# 2019-04-26工作记录-对交易创建到签名的完整过程分析

学习到第7模块的1-21章节，完整的实现了创建区块、创建区块链、创建钱包、创建交易、实现转账、交易签名、签名验证功能。现在对这些过程进行一些梳理记录。

## 1 公钥私钥、钱包、钱包地址

Wallet.go 定义钱包结构：

**type** Wallet **struct**{  
 PrivateKey ecdsa.PrivateKey  
 PublicKey []byte  
}

只包含私钥和公钥两部分。创建钱包需要调用NewWallet()----🡪newKeyPair():

//生成私钥和公钥

func newKeyPair() (ecdsa.PrivateKey,[]byte){

//生成椭圆曲线, secp256r1 曲线。 比特币当中的曲线是secp256k1

curve :=elliptic.P256()

private,err :=ecdsa.GenerateKey(curve,rand.Reader)

if err !=nil{

fmt.Println("error")

}

//拼接x和y坐标，就是公钥

pubkey :=append(private.PublicKey.X.Bytes(),private.PublicKey.Y.Bytes()...)

return \*private,pubkey

}

上面代码中看到，私钥是通过椭圆曲线生成的，而公钥又是拼接x、y坐标而成。

钱包中的公钥比较长，使用中不方便，常常利用公钥计算出25字节的字符串代表这个钱包，就是钱包地址。

Ripemd160 base58

公钥PublicKey ---------》 PubKeyHash ----------》 address

字节码64字节 字节码20字节 字符串25字节

*//根据公钥计算出对应的公钥hash值，先SHA256后ripemd160，得到20字节的hash值***func** HashPubKey(publicKey []byte) []byte {  
 *//将传入的公钥进行256运算，返回256位hash值* hash256 := sha256.Sum256(publicKey)  
  
 *//将上面的256位hash值进行160运算，返回160位的hash值* ripemd160 := ripemd160.New()  
 \_,err := ripemd160.Write(hash256[:])  
 **if** err != nil{  
 fmt.Print(**"error=%s\n"**,err)  
 }  
  
 publicRIPEMD160 := ripemd160.Sum(nil)  
 **return** publicRIPEMD160  
}

//获取钱包地址，根据公钥计算出比特币地址。

func (w \*Wallet) GetAddress() []byte {

//调用Ripemd160Hash返回160位的Pub Key hash

ripemd160Hash := HashPubKey(w.PublicKey)

//将version+Pub Key hash

version\_ripemd160Hash := append([]byte{version},ripemd160Hash...)

//调用CheckSum方法返回前四个字节的checksum

checkSumBytes := CheckSum(version\_ripemd160Hash)

//checkSumBytes := CheckSum(ripemd160Hash) //这样写是错误的

//将version+PubKeyhash+ checksum生成25个字节

bytes := append(version\_ripemd160Hash,checkSumBytes...)

//将这25个字节进行base58编码并返回

return Base58Encode(bytes)

}

## 2 钱包集

前面说过钱包，就是一对私钥和公钥，简化的用一个字符串地址代表。一个钱包地址就代表一个可以正常收发转账的账户，在区块链是独立存在的。正常社会中一个人可以拥有多个银行账户，分别处理不同的事务（工资、 代缴水电气费、 网络支付、 理财）。在区块链上一个人也可以创建出多个钱包，集中存放在钱包集中进行管理。这些钱包用map[string]\*Wallet保存， key是钱包地址，value是钱包对象Wallet指针。这个钱包集还可以写入磁盘文件：

const walletFile = "wallet.dat"

//定义钱包集，里面通过map存储了多个钱包  
type Wallets struct{  
 Store map[string]\*Wallet //这里的Store变量必须首字母大写，这样才能在序列化时被输出。  
}

区块链上的操作几乎都需要用户钱包地址，因此程序一启动就要准备好钱包集。如果发现钱包文件存在就读取文件内容，恢复钱包地址；不存在就创建文件，并新建地址。

初始化钱包集的函数如下：

//读取文件建立钱包集

func NewWallets() (\*Wallets,error){

wallets := Wallets{}

wallets.Store = make(map[string]\*Wallet)

//改造: 如果发现钱包文件存在就读取文件内容，恢复钱包地址；不存在就创建文件，并新建地址。

\_,err := os.Stat(walletFile)

if os.IsNotExist(err){ //检查文件是否存在

fmt.Printf("钱包文件（%s）不存在，创建钱包文件...\n",walletFile)

wallets.CreateWallet()

wallets.SaveToFile2() //钱包集重新写入文件

err=nil //必须清空错误信息

}else{

err = wallets.LoadFromFile()

}

return &wallets,err

}

读取钱包文件的函数如下：

//读取文件内容，反序列化成钱包集, 要求这个文件必须存在  
func (ws \*Wallets) LoadFromFile() error{  
 fileContent,err := ioutil.ReadFile(walletFile)  
 if err !=nil{  
 log.Panic(err)  
 return err  
 }  
  
 var wallets Wallets //接受反序列化的临时变量  
 gob.Register(elliptic.P256())  
 decoder := gob.NewDecoder(bytes.NewReader(fileContent))  
 err = decoder.Decode(&wallets)  
 if err !=nil{  
 log.Panic(err)  
 return err  
 }  
  
 ws.Store = wallets.Store //把当前对象的store替换掉  
 return nil  
}

写入钱包文件的函数如下：

//把ws序列化后写入文件保存

func (ws \*Wallets) SaveToFile2() {

var content bytes.Buffer

gob.Register(elliptic.P256()) //注册钱包地址中用的椭圆曲线

encoder := gob.NewEncoder(&content)

err := encoder.Encode(ws)

if err != nil {

log.Panic(err)

