# 实验3

实验要求：

1. 理解区块链交易中的优化机制和奖励的作用；
2. 熟悉Merkle树结构和SPV验证过程；
3. 了解比特币支付核心P2PKH脚本，解释什么是智能合约；
4. 了解区块链网络中节点角色，以及常见场景中的交易流程；
5. 在实验2基础上完善交易过程，实现不同节点各自的功能，完成本次实验内容；

目录

[6、交易（2） 3](#_Toc525125563)

[6.1 奖励 3](#_Toc525125564)

[6.2 UTXO集 4](#_Toc525125565)

[6.3 Merkle树 9](#_Toc525125566)

[6.4 P2PKH 14](#_Toc525125567)

[6.5 总结 16](#_Toc525125568)

[7、网络 16](#_Toc525125569)

[7.1 区块链网络 17](#_Toc525125570)

[7.2 节点角色 18](#_Toc525125571)

[7.3 网络简化 19](#_Toc525125572)

[7.4 实现 19](#_Toc525125573)

[7.5 场景 20](#_Toc525125574)

[7.6 版本 21](#_Toc525125575)

[7.7 getblocks 24](#_Toc525125576)

[7.8 inv 24](#_Toc525125577)

[7.9 getdata 25](#_Toc525125578)

[7.10 block和tx 26](#_Toc525125579)

[7.11 结果 30](#_Toc525125580)

[7.12 总结 34](#_Toc525125581)

# 6、交易（2）

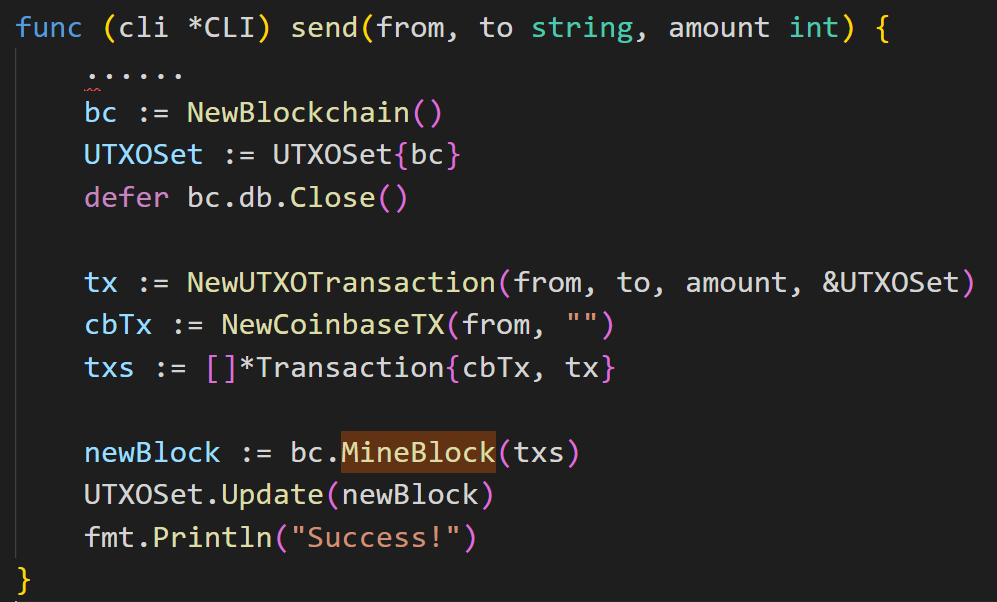
在这个系列文章的一开始，我们就提到了，区块链是一个分布式数据库。不过在之前的文章中，我们选择性地跳过了“分布式”这个部分，而是将注意力都放到了“数据库”部分。到目前为止，我们几乎已经实现了一个区块链数据库的所有元素。今天，我们将会分析之前跳过的一些机制。而在下一篇文章中，我们将会开始讨论区块链的分布式特性。

## 6.1 奖励

在上一篇文章中，我们略过的一个小细节是挖矿奖励。现在，我们已经可以来完善这个细节了。

挖矿奖励，实际上就是一笔coinbase交易。当一个挖矿节点开始挖出一个新块时，它会将交易从队列中取出，并在前面附加一笔coinbase交易。coinbase交易只有一个输出，里面包含了矿工的公钥哈希。

实现奖励，非常简单，更新send即可：



在我们的实现中，创建交易的人同时挖出了新块，所以会得到一笔奖励。

## 6.2 UTXO集

在 Part 3: 持久化和命令行接口中，我们研究了 Bitcoin Core 是如何在一个数据库中存储块的，并且了解到区块被存储在blocks数据库，交易输出被存储在chainstate数据库。会回顾一下chainstate的机构：

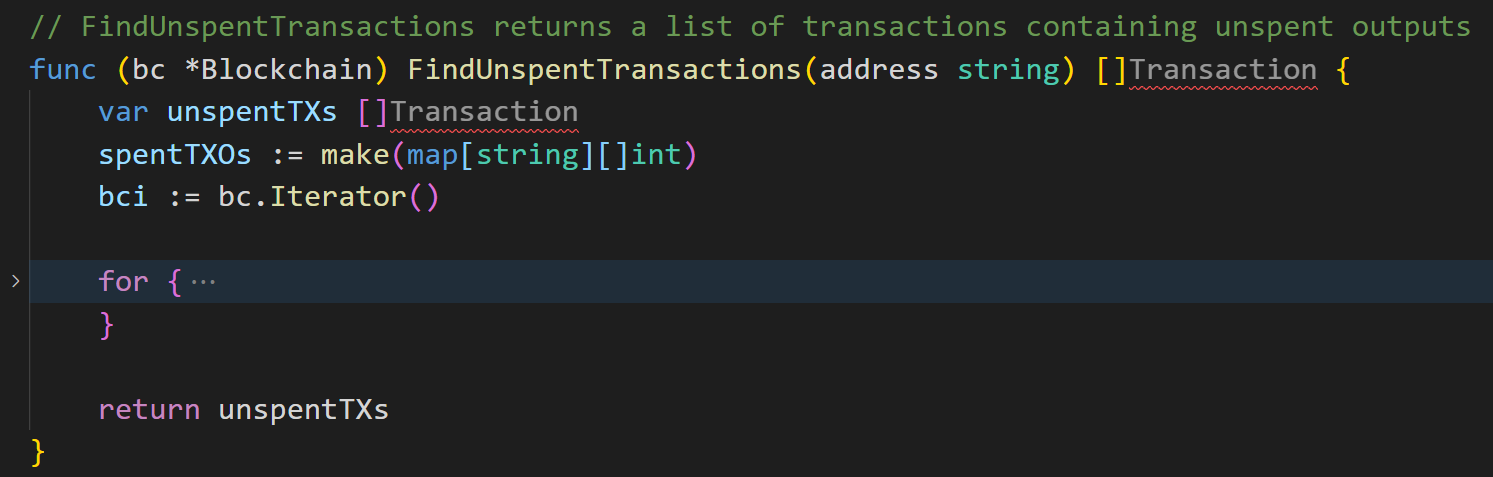
1. c + 32 字节的交易哈希 ->该笔交易的未花费交易输出记录
2. B + 32 字节的块哈希 ->未花费交易输出的块哈希

在之前那篇文章中，虽然我们已经实现了交易，但是并没有使用chainstate来存储交易的输出。所以，接下来我们继续完成这部分。

chainstate不存储交易。它所存储的是 UTXO 集，也就是未花费交易输出的集合。除此以外，它还存储了“数据库表示的未花费交易输出的块哈希”，不过我们会暂时略过块哈希这一点，因为我们还没有用到块高度（但是我们会在接下来的文章中继续改进）。

那么，我们为什么需要 UTXO 集呢？

来思考一下我们早先实现的Blockchain.FindUnspentTransactions方法：



这个函数找到有未花费输出的交易。由于交易被保存在区块中，所以它会对区块链里面的每一个区块进行迭代，检查里面的每一笔交易。截止 2021年10月21日，在比特币中已经有705873个块，整个数据库所需磁盘空间超过250 Gb。这意味着一个人如果想要验证交易，必须要运行一个全节点。此外，验证交易将会需要在许多块上进行迭代。

整个问题的解决方案是有一个仅有未花费输出的索引，这就是 UTXO 集要做的事情：这是一个从所有区块链交易中构建（对区块进行迭代，但是只须做一次）而来的缓存，然后用它来计算余额和验证新的交易。

好了，让我们来想一下实现 UTXO 集的话需要作出哪些改变。目前，找到交易用到了以下一些方法：

1. Blockchain.FindUnspentTransactions - 找到有未花费输出交易的主要函数。也是在这个函数里面会对所有区块进行迭代。
2. Blockchain.FindSpendableOutputs - 这个函数用于当一个新的交易创建的时候。如果找到有所需数量的输出。使用Blockchain.FindUnspentTransactions.
3. Blockchain.FindUTXO - 找到一个公钥哈希的未花费输出，然后用来获取余额。使用Blockchain.FindUnspentTransactions.
4. Blockchain.FindTransation - 根据 ID 在区块链中找到一笔交易。它会在所有块上进行迭代直到找到它。

可以看到，所有方法都对数据库中的所有块进行迭代。但是目前我们还没有改进所有方法，因为 UTXO 集没法存储所有交易，只会存储那些有未花费输出的交易。因此，它无法用于Blockchain.FindTransaction。

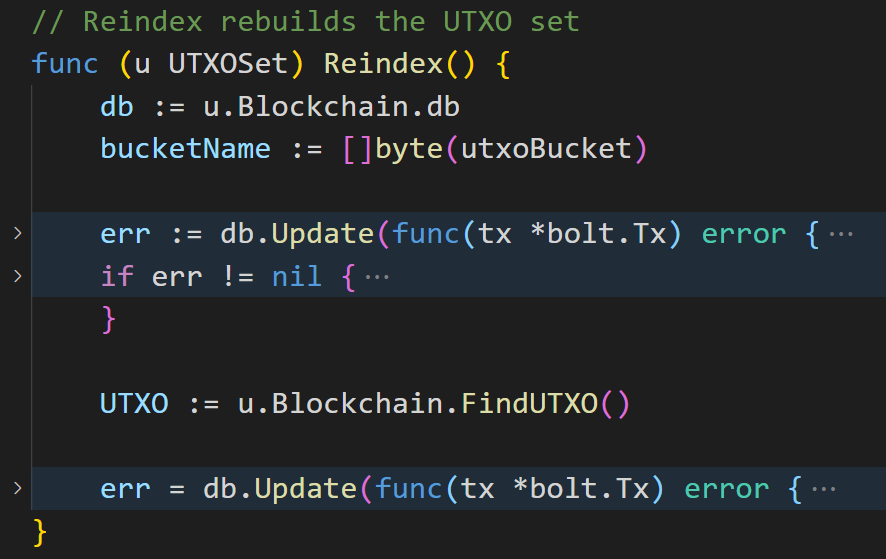
所以，我们想要以下方法：

1. Blockchain.FindUTXO - 通过对区块进行迭代找到所有未花费输出。
2. UTXOSet.Reindex - 使用UTXO找到未花费输出，然后在数据库中进行存储。这里就是缓存的地方。
3. UTXOSet.FindSpendableOutputs - 类似Blockchain.FindSpendableOutputs，但是使用 UTXO 集。
4. UTXOSet.FindUTXO - 类似Blockchain.FindUTXO，但是使用 UTXO 集。
5. Blockchain.FindTransaction跟之前一样。

