6交易

6.1 介绍

Transactions are the most important part of the bitcoin system. Everything else in bitcoin is designed to ensure that transactions can be created, propagated on the network, validated, and finally added to the global ledger of transactions (the blockchain). Transactions are data structures that encode the transfer of value between participants in the bitcoin system. Each transaction is a public entry in bitcoin's blockchain, the global double-entry bookkeeping ledger.
比特币交易是比特币系统中最重要的部分。
比特币中的所有其它东西都是为了保证：可以在网络上创建和传播交易，并最终添加到交易的全局账本中（区块链）。
交易是数据结构，它编码了比特币系统中参与者之间的价值转移。
每个交易都是比特币区块链中的一个公开账目，区块链是全局复式账本。

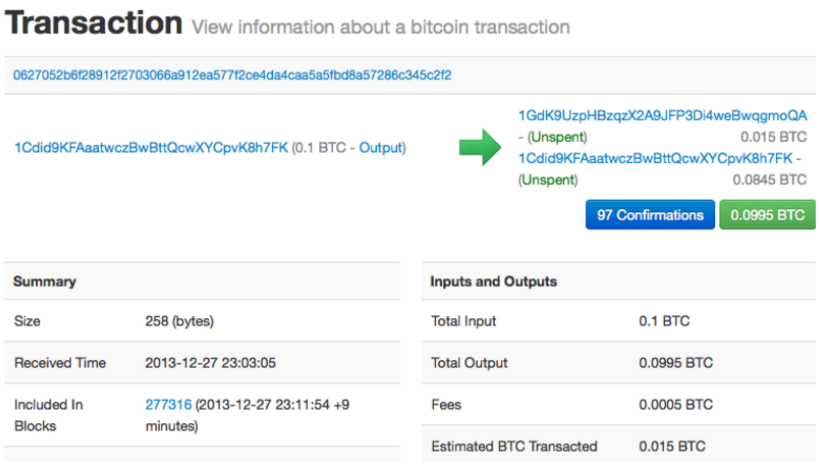
In this chapter we will examine all the various forms of transactions, what they contain, how to create them, how they are verified, and how they become part of the permanent record of all transactions. When we use the term "wallet" in this chapter, we are referring to the software that constructs transactions, not just the database of keys.
在本章中，我们将了解：各种形式的比特币交易、这些交易包含什么、如何创建这些交易、如何验证交易、交易如何成为永久记录的一部分。
本章中使用术语“钱包”时，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

In [ch02_bitcoin_overview], we looked at the transaction Alice used to pay for coffee at Bob's coffee shop using a block explorer (Alice's transaction to Bob's Cafe).
在第二章中，我们用区块浏览器看到了这个交易：Alice向Bob的咖啡店付款。

The block explorer application shows a transaction from Alice's "address" to Bob's "address." This is a much simplified view of what is contained in a transaction. In fact, as we will see in this chapter, much of the information shown is constructed by the block explorer and is not actually in the transaction.
区块浏览器应用程序显示：一个交易从Alice的“地址”到Bob的“地址”。
这是一个非常简化视图，说明了一个交易中包含什么。
实际上，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。

Figure 1. Alice's transaction to Bob's Cafe



6.2.1交易的幕后

Behind the scenes, an actual transaction looks very different from a transaction provided by a typical block explorer. In fact, most of the high-level constructs we see in the various bitcoin application user interfaces *do not actually exist* in the bitcoin system.
在幕后，实际的交易看起来与区块浏览器提供的交易非常不同。
实际上，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，并不存在于比特币系统中。

We can use Bitcoin Core's command-line interface (getrawtransaction and decoderawtransaction) to retrieve Alice's "raw" transaction, decode it, and see what it contains. The result looks like this:
我们可以使用Bitcoin Core的命令行界面（getrawtransaction和decodeawtransaction）来查看Alice的原始交易，对交易进行解码，看看交易包含什么。结果如下。

Alice's transaction decoded
解码后的Alice的交易。

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6
e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",
      "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",
    }
  ]
}
```

You may notice a few things about this transaction, mostly the things that are missing! Where is Alice's address? Where is Bob's address? Where is the 0.1 input "sent" by Alice?
你可能注意到关于这个交易的一些事情，主要是缺少了东西!

- Alice的地址在哪里?
- Bob的地址在哪里?
- Alice发送的0.1比特币输入在哪里?

In bitcoin, there are no coins, no senders, no recipients, no balances, no accounts, and no addresses. All those things are constructed at a higher level for the benefit of the user, to make things easier to understand.
在比特币中，没有比特币、发送者、接收者、余额、帐户、地址。
这些东西都是为了用户看起来方便而构造的高级视图，使客户更容易理解。

You may also notice a lot of strange and indecipherable fields and hexadecimal strings. Don't worry, we will explain each field shown here in detail in this chapter.
你可能还会注意到很多奇怪和难以辨认的字段，以及16进制字符串。
不必担心，本章将详细介绍这里显示的每个字段。

6.3交易的输入输出

The fundamental building block of a bitcoin transaction is a *transaction output*. Transaction outputs are indivisible chunks of bitcoin currency, recorded on the blockchain, and recognized as valid by the entire network. Bitcoin full nodes track all available and spendable outputs, known as *unspent transaction outputs*, or *UTXO*. The collection of all UTXO is known as the *UTXO set* and currently numbers in the millions of UTXO. The UTXO set grows as new UTXO is created and shrinks when UTXO is consumed. Every transaction represents a change (state transition) in the UTXO set.
比特币交易的基础构建块是“交易输出”。
在比特币中，交易输出不可分割，它记录在区块链中，并被全网认为有效。
比特币全节点跟踪所有可用和可花费的输出，称为UTXO（未花费的交易输出）。
所有UTXO的集合被称为UTXO集，目前有数百万个UTXO。
当新的UTXO被创建时，UTXO集就会变大；当UTXO被消耗时，UTXO集会缩小。
每个交易表示UTXO集的一个改变（状态转换）。

When we say that a user’s wallet has "received" bitcoin, what we mean is that the wallet has detected an UTXO that can be spent with one of the keys controlled by that wallet. Thus, a user’s bitcoin "balance" is the sum of all UTXO that user’s wallet can spend and which may be scattered among hundreds of transactions and hundreds of blocks. The concept of a balance is created by the wallet application. The wallet calculates the user’s balance by scanning the blockchain and aggregating the value of any UTXO the wallet can spend with the keys it controls. Most wallets maintain a database or use a database service to store a quick reference set of all the UTXO they can spend with the keys they control.
当我们说用户钱包“收到”比特币时，意思是：钱包检测一个UTXO，这个UTXO可以用钱包控制的一个密钥来花费。
这样，用户的比特币余额就是用户钱包可以花费的所有UTXO的总和，它们可能分散在数百个交易和区块中。
余额的概念是比特币钱包创造的。钱包这样计算用户的余额：扫描区块链，聚合所有可以花费的UTXO的价值。
多数钱包维护一个数据库或使用数据库服务，来存储它们可以花费的所有UTXO的快速参考集。

A transaction output can have an arbitrary (integer) value denominated as a multiple of satoshis. Just as dollars can be divided down to two decimal places as cents, bitcoin can be divided down to eight decimal places as satoshis. Although an output can have any arbitrary value, once created it is indivisible. This is an important characteristic of outputs that needs to be emphasized: outputs are *discrete* and *indivisible* units of value, denominated in integer satoshis. An unspent output can only be consumed in its entirety by a transaction.
一个交易输出可以有1聪（satoshi）的任意整数倍。
就像美元可分成两位小数的“分”一样，比特币可分成八位小数的“聪”。
虽然一个输出可以有任意值，但一旦被创造出来，就不可分割。
这是输出的一个重要特点：输出是离散的和不可分割的值。
一个UTXO只能在一个交易中被完全消费。

If an UTXO is larger than the desired value of a transaction, it must still be consumed in its entirety and change must be generated in the transaction. In other words, if you have an UTXO worth 20 bitcoin and want to pay only 1 bitcoin, your transaction must consume the entire 20-bitcoin UTXO and produce two outputs: one paying 1 bitcoin to your desired recipient and another paying 19 bitcoin in change back to your wallet. As a result of the indivisible nature of transaction outputs, most bitcoin transactions will have to generate change.
如果一个 UTXO比一个交易所需值大，它仍会被完整地消费掉，同时在交易中生成找零。
例如，你有一个20比特币的 UTXO，想要支付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：支付1比特币给收款人，支付19比特币找零给你的钱包。

因为交易输出的不可分割属性，所以多数比特币交易都会产生找零。

Imagine a shopper buying a $1.50 beverage, reaching into her wallet and trying to find a combination of coins and bank notes to cover the $1.50 cost. The shopper will choose exact change if available e.g. a dollar bill and two quarters (a quarter is $0.25), or a combination of smaller denominations (six quarters), or if necessary, a larger unit such as a $5 note. If she hands too much money, say $5, to the shop owner, she will expect $3.50 change, which she will return to her wallet and have available for future transactions.
想象一下，一位顾客要买1.5元的饮料。她掏出钱包，试图用零钱凑齐1.5 元。
如果可以的话，她会选刚刚好的零钱。
如果不行的话，她会用一张大额钞票，比如5元。
如果她把5元给了商店老板，会得到3.5元的找零，并把找零放回她的钱包以供以后交易使用。

Similarly, a bitcoin transaction must be created from a user's UTXO in whatever denominations that user has available. Users cannot cut an UTXO in half any more than they can cut a dollar bill in half and use it as currency. The user's wallet application will typically select from the user's available UTXO to compose an amount greater than or equal to the desired transaction amount.
类似的，一个比特币交易可以用用户可用的UTXO来创建。
用户不能把UTXO分成两半，就像不能把纸币撕成两半一样。
用户的钱包通常用用户的可用UTXO组成一个金额，大于或等于希望的交易金额。

As with real life, the bitcoin application can use several strategies to satisfy the purchase amount: combining several smaller units, finding exact change, or using a single unit larger than the transaction value and making change. All of this complex assembly of spendable UTXO is done by the user's wallet automatically and is invisible to users. It is only relevant if you are programmatically constructing raw transactions from UTXO.
就像现实生活一样，比特币应用可以使用一些策略来满足付款金额：用几个小额，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。
所有这些复杂的组成都是由用户钱包自动完成，用户看不到。
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。

A transaction consumes previously recorded unspent transaction outputs and creates new transaction outputs that can be consumed by a future transaction. This way, chunks of bitcoin value move forward from owner to owner in a chain of transactions consuming and creating UTXO.
一笔交易消费先前记录的UTXO，创建新的UTXO，可以在未来的交易中被消费。
通过这种方式，比特币价值从一个所有者转移给另一个所有者（消费和创建UTXO），形成一个交易链。

The exception to the output and input chain is a special type of transaction called the *coinbase* transaction, which is the first transaction in each block. This transaction is placed there by the "winning" miner and creates brand-new bitcoin payable to that miner as a reward for mining. This special coinbase transaction does not consume UTXO; instead, it has a special type of input called the "coinbase." This is how bitcoin's money supply is created during the mining process, as we will see in [mining].
对于输出与输入链来说，有一种特殊的交易，称为“币基交易”，它是每个区块的第一笔交易。
这种交易是由获胜的矿工放置的，给这个矿工创建了新的可花费比特币，它是挖矿的奖励。
这个特殊的币基交易不消费UTXO，它有一个特殊类型的输入，称为“币基”。
这就是挖矿过程中创建比特币货币供应的方式。

Tip: What comes first? Inputs or outputs, the chicken or the egg? Strictly speaking, outputs come first because coinbase transactions, which generate new bitcoin, have no inputs and create outputs from nothing.
提示：输入和输出，哪个先产生？先有鸡还是先有蛋？
严格来讲，先产生输出，因为币基交易（创造新比特币）没有输入，它是无中生有。

6.3.1交易输出

Every bitcoin transaction creates outputs, which are recorded on the bitcoin ledger. Almost all of these outputs, with one exception (see [op_return]) create spendable chunks of bitcoin called UTXO, which are then recognized by the whole network and available for the owner to spend in a future transaction.
每个比特币交易都会创建输出，并被比特币账本记录下来。
除特例之外（OP_RETURN），几乎所有的输出都能创建一定数量的比特币，称为UTXO。
UTXO被整个网络识别，拥有者可在以后的交易中使用它。

UTXO are tracked by every full-node bitcoin client in the UTXO set. New transactions consume (spend) one or more of these outputs from the UTXO set.
每个全节点比特币客户端在其UTXO集中跟踪UTXO。
新的交易从UTXO集中消耗（花费）一个或多个输出。

Transaction outputs consist of two parts:
- An amount of bitcoin, denominated in *satoshis*, the smallest bitcoin unit
- A cryptographic puzzle that determines the conditions required to spend the output

交易输出包含两部分：
- 一定量的比特币，面值为“聪”（比特币的最小单位）
- 一个加密谜题，它决定花费这个输出所需的条件

The cryptographic puzzle is also known as a *locking script*, a *witness script*, or a scriptPubKey.
这个加密谜题也被称为“锁定脚本”、“见证脚本”、“脚本公钥”。

The transaction scripting language, used in the locking script mentioned previously, is discussed in detail in Transaction Scripts and Script Language.
锁定脚本中使用的交易脚本语言会在后面详细讨论。

Now, let’s look at Alice’s transaction (shown previously in Transactions—Behind the Scenes) and see if we can identify the outputs. In the JSON encoding, the outputs are in an array (list) named vout:
现在，我们来看看 Alice 的交易，看看我们是否能认出输出。
在 JSON 编码中，输出位于名为 vout 的列表中。