}

err = ioutil.WriteFile(walletFile,content.Bytes(),0777)

if err != nil {

log.Panic(err)

}

}

## 3 普通的签名和验证

签名是对数据使用私钥进行加密。这个数据不是原始的数据形式，而是需要先计算出原始数据的Hash值，再进行加密运算得到r、s， 把r、s拼接起来就是签名数据了。

原始数据 数据Hash 私钥加密 签名数据（r+s）

“张三” 7b951fa2e14a21… r，s ccdc729eb1e93e160…

验证工作其实是使用公钥数据结构体对数据hash重新计算一遍签名，再和原有签名(r+s)进行比较，两者相同就表示验证成功，这些数据是这个人发出的，内容没有被更改。

示例代码：

func main(){

//调用函数生成公钥

privatekey,publickey :=newKeyPair()

//打印私钥 曲线上的x点

fmt.Printf("%x\n",privatekey.D.Bytes())

//打印公钥， 曲线上的x点和y点

fmt.Printf("%x\n",publickey)

//计算数据签名示例

h1:=sha256.Sum256([]byte("Json"))

//根据私钥和信息的哈希进行签名

r,s,err := ecdsa.Sign(rand.Reader, &privatekey, h1[:])

if err != nil {

fmt.Println("error:",err)

}

fmt.Printf("r=%x\n",r)

fmt.Printf("s=%x\n",s)

//r,s拼接在一起实现数字签名

signature := append(r.Bytes(), s.Bytes()...)

fmt.Printf("%x\n",signature)

//数据签名验证 ----------------------

//生成secp256k1椭圆曲线

curve := elliptic.P256()

//生成公钥数据结构体

len:=len(publickey)

x := big.Int{}

y := big.Int{}

x.SetBytes(publickey[:len/2]) // x轴的坐标

y.SetBytes(publickey[len/2:]) // y轴的坐标

rawPubKey := ecdsa.PublicKey{curve, &x,&y}

//验证签名，实际上就是使用公钥数据结构体对信息的hash串重新计算一遍签名，再和原有签名(r+s)进行比较

ret := ecdsa.Verify(&rawPubKey,h1[:],r, s)

if ret==false{

fmt.Printf("验证失败\n")

}else{

fmt.Printf("验证成功\n")

}

//这个乱改了h2，就会验证失败

h2 :=sha256.Sum256([]byte("Jsom"))

ret = ecdsa.Verify(&rawPubKey,h2[:],r, s)

if ret==false{

fmt.Printf("验证失败\n")

}else{

fmt.Printf("验证成功\n")

}

}

## 4 区块链、区块结构、交易关系图

本文中的区块结构是简化的，关键的内容与比特币的结构一致。







关于交易Vin和Vout的对应关系：



## 5 创建区块链过程

功能入口：test.go TestCLiArgs(),

blockchain.go：

*//定义区块链的基本结构，hash+存储的数据库***type** BlockChain **struct**{  
 tip []byte *//链上最新的区块的hash值* db \* bolt.DB *//数据库*}  
*//定义区块链的迭代器***type** BlockChainIterator **struct**{  
 currenthash []byte  
 db \* bolt.DB  
}

*// 注意，矿工的钱包地址一定要在钱包集中，不然以后转账时在钱包集中找不到矿工钱包地址，会出现map容器返回空指针。***const *minneraddress*** = **"14npxLBj8eGwCcGJPiuqoG4U6ssW7KA3hs"**

NewBlockChain(minneraddress)创建一个区块链. 不存在就创建，存在就获取最新的区块信息, 参数是矿工钱包地址:base58编码的字符串，不是字节数组。

//建立CoinBase挖矿奖励交易

transation := NewCoinbaseTX(address, genesisdata)

genesis := NewGensisBlock([]\*Transation{transation}) //建立创世区块

err = b.Put([]byte("L"),genesis.Hash) //新区块与“L”关联， L表示last，以后方便的寻找到区块链的最新头部

### 5.1区块链对象的创建过程说明：

第一次运行时没有区块链对象及数据库文件，就需要自动创建出区块链。

1 创建保存全部区块数据的数据库文件

***dbFile*const *dbFile*** = **"blockchain.db"**

db,err := bolt.Open(***dbFile***,0600,nil)

2 建立CoinBase，这是创世区块的挖矿奖励交易

transation := NewCoinbaseTX(address, ***genesisdata***)

3 建立创世区块

genesis := NewGensisBlock([]\*Transation{transation})

4 把创世区块序列化后存入数据桶中，映射的键值是创世区块的Hash值。

err = b.Put(genesis.Hash, genesis.Serialize())

5新区块hash值与字符串“L”关联， L表示last，以后方便的寻找到区块链的最新头部区块。根据区块hash值就可以从数据桶中找到区块的序列化数据，经过反序列化就能恢复成区块数据结构。

err = b.Put([]byte(**"L"**),genesis.Hash)