因此，从现在开始，两个最常用的函数将会使用 cache，来开始写代码吧。



我们将会使用一个单一数据库，但是我们会将 UTXO 集从存储在不同的 bucket 中。因此，UTXOSet跟Blockchain一起。



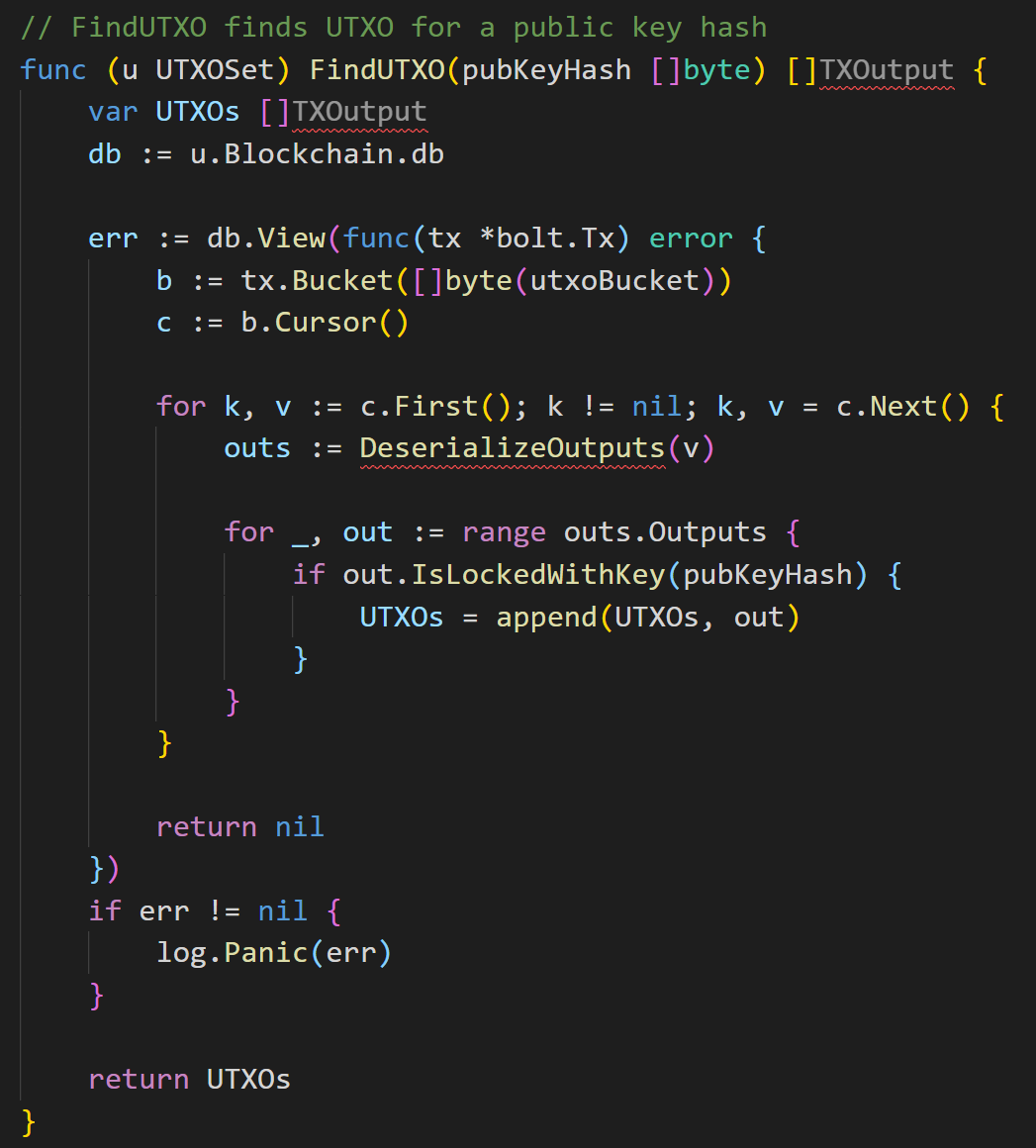
这个方法初始化了 UTXO 集。首先，如果 bucket 存在就先移除，然后从区块链中获取所有的未花费输出，最终将输出保存到 bucket 中。

Blockchain.FindUTXO几乎跟Blockchain.FindUnspentTransactions一模一样，但是现在它返回了一个TransactionID ->TransactionOutputs的 map。

现在，UTXO 集可以用于发送币：

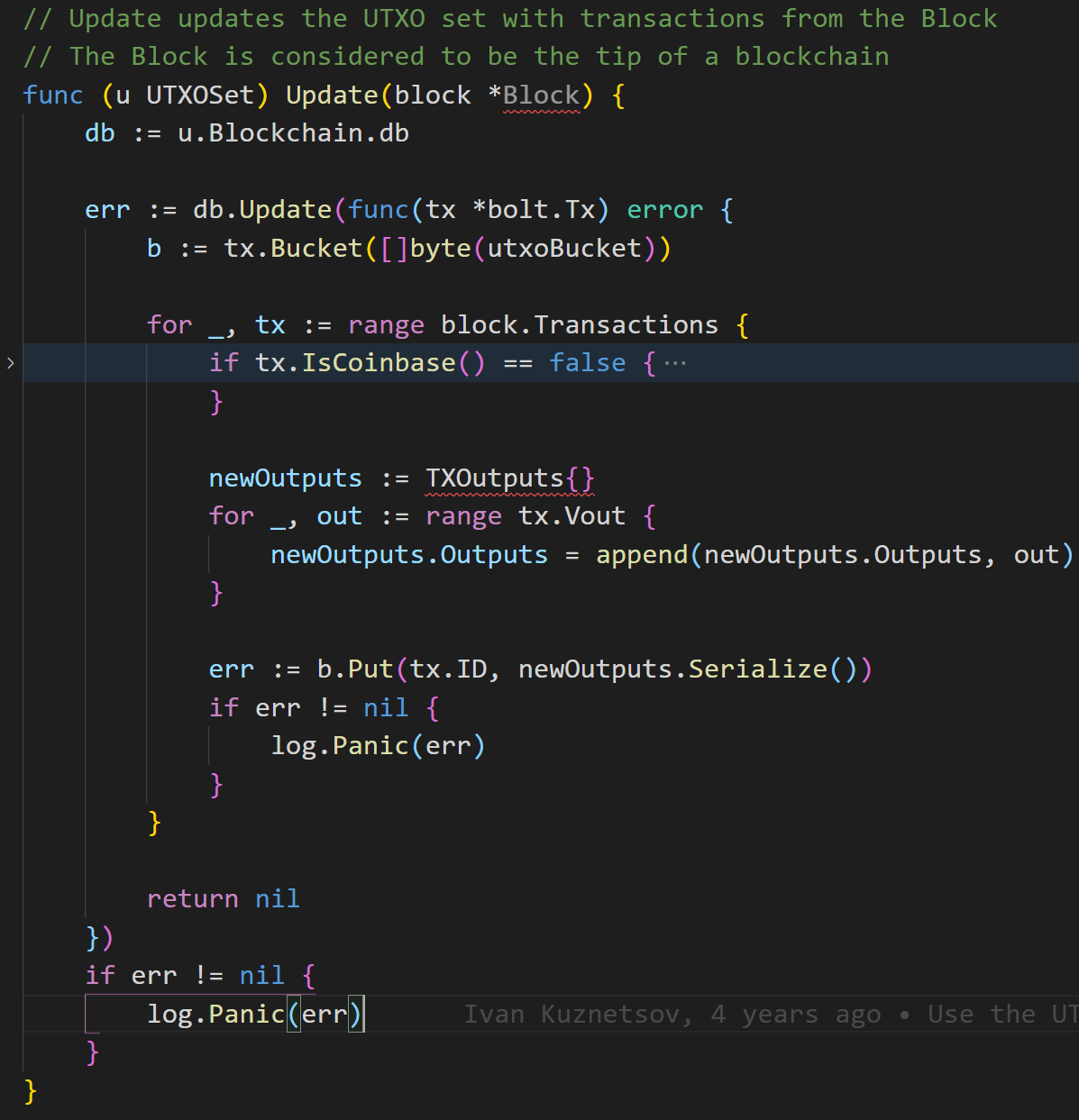


或者检查余额：



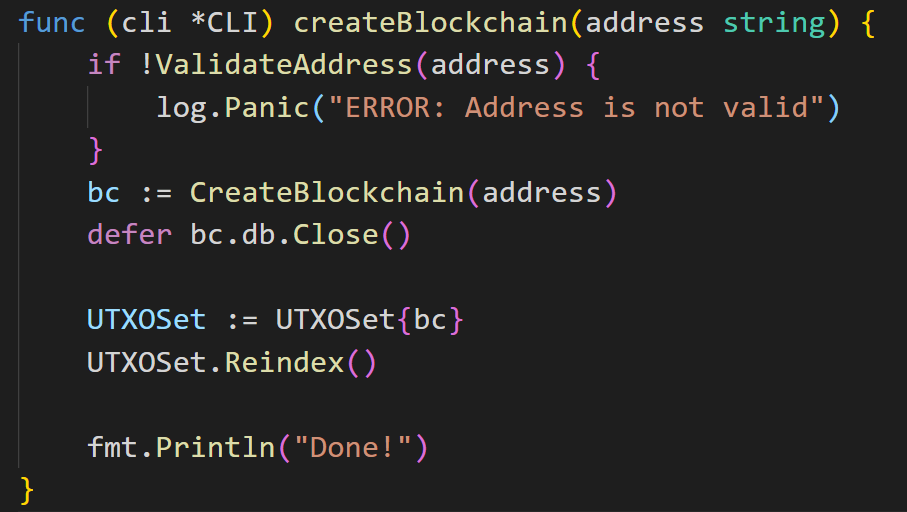
这是Blockchain方法的简单修改后的版本。这个Blockchain方法已经不再需要了。

有了 UTXO 集，也就意味着我们的数据（交易）现在已经被分开存储：实际交易被存储在区块链中，未花费输出被存储在 UTXO 集中。这样一来，我们就需要一个良好的同步机制，因为我们想要 UTXO 集时刻处于最新状态，并且存储最新交易的输出。但是我们不想每生成一个新块，就重新生成索引，因为这正是我们要极力避免的频繁区块链扫描。因此，我们需要一个机制来更新 UTXO 集：

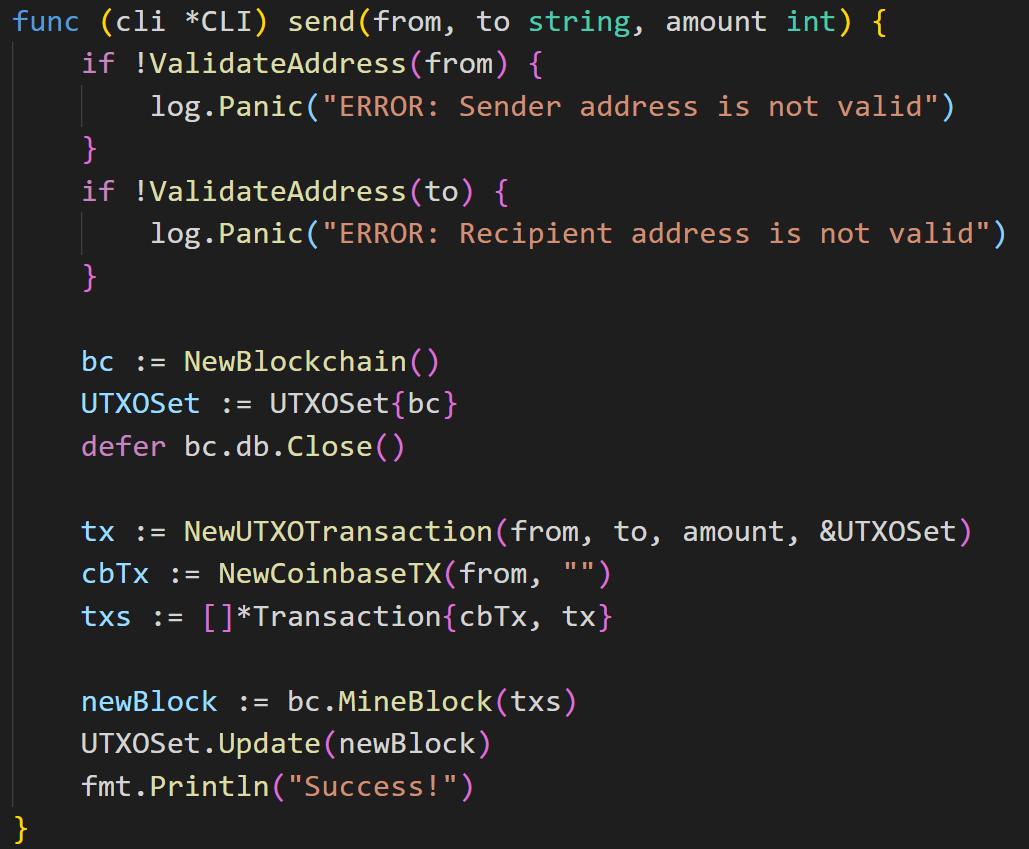


虽然这个方法看起来有点复杂，但是它所要做的事情非常直观。当挖出一个新块时，应该更新 UTXO 集。更新意味着移除已花费输出，并从新挖出来的交易中加入未花费输出。如果一笔交易的输出被移除，并且不再包含任何输出，那么这笔交易也应该被移除。相当简单！

