

实验 2 签名、验签以及 UTXO 的简单实现

【实验介绍】

在实验 1 中，为了方便地建立起一个区块链系统，我们简化了交易的结构，仅仅使用 UUID 作为交易的内容。以比特币为例，交易的内容一般为用户转账，携带着用户的信息和转账金额数。为了保障用户在网络中的匿名性和交易的可验证性和安全性，区块链系统一般使用地址来指代某个用户，并使用数字签名（Digital Signature）技术来辨别交易的真伪。同时，为了解决数字货币中的双花问题，即一笔钱被花费了两次，比特币使用 UTXO（Unspent TX Output）作为交易模型。本次实验的内容是在实验 1 的基础上添加账户模块（可看作一个简单的钱包）和 UTXO 模块，并实现交易的签名及验签功能，同时会修改其它模块以适应新功能。

在进行实验操作之前，我们首先了解非对称加密和数字签名的相关概念以及学习 UTXO 模型的工作原理。

(1) 非对称加密

非对称加密是指在使用前生成一个公钥(Public Key)和对应的私钥(Private Key)，在加密明文和解密密文的过程中，使用不同的密钥，即一个用于加密，一个用于解密。非对称加密虽然速度较慢，但其公私钥分开的优点，不用向外分发解密的私钥，安全性大大提高，多应用于签名验证和数字身份场景。

在非对称加密算法应用中，用私钥加密的数据要用对应的公钥才能解开，用公钥加密的数据要用对应的私钥才能解开。常见的非对称加密算法有 RSA、DSA（Digital Signature Algorithm）数字签名算法、ECC（Elliptic curve cryptography）椭圆曲线密码算法、ECDSA（Elliptic Curve Digital Signature Algorithm）椭圆曲线数字签名算法——DSA 和 ECC 的结合。

比特币中的私钥本质是一个密码学安全的随机数，通过 secp256k1（一组特定参数的 ECDSA）变换后得到公钥，公钥再经过 SHA256 和 RIPEMD160 哈希（两者结合又称 Hash160），添加前导字节并进行 Base58Check 编码，得到可读性高的短地址。从私钥到公钥再到地址都是单向的算法，以目前的算力几乎是不可逆的，保证了用户私钥的安全性。比特币中私钥、公钥和地址的关系如图 2-1 所示。

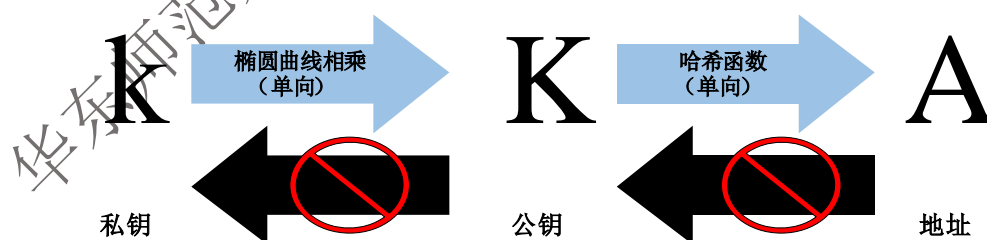


图 2-1 私钥、公钥、地址关系图

(2) 数字签名

数字签名在区块链中有着广泛的应用。从功能上来讲，数字签名与我们在纸质文件上的签名类似，是一种鉴别信息来源以及信息真伪的方法。从技术上来讲，数字签名使用密码学中的哈希（Hash）算法、非对称加密（Asymmetric Cryptography）算法来保证以下三点：①确认信息是由签名方发送的；②确认信息在传输过程中没有受到修改；③确认信息在传输过程中没有出现丢失。非对称加密实现数字签名如图 2-2 所示。

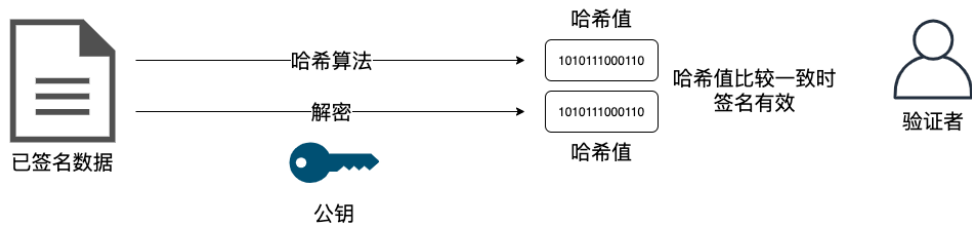


图 2-2 非对称加密实现数字签名

(3) UTXO 模型

比特币的区块链由一个个区块串联构成，而每个区块又包含一个或多个交易。如果我们观察任何一个交易，它总是由若干个输入（Input）和若干个输出（Output）构成，一个 Input 指向的是前面区块的某个 Output，只有 Coinbase 交易（铸币交易）没有输入，只有凭空输出。如图 2-3 所示，这些交易的 Input 和 Output 总是可以串联起来。

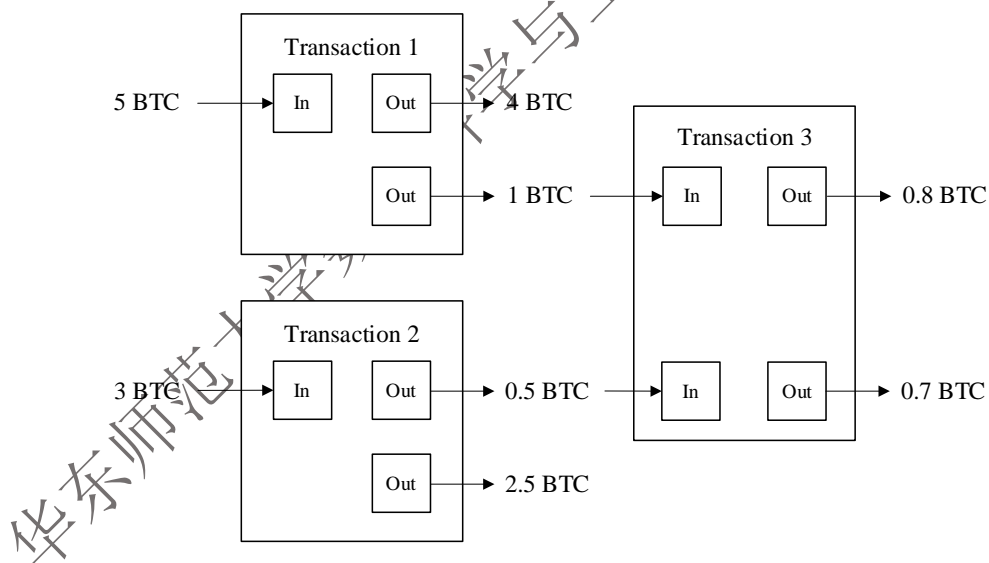


图 2-3 区块链交易间的关系

UTXO（Unspent TX Output）译为未花费的交易输出，UTXO 模型的设计基于一种思路：除了铸币（Coinbase）交易生成的比特币，任意一笔钱不会凭空产生，也不会凭空消失。UTXO 由四部分组成：

- 1) address: 拥有此 UTXO 的地址；
- 2) amount: 此 UTXO 的金额；
- 3) Signature Script: UTXO 解锁脚本，使用交易发送方私钥加密交易内容得到的签名；

4) Pubkey Script: UTXO 锁定脚本，包含交易获得方的公钥哈希。

如图 2-4 所示，展示了 Alice 对 Bob 进行一笔转账交易的过程：假设 Alice 有 amount 个比特币，这其实意味着，之前有一个交易把这些比特币转入 Alice 的地址，这个交易的输出（即 amount 个比特币）未被使用，Alice 拥有了这 amount 个比特币。

- 1) Alice 使用公钥解锁自己的 UTXO，构造新的 UTXO 和一笔转账交易；
- 2) Alice 使用私钥对 UTXO 的交易内容进行签名；将 UTXO 的输出地址设为 Bob 的钱包地址，并把 Bob 的公钥 Hash 作为解锁此 UTXO 的凭据；
- 3) 矿工将这一交易打包进新的区块，转账交易完成。

