Tedu.cn
达内教育

使用Go语言实现数字货币系统

Unit01 数据模型和工作量证明

Contents目录

01 Hash算法

1.Hash算法

1.1、Hash算法概念

Hash算法，又叫散列算法，是将任意长度的二进制值映射为较短的固定长度的二进制值，这个小的二进制值称为哈希值。简单的理解，就是把一段交易信息转换成一个固定长度的字符串。

1.2、Hash算法的特点：

1. 无法从一个哈希值恢复原始数据，也就是不可逆性。
2. 对于特定数据只有一个唯一的哈希，也就是唯一性。
3. 数据的微小变化都会使得哈希完全不同，也就是不可推断性。

Hash算法

1.3、Hash算法在安全领域的应用

1 文件校验

2 数字签名

1.4、哈希算法的种类

比较常用的有MD5、SHA1、SHA256等。SHA(Secure Hash Algorithm)代表安全哈希算法。SHA-1和SHA-2是该算法不同的两个版本。SHA-1是160位，SHA-2有不同的位数，其中最受欢迎的是256位。

2004年8月17日，在美国加州圣芭芭拉召开的国际密码大会上，山东大学王小云教授在国际会议上首次宣布了她及她的研究小组的研究成果——对MD5、HAVAL - 128、MD4和RIPEMD四个著名密码算法的破译结果。次年二月宣布破解SHA-1密码。

02 简单的区块模型

2.简单的区块模型

2.1 区块的描述

区块链是由区块顺序相连构成。所以，在区块链中存储有效信息的是区块。例如，比特币区块存储了比特币的交易信息。除此之外，区块还包含了一些技术信息，区块的版本、当前时间戳和前一个区块的哈希。

2.2 实现步骤

1> 定义区块类型

```
type Block struct {
    Timestamp     int64    //当前的时间戳，区块创建的时间
    Data          []byte   //区块存储的有效信息，(在数字货币中为交易信息)
    PrevBlockHash []byte //前一个区块的哈希(用于形成区块链)
    Hash          []byte   //当前区块的哈希(比特币的区块并不存储当前区块哈希)
}
```

2.简单的区块模型

2>添加区块类型的方法SetHash()，计算区块Hash

```
func (b *Block) SetHash()
```

3>添加创建新区块的函数

```
func NewBlock(data string, prevBlockHash []byte) *Block
```

03 简单的区块链模型

3.简单的区块链模型

3.1 区块链的描述

区块链是由区块顺序相连构成。所以，在区块链中存储有效信息的是区块。例如，比特币区块存储了比特币的交易信息。除此之外，区块还包含了一些技术信息，区块的版本、当前时间戳和前一个区块的哈希。

3.2 区块链的实现步骤

1> 添加区块链类型

```
type Blockchain struct {
    blocks []*Block // 切片保存多个块地址
}
```

3.简单的区块链模型

2>为区块链类型添加方法AddBlock

func (bc *Blockchain) AddBlock(data string)

3>添加一个创建创世块的函数

func NewBlock(data string, prevBlockHash []byte) *Block

4>添加一个创建包含创世块的区块链的函数

func NewBlockchain() *Blockchain

5> 在main函数中，调用相关函数，检查是否可以如期工作

04 工作量证明

4. 工作量证明

4.1 工作量证明的描述

比特币是通过工作量证明来竞争记账权，并获得比特币奖励。简单来讲就是谁能够根据区块数据更快的计算得到满足条件的哈希值，谁就可以胜出，这个块才会被添加到区块链中。我们把这个过程称为挖矿。

4.2 工作量证明算法

1>获取区块头数据data

2>为区块头数据data增加一个计数器counter，计数器初始值为零

3>将区块头数据data和计数器counter组合到一起计算哈希。

4>检查哈希是否符合一定条件。符合条件，结束；否则，计数器加一，重新执行步骤3~4

4.工作量证明

4.3 工作量证明算法的说明

哈希的条件根据具体要求而定。例如要求一个哈希的前 20 位必须是 0，要求的位数越多，难度会越大。在比特币中，这个要求会随着时间而不断变化。因为按照设计，必须保证每 10 分钟生成一个区块，而不论计算能力会随着时间增长，或者是会有越来越多的矿工进入网络，所以需要动态调整这个必要条件。

