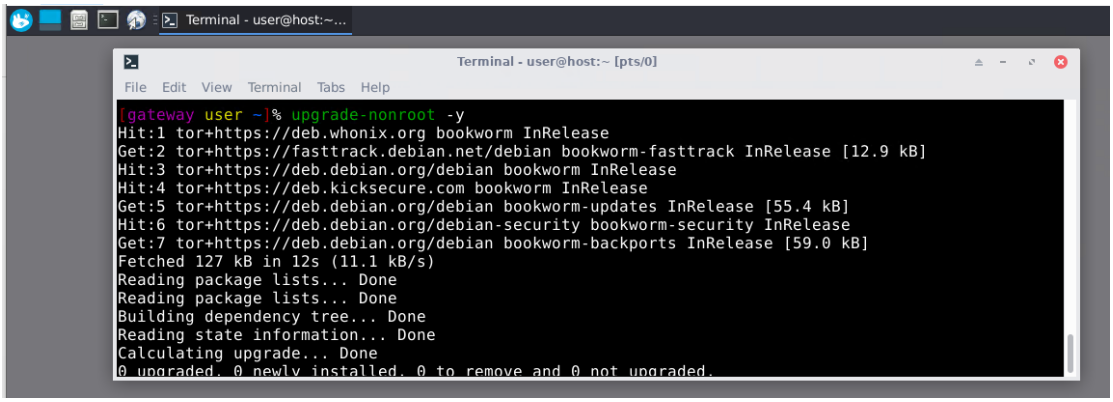


图 8-24 给系统打上最新的补丁

修改下次启动方式为硬盘只读方式，执行命令：`sudo sed -i 's|default="0"|default="1"|'`

/boot/grub/grub.cfg。这样相当于冻结了磁盘，以后就算你改动了，重启之后一切恢复原样。参考图 8-25。

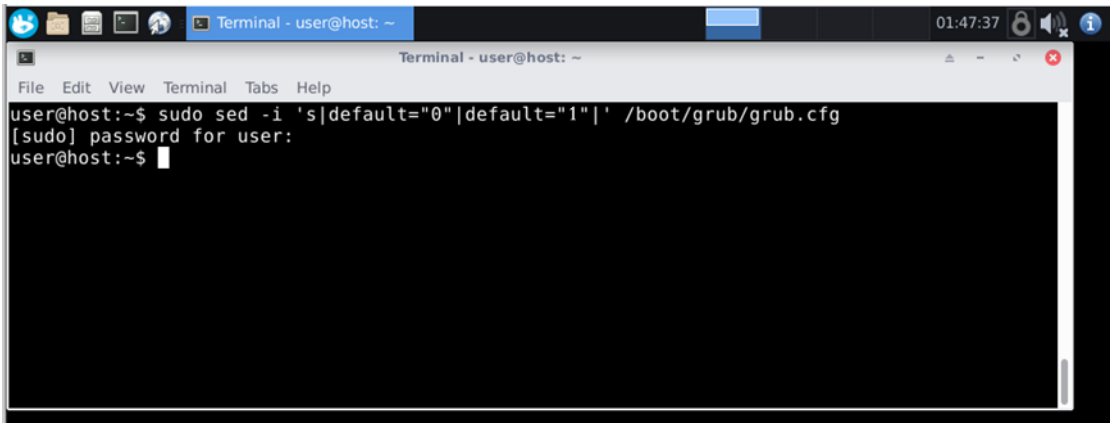


图 8-25 设置以只读方式启动

也就是在启动菜单上默认选择第二项。

禁用开机后自动登录，为了安全，只有知道密码的人才可以登录：

```
sudo mv -f /etc/lightdm/lightdm.conf.d/{40_autologin.conf,40_autologin.conf.old}
```

扩展 onion-grater 白名单，以便以后让在线工作站上的 Bisq 通过 Tor 网关：

```
sudo onion-grater-add 40_bisq
sudo onion-grater-list --used                                    //列举正在使用的 grater
/etc/onion-grater-merger.d/30_whonix-default.yml
/usr/local/etc/onion-grater-merger.d/40_bisq.yml
sudo onion-grater-list --available                               //有哪些可用的 gater
/usr/share/doc/onion-grater-merger/examples/40_bisq.yml
/usr/share/doc/onion-grater-merger/examples/40_bitcoind.yml
/usr/share/doc/onion-grater-merger/examples/40_cwtch.yml
/usr/share/doc/onion-grater-merger/examples/40_haveno.yml
/usr/share/doc/onion-grater-merger/examples/40_lnd.yml
/usr/share/doc/onion-grater-merger/examples/40_onion_authentication.yml
/usr/share/doc/onion-grater-merger/examples/40_onionshare.yml
/usr/share/doc/onion-grater-merger/examples/40_ricochet.yml
/usr/share/doc/onion-grater-merger/examples/40_wahay.yml
/usr/share/doc/onion-grater-merger/examples/40_zeronet.yml
```

最后参考图 8-26 所示关闭虚拟机。

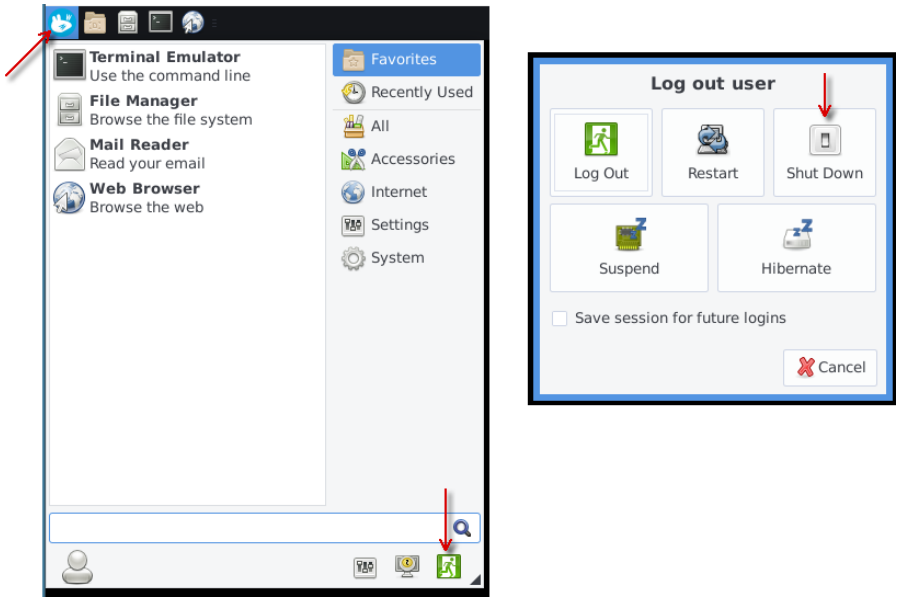
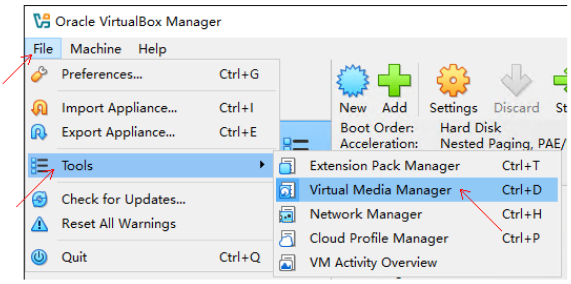


图 8-26 关闭虚拟机

设置磁盘为只读（按 Ctrl+d，或点击菜单 File->Tools->Virtual Media Manager），参考图 8-27。



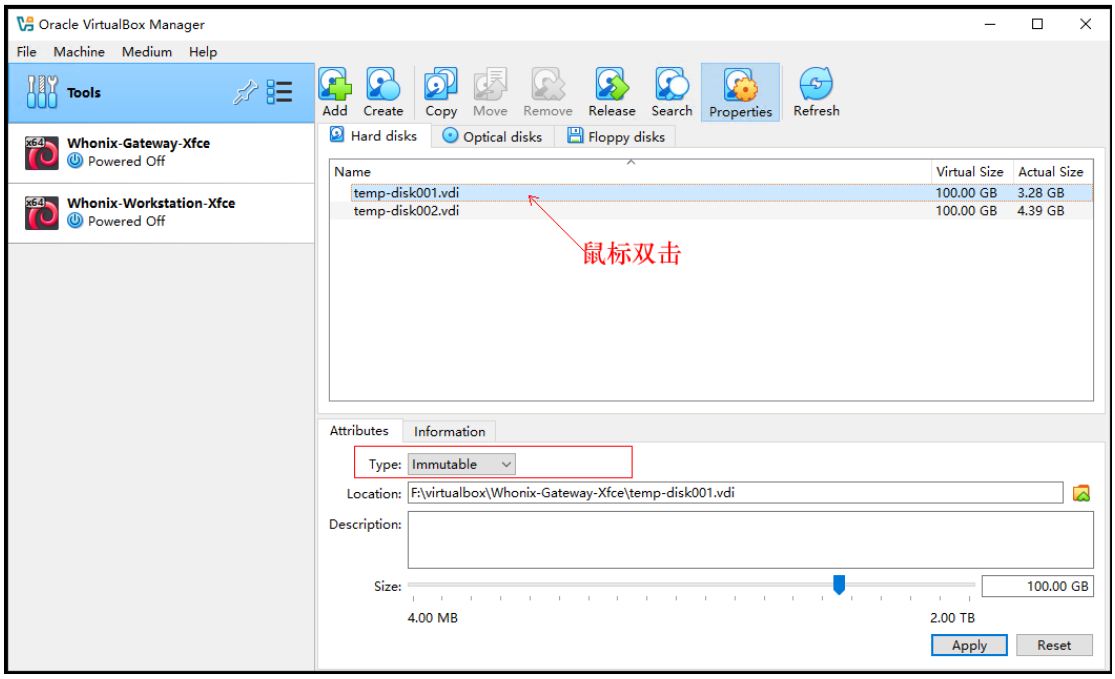


图 8-27 设置只读方式

Whonix-Gateway-XFCE 是 Whonix-Workstation-XFCE 等其他虚拟机的网关，所以平时本机都要启动。改名为 Gateway，以增强隐蔽性，参考图 8-28。

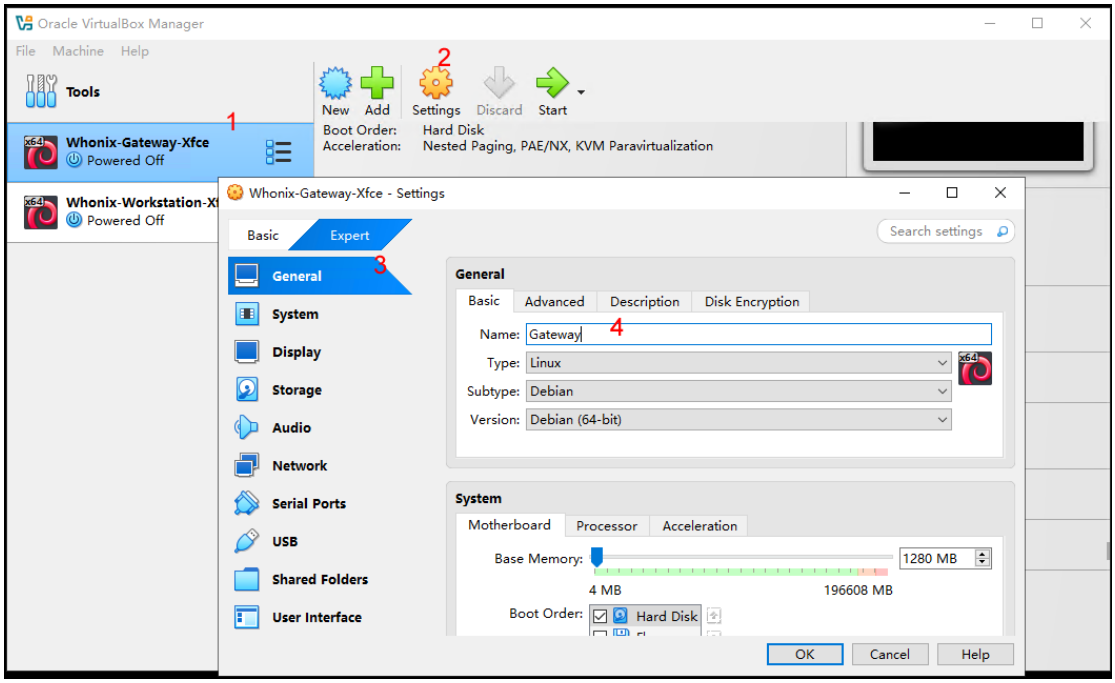


图 8-28 虚拟机改名

然后启动 Gateway，图 8-29 为启动后的登录界面。

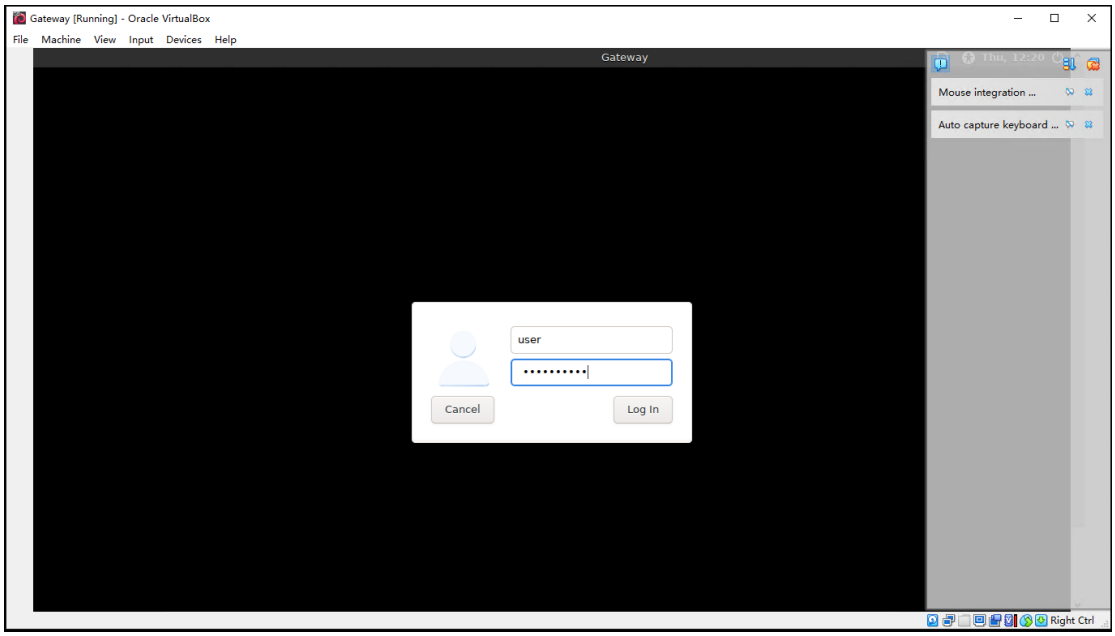


图 8-29 登录界面

登录后看看到右上角有 live 字样，鼠标悬停上面会弹出一个通知，告诉我们磁盘是只读模式，此后对硬盘内容的改动，

虚拟机重启后都将作废。如图 8-30 所示。

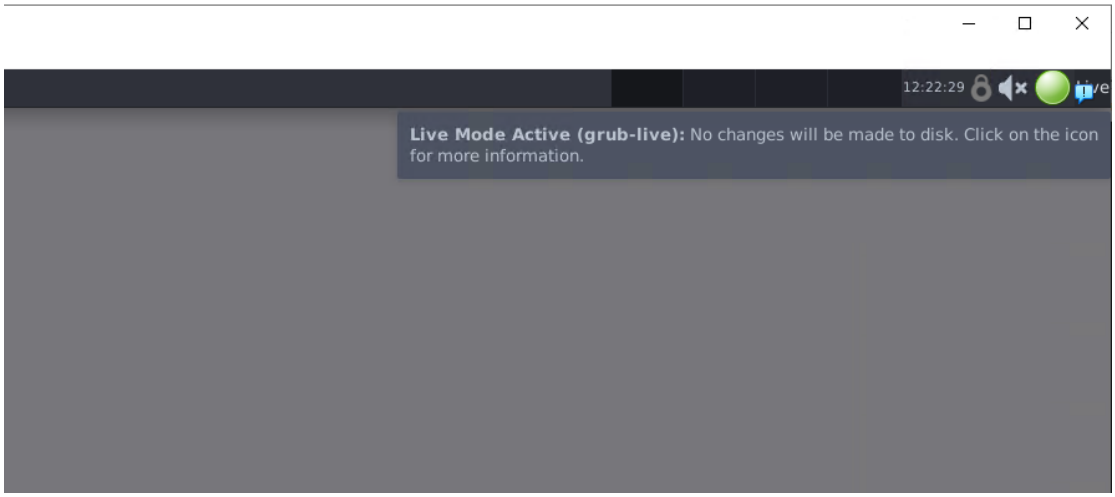


图 8-30 登录后的样子

8.1.4 准备在线工作站

专门通过 Tor 网关中的暗网访问互联网，从而隐匿自己的互联网出口 ip。

8.1.4.1 初始化

启动 Whonix-Workstation-XFCE 虚拟机，见图 8-31。

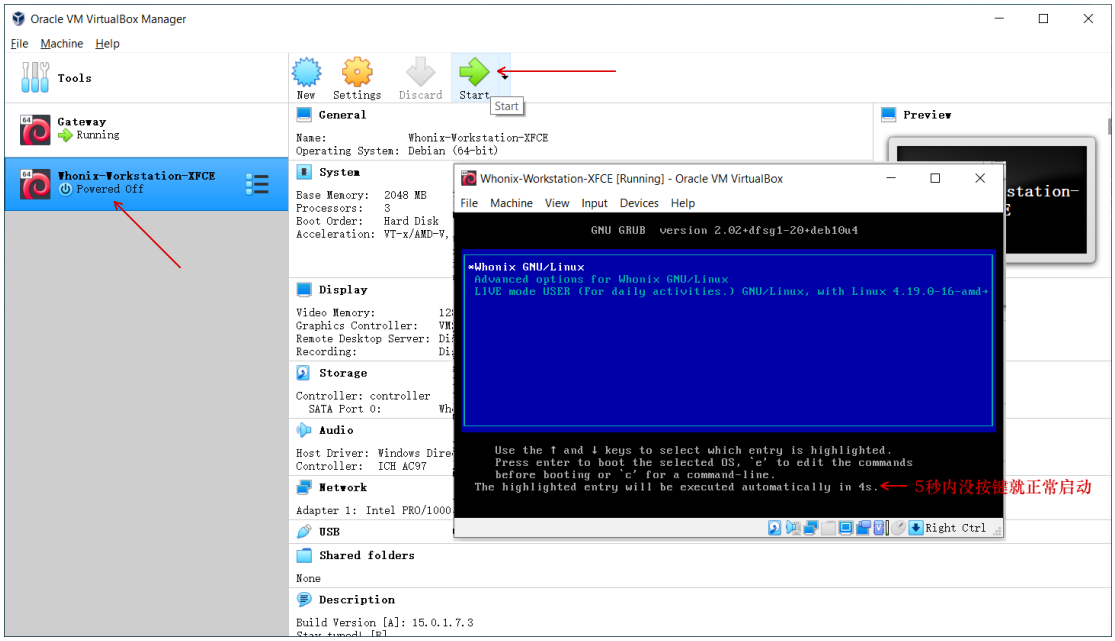


图 8-31 启动工作站

首先就是检查系统，类似图 8-12 所示的画面，点击 ok 按钮后出现如图 8-32 所示的画面。

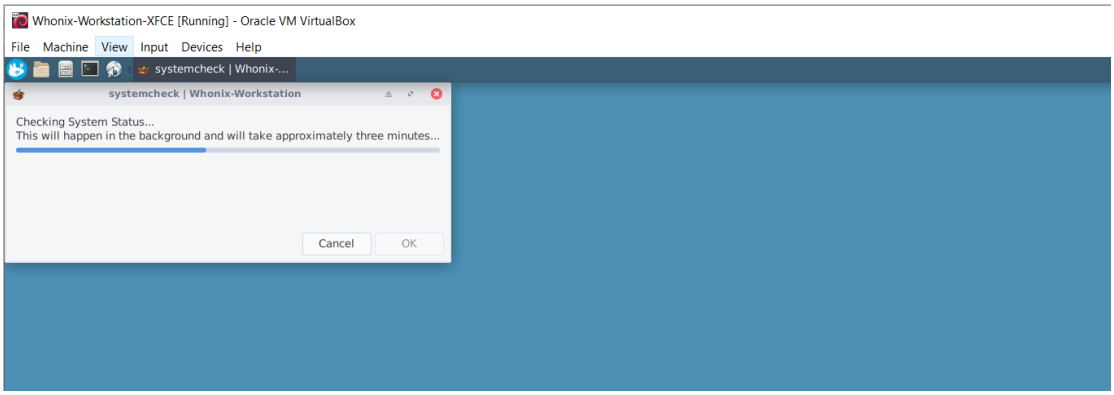


图 8-32 初始化工作站

检查后如果出现如图 8-33 所示的结果，表明一切 OK 啦。

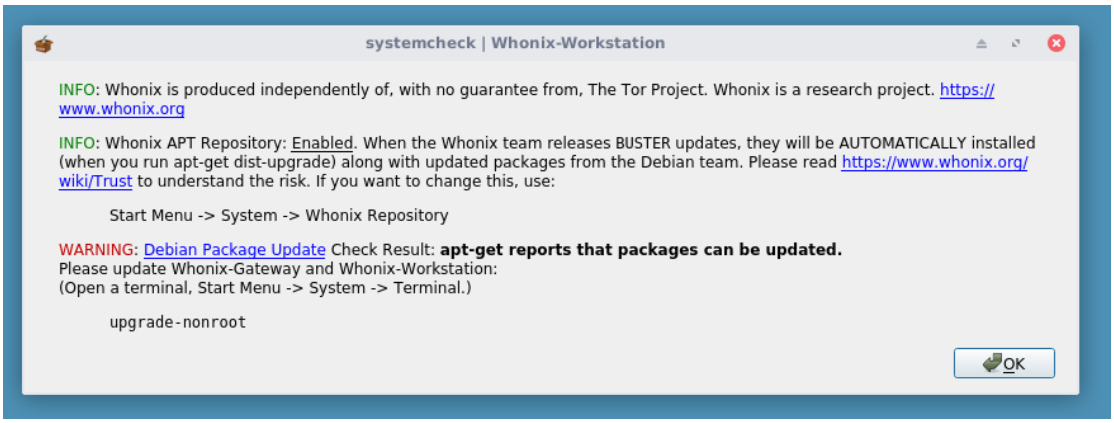


图 8-33 初始化成功

检查浏览器是否走 Tor 网络，参见图 8-34。



图 8-34 检查浏览器是否走 Tor 网

如果看到如图 8-35 所示的样子，就一切 ok 啦。可进一步在 https://www.findip-address.com/ 上查询地址 104.244.73.136，发现位于欧洲的卢森堡，说明我的暗网从卢森堡的某台电脑上出口到互联网的——如果警察逆向追踪我，就追到卢森堡的一台电脑，此后就追不下去了，所以警察会认为我就是卢森堡的一个市民。另外我的出口地址时经常变的，明天可能又是美国纽约的 ip。

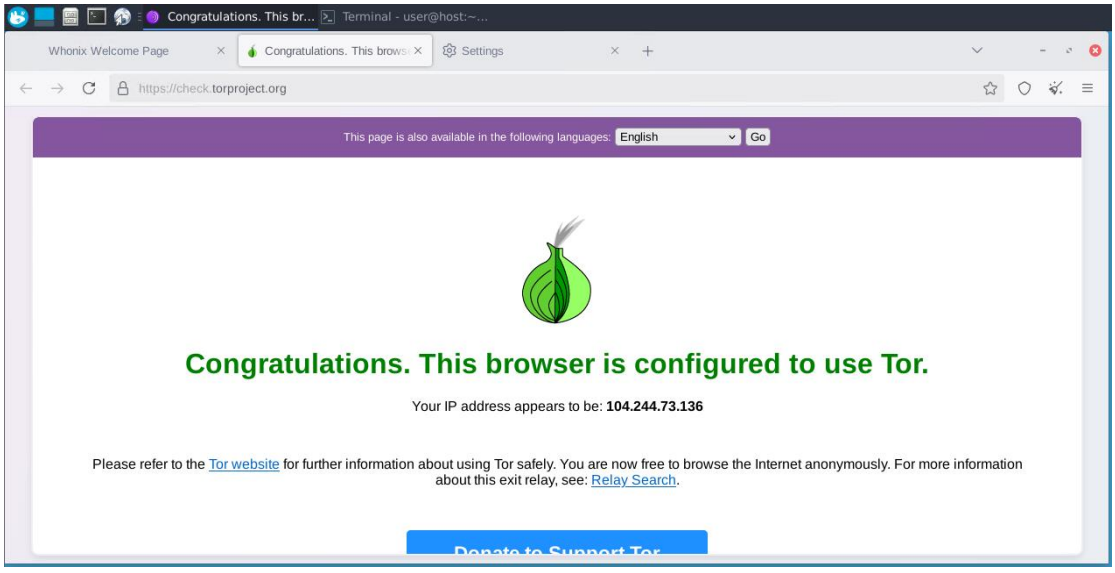


图 8-35 浏览器走了 Tor 网

再浏览网站 https://dnsleaktest.com/检查 DNS 是否被泄露，如果看到的是中国大陆之外的 ip，那说明没有泄露，参见图 8-36 所示为德国的勃兰登堡。

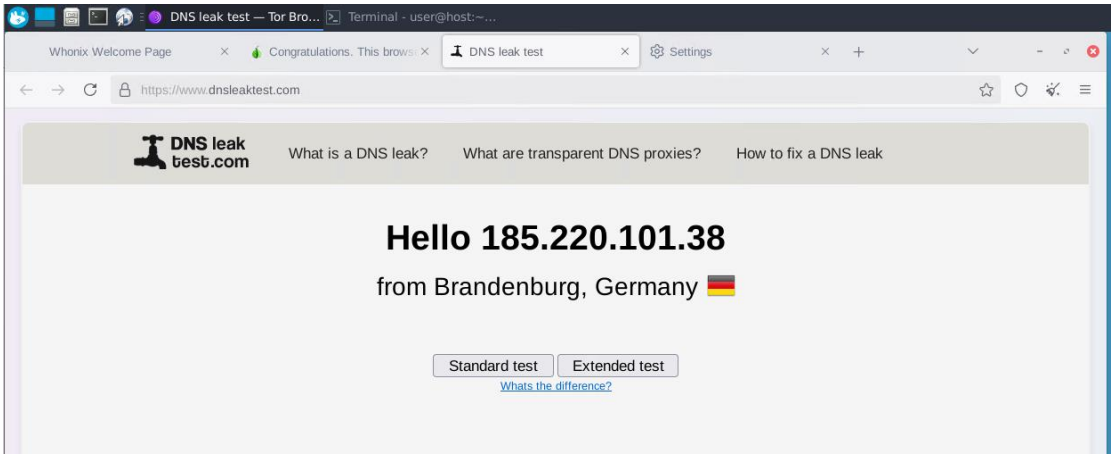


图 8-36 检查 DNS 是否泄露

我们平时上网，比如浏览京东的网站 www.jd.com，电脑首先要询问 DNS 服务器 www.jd.com 对应的 ip，然后获得 ip 之后才去正式打开京东网站页面。如果你电脑配置的 DNS 服务器被坏人掌握，那么坏人就知道你上了什么网站，或者他干脆把钓鱼网站的 ip 反馈给你，或者不给你反馈 ip，这称为域名污染。

记得哦，从此要养成一个好习惯，少用 chrome 浏览器和谷歌搜索，要习惯使用洋葱浏览器 Tor-browser 和 duckduckgo.com 搜索。什么？百度能不能用？如果你不怕死就用吧，魏则西的教训永远不要忘记。参考图 8-37。

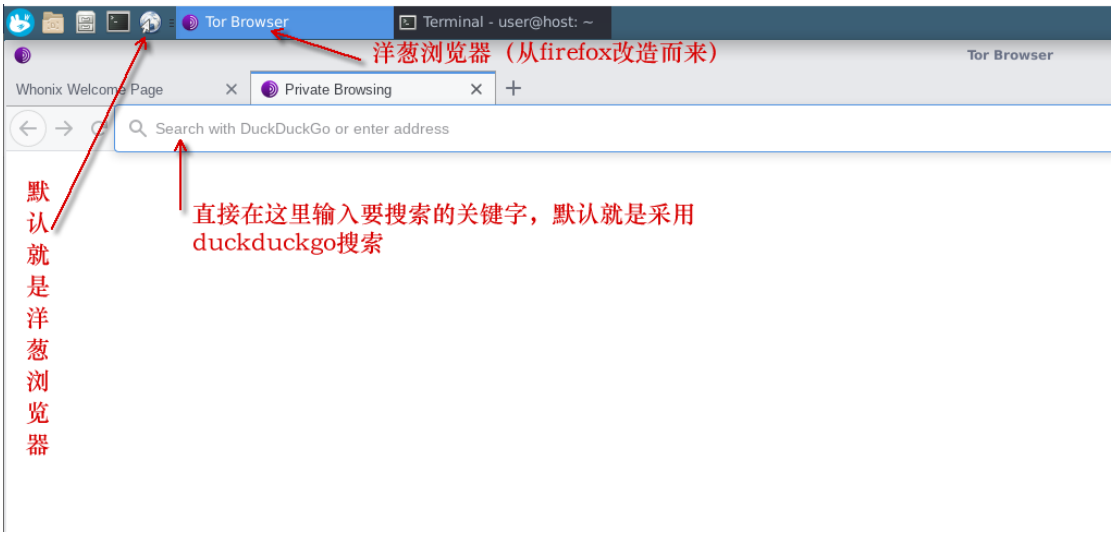


图 8-37 洋葱浏览器集成了 duckduckgo 搜索

修改用户 user 的密码（参考 § 7.1 小节中的方法）。卸载多余的软件并给操作系统打上最新的补丁，最后重启：

```
sudo apt purge ristretto vlc  -y
sudo apt autoremove -y
sudo upgrade-nonroot -y
sudo shutdown -r now
```

8.1.4.2 汉化

点击靠近左上角“  ”图标打开终端（见图 8-38），命令都在这个终端中输入和执行的。



图 8-38 命令终端

作为中国人，还是习惯看中文和输入中文，我们接下来对工作机进行汉化，执行命令：

```
sudo dpkg-reconfigure locales
```

按 End 键滚到靠近底部，按空格键选择 zh_CN.UTF-8，编辑文件/etc/default/locale:

```
sudo mousepad /etc/default/locale
```

确保里面有下面的两行内容：

```
LANG=zh_CN.UTF-8
LANGUAGE=zh_CN:zh
```

安装中文字体：

```
sudo apt install fonts-wqy-zenhei fonts-wqy-microhei -y
```

安装输入法，这里用的是 fcitx5 输入法：

```
~~sudo apt install fcitx fcitx-libpinyin -y~~
sudo apt install fcitx5 fcitx5-chinese-addons -y
```

然后重启虚拟机：

```
sudo shutdown -r now
```

之后启动 Mousepad 记事本，然后按 Ctrl+空格键在中英文输入法之间切换，请参考图 8-39。

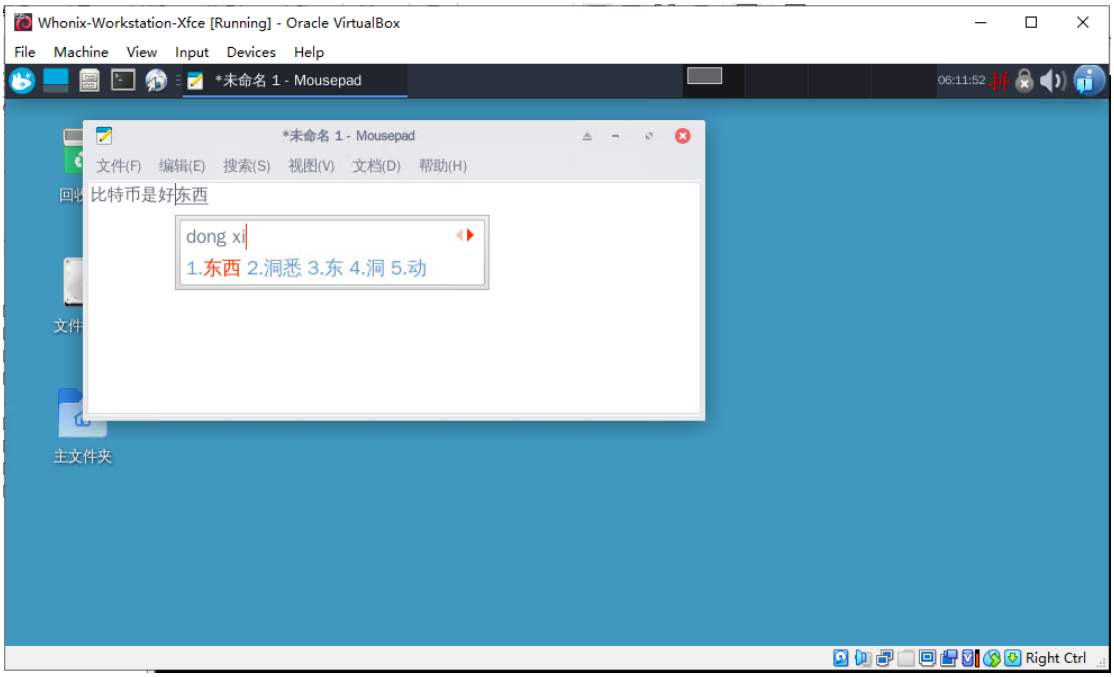


图 8-39 可以输入中文

8.1.4.3 准备相关文件

参考“§7.2 准备相关文件”小节操作。

然后关闭虚拟机，复制一台新的虚拟机（以后作为离线签名机器），见图 8-40。



图 8-40 复制虚拟机

接下来的两个界面全部采用默认值即可。再次启动 Whonix-Workstation-XFCE，设置下次开机自动进入磁盘只读模式：

```
sudo  sed  -i  's|default="0"|default="1"|'  /boot/grub/grub.cfg
```