现在让我们在必要的时候使用 UTXO 集：

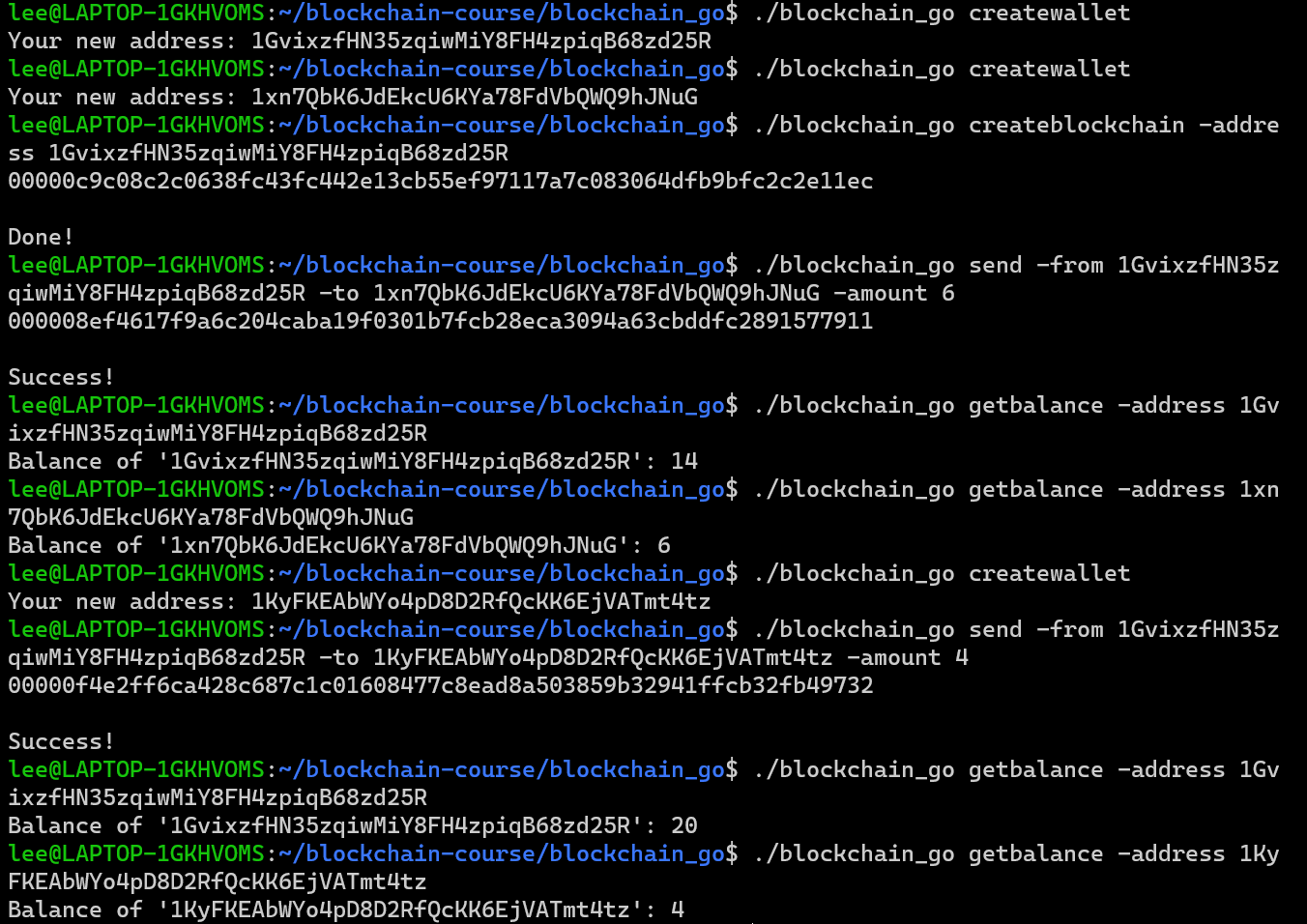


当一个新的区块链被创建以后，就会立刻进行重建索引。目前，这是Reindex唯一使用的地方，即使这里看起来有点“杀鸡用牛刀”，因为一条链开始的时候，只有一个块，里面只有一笔交易，Update已经被使用了。不过我们在未来可能需要重建索引的机制。



当挖出一个新块时，UTXO 集就会进行更新。

让我们来检查一下如否如期工作：



很好！1GvixzfHN35zqiwMiY8FH4zpiqB68zd25R地址接收到了 3 笔奖励：

1. 一次是挖出创世块
2. 一次是挖出块000008ef4617f9a6c204caba19f0301b7fcb28eca3094a63cbddfc2891577911
3. 一个是挖出块00000f4e2ff6ca428c687c1c01608477c8ead8a503859b32941ffcb32fb49732

## 6.3 Merkle树

在这篇文章中，我还想要再讨论一个优化机制。

上如上面所提到的，完整的比特币数据库（也就是区块链）需要超过250 Gb 的磁盘空间。因为比特币的去中心化特性，网络中的每个节点必须是独立，自给自足的，也就是每个节点必须存储一个区块链的完整副本。随着越来越多的人使用比特币，这条规则变得越来越难以遵守：因为不太可能每个人都去运行一个全节点。并且，由于节点是网络中的完全参与者，它们负有相关责任：节点必须验证交易和区块。另外，要想与其他节点交互和下载新块，也有一定的网络流量需求。

在中本聪的[比特币原始论文](https://bitcoin.org/bitcoin.pdf)中，对这个问题也有一个解决方案：简易支付验证（Simplified Payment Verification, SPV）。SPV 是一个比特币轻节点，它不需要下载整个区块链，也**不需要验证区块和交易**。相反，它会在区块链查找交易（为了验证支付），并且需要连接到一个全节点来检索必要的数据。这个机制允许在仅运行一个全节点的情况下有多个轻钱包。

为了实现 SPV，需要有一个方式来检查是否一个区块包含了某笔交易，而无须下载整个区块。这就是 Merkle 树所要完成的事情。

比特币用 Merkle 树来获取交易哈希，哈希被保存在区块头中，并会用于工作量证明系统。到目前为止，我们只是将一个块里面的每笔交易哈希连接了起来，将在上面应用了 SHA-256 算法。虽然这是一个用于获取区块交易唯一表示的一个不错的途径，但是它没有利用到 Merkle 树。

来看一下 Merkle 树：



每个块都会有一个 Merkle 树，它从叶子节点（树的底部）开始，一个叶子节点就是一个交易哈希（比特币使用双 SHA256 哈希）。叶子节点的数量必须是双数，但是并非每个块都包含了双数的交易。因为，如果一个块里面的交易数为单数，那么就将最后一个叶子节点（也就是 Merkle 树的最后一个交易，不是区块的最后一笔交易）复制一份凑成双数。

从下往上，两两成对，连接两个节点哈希，将组合哈希作为新的哈希。新的哈希就成为新的树节点。重复该过程，直到仅有一个节点，也就是树根。根哈希然后就会当做是整个块交易的唯一标示，将它保存到区块头，然后用于工作量证明。

Merkle 树的好处就是一个节点可以在不下载整个块的情况下，验证是否包含某笔交易。并且这些只需要一个交易哈希，一个 Merkle 树根哈希和一个 Merkle 路径。

