华东师范大学数据科学与工程学院实验报告

比特币系统的实现

一、实验目的

了解区块链中比特币系统的工作原理：

1、了解比特币系统中大致的结构，运行机制等；

2、熟悉矿工节点进行挖矿的过程，包括 merkle 值的计算，区块体以及区块头的打包，POW 共识机制，对 SHA256 算法的理解；

3、熟悉密码学的相关知识，包括公私钥钱包地址的生成过程与作用，交易与 UTXO 的验证过程等等；

4、了解 UTXO 模型，并且熟悉用户之间的交易过程；

5、了解 SPV 简单支付验证的必要性与过程，包括目标区块的定位，目标交易的验证路径寻找等；

6、知道比特币系统中一些简单问题的应对方法，比如双重支付等。

二、实验任务

1、模拟比特币系统中矿工节点挖矿的过程；

2、基于 UTXO 模型与密码学知识，模拟比特币系统中用户之间交易的过程；

3、模拟比特币系统中 SPV 过程；

三、使用环境

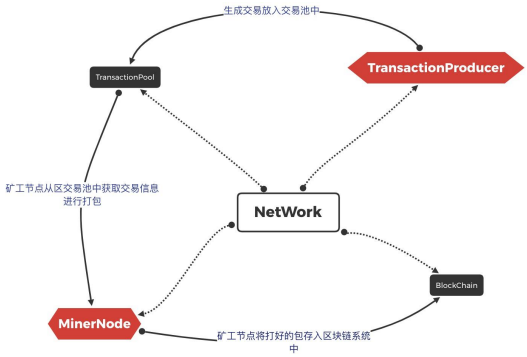
SDK1.8，intellij idea

四、实验过程

实验一：模拟比特币系统中矿工节点挖矿的过程

代码的理解:

network 对象作为一个载体，矿工节点 MinerNode，交易产生者 TransactionProducer，交易池 TransactionPool，区块链 BlockChain 的活动场所。



TransactionProducer 生成交易并写入交易池，MinerNode 不断从交易池中按批次取出交易。TransactionProducer 与 MinerNode 作为两个并行的线程，而交易池作为两个线程的临界区，在编写的过程中需要注意一些并发的问题。在 TransactionProducer 往交易池写入交易的时候需要拒绝 MinerNode 对交易池的访问，当 TransactionProducer 进入交易池时会判定交易池的数量是不是已满，如果是的话那就会等待 MinerNode 取走交易池中的交易，当 TransactionProducer 生产完交易后发现交易池已满，就会通知 MinerNode 运行，至此切换线程。

```
synchronized (transactionPool) {
    while (transactionPool.isFull()) {
        try {
            transactionPool.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    Transaction randomOne = getOneTransaction();
    transactionPool.put(randomOne);
```

```
if (transactionPool.isFull()) {
    transactionPool.notify();
}
```

其中 synchronized 关键字是保证代码块中的操作的原子性，即在运行代码块中代码时不会发生线程切换。同理 MinerNode 进入交易池，需要看看交易池是不是满了，如果没有满，那就必须等待，在取走交易池的所有交易之后有需要通知 TransacProducer 线程启动。

第一个实验的交易 Transaction 还是比较简陋的，其中包含的信息只有一些 data 和 timestamp，timestamp 是这笔交易产生的时间，有专门的函数产生，而 data 更是随机生成的用来区分交易

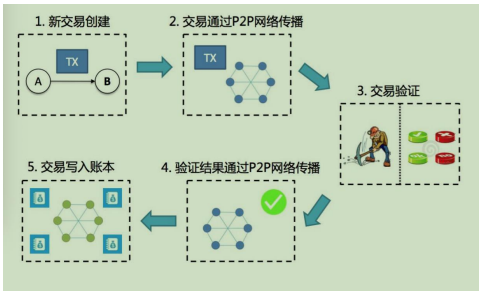
```
Transaction transaction = new Transaction(UUID.randomUUID().toString(), System.currentTimeMillis());
```

SHA256 可以看作时一个黑盒子，是一个确保一对一，不会产生碰撞的函数。输入是任意一串字符串（在代码中大部分是通过 toString（）方法实现），输出是一个 256 位 0/1 串。SHA256 另一个性质是，它正向运算可以实现，但是反向运算，即通过函数结果求得函数的输入是几乎不可能的事。这样的哈希函数，是区块链系统中安全性保证的基础，也是矿工节点挖矿的难度所在。

挖矿

在真实的比特币系统中，挖矿又称 POW（proove of word，工作量证明），真正体现了了比特币系统的分布式记账本的实质。

因为比特币是去中心化的，是没有一个中央的权威机构来管理系统中每个用户所产生的交易的。因此挖矿就是将系统网络中一些交易收集起来并且打包然后记入账本。

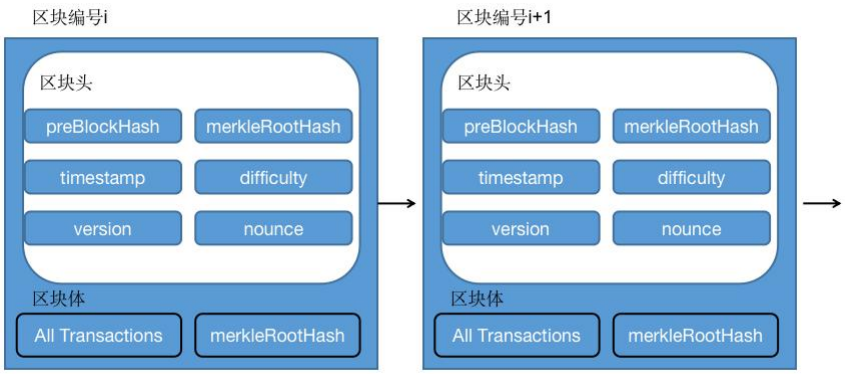


打包过程中首先需要验证每笔交易的真实性（实验一中的代码没有给出），打包后就是漫长的“试数”阶段，一旦“试数”成功，即区块生成，并且接入区块链中的末尾。这个信

息通过网络传播至各个节点（包括矿工与用户），他们将其记入自己所存的区块链副本中。真实的系统中为了激励更多的用户加入矿工节点行业，往往在他们打包好交易后和区块生成后给予丰厚的物质奖励，但是又为了防止过的区块生成过快与通货膨胀，系统也会根据当前时代的平均算力调整难度值。虽然系统中每个人都存有相应的副本，但是之后承认已知最长的区块链。因此，这就解决了区块链系统中没有中央权威机构却也能够记录信息的问题。

