**实验3 SPV简单支付验证**

**【实验介绍】**

实验2我们实现了交易的生成和验证上链过程，那如何再之后验证一笔交易是否已经上链成功呢，换句话说就是是否已经在网络中达成共识?在BTC中，我们判断一笔交易是否成功是查看它是否在网络的最长链中，当该交易所在区块往后没有分叉并继续添加六个区块后，我们就相信该笔交易被所有的节点承认，即完成共识。对于全节点，它想要查询一笔交易是否在区块链中，只需在本地存储的区块链数据中查找该交易的存在，但目前完整的BTC数据量已高达几百GB，这对于想要轻便交易并确认的节点来说是无法容忍的，如何在不存储完整数据的情况下查询某笔交易是否在链上呢？这只能求助于网络中的全节点，向它询问是否存在这笔交易，但是这样返回的结果也不一定是正确的。

在不可信的网络中，从全节点获取到的验证消息不一定是正确的，所以我们需要对查询的结果进行验证，确保对方无法欺骗我们，且这种验证过程需是简单高效的。对此，中本聪提出了SPV(Simplified Payment Verfication)，即简单支付验证，不运行完全节点也可验证交易，用户只需要保存所有的区块头信息block\_header即可完成验证 。

在实验正式进行前，首先介绍一下SPV节点和SPV验证。

1. SPV节点:

2019年的一个安全漏洞，导致一个公共钱包中的7000枚比特币被盗取，如果用户可以有一个本地钱包来保存跟自己有关的交易就可以大概率地避免这一问题，但是光保存是没用的，我们需要验证信息的真实性，如果像那种全节点一样保存整个区块链网络中的交易来本地验证的话，会存在两个问题，一是数据量是很大的，一般需要几百GB，需要较大的存储空间; 二是一个新用户如果想要加入到网络中的话，需要同步整个区块链网络中的信息，由于数据量过大，同步数据也会需要大量的时间成本; 对于大多数参与比特比网络的用户而言，是没有必要耗费大量的空间和时间成本的来构建全节点的，所以SPV节点就出现了，通过仅仅存储区块链网络中的所有区块头，然后向网络中的全节点申请验证，并通过返回的验证路径和本地存放的区块头信息计算来验证其结果的正确性。SPV节点的出现，使得加入比特币网络变得更容易，同时也使得其使用变得更加便捷，支付验证更加简单快速，同时也间接地帮助了区块链技术的扩展和延伸，例如侧链技术中就有使用到SPV。

1. SPV验证:

当SPV节点想验证某笔交易是否存在时，它将该交易的哈希(如下图3-1中的绿色方块)发送给一个全节点，请求该交易的merkle树验证路径，即图中黄色区块部分，当得到全节点发送过来的验证路径后（说明全节点反馈交易是存在的），SPV节点在本地使用它们重新计算包含该交易的区块的merkle根哈希，与本地区块头里存储的merkle根哈希作比较，判断是否相同，若相同，则该笔交易得到确认，否则说明对方在撒谎。

除了准确性得到保障以外，我们还可以看到其中处理的数据量是交易数目的log(n)数据级，相较于原来全节点那种遍历所有交易来验证的O(n)时间复杂度，效率得到了很大的提升，这使得一些低级终端也可能运行SPV轻节点进行区块链交易。同时SPV还提供一种隐私保护机制，如果当前SPV节点不想让全节点知道自己想要查哪条具体交易，可以通过Bloom过滤器来模糊查询内容后，再询问全节点，最后根据返回值找出自己需要的精确的信息(本实验中没有实现这一保护机制，有兴趣的同学可以自行了解一下bloom过滤器以及SPV节点是如何使用其来做到隐私保护的)。

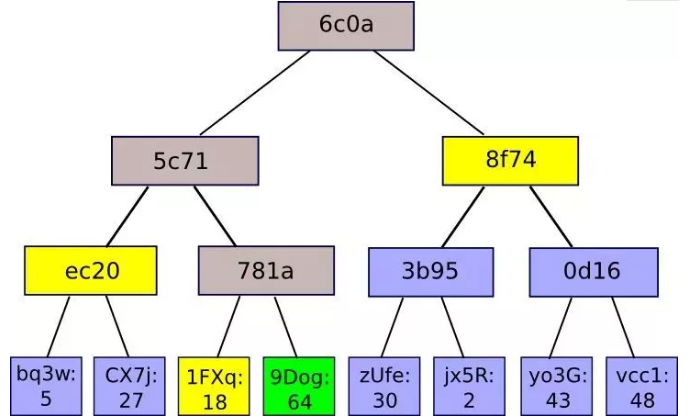


图3-1 merkle树验证

**【实验要求】**

（1）实现spv轻节点模块

（2）实现spv验证流程

（3）实现与旧有框架的交互

**【实验准备】**

**1. 环境配置**

本次实验使用和实验1，实验2相同的开发环境。

**2. 导入代码**

推荐读者使用自己在实验2完成的代码上进行此实验。若仅想学习SPV简单支付验证的工程原理，也可以直接导入附件里实验3的minichain压缩包进行实验，此压缩包含有实验2的参考代码。

**【实验过程】**

本次实验增加了spv轻节点模块及其验证流程的模拟，考虑到多次加入新模块使得旧有代码耦合度不断提高以及之前待解决的问题，我们重构了之前的代码框架，代码功能不变，运行效果和之前相同。

下图是旧有代码框架，空心箭头是逻辑关系，红线指代实际代码流的交互。旧有框架各模块之间的交互都是内部拥有对方的引用（即指针），而网络模块虽有很多模块的引用，但没有执行其作为中介的作用，这在设计上是不合理的（例如区块链里为什么会有用户的引用），且随着模块的增加，我们又得在模块中增加引用， 这会让系统变得复杂臃肿难以理解，如图3-2所示。

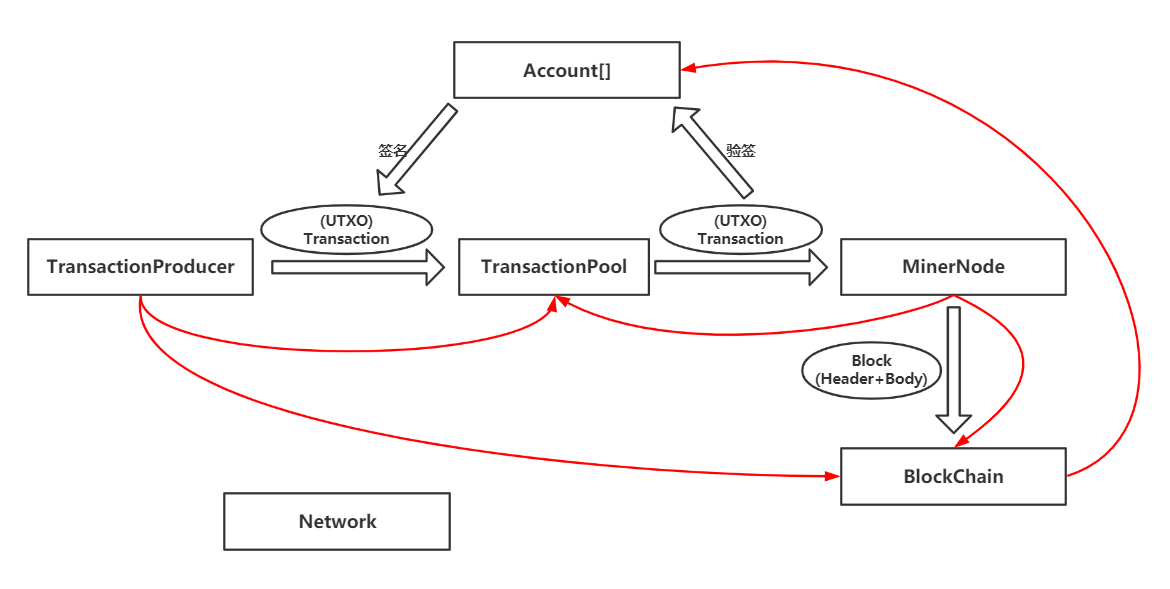


图3-2 旧框架的业务逻辑

而新的代码框架如下图所示（增加了spv轻节点），我们仅保留矿工节点对区块链的引用，因为在我们的系统中区块链存储于矿工节点上。我们切断了其他的引用联系，各模块和网络模块有一个双向引用，这样模块与模块都通过网络来交互，从而降低系统的耦合度，更贴近真实区块链的执行场景，即分布式网络的节点都通过网络来通信，如图3-3所示。（虽然我们这里的网络只是个单薄的类对象）

