华东师范大学数据科学与工程学院实验报告

课程名称：区块链与分享型数据库　　**年级**：2018 级　　**上机实践成绩**：
指导教师：张召　　**姓名**：池欣宁　　**学号**：10185501409
上机实践名称：区块链系统(minichain)的简单实现　　**上机实践日期**：2021.3.28
上机实践编号：　　**组号**：　　**上机实践时间**：

一、实验目的

通过代码的阅读和编写，从工程的角度理解区块链。在实践中加深对理论课知识学习的理解与体会。

二、实验任务

(1) 基本区块的基本结构及其必要属性
(2) 计算区块的哈希值/POW/挖矿
(3) 签名与验签--数字签名如何解决比特币中的身份认证问题
(4) UTXO 功能
(5) SPV 轻节点模块与验证

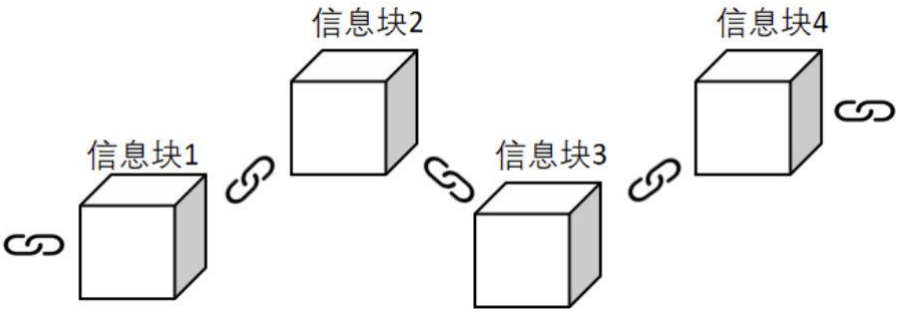
三、使用环境

Java 环境以及 IDE：IDEA：2020.2.3

四、实验过程

第一部分：基本区块的基本结构及其必要属性

区块链的简单理解是一个个区块串成的链，即一条散列值组成的链。也可以将它理解成一种仅提供**增查**功能特殊的**分布式**数据库。



区块：区块头与区块体

区块的结构定义

区块头	blockHeader;
区块体	blockBody;

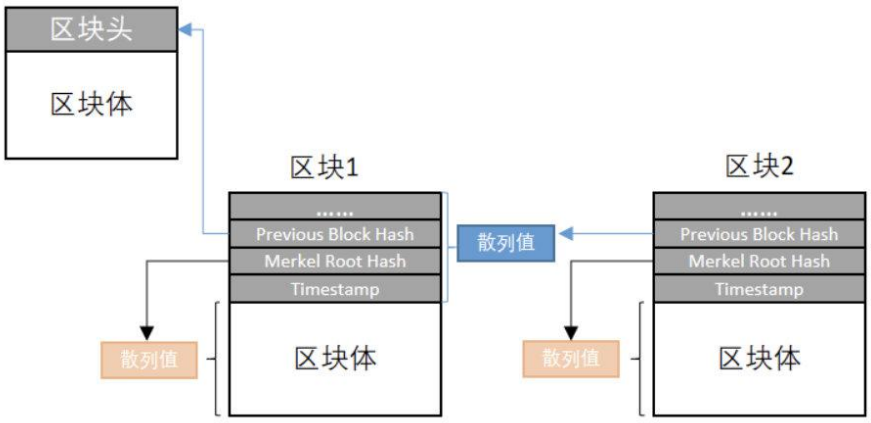
区块头(与真实 BlockChain 相比，略去 Version/Difficulty 等信息)

preBlockHash	前一个区块的哈希值
merkleRootHash	当前区块的哈希值

timestamp	时间戳
nonce	规定新区块的区块头的散列值必须满足某种特性，也是挖矿时在计算寻找的值。

区块体

transactions	从交易池(Transaction Producer)中取得的一批次交易
merkleRootHash	将区块中的这一批交易组织起来的用于交易验证的 Hash Value



