用Go构建区块链——6.交易2

本篇是"用Go构建区块链"系列的第六篇，主要对原文进行翻译。对应原文如下：

Building Blockchain in Go. Part 6: Transactions 2

1、介绍

在本系列的第一部分中，我说过区块链是一个分布式数据库。那时候，我们决定跳过"分布式"部分，专注于"数据库"部分。到目前为止，我们已经实现了几乎所有构成区块链数据库的东西。在本文中，我们将介绍一些在前面部分中跳过的机制，下一部分我们将开始研究区块链的分布式特性。

之前的章节：

1. 基本原型
2. 工作量证明
3. 持久化和命令行
4. 交易1
5. 地址

> 这部分介绍了重要的代码更改，所以在这里解释它们是没有意义的。请参阅此页面以查看自上一篇文章以来的所有更改。

2、奖励

在之前的文章中我们跳过的一件小事就是挖矿奖励。而且我们已经拥有了实现它的一切。

奖励只是一个coinbase交易。当一个挖矿节点开始挖掘新的区块时，它会从队列中获取交易信息，并向它们添加coinbase交易。coinbase交易的唯一输出包含矿工的公钥哈希。

实现奖励机制与更新 `send` 命令一样简单：

```
func (cli *CLI) send(from, to string, amount int) {
    ...
    bc := NewBlockchain()
    UTXOSet := UTXOSet{bc}
    defer bc.db.Close()

    tx := NewUTXOTransaction(from, to, amount, &UTXOSet)
    cbTx := NewCoinbaseTX(from, "")
    txs := []*Transaction{cbTx, tx}
```

```
    newBlock := bc.MineBlock(txs)
    fmt.Println("Success!")
}
```

在我们的实现中，创建交易的人挖到新区块时，会得到一笔奖励。

3、UTXO 集

在 第三部分：持久化和命令行 中，我们研究了比特币核心在数据库中存储块的方式。据说块存储在 `blocks` 数据库中，事务输出存储在 `chainstate` 数据库中。回顾下 `chainstate` 的数据结构吧：
1. 'c' + 32字节的交易哈希 -> 该笔交易的未花费交易输出记录
2. 'B' -> 32字节区块哈希 : 数据库表示的未花费交易输出的区块哈希

从那篇文章开始，我们已经实现了交易处理，但是我们还没有使用 `chainstate` 来存储它们的输出。所以，这就是我们接下来要做的事情。

`chainstate` 不存储交易。相反，它存储所谓的UTXO集合，或未花费的交易输出集合。除此之外，它存储"数据库表示未花费的交易输出的区块哈希"，这部分我们暂时忽略它，因为我们没有使用区块高度（但我们将在下一篇文章中实现它们）。

那么，为什么我们想要设置UTXO集合呢?

思考下我们之前实现的 `Blockchain.FindUnspentTransactions` 方法：

```
func (bc *Blockchain) FindUnspentTransactions(pubKeyHash []byte) []Transacti
on {
    ...
    bci := bc.Iterator()

    for {
        block := bci.Next()

        for _, tx := range block.Transactions {
            ...
        }

        if len(block.PrevBlockHash) == 0 {
            break
        }
    }
    ...
}
```

这个方法会找到那些未花费输出的交易。由于交易被保存在区块中，所以它会遍历区块链上的每个区块，来检查每笔交易。截至2017年9月18日，比特币中有485,860个块，整个数据库需要140 Gb以上的磁盘空间。这意味着必须运行一个完整节点来验证交易。而且，验证交易将需要遍历许多区块。

这个问题的解决方案是有一个只存储未花费输出的索引，这就是UTXO集合所做的：这是一个由所有区块链交易构建的缓存（通过迭代块，是的，但是这只能完成一次），并且稍后用于计算余额并验证新的交易。截至2017年9月，UTXO集合约为2.7 Gb的磁盘空间。

好的，让我们想想我们需要改变以实现UTXO集。目前，以下方法用于查找交易：

1. `Blockchain.FindUnspentTransactions` - 查找未花费输出的交易的主要功能。这是所有区块迭代发生的这个函数。
2. `Blockchain.FindSpendableOutputs` - 创建新交易时使用此功能。如果找到足够数量的输出持有所需的数量。使用 `Blockchain.FindUnspentTransactions` 。
3. `Blockchain.FindUTXO` - 找到公密哈希的未花费输出，用于获取余额。使用 `Blockchain.FindUnspentTransactions` 。
4. `Blockchain.FindTransaction` - 通过ID在区块链中查找交易。它遍历所有区块直到找到为止。