代码实现

在实验代码中，区块链的结构如下：



在区块头中：

preBlockHash:为前一个区块所有的信息（toString（））， 体现了区块链的链式结构；

```
String preBlockHash=blockChain.getNewestBlock().toString();
```

merkleRootHash：由该区块中所有的交易产生（下文会详细介绍）；

timestamp：为区块打包是的时间，java 自带包的函数生成；

difficulty:难度值，初始化时就已经设置为 4（下文介绍）；

version：系统版本，初始化时就设置为 1，在这里似乎不重要；

nounce：一个灵活的随机数，“试数阶段”所用的变量（下文介绍）；

在区块体中，存着已经打包好的所有交易的信息，在代码中是一个有序数组，和 merkleRootHash。

挖矿：

实验代码中，MinerNode 的工作流程如下：

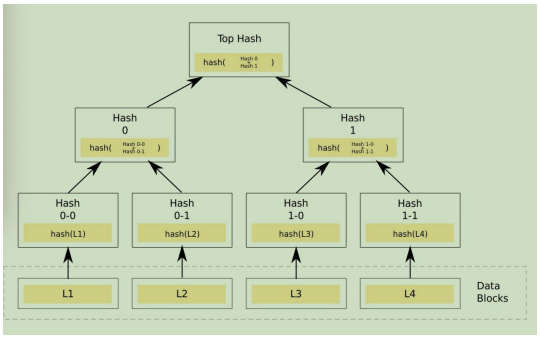
（1）首先 MinerNode 等待交易池中交易满，取出交易；

```
Transaction[] transactions = transactionPool.getAll();
```

（2）调用 getBolockBdoy（）方法，根据取出的交易生成该批次交易的 merkleRootHash，连同该批次交易一起封装成为区块体；

```
BlockBody blockBody = getBlockBody(transactions);
```

merkleRootHash 的生成过程：



merkleRootHash 的生成是一个自底向上的过程。首先每笔（L1……）交易通过 SHA256 函数变换为该笔交易的哈希（Hash（L1）……）；相邻两笔交易哈希串简单拼接，并且继续使用 SHA256 函数进行哈希变换，这样交易哈希数量减半；以此类推，经过一轮一轮的拼接与哈希后，直到得出最后的总哈希。整个过程就好比是一个二叉树的叶子结点不断生成父母节点。

在代码中（getBolockBdoy（））首先遍历所有的交易，将交易的哈希值存入 merkletree 列表中；

```
for(i = 0; i < MiniChainConfig.MAX_TRANSACTION_COUNT; i++){
    merkletree.add(SHA256Util.sha256Digest(transactions[i].toString()));
}
```

不断循环，取出 merkletree 中头两个元素，相拼接，去哈希并切写回 merkletree 中，直到 merkletree 中元素个数只为 1；

```
while(merkletree.size()!=1){
    String str1= merkletree.get(0);
    String str2= merkletree.get(1);
    merkletree.remove(1);
    merkletree.remove(0);
    merkletree.add(SHA256Util.sha256Digest(str1+str2));
}
```

最后剩下的元素，就是该批次交易的 merkleRootHash。

最后调用 BlockBody（）方法，生成区块体

```
BlockBody BlockBody =  new BlockBody(merkleRootHash,transactions);
```

e.g.代码中的 Test 类就是为了验证 BlockBody 生成的正确性，尤其是 merkleRootHash 生成的正确性。其原理就是生成一批次特定的交易，然后使用正确的方法计算出这一批交易的 merkleRootHash，打包成区块体后，与使用 getBolockBdoy（）方法生成的区块体比对，如果生成成功，那么说明编写的 getBolockBdoy（）方法正确。

（3）在 getBlock（）方法中生成区块头，和区块体结合，生成区块：

merkleRootHash 之前以及求得，直接调用获取；

```
String merkleRootHash=blockBody.getMerkleRootHash();
```

preBlockHash 通过调用 BlockChain 中的 getNewestBlock（）方法得到；

```
String preBlockHash=blockChain.getNewestBlock().toString();
```

nounce 字段随机生成；

```
long nonce=Math.abs(new Random().nextLong());
```

timestamp、difficulty、version 在构造函数的时候就已经初始化；

至此区块头所有信息已经具备，与区块体一起打包成区块；

```
Block Block=new Block(blockHeader,blockBody);
```

（4）挖矿，mine（）函数：

所谓的挖矿就是通过区块的所有信息使用 toString（）变成字符串形式，然后再对字符串使用 SHA256 函数变成区块哈希，最后观察区块哈希中是不是满足指定的形式，如果满足，那就说明打包成功。

一般的，这个“指定形式”，就是区块哈希串开头的“0”的个数。比如，某个时候比特币的区块链设置为一个区块哈希需要以 4 个“0”开头，但 Miner 打包好的区块得到的哈希开头只有 3 个“0”，因此这个区块就不满足条件，不能被接入区块链中。开头“0”的个数，就是代码中的 difficulty，系统指定“0”的个数越多，打包的区块成功率越低。

如果打包好的区块不满足设置的 difficulty，那么就需要改变这个区块的信息来使这个区块满足。但是由于区块中的字段除了 nounce 之外，其他所有的信息都是固定的，不可篡改的，因此矿工节点只能一次一次改变 nounce 的值，再判断区块满不满足条件。这

个就是“试数”阶段。

```
while (true) {
    String blockHash = SHA256Util.sha256Digest(block.toString());
    if (blockHash.startsWith(MinerUtil.hashPrefixTarget())) {
        System.out.println("Mined a new Block! Detail of the new Block : ");
        System.out.println(block.toString());
        System.out.println("And the hash of this Block is : " + SHA256Util.sha256Digest(block.toString()) +
                           ", you will see the hash value in next Block's preBlockHash field.");
        System.out.println();
        blockChain.addNewBlock(block);
        break;
    } else {
        //todo
        block = getBlock(blockBody);
    }
}
```

上述代码就是不断的循环判断区块满不满足条件，满的话就将区块接入已有的区块链末尾，如果不满足条件，那就需要重新随机一下 nounce 的值再继续。

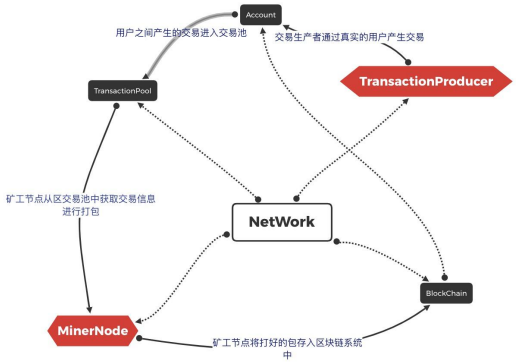
当 MinerNode 挖矿成功之后，就会将结果进行全网广播，使得大家对这个区块的承认，完成工作量证明（虽然这一过程，代码没有给出）

实验二

基于 UTXO 模型与密码学知识，模拟比特币系统中用户之间交易的过程

代码结构

由于实验一的的代码中，交易的产生是随机的，凭空产生的，而实验二的目的是需要模拟用户真实交易的过程，包括了身份验证，交易的正确性验证等，因此就需要引入 Account 这一个新角色，如下图所示：



