7.1介绍

在上一章中，我们介绍了比特币交易的基本元素，并且查看了最常见的交易脚本类型，即P2PKH脚本。在本章中，我们将介绍更高级的脚本，以及如何使用它来构建具有复杂条件的交易。

首先，我们将看看多重签名脚本。接下来，我们将检查第二个最常见的交易脚本Pay-to-Script-Hash，它打开了一个复杂脚本的整个世界。然后，我们将检查新的脚本操作符，通过时间锁定将比特币添加时间维度。

7.2多重签名

多重签名脚本设置了一个条件，其中N个公钥被记录在脚本中，并且至少有M个必须提供签名来解锁资金。这也称为M-N方案，其中N是密钥的总数，M是验证所需的签名的数量。例如，2/3的多重签名是三个公钥被列为潜在签名人，至少有2个有效的签名才能花费资金。此时，标准多重签名脚本限制在最多15个列出的公钥，这意味着您可以从1到15之间的多重签名或该范围内的任何组合执行任何操作。在本书发布之前，限制15个已列出d的密钥可能会被解除，因此请检查isStandard（）函数以查看当前网络接受的内容。

设置M-N多重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
```

M是花费输出所需的签名的数量，N是列出的公钥的总数。 设置2到3多重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
```

上述锁定脚本可由含有签名和公钥的脚本予以解锁： 或者由3个存档公钥中的任意2个相一致的私钥签名组合予以解锁。 两个脚本组合将形成一个验证脚本：

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

当执行时，只有当未解锁版脚本与解锁脚本设置条件相匹配时，组合脚本才显示得到结果为真（Ture）。

上述例子中相应的设置条件即为：未解锁脚本是否含有3个公钥中的任意2个相对应的私钥的有效签名。

CHECKMULTISIG执行中的bug

CHECKMULTISIG的执行中有一个bug，需要一些轻微的解决方法。 当CHECKMULTISIG执行时，它应该消耗堆栈(stack)上的M + N + 2个项目作为参数。 然而，由于该错误，CHECKMULTISIG将弹出（pop）超出预期的额外值或一个值。 我们来看看这个更详细的/使用以前的/验证示例：

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

首先，CHECKMULTISIG弹出最上面的项目，这是N（在这个例子中N是“3”）。然后它弹出N个项目，这是可以签名的公钥。在这个例子中，公钥A，B和C.然后，它弹出一个项目，即M，仲裁（需要多少个签名）。这里M = 2。此时，CHECKMULTISIG应弹出最终的M个项目，这些是签名，并查看它们是否有效。

然而，不幸的是，实施中的错误导致CHECKMULTISIG再弹出一个项目（总共M + 1个）。检查签名时，不考虑额外的项目，因此它对CHECKMULTISIG本身没有直接影响。但是，必须存在额外的值，因为如果不存在，则当CHECKMULTISIG尝试弹出空堆栈时，会导致堆栈错误和脚本失败（将交易标记为无效）。因为额外的项目被忽略，它可以是任何东西，但通常使用0。

