实验三 SPV验证

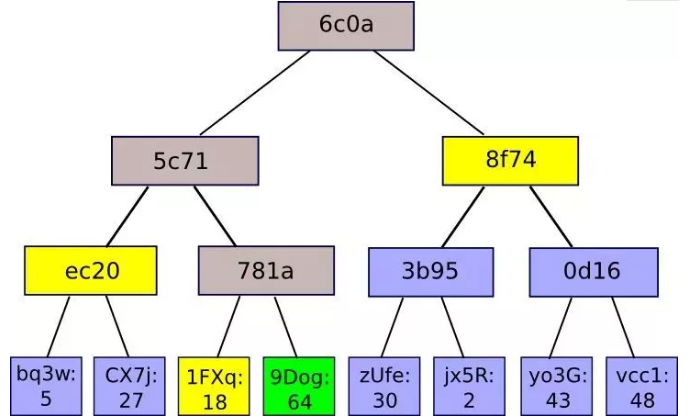
【实验介绍】

在BTC中，我们对于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，它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。

在不可信的网络中，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。对于SPV轻节点，它只存储区块头信息，数据量非常小，区块头包含该区块中交易哈希生成的merkle树根哈希，是与区块体中交易数据密切关联的唯一信息。

当spv轻节点想确认某笔交易时，它将该交易的哈希（如下图中的绿色方块）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，spv轻节点在本地使用它们重新计算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。但是，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？答案是否定的，用下图举个例子，查询的交易哈希和最终的merkle根哈希不变，全节点就需要杜撰出三个虚假的哈希通过层层double sha256哈希（BTC的计算规定）达到一样的根哈希。哈希摘要是单向映射算法，冲突的可能性本身已无限接近于0，我们平时在挖矿时达成某个哈希结果的前固定比特位为0已经是极其困难之事，而想要在多次哈希后仍保证全部比特位相同就更加不可能。除此之外，spv轻节点会向多个全节点查询交易信息确保查询结果的准确性。

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，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。



