# 6.1 简介

比特币交易是比特币系统中最重要的部分。根据比特币系统的设计原理，系统中任何其他的部分都是为了确保比特币交易可以被生成、能在比特币网络中得以传播和通过验证，并最终添加入全球比特币交易总账簿（比特币区块链）。比特币交易的本质是数据结构，这些数据结构中含有比特币交易参与者价值转移的相关信息。比特币区块链是一本全球复式记账总账簿，每个比特币交易都是在比特币区块链上的一个公开记录。

在这一章，我们将会剖析比特币交易的多种形式、所包含的信息、如何被创建、如何被验证以及如何成为所有比特币交易永久记录的一部分。当我们在本章中使用术语“钱包”时，我们指的是构建交易的软件，而不仅仅是密钥的数据库。

# 6.2交易细节

在[第二章比特币概述]中，我们使用区块浏览器查看了Alice曾经在Bob的咖啡店（Alice与Bob's Cafe的交易）支付咖啡的交易。

区块浏览器应用程序显示从Alice的“地址”到Bob的“地址”的交易。 这是一个非常简化的交易中包含的内容。 实际上，正如我们将在本章中看到的，所显示的大部分信息都是由区块浏览器构建的，实际上并不在交易中。

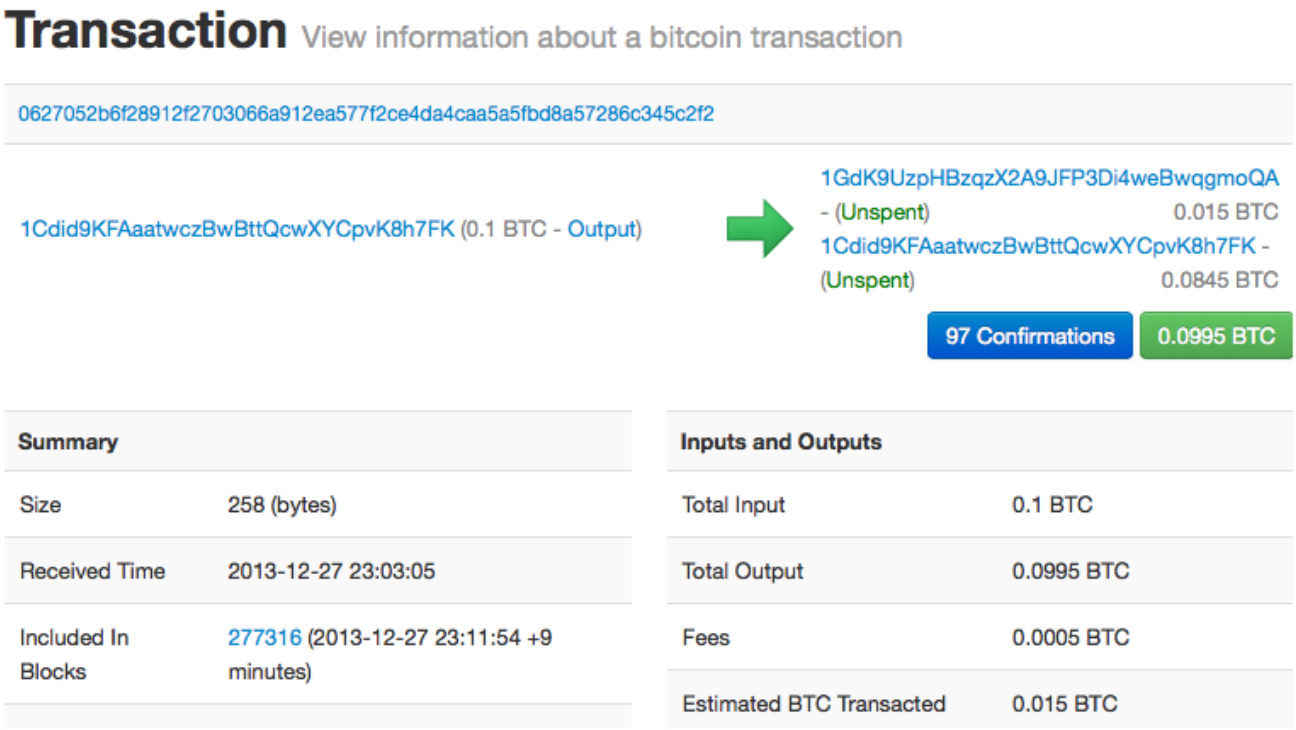



图1. Alice与Bob's Cafe的交易

## 6.2.1交易 - 幕后细节

在幕后，实际的交易看起来与典型的区块浏览器提供的交易非常不同。 事实上，我们在各种比特币应用程序用户界面中看到的大多数高级结构实际上并不存在于比特币系统中。

我们可以使用Bitcoin Core的命令行界面（getrawtransaction和decodeawtransaction）来检索Alice的“原始”交易，对其进行解码，并查看它包含的内容。 结果如下： Alice的交易被解码后是这个样子：

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid":"7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig":
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff
08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3
457eee41c04f4938de5cc17b4a10fa336a8d752adf",
      "sequence": 4294967295
    }
 ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
OP_EQUALVERIFY OP_CHECKSIG",
    }
  ]
}
```


您可能会注意到这笔交易似乎少了些什么东西，比如：Alice的地址在哪里？Bob的地址在哪里？ Alice发送的“0.1”个币的输入在哪里？ 在比特币里，没有具体的货币，没有发送者，没有接收者，没有余额，没有帐户，没有地址。为了使用者的便利，以及使事情更容易理解，所有这些都构建在更高层次上。

你可能还会注意到很多奇怪和难以辨认的字段以及十六进制字符串。 不必担心，本章将详细介绍这里所示的各个字段。

# 6.3交易的输入输出

比特币交易中的基础构建单元是交易输出。 交易输出是比特币不可分割的基本组合，记录在区块上，并被整个网络识别为有效。 比特币完整节点跟踪所有可找到的和可使用的输出，称为“未花费的交易输出”（unspent transaction outputs），即UTXO。 所有UTXO的集合被称为UTXO集，目前有数百万个UTXO。 当新的UTXO被创建，UTXO集就会变大，当UTXO被消耗时，UTXO集会随着缩小。每一个交易都代表UTXO集的变化（状态转换）。

当我们说用户的钱包已经“收到”比特币时，我们的意思是，钱包已经检测到了可用的UTXO。通过钱包所控制的密钥，我们可以把这些UTXO花出去。 因此，用户的比特币“余额”是指用户钱包中可用的UTXO总和，而他们可能分散在数百个交易和区块中。 “一个用户的比特币余额”，这个概念是比特币钱包应用创建的派生之物。比特币钱包通过扫描区块链并聚集所有属于该用户的UTXO来计算该用户的余额 。大多数钱包维护一个数据库或使用数据库服务来存储所有UTXO的快速参考集，这些UTXO由用户所有的密钥来控制花费行为。

一个UTXO可以是1“聪”（satoshi）的任意倍数（整数倍）。就像美元可以被分割成表示两位小数的“分”一样，比特币可以被分割成八位小数的“聪”。尽管UTXO可以是任意值，但一旦被创造出来，即不可分割。这是UTXO值得被强调的一个重要特性：UTXO是面值为“聪”的离散（不连续）且不可分割的价值单元，一个UTXO只能在一次交易中作为一个整体被消耗。

如果一个 UTXO比一笔交易所需量大，它仍会被当作一个整体而消耗掉，但同时会在交易中生成零头。例如，你有一个价值20比特币的 UTXO并且想支付1比特币，那么你的交易必须消耗掉整个20比特币的UTXO，并产生两个输出：一个支付了1比特币给接收人，另一个支付了19比特币的找零到你的钱包。这样的话，由于UTXO（或交易输出）的不可分割特性，大部分比特币交易都会产生找零。

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

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

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

一笔交易会消耗先前的已被记录（存在）的UTXO，并创建新的UTXO以备未来的交易消耗。通过这种方式，一定数量的比特币价值在不同所有者之间转移，并在交易链中消耗和创建UTXO。一笔比特币交易通过使用所有者的签名来解锁UTXO，并通过使用新的所有者的比特币地址来锁定并创建UTXO。

从交易的输出与输入链角度来看，有一个例外，即存在一种被称为“币基交易”（Coinbase Transaction）的特殊交易，它是每个区块中的第一笔交易，这种交易存在的原因是作为对挖矿的奖励，创造出全新的可花费比特币用来支付给“赢家”矿工。这也就是为什么比特币可以在挖矿过程中被创造出来，我们将在“挖矿”这一章进行详述。

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

## 6.3.1 交易输出

每一笔比特币交易都会创造输出，并被比特币账簿记录下来。除特例之外（见“数据输出操作符”（OP_RETURN）），几乎所有的输出都能创造一定数量的可用于支付的比特币，也就是UTXO。这些UTXO被整个网络识别，所有者可在未来的交易中使用它们。

UTXO在UTXO集（UTXOset）中被每一个全节点比特币客户端追踪。 新的交易从UTXO集中消耗（花费）一个或多个输出。

交易输出包含两部分：

- 一定量的比特币，面值为“聪”（satoshis），是最小的比特币单位；
- 确定花费输出所需条件的加密难题（cryptographic puzzle）

这个加密难题也被称为锁定脚本(locking script), 见证脚本(witness script), 或脚本公钥 (scriptPubKey)。

有关交易脚本语言会在后面121页的“交易脚本和脚本语言”一节中详细讨论。

现在，我们来看看 Alice 的交易（之前的章节“交易 - 幕后”所示），看看我们是否可以找到并识别输出。 在 JSON 编码中，输出位于名为 vout 的数组（列表）中：