4.4 工作量证明的实现

1> 定义挖矿的难度值

const targetBits = 16 //哈希的前16位为零，用十六进制表示前4位为零

2> 定义工作量证明类型

```
type ProofOfWork struct {
    block  *Block   //区块的指针
    target *big.Int //目标数值的指针
}
```

4.工作量证明

3> 添加一个工具类函数用于将int64类型的整数转化为byte[]

```
func IntToHex(num int64) []byte
```

4> 为工作量证明类型添加方法

```
// 1. 准备要计算的数据
func (pow *ProofOfWork) prepareData(nonce int) []byte
// 2. 运行一个工作量证明函数，得到块哈希和计数器数值
func (pow *ProofOfWork) Run() (int, []byte)
// 3.验证工作量证明方法
func (pow *ProofOfWork) Validate() bool
```

4.工作量证明

5> 添加一个创建工作量证明对象的函数

func NewProofOfWork(b *Block) *ProofOfWork

6> 修改区块Block类型，增加Nonce字段

7> 修改 NewBlock 函数，移除SetHash，换成工作量证明

Unit02 持久化和命令行接口

Contents目录

01 数据库选择和设计

1.数据库选择和设计

1.1 数据库的选择

在比特币原始论文 中，并没有提到要使用哪一个具体的数据库，它完全取决于开发者如何选择。 Bitcoin Core ， 最初由中本聪发布，现在是比特币的一个参考实现，它使用的是 LevelDB。

我们将要使用的是BoltDB。Bolt DB是一个纯键值存储的 Go 数据库。没有具体的数据类型，键和值都是字节序列，并且仅关注值的获取和设置。

BoltDB的特点：

1>非常简单和简约

2>用 Go 实现

3>不需要运行一个服务器

4>能够允许我们构造想要的数据结构

Hash算法

1.2 数据库的设计

Bitcoin Core 使用两个 “bucket” 来存储数据：

1> blocks，它存储了描述一条链中所有块的元数据

2>chainstate，存储了一条链的状态，也就是当前所有的未花费的交易输出，和一些元数据

因为目前还没有交易，所以我们只需要 blocks bucket.

[blockhash]=blockdata

[l]=last block hash

02 区块的持久化

2.区块链的持久化

2.1 区块的序列化

在 BoltDB 中，键和值只能是 []byte 类型，但是我们想要存储 Block 结构。所以，我们需要使用 encoding/gob 来对这些结构进行序列化。