但不在 virtualbox 中设置虚拟机硬盘只读，这样方便我们切换到磁盘写入模式。禁用开机后自动登录：

```
sudo mv /etc/lightdm/lightdm.conf.d/{40_autologin.conf,40_autologin.conf.old}
```

8.1.4.5 安装 Bisq

通过 https://bisq.network/交易真正实现了比特币的理想：去中心化、匿名和透明。但是必须在本地电脑上安装一个软件，而不能直接在网站上交易，这样意味着私钥只掌握你自己手里(此软件要在线使用，这就不能绝对保证私钥的安全性，所以建议在 bisq 中购买的加密币还是要通过手工方法转移到自己的伪纸质钱包中)。

本虚拟机的内存最少 6GB。开放防火墙：Applications → System → User Firewall Settings，在打开的文件中增加一行 EXTERNAL_OPEN_ALL=true，存盘退出，然后重载防火墙策略：Applications → System → Reload Firewall。

接下来的安装过程参考“§7.4 安装 Bisq”小节，只不过不要单独再安装 Tor。

参考 https://www.whonix.org/wiki/Bisq。

8.1.4.6 安装 jami

Jami 是一个去中心化的开源的点对点即时通信工具，可以用文字、语音和视频沟通，通信内容只有天知地知你知我知（我和你是通信的双方），放心，绝对没有老大哥盯着。

安装过程请参考“§7.5 安装 jami”小节。

8.1.4.7 扫尾工作

关闭 Whonix-Workstation-XFCE 后，把虚拟机的名字改名为 Online，禁用内外共享剪贴板和文件拖放功能。参考图 8-41。

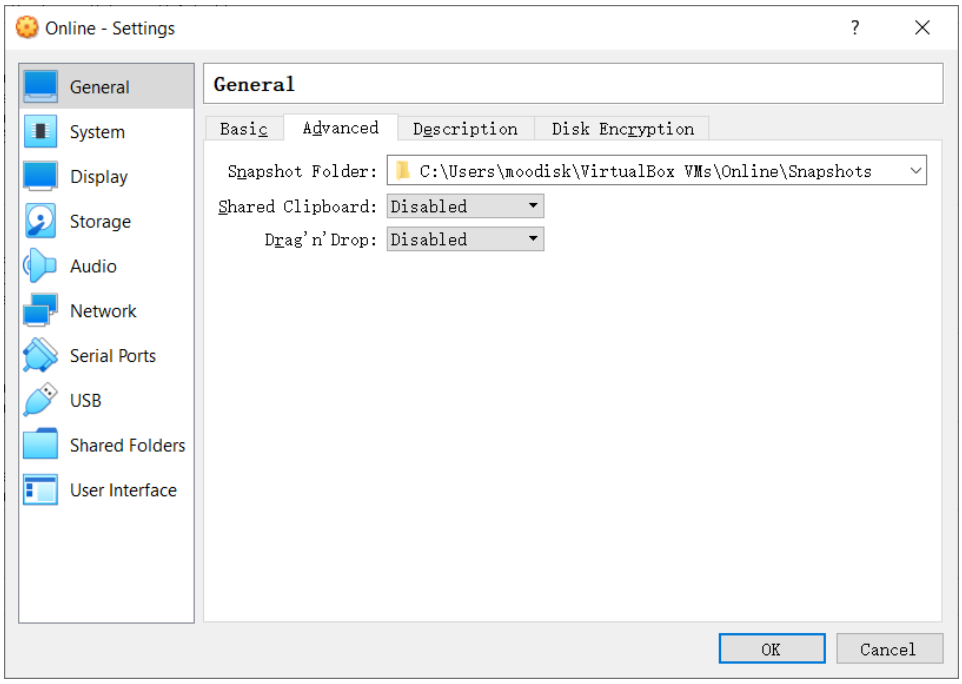


图 8-41 禁用内外共享剪贴板和文件拖放功能

以后启动 Online 虚拟机就会自动以自读模式启动了，参考图 8-42 所示。

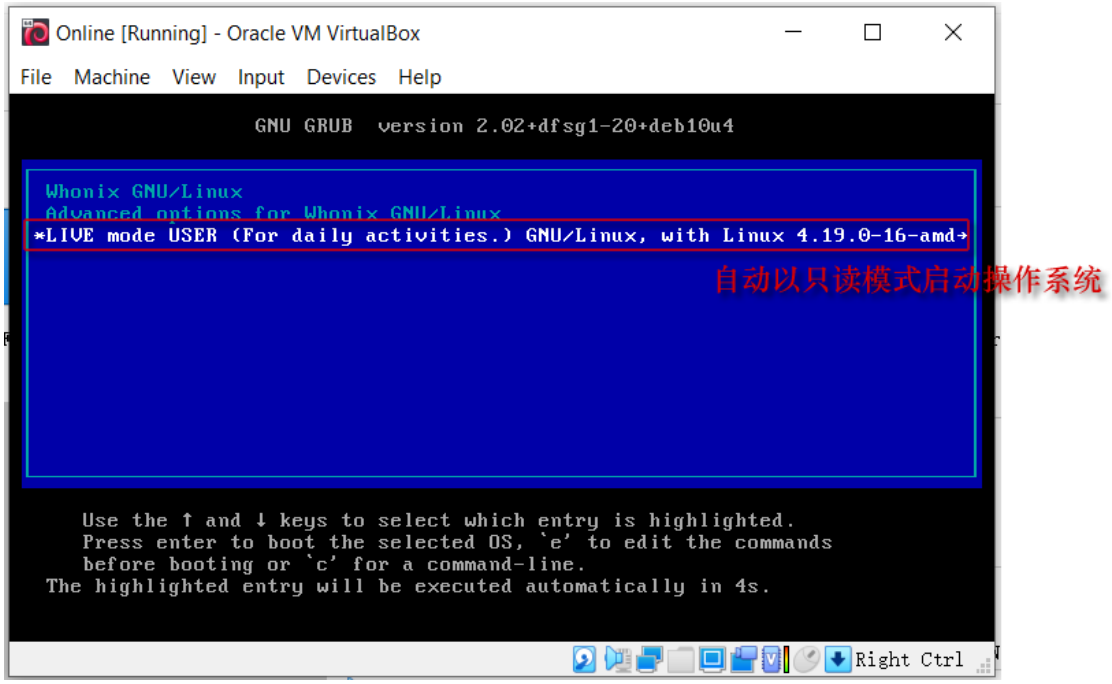


图 8-42 默认以只读方式启动

给虚拟机磁盘加密，这样就算别人把你的虚拟机拷贝走了，他也是没法启动虚拟机，也无法直接查看虚拟机磁盘中的内容。关闭虚拟机，进入设置界面，然后参考图 8-43 操作。

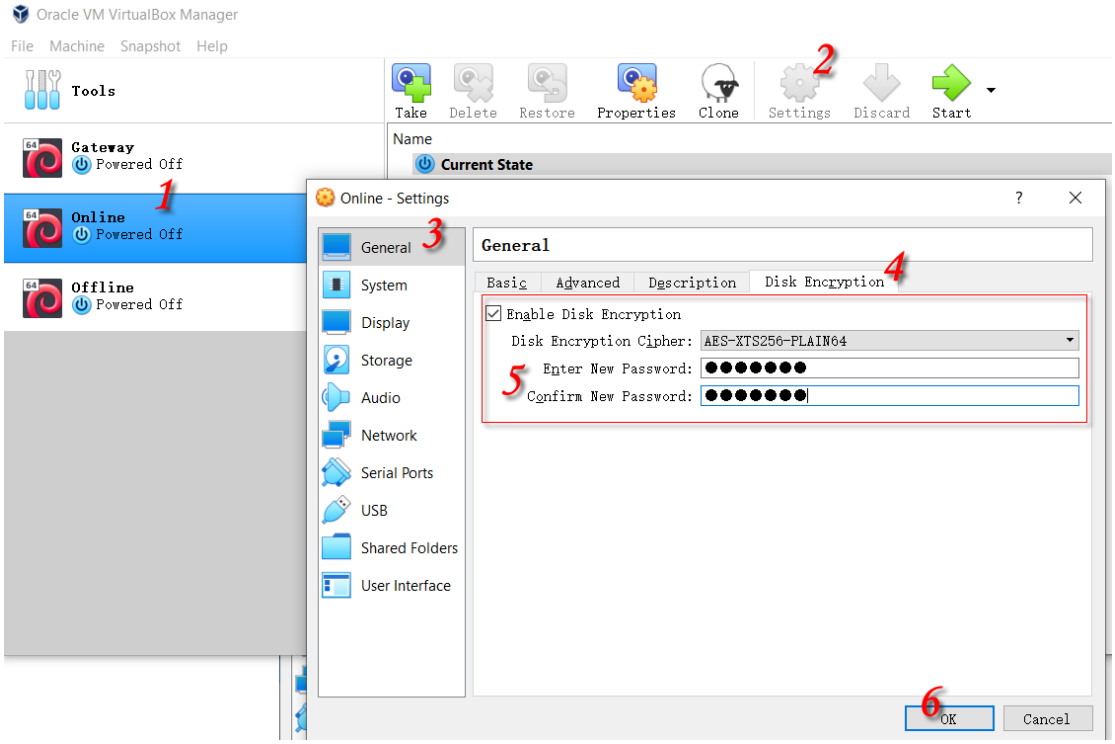


图 8-43 给虚拟机设置开机密码

密码要尽量长而且复杂。这样每次启动虚拟机时就要输入密码，参考图 8-44。

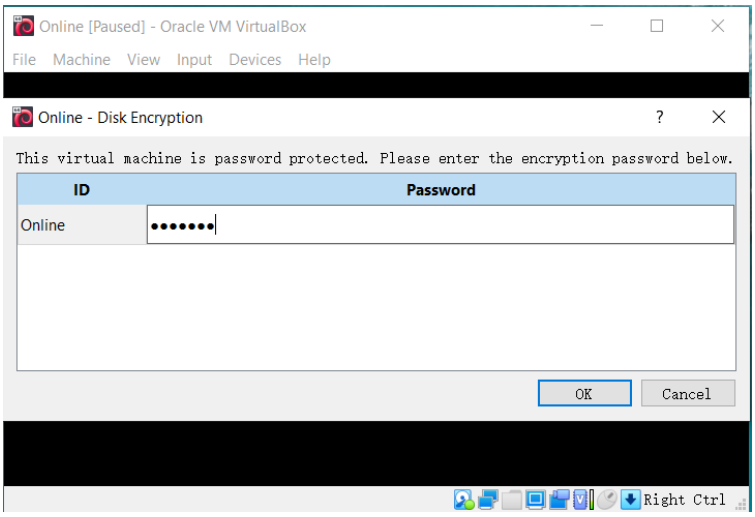


图 8-44 开机时提示输入密码

导出虚拟机也是要输了密码的。

8.1.5 配置离线签名机

把克隆出来的虚拟机改名为 Offline，并启动它。从这里 https://download-installer.cdn.mozilla.net/pub/firefox/releases/137.0.2/linux-x86_64/zh-CN/firefox-137.0.2.tar.xz 下载 firefox 浏览器，然后下载的文件 firefox-137.0.2.tar.xz 拖到 Offline 主机的桌面上，然后再剪切到主文件夹中的 softwares 目录中，解压后运行里面 firfox 启动浏览器。对浏览器做如下设置：一律使用隐私浏览模式，参考图 8-45。



图 8-45 设置隐私浏览模式

然后关机。

禁用网卡、剪贴板、声卡、USB 和文件拖放功能，参考图 8-46 和图 8-47。

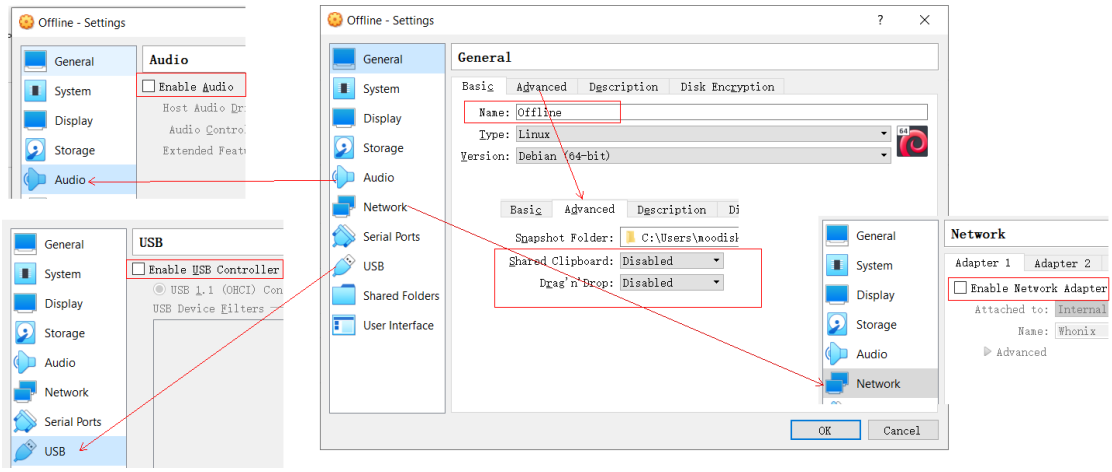


图 8-46 禁用不需要的功能

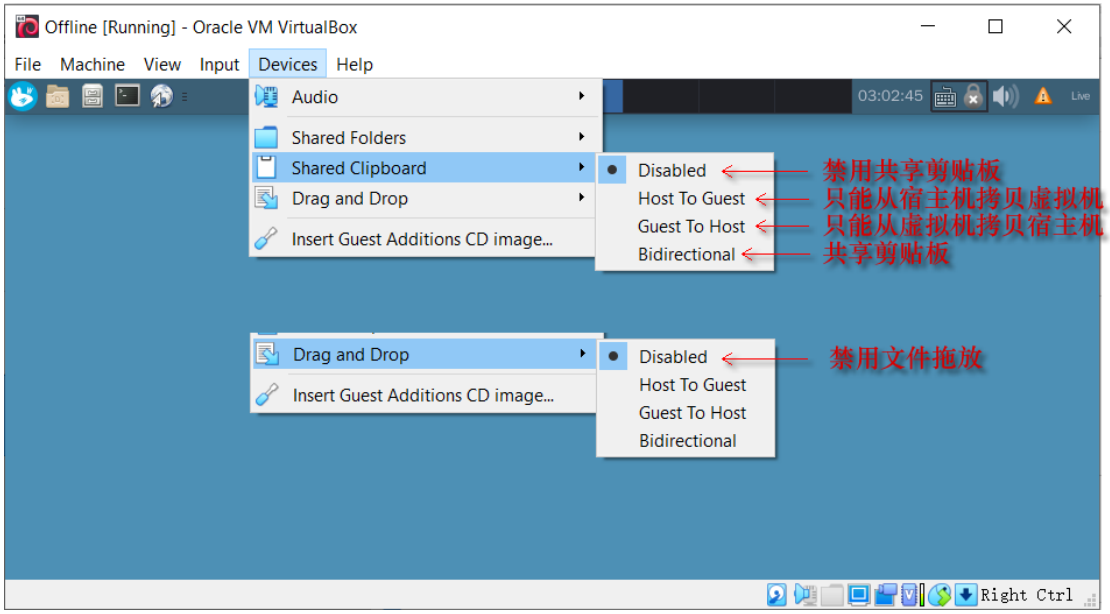


图 8-47 操纵剪贴板和文件拖放

最后参考 Gateway 虚拟机的方法设置磁盘为只读模式即可。禁用开机后自动登录：

```
sudo mv /etc/lightdm/lightdm.conf.d/{40_autologin.conf, 40_autologin.conf.old}
```

设置主机名为 Offline：

```
sudo hostnamectl set-hostname Offline
sudo echo '127.0.0.1 Offline' >> /etc/hosts
```

关闭虚拟机。设置 Offline 的磁盘为只读模式。

把你的已用密码保护的私钥等文件拷贝过来。

让下次开机后成为只读模式：

```
sudo  sed  -i  's|default="0"|default="1"|'  /boot/grub/grub.cfg
```

当签名完毕后，可以临时开启 Guest To Host 的剪贴板共享模式，但这时很危险，千万不要拷贝私钥，只拷贝原始的交易签名即可。一旦交易签名拷贝完毕，就立即禁用剪贴板共享。

关闭虚拟机，然后对虚拟机的硬盘进行加密，具体参考图 8-48 所示。

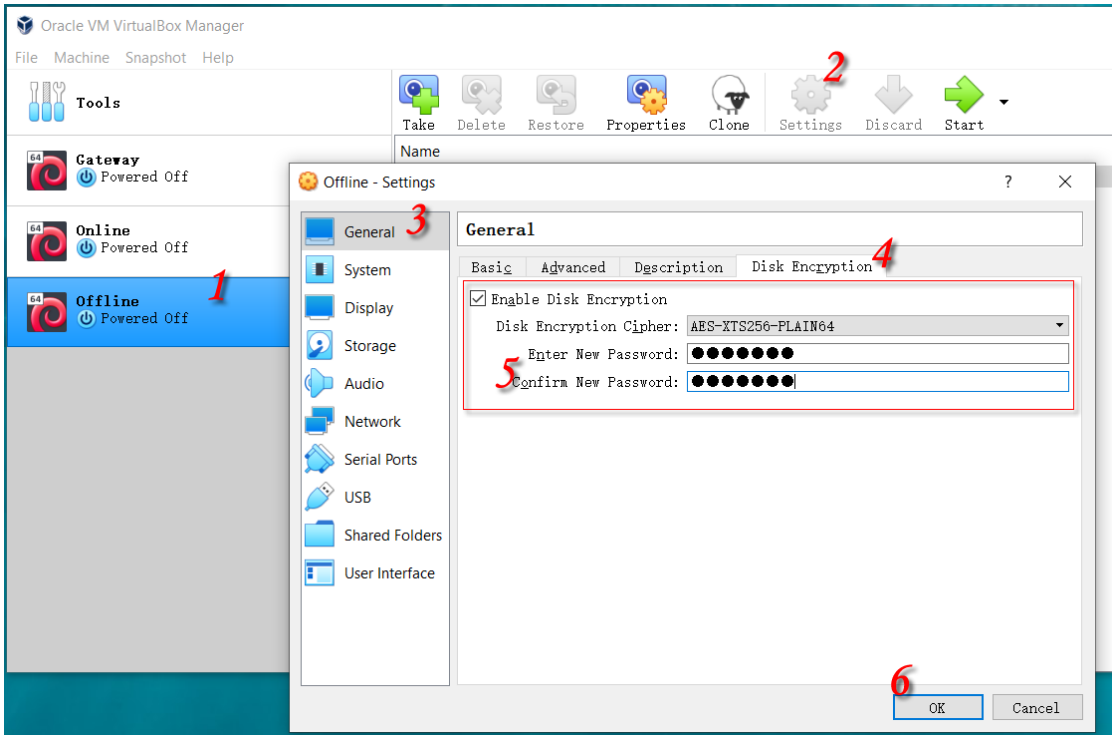


图 8-48 设置开机密码

最后总结一下:

这样我们有了三台虚拟机，即 Gateway、Online 和 Offline，Online 通过 Gateway 走 Tor 网络，而 Gateway 又通过宿主机的 VPN 出墙，而 Offline 完全没有网络，这样保存在其上的私钥不被泄密，再说了它还加了密呢。但是要千万注意，不要把解密后的私钥放到另外两台虚拟机上，在确保 Offline 机器的共享剪贴板被禁用之后，才允许在机器内部粘贴拷贝私钥，否则你在 Offline 内部一拷贝私钥，私钥就被复制到了宿主机的剪贴板中，这样私钥就离开了 Offline 机器了。不要让私钥离开 Offline 机器是一条铁律!

到此，请对三台虚拟机拍个快照，以方便将来虚拟机故障而回滚，参考图 8-49 所示，选择一台虚拟机，然后点击“Take”按钮拍摄一个快照，快照就命名 Snapshot 1。

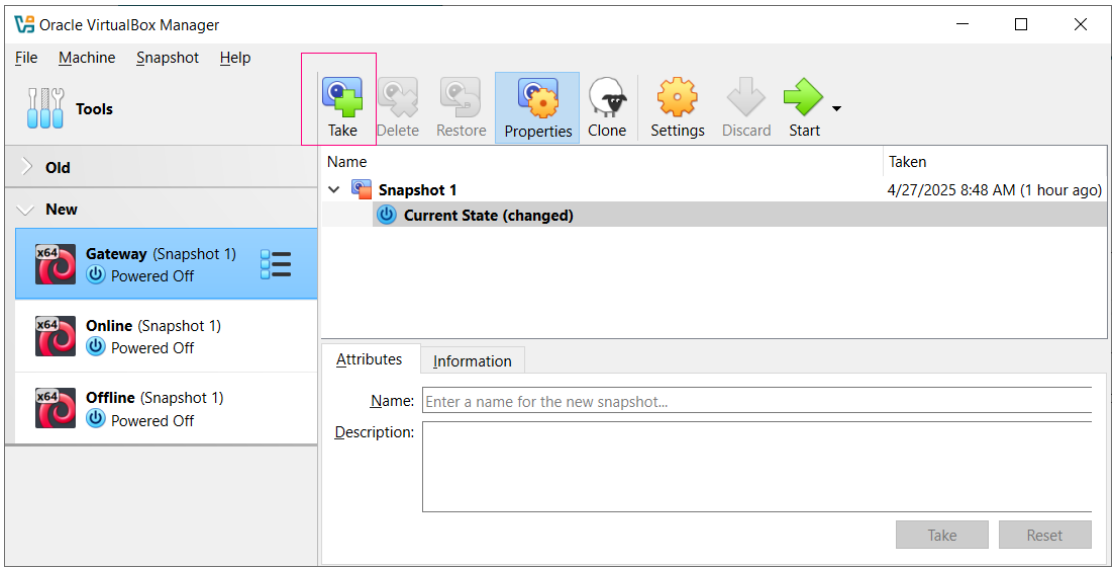


图 8-49 给虚拟机拍摄快照

8.1.6 备份和恢复虚拟机

接下来我们做备份，有两种方式，第一种就是直接备份虚拟机的目录，参考图 8-50 找到虚拟机所在的目录，然后拷贝到其他地方做备份。

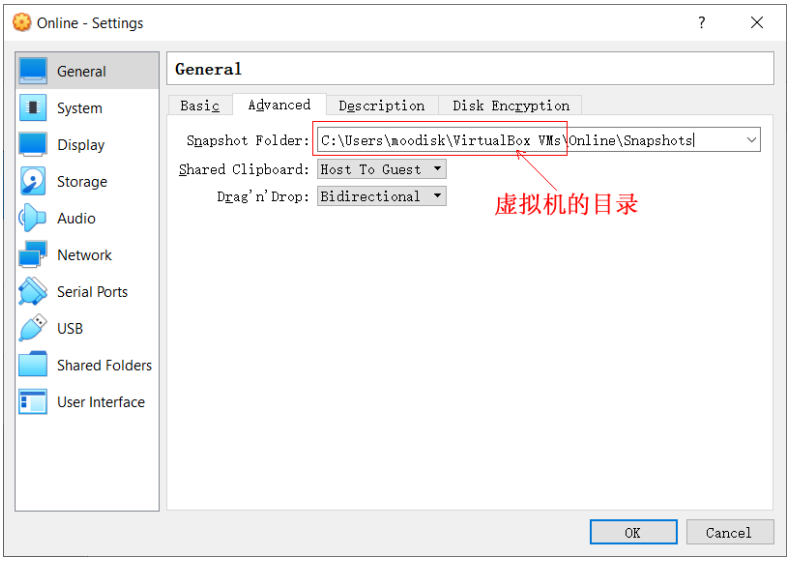


图 8-50 找出虚拟机在宿主机上的目录

第二种方式是导出来，参考图 8-51 操作。

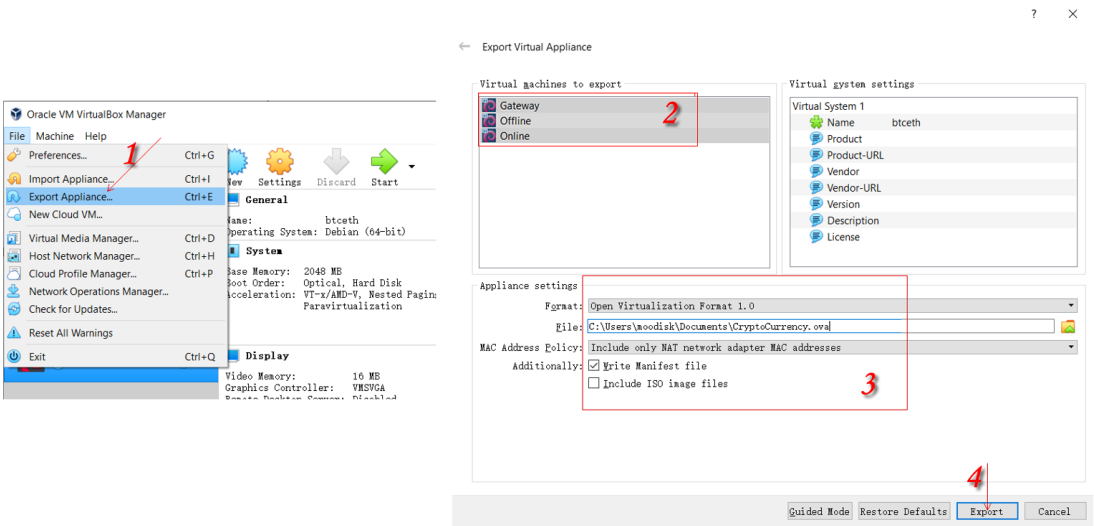


图 8-51 导出虚拟机

选中三台虚拟机，导出的虚拟机文件为 C:\Users\moodisk\Documents\CryptoCurrency.ova，然后就可以把这个文件备份出来。导出大概十几分钟就完成了，参见图 8-52 所示。

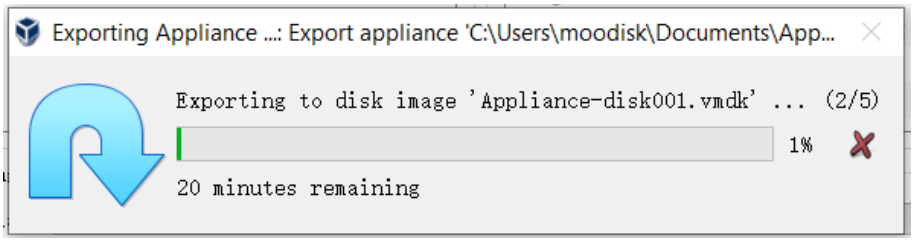


图 8-52 导出三台虚拟机

最后把文件 C:\Users\moodisk\Documents\CryptoCurrency.ova 备份出来即可。同时最好把当前的 virtualbox 安装包也备份一下，以免以后版本升级导致兼容问题。

要恢复虚拟机时，按图 8-53 操作。

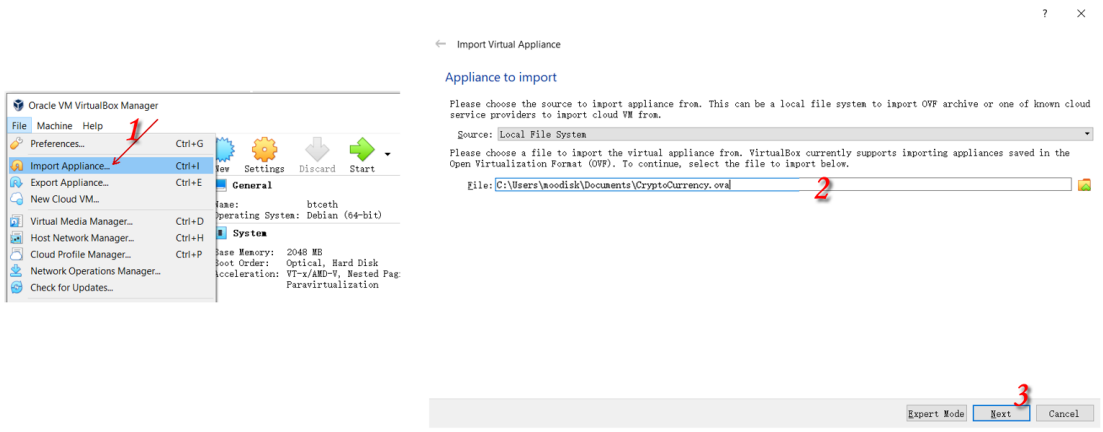


图 8-53 恢复虚拟机

8.2 钱包与交易

首先把 virtualbox 置为专家模式：File→Preferences，然后点击 Expert，最后点击 ok 按钮返回，这样就在 file→Tools 菜单中增加了“Virtual Media Manager”子菜单。

把磁盘设置为读写模式，因为后面创建的私钥要保存在 Offline 虚拟机的磁盘中，点击 File→Tools→Virtual Media Manager 打开图 8-54 所示的界面（或者直接按快捷键 Ctrl+d）。

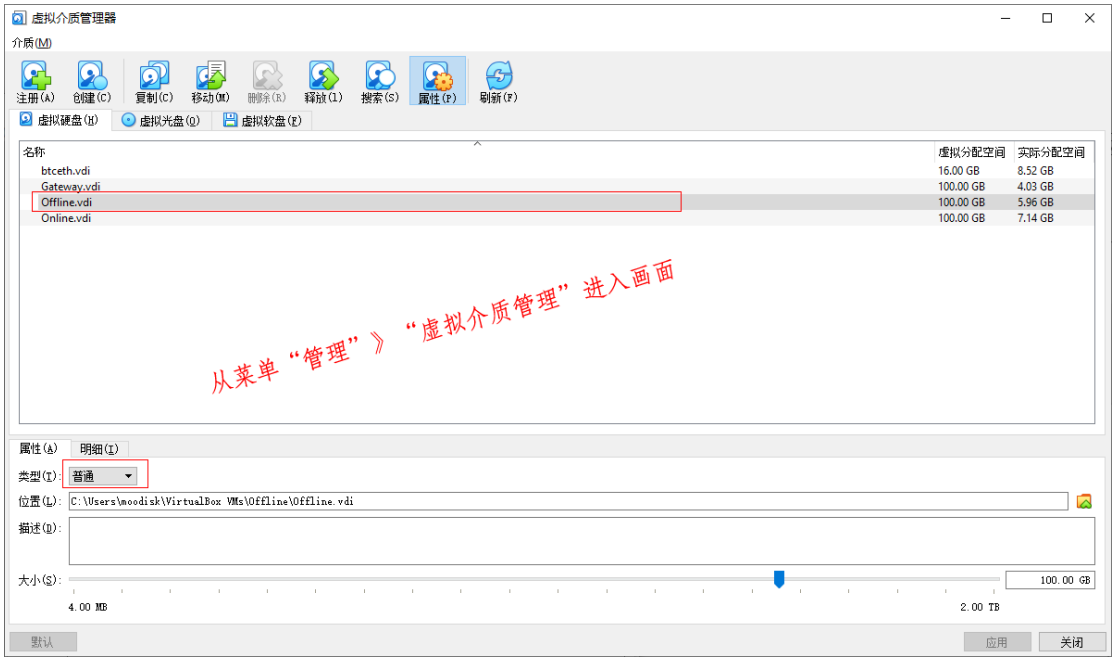


图 8-54 以读写模式启动虚拟机

启动 Offline 虚拟机，Gateway 和 Online 可以不启动。在启动的前几秒钟，往里面点击鼠标，然后移动上下光标键，选择第一项启动虚拟机，参见图 8-55。