因为这个bug成为共识规则的一部分，所以现在它必须永远被复制。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
```

这样解锁脚本就不是下面的：

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

而是：

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

从现在开始，如果你看到一个multisig解锁脚本，你应该期望看到一个额外的0开始，其唯一的目的是解决一个bug，意外地成为一个共识规则的解决方法。【译者注：即保证例子中有3个私钥签名（其中2有效签名，其中1个为0的无效签名）对应3个公钥用于检查多重签名，从而保证脚本不产生bug。】

7.3 P2SH（Pay-to-Script-Hash）

P2SH在2012年被作为一种新型、强大、且能大大简化复杂交易脚本的交易类型而引入。为进一步解释P2SH的必要性，让我们先看一个实际的例子。

在第1章中，我们曾介绍过Mohammed，一个迪拜的电子产品进口商。Mohammed的公司采用比特币多重签名作为其公司会计账簿记账要求。多重签名脚本是比特币高级脚本最为常见的运用之一，是一种具有相当大影响力的脚本。针对所有的顾客支付（即应收账款），Mohammed的公司要求采用多重签名交易。基于多重签名机制，顾客的任何支付都需要至少两个签名才能解锁，一个来自Mohammed，另一个来自其合伙人或拥有备份钥匙的代理人。这样的多重签名机制能为公司治理提供管控便利，同时也能有效防范盗窃、挪用和遗失。 最终的脚本非常长：

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

虽然多重签名十分强大，但其使用起来还是多有不便。基于之前的脚本，Mohammed必须在客户付款前将该脚本发送给每一位客户，而每一位顾客也必须使用特制的能产生客户交易脚本的比特币钱包软件，每位顾客还得学会如何利用脚本来完成交易。

此外，由于脚本可能包含特别长的公钥，最终的交易脚本可能是最初交易脚本长度的5倍之多。额外长度的脚本将给客户造成费用负担。最后，一个长的交易脚本将一直记录在所有节点的随机存储器的UTXO集中，直到该笔资金被使用。采用这种复杂输出脚本使得在实际交易中变得困难重重。

P2SH正是为了解决这一实际难题而被引入的，它旨在使复杂脚本的运用能与直接向比特币地址支付一样简单。在P2SH 支付中，复杂的锁定脚本被电子指纹所取代，电子指纹是指密码学中的哈希值。

当一笔交易试图支付UTXO时，要解锁支付脚本，它必须含有与哈希相匹配的脚本。P2SH的含义是，向与该哈希匹配的脚本支付，当输出被支付时，该脚本将在后续呈现。

在P2SH交易中，锁定脚本由哈希运算后的20字节的散列值取代，被称为赎回脚本。因为它在系统中是在赎回时出现而不是以锁定脚本模式出现。表7-1列示了非P2SH脚本，表7-2列示了P2SH脚本。

表7-1不含P2SH的复杂脚本

从表中可以看出，对于P2SH，详细描述了输出（赎回脚本）的条件的复杂脚本不会在锁定脚本中显示。

相反，只有它的散列值在锁定脚本中呈现，并且兑换脚本本身稍后呈现，作为解锁脚本在输出花费时的一部分。 这使得给矿工的交易费用从发送方转移到收款方，复杂的计算工作也从从发送方转移到收款方。

输出脚本（Redeem Script）中的**“2 Pubkey1 Pubkey2 Pubkey3 Pubkey4 Pubkey5 5 CHECKMULTISIG”**的内容，没有出现在锁定脚本（Locking Script 表中第二行内容）中，但对实现上很长的一大串的**“2 Pubkey1 Pubkey2 Pubkey3 Pubkey4 Pubkey5 5 CHECKMULTISIG”**（有520字节）进行哈希运算后的20字节的散列值取代之，然后将之（**“2 Pubkey1 Pubkey2 Pubkey3 Pubkey4 Pubkey5 5 CHECKMULTISIG”**）放到解锁脚本中（Unlocking Script）。这使得给矿工的交易费用从发送方转移到收款方，并且令复杂的计算工作也从从发送方转移到收款方。 【译者注：本段原文描述如下：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.】

让我们再看下Mohammed公司的例子，复杂的多重签名脚本和相应的P2SH脚本。 首先，Mohammed公司对所有顾客订单采用多重签名脚本： 2 <Mohammed's Public Key> 5 CHECKMULTISIG 如果占位符由实际的公钥（以04开头的520字节）替代，你将会看到的脚本会非常地长：

> 2
> 04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C39
> 5D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C58704A2192968D8655D6A935BEAF2CA2
> 3E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D
> 78D0E34224858008E8B49047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D
> 5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D99779650421D65CB
> D7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227
> 440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5043752580AFA1ECED3C68D446BCAB69AC0BA7D
> F50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000E
> B7ED238F4D800 5 CHECKMULTISIG

整个脚本都可由仅为20个字节的密码哈希所取代，首先采用SH256哈希算法，随后对其运用RIPEMD160算法。20字节 的脚本为：

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97
```

一笔P2SH交易运用锁定脚本将输出与哈希关联，而不是与前面特别长的脚本所关联。使用的锁定脚本为：

```
HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL
```

正如你所看到的，这个脚本比前面的长脚本简短多了。取代“向该5个多重签名脚本支付”，这个P2SH等同于“向含该哈希的脚本支付”。顾客在向Mohammed公司支付时，只需在其支付指令中纳入这个非常简短的锁定脚本即可。当Mohammed想要花费这笔UTXO时，附上原始赎回脚本（与UTXO锁定的哈希）和必要的解锁签名即可，如：

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

两个脚本经由两步实现组合。 首先，将赎回脚本与锁定脚本比对以确认其与哈希是否匹配：

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

假如赎回脚本与哈希匹配，解锁脚本会被执行以释放赎回脚本：

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

本章中描述的几乎所有脚本只能以P2SH脚本来实现。 它们不能直接用在UTXO的锁定脚本中。

7.3.1 P2SH地址

P2SH的另一重要特征是它能将脚本哈希编译为一个地址（其定义请见BIP0013 /BIP-13）。P2SH地址是基于Base58编码的一 个含有20个字节哈希的脚本，就像比特币地址是基于Base58编码的一个含有20个字节的公钥。由于P2SH地址采用5作为前缀，这导致基于Base58编码的地址以“3”开头。例如，Mohammed的脚本，基于Base58编码下的P2SH地址变 为“39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw”。

此时，Mohammed可以将该地址发送给他的客户，这些客户可以 采用任何的比特币钱包实现简单支付，就像这是一个比特币地址一样。以“3”为前缀给予客户这是一种特殊类型的地址的暗示，该地址与一个脚本相对应而非与一个公钥相对应，但是它的效果与比特币地址支付别无二致。 P2SH地址隐藏了所有的复杂性，因此，运用其进行支付的人将不会看到脚本。