6 有了数据库和最新的区块hash值，就能创建出全局唯一的区块链对象

bc :=BlockChain{tip,db}

核心函数：NewBlockChain

*//创建一个区块链. 不存在就创建，存在就获取最新的区块信息, 参数是矿工地址:base58编码的字符串，不是字节数组***func** NewBlockChain(address string) \*BlockChain{  
 **var** tip []byte  
  
 db,err := bolt.Open(***dbFile***,0600,nil)  
 **if** err !=nil{  
 log.Panic(err)  
 }  
 err = db.Update(**func**(tx \*bolt.Tx) error{  
  
 b:=tx.Bucket([]byte(***blockBucket***)) *//获得数据库的桶* **if** b==nil{  
 fmt.Println(**"区块链不存在，建立创世区块，建立新的区块链"**)  
 b,err:=tx.CreateBucket([]byte(***blockBucket***))  
 **if** err !=nil{  
 log.Panic(err)  
 }  
  
 *//建立CoinBase挖矿奖励交易* transation := NewCoinbaseTX(address, ***genesisdata***)  
 genesis := NewGensisBlock([]\*Transation{transation}) *//建立创世区块* err = b.Put(genesis.Hash, genesis.Serialize()) *//区块数据写入数据库桶中* **if** err !=nil{  
 log.Panic(err)  
 }  
 err = b.Put([]byte(**"L"**),genesis.Hash) *//新区块与“L”关联， L表示last，以后方便的寻找到区块链的最新头部* **if** err !=nil{  
 log.Panic(err)  
 }  
 tip = genesis.Hash  
 }**else**{  
 *//区块链存在，获取“L”对应的区块数据* tip = b.Get([]byte(**"L"**))  
 }  
  
 **return** nil  
 })  
  
 **if** err !=nil{  
 log.Panic(err)  
 }  
  
 *//根据tip和db建立区块链对象* bc :=BlockChain{tip,db}  
 **return** &bc  
}

### 5.2 创世区块的创建过程说明

1 初始化区块，填写版本号、网络难度、时间戳、coinbase交易。

2 核心工作是进行POW，对这个区块数据计算出满足指定难度要求的hash，这时的nonce就确定下来了。

3 填写nonce和hash，区块创建完成。

*//建立创世区块，输入参数是交易***func** NewGensisBlock(transations []\*Transation) \* Block{  
 *//初始化区块* block :=&Block{  
 []byte{},  
 1,  
 []byte{},  
 []byte{},  
 uint32(time.Now().Unix()),  
 453281356, *//453281356* 0,  
 transations,  
 }  
  
 pow := NewProofOfWork(block)  
 nonce, hash := pow.Run()  
 *//fmt.Printf("NewGensisBlock()： hash=%x\n",hash)* block.Nonce = nonce  
 block.Hash = hash  
  
 *//block.ToString()* **return** block  
}

### 5.3 普通区块的创建过程说明

创世区块是伴随着区块链建立生成的，之后都是矿工挖矿产生普通区块，一般会记录交易信息，如果这段出块时间内没有交易，区块内的交易记录填写为空。

功能入口：cli.go

*//根据命令行参数添加区块***func** (cli \*CLI) addBlock(){  
 cli.bc.MineBlock([]\*Transation{}) *//先添加一个空的交易列表*}

MineBlock挖矿的具体过程如下：

1 先检查输入的交易签名是否正确

2 找到“L”对应的最新区块头hash。

3 根据输入参数（交易记录、最新区块头hash）建立区块，NewBlock ()函数内部完成。

4 剩下的工作和NewBlockChain函数的后半截相同，要写入数据库，更新最新区块头hash。

*//这个就是链上的挖矿动作，链上添加一个区块，记录到数据库中***func** (bc \*BlockChain) MineBlock(transations []\*Transation){  
 *//先检查输入的交易签名是否正确* **for** \_,tx := **range** transations {  
 **if** bc.VerifyTransation(tx) == ***false*** {  
 log.Panic(**"BlockChain.MineBlock() : ERROR: Invalid transation!"**)  
 } **else** {  
 fmt.Println(**"BlockChain.MineBlock() :transation verify success!"**)  
 }  
 }  
  
 **var** lasthash []byte  
 err := bc.db.View(**func**(tx \* bolt.Tx)error{  
 b:= tx.Bucket([]byte(***blockBucket***))  
 lasthash = b.Get([]byte(**"L"**))  
 **return** nil  
 })  
 **if** err!=nil{  
 log.Panic(err)  
 }  
 newBlock := NewBlock(transations, lasthash)  
  
 *//把新区块写入数据库中* bc.db.Update(**func**(tx \*bolt.Tx) error{  
 b:=tx.Bucket([]byte(***blockBucket***))  
 err:=b.Put(newBlock.Hash,newBlock.Serialize())  
 **if** err !=nil{  
 log.Panic(err)  
 }  
 *//把“L”关联新区块hash值* err = b.Put([]byte(**"L"**),newBlock.Hash)  
 **if** err !=nil{  
 log.Panic(err)  
 }  
  
 bc.tip = newBlock.Hash  
 **return** nil  
 })  
}

### 5.4 遍历区块链

区块链是一个单向链表，每一个区块指向前一个区块，指针不是内存地址，而是区块的hash值。遍历只能从头到尾的方向进行。规定创世区块为尾部，最新区块为头部。自己建立一个迭代器，内部的指针是指向下一个区块hash。通过Next函数返回当前区块，更新指针指向下一个区块。当发现下一个区块的hash长度为0表示已经是创世区块了，遍历结束。

*//获取迭代器所指的当前块, 读完后就指向前一个区块的hash值。 根据每个hash值在数据库中找到对应的区块序列化数据。***func** (bci \*BlockChainIterator) Next() \*Block{  
 **var** block \*Block  
 err := bci.db.View(**func**(tx \*bolt.Tx) error{  
 b:=tx.Bucket([]byte(***blockBucket***))  
 data := b.Get(bci.currenthash)  
 block = DeserializeBlock(data)  
 **return** nil  
 })  
 **if** err!=nil{  
 log.Panic(err)  
 }  
 bci.currenthash = block.PrevBlockHash  
 **return** block  
}