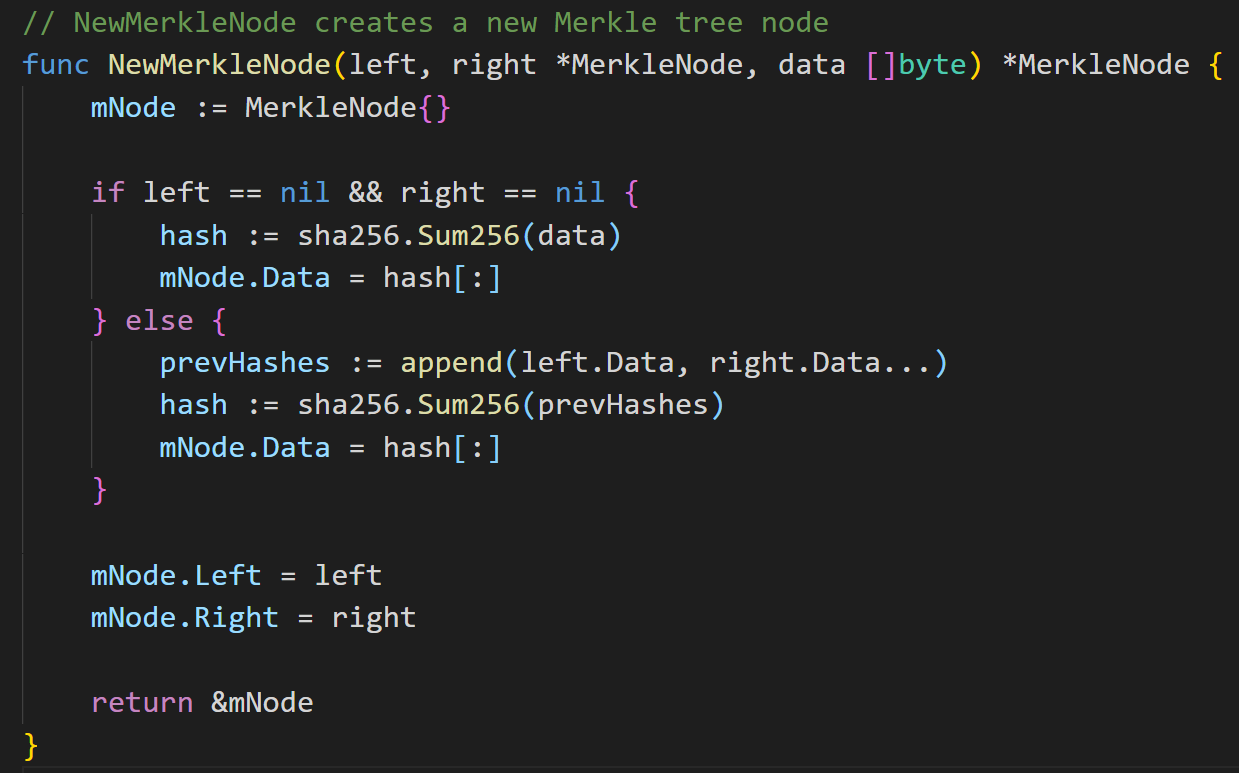
最后，来写代码：



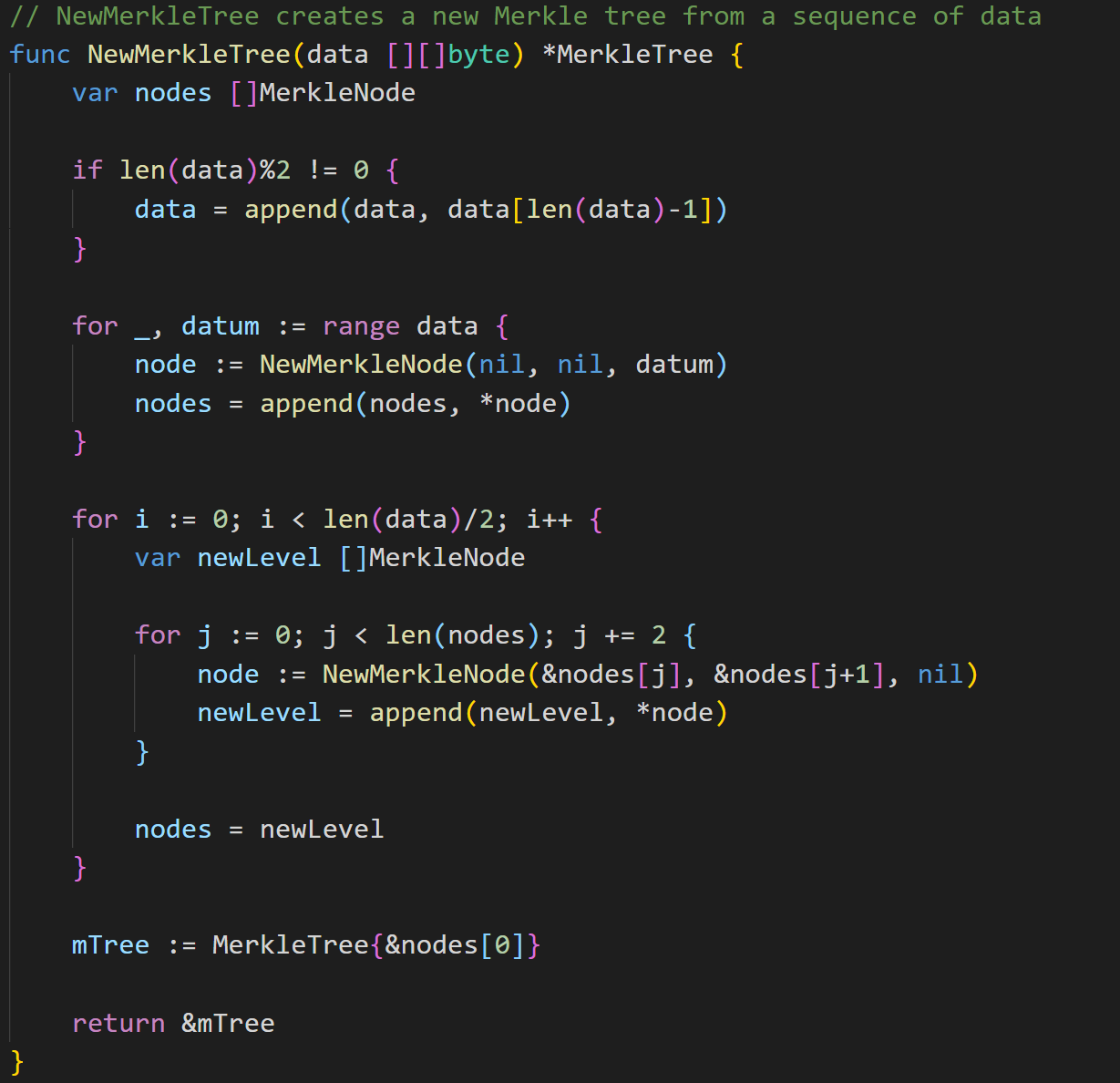


先从结构体开始。每个MerkleNode包含数据和指向左右分支的指针。MerkleTree实际上就是连接到下个节点的根节点，然后依次连接到更远的节点，等等。

让我们首先来创建一个新的节点：



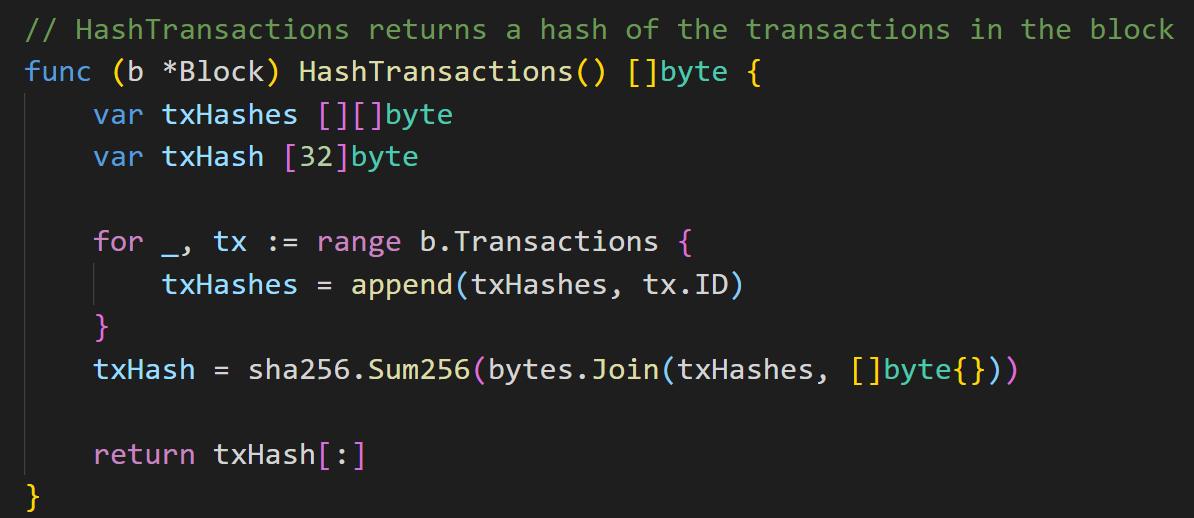
每个节点包含一些数据。当节点在叶子节点，数据从外界传入（在这里，也就是一个序列化后的交易）。当一个节点被关联到其他节点，它会将其他节点的数据取过来，连接后再哈希。



当生成一棵新树时，要确保的第一件事就是叶子节点必须是双数。然后，**数据**（也就是一个序列化后交易的数组）被转换成树的叶子，从这些叶子再慢慢形成一棵树。

​[btcsuite/btcd](https://github.com/btcsuite/btcd/blob/50de9da05b50eb15658bb350f6ea24368a111ab7/blockchain/merkle.go#L71-L155)是用数组实现的merkle树，因为这么做可以减少一半的内存使用。

现在，让我们来修改Block.HashTransactions，它用于在工作量证明系统中获取交易哈希：



首先，交易被序列化（使用encoding/gob），然后使用序列后的交易构建一个Mekle树。树根将会作为块交易的唯一标识符。

## 6.4 P2PKH

还有一件事情，我想要再谈一谈。

大家应该还记得，在比特币中有一个脚本（Script）编程语言，它用于锁定交易输出；交易输入提供了解锁输出的数据。这个语言非常简单，用这个语言写的代码其实就是一系列数据和操作符而已。比如如下示例：

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：将数据依次放入栈内，当遇到操作符时，就从栈内取出数据，并将操作符作用于数据，然后将结果作为栈顶元素。*脚本*的栈，实际上就是一个先进后出的内存存储：栈里的第一个元素最后一个取出，后面的每一个元素都会放到前一个元素之上。

让我们来对上面的脚本分部执行：

|  |  |  |  |
| --- | --- | --- | --- |
| 步骤 | 栈 | 脚本 | 说明 |
| 1 | 空 | 5 2 OP\_ADD 7 OP\_EQUAL | 一开始栈为空 |
| 2 | 5 | 2 OP\_ADD 7 OP\_EQUAL | 从脚本里面取出5放入栈上 |
| 3 | 5 2 | OP\_ADD 7 OP\_EQUAL | 从脚本里面取出2放入栈上 |
| 4 | 7 | 7 OP\_EQUAL | 遇到操作符OP\_ADD, 从栈里取出两个操作数5和2，相加后将结果放回栈上 |
| 5 | 7 7 | OP\_EQUAL | 从脚本里面取出7放到栈上 |
| 6 | true | 空 | 遇到操作符OP\_EQUAL，从栈里取出两个操作数并比较，将比较的结果放回栈内，脚本执行完毕，为空 |

OP\_ADD从栈内取两个元素，将这两个元素进行相加，然后将结果重新放回栈内。OP\_EQUAL从栈内取两个元素，然后对这两个元素进行比较：如果它们相等，就在栈上放一个true，否则放一个false。脚本执行的结果就是栈顶元素：在我们的案例中，如果是true，那么表明脚本执行成功。

现在来看一下在比特币中，是如何用脚本执行支付的：

<signature><pubKey> OP\_DUP OP\_HASH160 <pubKeyHash> OP\_EQUALVERIFY OP\_CHECKSIG

这个脚本叫做Pay to Public Key Hash(P2PKH)，这是比特币最常用的一个脚本。它所做的事情就是向一个公钥哈希支付，也就是说，用某一个公钥锁定一些币。这是**比特币支付的核心**：没有账户，没有资金转移；只有一个脚本检查提供的签名和公钥是否正确。

这个脚本实际存储为两个部分：

1. 第一个部分，<signature><pubkey>，存储在输入的ScriptSig字段。
2. 第二部分，OP\_DUP OP\_HASH160 <pubkeyHash> OP\_EQUALVERYFY OP\_CHECKSIG存储在输出的ScriptPubKey里面。

|  |  |  |
| --- | --- | --- |
| 步骤 | 栈 | 脚本 |
| 1 | 空 | <signature><pubKey> OP\_DUP OP\_HASH160 <pubKeyHash> OP\_EQUALVERIFY OP\_CHECKSIG |
| 2 | <signature> | <pubKey> OP\_DUP OP\_HASH160 <pubKeyHash> OP\_EQUALVERIFY OP\_CHECKSIG |
| 3 | <signature><pubkey> | OP\_DUP OP\_HASH160 <pubKeyHash> OP\_EQUALVERIFY OP\_CHECKSIG |
| 4 | <signature><pubKey><pubKey> | OP\_HASH160 <pubKeyHash> OP\_EQUALVERIFY OP\_CHECKSIG |
| 5 | <signature><pubKey><pubKeyHash> | <pubKeyHash> OP\_EQUALVERIFY OP\_CHECKSIG |
| 6 | <signature><pubKey><pubKeyHash><pubKeyHash> | OP\_EQUALVERIFY OP\_CHECKSIG |
| 7 | <signature><pubKey> | OP\_CHECKSIG |
| 8 | true或false | 空 |

因此，输出定了解锁的逻辑，输入提供解锁输出的“钥匙”。然我们来执行一下这个脚本：

OP\_DUP对栈顶元素进行复制。OP\_HASH160取栈顶元素，然后用RIPEMD160对它进行哈希，再将结果送回到栈上。OP\_EQUALVERIFY将栈顶的两个元素进行比较，如果它们不相等，终止脚本。OP\_CHECKSIG通过对交易进行哈希，并使用<signature>和pubKey来验证一笔交易的签名。最后的操作符有点复杂：它生成了一个修剪后的交易副本，对它进行哈希（因为它是一个被签名后的交易哈希），然后使用提供的<signature>和pubKey检查签名是否正确。

有了一个这样的脚本语言，实际上也可以让比特币成为一个智能合约平台：除了将一个单一的公钥转移资金，这个语言还使得一些其他的支付方案成为可能。

## 6.5 总结

我们已经实现了一个基于区块链的加密货币的几乎所有关键特性。我们已经有了区块链，地址，挖矿和交易。但是要想给这些所有的机制赋予生命，让比特币成为一个全球系统，还有一个不可或缺的环节：共识（consensus）。接下来，我们将会开始实现区块链的“去中心化（decenteralized）”。

# 7、网络

到目前为止，我们所构建的原型已经具备了区块链所有的关键特性：匿名，安全，随机生成的地址；区块链数据存储；工作量证明系统；可靠地存储交易。尽管这些特性都不可或缺，但是仍有不足。能够使得这些特性真正发光发热，使得加密货币成为可能的，是**网络（network）**。如果实现的这样一个区块链仅仅运行在单一节点上，有什么用呢？如果只有一个用户，那么这些基于密码学的特性，又有什么用呢？正是由于网络，才使得整个机制能够运转和发光发热。

