从0到1简易区块链开发手册V0.2

Author: brucefeng

Email: brucefeng@brucefeng.com

微信ID: brucefeng1991

编程语言: Golang

前言

这是我这段时间学习区块链开发以来打造的第一个区块链平台，之所以叫做简易区块链，是因为它确实比较简易，仅仅是实现了底层的一些功能，不足以作为一个真正的公链使用，但通过学习，我们能够通过代码更加理解比特币白皮书中描述的各种比特币原理，区块链世界，无论是研究理论的，还是实战开发的，甚至炒币玩资本的，都离不开比特币的影子，区块链技术毕竟是从比特币中剥离抽象而来，所以，作为一个技术人员，无论是研究以太坊，超级账本，甚至是各种公链，包括某些山寨公链，都需要先去理解比特币原理，而对于开发者而言，理解原理最好的方式就是将其通过代码实现，当然，我们这里实现的一些原理只是应用实战范围之内可以实现的，比如椭圆加密算法，我们要实现的只是使用椭圆加密去实现某些加密功能，而非用代码去实现一个完整的椭圆加密代码库，这个不再本文的讨论范围内，所以本文面向的群体是:

- 对比特币原理不了解，但没时间看太多的资料文献的初学者
- 对比特币原理有所了解，但是停留在理论阶段的研究者
- 没有对比特币进行研究，想直接研究以太坊，超级账本的实战者(大神除外)
- 对Golang熟悉，但是不知道如何入手区块链的开发者或者是像我一样的运维 :-)

本文中，我们先通过命令行的方式演示区块链的工作流程以及相关原理，涉及到比较重要的内容，比如Sha256哈希，椭圆加密，Base58编码等内容，我会根据时间以及后期的工作情况进行适当调整，这注定是一个短期内没有结尾的故事。

> 为表尊敬，写在前面，建议先阅读该文档
>
> 本文的学习资料来自这位**liuxhengxu**前辈翻译的资料
>
> 能将资料翻译得如此完美，相比其技术功能也是相当深厚的，感谢分享

建议大家可以先看该资料后再来看我的这系列文章，否则可能会有一些难度，由于该资料是通过循序渐进的方式进行版本迭代，慢慢引导开发者不断对原有的代码进行优化，拓展，非常认真并细心，希望大家时间充裕的时候以及对某些本文并未写清楚的地方，强烈建议阅读该资料。

本文在此基础上进行了一些修改(谈不上改进)，我摒弃一些过于基础的以及后期需要大量重构的代码，直接通过该项目的执行流程进行代码分析，这样可以稍微节省一些大家的时间，把有限的精力放在对业务有更大提升的技术研究上。

一. 功能描述

```
Usage:
        createwallet
                        -- 创建钱包
        getaddresslists
                        -- 获取所有的钱包地址
        createblockchain -address address
                        -- 创建创世区块
        send -from SourceAddress -to DestAddress -amount Amount
                        -- 转账交易
        printchain
                        -- 打印区块
        getbalance -address address
                        -- 查询余额
```

本文围绕着几个功能进行讲解

- 创建钱包

  通过椭圆加密算法创建钱包地址

- 获取钱包地址

  获取区块链中所有的钱包地址

- 创建创世区块

  实现创世区块的创建，并生成区块链

- 实现转账交易

  通过转账交易，生成区块，并存入区块链

- 打印区块

  打印出所有的区块信息，实现转账交易的溯源

- 查询余额

  查询出对应钱包的余额状态

随着代码的不断完善，我们将会对以上进行改进，并提供更多的功能点进行分析探讨，我们先通过下图简单演示一下如上功能

创建第一个钱包

```
$ ./mybtc  createwallet
区块链钱包不存
```

```
Author:brucefeng
Email:brucefeng@brucefeng.com
```

关于效果图，大家先大致看下即可，不需要刻意研究，在后期的课程中都会涉及。

二. 实现命令行功能

1.定义结构体

定义一个空结构体

```
type CLI struct {

}
```

2.结构体方法

> 重要！初学者必看
>
> 这里提到的结构体方法并不是真正实现功能的方法，而是命令行对象的方法，这些方法中会调用实际的功能对象方法进行功能实现,在本章节中，创建结构体方法即可，功能代码可以为空，如:

```
例子：
func (cli *CLI) CreateWallet() {
}
func (cli *CLI) GetAddressLists() {
}
.....
```