经过 6 个区块确认（证明该笔交易极大概率合法）后，这 amount 个比特币就属于 Bob 了。实际上 Bob 拥有的是这笔转账交易的 UTXO。

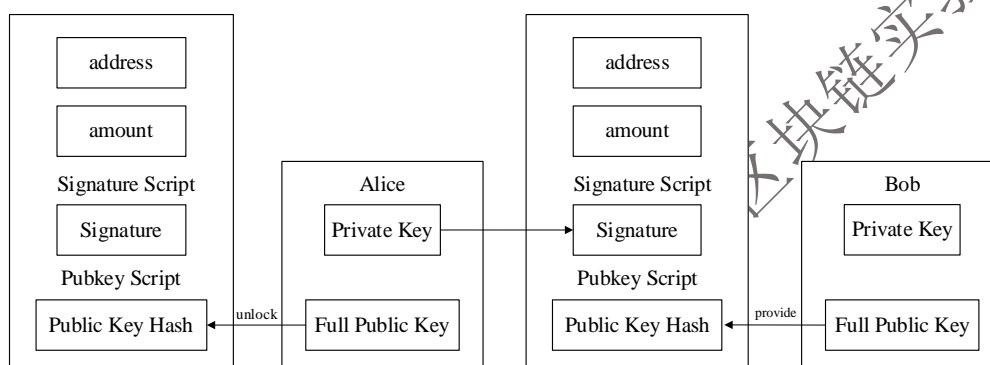


图 2-4 基于 UTXO 模型，Alice 转账给 Bob

上面提及了比特币中的解锁脚本（Signature Script）与锁定脚本（PubKey Script）。比特币脚本是一种基于栈结构的无状态脚本语言，只能进行有限的操作，图灵不完备。虽然简单，但在数字货币领域，也意味着更少的金融风险。

在 UTXO 模型中，交易只是代表了 UTXO 集合的变更。而账户和余额的概念是在 UTXO 集合上更高的抽象，账号和余额的概念只存在于钱包中。以比特币钱包为例，钱包管理的是一组私钥，对应的是一组公钥和地址。要查看钱包余额，必须从创世区块开始扫描每一笔交易，如果：

- 1) 遇到某笔交易的某个 Output 是钱包管理的地址之一，则钱包余额增加；
- 2) 遇到某笔交易的某个 Input 是钱包管理的地址之一，则钱包余额减少。

所以从狭义上来说，比特币钱包中并没有比特币，只有钱包地址关联的所有 UTXO 之和，代表着钱包地址拥有这些比特币的所有权。

【实验要求】

- (1) 实现账户类与钱包功能
- (2) 实现 UTXO 类与解锁脚本功能
- (3) 实现账户签名交易，矿工验签功能

【实验准备】

1. 环境配置

本次实验使用和实验 1 相同的开发环境。

2. 导入代码

推荐读者使用自己在实验 1 完成的代码上进行此实验。若仅想学习数字签名和比特币 UTXO 的工程原理，也可以直接导入附件里实验 2 的 minichain 压缩包进行实验，此压缩包含有实验 1 的参考代码。

【实验过程】

1. Util 工具类支持

如图 2-5 所示，右击 SHA256Util，重命名文件为 SecurityUtil，然后将附件中实验 2 文件夹下的 SecurityUtil.java 中的内容复制并覆盖原有代码。

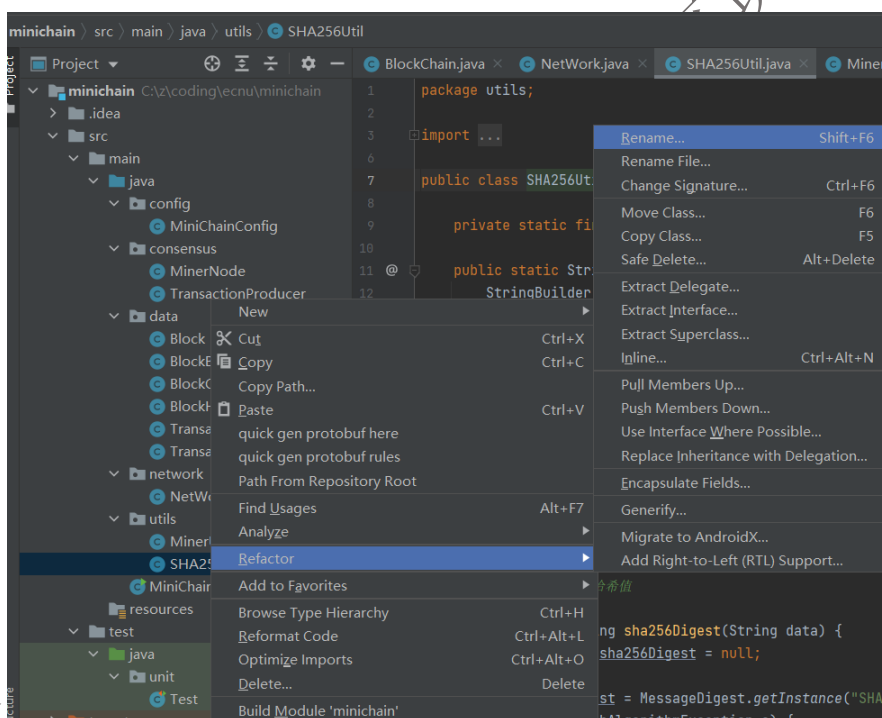


图 2-5 重命名 SHA256Util

在 utils 包下新建类 Base58Util，将附件中实验 2 文件夹下的 Base58Util.java 中的内容复制并粘贴到此类下，结果如图 2-6 所示。

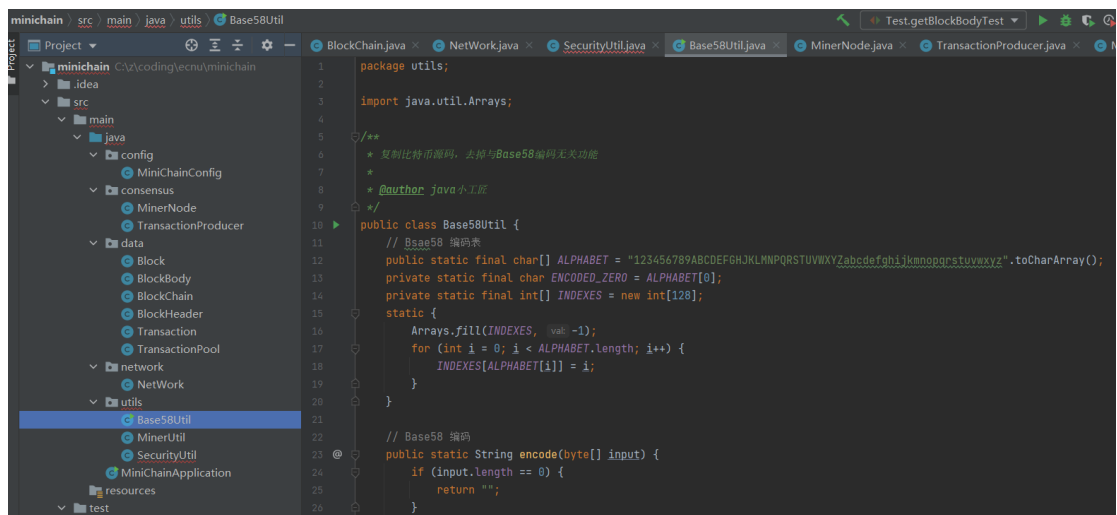


图 2-6 Base58Util.java

2. 新建 Account 类、UTXO 类

在 data 包下新建 Account 类和 UTXO 类，如图 2-7 所示。

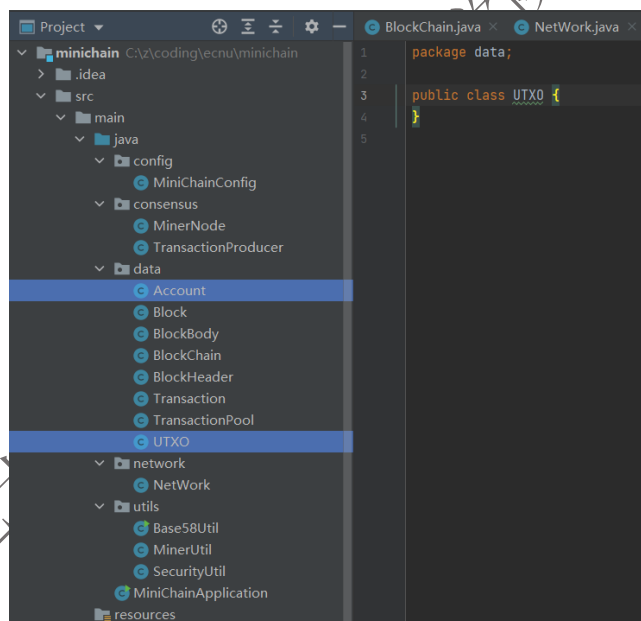


图 2-7 新建 Account 类和 UTXO 类

账户拥有公私钥，其通过椭圆曲线加密算法获得，故在 Account 类中添加构造方法如图 2-8 所示。

```

1 package data;
2
3 import utils.SecurityUtil;
4
5 import java.security.KeyPair;
6 import java.security.PrivateKey;
7 import java.security.PublicKey;
8
9 public class Account {
10
11     private final PublicKey publicKey;
12     private final PrivateKey privateKey;
13
14     public Account() {
15         KeyPair keyPair = SecurityUtil.secp256k1Generate();
16         this.privateKey = keyPair.getPrivate();
17         this.publicKey = keyPair.getPublic();
18     }
19
20 }
21