```
"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",
  }
]
```

As you can see, the transaction contains two outputs. Each output is defined by a value and a cryptographic puzzle. In the encoding shown by Bitcoin Core, the value is shown in bitcoin, but in the transaction itself it is recorded as an integer denominated in satoshis. The second part of each output is the cryptographic puzzle that sets the conditions for spending. Bitcoin Core shows this as scriptPubKey and shows us a human-readable representation of the script.
如你所见，这个交易包含两个输出。每个输出有一个价值和一个加密难题。
在Bitcoin Core显示的编码中，价值的单位比特币，但在交易中，它被记录为以聪为单位的整数。

每个输出的第二部分是设定消费条件的加密难题。Bitcoin Core 将其显示为scriptPubKey，并向我们显示了这个脚本的一个可读表示。

The topic of locking and unlocking UTXO will be discussed later, in Script Construction (Lock + Unlock). The scripting language that is used for the script in scriptPubKey is discussed in Transaction Scripts and Script Language. But before we delve into those topics, we need to understand the overall structure of transaction inputs and outputs.
稍后将讨论脚本的锁定和解锁（Lock/Unlock）。
在 ScriptPubKey中使用的脚本语言也在后面讨论。
但这之前，我们需要理解交易输入和输出的整个结构。

6.3.1.1交易序列化：输出

When transactions are transmitted over the network or exchanged between applications, they are *serialized*. Serialization is the process of converting the internal representation of a data structure into a format that can be transmitted one byte at a time, also known as a byte stream. Serialization is most commonly used for encoding data structures for transmission over a network or for storage in a file. The serialization format of a transaction output is shown in Transaction output serialization.
当交易在网络上传输，或在应用之间交换时，它们被序列化。
序列化是将数据结构的内部表示转换为可以一次发送一个字节的格式（字节流）。
序列化最常用于编码在网络上传输的数据结构，或存储在一个文件中。

Table 1. Transaction output serialization
表1：交易输出序列化

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

Most bitcoin libraries and frameworks do not store transactions internally as byte-streams, as that would require complex parsing every time you needed to access a single field. For convenience and readability, bitcoin libraries store transactions internally in data structures (usually object-oriented structures).
多数比特币库和框架不会在内部把交易存储为字节流，因为那样的话，每次访问一个字段都需要复杂的分析。
为了方便和可读，比特币库在内部将交易存储在数据结构中，通常是面向对象的结构。

The process of converting from the byte-stream representation of a transaction to a library's internal representation data structure is called *deserialization* or *transaction parsing*. The process of converting back to a byte-stream for transmission over the network, for hashing, or for storage on disk is called *serialization*. Most bitcoin libraries have built-in functions for transaction serialization and deserialization.
将字节流表示转换为一个库的内部表示数据结构，称为反序列化，或交易解析。
为了网络传输、哈希、存储在磁盘上，转换成字节流的过程称为序列化。
多数比特币库有内置函数，用于交易序列化和反序列化。

See if you can manually decode Alice's transaction from the serialized hexadecimal form, finding some of the elements we saw previously. The section containing the two outputs is highlighted in Alice's transaction, serialized and presented in hexadecimal notation to help you:
看看你是否能从序列化的十六进制格式，手动解码 Alice 的交易，找出我们前面看到的一些元素。
在例1中，包含两个输出的部分被加粗。

Example 1. Alice's transaction, serialized and presented in hexadecimal notation

例1：Alice的交易，序列化并以16进制表示

```
0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73
4d2804fe65fa35779000000008b483045022100884d142d86652a3f47
ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039
ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84
16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1
7b4a10fa336a8d752adfffffffff0260e3160000000000019 76a914ab6
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
19 76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac 00000000
```

Here are some hints:

- There are two outputs in the highlighted section, each serialized as shown in Transaction output serialization.
- The value of 0.015 bitcoin is 1,500,000 satoshis. That's 16 e3 60 in hexadecimal.
- In the serialized transaction, the value 16 e3 60 is encoded in little-endian (least-significant-byte-first) byte order, so it looks like 60 e3 16.
- The scriptPubKey length is 25 bytes, which is 19 in hexadecimal.

这里有一些提示：

- 加粗的部分有两个输出，每个都如表1被序列化。
- 0.015比特币 = 1,500,000聪。16进制是16 e3 60。
- 在序列化交易中，值16 e3 60以小端字节序（最低有效字节优先）进行编码，所以是60 e3 16。
- scriptPubKey的长度为25个字节，16进制就是0x19。
- 第二个输出的值是80 ef d0，即 8,450,000聪 = 0.0845比特币。

6.3.2交易输入

Transaction inputs identify (by reference) which UTXO will be consumed and provide proof of ownership through an unlocking script.
交易输入指明消费哪个UTXO，并通过一个解锁脚本提供所有权证明。

To build a transaction, a wallet selects from the UTXO it controls, UTXO with enough value to make the requested payment. Sometimes one UTXO is enough, other times more than one is needed. For each UTXO that will be consumed to make this payment, the wallet creates one input pointing to the UTXO and unlocks it with an unlocking script.
要构建一个交易，钱包从它控制的UTXO中选择足够的钱来付款。
有时一个UTXO就够了，有时需要多个UTXO。
对于用于此付款的每个UTXO，钱包创建一个指向UTXO的输入，并使用一个解锁脚本对它解锁。

Let's look at the components of an input in greater detail. The first part of an input is a pointer to an UTXO by reference to the transaction hash and an output index, which identifies the specific UTXO in that transaction. The second part is an unlocking script, which the wallet constructs in order to satisfy the spending conditions set in the UTXO. Most often, the unlocking script is a digital signature and public key proving ownership of the bitcoin. However, not all unlocking scripts contain signatures. The third part is a sequence number, which will be discussed later.
让我们更详细地看一下输入的组件。

1. 一个指向UTXO的指针，方法是引用交易哈希和输出索引，输出索引标识了那个交易中的指定UTXO。
2. 一个解锁脚本，钱包构建它以满足UTXO中设置的支付条件。多数情况下，解锁脚本是一个数字签名和公钥，以证明比特币的所有权。但并不是所有的解锁脚本都包含签名。
3. 一个序列号，稍后再讨论。

Consider our example in Transactions—Behind the Scenes. The transaction inputs are an array (list) called vin:

考虑我们之前的例子。交易输入是一个名为 `vin` 的列表。

The transaction inputs in Alice's transaction
Alice的交易中的交易输入。

```
"vin": [
  {
    "txid":
"7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb0
2204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc54123363767
89d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
]
```

As you can see, there is only one input in the list (because one UTXO contained sufficient value to make this payment). The input contains four elements:

- A transaction ID, referencing the transaction that contains the UTXO being spent
- An output index (vout), identifying which UTXO from that transaction is referenced (first one is zero)
- A scriptSig, which satisfies the conditions placed on the UTXO, unlocking it for spending
- A sequence number (to be discussed later)

如你所见，列表中只有一个输入。这个输入包含四个元素：

- 交易ID，引用包含UTXO的交易
- 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）
- scriptSig（解锁脚本），满足UTXO设置的条件，为了支付而解锁它。
- 序列号（稍后讨论）

In Alice's transaction, the input points to the transaction ID:
在 Alice 的交易中，输入指向的交易ID。

```
7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18
```

and output index 0 (i.e., the first UTXO created by that transaction). The unlocking script is constructed by Alice's wallet by first retrieving the referenced UTXO, examining its locking script, and then using it to build the necessary unlocking script to satisfy it.
输出索引是0（即由该交易的第一个UTXO）。
解锁脚本由Alice的钱包构建：首先获得引用的UTXO，检查它的锁定脚本，然后构建满足它的解锁脚本。

Looking just at the input you may have noticed that we don't know anything about this UTXO, other than a reference to the transaction containing it. We don't know its value (amount in satoshi), and we don't know the locking script that sets the conditions for spending it. To find this information, we must retrieve the referenced UTXO by retrieving the underlying transaction. Notice that because the value of the input is not explicitly stated, we must also use the referenced UTXO in order to calculate the fees that will be paid in this transaction (see Transaction Fees).
仅看这个输入，你可能已经注意到，除了引用包含UTXO的交易之外，我们不知道这个UTXO的任何内容。我们不知道它的价值，不知道锁定脚本。
为了找到这些信息，我们必须通过检索底层交易来获取被引用的UTXO。
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交易费。

It's not just Alice's wallet that needs to retrieve UTXO referenced in the inputs. Once this transaction is broadcast to the network, every validating node will also need to retrieve the UTXO referenced in the transaction inputs in order to validate the transaction.
不仅是Alice的钱包需要获取输入中引用的UTXO。

一旦该交易被广播到网络上，每个验证节点也将需要获取交易输入中引用的UTXO，以验证该交易。

Transactions on their own seem incomplete because they lack context. They reference UTXO in their inputs but without retrieving that UTXO we cannot know the value of the inputs or their locking conditions. When writing bitcoin software, anytime you decode a transaction with the intent of validating it or counting the fees or checking the unlocking script, your code will first have to retrieve the referenced UTXO from the blockchain in order to build the context implied but not present in the UTXO references of the inputs.
因为缺乏语境，交易本身似乎不完整。
它们在输入中引用UTXO，但是如果没有获取UTXO，我们无法知道输入的值或锁定条件。
当编写比特币软件时，无论何时解码交易以验证它，或计算交易费，或检查解锁脚本，你的代码首先必须从区块链中获取引用的UTXO，以构建隐含但不存在于输入的UTXO的语境。

For example, to calculate the amount paid in fees, you must know the sum of the values of inputs and outputs. But without retrieving the UTXO referenced in the inputs, you do not know their value. So a seemingly simple operation like counting fees in a single transaction in fact involves multiple steps and data from multiple transactions.
例如，为了计算要支付的交易费，你必须知道输入总额和输出总额。
如果不获取输入中引用的UTXO，则不知道它们的值。
因此，一个看似简单的操作，例如计算一个交易中的交易费，实际上涉及多个步骤和来自多个交易的数据。

We can use the same sequence of commands with Bitcoin Core as we used when retrieving Alice's transaction (getrawtransaction and decoderawtransaction). With that we can get the UTXO referenced in the preceding input and take a look:
我们可以使用Bitcoin Core相同的命令序列，就像我们在获取Alice的交易（getrawtransaction和decodeawtransaction）时一样。
有了它，我们可以得到在前面的输入中引用的UTXO，并查看。

Alice's UTXO from the previous transaction, referenced in the input
输入中引用的来自以前交易的Alice的UTXO。

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

We see that this UTXO has a value of 0.1 BTC and that it has a locking script (scriptPubKey) that contains "OP_DUP OP_HASH160…".
这个UTXO的值为0.1比特币，并且有一个包含“OP_DUP OP_HASH160 ...”的锁定脚本。

Tip: To fully understand Alice's transaction we had to retrieve the previous transaction(s) referenced as inputs. A function that retrieves previous transactions and unspent transaction outputs is very common and exists in almost every bitcoin library and API.
提示：为了充分理解Alice的交易，我们必须获取输入引用的以前交易。
获取以前交易和UTXO的概念是很常见，在几乎每个比特币库和API中都有。

6.3.2.1交易序列化：输入

When transactions are serialized for transmission on the network, their inputs are encoded into a byte stream as shown in Transaction input serialization.
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。

Table 2. Transaction input serialization
表2：交易输入序列化

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)

As with the outputs, let's see if we can find the inputs from Alice's transaction in the serialized format. First, the inputs decoded:
与输出一样，我们来看看是否可以从序列化格式的 `Alice` 的交易中找到输入。
首先，解码输入。

```
"vin": [
  {
    "txid":
"7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb0
2204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc54123363767
89d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
],
```

Now, let's see if we can identify these fields in the serialized hex encoding in Alice's transaction, serialized and presented in hexadecimal notation:
现在，我们来看看是否可以在序列化编码中找出这些字段。

Example 2. Alice's transaction, serialized and presented in hexadecimal notation
例2：Alice的交易，序列化并以16进制表示

