可编程支付原理

比特币的所有交易的信息都被记录在比特币的区块链中，任何用户都可以通过公钥查询到某个交易的输入和输出金额。当某个用户希望花费一个输出时，例如，小明想要把某个公钥地址的输出支付给小红，他就需要使用自己的私钥对这笔交易进行签名，而矿工验证这笔交易的签名是有效的之后，就会把这笔交易打包到区块中，从而使得这笔交易被确认。

但比特币的支付实际上并不是直接支付到对方的地址，而是一个脚本，这个脚本的意思是：谁能够提供另外一个脚本，让这两个脚本能顺利执行通过，谁就能花掉这笔钱：

```
FROM: UTXO Hash#index
AMOUNT: 0.5 btc
TO: OP_DUP OP_HASH160 <address> OP_EQUALVERIFY OP_CHECKSIG
```

所以，比特币交易的输出是一个锁定脚本，而下一个交易的输入是一个解锁脚本。必须提供一个解锁脚本，让锁定脚本正确运行，那么该输入有效，就可以花费该输出。

我们以真实的比特币交易为例，某个交易的某个输出脚本是`76a914dc...489c88ac`这样的二进制数据，注意这里的二进制数据是用十六进制表示的，而花费该输出的某个交易的输入脚本是`48304502...14cf740f`这样的二进制数据，也是十六进制表示的：

我们先来看锁定脚本，锁定脚本的第一个字节`76`翻译成比特币脚本的字节码就是`OP_DUP`，`a9`翻译成比特币脚本的字节码就是`OP_HASH160`。`14`表示这是一个20字节的数据，注意十六进制的`14`换算成十进制是20，于是我们得到20字节的数据。最后两个字节，`88`表示字节码`OP_EQUALVERIFY`，`ac`表示字节码`OP_CHECKSIG`，所以整个锁定脚本是：

```
        OP_DUP 76
    OP_HASH160 a9
          DATA 14 (dc5dc65c...fe9f489c)
OP_EQUALVERIFY 88
   OP_CHECKSIG ac
```

我们再来看解锁脚本。解锁脚本的第一个字节`48`表示一个72字节长度的数据，因为十六进制的`48`换算成十进制是72。接下来的字节`21`表示一个33字节长度的数据。因此，该解锁脚本实际上只有两个数据。

```
DATA 48 (30450221...68fa9b01)
DATA 21 (03dd8763...14cf740f)
```

接下来，我们就需要验证这个交易是否有效。要验证这个交易，首先，我们要把解锁脚本和锁定脚本拼到一块，然后，开始执行这个脚本：

```
          DATA 48 (30450221...68fa9b01)
          DATA 21 (03dd8763...14cf740f)
        OP_DUP 76
    OP_HASH160 a9
          DATA 14 (dc5dc65c...fe9f489c)
OP_EQUALVERIFY 88
   OP_CHECKSIG ac
```

比特币脚本是一种基于栈结构的编程语言，所以，我们要先准备一个空栈，用来执行比特币脚本。然后，我们执行第一行代码，由于第一行代码是数据，所以直接把数据压栈：

```
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
├────────────────────┤
|30450221...68fa9b01|
└────────────────────┘
```

紧接着执行第二行代码，第二行代码也是数据，所以直接把数据压栈：

```
|                    |
|                    |
|                    |
|                    |
├────────────────────┤
|03dd8763...14cf740f|
├────────────────────┤
|30450221...68fa9b01|
└────────────────────┘
```

接下来执行`OP_DUP`指令，这条指令会把栈顶的元素复制一份，因此，我们现在的栈里面一共有3份数据：

```
|                   |
|                   |
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|30450221...68fa9b01|
└───────────────────┘
```