由左图所示，区块体中任何一个字节变化->MerkleRootHash 的值会发生变化，Merkle Root Hash 在区块头里，区块头的散列值就会变化

```
chixinning@mymac ~/Desktop/ClassNotes4S2021/区块链与分享型数据库/BlockChainPy
main ± python3 main.py
创建一个新区块链

Block Hash: b27d1866e636de1c63945cc760d6bdc641de51d44365674436f11093bccc707f
Prev Block Hash:
TimeStamp: 1616740749
Data: 创世区块

Block Hash: 13f735701060600da0b219bd6fc4c1db1c1406a82e8ddfeafbae55571b81200a
Prev Block Hash: b27d1866e636de1c63945cc760d6bdc641de51d44365674436f11093bccc707
f
TimeStamp: 1616740749
Data: 第一个交易是转账10块

Block Hash: 56d50f2c4482b0db4169579ef640b1d764f3c41c92f89fbe1fce803b25df116b
Prev Block Hash: 13f735701060600da0b219bd6fc4c1db1c1406a82e8ddfeafbae55571b81200
a
TimeStamp: 1616740749
Data: 第二个交易是转账20
```

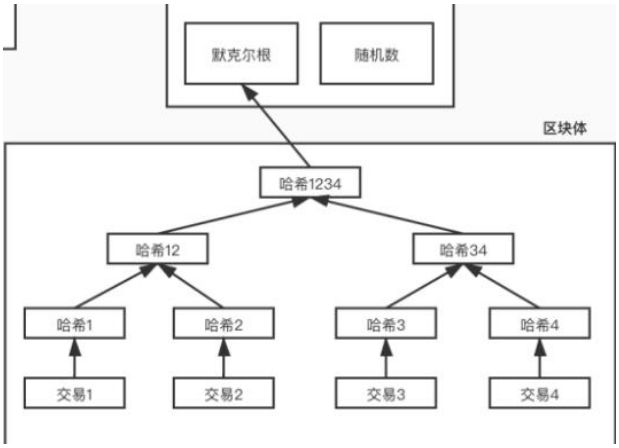
Genesis Block：比较特殊的区块，它的 PerviousBlock Hash 值便为空。

使用 python 同样可以简单地复现区块间最简单的逻辑，与课程 Java 代码相同，区块链是一条散列值组成的链。

第二部分：计算区块的哈希值/POW/挖矿

为了使区块链这个数据库具有公信力,即不是任何人都可以向区块链中新增一个区块。如果任何人都可以向区块链条中新增区块，那么随便任何人都可以说自己有过某些交易了，通过引入 POW 机制，对新增区块加以限制条件。

getBlockBody(Transactions[] transactions)的编写:



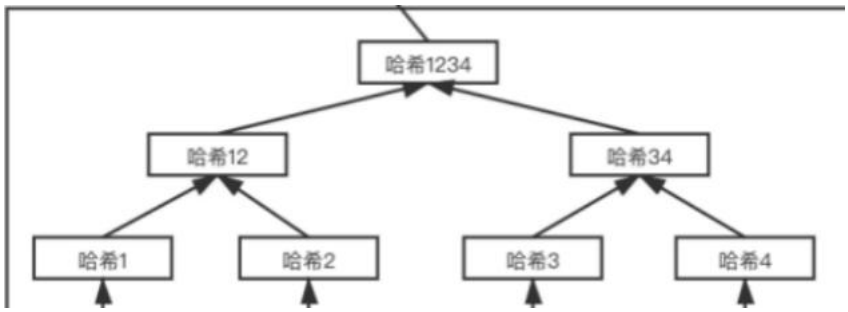
在第一部分区块的结构中可以发现，blockBody 部分由一组 Transactions[] transactions 组成。所以该函数的实现思路是，遍历传入的参数 transactions，将每一个交易首先计算哈希值，得到一个计算好哈希值后的数组。
然后在这个数组进行满二叉树自底向上的层次遍历，两两一组，得到字符串的拼接后，再计算一次哈希值保证 hashValue 的值的长度不变(防止字符串的简单拼接越加越长)。两两一组计算好的新哈希值，如图中的哈希 12，再加入到这个队列中，直到队列的大小为 1，便得到了默克尔根。