```

图 2-8 Account 构造方法

在实验介绍中，我们描述了比特币钱包地址的生成算法，算法流程如图2-9 所示。

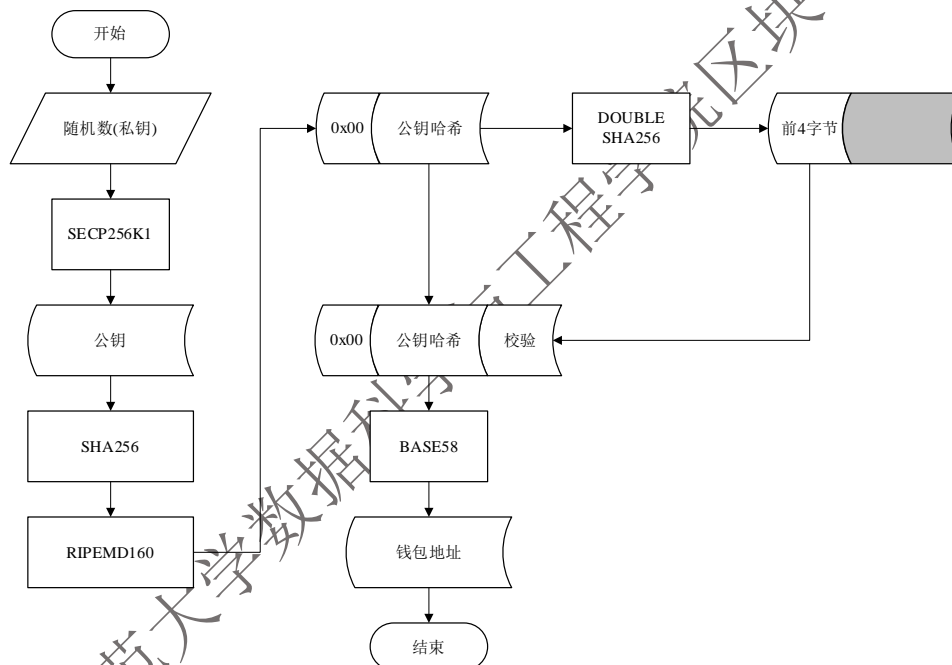


图 2-9 比特币钱包地址生成算法

编写对应的代码实现获得钱包地址的方法，如图 2-10 所示。

```

21  /**
22   * 根据账户的公钥计算钱包地址
23   * @return
24   */
25  public String getWalletAddress() {
26
27      // 公钥哈希: RIPEMD160(SHA256(PubK))
28      byte[] publicKeyHash = SecurityUtil.ripemd160Digest(SecurityUtil.sha256Digest(publicKey.getEncoded()));
29
30      // 0x00 + 公钥哈希
31      byte[] data = new byte[1 + publicKeyHash.length];
32      data[0] = (byte) 0;
33      for (int i = 0; i < publicKeyHash.length; ++i) {
34          data[1 + i] = publicKeyHash[i];
35      }
36      // 两次sha256哈希摘要
37      byte[] doubleHash = SecurityUtil.sha256Digest(SecurityUtil.sha256Digest(data));
38
39      // 0x00 + 公钥哈希 + 校验 (两次哈希后前4字节)
40      byte[] walletEncoded = new byte[1 + publicKeyHash.length + 4];
41      walletEncoded[0] = (byte) 0;
42      for (int i = 0; i < publicKeyHash.length; ++i) {
43          walletEncoded[1 + i] = publicKeyHash[i];
44      }
45      for (int i = 0; i < 4; ++i) {
46          walletEncoded[1 + publicKeyHash.length + i] = doubleHash[i];
47      }
48
49      // 对二进制地址进行BASE58编码, 得到钱包地址 (字符串形式)
50      String walletAddress = Base58Util.encode(walletEncoded);
51
52      return walletAddress;
53  }

```

图 2-10 Account 类_方法_获取钱包地址

按下 Alt+Insert 快捷键快速生成相应的 getter 方法和 toString 方法, 修改 toString 的内容使其公私钥的输出为十六进制字符串形式, 代码如图 2-11 所示。

```

55  public PublicKey getPublicKey() { return publicKey; }
56
57  public PrivateKey getPrivateKey() { return privateKey; }
58
59  @Override
60  public String toString() {
61      return "Account{" +
62          "publicKey=" + SecurityUtil.bytes2HexString(publicKey.getEncoded()) +
63          ", privateKey=" + SecurityUtil.bytes2HexString(privateKey.getEncoded()) +
64          '}';
65  }

```

图 2-11 Account 类_方法_getter 和 toString

3. 完善 Account 类、UTXO 类

在实验介绍中, 我们给出了 UTXO 具体的数据结构。我们可以忽略签名脚本的具体实现, 完善 UTXO 类的构造方法。如图 2-12 所示。

```
1 package data;
2
3 import utils.SecurityUtil;
4
5 import java.security.PublicKey;
6
7 public class UTXO {
8
9     private final String walletAddress;
10    private final int amount;
11    private final byte[] publicKeyHash;
12
13    /**
14     * 构建一个UTXO
15     * @param walletAddress 交易获得方的钱包地址
16     * @param amount 比特币数额
17     * @param publicKey 交易获得方的公钥（公钥是公开的）
18     */
19    @ public UTXO(String walletAddress, int amount, PublicKey publicKey) {
20        this.walletAddress = walletAddress;
21        this.amount = amount;
22        // 对公钥进行哈希摘要：RIPEMD160(SHA256(PubK)，作为解锁脚本数据
23        publicKeyHash = SecurityUtil.ripemd160Digest(SecurityUtil.sha256Digest(publicKey.getEncoded()));
24    }
```