你可以将这些区块链特性认为是规则（rule），类似于人类在一起生活，繁衍生息建立的规则，一种社会安排。区块链网络就是一个程序社区，里面的每个程序都遵循同样的规则，正是由于遵循着同一个规则，才使得网络能够长存。类似的，当人们都有着同样的想法，就能够构建更好的生活。如果有人遵循着不同的规则，那么他们就将生活在一个分裂的社区中。同样的，如果有区块链节点遵循不同的规则，那么也可能会形成一个分裂的网络。

**重点在于**：如果没有网络，或者大部分节点都不遵守同样的规则，那么规则就会形同虚设，毫无用处！

## 7.1 区块链网络

区块链网络是去中心化的，这意味着没有服务器，客户端也不需要依赖服务器来获取或处理数据。在区块链网络中，有的是节点，每个节点是网络的一个完全（full-fledged）成员。节点就是一切：它既是一个客户端，也是一个服务器。这一点需要牢记于心，因为这与传统的网页应用非常不同。

区块链网络是一个 P2P（Peer-to-Peer，端到端）的网络，即节点直接连接到其他节点。它的拓扑是扁平的，因为在节点的世界中没有层级之分。下面是它的示意图：



要实现这样一个网络节点更加困难，因为它们必须执行很多操作。每个节点必须与很多其他节点进行交互，它必须请求其他节点的状态，与自己的状态进行比较，当状态过时时进行更新。

## 7.2 节点角色

尽管节点具有完备成熟的属性，但是它们也可以在网络中扮演不同角色。比如：

1. 矿工这样的节点运行于强大或专用的硬件（比如 ASIC）之上，它们唯一的目标是，尽可能快地挖出新块。矿工是区块链中唯一可能会用到工作量证明的角色，因为挖矿实际上意味着解决PoW难题。在权益证明PoS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块的有效性，并对交易进行确认。为此，他们必须拥有区块链的完整拷贝。同时，全节点执行路由操作，帮助其他节点发现彼此。对于网络来说，非常重要的一段就是要有足够多的全节点。因为正是这些节点执行了决策功能：他们决定了一个块或一笔交易的有效性。
3. SPV表示 Simplified Payment Verification，简单支付验证。这些节点并不存储整个区块链副本，但是仍然能够对交易进行验证（不过不是验证全部交易，而是一个交易子集，比如，发送到某个指定地址的交易）。一个 SPV 节点依赖一个全节点来获取数据，可能有多个 SPV 节点连接到一个全节点。SPV 使得钱包应用成为可能：一个人不需要下载整个区块链，但是仍能够验证他的交易。

## 7.3 网络简化

为了在目前的区块链原型中实现网络，我们不得不简化一些事情。因为我们没有那么多的计算机来模拟一个多节点的网络。当然，可以使用虚拟机或是 Docker 来解决这个问题，但是这会使一切都变得更复杂。所以，我们想要在一台机器上运行多个区块链节点，同时希望它们有不同的地址。为了实现这一点，我们将使用**端口号作为节点标识符**，而不是使用 IP 地址，比如将会有这样地址的节点：**127.0.0.1:3000**，**127.0.0.1:3001**，**127.0.0.1:3002**等等。我们叫它端口节点（port node） ID，并使用环境变量NODE\_ID对它们进行设置。故而，你可以打开多个终端窗口，设置不同的NODE\_ID运行不同的节点。

这个方法也需要有不同的区块链和钱包文件。它们现在必须依赖于节点 ID 进行命名，比如 blockchain\_3000.db, blockchain\_30001.db and wallet\_3000.db, wallet\_30001.db 等等。

## 7.4 实现

所以，当下载 Bitcoin Core 并首次运行时，到底发生了什么呢？它必须连接到某个节点下载最新状态的区块链。考虑到你的电脑并没有意识到所有或是部分的比特币节点，那么连接到的“某个节点”到底是什么？

在 Bitcoin Core 中硬编码一个地址，已经被证实是一个错误：因为节点可能会被攻击或关机，这会导致新的节点无法加入到网络中。在 Bitcoin Core 中，硬编码了[DNS seeds](https://bitcoin.org/en/glossary/dns-seed)。虽然这些并不是节点，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，它会连接到一个种子节点，获取全节点列表，随后从这些节点中下载区块链。

不过在我们目前的实现中，无法做到完全的去中心化，因为会出现中心化的特点。我们会有三个节点：

1. 一个中心节点。所有其他节点都会连接到这个节点，这个节点会在其他节点之间发送数据。
2. 一个矿工节点。这个节点会在内存池中存储新的交易，当有足够的交易时，它就会打包挖出一个新块。
3. 一个钱包节点。这个节点会被用作在钱包之间发送币。但是与 SPV 节点不同，它存储了区块链的一个完整副本。

## 7.5 场景

本节的目标是实现如下场景：

1. 中心节点创建一个区块链。
2. 一个其他（钱包）节点连接到中心节点并下载区块链。
3. 另一个（矿工）节点连接到中心节点并下载区块链。
4. 钱包节点创建一笔交易。
5. 矿工节点接收交易，并将交易保存到内存池中。
6. 当内存池中有足够的交易时，矿工开始挖一个新块。
7. 当挖出一个新块后，将其发送到中心节点。
8. 钱包节点与中心节点进行同步。
9. 钱包节点的用户检查他们的支付是否成功。

这就是比特币中的一般流程。尽管我们不会实现一个真实的 P2P 网络，但是我们会实现一个真实的，也是比特币最常见最重要的用户场景。

## 7.6 版本

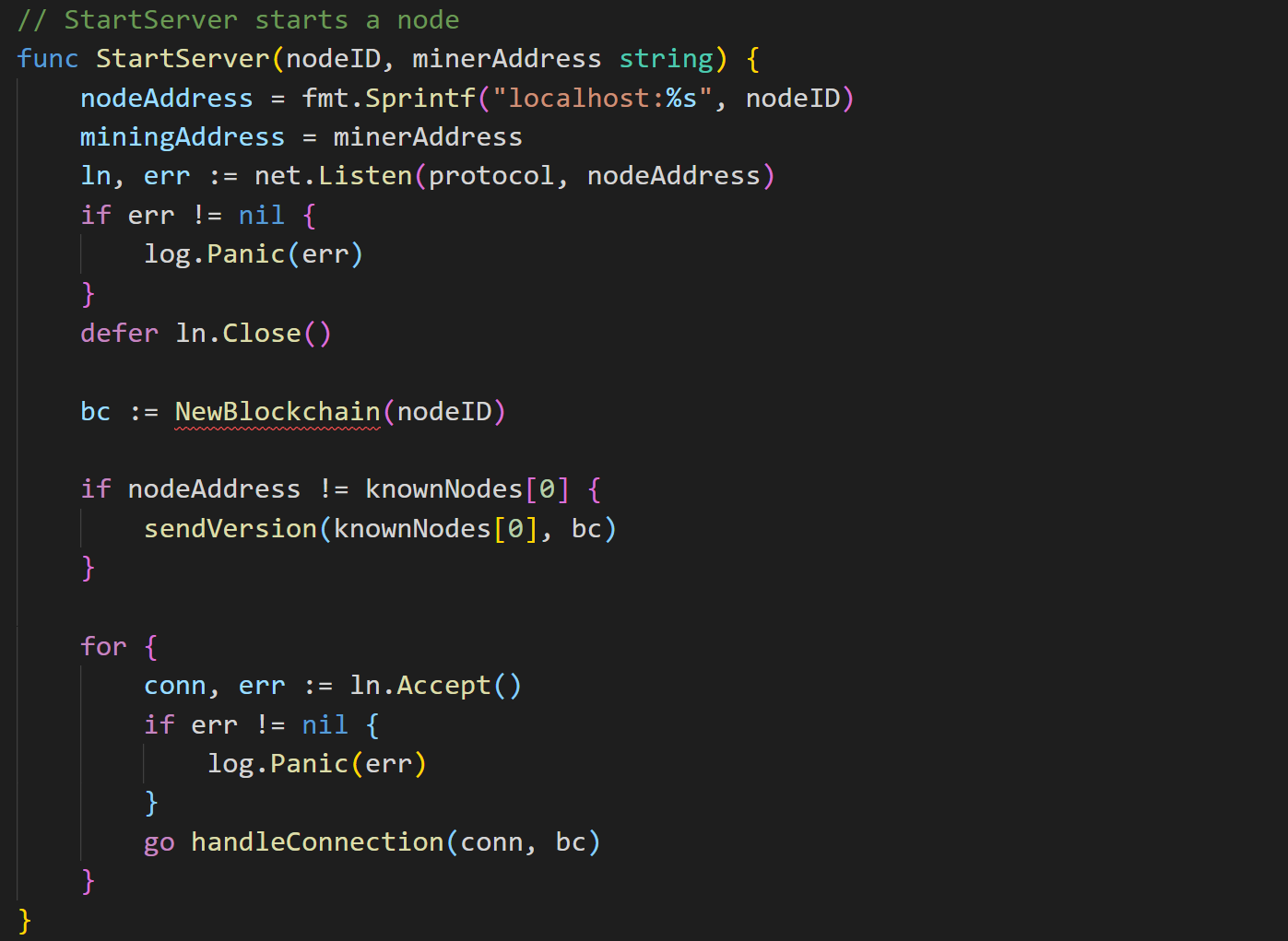
节点通过消息（message）进行交流。当一个新的节点开始运行时，它会从一个 DNS 种子获取几个节点，给它们发送version消息，看起来就像是这样：



由于我们仅有一个区块链版本，所以Version字段实际并不会存储什么重要信息。BestHeight存储区块链中节点的高度。AddFrom存储发送者的地址。

接收到version消息的节点应该做什么呢？它会响应自己的version消息。这是一种握手：如果没有事先互相问候，就不可能有其他交流。不过，这并不是出于礼貌：version用于找到一个更长的区块链。当一个节点接收到version消息，它会检查本节点的区块链是否比BestHeight的值更大。如果不是，节点就会请求并下载缺失的块。

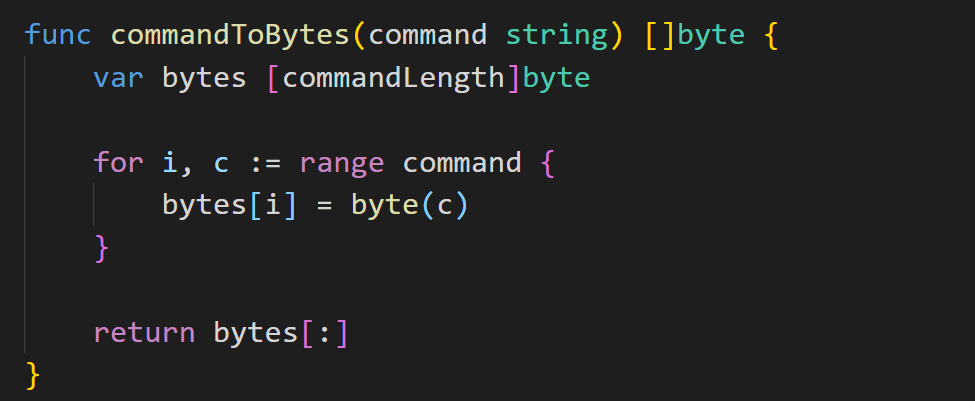
为了接收消息，我们需要一个服务器：



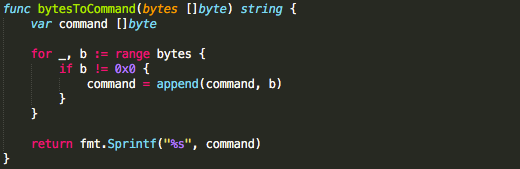
首先，我们对中心节点的地址进行硬编码：因为每个节点必须知道从何处开始初始化。minerAddress参数指定了接收挖矿奖励的地址。