【实验要求】

（1）实现spv轻节点模块

（2）实现spv验证流程

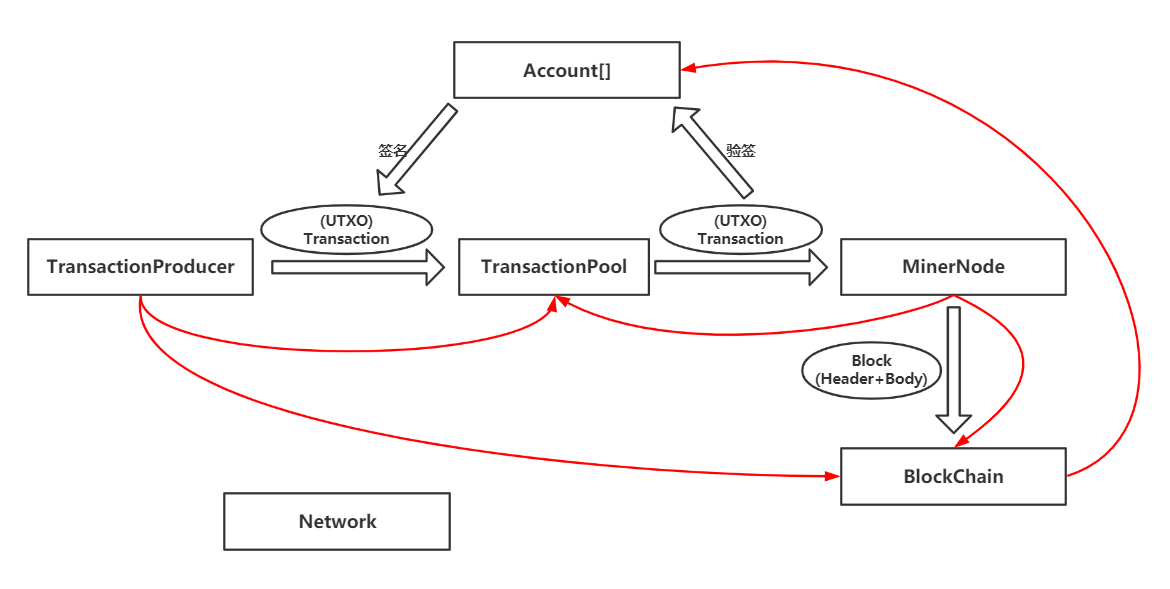
（3）实现与旧有框架的交互

【实验过程】

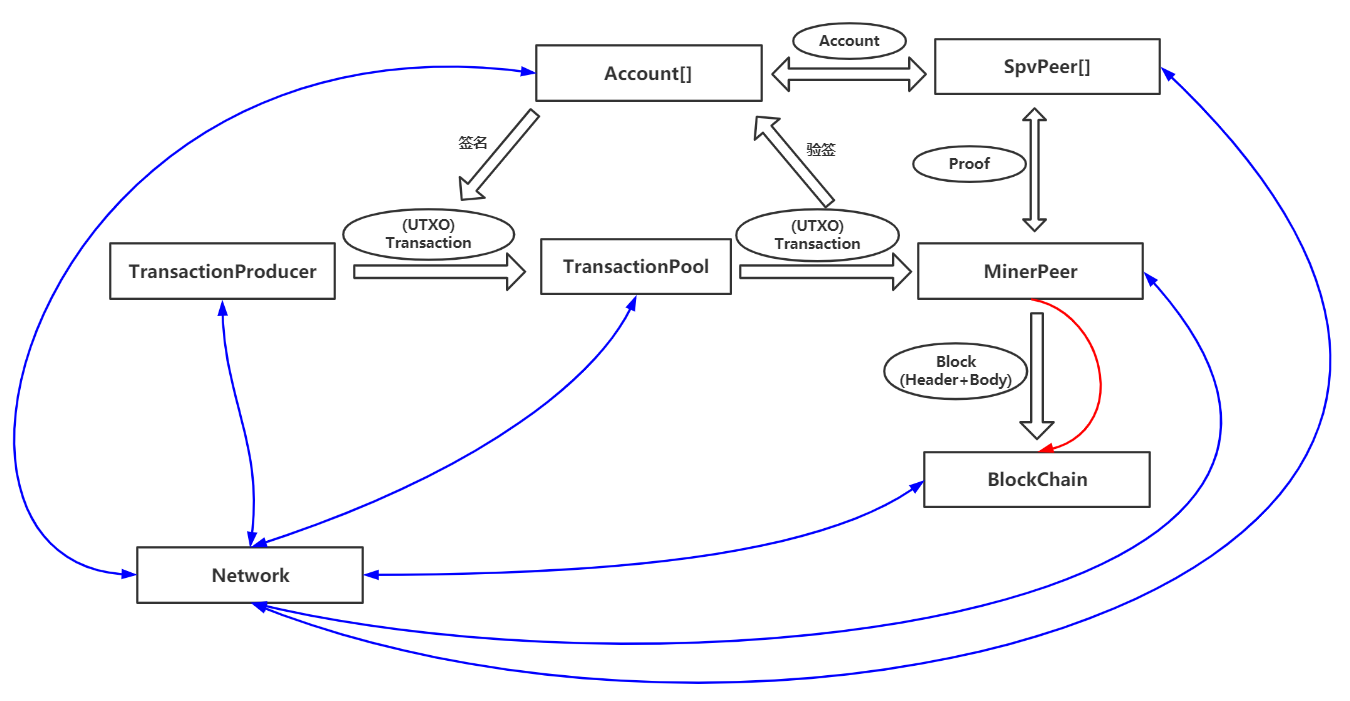
1. 旧有代码的简单重构

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。



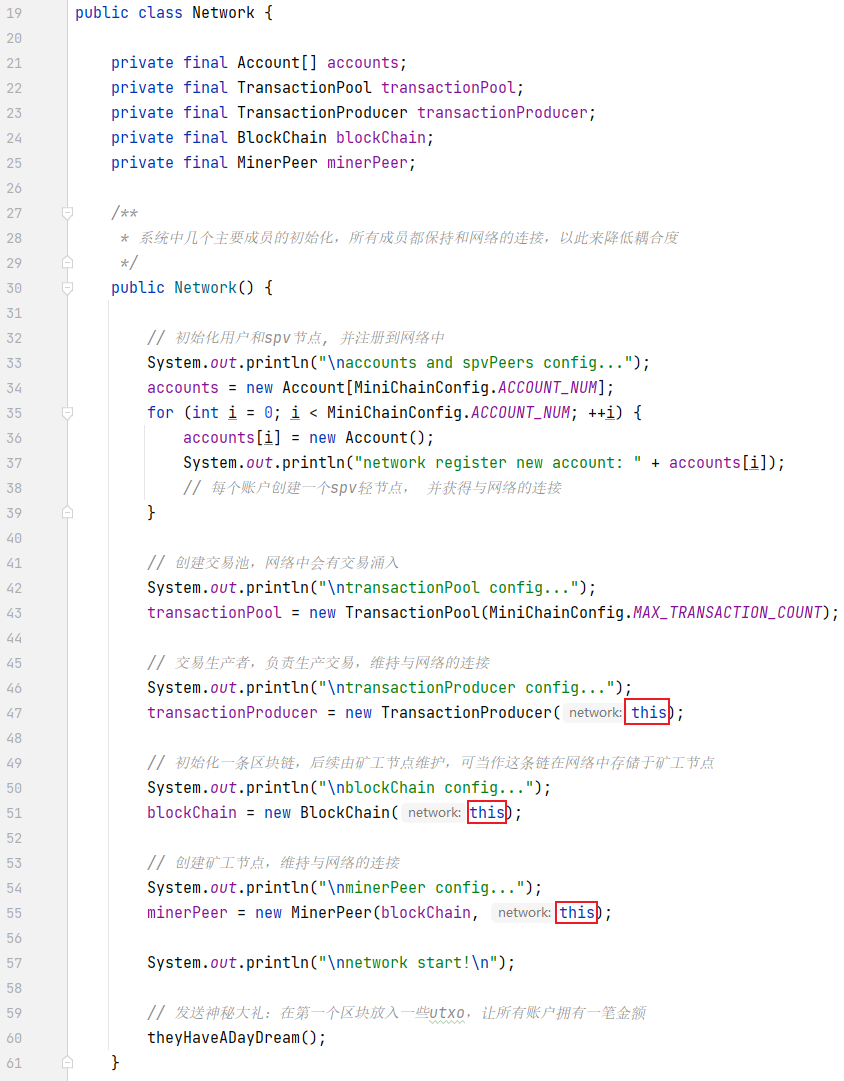
而新的代码框架如下图所示（增加了spv轻节点），我们仅保留矿工节点对区块链的引用，因为在我们的系统中区块链存储于矿工节点上。我们切断了其他的引用联系，各模块和网络模块有一个双向引用，这样模块与模块都通过网络来交互，从而降低系统的耦合度，初次之外，更贴近真实区块链的执行场景，即分布式网络的节点都通过网络来通信。（虽然我们这里的网络只是个单薄的类对象）。



