7高级交易和脚本

7.1介绍

In the previous chapter, we introduced the basic elements of bitcoin transactions and looked at the most common type of transaction script, the P2PKH script. In this chapter we will look at more advanced scripting and how we can use it to build transactions with complex conditions.
上一章介绍了比特币交易的基本元素，看到了最常见的交易脚本类型（P2PKH）。
本章介绍更高级的脚本，以及如何用它来构建具有复杂条件的交易。

First, we will look at *multisignature* scripts. Next, we will examine the second most common transaction script, *Pay-to-Script-Hash*, which opens up a whole world of complex scripts. Then, we will examine new script operators that add a time dimension to bitcoin, through *timelocks*. Finally, we will look at *Segregated Witness*, an architectural change to the structure of transactions.
首先，我们看看“多签名脚本”。
然后，看看第二种最常见的交易脚本P2SH，它打开了一个复杂脚本的世界。
然后，看看新的脚本操作码，它们通过时间锁为比特币加了一个时间维度。
最后，看看隔离见证，它是对交易结构的一个架构修改。

7.2多签名

Multisignature scripts set a condition where N public keys are recorded in the script and at least M of those must provide signatures to unlock the funds. This is also known as an M-of-N scheme, where N is the total number of keys and M is the threshold of signatures required for validation. For example, a 2-of-3 multisignature is one where three public keys are listed as potential signers and at least two of those must be used to create signatures for a valid transaction to spend the funds.
“多签名脚本”设置了一个条件，脚本中记录了N个公钥，必须提供M个签名才能解锁资金。
这称为M-N方案，N是密钥总数，M是验证所需的签名数量。
例如，2-3多签名，有3个公钥，至少要2个有效签名才能花费资金。

At this time, *standard* multisignature scripts are limited to at most 3 listed public keys, meaning you can do anything from a 1-of-1 to a 3-of-3 multisignature or any combination within that range. The limitation to 3 listed keys might be lifted by the time this book is published, so check the IsStandard() function to see what is currently accepted by the network. Note that the limit of 3 keys applies only to standard (also known as "bare") multisignature scripts, not to multisignature scripts wrapped in a Pay-to-Script-Hash (P2SH) script. P2SH multisignature scripts are limited to 15 keys, allowing for up to 15-of-15 multisignature. We will learn about P2SH in Pay-to-Script-Hash (P2SH).
当前，标准的多签名脚本限制最多3个公钥。
这个数量限制可能会改变，所以，查看IsStandard()函数，看看网络当前接受的值是什么。
注意，3个密钥的限制只应用于标准多签名脚本，而不应用于P2SH脚本中包装的多签名脚本。
P2SH多签名脚本限制为15个密钥。

The general form of a locking script setting an M-of-N multisignature condition is:
设置M-N多签名条件的锁定脚本的一般形式是：

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N CHECKMULTISIG
```

where N is the total number of listed public keys and M is the threshold of required signatures to spend the output.
N是列出的公钥总数，M是花费输出所需的签名数量。

A locking script setting a 2-of-3 multisignature condition looks like this:
2-3多签名条件的锁定脚本如下所示：

```
2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG
```

The preceding locking script can be satisfied with an unlocking script containing pairs of signatures and public keys:
上面的锁定脚本可用一个解锁脚本来满足，它包含了签名。（ddk：锁定脚本中包含了公钥）

```
<Signature B> <Signature C>
```

or any combination of two signatures from the private keys corresponding to the three listed public keys.
或者,由3个公钥中的任意2个相一致的私钥签名组合予以解锁。

The two scripts together would form the combined validation script:
两个脚本组合将形成验证脚本：

```
<Signature B> <Signature C>
2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG
```

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 this case, the condition is whether the unlocking script has a valid signature from the two private keys that correspond to two of the three public keys set as an encumbrance.
当执行时，只有解锁脚本匹配了锁定脚本的条件时，这个组合脚本的结果才为Ture。
在本例中，条件是：解锁脚本中是否有两个私钥的有效签名。

CHECKMULTISIG执行中的一个bug

There is a bug in CHECKMULTISIG's execution that requires a slight workaround. When CHECKMULTISIG executes, it should consume M+N+2 items on the stack as parameters. However, due to the bug, CHECKMULTISIG will pop an extra value or one value more than expected.
CHECKMULTISIG的执行中有一个bug，需要一个解决方法。
当CHECKMULTISIG执行时，它应该消耗栈上的M+N+2个项目作为参数。
然而，由于有bug，CHECKMULTISIG会弹出一个额外的值，或超出预期的一个值。

Let’s look at this in greater detail using the previous validation example:
我们使用前面的验证例子来详细看看。

```
<Signature B> <Signature C>
2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG
```

First, CHECKMULTISIG pops the top item, which is N (in this example "3"). Then it pops N items, which are the public keys that can sign. In this example, public keys A, B, and C. Then, it pops one item, which is M, the quorum (how many signatures are needed). Here M = 2. At this point, CHECKMULTISIG should pop the final M items, which are the signatures, and see if they are valid.
首先，CHECKMULTISIG弹出最上面的项，它是N，在本例中是3。
然后弹出N个项，这是可以签名的公钥。在本例中，是公钥A、B、C。
然后，弹出M（需要多少个签名），这里M = 2。
此时，应该弹出最后的M个项，这些是签名，并查看它们是否有效。

However, unfortunately, a bug in the implementation causes CHECKMULTISIG to pop one more item (M+1 total) than it should. The extra item is disregarded when checking the signatures so it has no direct effect on CHECKMULTISIG itself. However, an extra value must be present because if it is not present, when CHECKMULTISIG attempts to pop on an

empty stack, it will cause a stack error and script failure (marking the transaction as invalid). Because the extra item is disregarded it can be anything, but customarily 0 is used.
但是，不幸的是，实现中有一个bug，导致CHECKMULTISIG多弹出了一个项（总共M+1个）。
检查签名时不考虑这个额外的项，因此它对CHECKMULTISIG本身没有直接影响。
但是，必须存在一个额外的值，因为如果不存在，则当CHECKMULTISIG尝试弹出空栈时，会导致栈错误和脚本失败（将交易标记为无效）。
因为这个额外的项被忽略，所以它可以是任何东西，但通常使用0。

Because this bug became part of the consensus rules, it must now be replicated forever. Therefore the correct script validation would look like this:
因为这个bug已成为共识规则的一部分，所以现在它必须永远存在。
因此，正确的脚本验证将如下所示：

```
0 <Signature B> <Signature C>
2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG
```

Thus the unlocking script actually used in multisig is not:
这样，多签名中使用的解锁脚本不是下面这个：

```
<Signature B> <Signature C>
```

but instead it is:
而是这个：

```
0 <Signature B> <Signature C>
```

From now on, if you see a multisig unlocking script, you should expect to see an extra 0 in the beginning, whose only purpose is as a workaround to a bug that accidentally became a consensus rule.
从现在开始，如果你看到一个多签名解锁脚本，你应该期望看到开头有一个额外的0，其唯一的目的是解决一个bug，而这个解决方法意外地成了一个共识规则。

7.3 P2SH

Pay-to-Script-Hash (P2SH) was introduced in 2012 as a powerful new type of transaction that greatly simplifies the use of complex transaction scripts. To explain the need for P2SH, let’s look at a practical example.
P2SH是在2012年引入的，它是一种新的强大交易类型，能极大简化复杂交易脚本的使用。
为解释为什么需要P2SH，我们先看一个实际的例子。

In [ch01_intro_what_is_bitcoin] we introduced Mohammed, an electronics importer based in Dubai. Mohammed’s company uses bitcoin’s multisignature feature extensively for its corporate accounts. Multisignature scripts are one of the most common uses of bitcoin’s advanced scripting capabilities and are a very powerful feature. Mohammed’s company uses a multisignature script for all customer payments, known in accounting terms as "accounts receivable," or AR. With the multisignature scheme, any payments made by customers are locked in such a way that they require at least two signatures to release, from Mohammed and one of his partners or from his attorney who has a backup key. A multisignature scheme like that offers corporate governance controls and protects against theft, embezzlement, or loss.
Mohammed是迪拜的进口电子产品的商人。
Mohammed的公司在公司账目中广泛使用了比特币的多签名功能。
多签名脚本是比特币高级脚本最为常使用的一个，它有很强的功能。
对于所有客户支付（会计学中称为“应收账目AR”），公司使用了一个多签名脚本。
使用这个多签名机制，客户的支付被锁定为：至少需要两个签名才能解锁（Mohammed、3个合伙人、律师）。
多签名机制能为公司提供了治理控制，并能防范盗窃、挪用和丢失。

The resulting script is quite long and looks like this:
最终的脚本很长，如下：

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key>
<Attorney Public Key> 5 CHECKMULTISIG
```

Although multisignature scripts are a powerful feature, they are cumbersome to use. Given the preceding script, Mohammed would have to communicate this script to every customer prior to payment. Each customer would have to use special bitcoin wallet software with the ability to create custom transaction scripts, and each customer would have to understand how to create a transaction using custom scripts. Furthermore, the resulting transaction would be about five times larger than a simple payment transaction, because this script contains very long public keys. The burden of that extra-large transaction would be borne by the customer in the form of fees. Finally, a large transaction script like this would be carried in the UTXO set in RAM in every full node, until it was spent. All of these issues make using complex locking scripts difficult in practice.
虽然多签名十分强大，但其使用起来很不不便。

- 每个客户支付之前，Mohammed必须把脚本发送给客户。
  每个客户还必须使用特殊的比特币钱包（能够创建定制交易脚本），还要理解如何使用定制脚本来创建交易。
- 此外，生成的交易是简单支付交易的5倍长，因为脚本包含很长的公钥。
  这使客户要负担更多的交易费。
- 最后，在全节点中，RAM中要保存UTXO（其中有锁定脚本），直到这个UTXO被花掉为止。

所以，在实际中，使用复杂锁定脚本就很困难。

P2SH was developed to resolve these practical difficulties and to make the use of complex scripts as easy as a payment to a bitcoin address. With P2SH payments, the complex locking script is replaced with its digital fingerprint, a cryptographic hash. When a transaction attempting to spend the UTXO is presented later, it must contain the script that matches the hash, in addition to the unlocking script. In simple terms, P2SH means "pay

to a script matching this hash, a script that will be presented later when this output is spent."
P2SH用来解决这些实际困难，使复杂脚本的使用与支付给比特币地址支付一样简单。
有了P2SH 支付，使用锁定脚本的数字指纹来替代锁定脚本，数字指纹是一个加密哈希。
当后面的交易试图花费UTXO时，除了解锁脚本之外，它还必须包含与这个哈希相匹配的脚本。
简单来说，P2SH的含义是：向脚本的哈希进行支付，当花费这个输出时，再提供这个脚本。

In P2SH transactions, the locking script that is replaced by a hash is referred to as the *redeem script* because it is presented to the system at redemption time rather than as a locking script. Complex script without P2SH shows the script without P2SH and Complex script as P2SH shows the same script encoded with P2SH.
在P2SH交易中，使用哈希替代了锁定脚本，锁定脚本被称为赎回脚本，因为是在赎回时才提供脚本，而不是作为锁定脚本来提供。
表1显示了没用P2SH的脚本，表2显示了使用P2SH的脚本。

Table 1. Complex script without P2SH

Locking Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Unlocking Script	Sig1 Sig2

Table 2. Complex script as P2SH

Redeem Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Locking Script	HASH160 <20-byte hash of redeem script> EQUAL
Unlocking Script	Sig1 Sig2 <redeem script>

As you can see from the tables, with P2SH the complex script that details the conditions for spending the output (redeem script) is not presented in the locking script. Instead, only a hash of it is in the locking script and the redeem script itself is presented later, as part of the unlocking script when the output is spent. This shifts the burden in fees and complexity from the sender to the recipient (spender) of the transaction.
可以看出，对于P2SH，详细列出输出支付条件的复杂脚本（赎回脚本）并不在锁定脚本中。
它的哈希在锁定脚本中，在以后花费输出时，在解锁脚本中提供赎回脚本。
这使交易费和复杂性从付款方转给到了收款方。

Let’s look at Mohammed’s company, the complex multisignature script, and the resulting P2SH scripts.
我们再看看Mohammed公司的例子，复杂的多签名脚本，生成的P2SH脚本。

First, the multisignature script that Mohammed’s company uses for all incoming payments from customers:
首先，Mohammed公司对所有客户支付使用的多签名脚本：

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public
Key> <Attorney Public Key> 5 CHECKMULTISIG
```

If the placeholders are replaced by actual public keys (shown here as 520-bit numbers starting with 04) you can see that this script becomes very long:
如果显示实际的公钥（以04开头的520位），你看到的脚本会非常长。
参见4.2.3.1公钥的格式：公钥表示为520比特的数字（64字节，130个十六进制数字），它04开头，之后是(x,y) 坐标，所以格式为04 x y。

```
2
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5
C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587
04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0
B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49
```

```
047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D44
20D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965
0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044D
ADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5
043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A
1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800
5 CHECKMULTISIG
```

This entire script can instead be represented by a 20-byte cryptographic hash, by first applying the SHA256 hashing algorithm and then applying the RIPEMD160 algorithm on the result.
这个脚本可用一个加密哈希表示，只有20个字节：
首先使用SH256哈希算法，随后使用RIPEMD160算法。

We use libbitcoin-explorer (bx) on the command-line to produce the script hash, as follows:
我们在命令行中使用 `libbitcoin-explorer (bx)`来生成脚本哈希，如下：

```
echo \
2 \
[04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC
5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587] \
[04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D
0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49] \
[047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4
420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965] \
[0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044
DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5] \
[043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29
A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800] \
5 CHECKMULTISIG \
| bx script-encode | bx sha256 | bx ripemd160
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

The series of commands above first encodes Mohammed’s multisig redeem script as a serialized hex-encoded bitcoin Script. The next bx command calculates the SHA256 hash of that. The next bx command hashes again with RIPEMD160, producing the final script-hash:
上面命令的解释：

- 首先，把Mohammed的多签名赎回脚本编码为一个序列化十六进制编码的比特币脚本。
- 然后，计算它的SHA256哈希
- 最后，用RIPEMD160计算哈希，就生成了脚本哈希。

The 20-byte hash of Mohammed’s redeem script is:
这个赎回脚本的哈希是：

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

A P2SH transaction locks the output to this hash instead of the longer redeem script, using the locking script:
P2SH交易把输出锁定到这个哈希，使用如下锁定脚本：

```
HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL
```

which, as you can see, is much shorter. Instead of "pay to this 5-key multisignature script," the P2SH equivalent transaction is "pay to a script with this hash." A customer making a payment to Mohammed’s company need only include this much shorter locking script in his payment.
这个锁定脚本比前面的多签名锁定脚本短多了。
多签名锁定脚本是：向这5个密钥的多签名脚本支付。
P2SH锁定脚本表示：向具有这个哈希的脚本支付。
客户在支付时，只需在其支付中包含这个简短的锁定脚本即可。

When Mohammed and his partners want to spend this UTXO, they must present the original redeem script (the one whose hash locked the UTXO) and the signatures necessary to unlock it, like this:
当 Mohammed和合伙人想要花费这个UTXO时，他们必须提供原始的赎回脚本，以及解锁需要的签名：

```
<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG>
```

The two scripts are combined in two stages. First, the redeem script is checked against the locking script to make sure the hash matches:
这两个脚本用两步实现合并。

```
Sig1 Sig2 <redeem script>
HASH160 <20-byte hash of redeem script> EQUAL
```

首先，将赎回脚本与锁定脚本对比，以确认赎回脚本的哈希与锁定脚本中的哈希匹配：

```
<2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG>
HASH160 <redeem scriptHash> EQUAL
```

If the redeem script hash matches, the unlocking script is executed on its own, to unlock the redeem script:
如果匹配，就会执行解锁脚本，以解锁赎回脚本：

```
<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG
```

Almost all the scripts described in this chapter can only be implemented as P2SH scripts. They cannot be used directly in the locking script of an UTXO.
本章描述的几乎所有脚本都只能实现为P2SH脚本。
它们不能直接用在UTXO的锁定脚本中。