你可以看到，所有的方法都要遍历数据库中的区块。但现在我们无法改进它们，因为UTXO集不存储所有事务，但只存储那些没有使用输出的事务。因此，它不能用于 `Blockchain.FindTransaction` 。

所以，我们想要以下方法：

1. `Blockchain.FindUTXO` - 通过迭代区块来查找所有未花费的输出。
2. `UTXOSet.Reindex` - 用于FindUTXO查找未使用的输出，并将它们存储在数据库中。这就是缓存发生的地方。
3. `UTXOSet.FindSpendableOutputs` - 和 `Blockchain.FindSpendableOutputs` 很像，但使用的是UTXO设置。
4. `UTXOSet.FindUTXO` - 和 `Blockchain.FindUTXO` 很像，但使用的是UTXO设置。
5. `Blockchain.FindTransaction` 保持不变。

因此，两个最常用的函数将从现在起使用缓存！我们开始编码。

```
type UTXOSet struct {
    Blockchain *Blockchain
}
```

我们将使用单个数据库，但我们会将UTXO集存储在不同的存储桶(bucket)中。因此， `UTXOSet`

会和 `Blockchain` 相结合。

```
func (u UTXOSet) Reindex() {
    db := u.Blockchain.db
    bucketName := []byte(utxoBucket)

    err := db.Update(func(tx *bolt.Tx) error {
        err := tx.DeleteBucket(bucketName)
        _, err = tx.CreateBucket(bucketName)
    })

    UTXO := u.Blockchain.FindUTXO()

    err = db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket(bucketName)

        for txID, outs := range UTXO {
            key, err := hex.DecodeString(txID)
            err = b.Put(key, outs.Serialize())
        }
    })
}
```

此方法初始化了UTXO集。首先，如果已经存在了bucket，就把它删掉。然后从区块链中获取所有未花费的输出，最后将输出保存到存储桶(bucket)中。

`Blockchain.FindUTXO` 几乎跟 `Blockchain.FindUnspentTransactions` 完全相同，但现在它返回一 `TransactionID` → `TransactionOutputs` 的映射map。

现在，UTXO集可以用来发送货币：

```
func (u UTXOSet) FindSpendableOutputs(pubkeyHash []byte, amount int) (int, m
ap[string][]int) {
    unspentOutputs := make(map[string][]int)
    accumulated := 0
    db := u.Blockchain.db

    err := db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))
        c := b.Cursor()

        for k, v := c.First(); k != nil; k, v = c.Next() {
            txID := hex.EncodeToString(k)
            outs := DeserializeOutputs(v)
```

```
            for outIdx, out := range outs.Outputs {
                if out.IsLockedWithKey(pubkeyHash) && accumulated < amount {
                    accumulated += out.Value
                    unspentOutputs[txID] = append(unspentOutputs[txID], outI
dx)
                }
            }
        }
    })

    return accumulated, unspentOutputs
}
```

或者检查余额：

```
func (u UTXOSet) FindUTXO(pubKeyHash []byte) []TXOutput {
    var UTXOs []TXOutput
    db := u.Blockchain.db

    err := db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))
        c := b.Cursor()

        for k, v := c.First(); k != nil; k, v = c.Next() {
            outs := DeserializeOutputs(v)

            for _, out := range outs.Outputs {
                if out.IsLockedWithKey(pubKeyHash) {
                    UTXOs = append(UTXOs, out)
                }
            }
        }

        return nil
    })

    return UTXOs
}
```