其他的可以在后期逐步实现，为了让有基础的同学对项目整体提前有些印象，所以，代码内容我直接复制粘贴进来，不做删减，在后期的内容中，会逐步涉及到每个调用的对象方法或者函数的作用。

2.1 创建钱包

```
func (cli *CLI) CreateWallet() {
    _, wallets := GetWallets() //获取钱包集合对象
    wallets.CreateNewWallets() //创建钱包集合

}
```

2.2 获取钱包地址

```
func (cli *CLI) GetAddressLists() {
    fmt.Println("钱包地址列表为:")
        //获取钱包的集合，遍历，依次输出
    _, wallets := GetWallets() //获取钱包集合对象
    for address, _ := range wallets.WalletMap {
        fmt.Printf("\t%s\n", address)
    }
}
```

2.3 创建创世区块

```
func (cli *CLI) CreateBlockChain(address string) {
    CreateBlockChainWithGenesisBlock(address)
    bc :=GetBlockChainObject()
    if bc == nil{
        fmt.Println("没有数据库")
        os.Exit(1)
    }
    defer bc.DB.Close()
    utxoSet:=&UTXOSet{bc}
    utxoSet.ResetUTXOSet()
}
```

2.4 创建转账交易

```
func (cli *CLI) Send(from, to, amount []string) {
    bc := GetBlockChainObject()
    if bc == nil {
        fmt.Println("没有BlockChain，无法转账。。")
        os.Exit(1)
    }
    defer bc.DB.Close()

    bc.MineNewBlock(from, to, amount)
    //添加更新
    utsoSet :=&UTXOSet{bc}
    utsoSet.Update()
}
```

2.5 查询余额

```
func (cli *CLI) GetBalance(address string) {
    bc := GetBlockChainObject()
    if bc == nil {
        fmt.Println("没有BlockChain，无法查询。。")
        os.Exit(1)
    }
    defer bc.DB.Close()
    //total := bc.GetBalance(address,[]*Transaction{})
    utxoSet :=&UTXOSet{bc}
    total:=utxoSet.GetBalance(address)


    fmt.Printf("%s,余额是：%d\n", address, total)
}
```

2.6 打印区块

```
func (cli *CLI) PrintChains() {
    //cli.BlockChain.PrintChains()
    bc := GetBlockChainObject() //bc{Tip,DB}
    if bc == nil {
        fmt.Println("没有BlockChain，无法打印任何数据。。")
        os.Exit(1)
    }
    defer bc.DB.Close()
    bc.PrintChains()
}
```

3. 相关函数

3.1 判断参数是否合法

```
func isValidArgs() {
    if len(os.Args) < 2 {
        printUsage()
        os.Exit(1)
    }
}
```

判断终端命令是否有参数输入，如果没有参数，则提示程序使用说明，并退出程序

3.2 程序使用说明

```
func printUsage() {
    fmt.Println("Usage:")
    fmt.Println("\tcreatewallet\n\t\t\t-- 创建钱包")
    fmt.Println("\tgetaddresslists\n\t\t\t-- 获取所有的钱包地址")
    fmt.Println("\tcreateblockchain -address address\n\t\t\t-- 创建创世区块")
    fmt.Println("\tsend -from SourceAddress -to DestAddress -amount
Amount\n\t\t\t-- 转账交易")
    fmt.Println("\tprintchain\n\t\t\t-- 打印区块")
    fmt.Println("\tgetbalance -address address\n\t\t\t-- 查询余额")
}
```

3.3 JSON解析的函数

```
func JSONToArray(jsonString string) []string {
    var arr [] string
    err := json.Unmarshal([]byte(jsonString), &arr)
    if err != nil {
        log.Panic(err)
    }
    return arr
}
```

通过该函数将JSON字符串格式转成字符串数组，用于在多笔转账交易中实现同时多个账户进行两两转账的功能。

3.4 校验地址是否有效

```
func  IsValidAddress(address []byte) bool {

    //step1：Base58解码
    //version+pubkeyHash+checksum
    full_payload := Base58Decode(address)

    //step2：获取地址中携带的checkSUm
    checkSumBytes := full_payload[len(full_payload)-addressCheckSumLen:]
```

```
    versioned_payload := full_payload[:len(full_payload)-
addressCheckSumLen]

    //step3：versioned_payload，生成一次校验码
    checkSumBytes2 := CheckSum(versioned_payload)

    //step4：比较checkSumBytes，checkSumBytes2
    return bytes.Compare(checkSumBytes, checkSumBytes2) == 0

}
```

