- p2pk和p2pkh的区别

  网络中的绝大多数交易，都是p2pkh，p2pk比p2pkh更简单，但是由于安全上的考虑，已经不再推荐被使用

  p2pkh是指公钥被hash了，p2pk没有被hash，这样做的好处就是使得目标地址变短了，pubkey相对变长了

- MULTISIG `pubKey`验证`sig`的算法时间复杂度

  ```
  n->     3
  ikey-> (pubKey3)
         (pubKey2)
         (pubKey1)
  m->     2
  isig-> (sig2)
         (sig1)
         0
  ```

  如上图所示，指针`ikey`指向pubKey，指针`isig`指向sig，试问验证算法的时间复杂度

  - 如果是pubKey和sig是无序排序的，则时间复杂度为O(mn)
  - 如果pubKey和sig是有序的，则时间复杂度为O(n)，用双指针算法

- P2SH(pay-to-script-hash)

  pay-to-script-hash就是把币发到一个脚本的哈希，而不是公钥或者公钥哈希

  和MULTICHECKSIG一样，P2SH也不是比特币诞生之初就有的，它是2012年的BIP 16中提出的。提出P2SH的目的主要是因为在之前的交易中，都是由发送者负责指定赎出币的条件。这样的话，如果赎出币的过程比较复杂，譬如要使用MULTISIG，那么对付钱的用户，也就是买家，就不够友好。使用P2SH的方式，可以由币的接收方设计好执行的脚本，然后不论脚本多么复杂，发送方只需要将币发送到一个20字节的哈希地址就行。

  相当于减轻了买家的负担，但是加重了卖家的负担，原因如下：

  - 在支付用户的交易T1中的输出脚本为：

    `OP_HASH160 [20-byte-hash-value] OP_EQUAL`

    相当于给地址加了一把锁

- 卖家如果想用这笔钱，首先第一步需要把地址上的这把锁给解开，这里解开的逻辑可以参考**比特币的机制(一)**，卖家需要提供用户hash脚本之前的脚本原数据才可以解开这把锁，之后正常按照正常的交易逻辑进行

简单来说就是在原来的锁的基础之上又加了一把地址锁，因此是方便了买家，但是麻烦了卖家

- redeemScriptHash和P2SHaddress的区别

  下图是一个买家Bob发出的交易信息：

Description		Hex Bytes
Version byte		01000000
Input count		01
Previous tx hash (reversed)		acc6fb9ec2c3884d3a12a89e7078c83853d9b7912281cefb14bac00a2737d33a
Output index		00000000
scriptSig length of 138 bytes		8a
scriptSig	Push 71 bytes to stack	47
	<signature>	304402204e63d034c6074f17e9c5f8766bc7b5468a0dce5b69578bd08554e8f21434c58e0220763c6966f47c39068c8dcd3f3dbd8e2a4ea13ac9e9c899ca1fbc00e2558cbb8b01
	Push 65 bytes to stack	41
	<pubKey>	0431393af9984375830971ab5d3094c6a7d02db3568b2b06212a7090094549701bbb9e84d9477451acc42638963635899ce91bacb451a1bb6da73ddfbcf596bddf
Sequence		ffffffff
No. of outputs		01
Amount of 65600 in LittleEndian		4000010000000000
scriptPubKey length of 23 bytes		17
scriptPubKey	OP_HASH160	a9
	Push 20 bytes to stack	14
	redeemScriptHash	1a8b0026343166625c7475f01e48b5ede8c0252e
	OP_EQUAL	87
locktime		00000000

注意里面有一个参数为`redeemScriptHash`,接下来看一上面数据中scriptPubKey的生成过程：

1. 首先Bob需要创建2-of-3 multisig P2SH地址。为了创建这个地址，首先Bob需要生成3个十六进制的公钥地址。这里使用go-bitcoin-multisig生成3对公私钥对：