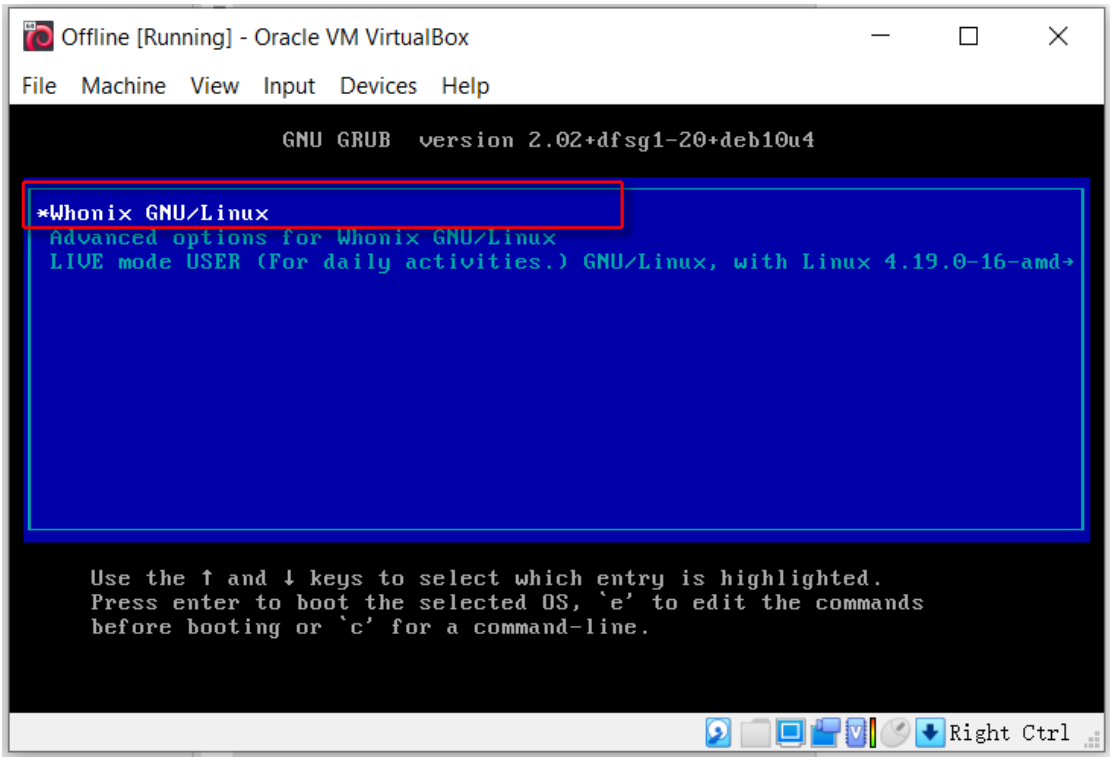


图 8-55 以读写模式启动

可以按键盘右侧的 ctrl 键从里卖弄退出鼠标来。确保 Offline 虚拟机已断网、剪贴板被禁用。

虚拟机启动后，请参考第 6 章创建钱包和放币交易，这里就不再赘述了。

9 终极操作环境

有两种方案可选：

方案 1：采用多块树莓派，以纯硬件的方式把安全性提高到军用级别。
方案 2：x86、16GB+裸机电脑+U 盘安装 Qubes OS，再在 Qubes OS 安装 Whonix。

这里我们采用方案 2，即把 Qubes OS 安装在 U 盘上，再在 Qubes OS 里创建四个 qubes，一个安装 Whonix-Gateway 做 Tor 网关，一个安装 Whonix-WorkStation 做在线工作站，第 3 个安装 Debian 保存诸如私钥等重要资料且对交易签名，始终保持离线状态，第 4 个安装 Debian 处理在线资料，不走 tor 网关。

U 盘至少做两份副本。

9.1 准备工作

9.1.1 准备硬件

1. 电脑：X86 CPU（如 Intel、AMD）、16GB+内存、至少 32GB 硬盘，CPU 能开启虚拟化 VT-x 或 VT-d。

2. U 盘：

从亚马逊或者委托别人从海外购买两个 Corsair CMFSS3B-256GB Flash Survivor Stealth 256GB USB 3.0 Flash Drive，这种 U 盘防水、防磁、防震，产品见图 9-1。



图 9-1 海盗船 u 盘

也可以购买 iStorage datAshur PRO2 128 GB，用了硬密码保护，见图 9-2 所示。不过安装 Qubes OS 时本身就对硬盘加密

了，所以 u 盘的硬件加密功能没有也无所谓。再说产品零件多了故障率也会增加。



图 9-2 datAshur U 盘

9.1.2 准备 QubesOS 安装 U 盘

从 https://mirrors.edge.kernel.org/qubes/iso/Qubes-R4.2.4-x86_64.iso 下载，从 https://github.com/pbatard/rufus/releases/download/v4.7/rufus-4.7.exe 下载烧写 u 盘的工具 rufus，参见图 9-3 操作

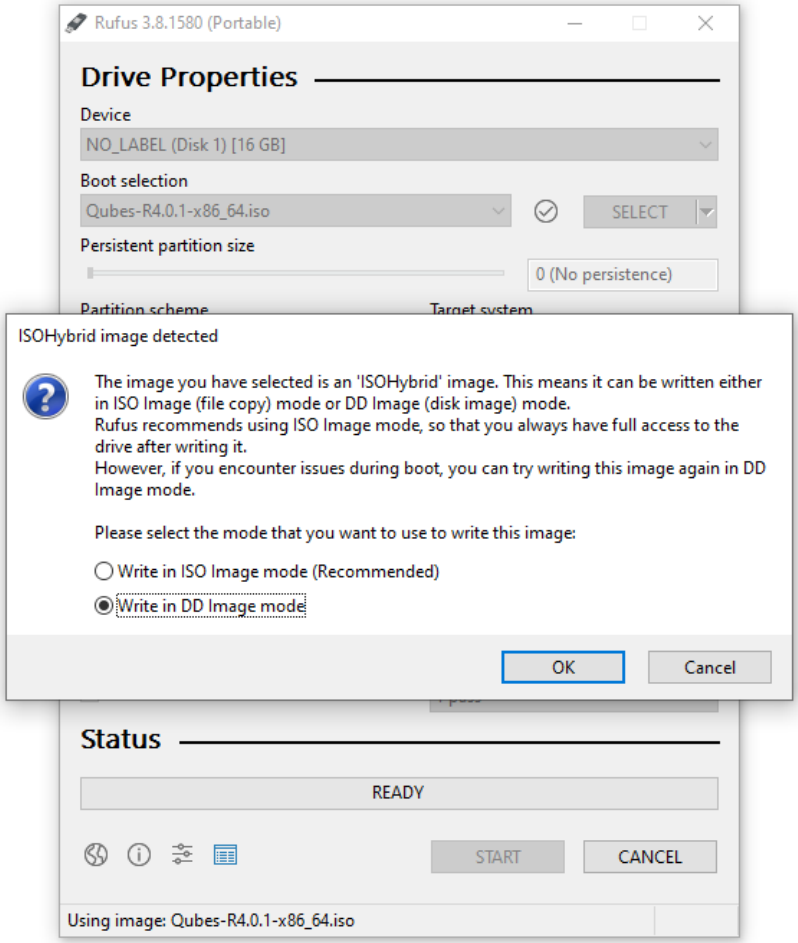


图 9-3 制作启动安装 u 盘

9.1.3 qubes 规划

总体规划参见图 9-4 所示。

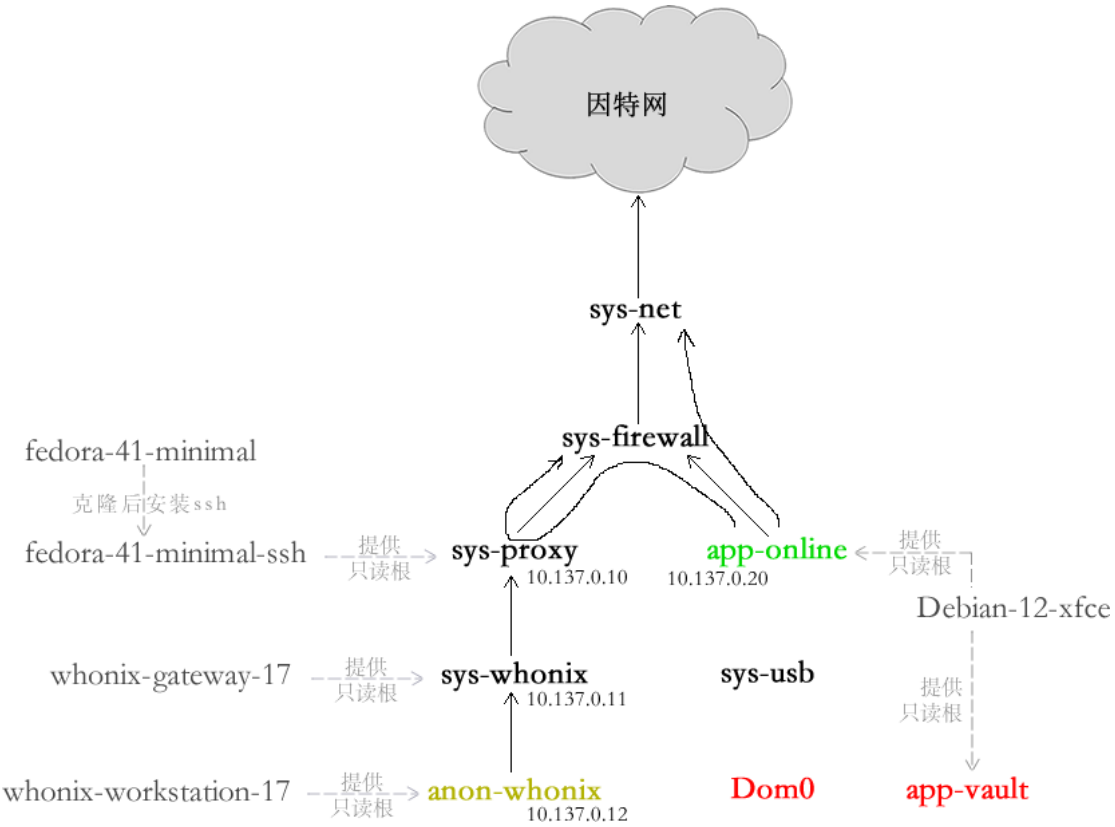


图 9-4 qubes 整体规划图

sys-usb 负责所有的 usb 设备，允许附加给任何正在运行的虚拟机，app-online 虚拟机负责日常上网任务，可以走 vpn 也可以直连互联网。

9.2 安装 Qubes OS

把启动 u 盘插入电脑，并从 u 盘启动电脑（在 BIOS 中设置或者启动时按 F12）。首先出现图 9-5 所示的启动菜单，选择第一项。

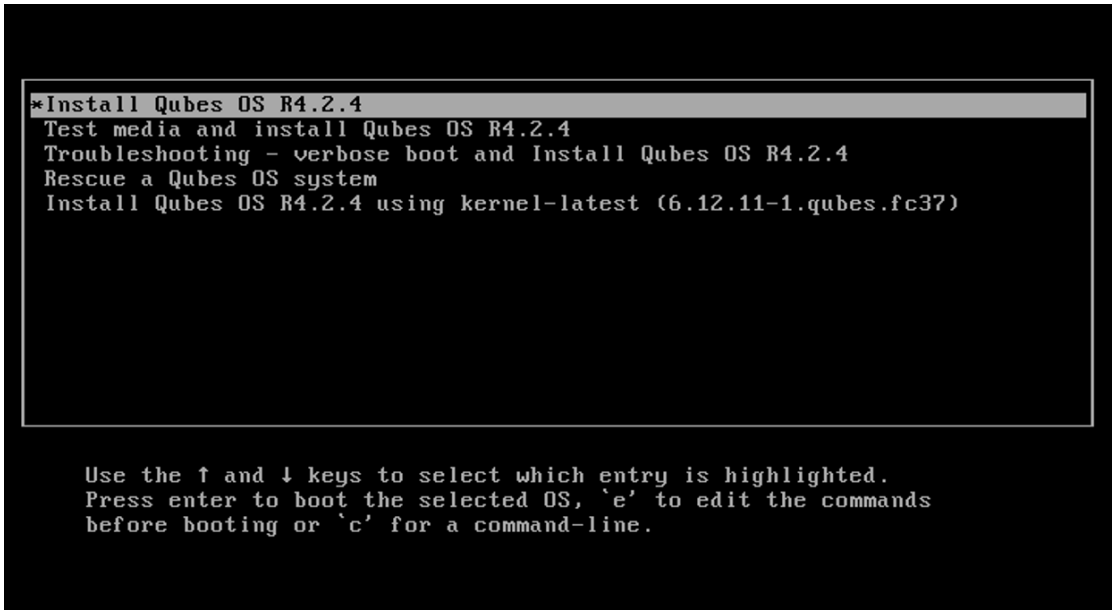


图 9-5 Qube OS 安装启动菜单

然后出现如图 9-6 所示的画面，选择简体中文。



图 9-6 选择语言

然后来到下一个画面，对有关项目设置后如图 9-7 所示的界面。

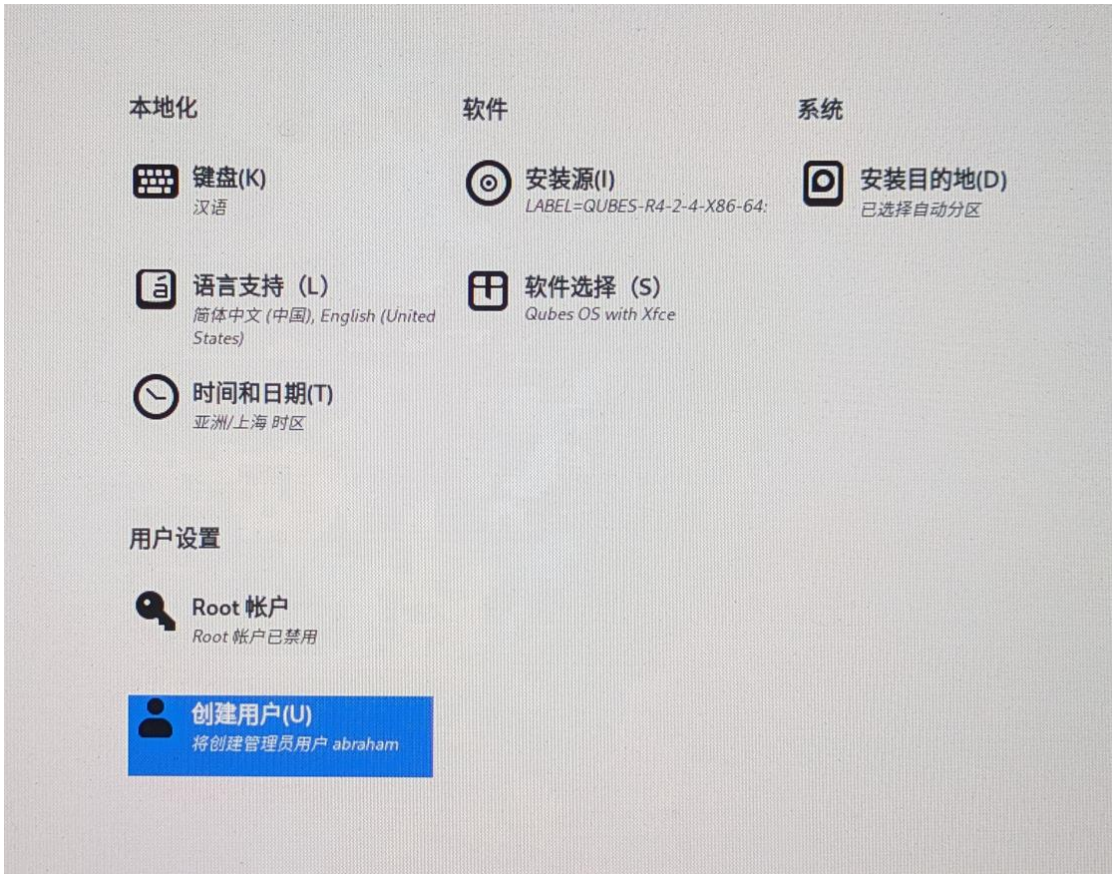


图 9-7 安装信息摘要

点击图中的“安装目的地”后出现图 9-8 的画面，直接点击“完成”后要求输入硬盘加密的密码。



图 9-8 硬盘加密密码

然后又回到 9-7 所示的安装信息摘要画面，往下滚动后再点击“创建用户”，出现 9-9 所示的画面。



图 9-9 创建用户

比如我创建用户 abraham，安装往后就是以此用户登陆的。再点击图 8-7 上的“语言支持”，增加美国英语。再点击图 9-7 上的“时间和日期”，先择亚洲的上海时区。最后点击图 8-7 上的“开始安装”按钮。整个安装过程大概耗时 20 分钟，见图 9-10 所示。



图 9-10 开始安装 Qubes OS

安装后重启，可看到如图 9-11 所示的画面。

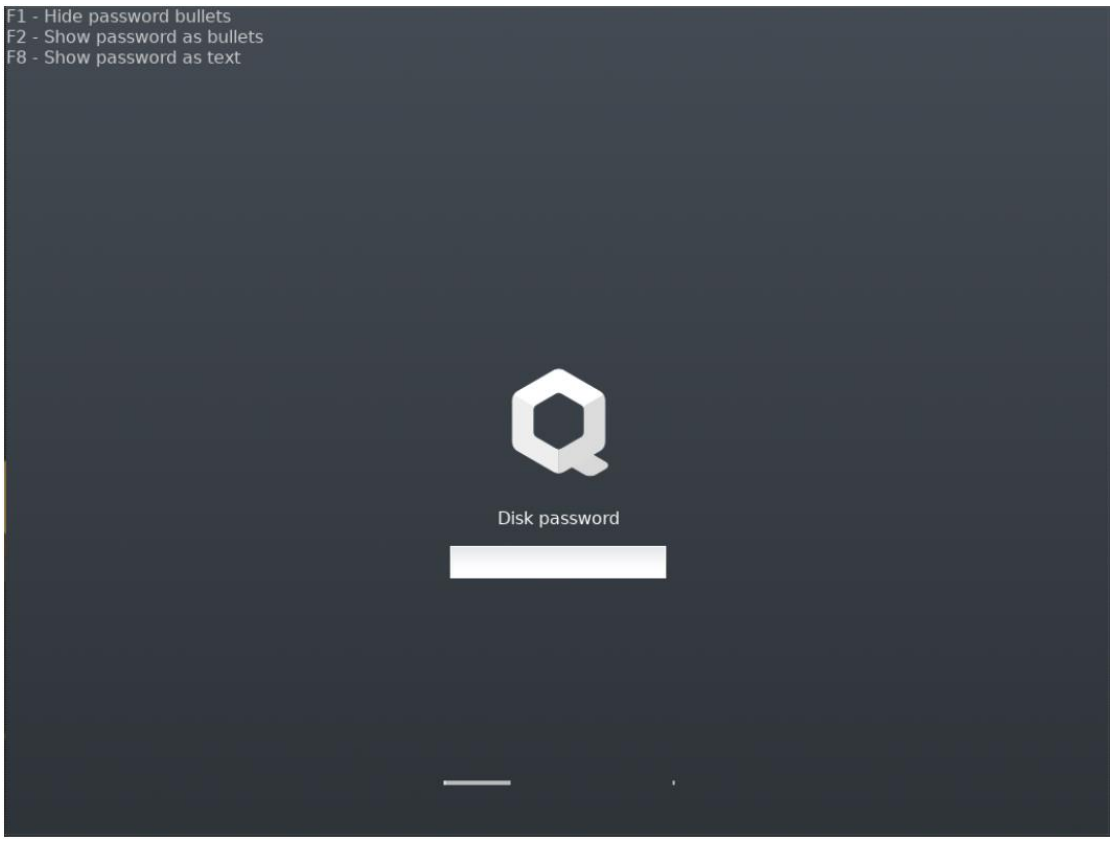


图 9-11 输入硬盘解锁密码

输入硬盘解锁密码之后开始解锁硬盘，然后出现 9-12 所示的界面。

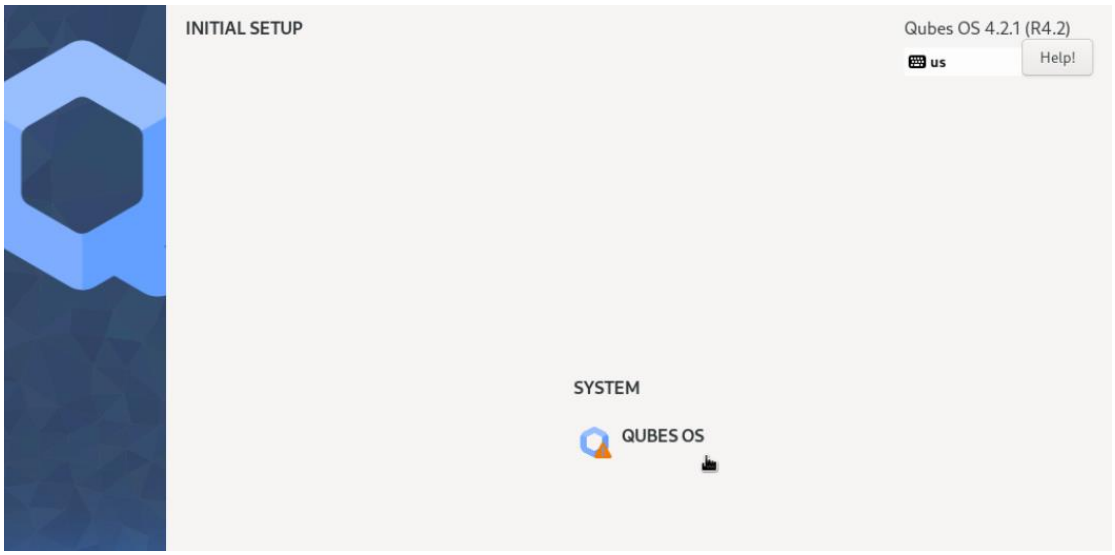


图 9-12 准备初始化

点击那个带感叹号的 QUBES OS 打开图 9-13 所示的界面。

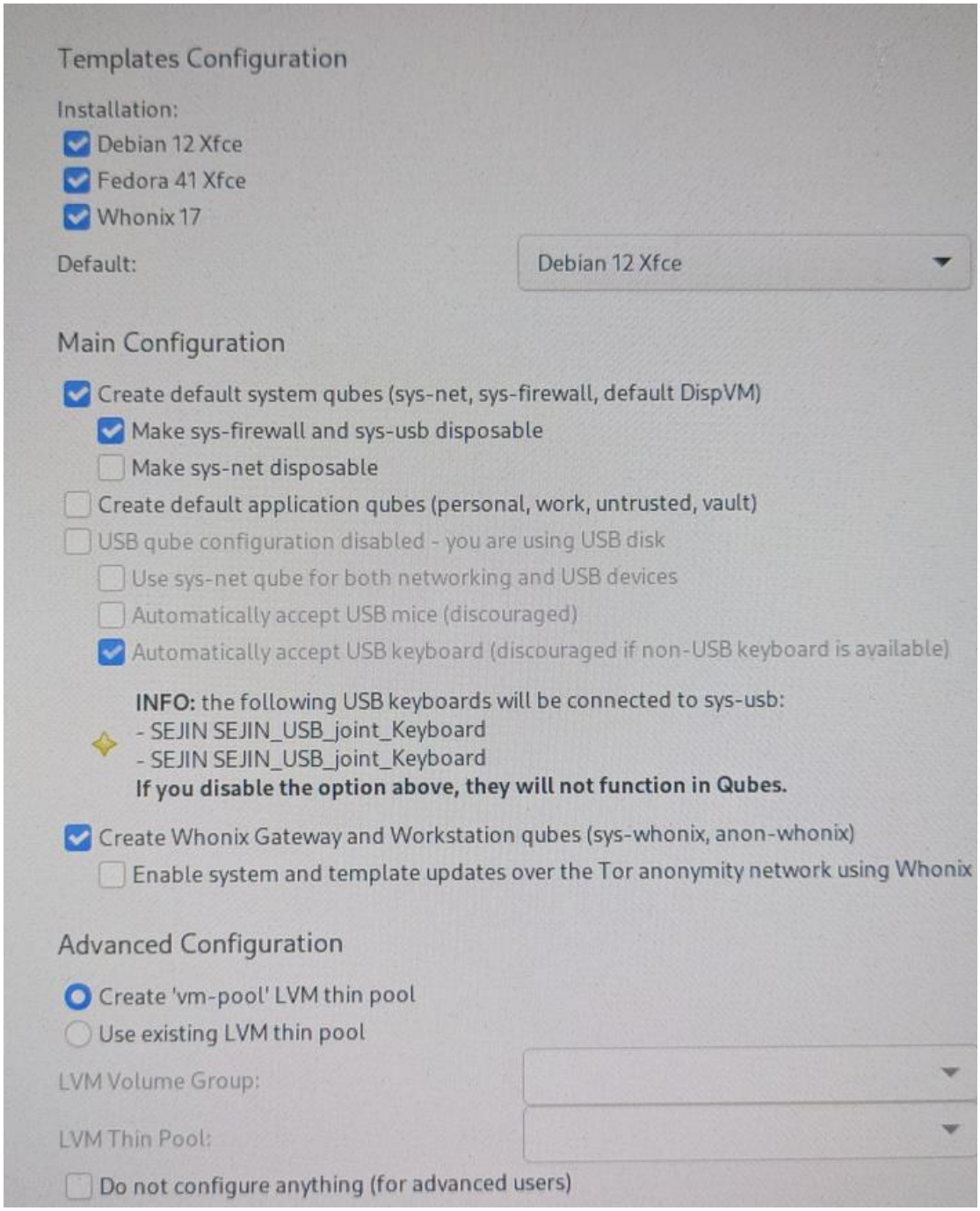


图 9-12 初始化 Qubes OS

安装三个模板 Debian 12 Xfce、Fedora 41 Xfce 和 Whonix17，把 Debian 12 Xfce 设为默认模板，其他选项参考图 9-12 选择。

最后出现如图 9-13 所示的登陆画面。

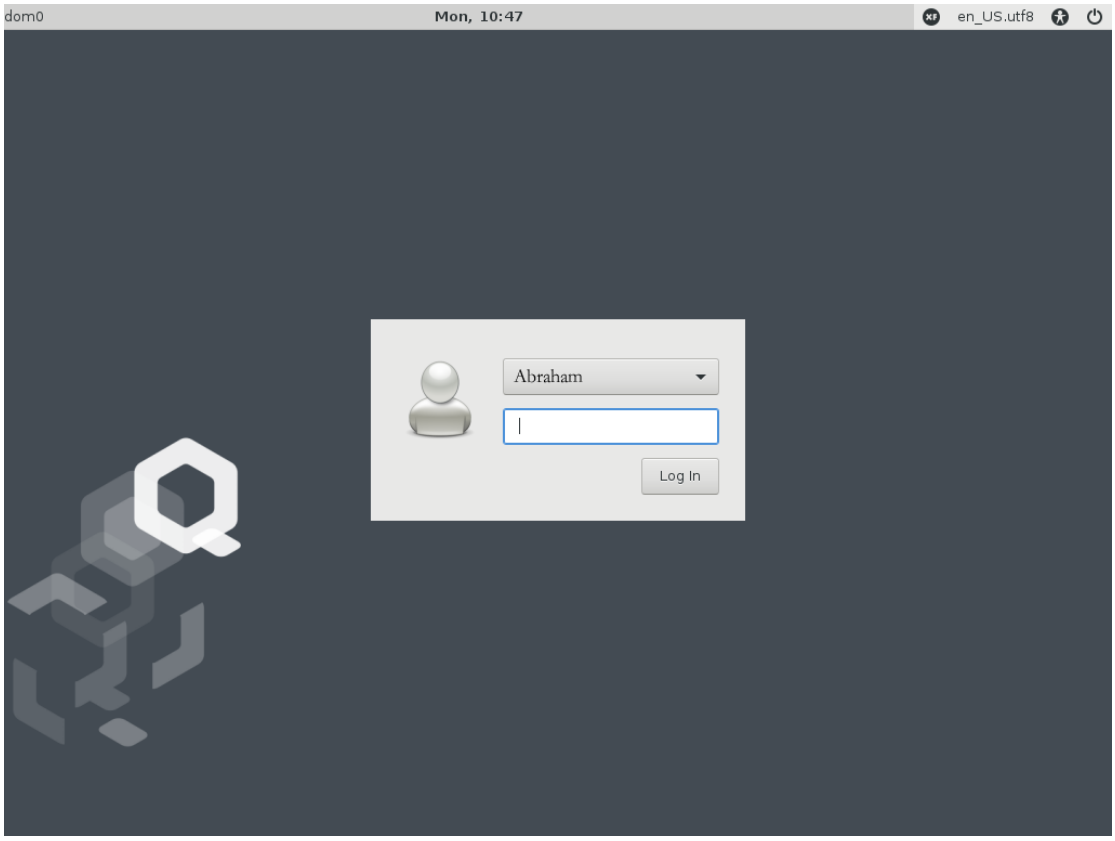


图 9-13 登陆界面

登陆之后，首先要确认鼠标接入 dom0 虚拟机，界面如图 9-14 所示。

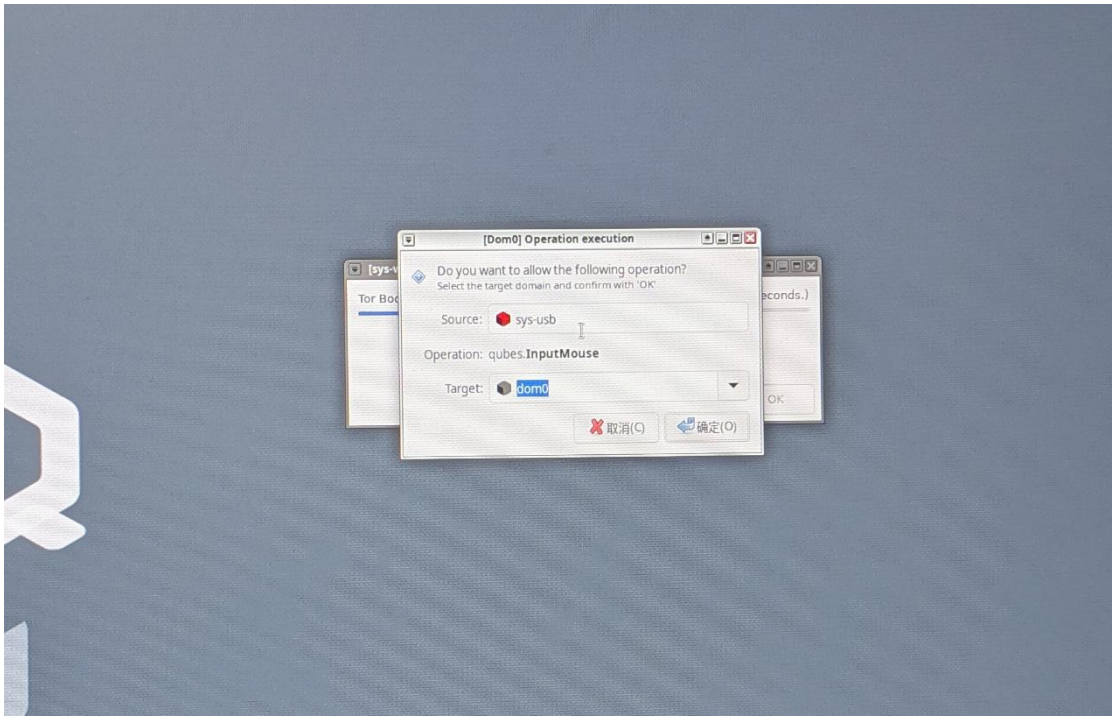


图 9-14 鼠标确认接入 dom0

然后打开 Dom0 终端：右键点击桌面 » 选择 "在此打开终端"，或者直接使用快捷键 Ctrl + Alt + T（默认快捷键可能因版本不同而变化），输入命令显示已有的 qubes 和模板，参考图 9-15。

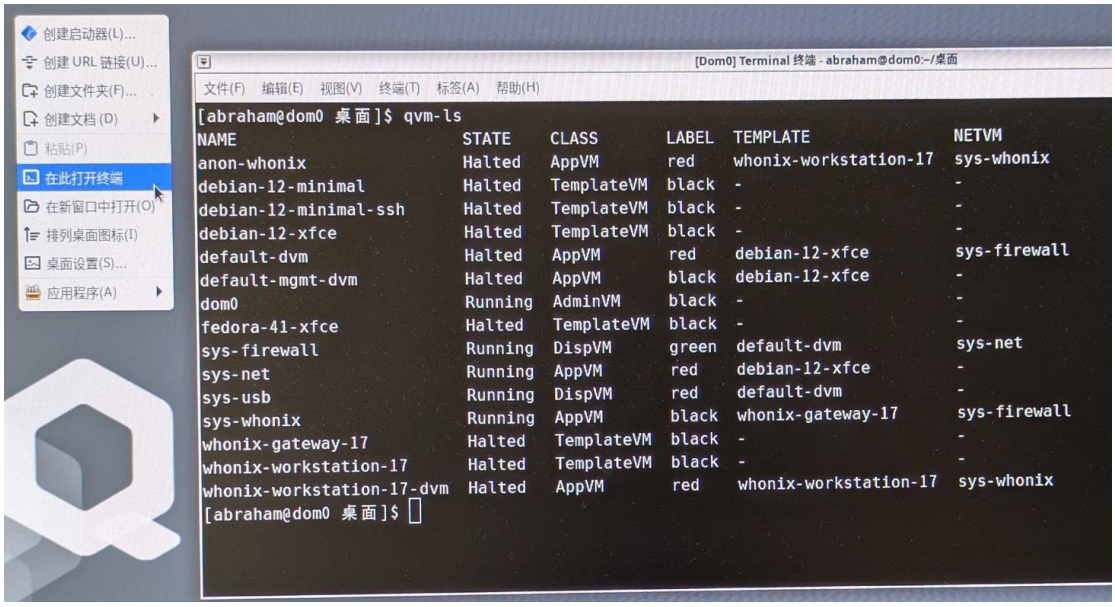


图 9-15 显示刚开始的 qubes

或者点击 Applications » Setting » Qubes Tools » Qubes Manager 打开 qube 管理器，参见图 9-16。

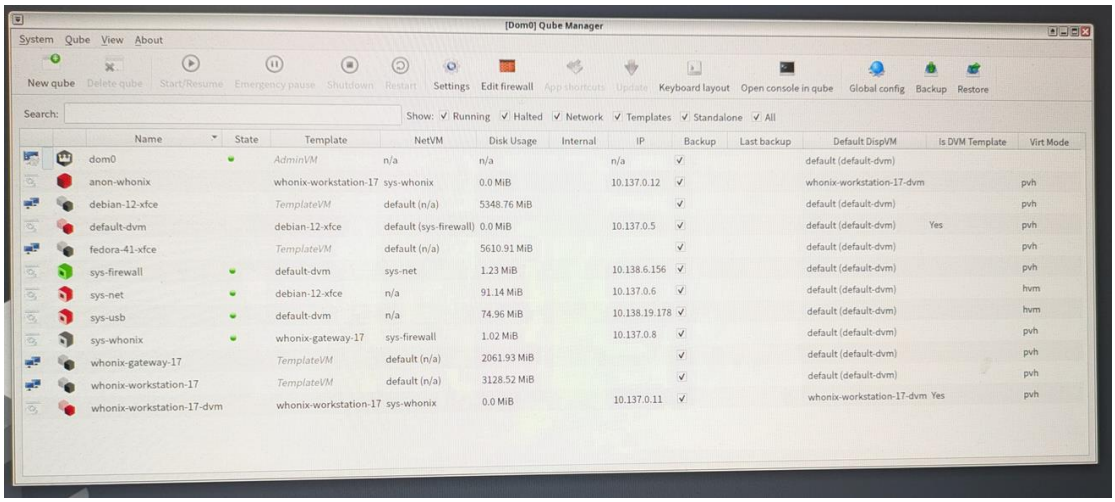


图 9-16 qube 管理器

每个 qube 就是一个虚拟机（xen 为虚拟化软件），虚拟机中除了/home、/usr/local 和/rw 目录外，其他文件都是来自模板，也即模板给虚拟机提供了只读根文件系统，把数据保存在/home、/user/local 或/rw 目录中才是持久化的，如果保存其他地方，那么一关机重启后就没了，所以修改其他目录和文件只能在模板中操作才行。