*//遍历打印区块链***func** (bc \*BlockChain) PrintBlockChain(){  
 bci := bc.iterator()  
  
 **for**{  
 block := bci.Next()  
 block.ToString()  
 fmt.Println()  
  
 *// PrevBlockHash长度==0表示这是创世区块，遍历结束* **if** len(block.PrevBlockHash)==0{  
 **break** }  
 }  
}

## 6 默克尔树建立过程

默克尔树根hash是区块头中必须写入的数据，需要根据这个区块中的交易记录计算出来。具体过程如下：

Merkle.go

//定义默克尔树节点

type MerkleNode struct{

Left \*MerkleNode

Right \*MerkleNode

Data []byte //存储左右节点拼接后计算出来的hash值

}

//定义默克尔树， 只要有个根节点就可以追溯整棵树节点

type MerkleTree struct{

RootNode \*MerkleNode

}

功能入口：test.go

*//测试创建区块的默克尔根*func TestCreateMerkleTreeRoot() {

…

block.createMerkleTreeRoot(Transations)

…

}

createMerkleTreeRoot (transations []\*Transation)工作过程：

1. 遍历交易列表，把每笔交易hash单独存入一个数组。

根据数组生成默克尔树（NewMerkleTree）。

1. 把默克尔树根的hash值写入区块头部。

NewMerkleTree函数工作工程：

1数组中的hash值作为初始叶节点，添加入树节点列表。

2计算树的层次，只要知道初始叶节点总数就能算出来每一层的节点数量。

假设初始叶节点总数nSize，那么上一层的节点总数为(nSize+1)/2, 通过迭代公式：

nSize = (nSize+1)/2就能得到每一层的个数。当nSize==1时停止。

3 遍历叶节点，两两划分计算出hash，遇到落单的节点就和自身计算出hash。这些计算出的hash作为新一层的节点存入节点列表。

4 继续遍历新一层节点，继续两两划分，经过两次hash256计算出hash存入节点列表。

5 重复上述过程2、3、4，直到这一层只有一个节点，就是树根节点了。

6 返回树根节点的hash值。

*//创建完整的默克尔树，输入参数是hash值数组，每一个hash值都是一个切片字节数组，就表现成了二维数组形式。***func** NewMerkleTree(data [][]byte) \*MerkleTree {  
  
 *// 定义节点列表* **var** nodes []MerkleNode  
  
 *// 创建全部初始叶节点* **for** \_,datum := **range**(data){  
 node := NewMerkleNode(nil,nil,datum)  
 nodes = append(nodes,\*node)  
 }  
  
 *//第一层循环代表树的层数，节点越多那么层级越多，第一层的循环次数就多。  
 //例如5个节点就3层，循环3次。 每次循环时nSize就代表本层的节点个数* j:=0 *//nodes中节点序号标记，表示了在每一层中第一个节点序号* **for** nSize:=len(nodes);nSize>1;nSize=(nSize+1)/2{  
 *//第二层循环，在每一层中两两计算出一个新节点，插入到节点列表尾巴，孤单节点就与自身计算出一个新节点。  
 //在nodes中存储了所有原始的节点，依靠每层的节点个数划分层* **for** i:=0;i<nSize;i+=2{  
 i2:=min(i+1,nSize-1) *//对于剩下的孤单节点i+1会越界，因此要限制在nSize-1.* node := NewMerkleNode(&nodes[j+i],&nodes[j+i2],nil)  
 nodes = append(nodes,\*node)  
 }  
 j+=nSize  
 }  
 *//循环结束，nodes中的最后一个节点就是树根* treeRoot := MerkleTree{&nodes[len(nodes)-1]}  
 **return** &treeRoot  
}

## 7 工作量证明机制POW

Pow是比特币的核心算法，现在就破解它，了解计算难度和为什么这样设计。

### 7.1 区块头的hash值含义

区块中包含了很多数据，为了防止恶意修改，只要计算出区块数据的hash值作为检查的手段就可以了。随便来一段区块数据都能立即计算出sha256的hash值，这样快不好吗？中本聪设计时就考虑应用场景：多台电脑同时独立计算，但只有一台最先算出hash的电脑能获得挖矿奖励。通过加大计算量和难度让不同性能的电脑计算出结果的速度有快有慢，性能优良的电脑计算获胜概率大，从中获得回报。

难度怎么设计呢？ 区块数据是固定的，添加一个随机数扰动nonce，让生成的hash符合这种形式：000000123456……,就是前面几位固定是0。 当然0越多越难以凑出这样的hash，只能遍历nonce，从0~2^32，看看能否凑出计算指定的计算结果。这个形式就是网络计算难度。当许多台电脑同时计算时，找到正确的nonce速度就快，也就是挖矿速度就快。中本聪设计时要求比特币每10分钟产生一个块，意味着10分钟左右才能计算出一次正确的hash。那么如何应对越来越厉害的专用矿机呢？一个方法是每4年奖励降低一半， 从一开始的100/50/25/12.5。奖励少了自然挖矿的就少了。另一个方法是检测网络中的出块时间间隔，发现少于10分钟就提高难度（hash前面更多的零），时间间隔变长了就降低难度。