图 2-12 完善 UTXO 构造方法

实验介绍中，我们提到 UTXO 需要解锁才能被交易使用，解锁脚本与锁定脚本的模型如图 2-13 所示，解锁脚本算法如图 2-14 所示。

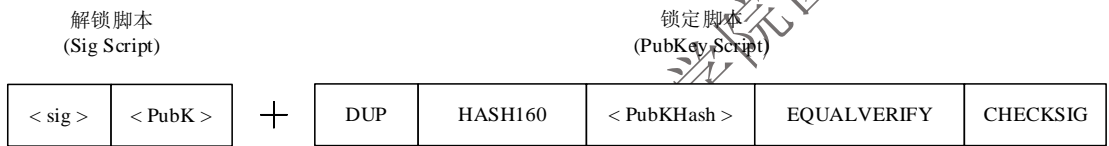


图 2-13 解锁脚本与锁定脚本模型

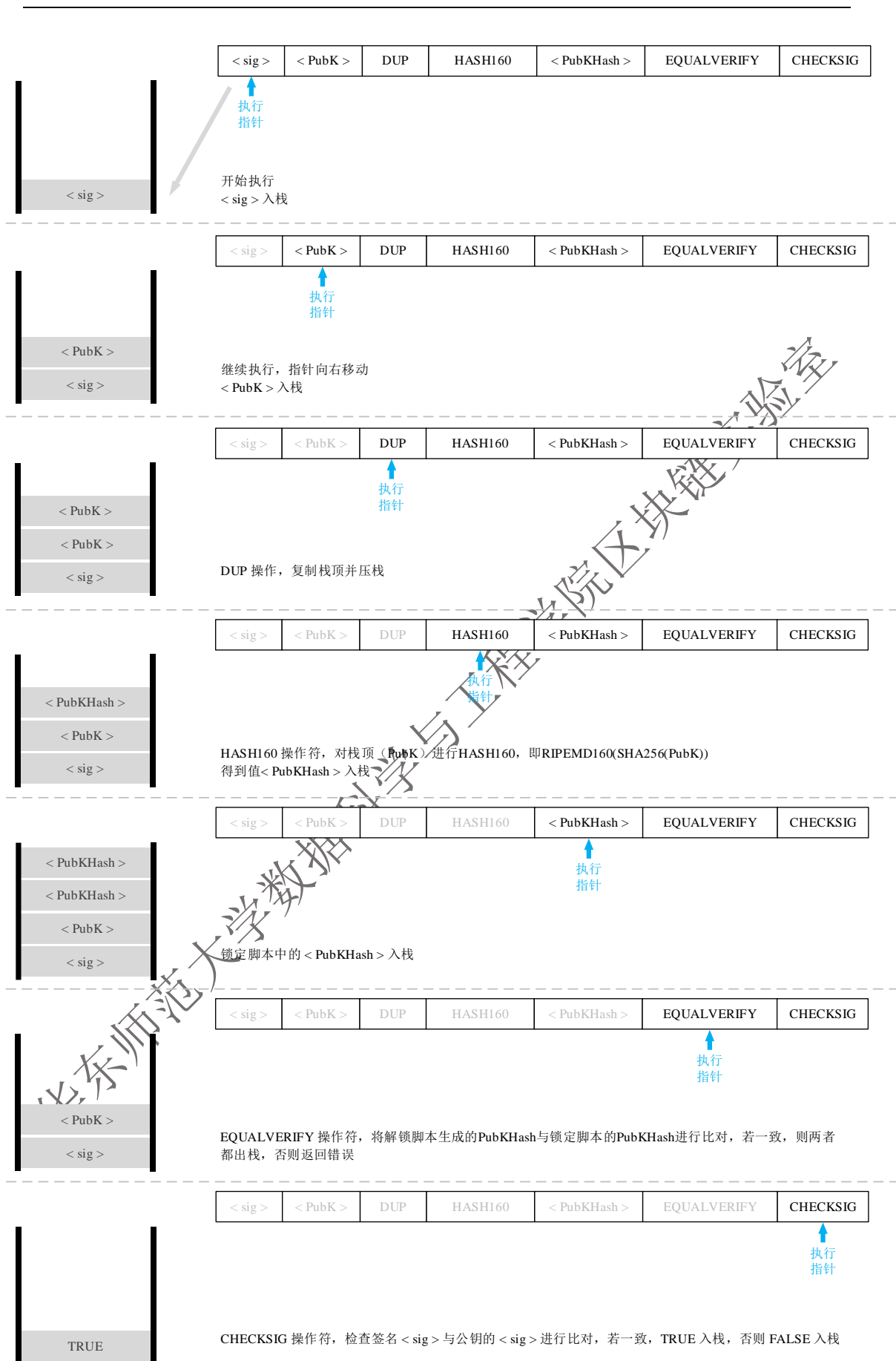


图 2-14 UTXO 解锁脚本算法

基于上述算法，我们使用 Java 实现一个解锁脚本方法，如图 2-15 所示。

```
34 @ public boolean unlockScript(byte[] sign, PublicKey publicKey) {
35     Stack<byte[]> stack = new Stack<>();
36     // <sig> 签名入栈
37     // 栈内: <Sig>
38     stack.push(sign);
39     // <PubK> 公钥入栈
40     // 栈内: <Sig> <PubK>
41     stack.push(publicKey.getEncoded());
42     // DUP 复制一份栈顶数据，peek()为java栈容器获取栈顶元素的函数
43     // 栈内: <Sig> <PubK> <PubK>
44     stack.push(stack.peek());
45     // HASH160 弹出栈顶元素，进行哈希摘要：RIPEMD160(SHA256(PubK)，并将其入栈
46     // 栈内: <Sig> <PubK> <PubHash>
47     byte[] data = stack.pop(); // 栈顶元素就是PubK
48     stack.push(SecurityUtil.ripemd160Digest(SecurityUtil.sha256Digest(data)));
49     // <PubHash> utxo先前保存的公钥哈希入栈
50     // 栈内: <Sig> <PubK> <PubHash> <PubHash>
51     stack.push(publicKeyHash);
52     // EQUALVERIFY 比较栈顶的两个公钥哈希是否相同，不相同则解锁失败
53     // 栈内: <Sig> <PubK>
54     byte[] publicKeyHash1 = stack.pop(); // 出栈并返回栈顶元素
55     byte[] publicKeyHash2 = stack.pop();
56     if (!Arrays.equals(publicKeyHash1, publicKeyHash2)) { // 一一比较比特数组内的每一个数据是否相等
57         return false;
58     }
59     // CHECKSIG 检查签名是否正确，正确则入栈 TRUE;
60     // 栈内:
61     byte[] publicKeyEncoded = stack.pop(); // 这里弹出的是二进制，在这里无法用来验签，故仍用 PublicKey形式的公钥验签
62     byte[] sign1 = stack.pop();
63     // 比特币网络中因为其脚本支持操作少的特性，需要入栈再检查，这里验证正确我们就直接返回了
64     // 栈内: TRUE (验证正确情况下)
65     return SecurityUtil.verify(publicKey.getEncoded(), sign1, publicKey);
66 }
```

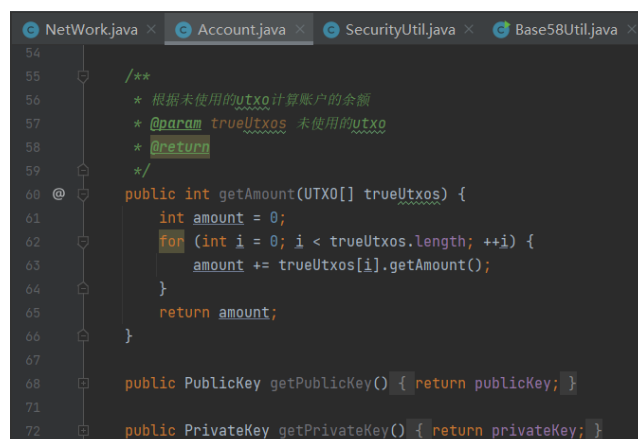
图 2-15 UTXO 类_方法_unlockScript

Alt+Insert 快捷键快速生成相应的 getter 方法和 toString 方法，修改 toString 的内容让公钥哈希输出十六进制字符串形式，添加换行符和制表符美化输出，代码如图 2-16 所示。

```
69 public String getWalletAddress() {
70     return walletAddress;
71 }
72
73 public int getAmount() {
74     return amount;
75 }
76
77 public byte[] getPublicKeyHash() {
78     return publicKeyHash;
79 }
80
81 @Override
82 public String toString() {
83     return "\n\tUTXO{" +
84         "walletAddress=" + walletAddress + '\t' +
85         "amount=" + amount +
86         ", publicKeyHash=" + SecurityUtil.bytes2HexString(publicKeyHash) +
87         '}';
88 }
89 }
```

图 2-16 UTXO 类_方法_toString

系统支持 UTXO 后，钱包就可以计算某个账户的余额，即将账户地址关联的 UTXO 汇总。回到 Account 类中，添加一个计算账户金额的方法，如图 2-17 所示。



```

54
55 /**
56  * 根据未使用的utxo计算账户的余额
57  * @param trueUtxos 未使用的utxo
58  * @return
59  */
60 @
61 public int getAmount(UTXO[] trueUtxos) {
62     int amount = 0;
63     for (int i = 0; i < trueUtxos.length; ++i) {
64         amount += trueUtxos[i].getAmount();
65     }
66     return amount;
67 }
68
69 public PublicKey getPublicKey() { return publicKey; }
70
71
72 public PrivateKey getPrivateKey() { return privateKey; }

```

图 2-17 Account 类_方法_计算账户余额

该方法的参数是 trueUtxos，意指与该账户关联的未被使用的 UTXO。而那些已经被使用了的 UTXO，我们将其称为 fakeUtxos。至此，我们完成了新类的添加，下面将在其他模块内进行修改。

4. Transactions 类

在实验 1 中，系统的交易类的数据就是一个名为 data 的字符串 (String)，现在我们实现了 UTXO，所以交易中要存放 UTXO。回顾实验介绍，一个交易包含若干个输入 UTXO 和若干个输出 UTXO。