7.3.1 P2SH地址

Another important part of the P2SH feature is the ability to encode a script hash as an address, as defined in BIP-13. P2SH addresses are Base58Check encodings of the 20-byte hash of a script, just like bitcoin addresses are Base58Check encodings of the 20-byte hash of a public key. P2SH addresses use the version prefix "5," which results in Base58Check-encoded addresses that start with a "3."
P2SH功能的另一重要部分是：它能把一个脚本哈希编码为一个BIP-13定义的地址。
P2SH地址是脚本的20字节哈希的Base58Check编码，
比特币地址是公钥的20字节哈希的Base58Check编码。
P2SH地址使用的版本前缀是“5”，所以，Base58Check编码的地址以“3”开头。

For example, Mohammed’s complex script, hashed and Base58Check-encoded as a P2SH address, becomes 39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw. We can confirm that with the bx command:
例如，Mohammed的P2SH地址是：39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw
我们可以用bx命令来确认它：

```
echo '54c557e07dde5bb6cb791c7a540e0a4796f5e97e' | bx address-encode -v 5
39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw
```

Now, Mohammed can give this "address" to his customers and they can use almost any bitcoin wallet to make a simple payment, as if it were a bitcoin address. The 3 prefix gives them a hint that this is a special type of address, one corresponding to a script instead of a public key, but otherwise it works in exactly the same way as a payment to a bitcoin address.
现在，Mohammed可以把这个地址给客户，客户可以使用几乎任何比特币钱包来做一个简单支付，就像这是一个比特币地址一样。
地址前缀“3”表示它对应一个脚本，而不是一个公钥，但其它方面与支付给一个比特币地址完全一样。

P2SH addresses hide all of the complexity, so that the person making a payment does not see the script.
P2SH地址隐藏了所有的复杂性，因此，付款人看不到这个脚本。

7.3.2 P2SH的优点

The P2SH feature offers the following benefits compared to the direct use of complex scripts in locking outputs:
相比直接在锁定输出中使用复杂脚本来说，P2SH具有以下优点。

- Complex scripts are replaced by shorter fingerprints in the transaction output, making the transaction smaller.
  在交易输出中，用短指纹替代了复杂脚本，使得交易更短。

- Scripts can be coded as an address, so the sender and the sender’s wallet don’t need complex engineering to implement P2SH.
  脚本能被编码为地址，所以，付款人和付款人的钱包不需要复杂的工作就能实现P2SH。

- P2SH shifts the burden of constructing the script to the recipient, not the sender.
  P2SH把构建脚本的负担转移至收款方，而不是付款方。

- P2SH shifts the burden in data storage for the long script from the output (which additionally to being stored on the blockchain is in the UTXO set) to the input (only stored on the blockchain).
  P2SH将长脚本数据存储的负担从输出（存储在UTXO集，耗费内存）转移至输入（存储在区块链上）。

- P2SH shifts the burden in data storage for the long script from the present time (payment) to a future time (when it is spent).
  P2SH将长脚本数据存储的负担从现在（付款时）转移至未来（花费时）。

- P2SH shifts the transaction fee cost of a long script from the sender to the recipient, who has to include the long redeem script to spend it.
  P2SH将长脚本的交易费从付款方转移至收款方。

7.3.3赎回脚本和验证

Prior to version 0.9.2 of the Bitcoin Core client, Pay-to-Script-Hash was limited to the standard types of bitcoin transaction scripts, by the IsStandard() function. That means that the redeem script presented in the spending transaction could only be one of the standard types: P2PK, P2PKH, or multisig.
在Bitcoin Core客户端0.9.2版本之前，P2SH被限制为标准类型的比特币交易脚本，通过IsStandard()函数。
这意味着，花费交易中的赎回脚本只能是标准类型：P2PK、P2PKH、多签名。

As of version 0.9.2 of the Bitcoin Core client, P2SH transactions can contain any valid script, making the P2SH standard much more flexible and allowing for experimentation with many novel and complex types of transactions.
从0.9.2版本开始，P2SH交易可以包含任意有效的脚本，这使得P2SH标准更加灵活，可以实现许多新颖和复杂的交易类型。

You are not able to put a P2SH inside a P2SH redeem script, because the P2SH specification is not recursive. Also, while it is technically possible to include RETURN (see Data Recording Output (RETURN)) in a redeem script, as nothing in the rules prevents

you from doing so, it is of no practical use because executing RETURN during validation will cause the transaction to be marked invalid.
不能在一个P2SH赎回脚本中放一个P2SH，因为P2SH规范不能递归。
还有，虽然在技术上可以在赎回脚本中包含RETURN（因为规则并不阻止这么做），但这样做并不实用，因为验证期间执行RETURN将导致交易无效。

Note that because the redeem script is not presented to the network until you attempt to spend a P2SH output, if you lock an output with the hash of an invalid redeem script it will be processed regardless. The UTXO will be successfully locked. However, you will not be able to spend it because the spending transaction, which includes the redeem script, will not be accepted because it is an invalid script. This creates a risk, because you can lock bitcoin in a P2SH that cannot be spent later. The network will accept the P2SH locking script even if it corresponds to an invalid redeem script, because the script hash gives no indication of the script it represents.
注意，因为只有在花费P2SH输出时，才向网络提供赎回脚本，所以，如果你用一个无效的赎回脚本的哈希来锁定输出，它还是会被处理。
这个UTXO会被成功地锁定，但是你再也不能花费这笔资金，因为你无法提供有效的赎回脚本。
这造成了一个风险，因为你可能把比特币锁定在一个以后不能花费的P2SH中。
网络会接受P2SH锁定脚本，即使它对应的是一个无效的赎回脚本，因为脚本哈希对脚本没有给出任何指示。

Warning：P2SH locking scripts contain the hash of a redeem script, which gives no clues as to the content of the redeem script itself. The P2SH transaction will be considered valid and accepted even if the redeem script is invalid. You might accidentally lock bitcoin in such a way that it cannot later be spent.
警告：P2SH锁定脚本包含一个赎回脚本的哈希，这个哈希对赎回脚本的内容没有提供任何线索。
P2SH交易会被认为是有效的，并被接受，即使赎回脚本是无效的。
你可能会偶然把锁定了比特币，但以后却不能花费。

7.4 用于数据记录的输出（RETURN）

Data Recording Output (RETURN)

Bitcoin’s distributed and timestamped ledger, the blockchain, has potential uses far beyond payments. Many developers have tried to use the transaction scripting language to take advantage of the security and resilience of the system for applications such as digital notary services, stock certificates, and smart contracts. Early attempts to use bitcoin’s script language for these purposes involved creating transaction outputs that recorded data on the blockchain; for example, to record a digital fingerprint of a file in such a way that anyone could establish proof-of-existence of that file on a specific date by reference to that transaction.
比特币的分布式和时间戳账本（即区块链）的潜在应用不只是支付。
许多开发者试图使用交易脚本语言，把这个系统的安全和弹性用于其它应用，例如：数字公证服务、股票凭证、智能合约。
把比特币脚本语言用于这些目的早期尝试涉及创建交易输出，用输出把数据记录在区块链中。
例如，用这种方法记录一个文件的数字指纹，则任何人可以用那个交易证明某天那个文件已经存在。

The use of bitcoin’s blockchain to store data unrelated to bitcoin payments is a controversial subject. Many developers consider such use abusive and want to discourage it. Others view it as a demonstration of the powerful capabilities of blockchain technology and want to encourage such experimentation.
这是一个有争议的话题：使用比特币的区块链来存储与比特币支付不相关的数据。
许多开发者认为这是滥用，并试图予以阻止。
另一些开发者则将之看做区块链技术强大功能的一个证明，并予以鼓励。

Those who object to the inclusion of nonpayment data argue that it causes "blockchain bloat," burdening those running full bitcoin nodes with carrying the cost of disk storage for data that the blockchain was not intended to carry. Moreover, such transactions create UTXO that cannot be spent, using the destination bitcoin address as a freeform 20-byte field. Because the address is used for data, it doesn’t correspond to a private key and the resulting UTXO can *never* be spent; it’s a fake payment. These transactions that can never be spent are therefore never removed from the UTXO set and cause the size of the UTXO database to forever increase, or "bloat."
反对的人认为这样做会引致“区块链膨胀”，使全节点要存储与交易无关的数据。
此外，这种交易创建的UTXO不能被花费，它将目的比特币地址用作一个随意写的20字节字段。
因为这个地址保存的是数据，没有对应的私钥，因此不会花费这个UTXO，它是一个假的支付。
因此，这些交易永远不会被花费，也不会从UTXO集中删除，导致UTXO数据库只会“膨胀”。

In version 0.9 of the Bitcoin Core client, a compromise was reached with the introduction of the RETURN operator. RETURN allows developers to add 80 bytes of nonpayment data to a transaction output. However, unlike the use of "fake" UTXO, the RETURN operator creates an explicitly *provably unspendable* output, which does not need to be stored in the UTXO set. RETURN outputs are recorded on the blockchain, so they consume disk space and contribute to the increase in the blockchain’s size, but they are not stored in the UTXO set and therefore do not bloat the UTXO memory pool and burden full nodes with the cost of more expensive RAM.
在Bitcoin Core客户端0.9版本中，通过引入Return操作码达成了妥协。
Return允许开发者在交易输出上加入80字节的非交易数据。
但是，与假的UTXO不同，RETURN操作码创建了一个明确可证明的不可花费输出，不需要把它存在UTXO集中。
RETURN输出记录在区块链上，所以它们会消耗磁盘空间，使区块链增大，但不存储在UTXO集中，因此不会使UTXO内存池膨胀，所以不会让全节点负担更多的RAM成本。

RETURN scripts look like this:
RETURN 脚本是这样：

```
RETURN <data>
```

The data portion is limited to 80 bytes and most often represents a hash, such as the output from the SHA256 algorithm (32 bytes). Many applications put a prefix in front of the data to help identify the application. For example, the Proof of Existence digital notarization service uses the 8-byte prefix DOCPROOF, which is ASCII encoded as 44 4f 43 50 52 4f 4f 46 in hexadecimal.
data限制为80字节，多表示为一个哈希，例如SHA256算法输出（32字节）。
许多应用在data前面放一个前缀，帮助识别这个应用。
例如，一种电子公正服务使用8个字节的前缀“DOCPROOF”，十六进制是 44 4f 43 50 52 4f 4f 46。

Keep in mind that there is no "unlocking script" that corresponds to RETURN that could possibly be used to "spend" a RETURN output. The whole point of RETURN is that you can’t spend the money locked in that output, and therefore it does not need to be held in the UTXO set as potentially spendable—RETURN is *provably unspendable*. RETURN is usually an output with a zero bitcoin amount, because any bitcoin assigned to such an output is effectively lost forever. If a RETURN is referenced as an input in a transaction, the script validation engine will halt the execution of the validation script and mark the transaction as invalid. The execution of RETURN essentially causes the script to "RETURN" with a FALSE and halt. Thus, if you accidentally reference a RETURN output as an input in a transaction, that transaction is invalid.
记住，没有解锁脚本对应RETURN。
RETURN的要点是：你不能花费在这个输出中锁定的钱，因此不需要把它放在UTXO集中。
RETURN通常是的零比特币输出，因为任何与该输出相对应的比特币都会永久丢失。
如果一个 RETURN 被作为一笔交易的输入，脚本验证引擎将会停止验证脚本的执行，将交易标记为无效。如果你偶然将 RETURN 的输出作为另一笔交易的输入，则该交易是无效的。

A standard transaction (one that conforms to the IsStandard() checks) can have only one RETURN output. However, a single RETURN output can be combined in a transaction with outputs of any other type.
一个标准交易（通过IsStandard()函数检验）只能有一个 RETURN 输出。
但是，一个RETURN 输出可以与其它类型的输出一起出现在一个交易中。

Two new command-line options have been added in Bitcoin Core as of version 0.10. The option datacarrier controls relay and mining of RETURN transactions, with the default set to "1" to allow them. The option datacarriersize takes a numeric argument specifying the maximum size in bytes of the RETURN script, 83 bytes by default, which, allows for a maximum of 80 bytes of RETURN data plus one byte of RETURN opcode and two bytes of PUSHDATA opcode.
Bitcoin Core版本0.10中添加了两个新的命令行选项。
选项datacarrier控制RETURN交易的传播和挖矿，默认为“1”，表示允许。
选项datacarriersize指定一个数字作为参数，它指定RETURN脚本的最大字节数，默认是83字节：data（最多80字节），RETURN操作码（1个字节），PUSHDATA操作码（2个字节）。

Note：RETURN was initially proposed with a limit of 80 bytes, but the limit was reduced to 40 bytes when the feature was released. In February 2015, in version 0.10 of Bitcoin Core, the limit was raised back to 80 bytes. Nodes may choose not to relay or mine RETURN, or only relay and mine RETURN containing less than 80 bytes of data.
说明：最初提出RETURN建议时限制为80字节，但是当这个功能被发布时，减少为40字节。
2015年2月，在Bitcoin Core 0.10版本中，这个限制又提高到80字节。
节点可以选择不对RETURN进行传播或挖矿，或只传播和挖矿少于80字节数据的RETURN。

ddk说明：
- 收款人把自己的比特币地址告知付款人，付款人构建交易，发布交易前，钱包软件会检查交易输出是否包含标准交易脚本，只有符合标准才会向网络传播。
- 网络节点也会检查交易输出是否符合标准，所以，在区块链上出现的交易都是符合标准的交易。

- 收款人需要检查区块中包含的交易的输出是不是自己的希望的脚本，以及包含了自己的比特币地址。这也就确保了自己能花掉收到的比特币。

7.5时间锁（Timelocks）

Timelocks are restrictions on transactions or outputs that only allow spending after a point in time. Bitcoin has had a transaction-level timelock feature from the beginning. It is implemented by the nLocktime field in a transaction. Two new timelock features were introduced in late 2015 and mid-2016 that offer UTXO-level timelocks. These are CHECKLOCKTIMEVERIFY and CHECKSEQUENCEVERIFY.
时间锁是对交易或输出的限制，它只允许在一个时间点之后才能花费。
最初时，比特币有一个交易级的时间锁功能。它是用交易中的nLocktime字段实现。
在2015年底和2016年中引入了两个新的时间锁功能，它们提供UTXO级的时间锁。
它们是CHECKLOCKTIMEVERIFY和CHECKSEQUENCEVERIFY。

Timelocks are useful for postdating transactions and locking funds to a date in the future. More importantly, timelocks extend bitcoin scripting into the dimension of time, opening the door for complex multistep smart contracts.
时间锁对这个很有用：推迟交易和将资金锁定到为了一个时间。
更重要的是，时间锁为比特币脚本加入了时间维度，从而为复杂多步骤智能合约提供了支持。

7.5.1交易Locktime（nLocktime）

From the beginning, bitcoin has had a transaction-level timelock feature. Transaction locktime is a transaction-level setting (a field in the transaction data structure) that defines the earliest time that a transaction is valid and can be relayed on the network or added to the blockchain.
最初时，比特币有一个交易级的时间锁功能。
交易的LockTime是一个交易级设置（交易数据结构中的一个字段），它定义交易有效的最早时间：可以在网络上传播这个交易，或记入区块链中。

Locktime is also known as nLocktime from the variable name used in the Bitcoin Core codebase. It is set to zero in most transactions to indicate immediate propagation and execution. If nLocktime is nonzero and below 500 million, it is interpreted as a block height, meaning the transaction is not valid and is not relayed or included in the blockchain prior to the specified block height. If it is above 500 million, it is interpreted as a Unix Epoch timestamp (seconds since Jan-1-1970) and the transaction is not valid prior to the specified time.
Locktime也称nLocktime，是来自Bitcoin Core代码中使用的变量名。
多数交易将其设置为零，表示立刻传播和执行。
如果nLockTime大于0，小于5亿，表示区块高度，意思是：在指定区块高度之前，这个交易无效，不会被传播，也不会被纳入区块链。
如果nLockTime大于5亿，表示Unix时间，意思是：在指定时间之前，这个交易是无效的。

Transactions with nLocktime specifying a future block or time must be held by the originating system and transmitted to the bitcoin network only after they become valid. If a transaction is transmitted to the network before the specified nLocktime, the transaction will be rejected by the first node as invalid and will not be relayed to other nodes. The use of nLocktime is equivalent to postdating a paper check.
如果交易的nLocktime指定为未来，这交易必须被发起系统保存，只有在交易有效后才被传播到网络上。

如果在交易在有效之前被传播到网络，第一个节点会拒绝这个交易，不会传播给其它节点。
使用nLocktime等于推迟一张支票。

7.5.1.1交易locktime的局限

nLocktime has the limitation that while it makes it possible to spend some outputs in the future, it does not make it impossible to spend them until that time. Let’s explain that with the following example.
nLocktime实现了：在未来某个时间之后，可以花费某些输出。
nLocktime没实现：在未来某个时间之前，不能花费某些输出。
我们用例子解释一下。

Alice signs a transaction spending one of her outputs to Bob’s address, and sets the transaction nLocktime to 3 months in the future. Alice sends that transaction to Bob to hold. With this transaction Alice and Bob know that:

- Bob cannot transmit the transaction to redeem the funds until 3 months have elapsed.
- Bob may transmit the transaction after 3 months.