下面是我自己第一周写的 getBlockBody 代码如下：

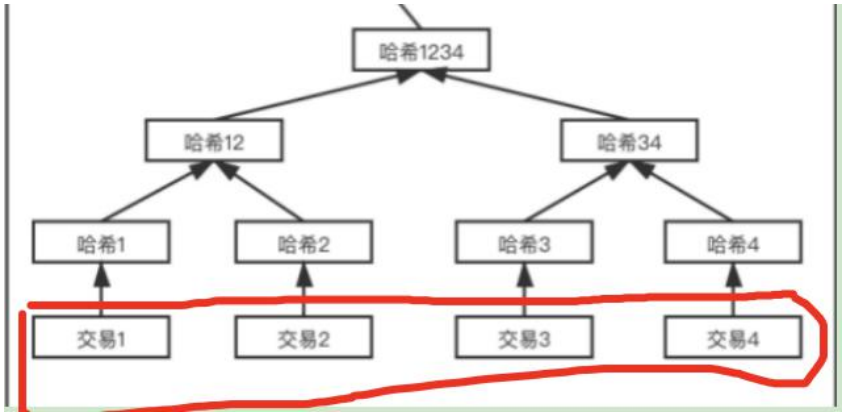
```
public BlockBody getBlockBody(Transaction[] transactions) {
    assert transactions != null && transactions.length == MiniChainConfig.MAX_TRANSACTION_COUNT;
    //todo
    //计算merkleRootHash
    //拿这64笔交易去计算
    //满二叉树的层次遍历，自底向上；merkleRootHash
    //字符串链接 掉哈希函数再算一遍
    List<String> hash_list = new ArrayList<>();
    for (int i =0;i<transactions.length;i++){
        String hash_value=SHA256Util.sha256Digest(transactions[i].toString());
        hash_list.add(hash_value);
    }
    while (hash_list.size()>1){
            int i=0;
            String hash1=hash_list.get(0);
            String hash2=hash_list.get(1);
            String hash_all=hash1+hash2;
            String hash_value=SHA256Util.sha256Digest(hash_all);
            hash_list.add(hash_value);
            hash_list.remove( index: 0);
            hash_list.remove( index: 0);

    }
    String merkleRootHash=hash_list.get(0);
    BlockBody blockBody= new BlockBody(merkleRootHash, transactions);
    return blockBody;
}
```

这里是层次遍历Hash后的数组

课后跟同学交流后思考了一个问题，即是先对每个交易取 HashValue 加入到队列中，再开始层次遍历，还是先将所有未哈希的交易放入队列中直接进行取哈希的层次遍历操作。
最后实践证明发现，如果是先对每个交易取 HashValue 加入到队列中，再开始层次遍历，即上面代码的过程，它层次遍历的树，是如下图所示：



但是，如果是先将所有未哈希的交易放入队列中，再直接进行取哈希的层次遍历操作，它所遍历的数组则是所有 HashValue 的值再加底下的一层交易数组，即并不是遍历的一个满二叉树，即如下图所示，所以会导致出错。

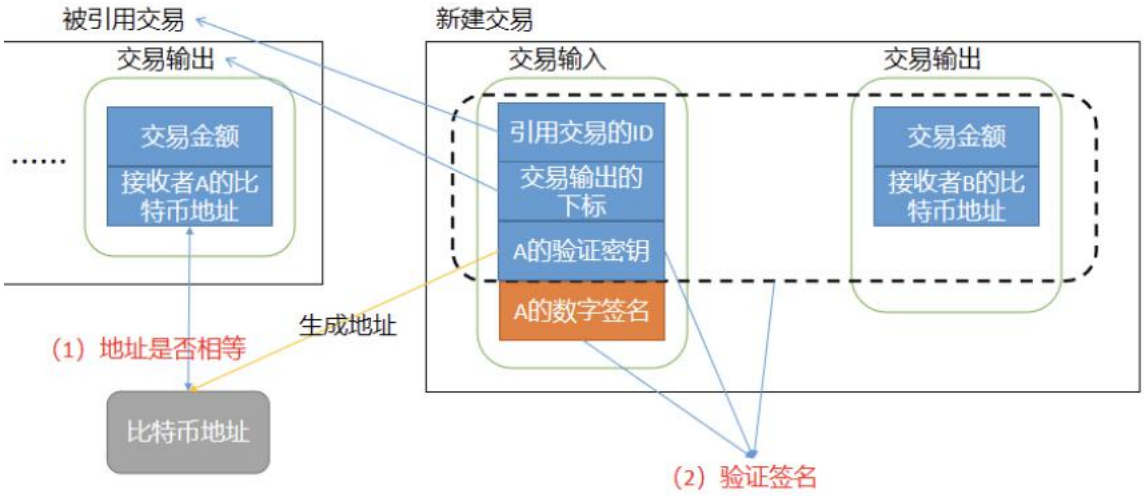


Mine 函数：穷举计算哈希值的过程
挖矿的本质，为区块找到一个合适的 Nonce 值(代码中使用 while 死循环一次次尝试)，使得区块头的散列值是一定数目(这里的数目由 config 中的 difficulty 决定，也与 blockHeader 中所记录的 difficulty 相同)的前导 0。
getBlock 函数
将得到的 blockBody 和 blockHeader 打包，即可得到一个新的 block。