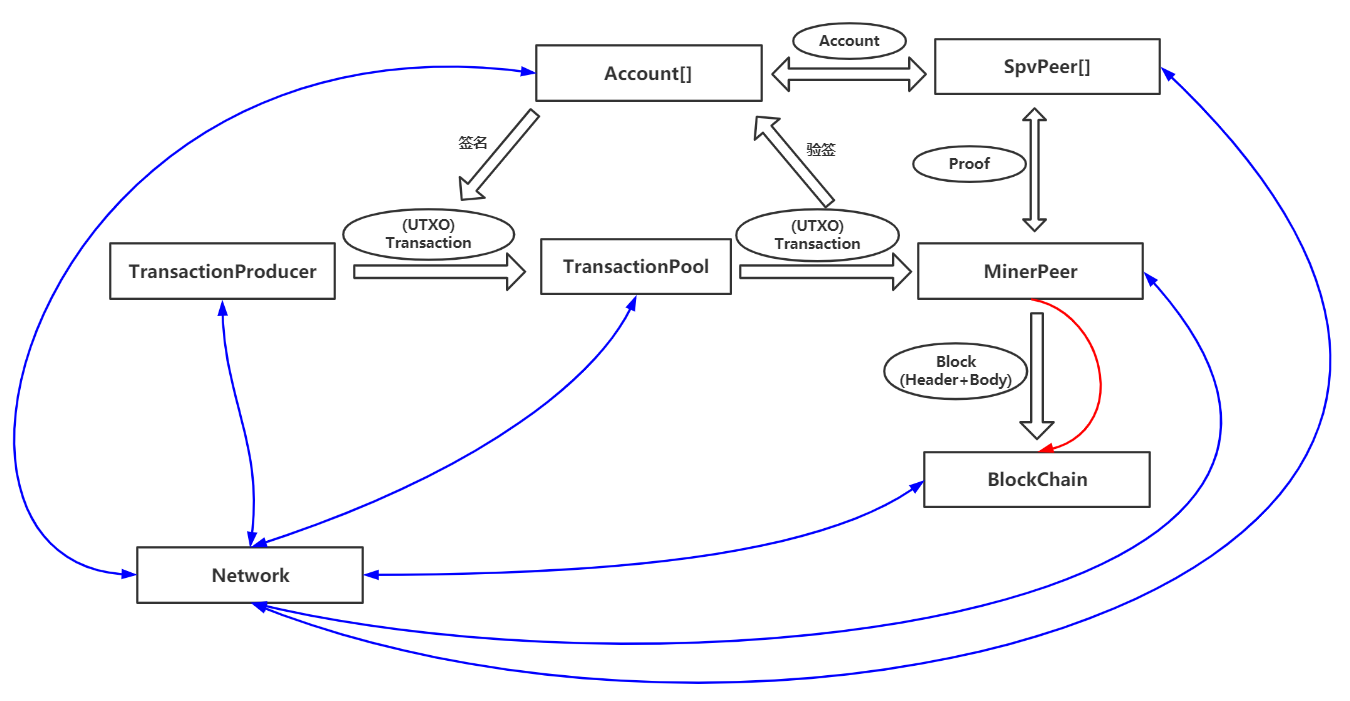


图3-3 新框架的业务逻辑

本次实验将在重构后的代码上完成，下面介绍代码细节：

**1. 重构部分**

（1）使用网络模块作为其他模块通信的中介

首先是网络模块，可将其当成一个虚拟的网络抽象，这个网络里有一些节点：账户（这里暂时只是身份，后续会绑定到spv节点上，这里暂时也将其当作节点）、交易池、交易生产者、矿工及其维护的区块链（这个区块链存储于矿工节点上）。如图3-4所示，在进行构造时，我们将该网络对象的引用（可看作C++的指针）传入到这些节点中，这些节点间的交互都通过网络来获取相应信息和服务（可假定其真的通过了一层网络）。