以下三个功能实现之前需要先调用该函数进行地址校验

- 创建创世区块
- 转账交易
- 查询余额

4.命令行主要方法Run

```
Usage:
        createwallet
                        -- 创建钱包
        getaddresslists
                        -- 获取所有的钱包地址
        createblockchain -address address
                        -- 创建创世区块
        send -from SourceAddress -to DestAddress -amount Amount
                        -- 转账交易
        printchain
                        -- 打印区块
        getbalance -address address
                        -- 查询余额
```

我们将如上功能展示的实现功能写在Run方法中，实现命令行功能的关键是了解**os.Args**与**flag**

> 关于这两个功能，此处不再赘述，否则篇幅会无限臃肿。

代码块均在方法体Run中，下文将分步骤对代码实现进行体现

```
func (cli *CLI) Run() {
}
```

4.1 判断命令行参数是否合法

```
isValidArgs()
```

4.2 创建flagset命令对象

```
    createWalletCmd := flag.NewFlagSet("createwallet", flag.ExitOnError)
    getAddresslistsCmd := flag.NewFlagSet("getaddresslists",
flag.ExitOnError)
    CreateBlockChainCmd := flag.NewFlagSet("createblockchain",
flag.ExitOnError)
    sendCmd := flag.NewFlagSet("send", flag.ExitOnError)
    printChainCmd := flag.NewFlagSet("printchain", flag.ExitOnError)
    getBalanceCmd := flag.NewFlagSet("getbalance", flag.ExitOnError)
    testMethodCmd := flag.NewFlagSet("test", flag.ExitOnError)
```

如上，通过flag.NewFlagSet方法创建命令对象，如createwallet,getaddresslists,createblockchain等命令对象

固定用法，掌握即可。

4.3 设置命令后的参数对象

```
    flagCreateBlockChainData := CreateBlockChainCmd.String("address",
"GenesisBlock", "创世区块的信息")
    flagSendFromData := sendCmd.String("from", "", "转账源地址")
    flagSendToData := sendCmd.String("to", "", "转账目标地址")
    flagSendAmountData := sendCmd.String("amount", "", "转账金额")
    flagGetBalanceData := getBalanceCmd.String("address", "", "要查询余额的账
户")
```

通过命令对象的String方法为命令后的参数对象

- createblockchain命令后的参数对象：address
- send命令后的参数对象: from | to |amount
- getbalance命令后的参数对象: address

其中createwallet，getaddresslists，printchain命令没有参数对象。

4.4 解析命令对象

```
switch os.Args[1] {
    case "createwallet":
        err := createWalletCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    case "getaddresslists":
        err := getAddresslistsCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    case "createblockchain":
        err := CreateBlockChainCmd.Parse(os.Args[2:])
        if err != nil {
```

```
            log.Panic(err)
        }
    case "send":
        err := sendCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    case "getbalance":
        err := getBalanceCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    case "printchain":
        err := printChainCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    case "test":
        err := testMethodCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    default:
        printUsage()
        os.Exit(1)

    }
```