第三部分：签名与验签--数字签名如何解决比特币中的身份认证问题。

区块链的密码学基础：
签名：数字信息->单向散列函数->散列值
密码学哈希和数据结构中的哈希有明显的不同:密码学哈希不可逆，也不会像数据结构哈希这么容易碰撞，其次，当输入信息发生微小变化时，会导致通过单向散列函数计算得到的散列值发生巨大的改变(雪崩现象)。
非对称加密与**数字签名**起到了身份鉴别与方否认的作用，根据理论课，在比特币系统中，账户的私钥用来签名，公钥用来验签和获得钱包地址。比特币的发送者在发起一个交易时可以使用其签名密钥来对交易信息进行签名，并将签名信息一并放入交易输入中。

签名交易：只要发送者签名了该交易后，根据数字签名的特性，发送者事后无法否认这笔交易。



钱包地址：比特币中将验证密钥编码成的易于阅读及记忆的字符串信息，即是在 getWalletAddress 中根据账户的公钥计算钱包地址。比特币地址是对版本号、验证密钥的散列值和校验码这三部分信息进行 Base58 编码的结果；

第四部分：UTXO 模型

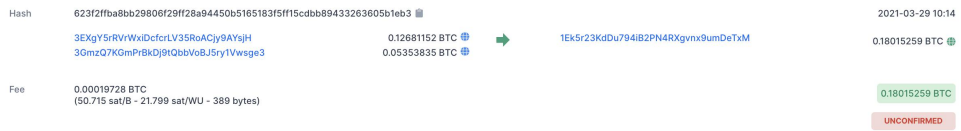
交易：

在第一周的实验课中，对于交易的内部，我们的交易类的数据就是一个名为 data 的字符串(String)，为了更贴合真实的应用场景，交易表示的是比特币的所有权从一个或多个账户向另外一个或多个账户转移的过程。

比特币交易的实现结构是由一个交易输入列表(inputUTXO)和一个交易输出列表(outputUTXO)构成:

如下图所示是真实场景中的一个交易:

Source:https://www.blockchain.com/explorer



课程代码中，交易的来源于我们模拟实现 genesisTransactions 函数，在 Transaction 类中添加相应函数，该函数会创建一批输出 utxo，为每个账户提供一笔金额，区块链中，有 0 个交易输入的交易是一种特殊的交易，被称为铸币交易（Coinbase），铸币交易可以理解成比特币的发行过程，矿工在完成 pow，付出了一定代价以后，将一个新的区块加入区块链，这些新产生的比特币的所有者就是这些矿工，这样一来，矿工拼命付出 cpu 计算力确实可以很形象地理解成挖矿的过程。

在 transaction 类中，由 inputUTXO 和 outputUTXO 构成了一个 transaction：

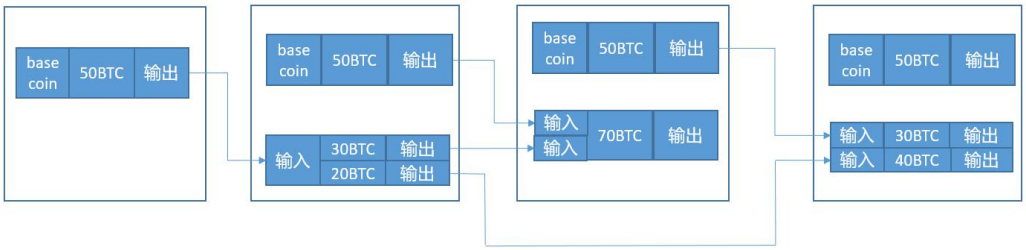
InputUTXO 和 outputUTXO 的结构:

walletAddress:比特币账户钱包地址

amount：交易的比特币数目

publicKeyHash:对公钥进行哈希摘要作为解锁脚本的数据。

所以 UTXO 就可以被形象的这样理解如下图：



UTXO 模型中，并没有账户余额的概念，在课程代码中，新增了一个那查询出某个账户可使用的 UTXO（有些已经被使用了）的功能，这个功能可以类比于账户余额的概念去理解。这个函数需要以 walletAddress(账户持有者的身份表示)为参数，通过遍历整个区块链每个块中每个交易的 inputUTXO 和 outputUTXO(与真实的比特币系统中通过遍历所有交易输出为指定账户的交易记录来求得该账户的余额一致)，并判断它们的钱包地址(walletAddress)是否是符合当前查询的地址。