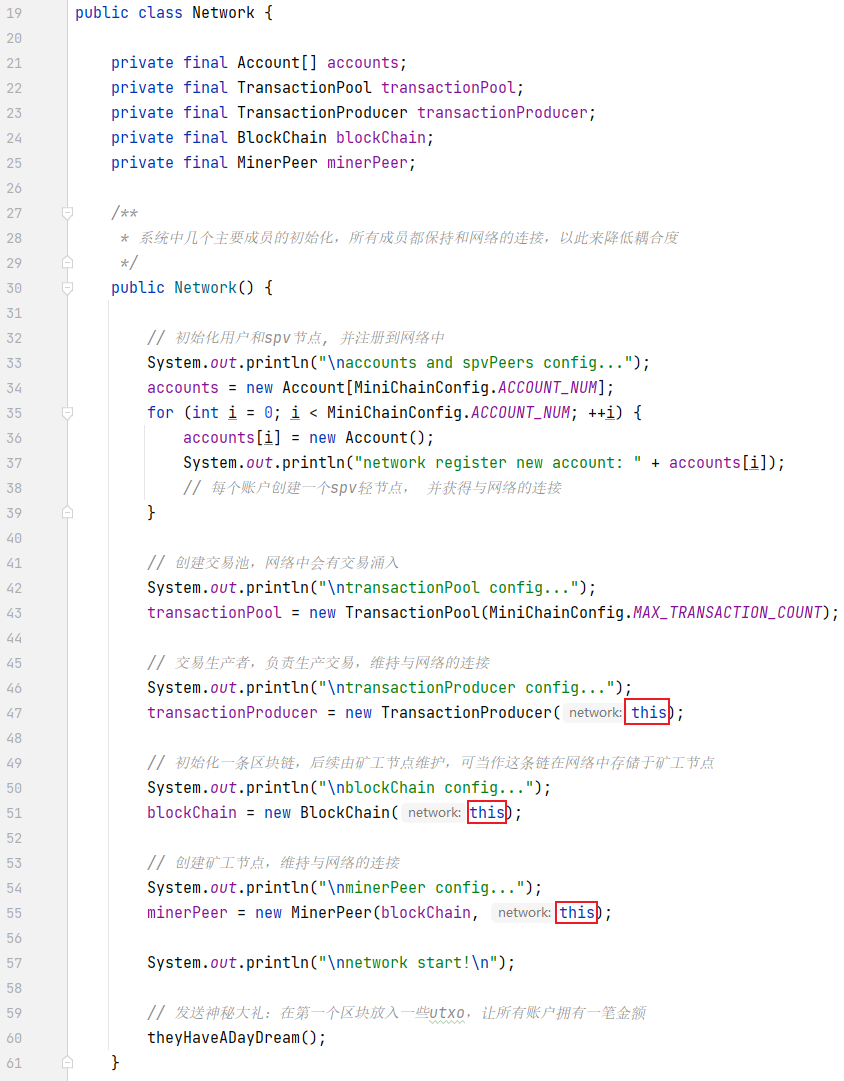


图3-4 Network类

如图3-5所示，矿工通过网络获得交易池信息，从中拿出交易打包成区块：



图3-5 MinerPeer与Network的交互

如图3-6所示，交易生产者通过网络获取账户数组构造交易，并通过网络放入交易池中：

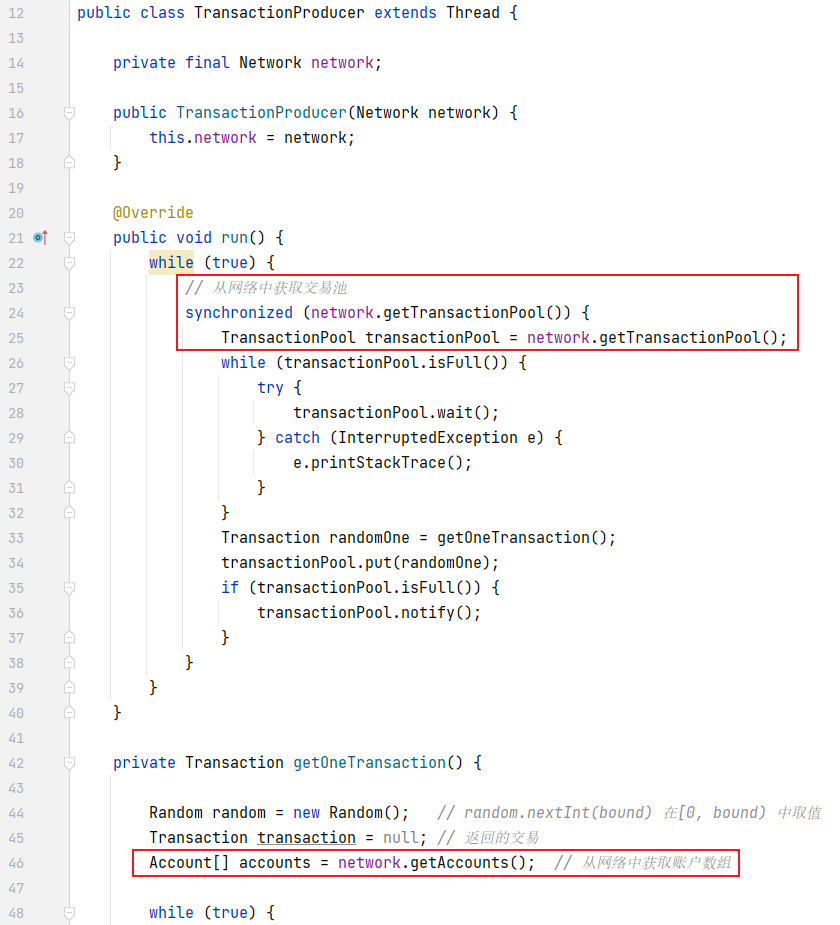


图3-6 TransactionProducer与Network的交互

除此之外还有其他地方对网络模块的使用，可在代码中查看。

通过以上修改，各模块由网络模块作为中介，防止之前那般各自互存一个对方引用的混乱关系，整体上降低了模块间的耦合度。

（2）初始产生utxo的函数由BlockChain类迁移至Network类

（初始产生utxo的函数放在哪都不合适，先将其放在Network类中，如图3-7所示，因为连接着账户数组和区块链对象，放在Network类中操作相对方便一些。

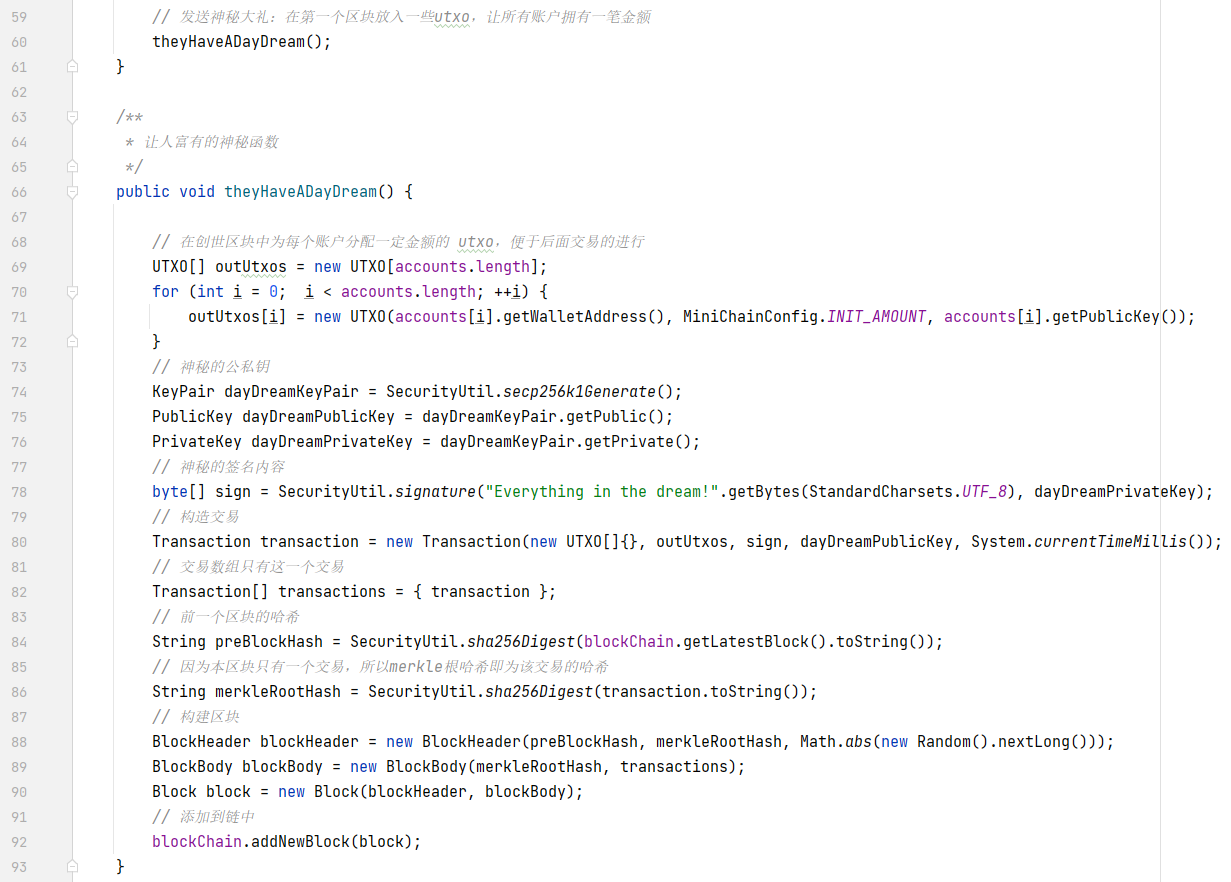


图3-7 在Network中初始化UTXO

（3）解决上次实验只能设定区块交易数为1的问题

这个问题不难解决，只是之前的代码量已经够多，而且相应的修改在逻辑上也不符合常理（本次的解决依然如此）。这里一种方法是给予交易池足够的“灵性”，让它检查放入池中的交易是否已经有池中交易使用的UTXO，如果有，则拒绝该交易入池，如图3-8所示:

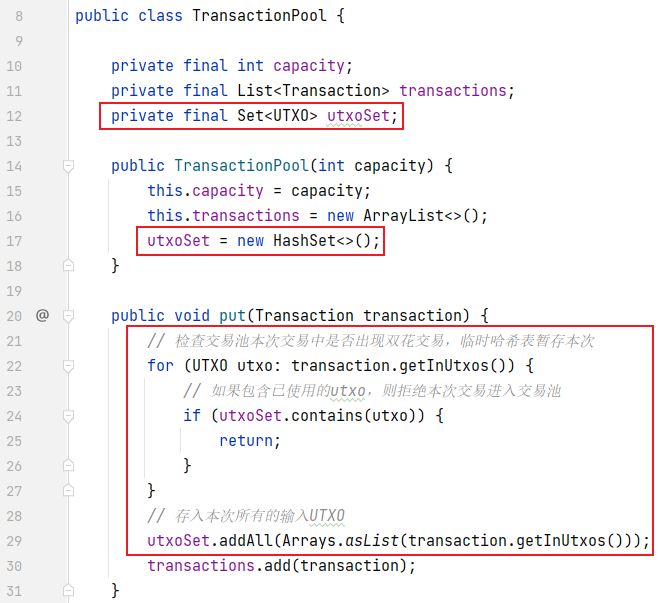


图3-8 设置区块的交易数量

（4）实现更真实的merkle根哈希计算方法，交易数不局限于2的幂（但不能为0）

两两节点可以向上计算哈希合并为一个节点，但如果当前层节点数为奇数时，就会多出一个节点，此时我们将其复制一份并计算，就解决了这个问题，代码如图3-9所示：

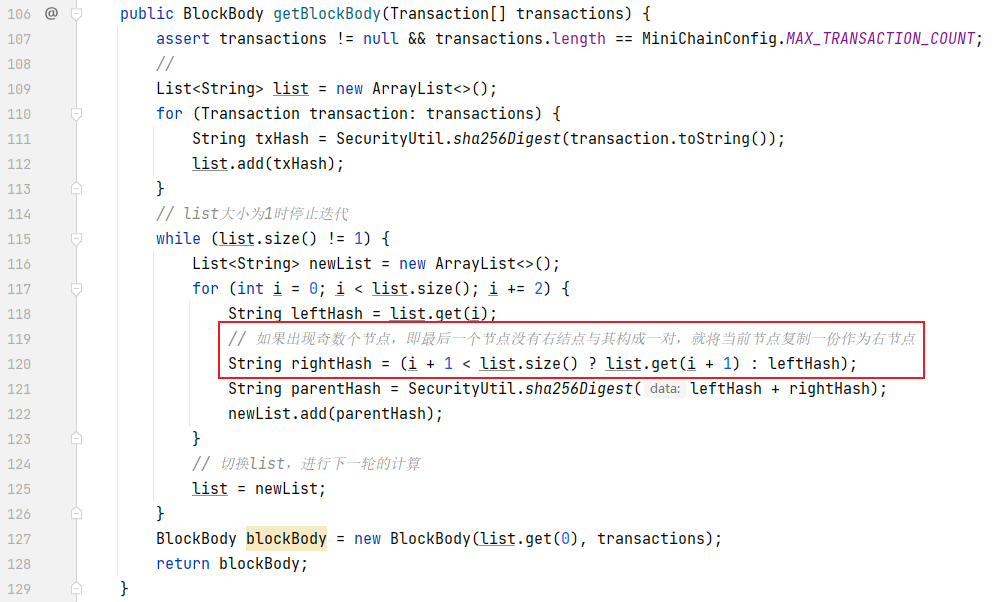


图3-9 Merkle根哈希计算

**2. SPV模块：SPV节点**

（1）SpvPeer类

首先，我们创建spv包（包是Java为了组织应用于某一功能或功能相同的类引入的，如本实验框架的data数据包、util通用工具包，在C++中相当于命名空间，Python中也有包的概念）。

从实验介绍我们知道，spv轻节点只存储区块头数据，同时在本实验中，我们将其关联至一个账户，即每个账户拥有一个轻节点，同时这个轻节点需连接到网络，故创建SpvPeer类如下图3-10所示，其拥有一个接受区块头的函数：