我们的消息，在底层就是字节序列。前 12 个字节指定了命令名（比如这里的version），后面的字节会包含**gob**编码的消息结构，commandToBytes看起来是这样：



它创建一个 12 字节的缓冲区，并用命令名进行填充，将剩下的字节置为空。下面一个相反的函数：



当一个节点接收到一个命令，它会运行bytesToCommand来提取命令名，并选择正确的处理器处理命令主体：



下面是version命令处理器：



首先，我们需要对请求进行解码，提取有效信息。所有的处理器在这部分都类似，所以我们会下面的代码片段中略去这部分。

然后节点将从消息中提取的BestHeight与自身进行比较。如果自身节点的区块链更长，它会回复version消息；否则，它会发送getblocks消息。

## 7.7 getblocks



getblocks意为 “给我看一下你有什么区块”（在比特币中，这会更加复杂）。注意，它并没有说“把你全部的区块给我”，而是请求了一个块哈希的列表。这是为了减轻网络负载，因为区块可以从不同的节点下载，并且我们不想从一个单一节点下载数十 GB 的数据。

处理命令十分简单：



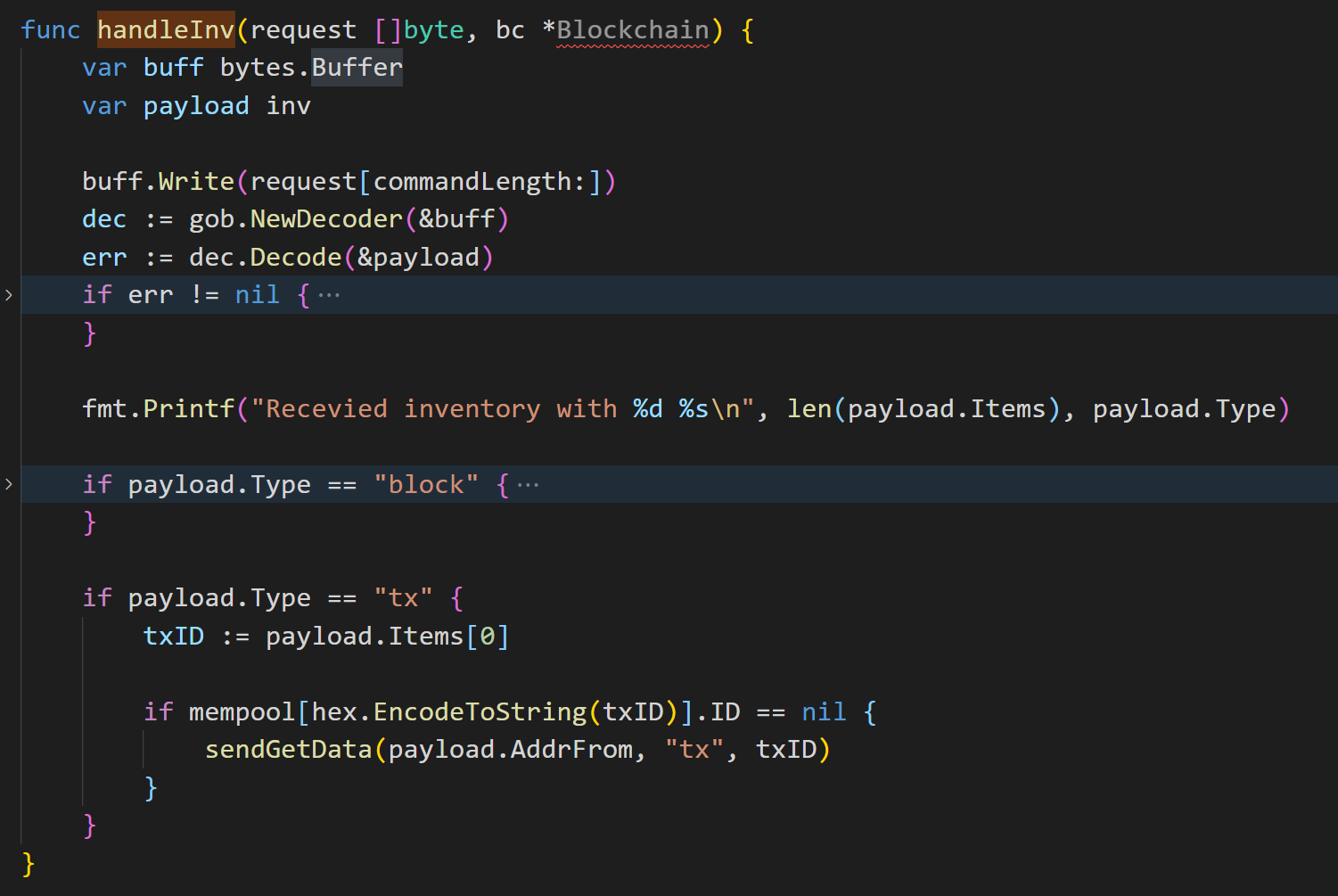
在我们简化版的实现中，它会返回**所有块哈希**。

## 7.8 inv



比特币使用inv来向其他节点展示当前节点有什么块和交易。再次提醒，它没有包含完整的区块链和交易，仅仅是哈希而已。Type字段表明了这是块还是交易。

处理inv稍显复杂：



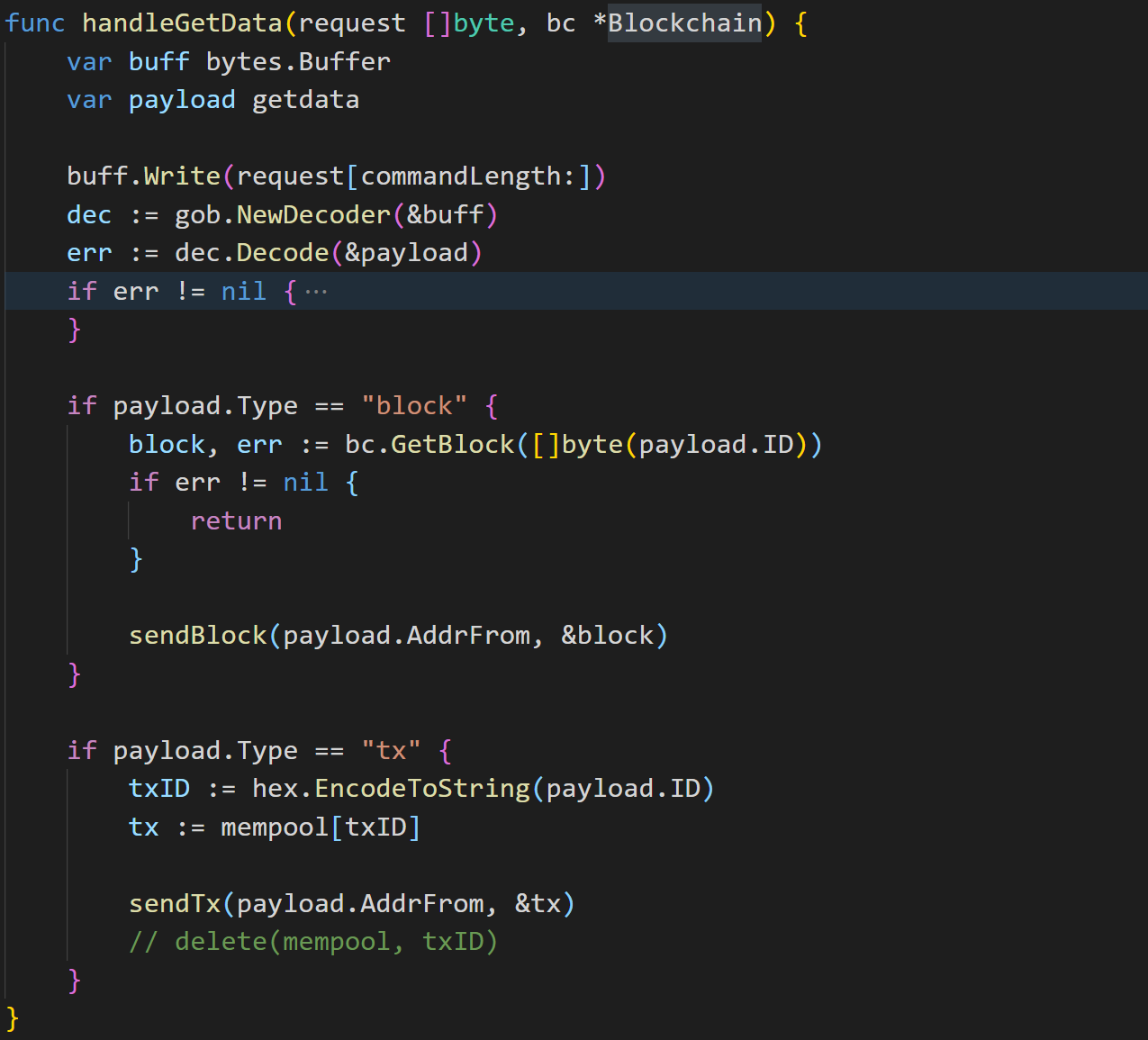
如果收到块哈希，我们想要将它们保存在blocksInTransit变量来跟踪已下载的块。这能够让我们从不同的节点下载块。在将块置于传送状态时，我们给inv消息的发送者发送getdata命令并更新blocksInTransit。在一个真实的 P2P 网络中，我们会想要从不同节点来传送块。

在我们的实现中，我们永远也不会发送有多重哈希的inv。这就是为什么当payload.Type == "tx"时，只会拿到第一个哈希。然后我们检查是否在内存池中已经有了这个哈希，如果没有，发送getdata消息。

## 7.9 getdata



getdata用于某个块或交易的请求，它可以仅包含一个块或交易的 ID。



这个处理器比较地直观：如果它们请求一个块，则返回块；如果它们请求一笔交易，则返回交易。注意，我们并不检查实际上是否已经有了这个块或交易。这是一个缺陷 :)

