用Go构建区块链——4.交易1

本篇是"用Go构建区块链"系列的第四篇，主要对原文进行翻译。对应原文如下：

Building Blockchain in Go. Part 4: Transactions 1

1、介绍

交易是比特币的核心，区块链的唯一目的是以安全可靠的方式存储交易，所以没有人可以在创建后修改它们。今天我们开始实现交易。但是因为这是一个相当大的话题，所以我将它分成两个部分：在这一部分中，我们将实现交易的通用机制，第二部分我们将详细讨论。

而且，由于代码变化很大，因此在这里对它们解释是没有意义的。你可以在这里看到所有的变化。

2、没有勺子（"There is no spoon"）

（译者注：There is no spoon，字面意思“没有汤勺”，是《黑客帝国》的一句台词。这里应该可以翻译为类似"无限力量或者魔法无限"的意思吧）

如果您曾经开发过Web应用程序，为了实现支付系统，您可能会在DB中创建这些表格：`accounts` 和 `transactions` 。一个账户会存储关于用户的信息，包括他们的个人信息和余额，而一个交易会存储关于从一个账户向另一个账户转移资金的信息。在比特币中，付款是以完全不同的方式实现的。这里：

1. 没有帐号;
2. 没有余额;
3. 没有地址;
4. 没有货币;
5. 没有发送者和接收者。

由于区块链是一个公开和开放的数据库，我们不希望存储有关钱包所有者的敏感信息。货币不会存在帐户中。交易不会将资金从一个地址转移到另一个地址。没有可保存帐户余额的字段或属性。只有交易。但是交易内容是什么?