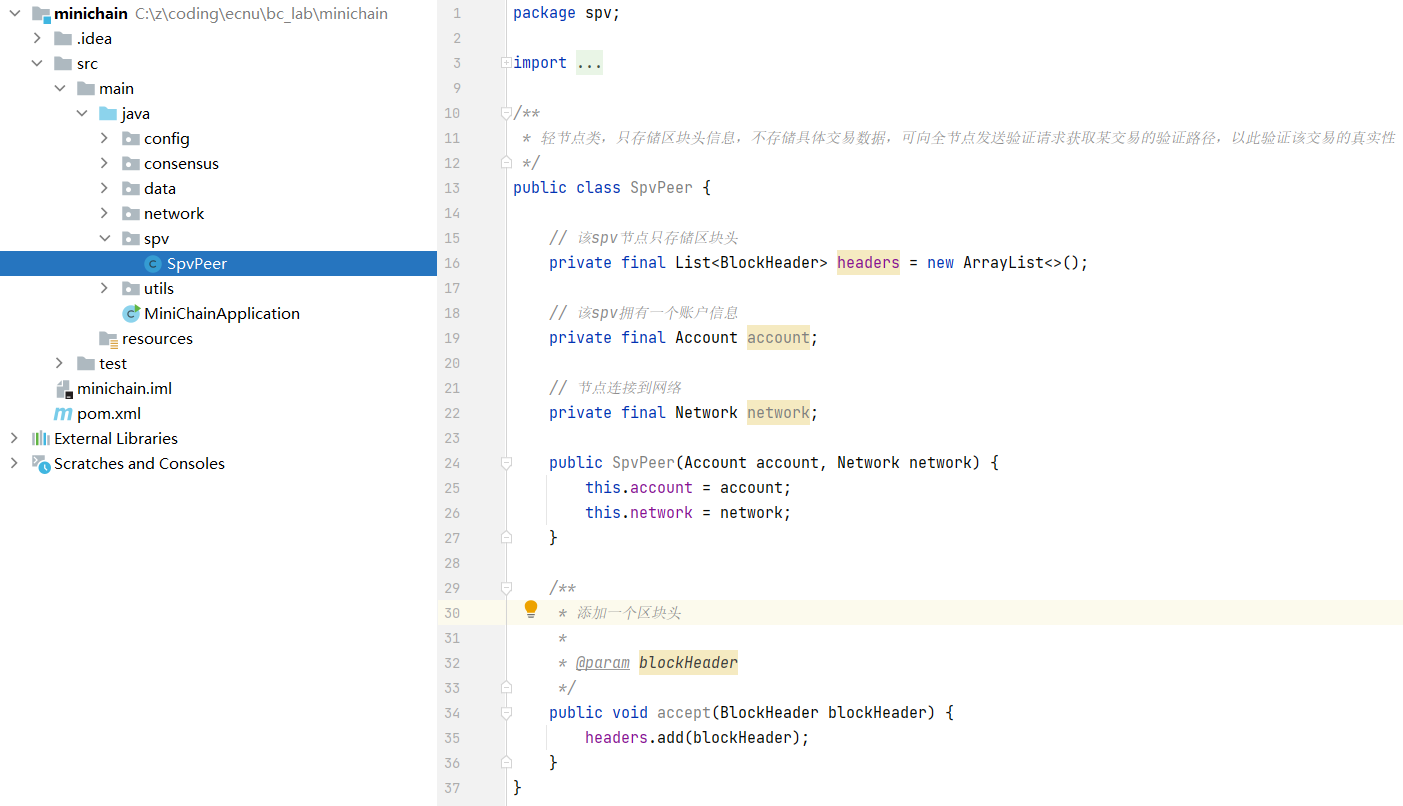


图3-10 Spv节点类

（2）Proof类

下面我们进行简单支付验证流程的设计，从实验介绍中我们知道，spv轻节点会向某个全节点发送交易哈希获取验证路径，即merkle树中的相应哈希值，再由轻节点重新计算根哈希，我们之前的代码由两个子节点计算上层父哈希的方式如下：



那么，我们还需知道每个节点在计算时是左节点还是右节点，即记录偏向信息，这里我们设计一个验证类来管理这些，在spv包下新建Proof类，编写代码如下图3-11，相应的getter函数请使用Alt+Insert快捷键快速生成（包括图片中未显示的Proof类的一些getter函数）。这里我们使用枚举限定路径节点的偏向，内部节点类维持路径节点的两个参数，列表存储路径节点，以及其他相关的数据成员。

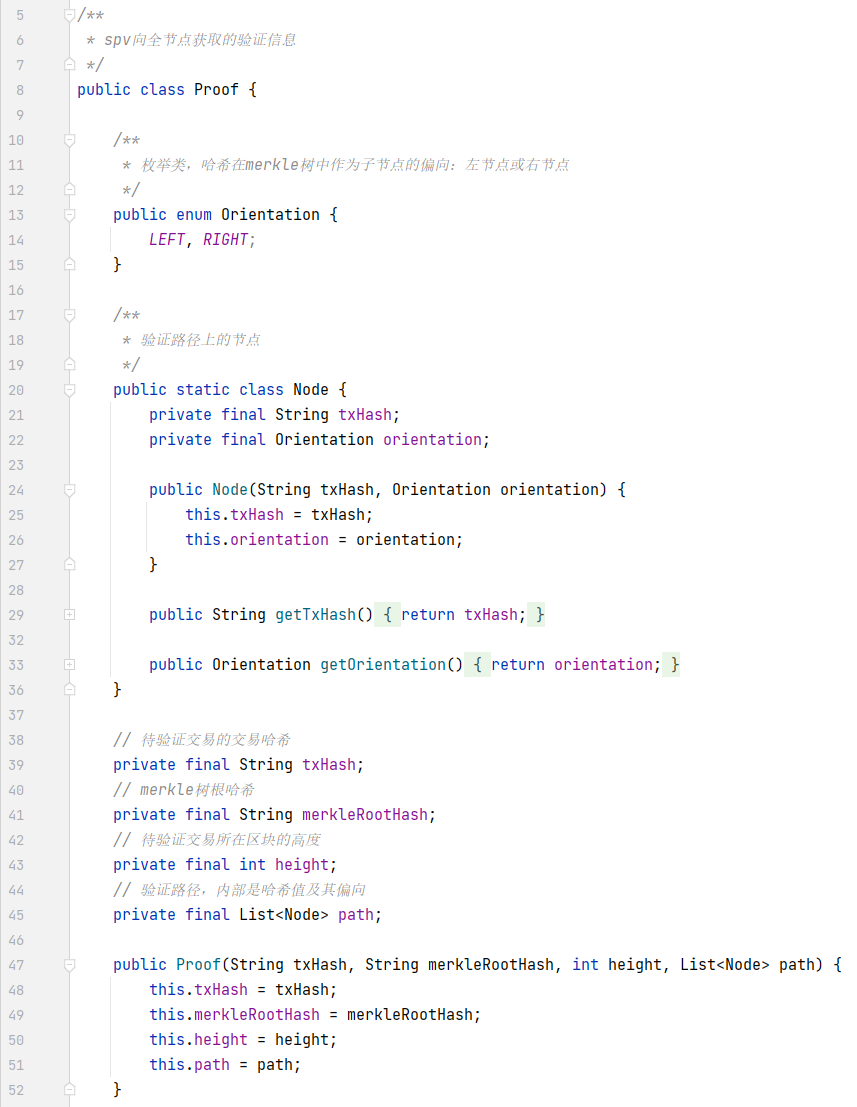


图3-11 Proof类

（3）simplifiedPaymentVerify函数

有了Proof类，我们可以就设计SpvPeer类的验证函数，该函数参数为某个交易。首先计算交易哈希值，向网络中的全节点发送交易哈希请求该交易的验证路径信息，获得验证路径后就开始计算根哈希，最终比较与本地区块头中的根哈希，完成最终的验证，故编写代码如下图3-12所示：（可以抠掉部分重要的代码，添加对应的功能注释，进行代码填空。比如，去掉下图中红框部分的代码）

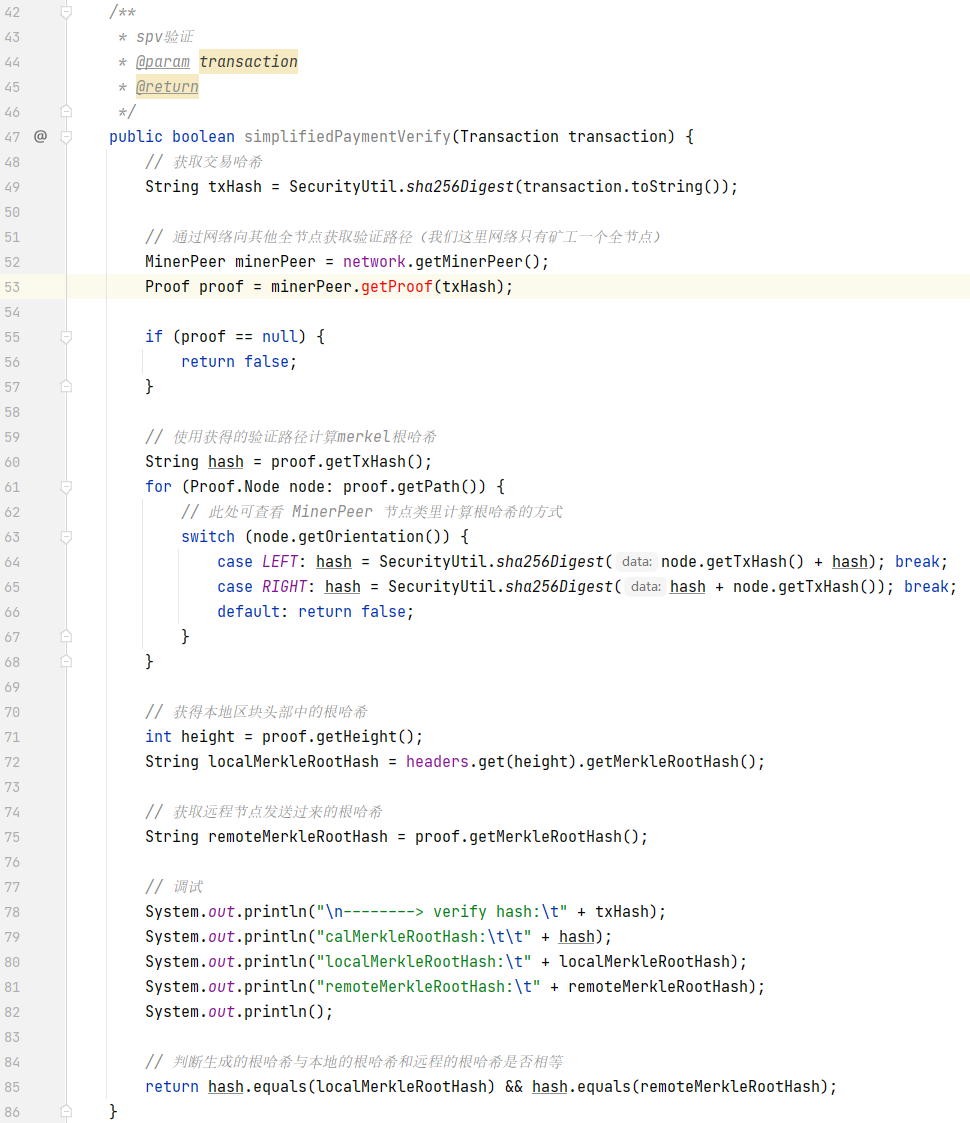


图3-12 验证函数simplifiedPaymentVerify()

（4）getProof函数

该函数依赖于矿工节点，故我们在矿工节点创建相应的函数，该函数返回该交易哈希的验证类Proof。首先需要定位该交易哈希原交易的所在区块，这在现实也是个棘手的问题，我们采用暴力遍历整个区块链中的所有交易并计算其哈希值是否等于要查询的哈希，锁定高度，代码如下图3-13所示：（同上，抠除部分代码）



