用Go构建区块链——7.网络

本篇是"用Go构建区块链"系列的最后一篇，主要对原文进行翻译。对应原文如下：

Building Blockchain in Go. Part 7: Network

1、介绍

到目前为止，我们已经构建了一个包含所有关键功能的区块链：匿名，安全和随机生成的地址; 区块链数据存储; 工作量证明系统; 以可靠的方式来存储交易。虽然这些功能至关重要，但这还不够。什么让这些功能真正发挥作用，以及使加密货币成为可能的因素是网络。只是在一台计算机上运行这种区块链实现有什么用处？当只有一个用户时，那些基于密码学的功能有什么好处？是网络使得这些机制可以工作起来，而且变得有用。

您可以将这些区块链功能视为规则，类似于人们想要共同生存和发展时所制定的规则。一种社会规则。区块链网络是遵循相同规则的程序社区，正式遵循这种规则使得社区得以存活。同样，当人们拥有相同的想法时，他们会变得更强大，并可以共同创造美好的生活。如果有人遵循不同的规则，他们将生活在一个单独的社会（州，公社等）。同样，如果区块链节点遵循不同的规则，它们将形成一个单独的网络。

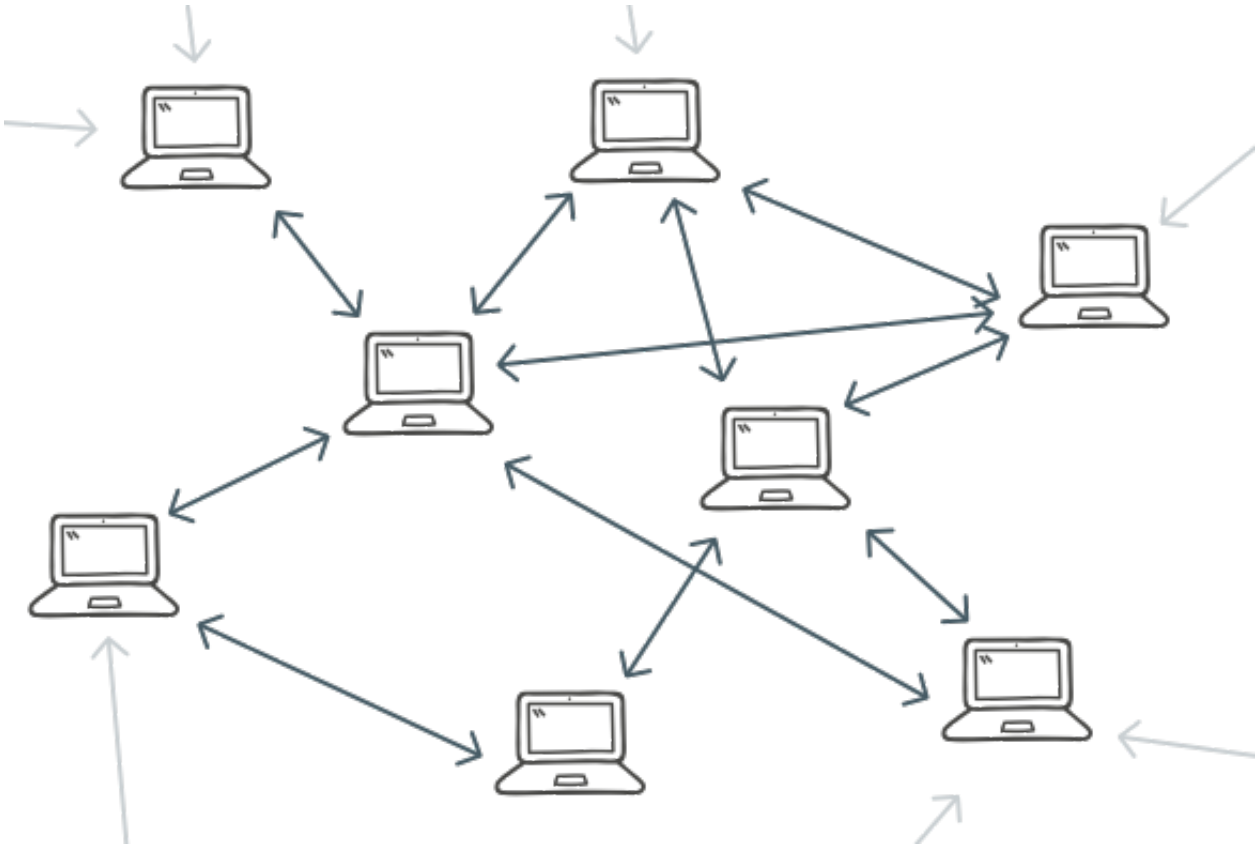
这非常重要：如果没有网络，没有大多数节点共享相同的规则，这些规则是无用的！

> 免责声明：不幸的是，我没有足够的时间来实现真正的P2P网络原型。在本文中，我将演示一个最常见的场景，涉及不同类型的节点。改善这种情况并使其成为P2P网络对您来说可能是一个很好的挑战和实践！另外，我不能保证除了本文中实现的其他场景以外的其他场景都可以使用。抱歉!
>
> 这部分的介绍有重大的代码更改，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、区块链网络

区块链网络是去中心化的，这意味着没有工作的服务器，客户端也不需要使用服务器来获取或处理数据。区块链网络中有节点，每个节点都是网络的正式成员。点就是一切：它既是客户端又是服务器。记住这一点非常重要，因为它与通常的Web应用程序非常不同。

区块链网络是一种P2P（对等）网络，这意味着节点彼此直接连接。它的拓扑结构是扁平的，因为节点角色没有层次结构。在这里它的示意图：



Business vector created by Dooder - Freepik.com

这种的网络节点更难以实现，因为它们必须执行大量操作。每个节点必须与多个其他节点交互，它必须请求其他节点的状态，将其与自己的状态进行比较，并在过期时更新其状态。

3、节点角色

尽管是全面的，区块链节点可以在网络中扮演不同的角色。它们分别是：

1. 矿工。