故删除旧有的内部代码，修改 Transaction 类结构如图 2-18 所示。



```

9 /**
10  * 对交易的抽象
11  */
12
13 public class Transaction {
14
15     private final UTXO[] inUtxos;
16     private final UTXO[] outUtxos;
17
18     private final byte[] sendSign; // 交易发送方的私钥签名
19     private final PublicKey sendPublicKey; // 交易发送方的公钥，方便矿工和其他节点进行检验，确保交易违背篡改
20     private final long timestamp;
21
22     2 related problems
23     public Transaction(UTXO[] inUtxos, UTXO[] outUtxos, byte[] sendSign, PublicKey sendPublicKey, long timestamp) {
24         this.inUtxos = inUtxos;
25         this.outUtxos = outUtxos;
26         this.sendSign = sendSign;
27         this.sendPublicKey = sendPublicKey;
28         this.timestamp = timestamp;
29     }
30 }

```

图 2-18 Transaction 类修改

Alt+Insert 快捷键快速生成相应的 getter 方法和 toString 方法，修改 toString 的内容让签名公钥输出十六进制字符串形式，添加换行符美化输出，代码如图 2-19 所示。

```

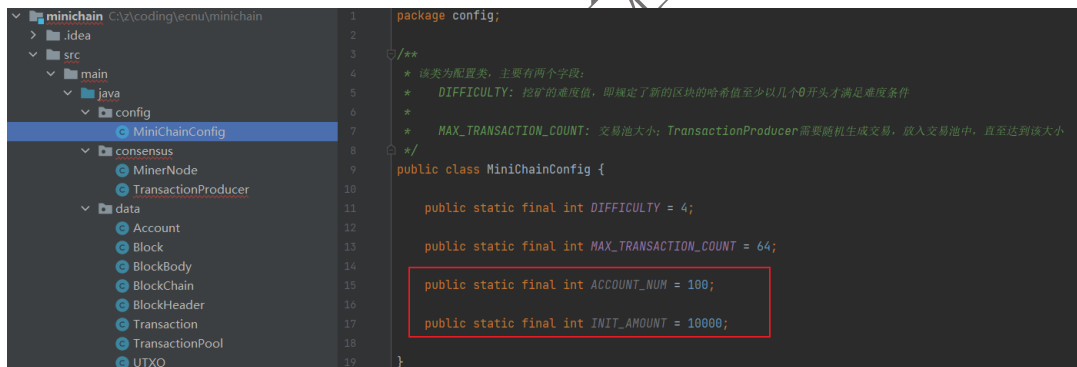
28     public UTXO[] getInUtxos() {
29         return inUtxos;
30     }
31
32     public UTXO[] getOutUtxos() {
33         return outUtxos;
34     }
35
36     public byte[] getSendSign() {
37         return sendSign;
38     }
39
40     public PublicKey getSendPublicKey() {
41         return sendPublicKey;
42     }
43
44     public long getTimestamp() {
45         return timestamp;
46     }
47
48     @Override
49     public String toString() {
50         return "\nTransaction{" +
51             "\n  inUtxos=" + Arrays.toString(inUtxos) +
52             "\n  outUtxos=" + Arrays.toString(outUtxos) +
53             "\n  sendSign=" + SecurityUtil.bytes2HexString(sendSign) +
54             "\n  sendPublicKey=" + SecurityUtil.bytes2HexString(sendPublicKey.getEncoded()) +
55             "\n  timestamp=" + timestamp +
56             "}";
57     }

```

图 2-19 Transaction 类_方法_toString

5. 添加账户

我们将初始化一些账户，并为他们提供一个固定金额的 UTXO，由于已有结构的限制，该账户数组将作为 BlockChain 类的数据成员。我们在 MiniChainConfig 配置类中配置相关参数，初始化账户为 100 个，初始 UTXO 金额为 10000，如图 2-20 所示。



```

1 package config;
2
3 /**
4  * 该类为配置类，主要有两个字段：
5  * DIFFICULTY: 挖矿的难度值，即确定了新的区块的哈希值至少以几个0开头才满足难度条件
6  * MAX_TRANSACTION_COUNT: 交易池大小，TransactionProducer需要随机生成交易，放入交易池中，直至达到该大小
7  */
8
9 public class MiniChainConfig {
10
11     public static final int DIFFICULTY = 4;
12
13     public static final int MAX_TRANSACTION_COUNT = 64;
14
15     public static final int ACCOUNT_NUM = 100;
16
17     public static final int INIT_AMOUNT = 10000;
18
19 }

```

图 2-20 MiniChainConfig 类修改

随后在 BlockChain 类中添加 Account 账户数组，然后修改构造方法，在初始区块中添加为这些账户提供的 UTXO，使每个账户在系统启动时就拥有 10000 比特币，便于后面交易的进行，代码如图 2-21 所示。

```

9  /**
10 * 区块链的类抽象，创建该对象时会自动生成创世区块，加入区块链中
11 */
12 public class Blockchain {
13
14     private final LinkedList<Block> chain = new LinkedList<>();
15     private final Account[] accounts; ①
16
17     public Blockchain() {
18         this.accounts = new Account[MiniChainConfig.ACCOUNT_NUM];
19         for (int i = 0; i < accounts.length; ++i) { ②
20             accounts[i] = new Account();
21         }
22         // 在创世区块中为每个账户分配一定金额的 utxo，便于后面交易的进行
23         Transaction[] transactions = genesisTransactions(accounts); ③
24
25         BlockHeader genesisBlockHeader = new BlockHeader( preBlockHash: null, merkleRootHash: null,
26                                                         Math.abs(new Random().nextLong()));
27         BlockBody genesisBlockBody = new BlockBody( merkleRootHash: null, transactions);
28         Block genesisBlock = new Block(genesisBlockHeader, genesisBlockBody); ④
29
30         System.out.println("Create the genesis Block!");
31         System.out.println("And the hash of genesis Block is : " + SecurityUtil.sha256Digest(genesisBlock.toString()) +
32                             ", you will see the hash value in next Block's preBlockHash field.");
33         System.out.println();
34         chain.add(genesisBlock);
35     }

```

图 2-21 Blockchain 类构造方法修改

下一步，实现 `genesisTransactions` 方法，在 `Blockchain` 类中添加相应方法，该方法会创建一批输出 `UTXO`，为每个账户提供一笔金额，只要该账户提供它的身份证明解锁便可使用。至于是谁平白无故地支付这么一笔交易并签名，代码中已经给出了答案，如图 2-22 所示。

```

43  /**
44 * 每个账户分配一定的金额
45 * @param accounts
46 * @return
47 */
48 private Transaction[] genesisTransactions(Account[] accounts) {
49     UTXO[] outUtxos = new UTXO[accounts.length];
50     for (int i = 0; i < accounts.length; ++i) {
51         outUtxos[i] = new UTXO(accounts[i].getWalletAddress(), MiniChainConfig.INIT_AMOUNT, accounts[i].getPublicKey());
52     }
53     KeyPair dayDreamKeyPair = SecurityUtil.secp256k1Generate();
54     PublicKey dayDreamPublicKey = dayDreamKeyPair.getPublic();
55     PrivateKey dayDreamPrivateKey = dayDreamKeyPair.getPrivate();
56     byte[] sign = SecurityUtil.signature("Everything in the dream!".getBytes(StandardCharsets.UTF_8), dayDreamPrivateKey);
57     return new Transaction[]{new Transaction(new UTXO[0], outUtxos, sign, dayDreamPublicKey, System.currentTimeMillis())};
58 }