```
0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa3577900
000000 8b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75
c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336
376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adffffffff0260e31600
000000001976a914ab6
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000 000
```

Hints:
- The transaction ID is serialized in reversed byte order, so it starts with (hex) 18 and ends with 79
- The output index is a 4-byte group of zeros, easy to identify
- The length of the scriptSig is 139 bytes, or 8b in hex
- The sequence number is set to FFFFFFFF, again easy to identify

提示：
- 交易ID以反字节序被序列化，因此以0x18开头，以0x79结尾
- 输出索引为4字节组的“0”
- scriptSig的长度为139个字节
- 序列号为FFFFFFFF

6.3.3 交易费

Most transactions include transaction fees, which compensate the bitcoin miners for securing the network. Fees also serve as a security mechanism themselves, by making it economically infeasible for attackers to flood the network with transactions. Mining and the fees and rewards collected by miners are discussed in more detail in [mining].
多数交易包含交易费，这是对矿工确保网络安全而给他们的补偿。
交易费本身也是一个安全机制，使经济上不利于攻击者通过交易来淹没网络。

This section examines how transaction fees are included in a typical transaction. Most wallets calculate and include transaction fees automatically. However, if you are constructing transactions programmatically, or using a command-line interface, you must manually account for and include these fees.
本节解释交易费是如何被包含在一个典型的交易中。
多数钱包自动计算并包含交易费。
但是，如果你以编程方式构造交易，或者使用命令行接口，你必须手动计算并包含这些交易费。

Transaction fees serve as an incentive to include (mine) a transaction into the next block and also as a disincentive against abuse of the system by imposing a small cost on every transaction. Transaction fees are collected by the miner who mines the block that records the transaction on the blockchain.
交易费是对矿工把一笔交易包含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。
成功挖到区块的矿工将得到该区块内包含的矿工费，区块被记录到区块链中。

Transaction fees are calculated based on the size of the transaction in kilobytes, not the value of the transaction in bitcoin. Overall, transaction fees are set based on market forces within the bitcoin network. Miners prioritize transactions based on many different criteria, including fees, and might even process transactions for free under certain circumstances. Transaction fees affect the processing priority, meaning that a transaction with sufficient fees is likely to be included in the next block mined, whereas a transaction with insufficient or no fees might be delayed, processed on a best-effort basis after a few blocks, or not processed at all. Transaction fees are not mandatory, and transactions without fees might be processed eventually; however, including transaction fees encourages priority processing.
交易费是根据交易字节数（千字节）来计算的，而不是根据交易的比特币价值来计算。
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，包括交易费，甚至可以在某些情况下免费处理交易。
交易费影响处理优先级，意味着有足够交费易费的交易会更可能被包含在下一个区块；反之，交易费不足或没有交易费的交易可能会被推迟，在以后的区块中基于尽力而为的原则来处理，或者根本不处理。
交易费不是强制的，没有交易费的交易最终也可能会被处理；但是，有交易费能鼓励优先处理。

Over time, the way transaction fees are calculated and the effect they have on transaction prioritization has evolved. At first, transaction fees were fixed and constant across the network. Gradually, the fee structure relaxed and may be influenced by market forces, based on network capacity and transaction volume. Since at least the beginning of 2016, capacity limits in bitcoin have created competition between transactions, resulting in higher fees and effectively making free transactions a thing of the past. Zero fee or very low fee transactions rarely get mined and sometimes will not even be propagated across the network.
随着时间的推移，交易费的计算方式，以及它们对交易优先级的影响，也在产生变化。
最初，交易费是固定的，是网络中的一个常量。
渐渐地，随着网络容量和交易量的变化，缴费结构开始放松，可能受市场力量的影响。
自从2016年初以来，比特币的容量限制已经造成交易之间的竞争，导致了更高的交易费，免费交易已经成为过去。
零交易费或低交易费的交易很少被挖矿，有时甚至不会在网络上传播。

In Bitcoin Core, fee relay policies are set by the minrelaytxfee option. The current default minrelaytxfee is 0.00001 bitcoin or a hundredth of a millibitcoin per kilobyte. Therefore, by default, transactions with a fee less than 0.00001 bitcoin are treated as free and are only relayed if there is space in the mempool; otherwise, they are dropped. Bitcoin nodes can override the default fee relay policy by adjusting the value of minrelaytxfee.
在 Bitcoin Core 中，交易费传播策略是由minrelaytxfee选项设置的。
当前的缺省值是0.00001比特币，或每千字节0.00001比特币。
因此，默认情况下，交易费低于0.00001比特币的交易是免费的，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，来改变默认的交易费传播策略。

Any bitcoin service that creates transactions, including wallets, exchanges, retail applications, etc., *must* implement dynamic fees. Dynamic fees can be implemented through a third-party fee estimation service or with a built-in fee estimation algorithm. If you're unsure, begin with a third-party service and as you gain experience design and implement your own algorithm if you wish to remove the third-party dependency.
任何创建交易的比特币服务（包括钱包、交易所、零售应用等），都必须实现动态交易费。
动态费用可以通过第三方交易费估算服务或内置的交易费估算算法来实现。
如果你不确定，那就从第三方服务开始；当你获得经验后，如果不想用第三方服务，可以设计和部署自己算法。

Fee estimation algorithms calculate the appropriate fee, based on capacity and the fees offered by "competing" transactions. These algorithms range from simplistic (average or median fee in the last block) to sophisticated (statistical analysis). They estimate the necessary fee (in satoshis per byte) that will give a transaction a high probability of being selected and included within a certain number of blocks. Most services offer users the option of choosing high, medium, or low priority fees. High priority means users pay higher fees but the transaction is likely to be included in the next block. Medium and low priority means users pay lower transaction fees but the transactions may take much longer to confirm.
交易费估算算法计算合适的交易费，它根据容量和其它交易提供的交易费。
这些算法的可以很简单（最后一个区块中的平均值或中位数），也可以很复杂（统计分析）。
它们估算必要的交易费（聪/字节），使交易更可能被选择和包含在一定数量的区块内。
多数服务给用户提供了选择高、中、低优先级交易费的选项。
高优先级意味着用户支付更高的交易费，但交易可能被打包进下一个区块中。
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。

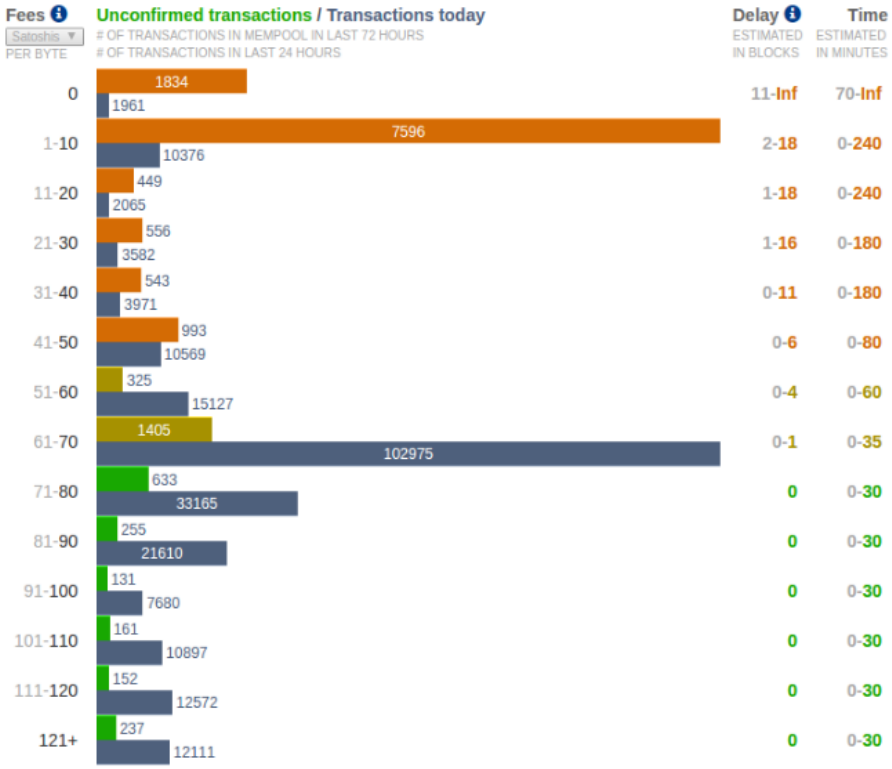
Many wallet applications use third-party services for fee calculations. One popular service is *http://bitcoinfees.21.co*, which provides an API and a visual chart showing the fee in satoshi/byte for different priorities.
许多钱包使用第三方服务计算交易费。
一个流行的服务是http://bitcoinfees.21.co
它提供了一个API和一个可视化图表，以"聪/字节"显示不同优先级的交易费。

Tip: Static fees are no longer viable on the bitcoin network. Wallets that set static fees will produce a poor user experience as transactions will often get "stuck" and remain unconfirmed. Users who don't understand bitcoin transactions and fees are dismayed by "stuck" transactions because they think they've lost their money.
提示：静态交易费在比特币网络上不再可行。
设置静态交易费的钱包给用户的体验不好，因为交易经常被卡住，不被确认。
不理解比特币交易和交易费的用户会因交易被卡住而感到沮丧，因为他们以为钱已经丢了。

The chart in Fee estimation service bitcoinfees.21.co shows the real-time estimate of fees in 10 satoshi/byte increments and the expected confirmation time (in minutes and number of blocks) for transactions with fees in each range. For each fee range (e.g., 61–70 satoshi/byte), two horizontal bars show the number of unconfirmed transactions (1405) and total number of transactions in the past 24 hours (102,975), with fees in that range. Based on the graph, the recommended high-priority fee at this time was 80 satoshi/byte, a fee likely to result in the transaction being mined in the very next block (zero block delay).

For perspective, the median transaction size is 226 bytes, so the recommended fee for a transaction size would be 18,080 satoshis (0.00018080 BTC).
图2显示了10聪/字节增量的交易费估算，以及期望的确认时间（分钟和区块数）。
对于每个收费范围（例如，61–70聪/字节），有两行，分别显示了未确认交易的数量（1405）和过去24小时的交易总数（102975）。
根据这个图，建议的高优先级交易费是80聪/字节，它会使交易在下一个区块（零区块延迟）中开采。
一般交易是226字节，因此单笔交易建议交易费为18,080聪。

Figure 2. Fee estimation service bitcoinfees.21.co
图2：交易费估算服务



The fee estimation data can be retrieved via a simple HTTP REST API, at https://bitcoinfees.21.co/api/v1/fees/recommended. For example, on the command line using the curl command:
交易费估算数据可以通过一个简单的HTTP REST API获取：https://bitcoinfees.21.co/api/v1/fees/recommended
例如，在命令行中使用curl命令：

Using the fee estimation API
使用交易费估算API

```
$ curl https://bitcoinfees.21.co/api/v1/fees/recommended
{"fastestFee":80,"halfHourFee":80,"hourFee":60}
```

The API returns a JSON object with the current fee estimate for fastest confirmation (fastestFee), confirmation within three blocks (halfHourFee) and six blocks (hourFee), in satoshi per byte.
这个API返回一个JSON对象，有当前交易费估计（聪/字节）：

- fastestFee ：最快确认
- halfHourFee：3个区块确认
- hourFee ：6个区块确认

6.3.4 把交易费加到交易中

The data structure of transactions does not have a field for fees. Instead, fees are implied as the difference between the sum of inputs and the sum of outputs. Any excess amount that remains after all outputs have been deducted from all inputs is the fee that is collected by the miners:
交易的数据结构没有交易费字段。
交易费是输入总额和输出总额之间的差值。
交易费会被矿工收走。

Transaction fees are implied, as the excess of inputs minus outputs:
交易费是隐含的，即输入减去输出。