本次实验将在重构后的代码上完成，下面介绍下一些修改的代码细节：

1. 使用网络模块作为其他模块通信的中介

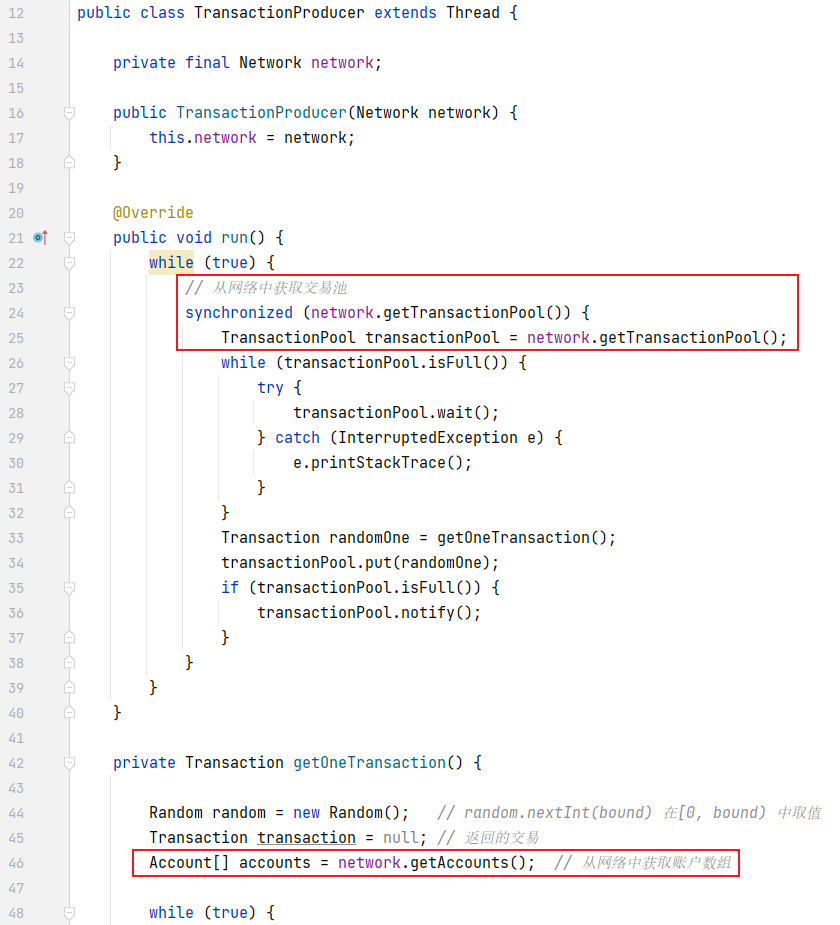
首先是网络模块，可将其当成一个虚拟的网络抽象，这个网络里有一些节点：账户（这里暂时只是身份，后续会绑定到spv轻节点上，这里暂时也将其当作节点）、交易池、交易生产者、矿工及其维护的区块链（这个区块链存储于矿工节点上）。如下图所示，在进行构造时，我们将该网络对象的引用（可看作C++的指针）传入到这些节点中，这些节点间的交互都通过网络来获取相应信息和服务（可假定其真的通过了一层网络☺）。



如下图所示，矿工通过网络获得交易池信息，从中拿出交易打包成区块：



再如交易生产者，也是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：



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。

1. 初始产生utxo的函数由BlockChain类迁移至Network类

这种奇异现象放哪里都不合适，只能放在Network类中，因为它连接着账户数组和区块链对象，操作相对方便一些：



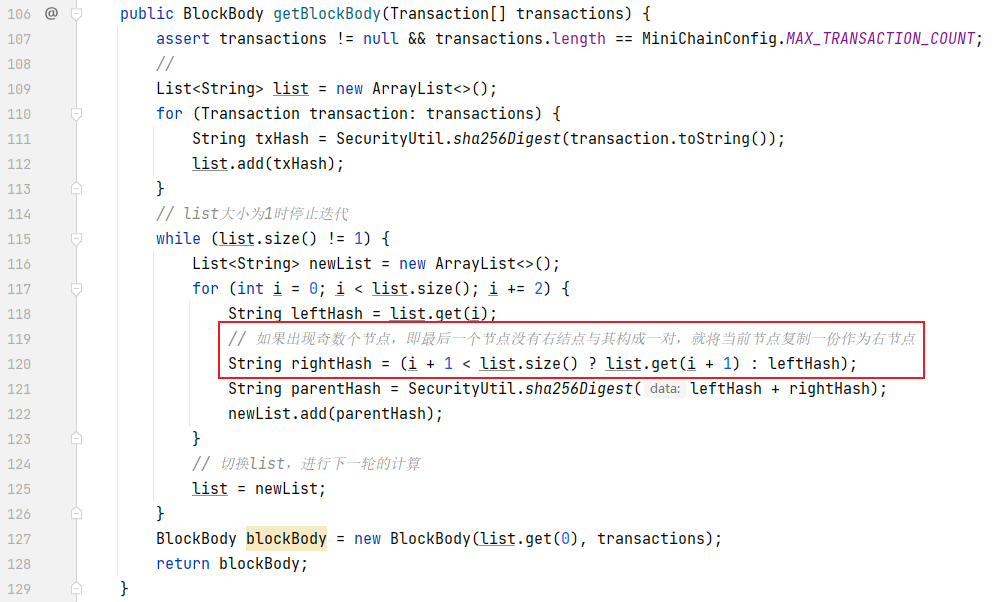
1. 解决上次实验只能设定区块交易数为1的问题

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。



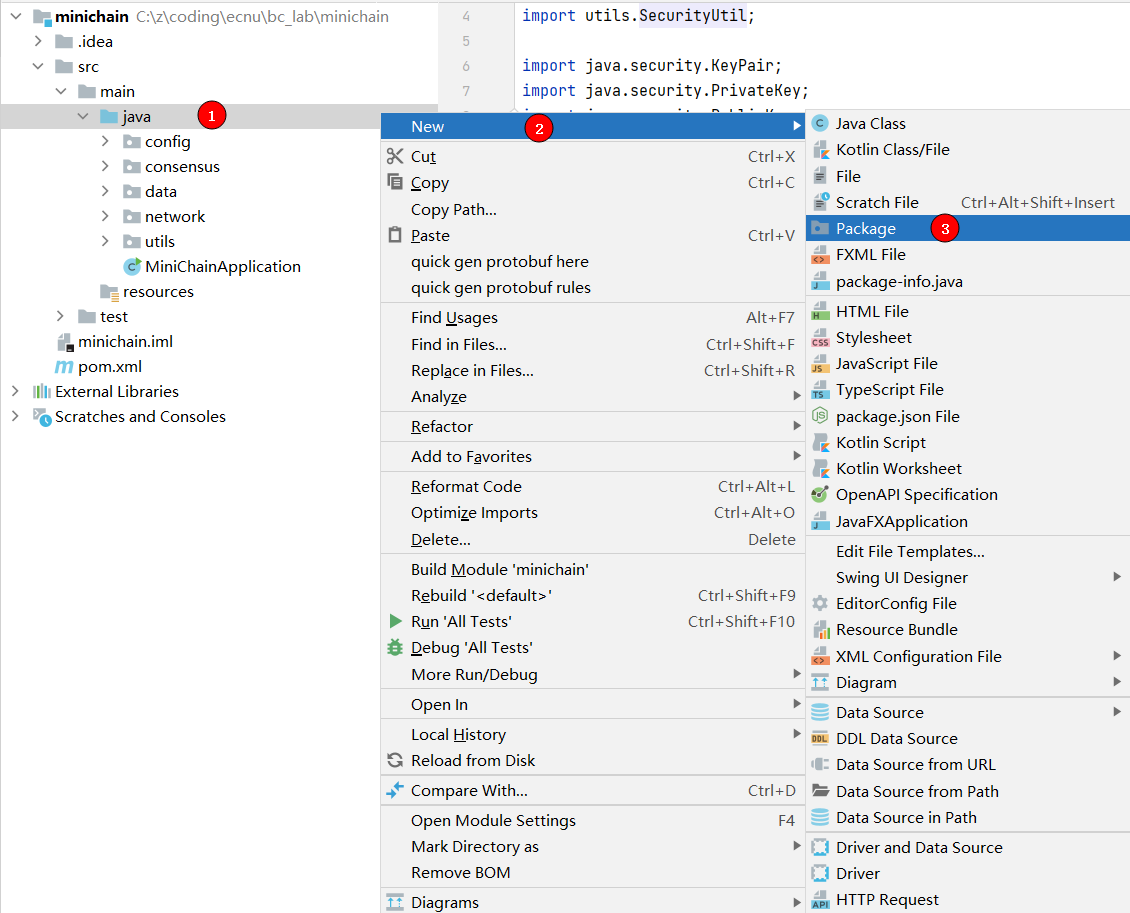
1. 实现更真实的merkle根哈希计算方法，交易数不局限于2的幂（但不能为0）