   这些节点运行在功能强大或专用的硬件设备（如ASIC）上，其唯一目标是尽快挖掘出新的区块。矿工只能在使用工作证明的区块链中使用，因为采矿实际上意味着解决PoW难题。例如，在证明权益区块链中，不存在挖掘。矿工是区块链中唯一可能使用到工作量证明系统的角色，因为挖矿实际上就是解决PoW的问题。例如在PoS权益证明的区块链中，没有挖矿。

2. 全节点。

   这些节点验证矿工挖出来的区块并验证交易。要做到这一点，他们必须拥有区块链的全部副本。而且，这样的节点执行这种路由操作，就像帮助其他节点发现对方一样。对于网络来说，拥有许多完整节点非常重要，因为由这些节点来做出决定的：它们决定一个区块或一笔交易是否有效。

3. SPV。

SPV代表简单支付验证。这些节点不存储完整的区块链副本，但它们仍然能够验证交易（并不是所有交易，而是一个子集，例如发送到某个特定地址的交易）。一个SPV节点依赖于完整节点来获取数据，并且可能有多个SPV节点连接到一个完整节点。SPV使得钱包应用成为可能：一个不需要下载完整的区块链，但仍然可以验证他们的交易。

4、网络简化

为了实现我们区块链中的网络，我们需要简化一些事情。问题是我们没有多台计算机来模拟具有多个节点的网络。我们可以使用虚拟机或Docker来解决这个问题，但它可能会让一切变得更加困难：您必须解决可能的虚拟机或Docker问题，而我的目标是专注于区块链实现。所以，我们希望在一台机器上运行多个区块链节点，同时我们希望它们拥有不同的地址。为了达到这个目的，我们将使用**端口作为节点标识符**，而不是IP地址。例如，会出现这样的地址节点：**127.0.0.1:3000**，**127.0.0.1:3001**，**127.0.0.1:3002**等。我们叫它为端口节点ID，并使用环境变量 **NODE_ID** 对它们进行设置。因此，您可以打开多个终端窗口，设置不同的**NODE_ID** s并运行不同的节点。

这种方法还需要拥有不同的区块链和钱包文件。现在，他们必须依靠节点ID进行命名，比如**blockchain_3000.db**, **blockchain_30001.db**和**wallet_3000.db**，**wallet_30001.db**等待。

5、实现

那么，当你下载Bitcoin Core并首次运行它时会发生什么？它必须连接到某他节点才能下载最新状态的区块链。考虑到你的计算机不知道所有的或者部分的比特币节点，那么这个节点是什么？

在Bitcoin Core中硬编码一个地址，已经被证实是一个错误：节点可能会受到攻击或关闭，这可能导致新节点无法加入网络。相反，在Bitcoin Core中，硬编码了DNS种子(DNS seeds)。虽然这些不是节点，但是DNS服务器知道一些节点的地址。当你启动一个全新的Bitcoin Core时，它将连接到其中一个种子节点上并获得全节点的列表，然后它将从中下载区块链。