9.3 创建并配置各个 qubes

9.3.1 Debian-12-minimal 模板

点击左上角的 Applications 图标，然后依次点击 Settings » Qubes Tools » Qubes Template manager 打开模板管理器，参见图 9-17 所示。

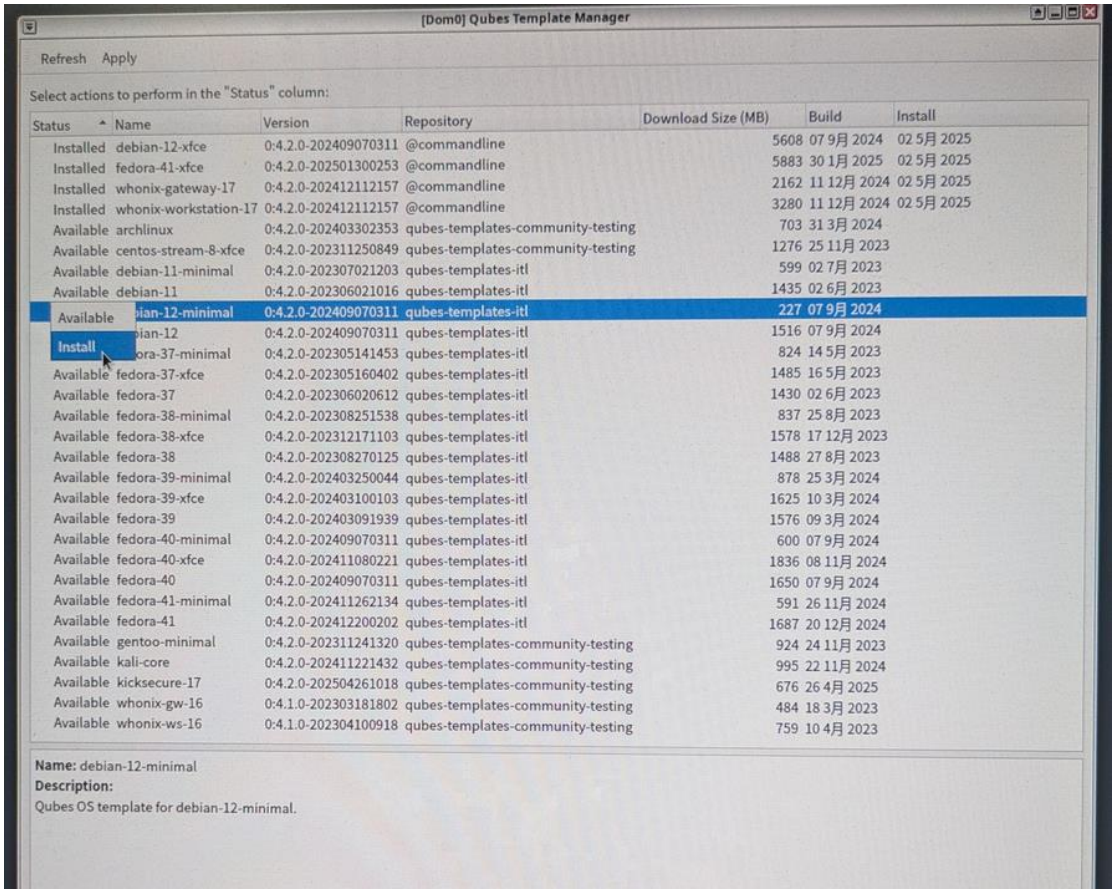


图 9-17 模板管理器

我们安装 debian-12-minimal 模板，这个模板差不多是最小的，还不到 300MB，点击左边的 Available 后再点击 Install，然后点击 Apply 开始安装模板。这个模板比较小，我们接下来用它来做 VPN 客户机。

执行下面命令克隆一份（接下来要在克隆出来的模板中安装一些程序）：

```
user@dom0:~$ qvm-clone debian-12-minimaldebian-12-minimal-ssh
```

接下来启动克隆的模板并安装需要的软件。首先打开模板的命令终端：

```
user@dom0:~$ qvm-run -u root debian-12-minimal-ssh xterm
```

在打开的命令终端中输入命令：

```
user@debian-12-minimal-ssh :~$ sudo apt update -y
```

```
user@debian-12-minimal-ssh :~$ sudo apt upgrade -y
user@debian-12-minimal-ssh :~$ sudo apt install qubes-core-agent-networking ssh less vim-tiny curl qubes-
usb-proxy -y
```

因为是用 ssh 搭建 socks5 梯子，所以最好采用公私钥登陆，这样就不用交互输入密码：

```
ssh-keygen -q -b 4096 -C "Socks5 VPN" -t rsa -f rsa_4096                    //产生一对公私钥
ssh-copy-id -i rsa_4096.pub alice@xx.yy.zzz.xyz                             //把公钥上传到 SSHD 服务器
```

以后就可以执行命令 ssh -C -N -D 0.0.0.0:8080 -i rsa_4096 alice@xx.yy.zzz.xyz 竖起梯子了。

创建一个 app 菜单项，以后点击它时执行 ssh 命令竖起梯子。继续在模板的终端中执行：

```
vi /usr/share/applications/ssh-vpn.desktop
[Desktop Entry]
Name=SSH VPN
Exec=/usr/bin/ssh -C -N -D 0.0.0.0:8080 -i ~/rsa_4096 siguobao@34.133.211.54
Icon=ssh
Comment=Ssh Socks5 Vpn
Type=Application
StartupNotify=false
Terminal=true
```

编辑两个 ssh.png 小图标，尺寸分别是 48x48、32x32，放在 /usr/share/icons/hicolor/48x48/apps/ 和 /usr/share/icons/locolor/32x32/apps/目录中。

在 dom0 的终端中执行命令：

```
user@dom0:~$ qvm-sync-appmenus debian-12-minimal-ssh
```

或者在 debian-12-minimal-ssh 的设置界面上点击 Applications 》Refresh Applications 按钮刷新可用的应用程序菜单。

如果只想在一个 AppVM 中显示菜单，就打开此 AppVM 的终端，然后在~/.local/share/applications 或 /usr/local/share/applications 目录先编辑 ssh-vpn.desktop 文件，然后在虚拟机的设置界面上点击 Applications 》Refresh Applications 按钮刷新可用的应用程序菜单，参考图 9-23 所示操作。

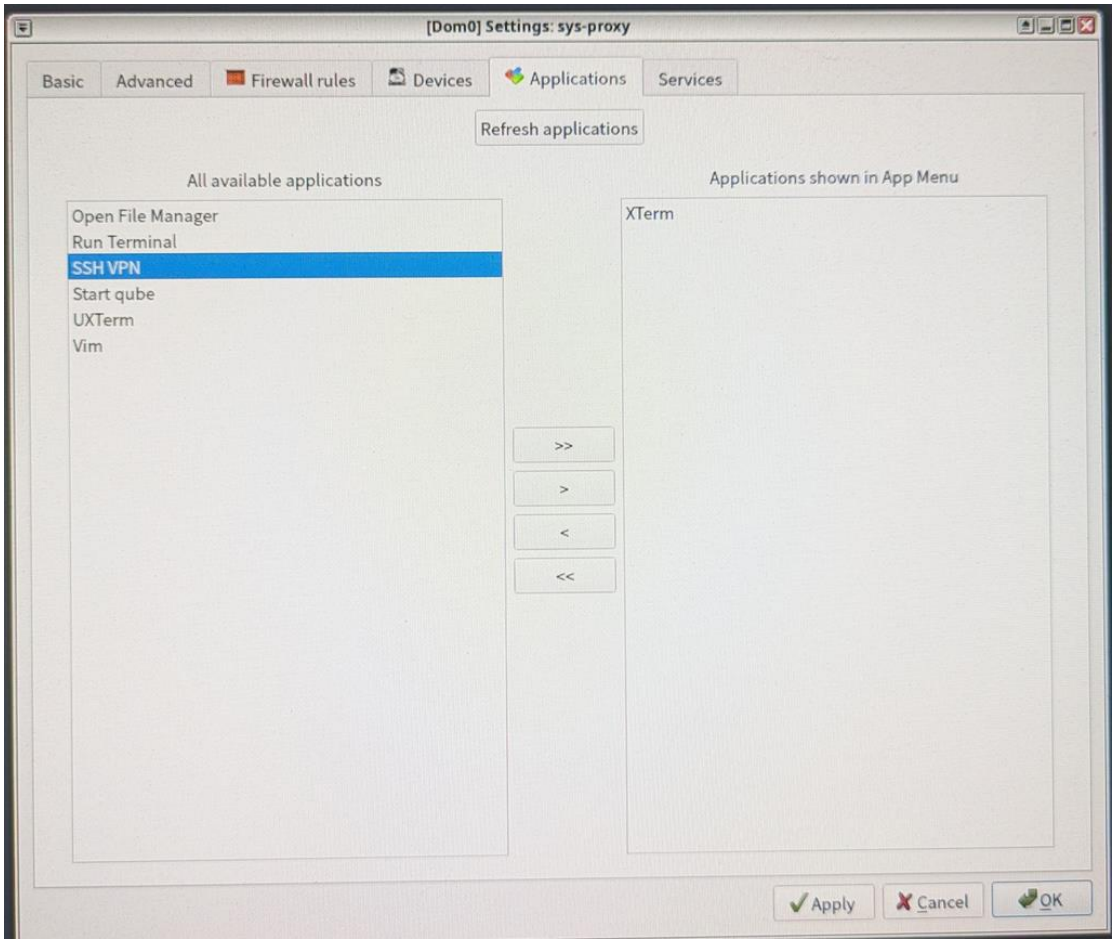


图 9-23 增加应用程序菜单

9.3.2 准备 sys-proxy

从模板 debian-12-minimal-ssh 创建虚拟机 sys-proxy，点击 Applications 》Settings 》 Create New Qube，打开如图 9-24 所示的画面。

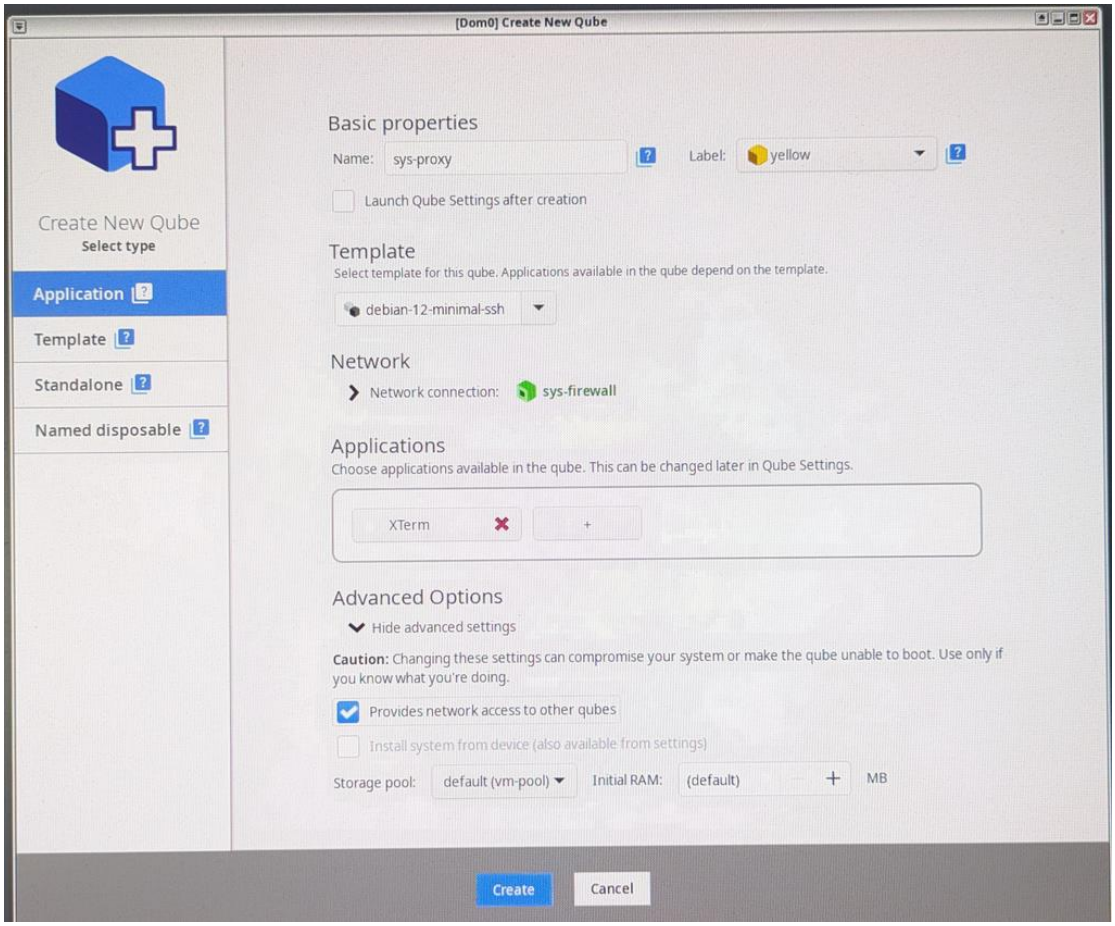


图 9-24 创建 sys-proxy 虚拟机

设置固定的 ip：

```
user@dom0:~$ qvm-prefs sys-proxy ip 10.137.0.10
```

设置防火墙：

```
user@dom0:~$ qvm-features sys-proxy qubes-firewall true        //开启增强防火墙规则，此后 qvm-firewall 命令才有效
user@dom0:~$ qvm-firewall sys-proxy action=accept dport=8080  //在 sys-firewall 中设置 8080 端口转发到 sys-proxy
```

进入 sys-proxy 设置防火墙（全部接受源端口 8080 的请求）：

```
user@sys-proxy:~$ sudo echo 'nft add rule ip qubes custom-input tcp dport 8080 accept' >>
/rw/config/rc.local
user@sys-proxy:~$ sudo echo 'nft add rule ip qubes custom-input ip saddr 10.137.0.11 accept' >>
/rw/config/rc.local
```

点击 Applications » Qubes » Services » sys-proxy » Setting 打开虚拟机的设置画面，点击 Applications 页上的“Refresh Applications”按钮刷新应用程序列表，把里面的 SSH VPN 移到右侧，这样 SSH VPN 就出现在 Applications » Qubes » Services » sys-proxy 的菜单上，点击它竖起梯子。然后在 sys-proxy 的终端执行命令：

```
user@sys-proxy:~$ curl --socks5 127.0.0.1:8080 ifconfig.me                    //如果返回 SSHD 服务器的地址就对了
```

9.3.3 配置 sys-whonix

1. 修改上游 qube 和固定 ip：

```
user@dom0:~$ qvm-prefs sys-whonix netvm sys-proxy
user@dom0:~$ qvm-prefs sys-whonix ip 10.137.0.11
```

2. 配置 Tor

Applications 》Qubes 》SERVICE 》sys-whonix 》Anon Connection Wizard，之后会出现如图 8-14 所示的画面，并参考那里的步骤配置 Tor，只不过梯子的地址为 10.137.0.10。

9.3.4 配置 anon-whonix

首先测试 Tor 网是否通畅：

```
user@anon-whonix:~$ torsocks curl -v https://check.torproject.org
```

9.3.4.1. 准备相关文件

把有关文件拷贝到 u 盘，然后把 u 盘附加到 anon-whonix 虚拟机（具体参考 9.4.2 小节），最后把 u 盘上的 www 目录拷贝到用户主目录中。

9.3.4.2 安装 Bisq

通过 u 盘把 Bisq 的安装文件拷贝到虚拟机，然后参考 7.4 小节安装。

9.3.5 准备 app-online

app-online 虚拟机专门用于在线浏览网站，既可以通过梯子，也可以直通互联网。先从 debian-12-xfce 模板克隆另一个模板 debian-12-xfce-cn，然后加装中文及中文输入法：

```
user@dom0:~$ qvm-clone debian-12-xfce debian-12-xfce-cn
user@dom0:~$ qvm-run -u root debian-12-xfce-cn xfce4-terminal
```

在打开的终端中执行：

```
root@debian-12-xfce-cn:~# apt install fcitx5-chinese-addons fonts-wqy-zenhei fonts-wqy-microhei -y
```

然后关闭模板，再从它创建虚拟机 app-online：

```
user@dom0:~$ qvm-shutdown debian-12-xfce-cn
user@dom0:~$ qvm-create --template debian-12-xfce-cn --label green app-online --property netvm none
```

```
user@dom0:~$ qvm-prefs app-online ip 10.137.0.20                    //设置固定 ip
user@dom0:~$ qvm-run app-online xfce4-terminal
```

在打开的终端中输入命令让 fcitx5 自动运行：

```
user@app-online:~$ mkdir ~/.config/autostart
user@app-online:~$ cp /usr/share/applications/org.fcitx.Fcitx5.desktop ~/.config/autostart/
```

关闭并重启 app-online：

```
user@dom0:~$ qvm-shutdown app-online
user@dom0:~$ qvm-start app-online
```

进入 app-online 的设置界面（Applications 》Qubes 》app-online 》Settings）的 Applications 页，刷新应用后选择左侧的 Fcitx 5 Configuration 加到右侧。然后启动 Fcitx 5 Configuration 添加拼音输入法，参考图 9-25。

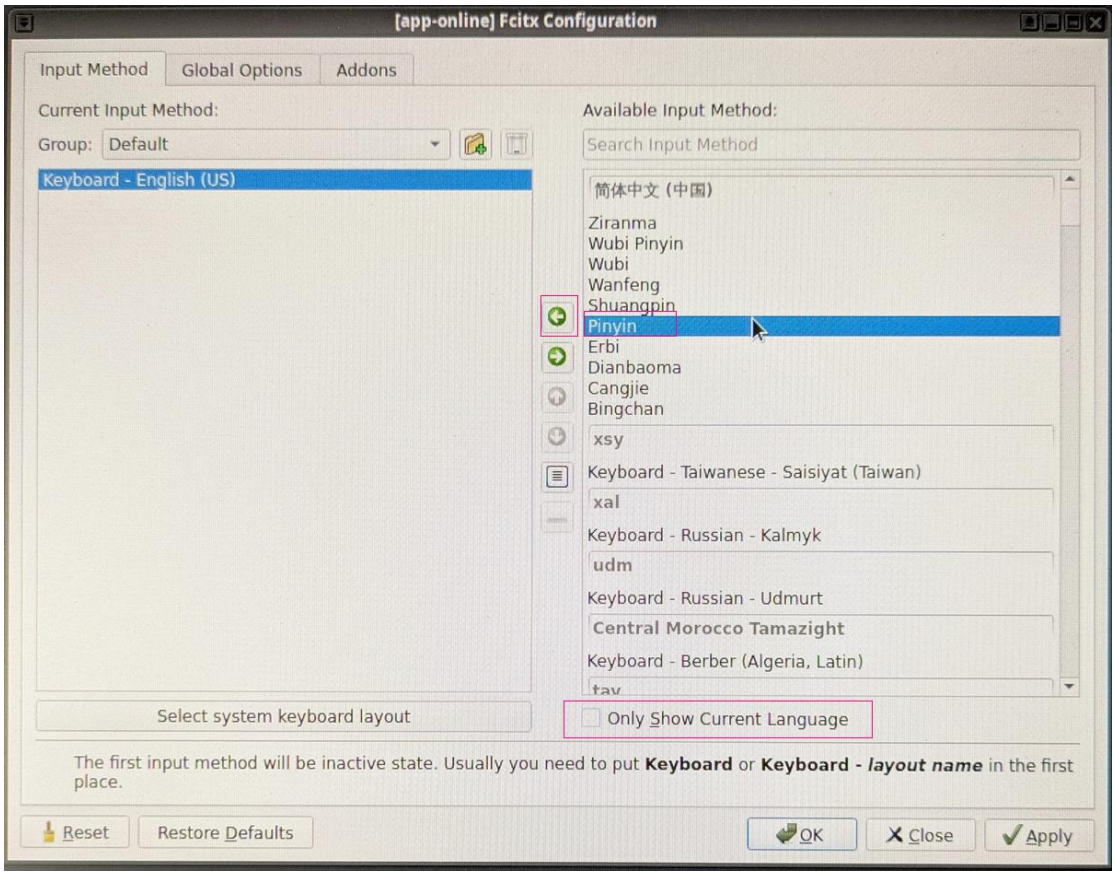


图 9-25 添加拼音

以后当启动 app-online 后会在屏幕右上角出现输入法的小键盘图标，点击它加入拼音输入法。以后运行 app-online 里的应用程序（比如 firefox）按 Ctrl+space 即可打开中文输入法。

9.3.5.1 准备相关文件

参考 8.1.4.3 小节操作。

9.3.5.2 安装 jami

从 u 盘考入 jami 安装包，然后参考 7.5 小节操作。

9.3.5.3 让浏览器走 VPN

路径是 app-online → sys-firewall → sys-proxy → sys-firewall → sys-net →外网。app-online 要访问 sys-proxy 上的 8080/tcp 端口，执行下面的命令添加防火墙策略：

1. app-online 接受来自 sys-proxy 的连接：

```
user@app-online:~$ sudo echo 'nft add rule ip qubes custom-input ip saddr 10.137.0.10 accept' >>
/rw/config/rc.local
```

2. sys-proxy 接受来自 app-online 的连接：

```
user@sys-proxy:~$ sudo echo 'nft add rule ip qubes custom-input ip saddr 10.137.0.20 accept' >>
/rw/config/rc.local
```

3. sys-firewall 转发双方的连接：

由于 sys-firewall 是 dispVM（一次性），保存的任何数据重启之后全部清空了。因此只能在它的模板上添加防火墙策略，但是其模板 default-dvm 被众多虚拟机使用，所以最好是先克隆一个出来：

```
user@dom0:~$ qvm-clone default-dvm default-dvm-fw
user@dom0:~$ qvm-run -u root default-dvm-fw xfce4-terminal
```

在新打开的终端中执行：

```
root@default-dvm-fw:~# echo 'nft add rule ip qubes custom-forward ip saddr 10.137.0.10 ip daddr
10.137.0.20 accept' >> /rw/config/qubes-firewall-user-script
root@default-dvm-fw:~# echo 'nft add rule ip qubes custom-forward ip saddr 10.137.0.20 ip daddr
10.137.0.10 accept' >> /rw/config/qubes-firewall-user-script
root@default-dvm-fw:~# shutdown -h now
```

然后在 sys-firewall 的设置界面上把模板改为 default-dvm-fw，重启 sys-firewall 即可。

4. 在 app-online 上测试(先要在 sys-proxy 上竖起梯子)：

```
user@app-online:~$ curl --socks5 10.137.0.10:8080 ifconfig.me                    //如果返回 ssh 服务器的 ip 表示 ok
```

或者启动 firefox 浏览器，然后参考“图 2-22 firefox 设置代理（一）”操作。

9.3.6 准备 app-vault

9.3.6.1 创建 app-vault

从模板 debian-12-xfce-cn 创建虚拟机，要求彻底禁用网络，使它成为孤岛，因为里面保存了私钥等重要资料。参考图 9-26 操作。

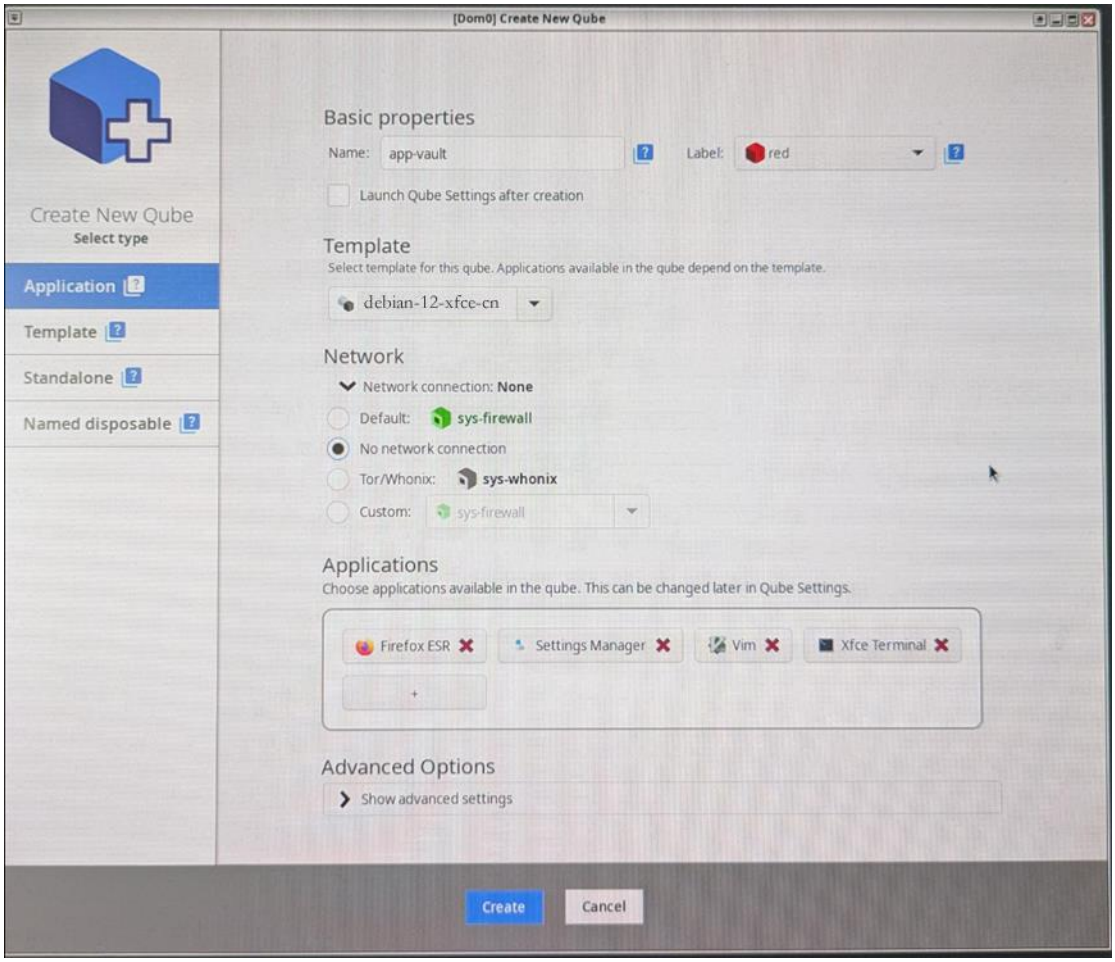


图 9-26 创建 app-vault 虚拟机

点选“No network connection”禁止使用网络，这样能绝对保正虚拟机的私钥安全。或者使用下面的命令创建 app-vault：

```
user@dom0:~$ qvm-create --template debian-12-xfce-cn --label red app-vault --property netvm none
```

只允许 app-vault 与 anon-whonix 之间拷贝粘贴内容：

```
user@dom0:~$ sudo vi /etc/qubes/policy.d/30-valut.policy
qubes.ClipboardCopy * app-vault anon-whonix allow
qubes.ClipboardCopy * anon-whonix app-vault allow
qubes.ClipboardPaste * app-vault anon-whonix allow
qubes.ClipboardPaste * anon-whonix app-vault allow
```

存盘退出 vi，策略理解生效。如果要彻底关闭剪贴板，请执行下面的命令：

```
user@dom0:~$ qvm-prefs app-vault gui-agent false
```

这样即使 app-vault 内部的程序之间也不能拷贝粘贴，需要时可重新开启：

```
user@dom0:~$ qvm-prefs app-vault gui-agent true
```

切记：只在 app-vault 内部拷贝粘贴被密码保护的私钥，把已经签名的交易拷贝到 anon-whonix，把 UTXO 所在的交易 Hex 码拷贝到 app-vault（签名时需要），非必要就不要暴露公钥。

不过最佳实践是这样的：不允许 app-vault 与其他任何虚拟机之间拷贝粘贴资料，app-vault 和 anon-whonix 之间拷贝交易 hex 码时通过文件方式，具体操作如下：

```
user@dom0:~$ sudo vi /etc/qubes-rpc/policy/30-disableClipboard.policy                    //禁用
内外拷贝
```

```
qubes.ClipboardCopy * appVM1 @anyvm deny
qubes.ClipboardCopy * @anyvm appVM1 deny
qubes.ClipboardPaste * appVM1 @anyvm deny
qubes.ClipboardPaste * @anyvm appVM1 deny
```

把文件 sign.txt 拷贝到 anon-whonix 虚拟机的/home/user/QubesIncoming/app-vault/sign.txt：

```
user@app-vault:~$ qvm-copy --target anon-whonix sign.txt
```

也可以移动文件：

```
user@app-vault:~$ qvm-move --target anon-whonix sign.txt
```

9.3.6.2 准备文件

从 u 盘拷贝 www 目录到 user 用户的主文件夹，再把加密后的私钥、钱包地址、公钥等拷贝到 user 主文件夹中，保存在 wallet 文件中，然后采用下面的命令对 wallet 文件进行加密：

```
user@app-vault:~$ gpg -c wallet                                    //-c 等价于--symmetric 参数，即用对称密码加密
```

加密的密码尽量复杂一些，最好参杂大小写字母、标点符号和数字，字符个数不低于 9。加密后的文件为 wallet.gpg，把 wallet 文件删除。解密时执行下面的命令：

```
user@app-vault:~$ gpg -d wallet.gpg
```

解密后的明文直接显示在屏幕上。

9.4 日常使用

9.4.1 拷贝和粘贴

1. 在 AppVM 之间拷贝内容：A → B

在 A 中选择要拷贝的内容，按 Ctrl+c 拷贝到 A 的剪贴板中，再按 Ctrl+Shift+c 从 A 的剪贴板拷贝到全局剪贴板，打开 B 的目标程序，然后按 Ctrl+Shift+v 从全局剪贴板拷贝到 B 的剪贴板，并清空全局剪贴板，最后按 Ctrl+v 拷贝到目的地。

2. 在 AppVM 之间拷贝或移动文件：A → B

打开 A 的文件管理器，选中要拷贝的文件，并右击，然后选择 Copy to Other AppVM 或者 Move to Other AppVM，在弹出的对话框中选择目的虚拟机，即 B。这样文件被拷贝到了 B 的/home/user/QubesIncoming/A/目录中。最后进入 B 中把文件拷贝到正确的位置即可。

也可以采用命令 qvm-copy 或 qvm_move。

3. 从 dom0 拷贝内容至 AppVM：dom0→A

在 dom0 中选择并按 ctrl+c 拷贝到剪贴板，点击通知区的 Qubes Clipboard，再点击 Copy dom0 clipboard 把内容拷贝到了全局剪贴板中，进入 A 后按 Ctrl+Shift+c 后再按 Ctrl+c 完成拷贝。

4. 从 dom0 拷贝文件到 AppVM：dom0 → A

使用命令 qvm-copy-to-vm A <file>把文件拷贝到了 A 的/home/user/QubesIncoming/dom0/目录中，然后进入 A 再把文件拷贝到正确的目录。或者采用命令 cat /path/to/file_in_dom0 | qvm-run --pass-io A 'cat > /path/to/file_name_in_appvm'

5. 从 AppVM 拷贝到 dom0： A → dom0

一般不要这样做，这可能会大致极大的风险，实在要拷贝，请采用命令 qvm-run --pass-io A 'cat /path/to/file_in_src_domain' > /path/to/file_name_in_dom0。