```
"vout": [
  {
    "value": 0.01500000,
    "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
OP_EQUALVERIFY
OP_CHECKSIG"
  },
  {
    "value": 0.08450000,
    "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
OP_EQUALVERIFY OP_CHECKSIG",
  }
]
```

如您所见，交易包含两个输出。 每个输出都由一个值和一个加密难题来定义。 在 Bitcoin Core 显示的编码中，该值显示以 bitcoin 为单位，但在交易本身中，它被记录为以 satoshis 为单位的整数。 每个输出的第二部分是设定支出条件的加密难题。 Bitcoin Core 将其显示为 scriptPubKey，并向我们展示了一个可读的脚本表示。

稍后将在脚本构造（Lock + Unlock）中讨论锁定和解锁UTXO的主题。 在 ScriptPubKey 中用于编辑脚本的脚本语言在章节Transaction Scripts（交易脚本）和Script Language（脚本语言）中讨论。 但在我们深入研究这些话题之前，我们需要了解交易输入和输出的整体结构。

### 6.3.1.1交易序列化 - 输出

当交易通过网络传输或在应用程序之间交换时，它们被序列化。 序列化是将内部的数据结构表示转换为可以一次发送一个字节的格式（也称为字节流）的过程。 序列化最常用于编码通过网络传输或用于文件中存储的数据结构。 交易输出的序列化格式**如下表所示**:

| Size | Field | Description |
|---|---|---|
| 8 bytes (little-endian) | Amount | Bitcoin value in satoshis ($10^{-8}$ bitcoin) |
| 1–9 bytes (VarInt) | Locking-Script Size | Locking-Script length in bytes, to follow |
| Variable | Locking-Script | A script defining the conditions needed to spend the output |

大多数比特币函数库和架构不会在内部将交易存储为字节流，因为每次需要访问单个字段时，都需要复杂的解析。为了方便和可读性，比特币函数库将交易内部存储在数据结构（通常是面向对象的结构）中。

从交易的字节流表示转换为函数库的内部数据结构表示的过程称为反序列化或交易解析。转换回字节流以通过网络传输、哈希化（hashing）或存储在磁盘上的过程称为序列化。大多数比特币函数库具有用于交易序列化和反序列化的内置函数。

看看是否可以从序列化的十六进制形式手动解码 Alice 的交易中，找到我们以前看到的一些元素。包含两个输出的部分在下面中已加粗显示:

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73
4d2804fe65fa35779000000008b483045022100884d142d86652a3f47
ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039
ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84
16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1
7b4a10fa336a8d752adfffffffff02**60e31600000000001976a914ab6**
**8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000**
**1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac** 00000000

这里有一些提示:

- 加粗显示的部分有两个输出，每个都如本节之前所述进行了序列化。
- 0.015比特币的价值是1,500,000 satoshis。 这是十六进制的16 e3 60。
- 在串行化交易中，值16 e3 60以小端（最低有效字节优先）字节顺序进行编码，所以它看起来像60 e3 16。
- scriptPubKey的长度为25个字节，以十六进制显示为19个字节。

## 6.3.2交易输入

交易输入将UTXO（通过引用）标记为将被消费，并通过解锁脚本提供所有权证明。

要构建一个交易，一个钱包从它控制的UTXO中选择足够的价值来执行被请求的付款。 有时一个UTXO就足够，其他时候不止一个。 对于将用于进行此付款的每个UTXO，钱包将创建一个指向UTXO的输入，并使用解锁脚本解锁它。

让我们更详细地看一下输入的组件。输入的第一部分是一个指向UTXO的指针，通过指向UTXO被记录在区块链中所在的交易的哈希值和序列号来实现。 第二部分是解锁脚本，钱包构建它用以满足设定在UTXO中的支出条件。 大多数情况下，解锁脚本是一个证明比特币所有权的数字签名和公钥，但是并不是所有的解锁脚本都包含签名。 第三部分是序列号，稍后再讨论。

考虑我们在之前交易幕后章节提到的例子。交易输入是一个名为 vin 的数组（列表）：

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff
08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3
457eee41c04f4938de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
]
```

如您所见，列表中只有一个输入（因为一个UTXO包含足够的值来完成此付款）。 输入包含四个元素:

- 一个交易ID，引用包含正在使用的UTXO的交易
- 一个输出索引（vout），用于标识来自该交易的哪个UTXO被引用（第一个为零）
- 一个 scriptSig（解锁脚本），满足放置在UTXO上的条件，解锁它用于支出
- 一个序列号（稍后讨论）

在 Alice 的交易中，输入指向的交易ID是：

```
7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18
```

输出索引是0（即由该交易创建的第一个UTXO）。解锁脚本由Alice的钱包构建，首先检索引用的UTXO，检查其锁定脚本，然后使用它来构建所需的解锁脚本以满足此要求。

仅仅看这个输入，你可能已经注意到，除了对包含它引用的交易之外，我们无从了解这个UTXO的任何内容。我们不知道它的价值（多少satoshi金额），我们不知道设置支出条件的锁定脚本。要找到这些信息，我们必须通过检索整个交易来检索被引用的UTXO。请注意，由于输入的值未明确说明，因此我们还必须使用被引用的UTXO来计算在此交易中支付的费用（参见后面交易费用章节）。

不仅仅是Alice的钱包需要检索输入中引用的UTXO。一旦将该交易广播到网络，每个验证节点也将需要检索交易输入中引用的UTXO，以验证该交易。

因为缺乏语境，交易本身似乎不完整。他们在输入中引用UTXO，但是没有检索到UTXO，我们无法知道输入的值或其锁定条件。当编写比特币软件时，无论何时解码交易以验证它或计算费用或检查解锁脚本，您的代码首先必须从块链中检索引用的UTXO，以构建隐含但不存在于输入的UTXO引用中的语境。例如，要计算支付总额的交易费，您必须知道输入和输出值的总和。但是，如果不检索输入中引用的UTXO，则不知道它们的值。因此，在单个交易中计算交易费用的简单操作，实际上涉及多个交易的多个步骤和数据。

我们可以使用与比特币核心相同的命令序列，就像我们在检索Alice的交易（getrawtransaction和decodeawtransaction）时一样。因此，我们可以得到在前面的输入中引用的UTXO，并查看：

输入中引用的来自Alice 以前的交易中的UTXO：

```
"vout": [
   {
     "value": 0.10000000,
     "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
OP_EQUALVERIFY OP_CHECKSIG"
   }
 ]
```

我们看到这个UTXO的值为0.1BTC，并且它有一个包含“OP_DUP OP_HASH160 ...”的锁定脚本（scriptPubKey）。

> **小贴士：**为了充分了解Alice的交易，我们必须检索引用以前的交易作为输入。 检索以前的交易和未花费的交易输出的函数是非常普遍的，并且存在于几乎每个比特币函数库和API中。

### 6.3.2.1交易序列化--交易输入

当交易被序列化以在网络上传输时，它们的输入被编码成字节流，如下表所示

| Size | Field | Description |
| --- | --- | --- |
| 32 bytes | Transaction Hash | Pointer to the transaction containing the UTXO to be spent |
| 4 bytes | Output Index | The index number of the UTXO to be spent; first one is 0 |
| 1–9 bytes (VarInt) | Unlocking-Script Size | Unlocking-Script length in bytes, to follow |
| Variable | Unlocking-Script | A script that fulfills the conditions of the UTXO locking script |
| 4 bytes | Sequence Number | Used for locktime or disabled (0xFFFFFFFF) |

与输出一样，我们来看看我们是否可以从序列化格式的 Alice 的交易中找到输入。 首先，将输入解码：

```
"vin":
[
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff
08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3
457eee41c04f4938de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
]
```

现在，我们来看看我们是否可以识别下面这些以十六进制表示法表示的字段：

0100000001**186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73**
**4d2804fe65fa35779000000008b483045022100884d142d86652a3f47**
**ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039**
**ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813**
**01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84**
**16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1**
**7b4a10fa336a8d752adfffffffff**0260e31600000000001976a914ab6
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000 000

**提示：**

- 交易ID以反转字节顺序序列化，因此以（十六进制）18开头，以79结尾
- 输出索引为4字节组的“0”，容易识别
- scriptSig的长度为139个字节，或十六进制为8b
- 序列号设置为FFFFFFFF，也容易识别

## 6.3.3 交易费

大多数交易包含交易费（矿工费），这是为了确保网络安全而给比特币矿工的一种补偿。费用本身也作为一个安全机制，使经济上不利于攻击者通过交易来淹没网络。对于挖矿、费用和矿工得到的奖励，在挖矿一章中将有更详细的讨论。

这一节解释交易费是如何被包含在一个典型的交易中的。大多数钱包自动计算并计入交易费。但是， 如果你以编程方式构造交易，或者使用命令行界面，你必须手动计算并计入这些费用。

交易费作为矿工打包（挖矿）一笔交易到下一个区块中的一种激励；同时作为一种抑制因素，通过对每一笔交易收取小额费用来防止对系统的滥用。成功挖到某区块的矿工将得到该区内包含的矿工费， 并将该区块添加至区块链中。