这些是相应Blockchain方法的简单修改版本。这些Blockchain方法不再需要了。

拥有UTXO集意味着我们的数据（交易）现在被分开到存储区中：实际交易存储在区块链中，未花费的输出存储在UTXO集中。这种分离需要强大的同步机制，因为我们希望UTXO集始终被更新并存储最近交易的输出。但我们不希望每个区块被挖出来时都要重建索引，因为这是我们想要避免频繁的区块链访问。因此，我们需要一个更新UTXO集合的机制：

```
func (u UTXOSet) Update(block *Block) {
    db := u.Blockchain.db

    err := db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(utxoBucket))

        for _, tx := range block.Transactions {
            if tx.IsCoinbase() == false {
                for _, vin := range tx.Vin {
                    updatedOuts := TXOutputs{}
                    outsBytes := b.Get(vin.Txid)
                    outs := DeserializeOutputs(outsBytes)

                    for outIdx, out := range outs.Outputs {
                        if outIdx != vin.Vout {
                            updatedOuts.Outputs = append(updatedOuts.Outputs
, out)
                        }
                    }

                    if len(updatedOuts.Outputs) == 0 {
                        err := b.Delete(vin.Txid)
                    } else {
                        err := b.Put(vin.Txid, updatedOuts.Serialize())
                    }

                }
            }

            newOutputs := TXOutputs{}
            for _, out := range tx.Vout {
                newOutputs.Outputs = append(newOutputs.Outputs, out)
            }

            err := b.Put(tx.ID, newOutputs.Serialize())
        }
    })
}
```

该方法看起来很多，但它做的很简单。当一个新区块被挖掘时，UTXO集应该被更新。更新意味着移除已花费的输出并增加新被挖掘到的交易的未花费输出。如果一个交易的输出被删除，不包含更多的输出，它也会被删除。非常简单！

现在让我们在需要的地方使用UTXO集合：

```
func (cli *CLI) createBlockchain(address string) {
```

```
    ...
    bc := CreateBlockchain(address)
    defer bc.db.Close()

    UTXOSet := UTXOSet{bc}
    UTXOSet.Reindex()
    ...
}
```

当一个新的区块链被创建以后，就会立刻进行重建索引。现在，这里是唯一使用到 `Reindex` 的地方，虽然看起来是多余的。在区块链开始时，只有一个区块有一个交易，并且 `Update` 可以用来代替。但是,我们在未来可能需要重建索引的机制。

```
func (cli *CLI) send(from, to string, amount int) {
    ...
    newBlock := bc.MineBlock(txs)
    UTXOSet.Update(newBlock)
}
```

UTXO 集合会在一个新的区块被挖掘后进行更新。

让我们来检查它工作是否正常

```
$ blockchain_go createblockchain -address 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1
00000086a725e18ed7e9e06f1051651a4fc46a315a9d298e59e57aeacbe0bf73

Done!

$ blockchain_go send -from 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1 -to 12DkLzLQ4B
3gnQt62EPRJGZ38n3zF4Hzt5 -amount 6
0000001f75cb3a5033aeecbf6a8d378e15b25d026fb0a665c7721a5bb0faa21b

Success!

$ blockchain_go send -from 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1 -to 12ncZhA5mF
TTnTmHq1aTPYBri4jAK8TacL -amount 4
000000cc51e665d53c78af5e65774a72fc7b864140a8224bf4e7709d8e0fa433

Success!

$ blockchain_go getbalance -address 1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1
Balance of '1F4MbuqjcuJGymjcuYQMUVYB37AWKkSLif': 20

$ blockchain_go getbalance -address 12DkLzLQ4B3gnQt62EPRJGZ38n3zF4Hzt5
Balance of '1XWu6nitBWe6J6v6MXmd5rhdP7dZsExbx': 6
```

```
$ blockchain_go getbalance -address 12ncZhA5mFTTnTmHq1aTPYBri4jAK8TacL
Balance of '13UASQpCR8Nr41PojH8Bz4K6cmTCqweskL': 4
```

太好了！ `1JnMDSqVoHi4TEFXNw5wJ8skPsPf4LHkQ1` 这个地址收到了3次奖励：

1. 一次来自挖掘出创世区块。
2. 一次来自挖掘出区块 `0000001f75cb3a5033aeecbf6a8d378e15b25d026fb0a665c7721a5bb0faa21b` 。
3. 一次来自挖掘出区块 `000000cc51e665d53c78af5e65774a72fc7b864140a8224bf4e7709d8e0fa433` 。

