# 【译】用Java创建你的第一个区块链-part1

区块链是分布式数据存储、点对点传输、共识机制、加密算法等计算机技术的新型应用模式。所谓共识机制是区块链系统中实现不同节点之间建立信任、获取权益的数学算法 。

## 前言

本系列教程旨在帮助你了解如何开发区块链技术。

**本章目标**

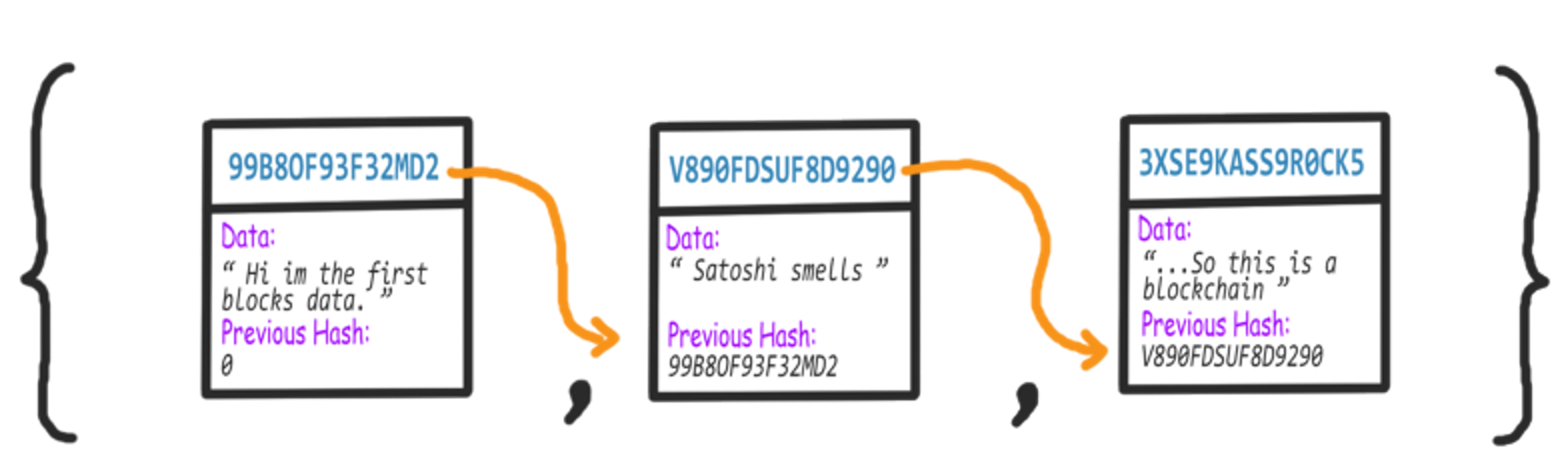
* 创建你第一个非常基本的区块链
* 实现一个简单的工作量证明系统即挖矿
* 在此基础上进行扩展

（我会假设你对面向对象编程有基本的了解）

值得注意的是，这里创建的区块链并不是功能完全的完全适合应用与生产的区块链，相反只是为了帮助你更好的理解区块链的概念。

### 创建区块链

区块链就是一串或者是一系列区块的集合，类似于链表的概念，每个区块都指向于后面一个区块，然后顺序的连接在一起。那么每个区块中的内容是什么呢？在区块链中的每一个区块都存放了很多很有价值的信息，主要包括三个部分：自己的数字签名，上一个区块的数字签名，还有一切需要加密的数据（这些数据在比特币中就相当于是交易的信息，它是加密货币的本质）。每个数字签名不但证明了自己是特有的一个区块，而且指向了前一个区块的来源，让所有的区块在链条中可以串起来，而数据就是一些特定的信息，你可以按照业务逻辑来保存业务数据。



**这里的hash指的就是数字签名**

**所以每一个区块不仅包含前一个区块的hash值，同时包含自身的一个hash值**，自身的hash值是通过之前的hash值和数据data通过hash计算出来的。如果前一个区块的数据一旦被篡改了，那么前一个区块的hash值也会同样发生变化（因为数据也被计算在内），这样也就导致了所有后续的区块中的hash值。**所以计算和比对hash值会让我们检查到当前的区块链是否是有效的**，也就避免了数据被恶意篡改的可能性，因为篡改数据就会改变hash值并破坏整个区块链。

#### 定义区块链的类快

**import** java.util.Date**;**

**public** **class** **Block** **{**

**public** String hash**;**

**public** String previousHash**;**

**private** String data**;** //our data will be a simple message.

**private** **long** timeStamp**;** //as number of milliseconds since 1/1/1970.

//Block Constructor.

**public** **Block(**String data**,**String previousHash **)** **{**

**this.**data **=** data**;**

**this.**previousHash **=** previousHash**;**

**this.**timeStamp **=** **new** Date**().**getTime**();**

**}**

**}**

正如你可以看到我们的基本块包含String hash，它将保存我们的数字签名。变量previoushash保存前一个块的hash和String data来保存我们的块数据

#### 创建数字签名

熟悉加密算法的朋友们，Java方式可以实现的加密方式有很多，例如BASE、MD、RSA、SHA等等，我在这里选用了SHA256这种加密方式，SHA（Secure Hash Algorithm）安全散列算法，这种算法的特点是数据的少量更改会在Hash值中产生不可预知的大量更改，hash值用作表示大量数据的固定大小的唯一值，而SHA256算法的hash值大小为256位。之所以选用SHA256是因为它的大小正合适，一方面产生重复hash值的可能性很小，另一方面在区块链实际应用过程中，有可能会产生大量的区块，而使得信息量很大，那么256位的大小就比较恰当了。

下面我创建了一个StringUtil方法来方便调用SHA256算法

**import** java.security.MessageDigest**;**

**public** **class** **StringUtil** **{**

//Applies Sha256 to a string and returns the result.

**public** **static** String **applySha256(**String input**){**

**try** **{**

MessageDigest digest **=** MessageDigest**.**getInstance**(**"SHA-256"**);**

//Applies sha256 to our input,

**byte[]** hash **=** digest**.**digest**(**input**.**getBytes**(**"UTF-8"**));**

// This will contain hash as hexidecimal

StringBuffer hexString **=** **new** StringBuffer**();**

**for** **(int** i **=** 0**;** i **<** hash**.**length**;** i**++)** **{**

String hex **=** Integer**.**toHexString**(**0xff **&** hash**[**i**]);**

**if(**hex**.**length**()** **==** 1**)** hexString**.**append**(**'0'**);**

hexString**.**append**(**hex**);**

**}**

**return** hexString**.**toString**();**

**}**

**catch(**Exception e**)** **{**

**throw** **new** **RuntimeException(**e**);**

**}**

**}**

**}**

或许你完全不理解上述代码的含义，但是你只要理解所有的输入调用此方法后均会生成一个独一无二的hash值（数字签名），而这个hash值在区块链中是非常重要的。

接下来让我们在Block类中应用方法 applySha256 方法，其主要的目的就是计算hash值，我们计算的hash值应该包括区块中所有我们不希望被恶意篡改的数据，在我们上面所列的Block类中就一定包括previousHash，data和timeStamp，

**public** String **calculateHash()** **{**

String calculatedhash **=** StringUtil**.**applySha256**(**

previousHash **+**

Long**.**toString**(**timeStamp**)** **+**

data

**);**

**return** calculatedhash**;**

**}**

然后把这个方法加入到Block的构造函数中去

**public** **Block(**String data**,**String previousHash **)** **{**

**this.**data **=** data**;**

**this.**previousHash **=** previousHash**;**

**this.**timeStamp **=** **new** Date**().**getTime**();**

//Making sure we do this after we set the other values.