### 7.2 计算网络难度值bits

实际的区块中都有一个bits字段，是一个大整数，代表网络计算难度，下面就解析这个bits数值。



*以bits=453281356为例，16进制表达是0x1B04864C  
//bits计算结果 000000000004864c000000000000000000000000000000000000000000000000  
//block hash 00000000000090ff2791fe41d80509af6ffbd6c5b10294e29cdf1b603acab92c  
//Previous Block 0000000000045b02ab29280b9df7e9513fa6fe274f0d7fd7ecf95c6d708ceb29  
//上面的hash值表明，生成的hash值必须小于bits计算结果。 换句话说，只要小于bits计算结果就满足要求。实际计算出的nonce=*3806873890**func** CaculateTargetValue(bits []byte) []byte{  
 *//第一个字节表示指数* exp := bits[0]  
 fmt.Printf(**"exp=%d\n"**,exp)  
 fmt.Printf(**"type=%T\n"**,exp)  
  
 *//计算后面3个字节* coefficient := bits[1:]  
  
 *//拼接出目标hash值格式* result := append(bytes.Repeat([]byte{0x00}, 32-int(exp)),coefficient...)  
 result = append(result,bytes.Repeat([]byte{0x00}, 32-len(result))...)  
  
 **return** result  
}

上面的难度要求前面有10个0，相当难算，我的笔记本每秒大概计算10万个nonce，算了2小时才检查18亿个nonce，离38亿的结果值远的很呢。我们的示例程序中不能用这么难的困难度，否则就跑不动了，用的简单方法要求hash前面有4个0。

### 7.3 示例代码中的简易难度

proofOfWork.go: 看一下这段代码，

*//计算工作量证明，只要提供区块和难度就能自动计算出来这个nonce***type** ProofOfWork **struct**{  
 block \* Block  
 target \* big.Int *//这就是bits对应大整数，这里直接硬代码赋值*}

**const *targetBits*** = 16 *//表示后面有256-16=240位为0，前面就是0x0001啦*

**func** NewProofOfWork(b \* Block) \*ProofOfWork{  
 target :=big.NewInt(1)  
 target.Lsh(target,uint(256-***targetBits***))  
 pow :=&ProofOfWork{b,target}  
 **return** pow  
}

我们直到hash是32字节256位， 我们直接指定目标hash的前面有n位，那么后面就有(256-n)个0。方法是通过1<< (256-n)。上面代码中n=16，256-16=240，表示1<<240,那么后面就是240位0。前面就是16位，二进制是0000 0000 0000 0001，刚好是0x0001，合起来就是target：

0x00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00，很大的长整数，表示网络难度。 只要计算出的hash前面小于0x0001，也就是前面有4个0，就算满足要求，比起之前标准的前面10个0不知道简单了多少倍。

### 7.4 nonce计算过程

1. Nonce=0，开始循环
2. 准备数据，把区块的数据组装成字节数组
3. 对字节数组进行两次SHA256计算出hash值。
4. 比较hash是否小于指定的target；

小于就达到目标，计算停止。

不小于就继续递增nonce值，回到第2步继续计算区块的hash值。

1. 区块的头部hash更新成计算的结果。

核心编码：

**func** (pow \*ProofOfWork) PrepareData(nonce uint32) []byte{  
 data :=bytes.Join(  
 [][]byte{  
 IntToHex(pow.block.Version),  
 pow.block.PrevBlockHash,  
 pow.block.Merkleroot,  
 IntToHex(pow.block.Time),  
 IntToHex(pow.block.Bits),  
 IntToHex(uint32(nonce)), },  
 []byte{},  
 )  
  
 **return** data  
}  
  
*//开始挖矿计算***func** (pow \* ProofOfWork) Run() (uint32,[]byte){  
 **var** nonce uint32 = 0  
 **var** secondHash [32]byte  
 **var** currentHash big.Int  
  
 **for** nonce < maxnonce {  
 *//序列化* data :=pow.PrepareData(nonce)  
 *//double hash* firstHash := sha256.Sum256(data)  
 secondHash = sha256.Sum256(firstHash[:])  
 currentHash.SetBytes(secondHash[:])  
 *//fmt.Printf("nonce=%d, currenthash=%x\n",nonce, secondHash)  
  
 //比较* **if** currentHash.Cmp(pow.target) ==-1{  
 **break** }**else**{  
 nonce++  
 }  
 }  
  
 **return** nonce,secondHash[:]  
}  
  
*//验证nonce是否正确***func** (pow \* ProofOfWork) Validate() bool{  
 **var** hashInt big.Int  
  
 data := pow.PrepareData(pow.block.Nonce)  
  
 firstHash := sha256.Sum256(data)  
 secondHash := sha256.Sum256(firstHash[:])  
 hashInt.SetBytes(secondHash[:])  
  
 isValid := hashInt.Cmp(pow.target)==-1  
 **return** isValid  
}

## 8 交易机制

终于要讲到交易机制了，这里可以说是最难的核心流程，涉及到UTXO 模型+区块链上的搜索+交易签名+签名验证，这里简单些，只讲UTXO模型，把交易签名和签名验证放到最后的章节讲述。

日常生活中我们都有银行存折，看到上面存入、支出、余额栏目，很好理解。但是比特币中的钱包地址只关联交易，而在每一笔交易中只有输入、输出记录，代表着支出、存入概念，故意没有余额。因此要想找出一个钱包地址（例如张三钱包）的当前余额，就需要遍历全部区块的全部交易，找到张三关联的输出交易、输入交易，累计计算出余额。想象一下，现在区块高度已经573344，也就是要遍历57万个区块，每个区块中可能有多达10条交易，把关于张三地址的交易全部查找一遍，存心要累死电脑啊。