Alice签署了一个交易，把她的一个输出支付给Bob，她将nLocktime设定为3个月之后。
Alice把这笔交易发给Bob保存。有了这个交易，Alice和Bob知道：

- 3个月之内，Bob不能传播这个交易，所以不能使用支付给他的资金。
- 3个月之后，Bob可以传播这个交易。

However:

- Alice can create another transaction, double-spending the same inputs without a locktime. Thus, Alice can spend the same UTXO before the 3 months have elapsed.
- Bob has no guarantee that Alice won’t do that.

但是：

- Alice可以创建另一个交易，双花同一输入，而没有locktime。
  这样，Alice可以在3个月之内花费同一UTXO。
- Bob不能保证Alice不会这样做。

It is important to understand the limitations of transaction nLocktime. The only guarantee is that Bob will not be able to redeem it before 3 months have elapsed. There is no guarantee that Bob will get the funds. To achieve such a guarantee, the timelock restriction must be placed on the UTXO itself and be part of the locking script, rather than on the transaction. This is achieved by the next form of timelock, called Check Lock Time Verify.
理解交易nLocktime的限制很重要。
唯一能保证的是：Bob在3个月之内，不能花费支付给他的资金。
但不能保证的是：Bob能获得支付给他的资金。
为了实现这个，必所把时间锁定限制放在UTXO上，并成为锁定脚本的一部分，而不是放在交易上。
这是通过下面的时间锁形式（CLTM）来实现的。

7.5.2检查锁定时间验证(CLTV)

In December 2015, a new form of timelock was introduced to bitcoin as a soft fork upgrade. Based on a specification in BIP-65, a new script operator called *CHECKLOCKTIMEVERIFY* (*CLTV*) was added to the scripting language. CLTV is a per-output timelock, rather than a per-transaction timelock as is the case with nLocktime. This allows for much greater flexibility in the way timelocks are applied.
2015年12月，比特币使用一个软分叉升级引入了一种新的时间锁形式。
根据BIP-65，添加了一种新的脚本操作码：CHECKLOCKTIMEVERIFY（CLTV）
CLTV是针对每个输出的时间锁，而nLocktime是针对每个交易的时间锁。
这样，在时间锁的应用方式上有了更大的灵活性。

In simple terms, by adding the CLTV opcode in the redeem script of an output it restricts the output, so that it can only be spent after the specified time has elapsed.
简单说，在输出的赎回脚本中添加了CLTV操作码，从而限制了这个输出，只能在指定时间之后花费它。

Tip: While nLocktime is a transaction-level timelock, CLTV is an output-based timelock.
提示：nLocktime是交易的时间锁，CLTV是输出的时间锁。

CLTV doesn't replace nLocktime, but rather restricts specific UTXO such that they can only be spent in a future transaction with nLocktime set to a greater or equal value.
CLTV没有取代nLocktime，而是限制特定的UTXO，
这样，只能在一个未来的交易中花费这个UTXO，这个未来交易的nLocktime要设置为更大的的值。

ddk说明：带有CLTV的交易在网上传播，写入区块链，但只能在未来的一个交易中花费，这个交易的nLocktime要大于CLTV设置的时间，所以，这个交易生效的时间必然在CLTV指定的时间之后。

The CLTV opcode takes one parameter as input, expressed as a number in the same format as nLocktime (either a block height or Unix epoch time). As indicated by the VERIFY suffix, CLTV is the type of opcode that halts execution of the script if the outcome is FALSE. If it results in TRUE, execution continues.
CLTV操作码用一个参数作为输入，这个参数是与nLocktime相同格式的数字（区块高度或Unix时间）。

如VERIFY后缀所表示的，CLTV是这种类型的操作码：
- 如果结果为FALSE，则停止执行脚本。
- 如果结果为TRUE，则继续执行脚本。

In order to lock an output with CLTV, you insert it into the redeem script of the output in the transaction that creates the output. For example, if Alice is paying Bob's address, the output would normally contain a P2PKH script like this:
为了用CLTV锁定一个输出，将其插入到输出的赎回脚本中。
例如，如果Alice向Bob支付，输出通常包含一个这样的P2PKH脚本：

```
DUP HASH160 <Bob's Public Key Hash> EQUALVERIFY CHECKSIG
```

To lock it to a time, say 3 months from now, the transaction would be a P2SH transaction with a redeem script like this:
为了把它锁定到一个时间（例如3个月以后），这个交易将是一个P2SH交易，其中有如下赎回脚本：

```
<now + 3 months> CHECKLOCKTIMEVERIFY DROP DUP HASH160 <Bob's Public Key Hash> EQUALVERIFY
CHECKSIG
```

where <now {plus} 3 months> is a block height or time value estimated 3 months from the time the transaction is mined: current block height + 12,960 (blocks) or current Unix epoch time + 7,760,000 (seconds). For now, don't worry about the DROP opcode that follows CHECKLOCKTIMEVERIFY; it will be explained shortly.

其中，<now + 3 months> 是从这个交易的挖矿时间之后3个月的区块高度或时间值：

- 当前块高度 + 12960（块）
- 当前Unix时间 + 7,760,000（秒）

先不必关心操作码DROP,下面很快就会解释。

When Bob tries to spend this UTXO, he constructs a transaction that references the UTXO as an input. He uses his signature and public key in the unlocking script of that input and sets the transaction nLocktime to be equal or greater to the timelock in the CHECKLOCKTIMEVERIFY Alice set. Bob then broadcasts the transaction on the bitcoin network.
当Bob试图花费这个UTXO时，他构建一个交易，用这个UTXO作为输入。
他在输入的解锁脚本中使用他的签名和公钥，并将"交易的nLocktime"设置为大于或等于"Alice在CHECKLOCKTIMEVERIFY中设置的时间"。
然后，Bob在比特币网络上广播这个交易。

Bob's transaction is evaluated as follows. If the CHECKLOCKTIMEVERIFY parameter Alice set is less than or equal the spending transaction's nLocktime, script execution continues (acts as if a "no operation" or NOP opcode was executed). Otherwise, script execution halts and the transaction is deemed invalid.
Bob的交易计算如下：

```
如果：Alice设置的CHECKLOCKTIMEVERIFY参数 <= 这个支付交易的nLocktime
    继续执行脚本。
否则：
    停止执行脚本，该交易被视为无效。
```

More precisely, CHECKLOCKTIMEVERIFY fails and halts execution, marking the transaction invalid if (source: BIP-65):

1. the stack is empty; or
2. the top item on the stack is less than 0; or
3. the lock-time type (height versus timestamp) of the top stack item and the nLocktime field are not the same; or
4. the top stack item is greater than the transaction's nLocktime field; or
5. the nSequence field of the input is 0xffffffff.

更确切地说，如果出现下列情况之一，CHECKLOCKTIMEVERIFY失败并停止执行，标记交易无效。

无效情况	解释
1栈为空	没有要比较的锁定时间参数
2栈顶项小于0	时间值不对
3栈顶项的锁定时间类型（高度或时间戳）与nLocktime字段不同	时间值的类型相同
4栈顶项大于交易的nLocktime	交易在CLTV指定时间之后
5输入的nSequence字段为0xFFFFFFFF	见7.5.4.1：对于有nLocktime或CLTV的交易，nSequence值必须设置为小于2^{31}，以使时间锁生效

Note：CLTV and nLocktime use the same format to describe timelocks, either a block height or the time elapsed in seconds since Unix epoch. Critically, when used together, the format of nLocktime must match that of CLTV in the outputs—they must both reference either block height or time in seconds.
说明：CLTV和nLocktime使用相同的格式来描述时间锁定，即区块高度或Unix时间。
关键是，一起使用时，nLocktime与CLTV的格式相同，都是区块高度或都是Unix时间。

After execution, if CLTV is satisfied, the time parameter that preceded it remains as the top item on the stack and may need to be dropped, with DROP, for correct execution of subsequent script opcodes. You will often see CHECKLOCKTIMEVERIFY followed by DROP in scripts for this reason.

执行之后，如果满足了CLTV，则它前面的时间参数仍然是在栈顶，需要用DROP抛弃它，才能继续正确执行后面的脚本。
为此，CHECKLOCKTIMEVERIFY后面跟着DROP。

By using nLocktime in conjunction with CLTV, the scenario described in Transaction locktime limitations changes. Alice can no longer spend the money (because it's locked with Bob's key) and Bob cannot spend it before the 3-month locktime has expired.
通过将nLocktime与CLTV结合使用，"交易locktime局限"中描述场景就发生了变化。

- Alice将不再能花费这个钱，因为它被Bob的密钥锁定了。
  因为Alice的交易已在区块链上，区块链保证了Alice无法进行双重花费。
- Bob需要等待三个月之后才能花费这个钱。
  如果Bob发布了一个交易，nLocktime小于CLTV，则不能花费Alice支付给他的钱。
  如果Bob发布了一个交易，nLocktime大于等于CLTV，可能出现如下情况：
  - 当前时间还未到nLocktime指定时间，则这个交易不会在网络上传播，因此Bob无法花费。
  - 当前时间已经超过nLocktime指定时间，则这个交易会被网络传播，则Bob可以花费了。

By introducing timelock functionality directly into the scripting language, CLTV allows us to develop some very interesting complex scripts.
The standard is defined in BIP-65 (CHECKLOCKTIMEVERIFY).
通过把时间锁定功能直接引入脚本语言，CLTV允许我们开发一些非常有趣的复杂脚本。
该标准定义在BIP-65(CHECKLOCKTIMEVERIFY)：
https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki

7.5.3相对时间锁

nLocktime and CLTV are both *absolute timelocks* in that they specify an absolute point in time. The next two timelock features we will examine are *relative timelocks* in that they specify, as a condition of spending an output, an elapsed time from the confirmation of the output in the blockchain.
nLocktime和CLTV都是绝对时间锁，它们指定了绝对时间点。
接下来的两个时间锁定功能是相对时间锁，它们将花费输出的条件指定为：
从输出在区块链中被确认开始，经过的一段时间。

Relative timelocks are useful because they allow a chain of two or more interdependent transactions to be held off chain, while imposing a time constraint on one transaction that is dependent on the elapsed time from the confirmation of a previous transaction. In other words, the clock doesn’t start counting until the UTXO is recorded on the blockchain. This functionality is especially useful in bidirectional state channels and Lightning Networks, as we will see in [state_channels].
相对时间锁很有用，因为它允许脱链保存一串交易，这些交易之间相互依赖，同时对一个交易的时间进行了限制：从前一个交易被确认开始所经过的时间。
换句话说，在这个UTXO被记录到区块链之前，这个时钟不会开始计数。
这个功能在“双向状态通道”和“闪电网络”中特别有用。

Relative timelocks, like absolute timelocks, are implemented with both a transaction-level feature and a script-level opcode. The transaction-level relative timelock is implemented as a consensus rule on the value of nSequence, a transaction field that is set in every transaction input. Script-level relative timelocks are implemented with the CHECKSEQUENCEVERIFY (CSV) opcode.
与绝对时间锁类似，相对时间锁也是用“交易级功能”和“脚本级操作码”来实现的。

- 交易级相对时间锁：实现为对nSequence的值的一个共识规则，它是每个交易输入中的一个字段。
- 脚本级相对时间锁：使用CHECKSEQUENCEVERIFY（CSV）操作码实现。

Relative timelocks are implemented according to the specifications in BIP-68, Relative lock-time using consensus-enforced sequence numbers and BIP-112, CHECKSEQUENCEVERIFY.
相对时间锁是根据BIP-68和BIP-112实现的：

- BIP-68 Relative lock-time using consensus-enforced sequence numbers
  使用共识执行序列号的相对时间锁
- BIP-112 CHECKSEQUENCEVERIFY （CSV）

BIP-68 and BIP-112 were activated in May 2016 as a soft fork upgrade to the consensus rules.
2016年5月，作为对共识规则的一个软分叉升级，激活了BIP-68和BIP-112。

7.5.4 使用nSequence的相对时间锁

Relative timelocks can be set on each input of a transaction, by setting the nSequence field in each input.
可以在交易的每个输入中设置相对时间锁，方法是：设置每个输入中的nSequence字段。

7.5.4.1 nSequence的最初意义

The nSequence field was originally intended (but never properly implemented) to allow modification of transactions in the mempool. In that use, a transaction containing inputs with nSequence value below 2^{32} - 1 (0xFFFFFFFF) indicated a transaction that was not yet "finalized." Such a transaction would be held in the mempool until it was replaced by another transaction spending the same inputs with a higher nSequence value. Once a transaction was received whose inputs had an nSequence value of 0xFFFFFFFF it would be considered "finalized" and mined.
nSequence字段最初是为了（但从未被正确实现过）：允许对内存池中的交易进行修改。
在那个用法中，nSequence值小于2^{32} – 1（0xFFFFFFFF）时，表示这个交易还没有“最终确定”。
这个交易被放在内存池中，直到被另一个交易（支付相同的输入，有更高的nSequence值）替换。
一旦收到一个交易，其输入的nSequence值为0xFFFFFFFF ，那么它被视为“最终确定”，可被挖矿。

The original meaning of nSequence was never properly implemented and the value of nSequence is customarily set to 0xFFFFFFFF in transactions that do not utilize timelocks. For transactions with nLocktime or CHECKLOCKTIMEVERIFY, the nSequence value must be set to less than 2^{31} for the timelock guards to have an effect, as explained below.
nSequence的原始含义从未被正确实现过，所以，它的值通常都设置为0xFFFFFFFF，即不使用时间锁。
对于有nLocktime或CLTV的交易，nSequence值必须设置为小于2^{31} ，以使时间锁生效，解释如下。

ddk问题：这是为什么?

7.5.4.2 nSequence作为一个共识执行的相对时间锁定

Since the activation of BIP-68, new consensus rules apply for any transaction containing an input whose nSequence value is less than 2^{31} (bit 1<<31 is not set). Programmatically, that means that if the most significant (bit 1<<31) is not set, it is a flag that means "relative locktime." Otherwise (bit 1<<31 set), the nSequence value is reserved for other uses such as enabling CHECKLOCKTIMEVERIFY, nLocktime, Opt-In-Replace-By-Fee, and other future developments.
由于BIP-68的激活，新的共识规则应用于这种交易：输入中的nSequence值小于2^{31}。
从编程上说，如果最高位为0，表示“相对锁定时间”。
否则（最高位设置为1），nSequence值被保留用于其它用途，例如：
使能CHECKLOCKTIMEVERIFY、nLocktime、Opt-In-Replace-By-Fee、其它未来的开发。

ddk问题：这些用途是什么意思？如何使用?

Transaction inputs with nSequence values less than 2^{31} are interpreted as having a relative timelock. Such a transaction is only valid once the input has aged by the relative timelock amount. For example, a transaction with one input with an nSequence relative timelock of 30 blocks is only valid when at least 30 blocks have elapsed from the time the UTXO referenced in the input was mined. Since nSequence is a per-input field, a transaction may contain any number of timelocked inputs, all of which must have sufficiently aged for the transaction to be valid. A transaction can include both timelocked inputs (nSequence < 2^{31}) and inputs without a relative timelock (nSequence >= 2^{31}).