**this.**hash **=** calculateHash**();**

**}**

#### 测试

在主方法中让我们创建一些区块，并把其hash值打印出来，来看看是否一切都在我们的掌控中。

第一个块称为创世纪区块，因为它是头区块，所以我们只需输入“0”作为前一个块的previous hash。

**public** **class** **NoobChain** **{**

**public** **static** **void** **main(**String**[]** args**)** **{**

Block genesisBlock **=** **new** Block**(**"Hi im the first block"**,** "0"**);**

System**.**out**.**println**(**"Hash for block 1 : " **+** genesisBlock**.**hash**);**

Block secondBlock **=** **new** Block**(**"Yo im the second block"**,**genesisBlock**.**hash**);**

System**.**out**.**println**(**"Hash for block 2 : " **+** secondBlock**.**hash**);**

Block thirdBlock **=** **new** Block**(**"Hey im the third block"**,**secondBlock**.**hash**);**

System**.**out**.**println**(**"Hash for block 3 : " **+** thirdBlock**.**hash**);**

**}**

**}**

打印：

Hash **for** block 1**:** f6d1bc5f7b0016eab53ec022db9a5d9e1873ee78513b1c666696e66777fe55fb

Hash **for** block 2**:** 6936612b3380660840f22ee6cb8b72ffc01dbca5369f305b92018321d883f4a3

Hash **for** block 3**:** f3e58f74b5adbd59a7a1fc68c97055d42e94d33f6c322d87b29ab20d3c959b8f

每一个区块都必须要有自己的数据签名即hash值，这个hash值依赖于自身的信息（data）和上一个区块的数字签名（previousHash），但这个还不是区块链，下面让我们存储区块到数组中，这里我会引入gson包，目的是可以用json方式查看整个一条区块链结构。

**import** java.util.ArrayList**;**

**import** com.google.gson.GsonBuilder**;**

**public** **class** **NoobChain** **{**

**public** **static** ArrayList**<**Block**>** blockchain **=** **new** ArrayList**<**Block**>();**

**public** **static** **void** **main(**String**[]** args**)** **{**

//add our blocks to the blockchain ArrayList:

blockchain**.**add**(new** Block**(**"Hi im the first block"**,** "0"**));**

blockchain**.**add**(new** Block**(**"Yo im the second block"**,**blockchain**.**get**(**blockchain**.**size**()-**1**).**hash**));**

blockchain**.**add**(new** Block**(**"Hey im the third block"**,**blockchain**.**get**(**blockchain**.**size**()-**1**).**hash**));**

String blockchainJson **=** **new** GsonBuilder**().**setPrettyPrinting**().**create**().**toJson**(**blockchain**);**

System**.**out**.**println**(**blockchainJson**);**

**}**

**}**

这样的输出结构就更类似于我们所期待的区块链的样子。

#### 检查区块链的完整性

在主方法中增加一个isChainValid()方法，目的是循环区块链中的所有区块并且比较hash值，这个方法用来检查hash值是否是于计算出来的hash值相等，同时previousHash值是否和前一个区块的hash值相等。或许你会产生如下的疑问，我们就在一个主函数中创建区块链中的区块，所以不存在被修改的可能性，但是你要注意的是，区块链中的一个核心概念就是去中心化，每一个区块可能是在网络中的某一个节点中产生的，所以很有可能某个节点把自己节点中的数据修改了，那么根据上述的理论数据改变会导致整个区块链的破裂，也就是区块链就无效了。

**public** **static** Boolean **isChainValid()** **{**

Block currentBlock**;**

Block previousBlock**;**

//loop through blockchain to check hashes:

**for(int** i**=**1**;** i **<** blockchain**.**size**();** i**++)** **{**

currentBlock **=** blockchain**.**get**(**i**);**

previousBlock **=** blockchain**.**get**(**i**-**1**);**

//compare registered hash and calculated hash:

**if(!**currentBlock**.**hash**.**equals**(**currentBlock**.**calculateHash**())** **){**

System**.**out**.**println**(**"Current Hashes not equal"**);**

**return** **false;**

**}**

//compare previous hash and registered previous hash

**if(!**previousBlock**.**hash**.**equals**(**currentBlock**.**previousHash**)** **)** **{**

System**.**out**.**println**(**"Previous Hashes not equal"**);**

**return** **false;**

**}**

**}**

**return** **true;**

**}**

任何区块链中区块的一丝一毫改变都会导致这个函数返回false，也就证明了区块链无效了。

在比特币网络中所有的网络节点都分享了它们各自的区块链，然而最长的有效区块链是被全网所统一承认的，如果有人恶意来篡改之前的数据，然后创建一条更长的区块链并全网发布呈现在网络中，我们该怎么办呢？这就涉及到了区块链中另外一个重要的概念工作量证明，这里就不得不提及一下hashcash，这个概念最早来自于Adam Back的一篇论文，主要应用于邮件过滤和比特币中防止双重支付。

### 挖矿

这里我们要求挖矿者做工作量证明，具体的方式是在区块中尝试不同的参数值直到它的hash值是从一系列的0开始的。让我们添加一个名为nonce的int类型以包含在我们的calculatehash（）方法中，以及需要的mineblock（）方法

**import** java.util.Date**;**

**public** **class** **Block** **{**

**public** String hash**;**

**public** String previousHash**;**

**private** String data**;** //our data will be a simple message.

**private** **long** timeStamp**;** //as number of milliseconds since 1/1/1970.

**private** **int** nonce**;**

//Block Constructor.

**public** **Block(**String data**,**String previousHash **)** **{**

**this.**data **=** data**;**

**this.**previousHash **=** previousHash**;**

**this.**timeStamp **=** **new** Date**().**getTime**();**

//Making sure we do this after we set the other values.

**this.**hash **=** calculateHash**();**

**}**

//Calculate new hash based on blocks contents

**public** String **calculateHash()** **{**

String calculatedhash **=** StringUtil**.**applySha256**(**

previousHash **+**

Long**.**toString**(**timeStamp**)** **+**

Integer**.**toString**(**nonce**)** **+**

data

**);**

**return** calculatedhash**;**

**}**

**public** **void** **mineBlock(int** difficulty**)** **{**

//Create a string with difficulty \* "0"

String target **=** **new** String**(new** **char[**difficulty**]).**replace**(**'\0'**,** '0'**);**

**while(!**hash**.**substring**(** 0**,** difficulty**).**equals**(**target**))** **{**

nonce **++;**

hash **=** calculateHash**();**

**}**

System**.**out**.**println**(**"Block Mined!!! : " **+** hash**);**

**}**

**}**

mineBlock()方法中引入了一个int值称为difficulty难度，低的难度比如1和2，普通的电脑基本都可以马上计算出来，我的建议是在4-6之间进行测试，普通电脑大概会花费3秒时间，在莱特币中难度大概围绕在442592左右，而在比特币中每一次挖矿都要求大概在10分钟左右，当然根据所有网络中的计算能力，难度也会不断的进行修改。

我们在NoobChain类 中增加difficulty这个静态变量。

**public** **static** **int** difficulty **=** 5**;**

这样我们必须修改主方法中让创建每个新区块时必须触发mineBlock()方法，而isChainValid()方法用来检查每个区块的hash值是否正确，整个区块链是否是有效的。

**import** java.util.ArrayList**;**

**import** com.google.gson.GsonBuilder**;**

**public** **class** **NoobChain** **{**

**public** **static** ArrayList**<**Block**>** blockchain **=** **new** ArrayList**<**Block**>();**

**public** **static** **int** difficulty **=** 5**;**

**public** **static** **void** **main(**String**[]** args**)** **{**

//add our blocks to the blockchain ArrayList:

blockchain**.**add**(new** Block**(**"Hi im the first block"**,** "0"**));**

System**.**out**.**println**(**"Trying to Mine block 1... "**);**

blockchain**.**get**(**0**).**mineBlock**(**difficulty**);**

blockchain**.**add**(new** Block**(**"Yo im the second block"**,**blockchain**.**get**(**blockchain**.**size**()-**1**).**hash**));**

System**.**out**.**println**(**"Trying to Mine block 2... "**);**

blockchain**.**get**(**1**).**mineBlock**(**difficulty**);**

blockchain**.**add**(new** Block**(**"Hey im the third block"**,**blockchain**.**get**(**blockchain**.**size**()-**1**).**hash**));**

System**.**out**.**println**(**"Trying to Mine block 3... "**);**

blockchain**.**get**(**2**).**mineBlock**(**difficulty**);**

System**.**out**.**println**(**"\nBlockchain is Valid: " **+** isChainValid**());**

String blockchainJson **=** **new** GsonBuilder**().**setPrettyPrinting**().**create**().**toJson**(**blockchain**);**