实现步骤：

1> 为Block类型添加Serialize 方法

```
func (b *Block) Serialize() []byte
```

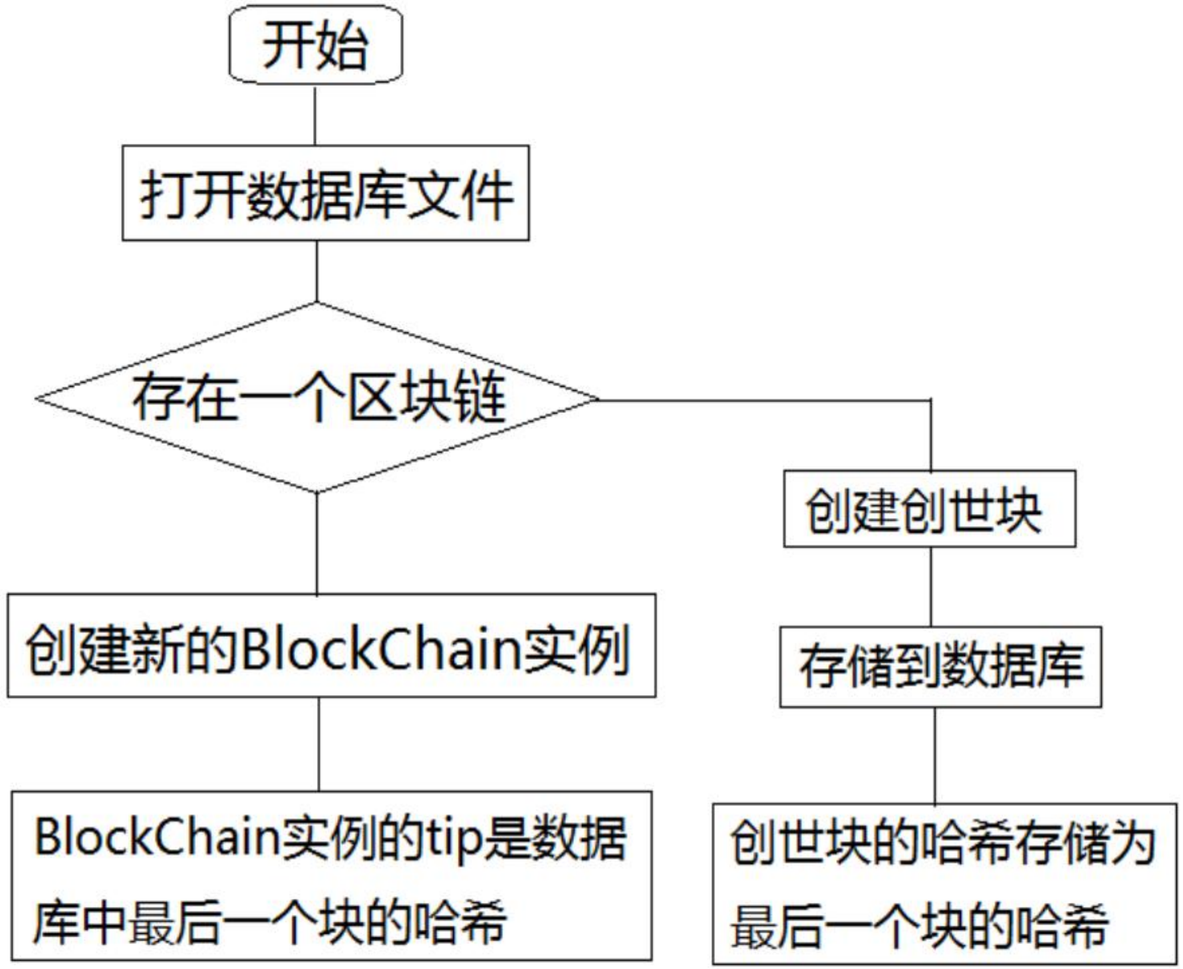
2> 添加一个DeserializeBlock函数

```
func DeserializeBlock(d []byte) *Block
```

2.区块链的持久化

2.2 区块的持久化

1>修改NewBlockchain函数，流程图如下：



2.区块链的持久化

2>修改区块链结构体类型

```
type Blockchain struct {
    tip []byte
    db  *bolt.DB
}
```

3> 修改AddBlock函数，向数据库中添加新块

func (bc *Blockchain) AddBlock(data string)

03 区块的遍历

3.区块的遍历

3.1 区块链的遍历

1> 创建区块链迭代器类型

```
type BlockchainIterator struct {
    currentHash []byte
    db          *bolt.DB
}
```

2> 为迭代器添加Next()方法

func (i *BlockchainIterator) Next() *Block

3.区块链的遍历

3>添加一个创建创世块的函数

func NewBlock(data string, prevBlockHash []byte) *Block

4>添加一个创建包含创世块的区块链的函数

func NewBlockchain() *Blockchain

5> 在main函数中，调用相关函数，检查是否可以如期工作

04 命令行接口

4. 命令行接口

4.1 要实现的命令

```
blockchain_go addblock "Pay 0.031337 for a coffee"
blockchain_go printchain
```

4.2 flag包简单用法

1>使用flag.NewFlagSet创建子命令

2>为子命令添加参数

3>解析子命令参数

4>使用参数

4.工作量证明

4.3 命令行接口的实现

1> 定义命令行类型CLI

2> 为命令类型CLI添加方法

```
func (cli *CLI) Run()
func (cli *CLI) printUsage()
func (cli *CLI) validateArgs()
func (cli *CLI) addBlock()
func (cli *CLI) printChain()
```