9.4.2 使用 U 盘

有两种方法，一种是点击鼠标方式，另一种是命令方式。

方式 1、鼠标方式：点击屏幕右上角的设备管理图标  ，会列出一系列的可用设备，USB 设备排在最下面，鼠标悬停其上会列出能附加的正在运行的虚拟机或模板，选择一个虚拟机则把此 U 盘链接上了，参见图 9-27 所示。

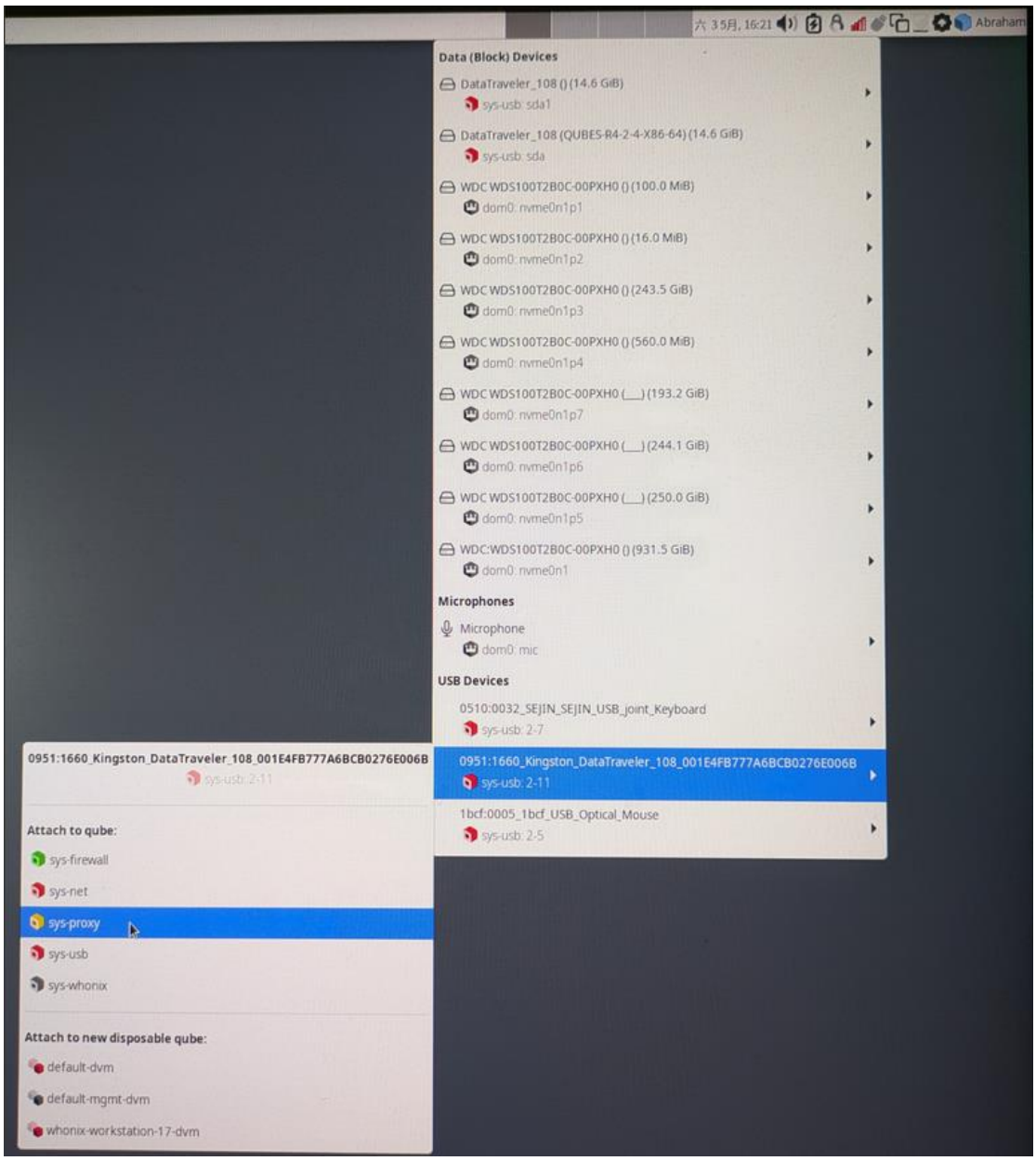


图 9-27 把 u 盘附加到某台虚拟机

然后就可以在那个虚拟机里使用 u 盘了，一般映射为/dev/sda 设备，使用命令 mount /dev/sda1 /mnt 挂载，然后就可以进入/mnt 目录看到 u 盘里的文件了。

方式 2、命令方式：在 dom0 的终端里，执行命令 qvm-usb 列出已识别的 u 盘，然后执行命令 qvm-usb attach <虚拟机名> < BACKEND:DEVID>，如 qvm-usb attach anon-whonix sys-usb:2-5。然后就可以在 anon-whonix 中使用 u 盘了。用完了执行命令 qvm-usb detach <虚拟机名> < BACKEND:DEVID>。

9.4.3 数据持久化

之前提到过，虚拟机中除了/home、/usr/local 和/rw 目录外，保存在其他地方的数据重启之后就没了，所以如果不是非必

要的话，想永久保存的数据就放在这三个目录当中。

但是有的程序的数据目录在其他地方，比如 Tor 的数据目录固定为/var/lib/tor，显然这个目录必须持久化！怎么办？只能采用绑定目录（bind-dir）技术了。

下面我们一持久化/var/lib/tor 目录为例说明整个操作过程。打开目标虚拟机（如 appVM）的命令终端，执行如下的命令：

```
user@appVM:~$ sudo mkdir -p /rw/config/qubes-bind-dirs.d
user@appVM:~$ sudo echo "binds+=( '/var/lib/tor' )" >> /rw/config/qubes-bind-dirs.d/50_user.conf
```

如果虚拟机对应的模板中已经有了/var/lib/tor 目录，那么重启虚拟机后会自动绑定/var/lib/tor 目录到/rw/bind-dirs/var/lib/tor，这样就实现了持久化。如果模板中没有/var/lib/tor 目录，那么请手工创建/rw/bind-dirs/var/lib/tor 目录，下次重启后会自动绑定到/var/lib/tor：

```
user@appVM:~$ sudo mkdir -p /rw/bind-dirs/var/lib/tor
```

注意/rw/bind-dirs/var/lib/tor 和/var/lib/tor 目录的权限一样。

如果还要持久化其他目录（如/etc/tor/torrc），请把 binds+=('/etc/tor/torrc')追加到/rw/config/qubes-bind-dirs.d/50_user.conf 末尾，其他操作类似（例如要不要创建目录/rw/bind-dirs/etc/tor/torrc）。

如果一个模板需要产生众多功能一样的虚拟机，且都要持久化某个目录，这时可直接针对模板操作，即把文件 50_user.conf 放在模板的/etc/qubes-bind-dirs.d 目录中，并确保被持久化的目录已经存在，这样全部基于此模板的虚拟机下次启动后都会持久化这个目录。

还有一种情况，只是某个应用软件需要持久化目录，那请把配置文件放置在/usr/lib/qubes-bind-dirs.d 目录中，安装某些软件时就会产生这样的配置文件。

这三个目录的优先级从高到低是：/rw/config/qubes-bind-dirs.d 》/etc/qubes-bind-dirs.d 》/usr/lib/qubes-bind-dirs.d。

一切的机理请查看/usr/lib/qubes/init/bind-dirs.sh，启动虚拟机时会执行它。

安装某个软件时会自动做的持久化配置文件放在/usr/lib/qubes-bind-dirs.d/目录中，比如 40_qubes-whonix.conf 持久化/var/lib/tor，但是我们有时想取消这样的持久化，这时可以这么做：在/rw/config/qubes-bind-dirs.d/50_user.conf 中追加一行 binds=("${binds[@]/'/var/lib/tor'}")，不建议直接修改/usr/lib/qubes-bind-dirs.d/40_qubes-whonix.conf，因为下次升级软件时可能会覆盖。

有时我们想创建最小状态的虚拟机——只有极少数的数据被持久化，这时我们可以开启虚拟机的 custom-persist 特征，以致：

/rw/config/rc.local 不再被执行了；/rw/config/qubes-firewall-user-script 被忽略；/rw/config/suspend-module-blacklist 被忽略；/rw/config/qubes-bind-dirs.d 目录被忽略；/home 和 /user/local 目录不再被持久化。

具体这样操作：

```
user@dom0:~$ qvm-service -e my-app-vm custom-persist                    //开启 my-app-vm 虚拟机的 custom-persist 特征
```

执行命令 qvm-features my-app-vm custom-persist.persist_dir /to/the/persistent_dir 持久化目录/to/the/persistent_dir，如：

```
user@dom0:~$ qvm-features my-app-vm custom-persist.torrc /etc/tor/torrc
user@dom0:~$ qvm-features my-app-vm custom-persist.home /home
```

可能在绑定目录时需要自动在/rw/bind-dirs 中创建目录，此时可执行命令 qvm-features my-app-vm custom-persist.path <dir|file>:<目录或文件的主人>:<目录或文件的组>:<用数字表示的权限>:/to/the/path，例如：

```
user@dom0:~$ qvm-features my-app-vm custom-persist.downloads dir:user:user:0755:/home/user/Downloads
user@dom0:~$ qvm-features my-app-vm custom-persist.my_ssh_known_hosts_file
file:user:user:0600:/home/user/.ssh/known_hosts
```

9.4.4 qrexec 策略系统

qrexec 是一种允许从一台虚拟机去操纵另一台虚拟机的 RPC 机制，qrexec 策略就是权限控制——允许谁在哪里能做什么。保存策略的文件以.policy 为扩展名，保存在/etc/qubes/policy.d 或/run/qubes/policy.d/目录下，后者中的策略文件重启之后会被删除。/etc/qubes/policy.d 目录：

```
/etc/qubes/policy.d/
├── 35-compat.policy
├── 90-admin-default.policy
├── 90-default.policy
├── include
│   ├── admin-local-ro
│   ├── admin-local-rwx
│   ├── admin-global-ro
│   ├── admin-global-rwx
(和一些用户自定义的策略文件)
```

文件名开头的数字同时表示优先级，数字越小优先级越高，高优先级的策略会覆盖低优先级的策略，如果你想定义自己的策略，不要去修改现有的策略文件，而是单独创建一个更高优先级的文件，如 30-cutom.policy。90-default.policy 文件的部分内容如下：

```
policy.RegisterArgument *          @anyvm         dom0       deny
qubes.ConnectTCP        *         @anyvm         @anyvm     deny

qubes.FeaturesRequest   *          @anyvm           dom0    allow

qubes.NotifyTools       *         @anyvm         dom0       allow

qubes.Filecopy          *         @anyvm         @anyvm     ask

qubes.GetDate           *         @tag:anon-vm    @anyvm     deny
qubes.GetDate           *         @anyvm         @anyvm     allow target=dom0
```

每一行就是一条策略，一条策略的语法如下：

```
qrexec.Service|* +ARGUMENT|* SRCQUBE DSTQUBE {allow|deny|ask} [PARAM=VALUE [PARAM=VALUE ...]]
```

其中：

Service 是 Qubes OS RPC 服务；

ARGUMENT 为调用服务时的参数，可为+（空参数）、*（任何参数）、+value（具体参数，如+firefox）；

SRCQUBE 和 DSTQUBE 分别为源虚拟机和目标虚拟机，它们具体的值可以是虚拟机名字（如 sys-firewall）、@adminvm（管理虚拟机，即 dom0）、@anyvm（任意虚拟机）、@default（默认目标虚拟机，结合 target=属性使用。诸如 qvm-copy 命令没有指明目标时旧使用这里定义的默认目标虚拟机）、@dispvm（从源虚拟机的 disposable 模板启动一个新的一次性虚拟机作为目标）、@dispvm:QUBENAME（从 QUBENAME 模板启动一次性虚拟机）、@dispvm:@tag:TAG（带有 TAG 标签的任何一次性虚拟机）、@tag:TAG（带有 TAG 标签的任何虚拟机）、@type:TYPE（类型为 TYPE 的所有虚拟机，类型有：DispVM、AppVM 和 TemplateVM）；

动作 allow 表示允许，deny 表示拒绝，ask 会弹出一个确认窗口，需要用户确认。

看一个例子：

```
qubes.Filecopy * work @default ask default_target=vault
```

此后如果在 work 虚拟机中执行 qvm-copy file.txt，会弹出一个对话框选择目标虚拟机（默认值是 vault）。

每次发起 Qubes RPC 调用，都会评估策略文件（文件名从小到大排序处理）去寻找匹配的规则，Service、ARGUMENT、

SRCQUBE 和 DSTQUBE 都匹配表示命中一条规则，并忽略剩余的规则。

比如禁用某台虚拟机（如 appVM1）的剪贴板功能：

```
qubes.ClipboardPaste * appVM1 @anyvm deny
qubes.ClipboardPaste * @anyvm appVM1 deny
```

Qubes OS 也提供了一个图形化的策略编辑器，点击 Applications 》Settings 》Qubes Tools 》Qubes OS Policy Editor，打开的界面参考图 9-28 所示。

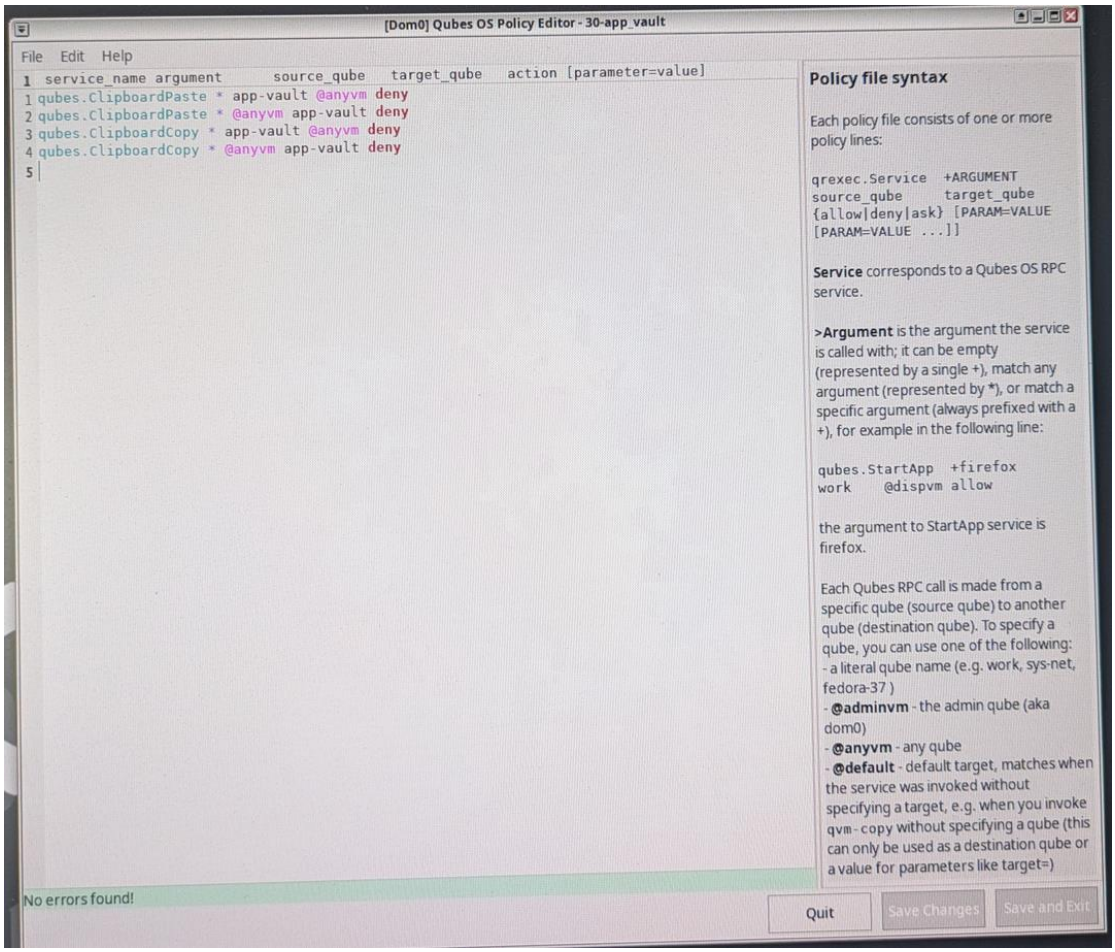


图 9-28 策略编辑器

9.4.5 常见命令

9.4.6 备份和恢复

备份：Applications 》System Tools 》Backup Qubes，然后参考向导走即可。

恢复：Applications 》System Tools 》Restore Backup，然后根据向导走即可。

克隆整个 U 盘：在 Linux 上很简单，采用 dd 命令即可，确保 U 盘未被挂载的情况下，使用命令：

```
lsblk
dd if=/dev/sdb of=/dev/sdc bs=4M status=progress
sync
eject /dev/sdc
```

或把 u 盘制作成文件：

```
dd if=/dev/sdb of=qubes-os.img bs=4M status=progress
```

对于 windows 可使用 DiskGenius，从 https://www.eassos.com/下载并安装它，然后启动后点击 Tools 》Clone Disk(D)，出现如图 9-29 所示的界面。

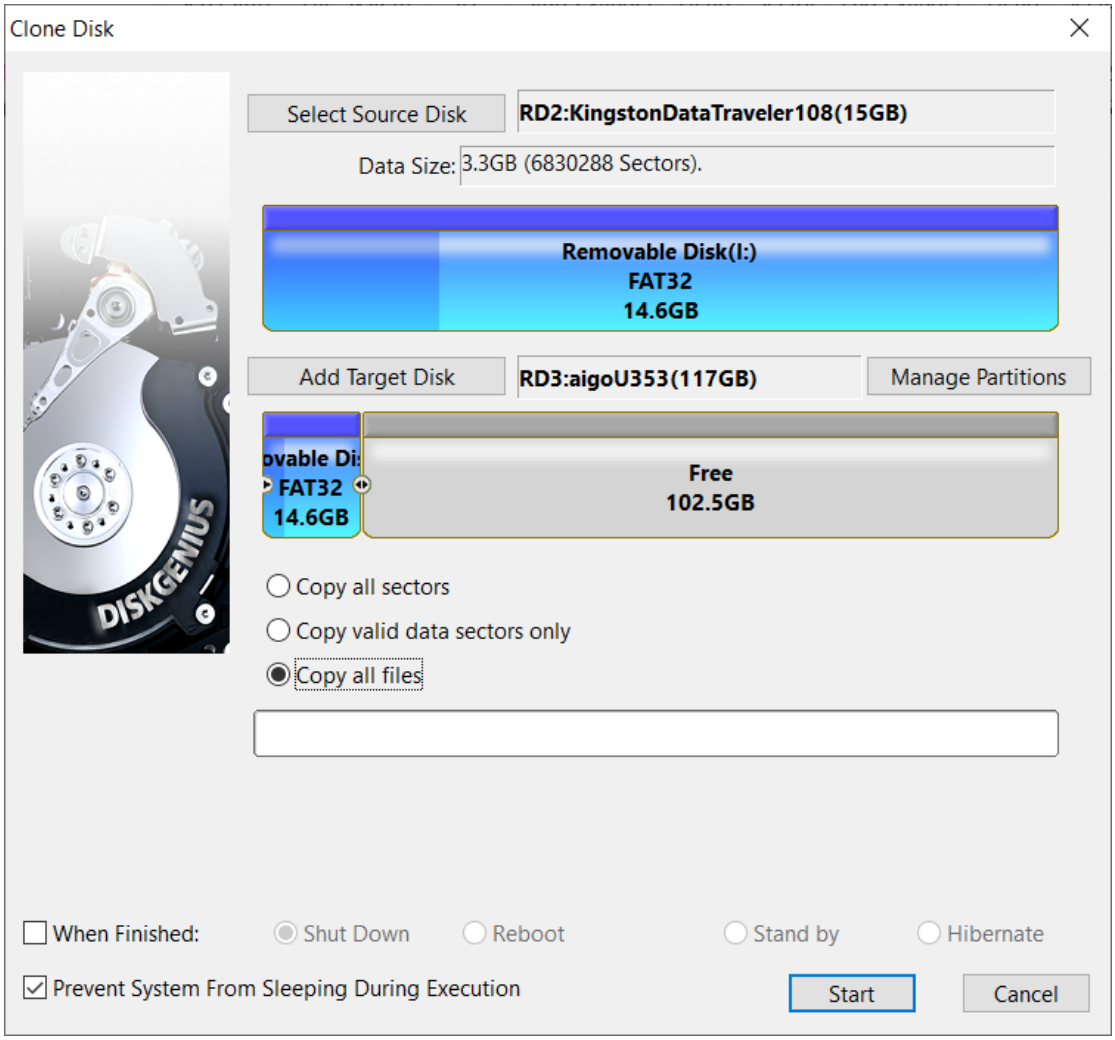


图 9-29 克隆 U 盘

利用 DiskGenius 也可以把 U 盘制作成文件，点击菜单 Disk 》Backup Disk to Image File 出现如图 9-30 所示的界面。

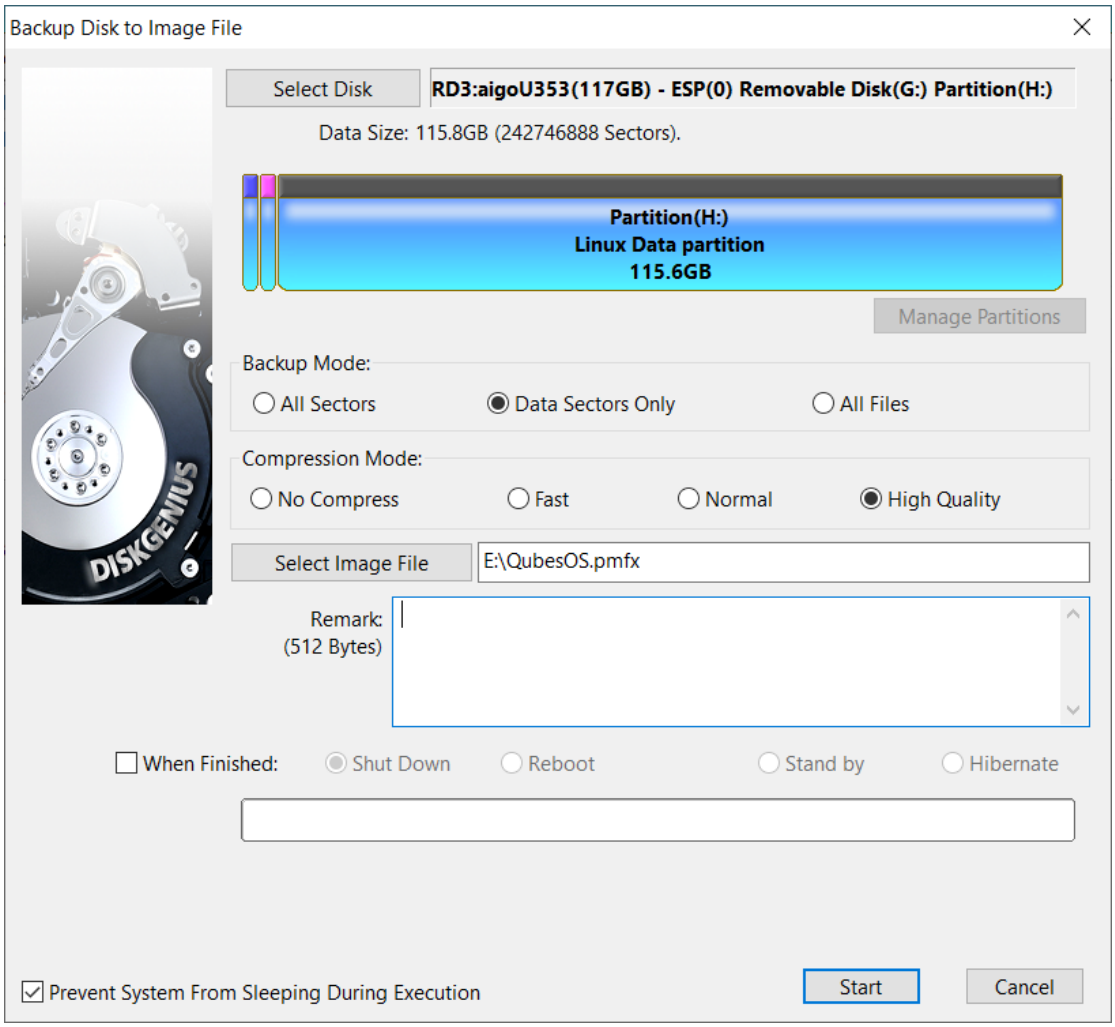


图 9-30 从 U 盘到文件

9.5 钱包与交易

凡是涉及私钥的操作都在 app-vault 中进行，比如新建钱包、给私钥加密、从私钥导出公钥和地址、交易签名等。要连网查询时才在 anon-whonix 上操作，包括买卖比特币时与对方交流沟通。与比特币交易无关的普通资料上网查询可以在 app-online 上，因为走 Tor 网速度慢、公共成本高，包括使用 jami 即时通讯工具。

关于钱包和交易的具体操作请参考第 6 章的相关小节，这里不再赘述。

附录 A：手工生成原始比特币交易(P2PKH)

本附录中，我们直接通过手工方法来生成原始的比特币交易串，来进一笔理解加密币交易数据的底层结构。

A.1 一进二出的交易

所谓的“一进二出”交易就是包含一个输入两个输出——花费一个 UTXO，转币到两个地址。

参考 https://medium.com/@ottosch/manually-creating-and-signing-a-bitcoin-transaction-87fbbfe46032

已经签名的原始交易码格式如图 A-1 所示：

	项目	值	例：1进1出
	4个字节的版本号	02000000	02000000
	1个字节的输入个数	xx	01
输入1	32个字节的交易ID	xxxxx……x	xxxxx……x
	4个字节的输出索引	xxxxxxxx	xxxxxxxx
	1个字节的解锁脚本长度	xx	xx
	可变长度的解锁脚本	<解锁脚本>	<解锁脚本>
	4个字节的顺序号	ffffffff	ffffffff
输入2	32个字节的交易ID	xxxxx……x	
	4个字节的输出索引	xxxxxxxx	
	1个字节的解锁脚本长度	xx	
	可变长度的解锁脚本	<解锁脚本>	
	4个字节的顺序号	ffffffff	
输入n	……	……	
	1个字节的输出个数	xx	01
输出1	8个字节的发生额(单位聪)	xxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxx
	1个字节的加锁脚本长度	xx	xx
	可变长度的锁定脚本	<锁定脚本>	<锁定脚本>
输出2	8个字节的发生额(单位聪)	xxxxxxxxxxxxxxxx	
	1个字节的加锁脚本长度	xx	
	可变长度的锁定脚本	<锁定脚本>	
输出m	……	……	
	4个字节的锁定时间	00000000	00000000

例子：1进1出的原始交易码：

020000000**1**xxxxx……xxxxxxxxxx**<解锁脚本>**ffffffff**01**xxxxxxxxxxxxxxxxxx**<锁定脚本>**00000000

图 A-1 已经签名的原始交易码格式

A.1.1 准备两个比特币测试地址

在 https://iancoleman.io/bip39/上申请两个比特币测试钱包，参考图 A-2 操作。

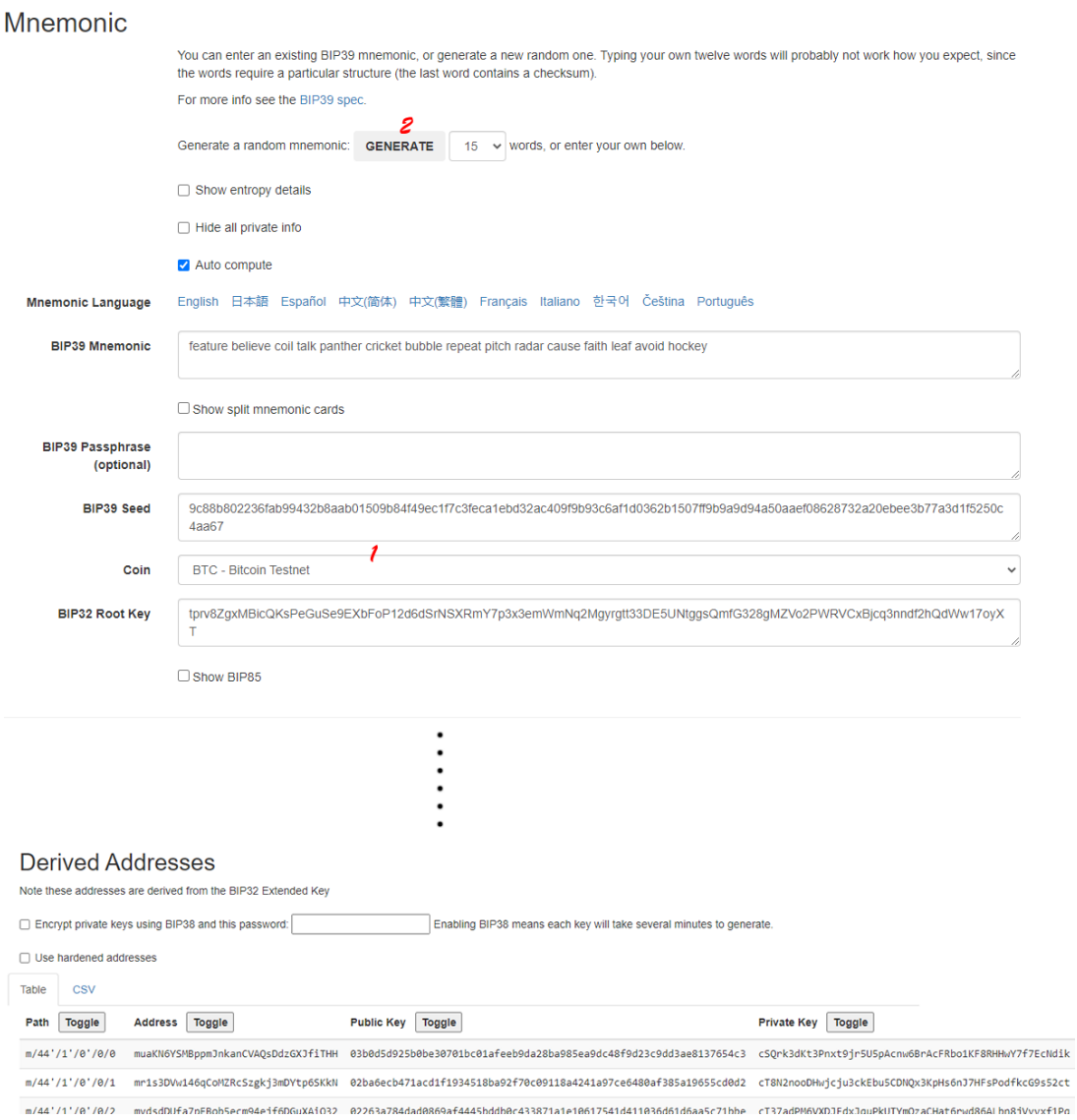


图 A-2 申请比特币钱包

再去 https://testnet-faucet.mempool.co/、https://bitcoinfaucet.uo1.net/或 https://coinfaucet.eu/en/btc-testnet/上充币。

下面构造从 mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN 转 0.0005 比特币到地址

muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH 的原始交易，这是 1 进 2 出的交易（包括找零）。交易格式为：

```
02000000 01 xxxxxx……xxxxxxxxxx xx <解锁脚本> ffffffff 02 xxxxxxxxxxxx xx <锁定脚本> xxxxxxxxxxxxxxxxxx <锁定脚本> 00000000
```

下面就是填充里面的 x。

A.1.2 确定 32 个字节的前交易 ID：xxxxx……x

从 https://sochain.com/api/v2/get_tx_unspent/BTCTEST/mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN 查看 UTXO，如图 A-3 所示。


```
{
  "status": "success",
  "data": {
    "network": "BTCTEST",
    "address": "mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN",
    "txs": [
      {
        "txid": "f6dfa68c702f063730e67ae1dc20f7d526a7888ddc3901adc80a7623c6815981",
        "output_no": 0,
        "script_asm": "OP_DUP OP_HASH160 7328eef2a4c9dde0bc4527f10c10f8caca2851a7 OP_EQUALVERIFY OP_CHECKSIG",
        "script_hex": "76a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac",
        "value": "0.00100000",
        "confirmations": 2,
        "time": 1656733342
      }
    ]
  }
}
```


交易ID ← "txid"　　输出索引 ← "output_no"　　锁住脚本 ← "script_hex"

图 A-3 转出钱包的一个 utxo

交易 ID “f6dfa68c702f063730e67ae1dc20f7d526a7888ddc3901adc80a7623c6815981”的小端存储为“815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff6”，可使用下面的 JavaScript 语句实现：

```
"f6dfa68c702f063730e67ae1dc20f7d526a7888ddc3901adc80a7623c6815981".replace(/(.{2})/g, "$1 ").split(" ").reverse().join("")
```

也可以从采用下面的 linux 命令转成小端存储：

```
$ echo f6dfa68c702f063730e67ae1dc20f7d526a7888ddc3901adc80a7623c6815981 | tac -rs ..; echo 815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff6
```

因此原始交易变成：

02000000**01**815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff6xxxxxxxxx<解锁脚本>ffffffff……

A.1.3 确定 4 个字节的输出索引：xxxxxxxx

上图中显示输出索引为 0，用四个字节表示就是 8 个 0：00000000：

02000000**01**815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff6**00000000**xx<解锁脚本>ffffffff……