如果交易的输入的nSequence小于2^{31}，表示“相对时间锁”。
这个交易只有在这种情况下才有效：输入被相对时间锁数量老化之后。
例如，一个交易的一个输入的nSequence是30个区块，那么，只有当从这个输入引用的UTXO被挖矿开始，至少30个区块之后，这个交易才是有效的。
因为nSequence是每个输入的字段，一个交易可能包含很多时间锁输入，所以，只有所有的时间锁都满足条件时，这个交易才是有效的。
一个交易可以同时包含：时间锁输入（nSequence < 2^{31}）、无相对时间锁定输入（nSequence >= 2^{31}）。

The nSequence value is specified in either blocks or seconds, but in a slightly different format than we saw used in nLocktime. A type-flag is used to differentiate between values counting blocks and values counting time in seconds. The type-flag is set in the 23rd least-significant bit (i.e., value 1<<22). If the type-flag is set, then the nSequence value is interpreted as a multiple of 512 seconds. If the type-flag is not set, the nSequence value is interpreted as a number of blocks.
nSequence值以区块或秒为单位指定，但与nLocktime中使用的格式略有不同。
用类型标志区分块数和秒数。这个类型标志设置在第23位（即值1 << 22）。
如果类型标志为1，则nSequence值为512秒的倍数。
如果类型标志为0，则nSequence值为区块数。

When interpreting nSequence as a relative timelock, only the 16 least significant bits are considered. Once the flags (bits 32 and 23) are evaluated, the nSequence value is usually "masked" with a 16-bit mask (e.g., nSequence & 0x0000FFFF).
当把nSequence解释为相对时间锁时，只考虑最低16个有效位。
一旦计算了标志（位32和23），nSequence值通常用16位掩码获得（例如nSequence & 0x0000FFFF）。

BIP-68 definition of nSequence encoding (Source: BIP-68) shows the binary layout of the nSequence value, as defined by BIP-68.
下图显示了BIP-68定义的nSequence值的二进制布局。

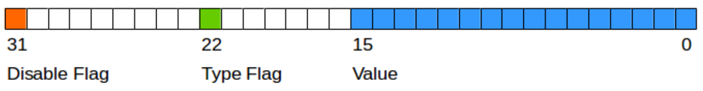

Figure 1. BIP-68 definition of nSequence encoding (Source: BIP-68)

Relative timelocks based on consensus enforcement of the nSequence value are defined in BIP-68.
The standard is defined in BIP-68, Relative lock-time using consensus-enforced sequence numbers.
BIP-68定义了相对时间锁，它基于nSequence值的共识执行。
BIP-68: Relative lock-time using consensus-enforced sequence numbers

7.5.5 使用csv的相对时间锁

Just like CLTV and nLocktime, there is a script opcode for relative timelocks that leverages the nSequence value in scripts. That opcode is CHECKSEQUENCEVERIFY, commonly referred to as CSV for short.
就像CLTV和nLocktime一样，有一个脚本操作码用于相对时间锁定，它利用脚本中的nSequence值。
该操作码是CHECKSEQUENCEVERIFY（CSV）。

The CSV opcode when evaluated in an UTXO’s redeem script allows spending only in a transaction whose input nSequence value is greater than or equal to the CSV parameter. Essentially, this restricts spending the UTXO until a certain number of blocks or seconds have elapsed relative to the time the UTXO was mined.
当在计算UTXO的赎回脚本时，只有在输入nSequence大于或等于CSV参数时，才能花费。
实质上，是限制这个UTXO被挖矿之后的相对时间以后，才能花费这个UTXO。

As with CLTV, the value in CSV must match the format in the corresponding nSequence value. If CSV is specified in terms of blocks, then so must nSequence. If CSV is specified in terms of seconds, then so must nSequence.
与CLTV一样，CSV中的值必须与相应nSequence值中的格式相匹配。
如果CSV指定的是块数，那么nSequence也应该指定块数。
如果CSV指定的是秒数，那么nSequence也应该指定秒数。

Relative timelocks with CSV are especially useful when several (chained) transactions are created and signed, but not propagated, when they’re kept "off-chain." A child transaction cannot be used until the parent transaction has been propagated, mined, and aged by the time specified in the relative timelock. One application of this use case can be seen in [state_channels] and [lightning_network].
有CSV的相对时间锁对于这种情况特别有用：

- 创建和签名了多个（串行）交易，但没有传播它们，它们被“脱链”保存。
- 在父交易已被传播、挖矿，直到消耗完相对锁定时间，才能使用子交易。

这种应用参见state_channels和lightning_network。

CSV is defined in detail in BIP-112, CHECKSEQUENCEVERIFY.
CSV 细节参见 BIP-112：CHECKSEQUENCEVERIFY

ddk总结：

绝对时间锁	nLocktime	交易级，发起节点限制交易在指定的绝对时间之后才有效
	CLTV	输出脚本中的操作码，锁定输出在未来某个时间之后才能花费
	合并使用	nLocktime >= CLTV时，交易才有效
相对时间锁	nSequence	每个输入，发起节点限制自己在指定的相对时间之后才有效
	CSV	输出脚本中的操作码，锁定输出在为了某个相对时间后才能花费
	合并使用	nSequence >= CSV时，交易才有效

7.5.6 Median-Time-Past

As part of the activation of relative timelocks, there was also a change in the way "time" is calculated for timelocks (both absolute and relative). In bitcoin there is a subtle, but very significant, difference between wall time and consensus time. Bitcoin is a decentralized network, which means that each participant has his or her own perspective of time. Events on the network do not occur instantaneously everywhere. Network latency must be factored into the perspective of each node. Eventually everything is synchronized to create a common ledger. Bitcoin reaches consensus every 10 minutes about the state of the ledger as it existed in the *past*.
作为激活相对时间锁的一部分，对于“时间”的计算方式也有一个变化，包括相对时间和绝对时间。
在比特币中，对于墙上时间和共识时间，有一个微妙但非常重要的区别。
比特币是一个去中心化的网络，这意味着，每个参与者都有自己的时间。
网络上的事件不会立刻发生在每个地方。从每个节点角度看，都会有网络延迟。
最终，所有的东西都要同步，以创建一个公共账目。
比特币对过去10分钟的账目的状态达成共识。

The timestamps set in block headers are set by the miners. There is a certain degree of latitude allowed by the consensus rules to account for differences in clock accuracy between decentralized nodes. However, this creates an unfortunate incentive for miners to lie about the time in a block so as to earn extra fees by including timelocked transactions that are not yet mature. See the following section for more information.
矿工们在区块头中设置时间戳。
考虑到去中心化节点之间的时钟精度的差异，共识规则允许一定的误差。
然而，这可能诱使矿工说谎，把还没有到期的时间锁定交易包含进来，从而获得额外的费用。

To remove the incentive to lie and strengthen the security of timelocks, a BIP was proposed and activated at the same time as the BIPs for relative timelocks. This is BIP-113, which defines a new consensus measurement of time called *Median-Time-Past*.
为了杜绝矿工说谎，加强时间锁的安全性，在相对时间锁的BIPs的同时，建议和激活了一个BIP。
这就是BIP-113，它定义了一个新的时间共识测量：Median-Time-Past

Median-Time-Past is calculated by taking the timestamps of the last 11 blocks and finding the median. That median time then becomes consensus time and is used for all timelock calculations. By taking the midpoint from approximately two hours in the past, the influence of any one block’s timestamp is reduced. By incorporating 11 blocks, no single miner can influence the timestamps in order to gain fees from transactions with a timelock that hasn’t yet matured.
Median-Time-Past的计算方法是：取最近11个区块的时间戳，计算它们的中位数。
这个中位时间值就称为了共识时间，并用于所有的时间锁计算。
通过取过去约两个小时的中间点，任何一个区块的时间戳的影响都减小了。
通过这个方法，没有一个矿工可以利用时间戳从尚未到期的交易中获取费用。

Median-Time-Past changes the implementation of time calculations for nLocktime, CLTV, nSequence, and CSV. The consensus time calculated by Median-Time-Past is always approximately one hour behind wall clock time. If you create timelock transactions, you should account for it when estimating the desired value to encode in nLocktime, nSequence, CLTV, and CSV.
Median-Time-Past改变了下列的时间计算：nLocktime、CLTV、nSequence、CSV
Median-Time-Past计算的共识时间总是墙上时间前大约一小时。
如果你要创建时间锁交易，当估计想要的值时，应该考虑到这个。

Median-Time-Past is specified in BIP-113.
Median-Time-Past的细节参见BIP-113.

7.5.7用时间锁防止交易费狙击

Fee-sniping is a theoretical attack scenario, where miners attempting to rewrite past blocks "snipe" higher-fee transactions from future blocks to maximize their profitability.
交易费狙击（Fee Sniping）是一个理论攻击场景：
试图重写过去区块的矿工，狙击未来区块的更高交易费的交易，从而使得利益最大化。

For example, let's say the highest block in existence is block #100,000. If instead of attempting to mine block #100,001 to extend the chain, some miners attempt to remine #100,000. These miners can choose to include any valid transaction (that hasn't been mined yet) in their candidate block #100,000. They don't have to remine the block with the same transactions. In fact, they have the incentive to select the most profitable (highest fee per kB) transactions to include in their block. They can include any transactions that were in the "old" block #100,000, as well as any transactions from the current mempool. Essentially they have the option to pull transactions from the "present" into the rewritten "past" when they re-create block #100,000.
例如，最高区块是＃100,000。
有些矿工不是努力挖取＃100,001来扩展区块链，而是试图重新挖取＃100,000。
这些矿工可以在他们的候选区块＃100,000中包含任何有效的交易（尚未被挖掘）。
他们不一定重新挖掘具有相同交易的区块。
事实上，他们有动力选择最有利可图（每kB最高费用）的交易包含在区块中。
他们可以包含处于“旧”块＃100,000中的任何交易，以及来自当前内存池的任何交易。
当他们重新创建区块＃100,000时，他们本质上可以将“现在的”交易拉回到“过去”。

Today, this attack is not very lucrative, because block reward is much higher than total fees per block. But at some point in the future, transaction fees will be the majority of the reward (or even the entirety of the reward). At that time, this scenario becomes inevitable.
今天，这种狙击并不是非常有利可图，因为区块奖励远远高于每个区块的总交易费。
但在未来某个时候，交易费将是奖励的大部分（甚至是全部），那时候这种情况就不可避免了。

To prevent "fee sniping," when Bitcoin Core creates transactions, it uses nLocktime to limit them to the "next block," by default. In our scenario, Bitcoin Core would set nLocktime to 100,001 on any transaction it created. Under normal circumstances, this nLocktime has no effect—the transactions could only be included in block #100,001 anyway; it's the next block.
为了防止“交易费狙击”，当Bitcoin Core创建交易时，默认情况下，它使用nLocktime将它们限制为“下一个区块”。
在我们的场景中，Bitcoin Core将在任何创建的交易上将nLocktime设置为100,001。
在正常情况下，这个nLocktime没有任何效果，交易只能包含在＃100,001块中，这是下一个区块。

But under a blockchain fork attack, the miners would not be able to pull high-fee transactions from the mempool, because all those transactions would be timelocked to block #100,001. They can only remine #100,000 with whatever transactions were valid at that time, essentially gaining no new fees.
但是在区块链分叉攻击的情况下，由于所有交易都将被时间锁阻止在＃100,001，所以矿工们无法从内存池中提取高交易费的交易。他们只能在当时有效的任何交易中重新挖矿#100,000，这实质上不会获得新的交易费。

To achieve this, Bitcoin Core sets the nLocktime on all new transactions to <current block # + 1> and sets the nSequence on all the inputs to 0xFFFFFFFE to enable nLocktime.
为了实现这个，Bitcoin Core将所有新交易的nLocktime设置为 <当前区块号 + 1>，
并将所有输入上的nSequence设置为0xFFFFFFFE，以启用nLocktime。

ddk问题：本节是什么意思？

7.6有流控制的脚本（条件语句）

One of the more powerful features of Bitcoin Script is flow control, also known as conditional clauses. You are probably familiar with flow control in various programming languages that use the construct IF...THEN...ELSE. Bitcoin conditional clauses look a bit different, but are essentially the same construct.
比特币脚本的一个更强大的功能是流控制，也称条件语句。
你可能熟悉各种编程语言中的流控制`IF...THEN...ELSE`。
比特币条件语句看起来有点不同，但本质上是相同的结构。

At a basic level, bitcoin conditional opcodes allow us to construct a redeem script that has two ways of being unlocked, depending on a TRUE/FALSE outcome of evaluating a logical condition. For example, if x is TRUE, the redeem script is A and the ELSE redeem script is B.
在基本层面上，比特币条件操作码允许构建一个具有两种解锁方式的赎回脚本，这取决于逻辑条件的计算结果是`TRUE`或`FALSE`。例如，如果为`TRUE`，则赎回脚本为`A`，否则赎回脚本为`B`。

Additionally, bitcoin conditional expressions can be "nested" indefinitely, meaning that a conditional clause can contain another within it, which contains another, etc. Bitcoin Script flow control can be used to construct very complex scripts with hundreds or even thousands of possible execution paths. There is no limit to nesting, but consensus rules impose a limit on the maximum size, in bytes, of a script.
此外，比特币条件表达式可以“嵌套”，即条件语句中可以包含条件语句。
比特币脚本流控制可用于构造非常复杂的脚本，具有很多个可能的执行路径。
嵌套没有限制，但共识规则对脚本的最大字节数有限制。

Bitcoin implements flow control using the IF, ELSE, ENDIF, and NOTIF opcodes. Additionally, conditional expressions can contain boolean operators such as BOOLAND, BOOLOR, and NOT.
比特币实现流控制的操作码是：`IF`，`ELSE`，`ENDIF`，`NOTIF`。
此外，条件表达式可以包含布尔运算符，如`BOOLAND`、`BOOLOR`、`NOT`。

At first glance, you may find the bitcoin’s flow control scripts confusing. That is because Bitcoin Script is a stack language. The same way that 1 {plus} 1 looks "backward" when expressed as 1 1 ADD, flow control clauses in bitcoin also look "backward."
乍看之下，比特币的流控制脚本令人不解，这是因为比特币脚本是一种堆栈语言。
1+1被表示为`1 1 ADD`，流控语句也是放在后面。

In most traditional (procedural) programming languages, flow control looks like this:
在多数传统编程语言中，流控制如下所示：

Pseudocode of flow control in most programming languages
多数编程语言中的流控制的伪码

```
if (condition):
  code to run when condition is true
else:
  code to run when condition is false
code to run in either case
```

In a stack-based language like Bitcoin Script, the logical condition comes before the IF, which makes it look "backward," like this:
在堆栈语言中，逻辑条件出现在`IF`之前，这使得它看起来像放在后面。

Bitcoin Script flow control
比特币脚本流控制

```
condition
IF
  code to run when condition is true
ELSE
  code to run when condition is false
ENDIF
code to run in either case
```

When reading Bitcoin Script, remember that the condition being evaluated comes *before* the IF opcode.
阅读比特币脚本时，记住：计算的条件是在IF操作码之前。

7.6.1有VERIFY操作码的条件语句

Another form of conditional in Bitcoin Script is any opcode that ends in VERIFY. The VERIFY suffix means that if the condition evaluated is not TRUE, execution of the script terminates immediately and the transaction is deemed invalid.
比特币脚本中的另一种形式的条件是：以VERIFY结尾的操作码。
VERIFY后缀表示：如果计算条件不为TRUE，脚本将终止执行，并且该交易为无效。

Unlike an IF clause, which offers alternative execution paths, the VERIFY suffix acts as a *guard clause*, continuing only if a precondition is met.
与提供其它执行路径的IF语句不同，VERIFY后缀充当保护语句，只有在满足条件的情况下才会继续。

For example, the following script requires Bob’s signature and a pre-image (secret) that produces a specific hash. Both conditions must be satisfied to unlock:
例如，以下脚本需要Bob的“签名”和“产生特定哈希的一个秘密”。
解锁时必须满足这两个条件：

A redeem script with an EQUALVERIFY guard clause.
1)有EQUALVERIFY保护语句的赎回脚本。

```
HASH160 <expected hash> EQUALVERIFY <Bob's Pubkey> CHECKSIG
```

To redeem this, Bob must construct an unlocking script that presents a valid pre-image and a signature:
为了兑现这个，Bob必须构建一个解锁脚本，提供有效的秘密和签名：

An unlocking script to satisfy the above redeem script
2)一个解锁脚本以满足上述赎回脚本。

```
<Bob's Sig> <hash pre-image>
```

Without presenting the pre-image, Bob can’t get to the part of the script that checks for his signature.
如果没有这个秘密（pre-image），Bob无法到达检查其签名的脚本部分。