图3-13 getProof() 函数(上)

然后第二步重新计算merkle根哈希，构建Proof验证路径返回，如下图3-14所示（下图与上图是同一个函数getProof()内）：（抠除代码）



图3-14 getProof()函数(下)

**3. SPV模块与其他模块交互**

（1）spv轻节点数组

编写好相应的类及其函数后，我们开始在网络中加入spv轻节点，并将其关联至账户，在Network类中添加如下图3-15所示代码：

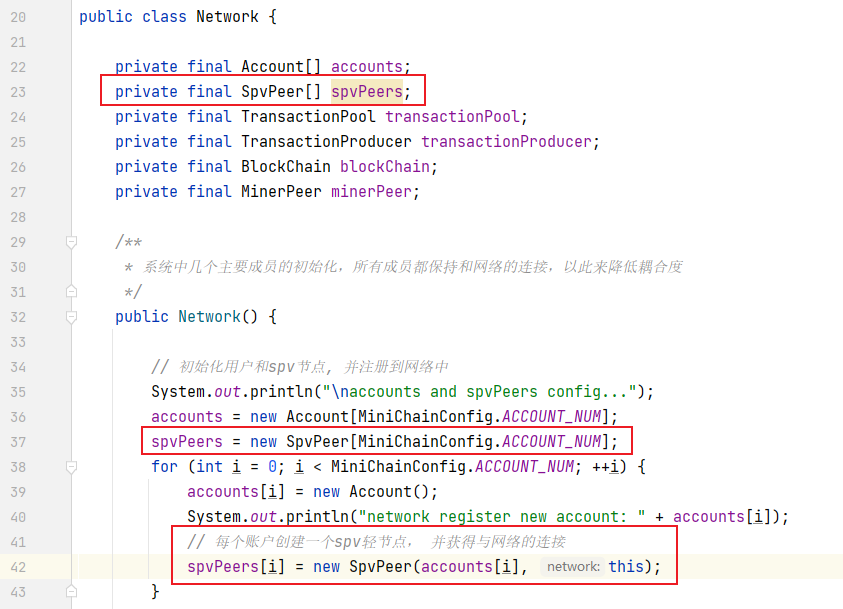


图3-15 Network中的spv轻节点

并添加getter函数：

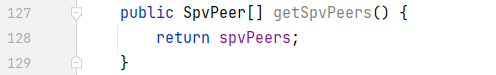


图3-16 添加getter

（2）广播区块头

这些spv轻节点仅存储区块头数据，故矿工每挖出一个区块就需要给网络中的spv节点发送最新区块的头部，在MinePeer节点中添加boardcast函数，其通过network获取网络中的spv轻节点集，将区块的头部数据发送出去，类似于p2p网络中的广播,实现代码如下图3-17所示：