Account 寄托在 BlockChain 平台，随着 BlockChain 的初始化而产生。TransactinoProducer 已经不是随机构造参数，凭空产生交易，放入交易池中。在这里 TransactinoProducer 要做的是通过两个 Account 之间模拟现实比特币系统中的交易流程，产生交易而放入交易池中。

公私钥

真实的区块链面对的对象的性质是多方参与互不信任，而且是没有中央权威机构的。因此为了防止个人的数据和交易信息被篡改就引入了密码学的公钥私钥的概念。一般公钥和私钥是成对出现的，公钥是可以公开的，而私钥是自己私密保存的，公开会产生许多安全问题。私钥可以对一个信息进行加密，公钥（只有与私钥配对的公钥）可以对这一个加密的信息进行解密，并且这个加密与解密的过程只有正向过程，逆向过程几乎不可能。公钥和私钥是一种非对称的加密方式，通常用来进行身份验证。

上图就是一个应用公私钥加密和身份验证的例子。某个用户 A 为了向其他用户证明某条记录的所有权时，用户 A 不仅要发送一条信息（Data），还需要发送这条信息先经过哈希 SHA256 变换与私钥加密之后的签名（135ff034），和一条用来解密的公钥。用户 B 收到这个消息后，首先用公钥将私钥解密得到一条哈希值，然后将 Data 作 SHA256 哈希变化得到另一条哈希值，如果这两条哈希值是一样的，那么就说明了这条信息是 A 发送的，而且发送的

过程中数据没有被恶意篡改，保证了过程的安全性。

交易的产生：

代码中的大致思路（在 getOneTransaction（）中），考虑用户 A 向用户 B 转账：

（1）首先需要获得用户 A 和 B 的钱包地址；

（2）获得 A 的余额以及可以使用的“余额”UTXO；

（3）A 解锁并且使用这些“余额”向 B 汇款；

（4）产生一笔交易；

根据代码详细介绍每一步骤：

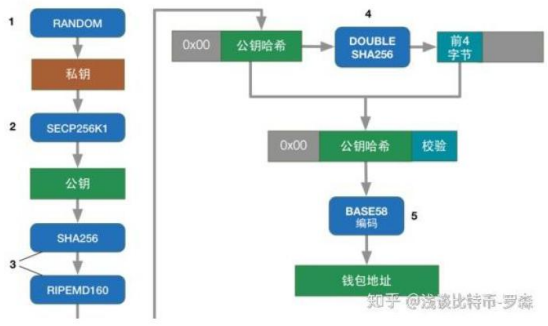
（1）获得 A 和 B 的钱包地址：

获得 A 的钱包地址是为了获得计算 A 钱包中的余额，获得 B 的钱包地址是为了交易过程需要交易获得方的钱包地址。

钱包地址的产生过程也比较复杂（代码在 Account 类中），大致如下图：

- 随机数发生器生成一个『私钥』
- 『私钥』经过SECP256K1算法处理生成了『公钥』。SECP256K1是一种椭圆曲线算法。
- 同SHA256一样，RIPEMD160也是一种Hash算法
- 将一个字节的地址版本号连接到『公钥哈希』头部（对于比特币网络的pubkey地址，这一字节为“0”），然后对其进行两次SHA256运算，将结果的前4字节作为『公钥哈希』的校验值，连接在其尾部。
- 将上一步结果使用BASE58进行编码(比特币定制版本)，就得到了『钱包地址』。

比如, 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa



2021-3-9 华东师范大学 张召 15

首先是一个随机一个数，通过这个随机数产生私钥，并且通过私钥产生公钥，代码中没有体现这个过程，而是使用以及封装好的方法产生公私钥

```
KeyPair keyPair= SecurityUtil.secp256k1Generate();

this.publicKey = keyPair.getPublic();

this.privateKey = keyPair.getPrivate();
```

对公钥进行一次 SHA256 算法和一次 RIPEMD160 算法得到公钥哈希

```
byte[] publicKeyHash=SecurityUtil.ripemd160Digest(SecurityUtil.sha256Digest(publicKey.getEncoded()));
```

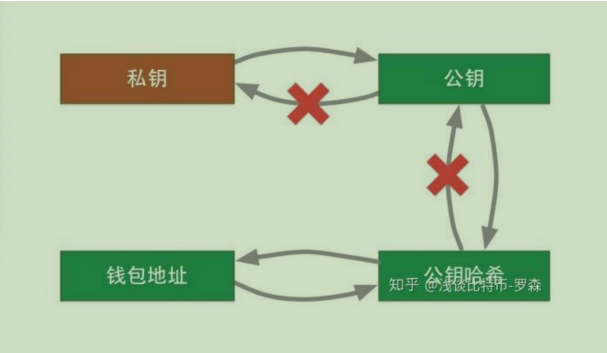
对公钥哈希进行两次哈希运算，结果取前四字节凭借在公钥哈希的末尾，然后在公钥哈希前面拼接 0x00

```
//两次哈希
byte[] doubleHash=SecurityUtil.sha256Digest(SecurityUtil.sha256Digest(data));

//前面拼接 0x00,后面拼接两次 hash 之后的前四字节
byte[] walletEncoded=new byte[1+publicKeyHash.length+4];
walletEncoded[0]=(byte)0;
for (int i=0;i<publicKeyHash.length;++i){
    walletEncoded[1+i]=publicKeyHash[i];

}
for(int i=0;i<4;++i){
    walletEncoded[1+publicKeyHash.length+i]=doubleHash[i];
}
```

对最终获得的公钥哈希进行 BASE58 编码就可以得到该用户的钱包地址

```
String walletAddress= Base58Util.encode(walletEncoded);
```

值得注意的是，整个钱包地址生产的过程只有 BASE58 编码这一过程可逆，其余都是不可逆的



（2）获得 A 的余额以及可以使用的“余额”UTXO；

像支付宝，银行，微信支付这些系统，我们想知道自己的余额是很简单的，因为他们由专门的数据库来记录每个用户的信息。但由于区块链系统是一个没有中央机构来记录我们的余额的，因此想要知道自己的余额就并不那么容易了。比特币系统实质上是一个分布式账本，记录的是一整条区块链，区块链中的所有交易。如果用户 A 想要知道自己的自己的余额就必须遍历整条区块，找出与自己有关的交易，就比如：