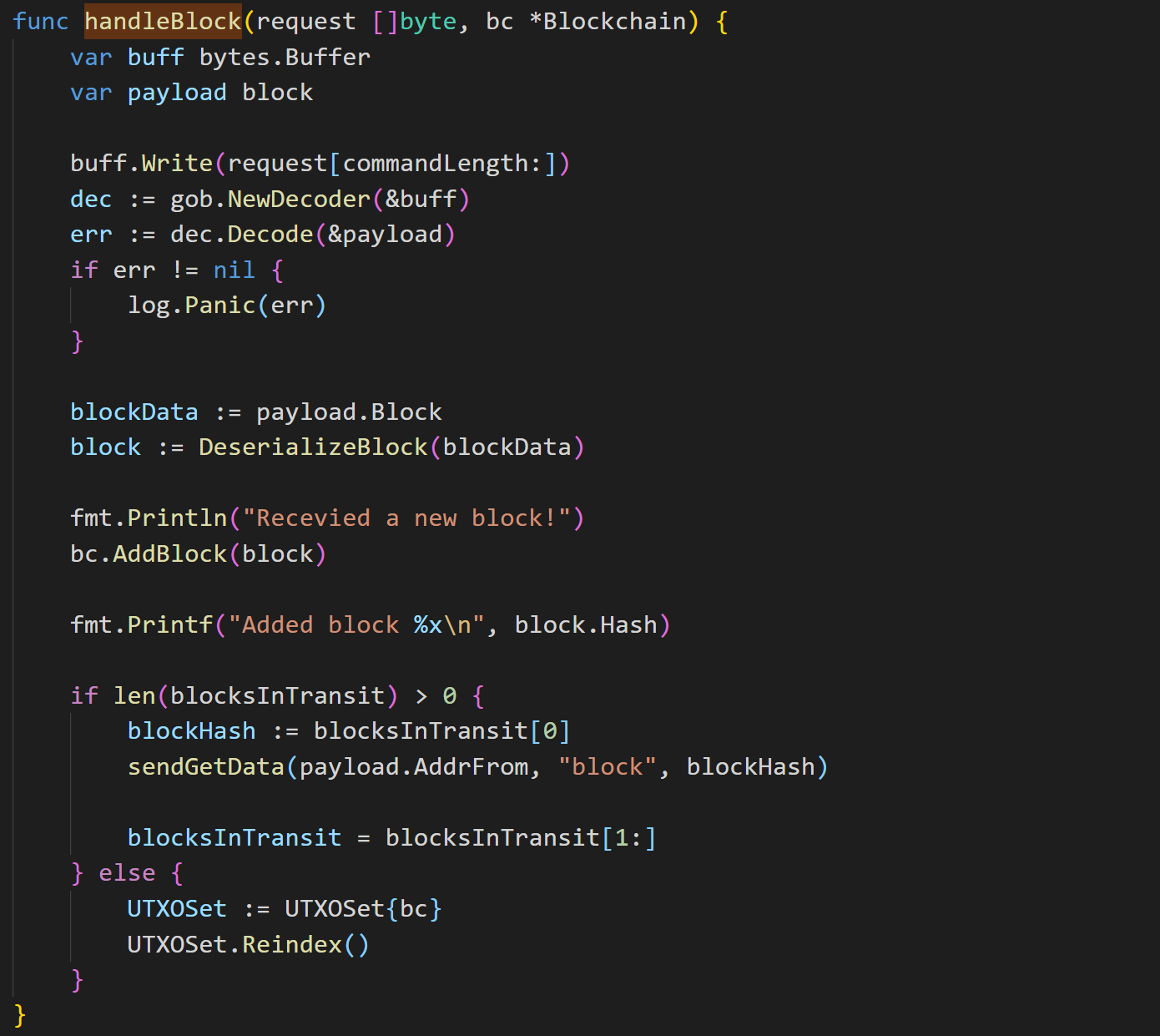
## 7.10 block和tx





实际完成数据转移的正是这些消息。

处理block消息十分简单：



当接收到一个新块时，我们把它放到区块链里面。如果还有更多的区块需要下载，我们继续从上一个下载的块的那个节点继续请求。当最后把所有块都下载完后，对 UTXO 集进行重新索引。

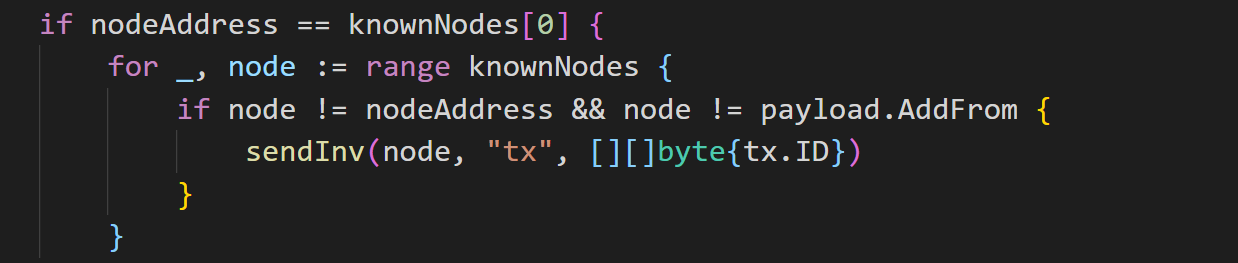
TODO：并非无条件信任，我们应该在将每个块加入到区块链之前对它们进行验证。

TODO: 并非运行UTXOSet.Reindex()，而是应该使用UTXOSet.Update(block)，因为如果区块链很大，它将需要很多时间来对整个 UTXO 集重新索引。

处理tx消息是最困难的部分：



首先要做的事情是将新交易放到内存池中（再次提醒，在将交易放到内存池之前，必要对其进行验证）。下个片段：

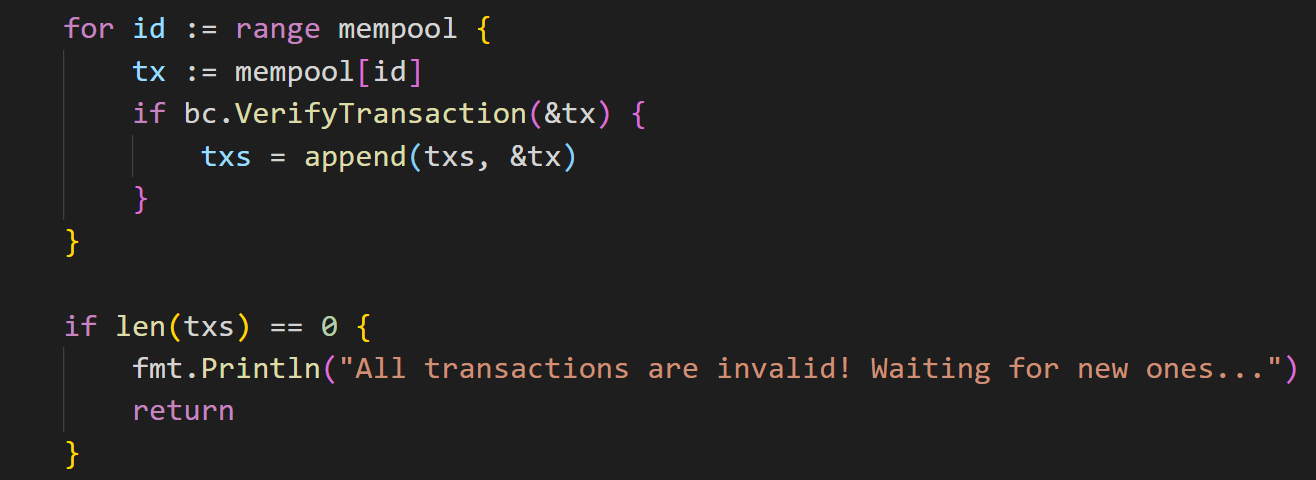


检查当前节点是否是中心节点。在我们的实现中，中心节点并不会挖矿。它只会将新的交易推送给网络中的其他节点。

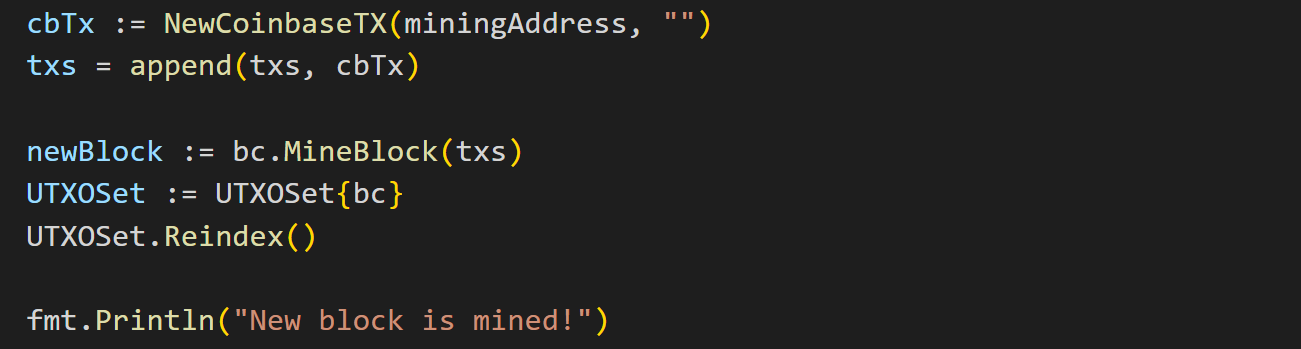
下一个很大的代码片段是矿工节点“专属”。让我们对它进行一下分解:



miningAddress只会在矿工节点上设置。如果当前节点（矿工）的内存池中有两笔或更多的交易，开始挖矿：

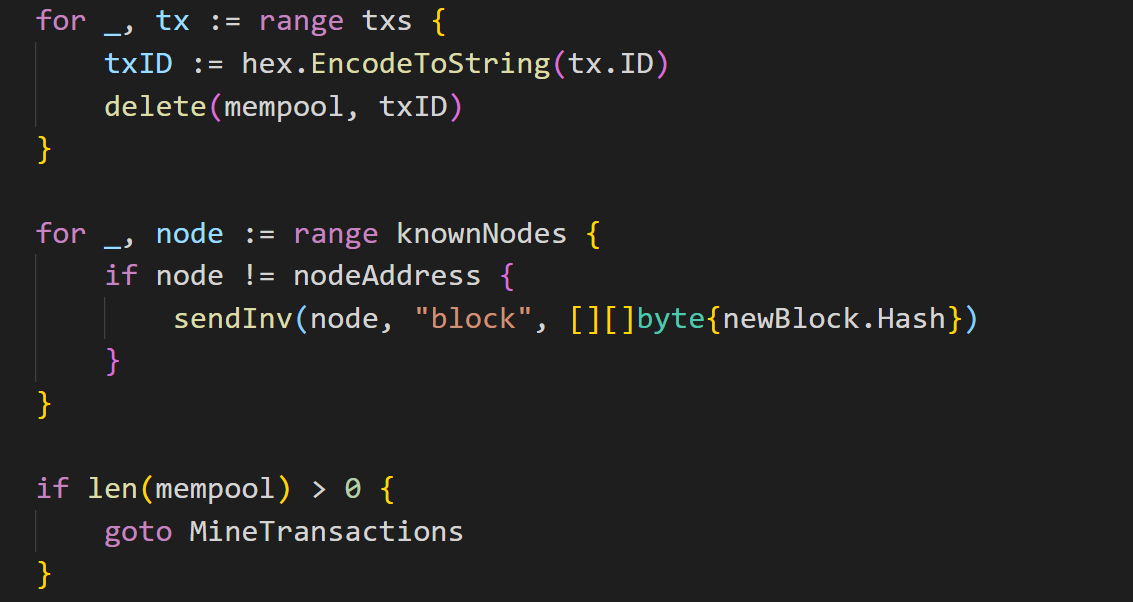


首先，内存池中所有交易都是通过验证的。无效的交易会被忽略，如果没有有效交易，则挖矿中断。



验证后的交易被放到一个块里，同时还有附带奖励的coinbase交易。当块被挖出来以后，UTXO 集会被重新索引。

TODO: 提醒，应该使用UTXOSet.Update而不是UTXOSet.Reindex.