交易费是基于交易的千字节规模来计算的，而不是比特币交易的价值。总的来说，交易费是根据比特币网络中的市场力量确定的。矿工会依据许多不同的标准对交易进行优先级排序，包括费用，他们甚至可能在某些特定情况下免费处理交易。但大多数情况下，交易费影响处理优先级，这意味着有足够费用的交易会更可能被打包进下一个挖出的区块中；反之交易费不足或者没有交易费的交易可能会被推迟，基于尽力而为的原则在几个区块之后被处理，甚至可能根本不被处理。交易费不是强制的，而且没有交易费的交易最终也可能会被处理，但是，交易费将提高处理优先级。

随着时间的推移，交易费的计算方式以及在交易处理优先级上的影响已经产生了变化。起初，交易费是固定的，是网络中的一个固定常数。渐渐地，随着网络容量和交易量的不断变化，并可能受到来自市场力量的影响，收费结构开始放松。自从至少2016年初以来，比特币网络容量的限制已经造成交易之间的竞争，从而导致更高的费用，免费交易彻底成为过去式。零费用或非常低费用的交易鲜少被处理，有时甚至不会在网络上传播。

在 Bitcoin Core 中，费用传递政策由minrelaytxfee选项设置。 目前默认的minrelaytxfee是每千字节0.00001比特币或者millibitcoin的1%。 因此，默认情况下，费用低于0.0001比特币的交易是免费的，但只有在内存池有空间时才会被转发; 否则，会被丢弃。 比特币节点可以通过调整minrelaytxfee的值来覆盖默认的费用传策略。

任何创建交易的比特币服务，包括钱包，交易所，零售应用等，都必须实现动态收费。动态费用可以通过第三方费用估算服务或内置的费用估算算法来实现。如果您不确定，那就从第三方服务开始，如果您希望去除第三方依赖，您应当有设计和部署自己算法的经验。

费用估算算法根据网络能力和“竞争”交易提供的费用计算适当的费用。这些算法的范围从十分简单的（最后一个块中的平均值或中位数）到非常复杂的（统计分析）均有覆盖。他们估计必要的费用（以字节为单位），这将使得交易具有很高的可能性被选择并打包进一定数量的块内。大多数服务为用户提供高、中、低优先费用的选择。高优先级意味着用户支付更高的交易费，但交易可能就会被打包进下一个块中。中低优先级意味着用户支付较低的交易费，但交易可能需要更长时间才能确认。

许多钱包应用程序使用第三方服务进行费用计算。一个流行的服务是http://bitcoinfees.21.co，它提供了一个API和一个可视化图表，以satoshi / byte为单位显示了不同优先级的费用。

> **小贴士：**静态费用在比特币网络上不再可行。 设置静态费用的钱包将导致用户体验不佳，因为交易往往会被“卡住”，并不被确认。 不了解比特币交易和费用的用户因交易被“卡住” 而感到沮丧，因为他们认为自己已经失去了资金。

下面费用估算服务bitcoinfees.21.co中的图表显示了10个satoshi / byte增量的费用的实时估计，以及每个范围的费用交易的预期确认时间（分钟和块数）。 对于每个收费范围（例如，61-70 satoshi /字节），两个水平栏显示过去24小时（102,975）中未确认交易的数量（1405）和交易总数，费用在该范围内。 根据图表，此时推荐的高优先费用为80 satoshi / 字节，这可能导致交易在下一个块（零块延迟）中开采。 据合理判断，一笔常规交易的大小约为226字节，因此单笔交易建议费用为18,080 satoshis（0.00018080 BTC）。

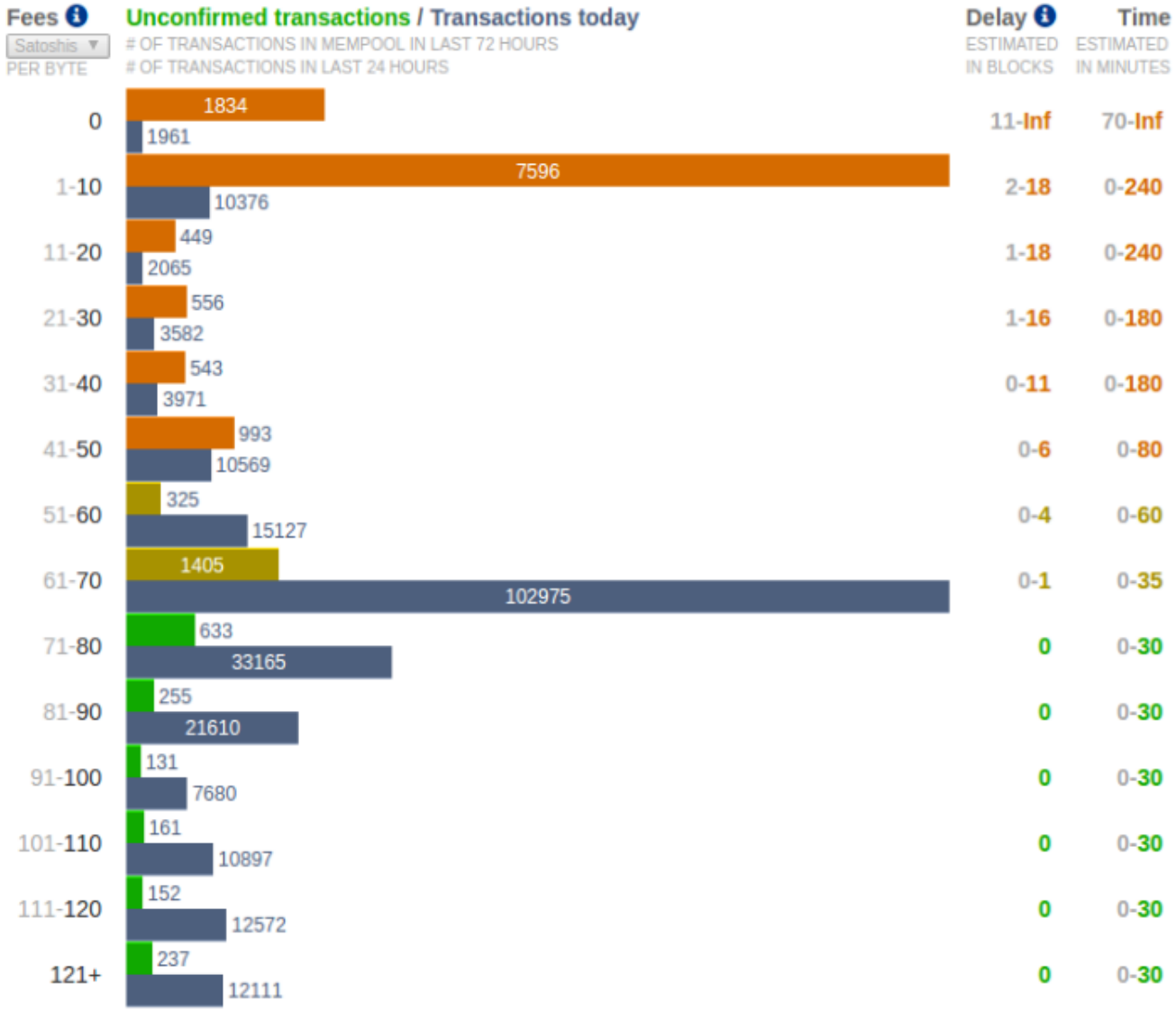
费用估算数据可以通过简单的HTTP REST API（https://bitcoinfees.21.co/api/v1/fees/recommended）来检索。例如，在命令行中使用curl命令：

运用费用估算API

```
$ curl https://bitcoinfees.21.co/api/v1/fees/recommended

{"fastestFee":80,"halfHourFee":80,"hourFee":60}
```

API通过费用估算以 satoshi per byte 的方式返回一个 JSON 对象，从而实现"最快确认" (fastestFee)，以及在三个块（halfHourFee）和六个块（hourFee）内确认。




## 6.3.4 把交易费加到交易中

交易的数据结构没有交易费的字段。相替代地，交易费是指输入和输出之间的差值。从所有输入中扣掉所有输出之后的多余的量会被矿工作为矿工费收集走：

> 交易费即输入总和减输出总和的余量：交易费 = 求和（所有输入） - 求和（所有输出）

正确理解交易比较困难，但又尤为重要。因为如果你要构建你自己的交易，你必须确保你没有因疏忽在交易中添加一笔大量交易费而大大减少了输入的可花费额。这意味着你必须计算所有的输入，如有必要则加上找零， 不然的话，结果就是你给了矿工一笔相当可观的劳务费！

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

> **警告：**如果你忘记了在手动构造的交易中增加找零的输出，系统会把找零当作交易费来处理。“不用找了！”也许不是你的真实意愿。

让我们重温一下Alice在咖啡店的交易来看看在实际中它如何运作。Alice想花0.015比特币购买咖啡。为了确保这笔交易能被立即处理，Alice想添加一笔交易费，比如说0.001。这意味着总花费会变成0.016。因此她的钱包需要凑齐 0.016或更多的UTXO。如果需要，还要加上找零。我们假设他的钱包有一个0.2比特币的UTXO可用。他的钱包就会消耗 掉这个UTXO，创造一个新的0.015的输出给Bob的咖啡店，另一个0.184比特币的输出作为找零回到Alice的钱包， 并留下未分配的0.001矿工费内含在交易中。

现在让我们看看另一种情况。Eugenia，我们在菲律宾的儿童募捐项目主管，完成了一次为孩子购买教材的筹款活动。她从世界范围内接收到了好几千份小额捐款，总额是50比特币。所以她的钱包塞满了非常小的UTXO。现在她想用比特币从本地的一家出版商购买几百本教材。