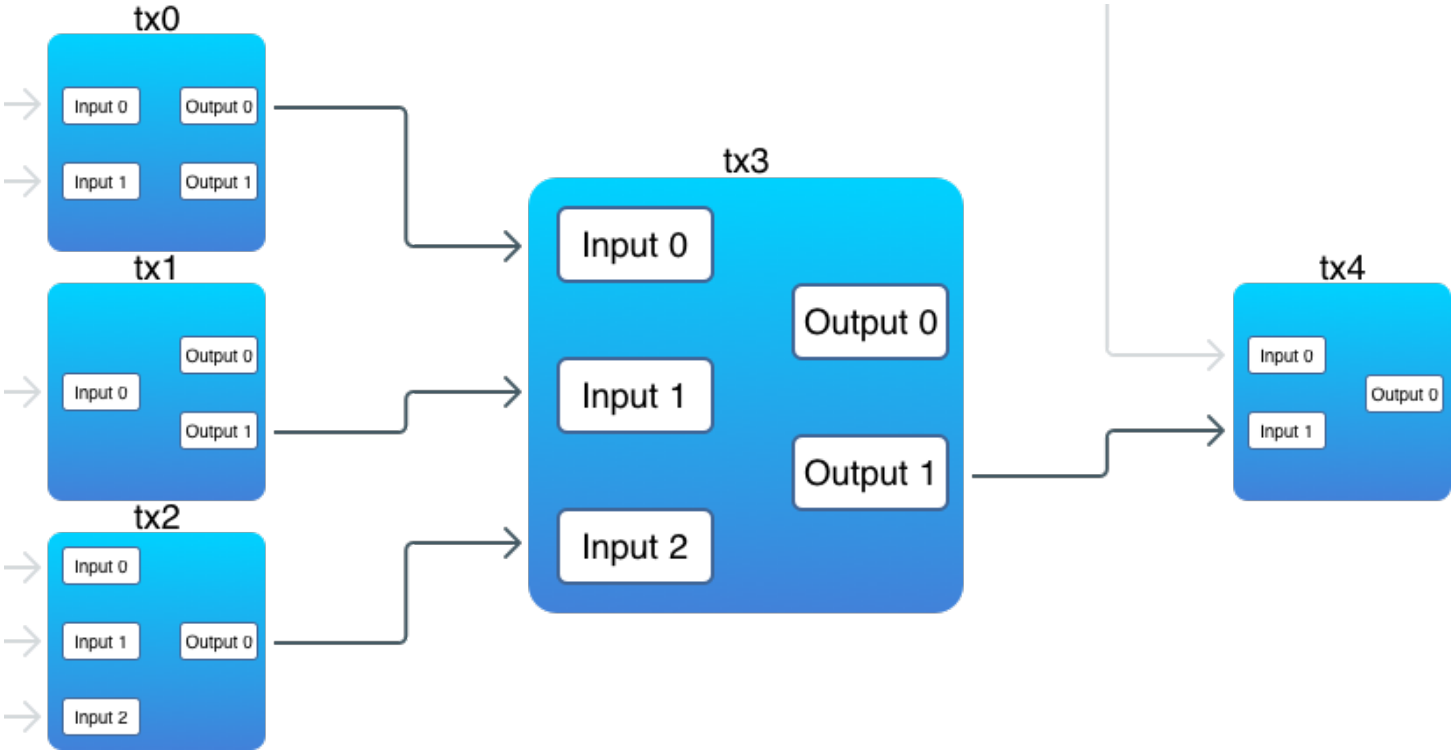
3、比特币交易

交易是输入和输出的组合：

```
type Transaction struct {
    ID   []byte
```

```
    Vin  []TXInput
    Vout []TXOutput
}
```

一笔新的交易的输入引用前一笔交易的输出（这里有个例外，我们将在后面讨论）。输出是货币实际存储的地方。下图演示了交易之间的的关系：



注意：

1. 有一些输出没有关联到输入；
2. 在一笔交易中，输入可以引用之前多个交易的输出；
3. 一个输入必须引用一个输出。

在整篇文章中，我们将使用诸如"钱"，"货币(coins)"，"花费"，"发送"，"账户"等词语。但比特币中没有这样的概念。交易只是用脚本锁定值，只能由锁定它的人解锁。

4、交易输出

首先从输出开始：

```
type TXOutput struct {
    Value        int
    ScriptPubKey string
}
```

事实上，这是存储"货币(coins)"的输出(注意一下上面的 `Value` 字段)。而存储意味着用一个谜题

锁定它们，这是存储在 `ScriptPubKey` 。在内部，比特币使用称为脚本的脚本语言，用于定义输出锁定和解锁逻辑。这种语言很原始（这是故意的，以避免可能的黑客和滥用），但我们不会详细讨论它。你可以在这里找到它的详细解释。

> 在比特币中，value字段存储satoshis的数量，而不是BTC的数量。一个satoshis是一亿分之一个比特币（0.00000001 BTC）的，因此，这是比特币的最小货币单位（如百分比）。

由于我们没有实现地址，现在我们将避免整个脚本相关的逻辑。 `ScriptPubKey` 将存储任意字符串（用户定义的钱包地址）。

> 顺便说一句，拥有这样的脚本语言意味着比特币也可以用作智能合约平台。

输出的一个重要的事情是它们是 `不可分割` 的，这意味着你不能引用它的一部分。在新的交易中引用输出时，它将作为一个整体进行使用。如果其值大于所需要的值，则会生成更改并将其发送回发送者。这与真实世界的情况类似，例如，您支付5美元的钞票用于花费1美元，并且找回4美元的东西。

5、交易输入

这里是输入：

```
type TXInput struct {
    Txid      []byte
    Vout      int
    ScriptSig string
}
```

如上所述，输入引用前一个输出： `Txid` 存储此类交易的ID， `Vout` 在交易中存储输出的索引。 `ScriptSig` 是一个提供数据用于在输出中使用的脚本 `ScriptPubKey` 。如果数据是正确的，输出可以被解锁，这个值也可以被用来生成新的输出；如果不正确，输出就不能被输入所引用。这是保证用户不能花钱属于其他人的货币的机制。