This script can be written with an IF instead:
该脚本可以用IF编写：

A redeem script with an IF guard clause

```
HASH160 <expected hash> EQUAL
IF
   <Bob's Pubkey> CHECKSIG
ENDIF
```

Bob’s unlocking script is identical:
Bob的解锁脚本是一样的：

An unlocking script to satisfy the above redeem script

满足上面赎回脚本的解锁脚本
```
<Bob's Sig> <hash pre-image>
```

The script with IF does the same thing as using an opcode with a VERIFY suffix; they both operate as guard clauses. However, the VERIFY construction is more efficient, using two fewer opcodes.
使用IF的脚本与使用具有VERIFY后缀的操作码相同，他们都作为保护语句。
然而，VERIFY的构造更有效率，使用较少的操作码。

So, when do we use VERIFY and when do we use IF? If all we are trying to do is to attach a precondition (guard clause), then VERIFY is better. If, however, we want to have more than one execution path (flow control), then we need an IF...ELSE flow control clause.
那么，我们什么时候使用VERIFY？什么时候使用IF？
如果我们想要做的是附加一个前提条件（保护语句），那么VERIFY更好。
如果我们想要有多个执行路径（流控制），那么就需要一个IF...ELSE流控语句。

Tip：An opcode such as EQUAL will push the result (TRUE/FALSE) onto the stack, leaving it there for evaluation by subsequent opcodes. In contrast, the opcode EQUALVERIFY suffix does not leave anything on the stack. Opcodes that end in VERIFY do not leave the result on the stack.
提示：诸如EQUAL之类的操作码会将结果（TRUE/FALSE）推送到堆栈上，留下它用于后续操作码的计算。相比之下，操作码EQUALVERIFY后缀不会在堆栈上留下任何东西。以VERIFY中结束的操作码不会将结果留在堆栈上。

7.6.2在脚本中使用流控制

A very common use for flow control in Bitcoin Script is to construct a redeem script that offers multiple execution paths, each a different way of redeeming the UTXO.
比特币脚本中流控制的一个常见的用途是：
构建一个赎回脚本，它提供多个执行路径，每个路径都有一种不同的赎回UTXO的方式。

Let’s look at a simple example, where we have two signers, Alice and Bob, and either one is able to redeem. With multisig, this would be expressed as a 1-of-2 multisig script. For the sake of demonstration, we will do the same thing with an IF clause:
我们来看一个简单的例子：有两个签名人（Alice和Bob），两人中任何一个都可以兑换。
使用多签名，这将被表示为1-2 多签名脚本。

我们使用IF子句做同样的事情：

```
IF
   <Alice's Pubkey> CHECKSIG
ELSE
   <Bob's Pubkey> CHECKSIG
ENDIF
```

Looking at this redeem script, you may be wondering: "Where is the condition? There is nothing preceding the IF clause!"
看这个赎回脚本，你可能会想：条件在哪里？IF子句之前什么也没有！

The condition is not part of the redeem script. Instead, the condition will be offered in the unlocking script, allowing Alice and Bob to "choose" which execution path they want.
这个条件不是赎回脚本的一部分，解锁脚本将提供该条件，允许Alice和Bob选择他们想要的执行路径。

Alice redeems this with the unlocking script:
Alice使用如下解锁脚本
```
<Alice's Sig> 1
```

The 1 at the end serves as the condition (TRUE) that will make the IF clause execute the first redemption path for which Alice has a signature.
最后的1作为条件（TRUE），将使IF子句执行有Alice签名的第一个赎回路径。

For Bob to redeem this, he would have to choose the second execution path by giving a FALSE value to the IF clause:
Bob为了赎回这个，他必须通过给IF子句赋一个FALSE值来选择第二个执行路径。

```
<Bob's Sig> 0
```

Bob’s unlocking script puts a 0 on the stack, causing the IF clause to execute the second (ELSE) script, which requires Bob’s signature.
Bob的解锁脚本在栈中放置一个0，导致IF子句执行第二个（ELSE）脚本，这需要Bob的签名。

Since IF clauses can be nested, we can create a "maze" of execution paths. The unlocking script can provide a "map" selecting which execution path is actually executed:
由于可以嵌套IF子句，所以我们可以创建复杂的执行路径。
解锁脚本可以提供一个选择执行路径实际执行的“地图”：

```
IF
  script A
ELSE
  IF
    script B
  ELSE
    script C
  ENDIF
ENDIF
```

In this scenario, there are three execution paths (script A, script B, and script C). The unlocking script provides a path in the form of a sequence of TRUE or FALSE values. To select path script B, for example, the unlocking script must end in 1 0 (TRUE, FALSE). These values will be pushed onto the stack, so that the second value (FALSE) ends up at the top of the stack. The outer IF clause pops the FALSE value and executes the first ELSE clause. Then the TRUE value moves to the top of the stack and is evaluated by the inner (nested) IF, selecting the B execution path.
在这种情况下，有三个执行路径（脚本A，脚本B，脚本C）。
解锁脚本以TRUE或FALSE值的形式提供路径。
要选择路径脚本B，解锁脚本必须以1 0（TRUE，FALSE）结束。
这些值将被压入到栈，以便第二个值（FALSE）在栈顶。
外部IF子句弹出FALSE值并执行第一个ELSE子句。
然后，TRUE值移动到栈顶，并通过内部（嵌套）IF来计算，选择B执行路径。

Using this construct, we can build redeem scripts with tens or hundreds of execution paths, each offering a different way to redeem the UTXO. To spend, we construct an unlocking script that navigates the execution path by putting the appropriate TRUE and FALSE values on the stack at each flow control point.
使用这个结构，可以构建有很多执行路径的赎回脚本，每个路径提供了一种不同的方式来赎回UTXO。
如果要花费，我们就构建一个解锁脚本，通过在每个流控制点的栈上放置相应的TRUE和FALSE值来到达执行路径。

7.7复杂的脚本例子

In this section we combine many of the concepts from this chapter into a single example. Our example uses the story of Mohammed, the company owner in Dubai who is operating an import/export business.
在本节中，我们将本章中的许多概念合并成一个例子。
这个例子是Mohammed公司的故事，他们正在经营进出口业务。

In this example, Mohammed wishes to construct a company capital account with flexible rules. The scheme he creates requires different levels of authorization depending on timelocks. The participants in the multisig scheme are Mohammed, his two partners Saeed and Zaira, and their company lawyer Abdul. The three partners make decisions based on a majority rule, so two of the three must agree. However, in the case of a problem with their keys, they want their lawyer to be able to recover the funds with one of the three partner signatures. Finally, if all partners are unavailable or incapacitated for a while, they want the lawyer to be able to manage the account directly.
在这个例子中，Mohammed希望用灵活的规则建立公司的资金账户。
他创建的方案需要不同级别的授权，授权依赖于时间锁。
多签名方案的参与者是Mohammed、合作伙伴Saeed和Zaira、公司律师Abdul。
三个合作伙伴根据多数原则作出决定，因此三者中必须有两个人同意。
然而，如果他们的密钥有问题，他们希望律师能够用三个合作伙伴签名之一收回资金。
最后，如果所有的合作伙伴一段时间都不可用或无行为能力，他们希望律师能够直接管理该帐户。

Here's the redeem script that Mohammed designs to achieve this (line number prefix as XX):
Variable Multi-Signature with Timelock
这是Mohammed设计的脚本：（有时间锁的可变多签名）

```
01  IF
02      IF
03          2
04      ELSE
05         <30 days> CHECKSEQUENCEVERIFY DROP
06         <Abdul the Lawyer's Pubkey> CHECKSIGVERIFY
07         1
08      ENDIF
09      <Mohammed's Pubkey> <Saeed's Pubkey> <Zaira's Pubkey> 3
CHECKMULTISIG
10  ELSE
11      <90 days> CHECKSEQUENCEVERIFY DROP
12      <Abdul the Lawyer's Pubkey> CHECKSIG
13  ENDIF
```

Mohammed's script implements three execution paths using nested IF...ELSE flow control clauses.
Mohammed的脚本使用嵌套的IF...ELSE流控语句，实现了三个执行路径。

In the first execution path, this script operates as a simple 2-of-3 multisig with the three partners. This execution path consists of lines 3 and 9. Line 3 sets the quorum of the multisig to 2 (2-of-3). This execution path can be selected by putting TRUE TRUE at the end of the unlocking script:
第1个执行路径：
该脚本作为三个合作伙伴的2-3多签名操作。
该执行路径由第3行和第9行组成，第3行将多重签名的定额设置为2（2-3）。

Unlocking script for the first execution path (2-of-3 multisig)
该执行路径可以通过在解锁脚本的末尾设置TRUE TRUE来选择：

```
0 <Mohammed's Sig> <Zaira's Sig> TRUE TRUE
```

`Tip`: The 0 at the beginning of this unlocking script is because of a bug in CHECKMULTISIG that pops an extra value from the stack. The extra value is disregarded by the CHECKMULTISIG, but it must be present or the script fails. Pushing 0 (customarily) is a workaround to the bug, as described in [A bug in CHECKMULTISIG execution](#).
提示：此解锁脚本开头的0是因为CHECKMULTISIG中的bug从栈中弹出一个额外的值。
额外的值被CHECKMULTISIG忽略，否则脚本签名将失败。压入0（通常）是解决这个bug的方法。

The second execution path can only be used after 30 days have elapsed from the creation of the UTXO. At that time, it requires the signature of Abdul the lawyer and one of the three partners (a 1-of-3 multisig). This is achieved by line 7, which sets the quorum for the multisig to 1. To select this execution path, the unlocking script would end in FALSE TRUE:
第2个执行路径：
只能在这个UTXO创建30天后才能使用。
那时候，需要Abdul（律师）和其中一个合作伙伴来签名。
这是通过第7行实现的，该行将多选的法定人数设置为1。

Unlocking script for the second execution path (Lawyer + 1-of-3)
要选择这个执行路径，解锁脚本将以FALSE TRUE结束：
```
0 <Saeed's Sig> <Abdul's Sig> FALSE TRUE
```

`Tip`: Why FALSE TRUE? Isn't that backward? Because the two values are pushed on to the stack, with FALSE pushed first, then TRUE pushed second. TRUE is therefore popped *first* by the first IF opcode.
提示：为什么先FALSE后TRUE？ 反了吗？
这是因为这两个值被压入栈，所以先压入FALSE，然后压入TRUE。
因此，第一个IF操作码首先弹出的是TRUE。

Finally, the third execution path allows Abdul the lawyer to spend the funds alone, but only after 90 days. To select this execution path, the unlocking script has to end in FALSE:
第3个执行路径：
允许律师单独花费资金，但只能在90天之后。
要选择这个执行路径，解锁脚本必须以FALSE结束：

Unlocking script for the third execution path (Lawyer only)
第3个执行路径的解锁脚本（只有律师）
```
<Abdul's Sig> FALSE
```

Try running the script on paper to see how it behaves on the stack.
在纸上执行这个脚本，查看它在栈上的行为。

A few more things to consider when reading this example. See if you can find the answers:
- Why can't the lawyer redeem the third execution path at any time by selecting it with FALSE on the unlocking script?
- How many execution paths can be used 5, 35, and 105 days, respectively, after the UTXO is mined?
- Are the funds lost if the lawyer loses his key? Does your answer change if 91 days have elapsed?
- How do the partners "reset" the clock every 29 or 89 days to prevent the lawyer from accessing the funds?
- Why do some CHECKSIG opcodes in this script have the VERIFY suffix while others don't?

思考题：
- 为什么律师不能随时通过在解锁脚本中选择FALSE来执行第三个执行路径？
  答：因为设置了锁定时间，只能在90天后才能花费。

- 在这个UTXO被挖矿后，在5、35、105天分别有多少个执行路径可以使用？
  答：5天（1条路径），35天（2条路径），105天（3条路径）

- 如果律师丢失了他的钥匙，资金是否丢失？如果91天过去了，你的答案是否会改变？
  答：可以通过2-3使用资金；但91天之后，如果律师也没有钥匙，则会失去资金。

- 合作伙伴如何每隔29天或89天"重置"一次时间，以防律师获得资金？
  答：They 'reset' the clock by moving the funds to a new output before the period ends.
  To 'renew' the time, they just need to spend the output by creating a new transaction with exactly the same output script, every time they want to renew it. Because the times use CSV, they are relative timelocks not absolute, so they don't need to be modified for each new transaction. Once a new transaction has been created which spends the output of the old one, that old output cannot be spent a second time so the lawyer must wait for the 30/90 days to elapse on the new transaction, and so on.

- 为什么这个脚本中的一些CHECKSIG操作码有VERIFY后缀，而其它的没有？
  答：因为在第2个路径中，CHECKSIG之后如果为FALSE应该停止执行，所以带VERIFY后缀。而第三个路径，CHECKSIG之后已经结束，所以不用带VERIFY后缀。

7.8隔离见证（segwit）

Segregated Witness

Segregated Witness (segwit) is an upgrade to the bitcoin consensus rules and network protocol, proposed and implemented as a BIP-9 soft-fork that was activated on bitcoin's mainnet on August 1st, 2017.
隔离见证（segwit）是对比"特币共识规则和网络协议"的一个升级，
作为BIP-9软分叉被建议和实施，2017-8-1在比特币主网上被激活。

In cryptography, the term "witness" is used to describe a solution to a cryptographic puzzle. In bitcoin terms, the witness satisfies a cryptographic condition placed on a unspent transaction output (UTXO).
在密码学中，"见证（witness）"用于描述加密难题的一个解。
在比特币中，"见证"满足了放置在UTXO上的一个加密条件。

In the context of bitcoin, a digital signature is *one type of witness*, but a witness is more broadly any solution that can satisfy the conditions imposed on an UTXO and unlock that UTXO for spending. The term "witness" is a more general term for an "unlocking script" or "scriptSig."
在比特币中，数字签名就是一种"见证"，
但"见证"是更为宽泛的任意解，它能满足UTXO上条件，解锁这个UTXO来花费。
"见证"是一个更通用的术语，表示"解锁脚本"或"scriptSig"。

Before segwit's introduction, every input in a transaction was followed by the witness data that unlocked it. The witness data was embedded in the transaction as part of each input. The term *segregated witness*, or *segwit* for short, simply means separating the signature or unlocking script of a specific output. Think "separate scriptSig," or "separate signature" in the simplest form.
在引入segwit之前，交易的每个输入都要有的见证数据。
见证数据作为每个输入的一部分被放在交易中。
segwit简单理解就是：把输出的签名或解锁脚本分离出来。
即：分离的解锁脚本（separate scriptSig），或分离的签名（separate signature）

Segregated Witness therefore is an architectural change to bitcoin that aims to move the witness data from the scriptSig (unlocking script) field of a transaction into a separate *witness* data structure that accompanies a transaction. Clients may request transaction data with or without the accompanying witness data.
因此，segwit是对比特币的结构性调整，目的是：把"见证数据"从交易的scriptSig字段移到一个分离的见证数据结构中，这个结构伴随着一个交易。
客户端可以请求"有或没有见证数据"的交易数据。

In this section we will look at some of the benefits of Segregated Witness, describe the mechanism used to deploy and implement this architecture change, and demonstrate the use of Segregated Witness in transactions and addresses.
在本节中，我们来看看：

- 优点：隔离见证的一些优点
- 机制：部署和实现这个结构性调整的机制
- 使用：隔离见证在交易和地址中的使用

Segregated Witness is defined by the following BIPs:
隔离见证是由下列BIP定义：

BIP-141	The main definition of Segregated Witness.	segwit的主要定义
BIP-143	Transaction Signature Verification for Version 0 Witness Program	用于版本0见证程序的交易签名验证
BIP-144	Peer Services—New network messages and serialization formats	Peer服务—新的网络消息和序列化格式
BIP-145	getblocktemplate Updates for Segregated Witness (for mining)	对segwit的getblocktemplate更新（用于挖矿）
BIP-173	Base32 address format for native v0-16 witness outputs	用于原始v0-16见证输出的Base32地址格式

7.8.1为什么需要隔离见证?