在我们的实现中，虽然还是中心化的。我们会有三个节点：

1. 一个中心节点。这是所有其他节点将连接到的节点，并且这是将在其他节点之间发送数据的节点。
2. 一个矿工节点。这个节点将在内存池中存储新的交易，当有足够的交易时，它会打包挖掘出一个新的区块。
3. 一个钱包节点。这个节点将用于在钱包之间发送币。与SPV节点不同，它将存储完整的区块链副本。

6、场景

本文的目标是实现以下场景：

1. 中心节点创建一个区块链。
2. 其他（钱包）节点连接到它并下载区块链。
3. 另外一个（矿工）节点连接到中心节点并下载区块链。
4. 钱包节点创建一个交易。
5. 矿工节点接收交易并将它保存在其内存池中。
6. 当内存池中有足够的交易时，矿工开始挖掘出新的区块。
7. 当一个新的区块被挖掘出来时，将它发送到中心节点。
8. 钱包节点与中心节点同步。
9. 钱包节点的用户检查他们的支付是否成功。

比特币看起来是这样的情况。即使我们不打算建立一个真正的P2P网络，我们将实现一个真正的，也是最重要的比特币用户场景。

7、版本

节点通过消息的方式进行通信。当一个新节点运行时，它从DNS种子中获得几个节点，并向它们发送**版本（version）**消息，在我们的实现中，看起来就像是这样：

```
type version struct {
    Version    int
    BestHeight int
    AddrFrom   string
}
```

我们只有一个区块链版本，所以该**Version**字段不会保留任何重要信息。**BestHeight**存储区块链中节点的长度。**AddFrom**存储发送者的地址。

接收到**version**消息的节点应该做什么呢？它会用自己的**version**信息回应。这是一种握手：没有彼此事先问候，就不可能有其他互动。但这不仅仅是礼貌：**version**用于寻找更长的区块链。当一个节点收到一条**version**消息时，它会检查本节点的区块链是否比**BestHeight**的值更大。如果不是，节点将请求并下载缺失的区块。

为了接收消息，我们需要一个服务器：

```
var nodeAddress string
var knownNodes = []string{"localhost:3000"}

func StartServer(nodeID, minerAddress string) {
    nodeAddress = fmt.Sprintf("localhost:%s", nodeID)
    miningAddress = minerAddress
    ln, err := net.Listen(protocol, nodeAddress)
    defer ln.Close()
```

```
    bc := NewBlockchain(nodeID)

    if nodeAddress != knownNodes[0] {
        sendVersion(knownNodes[0], bc)
    }

    for {
        conn, err := ln.Accept()
        go handleConnection(conn, bc)
    }
}
```

首先，我们对中心节点的地址进行硬编码：每个节点必须知道从何处开始初始化。**minerAddress**参数指定接收挖矿奖励的地址。这一部分：

```
if nodeAddress != knownNodes[0] {
    sendVersion(knownNodes[0], bc)
}
```

意味着如果当前节点不是中心节点，它必须向中心节点发送**version**消息来确定其区块链是否过时。

```
func sendVersion(addr string, bc *Blockchain) {
    bestHeight := bc.GetBestHeight()
    payload := gobEncode(version{nodeVersion, bestHeight, nodeAddress})

    request := append(commandToBytes("version"), payload...)

    sendData(addr, request)
}
```

我们的消息，在底层次上是字节序列。前12个字节指定命令名称（比如这里的**version**），后面的字节将包含**gob**编码过的消息结构。commandToBytes看起来像这样：

```
func commandToBytes(command string) []byte {
    var bytes [commandLength]byte

    for i, c := range command {
        bytes[i] = byte(c)
    }

    return bytes[:]
```

```
}
```

它创建一个12字节的缓冲区并用命令名填充它，剩下的字节为空。有一个相反的函数：

```
func bytesToCommand(bytes []byte) string {
    var command []byte

    for _, b := range bytes {
        if b != 0x0 {
            command = append(command, b)
        }
    }

    return fmt.Sprintf("%s", command)
}
```

当一个节点接收到一个命令时，它运行**bytesToCommand**提取命令名并用正确的处理程序处理命令体：

```
func handleConnection(conn net.Conn, bc *Blockchain) {
    request, err := ioutil.ReadAll(conn)
    command := bytesToCommand(request[:commandLength])
    fmt.Printf("Received %s command\n", command)

    switch command {
    ...
    case "version":
        handleVersion(request, bc)
    default:
        fmt.Println("Unknown command!")
    }

    conn.Close()
}
```