System**.**out**.**println**(**"\nThe block chain: "**);**

System**.**out**.**println**(**blockchainJson**);**

**}**

**public** **static** Boolean **isChainValid()** **{**

Block currentBlock**;**

Block previousBlock**;**

String hashTarget **=** **new** String**(new** **char[**difficulty**]).**replace**(**'\0'**,** '0'**);**

//loop through blockchain to check hashes:

**for(int** i**=**1**;** i **<** blockchain**.**size**();** i**++)** **{**

currentBlock **=** blockchain**.**get**(**i**);**

previousBlock **=** blockchain**.**get**(**i**-**1**);**

//compare registered hash and calculated hash:

**if(!**currentBlock**.**hash**.**equals**(**currentBlock**.**calculateHash**())** **){**

System**.**out**.**println**(**"Current Hashes not equal"**);**

**return** **false;**

**}**

//compare previous hash and registered previous hash

**if(!**previousBlock**.**hash**.**equals**(**currentBlock**.**previousHash**)** **)** **{**

System**.**out**.**println**(**"Previous Hashes not equal"**);**

**return** **false;**

**}**

//check if hash is solved

**if(!**currentBlock**.**hash**.**substring**(** 0**,** difficulty**).**equals**(**hashTarget**))** **{**

System**.**out**.**println**(**"This block hasn't been mined"**);**

**return** **false;**

**}**

**}**

**return** **true;**

**}**

**}**

打印：

Connected to the target VM**,** address: '127.0**.**0.1**:**61863'**,** transport: 'socket'

Trying to Mine block 1**...**

Block Mined**!!!** **:** 0000016667d4240e9c30f53015310b0ec6ce99032d7e1d66d670afc509cab082

Trying to Mine block 2**...**

Block Mined**!!!** **:** 000002ea55735bea4cac7e358c7b0d8d81e8ca24021f5f85211bf54fd4ac795a

Trying to Mine block 3**...**

Block Mined**!!!** **:** 000000576987e5e9afbdf19b512b2b7d0c56db0e6ca49b3a7e638177f617994b

Blockchain is Valid: **true**

**[**

**{**

"hash"**:** "0000016667d4240e9c30f53015310b0ec6ce99032d7e1d66d670afc509cab082"**,**

"previousHash"**:** "0"**,**

"data"**:** "first"**,**

"timeStamp"**:** 1520659506042**,**

"nonce"**:** 618139

**},**

**{**

"hash"**:** "000002ea55735bea4cac7e358c7b0d8d81e8ca24021f5f85211bf54fd4ac795a"**,**

"previousHash"**:** "0000016667d4240e9c30f53015310b0ec6ce99032d7e1d66d670afc509cab082"**,**

"data"**:** "second"**,**

"timeStamp"**:** 1520659508825**,**

"nonce"**:** 1819877

**},**

**{**

"hash"**:** "000000576987e5e9afbdf19b512b2b7d0c56db0e6ca49b3a7e638177f617994b"**,**

"previousHash"**:** "000002ea55735bea4cac7e358c7b0d8d81e8ca24021f5f85211bf54fd4ac795a"**,**

"data"**:** "third"**,**

"timeStamp"**:** 1520659515910**,**

"nonce"**:** 1404341

**}**

**]**

经过测试增加一个新的区块即挖矿必须花费一定时间，大概是3秒左右，你可以提高difficulty难度来看，它是如何影响数据难题所花费的时间的

如果有人在你的区块链系统中恶意篡改数据：

1. 他们的区块链是无效的。
2. 他们无法创建更长的区块链
3. 网络中诚实的区块链会在长链中更有时间的优势

因为篡改的区块链将无法赶上长链和有效链，除非他们比你网络中所有的节点拥有更大的计算速度，可能是未来的量子计算机或者是其他什么。

## 你已经完成了你的基本区块链！

你的区块链：

* 有很多区块组成用来存储数据
* 有数字签名让你的区块链链接在一起
* 需要挖矿的工作量证明新的区块
* 可以用来检查数据是否是有效的同时是未经篡改的

# 【译】用Java创建你的第一个区块链-part2:可交易

区块链是分布式数据存储、点对点传输、共识机制、加密算法等计算机技术的新型应用模式。所谓共识机制是区块链系统中实现不同节点之间建立信任、获取权益的数学算法 。

## 前言

本系列教程旨在帮助你了解如何开发区块链技术。[【译】用Java创建你的第一个区块链-part2](https://longfeizheng.github.io/2018/03/10/%E7%94%A8Java%E5%88%9B%E5%BB%BA%E4%BD%A0%E7%9A%84%E7%AC%AC%E4%B8%80%E4%B8%AA%E5%8C%BA%E5%9D%97%E9%93%BE-part1/)

**本章目标**

* 创建一个简单的钱包。
* 使用我们的区块链发送带签名的交易。
* 感觉很吊

**这样我们就有自己的加密货币**

值得注意的是，这里创建的区块链并不是功能完全的完全适合应用与生产的区块链，相反只是为了帮助你更好的理解区块链的概念。

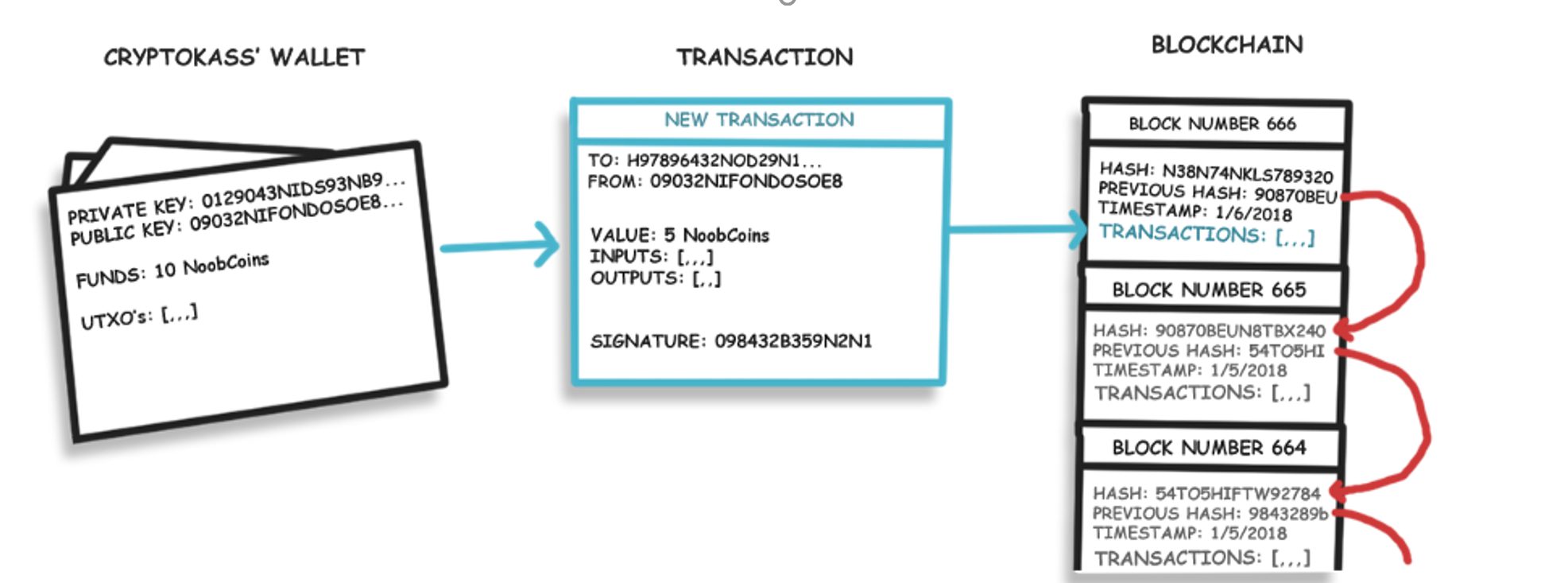
别担心，这实际上是很简单的，但比上一个教程要长!