现在Eugenia的钱包应用想要构造一个单笔大额付款交易，它必须从可用的、由很多小数额构成的大的UTXO集合中寻求钱币来源。这意味着交易的结果是从上百个小额UTXO中作为输入，但只有一个输出用来付给出版商。输入数量这么巨大的交易会比一千字节要大，也许总尺寸会达到两至三千字节。结果是它将需要比中等规模交易要高得多的交易费。

Eugenia的钱包应用会通过测量交易的大小，乘以每千字节需要的费用来计算适当的交易费。很多钱包会支付较大的交易费，确保交易得到及时处理。更高交易费不是因为Eugenia付的钱很多，而是因为她的交易很复杂并且尺寸更大——交易费是与参加交易的比特币值无关的。

# 6.4比特币交易脚本和脚本语言

比特币交易脚本语言，称为脚本，是一种类似Forth的逆波兰表达式的基于堆栈的执行语言。 如果听起来不知所云，是你可能还没有学习20世纪60年代的编程语言，但是没关系，我们将在本章中解释这一切。 放置在UTXO上的锁定脚本和解锁脚本都以此脚本语言编写。 当一笔比特币交易被验证时，每一个输入值中的解锁脚本与其对应的锁定脚本同时 （互不干扰地）执行，以确定这笔交易是否满足支付条件。

脚本是一种非常简单的语言，被设计为在执行范围上有限制，可在一些硬件上执行，可能与嵌入式装置一样简单。它仅需要做最少的处理，许多现代编程语言可以做的花哨的事情它都不能做。 但用于验证可编程货币，这是一个经深思熟虑的安全特性。

如今，大多数经比特币网络处理的交易是以“Alice付给Bob”的形式存在，并基于一种称为“P2PKH”（Pay-toPublic-Key-Hash）脚本。但是，比特币交易不局限于“支付给Bob的比特币地址”的脚本。事实上，锁定脚本可以被编写成表达各种复杂的情况。为了理解这些更为复杂的脚本，我们必须首先了解交易脚本和脚本语言的基础知识。

在本节中，我们将会展示比特币交易脚本语言的各个组件；同时，我们也会演示如何使用它去表达简单的使用条件以及如何通过解锁脚本去满足这些花费条件。

> **小贴士：**比特币交易验证并不基于静态模式，而是通过脚本语言的执行来实现的。这种语言允许表达几乎无限的各种条件。这也是比特币作为一种“可编程的货币”所拥有的力量。

## 6.4.1 图灵非完备性

比特币脚本语言包含许多操作码，但都故意限定为一种重要的模式——除了有条件的流控制以外，没有循环或复杂流控制能力。这样就保证了脚本语言的图灵非完备性，这意味着脚本有限的复杂性和可预见的执行次数。脚本并不是一种通用语言，这些限制确保该语言不被用于创造无限循环或其它类型的逻辑炸弹，这样的炸弹可以植入在一笔交易中，引起针对比特币网络的“拒绝服务”攻击。记住，每一笔交易都会被网络中的全节点验证，受限制的语言能

防止交易验证机制被作为一个漏洞而加以利用。

## 6.4.2 去中心化验证

比特币交易脚本语言是没有中心化主体的，没有任何中心主体能凌驾于脚本之上，也没有中心主体会在脚本被执行后对其进行保存。所以执行脚本所需信息都已包含在脚本中。可以预见的是，一个脚本能在任何系统上以相同的方式执行。如果您的系统验证了一个脚本，可以确信的是每一个比特币网络中的其他系统也将验证这个脚本，这意味着一个有效的交易对每个人而言都是有效的，而且每一个人都知道这一点。这种结果可预见性是比特币系统的一项至关重要的良性特征。

## 6.4.3 脚本构建（锁定与解锁）

比特币的交易验证引擎依赖于两类脚本来验证比特币交易：锁定脚本和解锁脚本。

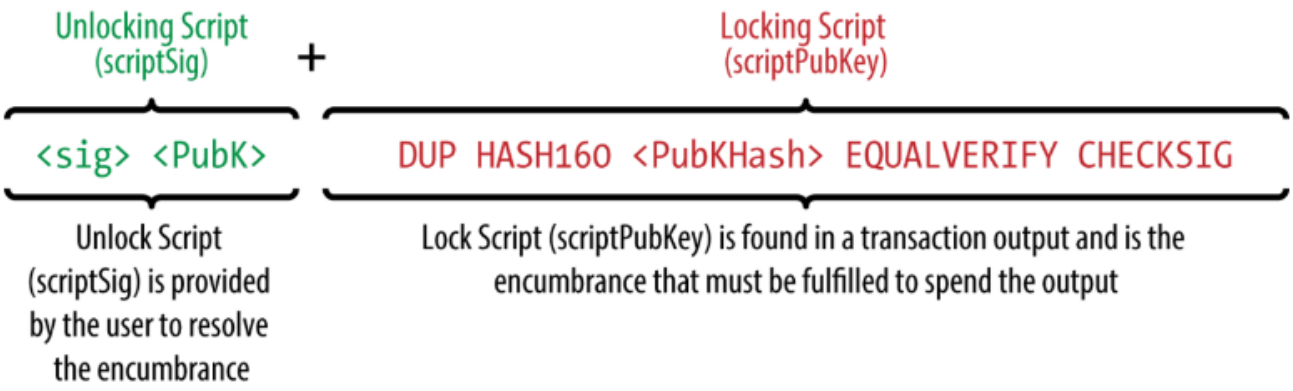
锁定脚本是一个放置在输出上面的花费条件：它指定了今后花费这笔输出必须要满足的条件。 由于锁定脚本往往含有一个公钥或比特币地址（公钥哈希值），在历史上它曾被称为脚本公钥（scriptPubKey）。由于认识到这种脚本技术存在着更为广泛的可能性，在本书中，我们将它称为“锁定脚本”（locking script）。在大多数比特币应用程序中，我们所称的“锁定脚本”将以scriptPubKey的形式出现在源代码中。您还将看到被称为见证脚本（witness script）的锁定脚本（参见[隔离见证]章节），或者更一般地说，它是一个加密难题（cryptographic puzzle）。 这些术语在不同的抽象层次上都意味着同样的东西。

解锁脚本是一个“解决”或满足被锁定脚本在一个输出上设定的花费条件的脚本，它将允许输出被消费。解锁脚本是每一笔比特币交易输入的一部分，而且往往含有一个由用户的比特币钱包（通过用户的私钥）生成的数字签名。由于解锁脚本常常包含一个数字签名，因此它曾被称作ScriptSig。在大多数比特币应用的源代码中，ScriptSig便是我们所说的解锁脚本。你也会看到解锁脚本被称作“见证”（witness 参见[隔离见证]章节）。在本书中，我们将它称为“解锁脚本”，用以承认锁定脚本的需求有更广的范围。但并非所有解锁脚本都一定会包含签名。

每一个比特币验证节点会通过同时执行锁定和解锁脚本来验证一笔交易。每个输入都包含一个解锁脚本，并引用了之前存在的UTXO。 验证软件将复制解锁脚本，检索输入所引用的UTXO，并从该UTXO复制锁定脚本。 然后依次执行解锁和锁定脚本。 如果解锁脚本满足锁定脚本条件，则输入有效（请参阅单独执行解锁和锁定脚本部分）。所有输入都是独立验证的，作为交易总体验证的一部分。

请注意，UTXO被永久地记录在区块链中，因此是不变的，并且不受在新交易中引用失败的尝试的影响。 只有正确满足输出条件的有效交易才能将输出视为“开销来源”，继而该输出将被从未花费的交易输出集（UTXO set）中删除。

下图是最常见类型的比特币交易（P2PKH:对公钥哈希的付款）的解锁和锁定脚本的示例，显示了在脚本验证之前从解锁和锁定脚本的并置产生的组合脚本：




### 6.4.3.1脚本执行堆栈

比特币的脚本语言被称为基于堆栈的语言，因为它使用一种被称为堆栈的数据结构。堆栈是一个非常简单的数据结构，可以被视为一叠卡片。栈允许两个操作：push和pop（推送和弹出）。 Push（推送）在堆栈顶部添加一个项目。 Pop（弹出）从堆栈中删除最顶端的项。栈上的操作只能作用于栈最顶端项目。堆栈数据结构也被称为“后进先出”（ Last-In-First-Out）或 “LIFO” 队列。