C 给 A 转账 5 元；

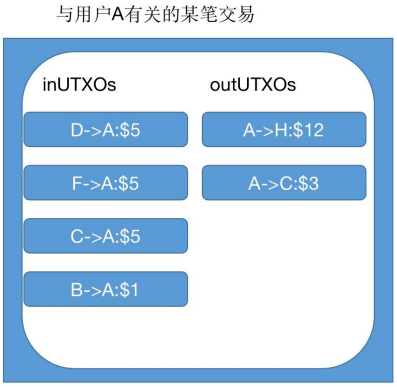
B 给 A 转账 5 元；

A 给 D 转账 6 元；

A 给 E 转账 2 元；

这样一计算，A 的余额就是 2 元。

为了适应这样的计算模式，每笔交易的结构就使用了 UTXO 模型



A 为了完成转账，需要从区块链中找出那些转向 A 的 UTXO 并将其作为 inUTXOs，而在 A 向其他账户支出的 outUTXOs 可以在之后的某笔交易中作为该用户的 inUTXOs 来使用。

在代码中，A 使用 getTrueUtxos 来获得 a 所有可以使用的 utxo，

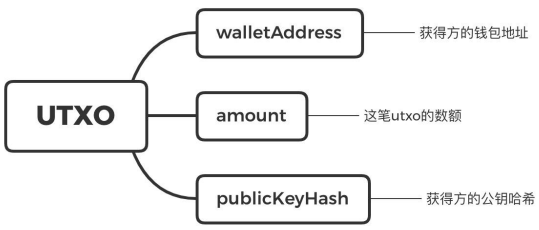
```
UTXO[] aTrueUtxos=blockChain.getTrueUtxos(aWalletAddress);
```

即遍历区块链找出那些作为 outUTXO 转给 A，但是没有作为 inUTXO 使用的 utxos

```
//遍历区块链中所有的区块，然后遍历所有的交易，获取 utxo
for(Block block:chain){
    BlockBody blockBody =block.getBlockBody();
    Transaction[] transactions=blockBody.getTransactions();
```

```
    for(Transaction transaction :transactions){
        UTXO[] inUtxos= transaction.getInUtxos();
        UTXO[] outUtxos=transaction.getOutUtxos();

        //统计出所有转给自己的 utxo 存入 trueUtxoSet
        for(UTXO utxo:outUtxos){
            if(utxo.getWalletAddress().equals(walletAddress)){
                trueUtxoSet.add(utxo);
            }
        }
        //去除用户自己花费的 utxo
        for(UTXO utxo:inUtxos){
            if(utxo.getWalletAddress().equals(walletAddress)){
                trueUtxoSet.remove(utxo);
            }
        }


    }
}
```

一个 utxo 的结构在，在 UTXO 类中有给出



然后使用 getAmount（）函数遍历 A 所有的可以使用的 utxo，将 utxo 中的 amount 相加得到 Amount：

```
for(int i=0;i<trueUtxos.length;++i){
            amount+=trueUtxos[i].getAmount();
        }
        return amount;
}
```

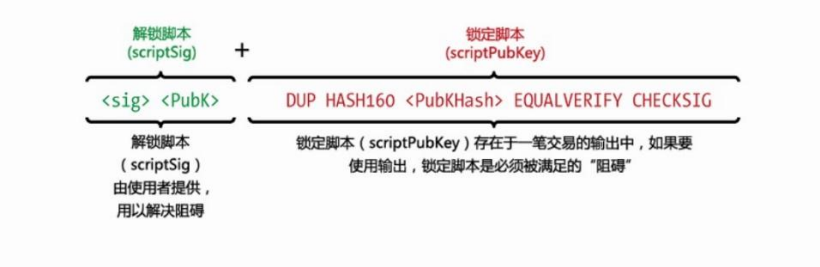
（3）A 解锁并且使用这些“余额”向 B 汇款；

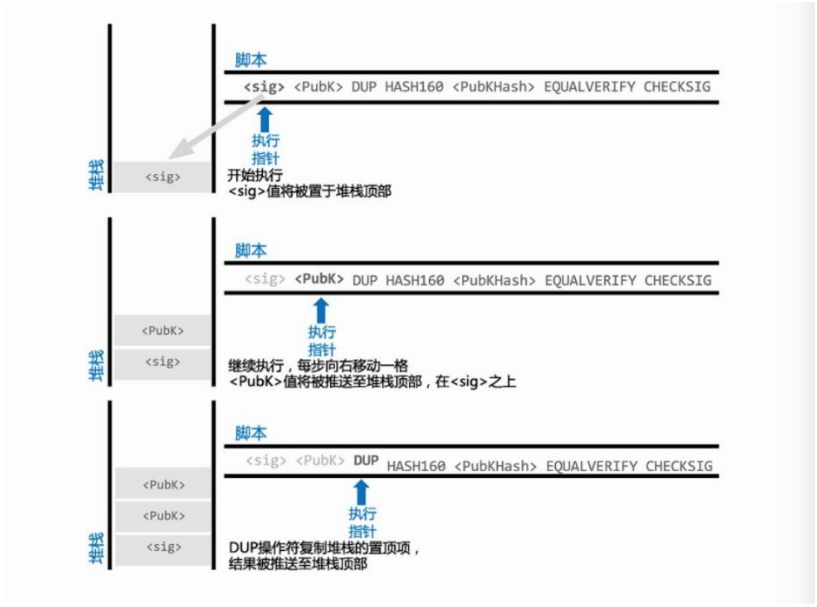
并不是任何人都可以使用 A 的 utxo，因此 A 在使用自己的的 utxo 时需要解锁。A 解锁自己的 UTXO 需要传入自己的签名和公钥

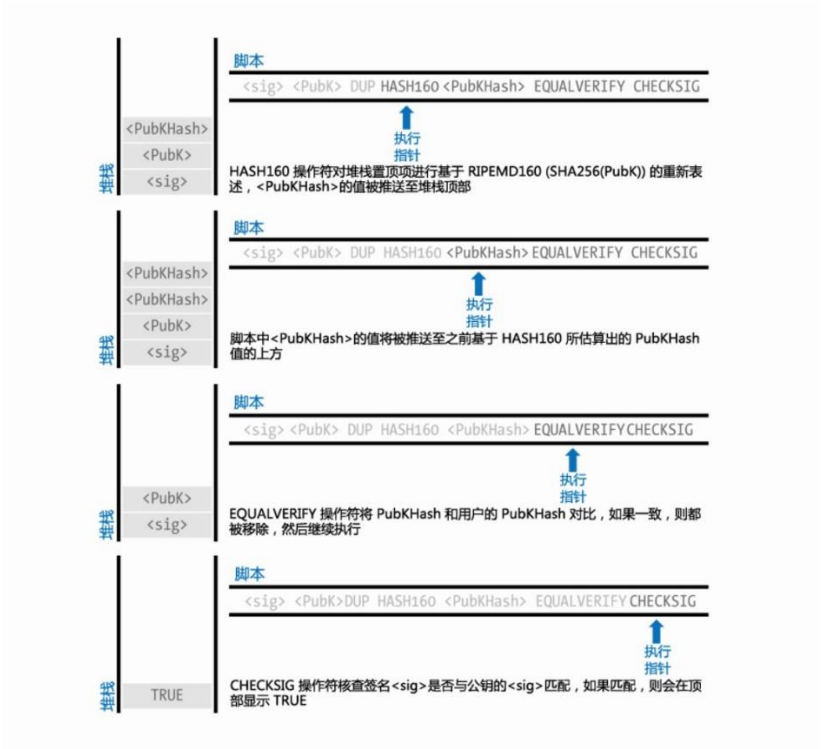
```
public boolean unlockScript(byte[] sign,PublicKey publicKey)
```