在上衣章节[【译】用Java创建你的第一个区块链-part2](https://longfeizheng.github.io/2018/03/10/%E7%94%A8Java%E5%88%9B%E5%BB%BA%E4%BD%A0%E7%9A%84%E7%AC%AC%E4%B8%80%E4%B8%AA%E5%8C%BA%E5%9D%97%E9%93%BE-part1/)，我们已经有了一个基本的区块链，但在区块链中存放的是一些无用的信息。今天我们将用交易取代这些信息（我们的区块将能够保存多个交易），允许我们创建一个非常简单的加密货币,我们的货币名字NoobCoin。

* 本教程是在上一边基础上实现的
* 导入[bounceycastle](https://www.bouncycastle.org/latest_releases.html)和GSON

### 准备一个钱包

在加密货币中，在区块链作为交易时，货币所有权可以进行转移，每个参与者都有一个自己私有的地址来发送或者是收取货币。，钱包可以存储这些地址。因此钱包就是可以在区块链上进行新交易的软件。



Don’t worry about the information on the transaction, that will be explained soon

让我们创建一个钱包类来保存公钥和私钥:

**package** noobchain**;**

**import** java.security.\***;**

**public** **class** **Wallet** **{**

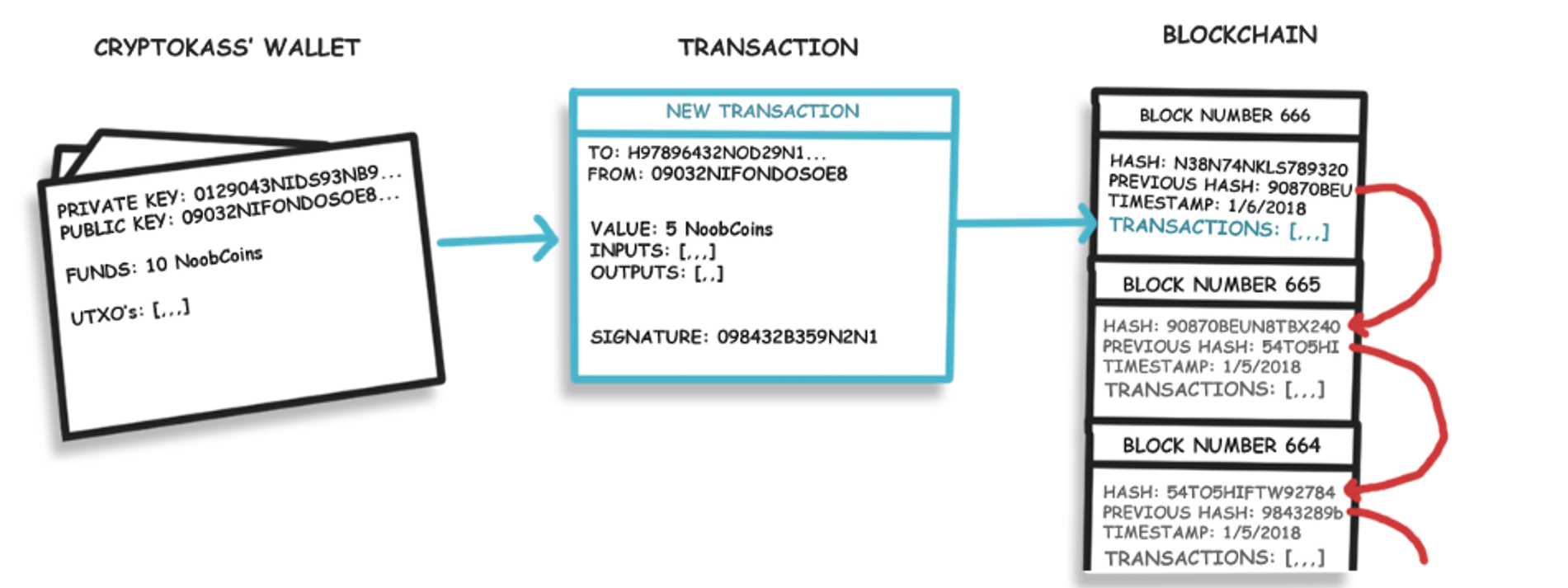
**public** PrivateKey privateKey**;**

**public** PublicKey publicKey**;**

**}**

公钥和私钥究竟是起到什么作用呢，其实公钥的作用就是地址，你可以分享你的公钥给别人以此来获取付款，而你的私钥的作用是为了对交易进行签名，这样其他人就不可以花费你的金额除非它拥有你的私钥，所以对于每个人而言我们必须保护好我们的私钥，不能透露我们的私钥信息给其他人。同时在我们进行交易的时候我们也会同时发送我们的公钥由此来验证我们的签名是有效的而且没有数据被篡改。

我们在密钥对KeyPair生成私有和公钥。我们将使用[椭圆曲线加密](https://en.wikipedia.org/wiki/Elliptic-curve_cryptography)来生成我们的密钥对KeyPair。让我们将generateKeyPair()方法添加到我们的钱包类中，并在构造函数中调用它:



私钥用于签署我们不想被篡改的数据。公钥用于验证签名。

**package** noobchain**;**

**import** java.security.\***;**

**public** **class** **Wallet** **{**

**public** PrivateKey privateKey**;**

**public** PublicKey publicKey**;**

**public** **Wallet(){**

generateKeyPair**();**

**}**

**public** **void** **generateKeyPair()** **{**

**try** **{**

KeyPairGenerator keyGen **=** KeyPairGenerator**.**getInstance**(**"ECDSA"**,**"BC"**);**

SecureRandom random **=** SecureRandom**.**getInstance**(**"SHA1PRNG"**);**

ECGenParameterSpec ecSpec **=** **new** ECGenParameterSpec**(**"prime192v1"**);**

// Initialize the key generator and generate a KeyPair

keyGen**.**initialize**(**ecSpec**,** random**);** //256 bytes provides an acceptable security level

KeyPair keyPair **=** keyGen**.**generateKeyPair**();**

// Set the public and private keys from the keyPair

privateKey **=** keyPair**.**getPrivate**();**

publicKey **=** keyPair**.**getPublic**();**

**}catch(**Exception e**)** **{**

**throw** **new** **RuntimeException(**e**);**

**}**

**}**

**}**

你不需要完全理解椭圆曲线加密算法的核心逻辑究竟是什么，你只需要它是用来创建公钥和私钥，以及公钥和私钥分别所起到的作用是什么就可以了。

现在我们已经又了钱包类的大概框架，下面我们再来看一下交易类。

### 交易和数字签名

每笔交易将携带一定以下信息：

1. 资金付款人的公匙信息。
2. 资金收款人的公匙信息。
3. 被转移资金的金额。
4. 输入，它是对以前的交易的引用，证明发送者有资金发送。
5. 输出，显示交易中收款方相关地址数量。(这些输出被引用为新交易的输入)
6. 一个加密签名，证明该交易是由地址的发送者是发送的，并且数据没有被更改。(阻止第三方机构更改发送的数量)

让我们创建这个新的交易类：

**import** java.security.\***;**

**import** java.util.ArrayList**;**

**public** **class** **Transaction** **{**

**public** String transactionId**;** // this is also the hash of the transaction.

**public** PublicKey sender**;** // senders address/public key.

**public** PublicKey reciepient**;** // Recipients address/public key.

**public** **float** value**;**

**public** **byte[]** signature**;** // this is to prevent anybody else from spending funds in our wallet.

**public** ArrayList**<**TransactionInput**>** inputs **=** **new** ArrayList**<**TransactionInput**>();**

**public** ArrayList**<**TransactionOutput**>** outputs **=** **new** ArrayList**<**TransactionOutput**>();**

**private** **static** **int** sequence **=** 0**;** // a rough count of how many transactions have been generated.

// Constructor:

**public** **Transaction(**PublicKey from**,** PublicKey to**,** **float** value**,** ArrayList**<**TransactionInput**>** inputs**)** **{**

**this.**sender **=** from**;**

**this.**reciepient **=** to**;**

**this.**value **=** value**;**

**this.**inputs **=** inputs**;**

**}**

// This Calculates the transaction hash (which will be used as its Id)

**private** String **calulateHash()** **{**

sequence**++;** //increase the sequence to avoid 2 identical transactions having the same hash