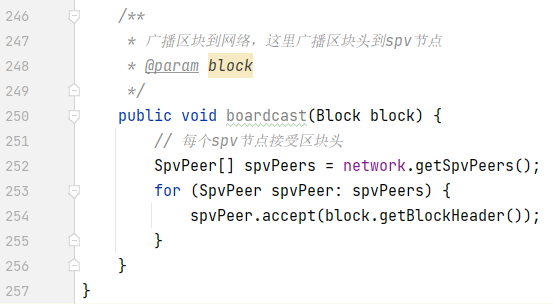


图3-17 矿工节点广播区块头

轻节点中的头部数据始终要保持与矿工中的全链数据同步，故在Network类构造函数中初始化blockChain时，需同步创世区块，如下图3-18：

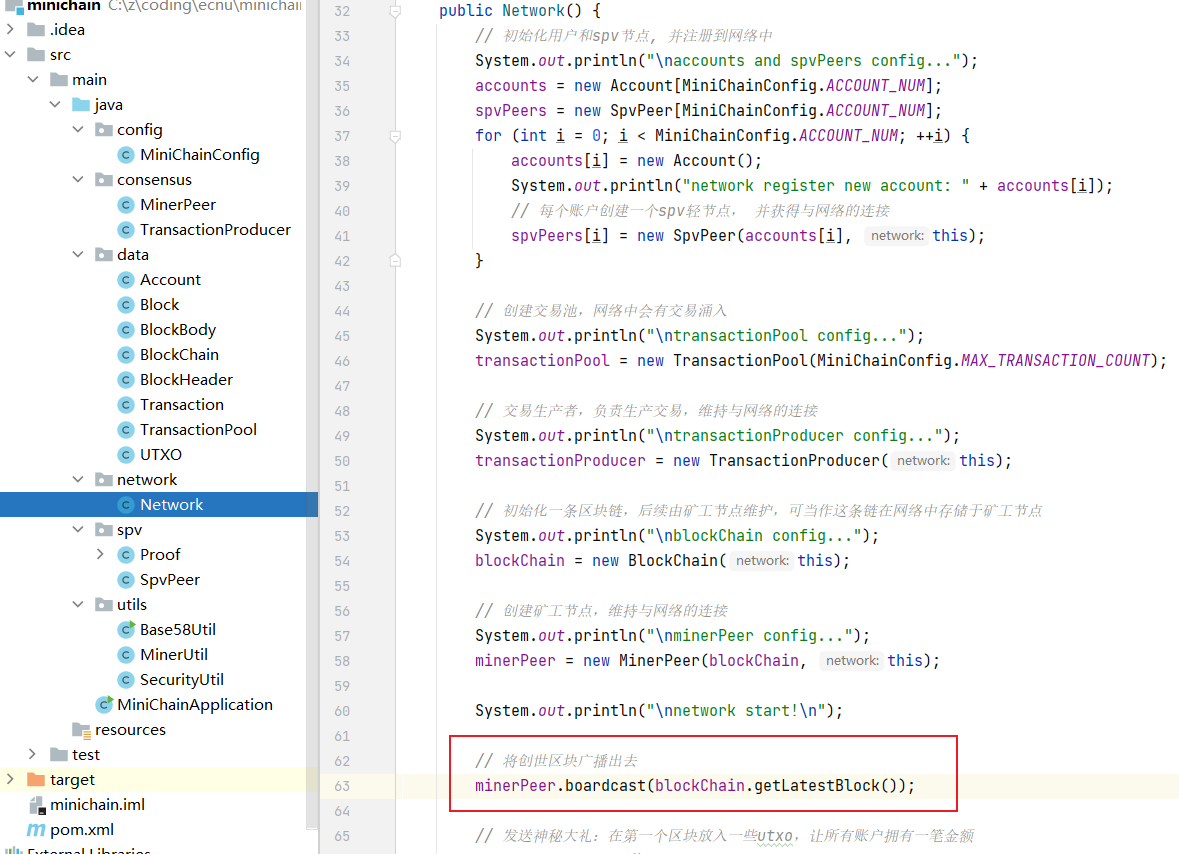


图3-18 广播创世块

对于因构造初始UTXO的第二个区块，需要广播，如图3-19：

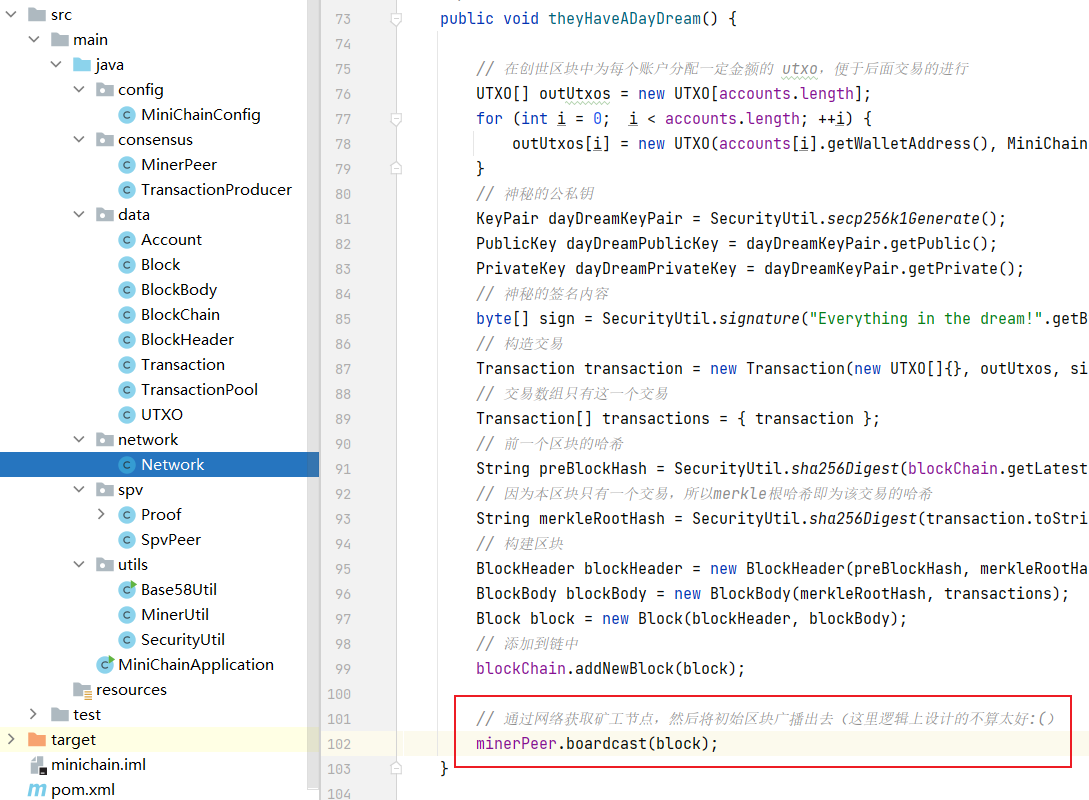


图3-19 广播初始化UTXO的区块

对于矿工每一个挖出的新块，也需要广播，如图3-20：

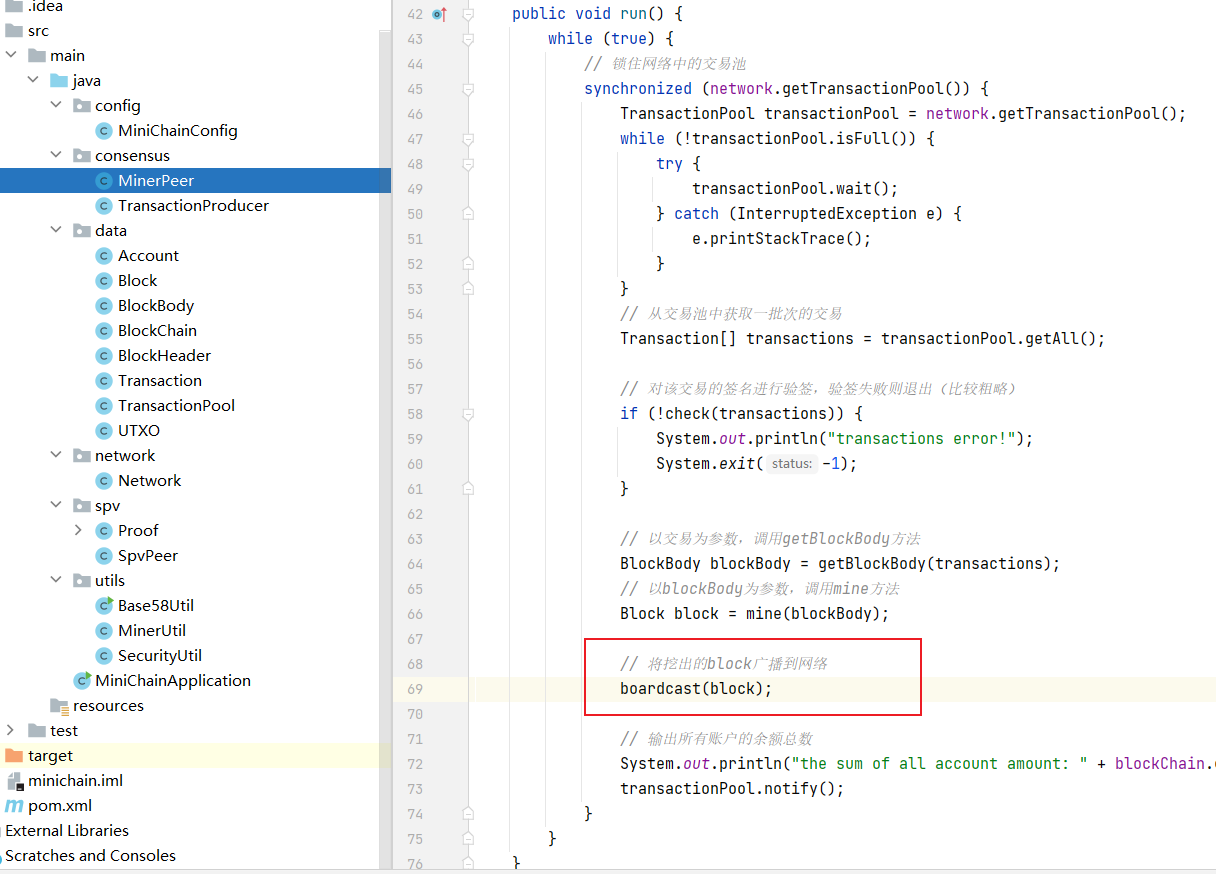


图3-20 广播新挖出来的块

（3）验证相关交易

spv轻节点获得最新块的头部数据后，如果它发现最新块中自己有相关参与的交易，它就可以对这些交易进行验证，首先要获得相关交易，由于交易是随机生成的，只能通过从区块链中获取，这里我们假定能从网络中搜索到与自己相关的交易，故可在网络模块添加获得与某账户相关的交易的函数，其参数为钱包地址，代码如下图3-21所示：（抠除代码）

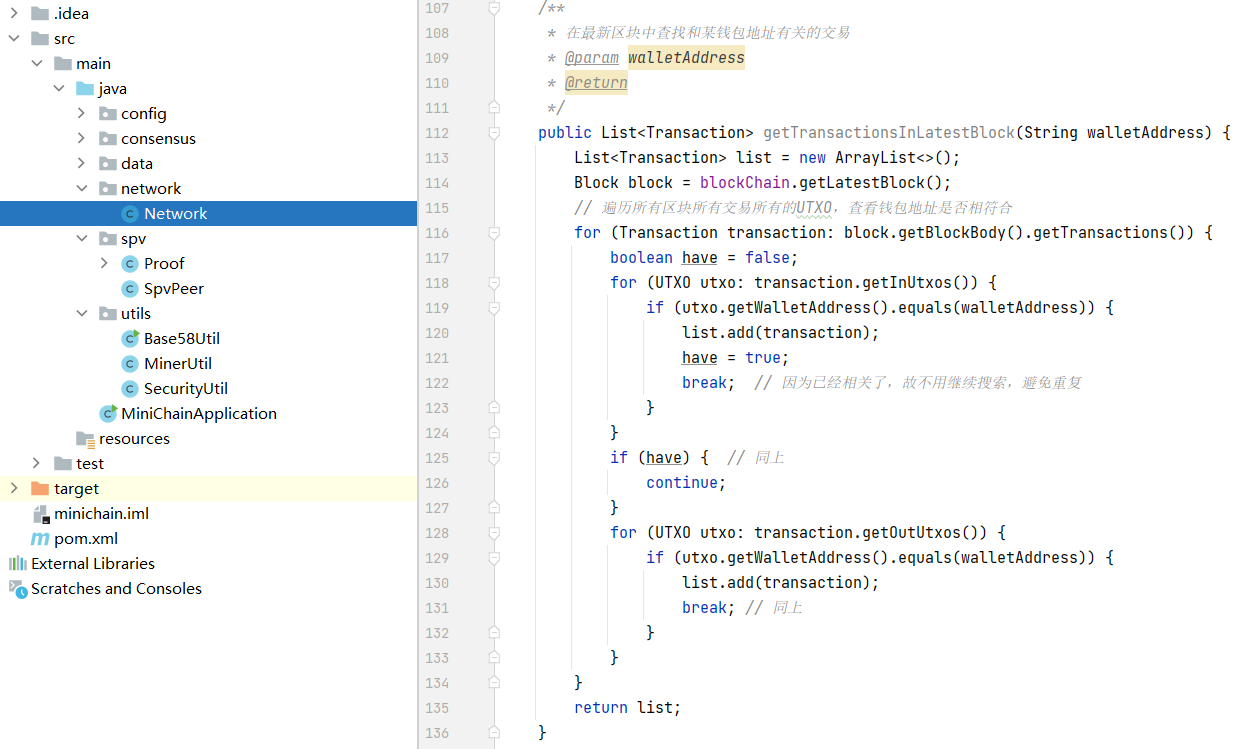


图3-21 获取最后一个区块相关交易函数 getTransacntionsInLatestBlock()

（4）验证

Spv轻节点每次接收区块头时，都检查下最新块是否有相关交易，有的话就一一对其进行验证，如图3-22：



图3-22 验证最新交易函数 verifyLatest()