练习题：构造一个确定的 UTXO 交易与构造一个复杂的 UTXO 交易（选做）：

因为在课程样例代码，getOneTransaction 的过程非常的清楚，即一笔交易的生成，在本课程中模拟为首先找出两个或多个账户，A 用户首先通过解锁，获取自己账户的使用权，再获取账户 A 的可用余额(getTrueUTXO），选择向 B 转账，同时输入对方的公钥以供生成公钥哈希，即与 B 账户构建 outputUTXO。当得到 outputUTXO 以后，A 账户在本地对整个交易进行私钥签名。

在这里，如果要构造确定的交易，只需要把账户改成确定的账户和确定的金额即可。而对于构造一个复杂的 UTXO 交易，只需要随机生成一个共同的私钥签名即可。

通过下图，可以很清楚的看到交易和找零过程：

因为实验 2 中的 transaction 的数量限制为 1，第一个 inputUTXO 由 daydream 的铸币交易生成。后续实验中，会对 transaction 的数量进行调整，且后面除了第一个 inputUTXO 都是真实的。

第五部分：SPV 轻节点模块与验证

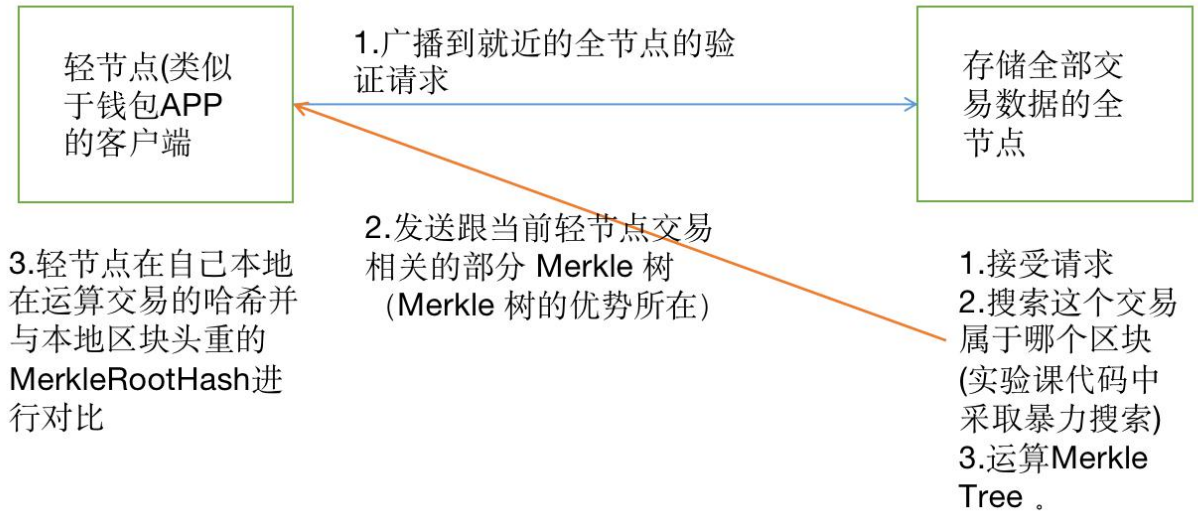
SPV：轻客户端需求 v.s.挖矿需求:安装全节点，很多时候是因为我们要挖矿。而安装轻客户端，通常就是把它当成一个钱包软件用。SPV 要解决的就是在轻客户端条件下的支付确认问题。

默克尔树:比特币系统中，使用默克尔树(哈希树)可以保证两个特点，默克尔树的典型应用场景包括：
快速比较大量数据：当两个默克尔树根相同时，则意味着所代表的数据必然相同（哈希算法决定的）。

快速定位修改。【Reference:https://zhuanlan.zhihu.com/p/39271872】
在比特币系统中，应用默克尔树，可以极大地提高了区块链的运行效率和可扩展性，使得区块头只需包含根哈希值而不必封装所有底层数据，同时，Merkle 树可支持“简化支付验证协议”（SPV），即在不运行完整区块链网络节点的情况下，也能够对交易数据进行检验。