传入公钥，是为了确定是判定公钥经过哈希变化之后与 UTXO 中的公钥哈希相匹配，即公钥是解开 UTXO 的钥匙；而传入 sign，是为确保是用户 A 带着钥匙去解锁的。这样就保证了某个用户账户的安全性

区块链中解锁 UTXO 的流程：







将用户的签名入栈

```
//签名入栈

stack.push(sign);
```

用户公钥入栈

```
//<sign><pubkey>

stack.push(publicKey.getEncoded());
```

复制一份公钥，哈希变换后至栈顶

```
//<sign><pubkey><pubkeyhash1>
byte[] data=stack.pop();
stack.push(SecurityUtil.ripemd160Digest(SecurityUtil.sha256Digest(data)));
```

utxo 中存储的公钥哈希入栈

```
//<sign><pubkey><pubkeyhash1><pubkeyhash2>
stack.push(publickKeyHash);
```

比对栈顶头两个元素，如果相同就说明配对成功，如果失败就说明解锁失败

```
if(!Arrays.equals(publicKeyHash1,publicKeyHash2)){
    return false;
}
```

此时栈中还剩下 sign 和 publicKey

最后一个操作是一个身份验证过程，由于 sign 是通过 A 的 publicKey 的编码，加上 A 的私钥生成的，如果 sign 通过公钥解密后得到的数据恰好是 A 的 A 的 publicKey 的编码那么身份验证成功，解锁者确实是 A。

```
SecurityUtil.verify(publicKey.getEncoded(),sign1,publicKey);
```

A 选取可以使用的 utxo 来作为向 B 支付的 inUTXOs

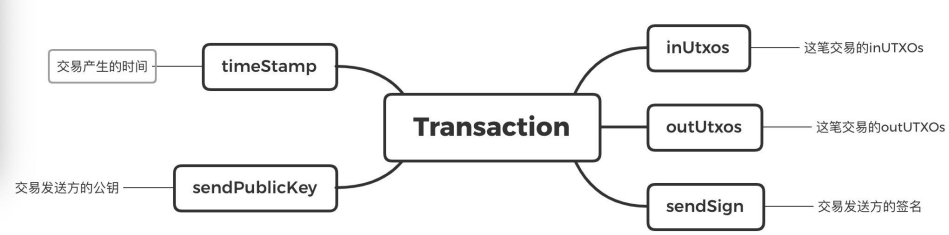
```
for(UTXO utxo :aTrueUtxos){
    //a 要解锁自己的脚本才能使用 utxo
    if(utxo.unlockScript(aUnlockSign,aAccount.getPublicKey())){
        inAmount+=utxo.getAmount();
        inUtxoList.add(utxo);
        if(inAmount>=txAmount){
            break;//直到能够大于交易总额
        }
    }
}
```

而 outUTXOs 一部分是转给 B，一部分是剩下的前，需要转给自己

```
//写下一笔记入 b 的 utxo
outUtxoList.add(new UTXO(bWalletAddress,txAmount,bAccount.getPublicKey()));


//把多余的钱转回 a 的钱包中，记入一笔 utxo
if(inAmount>txAmount){
    outUtxoList.add(new UTXO(aWalletAddress,inAmount-
txAmount,aAccount.getPublicKey()));

}
```

（4）产生一笔交易：

不同与实验一，实验二中交易的结构如下：



根据上图，即可封装好交易，然后进入交易池

```
transaction=new Transaction(inUtxos,outUtxos,sign,aAccount.getPublicKey(),timestamp);
```

发送方的签名 sendSign 与公钥哈希 sendPublicKey 是为了矿工节点在打包交易的时候验证交易的正确性：

因为 sendSign 是通过这笔交易的所有 utxos，通过用户 A 的私钥生成的

```
byte[] data =SecurityUtil.utxos2Bytes(inUtxos,outUtxos);
byte[] sign=SecurityUtil.signature(data,aAccount.getPrivateKey());
```

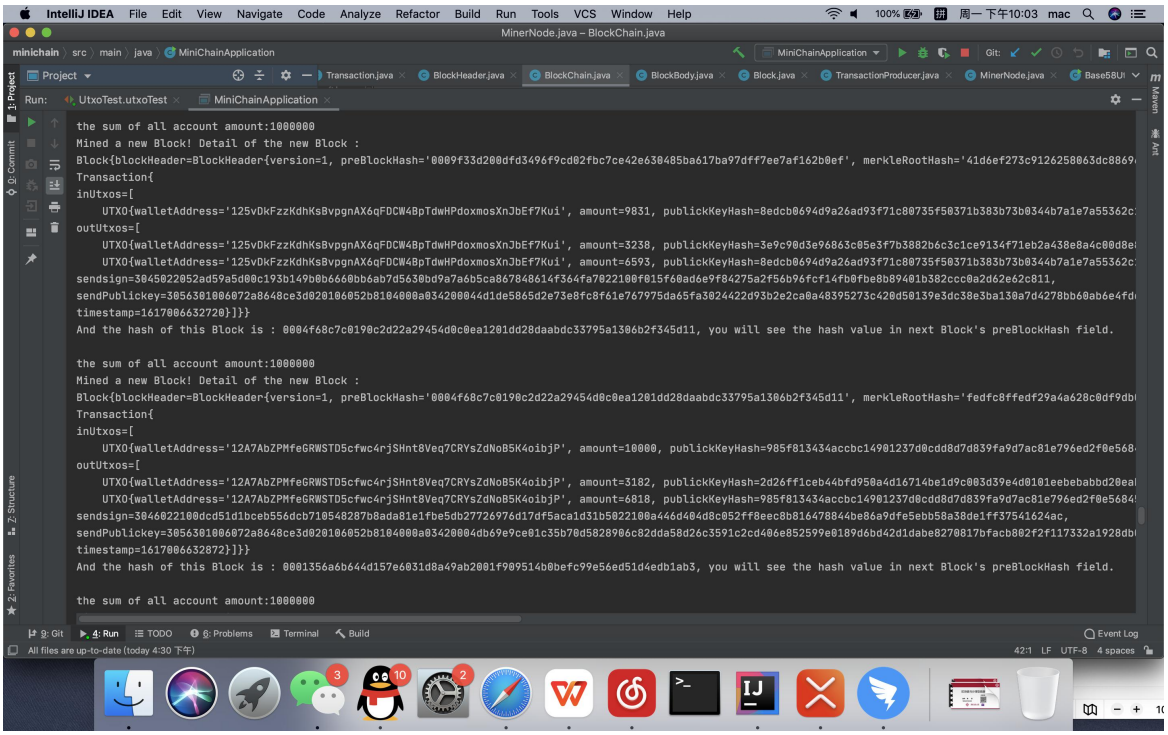
如果矿工节点在打包时，使用 A 的公钥 sendPublicKey 解密后得到的仍是这些 utxos，与交易的 utxos 一致，那就说明这笔交易是正确的，没有被篡改。

```
if(!SecurityUtil.verify(data,sign,publicKey)){
    return false;
}
```