4.4 在main函数中使用

Unit03 交易

Contents目录

01 交易的结构

1.交易的结构

在区块链中，交易一旦被创建，就没有任何人能够再去修改或是删除它。

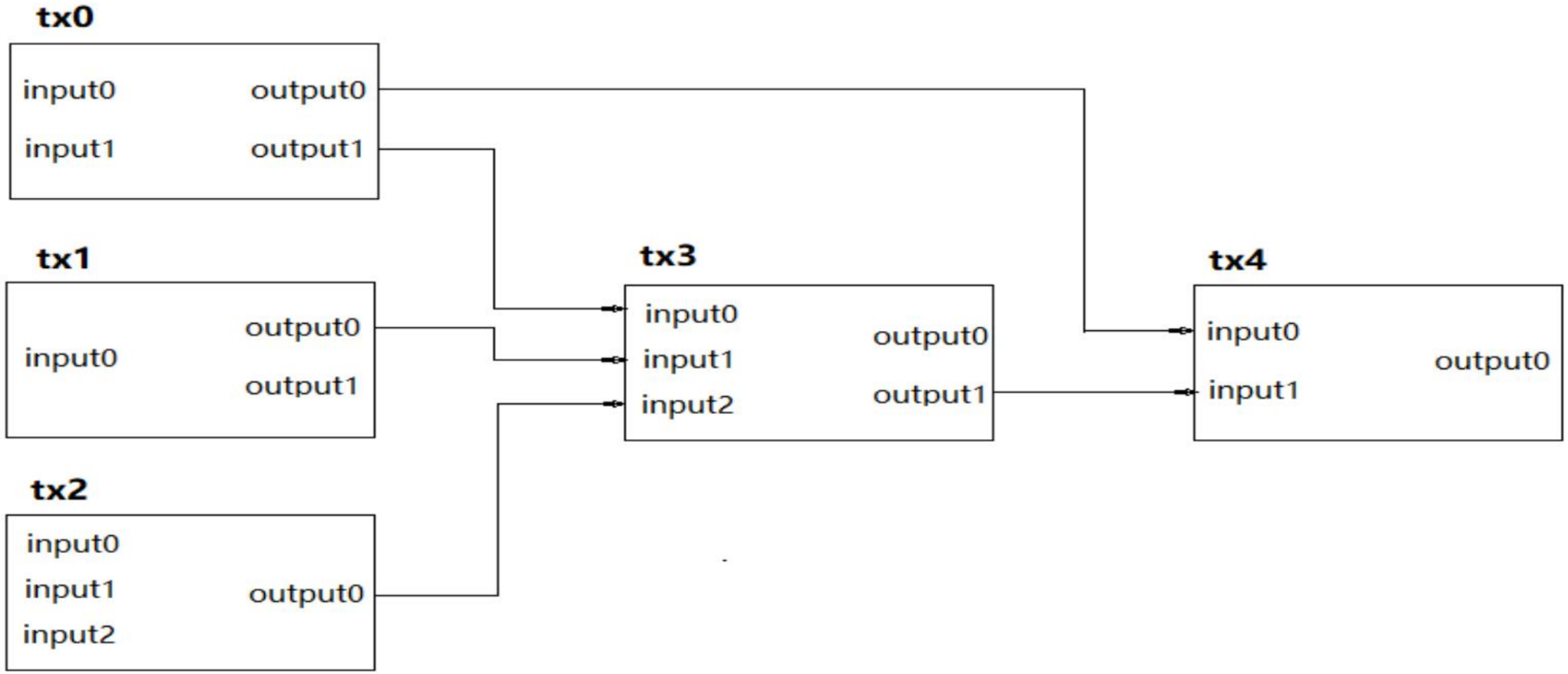
1.1 交易的结构

比特币的一笔交易由一些输入（input）和输出（output）组合而来

```
type Transaction struct {
    ID   []byte       //交易的 ID ，交易的 Hash 值
    Vin  []TXInput    //输入的集合
    Vout []TXOutput   //输出的集合
}
```