**4. 实验结果**

运行主程序，可看到验证情况如下图3-23所示：

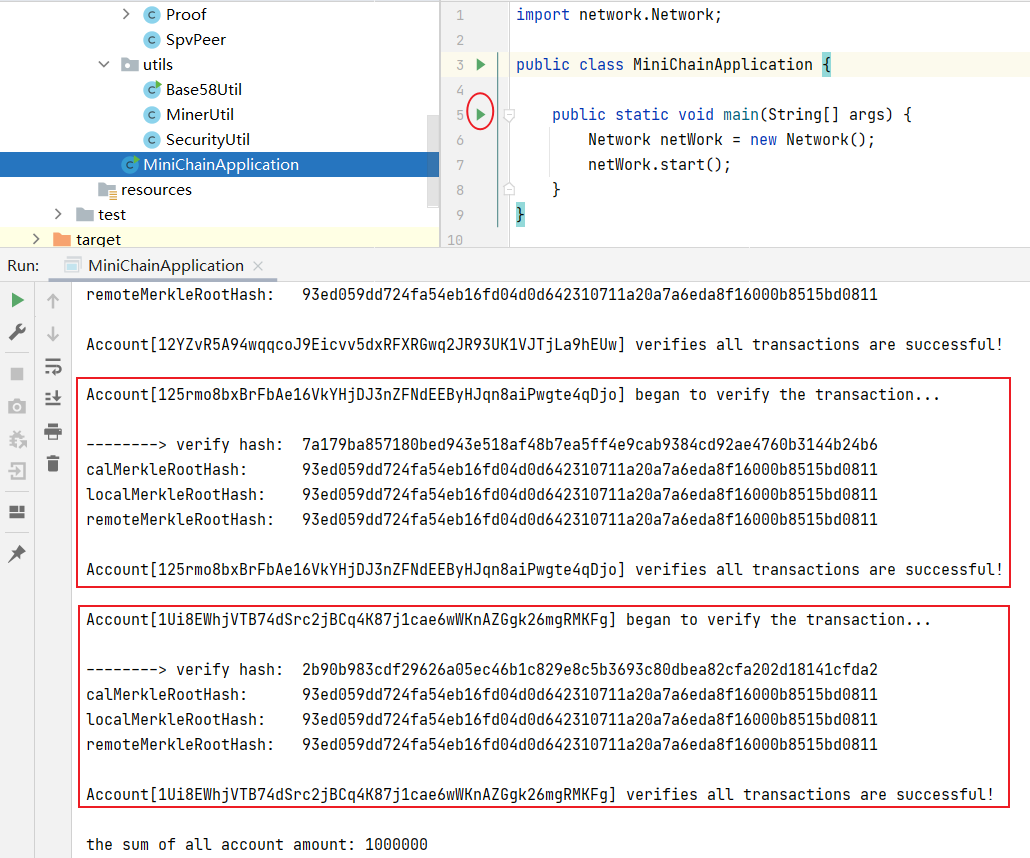


图3-23 程序运行结果

结果显示本地的Merkle根哈希和从矿工节点返回的根哈希以及对返回的验证路径验证后的根哈希三者是一样的，说明该交易是存在的。

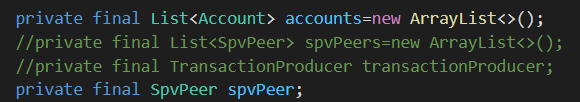
**5. 拓展**

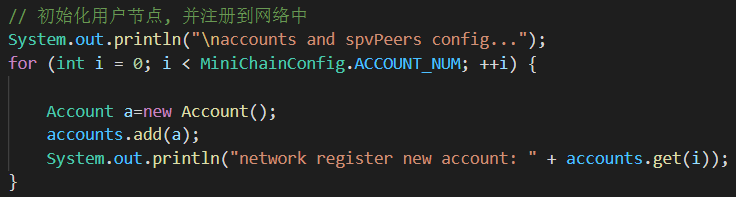
我们通过构建SPV类及其相关的证明函数，向大家展示SPV简单支付验证的流程。而区块链网络中的SPV节点有着更完整的功能和定位，例如比特币钱包，普通用户可以通过创建钱包，作为一个SPV节点加入网络当中，可以通过这个钱包查看自己的账户余额，进行交易和对某笔交易进行验证。

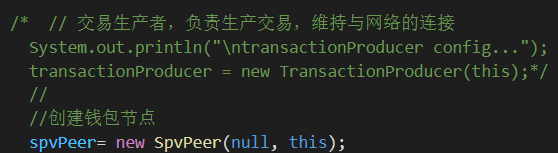
在经过上述的实验流程后，大家对其中的节点间如何交互，区块链网络中信息如何传输，相应函数的内容和作用应该都了较为熟悉的了解。不难发现，如果将TransactionProducer和SpvPeer合并，就可以得到一个类似钱包的SPV用户节点，而在实际的区块链网络中，交易也常常是用户提出的，所以本小节我们就通过改造原实验代码，实现一个钱包功能SPV节点（为了简化，我们的钱包节点也像矿工节点一样作为network进程中的一个线程来运行，方便函数的调用和参数的传递）。

1. 合并TransactionProducer和SpvPeer，重构network类

将TransactionProducer中的线程运行函数run()复制到SpvPeer中，去掉原来的交易生成 部分，然后将Network类中 的SpvPeer和Account 的关联去掉，默认Account是网络中 的其他用户，但是不向网络发送请求，去掉生产者TransactionProducer，单独创建一个 钱包SpvPeer，以下五处修改:

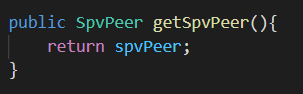




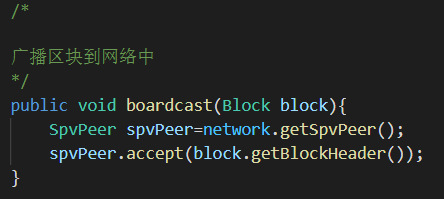


为了减少其他地方的代码修改，这里直接创建了一个钱包节点。因为前面区块生成时如果该节点还未生成的话，矿工节点没有地方可以广播区块头，且在SpvPeer创建后还需要向网络请求以前的区块头信息。并且这个创建过程还需要network监听，比较麻烦。故这里直接创建一个没有Account的SpvPeer，后续通过相应函数真正的创建一个钱包节点，即写入Account。



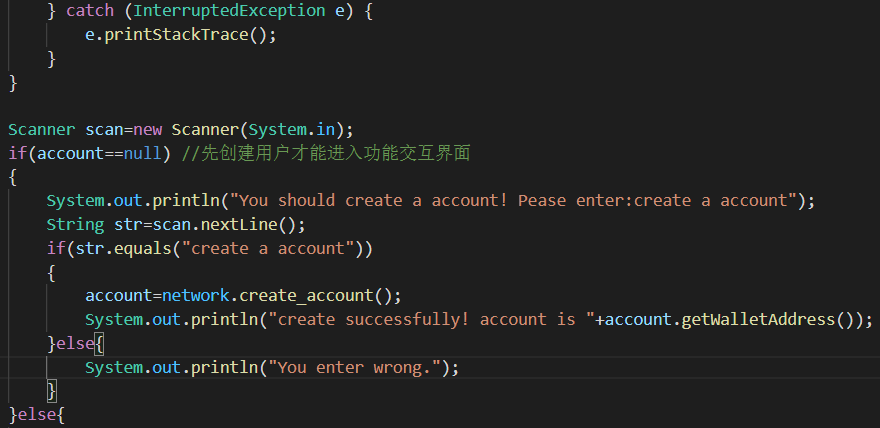


修改MinerPeer中的broadcast()：



1. 设计SpvPeer线程

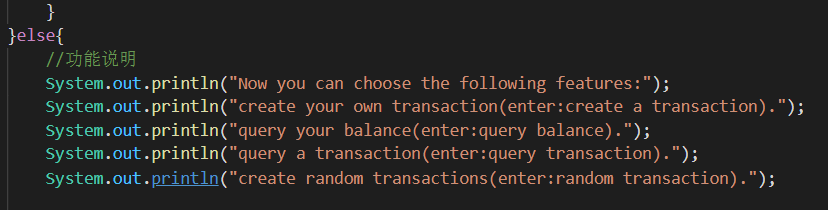
首先是创建账户，我们通过在终端键盘输入来模拟用户与钱包客户端的交互，代码 如下:



同时在network中构造create\_account()函数，其逻辑和一开始创建的初试账户的过 程类似，初始化给予一定金额，生成一个新的区块，代码如下：



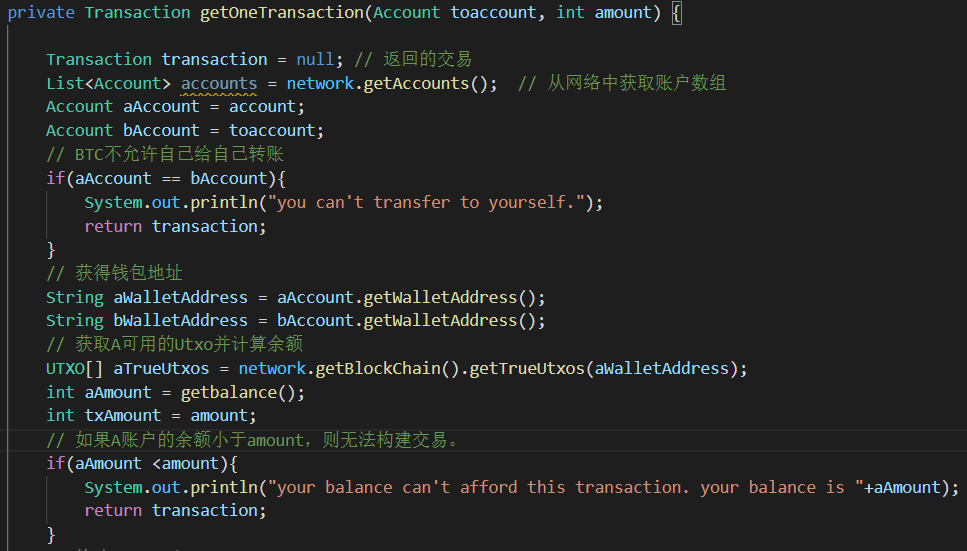
功能添加，三个主功能和一个辅助功能，如下：



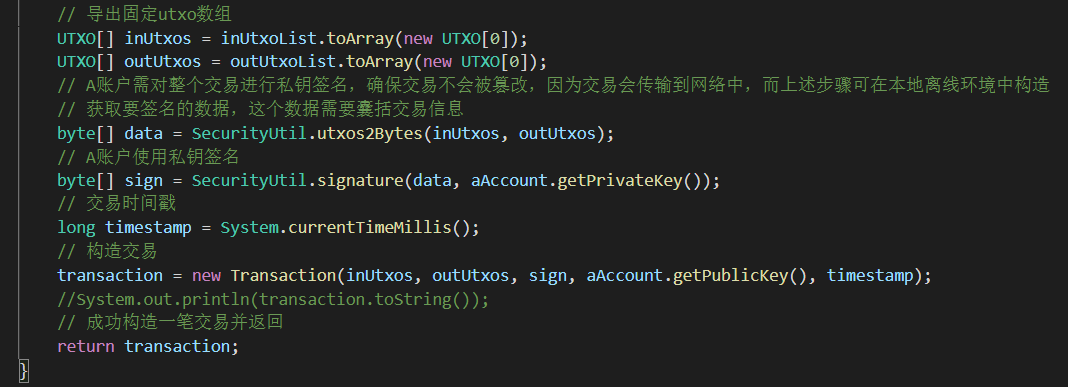
创建转账交易:



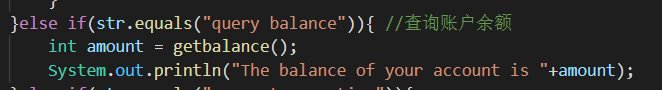
这里我们对原TransactionProducer中的getOneTransaction进行了改写，添加了两个 参数转账的账户对象Account(因为网络中维护了一个账户List，为了简便就直接采 用输入下标获取账户实例)，大体上变动不大，如下代码:



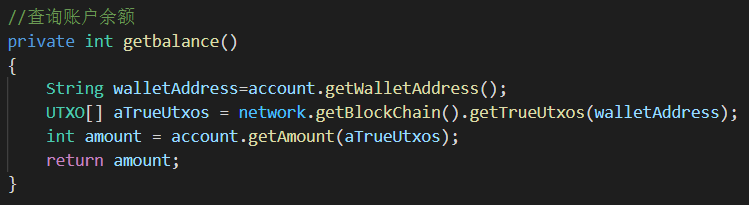




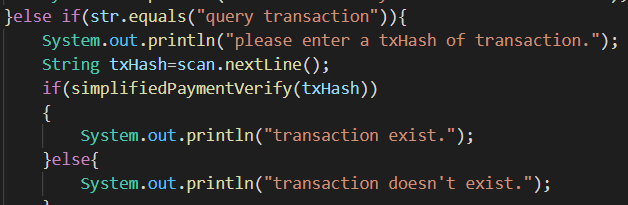
查询账户余额：



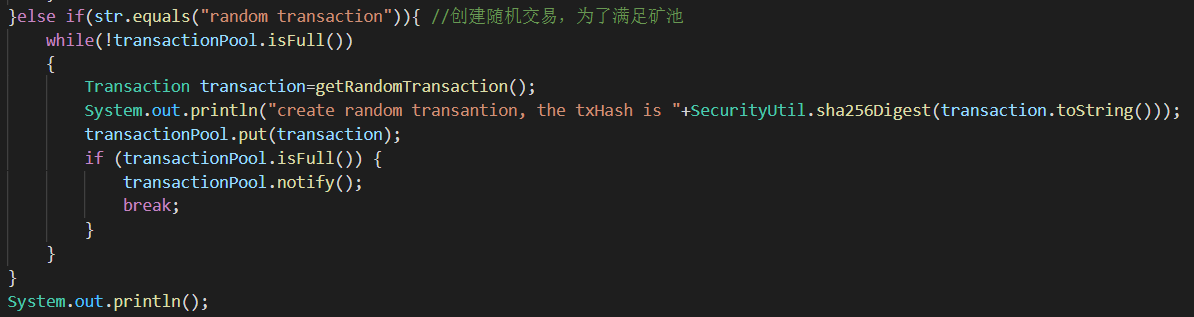
getbalance()函数，逻辑和创建交易中获取账户余额一致，代码如下：



验证交易，直接调用simplifiedPaymentVerify函数即可，修改一下该函数的参数为 String，同时也在verifyLatest()函数中做出相应更改或者可以直接删除该函数（该 函数只是为了展示前面实验进行SPV验证的结果而设计的）：

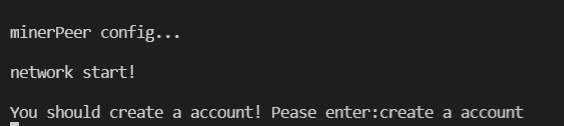


辅助功能，创建随机交易，直接复制原来的getOneTransaction()函数改个名字 getRandomTransaction()即可：

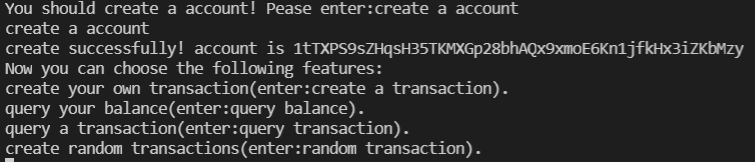


1. 实验结果

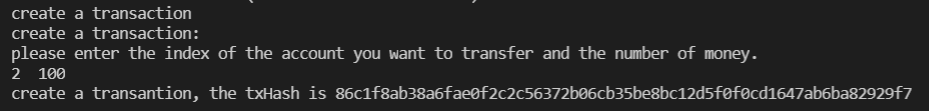
运行程序，初试界面如下：



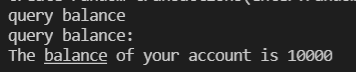
终端键盘输入create a account，出现如下图的功能界面，可以记录一下你账户地址， 后面可以查看区块中的交易是否是该地址发出的：

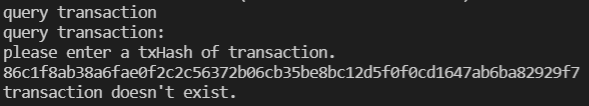


终端键盘输入create a transaction，并输入你要转账的账户下标和转账金额（2 100）， 出现以下结果截图，可以记录该交易在后续区块生成信息中查看有无或用于测试查 询交易的功能：

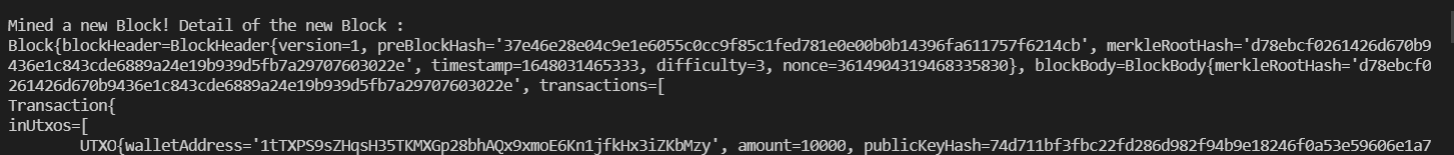


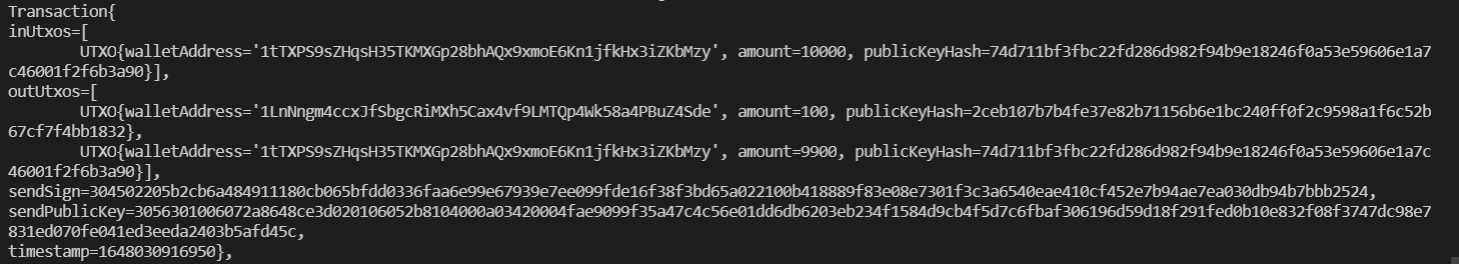
终端键盘输入query balance ,查询到你的账户余额，为什么余额没有改变，大家应 该都知道是因为交易还在交易池中没有被确认上链，我们可以通过query transaction 并输入上述的txHash，会发现网络返回给我们交易不存在。



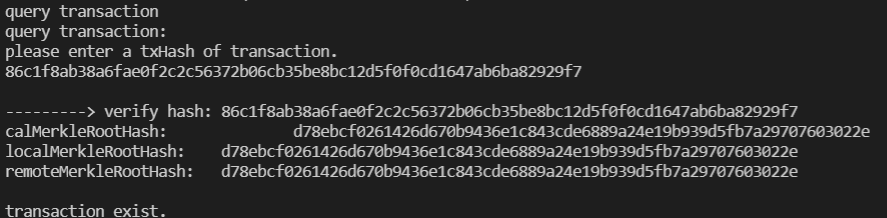


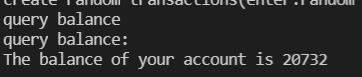
为了使得交易上链，我们终端键盘输入random transaction（那为什么我们不能再通 过该账户创建消息来使得交易池填满呢，大家可以思考一下），生成新的区块。我 们会发现我们的交易在该区块内的有记录。





最后再查询一下这条交易以及现在的账户余额，返回交易存在，区块头也是正确的 区块头，账户余额也正确。





还有其他交易中涉及该账户。

**【实验小结】**

本实验以共识作为切入点，首先说明了原有的全节点通过遍历整个区块链来验证交易的过程是需要很大的时间和空间花费，这种花费对于大部分参与区块链网络中的普通用户而言是难以负担且没有必要去负担的，因此这才有了中本聪在比特币中提出的SPV简单支付验证流程以及SPV节点的可行性。实验就SPV节点和SPV简单支付验证这两个概念进行了说明，并将其添加到前面实验原有的区块链网络中进行实操，通过相关代码，逻辑说明和实验结果图，让大家对于这两个概念及其实现的过程有了一定的了解。

**【习题】**

1. 为什么要使用SPV简单支付验证？

2. 为什么SPV简单支付验证可以保证其准确性（从其应用节点存放的数据结构和验证过程来说明）？

3. 全节点有没有可能既欺骗了我们，又保证最终计算的merkle根哈希与区块头中相同呢？

**【参考文献】**

1. 杨保华. 区块链原理, 设计与应用[M]. 机械工业出版社, 2017.
2. Andreas M. Antonopoulos. Mastering Bitcoin(精通比特币). 东南大学出版社
3. 邹均，张海宁，唐屹，李磊等. 区块链技术指南. 机械工业出版社