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，代码如下：

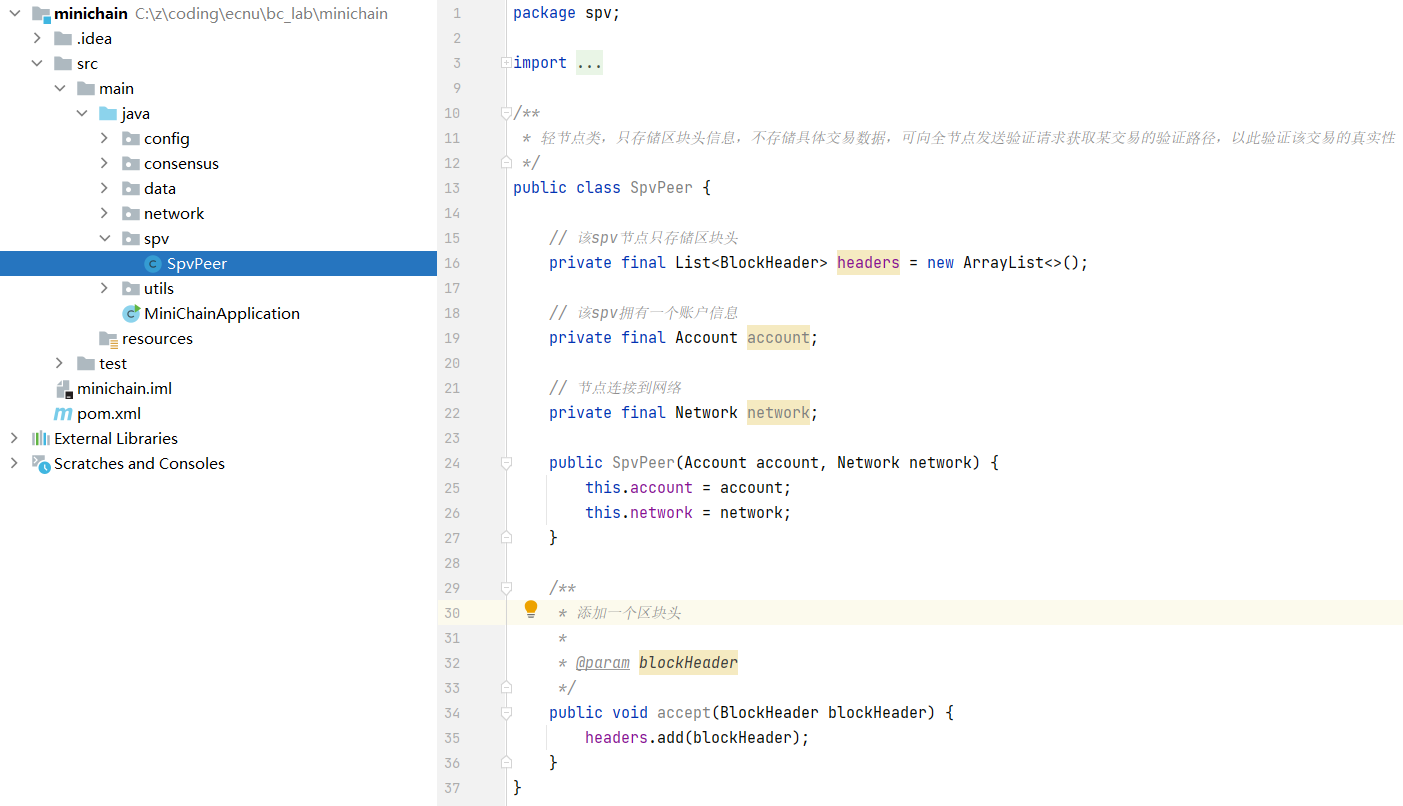


1. SPV轻节点
2. SpvPeer类

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，其拥有一个接受区块头的函数：

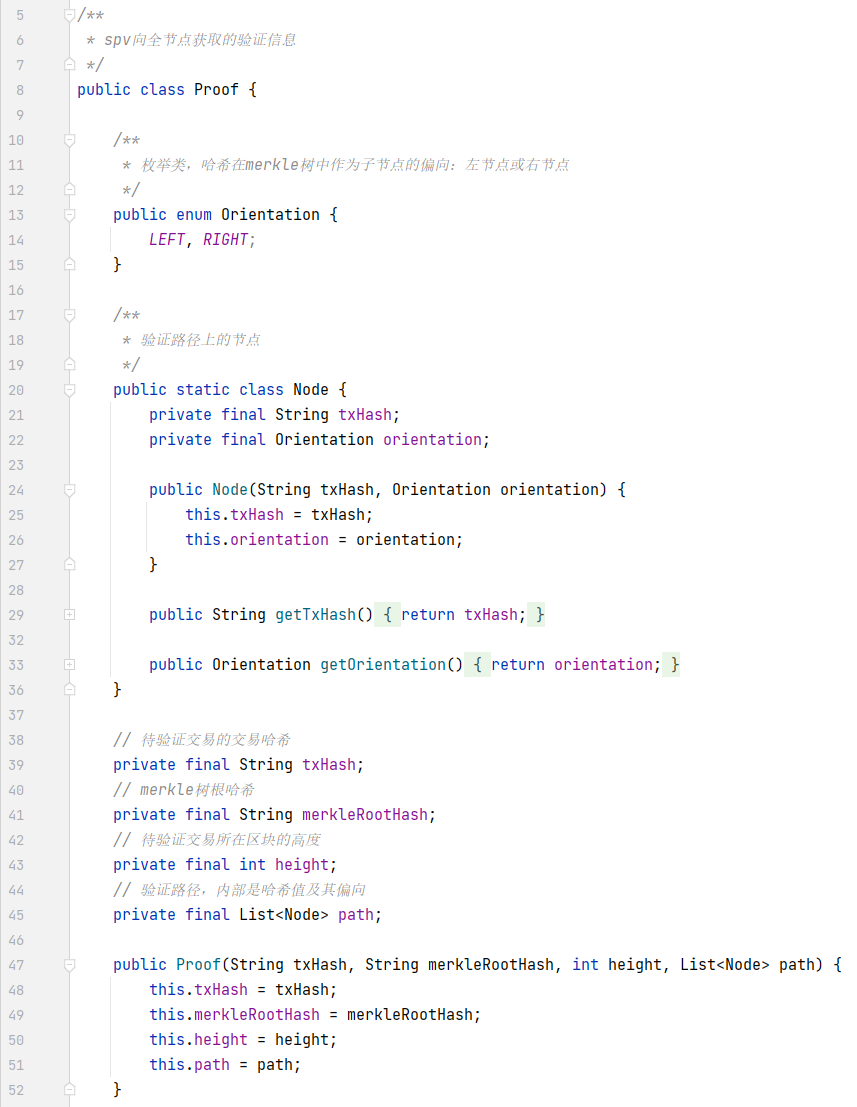


1. Proof类

下面我们就要进行简单支付验证流程的设计了，从实验介绍中我们知道，spv轻节点回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