同样，由于我们还没有实现地址， `ScriptSig` 因此将只存储任意用户定义的钱包地址。我们将在下一篇文章中实现公钥和签名检查。

我们总结一下。输出是储存"货币"的地方。每个输出都带有一个解锁脚本，它决定了解锁输出的逻辑。每个新交易都必须至少有一个输入和输出。输入引用前一个交易的输出，并提供 `ScriptSig` 输出的解锁脚本中使用的数据（字段），以解除锁定并使用其值创建新的输出。

但哪个先出现：输入还是输出?

6、鸡蛋

在比特币中，先有蛋，才有鸡。输入引用输出的逻辑是典型的 "鸡还是鸡蛋" 的问题：输入产生输出，输出使得输入成为可能。在比特币中，输出先于输入。

当矿工开始挖矿时，它会添加一个 `coinbase交易` 。coinbase交易是一种特殊类型的交易，不需要以前存在的输出。它无处不在地创造输出（即"货币"）。没有鸡的鸡蛋。这是矿工挖到新区块的奖励。

如您所知，区块链开始处有创世区块。这个区块在区块链中产生了第一个输出。由于没有以前的交易并且没有这样的输出，因此不需要先前的输出。

我们来创建一个coinbase交易：

```
func NewCoinbaseTX(to, data string) *Transaction {
    if data == "" {
        data = fmt.Sprintf("Reward to '%s'", to)
    }

    txin := TXInput{[]byte{}, -1, data}
    txout := TXOutput{subsidy, to}
    tx := Transaction{nil, []TXInput{txin}, []TXOutput{txout}}
    tx.SetID()

    return &tx
}
```

一个coinbase交易只有一个输入。在我们的实现中，它 `Txid` 是空的， `Vout` 等于-1。而且，coinbase交易不会存储脚本 `ScriptSig` 。相反，任意数据存储在那里。

> 在比特币中，第一个coinbase交易包含以下信息："The Times 03/Jan/2009 Chancellor on brink of second bailout for banks"。你可以自己看。

`subsidy` 是奖励的金额。在比特币中，这个数字没有存储在任何地方，只根据块的总数进行计算：块的数量除以 `210000` 。挖掘创世区块产生50 BTC，每 `210000` 块奖励减半。在我们的实现中，我们会将奖励作为常量存储（至少现在是😉）。

7、在区块链中存储交易

从现在开始，每个区块都必须存储至少一个交易，并且不可能在没有交易的情况下挖掘区块。这意味着我们应该删除并存储交易的 `Data` 字段 `Block` ：

```
type Block struct {
    Timestamp     int64
    Transactions  []*Transaction
```

```
    PrevBlockHash []byte
    Hash          []byte
    Nonce         int
}
```

`NewBlock` 并且 `NewGenesisBlock` 也必须相应地改变：

```
func NewBlock(transactions []*Transaction, prevBlockHash []byte) *Block {
    block := &Block{time.Now().Unix(), transactions, prevBlockHash, []byte{}
, 0}
    ...
}

func NewGenesisBlock(coinbase *Transaction) *Block {
    return NewBlock([]*Transaction{coinbase}, []byte{})
}
```

接下来要改变的是创建一个新的区块链：

```
func CreateBlockchain(address string) *Blockchain {
    ...
    err = db.Update(func(tx *bolt.Tx) error {
        cbtx := NewCoinbaseTX(address, genesisCoinbaseData)
        genesis := NewGenesisBlock(cbtx)

        b, err := tx.CreateBucket([]byte(blocksBucket))
        err = b.Put(genesis.Hash, genesis.Serialize())
        ...
    })
    ...
}
```