中本聪为什么要这么设计呢？ 我的体会是数据结构故意去除余额，要想获得余额必须计算全部相关交易，这样任何人也无法直接修改余额，要想对余额恶意修改，只能修改之前的交易，但是每一笔交易又是通过验证存在区块中的，区块也是验证通过的。每一笔交易又引用之前区块的交易输出，这样只能修改大量的区块交易数据才可能通过各种验证，攻击者只能靠调动强大的算力进行51%攻击，代价太大，得不偿失。

当然对于比特币10分钟一个块，每秒10笔交易量级来说主要是发生在计算pow随机数上，处理交易花的时间较少，交易笔数稀少，速度影响不大。如果要商业化运行，每秒100~1000笔交易，这种查找余额的方法太慢了。第二代比特币网络以太坊就增加了账户余额直接解决了这个问题。

那么在比特币中又是怎么解决这个查找余额的呢？使用了UTXO模型。简单的说，一笔交易的输入结构代表引用前一笔交易的输出，只要出现在交易输入区域的输出条目，就表示被花掉了。只有那些从未在输入区域Vin中出现的输出区域Vout才是自己的余额。核心思想表示如下：

1）最新节点为头，创世节点为尾，遍历节点，

2）遍历节点的交易列表

3） 把Vin中引用的交易存入已花费列表

4） 把Vout区域中的交易查找是否存在与已花费列表。如果存在就遍历下一笔交易，不存在表示找到了一笔未花费的交易，存入未花费列表。

5）累计未花费列表的总额就是余额。

### 8.1 核心函数FindUnspentTransations()

blockchain.go:

该函数输入参数是用户钱包地址的公钥hash值形式，找到所有属于自己的未花费交易，返回存储这些交易的数组。

*//==================自己改进的代码=======================================================*

*//找出指定用户address的所有未花费输出，需要遍历整个区块链*

*//改进了bug：当发现交易中的一笔未花费输出时没有记录下输出的序号Voutindex，而是添加这条交易。应该要保存这条交易和输出序号。*

*//返回：多了一个每条交易的输出序号数组*

*func (bc \*BlockChain) FindUnspentTransations2(pubkeyhash []byte) ([]Transation, map[string][]int){*

*var unspentTXs []Transation //所有未花费的交易记录*

*var unspendTXOs = make(map[string][]int) //未花费交易输出序号， key是交易的ID字符串，value是输出序号数组*

*/\*定义映射关系:*

*key: string（交易的hash值）*

*value: []int（存储已经花费的交易的序号）*

*表示这笔交易（hash）的输出序号，已经被花费了。 \*/*

*spendTXOs := make(map[string][]int) //已花费交易记录*

*// 第一层循环：遍历区块链的区块*

*bci :=bc.iterator()*

*for{*

*block := bci.Next()*

*//第二层循环：遍历该区块中的每一笔交易*

*for \_,tx := range block.Transations{*

*txID := hex.EncodeToString(tx.ID) //交易的hash值转成字符串形式*

*//第三层循环： 遍历这笔交易中的输出*

*loop3: for outIdx,out := range tx.Vout{*

*//如果这笔交易在已花费交易记录中存在，说明必然有一个输出被花费。*

*//通过循环找到记录的输出序号。序号对上了表示这个输出已经被花费，跳出来检查下一个输出。*

*if spendTXOs[txID] != nil { //找到这笔交易*

*for \_,spentOut := range spendTXOs[txID]{*

*if spentOut == outIdx{ //找到了这笔输出序号*

*continue loop3*

*}*

*}*

*}*

*// 程序跑到这里说明这笔输出未被花费，写入未花费交易记录。注意检查指定地址*

*// by the way, 最后一个区块的输出都是未被使用的*

*if out.CanBeUnlockedWith(pubkeyhash){*

*unspentTXs = append(unspentTXs, \*tx)*

*unspendTXOs[txID] = append(spendTXOs[txID], outIdx)*

*}*

*}*

*//遍历这笔交易中的输入，只要是输入就表示被使用了，需要添加到已花费交易记录中*

*//CoinBase交易没有输入，跳过*

*//注意 spendTXOs[inTxID]可能存入了多个输出序号，因此是个序号数组*

*if tx.isCoinBase() == false{*

*for \_,in := range tx.Vin{*

*if in.CanBeUnlockedWith(pubkeyhash){*

*inTxID := hex.EncodeToString(in.TXid)*

*//参数说明 交易的哈希值， 输出索引*

*spendTXOs[inTxID] = append(spendTXOs[inTxID], in.Voutindex)*

*}*

*}*

*}*

*}*

*if len(block.PrevBlockHash)==0{*

*break*

*}*

*}*

*//fmt.Println(unspentTXs)*

*return unspentTXs,unspendTXOs*

*}*

8.2 核心函数FindUTXO()找到UTXO，返回的是未使用的交易输出数组

**func (bc \*BlockChain) FindUTXO2(pubkeyhash []byte) []TXOutput{**

**var UTXOs []TXOutput**

**unspentTXs,unspendTXOs := bc.FindUnspentTransations2(pubkeyhash)**

**for \_,tx :=range unspentTXs{**

**txID := hex.EncodeToString(tx.ID) //交易的hash值转成字符串形式**

**for \_,outIdx:= range unspendTXOs[txID]{**

**out := tx.Vout[outIdx]**

**if out.CanBeUnlockedWith(pubkeyhash){**

**UTXOs = append(UTXOs,out)**

**}**

**}**

**}**

**return UTXOs**

**}**

8.3 根据转账金额，找到能满足转账金额的未花费交易输出

现在看看转账操作，都要A转给B地址X金额，那么首先就要查找A地址的所有未花费交易， 再依据一定规则从中找出能一部分交易：总和total>=X。例如先花费小金额的输出。如果不能满足金额要求说明A的余额不足。能满足的话就可以构造一笔交易，如果还有剩余零头就作为一笔输出归还给A地址。