Segregated Witness is an architectural change that has several effects on the scalability, security, economic incentives, and performance of bitcoin:
setwit是一个结构性调整，它对比特币有下列影响：可扩展性、安全性、经济激励、性能。

（1）Transaction Malleability 交易可塑性
By moving the witness outside the transaction, the transaction hash used as an identifier no longer includes the witness data. Since the witness data is the only part of the transaction that can be modified by a third party (see Transaction identifiers), removing it also removes the opportunity for transaction malleability attacks. With Segregated Witness, transaction hashes become immutable by anyone other than the creator of the transaction, which greatly improves the implementation of many other protocols that rely on advanced bitcoin transaction construction, such as payment channels, chained transactions, and lightning networks.
将见证从交易中移出后，用作标识的交易哈希不再包含见证数据。
因为见证数据是交易中唯一可被第三方修改的部分，把它移除也消除了交易可塑性攻击的机会。
通过segwit，任何人（除了交易的创建者）都不能更改交易哈希，这极大改善了其它许多依赖于高级比特币交易架构的协议的实现，例如：支付通道、跨连交易（chained transactions）、闪电网络。

（2）Script Versioning 脚本版本
With the introduction of Segregated Witness scripts, every locking script is preceded by a *script version* number, similar to how transactions and blocks have version numbers. The addition of a script version number allows the scripting language to be upgraded in a backward-compatible way (i.e., using soft fork upgrades) to introduce new script operands, syntax, or semantics. The ability to upgrade the scripting language in a nondisruptive way will greatly accelerate the rate of innovation in bitcoin.
通过引入segwit脚本，每个锁定脚本前面都有一个脚本版本号，类似于交易和区块的版本号。
这样，能以向后兼容的方式（即软分叉升级）来升级脚本语言，以引入新的脚本操作码、语法或语义。
这个能力大大加快比特币的创新速度。

（3）Network and Storage Scaling 网络和存储伸缩性
The witness data is often a big contributor to the total size of a transaction. More complex scripts such as those used for multisig or payment channels are very large. In some cases these scripts account for the majority (more than 75%) of the data in a transaction. By moving the witness data outside the transaction, Segregated Witness improves bitcoin's scalability. Nodes can prune the witness data after validating the signatures, or ignore it altogether when doing simplified payment verification. The witness data doesn't need to be transmitted to all nodes and does not need to be stored on disk by all nodes.
在一个交易中，见证数据常常占据主要大小。
更复杂的脚本会非常大，例如用于多签名或支付通道的脚本。
在某些情况下，这些脚本占据交易数据的大部分（超过75%）。
通过把见证数据移出交易，segwit改善了比特币的伸缩性。
节点可以在验证了签名之后删除见证数据，或在进行简单支付验证时完全忽略它。
不需要把见证数据传输给所有节点，不需要所有节点在磁盘上存储它。

（4）Signature Verification Optimization　签名验证优化
Segregated Witness upgrades the signature functions (CHECKSIG, CHECKMULTISIG, etc.) to reduce the algorithm's computational complexity. Before segwit, the algorithm used to produce a signature required a number of hash operations that was proportional to the size of the transaction. Data-hashing computations increased in $O(n^2)$ with respect to the number of signature operations, introducing a substantial computational burden on all nodes verifying the signature. With segwit, the algorithm is changed to reduce the complexity to $O(n)$.
segwit升级了签名功能（CHECKSIG、CHECKMULTISIG、等），以减少算法的计算复杂性。
在segwit之前，用于生成一个签名的算法需要一些哈希操作，这些操作与交易的大小成比例。

对于签名操作的数量，数据哈希计算以O(n^2)增加，这给验证这个签名的所有节点引入了相当大的计算负担。
有了segwit，这个算法把复杂性减少到O(n)。

ddk：这段什么意思？

（5）**Offline Signing Improvement 离线签名改进**
Segregated Witness signatures incorporate the value (amount) referenced by each input in the hash that is signed. Previously, an offline signing device, such as a hardware wallet, would have to verify the amount of each input before signing a transaction. This was usually accomplished by streaming a large amount of data about the previous transactions referenced as inputs. Since the amount is now part of the commitment hash that is signed, an offline device does not need the previous transactions. If the amounts do not match (are misrepresented by a compromised online system), the signature will be invalid.
segwit签名包含了被签名的哈希中的每个输入所引用的值（金额）。
以前，离线签名设备（如硬件钱包）在对一个交易签名之前，必须验证每个输入的金额。
这通常是通过连接大量作为输入引用的先前交易的数据来完成。
因为金额现在是被签名的承诺哈希的一部分，所以，离线设备不需要以前的交易。
如果金额不匹配（被一个受损在线系统歪曲），这个签名就会无效。

ddk问题：这段什么意思？7.8.1中讲的这些优点是什么意思?

7.8.2隔离见证工作原理

At first glance, Segregated Witness appears to be a change to how transactions are constructed and therefore a transaction-level feature, but it is not. Rather, Segregated Witness is a change to how individual UTXO are spent and therefore is a per-output feature.
初看时，segwit似乎是更改了交易的构建方法，以为它是交易级别的特性，但并非如此。
实际上，segwit更改了如何花费一个UTXO，因此它是每个输出的特性。

A transaction can spend Segregated Witness outputs or traditional (inline-witness) outputs or both. Therefore, it does not make much sense to refer to a transaction as a “Segregated Witness transaction.” Rather we should refer to specific transaction outputs as “Segregated Witness outputs."
一个交易可以花费“segwit输出”或“传统（内联见证）输出”，或者同时花费两者。
因此，把一个交易称作“segwit交易”是没有意义的。
我们应该称某个交易输出是“segwit输出”。

When a transaction spends an UTXO, it must provide a witness. In a traditional UTXO, the locking script requires that witness data be provided *inline* in the input part of the transaction that spends the UTXO. A Segregated Witness UTXO, however, specifies a locking script that can be satisfied with witness data outside of the input (segregated).
当一个交易花费一个UTXO时，它必须提供一个见证。
在传统UTXO中，锁定脚本要求在花费这个UTXO的交易的输入中提供见证数据。
但是，segwit UTXO指定了一个锁定脚本，使用输入之外的见证数据来满足这个锁定脚本。

7.8.3软分叉(向后兼容)

Segregated Witness is a significant change to the way outputs and transactions are architected. Such a change would normally require a simultaneous change in every bitcoin node and wallet to change the consensus rules—what is known as a hard fork. Instead,

segregated witness is introduced with a much less disruptive change, which is backward compatible, known as a soft fork. This type of upgrade allows nonupgraded software to ignore the changes and continue to operate without any disruption.
segwit对输出和交易的结构方式做了重大改变。
这种改变通常需要同时改变每个比特币节点和钱包，以改变共识规则，即“硬分叉”。
但是，segwit的引入却是通过一个更少破坏性的改变，这种改变能向后兼容，即“软分叉”。
这种类型的升级允许未升级的软件忽略那些改变，继续操作而不会造成任何破坏。

Segregated Witness outputs are constructed so that older systems that are not segwit-aware can still validate them. To an old wallet or node, a Segregated Witness output looks like an output that *anyone can spend*. Such outputs can be spent with an empty signature, therefore the fact that there is no signature inside the transaction (it is segregated) does not invalidate the transaction. Newer wallets and mining nodes, however, see the Segregated Witness output and expect to find a valid witness for it in the transaction’s witness data.
segwit输出被构造，使得不认识segwit的老系统仍然能够验证它们。
对于老的钱包或节点来说，一个segwit输出看起来就像任何人都能花费的一个输出。
这种输出可以用一个空签名来花费，因此，交易中没有签名（签名被隔离）并不会导致该交易无效。
但是，新的钱包和挖矿节点能认出segwit输出，并希望在交易的见证数据中为它找到一个有效的见证。

7.8.4隔离见证输出和交易的示例

Let’s look at some of our example transactions and see how they would change with Segregated Witness. We’ll first look at how a Pay-to-Public-Key-Hash (P2PKH) payment is transformed with the Segregated Witness program. Then, we’ll look at the Segregated Witness equivalent for Pay-to-Script-Hash (P2SH) scripts. Finally, we’ll look at how both of the preceding Segregated Witness programs can be embedded inside a P2SH script.
我们来看一些交易例子，看看它们如何随segwit而改变。
首先，看看“隔离见证程序”如何转换一个P2PKH支付。
然后，看看与P2SH脚本等价的segwit。
最后，看看在一个P2SH脚本中，如何嵌入上面两个隔离见证程序。

7.8.4.1 P2WPKH

P2PKH	Pay-to-Public-Key-Hash
P2WPKH	Pay-to-Witness-Public-Key-Hash

In [cup of coffee], Alice created a transaction to pay Bob for a cup of coffee. That transaction created a P2PKH output with a value of 0.015 BTC that was spendable by Bob. The output’s script looks like this:
在前面的例子中，Alice在Bob的咖啡店买了一杯咖啡，做了付款交易。
这个交易创建了一个P2PKH输出，给Bob支付了0.015比特币。
这个输出脚本是这样：

Example P2PKH output script
示例P2PKH输出脚本

```
DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 EQUALVERIFY CHECKSIG
```

With Segregated Witness, Alice would create a Pay-to-Witness-Public-Key-Hash (P2WPKH) script, which looks like this:
如果使用segwit，Alice会创建一个P2WPKH脚本，是这样：

Example P2WPKH output script
示例P2WPKH输出脚本

```
0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
```

As you can see, a Segregated Witness output’s locking script is much simpler than a traditional output. It consists of two values that are pushed on to the script evaluation stack. To an old (nonsegwit-aware) bitcoin client, the two pushes would look like an output that anyone can spend and does not require a signature (or rather, can be spent with an empty signature).
正如你所见，segwit输出的锁定脚本比传统输出简单多了。
它包含两个值，这两个值被压入脚本计算堆栈中。
对老的比特币客户端来说，这两个压入值看起来就像任何人都能花费的输出，并且不需要签名（或者能用一个空签名来花费）。

To a newer, segwit-aware client, the first number (0) is interpreted as a version number (the *witness version*) and the second part (20 bytes) is the equivalent of a locking script known as a *witness program*. The 20-byte witness program is simply the hash of the public key, as in a P2PKH script
对新的比特币客户端来说：

- 数字0表示版本号（见证版本）
- 第2部分的20字节相当于一个锁定脚本，称为“见证程序”(witness program）。
  这20字节的见证程序只是公钥的哈希，就像 P2PKH 脚本中的一样。

Now, let’s look at the corresponding transaction that Bob uses to spend this output. For the original script (nonsegwit), Bob’s transaction would have to include a signature within the transaction input:
现在，我们来看看Bob用来花费这个输出的交易。
对于非segwit脚本，Bob的交易必须在交易输入中包含一个签名：

Decoded transaction showing a P2PKH output being spent with a signature
下面是解码的交易，显示了是用一个签名来花费一个P2PKH 输出。

```
[...]
“Vin” : [
          "txid"     :
"0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",
          "vout"     : 0,
          "scriptSig": “<Bob’s scriptSig>”,
]
[...]
```

However, to spend the Segregated Witness output, the transaction has no signature on that input. Instead, Bob’s transaction has an empty scriptSig and includes a Segregated Witness, outside the transaction itself:
但是，为了花费segwit输出，这个交易在输入中没有签名。
而是，Bob的交易有一个空的scriptSig ，并且在这个交易之外有一个segwit。

Decoded transaction showing a P2WPKH output being spent with separate witness data
下面是解码的交易，显示了用segwit数据花费一个P2WPKH 输出。

```
[...]
“Vin” : [
          "txid"     :
"0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",
          "vout"     : 0,
          "scriptSig": “”,
]
[...]
“witness”: “<Bob’s witness data>”
[...]
```

7.8.4.2 Wallet construction of P2WPKH

It is extremely important to note that P2WPKH should only be created by the payee (recipient) and not converted by the sender from a known public key, P2PKH script, or address. The sender has no way of knowing if the recipient’s wallet has the ability to construct segwit transactions and spend P2WPKH outputs.
重要的是，P2WPKH 只应由收款人创建，而不是由付款人用一个已知的公钥、P2PKH脚本或地址来转换。

付款人没法知道：收款人的钱包是否有能力构建segwit交易和花费P2WPKH 输出。

Additionally, P2WPKH outputs must be constructed from the hash of a *compressed* public key. Uncompressed public keys are nonstandard in segwit and may be explicitly disabled by a future soft fork. If the hash used in the P2WPKH came from an uncompressed public key, it may be unspendable and you may lose funds. P2WPKH outputs should be created by the payee’s wallet by deriving a compressed public key from their private key.
另外，P2WPKH 输出必须用压缩公钥的哈希来构建。
未压缩公钥在segwit中不是标准的，可能会被将来的软分叉中明确禁用。
如果 P2WPKH 中使用的哈希来自未压缩公钥，那么可能不能花费它，你有可能丢失资金。
P2WPKH 输出应该由收款人的钱包来创建：从他的私钥中导出一个压缩公钥。

Warning: P2WPKH should be constructed by the payee (recipient) by converting a compressed public key to a P2WPKH hash. You should never transform a P2PKH script, bitcoin address, or uncompressed public key to a P2WPKH witness script.
警告： P2WPKH 应该由收款人构建（把一个压缩公钥转换为一个P2WPKH哈希）。
你绝不应把一个P2PKH脚本、比特币地址或未压缩公钥转换成一个P2WPKH见证脚本。

7.8.4.3 P2WSH

P2SH	Pay-to-Script-Hash
P2WSH	Pay-to-Witness-Script-Hash

The second type of witness program corresponds to a Pay-to-Script-Hash (P2SH) script. We saw this type of script in Pay-to-Script-Hash (P2SH). In that example, P2SH was used by Mohammed's company to express a multisignature script. Payments to Mohammed's company were encoded with a locking script like this:
第2种类型的见证程序对应一个P2SH脚本。
在前面的例子中，Mohammed的公司使用P2SH 来表达一个多签名脚本。
用如下锁定脚本编码给这个公司的支付：

Example P2SH output script
示例P2SH输出脚本

```
HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL
```

This P2SH script references the hash of a *redeem script* that defines a 2-of-3 multisignature requirement to spend funds. To spend this output, Mohammed's company would present the redeem script (whose hash matches the script hash in the P2SH output) and the signatures necessary to satisfy that redeem script, all inside the transaction input:
这个P2SH脚本引用了一个赎回脚本的哈希，这个赎回脚本定义了一个2-3多签名。
为了花费该输出，公司要提供这个赎回脚本（其哈希与P2SH 输出中的脚本哈希相同），并且还有满足那个赎回脚本的签名，所有这些都在交易输入中。

Decoded transaction showing a P2SH output being spent
下面是解码的交易，显示了花费一个P2SH输出。

```
[...]
"Vin" : [
          "txid"    : "abcdef12345...",
          "vout"    : 0,
          "scriptSig": "<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>",
]
```

Now, let's look at how this entire example would be upgraded to segwit. If Mohammed's customers were using a segwit-compatible wallet, they would make a payment, creating a Pay-to-Witness-Script-Hash (P2WSH) output that would look like this:
现在，我们看看这个例子如何升级为segwit。
如果客户使用了一个segwit兼容的钱包，他们会创建一个P2WSH输出，看起来是这样：

Example P2WSH output script
示例P2WSH输出脚本

```
0 a9b7b38d972cabc7961dbfbcb841ad4508d133c47ba87457b4a0e8aae86dbb89
```

Again, as with the example of P2WPKH, you can see that the Segregated Witness equivalent script is a lot simpler and omits the various script operands that you see in P2SH scripts. Instead, the Segregated Witness program consists of two values pushed to the stack: a witness version (0) and the 32-byte SHA256 hash of the redeem script.
这与P2WPKH类似，segwit比普通脚本简单多了，省去了P2SH脚本中的各种操作码。
segwit程序包含两个要压入栈的值：见证版本（0）、赎回脚本的SHA256哈希（32字节）。

Tip: While P2SH uses the 20-byte RIPEMD160(SHA256(script)) hash, the P2WSH witness program uses a 32-byte SHA256(script) hash. This difference in the selection of the hashing algorithm is deliberate and is used to differentiate between the two types of witness programs (P2WPKH and P2WSH) by the length of the hash and to provide stronger security to P2WSH (128 bits of security in P2WSH versus 80 bits of security in P2SH).
提示：
- P2SH的哈希是20字节，使用RIPEMD160(SHA256(script))
- P2WSH的哈希是32字节，使用SHA256（script）