   `go-bitcoin-multisig keys --count 3 --concise`

   生成的结果如下：

```
--------------
KEY #1
Private key:
```

```
5JruagvxNLXTnkksyLMfgFgf3CagJ3Ekxu5oGxpTm5mPfTAPez3
Public key hex:
04a882d414e47...
Public Bitcoin address:
1JzVFZSN1kxGLTHG41EVvY5gHxLAX7q1Rh
--------------
--------------
KEY #2
Private key:
5JX3qAwDEEaapvLXRfbXRMSiyRgRSW9WjgxeyJQWwBugbudCwsk
Public key hex:
046ce31db9bdd...
Public Bitcoin address:
14JfSvgEq8A8S7qcvxeaSCxhn1u1L71vo4
--------------
--------------
KEY #3
Private key:
5JjHVMwJdjPEPQhq34WMUhzLcEd4SD7HgZktEh8WHstWcCLRceV
Public key hex:
0411ffd36c70...
Public Bitcoin address:
1Kyy7pxzSKG75L9HhahRZgYoer9FePZL4R
--------------
```

这样就拥有了三个十六进制的公钥：

Key A:
`04a882d414e47803...`

Key B:
`046ce31db9bdd543...`

Key C:
`0411ffd36c70776...`

2. 之后我们指明我们需要一个2-of-3的地址，并且将我们的3个公钥作为输入，以生成该P2SH地址：

   `go-bitcoin-multisig address --m 2 --n 3 --public-keys 04a882d414e478039cd5b52a92ffb13dd5e6bd4515497439dffd691a...`

   上述命令的输出是：

```
---------------------
Your *P2SH ADDRESS* is:
347N1Thc213QqfYCz3PZkjoJpNv5b14kBd
Give this to sender funding multisig address with Bitcoin.
---------------------
---------------------
Your *REDEEM SCRIPT* is:
524104a882d414e478039...
Keep private and provide this to redeem multisig balance later.
---------------------
```

3. 将生成的P2SH地址提供给Alice

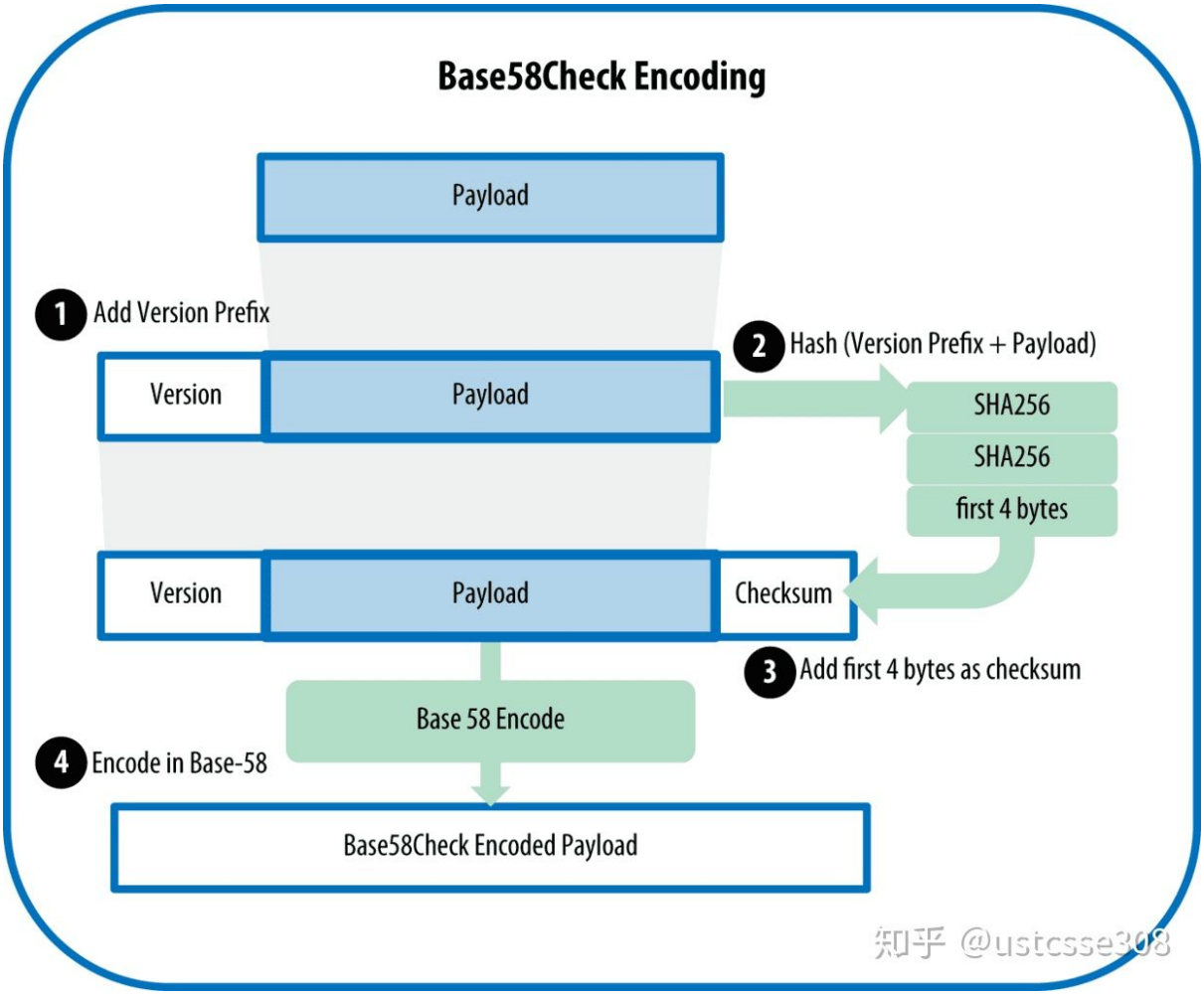
到此为止，生成的P2SH地址为`347N1...`，但是在redeemScriptHash中的地址却是`1a8b00...`，这是因为`347N1...`是`1a8b00...`经过base58check之后的结果，下面两个地址在逻辑上是等价的：