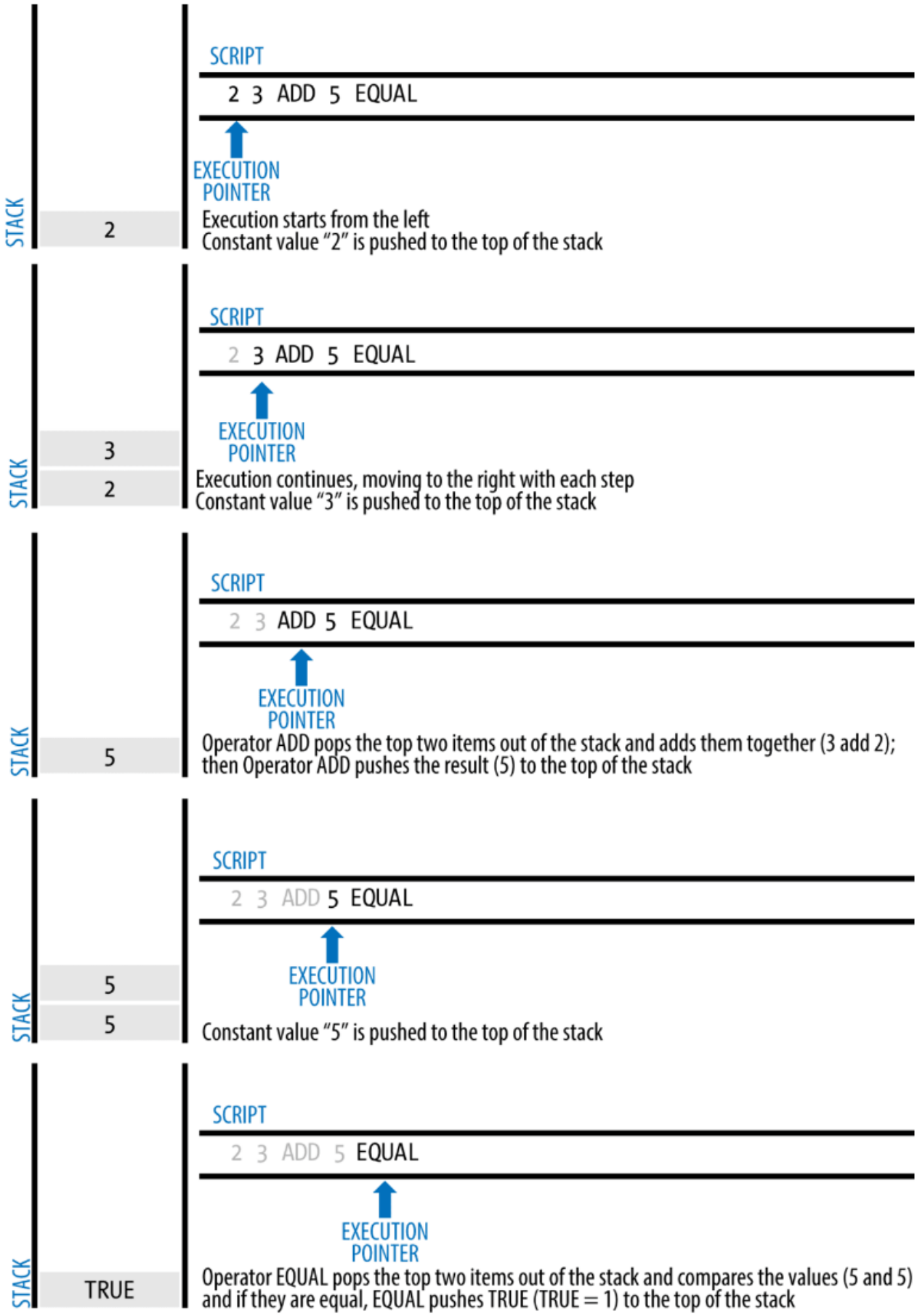
脚本语言通过从左到右处理每个项目来执行脚本。数字（数据常量）被推到堆栈上。操作码（Operators）从堆栈中推送或弹出一个或多个参数，对其进行操作，并可能将结果推送到堆栈上。例如，操作码 OP_ADD 将从堆栈中弹出两个项目，添加它们，并将结果的总和推送到堆栈上。

条件操作码（Conditional operators）对一个条件进行评估，产生一个 TRUE 或 FALSE 的布尔结果（boolean result）。例如， OP_EQUAL 从堆栈中弹出两个项目，如果它们相等，则推送为 TRUE（由数字1表示），否则推送为 FALSE（由数字0表示）。比特币交易脚本通常包含条件操作码，以便它们可以产生用来表示有效交易的 TRUE 结果。

### 6.4.3.2一个简单的脚本

现在让我们将学到的关于脚本和堆栈的知识应用到一些简单的例子中。

如图6-4，在比特币的脚本验证中，执行简单的数学运算时，脚本“ 2 3 OP*ADD 5 OP_EQUAL ”演示了算术加法操作码 OP_ADD ，该操作码将两个数字相加，然后把结果推送到堆栈， 后面的条件操作符 OP_EQUAL 是验算之前的两数之和是否等于 5 。为了简化起见，前缀OP*在一步步的演示过程中将被省略。有关可用脚本操作码和函数的更多详细信息，请参见[交易脚本]。


SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
2
Execution starts from the left
Constant value "2" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
3
2
Execution continues, moving to the right with each step
Constant value "3" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
5
Operator ADD pops the top two items out of the stack and adds them together (3 add 2);
then Operator ADD pushes the result (5) to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
5
5
Constant value "5" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
TRUE
Operator EQUAL pops the top two items out of the stack and compares the values (5 and 5)
and if they are equal, EQUAL pushes TRUE (TRUE = 1) to the top of the stack

尽管绝大多数解锁脚本都指向一个公钥哈希值（本质上就是比特币地址），因此如果想要使用资金则需验证所有权，但脚本本身并不需要如此复杂。任何解锁和锁定脚本的组合如果结果为真（TRUE），则为有效。前面被我们用于说明脚本语言的简单算术操作码同样也是一个有效的锁定脚本，该脚本能用于锁定交易输出。

使用部分算术操作码脚本作为锁定脚本的示例：

```
3 OP_ADD 5 OP_EQUAL
```

该脚本能被以如下解锁脚本为输入的一笔交易所满足：

```
2
```

验证软件将锁定和解锁脚本组合起来，结果脚本是：

```
2 3 OP_ADD 5 OP_EQUAL
```

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

> **小贴士：**如果堆栈顶部的结果显示为TRUE（标记为{{0x01}}），即为任何非零值，或脚本执行后堆栈为空情形，则交易有效。如果堆栈顶部的结果显示为FALSE（0字节空值，标记为{{}}）或脚本执行被操作码明确禁止，如OP_VERIFY、 OP_RETURN，或有条件终止如OP_ENDIF，则交易无效。详见[tx_script_ops]相关内容。

以下是一个稍微复杂一点的脚本，它用于计算 2+7-3+1 。注意，当脚本在同一行包含多个操作码时，堆栈允许一个操作码的结果由于下一个操作码执行。

```
2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL
```

请试着用纸笔自行演算脚本，当脚本执行完毕时，你会在堆栈得到正确的结果。

#### 6.4.3.3 解锁和锁定脚本的单独执行

在最初版本的比特币客户端中，解锁和锁定脚本是以连锁的形式存在，并被依次执行的。出于安全因素考虑，在2010 年比特币开发者们修改了这个特性——因为存在“允许异常解锁脚本推送数据入栈并且污染锁定脚本”的漏洞。而在当前的方案中，这两个脚本是随着堆栈的传递被分别执行的。下面将会详细介绍。

首先，使用堆栈执行引擎执行解锁脚本。如果解锁脚本在执行过程中未报错（例如：没有“悬挂”操作码），则复制主堆栈（而不是备用堆栈），并执行锁定脚本。如果从解锁脚本中复制而来的堆栈数据执行锁定脚本的结果为“TRUE"，那么解锁脚本就成功地满足了锁定脚本所设置的条件，因此，该输入是一个能使用该UTXO的有效授权。如果在合并脚本后的结果不是”TRUE“以外的任何结果，输入都是无效的，因为它不能满足UTXO中所设置的使用该笔资金的条件。

### 6.4.4 P2PKH（Pay-to-Public-Key-Hash）

比特币网络处理的大多数交易花费的都是由“付款至公钥哈希”（或P2PKH）脚本锁定的输出，这些输出都含有一个锁定脚本，将输入锁定为一个公钥哈希值，即我们常说的比特币地址。由P2PKH脚本锁定的输出可以通过提供一个公钥和由相应私钥创建的数字签名来解锁（使用）。参见数字签名ECDSA相关内容。

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

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

脚本中的 Cafe Public Key Hash 即为咖啡馆的比特币地址，但该地址不是基于Base58Check编码。事实上，大多数比特币地址的公钥哈希值都显示为十六进制码，而不是大家所熟知的以1开头的基于Bsase58Check编码的比特币地址。