是故意这样选择的，这样，通过哈希长度能区分两种类型的见证程序（P2WPKH和P2WSH），并为P2WSH提供更强的安全性（P2WSH是128位安全性，而P2SH是 80 位安全性）。

Mohammed's company can spend outputs the P2WSH output by presenting the correct redeem script and sufficient signatures to satisfy it. Both the redeem script and the signatures would be segregated *outside* the spending transaction as part of the witness data. Within the transaction input, Mohammed's wallet would put an empty scriptSig:
Mohammed的公司可以这样花费这个P2WSH输出：提供正确的赎回脚本和足够的满足它的签名。
赎回脚本和签名都是放在这个花费交易之外，是见证数据的一部分。
在这个交易输入中，Mohammed的钱包会放置一个空的scriptSig：

Decoded transaction showing a P2WSH output being spent with separate witness data
下面是解码的交易，显示了用segwit数据花费的一个P2WSH输出。

```
[...]
"Vin" : [
          "txid"     : "abcdef12345...",
          "vout"     : 0,
          "scriptSig": "",
]
[...]
"witness": "<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>"
[...]
```

7.8.4.4 区分P2WPKH和P2WSH

In the previous two sections, we demonstrated two types of witness programs: Pay-to-Witness-Public-Key-Hash (P2WPKH) and Pay-to-Witness-Script-Hash (P2WSH). Both types of witness programs consist of a single byte version number followed by a longer hash. They look very similar, but are interpreted very differently: one is interpreted as a public key hash, which is satisfied by a signature and the other as a script hash, which is satisfied by a redeem script. The critical difference between them is the length of the hash:

• The public key hash in P2WPKH is 20 bytes	P2WPKH的公钥哈希是20字节
• The script hash in P2WSH is 32 bytes	P2WSH的脚本哈希是32字节

在前两节中，我们展示了两种见证程序：P2WPKH和P2WSH。
这两种见证程序包含：版本号（1字节），一个哈希。
它们看起来非常相似，但是对它们的解释非常不同：
- P2WPKH被解释为一个公钥哈希，用签名满足它。
- P2WSH被解释为一个脚本哈希，用一个赎回脚本来满足它。

它们之间的关键区别是哈希值的长度：
- P2WPKH的公钥哈希是20字节
- P2WSH的脚本哈希是32字节

This is the one difference that allows a wallet to differentiate between the two types of witness programs. By looking at the length of the hash, a wallet can determine what type of witness program it is, P2WPKH or P2WSH.
这个区别使钱包可以区分这两种类型的见证程序。

通过查看哈希值的长度，钱包可以确定它是哪种类型的见证程序（P2WPKH 或P2WSH）。

7.8.5升级到隔离见证

As we can see from the previous examples, upgrading to Segregated Witness is a two-step process. First, wallets must create special segwit type outputs. Then, these outputs can be spent by wallets that know how to construct Segregated Witness transactions.
正如前面的例子所看到的，升级到segwit需要经过两步过程。

- 首先，钱包必须创建特殊的segwit类型输出。
- 然后，这些输出可以被知道如何构建segwit交易的钱包花费。

In the examples, Alice’s wallet was segwit-aware and able to create special outputs with Segregated Witness scripts. Bob’s wallet is also segwit-aware and able to spend those outputs. What may not be obvious from the example is that in practice, Alice’s wallet needs to *know* that Bob uses a segwit-aware wallet and can spend these outputs. Otherwise, if Bob’s wallet is not upgraded and Alice tries to make segwit payments to Bob, Bob’s wallet will not be able to detect these payments.
在这些例子中，Alice的钱包理解segwit，并且能够用segwit脚本创建特殊输出。
Bob的钱包也理解segwit，并能够花费这些输出。
在这个例子中，可能不明显的是，在实际中，Alice的钱包需要知道Bob使用了segwit钱包，并能够花费这些输出。否则，如果Bob的钱包没有升级，而Alice试图对Bob进行segwit付款，那么Bob的钱包就不能检测到这些付款。

Tip: For P2WPKH and P2WSH payment types, both the sender and the recipient wallets need to be upgraded to be able to use segwit. Furthermore, the sender’s wallet needs to know that the recipient’s wallet is segwit-aware.
提示 对于P2WPKH和P2WSH支付类型，付款人和收款人的钱包都需要升级，才能使用segwit。
此外，付款人的钱包需要知道收款人的钱包是否理解segwit。

Segregated Witness will not be implemented simultaneously across the entire network. Rather, Segregated Witness is implemented as a backward-compatible upgrade, where *old and new clients can coexist*. Wallet developers will independently upgrade wallet software to add segwit capabilities. The P2WPKH and P2WSH payment types are used when both sender and recipient are segwit-aware. The traditional P2PKH and P2SH will continue to work for nonupgraded wallets. That leaves two important scenarios, which are addressed in the next section:
segwit不会在整个网络中同时实施。
segwit被实施为向后兼容的升级，其中老客户端和新客户端可以共存。
钱包开发人员要独立升级钱包软件，以添加segwit功能。
当付款人和收款人都理解segwit时，就使用P2WPKH和P2WSH付款类型。
传统的P2PKH和P2SH将继续为非升级的钱包工作。
这留下了两个重要的场景，下一节将讨论：

- Ability of a sender’s wallet that is not segwit-aware to make a payment to a recipient’s wallet that can process segwit transactions
  付款人钱包不能理解segwit，收款人钱包能理解segwit，
  付款人要能给收款人做支付。

- Ability of a sender’s wallet that is segwit-aware to recognize and distinguish between recipients that are segwit-aware and ones that are not, by their addresses.
  付款人钱包能理解segwit，它要能够通过地址来认出和区分两种收款人：
  能理解segwit和不能理解segwit。

7.8.5.1在P2SH中嵌入segwit

Let’s assume, for example, that Alice’s wallet is not upgraded to segwit, but Bob’s wallet is upgraded and can handle segwit transactions. Alice and Bob can use "old" non-segwit transactions. But Bob would likely want to use segwit to reduce transaction fees, taking advantage of the discount that applies to witness data.
举个例子，假设Alice的钱包没有升级到segwit，而Bob的钱包已经升级到segwit。
Alice和Bob可以使用老的非segwit交易。
但是Bob可能想使用segwit来减少交易费，利用应用于见证数据的折扣。

In this case Bob’s wallet can construct a P2SH address that contains a segwit script inside it. Alice’s wallet sees this as a "normal" P2SH address and can make payments to it without any knowledge of segwit. Bob’s wallet can then spend this payment with a segwit transaction, taking full advantage of segwit and reducing transaction fees.
在这种情况下，Bob的钱包可以构建一个P2SH地址，在其中包含一个segwit脚本。
Alice的钱包把它看做一个普通的P2SH地址，并可以向它支付。
然后，Bob的钱包可以用一个segwit交易来花费这笔钱，充分利用了setwit，并且减少了易费用。

Both forms of witness scripts, P2WPKH and P2WSH, can be embedded in a P2SH address. The first is noted as P2SH(P2WPKH) and the second is noted as P2SH(P2WSH).
两种形式的见证脚本（P2WPKH和P2WSH）都可以嵌入到一个P2SH地址中。
第一个表示为P2SH（P2WPKH），第二个表示为P2SH（P2WSH）。

7.8.5.2 P2WPKH在P2SH中

The first form of witness script we will examine is P2SH(P2WPKH). This is a Pay-to-Witness-Public-Key-Hash witness program, embedded inside a Pay-to-Script-Hash script, so that it can be used by a wallet that is not aware of segwit.
我们先看第一种形式的见证脚本：P2SH（P2WPKH）。
这是一个P2WPKH见证程序，嵌入在P2SH脚本中，所以它可以被不理解segwit的钱包使用。

Bob’s wallet constructs a P2WPKH witness program with Bob’s public key. This witness program is then hashed and the resulting hash is encoded as a P2SH script. The P2SH script is converted to a bitcoin address, one that starts with a "3," as we saw in the Pay-to-Script-Hash (P2SH) section.
Bob的钱包用Bob的公钥构造了一个P2WPKH见证程序。
然后对这个见证程序进行哈希，生成的哈希被编码为一个P2SH脚本。
这个P2SH脚本被转换为一个比特币地址，它以“3”开始，正如我们在P2SH章节看到的那样。

Bob’s wallet starts with the P2WPKH witness program we saw earlier:
Bob的钱包从我们之前看到的P2WPKH 见证程序开始：

Bob’s P2WPKH witness program　Bob的P2WPKH见证程序
```
0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
```

The P2WPKH witness program consists of the witness version and Bob’s 20-byte public key hash.
这个P2WPKH见证程序包含：见证版本、Bob的20字节公钥哈希。

Bob’s wallet then hashes the preceding witness program, first with SHA256, then with RIPEMD160, producing another 20-byte hash.
然后，Bob的钱包哈希之前的见证程序，先用SHA256，然后用RIPEMD160，产生另一个20字节的散列：

Let’s use bx on the command-line to replicate that:
我们在命令行上用bx来复制它：

HASH160 of the P2WPKH witness program
P2WPKH见证程序的HASH160

```
echo \
'0 [ab68025513c3dbd2f7b92a94e0581f5d50f654e7]'\
 | bx script-encode | bx sha256 | bx ripemd160
3e0547268b3b19288b3adef9719ec8659f4b2b0b
```

Next, the redeem script hash is converted to a bitcoin address. Let’s use bx on the command-line again:

然后，这个赎回脚本哈希被转换为一个比特币地址。
我们再次使用bx。

P2SH address
```
echo \
'3e0547268b3b19288b3adef9719ec8659f4b2b0b' \
| bx address-encode -v 5
37Lx99uaGn5avKBxiW26HjedQE3LrDCZru
```

Now, Bob can display this address for customers to pay for their coffee. Alice’s wallet can make a payment to 37Lx99uaGn5avKBxiW26HjedQE3LrDCZru, just as it would to any other bitcoin address.
现在，Bob可以把这个地址显示给顾客，让他们付钱买咖啡。
Alice的钱包可以支付给37Lx99uaGn5avKBxiW26HjedQE3LrDCZru，就像支付给其它比特币地址一样。

To pay Bob, Alice’s wallet would lock the output with a P2SH script:
为了向Bob支付，Alice钱包会用一个P2SH脚本锁定这个输出：

```
HASH160 3e0547268b3b19288b3adef9719ec8659f4b2b0b EQUAL
```

Even though Alice’s wallet has no support for segwit, the payment it creates can be spent by Bob with a segwit transaction.
即使Alice的钱包不支持segwit，它创建的支付也能被Bob用一个segwit交易来花费。

7.8.5.3 P2WSH在P2SH中

Similarly, a P2WSH witness program for a multisig script or other complicated script can be embedded inside a P2SH script and address, making it possible for any wallet to make payments that are segwit compatible.
类似的，一个P2WSH见证程序（用多签名脚本或其它复杂脚本）可以嵌入到一个P2SH脚本和地址中，使任何理解segwit钱包都可以进行支付。

As we saw in Pay-to-Witness-Script-Hash (P2WSH), Mohammed’s company is using Segregated Witness payments to multisignature scripts. To make it possible for any client to pay his company, regardless of whether their wallets are upgraded for segwit, Mohammed’s wallet can embed the P2WSH witness program inside a P2SH script.
正如我们在P2WSH中看到的，穆罕默德的公司正在使用隔离见证付快给多重签名脚本。
为了让任何客户都能给他的公司付款，无论他们的钱包是否做了segwit升级，穆罕默德的钱包都可以在一个P2SH脚本中嵌入这个P2WSH见证程序。

First, Mohammed’s wallet hashes the redeem script with SHA256 (just once). Let’s use bx to do that on the command-line:

首先，穆罕默德的钱包用SHA256对赎回脚本进行哈希（只一次）。
我们用bx来做这个：

Mohammed’s wallet creates a P2WSH witness program
穆罕默德的钱包创建了一个P2WSH见证程序

```
echo \
2 \
[04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC
5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587] \
[04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D
0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49] \
[047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4
420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965] \
[0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044
DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5] \
[043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29
A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800] \
5 CHECKMULTISIG \
| bx script-encode | bx sha256
9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73
```

Next, the hashed redeem script is turned into a P2WSH witness program:
然后，这个被哈希的赎回脚本被转换为一个P2WSH见证程序。

```
0 9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73
```

Then, the witness program itself is hashed with SHA256 and RIPEMD160, producing a new 20-byte hash, as used in traditional P2SH. Let’s use bx on the command-line to do that:
然后，用SHA256和RIPEMD160对这个见证程序本身进行散列，产生一个新的20字节哈希，就如传统P2SH中使用一样。我们用bx来做这个：

The HASH160 of the P2WSH witness program
P2WSH见证程序的HASH160

```
 echo \
'0 [9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73]'\
 | bx script-encode | bx sha256 | bx ripemd160
86762607e8fe87c0c37740cddee880988b9455b2
```

Next, the wallet constructs a P2SH bitcoin address from this hash. Again, we use bx to calculate on the command-line:
然后，这个钱包用这个哈希构建一个P2SH比特币地址。
我们还用bx来计算。

P2SH bitcoin address

```
echo \
'86762607e8fe87c0c37740cddee880988b9455b2'\
 | bx address-encode -v 5
3Dwz1MXhM6EfFoJChHCxh1jWHb8GQqRenG
```

Now, Mohammed’s clients can make payments to this address without any need to support segwit. To send a payment to Mohammed, a wallet would lock the output with the following P2SH script:
现在，穆罕默德的客户可以付款到这个地址，而不需要支持segwit。
为了发送支付给默罕默德，钱包会用下面的P2SH脚本锁定这个输出。

P2SH script used to lock payments to Mohammed’s multisig
用于锁定给默罕默德的多签名支付的P2SH脚本

```
HASH160 86762607e8fe87c0c37740cddee880988b9455b2 EQUAL
```

Mohammed’s company can then construct segwit transactions to spend these payments, taking advantage of segwit features including lower transaction fees.
然后，默罕默德的公司能够构建segwit交易来花费这些支付，充分利用segwit特性，包括更低的交易费。

7.8.5.4 segwit地址

Even after segwit activation, it will take some time until most wallets are upgraded. At first, segwit will be embedded in P2SH, as we saw in the previous section, to ease compatibility between segit-aware and unaware wallets.
即使在segwit激活之后，多数钱包升级也需要一些时间。
首先，segwit会被嵌入在P2SH中，就像我们在前一节看到的，这使得理解和不理解segwit的钱包容易兼容。

However, once wallets are broadly supporting segwit, it makes sense to encode witness scripts directly in a native address format designed for segwit, rather than embed it in P2SH.
但是，一旦钱包广泛都能够支持segwit，这样做就有意义了：直接用为segwit设计的native地址格式来编码见证脚本，而不是嵌入在P2SH中。

The native segwit address format is defined in BIP-173:
BIP-173定义了native segwit地址格式：

BIP-173 Base32 address format for native v0-16 witness outputs

BIP-173 only encodes witness (P2WPKH and P2WSH) scripts. It is not compatible with non-segwit P2PKH or P2SH scripts. BIP-173 is a checksummed Base32 encoding, as compared to the Base58 encoding of a "traditional" bitcoin address. BIP-173 addesses are also called *bech32* addresses, pronounced "beh-ch thirty two", alluding to the use of a "BCH" error detection algorithm and 32-character encoding set.
BIP-173只编码见证（P2WPKH和P2WSH）脚本。
它与非segwit的P2PKH或P2SH脚本不兼容。
BIP-173是一个校验和Base32编码，而传统的比特币地址是Base58编码。
BIP-173地址也被称为bech32地址，表示使用“BCH”错误检测算法和32字符编码集。

BIP-173 addresses use 32 lower-case-only alphanumeric character set, carefully selected to reduce errors from misreading or mistyping. By choosing a lower-case-only character set, bech32 is easier to read, speak, and 45% more efficient to encode in QR codes.
BIP-173地址使用32个小写字母数字字符集，仔细选择以减少读写错误。
通过选择一个小写的字符集，bech32更容易读和说，并且用二维码编码时提高45%的效率。

The BCH error detection algorithm is a vast improvement over the previous checksum algorithm (from Base58Check), allowing not only detection but also *correction* of errors. Address-input interfaces (such as text-fields in forms) can detect and highlight which character was most likely mistyped when they detect an error.
BCH错误检测算法比以前的校验和算法（来自Base58Check）有很大改进，不仅可以检测，还可以纠错。地址输入接口（例如文本窗口）在检测到错误时，可以高亮可能的错误字符。