1.交易的结构

交易结构示意图



1.交易的结构

注意的问题：

1.一笔交易的输入可以引用之前多笔交易的输出。例如，tx3引入了tx0，tx1，tx2。

2.一个输入必须引用一个输出。

3.有一些输出并没有被关联到某个输入上，这些输出叫做未花费交易输出。

4.交易输出不可再分(找零机制)

在一笔交易中不会出现同一个地址的两个输出

1.交易的结构

1.2 交易输出

```
type TXOutput struct {
    Value        int    //存储了“币”
    ScriptPubKey string //存储了一个数学难题，这个数学难题用来对输出进行锁定
}
```

1.3 交易输入(完全引用了输出)

```
type TXInput struct {
    Txid      []byte //一个输入引用了之前一笔交易的一个输出,这笔交易的 ID
    Vout      int    //在交易中的输出的索引(一个交易通常包含多个输出)
    ScriptSig string //作用于输出的数据，用于解锁输出
}
```

02 修改区块

2.修改区块

2.1 修改块结构，将Data换成交易

2.2 创币交易的函数

func NewCoinbaseTX(to, data string) *Transaction

2.3 NewBlock和NewGenesisBlock 也必须做出相应改变

2.修改区块

2.4 增加创建区块链的函数CreateBlockChain，在函数中创建数据库文件，并添加创世块数据。

2.5 修改NewBlockChain函数，通过读取数据库中的区块链中最后一个块的哈希，创建一个新的区块链对象。

2.6 修改工作量证明，首先在block.go中为Block增加HashTransactions()方法。然后，修改工作量证明的prepareData函数

03 未花费交易输出

3.未花费交易输出

3.1 未花费交易输出概念

未花费交易输出（unspent transactions outputs, UTXO）。未花费（unspent）指的是这个输出还没有被包含在任何交易的输入中，或者说没有被任何输入引用。

在交易结构示意图中，未花费的输出是：

tx1, output 1；

tx3, output 0；

tx4, output 0；

3.区块链的遍历

3.2 未花费交易输出的实现

1> 定义在输入和输出上的锁定和解锁方法

```
func (in *TXInput) CanUnlockOutputWith(unlockingData string) bool {
    return in.ScriptSig == unlockingData
}
func (out *TXOutput) CanBeUnlockedWith(unlockingData string) bool {
    return out.ScriptPubKey == unlockingData
}
```

3.区块链的遍历

2> 查找指定地址的包含未花费输出的交易

func (bc *Blockchain) FindUnspentTransactions(address string) []Transaction

3> 查找指定地址的未花费交易输出

func (bc *Blockchain) FindUTXO(address string) []TXOutput

4>查找指定地址的可花费交易输出

func (bc *BlockChain) FindSpendableOutputs(address string, amount int) (int, map[string][]int)

3.区块链的遍历

5> 在CLI结构中增加方法,获取指定地址的余额

func (cli *CLI) getBalance(address string)

6> 实现普通交易

func NewUTXOTransaction(from, to string, amount int, bc *Blockchain) *Transaction

7>查找指定地址的可花费交易输出

func (bc *BlockChain) FindSpendableOutputs(address string, amount int) (int, map[string][]int)

3.区块链的遍历

8> 修改Blockchain.MineBlock

func (bc *Blockchain) MineBlock(transactions []*Transaction)

9> 增加send方法实现交易

func (cli *CLI) send(from, to string, amount int)

10>修改遍历区块链的代码

func (cli *CLI) printChain()

11>修改cli.go文件：增加createBlockChain,修改printusage和run函数

3.区块链的遍历

8> 修改Blockchain.MineBlock

func (bc *Blockchain) MineBlock(transactions []*Transaction)

9> 增加send方法实现交易

func (cli *CLI) send(from, to string, amount int)

10>修改遍历区块链的代码

func (cli *CLI) printChain()

11>修改cli.go文件：增加createBlockChain,修改printusage和run函数

3.区块链的遍历

12> 修改main函数

func (bc *Blockchain) MineBlock(transactions []*Transaction)

```
func main() {
    cli := CLI{}
    cli.Run()
}
```