4、Merkle树

还有一个我想在这篇文章中讨论的优化机制。

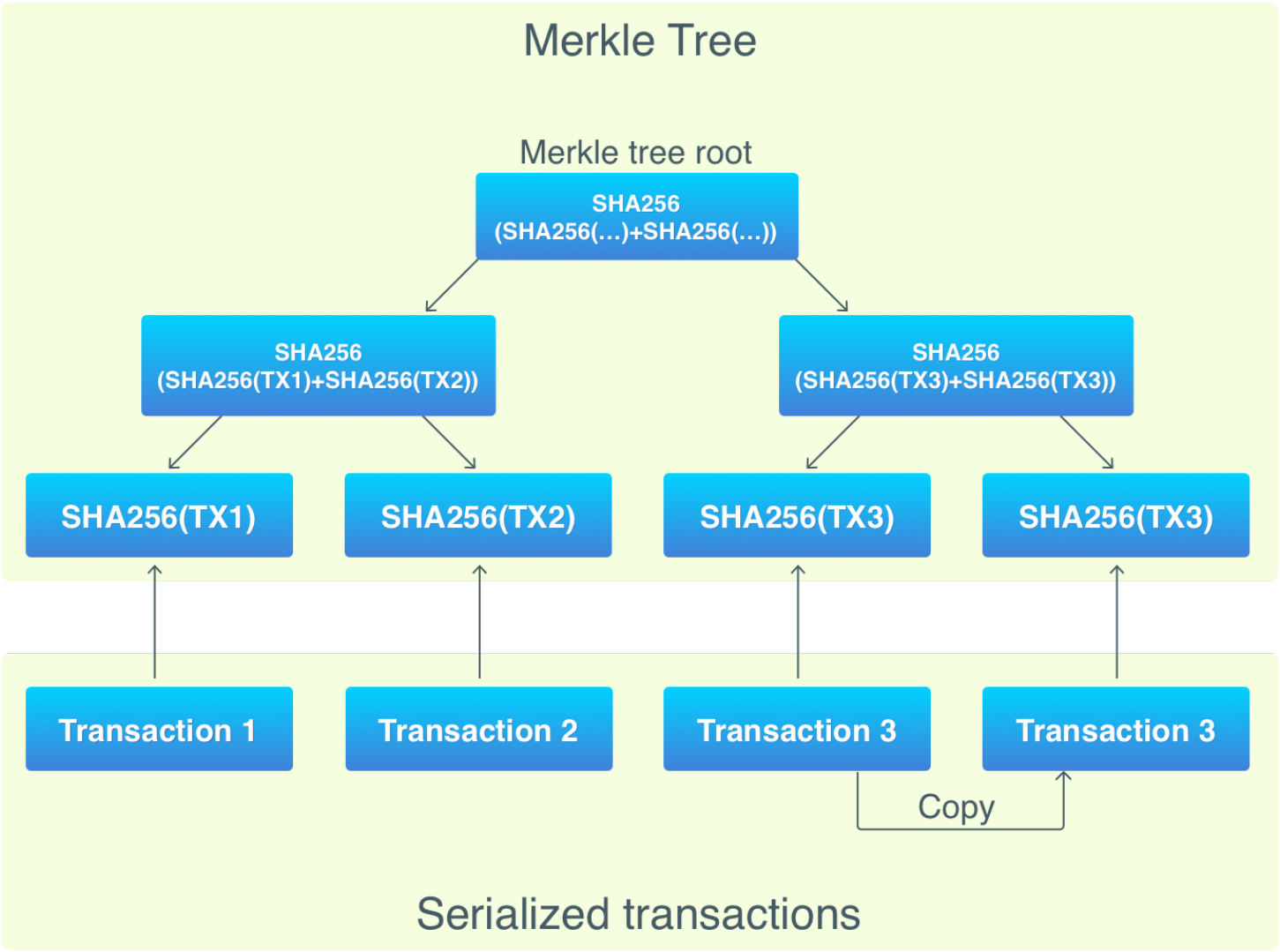
如上所述，完整的比特币数据库（即区块链）需要超过140 Gb的磁盘空间。由于比特币的去中心化特性，网络中的每个节点都必须是独立的，即每个节点都必须存储完整的区块链副本。随着越来越多的人开始使用比特币，这条规则变得更加难以遵循：每个人都不可能运行一个完整的节点。另外，由于节点是网络的完整参与者，他们有责任：他们必须验证交易和区块。此外，需要有一定的互联网流量来和其他节点交互，下载新的区块。