```

图 2-22 Blockchain 类_方法_genesisTransactions

在 `Blockchain` 类中，我们还需要实现一个方法 `getTrueUtxos`，查询出某个账户可使用的 `UTXO`，定义为 `trueUtxos` 数组，以便构造后续交易使用。在 `Account` 类的 `getAmount` 方法中，正需要 `trueUtxos` 作为参数。而查询某账户的 `trueUtxos` 需要遍历整个区块链每个块中每个交易的 `inUtxos` 和 `outUtxos`，并判断它们的钱包地址是否关联当前账户的地址。实现代码如图 2-23 所示。

```

77  /**
78  * 遍历整个区块链获得某钱包地址相关的utxo，获得真正的utxo，即未被使用的utxo
79  * @param walletAddress 钱包地址
80  * @return
81  */
82  public UTXO[] getTrueUtxos(String walletAddress) {
83      // 使用哈希表存储结果，保证每个utxo唯一
84      Set<UTXO> trueUtxoSet = new HashSet<>();
85      // 遍历每个区块
86      for (Block block : chain) {
87          BlockBody blockBody = block.getBlockBody();
88          Transaction[] transactions = blockBody.getTransactions();
89          // 遍历区块中的所有交易
90          for (Transaction transaction : transactions) {
91              UTXO[] inUtxos = transaction.getInUtxos();
92              UTXO[] outUtxos = transaction.getOutUtxos();
93              // 交易中的inUtxo是已使用的utxo，故需要删除
94              for (UTXO utxo : inUtxos) {
95                  if (utxo.getWalletAddress().equals(walletAddress)) {
96                      trueUtxoSet.remove(utxo);
97                  }
98              }
99              // 交易中的outUtxo是新产生的utxo，可作为后续交易使用
100             for (UTXO utxo : outUtxos) {
101                 if (utxo.getWalletAddress().equals(walletAddress)) {
102                     trueUtxoSet.add(utxo);
103                 }
104             }
105         }
106     }
107     // 转化为数组形式返回
108     UTXO[] trueUtxos = new UTXO[trueUtxoSet.size()];
109     trueUtxoSet.toArray(trueUtxos);
110     return trueUtxos;
111 }

```

图 2-23 Blockchain 类_方法_getTrueUtxos

按下 Alt+Insert 快捷键快速生成 accounts 的 getter 方法，如图 2-24 所示。

```

117     public Account[] getAccounts() {
118         return accounts;
119     }

```

图 2-24 Blockchain 方法_getAccounts

6. 随机生成一笔交易

现在，系统有了一定量的账户，并且能查清他们的余额，他们可以在链上开始交易了。实验 1 中，我们的交易是随机数据，现在要开始真正的比特币交易。TransactionProducer 类负责产生交易，现在它需要与账户和链交互，故需要添加一个 blockchain 数据成员并修改其构造方法，如图 2-25 所示。

```

9  /**
10  * 生成随机交易
11  */
12  public class TransactionProducer extends Thread {
13
14      private final TransactionPool transactionPool;
15      private final Blockchain blockchain;
16
17      public TransactionProducer(TransactionPool transactionPool, Blockchain blockchain) {
18          this.transactionPool = transactionPool;
19          this.blockchain = blockchain;
20      }

```

图 2-25 TransactionProducer 类修改

IDE 提示相关错误，点击上图红字跳转到 NetWork 类，修改相关代码，如图 2-26 所示。

```

13 public class NetWork {
14
15     private final Blockchain blockChain = new Blockchain();
16     private MinerNode minerNode;
17     private TransactionPool transactionPool;
18     private TransactionProducer transactionProducer;
19
20     /**
21      * 系统中几个主要成员的初始化
22      */
23     public NetWork() {
24         transactionPool = new TransactionPool(MiniChainConfig.MAX_TRANSACTION_COUNT);
25         transactionProducer = new TransactionProducer(transactionPool, blockChain);
26         minerNode = new MinerNode(transactionPool, blockChain);
27     }
28

```

图 2-26 NetWork 构造方法修改

在 TransactionProduce 类中，修改 getOneTransaction 方法，获取链上的真实账户地址，生成随机的交易额，构造真实的 UTXO 和交易，并对交易进行签名，代码如图 2-27 所示。

```

42 private Transaction getOneTransaction() {
43
44     Random random = new Random(); // random.nextInt(bound) 在[0, bound) 中取值
45     Transaction transaction = null; // 返回的交易
46     Account[] accounts = blockchain.getAccounts(); // 获取账户数组
47
48     while (true) {
49         // 随机获取两个账户A和B
50         Account aAccount = accounts[random.nextInt(accounts.length)];
51         Account bAccount = accounts[random.nextInt(accounts.length)];
52         // BTC不允许自己给自己转账
53         if (aAccount == bAccount) {
54             continue;
55         }
56
57         // 获得钱包地址
58         String aWalletAddress = aAccount.getWalletAddress();
59         String bWalletAddress = bAccount.getWalletAddress();
60
61         // 获取A可用的Utxo并计算余额
62         UTXO[] aTrueUtxos = blockchain.getTrueUtxos(aWalletAddress);
63         int aAmount = aAccount.getAmount(aTrueUtxos);
64         // 如果A账户的余额为0, 则无法构建交易, 重新随机生成
65         if (aAmount == 0) {
66             continue;
67         }
68
69         // 随机生成交易数额 [1, aAmount] 之间
70         int txAmount = random.nextInt(aAmount) + 1;
71         // 构建InUtxo和OutUtxo
72         List<UTXO> inUtxoList = new ArrayList<>();
73         List<UTXO> outUtxoList = new ArrayList<>();
74
75         // A账户需先解锁才能使用自己的utxo, 解锁需要私钥签名和公钥去执行解锁脚本, 这里先生成需要解锁的签名
76         // 签名的数据我们约定为公钥的二进制数据
77         byte[] aUnlockSign = SecurityUtil.signature(aAccount.getPublicKey().getEncoded(), aAccount.getPrivateKey());
78
79         // 选择输入总额>=交易数额的 utxo
80         int inAmount = 0;
81         for (UTXO utxo : aTrueUtxos) {
82             // 解锁成功才能使用该utxo
83             if (utxo.unlockScript(aUnlockSign, aAccount.getPublicKey())) {
84                 inAmount += utxo.getAmount();
85                 inUtxoList.add(utxo);
86                 if (inAmount >= txAmount) {
87                     break;
88                 }
89             }
90         }
91         // 可解锁的utxo总额仍不足以支付交易数额, 则重新随机
92         if (inAmount < txAmount) {
93             continue;
94         }
95
96         // 构建输出OutUtxos, A账户向B账户支付txAmount, 同时输入对方的公钥以供生成公钥哈希
97         outUtxoList.add(new UTXO(bWalletAddress, txAmount, bAccount.getPublicKey()));
98         // 如果有余额, 则“找零”, 即给自己的utxo
99         if (inAmount > txAmount) {
100             outUtxoList.add(new UTXO(aWalletAddress, amount: inAmount - txAmount, aAccount.getPublicKey()));
101         }
102
103         // 导出固定utxo数组
104         UTXO[] inUtxos = inUtxoList.toArray(new UTXO[0]);
105         UTXO[] outUtxos = outUtxoList.toArray(new UTXO[0]);
106
107         // A账户需对整个交易进行私钥签名, 确保交易不会被篡改, 因为交易会传输到网络中, 而上述步骤可在本地离线环境中构造
108         // 获取要签名的数据, 这个数据需要囊括交易信息
109         byte[] data = SecurityUtil.utxos2Bytes(inUtxos, outUtxos);
110         // A账户使用私钥签名
111         byte[] sign = SecurityUtil.signature(data, aAccount.getPrivateKey());
112         // 交易时间戳
113         long timestamp = System.currentTimeMillis();
114         // 构造交易
115         transaction = new Transaction(inUtxos, outUtxos, sign, aAccount.getPublicKey(), timestamp);
116         // 成功构造一笔交易, 推出循环
117         break;
118     }
119     // 返回随机生成的交易
120     return transaction;
121 }
122
123

```

图 2-27 TransactionProducer 类_方法_getOneTransaction

7. 矿工检查工作

在前面的步骤中，我们实现了交易的签名，所以矿工需使用构造交易的公钥进行验签，确保交易不被篡改，故在 MinerNode 类添加如下代码，如图 2-28 红色方框部分所示。

```
35 public void run() {
36     while (true) {
37         synchronized (transactionPool) {
38             while (!transactionPool.isFull()) {
39                 try {
40                     transactionPool.wait();
41                 } catch (InterruptedException e) {
42                     e.printStackTrace();
43                 }
44             }
45
46             // 从交易池中获取一批次的交易
47             Transaction[] transactions = transactionPool.getAll();
48
49             // 对该交易的签名进行验签，验签失败则退出（比较粗略）
50             if (!check(transactions)) {
51                 System.out.println("transactions error!");
52                 System.exit(-1);
53             }
54
55             // 以交易为参数，调用getBlockBody方法
56             BlockBody blockBody = getBlockBody(transactions);
57
58             // 以blockBody为参数，调用mine方法
59             mine(blockBody);
60
61             transactionPool.notify();
62         }
63     }
64 }
65
66 /**
67  * 矿工检查每笔交易的签名是否正确，是否有被篡改
68  * @param transactions
69  * @return
70  */
71 private boolean check(Transaction[] transactions) {
72     for (int i = 0; i < transactions.length; ++i) {
73         Transaction transaction = transactions[i];
74         // 签名的数据是该交易的 inUtxos 和 outUtxos
75         byte[] data = SecurityUtil.utxos2Bytes(transaction.getInUtxos(), transaction.getOutUtxos());
76         byte[] sign = transaction.getSendSign();
77         PublicKey publicKey = transaction.getSendPublicKey();
78         if (!SecurityUtil.verify(data, sign, publicKey)) {
79             return false;
80         }
81     }
82     return true;
83 }
```