好了，这就是**version**命令处理函数的样子：

```
func handleVersion(request []byte, bc *Blockchain) {
    var buff bytes.Buffer
    var payload verzion

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)
    err := dec.Decode(&payload)
```

```
    myBestHeight := bc.GetBestHeight()
    foreignerBestHeight := payload.BestHeight

    if myBestHeight < foreignerBestHeight {
        sendGetBlocks(payload.AddrFrom)
    } else if myBestHeight > foreignerBestHeight {
        sendVersion(payload.AddrFrom, bc)
    }

    if !nodeIsKnown(payload.AddrFrom) {
        knownNodes = append(knownNodes, payload.AddrFrom)
    }
}
```

首先，我们需要解码请求并提取有效载荷。这与所有处理器类似，所以我将在后面的代码片段中省略这一部分。

然后一个节点将其**BestHeight**与消息中的一个进行比较。如果节点的区块链更长，它会回复**version**消息; 否则，它会发送**getblocks**消息。

8、getblocks

```
type getblocks struct {
    AddrFrom string
}
```

getblocks意味着"向我展示你拥有的块"（在比特币中，它更复杂）。注意，它不会说"给我所有的区块"，而是要求一个区块哈希列表。这样做是为了减少网络负载，因为可以从不同的节点下载区块，我们不希望从一个节点下载几十GB的数据。

处理命令如下所示：

```
func handleGetBlocks(request []byte, bc *Blockchain) {
    ...
    blocks := bc.GetBlockHashes()
    sendInv(payload.AddrFrom, "block", blocks)
}
```

在我们的简化实现中，它将返回**所有区块哈希**。

9、inv

```
type inv struct {
    AddrFrom string
    Type     string
    Items    [][]byte
}
```

比特币使用**inv**向其他节点显示当前节点具有哪些区块或交易。再次提示，它不包含整个区块和交易，仅仅是它们的哈希值。该**Type**字段表示这些是区块还是交易。

处理**inv**更困难：

```
func handleInv(request []byte, bc *Blockchain) {
    ...
    fmt.Printf("Recevied inventory with %d %s\n", len(payload.Items), payloa
d.Type)

    if payload.Type == "block" {
        blocksInTransit = payload.Items

        blockHash := payload.Items[0]
        sendGetData(payload.AddrFrom, "block", blockHash)

        newInTransit := [][]byte{}
        for _, b := range blocksInTransit {
            if bytes.Compare(b, blockHash) != 0 {
                newInTransit = append(newInTransit, b)
            }
        }
        blocksInTransit = newInTransit
    }

    if payload.Type == "tx" {
        txID := payload.Items[0]

        if mempool[hex.EncodeToString(txID)].ID == nil {
            sendGetData(payload.AddrFrom, "tx", txID)
        }
    }
}
```

如果收到块哈希，我们希望将它们保存在**blocksInTransit**变量中以跟踪下载的区块。这允许我们从不同节点下载块。在将块放入传输状态之后，我们将**getdata**命令发送给**inv**消息的发送者并进行更新**blocksInTransit**。在真实的P2P网络中，我们希望从不同节点传输块。

在我们的实现中，我们永远不会发送**inv**多个哈希值。这就是为什么**payload.Type == "tx"**只有第一个哈希被采用时。然后我们检查我们的内存池中是否已经有这个哈希，如果没有，就发送**getdata**消息。

10、getdata

```
type getdata struct {
    AddrFrom string
    Type     string
    ID       []byte
}
```

getdata 用于某个区块或交易的请求，并且它仅包含一个区块或交易的ID。

```
func handleGetData(request []byte, bc *Blockchain) {
    ...
    if payload.Type == "block" {
        block, err := bc.GetBlock([]byte(payload.ID))

        sendBlock(payload.AddrFrom, &block)
    }

    if payload.Type == "tx" {
        txID := hex.EncodeToString(payload.ID)
        tx := mempool[txID]

        sendTx(payload.AddrFrom, &tx)
    }
}
```

该处理程序很简单：如果他们请求一个区块，则返回这个区块; 如果请求一笔交易，则返回交易。请注意，我们不检查我们是否确的有这个区块或交易。这是一个缺陷:)