```
Fees = Sum(Inputs) – Sum(Outputs)
```

This is a somewhat confusing element of transactions and an important point to understand, because if you are constructing your own transactions you must ensure you do not inadvertently include a very large fee by underspending the inputs. That means that you must account for all inputs, if necessary by creating change, or you will end up giving the miners a very big tip!
交易有一个容易混淆的地方，但理解它很重要。
因为如果你自己构建交易，必须确保你不会因没有花完输入，而给矿工提供了一大笔交易费。
这意味着，你必须计算所有的输入，如有必要则创建找零，不然的话，你就给了矿工一笔可观的小费！

For example, if you consume a 20-bitcoin UTXO to make a 1-bitcoin payment, you must include a 19-bitcoin change output back to your wallet. Otherwise, the 19-bitcoin "leftover" will be counted as a transaction fee and will be collected by the miner who mines your transaction in a block. Although you will receive priority processing and make a miner very happy, this is probably not what you intended.
例如，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。
尽管你的交易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。

Tip: If you forget to add a change output in a manually constructed transaction, you will be paying the change as a transaction fee. "Keep the change!" might not be what you intended.
提示：如果你在手工构建的交易中忘记了增加找零输出，系统会把找零当作交易费来处理。
“不用找零！”也许不是你的真实意愿。

Let’s see how this works in practice, by looking at Alice’s coffee purchase again. Alice wants to spend 0.015 bitcoin to pay for coffee. To ensure this transaction is processed promptly, she will want to include a transaction fee, say 0.001. That will mean that the total cost of the transaction will be 0.016. Her wallet must therefore source a set of UTXO that adds up to 0.016 bitcoin or more and, if necessary, create change. Let’s say her wallet has a 0.2-bitcoin UTXO available. It will therefore need to consume this UTXO, create one output to Bob’s Cafe for 0.015, and a second output with 0.184 bitcoin in change back to her own wallet, leaving 0.001 bitcoin unallocated, as an implicit fee for the transaction.
我们再来看看Alice在咖啡店的交易。
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。这意味着总花费是0.016。因此钱包要凑齐0.016或更多的UTXO。如果需要，还要创建找零。
假设钱包有一个0.2比特币的UTXO可用。钱包会消耗掉这个UTXO，创造一个0.015输出给Bob的咖啡店，另一个0.184比特币输出作为找零回到Alice的钱包，并留下未分配的0.001交易费内含在交易中。

Now let’s look at a different scenario. Eugenia, our children’s charity director in the Philippines, has completed a fundraiser to purchase schoolbooks for the children. She received several thousand small donations from people all around the world, totaling 50

bitcoin, so her wallet is full of very small payments (UTXO). Now she wants to purchase hundreds of schoolbooks from a local publisher, paying in bitcoin.
现在我们看看另一个场景。
Eugenia是菲律宾的儿童募捐项目主管，已经完成了一次为孩子购买教材的筹款活动。
她从世界各地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小额UTXO。
现在她想用比特币从本地的一家出版商那里购买几百本教材。

As Eugenia's wallet application tries to construct a single larger payment transaction, it must source from the available UTXO set, which is composed of many smaller amounts. That means that the resulting transaction will source from more than a hundred small-value UTXO as inputs and only one output, paying the book publisher. A transaction with that many inputs will be larger than one kilobyte, perhaps several kilobytes in size. As a result, it will require a much higher fee than the median-sized transaction.
Eugenia的钱包想要构建一个单笔大额付款交易，它必须来自可用的UTXO集，它由许多小额构成。
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。
输入数量很大，会超过一千字节，也许有几千字节。结果是需要比一般交易支付高得多的交易费。

Eugenia's wallet application will calculate the appropriate fee by measuring the size of the transaction and multiplying that by the per-kilobyte fee. Many wallets will overpay fees for larger transactions to ensure the transaction is processed promptly. The higher fee is not because Eugenia is spending more money, but because her transaction is more complex and larger in size—the fee is independent of the transaction's bitcoin value.
Eugenia的钱包会通过测量交易的大小，乘以每千字节需要的交易费，来计算适当的交易费。
很多钱包会对较大交易超额支付交易费，以确保交易得到及时处理。
更高交易费不是因为Eugenia付了更多的钱，而是因为她的交易更复杂、字节数更多，交易费与交易的价值无关。

6.4交易脚本和脚本语言

The bitcoin transaction script language, called *Script*, is a Forth-like reverse-polish notation stack-based execution language. If that sounds like gibberish, you probably haven't studied 1960s programming languages, but that's ok—we will explain it all in this chapter.
比特币交易脚本语言，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，是因为你可能没学过1960年代的编程语言，但没关系，我们会在本章中解释。

Both the locking script placed on an UTXO and the unlocking script are written in this scripting language. When a transaction is validated, the unlocking script in each input is executed alongside the corresponding locking script to see if it satisfies the spending condition.
“放置在UTXO上的锁定脚本”和“解锁脚本”都是用这种脚本语言编写的。
当验证一个交易时，把每个输入的解锁脚本和对应的锁定脚本放在一起，看看它是否满足支付条件。

Script is a very simple language that was designed to be limited in scope and executable on a range of hardware, perhaps as simple as an embedded device. It requires minimal processing and cannot do many of the fancy things modern programming languages can do. For its use in validating programmable money, this is a deliberate security feature.
脚本是一种非常简单的语言，限制在一定范围，可以在一些硬件（例如嵌入设备）上执行。
它需要很少的处理，不能做许多现代编程语言可以做的花哨的事情。
但用于验证可编程货币，这是一个经过深思熟虑的安全特性。

Today, most transactions processed through the bitcoin network have the form "Payment to Bob's bitcoin address" and are based on a script called a Pay-to-Public-Key-Hash script. However, bitcoin transactions are not limited to the "Payment to Bob's bitcoin address" script. In fact, locking scripts can be written to express a vast variety of complex conditions. In order to understand these more complex scripts, we must first understand the basics of transaction scripts and script language.
如今，比特币网络上处理的多数交易的形式是“Alice付款给Bob”，并且基于P2PKH脚本。
但是，比特币交易不局限于这种脚本。
实际上，锁定脚本可被编写成表达各种复杂的条件。
为了理解这些更复杂的脚本，我们必须首先了解交易脚本和脚本语言的基础。

In this section, we will demonstrate the basic components of the bitcoin transaction scripting language and show how it can be used to express simple conditions for spending and how those conditions can be satisfied by unlocking scripts.
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。

Tip: Bitcoin transaction validation is not based on a static pattern, but instead is achieved through the execution of a scripting language. This language allows for a nearly infinite variety of conditions to be expressed. This is how bitcoin gets the power of "programmable money."
提示：比特币交易验证不是基于静态模式，而是通过执行一个脚本语言来实现的。
这种语言允许表达几乎无限种条件，因此，比特币有了“可编程货币”的能力。

6.4.1图灵不完全性

The bitcoin transaction script language contains many operators, but is deliberately limited in one important way—there are no loops or complex flow control capabilities other than conditional flow control. This ensures that the language is not *Turing Complete*, meaning that scripts have limited complexity and predictable execution times.

比特币交易脚本语言包含许多操作码，但故意被限制为：除了有条件流控制以外，没有循环或复杂流控制能力。
这确保了这种语言不是“图灵完备的”，即，这种脚本有有限的复杂性，和可预见的执行时间。

Script is not a general-purpose language. These limitations ensure that the language cannot be used to create an infinite loop or other form of "logic bomb" that could be embedded in a transaction in a way that causes a denial-of-service attack against the bitcoin network. Remember, every transaction is validated by every full node on the bitcoin network. A limited language prevents the transaction validation mechanism from being used as a vulnerability.
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建无限循环或其它形式的逻辑炸弹，如果嵌入在交易中，会导致针对对比特币网络的DoS攻击。
记住，每笔交易都会被比特币网络中的每个全节点验证。
受限语言能防止交易验证机制被用作漏洞。

6.4.2无状态验证

The bitcoin transaction script language is stateless, in that there is no state prior to execution of the script, or state saved after execution of the script. Therefore, all the information needed to execute a script is contained within the script.