**return** StringUtil**.**applySha256**(**

StringUtil**.**getStringFromKey**(**sender**)** **+**

StringUtil**.**getStringFromKey**(**reciepient**)** **+**

Float**.**toString**(**value**)** **+** sequence

**);**

**}**

**}**

我们还应该创建空的transactioninput和transactionoutput类，不要担心我们可以在后面填写它们。

我们的交易类还将包含生成/验证签名和验证交易的相关方法，那么签名的意图是什么？签名在我们的区块链中执行了两个非常重要的任务：第一，签名用来保证只有货币的拥有者才可以用来发送自己的货币，第二，签名用来阻止其他人试图篡改提交的交易。

**即私钥被用来签名数据，而公钥用来验证其完整性。**

举个例子：Bob 想要发送2个加密货币给Sally，他们用各自的钱包创建了交易，并提交到全网的区块链中作为一个新的区块，一个挖矿者试图篡改接受者把2个加密货币给John，但是幸运的事，Bob在交易数据中已经用私钥进行了签名，这就允许了全网中的任何节点使用小明的公匙进行验证数据是否已经被篡改（因为没有其他人的公钥可以用来验证小明发出的这笔交易）。

从前面的代码中我们可以看到我们的签名将是一堆字符串，所以让我们创建一个方法来生成它们。首先我们在StringUtil类中创建产生签名的方法。

**public** **static** **byte[]** **applyECDSASig(**PrivateKey privateKey**,** String input**)** **{**

Signature dsa**;**

**byte[]** output **=** **new** **byte[**0**];**

**try** **{**

dsa **=** Signature**.**getInstance**(**"ECDSA"**,** "BC"**);**

dsa**.**initSign**(**privateKey**);**

**byte[]** strByte **=** input**.**getBytes**();**

dsa**.**update**(**strByte**);**

**byte[]** realSig **=** dsa**.**sign**();**

output **=** realSig**;**

**}** **catch** **(**Exception e**)** **{**

**throw** **new** **RuntimeException(**e**);**

**}**

**return** output**;**

**}**

//Verifies a String signature

**public** **static** **boolean** **verifyECDSASig(**PublicKey publicKey**,** String data**,** **byte[]** signature**)** **{**

**try** **{**

Signature ecdsaVerify **=** Signature**.**getInstance**(**"ECDSA"**,** "BC"**);**

ecdsaVerify**.**initVerify**(**publicKey**);**

ecdsaVerify**.**update**(**data**.**getBytes**());**

**return** ecdsaVerify**.**verify**(**signature**);**

**}catch(**Exception e**)** **{**

**throw** **new** **RuntimeException(**e**);**

**}**

**}**

**public** **static** String **getStringFromKey(**Key key**)** **{**

**return** Base64**.**getEncoder**().**encodeToString**(**key**.**getEncoded**());**

**}**

不用担心你不能理解这些方法的内容，你所需要知道的就是applyECDSASig方法的输入参数为付款人的私钥和需要加密的数据信息，签名后返回字节数组。而verifyECDSASig方法的输入参数为公钥、加密的数据和签名，调用该方法后返回true或false来说明签名是否是有效的。getStringFromKey返回任何key的编码字符串

让我们使用签名的方法在交易类中，增加generatesiganature() 和 varifiysignature()方法：

//Signs all the data we dont wish to be tampered with.

**public** **void** **generateSignature(**PrivateKey privateKey**)** **{**

String data **=** StringUtil**.**getStringFromKey**(**sender**)** **+** StringUtil**.**getStringFromKey**(**reciepient**)** **+** Float**.**toString**(**value**)** **;**

signature **=** StringUtil**.**applyECDSASig**(**privateKey**,**data**);**

**}**

//Verifies the data we signed hasnt been tampered with

**public** **boolean** **verifiySignature()** **{**

String data **=** StringUtil**.**getStringFromKey**(**sender**)** **+** StringUtil**.**getStringFromKey**(**reciepient**)** **+** Float**.**toString**(**value**)** **;**

**return** StringUtil**.**verifyECDSASig**(**sender**,** data**,** signature**);**

**}**

在现实中，您可能希望签名更多信息，例如使用的输出/输入和/或时间戳（现在我们只是签名最低限度的信息）

签名将由矿工验证，只有签名验证成功后交易才能被添加到区块中去。

当我们检查区块链的有效性时，我们也可以检查签名

### 测试钱包和签名

现在我们简单的进行一些测试，在主方法中，我们增加了一些新的变量也替换了我们之前在主方法中的一些内容。

**import** java.security.Security**;**

**import** java.util.ArrayList**;**

**import** java.util.Base64**;**

**import** com.google.gson.GsonBuilder**;**

**public** **class** **NoobChain** **{**

**public** **static** ArrayList**<**Block**>** blockchain **=** **new** ArrayList**<**Block**>();**

**public** **static** **int** difficulty **=** 5**;**

**public** **static** Wallet walletA**;**

**public** **static** Wallet walletB**;**

**public** **static** **void** **main(**String**[]** args**)** **{**

//Setup Bouncey castle as a Security Provider

Security**.**addProvider**(new** org**.**bouncycastle**.**jce**.**provider**.**BouncyCastleProvider**());**

//Create the new wallets

walletA **=** **new** Wallet**();**

walletB **=** **new** Wallet**();**

//Test public and private keys

System**.**out**.**println**(**"Private and public keys:"**);**

System**.**out**.**println**(**StringUtil**.**getStringFromKey**(**walletA**.**privateKey**));**

System**.**out**.**println**(**StringUtil**.**getStringFromKey**(**walletA**.**publicKey**));**

//Create a test transaction from WalletA to walletB

Transaction transaction **=** **new** Transaction**(**walletA**.**publicKey**,** walletB**.**publicKey**,** 5**,** **null);**

transaction**.**generateSignature**(**walletA**.**privateKey**);**

//Verify the signature works and verify it from the public key

System**.**out**.**println**(**"Is signature verified"**);**

System**.**out**.**println**(**transaction**.**verifiySignature**());**

**}**

确定要添加bounceycastle依赖

我们创建了两个钱包walleta和walletb，然后打印了walleta的私钥和公钥。生成一个交易并使用walleta的私钥对其进行签名。

打印

Connected to the target VM**,** address: '127.0**.**0.1**:**55757'**,** transport: 'socket'

Private and **public** keys:

MHsCAQAwEwYHKoZIzj0CAQYIKoZIzj0DAQEEYTBfAgEBBBiJzZiesBnBWwwB**+**uog**+**fJX84mx4lPUTUagCgYIKoZIzj0DAQGhNAMyAAQfPzad0zqBUGQAany2rRZE1**+**2ml5IOCZST8LywjBQT8ux**+**3UPVbr2u0**+**LaExxz1WI**=**

MEkwEwYHKoZIzj0CAQYIKoZIzj0DAQEDMgAEHz82ndM6gVBkAGp8tq0WRNftppeSDgmUk**/**C8sIwUE**/**Lsft1D1W69rtPi2hMcc9Vi

Is signature verified

**true**

接下来我们将创建并验证输入和输出，并把交易保存到区块链中去。

### 输入和输出 1：如何验证货币是你的

如果你拥有1比特币，你必须前面就得接收1比特币。比特币的账本不会在你的账户中增加一个比特币也不会从发送者那里减去一个比特币，发送者只能指向他/她之前收到过一个比特币，所以一个交易输出被创建用来显示一个比特币发送给你的地址（交易的输入指向前一个交易的输出）。

**你的钱包余额是所有未使用的交易输出的总和**

从这一个点出发，我们会依照比特币中的说明，把所有未使用的交易输出称为[UTXO](http://8btc.com/article-4381-1.html).