，然后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：

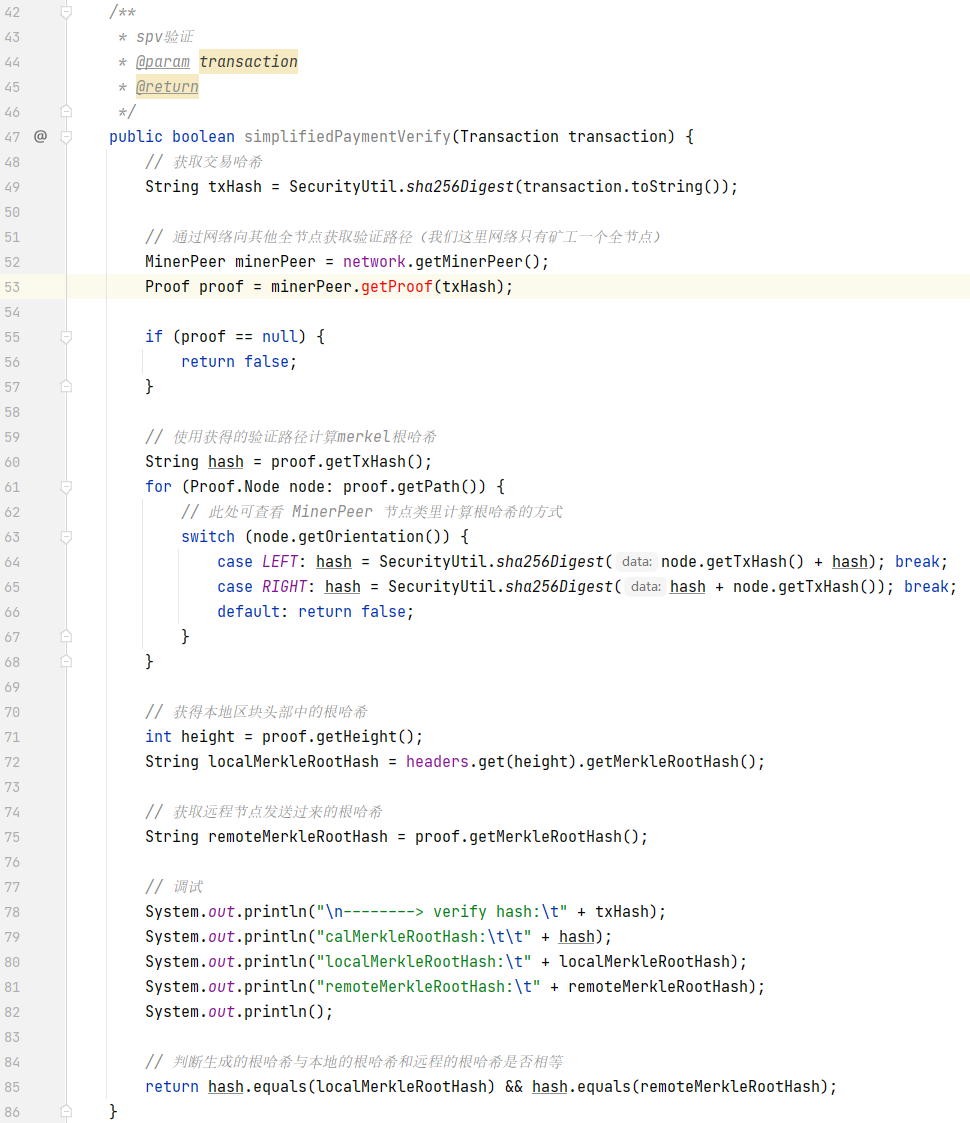


那么，我们还需知道每个节点在计算时是左节点还是右节点，即记录偏向信息，这里我们设计一个验证类来管理这些，在spv包下新建Proof类，编写代码如下图，相应的getter函数请使用Alt+Insert快捷键快速生成（包括图片中未显示的Proof类的一些getter函数）。这里我们使用枚举限定路径节点的偏向，内部节点类维持路径节点的两个参数，列表存储路径节点，以及其他相关的数据成员。