默克尔树 v.s. Hash List：
在默克尔树中，验证交易是否存在，并不需要验证全部节点，而是只需要验证部分节点。HashList 则需要验证全部，在目前完整的 BTC 数据高达几百 GB 的情况下，这种数据开销对于简单验证无法容忍，在不存储完整数据的情况下，利用存储了全部数据的全节点验证某币交易是否在链上，则是 SPV 简单交易验证的动机。
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。

inputUTXO v.s. outputUTXO:【Reference:https://zhuanlan.zhihu.com/p/46292104
outputUTXO：轻客户端可以构建交易，并且对交易进行签名，最后广播到全网。
inputUTXO：轻客户端只会去下载跟自己的账户相关的交易。
SPV 便解决了在请客户端本地没有保存区块体的情况下的交易验证的问题。一个交易在区块链上生效需要经过**六次**确认。
下面是根据理论，结合相关上机代码，总结出的 SPV 验证的过程：



下图则是在本机上的 SPV 的实验结果：

```
Account[12dHWULjAk2BHFCCixz7AaT2bpyuVV4thgeA7gRMb2hYso4T8ha] began to verify the transaction...

------->verify hash:    88edd45fbf2ce9e4c2d307ec91e7d330961405bc74dfdc24656939050e274b6e
calMerkleRootHash:      2babec0eace5a01a8db4145030fb5cc2300f80abc1d61ae66c3b3c1ff10be2d7
localMerkleRootHash:    2babec0eace5a01a8db4145030fb5cc2300f80abc1d61ae66c3b3c1ff10be2d7
remoteMerkleRootHahs:   2babec0eace5a01a8db4145030fb5cc2300f80abc1d61ae66c3b3c1ff10be2d7
```

从结果可以看出 calMerkleRootHash,localMerkleRootHash 和 remoteMerkleRootHash 的值是保持一致的。

第六部分：其它

真实区块链场景与此次 minichain 架构中的不同：

1. Transaction 的数目限制的问题：

在第二次实验中，将 transaction 的数目限制为 1，引入了计算验证 UTXO 账户总额来验证 UTXO 交易场景是否正确，在后面与助教学长的讨论中了解到仅是在第二次实验中 transaction 的数目限制为 1 的情况下，才可以使用这种验证交易的方式，即这种验证的局限性。在第三次实验中，结合验证问题对 Transaction 的数目进行了修改。

2. 双花(UTXO)验证:

在第二次实验中，对于计算所有 UTXO 账户余额的方法不符合真实的应用场景，所以在第三次实验中进行了改进，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，更贴近了理论课上所学的验证问题。

3. 初始的 UTXO 产生：

在真实的区块链场景中，初始的比特币由铸币交易产生，在课程代码对区块链的模拟中便是 theyHaveADayDream 函数，它在第三次实验被修改后放在了 Network 类下，对于 miniChain 的代码整体结构朝着更符合逻辑的结构调整。

五、总结

在此次实验中收获颇丰，受益匪浅，现总结如下：

① **在实验中加深了很多 Java 编程思想的理解**，比如第三次实验中的对于 OOP 编程思想的学习，是“高内聚，低耦合”的思想的一次很好的体验和时间。第三次实验中对整体结构中的调整，将很多功能的实现放入到与 network 的交互过程中，而不是像第一次代码中有很多类间与类间的交互，即让 network 类作为中心，更好的展现出它与区块链中其他功能和对象的交互作用。

② **同时，在实验中学习了很多 Java 编程的基本语法和 Java 的特性**:

比如，**包(Package)**是 Java 为了组织应用于某一功能或功能相同的类引入的，在 miniChain 实验中，data 数据包、util 通用工具包便是如此。包可以类比于其它编程语言来理解，如在 C++中相当于命名空间以及 Python 中也有包的概念。