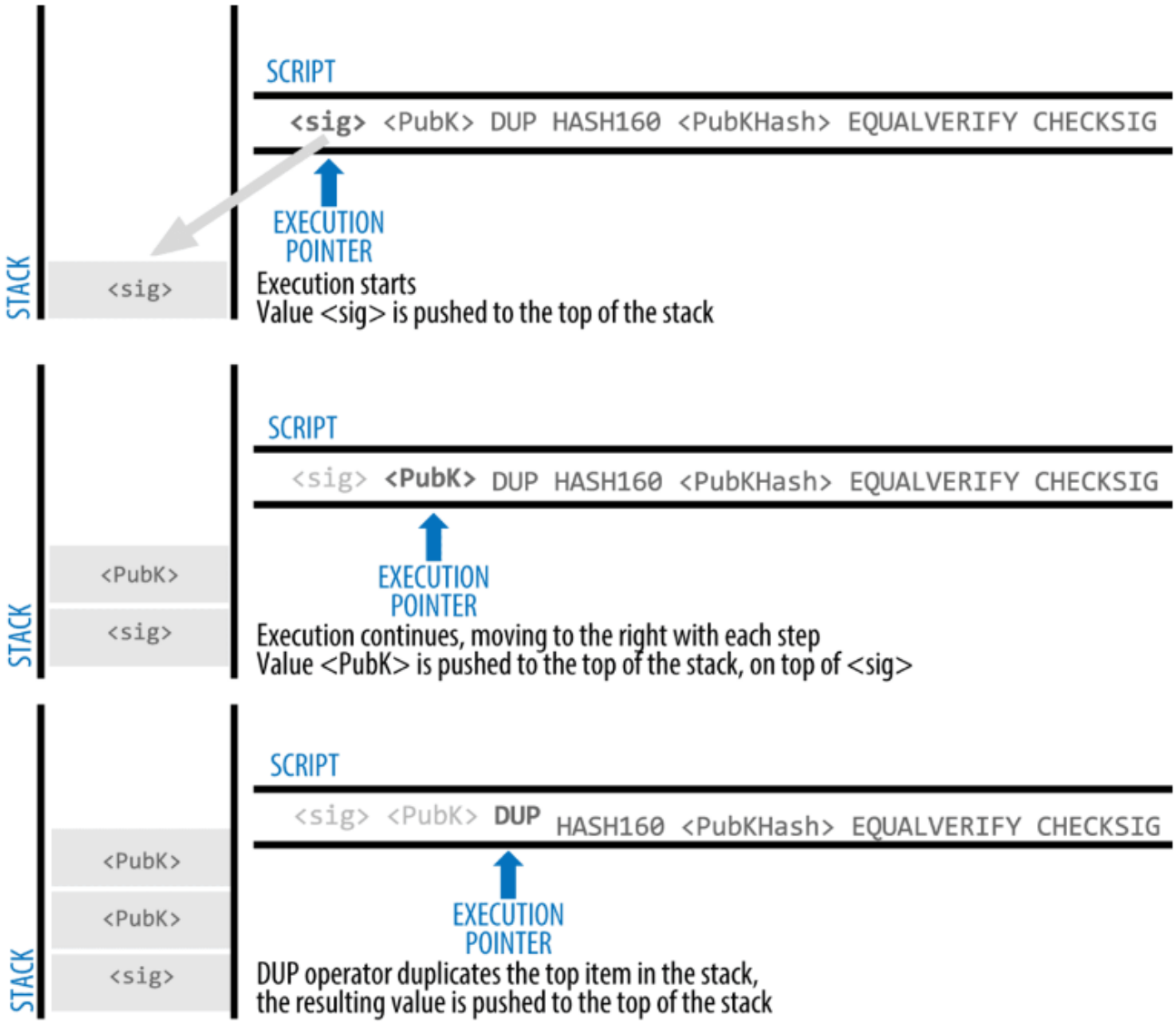
上述锁定脚本相应的解锁脚本是：

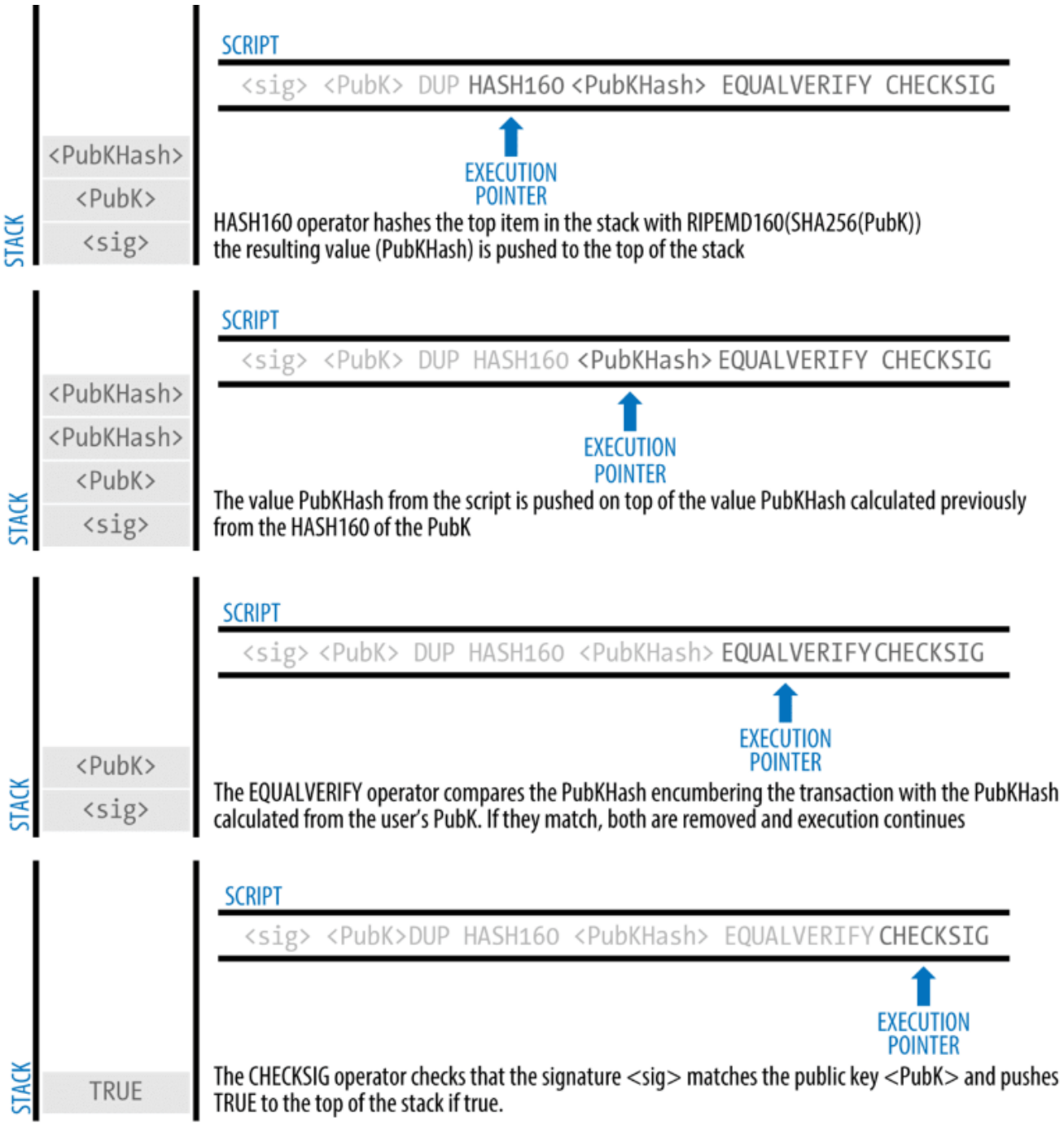
```
<Cafe Signature> <Cafe Public Key>
```

将两个脚本结合起来可以形成如下组合验证脚本：

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

只有当解锁脚本与锁定脚本的设定条件相匹配时，执行组合验证脚本时才会显示结果为真（TRUE）。换句话说，只有当解锁脚本得到了咖啡馆的有效签名，交易执行结果才会被通过（结果为真），该有效签名是从与公钥哈希相匹配的咖啡馆的私钥中所获取的。 图6-5和图6-6（分两部分）显示了组合脚本一步步检验交易有效性的过程。


SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<sig>
Execution starts
Value <sig> is pushed to the top of the stack
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubK>
<sig>
Execution continues, moving to the right with each step
Value <PubK> is pushed to the top of the stack, on top of <sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubK>
<PubK>
<sig>
DUP operator duplicates the top item in the stack,
the resulting value is pushed to the top of the stack




## 6.5数字签名（ECDSA）

到目前为止，我们还没有深入了解"数字签名"的细节。在本节中，我们将研究数字签名的工作原理，以及如何在不揭示私钥的情况下提供私钥的所有权证明。

比特币中使用的数字签名算法是椭圆曲线数字签名算法（Elliptic Curve Digital Signature Algorithm）或ECDSA。ECDSA是用于基于椭圆曲线私钥/公钥对的数字签名的算法，如椭圆曲线章节[elliptic_curve]所述。 ECDSA用于脚本函数OP_CHECKSIG，OP_CHECKSIGVERIFY，OP_CHECKMULTISIG和OP_CHECKMULTISIGVERIFY。每当你锁定脚本中看到这些时，解锁脚本都必须包含一个ECDSA签名。

数字签名在比特币中有三种用途（请参阅下面的侧栏）。第一，签名证明私钥的所有者，即资金所有者，已经授权支出这些资金。第二，授权证明是不可否认的（不可否认性）。第三，签名证明交易（或交易的具体部分）在签字之后没有也不能被任何人修改。

请注意，每个交易输入都是独立签名的。这一点至关重要，因为不管是签名还是输入都不必由同一“所有者”实施。事实上，一个名为 “CoinJoin” 的特定交易方案（多重签名方案？）就使用这个特性来创建多方交易来保护隐私。

> **注意：**每个交易输入和它可能包含的任何签名完全独立于任何其他输入或签名。 多方可以协作构建交易，并各自仅签一个输入。
>
> **维基百科对 “数字签名 ”的定义：**
>
> *数字签名是用于证明数字消息或文档的真实性的数学方案。 有效的数字签名给了一个容易接受的理由去相信：1）该消息是由已知的发送者（身份认证性）创建的； 2）发送方不能否认已发送消息（不可否认性； 3）消息在传输中未被更改（完整性）。*

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

# 6.5.1数字签名如何工作

数字签名是一种由两部分组成的数学方案：第一部分是使用私钥（签名密钥）从消息（交易）创建签名的算法； 第二部分是允许任何人验证签名的算法，给定消息和公钥。

## 6.5.1.1创建数字签名

在比特币的ECDSA算法的实现中，被签名的“消息”是交易，或更确切地说是交易中特定数据子集的哈希值（参见签名哈希类型（SIGHASH））。签名密钥是用户的私钥，结果是签名：