创建TransactionInput 类：

**public** **class** **TransactionInput** **{**

**public** String transactionOutputId**;** //Reference to TransactionOutputs -> transactionId

**public** TransactionOutput UTXO**;** //Contains the Unspent transaction output

**public** **TransactionInput(**String transactionOutputId**)** **{**

**this.**transactionOutputId **=** transactionOutputId**;**

**}**

**}**

这个类将用于引用尚未使用的transactionoutput。transactionOutputId将用于查找相关的TransactionOutput，允许矿工检查您的所有权。

创建TransactionOutputs 类：

**import** java.security.PublicKey**;**

**public** **class** **TransactionOutput** **{**

**public** String id**;**

**public** PublicKey reciepient**;** //also known as the new owner of these coins.

**public** **float** value**;** //the amount of coins they own

**public** String parentTransactionId**;** //the id of the transaction this output was created in

//Constructor

**public** **TransactionOutput(**PublicKey reciepient**,** **float** value**,** String parentTransactionId**)** **{**

**this.**reciepient **=** reciepient**;**

**this.**value **=** value**;**

**this.**parentTransactionId **=** parentTransactionId**;**

**this.**id **=** StringUtil**.**applySha256**(**StringUtil**.**getStringFromKey**(**reciepient**)+**Float**.**toString**(**value**)+**parentTransactionId**);**

**}**

//Check if coin belongs to you

**public** **boolean** **isMine(**PublicKey publicKey**)** **{**

**return** **(**publicKey **==** reciepient**);**

**}**

**}**

交易输出类将显示从交易中发送给每一方的最终金额。这些作为新交易中的输入参考，作为证明你可以发送的金额数量。

### 输入和输出 2：处理交易

区块链可能会收到很多交易，而区块链可能会非常长，因为我们必须查找并检查其输入，所以可能需要非常长的时间来处理新的交易。为了解决这个问题，我们保存了一个额外的集合称之为为使用的交易作为可用的输入,所以在主函数中增加一个集合称为UTXO。

**public** **class** **NoobChain** **{**

**public** **static** ArrayList**<**Block**>** blockchain **=** **new** ArrayList**<**Block**>();**

**public** **static** HashMap**<**String**,**TransactionOutputs**>** UTXOs **=** **new** HashMap**<**String**,**TransactionOutputs**>();** //list of all unspent transactions.

**public** **static** **int** difficulty **=** 5**;**

**public** **static** Wallet walletA**;**

**public** **static** Wallet walletB**;**

**public** **static** **void** **main(**String**[]** args**)** **{**

hashmap允许我们使用键来查找值，但是您需要导入java.util.hashmap

重点来了，我们在交易类中增加一个processTransaction方法，这个方法是把一切放在一起用来处理交易。

//Returns true if new transaction could be created.

**public** **boolean** **processTransaction()** **{**

**if(**verifiySignature**()** **==** **false)** **{**

System**.**out**.**println**(**"#Transaction Signature failed to verify"**);**

**return** **false;**

**}**

//gather transaction inputs (Make sure they are unspent):

**for(**TransactionInput i **:** inputs**)** **{**

i**.**UTXO **=** NoobChain**.**UTXOs**.**get**(**i**.**transactionOutputId**);**

**}**

//check if transaction is valid:

**if(**getInputsValue**()** **<** NoobChain**.**minimumTransaction**)** **{**

System**.**out**.**println**(**"#Transaction Inputs to small: " **+** getInputsValue**());**

**return** **false;**

**}**

//generate transaction outputs:

**float** leftOver **=** getInputsValue**()** **-** value**;** //get value of inputs then the left over change:

transactionId **=** calulateHash**();**

outputs**.**add**(new** TransactionOutput**(** **this.**reciepient**,** value**,**transactionId**));** //send value to recipient

outputs**.**add**(new** TransactionOutput**(** **this.**sender**,** leftOver**,**transactionId**));** //send the left over 'change' back to sender

//add outputs to Unspent list

**for(**TransactionOutput o **:** outputs**)** **{**

NoobChain**.**UTXOs**.**put**(**o**.**id **,** o**);**

**}**

//remove transaction inputs from UTXO lists as spent:

**for(**TransactionInput i **:** inputs**)** **{**

**if(**i**.**UTXO **==** **null)** **continue;** //if Transaction can't be found skip it

NoobChain**.**UTXOs**.**remove**(**i**.**UTXO**.**id**);**

**}**

**return** **true;**

**}**

//returns sum of inputs(UTXOs) values

**public** **float** **getInputsValue()** **{**

**float** total **=** 0**;**

**for(**TransactionInput i **:** inputs**)** **{**

**if(**i**.**UTXO **==** **null)** **continue;** //if Transaction can't be found skip it

total **+=** i**.**UTXO**.**value**;**

**}**

**return** total**;**

**}**

//returns sum of outputs:

**public** **float** **getOutputsValue()** **{**

**float** total **=** 0**;**

**for(**TransactionOutput o **:** outputs**)** **{**

total **+=** o**.**value**;**

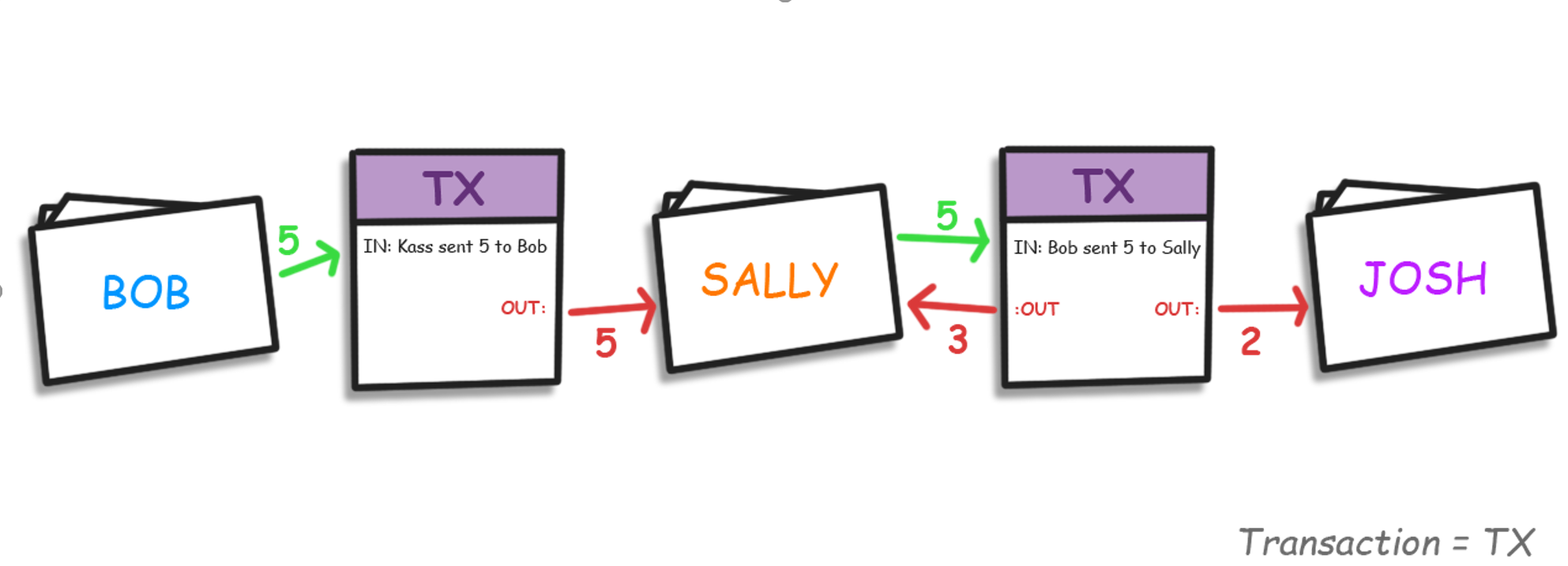
**}**

**return** total**;**

**}**

我们还添加了getinputsvalue方法

通过这种方法，我们执行一些检查以确保交易有效，然后收集输入并生成输出。最重要的是，最后，我们抛弃了输入在我们的UTXO列表，这就意味着一个可以使用的交易输出必须之前一定是输入，所以输入的值必须被完全使用，所以付款人必须改变它们自身的金额状态。



红色箭头是输出。请注意，绿色输入是对之前输出的参考

最后让我们修改我们的wallet类

* 搜集余额（通过循环遍历UTXO列表来检查交易的输出是否是我的）并
* 创建交易

**import** java.security.\***;**

**import** java.security.spec.ECGenParameterSpec**;**

**import** java.util.ArrayList**;**

**import** java.util.HashMap**;**

**import** java.util.Map**;**

**public** **class** **Wallet** **{**

**public** PrivateKey privateKey**;**

**public** PublicKey publicKey**;**

**public** HashMap**<**String**,**TransactionOutput**>** UTXOs **=** **new** HashMap**<**String**,**TransactionOutput**>();** //only UTXOs owned by this wallet.