然后，执行`OP_HASH160`指令，这条指令会计算栈顶数据的hash160，也就是先计算SHA-256，再计算RipeMD160。对十六进制数据`03dd8763f8c3db6b77bee743ddafd33c969a99cde9278deb441b09ad7c14cf740f`计算hash160后得到结果`dc5dc65c7e6cc3c404c6dd79d83b22b2fe9f489c`，然后用结果替换栈顶数据：

```
|                   |
|                   |
├───────────────────┤
|dc5dc65c...fe9f489c|
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|30450221...68fa9b01|
└───────────────────┘
```

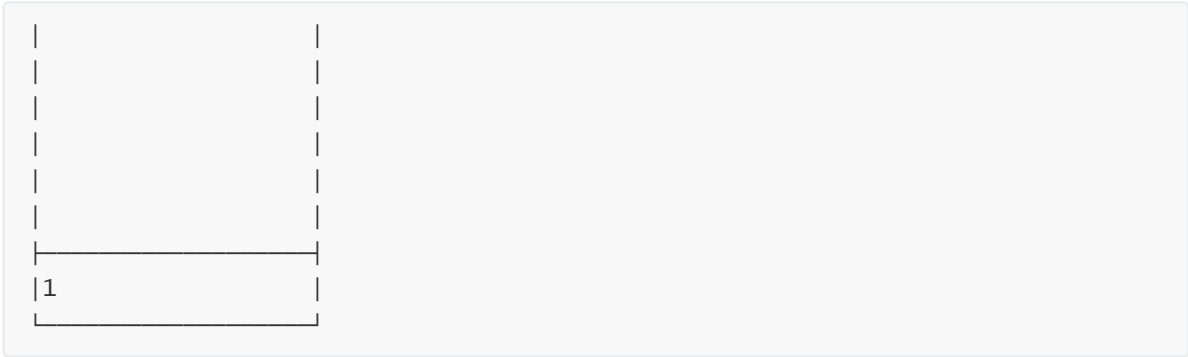
接下来的指令是一条数据，所以直接压栈：

```
├───────────────────┤
|dc5dc65c...fe9f489c|
├───────────────────┤
|dc5dc65c...fe9f489c|
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|30450221...68fa9b01|
└───────────────────┘
```

然后，执行`OP_EQUALVERIFY`指令，它比较栈顶两份数据是否相同，如果相同，则验证通过，脚本将继续执行，如果不同，则验证失败，整个脚本就执行失败了。在这个脚本中，栈顶的两个元素是相同的，所以验证通过，脚本将继续执行：

```
|                   |
|                   |
|                   |
|                   |
├───────────────────┤
|03dd8763...14cf740f|
├───────────────────┤
|30450221...68fa9b01|
└───────────────────┘
```

最后，执行`OP_CHECKSIG`指令，它使用栈顶的两份数据，第一份数据被看作公钥，第二份数据被看作签名，这条指令就是用公钥来验证签名是否有效。根据验证结果，成功存入`1`，失败存入`0`：



最后，当整个脚本执行结束后，检查栈顶元素是否为`0`，如果不为`0`，那么整个脚本就执行成功，这笔交易就被验证为有效的。

上述代码执行过程非常简单，因为比特币的脚本不含条件判断、循环等复杂结构。上述脚本就是对输入的两个数据视作签名和公钥，然后先验证公钥哈希是否与地址相同，再根据公钥验证签名，这种标准脚本称之为P2PKH（Pay to Public Key Hash）脚本。

输出

当小明给小红支付一笔比特币时，实际上小明创建了一个锁定脚本，该锁定脚本中引入了小红的地址。要想通过解锁脚本花费该输出，只有持有对应私钥的小红才能创建正确的解锁脚本（因为解锁脚本包含的签名只有小红的私钥才能创建），因此，小红事实上拥有了花费该输出的权利。

使用钱包软件创建的交易都是标准的支付脚本，但是，比特币的交易本质是成功执行解锁脚本和锁定脚本，所以，可以编写各种符合条件的脚本。比如，有人创建了一个交易，它的锁定脚本像这样：

```
OP_HASH256
      DATA 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
  OP_EQUAL
```