*//找出能满足指定（地址+金额）的 未花费交易输出，用这些交易作为输入够转账了*

*func (bc \*BlockChain) FindSpendableOutputs2(pubkeyhash []byte, amount int) (int,map[string][]int){*

*unspentOutputs := make(map[string][]int)*

*unspentTXs,unspendTXOs := bc.FindUnspentTransations2(pubkeyhash)*

*accumulated :=0 //检查累计金额*

*breakPoint: for \_,tx := range unspentTXs{*

*txID := hex.EncodeToString(tx.ID) //交易的hash值转成字符串形式*

*for \_,outIdx:= range unspendTXOs[txID]{*

*out := tx.Vout[outIdx]*

*if out.CanBeUnlockedWith(pubkeyhash) && accumulated <amount {*

*accumulated += out.Value*

*unspentOutputs[txID] = append(unspentOutputs[txID],outIdx)*

*if accumulated >= amount{*

*break breakPoint*

*}*

*}*

*}*

*}*

*return accumulated,unspentOutputs*

*}*

### 8.4 转账操作函数NewUTXOTransation()

现在前面的准备工作都做好了，就能处理具体的转账操作了。一般流程如下：

1从钱包集中找到发送方钱包，里面有公钥私钥等数据可用。

2找到可用的UTXO，检查余额是否足够，不够的话提示余额不足。

3构建Vin数据，可能有多条TXInput

4构建Vout数据，注意剩下的余额要转给发送方

5根据Vin和Vout填写交易结构体，注意要调用hash()方法计算这笔交易的hash值

6用私钥对交易进行签名

到此一笔完整的交易就创建出来了。

Transation.go :

*//根据发送方、接收方、转账金额创建出对应的交易  
func NewUTXOTransation(from,to string,amount int, bc \*BlockChain) \*Transation{  
 var inputs []TXInput  
 var outputs []TXOutput  
  
 wallets,err := NewWallets()  
 if err !=nil{  
 log.Panic(err)  
 }  
 //根据发送方地址找到对应的钱包，里面包含了公钥和私钥，可用于签名  
 wallet := wallets.GetWallet(from)  
  
 acc,validoutputs :=bc.FindSpendableOutputs2(HashPubKey(wallet.PublicKey),amount)  
 if acc < amount{  
 log.Panic("Error: Not enough funds")  
 }  
  
 //遍历输出，得到每笔交易的hash和输出序号  
 for txid,outs := range validoutputs{  
 txID,err :=hex.DecodeString(txid) //把交易hash值从字符串形式转成字节切片形式  
 if err !=nil{  
 log.Panic(err)  
 }  
  
 //遍历每一个序号, 把这笔输出作为新交易的Vin项。  
 //交易输入需要用户的公钥，只能从钱包集中找到指定的钱包，再得到公钥。  
 for \_,out :=range outs{  
 input := TXInput{txID,out,nil,wallet.PublicKey}  
 inputs = append(inputs,input)  
 }  
 }  
  
 //开始填写Vout项，注意这些Vin总金额可能>转账金额，要把剩下的余额还给发送方  
 outputs = append(outputs,\*NewTXOutput(amount,to))  
 if acc > amount{  
 outputs = append(outputs,\*NewTXOutput(acc-amount,from))  
 }  
  
 //根据Vin和Vout填写交易结构体，注意要调用hash()方法计算这笔交易的hash值  
 tx := Transation{nil,inputs,outputs}  
 tx.ID = tx.Hash()  
  
 //用私钥对交易进行签名  
 bc.SignTransation(&tx, wallet.PrivateKey)  
 return &tx  
}*

8.5 转账命令函数send()

cli.go 只要输入发送方、接收方、金额就自动完成创建交易、创建区块工作。

*//转账操作，先生成一笔新交易， 存入挖矿所得的新区块中  
func (cli \*CLI) send(from, to string, amount int){  
 tx := NewUTXOTransation(from,to,amount,cli.bc)  
 cli.bc.MineBlock([]\*Transation{tx})  
 fmt.Printf("send success!\n")  
}*

## 9 交易签名和交易验证

上面讲解了所有的交易过程，现在就说说交易中的签名和验证是怎么进行的。

### 9.1基本原理：

交易的任何一笔输出都要指定接收方A，而且必须限定只有接收方才能使用这笔钱。那么用接受方的公钥hash进行签名，这样以后A作为发送方引用这笔输出时，就可以用A自己的私钥验证这个签名，别人是无法验证成功的。通过这样的机制保证只有A能动用自己的UTXO。

### 9.2 coninbase交易上锁

所有的钱都来自于挖矿生成的coninbase交易。Coinbase交易的输入是空，用不着加密验证，但是输出就需要指定矿工的公钥hash。

核心函数： transation.go

*//根据金额与地址新建一个输出  
func NewTXOutput(value int ,address string) \*TXOutput{  
 txo := TXOutput{value,nil}  
 //txo.PubkeyHash = []byte(address)  
 txo.Lock([]byte(address)) //设置公钥hash  
 return &txo  
}*