1. 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：

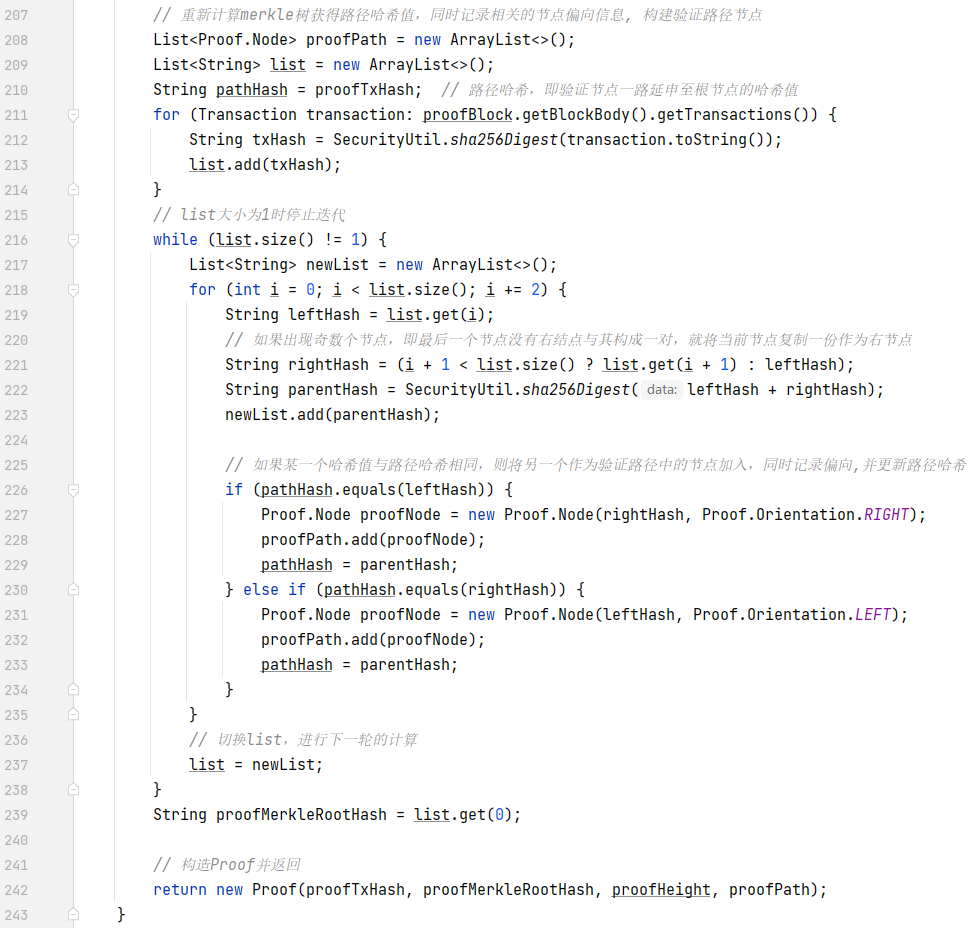


1. getProof函数

该函数依赖于矿工节点的getProof函数，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，我们这里就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：