匹配对应的命令，用命令对象的Parse方法对**os.Args[2:]**进行解析。

4.5 执行对应功能

```
    //4.1 创建钱包--->交易地址
    if createWalletCmd.Parsed() {
        cli.CreateWallet()
    }
    //4.2 获取钱包地址
    if getAddresslistsCmd.Parsed() {
        cli.GetAddressLists()
    }
    //4.3 创建创世区块
    if CreateBlockChainCmd.Parsed() {
        if !IsValidAddress([]byte(*flagCreateBlockChainData)) {
            fmt.Println("地址无效，无法创建创世前区块")
            printUsage()
            os.Exit(1)
        }
        cli.CreateBlockChain(*flagCreateBlockChainData)
    }
```

```
    //4.4 转账交易
    if sendCmd.Parsed() {
        if *flagSendFromData == "" || *flagSendToData == "" ||
*flagSendAmountData == "" {
            fmt.Println("转账信息有误")
            printUsage()
            os.Exit(1)
        }
        //添加区块
        from := JSONToArray(*flagSendFromData)     //[]string
        to := JSONToArray(*flagSendToData)         //[]string
        amount := JSONToArray(*flagSendAmountData) //[]string
        for i := 0; i < len(from); i++ {
            if !IsValidAddress([]byte(from[i])) ||
!IsValidAddress([]byte(to[i])) {
                fmt.Println("地址无效，无法转账")
                printUsage()
                os.Exit(1)
            }
        }

        cli.Send(from, to, amount)
    }
    //4.5 查询余额
    if getBalanceCmd.Parsed() {
        if !IsValidAddress([]byte(*flagGetBalanceData)) {
            fmt.Println("查询地址有误")
            printUsage()
            os.Exit(1)
        }
        cli.GetBalance(*flagGetBalanceData)
    }

    //4.6 打印区块信息
    if printChainCmd.Parsed() {
        cli.PrintChains()
    }
```

5. 测试代码

> 在main.go中中添加测试代码

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

编译运行

```
$ go build -o mybtc main.go
```

> 测试思路
>
> 1. 查看命令行列表是否可以正常显示
> 2. 输入非法字符查看是否有错误提示
>
> 业务功能此处暂未实现，测试时忽略。

下一篇文章将介绍如何实现钱包/地址的生成功能，即**创建钱包**。

三.创建钱包

1.概念

创建钱包其实就是创建比特币地址，在比特币世界中，没有账户概念，不需要也不会在任何地方存储个人数据（比如姓名，身份证件号码等）。但是，我们总要有某种途径识别出你是交易输出的所有者（也就是说，你拥有在这些输出上锁定的币），这就是比特币地址（address）需要完成的使命。

关于钱包这个概念，我个人觉得imtoken在用户引导那部分写得很清楚,此处将链接给到大家,有兴趣的可以去看看

https://www.cnblogs.com/fangbei/p/imToken-clearance.html

我们来看一下一个真实的比特币账户，1FSzfZ27CVTkfNw6TWxnHPaRLRCgpWvbFC ，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。但是，地址（尽管地址也是独一无二的）并不是用来证明你是一个“钱包”所有者的信物。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，因为原生的公钥人类很难阅读。在比特币中，你的身份（identity）就是一对（或者多对）保存在你的电脑（或者你能够获取到的地方）上的公钥（public key）和私钥（private key）。比特币基于一些加密算法的组合来创建这些密钥，并且保证了在这个世界上没有其他人能够取走你的币，除非拿到你的密钥。

关于如何创建一个钱包以及钱包集合，通过下图进行简单展示


创建钱包
Author:brucefeng
Email:brucefeng@brucefeng.com
钱包 wallet.go
1. 结构体定义
名称
Wallet
属性
私钥 PrivateKey
公钥 PublickKey
2. 【函数】生成一对秘钥 newKeyPair
1. 通过椭圆加密算法，得到一个椭圆曲线值
2. 生成私钥
3. 通过私钥生成公钥
将返回的私钥与公钥赋值给钱包对象
3. 【函数】创建钱包对象 NewWallet
定义两个常量
1. version = byte(0x00)
2. addressCheckSumLen = 4
4. 【方法】根据公钥获取地址GetAddress
【函数】1. 得到公钥哈希 PubKeyHash
原始公钥-->公钥哈希
1. sha256 加密
2. ripemd160 加密
【函数】2. 获取地址 GetAddressByPubKeyHash
公钥哈希-->地址address
1. 添加版本号 拼接
[]byte{version}
pubKeyHash
【函数】2. 获取CheckSum
1. 第一次sha256加密
2. 第二次sha256加密
3. 返回第二次Hash后的前四位
3. 拼接全部数据
拼接后的版本号
CheckSum
定义变量编码集
【函数】4. 对拼接数据进行Base58编码
Base58Encode，返回address
关联函数 ReverseBytes 字节数组反转
解密函数 Base58Decode
钱包集合 wallets.go
定义一个常量 存储钱包数据的本地文件
1. 结构体定义
名称
Wallets
属性
WalletMap map[string]*Wallet
【方法】2.创建钱包集合
1. 创建钱包对象
2. 获取钱包地址
3. 将钱包地址存入钱包集合
【方法】4. 将钱包集合本地化存储 SaveFile
1. 序列化Wallets对象
2. 将数据存储到文件中
【函数】2. 获取钱包集合 GetWallets
1. 钱包文件不存在
创建一个空的钱包集合对象
2. 钱包文件已存在
1. 读取本地的钱包文件中的数据
2. 注册接口 elliptic.P256()
3. 将数据反序列化成钱包集合对象