7.3.2 P2SH的优点

与直接使用复杂脚本以锁定输出的方式相比，P2SH具有以下特点：

- 在交易输出中，复杂脚本由简短电子指纹取代，使得交易代码变短。
- 脚本能被编译为地址，支付指令的发出者和支付者的比特币钱包不需要复杂工序就可以执行P2SH。
- P2SH将构建脚本的重担转移至接收方，而非发送方。
- P2SH将长脚本数据存储的负担从输出方（存储于UTXO集，影响内存）转移至输入方（存储在区块链里面）。
- P2SH将长脚本数据存储的重担从当前（支付时）转移至未来（花费时）。
- P2SH将长脚本的交易费成本从发送方转移至接收方，接收方在使用该笔资金时必须含有赎回脚本。

7.3.3赎回脚本和标准确认

在0.9.2版比特币核心客户端之前，P2SH仅限于标准比特币交易脚本类型（即通过标准函数检验的脚本）。这也意味着使用该笔资金的交易中的赎回脚本只能是标准化的P2PK、P2PKH或者多重签名，而非RETURN 和P2SH。

作为0.9.2版的比特币核心客户端，P2SH交易能包含任意有效的脚本，这使得P2SH标准更为灵活，也可以用于多种新的或复杂类型的交易。

请记住不能将P2SH植入P2SH赎回脚本，因为P2SH不能自循环。虽然在技术上可以将RETURN包含在赎回脚本中，但由于规则中没有策略阻止您执行此操作，因此在验证期间执行RETURN将导致交易被标记为无效，因此这是不实际的。

需要注意的是，因为赎回脚本只有在你试图发送一个P2SH输出时才会在比特币网络中出现，假如你将输出与一个无效的交易哈希锁定，则它将会被忽略。该UTXO将会被成功锁定，但是你将不能使用该笔资金，因为交易中含有赎回脚本，该脚本因是一个无效的脚本而不能被接受。这样的处理机制也衍生出一个风险，你可能将比特币锁定在一个未来不能被花费的P2SH中。因为比特币网络本身会接受这一P2SH，即便它与无效的赎回脚本所对应（因为该赎回脚本哈希没有对其所表征的脚本给出指令）。

注释 P2SH锁定脚本包含一个赎回脚本哈希，该脚本对于赎回脚本本身未提供任何描述。P2SH交易即便在赎回脚本无效的情况下也会被认为有效。如果处理不当，有可能会出现一个事故，即你的比特币可能会被锁死在P2SH这个交易中，导致你以后再也不能花费这笔比特币了。

7.4 数据记录输出（RETURN操作符）

比特币的去中心特点和时间戳账本机制，即区块链技术，其潜在运用将大大超越支付领域。许多开发者试图充分发挥交易脚本语言的安全性和可恢复性优势，将其运用于电子公证服务、证券认证和智能合约等领域。很多早期的开发者利用比特币这种能将交易数据放到区块链上的技术进行了很多尝试，例如，为文件记录电子指纹，则任何人都可以通过该机制在特定的日期建立关于文档存在性的证明。

运用比特币的区块链技术存储与比特币支付不相关数据的做法是一个有争议的话题。许多开发者认为其有滥用的嫌疑，因而试图予以阻止。另一些开发者则将之视为区块链技术强大功能的有力证明，从而试图给予大力支持。那些反对非支付相关应用的开发者认为这样做将引致“区块链膨胀”，因为所有的区块链节点都将以消耗磁盘存储空间为成本，负担存储此类 数据的任务。

更为严重的是，此类交易仅将比特币地址当作自由组合的20个字节而使用，进而会产生不能用于交易的UTXO。因为比特币地址只是被当作数据使用，并不与私钥相匹配，所以会导致UTXO不能被用于交易，因而是一种伪支付行为。因此，这些交易永远不会被花费，所以永远不会从UTXO集中删除，并导致UTXO数据库的大小永远增加或“膨胀”。

在0.9版的比特币核心客户端上，通过采用Return操作符最终实现了妥协。Return允许开发者在交易输出上增加80字节的非交易数据。然后，与伪交易型的UTXO不同，Return创造了一种明确的可复查的非交易型输出，此类数据无需存储于UTXO集。Return输出被记录在区块链上，它们会消耗磁盘空间，也会导致区块链规模的增加，但 它们不存储在UTXO集中，因此也不会使得UTXO内存膨胀，更不会以消耗代价高昂的内存为代价使全节点都不堪重负。RETURN 脚本的样式：

```
RETURN <data>
```

“data”部分被限制为80字节，且多以哈希方式呈现，如32字节的SHA256算法输出。许多应用都在其前面加上前缀以辅助认定。例如，电子公正服务的证明材料采用8个字节的前缀“DOCPROOF”，在十六进制算法中，相应的ASCII码为 44 4f 43 50 52 4f 4f 46。