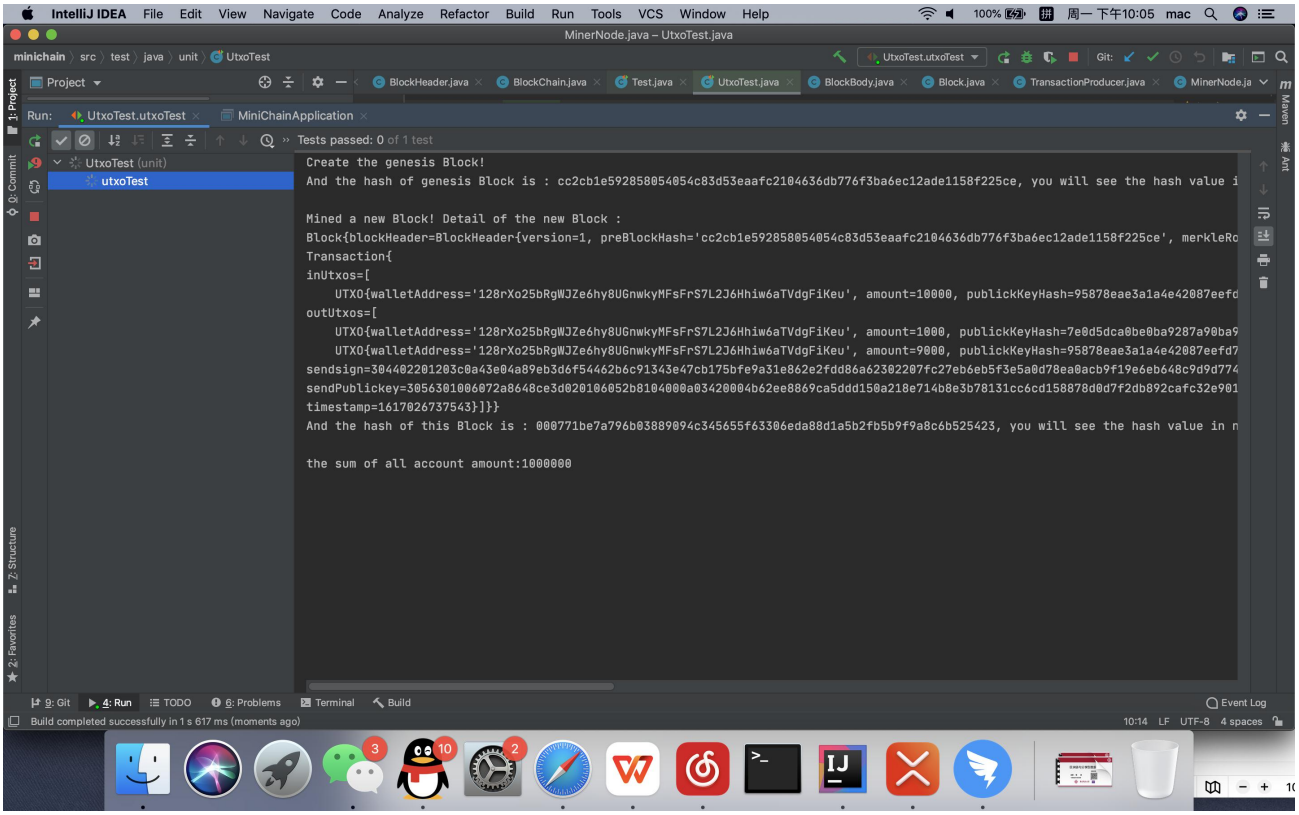
至此，交易流程已经介绍完毕。

为了保证系统代码能够运行起来，再初始化时，需要向所有用户分配一定的余额，即系统生成一笔交易，钱包地址为每个用户的钱包地址，inUtxos 为空，outUtxos 为转向用户的 utxos，交易的签名与公钥都是随机生成。这些交易打包后就作为区块链的头。（具体的过程代码就不展出了）

getOneTransaction（）就是不断的随机指定两个用户，随机化交易金额来完成交易的，运行结果



测试函数是需要指定两个用户，指定金额，代码和 getOneTransaction（）相类似，改掉其中的随机化因素就好了，运行结果



“双花”问题：

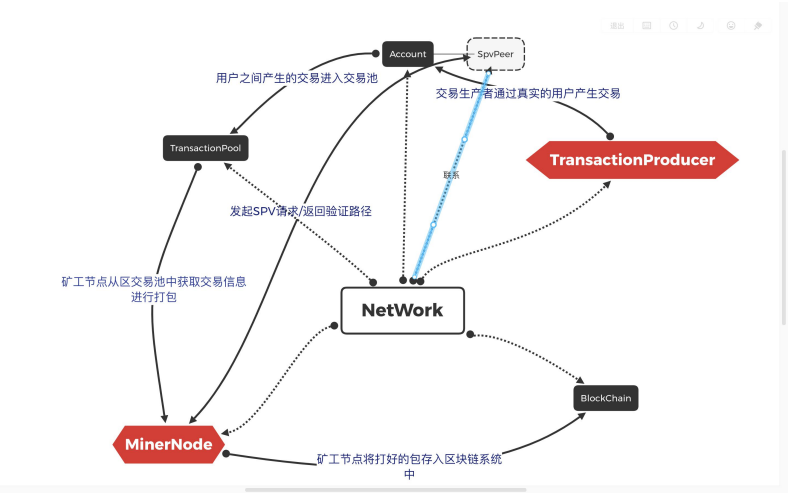
由于在运行中会存在“双花”现象，即在一批交易生成过程中，某个用户的可以使用的 UTXOs 会存在两次或这两次以上的使用情况，这是由于某笔 UTXO 在使用之后，如果没有被打包记入区块链中，那么这笔 UTXO 就会被认为是没有被使用的，就会可能在该用户下一次转账的时候被使用。为了避免双花现象，我们把交易池的容量变为 1，这样一旦一笔交易产生，交易池就会满，这样就不会产生交易了，直到矿工取出交易，打包，并且挖矿成功接入区块链中，交易生产这才会被唤醒，继续产生交易。这样就不会出现某笔 UTXO 已经被使用，但是因为没有被记入区块链而查不到的现象。这个做法显然是治标不治本的，但是由于这个实验的重点不在这，所以“双花”现象将在下一个实验被彻底解决。