图 创建钱包与钱包集合

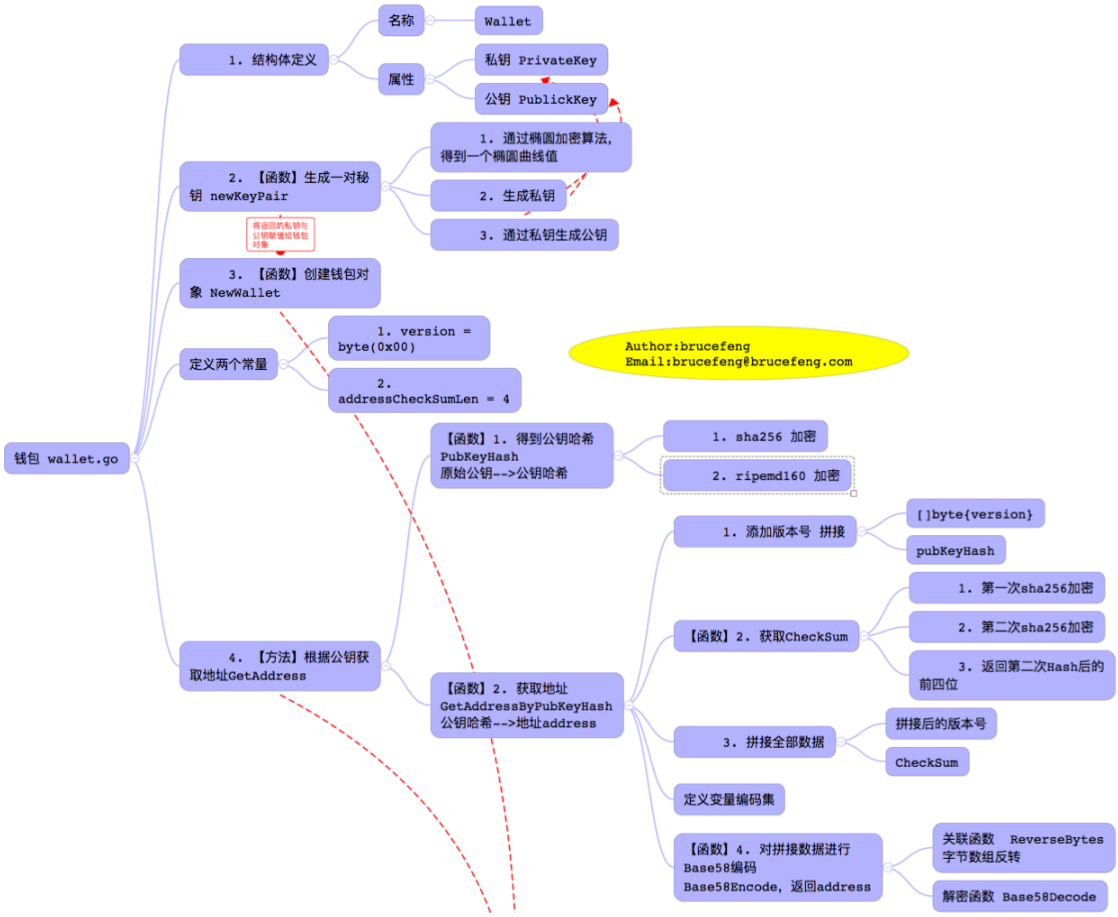


图 创建钱包wallet

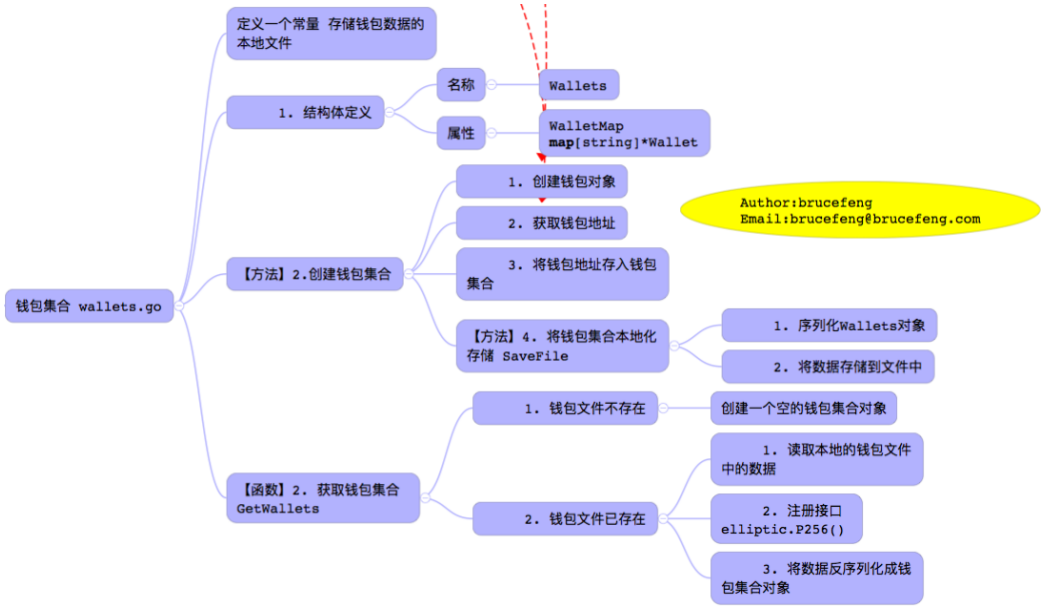


图 创建钱包集合

2. 定义钱包结构体

```
type Wallet struct {
    //1.私钥
    PrivateKey ecdsa.PrivateKey
    //2.公钥
    PublickKey []byte //原始公钥
}
```