现在，该函数将获得一个地址，该地址将获得挖掘创世区区块的奖励。

8、工作量证明（Proof-of-Work）

工作量证明算法必须考虑存储在区块中的交易，以保证区块链作为交易存储的一致性和可靠性。所以现在我们必须修改 `ProofOfWork.prepareData` 方法：

```
func (pow *ProofOfWork) prepareData(nonce int) []byte {
    data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
```

```
            pow.block.HashTransactions(), // This line was changed
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
    )

    return data
}
```

不像之前使用 `pow.block.Data` ，我们现在用 `pow.block.HashTransactions()` ：

```
func (b *Block) HashTransactions() []byte {
    var txHashes [][]byte
    var txHash [32]byte

    for _, tx := range b.Transactions {
        txHashes = append(txHashes, tx.ID)
    }
    txHash = sha256.Sum256(bytes.Join(txHashes, []byte{}))

    return txHash[:]
}
```

再次，我们使用哈希计算作为提供数据的唯一表示的机制。我们希望区块中的所有交易都由独一无二的哈希作为唯一标识。为了达到这个目的，我们得到每个交易的哈希值，连接它们，并获得连接组合的哈希值。

> 比特币使用更复杂的技术：它将所有包含在区块中的交易表示为Merkle树，并在工作量证明系统中使用该树的根哈希。这种方法允许快速检查块是否包含某个交易，只需要有根哈希，而不用下载所有交易。

让我们来检查一下目前为止的一切是否正确：

```
$ blockchain_go createblockchain -address Ivan
00000093450837f8b52b78c25f8163bb6137caf43ff4d9a01d1b731fa8ddcc8a

Done!
```

好！我们收到了第一笔挖矿奖励。但我们如何检查余额?

9、未花费交易输出（Unspent Transaction Outputs）

我们需要找到所有未花费的交易输出（UTXO）。未花费意味着这些输出在任何输入中都未被引用。在上图中，未花费的输出是：

1. tx0, output 1;
2. tx1, output 0;
3. tx3, output 0;
4. tx4, output 0。

当然，当我们检查余额时，我们不需要所有这些，只要找能被我们的 key 所解锁的那些就可以了（当前我们没有实现密钥，暂时会用用户自定义地址来代替）。首先，我们来定义输入和输出上的锁定 - 解锁方法：

```
func (in *TXInput) CanUnlockOutputWith(unlockingData string) bool {
    return in.ScriptSig == unlockingData
}

func (out *TXOutput) CanBeUnlockedWith(unlockingData string) bool {
    return out.ScriptPubKey == unlockingData
}
```

这里我们只是比较脚本字段 `unlockingData` 。在后续文章中，我们基于私钥实现了地址以后，会对这部分进行改进。

下一步 - 查找包含未花费输出的交易 - 相当困难：

```
func (bc *Blockchain) FindUnspentTransactions(address string) []Transaction
{
  var unspentTXs []Transaction
  spentTXOs := make(map[string][]int)
  bci := bc.Iterator()

  for {
    block := bci.Next()

    for _, tx := range block.Transactions {
      txID := hex.EncodeToString(tx.ID)

    Outputs:
      for outIdx, out := range tx.Vout {
        // Was the output spent?
        if spentTXOs[txID] != nil {
          for _, spentOut := range spentTXOs[txID] {
            if spentOut == outIdx {
              continue Outputs
```

```
            }
          }
        }

        if out.CanBeUnlockedWith(address) {
          unspentTXs = append(unspentTXs, *tx)
        }
      }

      if tx.IsCoinbase() == false {
        for _, in := range tx.Vin {
          if in.CanUnlockOutputWith(address) {
            inTxID := hex.EncodeToString(in.Txid)
            spentTXOs[inTxID] = append(spentTXOs[inTxID], in.Vout)
          }
        }
      }
    }

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

  return unspentTXs
}
```

由于交易存储在区块中，因此我们必须检查区块链中的每个区块。我们从输出开始：

```
if out.CanBeUnlockedWith(address) {
    unspentTXs = append(unspentTXs, tx)
}
```