`347N1Thc213QqfYCz3PZkjoJpNv5b14kBd`

`1a8b0026343166625c7475f01e48b5ede8c0252e`

传输的过程中传输的是`347N1...`，这是因为Base58具有检错性质

- **Base58的特点**



为了更简洁方便地表示长串的数字。譬如，十进制计数系统使用0-9十个数字，而十六进制系统使用了额外的 A-F 六个字母。同样的数字，它的十六进制表 示就会比十进制表示更短。更进一步，Base64使用了26个小写字母、26个大写字母、10个数字以及两个符号（例 如“+”和“/”）。Base58是Base64编码格式的子集，同样使用大小写字母和10个数字，但舍弃了一些容易错 读和在特定字体中容易混淆的字符。具体地，Base58不含Base64中的0（数字0）、O（大写字母o）、l（小写字母 L）、I（大写字母i），以及“+”和“/”两个字符。简而言之，Base58就是由不包括（0，O，l，I）的大小写字母和数字组成。之所以做出这样的选择，就是对人友好，让人在看到Base58编码的数据之后不会疑惑，从而防止出错。这是因为，如果在比特币交易中如果因为看不清楚地址而输错了目标地址，那么付出去的钱是拿不回来的，所以一定要防止这种错误。

base58具有检错功能，这是因为base在传输过程中会加入checksum

- TimeLock

  在刚开始看交易的细节时，我们就遇到过Lock Time这个域。Lock Time顾名思义，就是锁定一些币，在达到某个时间或者某个区块之前不能使用这些币。在之前的交易中这个值都是0，也即不用锁定。那么在什么情况下需要使用lock time呢?

  虽说比特币交易比传统的交易费用低——譬如信用卡，当使用信用卡时，如果花费的金额较低，商家可能会拒绝接受信用卡，因为每一笔信用卡使用都需要付手续费，但是为了鼓励矿工尽快将自己的交易打包，一般都会在交易中预留交易费用。但是，有些情况下，可能需要快速地变更支付的费用，因此，就有必要防止快速而经常地进行交易而导致的交易费用。