图 2-28 MinerNode 类修改

8. 运行

将 Test 测试代码中因代码修改而出错的地方注释，如图 2-29 所示。

```
11 public class Test {
12
13     private MinerNode minerNode;
14
15     @Before
16     public void setUp() { minerNode = new MinerNode(transactionPool, blockChain); }
17
18     @org.junit.Test
19     public void getBlockBodyTest() {
20         Transaction[] transactions = new Transaction(MiniChainConfig.MAX_TRANSACTION_COUNT);
21         for (int i = 0; i < MiniChainConfig.MAX_TRANSACTION_COUNT; ++i) {
22             // transactions[i] = new Transaction("com.acnu.dase.minichain" + i, 0);
23         }
24         BlockBody blockBody = minerNode.getBlockBody(transactions);
25         Assert.assertTrue("ac81624dc5ef6e92d9e9d722b00ce5bf4c86eb49b5f19787c96fab2f947a".equals(SecurityUtil.sha256Digest(blockBody.toString())));
26     }
27 }
28
29
30 }
```

图 2-29 注释测试方法

运行 MiniChainApplication 类中的 main 方法启动 minichain，读者可在打印信息中看到每个新块的信息（如果很长时间无打印信息可将 MiniChainConfig 里的 DIFFICULTY 参数降低），里面有若干个签名交易，交易中有若干输入 UTXO 和输出 UTXO，如图 2-30 所示。

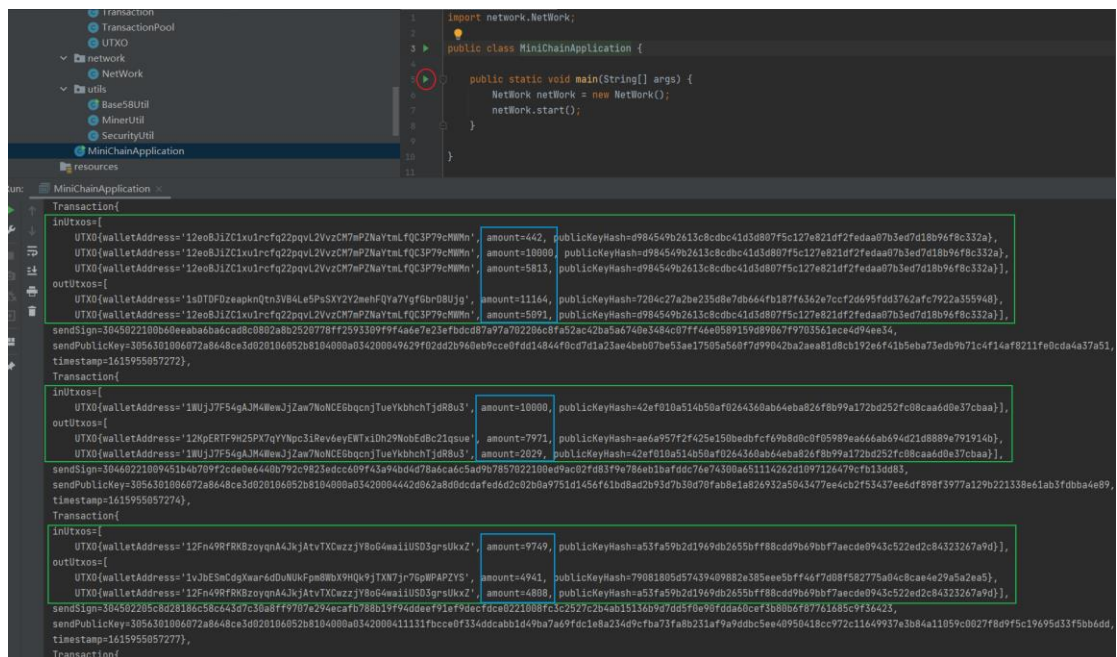


图 2-30 运行结果

9. 验证

为了验证交易中的 UT XO 是否正确，我们可以在系统每出一个块时统计所有账户的总余额，如果所有账户的总额等于初始账户数×初始账户金额（此处是 100000），就能在一定程度上保证程序运行正确。在 Blockchain 类中添加统计下所有账户的总余额的方法，如图 2-31 所示。

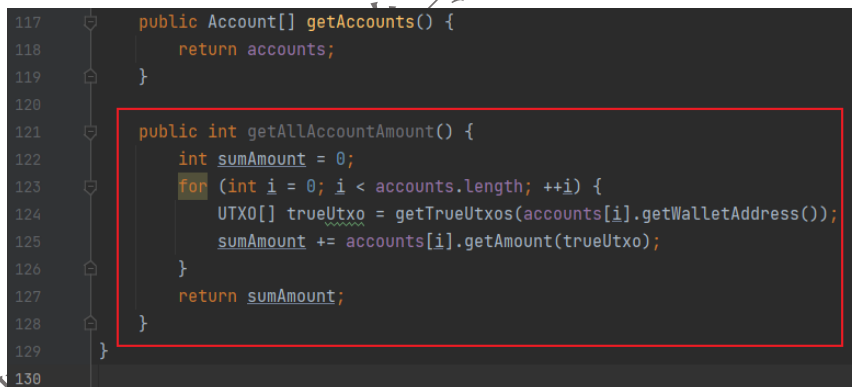


图 2-31 Blockchain 类_方法_getAllAccountAmount

在 MineNode 类中的 run 方法添加输出代码，如图 2-32 所示。

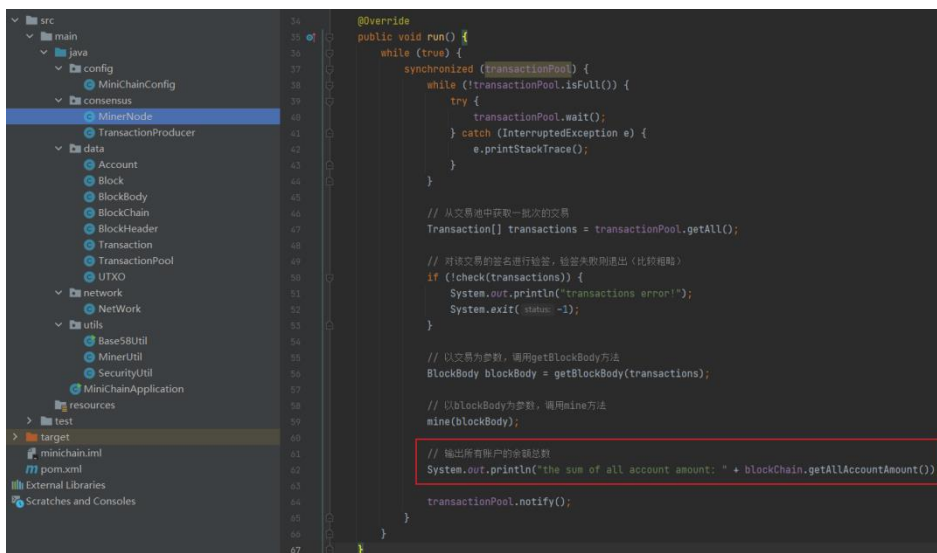


图 2-33 输出所有账户的余额总数

重新运行程序，样例输出如图 2-33 所示。

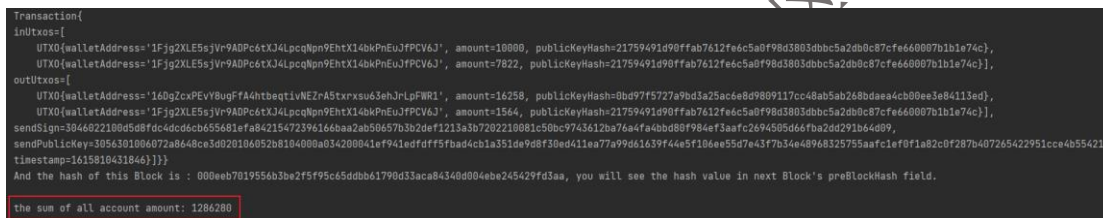


图 2-33 样例输出 1—所有账户的余额总数

读者会发现总金额与我们预期的 1000000 不一样。这因为当前每个块的交易数是 64 个，有些交易使用了重复的输入 UTXO，故会让总金额慢慢变大。在实际的比特币网络中，有更多更严格的检查机制。虽然可以用相关容器做修改，但会带来更多冗余的代码量。所以，我们适当做些妥协，将每个块的交易数限制为 1 个（默克尔树计算不受影响），如图 2-34 所示。