13> 修改CLI类型

```
type CLI struct {
    //bc *Blockchain
}
```

Unit04 地址

Contents目录

01 比特币地址

1.比特币地址

这就是一个真实的比特币地址：1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa。这是史上第一个比特币地址，据说属于中本聪。

比特币地址是完全公开的，如果你想要给某个人发送币，只需要知道他的地址就可以了。实际上，所谓的地址，只不过是将公钥表示成人类可读的形式而已。

02 数字签名

2.数字签名

2.1 公钥加密

公钥加密（public-key cryptography）算法使用的是成对的密钥：公钥和私钥。公钥并不是敏感信息，可以告诉其他人。但是，私钥绝对不能告诉其他人。

在加密货币的世界中，你的私钥代表的就是你，私钥就是一切。

本质上，比特币钱包也只不过是这样的密钥对而已。当你安装一个钱包应用，或是使用一个比特币客户端来生成一个新地址时，它就会为你生成一对密钥。在比特币中，谁拥有了私钥，谁就可以控制所以发送到这个公钥的币。

2.数字签名

2.2 数字签名的过程描述

发送方将交易数据的Hash使用私钥签名，将交易数据、签名后的数据和发送方的公钥发送给接收方。

接收方收到数据后，使用发送的公钥解密签名后的数据，得到签名钱的数据也就是交易数据的Hash值。然后接收方可以将交易数据也求一次Hash值。如果两个Hash值相等，表示数据没有被篡改。

而且，只要可以使用发送方的公钥解密签名后的数据，就能确定发送方。所以，发送方式不可抵赖的。

2.数字签名

2.3 椭圆曲线加密

比特币使用椭圆曲线来产生私钥和公钥。椭圆曲线是一个复杂的数学概念，我们并不打算在这里作太多解释。我们只要知道这些曲线可以生成非常大的随机数就够了。

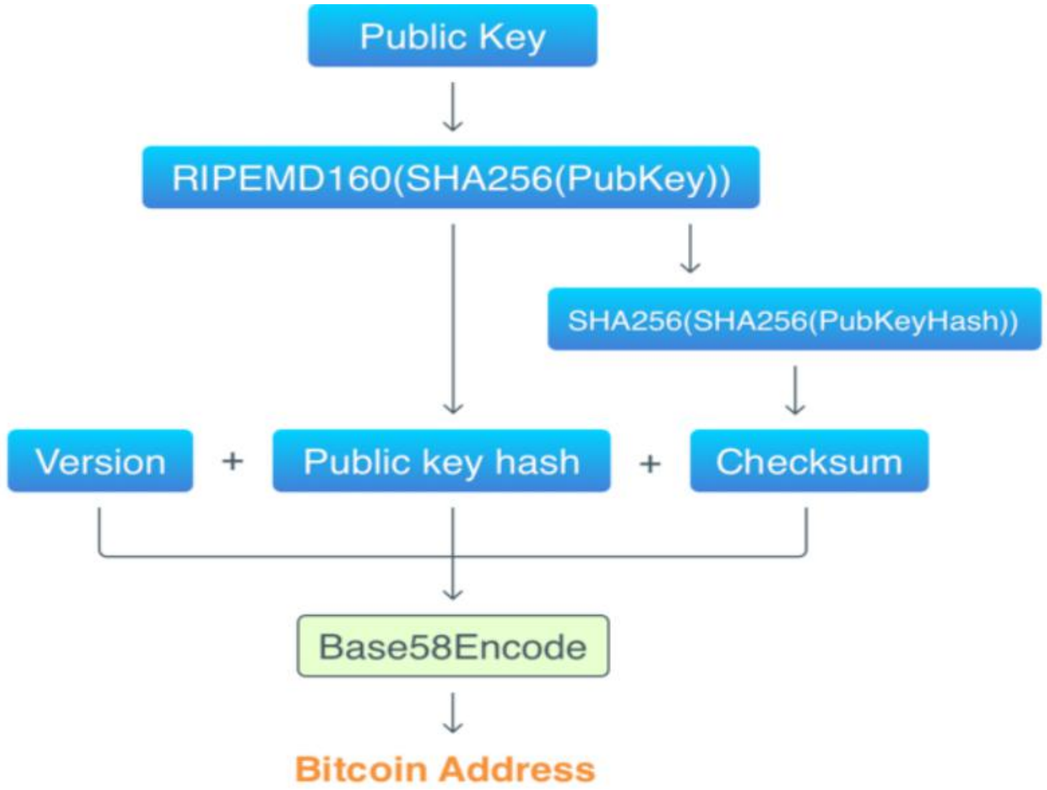
比特币使用的是 ECDSA（Elliptic Curve Digital Signature Algorithm）算法来对交易进行签名。