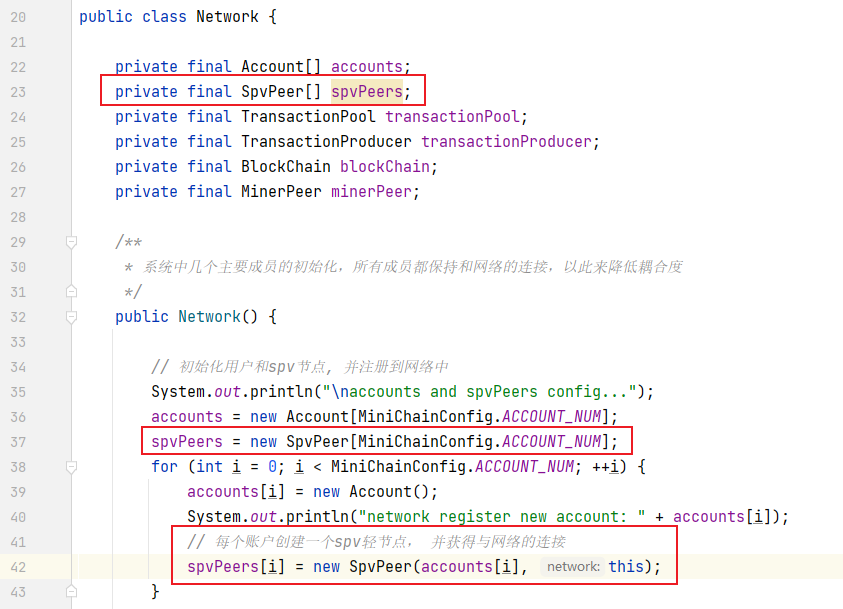


然后重新计算merkle根哈希，构建Proof验证路返回（下图与上图是一个函数）：

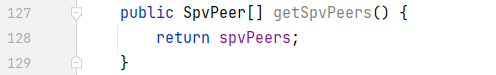


1. spv轻节点数组

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类中添加如下代码：

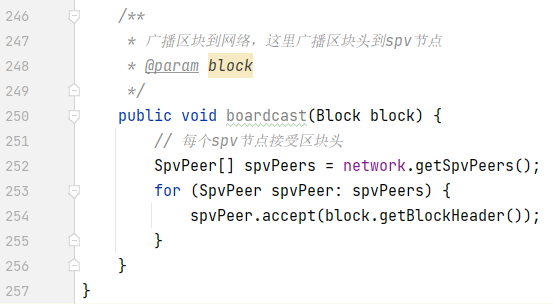


并添加getter函数：

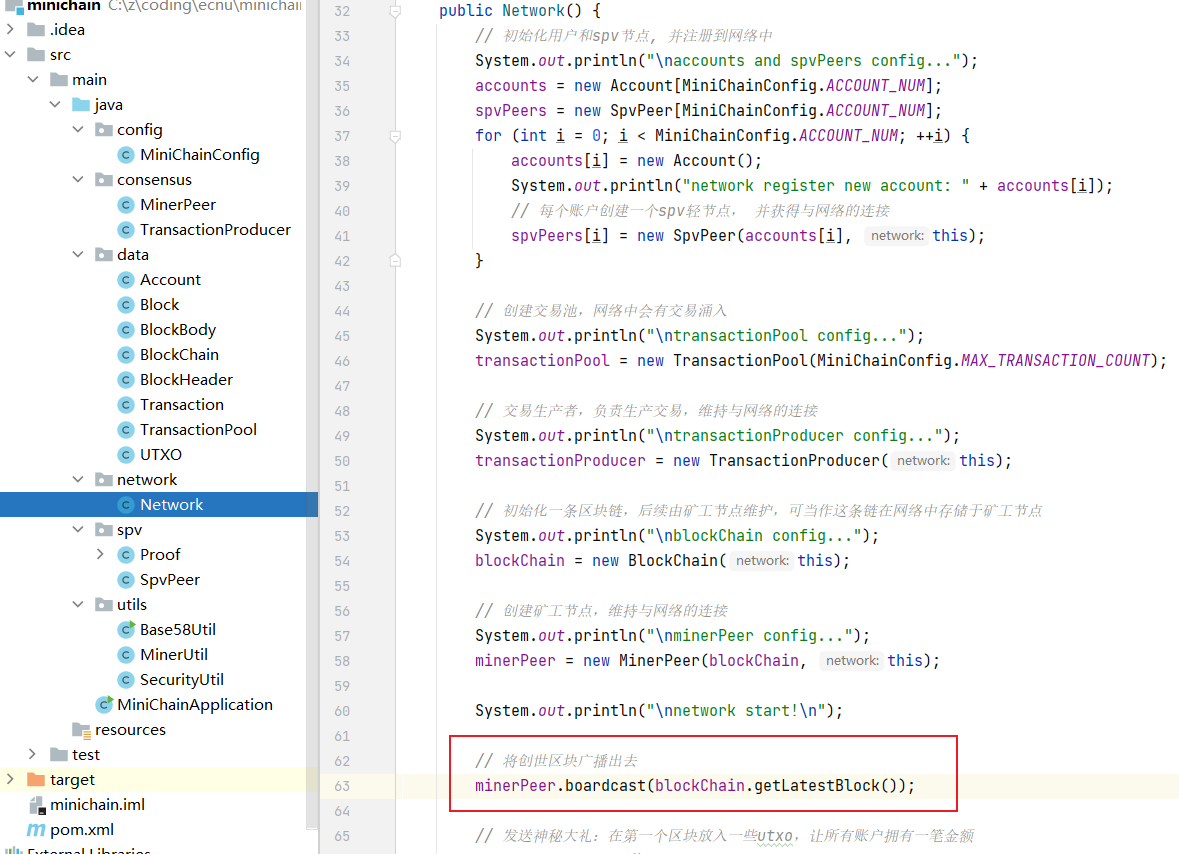


1. 广播区块头

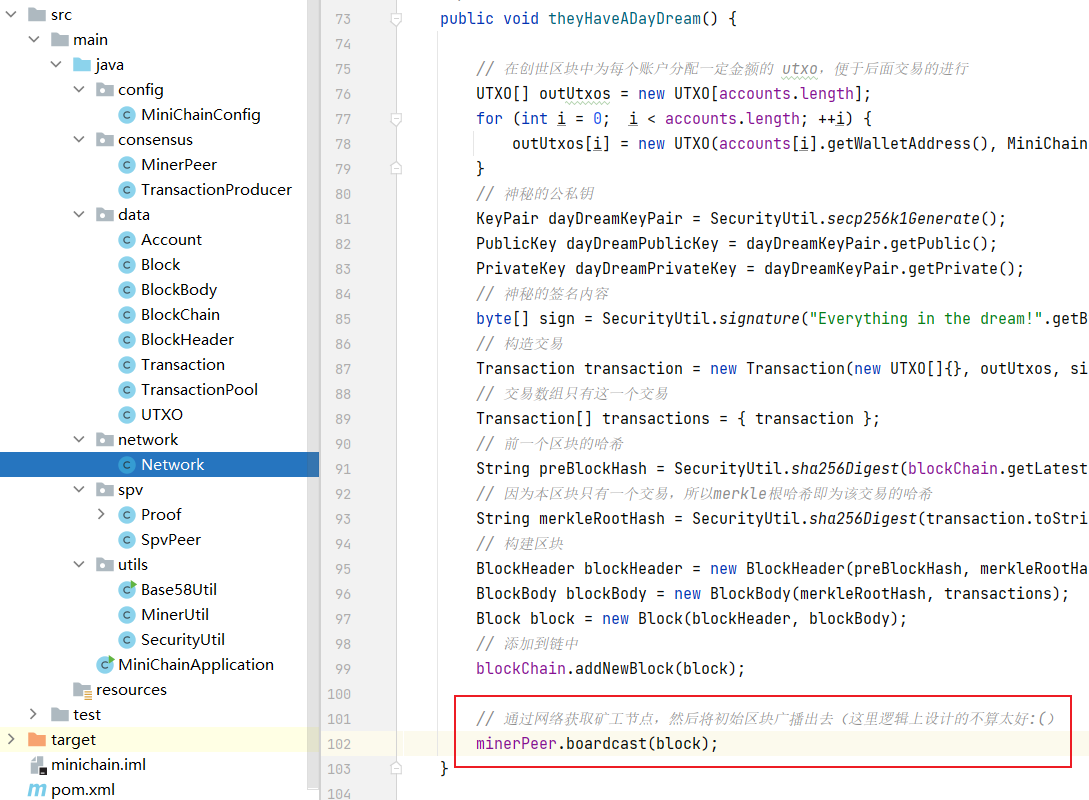
这些spv轻节点仅存储区块头数据，故矿工每挖出一个区块就需要给网络中的spv节点发送最新区块的头部，在MinePeer节点中添加boardcast函数，其通过network获取网络中的spv轻节点集，将区块的头部数据发送出去，类似于p2p网络中的广播：