定义钱包Wallet的属性为私钥：PrivateKey，类型为系统内置的结构体对象ecdsa.PrivateKey，公钥：PublickKey，类型为字节数组

3. 生成钱包地址

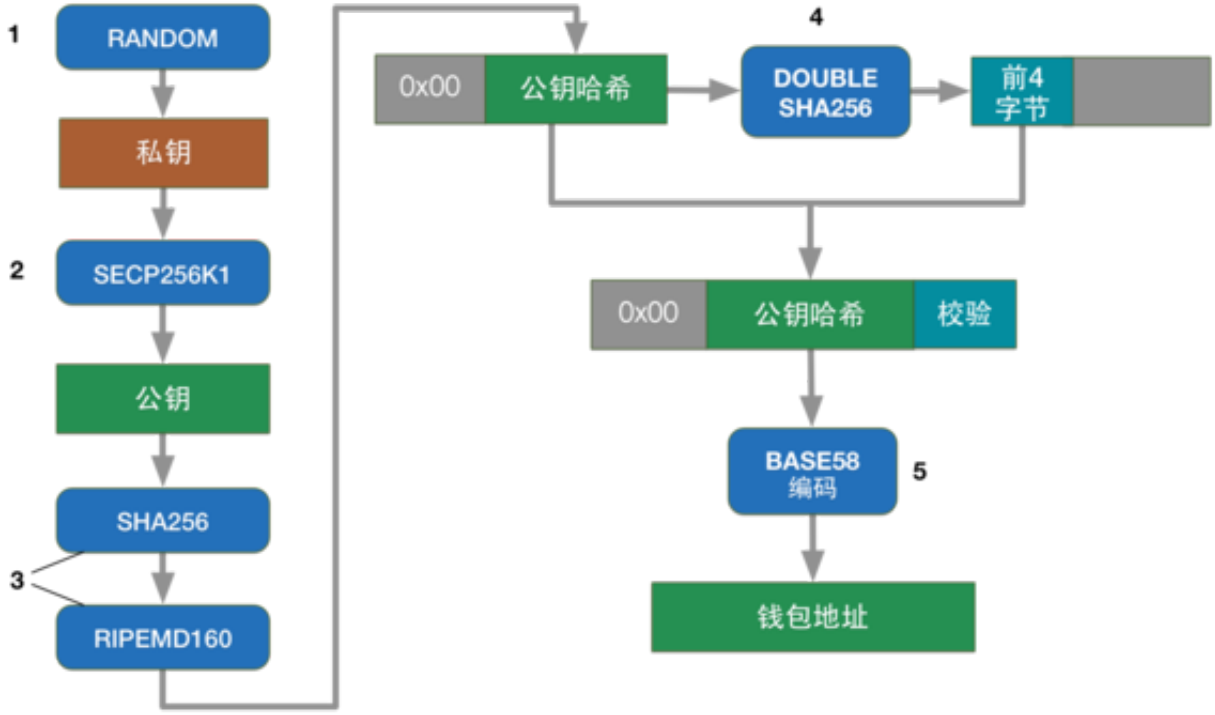


图 从私钥到生成钱包地址的过程图

3.1 通过椭圆曲线算法产生密钥对

```
func newKeyPair() (ecdsa.PrivateKey, []byte) {
    //椭圆加密
    curve := elliptic.P256() //根据椭圆加密算法，得到一个椭圆曲线值
    //生成私钥
    privateKey, err := ecdsa.GenerateKey(curve, rand.Reader) //*Private
    if err != nil {
        log.Panic(err)
    }

    //通过私钥生成原始公钥
    publicKey := append(privateKey.PublicKey.X.Bytes(),
privateKey.PublicKey.Y.Bytes()...)
    return *privateKey, publicKey
```

```
}
```

椭圆曲线加密：(ECC：ellipse curve Cryptography)，非对称加密

- 根据椭圆曲线算法，产生随机私钥
- 根据私钥，产生公钥

3.2 创建钱包对象

```
func NewWallet() *Wallet {
    privateKey, publicKey := newKeyPair()
    return &Wallet{privateKey, publicKey}
}
```

通过newKeyPair函数将返回的私钥与公钥生成钱包对象Wallet

3.3 定义常量值

```
const version = byte(0x00)
const addressCheckSumLen = 4
```

- version: 版本前缀，比特币中固定为0
- addressCheckSumLen: 用于获取校验码的长度变量，取添加版本+数据进行两次SHA256之后的前4个字节