在中本聪公布的比特币原始论文中，针对这种情况有一种解决方案：简单支付验证(Simplified Payment Verification, SPV)。SPV 是一个比特币的轻量节点，这种节点不会下载整个区块链， `也不会验证区块和交易` 。而是找到区块中的交易(以验证支付)然后链接到完整节点以检索必要的数据。这种机制允许多个轻量钱包节点只运行一个完整的节点。

为了使SPV成为可能，应该有一种方法来检查一个区块是否包含某个交易而不下载整个区块。Merkle树就是在这里发挥作用的。

比特币使用Merkle树来获取交易哈希，然后将其存储在区块头中，并会用于工作量证明系统。到目前为止，我们只是将一个区块里面的每笔交易哈希连接了起来，其中应用了 `SHA-256` 算法。这也是一种获取区块交易唯一表示的一种好方法，但是它并没有 Merkle 树的优势。

我们来看一下Merkle树：



Merkle树是为每个区块构建的，它以叶子（树的底部）开始，叶子是一个交易哈希（比特币使用两次 `SHA256` 哈希）。叶子的数量必须是偶数，但不是每个区块都包含偶数个交易。如果存在奇数个交易，则最后一个交易将被复制（只是在Merkle树中，而不是在区块链中！）。

从底部开始，叶子节点会成对分组，它们的哈希会连接在一起，并从连接的哈希中获取新的哈希。新哈希形成新的树节点。重复这个过程，直到只有一个节点被称为树的根节点。然后将根哈希用作交易的唯一表示，保存在区块头中，并用于工作量证明系统中。

Merkle树的好处是节点可以在不下载整个区块的情况下，验证是否包含某笔交易。为此，只需要一个交易哈希，一个Merkle树的根哈希和一个Merkle路径。

最后，让我们编写代码：

```
type MerkleTree struct {
    RootNode *MerkleNode
}

type MerkleNode struct {
    Left  *MerkleNode
    Right *MerkleNode
```

```
    Data  []byte
}
```

我们从结构体开始。每个 `MerkleNode` 都有数据，并链接到它们的分支。`MerkleTree` 实际上是连接到下一个节点的根节点，它们又链接到更远的节点等等。

我们首先来创建一个新的节点：

```
func NewMerkleNode(left, right *MerkleNode, data []byte) *MerkleNode {
    mNode := MerkleNode{}

    if left == nil && right == nil {
        hash := sha256.Sum256(data)
        mNode.Data = hash[:]
    } else {
        prevHashes := append(left.Data, right.Data...)
        hash := sha256.Sum256(prevHashes)
        mNode.Data = hash[:]
    }

    mNode.Left = left
    mNode.Right = right

    return &mNode
}
```

每个节点都包含一些数据。当一个节点是叶子节点时，数据从外界传递（在我们的例子中是一个序列化的交易）。当一个节点链接到其他节点时，它会将其他节点的数据取过来，连接后再哈希。

```
func NewMerkleTree(data [][]byte) *MerkleTree {
    var nodes []MerkleNode

    if len(data)%2 != 0 {
        data = append(data, data[len(data)-1])
    }

    for _, datum := range data {
        node := NewMerkleNode(nil, nil, datum)
        nodes = append(nodes, *node)
    }

    for i := 0; i < len(data)/2; i++ {
        var newLevel []MerkleNode
```

```
        for j := 0; j < len(nodes); j += 2 {
            node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
            newLevel = append(newLevel, *node)
        }

        nodes = newLevel
    }

    mTree := MerkleTree{&nodes[0]}

    return &mTree
}
```

当一个树被创建出，首先保证叶节点必须为偶数个。然后，`数据`（也就是被序列化的交易数组）被转换为树的叶子节点，并从这些叶子节点中生长成一棵树。

现在，让我们修改一下`Block.HashTransactions`，在工作量证明系统中使用它来获取交易哈希：

```
func (b *Block) HashTransactions() []byte {
    var transactions [][]byte

    for _, tx := range b.Transactions {
        transactions = append(transactions, tx.Serialize())
    }
    mTree := NewMerkleTree(transactions)

    return mTree.RootNode.Data
}
```

首先，交易被序列化了(使用`encoding/gob`)，然后它们被用来构建一个Merkle树。树的根节点将会被用作区块交易的独特 ID。

5、P2PKH

还有一件事，我想更详细地讨论。

正如你记得的，在比特币中有脚本（Script）编程语言，它用于锁定交易输出; 并且交易输入提供数据来解锁输出。语言很简单，这种语言的代码只是一系列数据和操作符。考虑这个例子：

```
5 2 OP_ADD 7 OP_EQUAL
```