比特币交易脚本语言是无状态的，在脚本执行之前没有状态，执行脚本之后也不会保存状态。
所以，执行脚本所需的所有信息都都包含在脚本中。

A script will predictably execute the same way on any system. If your system verifies a script, you can be sure that every other system in the bitcoin network will also verify the script, meaning that a valid transaction is valid for everyone and everyone knows this. This predictability of outcomes is an essential benefit of the bitcoin system.
一个脚本能在任何系统以相同的方式可预测地执行。
如果你的系统验证一个脚本有效，你可以确信比特币网络中的其它系统也会验证这个脚本有效，这意味着，一个有效的交易对每个人都是有效的，而且每一个人都知道它是有效的。
结果的可预测性是比特币系统的一个基本利益。

6.4.3脚本构建（锁定+解锁）

Bitcoin’s transaction validation engine relies on two types of scripts to validate transactions: a locking script and an unlocking script.
比特币的交易验证引擎依赖于两类脚本来验证交易：锁定脚本、解锁脚本。

A locking script is a spending condition placed on an output: it specifies the conditions that must be met to spend the output in the future. Historically, the locking script was called a *scriptPubKey*, because it usually contained a public key or bitcoin address (public key hash). In this book we refer to it as a "locking script" to acknowledge the much broader range of possibilities of this scripting technology. In most bitcoin applications, what we refer to as a locking script will appear in the source code as scriptPubKey. You will also see the locking script referred to as a *witness script* (see [segwit]) or more generally as a *cryptographic puzzle*. These terms all mean the same thing, at different levels of abstraction.
锁定脚本是放置在输出上的花费条件：它指定了花费这笔输出必须要满足的条件。
由于锁定脚本常有一个公钥或比特币地址（公钥哈希值），在历史上曾被称为脚本公钥（`scriptPubKey`）。
由于这种脚本技术有更广泛的应用，所以，我们将它称为“锁定脚本”。
在多数比特币应用中，我们所称的“锁定脚本”将以`scriptPubKey`的出现在源代码中。
你还将看到这样的锁定脚本：见证脚本（`witness script`），或加密谜题（`cryptographic puzzle`）。
这些术语指的是一回事，只是在不同的在不同的抽象层次上。

An unlocking script is a script that "solves," or satisfies, the conditions placed on an output by a locking script and allows the output to be spent. Unlocking scripts are part of every transaction input. Most of the time they contain a digital signature produced by the user’s wallet from his or her private key. Historically, the unlocking script was called *scriptSig*, because it usually contained a digital signature. In most bitcoin applications, the source code refers to the unlocking script as scriptSig. You will also see the unlocking script referred to as a *witness* (see [segwit]). In this book, we refer to it as an "unlocking script" to acknowledge the much broader range of locking script requirements, because not all unlocking scripts must contain signatures.
解锁脚本是一个脚本，它解决或满足锁定脚本上的调剂那，从而允许花费此输出。
解锁脚本是每个交易输入的一部分，而且常有一个由用户钱包（通过私钥）生成的数字签名。
由于解锁脚本常有一个数字签名，因此曾被称作`ScriptSig`。
在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`）。
在本书中，我们将它称为“解锁脚本”，因为它有更广的应用范围。
但并非所有解锁脚本都一定会包含签名。

Every bitcoin validating node will validate transactions by executing the locking and unlocking scripts together. Each input contains an unlocking script and refers to a previously existing UTXO. The validation software will copy the unlocking script, retrieve the UTXO referenced by the input, and copy the locking script from that UTXO. The unlocking and locking script are then executed in sequence. The input is valid if the unlocking script satisfies the locking script conditions (see Separate execution of unlocking and locking scripts). All the inputs are validated independently, as part of the overall validation of the transaction.
每个比特币验证节点会通过同时执行锁定脚本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`。
验证软件将拷贝解锁脚本，获取输入引用的`UTXO`，并从该`UTXO`拷贝锁定脚本。
然后，依次执行解锁脚本和锁定脚本。
如果解锁脚本满足锁定脚本的条件，则输入有效。
所有输入都是独立验证的，是交易总体验证的一部分。

Note that the UTXO is permanently recorded in the blockchain, and therefore is invariable and is unaffected by failed attempts to spend it by reference in a new transaction. Only a valid transaction that correctly satisfies the conditions of the output results in the output being considered as "spent" and removed from the set of unspent transaction outputs (UTXO set).
注意，UTXO被永久记录在区块链中，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交易才能花费此输出，继而该输出会从UTXO集中删除。

Combining scriptSig and scriptPubKey to evaluate a transaction script is an example of the unlocking and locking scripts for the most common type of bitcoin transaction (a payment to a public key hash), showing the combined script resulting from the concatenation of the unlocking and locking scripts prior to script validation.
下图是最常见的比特币交易（P2PKH）的解锁脚本和锁定脚本的示例，
显示了在脚本验证之前，把解锁脚本和锁定脚本合并为组合脚本。

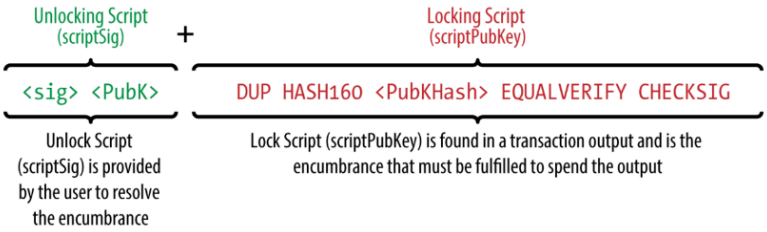


Figure 3. Combining scriptSig and scriptPubKey to evaluate a transaction script

6.4.3.1脚本执行堆栈

Bitcoin's scripting language is called a stack-based language because it uses a data structure called a *stack*. A stack is a very simple data structure that can be visualized as a stack of cards. A stack allows two operations: push and pop. Push adds an item on top of the stack. Pop removes the top item from the stack. Operations on a stack can only act on the topmost item on the stack. A stack data structure is also called a Last-In-First-Out, or "LIFO" queue.
比特币的脚本语言被称为基于堆栈的语言，因为它使用堆栈数据结构。
堆栈是一种非常简单的数据结构，允许两个操作：push和pop（压入和弹出）。
Push（压入）在栈顶添加一个项目。Pop（弹出）从栈定删除一个项目。
栈上的操作只能作用于栈顶的项目。堆栈数据结构也被称为"后进先出"队列。

The scripting language executes the script by processing each item from left to right. Numbers (data constants) are pushed onto the stack. Operators push or pop one or more parameters from the stack, act on them, and might push a result onto the stack. For example, OP_ADD will pop two items from the stack, add them, and push the resulting sum onto the stack.
脚本语言执行脚本是从左到右处理每个项目。
数字（数据常量）被压入栈上。
操作码（Operator）从栈中压入或弹出一个或多个参数，对其进行操作，并可能将结果压入栈上。
例如，操作码 OP_ADD 从栈中弹出两个项目，求它们的和，并将结果压入栈上。

Conditional operators evaluate a condition, producing a boolean result of TRUE or FALSE. For example, OP_EQUAL pops two items from the stack and pushes TRUE (TRUE is represented by the number 1) if they are equal or FALSE (represented by zero) if they are not equal. Bitcoin transaction scripts usually contain a conditional operator, so that they can produce the TRUE result that signifies a valid transaction.
条件操作码（conditional operator）对一个条件进行计算，产生布尔结果（TRUE或FALSE）。
例如，OP_EQUAL从栈中弹出两个项目，如果它们相等，则压入TRUE（1），否则压入FALSE（0）。
比特币交易脚本通常包含条件操作码，这样就可以产生表示有效交易的 TRUE 结果。

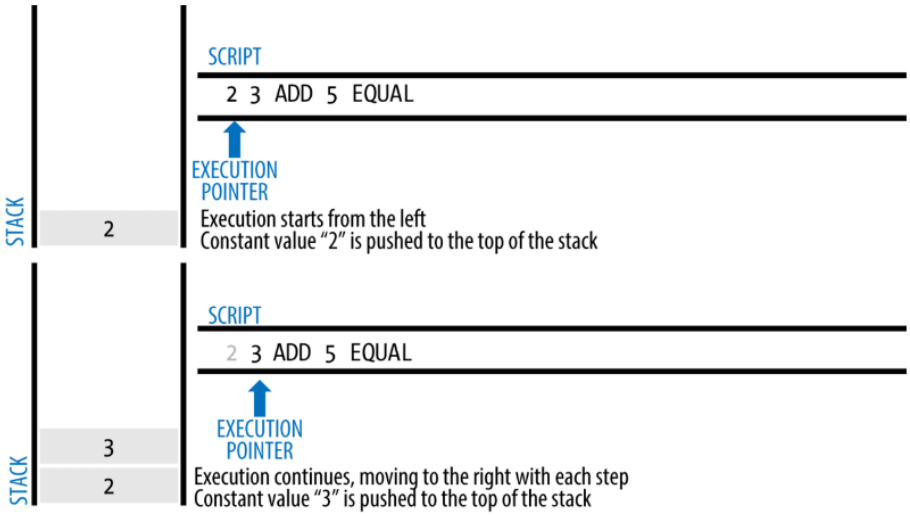
6.4.3.2一个简单的脚本

Now let’s apply what we’ve learned about scripts and stacks to some simple examples.
现在，我们将应用学过的知识，看一些简单的例子。

In Bitcoin’s script validation doing simple math, the script 2 3 OP_ADD 5 OP_EQUAL demonstrates the arithmetic addition operator OP_ADD, adding two numbers and putting the result on the stack, followed by the conditional operator OP_EQUAL, which checks that the resulting sum is equal to 5. For brevity, the OP_ prefix is omitted in the step-by-step example. For more details on the available script operators and functions, see [tx_script_ops].
如图4，脚本“ 2 3 OP_ADD 5 OP_EQUAL ”演示了：

- 算术加法操作码 OP_ADD ，它将两个数字相加，然后把结果压入栈。
- 然后是条件操作码 OP_EQUAL，检查结果是否等于 5 。

为了简化起见，前缀OP_在演示过程中将被省略。
有关可用的脚本操作码和功能的更多信息，参见[交易脚本]。



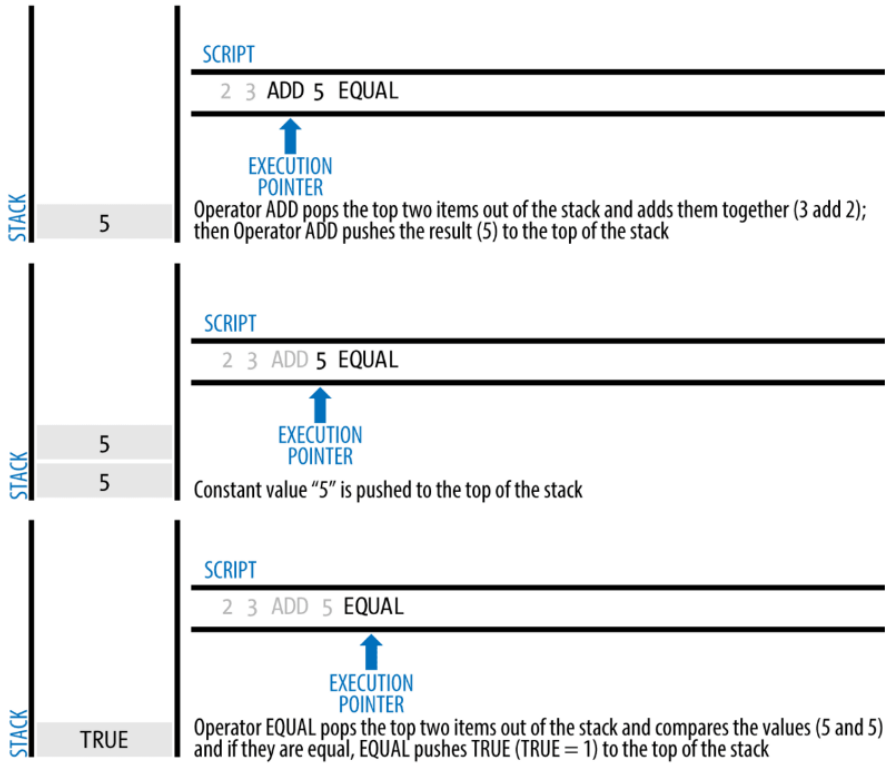


Figure 4. Bitcoin's script validation doing simple math
图4：比特币的脚本验证，做简单的算数

Although most locking scripts refer to a public key hash (essentially, a bitcoin address), thereby requiring proof of ownership to spend the funds, the script does not have to be that complex. Any combination of locking and unlocking scripts that results in a TRUE value is valid. The simple arithmetic we used as an example of the scripting language is also a valid locking script that can be used to lock a transaction output.
虽然多数锁定脚本都指向一个公钥哈希（本质上就是比特币地址），因此需要证明所有权，但脚本不一定很复杂。
任何解锁脚本和锁定脚本的组合的结果为真（TRUE）时，表示有效。
前面例子中的脚本语言使用的简单算术也是一个有效的锁定脚本，该脚本能用于锁定交易输出。

Use part of the arithmetic example script as the locking script:
使用部分算术例子脚本作为锁定脚本：