请记住 RETURN 不涉及可用于支付的解锁脚本的特点， RETURN 不能使用其输出中所锁定的资金，因此它也就没有必要记录在蕴含潜在成本的UTXO集中，所以 RETURN 实际是没有成本的。

RETURN 常为一个金额为0的比特币输出， 因为任何与该输出相对应的比特币都会永久消失。假如一笔 RETURN 被作为一笔交易的输入，脚本验证引擎将会阻止验证脚本的执行，将标记交易为无效。如果你碰巧将 RETURN 的输出作为另一笔交易的输入，则该交易是无效的。

一笔标准交易（通过了 isStandard() 函数检验的）只能有一个 RETURN 输出。但是单个RETURN 输出能与任意类型的输出交易进行组合。

Bitcoin Core中添加了两个新版本的命令行选项。 选项datacarrier控制RETURN交易的中继和挖掘，默认设置为“1”以允许它们。 选项datacarriersize采用一个数字参数，指定RETURN脚本的最大大小（以字节为单位），默认为83字节，允许最多80个字节的RETURN数据加上一个字节的RETURN操作码和两个字节的PUSHDATA操作码。

注释 最初提出了RETURN，限制为80字节，但是当功能被释放时，限制被减少到40字节。 2015年2月，在Bitcoin Core的0.10版本中，限制提高到80字节。 节点可以选择不中继或重新启动RETURN，或者只能中继和挖掘包含少于80字节数据的RETURN。

7.5时间锁（Timelocks）

时间锁是只允许在一段时间后才允许支出的交易。比特币从一开始就有一个交易级的时间锁定功能。它由交易中的nLocktime字段实现。在2015年底和2016年中期推出了两个新的时间锁定功能，提供UTXO级别的时间锁定功能。这些是CHECKLOCKTIMEVERIFY和CHECKSEQUENCEVERIFY。

时间锁对于后期交易和将资金锁定到将来的日期很有用。更重要的是，时间锁将比特币脚本扩展到时间的维度，为复杂的多级智能合同打开了大门。

7.5.1交易锁定时间（nLocktime）

比特币从一开始就有一个交易级的时间锁功能。交易锁定时间是交易级设置（交易数据结构中的一个字段），它定义交易有效的最早时间，并且可以在网络上中继或添加到区块链中。

锁定时间也称为nLocktime，是来自于Bitcoin Core代码库中使用的变量名称。在大多数交易中将其设置为零，以指示即时传播和执行。如果nLocktime不为零，低于5亿，则将其解释为块高度，这意味着交易无效，并且在指定的块高度之前未被中继或包含在块链中。

如果超过5亿，它被解释为Unix纪元时间戳（自Jan-1-1970之后的秒数），并且交易在指定时间之前无效。指定未来块或时间的nLocktime的交易必须由始发系统持有，并且只有在有效后才被发送到比特币网络。如果交易在指定的nLocktime之前传输到网络，那么第一个节点就会拒绝该交易，并且不会被中继到其他节点。使用nLocktime等同于一张延期支票。

7.5.1.1交易锁定时间限制

nLocktime就是一个限制，虽然它可以在将来花费，但是到现在为止，它并不能使用它们。我们来解释一下，下面的例子。

Alice签署了一笔交易，支付给Bob的地址，并将交易nLocktime设定为3个月。Alice把这笔交易发送给Bob。有了这个交易，Alice和Bob知道：

- 在3个月过去之前，Bob不能完成交易进行变现。
- Bob可以在3个月后接受交易。

  然而：

- Alice可以创建另一个交易，双重花费相同的输入，而不需要锁定时间。 因此，Alice可以在3个月过去之前花费相同的UTXO。
- Bob不能保证Alice不会这样做。

了解交易nLocktime的限制很重要。 唯一的保证是Bob在3个月过去之前无法兑换它。 不能保证Bob得到资金。 为了实现这样的保证，时间限制必须放在UTXO本身上，并成为锁定脚本的一部分，而不是交易。

这是通过下一种形式的时间锁定来实现的，称为检查锁定时间验证(CLTV)。

7.5.2检查锁定时间验证Check Lock Time Verify (CLTV)

2015年12月，引入了一种新形式的时间锁进行比特币软分叉升级。根据BIP-65中的规范，脚本语言添加了一个名为CHECKLOCKTIMEVERIFY（CLTV）的新脚本操作符。 CLTV是每个输出的时间锁定，而不是每个交易的时间锁定，与nLocktime的情况一样。这允许在应用时间锁的方式上具有更大的灵活性。 简单来说，通过在输出的赎回脚本中添加CLTV操作码来限制输出，从而只能在指定的时间过后使用。

注释 当nLocktime是交易级时间锁定时，CLTV是基于输出的时间锁。

CLTV不会取代nLocktime，而是限制特定的UTXO，并通过将nLocktim设置为更大或相等的值，从而达到在未来才能花费这笔钱的目的。

CLTV操作码采用一个参数作为输入，表示为与nLocktime（块高度或Unix纪元时间）相同格式的数字 。如VERIFY后缀所示，CLTV如果结果为FALSE，则停止执行脚本的操作码类型。如果结果为TRUE，则继续执行。