From the BIP-173 specification, here are some examples of bech32 addresses:
来自BIP-173规范，这里有一些bech32地址的例子：

Mainnet P2WPKH bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4
Testnet P2WPKH tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx
Mainnet P2WSH
bc1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3qccfmv3
Testnet P2WSH
tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7

As you can see in these examples, a segwit bech32 string is up to 90 characters long and consists of three parts:
可以看出，一个segwit bech32字符串最多有90个字符，由三个部分组成：

The human readable part 人可读部分
This prefix "bc" or "tb" identifying mainnet or testnet.
前缀bc表示mainet，前缀tb表示testnet。

The separator 分隔符
The digit "1", which is not part of the 32-character encoding set and can only appear in this position as a separator
数字1，它不是32字符编码集的一部分，只能出现在作为分隔符的位置。

The data part 数据部分
A minimum of 6 alphanumeric characters, the checksum encoded witness script
至少6个字母数字字符，校验和编码的见证脚本

At this time, only a few wallets accept or produce native segwit bech32 addresses, but as segwit adoption increases, you will see these more and more often.
当前，只有一些钱包接受或产生native segwit bech32地址，但随着segwit被采用增加，你会越来越多的看到它们。

7.8.5.5交易标识

One of the greatest benefits of Segregated Witness is that it eliminates third-party transaction malleability.
隔离见证的一个最大好处是：消除了第三方交易延展性。

Before segwit, transactions could have their signatures subtly modified by third parties, changing their transaction ID (hash) without changing any fundamental properties (inputs, outputs, amounts). This created opportunities for denial-of-service attacks as well as attacks against poorly written wallet software that assumed unconfirmed transaction hashes were immutable.
在segwit之前，交易的签名可能被第三方巧妙地修改，从而改变了它们的交易ID（哈希），而没有改变任何基本属性（输入、输出、金额）。
这为DoS攻击创造了机会，以及针对编写不好的钱包软件的攻击，这些软件假定未经证实的交易哈希是不可变的。

With the introduction of Segregated Witness, transactions have two identifiers, txid and wtxid. The traditional transaction ID txid is the double-SHA256 hash of the serialized transaction, without the witness data. A transaction wtxid is the double-SHA256 hash of the new serialization format of the transaction with witness data.
通过引入隔离见证，交易有两个标识：txid和wtxid。
传统的txid是序列化交易的双SHA256哈希，没有见证数据。
交易的wtxid是有见证数据的交易的新序列化格式的双SHA256哈希。

The traditional txid is calculated in exactly the same way as with a nonsegwit transaction. However, since the segwit transaction has empty scriptSigs in every input, there is no part of the transaction that can be modified by a third party. Therefore, in a segwit transaction, the txid is immutable by a third party, even when the transaction is unconfirmed.
传统的txid的计算方式与nonsegwit交易完全相同。
但是，由于segwit交易在每个输入中都有空的scriptSig，因此没有可由第三方修改的交易部分。
因此，在一个segwit交易中，即使交易还未确认，第三方也不能修改txid。

The wtxid is like an "extended" ID, in that the hash also incorporates the witness data. If a transaction is transmitted without witness data, then the wtxid and txid are identical. Note than since the wtxid includes witness data (signatures) and since witness data may be malleable, the wtxid should be considered malleable until the transaction is confirmed. Only the txid of a segwit transaction can be considered immutable by third parties and only if *all* the inputs of the transaction are segwit inputs.

wtxid就像一个“扩展的”ID，因为这个哈希也包含了见证数据。

如果被传输的交易没有见证数据，则wtxid和txid是相同的。

注意，因为wtxid包含了见证数据（签名），并且因为见证数据可能是可延展的，所以，在交易确认之前，wtxid应该被认为是可延展的。

只有segwit交易的txid被认为不能被第三方修改，并且在这个额交易的所有输入都是segwit输入。

Tip: Segregated Witness transactions have two IDs: txid and wtxid. The txid is the hash of the transaction without the witness data and the wtxid is the hash inclusive of witness data. The txid of a transaction where all inputs are segwit inputs is not susceptible to third-party transaction malleability.

提示： 隔离见证交易有两个ID：txid和wtxid。

txid是没有见证数据的交易的哈希，wtxid是包含见证数据的哈希。

如果一个交易的所有输入都是segwit输入，那么它的txid就不容易受第三方交易延展性的影响。

7.8.6隔离见证的新签名算法

Segregated Witness modifies the semantics of the four signature verification functions (CHECKSIG, CHECKSIGVERIFY, CHECKMULTISIG, and CHECKMULTISIGVERIFY), changing the way a transaction commitment hash is calculated.
“隔离见证”修改了四个签名验证函数（CHECKSIG，CHECKSIGVERIFY，CHECKMULTISIG，CHECKMULTISIGVERIFY）的语义，这改变了一个交易承诺哈些被计算的方式。

Signatures in bitcoin transactions are applied on a *commitment hash*, which is calculated from the transaction data, locking specific parts of the data indicating the signer’s commitment to those values. For example, in a simple SIGHASH_ALL type signature, the commitment hash includes all inputs and outputs.
比特币交易中的签名应用于一个交易哈希上（用交易数据计算这个承诺哈希），它锁定了数据的特定部分，表明签名者对这些值的承诺。
例如，在一个简单的SIGHASH_ALL类型签名中，承诺哈希包括所有的输入和输出。

Unfortunately, the way the commitment hash was calculated introduced the possibility that a node verifying the signature can be forced to perform a significant number of hash computations. Specifically, the hash operations increase in O(n^2) with respect to the number of signature operations in the transaction. An attacker could therefore create a transaction with a very large number of signature operations, causing the entire bitcoin network to have to perform hundreds or thousands of hash operations to verify the transaction.
不幸的是，计算承诺哈希的方式引入了这个可能性：验证签名的节点可能被迫执行大量哈希计算。
具体而言，根据交易中的签名操作的数量，哈希操作增加了O（n^2）。
因此，攻击者可以创建一个有大量签名操作的交易，导致整个比特币网络不得不执行成千上万次哈希操作来验证这个交易。

Segwit represented an opportunity to address this problem by changing the way the commitment hash is calculated. For segwit version 0 witness programs, signature verification occurs using an improved commitment hash algorithm as specified in BIP-143.
Segwit代表了解决这个问题的一个机会：改变承诺散列的计算方式。
对于segwit版本0见证程序，使用BIP-143中规定的改进的承诺哈希算法进行签名验证。

The new algorithm achieves two important goals.
Firstly, the number of hash operations increases by a much more gradual O(n) to the number of signature operations, reducing the opportunity to create denial-of-service attacks with overly complex transactions.
Secondly, the commitment hash now also includes the value (amounts) of each input as part of the commitment. This means that a signer can commit to a specific input value without needing to "fetch" and check the previous transaction referenced by the input. In the case of offline devices, such as hardware wallets, this greatly simplifies the communication between the host and the hardware wallet, removing the need to stream previous transactions for validation. A hardware wallet can accept the input value "as stated" by an untrusted host. Since the signature is invalid if that input value is not correct, the hardware wallet doesn’t need to validate the value before signing the input.
新算法实现了两个重要目标。
首先，哈希操作的数量是以签名操作的数量的O（n）增加，这减少了用过于复杂的交易创建DoS攻击的机会。
其次，承诺哈希现在还包括每个输入的值（金额），作为承诺的一部分。这意味着，签名者可以承诺一个特定的输入值，而不需要取出和检查这输入引用的之前的交易。
在离线设备（如硬件钱包）中，这极大地简化了主机与硬件钱包之间的通信，消除了对以前的交易流进行验证的需要。硬件钱包可能接受不可信主机提供的输入值。由于如果输入值不正确导致签名无效，则硬件钱包在对输入签名之前不需要验证这个值。

7.8.7隔离见证的经济激励

Bitcoin mining nodes and full nodes incur costs for the resources used to support the bitcoin network and the blockchain. As the volume of bitcoin transactions increases, so does the cost of resources (CPU, network bandwidth, disk space, memory). Miners are compensated for these costs through fees that are proportional to the size (in bytes) of each transaction. Nonmining full nodes are not compensated, so they incur these costs because they have a need to run an authoritative fully validating full-index node, perhaps because they use the node to operate a bitcoin business.
比特币挖掘节点和全节点会产生对资源的成本，用于支持比特币网络和区块链。
随着比特币交易量的增加，资源成本（CPU、网络带宽、磁盘空间、内存）也在增加。
矿工通过交易费（与字节数成比例）来获得成本补偿。
不挖矿的全节点没有补偿，所以它们承担这些成本，因为它们需要运行一个权威的充分验证全索引节点，可能是因为他们使用这个节点来运营一个比特币业务。

Without transaction fees, the growth in bitcoin data would arguably increase dramatically. Fees are intended to align the needs of bitcoin users with the burden their transactions impose on the network, through a market-based price discovery mechanism.
如果没有交易费，比特币数据的增长可能会大幅增加。
交易费是通过一个市场价格发现机制，来于平衡"比特币用户的需要"与"交易给网络施加的负担"。

The calculation of fees based on transaction size treats all the data in the transaction as equal in cost. But from the perspective of full nodes and miners, some parts of a transaction carry much higher costs. Every transaction added to the bitcoin network affects the consumption of four resources on nodes:
基于交易规模的费用计算将交易中的所有数据视为相同的成本。
但是从全节点和矿工的角度来看，交易的某些部分有更高的成本。
给比特币网络增加的每个交易都会影响节点上四种资源的消耗：

Disk Space 磁盘空间
Every transaction is stored in the blockchain, adding to the total size of the blockchain. The blockchain is stored on disk, but the storage can be optimized by "pruning" older transactions.
每个交易都存储在区块链中，增加了区块链的总大小。
区块链存储在磁盘上，但可以通过"剪掉"老的交易来优化存储。

CPU
Every transaction must be validated, which requires CPU time.
每个交易都必须被验证，这需要CPU时间。

Bandwidth 带宽
Every transaction is transmitted (through flood propagation) across the network at least once. Without any optimization in the block propagation protocol, transactions are transmitted again as part of a block, doubling the impact on network capacity.
每个交易至少通过网络传输一次（通过泛洪传播）。
在区块传播协议中没有任何优化，交易会作为区块的一部分被再次传播，这对加倍影响了网络容量。

Memory 内存
Nodes that validate transactions keep the UTXO index or the entire UTXO set in memory to speed up validation. Because memory is at least one order of magnitude more expensive than disk, growth of the UTXO set contributes disproportionately to the cost of running a node.
验证交易的节点在内存中保存UTXO索引或整个UTXO集，以加速验证。
因为内存至少比磁盘贵一个数量级，所以UTXO集的增长对运行节点的成本贡献不成比例。

As you can see from the list, not every part of a transaction has an equal impact on the cost of running a node or on the ability of bitcoin to scale to support more transactions. The most expensive part of a transaction are the newly created outputs, as they are added to the in-memory UTXO set. By comparison, signatures (aka witness data) add the least burden to the network and the cost of running a node, because witness data are only validated once and then never used again. Furthermore, immediately after receiving a new transaction and validating witness data, nodes can discard that witness data. If fees are calculated on transaction size, without discriminating between these two types of data, then the market incentives of fees are not aligned with the actual costs imposed by a transaction. In fact, the current fee structure actually encourages the opposite behavior, because witness data is the largest part of a transaction.
从表中可看出，并不是交易的每个部分都对运行节点的成本或比特币支持更多交易的能力产生同等影响。
交易中最昂贵的部分是新创建的输出，因为它们被添加到内存中的UTXO集。
相比之下，签名（又名见证数据）为网络和运行一个节点增加了最小的负担，因为见证数据只被验证一次，之后又不再使用。
此外，在收到新的交易并验证见证数据之后，节点可以丢弃该那个见证数据。
如果按照交易规模计算交易费，而不区分这两种数据，那么交易费的市场激励就与交易的实际成本不符。
事实上，目前的交易费结构鼓励了相反的行为，因为见证数据是交易的最大部分。

The incentives created by fees matter because they affect the behavior of wallets. All wallets must implement some strategy for assembling transactions that takes into consideration a number of factors, such as privacy (reducing address reuse), fragmentation (making lots of loose change), and fees. If the fees are overwhelmingly motivating wallets to use as few inputs as possible in transactions, this can lead to UTXO picking and change address strategies that inadvertently bloat the UTXO set.
交易费创创造的激励关系重大，因为它们影响钱包的行为。
所有的钱包都必须实行一些策略来组合交易，这些策略要考虑到隐私（减少地址重复使用）、碎片化（大量松散零钱）、交易费。
如果交易费压倒性地促使钱包在交易中使用尽可能少的输入，这可能导致UTXO选择和改变地址策略，从而不经意地膨胀UTXO集。

Transactions consume UTXO in their inputs and create new UTXO with their outputs. A transaction, therefore, that has more inputs than outputs will result in a decrease in the UTXO set, whereas a transaction that has more outputs than inputs will result in an increase in the UTXO set. Let's consider the *difference* between inputs and outputs and call that the "Net-new-UTXO." That's an important metric, as it tells us what impact a transaction will have on the most expensive network-wide resource, the in-memory UTXO set. A transaction with positive Net-new-UTXO adds to that burden. A transaction with a negative Net-new-UTXO reduces the burden. We would therefore want to encourage transactions that are either negative Net-new-UTXO or neutral with zero Net-new-UTXO.
交易消耗了其输入中的UTXO，并在输出中创建了新的UTXO。
因此，交易输入比输出多将导致UTXO集合的减少，而输出多于输入的交易将导致UTXO集合的增加。
我们考虑输入和输出之间的差异，并称之为"净新UTXO"。
这是一个重要指标，因为它告诉我们一个交易会对最昂贵的网络资源、内存中的UTXO集产生什么影响。
正值Net-new-UTXO交易增加了这一负担，负值Net-new-UTXO交易减轻了负担。
因此，我们希望鼓励交易的Net-new-UTXO是负值或零。

Let's look at an example of what incentives are created by the transaction fee calculation, with and without Segregated Witness. We will look at two different transactions. Transaction A is a 3-input, 2-output transaction, which has a Net-new-UTXO metric of –1, meaning it consumes one more UTXO than it creates, reducing the UTXO set by one. Transaction B is a 2-input, 3-output transaction, which has a Net-new-UTXO metric of 1, meaning it adds one UTXO to the UTXO set, imposing additional cost on the entire bitcoin network. Both transactions use multisignature (2-of-3) scripts to demonstrate how complex scripts increase the impact of segregated witness on fees. Let's assume a transaction fee of 30 satoshi per byte and a 75% fee discount on witness data:
我们来看一个例子，说明有无隔离见证时，交易费用计算产生了什么激励。
我们看两个不同的交易。交易A是3输入和2输出，交易B是2输入和3输出。
这两个交易都使用多签名（2-3）脚本来说明复杂脚本如何增加隔离见证对费用的影响。

我们假设交易费为每字节30聪，对见证数据是75%的折扣费用：

	交易A的费用（聪）	**交易B的费用（聪）**
无隔离见证	25710	18990
有隔离见证	8130	12045

Both transactions are less expensive when segregated witness is implemented. But comparing the costs between the two transactions, we see that before Segregated Witness, the fee is higher for the transaction that has a negative Net-new-UTXO. After Segregated Witness, the transaction fees align with the incentive to minimize new UTXO creation by not inadvertently penalizing transactions with many inputs.

当实现了隔离见证时，这两个交易都更便宜了。
但是比较这两个交易的成本，我们发现，在隔离见证之前，交易A的收费更高；在隔离见证之后，交易A的收费更低，这激励了更少的新的UTXO创建，而没有无意中惩罚许多输入的交易。

Segregated Witness therefore has two main effects on the fees paid by bitcoin users. Firstly, segwit reduces the overall cost of transactions by discounting witness data and increasing the capacity of the bitcoin blockchain. Secondly, segwit’s discount on witness data corrects a misalignment of incentives that may have inadvertently created more bloat in the UTXO set.
因此，隔离见证对比特币用户支付的交易费有两个主要的影响。
首先，segwit通过折扣见证数据和增加比特币区块链的容量来降低交易的总体成本。
其次，segwit对见证数据的折扣纠正了可能无意中在UTXO集中产生更多膨胀的激励的错位。