输入脚本长度 xx 和<输入脚本>暂时不能确定，不妨输入脚本长度先设为 0，即 00

02000000**01**815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff6**00000000****00**<解锁脚本>ffffffff**02**xxxxxxxxxxxxxxxxxx<锁定脚本>xxxxxxxxxxxxxxxxxx<锁定脚本>00000000

A.1.4 确定第 1 个输出中的 8 个字节的发生额：xxxxxxxxxxxxxxxx

我们要转 0.0005 个比特币，换算成单位聪就是 50000，再转换为十六进制就是 c350，扩展为八个字节就是 000000000000c350，小端存储为 50c3000000000000，这一系列转换可以采用下面的 JS 语句完成：

```
("0000000000000000"+(50000).toString(16)).substr(-16).replace(/(.{2})/g, "$1 ").split(" ").reverse().join("")
```

(50000).toString(16)把十进制的 50000 转换为 16 进制表示的字符串（即 c350）。

因此交易变为：

```
0200000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff60000000000<解锁脚本
>ffffffff0250c3000000000000xx<锁定脚本>xxxxxxxxxxxxxxxxxx<锁定脚本>00000000
```

A.1.5 确定第 1 个输出中的输出脚本和长度：xx<输出脚本>

对于 P2PKH 地址类型，输出脚本的常用格式为：

```
OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

https://en.bitcoin.it/wiki/Script 列出了各种操作符对应的十六进制数，即 OP_DUP=76，OP_HASH160=a9，OP_EQUALVERIFY=88，OP_CHECKSIG=ac。而<PubKeyHash>是 base58 格式的比特币地址的反 base58 解码后去掉开头 1 个字节和末尾 4 个字节所剩下的 20 个字节，即 ripemd160，采用下面的 JS 语句可实现：

```
bitcore=require("bitcore-lib");
bitcore.encoding.Base58Check.decode("muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH").toString('hex').substr(2);
"9a3498d201c9a1498cefccc51efb56e0a902ed4f"
```

muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH 就是收币地址（对方地址）。

也可以采用下面的 Python 语句可完成：

```
$ apt install python3-base58
$ python3
>>> from base58 import b58decode_check
>>> b58decode_check('muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH')[1:].hex()
'9a3498d201c9a1498cefccc51efb56e0a902ed4f'
```

于是得到输出脚本为 76 a9 14 9a3498d201c9a1498cefccc51efb56e0a902ed4f 88 ac，输出脚本长度（不包含空格）为 25 个字节，用十六斤制表示就是 19。最后得到：

```
1976a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88ac
```

于是交易变成：

```
0200000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff60000000000<解锁脚本
>ffffffff0250c3000000000000 1976a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88ac xxxxxxxx
xxxxxxxxxx<锁定脚本>00000000
```

采用类似的方法确定第 2 个输出，这是找零输出（返回到转出账号），发生额等于 0.001-0.0005-0.00005(交易费)，即 0.00045 比特币=45000 聪。所以 8 个字节的发生额为 c8af000000000000，输出脚本为 76a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac，输出脚本长度为 0x19。最终的交易变为：

02000000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff60000000000<解锁脚本>ffffffff0250c3000000000000001976a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acc8af000000000000019976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000000

A.1.6 对交易签名

为了采用 openssl 签名，转出钱包的私钥需要加上前缀 302e0201010420 和后缀 a00706052b8104000a。但首先要把 WIF 压缩格式私钥转化为 HEX 格式，可在 www.bitaddress.org 的“Wallet Details”中转化（注意：如果是测试钱包，请在网址后追加 ?testnet=true ）。比如我的测试私钥转化为 HEX 格式为 A579999EC07944018C3B85ECAA85F1D73C09C3C9129CD71F16B023C1374ce664

```
$ echo "302e0201010420 A579999EC07944018C3B85ECAA85F1D73C09C3C9129CD71F16B023C1374ce664
a00706052b8104000a" | xxd -p -r > priv.hex
$ openssl ec -inform d < priv.hex > priv.pem
```

这样就得到符合 openssl 签名格式的私钥 priv.pem。

1. 对第 1 个输入签名（如果有多个输入，每个输入需要单独签名）

在交易后面添加 01000000（表明签名类型为 SIGHASH_ALL），即：

02000000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff60000000000<解锁脚本>ffffffff0250c3000000000000001976a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acc8af000000000000019976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac0000000001000000

用 UTXO 的输出脚本替换交易中的<输入脚本>，长度 00 更换为 x19，即：

02000000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff60000000019**76a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac**ffffffff0250c3000000000000001976a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acc8af000000000000019976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac0000000001000000

对交易做哈希运算：

```
$ echo -n
020000000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff6000000001976a9147328eef2a4c9dde0
bc4527f10c10f8caca2851a788acffffffff0250c3000000000000001976a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88ac
c8af000000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac0000000001000000 | xxd -p -r |
sha256sum -b | xxd -p -r | sha256sum -b | xxd -p -r > msg
```

哈希结果保存在 msg 文件中。正式签名：

```
$ openssl pkeyutl -inkey priv.pem -sign -in msg -pkeyopt digest:sha256 | xxd -p -c 256
30440220395fd4517b7f2fa4a910183d9973348702acda2cb7d1551a895ba17ce6a318ef022041818626f12c116bd1ae98dad79895
b944cd619faf68a73fef04ce7e8504545e
```

解析：30-指明签名的结构，44-总长度，02-整数类型，20-<r>长度，红色内容就是<r>，后面的 20-<s>长度，蓝色内容就是<s>。

要求：r 和 s 为正整数，所以最高位字节小于 0x7f，且 s 不能大于 n/2，n 就是椭圆曲线的阶，即 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141，验证是否满足条件：

```
$ python3
>>> s = 0x41818626f12c116bd1ae98dad79895b944cd619faf68a73fef04ce7e8504545e
>>> n = 0xfffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
>>> s < n/2
True
```

```
>>>
```

如果不满足，就尝试多次签名直到符合要求。最后追加 0x01 表明是签名哈希：

```
30440220395fd4517b7f2fa4a910183d9973348702acda2cb7d1551a895ba17ce6a318ef022041818626f12c11
6bd1ae98dad79895b944cd619faf68a73fef04ce7e8504545e01
```

最终产生的输入脚本（格式为<sig> <public key>）：



其中 6a 为输入脚本的总长度，47 为签名长度，21 为公钥长度。

最后把输入脚本和长度替换交易中的 00<输入脚本>，即从

```
0200000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff60000000000<解锁脚本
>ffffffff0250c3000000000001976a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acc8af000000000
00019 76a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000000
```

替换后为：

```
0200000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff600000000 6a47304402
20395fd4517b7f2fa4a910183d9973348702acda2cb7d1551a895ba17ce6a318ef022041818626f12c116b
d1ae98dad79895b944cd619faf68a73fef04ce7e8504545e012102ba6ecb471acd1f1934518ba92f70c091
18a4241a97ce6480af385a19655cd0d2ffffffff0250c3000000000001976a9149a3498d201c9a1498cefccc5
1efb56e0a902ed4f88acc8af0000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000
000
```

即我们最后得到的签名交易原始数据为：

```
0200000001815981c623760ac8ad0139dc8d88a726d5f720dce17ae63037062f708ca6dff6000000006a4730440220395fd4517b7f2
fa4a910183d9973348702acda2cb7d1551a895ba17ce6a318ef022041818626f12c116bd1ae98dad79895b944cd619faf68a73fef04c
e7e8504545e012102ba6ecb471acd1f1934518ba92f70c09118a4241a97ce6480af385a19655cd0d2ffffffff0250c300000000000019
76a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acc8af0000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2
851a788ac00000000
```

先试试能否在 https://live.blockcypher.com/btc/decodetx/网站上对交易原始数据进行解码（此网站需要爬梯出去才行），参见图 A-4。

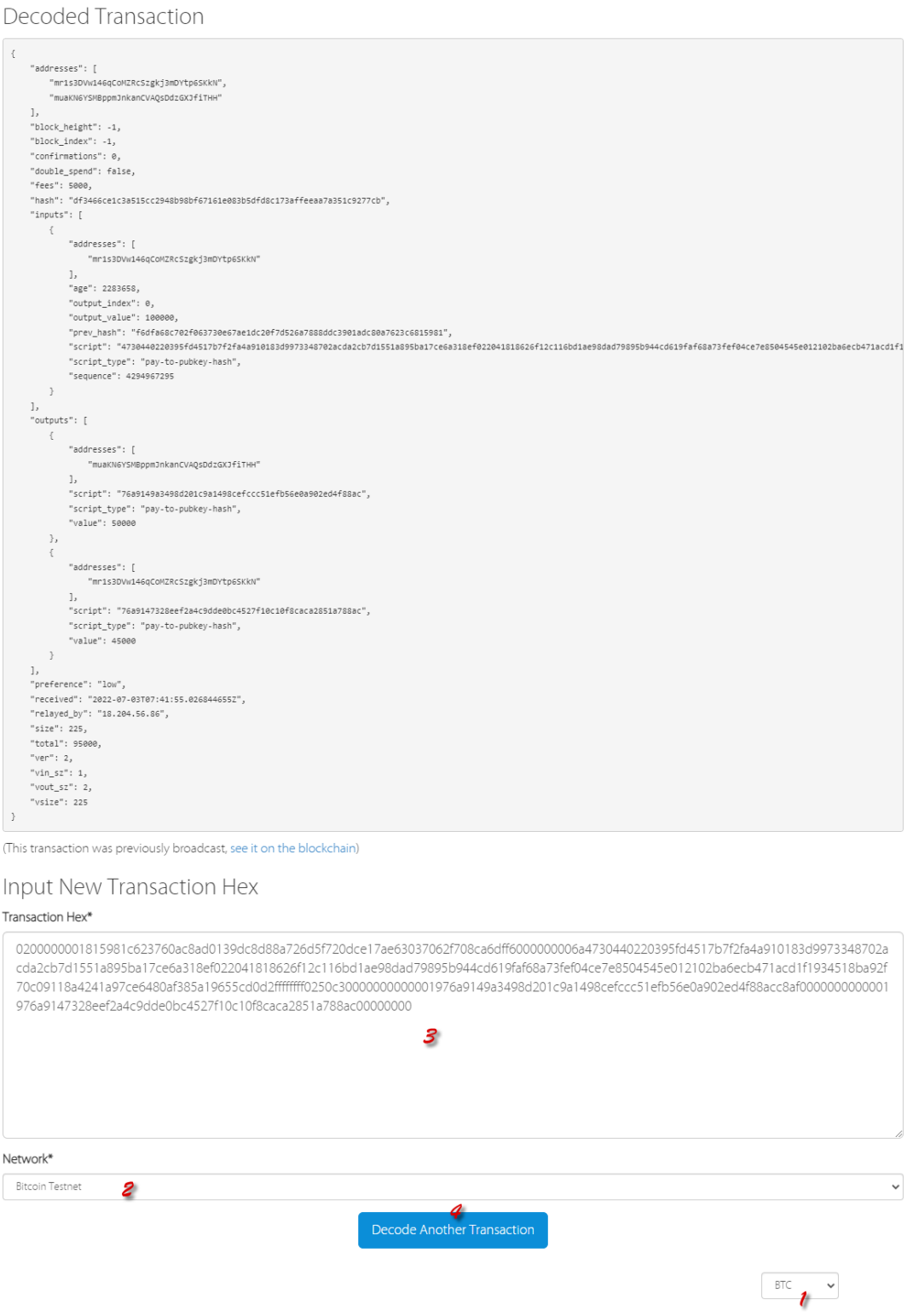


图 A-4 先解码并验证

解码成功。接下来就可以正式在 https://live.blockcypher.com/btc/pushtx/上发布交易了，具体参见图 A-5 所示。

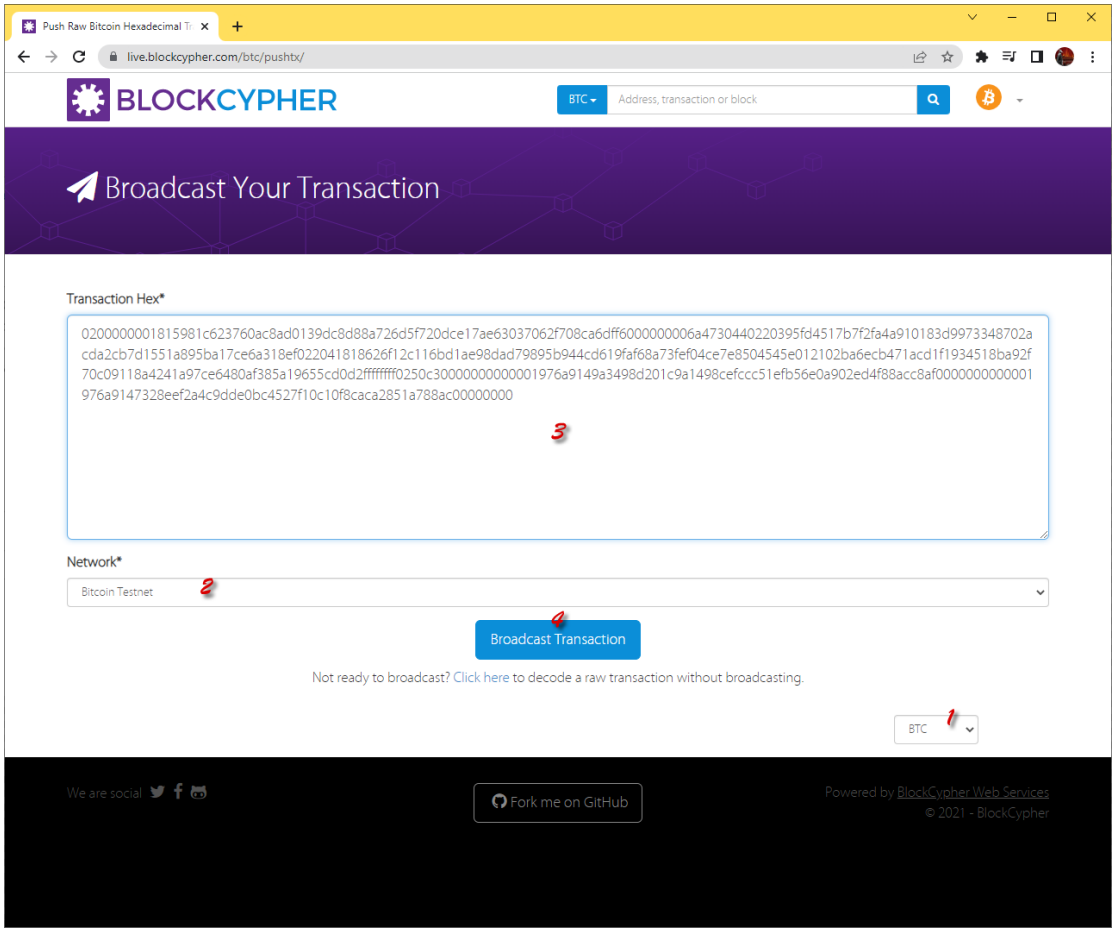


图 A-5 发布交易

发布之后大概一分钟后就可以查看是否到账，直接打开 https://live.blockcypher.com/btc-testnet/address/muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH，发现已经到账了，如图 A-6 所示。

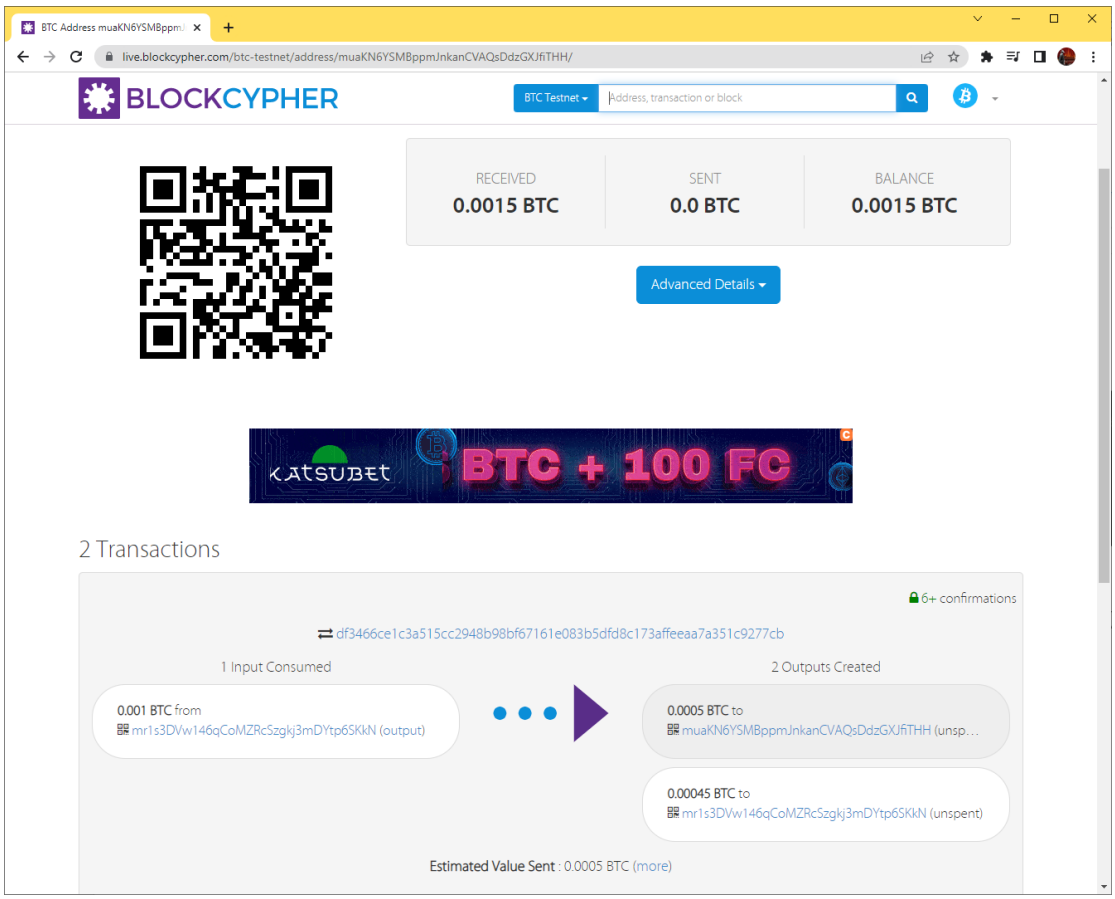


图 A-6 成功发布

A.2 二进一出的交易

签名后的原始交易码格式：

02000000**02**xxxxx……xxxxxxxxxx**<解锁脚本>**ffffffff**xxxxx**……xxxxxxxxxx**<解锁脚本>**ffffffff**01**xxxxxxxxxxxxxxxxxx**<锁定脚本>**00000000

这次我们把 muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH 地址中的全部比特币转给 mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN 钱包。先查看转出钱包上的全部 UTXO，如 A-7。

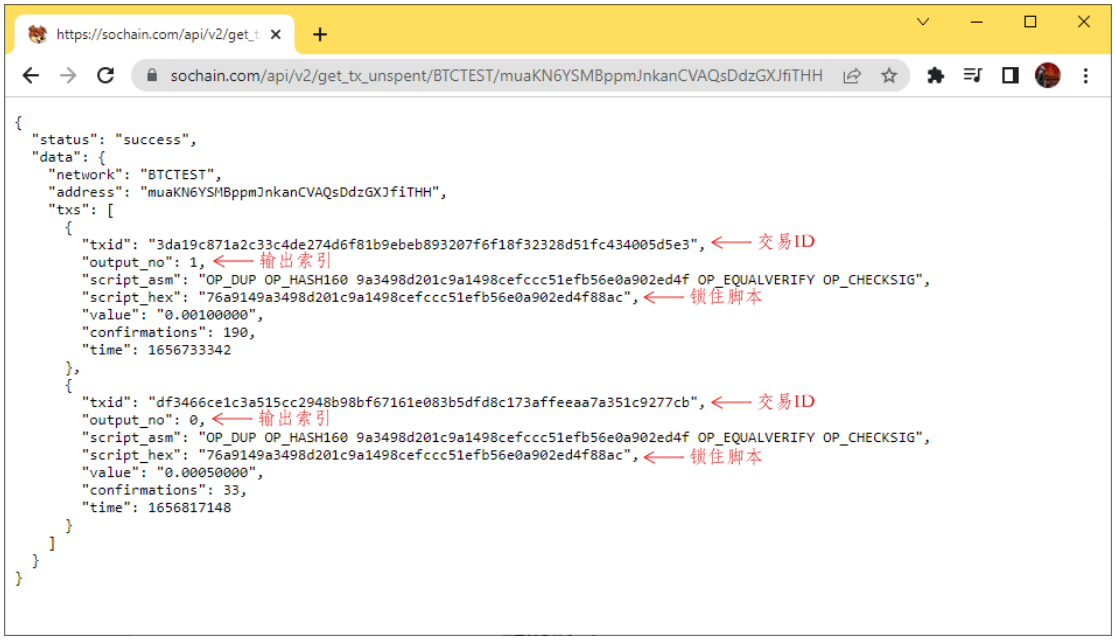


图 A-7 先查出 utxo

发现存在两个 UTXO。

签名前的交易如下：

```
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d0100000000<解锁脚本
>ffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df0000000000<解锁脚本
>ffffffff01683602000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000000
```

对于 2 进的签名要特别注意，要分别对每一个输入进行签名。

一、先对第一个输入签名

把第一个 UTXO 的 script_hex 替换交易中的第一个“<输入脚本>”为 76a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88ac，长度替换为 19。删除第二个“<输入脚本>”（保留其长度 00）：

```
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d0100000019
76a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df0000000000ffffffff01683602000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000000
```

在交易后追加 01000000：

```
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d0100000019
76a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df0000000000ffffffff01683602000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac0000000001000000
```

首先把转出钱包的私钥转换为与 openssl 签名兼容的私钥：

```
$ echo "302e0201010420 901F8E7D3E00D8AD969D115583F769718F460AB0E7B1D4F5735C09B1E4215F6C
a00706052b8104000a" | xxd -p -r > priv.hex
$ openssl ec -inform d < priv.hex > priv.pem
```

对交易做哈希运算：

```
$ echo
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d010000001976a9149a3498d201c9a149
8cefccc51efb56e0a902ed4f88acffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df000000
0000ffffffff0168360200000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac0000000001000000 | xxd -
p -r | sha256sum -b | xxd -p -r | sha256sum -b | xxd -p -r > msg
```

哈希结果保存在 msg 文件中。正式签名：

```
$ openssl pkeyutl -inkey priv.pem -sign -in msg -pkeyopt digest:sha256 | xxd -p -c 256
```

得到签名：

3046022100b124a74545f68625cd9b5300c315b4cafb95cba5ab9578d737ab05ba49a21e2f022100b3084ab622589bf1bd8558f0ab1d582bf705989dcac9664373a357159875b0d4

验证 s 是否满足要求：

```
$ python3
>>> s = 0x00b3084ab622589bf1bd8558f0ab1d582bf705989dcac9664373a357159875b0d4
>>> n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
>>> s < n/2
False
>>>
```

这样的 s 不行，重新做一次签名：

```
$ openssl pkeyutl -inkey priv.pem -sign -in msg -pkeyopt digest:sha256 | xxd -p -c 256
```

得到签名：

3044022077031eba150da785e52212d5b084adf2dc2c88747d41f9b1d92837c95b30b886022061627 6fa4c6d9abc23a732344ce51b1e6992d0f8b0b1fe3b0641b80ae367f466

再次验证：

```
$ python3
>>> s = 0x616276fa4c6d9abc23a732344ce51b1e6992d0f8b0b1fe3b0641b80ae367f466
>>> n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
>>> s < n/2
True
>>>
```

在签名后追加 0x01（表明是签名）：

3044022077031eba150da785e52212d5b084adf2dc2c88747d41f9b1d92837c95b30b886022061627 6fa4c6d9abc23a732344ce51b1e6992d0f8b0b1fe3b0641b80ae367f46601

最终产生输入脚本（即<输入脚本总长度><签名长度><签名><公钥长度><公钥>）：



签名的长度为 47（十六进制），公钥的长度是 21，输入脚本总长度为 6a（即 47+21+2），因为 47 和 21 本身占据 2 个字节，所以要加上 2。

即：

6a473044022077031eba150da785e52212d5b084adf2dc2c88747d41f9b1d92837c95b30b8860220616276fa4c6d9abc23a732344ce51b1e6992d0f8b0b1fe3b0641b80ae367f466012103b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3

公钥的大小写字母不敏感，但是为了统一效果，建议转化为小写，方法如下：

```
btceth@btceth:~$ python3
>>> a = "03B0D5D925B0BE30701BC01AFEEB9DA28BA985EA9DC48F9D23C9DD3AE8137654C3"
>>> print(a.lower());
03b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3
```

得到的签名还不能写入交易中，还得继续对第 2 个输入签名。

```
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d0100000000<解锁脚本
>ffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df0000000000<解锁脚本
>ffffffff0168360200000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000000
```

把第二个 UTXO 的 script_hex（即 76a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88ac）替换交易中的第二个“<输入脚本>”，长度替换为 19，删除第一个“<输入脚本>”（保留其长度 00），再追加 01000000：

```
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d0100000000ffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df0000000019 76a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acffffffff0168360200000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac0000000001000000
```

对交易做哈希运算：

```
$ echo
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d0100000000ffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df000000001976a9149a3498d201c9a1498cefccc51efb56e0a902ed4f88acffffffff0168360200000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac0000000001000000 | xxd -p -r | sha256sum -b | xxd -p -r | sha256sum -b | xxd -p -r > msg
```

哈希结果保存在 msg 文件中。正式签名：

```
$ openssl pkeyutl -inkey priv.pem -sign -in msg -pkeyopt digest:sha256 | xxd -p -c 256
```

304402207215c65755ab28648d6c5f9b0da1a4cd4e7de197d0358e0b49d44f437cc959e90220590668a6d7bf832faa7dcdd16644d671b96c77e3a1712c95bbe6b534ea4a3451

验证 s 是否满足要求：

```
$ python3
>>> s = 0x590668a6d7bf832faa7dcdd16644d671b96c77e3a1712c95bbe6b534ea4a3451
>>> n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
>>> s < n/2
True
>>>
```

在签名后追加 0x01（表明是签名）：

304402207215c65755ab28648d6c5f9b0da1a4cd4e7de197d0358e0b49d44f437cc959e90220590668a6d7bf832faa7dcdd16644d671b96c77e3a1712c95bbe6b534ea4a345101

最终产生输入脚本（<sig> <public key>）：



即：

6a47304402207215c65755ab28648d6c5f9b0da1a4cd4e7de197d0358e0b49d44f437cc959e90220590668a6d7bf832faa7dcdd16644d671b96c77e3a1712c95bbe6b534ea4a3451012103b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3

产生最终的交易，替换两个<输入脚本>和长度，即：

```
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d0100000000<解锁脚本>ffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df0000000000<解锁脚本>ffffffff01683602000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000000
```

替换后为：

```
0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d010000006a47304402207031eba150da785e52212d5b084adf2dc2c88747d41f9b1d92837c95b30b8860220616276fa4c6d9abc23a732344ce51b1e6992d0f8b0b1fe3b0641b80ae367f466012103b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3ffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df000000006a47304402207215c65755ab28648d6c5f9b0da1a4cd4e7de197d0358e0b49d44f437cc959e90220590668a6d7bf832faa7dcdd16644d671b96c77e3a1712c95bbe6b534ea4a3451012103b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3ffffffff01683602000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000000
```

即最终的已签名交易原始码为：

0200000002e3d5054043fc518d32328ff1f6073289ebebb9816f4d27dec4332c1a879ca13d010000006a47304402207031eba150da785e52212d5b084adf2dc2c88747d41f9b1d92837c95b30b8860220616276fa4c6d9abc23a732344ce51b1e6992d0f8b0b1fe3b0641b80ae367f466012103b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3ffffffffcb77921c357aaaeeff3a178cfd5d3b081e1667bf988b94c25c513a1cce6634df000000006a47304402207215c65755ab28648d6c5f9b0da1a4cd4e7de197d0358e0b49d44f437cc959e90220590668a6d7bf832faa7dcdd16644d671b96c77e3a1712c95bbe6b534ea4a3451012103b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3ffffffff01683602000000000001976a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac00000000

首先在 https://live.blockcypher.com/btc/decodetx/解码验证，参见图 A-8。

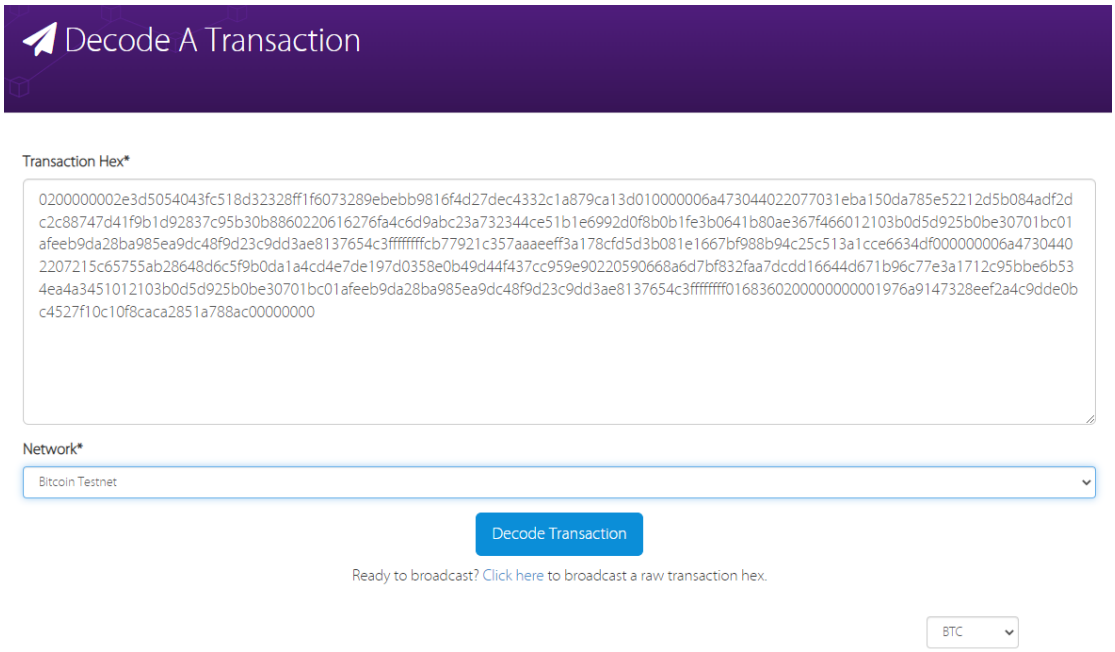


图 A-8 解码交易并验证

解码如图 A-9 所示，发现无措。

Decoded Transaction