**public** **Wallet()** **{...**

**public** **void** **generateKeyPair()** **{...**

//returns balance and stores the UTXO's owned by this wallet in this.UTXOs

**public** **float** **getBalance()** **{**

**float** total **=** 0**;**

**for** **(**Map**.**Entry**<**String**,** TransactionOutput**>** item: NoobChain**.**UTXOs**.**entrySet**()){**

TransactionOutput UTXO **=** item**.**getValue**();**

**if(**UTXO**.**isMine**(**publicKey**))** **{** //if output belongs to me ( if coins belong to me )

UTXOs**.**put**(**UTXO**.**id**,**UTXO**);** //add it to our list of unspent transactions.

total **+=** UTXO**.**value **;**

**}**

**}**

**return** total**;**

**}**

//Generates and returns a new transaction from this wallet.

**public** Transaction **sendFunds(**PublicKey \_recipient**,float** value **)** **{**

**if(**getBalance**()** **<** value**)** **{** //gather balance and check funds.

System**.**out**.**println**(**"#Not Enough funds to send transaction. Transaction Discarded."**);**

**return** **null;**

**}**

//create array list of inputs

ArrayList**<**TransactionInput**>** inputs **=** **new** ArrayList**<**TransactionInput**>();**

**float** total **=** 0**;**

**for** **(**Map**.**Entry**<**String**,** TransactionOutput**>** item: UTXOs**.**entrySet**()){**

TransactionOutput UTXO **=** item**.**getValue**();**

total **+=** UTXO**.**value**;**

inputs**.**add**(new** TransactionInput**(**UTXO**.**id**));**

**if(**total **>** value**)** **break;**

**}**

Transaction newTransaction **=** **new** Transaction**(**publicKey**,** \_recipient **,** value**,** inputs**);**

newTransaction**.**generateSignature**(**privateKey**);**

**for(**TransactionInput input: inputs**){**

UTXOs**.**remove**(**input**.**transactionOutputId**);**

**}**

**return** newTransaction**;**

**}**

**}**

你可以随时为钱包添加一些其他功能，例如记录您的交易历史记录。

### 添加交易到区块中

现在我们已经有了一个有效的交易系统，我们需要把交易加入到我们的区块链中。我们把交易列表替换我们块中无用的数据，但是在一个单一的区块中可能存放了1000个交易，这就会导致大量的hash计算，不用担心在这里我们使用了交易的merkle root，稍后你会看到。让我们增加一个帮助方法来创建merkle root在StringUtils类中。

在StringUtils类中创建getMerkleRoot方法

//Tacks in array of transactions and returns a merkle root.

**public** **static** String **getMerkleRoot(**ArrayList**<**Transaction**>** transactions**)** **{**

**int** count **=** transactions**.**size**();**

ArrayList**<**String**>** previousTreeLayer **=** **new** ArrayList**<**String**>();**

**for(**Transaction transaction **:** transactions**)** **{**

previousTreeLayer**.**add**(**transaction**.**transactionId**);**

**}**

ArrayList**<**String**>** treeLayer **=** previousTreeLayer**;**

**while(**count **>** 1**)** **{**

treeLayer **=** **new** ArrayList**<**String**>();**

**for(int** i**=**1**;** i **<** previousTreeLayer**.**size**();** i**++)** **{**

treeLayer**.**add**(**applySha256**(**previousTreeLayer**.**get**(**i**-**1**)** **+** previousTreeLayer**.**get**(**i**)));**

**}**

count **=** treeLayer**.**size**();**

previousTreeLayer **=** treeLayer**;**

**}**

String merkleRoot **=** **(**treeLayer**.**size**()** **==** 1**)** **?** treeLayer**.**get**(**0**)** **:** ""**;**

**return** merkleRoot**;**

**}**

我会尽快用一个实际的merkleroot取代这个方法，但是现在使用这个可以正常运行

修来Block 类

**import** java.util.ArrayList**;**

**import** java.util.Date**;**

**public** **class** **Block** **{**

**public** String hash**;**

**public** String previousHash**;**

**public** String merkleRoot**;**

**public** ArrayList**<**Transaction**>** transactions **=** **new** ArrayList**<**Transaction**>();** //our data will be a simple message.

**public** **long** timeStamp**;** //as number of milliseconds since 1/1/1970.

**public** **int** nonce**;**

//Block Constructor.

**public** **Block(**String previousHash **)** **{**

**this.**previousHash **=** previousHash**;**

**this.**timeStamp **=** **new** Date**().**getTime**();**

**this.**hash **=** calculateHash**();** //Making sure we do this after we set the other values.

**}**

//Calculate new hash based on blocks contents

**public** String **calculateHash()** **{**

String calculatedhash **=** StringUtil**.**applySha256**(**

previousHash **+**

Long**.**toString**(**timeStamp**)** **+**

Integer**.**toString**(**nonce**)** **+**

merkleRoot

**);**

**return** calculatedhash**;**

**}**

//Increases nonce value until hash target is reached.

**public** **void** **mineBlock(int** difficulty**)** **{**

merkleRoot **=** StringUtil**.**getMerkleRoot**(**transactions**);**

String target **=** StringUtil**.**getDificultyString**(**difficulty**);** //Create a string with difficulty \* "0"

**while(!**hash**.**substring**(** 0**,** difficulty**).**equals**(**target**))** **{**

nonce **++;**

hash **=** calculateHash**();**

**}**

System**.**out**.**println**(**"Block Mined!!! : " **+** hash**);**

**}**

//Add transactions to this block

**public** **boolean** **addTransaction(**Transaction transaction**)** **{**

//process transaction and check if valid, unless block is genesis block then ignore.

**if(**transaction **==** **null)** **return** **false;**

**if((**previousHash **!=** "0"**))** **{**

**if((**transaction**.**processTransaction**()** **!=** **true))** **{**

System**.**out**.**println**(**"Transaction failed to process. Discarded."**);**

**return** **false;**

**}**

**}**

transactions**.**add**(**transaction**);**

System**.**out**.**println**(**"Transaction Successfully added to Block"**);**

**return** **true;**

**}**

**}**

需要注意的是我们还更新了Block构造函数，因为我们不再需要传递字符串数据，并将merkle root包含在计算哈希方法中。addTransaction方法用来增加交易，只有满足条件下才可以成功的在区块中增加交易。

我们已经实现了一个可交易的区块链。

### 最后的测试

我们应该测试从钱包发送货币，更新区块链并进行有效性检查。但首先我们需要一种将新硬币引入混合的方法。有很多方法来创建新的硬币。在比特币区块链上，有很多方法可以创造新的比特币:矿工可以将交易包括在内，作为对每个矿工挖矿的奖励。但在这里我们只希望在创世纪区块中释放货币。就像比特币中一下，所以我们修改我们的主函数以达到下面的目的。

1. 创世纪区块发布100个货币给walletA
2. 一个更新的链有效性检查，考虑到交易。
3. 测试交易看是否一切正常。

**public** **class** **NoobChain** **{**

**public** **static** ArrayList**<**Block**>** blockchain **=** **new** ArrayList**<**Block**>();**

**public** **static** HashMap**<**String**,**TransactionOutput**>** UTXOs **=** **new** HashMap**<**String**,**TransactionOutput**>();**

**public** **static** **int** difficulty **=** 3**;**

**public** **static** **float** minimumTransaction **=** 0.1f**;**

**public** **static** Wallet walletA**;**

**public** **static** Wallet walletB**;**

**public** **static** Transaction genesisTransaction**;**

**public** **static** **void** **main(**String**[]** args**)** **{**

//add our blocks to the blockchain ArrayList:

Security**.**addProvider**(new** org**.**bouncycastle**.**jce**.**provider**.**BouncyCastleProvider**());** //Setup Bouncey castle as a Security Provider

//Create wallets:

walletA **=** **new** Wallet**();**

walletB **=** **new** Wallet**();**

Wallet coinbase **=** **new** Wallet**();**

//create genesis transaction, which sends 100 NoobCoin to walletA:

genesisTransaction **=** **new** Transaction**(**coinbase**.**publicKey**,** walletA**.**publicKey**,** 100f**,** **null);**

genesisTransaction**.**generateSignature**(**coinbase**.**privateKey**);** //manually sign the genesis transaction

genesisTransaction**.**transactionId **=** "0"**;** //manually set the transaction id

genesisTransaction**.**outputs**.**add**(new** TransactionOutput**(**genesisTransaction**.**reciepient**,** genesisTransaction**.**value**,** genesisTransaction**.**transactionId**));** //manually add the Transactions Output

UTXOs**.**put**(**genesisTransaction**.**outputs**.**get**(**0**).**id**,** genesisTransaction**.**outputs**.**get**(**0**));** //its important to store our first transaction in the UTXOs list.