这有点像一个数学谜题。它的意思是说，谁能够提供一个数据，它的hash256等于`6fe28c0a...`，谁就可以花费这笔输出。所以，解锁脚本实际上只需要提供一个正确的数据，就可以花费这笔输出。点这里查看谁花费了该输出。

比特币的脚本通过不同的指令还可以实现更灵活的功能。例如，多重签名可以让一笔交易只有在多数人同意的情况下才能够进行。最常见的多重签名脚本可以提供3个签名，只要任意两个签名被验证成功，这笔交易就可以成功。

```
FROM: UTXO Hash#index
AMOUNT: 10.5 btc
TO: P2SH: OP_2 pk1 pk2 pk3 OP_3 OP_CHECKMULTISIG
```

也就是说，3个人中，只要任意两个人同意用他们的私钥提供签名，就可以完成交易。这种方式也可以一定程度上防止丢失私钥的风险。3个人中如果只有一个人丢失了私钥，仍然可以保证这笔输出是可以被花费的。

支付的本质

从比特币支付的脚本可以看出，比特币支付的本质是由程序触发的数字资产转移。这种支付方式无需信任中介的参与，可以在零信任的基础上完成数字资产的交易，这也是为什么数字货币又被称为可编程的货币。

由此催生出了智能合约：当一个预先编好的条件被触发时，智能合约可以自动执行相应的程序，自动完成数字资产的转移。保险、贷款等金融活动在将来都可以以智能合约的形式执行。智能合约以程序来替代传统的纸质文件条款，并由计算机强制执行，将具有更高的更低的信任成本和运营成本。

小结

比特币采用脚本的方式进行可编程支付：通过执行解锁脚本确认某个UTXO的资产可以被私钥持有人转移给其他人。

多重签名

由比特币的签名机制可知，如果丢失了私钥，没有任何办法可以花费对应地址的资金。

这样就使得因为丢失私钥导致资金丢失的风险会很高。为了避免一个私钥的丢失导致地址的资金丢失，比特币引入了多重签名机制，可以实现分散风险的功能。

具体来说，就是假设N个人分别持有N个私钥，只要其中M个人同意签名就可以动用某个“联合地址”的资金。

多重签名地址实际上是一个Script Hash，以2-3类型的多重签名为例，它的创建过程如下：

```
const bitcoin = require('bitcoinjs-lib');
```

```
let
    pubKey1 =
'026477115981fe981a6918a6297d9803c4dc04f328f22041bedff886bbc2962e01',
    pubKey2 =
'02c96db2302d19b43d4c69368babace7854cc84eb9e061cde51cfa77ca4a22b8b9',
    pubKey3 =
'03c6103b3b83e4a24a0e33a4df246ef11772f9992663db0c35759a5e2ebf68d8e9',
    pubKeys = [pubKey1, pubKey2, pubKey3].map(s => Buffer.from(s, 'hex')); // 注意把string转换为Buffer

// 创建2-3 RedeemScript:
let redeemScript = bitcoin.script.multisig.output.encode(2, pubKeys);
console.log('Redeem script: ' + redeemScript.toString('hex'));

// 编码:
let scriptPubKey =
bitcoin.script.scriptHash.output.encode(bitcoin.crypto.hash160(redeemScript));
let address = bitcoin.address.fromOutputScript(scriptPubKey);

console.log('Multisig address: ' + address); //
36NUkt6FWUi3LAWBqWRdDmdTWbt91Yvfu7
```

```
Redeem script:
522102647711598 1fe981a6918a6297d9803c4dc04f328f22041bedff886bbc2962e012102c96db2
302d19b43d4c69368babace7854cc84eb9e061cde51cfa77ca4a22b8b92103c6103b3b83e4a24a0e
33a4df246ef11772f9992663db0c35759a5e2ebf68d8e953ae
Multisig address: 36NUkt6FWUi3LAWBqWRdDmdTWbt91Yvfu7
```