```
{
    "addresses": [
        "muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH",
        "mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN"
    ],
    "block_height": -1,
    "block_index": -1,
    "confirmations": 0,
    "double_spend": false,
    "fees": 5000,
    "hash": "a6fd54290082a39c689b97bdb10a812ddbe91cfc4ac90d7fe1951ce1683d7a04",
    "inputs": [
        {
            "addresses": [
                "muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH"
            ],
            "age": 2283658,
            "output_index": 1,
            "output_value": 100000,
            "prev_hash": "3da19c871a2c33c4de274d6f81b9ebeb893207f6f18f32328d51fc434005d5e3",
            "script": "473044022077031eba150da785e52212d5b084adf2dc2c88747d41f9b1d92837c95b30b8860220616276fa4c6d9abc23a732344ce51b1e6992d0f8b0b1fe3b0641b80ae367f466012103b0d5d925b0be307
            "script_type": "pay-to-pubkey-hash",
            "sequence": 4294967295
        },
        {
            "addresses": [
                "muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH"
            ],
            "age": 2283815,
            "output_index": 0,
            "output_value": 50000,
            "prev_hash": "df3466ce1c3a515cc2948b98bf67161e083b5dfd8c173affeeaa7a351c9277cb",
            "script": "473044022072115c65755ab28648d6c5f9b0da1a4cd4e7de197d0358e0b49d44f437cc959e902205906688a6d7bf832faa7dcdd16644d671b96c77e3a1712c95bbe6b534ea4a3451012103b0d5d925b0be307
            "script_type": "pay-to-pubkey-hash",
            "sequence": 4294967295
        }
    ],
    "outputs": [
        {
            "addresses": [
                "mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN"
            ],
            "script": "76a9147328eef2a4c9dde0bc4527f10c10f8caca2851a788ac",
            "script_type": "pay-to-pubkey-hash",
            "value": 145000
        }
    ],
    "preference": "low",
    "received": "2022-07-04T01:33:07.755709095Z",
    "relayed_by": "18.204.56.86",
    "size": 338,
    "total": 145000,
    "ver": 2,
    "vin_sz": 2,
    "vout_sz": 1,
    "vsize": 338
}
```


图 A-9 解码成功

正式在 https://live.blockcypher.com/btc/pushtx/上发布交易，参见图 A-10。

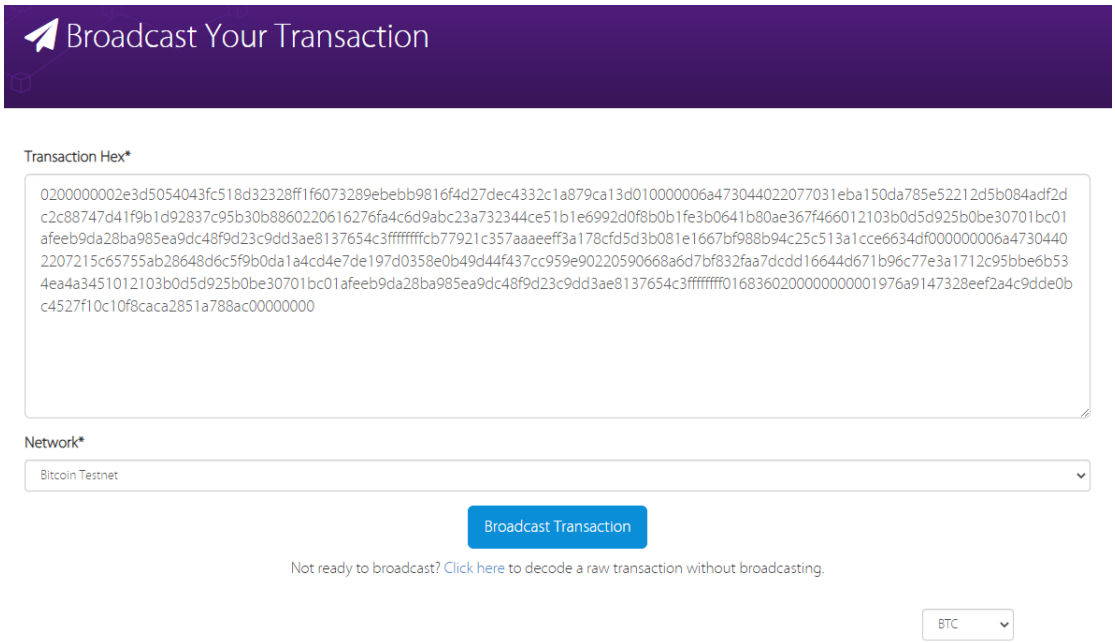


图 A-10 发布交易

发布结果参见图 A-11 所示。

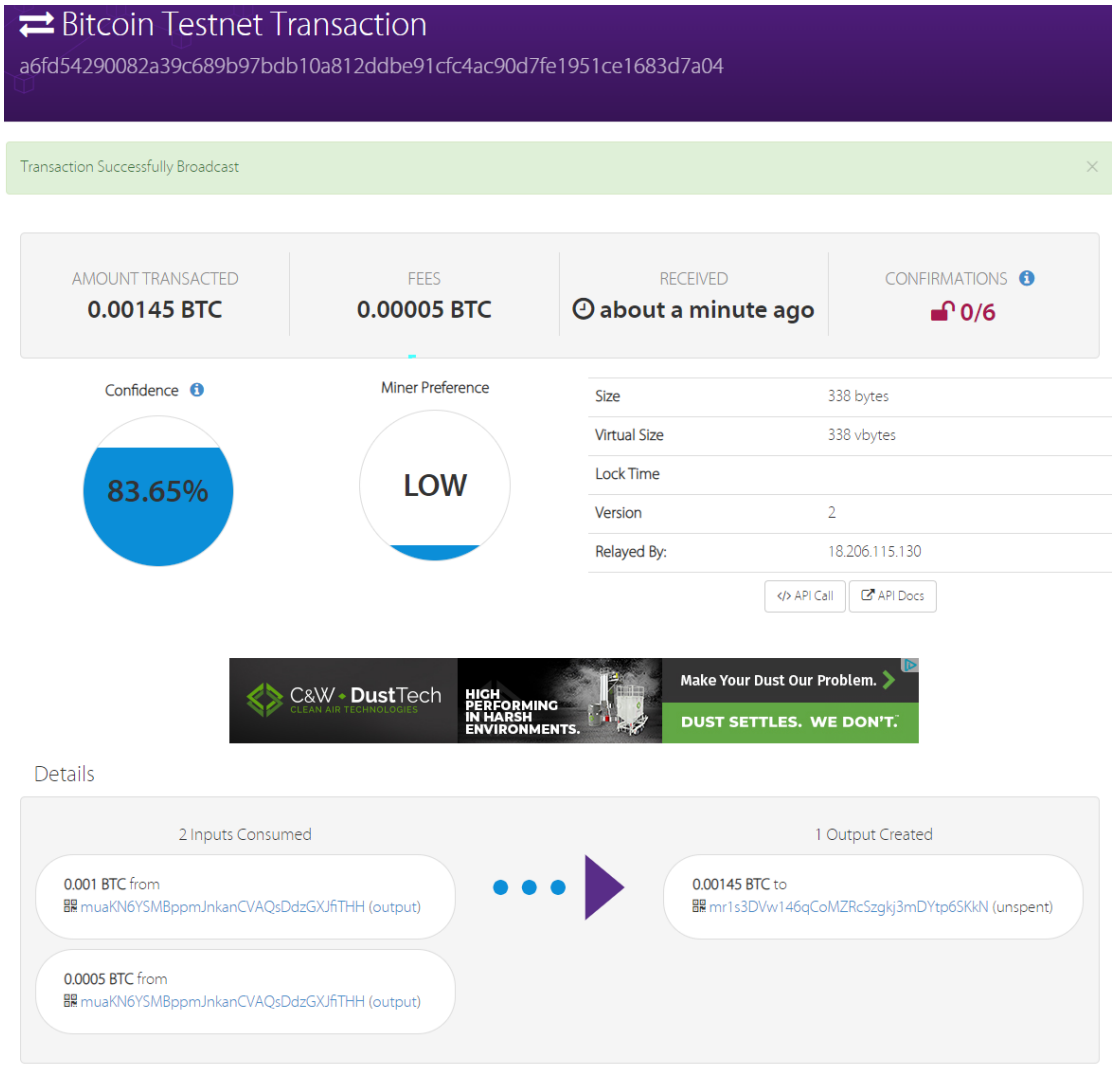


图 A-11 成功发布

最后在 https://live.blockcypher.com/btc-testnet/上查看到账情况，参见图 A-12 所示。

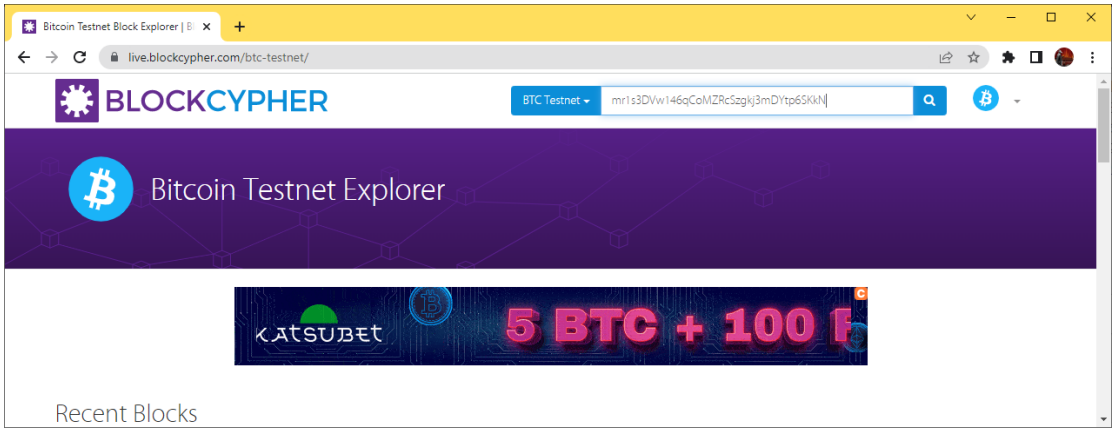


图 A-12 查看到账情况

发现到账了，参见图 A-13 所示。

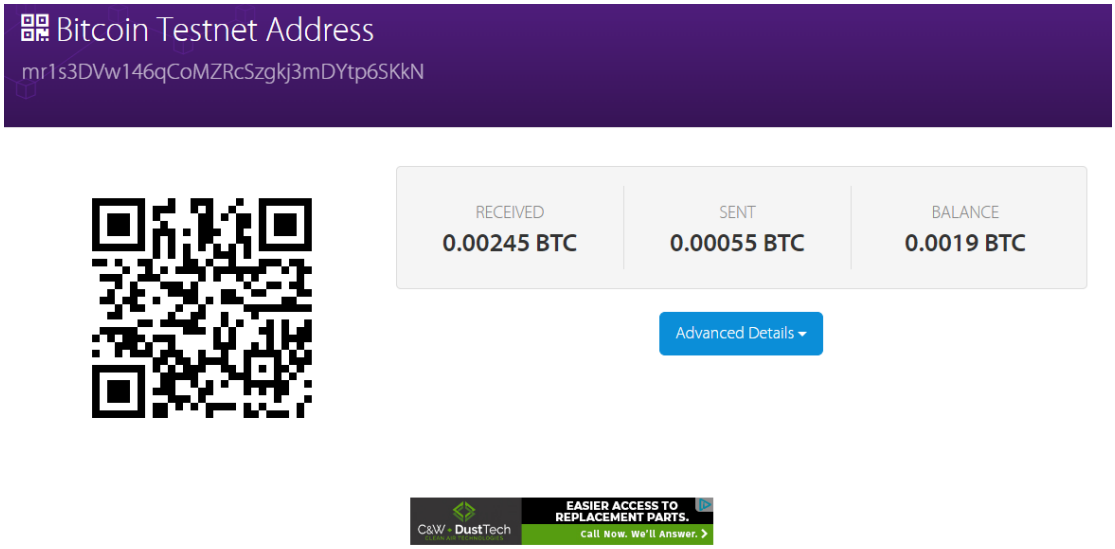

Bitcoin Testnet Address
mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN
RECEIVED
0.00245 BTC
SENT
0.00055 BTC
BALANCE
0.0019 BTC
Advanced Details
C&W + DustTech
EASIER ACCESS TO REPLACEMENT PARTS.
Call Now. We'll Answer.
3 Transactions
1/6 confirmations
a6fd54290082a39c689b97bdb10a812ddbe91cfc4ac90d7fe1951ce1683d7a04
2 Inputs Consumed
0.001 BTC from
muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH (output)
0.0005 BTC from
muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH (output)
1 Output Created
0.00145 BTC to
mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN (unspent)
Value Transacted : 0.00145 BTC

图 A-13 已经到账

附录 B：手工生成原始比特币交易(SegWit)

这次构建隔离见证的交易。

B.1 准备四个地址

1. 一个 BIP44 地址（P2PKH）：

私钥：

cMtwe4bv7dLrxh1mrZjzJRxhztdy1EgXVrXD5ZwqdRoyFMjUiEP8（WIF）

09624c6ac1424fcbc5d1c1b87e7c27ef0b977da5714a6d0b5bd8fe85527a0d66（HEX）

公钥：02699c92c9e62eeb2ec0054a5a66dc8cf89953d42741729e06e45334c2bc316dc4

地址：mgVCkR5QYHM5SA1rSMeVwWKBAxGxFe4aBu

2. 两个 BIP84 地址（原生 P2WPKH）：

私钥：

cPn3BoLZqyK8BavPGiArieLXm7WvoLpHd2vA7f1D3prLEqEhYhri（WIF）

4181c4b9b9e1f83c765a17a4dbb601684e8b607d9046251833602e074db09fe0（HEX）

公钥：02d77cc64a910abf4b6700e75131fa5ec12b8de422b1521dc1d8625559915cbf77

地址：tb1qegvl9nlvrd7yvd34dr3wr4nspgmmhkr0prmj42

私钥：cNunX1Zmx8Jy7eE2fno3HnFcenVybeBX1YcVQ3psn1dE9Hg5gPpW

公钥：035c84c164b5a7a59dc587b39a0599af0023e706fa0bcaec2fcf02ef1d0e69e309

地址：tb1ql303ul9axrgmaa5r72e20h30z4pv5jewdkzfkp

3. 一个 BIP49 地址（P2SH-P2WPKH）：

私钥：

cQkEBQnwvvgtXjS1nqwTDEJMvQwUVWRK5eySokuerHHpuN3vCocj（WIF）

5e69b61ff0aff209253a0aa74c33c841fac1f14bb47fdfc47a2abfcb9d57ed49（HEX）

公钥：028b2da00c8c91ea6b0e78cd4fbd286c231bd88a8082ccfd9f57160eda5be3b695

地址：2N8cUc7tDFhiM11kao82VrX3suTjhbj6JBW

可在 https://www.bitaddress.org/bitaddress.org-v3.3.0-SHA256-dec17c07685e1870960903d8f58090475b25af946fe95a734f88408cef4aa194.html?testnet=true 网站上把 WIF 格式的提要转换为 HEX 格式。

在 https://testnet-faucet.mempool.co/、https://bitcoinfaucet.uo1.net/或 https://coinfaucet.eu/en/btc-testnet/上分别给上面三个非

红色地址充币。

查看三个已充币地址的 UTXO:

https://sochain.com/api/v2/get_tx_unspent/BTCTEST/mgVCkR5QYHM5SA1rSMeVwWKBAxGxFe4aBu：


```
{
  "status": "success",
  "data": {
    "network": "BTCTEST",
    "address": "mgVCkR5QYHM5SA1rSMeVwWKBAxGxFe4aBu",
    "txs": [
      {
        "txid": "759b97690493c4b94e14f9c78d93e661f28a8fdc30f9c71beb794e2e02e59399",
        "output_no": 0,
        "script_asm": "OP_DUP OP_HASH160 0aa35c213b8e3de1bad9325b7696dde7f443f2ca OP_EQUALVERIFY OP_CHECKSIG",
        "script_hex": "76a9140aa35c213b8e3de1bad9325b7696dde7f443f2ca88ac",
        "value": "0.01881889",
        "confirmations": 1,
        "time": 1657068695
      }
    ]
  }
}
```


https://sochain.com/api/v2/get_tx_unspent/BTCTEST/tb1qegvl9nlvrd7yvd34dr3wr4nspgmmhkr0prmj42：


```
{
  "status": "success",
  "data": {
    "network": "BTCTEST",
    "address": "tb1qegvl9nlvrd7yvd34dr3wr4nspgmmhkr0prmj42",
    "txs": [
      {
        "txid": "2a363129c0f9328c043e04d03441005d22c231a60ebaa1b752c9bfcff3c0c2e6",
        "output_no": 0,
        "script_asm": "0 ca19f2cfec1b7c46363568e2e1d6700a37bbd86f",
        "script_hex": "0014ca19f2cfec1b7c46363568e2e1d6700a37bbd86f",
        "value": "0.00010000",
        "confirmations": 1,
        "time": 1657068695
      }
    ]
  }
}
```


https://sochain.com/api/v2/get_tx_unspent/BTCTEST/2N8cUc7tDFhiM11kao82VrX3suTjhbj6JBW：


```
{
  "status": "success",
  "data": {
    "network": "BTCTEST",
    "address": "2N8cUc7tDFhiM11kao82VrX3suTjhbj6JBW",
    "txs": [
      {
        "txid": "55bdf19321b7cf9b3533f8f915b391febb34930eae05a9c2f4a0d3e2fa1d6807",
        "output_no": 0,
        "script_asm": "OP_HASH160 a88ed306eaf882d6e3bb536b1fc33d99a53eedd1 OP_EQUAL",
        "script_hex": "a914a88ed306eaf882d6e3bb536b1fc33d99a53eedd187",
        "value": "0.00010000",
        "confirmations": 1,
        "time": 1657068695
      }
    ]
  }
}
```


由于地址类型不同，所以输出脚本也不同，这样输入脚本也是不同的。接下来我们来一步一步构造三进一出的原始交易，转出地址就是上面的红色字体部分。显然这是一个隔离见证交易，因为按规定只要包含一个非 P2PKH 地址的交易就是隔离见证交易。如图 B-1 所示。

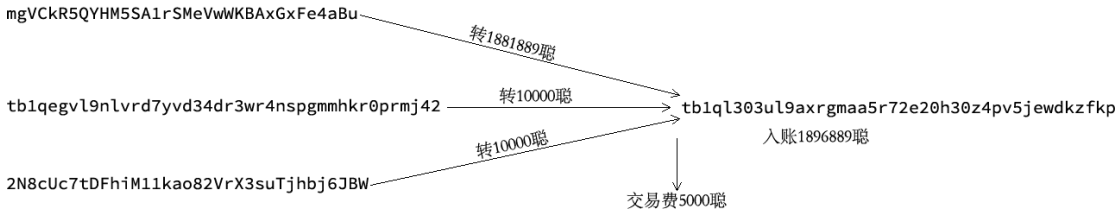

mgVCkR5QYHM5SA1rSMeVwWKBAxGxFe4aBu
转1881889聪
tb1qegvl9nlvrd7yvd34dr3wr4nspgmmhkr0prmj42
转10000聪
tb1ql303ul9axrgmaa5r72e20h30z4pv5jewdkzfkp
入账1896889聪
转10000聪
2N8cUc7tDFhiM11kao82VrX3suTjhbj6JBW
交易费5000聪

图 B-1 转账示意图

B.2 签名

三个输入属于不同类型，第一个输入是传统的 P2PKH，所以采用传统的签名方法。

B.2.1 对第一个输入签名（P2PKH）

版本 2（即 00000002，换成小端存储就是 02000000）和 3 个输入，所以交易的开头就是

```
02000000 03 ……
```

然后就是第一个输入的 txid（换成小端存储格式）：

```
$ echo 759b97690493c4b94e14f9c78d93e661f28a8fdc30f9c71beb794e2e02e59399 | tac -rs ..; echo
9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75
```

输出序号为 00000000，所以交易成了：

```
02000000 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 ……
```

接下来就是输出脚本的长度 0x19、输出脚本（UTXO 中的"script_hex"字段）和 0xffffffff：

```
02000000 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 19
76a9140aa35c213b8e3de1bad9325b7696dde7f443f2ca88ac ffffffff ……
```

注意：txid、输出脚本请参考前面第一个地址的 UTXO。

接下来是另外两个隔离见证的输入，对于传统签名，这两个隔离见证的输出脚本必须省略（保留长度为 0x00），它们的 txid 和输出序号参考相应的 UTXO（"txid"和"output_no"字段）。

两个 txid 换成小端格式：

```
$ echo 2a363129c0f9328c043e04d03441005d22c231a60ebaa1b752c9bfcff3c0c2e6 | tac -rs ..; echo
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a

$ echo 55bdf19321b7cf9b3533f8f915b391febb34930eae05a9c2f4a0d3e2fa1d6807 | tac -rs ..; echo
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55
```

输出序号都是 0x00000000，于是交易变为：

```
02000000 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 19
76a9140aa35c213b8e3de1bad9325b7696dde7f443f2ca88ac ffffffff
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 00 ffffffff
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000 00 ffffffff ……
```

整个交易只有一个输出，即 01，入账 1896889 聪，可采用下面的 Python 语句转换为小端存储格式：

```
$ python3
>>> int.to_bytes(1896889,8,'little').hex()
'b9f11c0000000000'
>>>
```

因此交易变成：

```
02000000 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 19
76a9140aa35c213b8e3de1bad9325b7696dde7f443f2ca88ac ffffffff
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 00 ffffffff
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000 00 ffffffff 01
b9f11c0000000000 ……
```

因为输出地址为 P2WPKH ，所以输出脚本格式为 OP_0 <pubkey hash>。当矿工节点遇到这样的输出脚本时就会转换为

传统格式，即 OP_DUP OP_HASH160 <pubkey hash> OP_EQUALVERIFY OP_CHECKSIG。首先对入账钱包的公钥做哈希运算得到地址（RIPEMD 160 格式）：

```
$ echo 035c84c164b5a7a59dc587b39a0599af0023e706fa0bcaec2fcf02ef1d0e69e309 | xxd -p -r | \
sha256sum -b | xxd -p -r | openssl ripemd
(stdin)= fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e
```

操作符 OP_0 对应 0x00，公钥的哈希长度为 0x14，公钥哈希、哈希长度和操作符加起来总长度为 0x16，所以交易变为：

```
02000000 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 19
76a9140aa35c213b8e3de1bad9325b7696dde7f443f2ca88ac ffffffff
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 00 ffffffff
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000 00 ffffffff 01 b9f11c0000000000
16 00 14 fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e ……
```

最后加上锁定时间 000000 和签名哈希标志 01000000，最终的交易为：

```
02000000 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 19
76a9140aa35c213b8e3de1bad9325b7696dde7f443f2ca88ac ffffffff
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 00 ffffffff
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000 00 ffffffff 01 b9f11c0000000000
16 00 14 fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e 00000000 01000000
```

去掉空格就是：

```
02000000039993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75000000001976a9140aa35c213b8e3de1
bad9325b7696dde7f443f2ca88acffffffffe6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a000000
0000ffffffff07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55000000000000ffffffff01b9f11c0000
000000160014fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e000000001000000
```

下面对交易做哈希运算并签名。

首先把第一个比特币私钥转换为兼容 openssl 签名的私钥：

```
$ echo "302e0201010420 09624c6ac1424fcbc5d1c1b87e7c27ef0b977da5714a6d0b5bd8fe85527a0d66 \
a00706052b8104000a" | xxd -p -r > priv-legacy.hex
$ openssl ec -inform d < priv-legacy.hex > priv-legacy.pem
```

兼容签名私钥保存在 priv-legacy.pem 文件中。下面正式签名：

```
$ echo
02000000039993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75000000001976a9140aa35c213b8e3de1
bad9325b7696dde7f443f2ca88acffffffffe6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a000000
0000ffffffff07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55000000000000ffffffff01b9f11c0000
000000160014fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e000000001000000 | xxd -p -r | sha256sum -b | xxd -p -
r | sha256sum -b | xxd -p -r > msg-legacy

$ openssl pkeyutl -inkey priv-legacy.pem -sign -in msg-legacy -pkeyopt digest:sha256 | xxd -p -c 256
304402200a7af1008a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d395
1f66565d6b9932b2283e59a6422a283c14
```

验证签名是否合法：

```
$ python3
>>> s = 0x2ea39f5f9e2491ec5942b00a48d3951f66565d6b9932b2283e59a6422a283c14
>>> n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
>>> s < n/2
True
>>>
```

最后再追加 0x01，即签名为：

```
304402200a7af1008a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d395
1f66565d6b9932b2283e59a6422a283c1401
```

B.2.2 对第二个输入签名（P2WPKH）

现在对第二个输入签名，这是一个隔离见证地址，签名的交易格式与 P2PKH 完全不同，格式说明参见 https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki#Specification。首先是交易版本 02000000，接下来是三个输入的哈希值，txid 和输出索引唯一定义了一个输入。可采用下面的命令计算哈希值：

```
$ echo "9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 \
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 \
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000" | xxd -p -r | \
sha256sum -b | xxd -p -r | sha256sum -b
462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a *-
```

注意：txid 和输出索引都是小端格式。因此交易的首部为：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a ……
```

再计算三个输入的顺序号（都是 0xffffffff）的哈希值：

```
$ echo "ffffffff ffffffff ffffffff" | xxd -p -r | sha256sum -b | xxd -p -r | sha256sum -b
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8 *-
```

交易变为：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8 ……
```

接下来是输出点（outpoint），即第二个输入（txid+输出索引，要求小端格式），此时交易变为：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 ……
```

接下来就是输出脚本，P2WPKH 与 P2PKH 相同，即 OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG ➔ 19 76 a9 14 <pubkey hash> 88 ac，其中 0x14 是<pubkey hash>的长度，0x19 是“76 a9 14 <pubkey hash> 88 ac”的长度。采用下面的语句对第二个钱包的公钥做哈希运算：

```
$ echo 02d77cc64a910abf4b6700e75131fa5ec12b8de422b1521dc1d8625559915cbf77 | xxd -p -r | \
sha256sum -b | xxd -p -r | openssl ripemd
(stdin)= ca19f2cfec1b7c46363568e2e1d6700a37bbd86f
```

因此交易变成：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000
76a914ca19f2cfec1b7c46363568e2e1d6700a37bbd86f88ac ……
```

接下来就是从地址要转走多少比特币，我们要全部转走，即 10000 聪，采用下面语句转换为 8 字节的小端存储：

```
$ python3
>>> int.to_bytes(10000,8,'little').hex()
'1027000000000000'
>>>
```

最后追加顺序号 ffffffff，这样交易变为：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000
76a914ca19f2cfec1b7c46363568e2e1d6700a37bbd86f88ac 1027000000000000 ffffffff ……
```

接下来就是输出了，即全部输出的哈希值，每个输出的格式是“<转入比特币(单位聪)><输出脚本长度><输出脚本>”。我们只有一个输出，即入账=b9f11c0000000000、输出脚本长度=0x16、输出脚本

=0014fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e（格式为 OP_0 <pubkey hash>，在 A.2.1 小节中已经算出）。首先对入账钱包的公钥做哈希运算得到<输出脚本>：

```
$ echo "b9f11c0000000000 16 0014fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e" | xxd -p -r | sha256sum -b | xxd
-p -r | sha256sum -b
8e1b4d840b0acf5ccc59f32aca2831a32305ce65fd8cd735fe0b27bf7a5ff310 *-
```

现在交易成了：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000
76a914ca19f2cfec1b7c46363568e2e1d6700a37bbd86f88ac 1027000000000000 ffffffff
8e1b4d840b0acf5ccc59f32aca2831a32305ce65fd8cd735fe0b27bf7a5ff310 ……
```

最后加上锁定时间 00000000 和哈希标志 01000000：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000
76a914ca19f2cfec1b7c46363568e2e1d6700a37bbd86f88ac 1027000000000000 ffffffff
8e1b4d840b0acf5ccc59f32aca2831a32305ce65fd8cd735fe0b27bf7a5ff310 00000000 01000000
```

去掉空格后的交易为：

```
02000000462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a82a7d5bb59fc957ff7f737ca0b8be713c7
05d6173783ad5edb067819bed70be8e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a0000000076a9
14ca19f2cfec1b7c46363568e2e1d6700a37bbd86f88ac1027000000000000ffffffff8e1b4d840b0acf5ccc59f32aca2831a32305
ce65fd8cd735fe0b27bf7a5ff3100000000001000000
```

下面对交易做哈希并签名。

把第二个比特币私钥转换为满足 openssl 签名的私钥：

```
$ echo "302e0201010420 4181c4b9b9e1f83c765a17a4dbb601684e8b607d9046251833602e074db09fe0 \
a00706052b8104000a" | xxd -p -r > priv-segwit.hex
$ openssl ec -inform d < priv-segwit.hex > priv-segwit.pem
```

正式签名：

```
$ echo
02000000462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a82a7d5bb59fc957ff7f737ca0b8be713c7
05d6173783ad5edb067819bed70be8e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a0000000076a9
14ca19f2cfec1b7c46363568e2e1d6700a37bbd86f88ac1027000000000000ffffffff8e1b4d840b0acf5ccc59f32aca2831a32305
ce65fd8cd735fe0b27bf7a5ff3100000000001000000 | xxd -p -r | sha256sum -b | xxd -p -r | sha256sum -b | xxd -
p -r > msg-segwit

$ openssl pkeyutl -inkey priv-segwit.pem -sign -in msg-segwit -pkeyopt digest:sha256 | xxd -p -c 256
3044022014ad39d4930e49f63ce97623299a9750412fa731b24cfcc5ddcfe9c41075b46702203bffa82e7b3966bfe302522c9b06f4
9006624f40845383b31e10eb446886709a
```

追加 01 就得到最终的签名：

3044022014ad39d4930e49f63ce97623299a9750412fa731b24cfcc5ddcfe9c41075b46702203bffa82e7b3966bfe302522c9b06f490
06624f40845383b31e10eb446886709a01

B.2.3 对第三个输入签名（P2SH-P2WPKH）

开始三项（版本、全部输入的哈希值、全部顺序号的哈希值）与第二个输入相同，即：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8 ……
```

接下来是输出点（outpint），即第三个输入（txid+输出索引号，要求小端格式），此后交易变为：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000 ……
```

接下来就是输出脚本（P2WPKH），即 19 76 a9 14 <pubkey hash> 88 ac，其中 14 是<pubkey hash>的长度，19 是“76 a9 14 <pubkey hash> 88 ac”的长度。采用下面的语句对第三个钱包的公钥做哈希算：

```
$ echo 028b2da00c8c91ea6b0e78cd4fbd286c231bd88a8082ccfd9f57160eda5be3b695 | xxd -p -r | \
sha256sum -b | xxd -p -r | openssl ripemd
(stdin)= 4c7abacf1fb6504e6f495241edd80b7b37e83c15
```

因此交易变成：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000
1976a9144c7abacf1fb6504e6f495241edd80b7b37e83c1588ac ……
```

加下来是转出比特币数（1027000000000000）和顺序号（ffffffff）：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000
1976a9144c7abacf1fb6504e6f495241edd80b7b37e83c1588ac 1027000000000000 ffffffff ……
```

最后是全部输出的哈希值、锁定时间（00000000）和签名哈希标志（01000000），这些与第二个输入一样，我们的交易变成：

```
02000000 462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a
82a7d5bb59fc957ff7f737ca0b8be713c705d6173783ad5edb067819bed70be8
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000
1976a9144c7abacf1fb6504e6f495241edd80b7b37e83c1588ac 1027000000000000 ffffffff
8e1b4d840b0acf5ccc59f32aca2831a32305ce65fd8cd735fe0b27bf7a5ff310 00000000 01000000
```

去掉空格后：

```
02000000462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a82a7d5bb59fc957ff7f737ca0b8be713c7
05d6173783ad5edb067819bed70be807681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55000000001976
a9144c7abacf1fb6504e6f495241edd80b7b37e83c1588ac10270000000000000ffffffff8e1b4d840b0acf5ccc59f32aca2831a323
05ce65fd8cd735fe0b27bf7a5ff3100000000001000000
```

下面对交易做哈希并签名。

首先把第三个比特币私钥转换为满足 openssl 签名格式的私钥：

```
$ echo "302e0201010420 5e69b61ff0aff209253a0aa74c33c841fac1f14bb47fdfc47a2abfcb9d57ed49 \
a00706052b8104000a" | xxd -p -r > priv-wrapped.hex
$ openssl ec -inform d < priv-wrapped.hex > priv-wrapped.pem
```

正式签名：

```
$ echo
02000000462e222f41ec655c428d9a78d6b9d6b10ad117c2de6ce74de8e6284ece92a37a82a7d5bb59fc957ff7f737ca0b8be713c7
05d6173783ad5edb067819bed70be807681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55000000001976
a9144c7abacf1fb6504e6f495241edd80b7b37e83c1588ac10270000000000000ffffffff8e1b4d840b0acf5ccc59f32aca2831a323
05ce65fd8cd735fe0b27bf7a5ff3100000000001000000 | xxd -p -r | sha256sum -b | xxd -p -r | sha256sum -b | xxd
-p -r > msg-wrapped

$ openssl pkeyutl -inkey priv-wrapped.pem -sign -in msg-wrapped -pkeyopt digest:sha256 | xxd -p -c 256
3044022060d24413c44266af3a3237e4c2f4df3140ee2b8d5f2f9ab0df5bbbb5aaea352b022057b066c92a0fc54a7835b3ae998642
9385421c9b001f6dd0131b31a459ba4002
```

再追加 0x01 就得到最终的签名：

3044022060d24413c44266af3a3237e4c2f4df3140ee2b8d5f2f9ab0df5bbbb5aaea352b02257b066c92a0fc54a7835b3ae9986429385421c9b001f6dd0131b31a459ba40020101

B.2.4 构造最终交易

隔离见证交易的具体格式请参见 https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki#Specification。

首先是版本号（02000000）、SegWit 标记（00）和标志（01），因此交易的开头是：

```
02000000 00 01 ……
```

然后是输入个数（0x03）：

```
02000000 00 01 03 ……
```

第一个输入是传统的 P2PKH 输入，序列化为<txid 小端格式><输出索引><输入脚本长度><输入脚本><ffffffff>，其中“输入脚本”就是“<签名长度><签名><公钥长度><公钥>”，因此：

```
9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 6a 47
304402200a7af1008a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d395
1f66565d6b9932b2283e59a6422a283c1401 21 02699c92c9e62eeb2ec0054a5a66dc8cf89953d42741729e06e45334c2bc316dc4
ffffffff
```

其中 47（十六进制）为签名的长度，21 为公钥的长度，而 6a 是整个输入脚本的长度（47+21+2），之所以要加上 2，那是因为 47 和 21 本省占据了两个字节。因此交易变成：

```
02000000 00 01 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 6a 47
304402200a7af1008a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d395
1f66565d6b9932b2283e59a6422a283c1401 21 02699c92c9e62eeb2ec0054a5a66dc8cf89953d42741729e06e45334c2bc316dc4
ffffffff ……
```

第二个输入比较简单，只有<txid 小端格式><输出索引><00><ffffffff>，输入脚本单独放在 witness 节。因此交易变成：

```
02000000 00 01 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 6a 47
304402200a7af1008a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d395
1f66565d6b9932b2283e59a6422a283c1401 21 02699c92c9e62eeb2ec0054a5a66dc8cf89953d42741729e06e45334c2bc316dc4
ffffffff e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 00 ffffffff ……
```

第三个输入为 P2SH-P2WPKH，序列化后为<txid 小端格式><输出索引小端格式><赎回脚本长度><赎回脚本><ffffffff>，其中<赎回脚本>的格式为“OP_0 <pubkey hash>”。因此交易成了：

```
02000000 00 01 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 6a 47
304402200a7af1008a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d395
1f66565d6b9932b2283e59a6422a283c1401 21 02699c92c9e62eeb2ec0054a5a66dc8cf89953d42741729e06e45334c2bc316dc4
ffffffff e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 00 ffffffff
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000 17 16 00 14
4c7abacf1fb6504e6f495241edd80b7b37e83c15 ffffffff ……
```