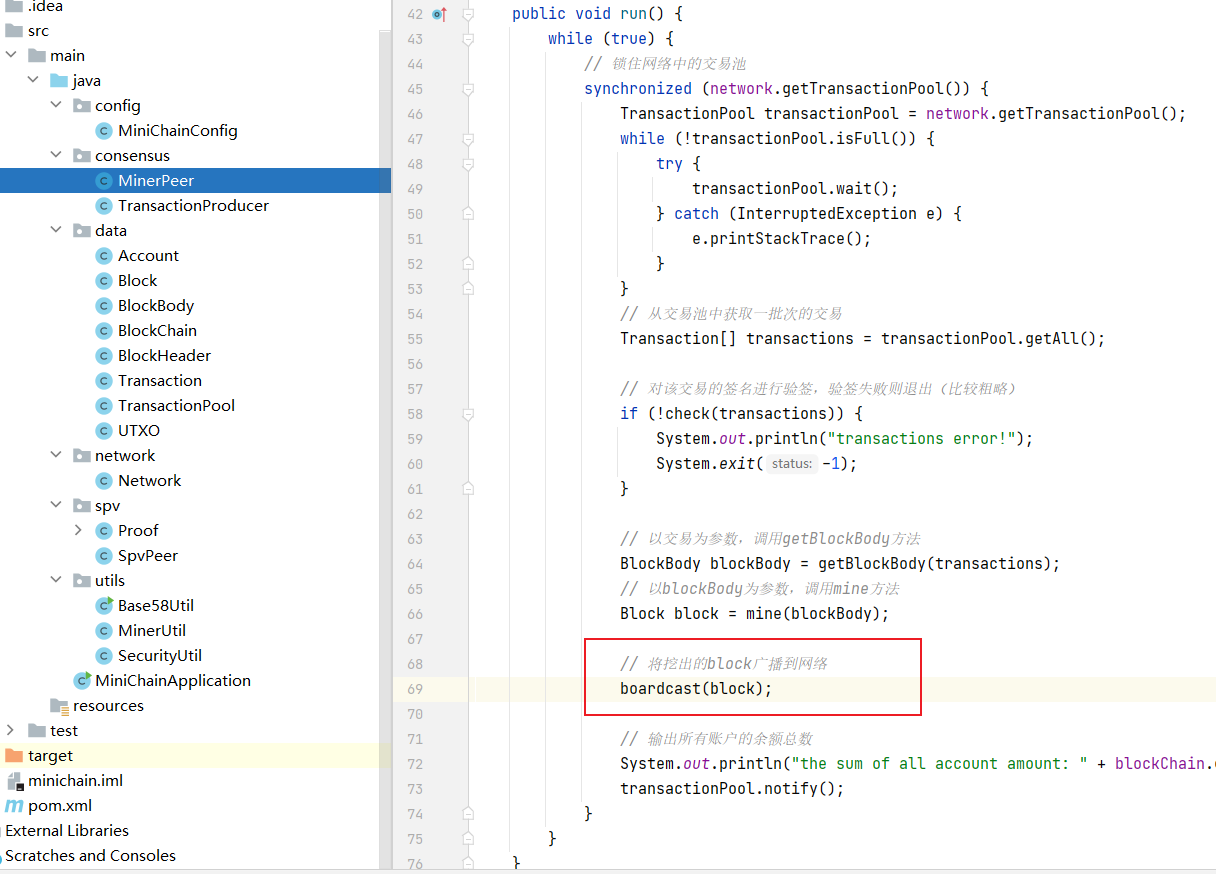
轻节点中的头部数据始终要保持与矿工中的全链数据同步，故在Network类构造函数中初始化blockChain时，需同步创世区块：



对于因构造初始UTXO的第二个区块，需要广播：

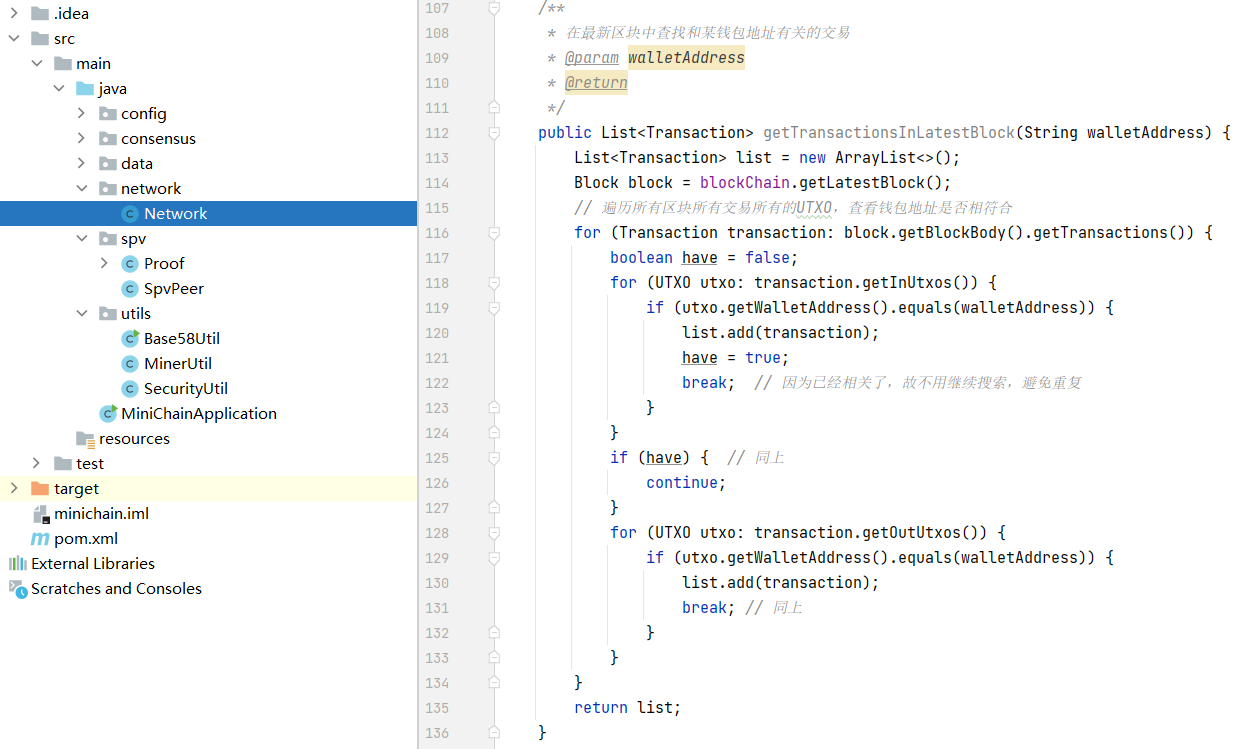


对于矿工每一个挖出的新块，也需要广播：



1. 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，其参数为钱包地址：



1. 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：



运行主程序，可看到验证情况：