实验三、模拟比特币系统中 SPV 过程

代码结构

对于实验二 Accounts 角色是寄托在 BlcokChain 上的，随着 BlockChain 的初始化 Accounts 而得到初始化，这与现实中的比特币系统不符；

这次实验还需要为每个账户添加 SPV 功能（简单支付验证功能），因此需要将往每个账户里面存储区块头有关的信息，因此引入 SpvPeer 用来和 Account 用户绑定，存储这个账户所拥有的区块头信息，Account 用来存储这个账户的公私钥，因此在本实验中将代码结构进行如下调整：



基于上图，所有的角色的初始化都转移到了 Network 中，包括了：

MinerNode，TransactionProducer 线程的启动；

TransactionPool，BlockChain，Account，SpvPeer 的初始化；

Account，SpvPeer 之间的绑定；

```
spvPeers[i] = new SpvPeer(accounts[i], this);
```

最开始为每个用户分配一笔资产的交易的初始化入链。在实验三中，这个区块位于第二个区块链，而实验二这个初始化是在 BlockChain 中位于第一个区块；

每个 SpvPeer 往自己本地存入头两个区块的区块头信息，通过 boardcast（）方法

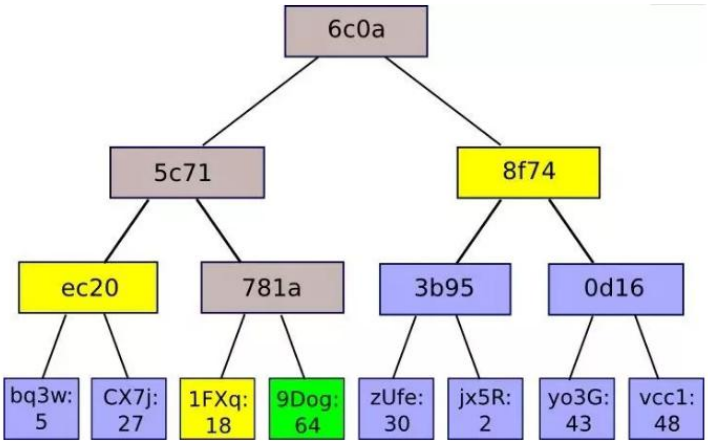
```
for (SpvPeer spvPeer: spvPeers) {
    spvPeer.accept(block.getBlockHeader());
}
```

SPV 简单支付验证

区块链的中的对象是多方参与互不信任的，到目前位置，我们已经实现了公私钥的运用，交易过程的身份验证，UTXO 解锁等功能，但是某一个用户仍然可以认为在交易产生之后到交易被正式打包入链这一个时间段会存在交易被篡改的可能性。因此这个代码实现了，在实验三的代码实现了每当一个新的区块入链时，所有的账户都可以对比本地的信息和远程的信息，对与自己有关的交易信息进行验证，以防被篡改。

考虑区块链系统中，网络，存储和计算等问题，每个用户的轻节点只存储区块链的区块头信息，因为在验证时起主要作用的信息就是 merkleRootHash。这样，在一些存储和计算性能比较差的机器上也可以进行简单支付验证。

具体的 SPV 过程：



（1）当一个用户想要验证上图中绿色的交易时，向一个存储了所有区块链信息的全节点（往往是一个矿工节点）发起请求，提供该交易的交易哈希；

（2）全节点收到请求信息后，遍历区块链，定位到含有目标目标交易的区块。如果定位不到，就说明交易还没有入链或者交易被篡改；

（3）全节点定位到目标区块后，返回交易验证路径，如上图黄色部分的交易哈希；

（4）轻节点收到验证路径后，使用待验证的交易哈希与这些验证路径上的交易哈希，依次拼接计算哈希，直到计算出这笔交易所在区块的 merkleRootHash；

（5）轻节点对比本地存储的 merkleRootHash，全节点返回的 merkleRootHash 和计算得出的 merkleRootHash，如果三者相同，那么交易验证成功，否则验证失败；

代码实现

在代码中，由于每个账户都是虚拟的，没有主观意识的，因此为了引入 SPV 功能，在一个矿工节点打包后，所有的区块都检查这个区块中是不是有与自己有关的交易，有的话才进行简单支付验证。具体流程以及代码解释：

（1）当矿工节点打包好一个区块并且入链后，它通过网络广播这个区块，每个轻节点将这个区块头信息记入自己的本地；

```
minerPeer.boardcast(block)
```

（2）每个轻节点在记入区块头信息的同时，还要验证该交易是否与自己有关

```
verifyLatest();
```

getTransactionInLatestBlcok（）中

通过网络，获取最新入链的区块体

```
Block block = blockChain.getLatestBlock();
```

遍历遍历这个区快中的每笔交易，在一次遍历每笔交易中的 UTXO，包括了 inUtxos 和 outUtxos

```
for (Transaction transaction: block.getBlockBody().getTransactions()) {

for (UTXO utxo: transaction.getInUtxos()) {

for (UTXO utxo: transaction.getOutUtxos()) {
```

只要发现该交易的某笔 UTXOs 的钱包地址与本地 Account 的钱包地址一样那就将这笔交易加入该用户待验证的交易列表中去

```
if (utxo.getWalletAddress().equals(walletAddress)) {

    list.add(transaction);
```

得到需要验证的交易列表后，SPV 就可以验证这些交易了

（3）对于每笔待验证的交易，轻节点向全节点发送交易的哈希值

```
String txHash = SecurityUtil.sha256Digest(transaction.toString());
```

代码中是 simplifiedPaymentVerify 在中调用 minerPeer 类中 getProof（）方法

```
Proof proof = minerPeer.getProof(txHash)
```

（4）全节点（getProof（））接收到待验证的交易哈希后就开始计算并返回验证路径：

首先是定位到目标的区块及交易

```
for (Block block: blockChain.getBlocks()) {

    for (Transaction transaction: block.getBlockBody().getTransactions()) {
```

定位不到就说明交易被篡改或者还没入链，当然这里不会出现这样的情况

定位到之后就开始计算路径：

在代码中就相当于重现一次 merkleRootHash 的计算过程，只不过在计算的时候，需要把与目标交易哈希有关的临界点记录下来，并且记录是他的左邻居还是右邻居。