  例如，用户需要在一段时间内连续地使用咖啡店的wifi，咖啡店希望每天支付一次流量费用。但是如果每天产生一笔交易，交易费用会很高。可以提出一种zero-trust的方案，意味着，交易是完全自动的， 只需要在最初预留一部分钱，然后系统会自动地按需进行支付，而咖啡店也能够放心地让用户使用而不至于担心用户会赖账。而真正进行广播，也即需要支付交易费用的交易的数量也能受到控制。

  思路是这样的：

  假设Alice是用户，Bob代表咖啡店。首先Alice生成一个交易Tx1，譬如支付100个币到一个2-of-2的multisig地址，也即这笔钱需要Alice和Bob共同签名才能使用。Alice首先对这个交易进行签名，然后广播这个交易。

  Bob看到这个交易之后可以让Alice使用wifi。接下来每天Alice生成新的一个交易发给Bob，使用Tx1中的钱支付给Bob，譬如第一天支付1个币给Bob，99个币给Alice；第二天支付2个币给Bob，98个币给Alice；等等。每天Bob看到这个交易，就会同意Alice继续使用网络。因为Tx1是2-of-2的交易类型，所以Bob看到Alice的签名，如果他想要获得支付，只要完成自己的签名部分就行了，所以Bob可以放心Alice不会赖账。

  当第28天Alice的工作完成不再需要咖啡店的网络了，就会通知Bob，对第28天的交易进行签名，也即总共支付28个币给Bob，剩余的72个币会返还给Alice。

  我们来想一下，这个过程中，Bob可以放心，对Alice会不会有损失?

  如果Bob是诚实的，这个过程会很顺利；但是如果Bob比较坑，在Alice使用完网络之后他一直不签名，那么Alice预付的100个币就一直锁死在网络中了。虽然Bob没有获得自己应得的那部分钱，但是Alice的损失更大。

  为了防止出现这种情况，可以使用lock_time。

  1. 首先Alice创建public key (K1)，然后请求Bob的公钥(K2)。
  2. 创建一个OP_CHECKMULTISIG交易Tx1，支付100个币到Multisig地址，也即需要Alice和Bob两人签名才能使用。Alice对这个交易签名，但是暂时并不广播。

3. Alice创建退款交易Tx2，Tx2使用Tx1的输出作为输入，并且将所有的钱都返回给Alice。这个交易设置了lock_time，譬如30天之后。Alice将这个交易提供给Bob。
4. Tx2主要是为了防止Bob坑，所以Bob为了证明自己不坑，会给Tx2签名，然后将签名返回给Alice。
5. Alice验证Bob的签名，如果正确，说明她的退款有保障，因此也就可以放心。
6. Alice此时对Tx1进行签名（这是对Tx1的input的支付签名），并且将签名发送给Bob。此时Alice或者Bob可以发布Tx1。此时Alice的100个币相当于被锁定了。
7. 然后Alice创建新的交易Tx3，使用Tx1的输出作为输入。Tx3类似于Tx2，但是有两个输出，譬如1个币给Bob，99个币给Alice。Alice对这个交易签名，发给Bob。
8. Bob收到Tx3和Alice的之后，验证签名的正确性。此时Bob如果加上自己的签名，就可以发布和广播这个交易，并获得1个币。但是因为Alice还在持续地使用Bob提供的服务，马上对这个交易进行签名，很明显是不明智的。
9. 之后每天Alice会继续创建类似的交易Tx3，都是用Tx1的输出作为输入。但是每次支付给Bob的币都在增多，留给自己的在减少。Bob收到之后进行验证。
10. 当Alice决定停止使用服务的时候，通知Bob，Bob对收到的最后一个Tx3进行签名并且广播。

如果Alice想利用Tx2进行双重支付，会不会成功呢？这时就是locktime起作用的时候了。因为这个Tx2不会立刻生效，所以Bob签字的Tx3会被首先确认，之后Tx2因为和Tx3使用的同一个输入，所以Tx2就是一个无效交易，因此双重支付不会成功。

- 五大交易脚本
    - **P2PKH（Pay-to-Public-Key-Hash）**
    - **P2PK(Pay-to-Public-Key)**
    - **多重签名**
    - **数据记录输出**
    - **P2SH（Pay-to-Script-Hash）**