如果一笔输出被我们所搜索的未花费交易输出的相同地址锁住了，那么这就是我们想要的输出。但在使用之前，我们需要检查一个输出是否已经在输入中被引用：

```
if spentTXOs[txID] != nil {
    for _, spentOut := range spentTXOs[txID] {
        if spentOut == outIdx {
            continue Outputs
        }
    }
}
```

我们跳过那些在输入中引用的值（它们的值被移到其他输出中，因此我们无法计算它们）。在检查输出之后，我们收集所有可以解锁输出的输入，并锁定提供的地址（这不适用于coinbase交易，因为它们不解锁输出）：

```
func (bc *Blockchain) FindUTXO(address string) []TXOutput {
	var UTXOs []TXOutput
	unspentTransactions := bc.FindUnspentTransactions(address)

	for _, tx := range unspentTransactions {
		for _, out := range tx.Vout {
			if out.CanBeUnlockedWith(address) {
				UTXOs = append(UTXOs, out)
			}
		}
	}

	return UTXOs
}
```

该函数返回一个包含未使用输出的交易列表。为了计算余额，我们需要一个函数来处理交易并仅返回输出：

```
func (bc *Blockchain) FindUTXO(address string) []TXOutput {
	var UTXOs []TXOutput
	unspentTransactions := bc.FindUnspentTransactions(address)

	for _, tx := range unspentTransactions {
		for _, out := range tx.Vout {
			if out.CanBeUnlockedWith(address) {
				UTXOs = append(UTXOs, out)
			}
		}
	}

	return UTXOs
}
```

就这样！现在，我们可以实现 `getbalance` 命令了：

```
func (cli *CLI) getBalance(address string) {
    bc := NewBlockchain(address)
    defer bc.db.Close()

    balance := 0
```

```
    UTXOs := bc.FindUTXO(address)

    for _, out := range UTXOs {
        balance += out.Value
    }

    fmt.Printf("Balance of '%s': %d\n", address, balance)
}
```

账户余额是由账户地址锁定的所有未花费的交易输出值的总和。

在挖掘了创始区块之后，来检查一下我们的余额：

```
$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 10
```

这是我们的第一笔钱！

10、发送货币

现在，我们要发送一些货币给别人。为此，我们需要创建一个新的交易，将它放在一个区块中，然后挖掘区块。到目前为止，我们只实现了coinbase交易（这是一种特殊类型的交易），现在我们需要一个普通交易：

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain) *Transa
ction {
    var inputs []TXInput
    var outputs []TXOutput

    acc, validOutputs := bc.FindSpendableOutputs(from, amount)

    if acc < amount {
        log.Panic("ERROR: Not enough funds")
    }

    // Build a list of inputs
    for txid, outs := range validOutputs {
        txID, err := hex.DecodeString(txid)

        for _, out := range outs {
            input := TXInput{txID, out, from}
            inputs = append(inputs, input)
        }
    }
```

```
    // Build a list of outputs
    outputs = append(outputs, TXOutput{amount, to})
    if acc > amount {
        outputs = append(outputs, TXOutput{acc - amount, from}) // a change
    }

    tx := Transaction{nil, inputs, outputs}
    tx.SetID()

    return &tx
}
```

在创建新的输出之前，我们首先必须找到所的有未花费输出并确保它们存储足够的值。这是 `FindSpendableOutputs` 方法做的事情了。之后，为每个找到的输出创建一个引用它的输入。接下来，我们创建两个输出：

1. 一个与接收者地址锁定的。这是货币实际转移的另外一个地址。
2. 一个与发送者地址锁定在一起。这是找零。只有在未花费的输出值比新交易所需的值更多时才会创建。记住：输出是 `不可见的` 。

`FindSpendableOutputs` 方法基于 `FindUnspentTransactions` 我们之前定义的方法：