*//交易输出的上锁，这个公钥的hash值就对应着一个比特币地址，也就是钱包地址*

*func (out \*TXOutput) Lock(address []byte){*

*decodeAddress := Base58Decode(address)*

*pubkeyhash := decodeAddress[1:len(decodeAddress)-addressChecksumLen]*

*out.PubkeyHash = pubkeyhash*

*}*

### 9.3普通交易的签名过程

普通交易的签名过程分为Vin和Vout两部分单独处理。其中针对Vout的处理比较简单，与coinbase交易的输出签名相同，在调用NewTXOutput()时就指定了接收方公钥hash。

主要的签名工作在Vin部分的处理。注意看这段代码：

1 transation.go NewUTXOTransation()🡪 创建交易，调用签名函数，用私钥签名

*//遍历每一个序号, 把这笔输出作为新交易的Vin项。  
//交易输入需要用户的公钥，只能从钱包集中找到指定的钱包，再得到公钥。***for** \_,out :=**range** outs{  
 input := TXInput{txID,out,nil,wallet.PublicKey}  
 inputs = append(inputs,input)  
}

这里创建的输入项Signature为空，Pubkey填写成接收方公钥wallet.PublicKey。

2transation.go NewUTXOTransation()

*//用私钥对交易进行签名*bc.SignTransation(&tx, wallet.PrivateKey)

blockchain.go SignTransation() 🡪 找到输入引用的所有以前的交易，开始签名。注意：传入的参数是接收方私钥wallet.PrivateKey。

3 transation.go Sign(): 详细步骤：

(1)检查这笔交易的Vin引用的交易ID是否在输入参数prevTXs中。如果不存在表示之前的过程有错误。

(2)创建副本,用于计算签名。注意这个副本输入项中Signature和Pubkey都置为空。

(3)取得每笔Vin引用的交易TX

(4)把Vin的每笔输入Signature置空，Pubkey设置为输出的公钥hash值。

(5)对整体交易副本计算hash值，得到这笔交易的hash。注意这里的hash是会变的，会随着每一笔输入重新计算生成。

(6)对得到的交易hash调用ecdsa.Sign计算签名结果r、s，组合起来得到signature。

(7)对原来的交易输入设置签名Signature。



//对交易进行签名

func (tx \*Transation) Sign(privkey ecdsa.PrivateKey, prevTXs map[string]Transation) {

//coninbase交易不用签名

if tx.isCoinBase(){

return

}

//合法性检查过程

for \_,vin := range tx.Vin{

if prevTXs[hex.EncodeToString(vin.TXid)].ID ==nil{

log.Panic("Error: Vin中引用的交易ID不存在！")

}

}

// 创建副本,用于计算签名

txcopy := tx.TrimmedCopy()

for inID,vin := range txcopy.Vin{

prevTX := prevTXs[hex.EncodeToString(vin.TXid)] //拿到前一笔交易的结构体

txcopy.Vin[inID].Signature = nil

txcopy.Vin[inID].Pubkey = prevTX.Vout[vin.Voutindex].PubkeyHash

txcopy.ID = txcopy.Hash()

r,s,err := ecdsa.Sign(rand.Reader,&privkey, txcopy.ID)

if err != nil{

log.Panic(err)

}

signature := append(r.Bytes(),s.Bytes()...)

tx.Vin[inID].Signature = signature

}

}

到这里，对输入添加了接收方私钥签名，对输出指定了接收方公钥hash，生成了完整的交易。

4 验证签名过程很深，一步步调用过程如下：

(1)首先在验证签名的入口在MineBlock()函数，

//转账操作，先生成一笔新交易， 存入挖矿所得的新区块中  
func (cli \*CLI) send(from, to string, amount int){  
 tx := NewUTXOTransation(from,to,amount,cli.bc) //会进行交易签名  
 cli.bc.MineBlock([]\*Transation{tx}) //会验证交易签名  
 fmt.Printf("send success!\n")  
}

(2) blockchain.go: MineBlock() 校验每一笔新交易

*//先检查输入的交易签名是否正确* **for** \_,tx := **range** transations {  
 **if** bc.VerifyTransation(tx) == ***false*** {  
 log.Panic(**"BlockChain.MineBlock() : ERROR: Invalid transation!"**)  
 } **else** {  
 fmt.Println(**"BlockChain.MineBlock() :transation verify success!"**)  
 }  
 }

(3) blockchain.go: VerifyTransation() 找出所有的输入引用的交易，作为参数传递给真正校验函数Transation.Verify()

(4) transation.go: Verify() 检查输入参数中的每一笔交易是否合法，具体步骤如下：

a) coinbase交易不用检验，直接跳过。

b) 校验交易会破坏这个交易内容，克隆一个交易副本

c) 开始对交易副本的每一笔输入vin进行处理，清空Signature，设置Pubkey为接收方方公钥hash。

d) 计算副本的Hash值—ID

e) 读取正本输入项的签名Signature，分解成r、s，

f) 读取正本输入项的公钥Pubkey，分解成x、y坐标，

g) 通过x、y坐标恢复公钥数据结构体rawPubkey

h) 使用rawPubkey检验r,s,ID是否一致。 不一致说明签名验证失败，交易非法。一致说明签名验证通过，这笔Vin合法有效。

校验算法的核心奥妙在于每一步中的副本Hash值-ID计算方法和结果与签名过程中的完全相同，这样才能保证对同一个ID数据进行签名和校验。