2.数字签名

2.4 Base58编码

比特币使用 Base58 算法将公钥转换成人类可读的形式。这个算法跟著名的 Base64 很类似，区别在于它使用了更短的字母表。

下图是从一个公钥获得一个地址的过程：



03 实现和使用地址

3.实现和使用地址

3.1 钱包包含了多个密钥对。所以，从定义钱包结构开始(在wallet.go)
3.2 修改输入和输出来使用地址
3.3 修改相关输入输入相关的函数
1>修改blockchain.go中的：
FindUnspentTransactions 一个out一个in
FindSpendableOutputs 一个out
FindUTXO 一个out
2>修改transaction.go中的：
NewCoinbaseTX out和in的构造函数的参数
NewUTXOTransaction out和in的构造函数的参数

3.简单的区块链模型

2>为区块链类型添加方法AddBlock

func (bc *Blockchain) AddBlock(data string)

3>添加一个创建创世块的函数

func NewBlock(data string, prevBlockHash []byte) *Block

4>添加一个创建包含创世块的区块链的函数

func NewBlockchain() *Blockchain

5> 在main函数中，调用相关函数，检查是否可以如期工作

04 实现签名交易

4. 实现签名交易

4.1 创建一个剪切后的交易用于签名

4.2 对交易中的每个输入进行签名

4.3 对交易进行验证

4.4 为区块链类型添加签名和验证交易方法

4.5 在创建交易时进行签名，在将交易放入区块之前进行验证

Unit05 UTXO集

Contents目录

01 挖矿奖励

1.挖矿奖励

在上一篇文章中，我们略过的一个小细节是挖矿奖励。现在，我们已经可以来完善这个细节了。挖矿奖励，实际上就是一笔 coinbase 交易。

当一个挖矿节点开始挖出一个新块时，它会将交易从队列中取出，并在前面附加一笔 coinbase 交易。coinbase 交易只有一个输出，里面包含了矿工的公钥哈希。

02 UTXO集概念

2.UTXO集概念

2.1 UTXO 集，也就是未花费交易输出的集合。引入UTXO，可以避免对整个区块链全链扫描，提高系统的运行效率。

2.2 UTXO 集的存储

在比特币中，区块被存储在blocks数据库，交易输出被存储在chainstate数据库。会回顾一下chainstate的结构：

c + 32 字节的交易哈希 -> 该笔交易的未花费交易输出记录

B + 32 字节的块哈希 -> 未花费交易输出的块哈希

在这儿，我们只考虑交易哈希 -> 该笔交易的未花费交易输出记录

03 实现UTXO集

3.实现UTXO集

3.1 添加utxoset.go文件，并在文件中，定义UTXOSet 结构体。

```
const utxoBucket = "chainstate"  //数据库中 bucket 的名称
type UTXOSet struct {
    Blockchain *Blockchain
}
```

3.2 重新构造UTXO集合

func (u UTXOSet) Reindex()

3.3 为区块链类型添加获取UTXO集的方法

func (bc *Blockchain) FindUTXO() map[string]TXOutputs

3.实现UTXO集

3.4 TXOutputs(交易输出集)类型的定义

3.5 为了实现区块数据与UTXO集数据同步，添加更新方法

func (u UTXOSet) Update(block *Block)

3.6 使用UTXO集

1> 创建区块时，调用UTXO集的Reindex方法

2> 普通交易时，调用UTXO集的Update方法

关注达内科技官方微信

谢谢观赏！