首先，我们需要所有公钥列表，这里是3个公钥。然后，通过`bitcoin.script.multisig.output.encode()`方法编码为2-3类型的脚本，对这个脚本计算hash160后，使用Base58编码即得到总是以`3`开头的多重签名地址，这个地址实际上是一个脚本哈希后的编码。

> 以3开头的地址就是比特币的多重签名地址，但从地址本身无法得知签名所需的M/N。

如果我们观察Redeem Script的输出，它的十六进制实际上是：

```
52
21 026477115981fe981a6918a6297d9803c4dc04f328f22041bedff886bbc2962e01
21 02c96db2302d19b43d4c69368babace7854cc84eb9e061cde51cfa77ca4a22b8b9
21 03c6103b3b83e4a24a0e33a4df246ef11772f9992663db0c35759a5e2ebf68d8e9
53
ae
```

翻译成比特币的脚本指令就是：

```
OP_2
PUSHDATA(33) 026477115981fe981a6918a6297d9803c4dc04f328f22041bedff886bbc2962e01
PUSHDATA(33) 02c96db2302d19b43d4c69368babace7854cc84eb9e061cde51cfa77ca4a22b8b9
PUSHDATA(33) 03c6103b3b83e4a24a0e33a4df246ef11772f9992663db0c35759a5e2ebf68d8e9
OP_3
OP_CHECKMULTISIG
```

`OP_2` 和 `OP_3` 构成2-3多重签名，这两个指令中间的3个 `PUSHDATA(33)` 就是我们指定的3个公钥，最后一个 `OP_CHECKMULTISIG` 表示需要验证多重签名。

发送给多重签名地址的交易创建的是P2SH脚本，而花费多重签名地址的资金需要的脚本就是M个签名+Redeem Script。

注意：从多重签名的地址本身并无法得知该多重签名使用的公钥，以及M-N的具体数值。必须将Redeem Script公示给每个私钥持有人，才能够验证多重签名地址是否正确（即包含了所有人的公钥，以及正确的M-N数值）。要花费多重签名地址的资金，除了M个私钥签名外，必须要有Redeem Script（可由所有人的公钥构造）。只有签名，没有Redeem Script是不能构造出解锁脚本来花费资金的。因此，保存多重签名地址的钱包必须同时保存Redeem Script。

利用多重签名，可以实现：

- 1-2，两人只要有一人同意即可使用资金；
- 2-2，两人必须都同意才可使用资金；
- 2-3，3人必须至少两人同意才可使用资金；
- 4-7，7人中多数人同意才可使用资金。

最常见的多重签名是2-3类型。例如，一个提供在线钱包的服务，为了防止服务商盗取用户的资金，可以使用2-3类型的多重签名地址，服务商持有1个私钥，用户持有两个私钥，一个作为常规使用，一个作为应急使用。这样，正常情况下，用户只需使用常规私钥即可配合服务商完成正常交易，服务商因为只持有1个私钥，因此无法盗取用户资金。如果服务商倒闭或者被黑客攻击，用户可使用自己掌握的两个私钥转移资金。

大型机构的比特币通常都使用多重签名地址以保证安全。例如，某个交易所的3-6多重签名地址 3D2oetdNuZUqQHPJmcMDDHYoqkyNVsFk9r。

利用多重签名，可以使得私钥丢失的风险被分散到N个人手中，并且，避免了少数人窃取资金的问题。

比特币的多重签名最多允许15个私钥参与签名，即可实现1-2至15-15的任意组合（$1 \leqslant M \leqslant N \leqslant 15$）。

小结

多重签名可以实现N个人持有私钥，其中M个人同意即可花费资金的功能。

多重签名降低了单个私钥丢失的风险。

支付比特币到一个多重签名地址实际上是创建一个P2SH输出。