3.4 根据公钥获取地址

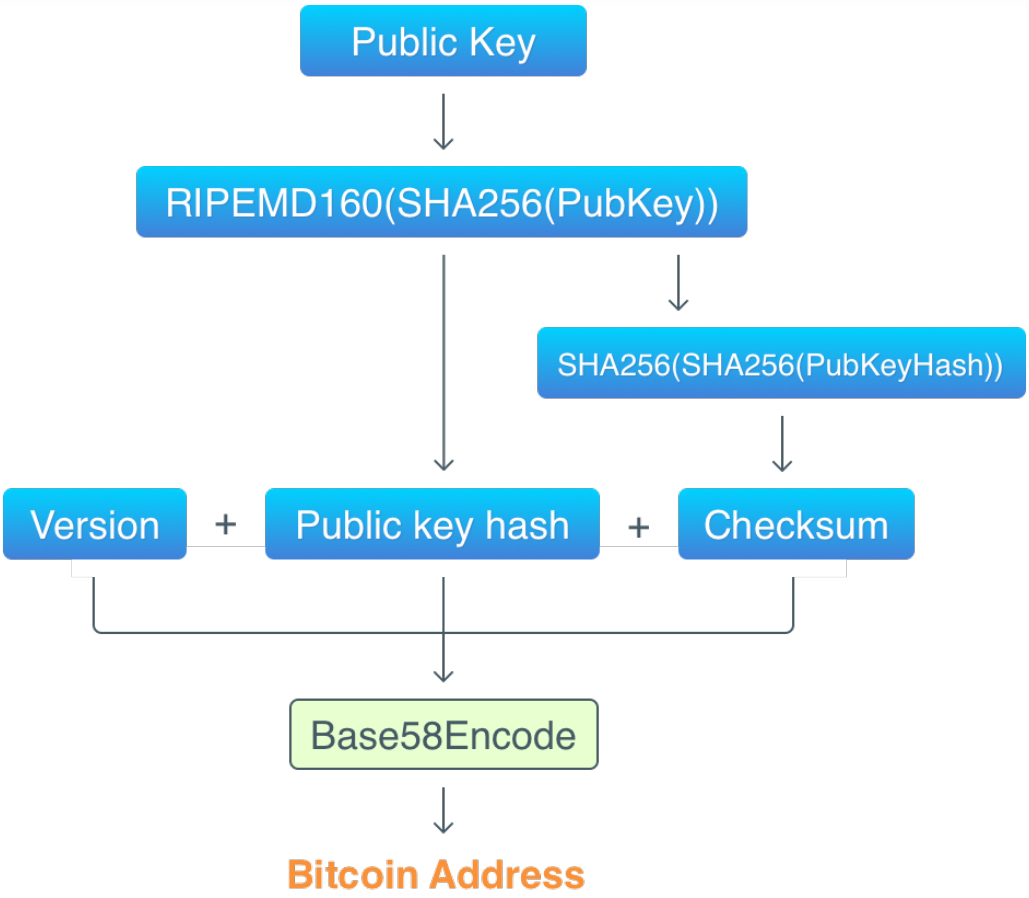


图 从**公钥**到生成钱包地址的过程图

```
func PubKeyHash(publickKey []byte) []byte {
    //1.sha256
    hasher := sha256.New()
    hasher.Write(publickKey)
    hash1 := hasher.Sum(nil)

    //2.ripemd160
    hasher2 := ripemd160.New()
    hasher2.Write(hash1)
    hash2 := hasher2.Sum(nil)

    //3.返回公钥哈希
    return hash2
}
```

通过公钥生成公钥哈希的步骤已完成。

```
func GetAddressByPubKeyHash(pubKeyHash []byte) []byte {
    //添加版本号:
    versioned_payload := append([]byte{version}, pubKeyHash...)

    //根据versioned_payload-->两次sha256,取前4位，得到checkSum
```

```
    checkSumBytes := CheckSum(versioned_payload)

    //拼接全部数据
    full_payload := append(versioned_payload, checkSumBytes...)

    //Base58编码
    address := Base58Encode(full_payload)
    return address
}
```

> 相关函数如下

- 生成校验码

```
func CheckSum(payload [] byte) []byte {
    firstHash := sha256.Sum256(payload)
    secondHash := sha256.Sum256(firstHash[:])
    return secondHash[:addressCheckSumLen]
}
```

通过两次sha256哈希得到校验码，返回校验码前四位

- 字节数组转Base58加密

```
var b58Alphabet =
[]byte("123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz")
func Base58Encode(input []byte)[]byte{
    var result [] byte
    x := big.NewInt(0).SetBytes(input)