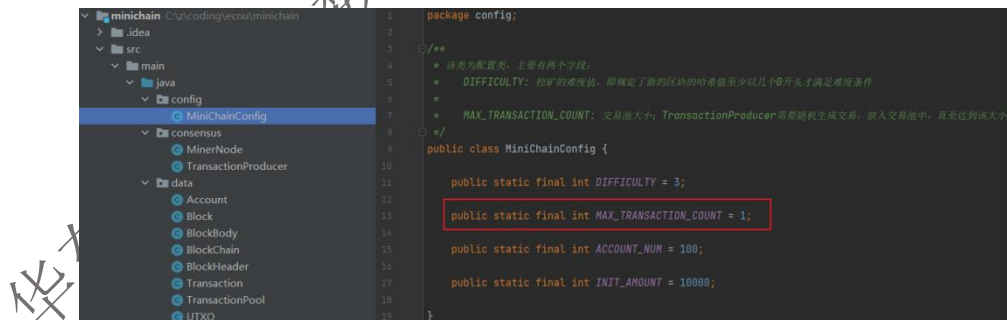


图 2-34 修改块的大小

重新运行程序，如图 2-35 所示，可以看到所有账户的总余额与预期的 1000000 一致，并且一个交易中的输入 UTXO 和输出 UTXO 在金额上时相等的。

```
C:\bin\Java\jdk1.8.0_261\bin\java.exe ...
Create the genesis Block!
And the hash of genesis Block is : 8ff6595d38ff5e95df83d92fe6f42c4efa92b5e16fd3718fc840184521844, you will see the hash value in next Block's preBlockHash field.

Mined a new Block! Detail of the new Block :
Block{BlockHeader=BlockHeader{version=1, preBlockHash='8ff6595d38ff5e95df83d92fe6f42c4efa92b5e16fd3718fc840184521844', merkleRootHash='8596d2987fe945f2fba8083f4548bced0946bf349933eba047db23
Transaction{
  inUtxos=[
    UTXO{walletAddress='105eqkCsisjngftdJAR1M2ZMQKNFGN1P1EJhenNTR8enfydJ0', amount=10000, publicKeyHash=1b6df3bd7d4f322a4038015da239e2b8c2a396b39a401904c8a5d526b2e0fc53}],
    outUtxos=[
      UTXO{walletAddress='12eShhTe6mRrLatBQJb6LtfAeHpfogSooKQNZUa9icRdLkX', amount=5082, publicKeyHash=f5c0e4e424ad1c17693fca8f8b9ea5c03675e8641666e4e76966c16f51d31be},
      UTXO{walletAddress='105eqkCsisjngftdJAR1M2ZMQKNFGN1P1EJhenNTR8enfydJ0', amount=4918, publicKeyHash=1b6df3bd7d4f322a4038015da239e2b8c2a396b39a401904c8a5d526b2e0fc53}],
      sendSigns=3046022100bf9894c7f38ef933ce95e7ee4212a936da7b1c94f332254784196be9c44b1de022180b38d23f1e04ae7065408c7b045d7a0e314599ee4e5519122fde674efee7a90da,
      sendPublicKey=3056301006072a8648ce3d020106052b8184000a034200040b69981400d32ec6acea341a20d7e63b0f102c08fa83d347ad4a0f6d87968d24ca240d51849df789f8a883fd3573d8c3141f6bb3f6158863c32aabec80f8ea,
      timestamp=1615810520024)}}}
And the hash of this Block is : 000d7f22b782e58dbabf67c7aca6d2eb4be6da943cedd94bb08237ae77009c, you will see the hash value in next Block's preBlockHash field.

the sum of all account amount: 100000
Mined a new Block! Detail of the new Block :
Block{BlockHeader=BlockHeader{version=1, preBlockHash='000d7f22b782e58dbabf67c7aca6d2eb4be6da943cedd94bb08237ae77009c', merkleRootHash='fa04bd38fe97497bdcf3b96df986e9ac7bb8b8fc3867252924957
Transaction{
  inUtxos=[
    UTXO{walletAddress='1nnJVnyXt7L5FwgiffsvnR5QuaXAY09QLVhVXBtqf1q12Th0y', amount=10000, publicKeyHash=677e773068d190de864ded8be33b32af2d6808ae568942d13707b3bbf5b69e5}],
    outUtxos=[
      UTXO{walletAddress='1jkgH2zaifKqbyUHAfAPBZK7h1St4y5Qczo5Wqvz2Z8gM8XY', amount=9225, publicKeyHash=284f8e19d6d7b7fe1f2b3d2fc5da892e3b68445cc81027d706f2f496052bb9a},
      UTXO{walletAddress='1nnJVnyXt7L5FwgiffsvnR5QuaXAY09QLVhVXBtqf1q12Th0y', amount=775, publicKeyHash=677e773068d190de864ded8be33b32af2d6808ae568942d13707b3bbf5b69e5}],
      sendSigns=304502200b405151409c4d90763f5be5682470bae187441c9e194049a5a903cbf5f16164022100ad49ac6b52a13b3c644e87ad11fcea9ed5fb18a48ffa0a3bdbe41aff0a775dfe5,
      sendPublicKey=3056301006072a8648ce3d020106052b8184000a034200040b69981400d32ec6acea341a20d7e63b0f102c08fa83d347ad4a0f6d87968d24ca240d51849df789f8a883fd3573d8c3141f6bb3f6158863c32aabec80f8ea,
      timestamp=1615810520089)}}}
And the hash of this Block is : 000f51224367ad18e1ac32327b25c121160b0c03b153d1c602940e52285c4742, you will see the hash value in next Block's preBlockHash field.

the sum of all account amount: 100000
```

图 2-35 样例输出 2—所有账户的余额总数

【实验小结】

本实验从密码学角度切入区块链系统，简要阐述了非对称加密、数字签名和 UTXO 的基本概念。并以比特币为例，描述了一般区块链系统中私钥、公钥以及地址之间的关系，同时给出了非对称加密在数字签名算法中的工作原理。另外，本实验给出了一个 Alice 向 Bob 转账的例子进一步说明比特币如何巧妙地使用 UTXO 交易模型来解决双花问题。

通过本实验的理论学习与工程实践，读者已经能初步理解常见的密码学技术在区块链系统中的工作原理。但由于篇幅限制，密码学在区块链系统中的其他应用如用于隐私保护的群签名、环签名、盲签名和门限签名，以及用于身份认证的数字证书等，均没有在本实验中进行介绍。感兴趣的读者可以阅读其他相关资料，在本实验提供的简单区块链系统中进行相应代码的实现。

【习题】

1. 在非对称加密算法和数字签名的过程中，私钥和公钥的作用是否相同？若不同，请描述不同之处。
2. UTXO 模型和账户模型的各自优缺点是什么？
3. 在实验完成的基础上，在 test/java/unit 包中新建一个测试类 UtxoTest，构造一个确定的 UTXO 交易：账户 A(accounts[1])转账 1000BTC 给账户 B(accounts[2])。

【参考文献】

- [1] 陈少真. 密码学基础[M]. 科学出版社, 2008.
- [2] 王厚涛.SSL VPN 安全技术研究及改进[D].北京邮电大学,2011.
- [3] 何昊.区块链架构之美[M].电子工业出版社,2021.
- [4] 汪朝晖, 张振峰. SM2 椭圆曲线公钥密码算法综述[J]. 信息安全研究, 2016.
- [5] 秦波, 陈李昌豪, 伍前红, 等. 比特币与法定数字货币[J]. 密码学报, 2017, 4(2): 176-186.