为了使用CLTV锁定输出，将其插入到创建输出的交易中的输出的赎回脚本中。例如，如果Alice支付Bob的地址，输出通常会包含一个这样的P2PKH脚本：

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

要锁定一段时间，比如说3个月以后，交易将是一个P2SH交易，其中包含一个赎回脚本：

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

其中是从交易开始被挖矿时间起计3个月的块高度或时间值：当前块高度+12,960（块）或当前Unix纪元时间+7,760,000（秒）。现在，不要担心CHECKLOCKTIMEVERIFY之后的DROP操作码,下面很快就会解释。

当Bob尝试花费这个UTXO时，他构建了一个引用UTXO作为输入的交易。他使用他的签名和公钥在该输入的解锁脚本，并将交易nLocktime设置为等于或更大于Alice设置的CHECKLOCKTIMEVERIFY 时间锁。然后，Bob在比特币网络上广播交易。

Bob的交易评估如下。如果Alice设置的CHECKLOCKTIMEVERIFY参数小于或等于支出交易的nLocktime，脚本执行将继续（就好像执行“无操作”或NOP操作码一样）。否则，脚本执行停止，并且该交易被视为无效。 更确切地说，CHECKLOCKTIMEVERIFY失败并停止执行，标记交易无效（来源：BIP-65）：

1. 堆栈是空的要么
2. 堆栈中的顶部项小于0;要么
3. 顶层堆栈项和nLocktime字段的锁定时间类型（高度或者时间戳）不相同;要么
4. 顶层堆栈项大于交易的nLocktime字段;要么
5. 输入的nSequence字段为0xffffffff。

注释 CLTV和nLocktime使用相同的格式来描述时间锁定，无论是块高度还是自Unix纪元以秒钟以来所经过的时间。 最重要的是，在一起使用时，nLocktime的格式必须与输入中的CLTV格式相匹配，它们必须以秒为单位引用块高度或时间。

执行后，如果满足CLTV，则其之前的时间参数仍然作为堆栈中的顶级项，并且可能需要使用DROP进行删除，才能正确执行后续脚本操作码。 为此，您将经常在脚本中看到CHECKLOCKTIMEVERIFY＋DROP在一起使用。

通过将nLocktime与CLTV结合使用，交易锁定时间限制中描述的情况发生变化。 因为Alice锁定了UTXO本身，所以现在Bob或Alice在3个月的锁定时间到期之前不可能花费它。

通过将时间锁定功能直接引入到脚本语言中，CLTV允许我们开发一些非常有趣的复杂脚本。 该标准在BIP-65（CHECKLOCKTIMEVERIFY）中定义（附录部分）。

7.5.3相对时间锁

nLocktime和CLTV都是绝对时间锁定，它们指定绝对时间点。接下来的两个时间锁定功能，我们将要考察的是相对时间锁定，因为它们将消耗输出的条件指定为从块链接中的输出确认起的经过时间。

相对时间锁是有用的，因为它们允许将两个或多个相互依赖的交易链接在一起，同时对依赖于从先前交易的确认所经过的时间的一个交易施加时间约束。换句话说，在UTXO被记录在块状块之前，时钟不开始计数。这个功能在双向状态通道和闪电网络中特别有用，我们将在后面章节[state_channels]中看到。

相对时间锁，如绝对时间锁定，同时具有交易级功能和脚本级操作码。交易级相对时间锁定是作为对每个交易输入中设置的交易字段nSequence的值的共识规则实现的。脚本级相对时间锁定使用CHECKSEQUENCEVERIFY（CSV）操作码实现。

相对时间锁是根据BIP-68与BIP－112的规范共同实现的，其中BIP-68通过与相对时间锁运用一致性增强的数字序列实现，BIP-112中是运用到了CHECKSEQUENCEVERIFY这个操作码实现。

BIP-68和BIP-112是在2016年5月作为软分叉升级时被激活的一个共识规则。

7.5.4 nSequence相对时间锁

相对时间锁定可以在每个输入中设置好，其方法是在每个输入中加多一个nSequence字段。

7.5.4.1 nSequence的本义

nnSequence字段的设计初心是想让交易能在在内存中修改，可惜后面从未运用过，使用nSequence这个字段时，如果输入的交易的序列值小于2^32 （0xFFFFFFFF），就表示尚未“确定”的交易。

这样的交易将在内存池中保存，直到被另一个交易消耗相同输入并具有较大nSequence值的代替。一旦收到一个交易，其投入的nSequence值为2^32，那么它将被视为“最终确定”并开采。 nSequence的原始含义从未被正确实现，并且在不利用时间锁定的交易中nSequence的值通常设置为2^32。对于具有nLocktime或CHECKLOCKTIMEVERIFY的交易，nSequence值必须设置为小于2^32，以使时间锁定器有效。通常设置为2^32 - 1（0xFFFFFFFE）。