11、block和tx

```
type block struct {
    AddrFrom string
    Block    []byte
}

type tx struct {
    AddFrom     string
    Transaction []byte
```

```
}
```

这是实际传输数据的这些消息。

处理**block**消息很简单：

```
func handleBlock(request []byte, bc *Blockchain) {
    ...

    blockData := payload.Block
    block := DeserializeBlock(blockData)

    fmt.Println("Recevied a new block!")
    bc.AddBlock(block)

    fmt.Printf("Added block %x\n", block.Hash)

    if len(blocksInTransit) > 0 {
        blockHash := blocksInTransit[0]
        sendGetData(payload.AddrFrom, "block", blockHash)

        blocksInTransit = blocksInTransit[1:]
    } else {
        UTXOSet := UTXOSet{bc}
        UTXOSet.Reindex()
    }
}
```

当我们收到一个新的区块时，我们将其放入我们的区块链中。如果有更多的区块要下载，我们会从我们下载前一个区块的同一节点请求它们。当我们最终下载所有区块时，UTXO集就会被重新索引。

> TODO：并非无条件信任，我们应该在将每个区块加入到区块链之前对它们进行验证。
> TODO：应该使用UTXOSet.Update(block)，而不是运行UTXOSet.Reindex()，因为如果区块链很大，重新索引整个UTXO集合需要花费很多时间。

处理tx消息是最困难的部分：

```
func handleTx(request []byte, bc *Blockchain) {
    ...
    txData := payload.Transaction
    tx := DeserializeTransaction(txData)
    mempool[hex.EncodeToString(tx.ID)] = tx
```

```
    if nodeAddress == knownNodes[0] {
        for _, node := range knownNodes {
            if node != nodeAddress && node != payload.AddFrom {
                sendInv(node, "tx", [][]byte{tx.ID})
            }
        }
    } else {
        if len(mempool) >= 2 && len(miningAddress) > 0 {
        MineTransactions:
            var txs []*Transaction

            for id := range mempool {
                tx := mempool[id]
                if bc.VerifyTransaction(&tx) {
                    txs = append(txs, &tx)
                }
            }

            if len(txs) == 0 {
                fmt.Println("All transactions are invalid! Waiting for new o
nes...")
                return
            }

            cbTx := NewCoinbaseTX(miningAddress, "")
            txs = append(txs, cbTx)

            newBlock := bc.MineBlock(txs)
            UTXOSet := UTXOSet{bc}
            UTXOSet.Reindex()

            fmt.Println("New block is mined!")

            for _, tx := range txs {
                txID := hex.EncodeToString(tx.ID)
                delete(mempool, txID)
            }

            for _, node := range knownNodes {
                if node != nodeAddress {
                    sendInv(node, "block", [][]byte{newBlock.Hash})
                }
            }

            if len(mempool) > 0 {
                goto MineTransactions
            }
        }
```

```
    }
}
```

首先要做的是将新交易放入内存池中（再次提示，交易必须在放入内存池之前进行验证）。下一步：

```
if nodeAddress == knownNodes[0] {
    for _, node := range knownNodes {
        if node != nodeAddress && node != payload.AddFrom {
            sendInv(node, "tx", [][]byte{tx.ID})
        }
    }
}
```

检查当前节点是否是中心节点。在我们的实现中，中心节点不会挖掘区块。相反，它会将新的交易转发到网络中的其他节点。

下一个很大的代码段只适用于矿工节点。让我们分成更小的部分：

```
if len(mempool) >= 2 && len(miningAddress) > 0 {
```

miningAddress仅在矿工节点上设置。当前（矿工）节点的内存池中有两笔或更多的交易时，开始挖矿。

```
for id := range mempool {
    tx := mempool[id]
    if bc.VerifyTransaction(&tx) {
        txs = append(txs, &tx)
    }
}

if len(txs) == 0 {
    fmt.Println("All transactions are invalid! Waiting for new ones...")
    return
}
```

首先，内存池中的所有交易都经过验证。无效的交易被忽略，如果没有有效的交易，则挖矿会被中断。

```
cbTx := NewCoinbaseTX(miningAddress, "")
txs = append(txs, cbTx)
```

```
newBlock := bc.MineBlock(txs)
UTXOSet := UTXOSet{bc}
UTXOSet.Reindex()

fmt.Println("New block is mined!")
```