*((Sig = F{sig}(F{hash}(m), dA)))*

这里的：

- *dA* 是签名私钥
- *m* 是交易（或其部分）
- *F~hash~* 是散列函数
- *F~sig~* 是签名算法
- *Sig* 是结果签名

ECDSA数学**运算**的更多细节可以在ECDSA Math章节中找到。

函数*F~sig~* 产生由两个值组成的签名*Sig*，通常称为*R*和*S*：

```
Sig = (R, S)
```

现在已经计算了两个值R和S，它们就序列化为字节流，使用一种称为“分辨编码规则”（*Distinguished Encoding Rules*）或 DER的国际标准编码方案。

## 6.5.1.2签名序列化（DER）

我们再来看看Alice创建的交易。 在交易输入中有一个解锁脚本，其中包含Alice的钱包中的以下DER编码签名：

```
3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff0
8df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301
```

该签名是Alice的钱包生成的R和S值的序列化字节流，证明她拥有授权花费该输出的私钥。 序列化格式包含以下9个元素：

- *0x30*表示DER序列的开始
- *0x45* - 序列的长度（69字节）
- *0x02* - 一个整数值
- *0x21* - 整数的长度（33字节）
- *R-00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb*
- *0x02* - 接下来是一个整数
- *0x20* - 整数的长度（32字节）
- *S-4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813*
- 后缀（*0x01*）指示使用的哈希的类型（SIGHASH_ALL）

看看您是否可以使用此列表解码 Alice 的序列化（DER编码）签名。 重要的数字是R和S; 数据的其余部分是DER编码方案的一部分。

## 6.5.2验证签名

要验证签名，必须有签名（*R*和*S*）、序列化交易和公钥（对应于用于创建签名的私钥）。本质上，签名的验证意味着"只有生成此公钥的私钥的所有者，才能在此交易上产生此签名。"

签名验证算法采用消息（交易或其部分的哈希值）、签名者的公钥和签名（R和S值），如果签名对该消息和公钥有效，则返回 TRUE 值。

## 6.5.3签名哈希类型（SIGHASH）

数字签名被应用于消息，在比特币中，就是交易本身。签名意味着签字人对特定交易数据的承诺（*commitment*）。在最简单的形式中，签名适用于整个交易，从而承诺（commit）所有输入，输出和其他交易字段。但是，在一个交易中一个签名可以只承诺（*commit*）一个数据子集，这对于我们将在本节中看到的许多场景是有用的。

比特币签名具有指示交易数据的哪一部分包含在使用 SIGHASH 标志的私钥签名的哈希中的方式。 SIGHASH 标志是附加到签名的单个字节。每个签名都有一个SIGHASH标志，该标志在不同输入之间也可以不同。具有三个签名输入的交易可以具有不同SIGHASH标志的三个签名，每个签名签署（承诺）交易的不同部分。

记住，每个输入可能在其解锁脚本中包含一个签名。因此，包含多个输入的交易可以拥有具有不同SIGHASH标志的签名，这些标志在每个输入中承诺交易的不同部分。还要注意，比特币交易可能包含来自不同"所有者"的输入，他们在部分构建（和无效）的交易中可能仅签署一个输入，继而与他人协作收集所有必要的签名后再使交易生效。许多SIGHSASH标志类型，只有在你考虑到由许多参与者在比特币网络之外共同协作去更新仅部分签署了的交易，才具有意义。

有三个SIGHASH标志：ALL，NONE和SINGLE，如下表所示。

| SIGHASH flag | Value | Description |
|---|---|---|
| ALL | 0x01 | Signature applies to all inputs and outputs |
| NONE | 0x02 | Signature applies to all inputs, none of the outputs |
| SINGLE | 0x03 | Signature applies to all inputs but only the one output with the same index number as the signed input |

另外还有一个修饰符标志SIGHASH_ANYONECANPAY，它可以与前面的每个标志组合。 当设置ANYONECANPAY时，只有一个输入被签名，其余的（及其序列号）打开以进行修改。 ANYONECANPAY的值为0x80，并通过按位OR运算，得到如下所示的组合标志：

| SIGHASH flag | Value | Description |
|---|---|---|
| ALL\|ANYONECANPAY | 0x81 | Signature applies to one inputs and all outputs |
| NONE\|ANYONECANPAY | 0x82 | Signature applies to one inputs, none of the outputs |
| SINGLE\|ANYONECANPAY | 0x83 | Signature applies to one input and the output with the same index number |

SIGHASH标志在签名和验证期间应用的方式是建立交易的副本和删节其中的某些字段（设置长度为零并清空），继而生成的交易被序列化，SIGHASH标志被添加到序列化交易的结尾，并将结果哈希化，得到的哈希值本身即是被签名的“消息”。 基于SIGHASH标志的使用，交易的不同部分被删节。 所得到的哈希值取决于交易中数据的不同子集。 在哈希化前，SIGHASH作为最后一步被包含在内，签名也会对SIGHASH类型进行签署，因此不能更改（例如，被矿工）。

> **小贴士：**所有SIGHASH类型对应交易nLocktime字段（请参阅[transaction_locktime_nlocktime]部分）。 此外，SIGHASH类型本身在签名之前附加到交易，因此一旦签名就不能修改它。

在Alice的交易（参见序列化签名（DER）的列表）的例子中，我们看到DER编码签名的最后一部分是01，这是SIGHASH_ALL标志。这会锁定交易数据，因此Alice的签名承诺的是所有的输入和输出状态。 这是最常见的签名形式。

我们来看看其他一些SIGHASH类型，以及如何在实践中使用它们：

*ALL | ANYONECANPAY*

这种构造可以用来做“众筹”交易，试图筹集资金的人可以用单笔输出来构建一个交易，单笔输出将“目标”金额付给众筹发起人。这样的交易显然是无效的，因为它没有输入。但是现在其他人可以通过添加自己的输入作为捐赠来修改它们，他们用ALL | ANYONECANPAY签署自己的输入，除非收集到足够的输入以达到输出的价值，交易无效，每次捐赠是一项“抵押”，直到募集整个目标金额才能由募款人收取。

*NONE*

该结构可用于创建特定数量的“不记名支票”或“空白支票”。它对输入进行承诺，但允许输出锁定脚本被更改。任何人都可以将自己的比特币地址写入输出锁定脚本并兑换交易。然而，输出值本身被签名锁定。

*NONE | ANYONECANPAY*

这种构造可以用来建造一个“吸尘器”。在他们的钱包中拥有微小UTXO的用户无法花费这些费用，因为手续费用超过了这些微小UTXO的价值。借助这种类型的签名，微小UTXO可以为任何人捐赠，以便随时随地收集和消费。

有一些修改或扩展SIGHASH系统的建议。作为Elements项目的一部分，一个这样的提案是Blockstream的Glenn Willen提出的Bitmask Sighash模式。这旨在为SIGHASH类型创建一个灵活的替代品，允许“任意的，输入和输出的矿工可改写位掩码”来表示“更复杂的合同预付款方案，例如已分配的资产交换中有变更的已签名的报价”。

> **注释:** 您不会在用户的钱包应用程序中看到SIGHASH标志作为一个功能呈现。 少数例外，钱包会构建P2PKH脚本，并使用SIGHASH_ALL标志进行签名。 要使用不同的SIGHASH标志，您必须编写软件来构造和签署交易。 更重要的是，SIGHASH标志可以被专用的比特币应用程序使用，从而实现新颖的用途。

## 6.5.4 ECDSA数学

如前所述，签名由产生由两个值*R*和*S*组成的签名的数学函数*F*~sig~ 创建。在本节中，我们将查看函数*F*~sig~ 的更多细节。

签名算法首先生成一个 *ephemeral*（临时）私公钥对。 在涉及签名私钥和交易哈希的变换之后，该临时密钥对用于计算*R*和*S*值。

临时密钥对基于随机数k，用作临时私钥。 从*k*，我们生成相应的临时公钥*P*（以P = k *G计算，与派生比特币公钥相同）;参见[pubkey]部分）。数字签名的R值则是临时公钥P的*x*坐标。

从那里，算法计算签名的S值，使得：

$$S = k^{-1} (Hash(m) + dA * R) \bmod p$$

其中：

- *k*是临时私钥
- *R*是临时公钥的x坐标
- *dA*是签名私钥
- *m*是交易数据
- *p*是椭圆曲线的主要顺序

验证是签名生成函数的倒数，使用R，S值和公钥来计算一个值P，该值是椭圆曲线上的一个点（签名创建中使用的临时公钥）：

$$P = S^{-1} * Hash(m) * G + S^{-1} * R * Qa$$

其中：

- *R*和*S*是签名值
- *Qa*是Alice的公钥
- *m*是签署的交易数据
- *G*是椭圆曲线发生器点

如果计算点*P*的*x*坐标等于*R*，则验证者可以得出结论，签名是有效的。

请注意，在验证签名时，私钥既不知道也不显示。

> **小贴士：**ECDSA的数学很复杂，难以理解。 网上有一些很棒的指南可能有帮助。 搜索"ECDSA解释"或尝试这个：http://bit.ly/2r0HhGB。

## 6.5.5随机性在签名中的重要性

如我们在ECDSA Math中所看到的，签名生成算法使用随机密钥*k*作为临时私有-公钥对的基础。 *k* 的值不重要，只要它是随机的。如果使用相同的值 *k* 在不同的消息（交易）上产生两个签名，那么签名私钥可以由任何人计算。在签名算法中重用相同的 *k* 值会导致私钥的暴露！

> **警告** 如果在两个不同的交易中，在签名算法中使用相同的值 *k*，则私钥可以被计算并暴露给世界！

这不仅仅是一个理论上的可能性。我们已经看到这个问题导致私人密钥在比特币中的几种不同实现的交易签名算法中的暴露。人们由于无意中重复使用 *k* 值而将资金窃取。重用 *k* 值的最常见原因是未正确初始化的随机数生成器。