7.5.4.2 nSequence作为一个共同执行的相对时间锁定

由于BIP-68的激活，新的共识规则适用于任何包含nSequence值小于2^31的输入的交易（bit 1<<31 is not set）。以编程方式，这意味着如果没有设置最高有效（bit 1<<31），它是一个表示“相对锁定时间”的标志。否则（bit 1<<31set），nSequence值被保留用于其他用途，例如启用CHECKLOCKTIMEVERIFY，nLocktime，Opt-In-Replace-By-Fee以及其他未来的新产品。

一笔输入交易，当输入脚本中的nSequence值小于2^31时，就是相对时间锁定的输入交易。这种交易只有到了相对锁定时间后才生效。例如，具有30个区块的nSequence相对时间锁的一个输入的交易只有在从输入中引用的UTXO开始的时间起至少有30个块时才有效。由于nSequence是每个输入字段，因此交易可能包含任何数量的时间锁定输入，所有这些都必须具有足够的时间以使交易有效。

交易可以包括时间锁定输入（nSequence <2^31）和没有相对时间锁定（nsequence> = 2^31）的输入。nSequence值以块或秒为单位指定，但与nLocktime中使用的格式略有不同。类型标志用于区分计数块和计数时间（以秒为单位）的值。类型标志设置在第23个最低有效位（即值1 << 22）。如果设置了类型标志，则nSequence值将被解释为512秒的倍数。如果未设置类型标志，则nSequence值被解释为块数。

当将nSequence解释为相对时间锁定时，只考虑16个最低有效位。一旦评估了标志（位32和23），nSequence值通常用16位掩码（例如nSequence＆0x0000FFFF）“屏蔽”。 下图显示由BIP-68定义的nSequence值的二进制布局。 Figure 1. BIP-68 definition of nSequence encoding (Source: BIP-68)

基于nSequence值的一致执行的相对时间锁定在BIP-68中。标准定义在BIP-68, Relative lock-time using consensus-enforced sequence numbers.

7.5.5 带CSV的相对时间锁

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，通常简称为CSV。 在UTXO的赎回脚本中评估时，CSV操作码仅允许在输入nSequence值大于或等于CSV参数的交易中进行消耗。实质上，这限制了UTXO的消耗，直到UTXO开采时间过了一定数量的块或秒。

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是根据块指定的，那么nSequence也是如此。如果以秒为单位指定CSV，那么nSequence也是如此。

当几个（已经形成链）交易被保留为“脱链”时，创建和签名这几个（已经形成链）交易但不传播时，CSV的相对时间锁特别有用。在父交易已被传播，直到消耗完相对锁定时间，才能使用子交易。这个用例的一个应用可以在state_channels和lightning_network/请加上链接 章节中看到。 CSV 细节参见 BIP-112, CHECKSEQUENCEVERIFY.

7.5.6中位时间过去Median-Time-Past

作为激活相对时间锁定的一部分，时间锁定（绝对和相对）的“时间”方式也发生了变化。在比特币中，墙上时间（wall time）和共识时间之间存在微妙但非常显著的差异。比特币是一个分散的网络，这意味着每个参与者都有自己的时间观。网络上的事件不会随时随地发生。网络延迟必须考虑到每个节点的角度。最终，所有内容都被同步，以创建一个共同的分类帐。比特币在过去存在的分类账状态中每10分钟达成一个新的共识。

区块头中设置的时间戳由矿工设定。共识规则允许一定的误差来解决分散节点之间时钟精度的问题。然而，这诱惑了矿工去说谎，以便通过包括还不在范围内的时间交易来赚取额外矿工费。有关详细信息，请参阅以下部分。

为了杜绝矿工说谎，加强时间安全性，在相对时间锁的基础上又新增了一个BIP。这是BIP-113，它定义了一个称为“中位时间过去 /（Median-Time-Past）”的新的共识测量机制。通过取最后11个块的时间戳并计算其中位数作为“中位时间过去”的值。这个中间时间值就变成了共识时间，并被用于所有的时间计算。过去约两个小时的中间点，任何一个块的时间戳的影响减小了。通过这个方法，没有一个矿工可以利用时间戳从具有尚未成熟的时间段的交易中获取非法矿工费。

Median-Time-Past更改了nLocktime，CLTV，nSequence和CSV的时间计算的实现。由Median-Time-Past计算的共识时间总是大约在挂钟时间后一个小时。如果创建时间锁交易，那么要在nLocktime，nSequence，CLTV和CSV中进行编码的估计所需值时，应该考虑它。 Median-Time-Past细节参见BIP-113.