```
3 OP_ADD 5 OP_EQUAL
```

which can be satisfied by a transaction containing an input with the unlocking script:
如果输入包含下面的锁定脚本，则交易就是有效的：

```
2
```

The validation software combines the locking and unlocking scripts and the resulting script is:
验证软件将锁定脚本和解锁脚本组合起来，结果是：

```
2 3 OP_ADD 5 OP_EQUAL
```

As we saw in the step-by-step example in Bitcoin's script validation doing simple math, when this script is executed, the result is OP_TRUE, making the transaction valid. Not only is this a valid transaction output locking script, but the resulting UTXO could be spent by anyone with the arithmetic skills to know that the number 2 satisfies the script.
正如上图所示，脚本被执行时，结果是OP_TRUE，所以这个交易有效。
这不仅是一个有效的交易输出锁定脚本，而且知道这个方法的任何人都可以花费这个UTXO。

Tip：Transactions are valid if the top result on the stack is TRUE (noted as { 0x01}), any other nonzero value, or if the stack is empty after script execution. Transactions are invalid if the top value on the stack is FALSE (a zero-length empty value, noted as {}) or if script execution is halted explicitly by an operator, such as OP_VERIFY, OP_RETURN, or a conditional terminator such as OP_ENDIF.
See [tx_script_ops] for details.
提示：如果栈顶的结果是TRUE（1），或任何非零值，或脚本执行后栈为空，则这个交易是有效的。
如果栈顶的结果是FALSE（0），或脚本执行被操作码明确停止（如OP_VERIFY、OP_RETURN），或有条件终止（如OP_ENDIF），则这个交易是无效的。详见[tx_script_ops]相关内容。

The following is a slightly more complex script, which calculates 2 + 7 - 3 + 1. Notice that when the script contains several operators in a row, the stack allows the results of one operator to be acted upon by the next operator:
下面是一个稍微复杂的脚本，它计算 2+7-3+1。
注意，当脚本在同一行包含多个操作码时，栈允许一个操作码的结果用于下一个操作码的执行。

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

Try validating the preceding script yourself using pencil and paper. When the script execution ends, you should be left with the value TRUE on the stack.
试试用纸笔演算一下这个脚本。
当脚本执行完时，栈上得到的结果应该是TRUE。

6.4.3.3 分别执行解锁脚本和锁定脚本

In the original bitcoin client, the unlocking and locking scripts were concatenated and executed in sequence. For security reasons, this was changed in 2010, because of a vulnerability that allowed a malformed unlocking script to push data onto the stack and corrupt the locking script. In the current implementation, the scripts are executed separately with the stack transferred between the two executions, as described next.
在最初的比特币客户端中，解锁脚本和锁定脚本被连接起来，顺序执行。
出于安全考虑，2010 年做了修改，因为有一个漏洞允许异常解锁脚本压入数据，导致锁定脚本崩溃。
在当前的实现中，脚本被分开执行，在两个执行之间传递这个栈，如下描述。

First, the unlocking script is executed, using the stack execution engine. If the unlocking script is executed without errors (e.g., it has no "dangling" operators left over), the main stack is copied and the locking script is executed. If the result of executing the locking script with the stack data copied from the unlocking script is "TRUE," the unlocking script has succeeded in resolving the conditions imposed by the locking script and, therefore, the input is a valid authorization to spend the UTXO. If any result other than "TRUE" remains after execution of the combined script, the input is invalid because it has failed to satisfy the spending conditions placed on the UTXO.
首先，执行解锁脚本，使用堆栈执行引擎。
如果解锁脚本执行没有错误（例如：没有留下“悬空”操作码），则拷贝主栈，执行锁定脚本。
如果执行结果是TRUE，则解锁脚本就成功地满足了锁定脚本的条件，因此，该输入是有效的。
如果结果不是TRUE，则输入是无效的，因为它不能满足UTXO中的条件。

6.4.4 P2PKH

The vast majority of transactions processed on the bitcoin network spend outputs locked with a Pay-to-Public-Key-Hash or "P2PKH" script. These outputs contain a locking script that locks the output to a public key hash, more commonly known as a bitcoin address. An output locked by a P2PKH script can be unlocked (spent) by presenting a public key and a digital signature created by the corresponding private key (see Digital Signatures (ECDSA)).
比特币网络处理的大多数交易花费的是由P2PKH脚本锁定的输出。
这些输出有一个锁定脚本，将这个输出锁定到一个公钥哈希，即比特币地址。
可用于一个公钥和数字签名（使用对应的私钥来创建）来解锁P2PKH脚本锁定的输出。

For example, let's look at Alice's payment to Bob's Cafe again. Alice made a payment of 0.015 bitcoin to the cafe's bitcoin address. That transaction output would have a locking script of the form:
例如，我们再来看看Alice买咖啡的例子。
Alice向Bob咖啡店的比特币地址支付了0.015比特币。
这个交易的输出有一个锁定脚本：

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

The Cafe Public Key Hash is equivalent to the bitcoin address of the cafe, without the Base58Check encoding. Most applications would show the *public key hash* in hexadecimal encoding and not the familiar bitcoin address Base58Check format that begins with a "1."
<Cafe Public Key Hash> 是咖啡馆的比特币地址，不是Base58Check编码。
多数应用会把公钥哈希显示为十六进制，而不是以1开头的Bsase58Check编码的比特币地址。

The preceding locking script can be satisfied with an unlocking script of the form:
可以用下面的解锁脚本满足上面的锁定脚本：

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

The two scripts together would form the following combined validation script:
将两个脚本结合起来，就成了下面的验证脚本：

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

When executed, this combined script will evaluate to TRUE if, and only if, the unlocking script matches the conditions set by the locking script. In other words, the result will be TRUE if the unlocking script has a valid signature from the cafe's private key that corresponds to the public key hash set as an encumbrance.
只有当解锁脚本满足锁定脚本的条件时，执行结果才是TRUE。
换句话说，只有当解锁脚本有一个有效的签名，执行结果才会是TURE。

Figures #P2PubKHash1 and #P2PubKHash2 show (in two parts) a step-by-step execution of the combined script, which will prove this is a valid transaction.
图5和图6显示了这个组合脚本的执行过程，它证明了这是一个有效的交易。

1. <sig> 把<sig>值压入栈顶
2. <PubK> 把<PubK>值压入栈顶
3. DUP 复制栈顶的项，并压入栈顶
4. HASH160 对栈顶项执行哈希RIPEMD160(SHA256(PubK))，把结果值（PubKHash）压入栈顶。
5. <PubKHash> 压入脚本中的<PubKHash>值
6. EQUALVERIFY 比较栈顶的两个PbuKHash，如果相等，就删除它俩，并继续执行。

7. CHECKSIG 检查签名<sig>是否匹配公钥<PubK>，如果匹配，则在栈顶压入TRUE

Figure 5. Evaluating a script for a P2PKH transaction (part 1 of 2)
图5：第1部分，计算一个P2PKH交易的脚本

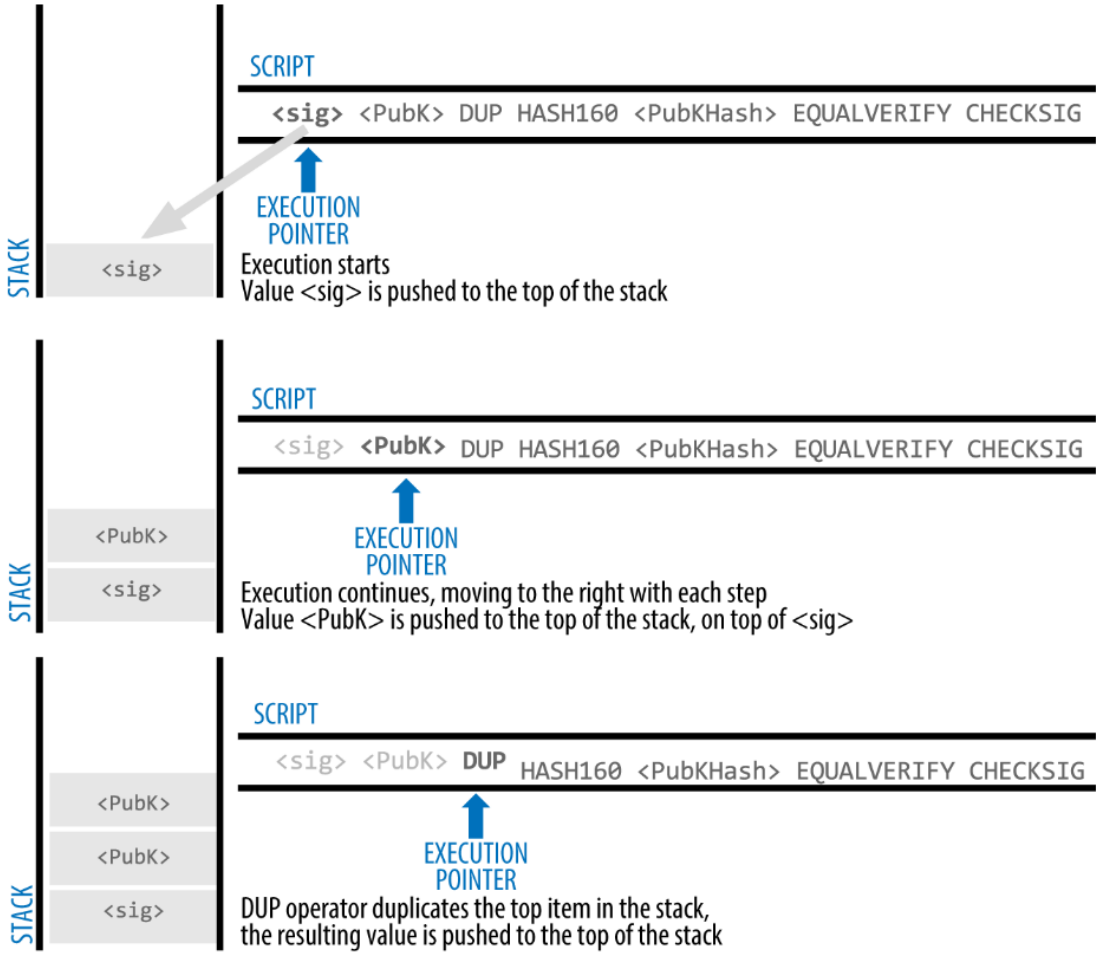
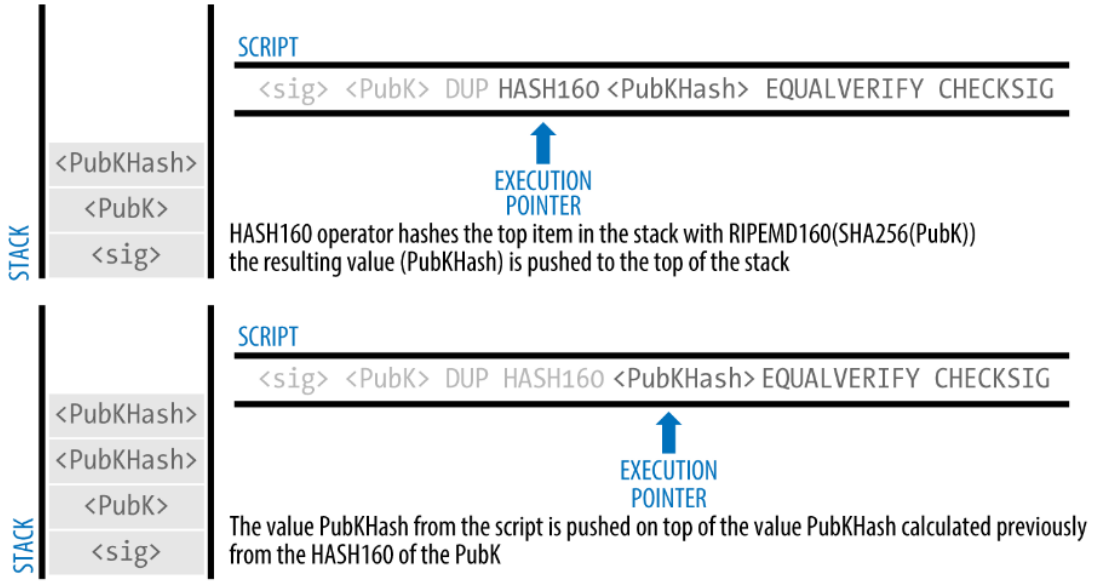
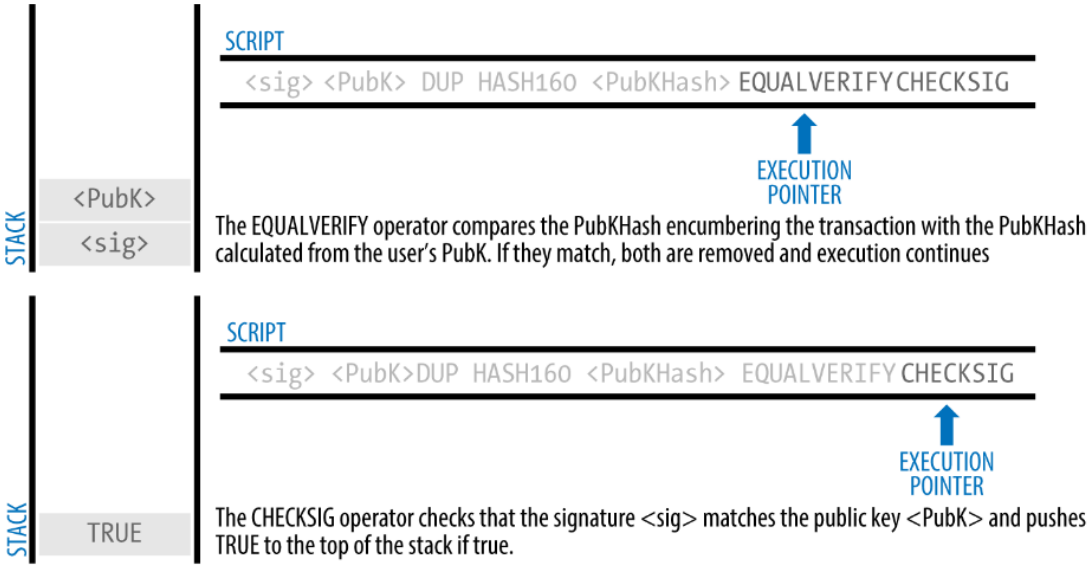


Figure 6. Evaluating a script for a P2PKH transaction (part 2 of 2)
图6：第2部分，计算一个P2PKH交易的脚本




SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubK>
<sig>
The EQUALVERIFY operator compares the PubKHash encumbering the transaction with the PubKHash calculated from the user's PubK. If they match, both are removed and execution continues
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
TRUE
The CHECKSIG operator checks that the signature <sig> matches the public key <PubK> and pushes TRUE to the top of the stack if true.

6.5数字签名（ECDSA）

So far, we have not delved into any detail about "digital signatures." In this section we look at how digital signatures work and how they can present proof of ownership of a private key without revealing that private key.
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，我们看看数字签名的工作原理，以及如何在不透漏私钥的情况下证明拥有私钥。

The digital signature algorithm used in bitcoin is the *Elliptic Curve Digital Signature Algorithm*, or *ECDSA*. ECDSA is the algorithm used for digital signatures based on elliptic curve private/public key pairs, as described in [elliptic_curve]. ECDSA is used by the script functions OP_CHECKSIG, OP_CHECKSIGVERIFY, OP_CHECKMULTISIG, and OP_CHECKMULTISIGVERIFY. Any time you see those in a locking script, the unlocking script must contain an ECDSA signature.
比特币中使用的数字签名算法是ECDSA（椭圆曲线数字签名算法）。
ECDSA是用于基于椭圆曲线私钥/公钥对的用于数字签名的算法。
下列脚本函数使用了ECDSA：

- OP_CHECKSIG
- OP_CHECKSIGVERIFY
- OP_CHECKMULTISIG
- OP_CHECKMULTISIGVERIFY

当你在锁定脚本中看到这些操作码时，解锁脚本都必须包含ECDSA签名。

A digital signature serves three purposes in bitcoin (see the following sidebar). First, the signature proves that the owner of the private key, who is by implication the owner of the funds, has *authorized* the spending of those funds. Secondly, the proof of authorization is *undeniable* (nonrepudiation). Thirdly, the signature proves that the transaction (or specific parts of the transaction) have not and *cannot be modified* by anyone after it has been signed.
在比特币中，数字签名有三种目的：

- 证明拥有私钥，即拥有资金，有权花费这些资金。 （身份认证）
- 证明授权是不可否认的 （不可否认）
- 证明交易（或特定部分）在签字之后，没有被修改。 （完整性）

Note that each transaction input is signed independently. This is critical, as neither the signatures nor the inputs have to belong to or be applied by the same "owners." In fact, a specific transaction scheme called "CoinJoin" uses this fact to create multi-party transactions for privacy.
注意，每个交易输入都是独立签名的。
这一点至关重要，因为这些签名和输入不必输入同一个人。
实际上，一个特殊的交易方案（CoinJoin）使用这个事实来创建多方交易，以保护隐私。

Note：Each transaction input and any signature it may contain is *completely* independent of any other input or signature. Multiple parties can collaborate to construct transactions and sign only one input each.
注意：每个交易输入和它可能包含的任何签名都完全独立于任何其它输入或签名。
多方可以协作构建交易，并各自仅签一个输入。

Wikipedia’s Definition of a "Digital Signature" 维基百科：数字签名
https://en.wikipedia.org/wiki/Digital_signature
A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (nonrepudiation), and that the message was not altered in transit (integrity).
Source: https://en.wikipedia.org/wiki/Digital_signature
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：

- 该消息是由已知的发送者创建的（身份认证）
- 发送方不能否认已发送消息 （不可否认）
- 消息在传输中未被更改 （完整性）

6.5.1数字签名工作原理

A digital signature is a *mathematical scheme* that consists of two parts. The first part is an algorithm for creating a signature, using a private key (the signing key), from a message (the transaction). The second part is an algorithm that allows anyone to verify the signature, given also the message and a public key.
数字签名是一个数学方案，由两部分组成：

- 一个算法，用于使用私钥（签名密钥）从消息（交易）创建签名。
- 一个算法，允许任何人验证这个签名，同时提供这个消息个一个公钥。

6.5.1.1创建一个数字签名

In bitcoin's implementation of the ECDSA algorithm, the "message" being signed is the transaction, or more accurately a hash of a specific subset of the data in the transaction (see Signature Hash Types (SIGHASH)). The signing key is the user's private key. The result is the signature:
在比特币的ECDSA算法实现中，被签名的"消息"是交易，更准确地说，是交易中特定数据子集的哈希。签名密钥是用户的私钥，结果是签名：

$$Sig = F_{sig}(F_{hash}(m), dA)$$

where:

- *m* is the transaction (or parts of it)交易（或交易的一部分）
- *dA* is the signing private key 签名私钥
- F_{hash} is the hashing function 哈希函数
- F_{sig} is the signing algorithm 签名算法
- *Sig* is the resulting signature 生成的签名

More details on the mathematics of ECDSA can be found in ECDSA Math.
后面会讲解ECDSA数学运算的细节。

The function F_{sig} produces a signature Sig that is composed of two values, commonly referred to as R and S:
F_{sig} 生成一个签名Sig，它由两个值组成：Sig = (R,S)

Now that the two values R and S have been calculated, they are serialized into a byte-stream using an international standard encoding scheme called the *Distinguished Encoding Rules*, or *DER*.
有了R和S，就把它们序列化为一个字节流，使用的是DER编码方案。

6.5.1.2签名的序列化（DER）

Let's look at the transaction Alice created again. In the transaction input there is an unlocking script that contains the following DER-encoded signature from Alice's wallet:
我们再来看看Alice的交易。
在交易输入中，有一个解锁脚本，它包含下面这个DER编码签名：

```
3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02
204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301
```

That signature is a serialized byte-stream of the R and S values produced by Alice's wallet to prove she owns the private key authorized to spend that output. The serialization format consists of nine elements as follows:
这个签名是Alice的钱包生成的R和S值的序列化字节流，证明她拥有私钥，可以花费该输出。
序列化格式包含以下9个元素：

- 0x30 —indicating the start of a DER sequence 表示DER序列的开始
- 0x45 —the length of the sequence (69 bytes) 序列的长度（69字节）
- 0x02 —an integer value follows 后面跟一个整数值
- 0x21 —the length of the integer (33 bytes) 整数的长度（33字节）
- **R**—00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb
- 0x02 —another integer follows 后面跟一个整数值
- 0x20 —the length of the integer (32 bytes) 整数的长度（32字节）
- **S**—4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
- A suffix (0x01) indicating the type of hash used (SIGHASH_ALL)
  后缀（0x01）表示使用的哈希类型是SIGHASH_ALL