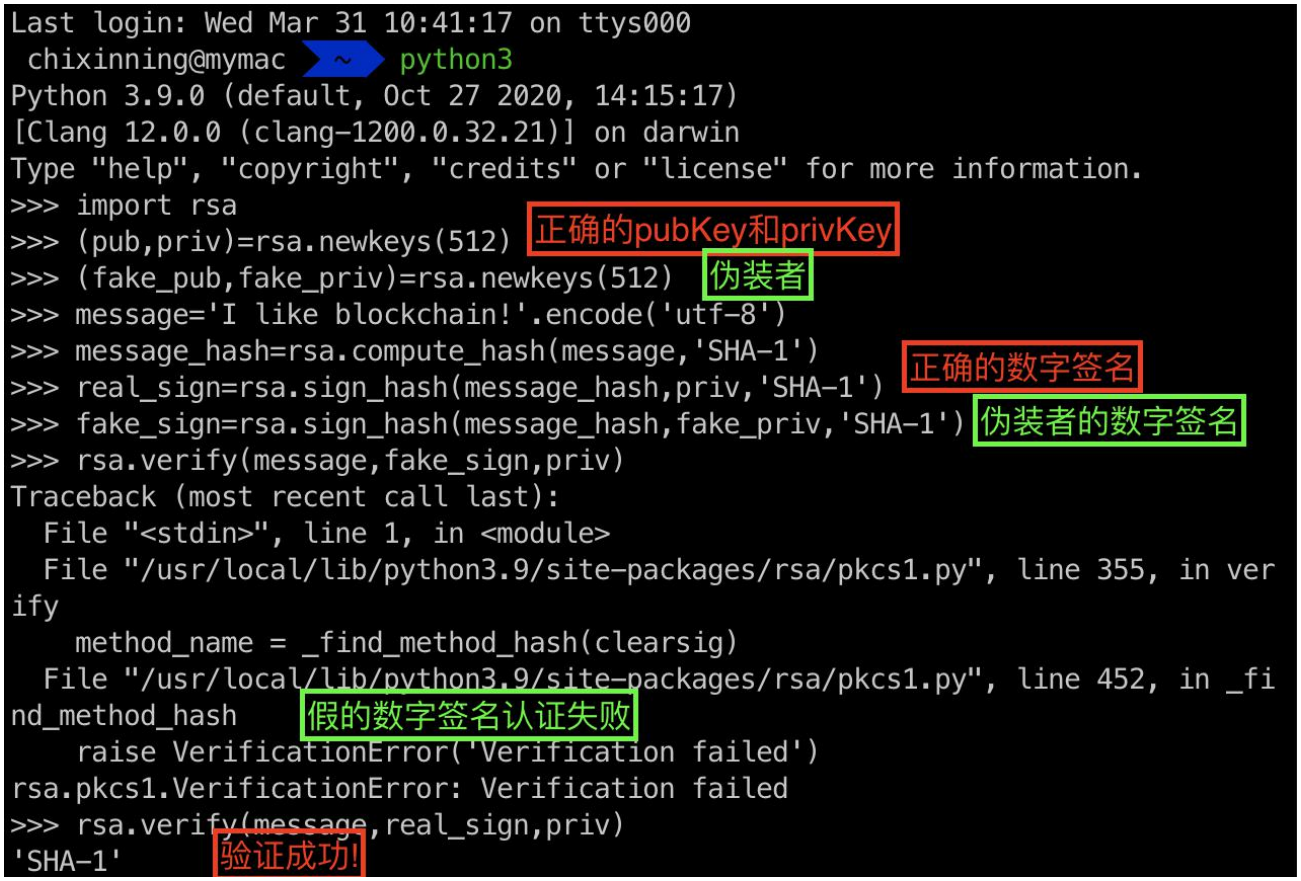
Java 中 **Final 关键字**的定义与使用，在这里是为了简单模仿不可篡改的特点。

Transaction producer 和 Miner 中线程(Thread)的使用，当 transaction pool 满了以后，transaction producer 则进入休眠状态。

同样的，还有 **IDEA** 很多神奇的功能,快捷键以及强大的代码订正和修改能力，如 Generate 功能中的 Getter 与 Constructor，toString 的改写；

积累了一些发现编译报错的原因的经验，以及阅读工程项目代码的能力，对于 java 编程的规范，变量的命名，甚至是注释添加相关，都有了一定的学习与提升。

③ **对于区块链的理论原理的理解有了更深的认识。**在课后花了很大的功夫理解比特币背后的密码学基础，即对称加密与非对称加密的理解。在网上阅读相关材料时，对非对称加密有一句理解十分贴切：A 给 B 了一把打开的锁，A 需要 B 用它锁住重要的东西寄回给他。钥匙需要 A 自己留着谁也不给。在这里锁=公钥；钥匙=私钥；同时在课后使用 python 也简单进行了非对称加密理解的一些尝试如下图:



其余的一些理解与体会在实验过程中有所体现，在此不做赘述。