7.5.7针对费用狙击（Fee Sniping）的时间锁定

费用狙击是一种理论攻击情形，矿工试图从将来的块（挑选手续费较高的交易）重写过去的块，实现“狙击”更高费用的交易，以最大限度地提高盈利能力。

例如，假设存在的最高块是块＃100,000。如果不是试图把＃100,001号的矿区扩大到区块链，那么一些矿工们会试图重新挖矿＃100,000。这些矿工可以选择在候选块＃100,000中包括任何有效的交易（尚未开采）。他们不必使用相同的交易来恢复块。事实上，他们有动力选择最有利可图（最高每kBB）的交易来包含在其中。它们可以包括处于“旧”块＃100,000中的任何交易，以及来自当前内存池的任何交易。当他们重新创建块＃100,000时，他们本质上可以将交易从“现在”提取到重写的“过去”中。

今天，这种袭击并不是非常有利可图，因为回报奖励（因为包括一定数量的比特币奖励）远远高于每个区块的总费用。但在未来的某个时候，交易费将是奖励的大部分（甚至是奖励的整体）。那时候这种情况变得不可避免了。

为了防止“费用狙击”，当Bitcoin Core /钱包 创建交易时，默认情况下，它使用nLocktime将它们限制为“下一个块”。在我们的环境中，Bitcoin Core /钱包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。 但是在区块链分叉攻击的情况下，由于所有这些交易都将被时间锁阻止在＃100,001，所以矿工们无法从筹码中提取高额交易。他们只能在当时有效的任何交易中重新挖矿#100,000，这导致实质上不会获得新的费用。 为了实现这一点，Bitcoin Core/钱包将所有新交易的nLocktime设置为，并将所有输入上的nSequence设置为0xFFFFFFFE以启用nLocktime。

7.6具有流量控制的脚本（条件子句 (Conditional Clauses)）

比特币脚本的一个更强大的功能是流量控制，也称为条件条款。您可能熟悉使用构造IF ... THEN ... ELSE的各种编程语言中的流控制。比特币条件条款看起来有点不同，但是基本上是相同的结构。

在基本层面上，比特币条件操作码允许我们构建一个具有两种解锁方式的赎回脚本，这取决于评估逻辑条件的TRUE / FALSE结果。例如，如果x为TRUE，则赎回脚本为A，ELSE赎回脚本为B. 此外，比特币条件表达式可以无限期地“嵌套”，这意味着这个条件语句可以包含其中的另外一个条件，另外一个条件其中包含别的条件等等 。Bitcoin脚本流控制可用于构造非常复杂的脚本，具有数百甚至数千个可能的执行路径。嵌套没有限制，但协商一致的规则对脚本的最大大小（以字节为单位）施加限制。

比特币使用IF，ELSE，ENDIF和NOTIF操作码实现流量控制。此外，条件表达式可以包含布尔运算符，如BOOLAND，BOOLOR和NOT。

乍看之下，您可能会发现比特币的流量控制脚本令人困惑。那是因为比特币脚本是一种堆栈语言。同样的方式，当1+1看起来“向后”当表示为1 1 ADD时，比特币中的流控制条款也看起来“向后”（backward）。 在大多数传统（程序）编程语言中，流控制如下所示： 大多数编程语言中的流控制伪代码

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

在基于堆栈的语言中，比如比特币脚本，逻辑条件出现在IF之前，这使得它看起来像“向后”，如下所示： Bitcoin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
```

阅读Bitcoin脚本时，请记住，评估的条件是在IF操作码之前。

7.6.1带有VERIFY操作码的条件子句

比特币脚本中的另一种条件是任何以VERIFY结尾的操作码。 VERIFY后缀表示如果评估的条件不为TRUE，脚本的执行将立即终止，并且该交易被视为无效。 与提供替代执行路径的IF子句不同，VERIFY后缀充当保护子句，只有在满足前提条件的情况下才会继续。

例如，以下脚本需要Bob的签名和产生特定哈希的前图像（秘密地）。

解锁时必须满足这两个条件：

1)具有EQUALVERIFY保护子句的赎回脚本。

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

为了兑现这一点，Bob必须构建一个解锁脚本，提供有效的前图像和签名：

2)一个解锁脚本以满足上述赎回脚本。

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

没有前图像，Bob无法访问检查其签名的脚本部分。

该脚本可以用IF编写： 具有IF保护条款的兑换脚本

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

Bob的解锁脚本是一样的： 解锁脚本以满足上述兑换脚本

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

使用IF的脚本与使用具有VERIFY后缀的操作码相同; 他们都作为保护条款。 然而，VERIFY的构造更有效率，使用较少的操作码。

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（保护条款），那么验证是更好的。 然而，如果我们想要有多个执行路径（流控制），那么我们需要一个IF ... ELSE流控制子句。

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评估。 相比之下，操作码EQUALVERIFY后缀不会在堆栈上留下任何东西。 在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

比特币脚本中流量控制的一个非常常见的用途是构建一个提供多个执行路径的赎回脚本，每个脚本都有一种不同的赎回UTXO的方式。

我们来看一个简单的例子，我们有两个签名人，Alice和Bob，两人中任何一个都可以兑换。 使用多重签名，这将被表示为1-of-2 多重签名脚本。 为了示范，我们将使用IF子句做同样的事情：

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

看这个赎回脚本，你可能会想：“条件在哪里？”IF子句之前没有什么！“ 条件不是赎回脚本的一部分。

相反，该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用解锁脚本兑换了这个：