```
30 45
02 21 00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb
02 20 4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01
```

See if you can decode Alice's serialized (DER-encoded) signature using this list. The important numbers are R and S; the rest of the data is part of the DER encoding scheme.
看看你是否能使用这个列表解码 Alice 的序列化（DER编码）签名。
重要的数字是R和S，其余的是DER编码方案的一部分。

6.5.2验证签名

To verify the signature, one must have the signature (R and S), the serialized transaction, and the public key (that corresponds to the private key used to create the signature). Essentially, verification of a signature means "Only the owner of the private key that generated this public key could have produced this signature on this transaction."
为了验证签名，必须有：签名（R,S）、序列化交易、公钥。
本质上，签名验证的意思是：只有生成此公钥的私钥的所有者，才能在此交易上生成此签名。

The signature verification algorithm takes the message (a hash of the transaction or parts of it), the signer's public key and the signature (R and S values), and returns TRUE if the signature is valid for this message and public key.
签名验证算法使用这个消息（交易或部分交易的哈希）、公钥、签名（R,S），
如果签名对该消息和公钥是有效的，则返回 TRUE。

6.5.3签名哈希类型（SIGHASH）

Digital signatures are applied to messages, which in the case of bitcoin, are the transactions themselves. The signature implies a *commitment* by the signer to specific transaction data. In the simplest form, the signature applies to the entire transaction, thereby committing all the inputs, outputs, and other transaction fields. However, a signature can commit to only a subset of the data in a transaction, which is useful for a number of scenarios as we will see in this section.
对消息应用数字签名，在比特币中，消息就是交易本身。

签名表示签字人对特定交易数据的保证（commitment）。
在最简单的形式中，对整个交易进行签名，从而保证所有的输入、输出和其它交易字段。
但是，一个签名也可以只保证交易中的数据子集，这对某些场景是有用的。

Bitcoin signatures have a way of indicating which part of a transaction's data is included in the hash signed by the private key using a SIGHASH flag. The SIGHASH flag is a single byte that is appended to the signature. Every signature has a SIGHASH flag and the flag can be different from input to input. A transaction with three signed inputs may have three signatures with different SIGHASH flags, each signature signing (committing) different parts of the transaction.
比特币签名有一个方法，使用一个SIGHASH标志来表示交易的哪些数据包含在哈希中，这个哈希被私钥签名。
SIGHASH标志是一个字节，附加在签名后面。
每个签名有一个SIGHASH标志，不同输入的标红字可以不同。
有三个签名输入的交易可以有三个不同的SIGHASH标志的签名，每个签名保证交易的不同部分。

Remember, each input may contain a signature in its unlocking script. As a result, a transaction that contains several inputs may have signatures with different SIGHASH flags that commit different parts of the transaction in each of the inputs. Note also that bitcoin transactions may contain inputs from different "owners," who may sign only one input in a partially constructed (and invalid) transaction, collaborating with others to gather all the necessary signatures to make a valid transaction. Many of the SIGHASH flag types only make sense if you think of multiple participants collaborating outside the bitcoin network and updating a partially signed transaction.
记住，每个输入可以在其解锁脚本中包含一个签名。因此，有多个输入的交易可以有具有不同SIGHASH标志的签名，它们在每个输入中保证交易的不同部分。
还要注意，比特币交易可能包含来自不同“拥有者”的输入，他们只能签署一个输入，协作收集所有的签名后才能使交易生效。
许多SIGHSASH标志类型只有在这种情况下才有意义：你想让多个参与者才比特币网络之外协作，并更新一个部分签名的交易。

There are three SIGHASH flags: ALL, NONE, and SINGLE, as shown in SIGHASH types and their meanings.
有三个SIGHASH标志：ALL、NONE、SINGLE

Table 3. SIGHASH types and their meanings
表3：SIGHASH类型和它们的含义

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

SIGHASH标志	值	描述
ALL	0x01	签名对象：所有输入和输出
NONE	0x02	签名对象：所有输入 （不包括任何输出）
SINGLE	0x03	签名对象：所有输入，但只有一个输出（与被签名输入有相同的索引号）

In addition, there is a modifier flag SIGHASH_ANYONECANPAY, which can be combined with each of the preceding flags. When ANYONECANPAY is set, only one input is signed, leaving the rest (and their sequence numbers) open for modification. The ANYONECANPAY

has the value 0x80 and is applied by bitwise OR, resulting in the combined flags as shown in SIGHASH types with modifiers and their meanings.
此外，有一个修饰标志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）可以修改。
ANYONECANPAY的值为0x80，得到如下组合标志。

Table 4. SIGHASH types with modifiers and their meanings

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	值	描述
ALL|ANYONECANPAY	0x81	Signature applies to one input and all outputs
NONE|ANYONECANPAY	0x82	Signature applies to one input, none of the outputs
SINGLE|ANYONECANPAY	0x83	Signature applies to one input and the output with the same index number

The way SIGHASH flags are applied during signing and verification is that a copy of the transaction is made and certain fields within are truncated (set to zero length and emptied). The resulting transaction is serialized. The SIGHASH flag is added to the end of the serialized transaction and the result is hashed. The hash itself is the "message" that is signed.
在签名和验证期间，SIGHASH标志的使用方法是：

- 拷贝这个交易，删去某些字段（设置为长度为零并清空）
- 对结果进行序列化，把SIGHASH标志加到序列化交易的结尾
- 对结果计算哈希，这个哈希是被签名的“消息”。

Depending on which SIGHASH flag is used, different parts of the transaction are truncated. The resulting hash depends on different subsets of the data in the transaction. By including the SIGHASH as the last step before hashing, the signature commits the SIGHASH type as well, so it can’t be changed (e.g., by a miner).
根据使用的SIGHASH标志，交易的不同部分被删去。
得到的哈希依赖于交易中数据的不同子集。
在哈希之前，在最后一步包含这个SIGHASH，签名也保证了SIGHASH类型，这样它不会被修改。

Note：All SIGHASH types sign the transaction nLocktime field (see [transaction_locktime_nlocktime]). In addition, the SIGHASH type itself is appended to the transaction before it is signed, so that it can’t be modified once signed.
注意：所有SIGHASH类型都对交易nLocktime字段进行签名。
此外，在签名之前，把SIGHASH类型本身附加到交易上，这样，签名之后就不能修改它了。

In the example of Alice’s transaction (see the list in Serialization of signatures (DER)), we saw that the last part of the DER-encoded signature was 01, which is the SIGHASH_ALL flag. This locks the transaction data, so Alice’s signature is committing the state of all inputs and outputs. This is the most common signature form.
在Alice的交易中，我们看到DER编码签名的最后是01，这是SIGHASH_ALL标志。
它锁定交易数据，因此Alice的签名保证了所有输入和输出的状态。这是最常见的签名形式。

Let’s look at some of the other SIGHASH types and how they can be used in practice:
我们来看看其它SIGHASH类型，以及如何在实际中使用它们。

ALL|ANYONECANPAY	0x81	签名对象：一个输入，所有输出

This construction can be used to make a "crowdfunding"-style transaction. Someone attempting to raise funds can construct a transaction with a single output. The single output pays the "goal" amount to the fundraiser. Such a transaction is obviously not valid, as it has no inputs. However, others can now amend it by adding an input of their own, as a donation. They sign their own input with ALL|ANYONECANPAY. Unless enough inputs are gathered to reach the value of the output, the transaction is invalid. Each donation is a "pledge," which cannot be collected by the fundraiser until the entire goal amount is raised.
这种类型用来做“众筹”类型的交易。
希望筹集资金的人可以创建一个交易，有一个输出。这个输出把“目标”金额付给发起人。
这样的交易显然是无效的，因为它没有输入。
但现在其他人可以把自己的输入加到这个交易上。他们用ALL|ANYONECANPAY签名自己的输入。
除非收集的输入达到输出的价值，否则这个交易是无效的。
每次捐款就是一个“保证”，募集到目标金额之后，发起人才能用钱。

ddk说明：

- 发起人创建了一个交易，只有一个输出（目标资金M，支付给发起人）
  因为这个只有输出，没有输入，所以是无效的。
- 捐款人可以给这个交易增加输入，并用ALL|ANYONECANPAY进行签名。
  这样，捐款人对自己的输入和交易输出做了保证。
- 当输入总额小于输出目标资金M时，这个交易无效；
  当输入总额大于等于输出目标资金M时，这个交易就有效了。
- 当交易有效时，可以在网络上传播这个交易。
  比特币节点验证这个交易的所有输入，确认有效后，发起人就可以花费获得的资金了。

NONE	0x02	签名对象：所有输入 （不包括任何输出）

This construction can be used to create a "bearer check" or "blank check" of a specific amount. It commits to the input, but allows the output locking script to be changed. Anyone can write their own bitcoin address into the output locking script and redeem the transaction. However, the output value itself is locked by the signature.
这种类型可用于创建一个特定账户的“不记名支票”或“空白支票”。
它保证输入，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中，并执行交易。
但是，输出值本身被这个签名锁定。

ddk说明：

- 发起人创建了一个交易，只对输入做了签名，而没有签名输出。
  输出地址未锁定，输出值被锁定。
- 这样，拿到这个交易数据的人，可以修改这个交易。

参见：https://raghavsood.com/blog/2018/06/10/bitcoin-signature-types-sighash
SIGHASH_SINGLE - This can be used to send BTC in scenarios where you only want to make the transfer provided some other parties are making a transfer too. Essentially, you agree to move a certain amount of BTC to a certain output, but only if the other input parties to the transaction also move their BTC. This can be used to create a transaction where you and a friend need to pay someone 1.5 BTC, and you are contributing 1 BTC. However, your friend only has a 1 BTC output. Thus, you can use SIGHASH_SINGLE to create a transaction with both the 1 BTC inputs, and a 1.5 BTC output to whoever needs to be paid. Your friend can then add an output for their change address, and use SIGHASH_ALL or SIGHASH_SINGLE (if their change address is at the same index as their input) and complete the transaction.
这一个有点儿让人困惑。表面上看，似乎你在烧钱，因为没有对任何输出签名。
事实上，如果你创建了一个交易，只有一个输入，并用_NONE进行签名，矿工就能修改输出给任何人。

这主要被用于多方贡献输入的场景。
在这种场景中，这样的签名实际意味着：我同意花我的钱，只要所有其他人也都花了他们的钱。
期望是，有一个签名者使用ALL来保证这个的所有输出的安全，并将钱发送给一个相互同意的输出集。

NONE|ANYONECANPAY	0x82	签名对象：一个输入 （不包括任何输出）

This construction can be used to build a "dust collector." Users who have tiny UTXO in their wallets can't spend these without the cost in fees exceeding the value of the dust. With this type of signature, the dust UTXO can be donated for anyone to aggregate and spend whenever they want.
这种类型可用来建造一个“吸尘器”。
用户的钱包中有小额UTXO，但因为交易费超过这些钱，所以无法花费这些钱。
有了这种类型的签名，小额UTXO可以捐给任何人，以聚合和花费这些钱。

ddk说明：

- 例如，A有一些小额UTXO，但无法花费。
- B可以创建一个交易，A把小额UTXO加到这个交易上，这样，B就可以花费了。

There are some proposals to modify or expand the SIGHASH system. One such proposal is *Bitmask Sighash Modes* by Blockstream's Glenn Willen, as part of the Elements project. This aims to create a flexible replacement for SIGHASH types that allows "arbitrary, miner-rewritable bitmasks of inputs and outputs" that can express "more complex contractual precommitment schemes, such as signed offers with change in a distributed asset exchange."
有一些修改或扩展SIGHASH系统的建议。
一个建议是Blockstream's Glenn Willen提出的Bitmask Sighash模式。
目的是为SIGHASH类型创建一个灵活的替代品，允许“任意的、矿工可写的输入和输出位掩码”，
它可以表达“更复杂的合同预付款方案，例如在分布式资产交换中变更签名要约”。

Note：You will not see SIGHASH flags presented as an option in a user's wallet application. With few exceptions, wallets construct P2PKH scripts and sign with SIGHASH_ALL flags. To use a different SIGHASH flag, you would have to write software to construct and sign transactions. More importantly, SIGHASH flags can be used by special-purpose bitcoin applications that enable novel uses.
说明：在用户钱包中，你不会看到SIGHASH标志。
多数情况下，钱包构建P2PKH脚本，使用SIGHASH_ALL标志进行签名。
为了使用不同的SIGHASH标志，你必须编写软件来构建和签名交易。
更重要的是，专用比特币应用使用SIGHASH标志，从而实现新颖的用途。

6.5.4 ECDSA数学

As mentioned previously, signatures are created by a mathematical function F_{sig} that produces a signature composed of two values *R* and *S*. In this section we look at the function F_{sig} in more detail.
如前所述，签名由一个数学函数 F_{sig} 生成的，产生的签名有两个值组成：R和S。
在本节中，我们看看这个函数的细节。

The signature algorithm first generates an *ephemeral* (temporary) private public key pair. This temporary key pair is used in the calculation of the *R* and *S* values, after a transformation involving the signing private key and the transaction hash.
签名算法首先生成一个临时密钥对。
在一个转化后（涉及签名私钥和交易哈希），这个临时密钥对用于计算R和S的值。

The temporary key pair is based on a random number *k*, which is used as the temporary private key. From *k*, we generate the corresponding temporary public key *P* (calculated