当一笔交易被挖出来以后，就会被从内存池中移除。当前节点所连接到的所有其他节点，接收带有新块哈希的inv消息。在处理完消息后，它们可以对块进行请求。

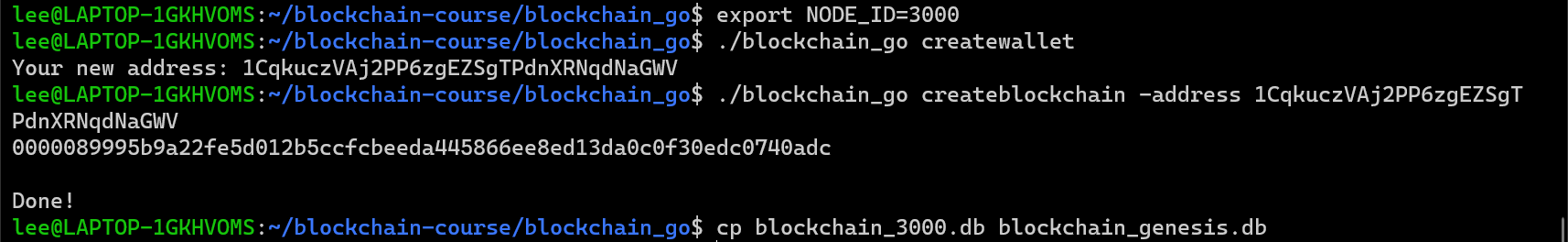
## 7.11 结果

来回顾一下上面定义的场景。

首先，在第一个终端窗口中将NODE\_ID设置为 3000（export NODE\_ID=3000）。为了展示什么节点执行什么操作，会使用像**NODE 3000**或**NODE 3001**进行标识。

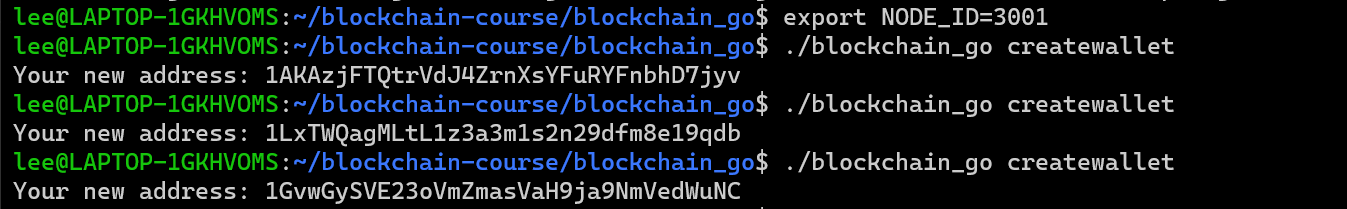
#### NODE 3000

创建一个钱包和一个新的区块链，然后，会生成一个仅包含创世块的区块链。我们需要保存块，并在其他节点使用。创世块承担了一条链标识符的角色（在 Bitcoin Core 中，创世块是硬编码的）



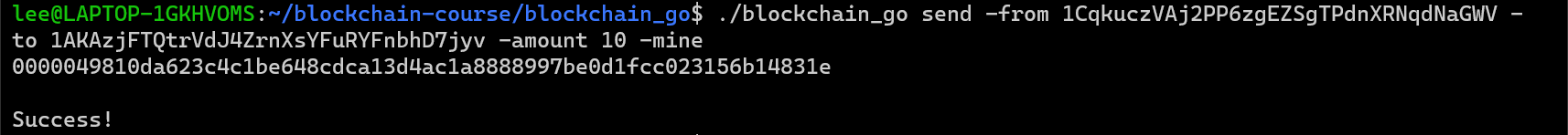
#### NODE 3001

接下来，打开一个新的终端窗口，将 node ID 设置为 3001。这会作为一个钱包节点。通过blockchain\_gocreatewallet生成一些地址，我们把这些地址叫做 WALLET\_1, WALLET\_2, WALLET\_3.



#### NODE 3000

向钱包地址发送一些币：



-mine标志指的是块会立刻被同一节点挖出来。我们必须要有这个标志，因为初始状态时，网络中没有矿工节点。

启动节点：



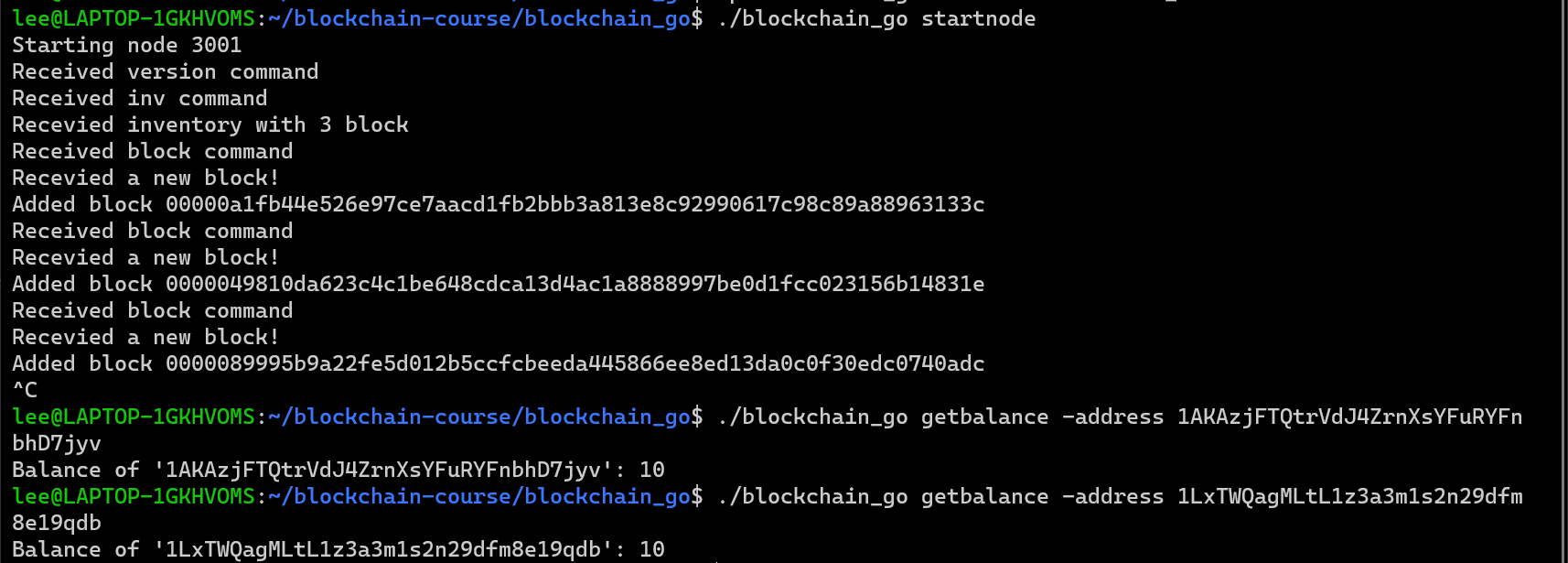
这个节点会持续运行，直到本文定义的场景结束。

#### NODE 3001

启动上面保存创世块节点的区块链，然后运行节点：



它会从中心节点下载所有区块。为了检查一切正常，暂停节点运行并检查余额：

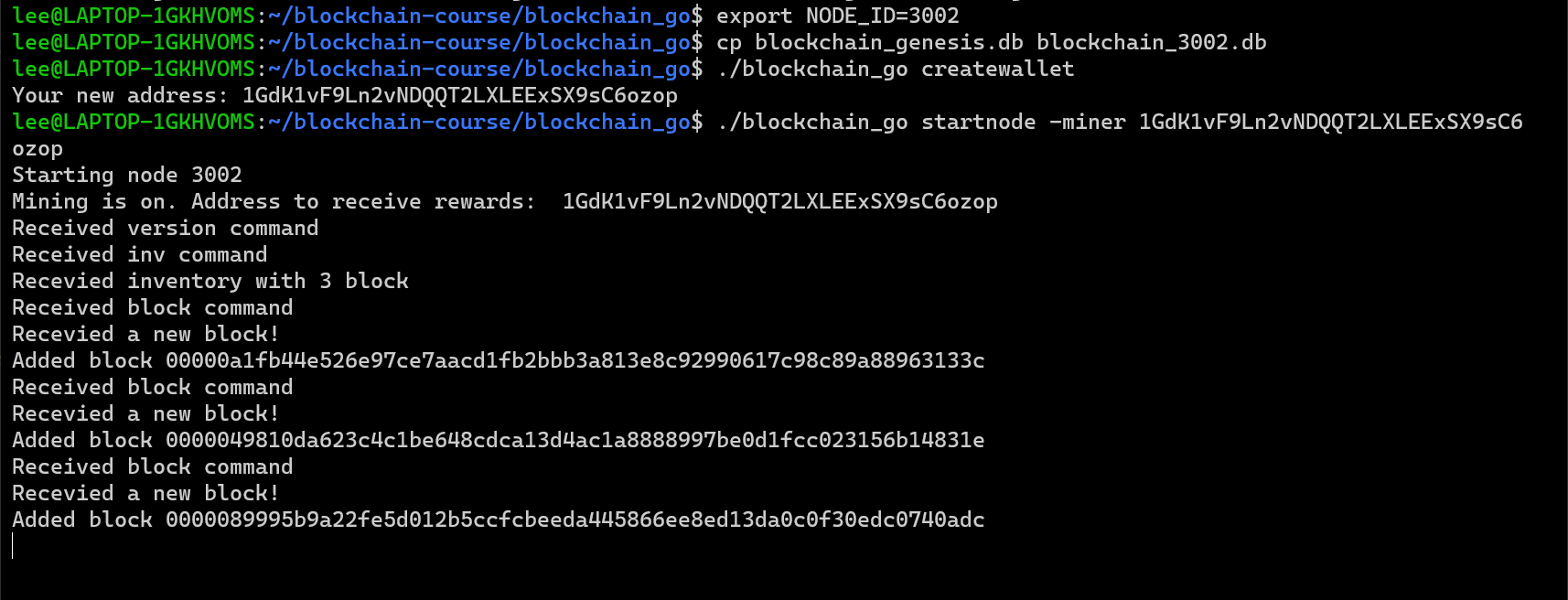


你还可以检查CENTRAL\_NODE地址的余额，因为 node 3001 现在有它自己的区块链：



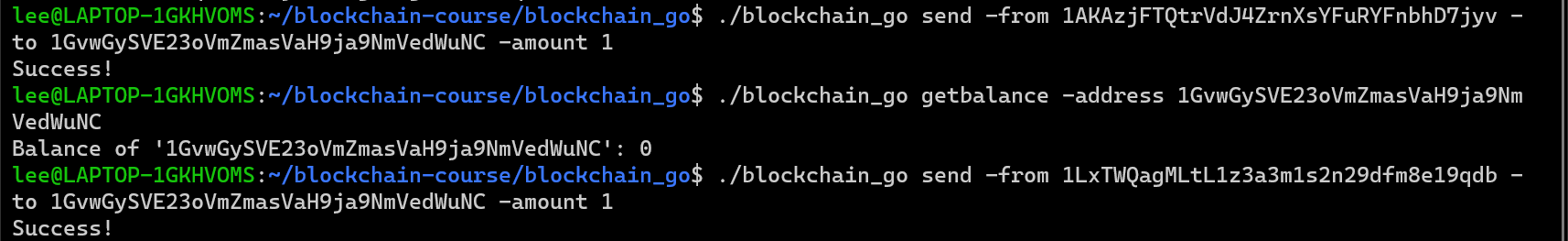
#### NODE 3002

打开一个新的终端窗口，将它的 ID 设置为 3002，然后生成一个钱包。这会是一个矿工节点。初始化区块链，启动节点：



#### NODE 3001

发送一些币：



#### NODE 3002

迅速切换到矿工节点，你会看到挖出了一个新块！同时，检查中心节点的输出。

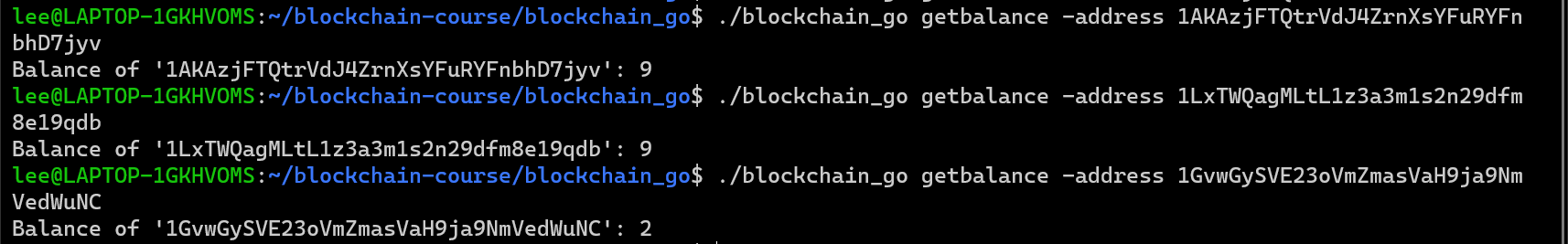


#### NODE 3001

切换到钱包节点并启动节点，它会下载最近挖出来的快：



暂停节点并检查余额：



就是这么多了！

## 7.12 总结

这是本系列的最后了。本文已经回答了关于比特币技术的一些问题，也给读者提出了一些问题，这些问题读者可以自行寻找答案。在比特币技术中还有隐藏着很多有趣的事情！好运！