```
<Alice's Sig> 1
```

最后的1作为条件（TRUE），将使IF子句执行Alice具有签名的第一个兑换路径。

为了兑换这个Bob，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的解锁脚本在堆栈中放置一个0，导致IF子句执行第二个（ELSE）脚本，这需要Bob的签名。

由于可以嵌套IF子句，所以我们可以创建一个“迷宫”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
```

在这种情况下，有三个执行路径（脚本A，脚本B和脚本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，例如，解锁脚本必须以1 0（TRUE，FALSE）结束。

这些值将被推送到堆栈，以便第二个值（FALSE）结束于堆栈的顶部。 外部IF子句弹出FALSE值并执行第一个ELSE子句。 然后，TRUE值移动到堆栈的顶部，并通过内部（嵌套）IF来评估，选择B执行路径。

使用这个结构，我们可以用数十或数百个执行路径构建赎回脚本，每个脚本提供了一种不同的方式来兑换UTXO。要花费，我们构建一个解锁脚本，通过在每个流量控制点的堆栈上放置相应的TRUE和FALSE值来导航执行路径。

7.7复杂的脚本示例

在本节中，我们将本章中的许多概念合并成一个例子。 我们的例子使用了迪拜公司所有者Mohammed的故事，他们正在经营进出口业务。

在这个例子中，Mohammed希望用灵活的规则建立公司资本账户。他创建的方案需要不同级别的授权，具体取决于时间锁定。

多重签名的计划的参与者是Mohammed，他的两个合作伙伴Saeed和Zaira，以及他们的公司律师Abdul。三个合作伙伴根据多数规则作出决定，因此三者中的两个必须同意。然而，如果他们的钥匙有问题，他们希望他们的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。

这是Mohammed设计的脚本： 具有时间锁定（Timelock）变量的多重签名

```
IF
  IF
    2
  ELSE
    <30 days> CHECKSEQUENCEVERIFY DROP
    <Abdul the Lawyer's Pubkey> CHECKSIGVERIFY
    1
  ENDIF
  <Mohammed's Pubkey> <Saeed's Pubkey> <Zaira's Pubkey> 3 CHECKMULTISIG
ELSE
  <90 days> CHECKSEQUENCEVERIFY DROP
  <Abdul the Lawyer's Pubkey> CHECKSIG
ENDIF
```

Mohammed的脚本使用嵌套的IF ... ELSE流控制子句来实现三个执行路径。

在第一个执行路径中，该脚本作为三个合作伙伴的简单的2-of-3 multisig操作。

该执行路径由第3行和第9行组成。第3行将multisig的定额设置为2（2 - 3）。

该执行路径可以通过在解锁脚本的末尾设置TRUE TRUE来选择： 解锁第一个执行路径的脚本（2-of-3 multisig）

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

提示 此解锁脚本开头的0是因为CHECKMULTISIG中的错误从堆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。 推送0（通常）是解决bug的方法，如CHECKMULTISIG执行中的错误章节所述。

第二个执行路径只能在UTXO创建30天后才能使用。 那时候，它需要签署Abdul（律师）和三个合作伙伴之一（三分之一）。

这是通过第7行实现的，该行将多选的法定人数设置为1。要选择此执行路径，解锁脚本将以FALSE TRUE结束： 解锁第二个执行路径的脚本(Lawyer + 1-of-3）

```
0 <Saeed's Sig> <Abdul's Sig> FALSE TRUE
```

提示 为什么先FALSE后TRUE？ 反了吗？这是因为这两个值被推到堆栈，所以先push FALSE，然后push TRUE。因此，第一个IF操作码首先弹出的是TRUE。

最后，第三个执行路径允许律师单独花费资金，但只能在90天之后。 要选择此执行路径，解锁脚本必须以FALSE结束： 解锁第三个执行路径的脚本（仅适用于律师）

```
<Abdul's Sig> FALSE
```

在纸上运行脚本来查看它在堆栈(stack)上的行为。

阅读这个例子还需要考虑几件事情。 看看你能找到答案吗？

- 为什么律师可以随时通过在解锁脚本中选择FALSE来兑换第三个执行路径？
- 在UTXO开采后分别有多少个执行路径可以使用5,35与105天？
- 如果律师失去钥匙，资金是否流失？ 如果91天过去了，你的答案是否会改变？
- 合作伙伴如何每隔29天或89天“重置”一次，以防止律师获得资金？
- 为什么这个脚本中的一些CHECKSIG操作码有VERIFY后缀，而其他的没有？