`5`，`2`，和`7`是数据，`OP_ADD`和`OP_EQUAL`是操作符。脚本代码从左到右执行：每个数据都

被放入堆栈，下一个操作符被应用到顶层堆栈元素。脚本的堆栈只是一个简单的FILO（First Input Last Output）存储器：堆栈中的第一个元素是最后一个元素，每个元素都放在前一个元素上。

让我们将上述脚本的执行分解成以下几个步骤：

1. 栈：空。脚本：5 2 OP_ADD 7 OP_EQUAL
2. 栈：5。脚本：2 OP_ADD 7 OP_EQUAL
3. 栈：5 2。脚本：OP_ADD 7 OP_EQUAL
4. 栈：7。脚本：7 OP_EQUAL
5. 栈：7 7。脚本：OP_EQUAL
6. 栈：true。脚本：空

OP_ADD从堆栈中获取两个元素，求和，并将总和推入堆栈。OP_EQUAL从堆栈中获取两个元素并对它们进行比较：如果它们相等，则将它推true入堆栈; 否则它推入false。脚本执行的结果是顶层堆栈元素的值：在我们的例子中true，这意味着脚本成功完成。

现在让我们看一下比特币用于执行支付的脚本：

```
<signature> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

这个脚本被称为Pay to Public Key Hash（P2PKH），这是比特币中最常用的脚本。它从字面上支付公共密钥哈希值，即用某个公钥锁定硬币。这是`比特币支付的核心`：没有账户，没有资金在它们之间转移; 只有一个脚本检查提供的签名和公钥是否正确。

该脚本实际上存储在两部分中：

1. 第一部分， 存储在输入ScriptSig字段中。
2. 第二部分，OP_DUP OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG存储在输出中ScriptPubKey。

因此，它是定义解锁逻辑的输出，并且它是提供数据以解锁输出的输入。让我们执行脚本：

1. 堆栈：空
   脚本： OP_DUP OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG

2. 堆栈：`<signature>`
   脚本： OP_DUP OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG

3. 堆栈：`<signature> <pubKey>`
   脚本：OP_DUP OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG

4. 堆栈：`<signature> <pubKey> <pubKey>`
   脚本：OP_HASH160 OP_EQUALVERIFY OP_CHECKSIG

5. 堆栈：`<signature> <pubKey> <pubKeyHash>`
   脚本： OP_EQUALVERIFY OP_CHECKSIG

6. 堆栈：`<signature> <pubKey> <pubKeyHash> <pubKeyHash>`
   脚本：OP_EQUALVERIFY OP_CHECKSIG

7. 堆栈：`<signature> <pubKey>`
   脚本：OP_CHECKSIG

8. 堆栈：true或false。脚本：空。

`OP_DUP` 复制顶层堆栈元素。 `OP_HASH160` 取顶部堆栈元素并将用 `RIPEMD160` 哈希计算; 结果被推回到堆栈。 `OP_EQUALVERIFY` 比较两个顶层堆栈元素，如果它们不相等，则会中断脚本。 `OP_CHECKSIG` 通过哈希交易并使用 `<signature>` 和验证交易的签名 `<pubKey>` 。后者的运算符相当复杂：它做了一个简单的交易副本, 对它哈希(因为这是被签名的交易哈希), 然后用提供的 `signature` 和 `pubKey` 验证签名.

拥有这样的脚本语言可以让比特币成为一个智能合约平台：除了转移到一个单一的密钥之外，该语言还可以实现其他支付方案。

6、总结

就是这样！我们已经实现了几乎所有基于区块链的加密货币的关键功能。我们拥有区块链，地址，采矿和交易。但还有一件事让所有这些机制生机勃勃，并使比特币成为全球系统：共识。在下一篇文章中，我们将开始实现区块链的"去中心化"部分。敬请关注！

链接：

1. 完整的源代码
2. UTXO集
3. Merkle树
4. 脚本
5. “Ultraprune”比特币核心提交
6. UTXO集统计
7. 智能合约和比特币
8. 为何每个比特币用户都应该了解"SPV安全"