```
func (bc *Blockchain) FindSpendableOutputs(address string, amount int) (int,
 map[string][]int) {
    unspentOutputs := make(map[string][]int)
    unspentTXs := bc.FindUnspentTransactions(address)
    accumulated := 0

Work:
    for _, tx := range unspentTXs {
        txID := hex.EncodeToString(tx.ID)

        for outIdx, out := range tx.Vout {
            if out.CanBeUnlockedWith(address) && accumulated < amount {
                accumulated += out.Value
                unspentOutputs[txID] = append(unspentOutputs[txID], outIdx)

                if accumulated >= amount {
                    break Work
                }
            }
        }
    }

    return accumulated, unspentOutputs
```

```
}
```

该方法迭代所有的未花费交易并累积这些值。当累计值大于或等于我们要转移的金额时，它停止并返回按交易ID分组的累计值和输出索引。我们不想花更多的钱。

现在我们可以修改该Blockchain.MineBlock方法：

```
func (bc *Blockchain) MineBlock(transactions []*Transaction) {
    ...
    newBlock := NewBlock(transactions, lastHash)
    ...
}
```

最后，让我们实现 `send` 命令：

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

    tx := NewUTXOTransaction(from, to, amount, bc)
    bc.MineBlock([]*Transaction{tx})
    fmt.Println("Success!")
}
```

发送货币意味着创建一个交易并通过挖掘一个区块将其添加到区块链。但比特币不会立即做到这一点（就像我们一样）。相反，它将所有新交易放入内存池中，并且当矿工准备去挖新的区块时，它将从内存池中获取所有交易并创建候选区块。交易只有在包含它们的区块被挖掘并添加到区块链时才会被确认。

让我们来看看发送货币的工作原理：

```
$ blockchain_go send -from Ivan -to Pedro -amount 6
00000001b56d60f86f72ab2a59fadb197d767b97d4873732be505e0a65cc1e37

Success!

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 4

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 6
```

太好了！现在，让我们创建更多的交易并确保从多个输出发送正常工作：

```
$ blockchain_go send -from Pedro -to Helen -amount 2
00000099938725eb2c7730844b3cd40209d46bce2c2af9d87c2b7611fe9d5bdf

Success!

$ blockchain_go send -from Ivan -to Helen -amount 2
000000a2edf94334b1d94f98d22d7e4c973261660397dc7340464f7959a7a9aa

Success!
```

现在，海伦（Helen）的钱币被锁定在两个输出上：一个来自佩德罗（Pedro），另一个来自伊万（Ivan）。让我们把它们发给别人：

```
$ blockchain_go send -from Helen -to Rachel -amount 3
000000c58136cffa669e767b8f881d16e2ede3974d71df43058baaf8c069f1a0

Success!

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 2

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 4

$ blockchain_go getbalance -address Helen
Balance of 'Helen': 1

$ blockchain_go getbalance -address Rachel
Balance of 'Rachel': 3
```

看起来很好！现在我们来测试一个失败：

```
$ blockchain_go send -from Pedro -to Ivan -amount 5
panic: ERROR: Not enough funds

$ blockchain_go getbalance -address Pedro
Balance of 'Pedro': 4

$ blockchain_go getbalance -address Ivan
Balance of 'Ivan': 2
```

11、总结

唷！这并不容易，但我们现在有交易了！尽管缺少像比特币一样的加密货币的一些关键特征：

1. 地址。我们还没有真实的，基于私钥的地址。
2. 奖励。挖矿绝对没有利润！
3. UTXO集。达到账户余额需要扫描整个区块链，当区块数量很多时可能需要很长时间。此外，如果我们想验证之后的交易，可能需要很长时间。UTXO集旨在解决这些问题并快速处理交易。
4. 内存池。这是交易在打包成块之前存储的地方。在我们目前的实现中，一个区块只包含一个交易，而且效率很低。

链接：

1. 完整的源代码
2. 交易
3. Merkle树
4. Coinbase