as *P* = *k***G*, in the same way bitcoin public keys are derived; see [pubkey]). The *R* value of the digital signature is then the x coordinate of the ephemeral public key *P*.
这个临时密钥对基于一个随机数k，用作临时私钥。
用k生成临时公钥P（P = k*G）。
数字签名的R值就是临时公钥P的x坐标。

From there, the algorithm calculates the *S* value of the signature, such that:
然后，这个算法计算签名的S值，使得：
$S = k^{-1} (Hash(m) + dA * R) \mod p$

where:
- *k* is the ephemeral private key 临时私钥
- *R* is the x coordinate of the ephemeral public key 临时公钥的x坐标
- *dA* is the signing private key 签名私钥
- *m* is the transaction data 交易数据
- *p* is the prime order of the elliptic curve 椭圆曲线的素数阶

Verification is the inverse of the signature generation function, using the *R*, *S* values and the public key to calculate a value *P*, which is a point on the elliptic curve (the ephemeral public key used in signature creation):
验证是签名生成函数的反向计算，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）：
$P = S^{-1} * Hash(m) * G + S^{-1} * R * Qa$

where:
- *R* and *S* are the signature values 签名值
- *Qa* is Alice's public key Alice的公钥
- *m* is the transaction data that was signed 被签名的数据
- *G* is the elliptic curve generator point 椭圆曲线生成器点

If the x coordinate of the calculated point *P* is equal to *R*, then the verifier can conclude that the signature is valid.
如果P的x坐标等于R，则签名是有效的。

Note that in verifying the signature, the private key is neither known nor revealed.
注意，在验证签名时，不用知道私钥，也不需要透漏私钥。

Tip: ECDSA is necessarily a fairly complicated piece of math; a full explanation is beyond the scope of this book. A number of great guides online take you through it step by step: search for "ECDSA explained" or try this one: http://bit.ly/2r0HhGB.
提示：ECDSA是一个相当复杂的数学计算，完整的解释超出了本书的范围。
网上有一些很好的指南帮助你一步一步理解：
搜索"ECDSA explained"或尝试这个http://bit.ly/2r0HhGB

6.5.5随机性在签名中的重要性

As we saw in ECDSA Math, the signature generation algorithm uses a random key *k*, as the basis for an ephemeral private/public key pair. The value of *k* is not important, *as long as it is random*. If the same value *k* is used to produce two signatures on different messages (transactions), then the signing *private key* can be calculated by anyone. Reuse of the same value for *k* in a signature algorithm leads to exposure of the private key!
正如我们在上一节中看到的，签名生成算法使用了一个随机密钥k，作为临时密钥对的基础。
k的值不重要，只要它是随机的。
如果使用相同的值 k 对不同的消息（交易）生成两个签名，那么，任何人都可以计算出签名私钥。
在签名算法中重用相同的 k 值会导致泄露私钥！

Waring：If the same value *k* is used in the signing algorithm on two different transactions, the private key can be calculated and exposed to the world!
警告：如果在两个不同的交易中，签名算法使用了相同的值 `k`，则私钥可能被算出和泄露！

This is not just a theoretical possibility. We have seen this issue lead to exposure of private keys in a few different implementations of transaction-signing algorithms in bitcoin. People have had funds stolen because of inadvertent reuse of a *k* value. The most common reason for reuse of a *k* value is an improperly initialized random-number generator.
这不仅仅是理论上的可能性。
我们已经看到：在比特币中，一些交易签名算法的实现，因为这个问题导致了私钥泄露。
人们由于无意中重复使用 `k` 值，而导致资金被窃。
重用 `k` 值的最常见原因是未正确初始化随机数生成器。

To avoid this vulnerability, the industry best practice is to not generate *k* with a random-number generator seeded with entropy, but instead to use a deterministic-random process seeded with the transaction data itself. This ensures that each transaction produces a different *k*. The industry-standard algorithm for deterministic initialization of *k* is defined in RFC 6979, published by the Internet Engineering Task Force.
为了避免这个漏洞，业界最佳实践是：
不要随机数做种子的随机数生成器来生成k，
而是用交易数据本身做种子的确定性随机过程。
这确保每个交易生成不同的 `k` 值。
`IETF RFC 6979`定义了业界标准算法，用于 `k` 值的确定性初始化。

If you are implementing an algorithm to sign transactions in bitcoin, you *must* use RFC 6979 or a similarly deterministic-random algorithm to ensure you generate a different *k* for each transaction.
如果你正在实现一个算法来对交易进行签名，必须使用`RFC 6979`或类似的确定性随机算法，以确保为每个交易生成的 `k` 值都不同。

6.6比特币地址、余额、其它抽象

We began this chapter with the discovery that transactions look very different "behind the scenes" than how they are presented in wallets, blockchain explorers, and other user-facing applications. Many of the simplistic and familiar concepts from the earlier chapters, such as bitcoin addresses and balances, seem to be absent from the transaction structure. We saw that transactions don't contain bitcoin addresses, per se, but instead operate through scripts that lock and unlock discrete values of bitcoin. Balances are not present anywhere in this system and yet every wallet application prominently displays the balance of the user's wallet.
交易的幕后看起来与它在钱包、区块链浏览器和其它用户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，例如比特币地址和余额，似乎不在交易结构中。
我们看到，交易并不包含比特币地址，而是通过锁定来操作，脚本锁定和解锁了比特币的价值。
这个系统中也不存在余额，但每个钱包都会给用户显示余额。

Now that we have explored what is actually included in a bitcoin transaction, we can examine how the higher-level abstractions are derived from the seemingly primitive components of the transaction.
既然已经知道了比特币交易中实际包含的内容，我们就来看看如何从交易的原始数据中导出高级抽象。

Let's look again at how Alice's transaction was presented on a popular block explorer (Alice's transaction to Bob's Cafe).
我们再来看看区块浏览器中呈现的Alice的交易。

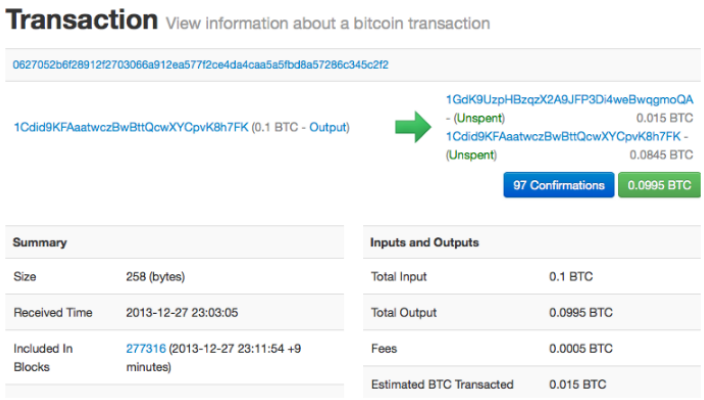


Figure 7. Alice's transaction to Bob's Cafe

On the left side of the transaction, the blockchain explorer shows Alice's bitcoin address as the "sender." In fact, this information is not in the transaction itself. When the blockchain explorer retrieved the transaction it also retrieved the previous transaction referenced in the input and extracted the first output from that older transaction. Within that output is a locking script that locks the UTXO to Alice's public key hash (a P2PKH script). The blockchain explorer extracted the public key hash and encoded it using Base58Check encoding to produce and display the bitcoin address that represents that public key.
在这个交易的左侧，区块浏览器把Alice的比特币地址显示为“发送者”。
实际上，这个信息并不在交易中。当区块链接浏览器获取到这个交易时，它还获取到了输入引用的前一个交易。在那个交易中，输出是一个锁定脚本，把这个UTXO锁定给Alice的公钥哈希。
区块链浏览器提取公钥哈希，使用Base58Check对其进行编码，以生成和显示表示该公钥的比特币地址。

Similarly, on the right side, the blockchain explorer shows the two outputs; the first to Bob's bitcoin address and the second to Alice's bitcoin address (as change). Once again, to create these bitcoin addresses, the blockchain explorer extracted the locking script from each output, recognized it as a P2PKH script, and extracted the public-key-hash from

within. Finally, the blockchain explorer reencoded that public key hash with Base58Check to produce and display the bitcoin addresses.
同样，在右侧，区块浏览器显示了两个输出。
第1个到Bob的比特币地址，第2个到Alice的比特币地址（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，认出它是P2PKH脚本，就从中提取公钥哈希。最后，用Base58Check编码公钥哈希以生成和显示比特币地址。

If you were to click on Bob's bitcoin address, the blockchain explorer would show you the view in The balance of Bob's bitcoin address.
如果你点击Bob的比特币地址，区块链接浏览器会显示图8。

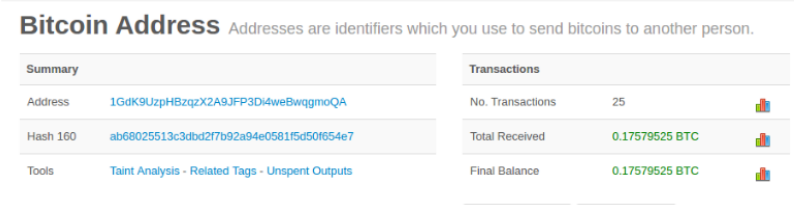


Figure 8. The balance of Bob's bitcoin address
图8：Bob的比特币地址的余额

The blockchain explorer displays the balance of Bob's bitcoin address. But nowhere in the bitcoin system is there a concept of a "balance." Rather, the values displayed here are constructed by the blockchain explorer as follows.
区块链浏览器显示了Bob的比特币地址的余额。
但是比特币系统中没有“余额”的概念。
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。

To construct the "Total Received" amount, the blockchain explorer first will decode the Base58Check encoding of the bitcoin address to retrieve the 160-bit hash of Bob's public key that is encoded within the address. Then, the blockchain explorer will search through the database of transactions, looking for outputs with P2PKH locking scripts that contain Bob's public key hash. By summing up the value of all the outputs, the blockchain explorer can produce the total value received.
为了获得“Total Received”的值，区块链浏览器首先解码比特币地址的Base58Check编码，以获取Bob的公钥哈希（160位）。
然后，区块链浏览器搜索交易数据库，查找这种输出：P2PKH锁定脚本包含Bob的公钥哈希。
通过汇总所有输出的值，浏览器就得到了“Total Received”的值。

Constructing the current balance (displayed as "Final Balance") requires a bit more work. The blockchain explorer keeps a separate database of the outputs that are currently unspent, the UTXO set. To maintain this database, the blockchain explorer must monitor the bitcoin network, add newly created UTXO, and remove spent UTXO, in real time, as they appear in unconfirmed transactions. This is a complicated process that depends on keeping track of transactions as they propagate, as well as maintaining consensus with the bitcoin network to ensure that the correct chain is followed. Sometimes, the blockchain explorer goes out of sync and its perspective of the UTXO set is incomplete or incorrect.
得到当前余额（Final Balance）需要多一点工作。
区块链接浏览器保存一个单独的输出数据库，它们是当前的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，删除已花费的UTXO，在它们出现在未确认交易中时实时地做。
这是一个复杂的过程，依赖于跟踪被传播的交易，还要维护与比特币网络的共识，以保证跟踪正确的链。有时，区块链浏览器未能保持同步，它的UTXO集是不完整或不正确的。

From the UTXO set, the blockchain explorer sums up the value of all unspent outputs referencing Bob's public key hash and produces the "Final Balance" number shown to the user.
从UTXO集中，区块链浏览器汇总了引用Bob的公钥哈希的所有UTXO的值，就生成了Final Balance。

In order to produce this one image, with these two "balances," the blockchain explorer has to index and search through dozens, hundreds, or even hundreds of thousands of transactions.
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和搜索很多交易。

In summary, the information presented to users through wallet applications, blockchain explorers, and other bitcoin user interfaces is often composed of higher-level abstractions that are derived by searching many different transactions, inspecting their content, and manipulating the data contained within them. By presenting this simplistic view of bitcoin transactions that resemble bank checks from one sender to one recipient, these applications have to abstract a lot of underlying detail. They mostly focus on the common types of transactions: P2PKH with SIGHASH_ALL signatures on every input. Thus, while bitcoin applications can present more than 80% of all transactions in an easy-to-read manner, they are sometimes stumped by transactions that deviate from the norm. Transactions that contain more complex locking scripts, or different SIGHASH flags, or many inputs and outputs, demonstrate the simplicity and weakness of these abstractions.
总之，钱包、区块链浏览器和其它比特币用户界面呈现给用户的信息，通常有更高级抽象组成，它们是这样得到的：搜索不同的交易，检查交易的内容，操作交易中包含的数据。
为了把比特币交易呈现出类似于银行支票从发送人到接收人的这种简单视图，这些应用必须抽象许多底层细节。
它们主要关注常见的交易类型：P2PKH在每个输入中有SIGHASH_ALL签名。
因此，虽然比特币应用可以以易阅的方式呈现80%以上的交易，但有时会被不常见的交易难住。
下面这些交易就说明了这些抽象的简单性和弱点：

- 包含更复杂的锁定脚本
- 包含不同SIGHASH标志
- 有许多输入和输出

Every day, hundreds of transactions that do not contain P2PKH outputs are confirmed on the blockchain. The blockchain explorers often present these with red warning messages saying they cannot decode an address. The following link contains the most recent "strange transactions" that were not fully decoded: https://blockchain.info/strange-transactions.
每天，区块链上会确认数百个不包含P2PKH输出的交易。
区块链浏览器经常用红色警告信息来呈现它们，表示无法解码地址。
下面的链接包含最近的“奇怪交易”，它们不能被完全解码：
https://blockchain.info/strange-transactions

As we will see in the next chapter, these are not necessarily strange transactions. They are transactions that contain more complex locking scripts than the common P2PKH. We will learn how to decode and understand more complex scripts and the applications they support next.
我们将在下一章看到，这些不一定是奇怪的交易。
它们包含更复杂的锁定脚本，而不是常见的P2PKH。
我们将学习如何解码和理解更复杂的脚本，以及它们支持的应用。