第三个钱包公钥的哈希值在上一小节已经计算出来，0x14 是公钥哈希值的长度，0x00 就是 OP_0，0x16 就是“00144c7abacf1fb6504e6f495241edd80b7b37e83c15”的长度，而 0x17 就是“1600144c7abacf1fb6504e6f495241edd80b7b37e83c15”的长度。

输出与 A.2.1 小节中的一样，直接拷贝过来即可：

```
02000000 00 01 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 6a 47
304402200a7af1008a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d395
1f66565d6b9932b2283e59a6422a283c1401 21 02699c92c9e62eeb2ec0054a5a66dc8cf89953d42741729e06e45334c2bc316dc4
ffffffff e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 00 ffffffff
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000 17 16 00 14
```

```
4c7abacf1fb6504e6f495241edd80b7b37e83c15 ffffffff 01 b9f11c0000000000 16 00 14
fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e ……
```

接下来是隔离见证部分，每一个输入都要在这里占据一项，每一项包括：条目数量、条目 1 长度、条目 1、条目 2 长度、条目 2、……。第一个输入为传统类型，没有隔离见证，所以条目数量为 0x00，其他两个输入包含两个条目，即签名和公钥，因此有：

```
00
--
02
47
3044022014ad39d4930e49f63ce97623299a9750412fa731b24cfcc5ddcfe9c41075b46702203bffa82e7b3966bfe302522c9b06f4
9006624f40845383b31e10eb446886709a01
21 02d77cc64a910abf4b6700e75131fa5ec12b8de422b1521dc1d8625559915cbf77
--
02
47
3044022060d24413c44266af3a3237e4c2f4df3140ee2b8d5f2f9ab0df5bbbb5aaea352b022057b066c92a0fc54a7835b3ae998642
9385421c9b001f6dd0131b31a459ba400201
21 028b2da00c8c91ea6b0e78cd4fbd286c231bd88a8082ccfd9f57160eda5be3b695
```

因此现在交易变为：

```
02000000 00 01 03 9993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75 00000000 6a 47
304402200a7af1008a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d395
1f66565d6b9932b2283e59a6422a283c1401 21 02699c92c9e62eeb2ec0054a5a66dc8cf89953d42741729e06e45334c2bc316dc4
ffffffff e6c2c0f3cfbfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a 00000000 00 ffffffff
07681dfae2d3a0f4c2a905ae0e9334bbfe91b315f9f833359bcfb72193f1bd55 00000000 17 16 00 14
4c7abacf1fb6504e6f495241edd80b7b37e83c15 ffffffff 01 b9f11c0000000000 16 00 14
fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e 00 02 47
3044022014ad39d4930e49f63ce97623299a9750412fa731b24cfcc5ddcfe9c41075b46702203bffa82e7b3966bfe302522c9b06f4
9006624f40845383b31e10eb446886709a01 21 02d77cc64a910abf4b6700e75131fa5ec12b8de422b1521dc1d8625559915cbf77
02 47
3044022060d24413c44266af3a3237e4c2f4df3140ee2b8d5f2f9ab0df5bbbb5aaea352b022057b066c92a0fc54a7835b3ae998642
9385421c9b001f6dd0131b31a459ba400201 21 028b2da00c8c91ea6b0e78cd4fbd286c231bd88a8082ccfd9f57160eda5be3b695
```

最后加上锁定时间 00000000，并去掉空格，最终的交易如下：

```
020000000001039993e5022e4e79eb1bc7f930dc8f8af261e6938dc7f9144eb9c4930469979b75000000006a47304402200a7af100
8a8df5d7b6d5da5050fa786e162ff78bc7a6022155ab3161f23594f402202ea39f5f9e2491ec5942b00a48d3951f66565d6b9932b2
283e59a6422a283c140121026 99c92c9e62eeb2ec0054a5a66dc8cf89953d42741729e06e45334c2bc316dc4ffffffffe6c2c0f3cf
bfc952b7a1ba0ea631c2225d004134d0043e048c32f9c02931362a00000000 00ffffffff07681dfae2d3a0f4c2a905ae0e9334bbfe
91b315f9f833359bcfb72193f1bd5500000000171600144c7abacf1fb6504e6f495241edd80b7b37e83c15ffffffff01b9f11c0000
000000160014fc5f1e7cbd30d1bef683f2b2a7de2f1542ca4b2e00024730440220 14ad39d4930e49f63ce97623299a9750412fa731
b24cfcc5ddcfe9c41075b46702203bffa82e7b3966bfe302522c9b06f49006624f40845383b31e10eb446886709a012102d77cc64a
910abf4b6700e75131fa5ec12b8de422b1521dc1d8625559915cbf77024730440220 60d24413c44266af3a3237e4c2f4df3140ee2b
8d5f2f9ab0df5bbbb5aaea352b022057b066c92a0fc54a7835b3ae9986429385421c9b001f6dd0131b31a459ba40020121028b2da0
0c8c91ea6b0e78cd4fbd286c231bd88a8082ccfd9f57160eda5be3b69500000000
```

最后在 https://live.blockcypher.com/btc/decodetx/ 上看看能否解码，然后在 https://live.blockcypher.com/btc/pushtx/ 发布交易。

附录 C：使用 MetaMask 钱包

MetaMask 钱包在电脑上，作为浏览器的插件安装，通行的浏览器有 firefox 和 chrome，也提供了移动设备上的 app。

用浏览器打开网站 https://metamask.io/download，然后点击对应的浏览器图标，再点击 Add to Firefox，然后弹出如图 C-1 所示的左侧部分。



图 C-1 安装浏览器插件

安装完后会在浏览器右上角出现图标，点击它开始初始化，参考图 C-2 所示。

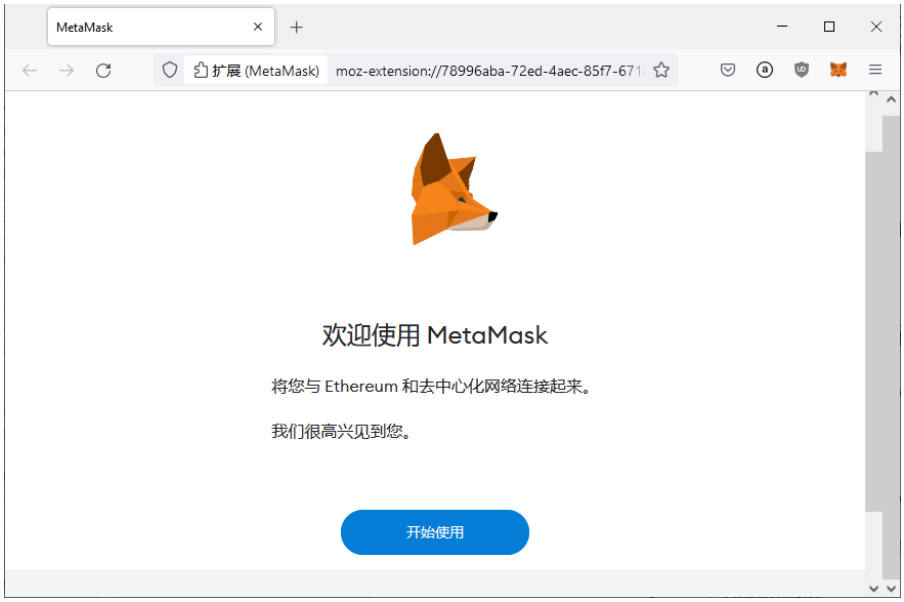


图 C-2 开始初始化钱包

点击图中的“开始使用”按钮。接下来的界面问你想不想反馈改建意见，选择 不，谢谢，然后出现图 C-3 所示的画面。

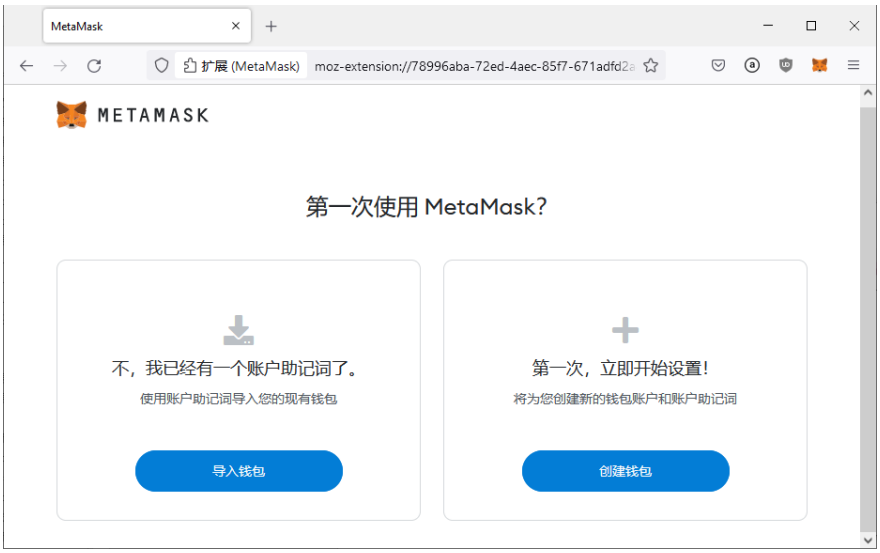


图 C-3 导入还是创建钱包

如果你已经有了 MetaMask 钱包，可以选择“导入钱包”（当然也可以再“创建钱包”），否则，只能选择“创建钱包”了。点击“创建钱包”之后出现如图 C-4 所示的界面，提示输入密码，这个密码是以后登录钱包的密码，防备别人使用你的电脑时进入钱包。



图 C-4 给钱包设置密码

点击“创建”后再点击下一步，出现如图 C-5 左侧所示的界面，点击那把锁，就能看到 12 个助记词（图 C-5 右侧），切记切记：要备份保管好这 12 个单词。以后恢复钱包或把钱包导入到其他设备时都要输入这写助记词。



图 C-5 钱包的助记词

点击“下一步”后出现如图 C-6 左侧所示的界面，按照助记单词的先后次序点击输入（见图 C-6 右侧部分）。

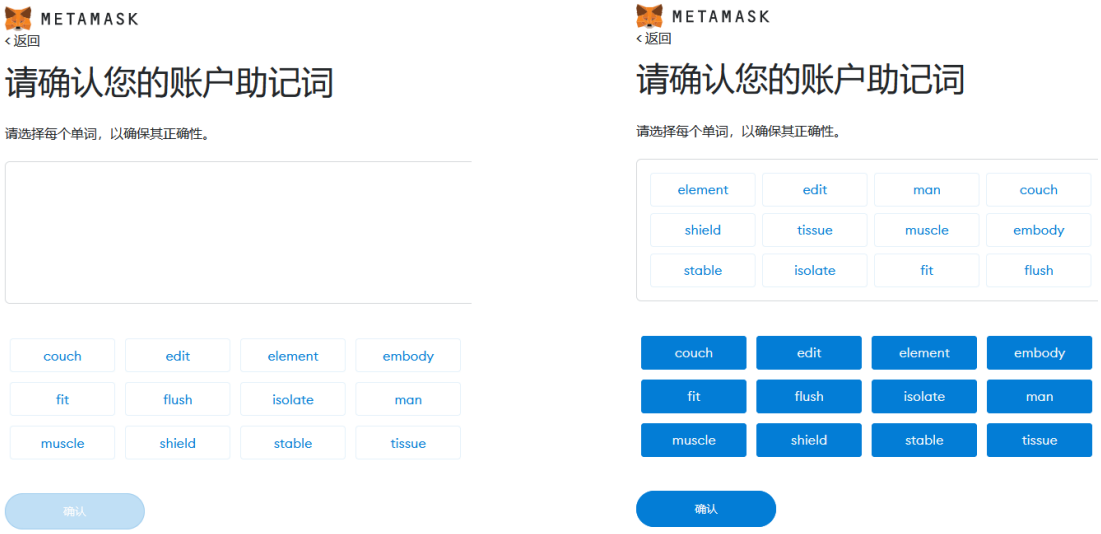


图 C-6 再三确认是否已记下助记词

最后点击“确认”直至完成，最后的画面如图 C-7 所示。

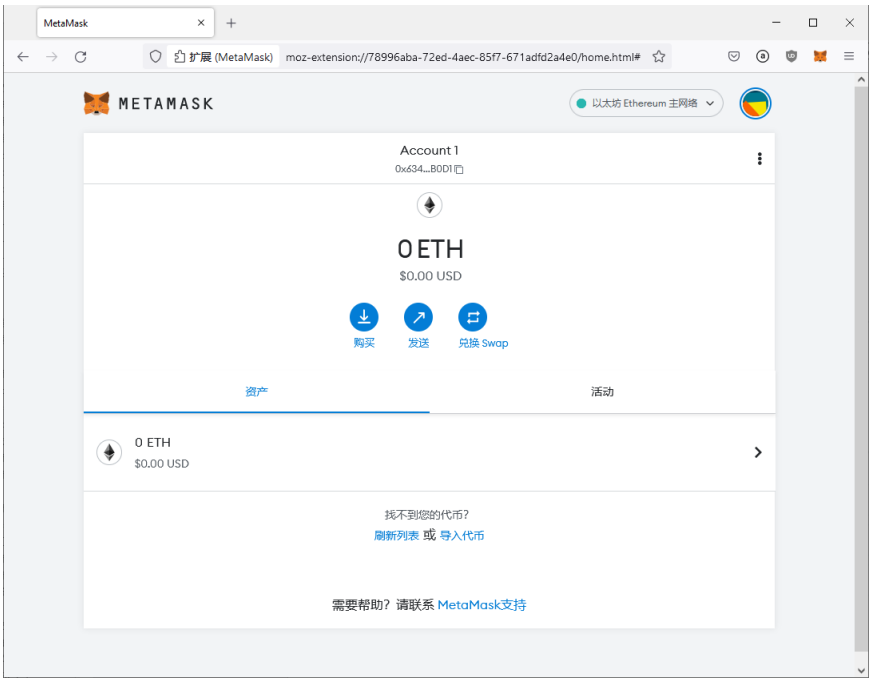


图 C-7 新创建的以太币钱包

现在我们看看如何导入钱包，在图 C-3 上点击“导入钱包”。接下来需要输入助记单词和钱包密码，最后点击“导入”，参见图 C-8。



图 C-8 输入助记词导入钱包

为了以后能用钱包做测试开发，可以设置显示测试网络，点击右上角的 ，然后点击“设置”，再点击“高级”，启用“显示测试网络”，参考图 C-9 左侧部分。

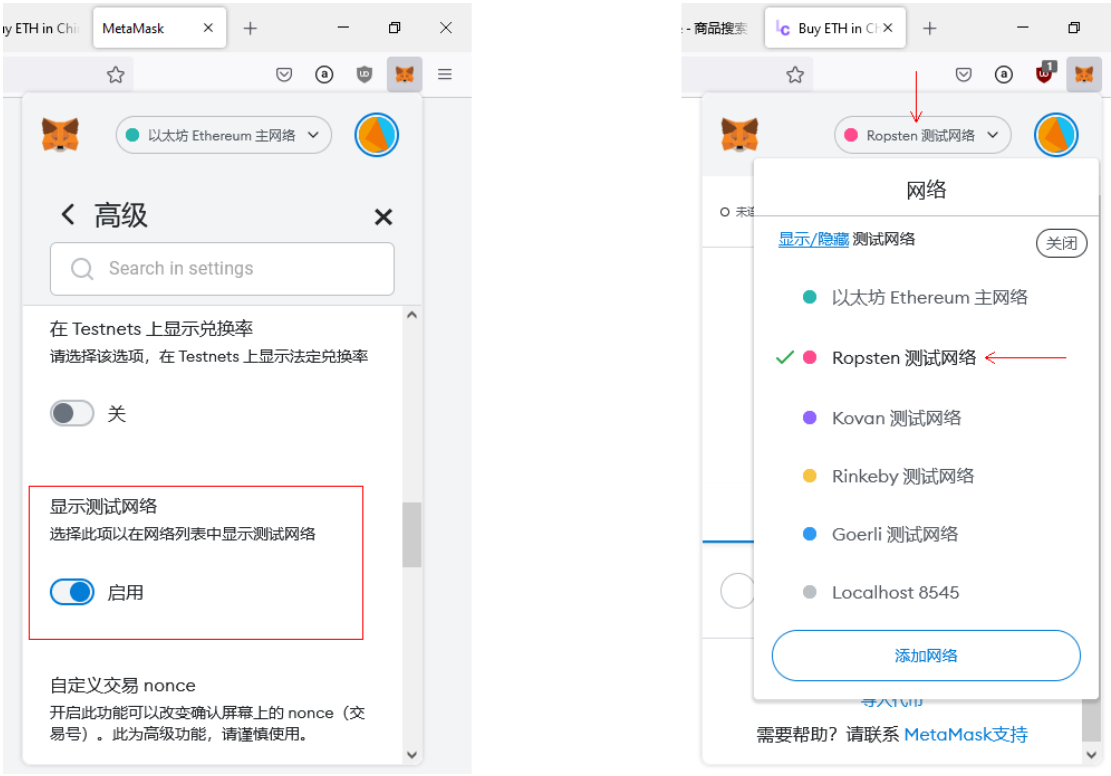
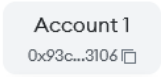


图 C-9 启用“显示测试网络”



以后就可以随时在以太主链和测试链之间来回切换了，见图 C-9 右侧部分。从测试链给钱包充值，点击复制钱包地址，然后去网站 https://faucet.egorfine.com/充值，会充入一个以太币，参见图 C-10 所示。

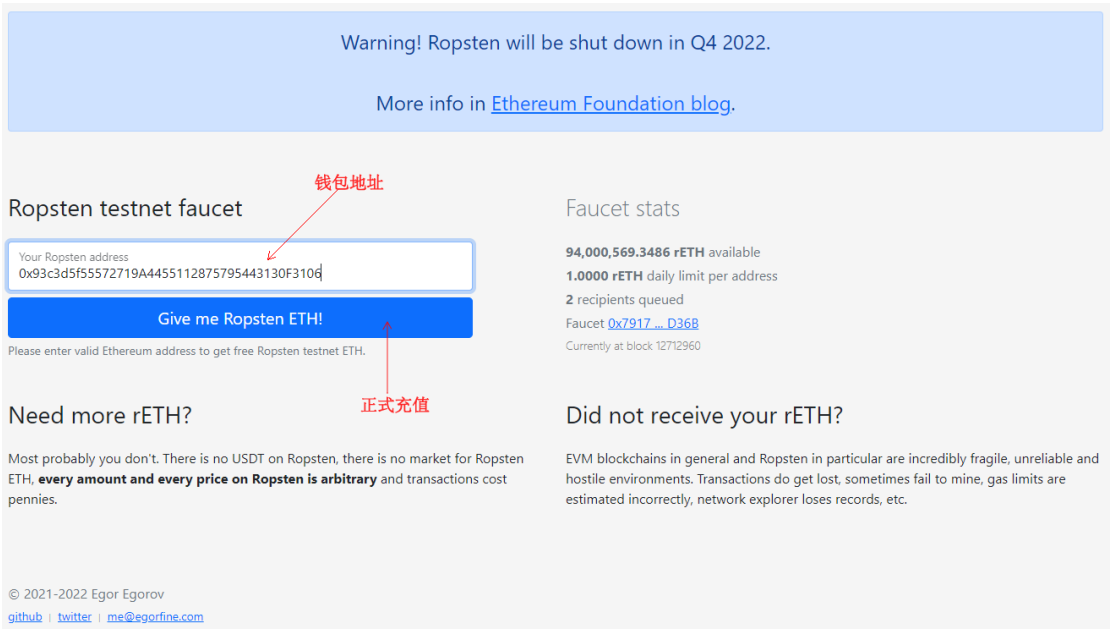


图 C-10 从测试链充值

附录 D：比特币测试地址

交易 id：466200308696215bbc949d5141a49a4138ecdfdfaa2a8029c1f9bcecd1f96177

7548329a72f9982bbe50fecad6fe9b4242877c75b1c950c3660b839e41f2e989

地址：

bc1p8dqa4wjvnt890qmfws83te0v3qxzsfu7ul63kp7u56w8qc0qwp5qv995qn

bc1pyfw56zu5vsq0ulu9kytasgw4xwnm3eysll6tfdz8d9gtht97k7tqxsz78n

bc1qwafvze0200nh9vkq4jmlf4sy0tn0ga5w0zpkpg

bc1qujepl0k5n0ga2e86yskvxa6auehpf6dlf84dx0

3Nntbr1ReGL4hCzgr8fGhFvKJvKzcAYENC

P2SH 地址：327MUKo9e53mUw5y3w31zSzUhmLsLNgdKV

P2TR 地址：bc1p9l9dw3cz09jje30c3wysseuddax40t6kyuvrkqlusszvknskmzys00k7zm

https://aandds.com/blog/bitcoin-taproot.html

比如我申请的两个比特币测试地址为：

1、 密钥：cSQrk3dKt3Pnxt9jr5U5pAcnw6BrAcFRbo1KF8RHHwY7f7EcNdik

公钥：03b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3

P2PKH 地址：muaKN6YSMBppmJnkanCVAQsDdzGXJfiTHH

P2WPKH 地址：tb1qng6f35spexs5nr80enz3a76kuz5s9m20um22xx

P2TR 地址：tb1p0980z73rdgfachyf9dq4tspzhpl35e9adutfyxeldapzec0afrjs5k8qrs

2、 密钥：cT8N2nooDHwjcju3ckEbu5CDNQx3KpHs6nJ7HFsPodfkcG9s52ct

公钥：02ba6ecb471acd1f1934518ba92f70c09118a4241a97ce6480af385a19655cd0d2

P2PKH 地址：mr1s3DVw146qCoMZRcSzgkj3mDYtp6SKkN

P2WPKH 地址：tb1qwv5wau4ye8w7p0z9ylcscy8cet9zs5d80aut9s

P2TR 地址：tb1pywgys8rc0g2g50vhxuzryawpyc409vvuppda46uezhvusg4yutjqu2lgm5

3、 密钥：cSyAHuVqHei9YrAEcmjaVf2FH9SHizwRLGCqQ56p1R7zgC2yUdRu

被密码 Y*2*保护：6PYXEV3JjJ6HFF42eAJc81rRdS9PPvdx5A1z3EJTzAnV1W81CdiPUZ34ro

公钥：02869adf9d5a84bbdd4aaa4c170c1174997b26e3499b07e87f792d8f05f6e695aa

P2WPKH 地址：tb1qqxc0yef599p7r9gmkayf8pln5d7kunwc9qwd38

P2TR 地址：tb1pmjw523jfl38hc5fq7elhq0m2u6xhtykpc29mg47ghp8szc5rupqslygy60（P2TR）

P2PKH 地址：mfftwjEbL3f2iWfKfrLRJtjQSXZEsiJUtu

4、P2SH 和 P2WSH 多签地址（2-of-3）：

三个私钥见上面的。

赎回脚本：OP_2 03b0d5d925b0be30701bc01afeeb9da28ba985ea9dc48f9d23c9dd3ae8137654c3 02ba6ecb471acd1f1934518ba92f70c09118a4241a97ce6480af385a19655cd0d2 02869adf9d5a84bbdd4aaa4c170c1174997b26e3499b07e87f792d8f05f6e695aa OP_3 OP_CHECKMULTISIG

P2SH 地址：2MvSmeV7P9djc6KcefGDJwR7j478ALi8sYs

P2WSH 地址：tb1q8k9cu8ynerdj3qlxsjq75040cgkg6eqj9hp99kw6kzqec67mws0sxm8ck2

5、P2SH-P2WPKH 地址：

密钥：cSQo7aCz6TcmtzRXVkJSiL6qhAvTCtK5yx2Wd1jstbVWCNFNXe2E

公钥：03f3ac2a5ba8a7d78f094d17106764a6737ea69803b38a960ed15cdc55e484738e

地址：2NAbxnF8ZbNpLh7YtFH1YZiQCpHUNQeTzFj

见证脚本：0014b23450d31f49a43dcb05ab26bd1f2348fd533e47

赎回脚本：a914be662dcac3e5d09e4cbe13deb11f39a8b2ef534487

6、P2WSH 自定赎回脚本地址：

密钥：cNTDA6HRLTgDoMB298E7TxoPrhkWgWERZMufbh4mj1ZeksfZrA8V

公钥：0200e629014f0cd042f5d0d233cbb2b706ee13cc8b4ae560e10698f7f53acbb59e

P2WSH 钱包地址：tb1qq00eppgntelkrnh09f0pdsrr4y7ahchdudvq9wnfg09yky4lsahskv47ha

花钱暗语：中国大妈

赎回脚本：OP_HASH160 16d356ebcc092cdd275097636259c25288b22558 OP_EQUALVERIFY 0200e629014f0cd042f5d0d233cbb2b706ee13cc8b4ae560e10698f7f53acbb59e OP_CHECKSIG

（赎回脚本格式：OP_HASH160 <花钱暗语的哈希值的哈希值> OP_EQUALVERIFY <公钥> OP_CHECKSIG）

输出脚本：002003df9085135e7f61ceef2a5e16c063a93ddbe2ede35802ba6943ca4b12bf876f

见证数据格式：<签名> <花钱暗语的哈希值> <赎回脚本>

在 https://testnet-faucet.mempool.co/上给地址充币：

附录 E：莱特币

莱特币就是比特币的分叉，所以大部分机制与比特币一样，钱包地址值用三种：P2PKH（对应路径 m/44'/2'/0'/0/0）、P2SH-P2WPKH（对应路径 m/49'/2'/0'/0/0）、P2WPKH（对应路径 m/84'/2'/0'/0/0）。1LTC=10^8 Litoshi。莱特币网络每 2.5 分钟出一个区块。发行总量总量预期为 8400 万。

主网查询：https://blockchair.com/litecoin、https://chain.so/（主网和测试网都可以）

获得 UTXO：https://api.blockcypher.com/v1/ltc/main/addrs/${address}?unspentOnly=true、https://litecoinspace.org/api/address/{address}/utxo

获取交易原始 hex 码：https://litecoinspace.org/api/tx/<txid>/hex、https://api.blockchair.com/litecoin/raw/transaction/<txid>

发布交易：https://api.blockcypher.com/v1/ltc/main/txs/push

测试地址充值：https://cypherfaucet.com/ltc-testnet

查询网站：https://litecoinspace.org/testnet

获得 UTXO：https://api.blockcypher.com/v1/ltc/test3/addrs/${address}?unspentOnly=true、https://litecoinspace.org/testnet/api/address/{address}/utxo

获取交易原始 hex 码：https://litecoinspace.org/testnet/api/tx/<txid>/hex

发布交易：https://api.blockcypher.com/v1/ltc/test3/txs/push

主网: https://blockchair.com/litecoin

测试网: https://blockchair.com/litecoin/testnet

下面是测试钱包。

助记词：observe size describe badge auction lumber banner shadow age north trend idle pink dose truly pumpkin humor harvest spy spend swap uncle ribbon eager

一、钱包 1：m/84'/1'/0'/0/0

私钥：cNj53qCo4RgSMxkxDNY41G7mfwTmcyBqcvSTteZPpc6kmeqWguNe

6PYLvaNtSuiiUu17E5MYFDSf8wXcSZWNQA5JjU8V8g8RUE7MYhsAh6ibXZ （私钥被加密 123 保护）

公钥：028d984f15e39a0468e553f1b49193e55a431cc79bcce0f30866675f284cd76e19

地址：tltc1qpjay39ntltfg5umy0wt5l3xhxgxkr7rfa2a7dr（P2WPKH）

交易：2a5c511e9c19bc2bac9ee3a0c6f8a9f0b2dea8d5cec9b585821ab0fda0587cc9

二、钱包 2：m/84'/1'/0'/0/1

私钥：cR31PQypiztoNAKLUdGEr39YkFoB9bmxCbqLpQRRjTgP9yePgZja

公钥：039a483796160d3e6ba811e10008d6c4118c7312d7ba6dba8edcfb284a4248c21f

地址：tltc1q32qsz8kyz0evy9z2ryng46esw9wmtfsvpjzm3y（P2WPKH）

三、钱包 3：m/84'/1'/0'/0/2

私钥：cUaA7hsWR1qmQDo3QQSyhhtRiPifANNoZvgtuYwVvSWS5mvovvd4

公钥：03a53317601d39755f0d4c614f62b30ce634b8d0b42f8c8d45c2023259b438da64

地址：tltc1qj4kzqyy25d88gh8wrslzwadrywfhzlpfchhjhg（P2WPKH）

mu92WJzSqF9jMr1sEphMrqMkk6ZfXb3ZtP （P2PKH）

附录 F：索尔币

即 solana 区块链上的 sol 币，它的钱包路径为 m/44'/501'/i'/j'（第 *i+1* 个账户里的第 *j+1* 个钱包），其中：m - 主节点、44' - BIP-44 规范（表示遵循 BIP44 标准）、501' - Solana 的币种注册号（由 SLIP-44 分配）、i - 账户索引（取值范围 [0, 2147483647]）、j - 地址索引（取值范围 [0, 2147483647]），比如 m/44'/501'/0'/2'。Solana 使用 Ed25519 椭圆曲线，而不是比特币使用的 secp256k1，私钥和公钥都是 32 个字节，通常把公钥附加在私钥右侧从而合并成一个 64 个字节的密钥，最后编码成 Base58。Solana 的 HD 路径在测试网和主网之间没有区别，使用相同的派生路径。区分网络是通过 RPC 端点实现的。余额查询：https://explorer.solana.com/address/${publicKey}。SOL 币不使用类似比特币和以太币那样传统的地址，它的“地址”就是公钥的 Base58 编码格式（字母大小写敏感），比如 9bAaNbiHziiPjxYJMaBJtNvqghUHUWSFk1THTvXMj5tu，遵循“私钥 → 公钥 → Base58 编码 = 账户标识”。账户需维持最低余额否则会被回收，2 年内未使用且余额不足的账户会被回收。1SOL=10^9 Lamports（十亿）。

开发 js 库@solana/web3.js 可从 https://app.unpkg.com/@solana/web3.js@1.98.4/files/lib 下载，即 index.iife.min.js。

https://solana.com/developers/cookbook

测试水龙头：https://faucet.solana.com/（需要梯子）（需要 guthub 账户认证）

在线查询：Devnet: https://explorer.solana.com/?cluster=devnet、Mainnet: https://explorer.solana.com/

HD 钱包：

助记词：reform tenant kite attract exotic force juice leg fever toss fade ask idle nominee setup vanish summer first mobile bid caught goose swarm tower

钱包 1：

密钥：
CahgzRecDeBx4J1ddRvRKoMBozarjvcWf2RwWqkWXKm9neJ6KPKBnEmX5KasqUy73WFMZmQM6htWobD4z2mFthdtddRx7

地址：HjEy6ZKdy7bacZ4N87PkCRb2toRn7vGpvyQoYc1Zu7sz（也是公钥）（已在 devnet 充值 1SOL）

路径：m/44'/501'/0'/0'

业务签名（txid）：
yn5mw78n1yE54QtyJat2gQx4RJa9pQkfE854mgqwgRqaufcFoF8xb1oJfaK6zJkymH7uuncMpmt5WpVPUHXM6Rd

钱包 2：

密钥：
SAgbmAn37yv2c9EDg8D78u7M9zVxNTieLrUTmZ333VqADpzsWWQhEW8a8r4zueTLmhphbxfXyMLFuQuxLymGPvAhWcdC7

地址：Ac7fqMjTiogkoJPFtS3CtfFyWcfCZ65s9SEaEQrYm6Xx（也是公钥）（已在 devnet 充值 2.5SOL）

路径：m/44'/501'/0'/1'

附录 G：狗狗币

测试了，如果互联网出口在香港，就可以充值：https://faucet.doge.toys/

钱包 1：

私钥：chXcVtks2d5iWwneXFnypc6npXe1F3PbWviVT4KUzCoZYBZMLcUt

公钥：03098462cf1790123af2eeaf790586be4fe2a4ea675925d7e18fc6f8ba36118b64

地址：nXfTTzh9A4d5oPtxkFRYX14s3uZj4xPTm4（P2PKH）（充值了）

路径：m/44'/1'/0'/0/0

钱包 2：

私钥：cfoYyEDjh5R49AKQJ8gyN6XnQTfcSbh8g3FDVYTYiCsowPjfrniB

公钥：02ac6ae2ca5ff8696efb9a6a6a9dacfef7c9119df3ca625983eaf9d3ee4f121a65

地址：nqEug5ucDT7VQRBfABgrUTNrmNwoa8Pvxe（P2PKH）

路径：m/44'/1'/0'/0/1

钱包 3：

助记词：horror fat robot similar evidence exit script duck defy unique anchor clock agent art mushroom print royal list century defense add gorilla rebuild winter

私钥：cirJ6PinEV9BER9MSPKe6qQmkFyvxUkv7Hzy8ujRexUP95nD9WaM

公钥：032a7ed1d4ce2ee10a5e6305209510b07276d9e47c5936531a7b0a0cb1d3ee0722

地址：noZwGKDz6k9htdxnahx3cSWcAeriqQvNeN（P2PKH）（充值了）

路径：m/44'/1'/0'/0/0