为了避免这个漏洞，业界最佳实践不是用熵播种的随机数生成器生成 *k* 值，而是使用交易数据本身播种的确定性随机进程。这确保每个交易产生不同的 *k* 值。在互联网工程任务组（Internet Engineering Task Force）发布的RFC 6979中定义了 *k* 值的确定性初始化的行业标准算法。

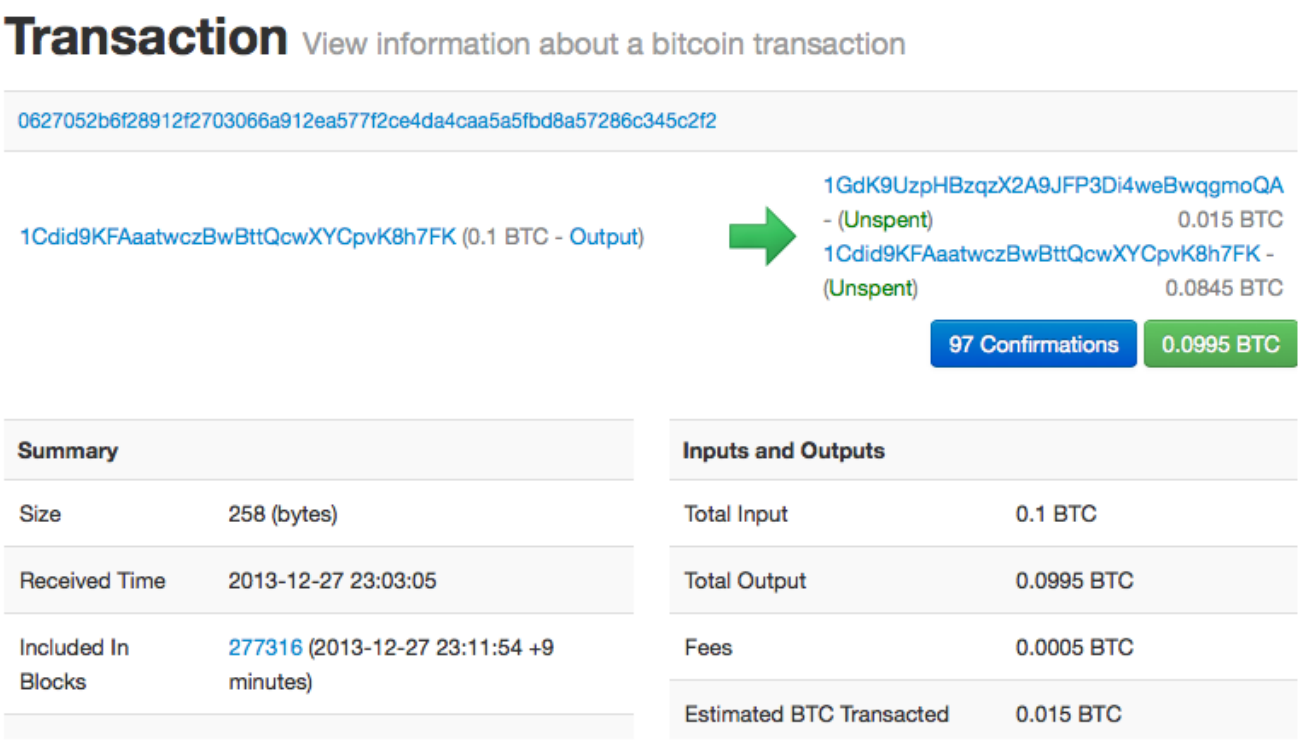
如果您正在实现一种用于在比特币中签署交易的算法，则必须使用RFC 6979或类似的确定性随机算法来确保为每个交易生成不同的 *k* 值。

# 6.6比特币地址，余额和其他摘要

在本章开始，我们发现交易的 “幕后”看起来与它在钱包、区块链浏览器和其它面向用户的应用程序中呈现的非常不同。 来自前几章的许多简单而熟悉的概念，如比特币地址和余额，似乎在交易结构中不存在。 我们看到交易本身并不包含比特币地址，而是通过锁定和解锁比特币离散值的脚本进行操作。 这个系统中的任何地方都不存在余额，而每个钱包应用程序都明明白白地显示了用户钱包的余额。

现在我们已经探讨了一个比特币交易中实际包含的内容，我们可以检查更高层次的抽象概念是如何从交易的看似原始的组成部分中派生出来的。

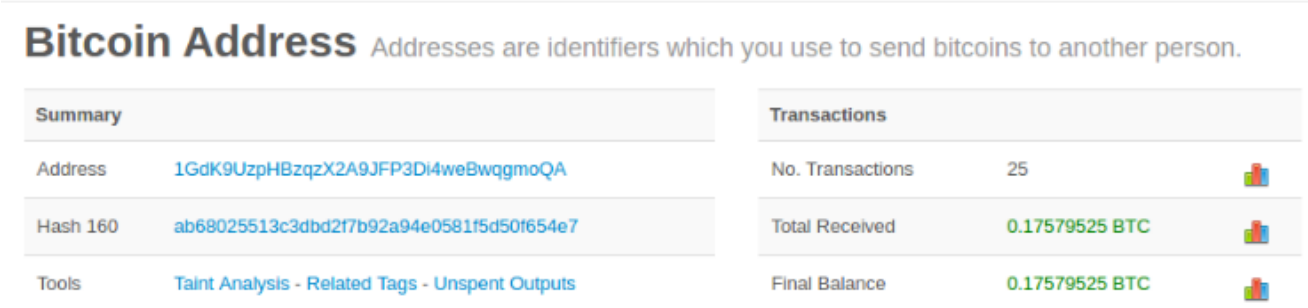
我们再来看看Alice的交易是如何在一个受欢迎的区块浏览器（前面章节Alice与Bob's Cafe的交易）中呈现的：




在交易的左侧，区块浏览器将Alice的比特币地址显示为“发送者”。其实这个信息本身并不在交易中。当区块链接浏览器检索到交易时，它还检索在输入中引用的先前交易，并从该旧交易中提取第一个输出。在该输出内是一个锁定脚本，将UTXO锁定到Alice的公钥哈希（P2PKH脚本）。块链浏览器提取公钥哈希，并使用Base58Check编码对其进行编码，以生成和显示表示该公钥的比特币地址。

同样，在右侧，区块浏览器显示了两个输出;第一个到Bob的比特币地址，第二个到Alice的比特币地址（作为找零）。再次，为了创建这些比特币地址，区块链浏览器从每个输出中提取锁定脚本，将其识别为P2PKH脚本，并从内部提取公钥哈希。最后，块链浏览器重新编码了使用Base58Check的公钥哈希生成和显示比特币地址。

如果您要点击Bob的比特币地址，则块链接浏览器将显示Bob的比特币地址的余额：




区块链浏览器显示了Bob的比特币地址的余额。但是比特币系统中却没有“余额”的概念。这么说吧，这里显示的余额其实是由区块链游览器按如下方式构建出来的：

为了构建“总接收”数量，区块链浏览器首先解码比特币地址的Base58Check编码，以检索编码在地址中的Bob的公钥的160位哈希值。然后，区块链浏览器搜索交易数据库，使用包含Bob公钥哈希的P2PKH锁定脚本寻找输出。通过总结所有输出的值，浏览器可以产生接收的总值。

完成构建当前余额（显示为“最终余额”）需要更多的工作。区块链接浏览器将当前未被使用的输入保存为一个分离的数据库——UTXO集。为了维护这个数据库，区块链浏览器必须监视比特币网络，添加新创建的UTXO，并在已被使用的UTXO出现在未经确认的交易中时，实时地删除它们。这是一个复杂的过程，不但要实时地跟踪交易在网络上的传播，同时还要保持与比特币网络的共识，确保在正确的链上。有时区块链浏览器未能保持同步，导致其对UTXO集的跟踪扫描不完整或不正确。

通过计算UTXO集，区块链浏览器总结了引用Bob的公钥哈希的所有未使用输出的值，并产生向用户显示的“最终余额”数目。

为了生成这张图片，得到这两个“余额”，区块链浏览器必须索引并搜索数十、数百甚至数十万的交易。

总之，通过钱包应用程序、区块链浏览器和其他比特币用户界面呈现给用户的信息通常源于更高层次的，通过搜索许多不同的交易，检查其内容以及操纵其中包含的数据而的抽象而构成。为了呈现出比特币交易类似于银行支票从发送人到接收人的这种简单视图，这些应用程序必须抽象许多底层细节。他们主要关注常见的交易类型：每个输入上具有SIGHASH_ALL签名的P2PKH。因此，虽然比特币应用程序以易于阅读的方式呈现所有了80%以上的交易，但有时候会被偏离了常规的交易 难住。包含更复杂的锁定脚本，或不同SIGHASH标志，或多个输入和输出的交易显示了这些抽象的简单性和弱点。

每天都有数百个不包含P2PKH输出的交易在块上被确认。 blockchain浏览器经常向他们发出红色警告信息，表示无法解码地址。以下链接包含未完全解码的最新的“奇怪交易”：https：//blockchain.info/strange-transactions。

正如我们将在下一章中看到的，这些并不一定是奇怪的交易。它们是包含比常见的P2PKH更复杂的锁定脚本的交易。我们将学习如何解码和了解更复杂的脚本及其支持的应用程序。