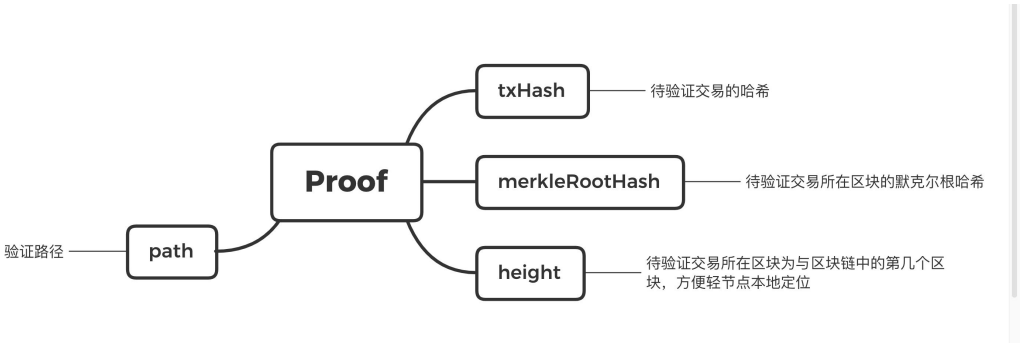
```
if (pathHash.equals(leftHash)) {

    Proof.Node proofNode = new Proof.Node(rightHash, Proof.Orientation.RIGHT);
```

```
    proofPath.add(proofNode);
    //更新目标交易哈希以供下一次迭代
    pathHash = parentHash;
} else if (pathHash.equals(rightHash)) {
    Proof.Node proofNode = new Proof.Node(leftHash, Proof.Orientation.LEFT);
    proofPath.add(proofNode);
    pathHash = parentHash;
}
```

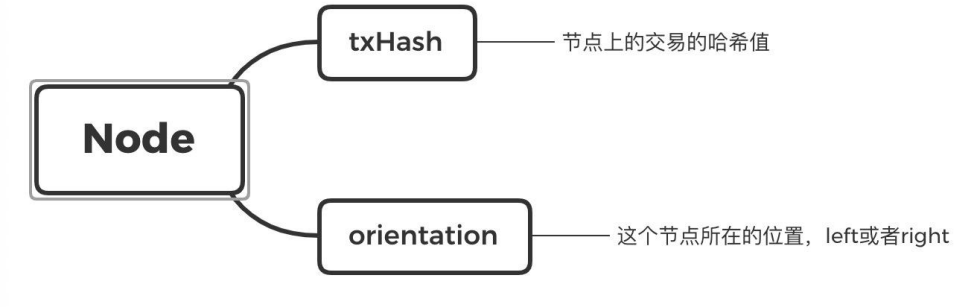
记录完之后在关系目标交易哈希为上次计算的父母节点就好了

这样，在 merkleRootHash 诞生的时候，所有与待验证交易有关的路径都已经被记录好，存放于 proofPath 中

最后矿工节点打包好之后将新打包成 proof 包返回给轻节点。proof 包的结构位于 Proof 类中



path 是由路径上的交易节点依次排列组成串



（5）轻节点得到返回的验证路径后开始本地验证（在 simplifiedPaymentVerify（）方法中）

通过交易路径，计算出 merkleRootHash

通过遍历交易路径中的节点，与本地的交易哈希做拼接和哈希计算，在计算的过程需要观察交易路径节点 node 中位置变量 orientation，注意拼接过程中两别哈希的前后顺序

```
String hash = proof.getTxHash();
for (Proof.Node node: proof.getPath()) {
    switch (node.getOrientation()) {
        case LEFT: hash = SecurityUtil.sha256Digest(node.getTxHash() + hash); break;
        case RIGHT: hash = SecurityUtil.sha256Digest(hash + node.getTxHash()); break;
        default: return false;
    }
}
```

（6）最后从 proof 包中得到全节点中远程的 merkleRootHash--remoteMerkleRootHash

```
String remoteMerkleRootHash= proof.getMerkleRootHash();
```

基于 proof 中的 height，定位本地的 merkleRootHash--localMerkleRootHash

```
String localMerkleRootHash = headers.get(height).getMerkleRootHash();
```

比对通过交易路径计算出来的 hash，和 localMerkleRootHash，remoteMerkleRootHash，如果三者相等，那么验证成功

```
return hash.equals(localMerkleRootHash) && hash.equals(remoteMerkleRootHash);
```

“双花”问题

一笔 UTXO 同时被用作两个地方，因为在交易没有每被打包加入区块链后，这笔 UTXO 是没有被记录的，因此会被认为这笔 UTXO 没有被使用。

实验三的代码为了解决这个问题，恢复交易池对多个用户的支持，在交易池类中增加了一个变量用来存储已经使用了 UTXO

```
private final Set<UTXO> utxoSet;
```

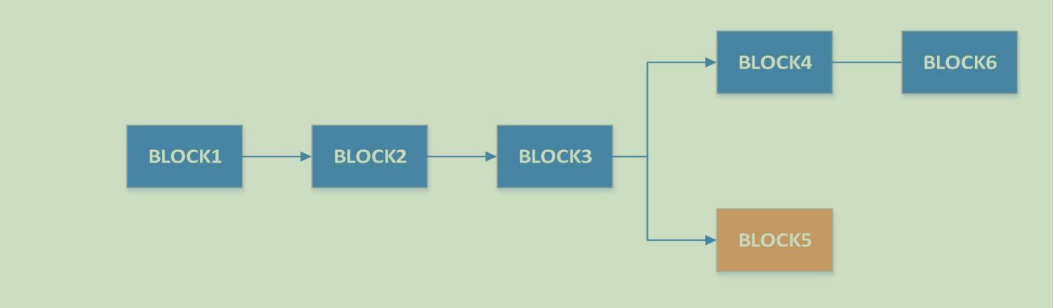
对于每笔想要入池的交易，首先得看看交易中的 inUTXOs 已经在 utxoSet 中，如果在那就说明这比较中存在一笔 inUTXO 已经被使用，这笔交易不能进入交易池

```
for (UTXO utxo: transaction.getInUtxos()) {
    // 如果包含已使用的 utxo，则拒绝本次交易进入交易池
    if (utxoSet.contains(utxo)) {
        return;
    }
}
```

对于满足入池条件的交易，需要将这些交易的 inUTXO 存入 utxoSet，因为这些 UTXO 已经在这笔交易中作为 inUTXOs 被使用。

```
utxoSet.addAll(Arrays.asList(transaction.getInUtxos()));
```

当然，现实中也会存在一笔 UTXO 几乎同时被使用到不同的地方，然后同时打包接入区块链末尾，形成分叉



然而，区块链系统中的用户已经达成公式，他们只会承认最长的那条链，当一个条链的长度超过另一条链的长度大约 6 的区块后，那条短的链就会被退回。那笔 UTXO 最终也只被使用了一次。

“双花”问题得以解决。

五、总结

学到了“实验目的”所需要掌握的东西，代码逻辑已经搞懂

感叹学长扎实的代码功力和理论基础，自愧不如

由于代码极大部分不是原创，所以实验报告写的心力交瘁，写了四天以上，因为不想在实验报告上拉垮导致分数不高。