已验证的交易正被放入一个区块，以及一个带有奖励的coinbase交易。挖矿结束后，UTXO 集被重新索引。

> TODO：同样，应该使用UTXOSet.Update来代替UTXOSet.Reindex

```
for _, tx := range txs {
    txID := hex.EncodeToString(tx.ID)
    delete(mempool, txID)
}

for _, node := range knownNodes {
    if node != nodeAddress {
        sendInv(node, "block", [][]byte{newBlock.Hash})
    }
}

if len(mempool) > 0 {
    goto MineTransactions
}
```

交易开始后，它将从内存池中移除。当前节点连接到的的所有其他节点，接收带有新块哈希的**inv**消息。他们可以在处理消息后请求该区块。

12、结果

让我们来回顾下我们之前定义的场景。

首先，在第一个终端窗口中设置**NODE_ID**为3000（**export NODE_ID=3000**）。在下一节之前我会用类似于 **NODE 3000** 或 **NODE 3001**来代替，你要了解哪个节点做什么。

NODE 3000

创建一个钱包和一个新的区块链：

```
$ blockchain_go createblockchain -address CENTREAL_NODE
```

（为了清晰和简洁，我将使用假地址）

之后，区块链会包含一个创世区块。我们需要保存块并将其用于其他节点。创世区块作为区块链的标识符（在比特币核心中，创世区块是硬编码的）。

```
$ cp blockchain_3000.db blockchain_genesis.db
```

NODE 3001

接下来，打开一个新的终端窗口并将节点ID设置为3001.这将是一个钱包节点。通过**blockchain_go createwallet**生成一些地址，我们把这些地址叫做**WALLET_1**，**WALLET_2**，**WALLET_3**。

NODE 3000

发送一些币到钱包地址：

```
$ blockchain_go send -from CENTREAL_NODE -to WALLET_1 -amount 10 -mine
$ blockchain_go send -from CENTREAL_NODE -to WALLET_2 -amount 10 -mine
```

-mine标志表示该区块将立即被同一节点挖掘。我们必须有这个标志，因为最初网络中没有矿工节点。

启动节点：

```
$ blockchain_go startnode
```

节点必须运行，直到场景结束。

NODE 3001

用上面保存的创始区块启动节点的区块链：

```
$ cp blockchain_genesis.db blockchain_3001.db
```

运行节点：

```
$ blockchain_go startnode
```

它会从中心节点下载所有的区块。要检查一切正常，请停止节点并检查余额：

```
$ blockchain_go getbalance -address WALLET_1
Balance of 'WALLET_1': 10

$ blockchain_go getbalance -address WALLET_2
Balance of 'WALLET_2': 10
```

另外，您可以检查**CENTRAL_NODE**地址的余额，因为节点3001现在有它的区块链：

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

NODE 3002

打开一个新的终端窗口并将其ID设置为3002，并生成一个钱包。这将是一个矿工节点。初始化区块链：

```
$ cp blockchain_genesis.db blockchain_3002.db
```

并启动节点：

```
$ blockchain_go startnode -miner MINER_WALLET
```

NODE 3001

发送一些币：

```
$ blockchain_go send -from WALLET_1 -to WALLET_3 -amount 1
$ blockchain_go send -from WALLET_2 -to WALLET_4 -amount 1
```

NODE 3002

快速切换到矿工节点，你会看到它挖出了一个新的区块！另外，检查中心节点的输出。

NODE 3001

切换到钱包节点并启动它：

```
$ blockchain_go startnode
```

它会下载新被挖出的区块！

停止并检查余额：

```
$ blockchain_go getbalance -address WALLET_1
Balance of 'WALLET_1': 9

$ blockchain_go getbalance -address WALLET_2
Balance of 'WALLET_2': 9

$ blockchain_go getbalance -address WALLET_3
Balance of 'WALLET_3': 1

$ blockchain_go getbalance -address WALLET_4
Balance of 'WALLET_4': 1

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

搞定，收工！

13、总结

这是该系列的最后一部分。我本可以发布更多的文章来实现P2P网络的真实原型，但我没有时间这样做。我希望这篇文章能够回答您关于比特币技术的一些问题，并提出新的问题，您可以自己找到答案。比特币技术中隐藏着更多有趣的东西！祝你好运！

PS：你可以从实现**addr**消息来开始优化这个网络，就像比特币网络协议中所描述的（链接在下面）那样。这是一个非常重要的信息，因为它允许节点相互发现。我开始着手实现它，但还没有完成！

链接：

1. 源代码
2. 比特币协议文档
3. 比特币网络