System**.**out**.**println**(**"Creating and Mining Genesis block... "**);**

Block genesis **=** **new** Block**(**"0"**);**

genesis**.**addTransaction**(**genesisTransaction**);**

addBlock**(**genesis**);**

//testing

Block block1 **=** **new** Block**(**genesis**.**hash**);**

System**.**out**.**println**(**"\nWalletA's balance is: " **+** walletA**.**getBalance**());**

System**.**out**.**println**(**"\nWalletA is Attempting to send funds (40) to WalletB..."**);**

block1**.**addTransaction**(**walletA**.**sendFunds**(**walletB**.**publicKey**,** 40f**));**

addBlock**(**block1**);**

System**.**out**.**println**(**"\nWalletA's balance is: " **+** walletA**.**getBalance**());**

System**.**out**.**println**(**"WalletB's balance is: " **+** walletB**.**getBalance**());**

Block block2 **=** **new** Block**(**block1**.**hash**);**

System**.**out**.**println**(**"\nWalletA Attempting to send more funds (1000) than it has..."**);**

block2**.**addTransaction**(**walletA**.**sendFunds**(**walletB**.**publicKey**,** 1000f**));**

addBlock**(**block2**);**

System**.**out**.**println**(**"\nWalletA's balance is: " **+** walletA**.**getBalance**());**

System**.**out**.**println**(**"WalletB's balance is: " **+** walletB**.**getBalance**());**

Block block3 **=** **new** Block**(**block2**.**hash**);**

System**.**out**.**println**(**"\nWalletB is Attempting to send funds (20) to WalletA..."**);**

block3**.**addTransaction**(**walletB**.**sendFunds**(** walletA**.**publicKey**,** 20**));**

System**.**out**.**println**(**"\nWalletA's balance is: " **+** walletA**.**getBalance**());**

System**.**out**.**println**(**"WalletB's balance is: " **+** walletB**.**getBalance**());**

isChainValid**();**

**}**

**public** **static** Boolean **isChainValid()** **{**

Block currentBlock**;**

Block previousBlock**;**

String hashTarget **=** **new** String**(new** **char[**difficulty**]).**replace**(**'\0'**,** '0'**);**

HashMap**<**String**,**TransactionOutput**>** tempUTXOs **=** **new** HashMap**<**String**,**TransactionOutput**>();** //a temporary working list of unspent transactions at a given block state.

tempUTXOs**.**put**(**genesisTransaction**.**outputs**.**get**(**0**).**id**,** genesisTransaction**.**outputs**.**get**(**0**));**

//loop through blockchain to check hashes:

**for(int** i**=**1**;** i **<** blockchain**.**size**();** i**++)** **{**

currentBlock **=** blockchain**.**get**(**i**);**

previousBlock **=** blockchain**.**get**(**i**-**1**);**

//compare registered hash and calculated hash:

**if(!**currentBlock**.**hash**.**equals**(**currentBlock**.**calculateHash**())** **){**

System**.**out**.**println**(**"#Current Hashes not equal"**);**

**return** **false;**

**}**

//compare previous hash and registered previous hash

**if(!**previousBlock**.**hash**.**equals**(**currentBlock**.**previousHash**)** **)** **{**

System**.**out**.**println**(**"#Previous Hashes not equal"**);**

**return** **false;**

**}**

//check if hash is solved

**if(!**currentBlock**.**hash**.**substring**(** 0**,** difficulty**).**equals**(**hashTarget**))** **{**

System**.**out**.**println**(**"#This block hasn't been mined"**);**

**return** **false;**

**}**

//loop thru blockchains transactions:

TransactionOutput tempOutput**;**

**for(int** t**=**0**;** t **<**currentBlock**.**transactions**.**size**();** t**++)** **{**

Transaction currentTransaction **=** currentBlock**.**transactions**.**get**(**t**);**

**if(!**currentTransaction**.**verifiySignature**())** **{**

System**.**out**.**println**(**"#Signature on Transaction(" **+** t **+** ") is Invalid"**);**

**return** **false;**

**}**

**if(**currentTransaction**.**getInputsValue**()** **!=** currentTransaction**.**getOutputsValue**())** **{**

System**.**out**.**println**(**"#Inputs are note equal to outputs on Transaction(" **+** t **+** ")"**);**

**return** **false;**

**}**

**for(**TransactionInput input: currentTransaction**.**inputs**)** **{**

tempOutput **=** tempUTXOs**.**get**(**input**.**transactionOutputId**);**

**if(**tempOutput **==** **null)** **{**

System**.**out**.**println**(**"#Referenced input on Transaction(" **+** t **+** ") is Missing"**);**

**return** **false;**

**}**

**if(**input**.**UTXO**.**value **!=** tempOutput**.**value**)** **{**

System**.**out**.**println**(**"#Referenced input Transaction(" **+** t **+** ") value is Invalid"**);**

**return** **false;**

**}**

tempUTXOs**.**remove**(**input**.**transactionOutputId**);**

**}**

**for(**TransactionOutput output: currentTransaction**.**outputs**)** **{**

tempUTXOs**.**put**(**output**.**id**,** output**);**

**}**

**if(** currentTransaction**.**outputs**.**get**(**0**).**reciepient **!=** currentTransaction**.**reciepient**)** **{**

System**.**out**.**println**(**"#Transaction(" **+** t **+** ") output reciepient is not who it should be"**);**

**return** **false;**

**}**

**if(** currentTransaction**.**outputs**.**get**(**1**).**reciepient **!=** currentTransaction**.**sender**)** **{**

System**.**out**.**println**(**"#Transaction(" **+** t **+** ") output 'change' is not sender."**);**

**return** **false;**

**}**

**}**

**}**

System**.**out**.**println**(**"Blockchain is valid"**);**

**return** **true;**

**}**

**public** **static** **void** **addBlock(**Block newBlock**)** **{**

newBlock**.**mineBlock**(**difficulty**);**

blockchain**.**add**(**newBlock**);**

**}**

**}**

打印：

Connected to the target VM**,** address: '127.0**.**0.1**:**57085'**,** transport: 'socket'

Creating and Mining Genesis block**...**

Transaction Successfully added to Block

Block Mined**!!!** **:** 000b5a7ca6bf07639122cb31e884996895a482c281dd89c203824f1e93a661bf

WalletA's balance is: 100.0

WalletA is Attempting to send **funds** **(**40**)** to WalletB**...**

Transaction Successfully added to Block

Block Mined**!!!** **:** 000c7f814357abfea86ad1b38ec1dc3afed2afc9107f2bacc933d8136bf34df0

WalletA's balance is: 60.0

WalletB's balance is: 40.0

WalletA Attempting to send more **funds** **(**1000**)** than it has**...**

#Not Enough funds to send transaction**.** Transaction Discarded**.**

Block Mined**!!!** **:** 000b3822fb7985db438712816727d4bc382926a1c4654062aad4071d9b9fad98

WalletA's balance is: 60.0

WalletB's balance is: 40.0

WalletB is Attempting to send **funds** **(**20**)** to WalletA**...**

Transaction Successfully added to Block

WalletA's balance is: 80.0

WalletB's balance is: 20.0

Blockchain is valid

现在钱包能够安全地在您的区块链上发送金额，只要他们有金额发送。这意味着你有你自己的本地加密货币

### 可以进行交易的区块链

你已经成功地创建了自己的加密货币。你的区块链：

* 允许用户创建钱包
* 使用椭圆曲线加密方式为钱包提供公钥和私钥
* 通过使用数字签名算法证明所有权，确保资金转移
* 允许用户在区块链上进行交易

原文链接：[Creating Your First Blockchain with Java. Part 2-Transactions](https://medium.com/programmers-blockchain/creating-your-first-blockchain-with-java-part-2-transactions-2cdac335e0ce)