    base :=big.NewInt(int64(len(b58Alphabet)))
    zero:=big.NewInt(0)
    mod:= &big.Int{}
    for x.Cmp(zero) !=0{
        x.DivMod(x,base,mod)
        result = append(result,b58Alphabet[mod.Int64()])
    }
    ReverseBytes(result)
    for b:=range input{
        if b == 0x00{
            result = append([]byte{b58Alphabet[0]},result...)
        }else {
            break
        }
    }

    return result

}
```

以上功能函数定义好之后，定义Wallet的方法GetAddress返回钱包address

```
func (w *Wallet) GetAddress() []byte {
    pubKeyHash := PubKeyHash(w.PublickKey)
    address := GetAddressByPubKeyHash(pubKeyHash)
    return address
}
```

至此，我们已经能够生成一个比特币地址了，可以通过https://www.blockchain.com/explorer进行钱包地址查看余额。

4.定义钱包集合结构体

```
type Wallets struct {
    WalletMap map[string]*Wallet
}
```

定义钱包集合结构体Wallets，属性为WalletMap,类型为Wallet集合

5.创建钱包集合

```
func (ws *Wallets) CreateNewWallets() {
    wallet := NewWallet()
    var address []byte
    address = wallet.GetAddress()
    fmt.Printf("创建的钱包地址：%s\n", address)
    ws.WalletMap[string(address)] = wallet
    //将钱包集合存入到本地文件中
    ws.SaveFile()
}
```

- 创建一个钱包对象
- 通过GetAddress获取钱包对象的地址
- 将钱包地址作为钱包集合的key,钱包对象作为value存储至钱包集合中
- 通过SaveFile将钱包集合存入到本地文件中

5.1 定义常量存储钱包数据

```
const walletsFile = "Wallets.dat" //存储钱包数据的本地文件名
```

5.2 本地化存储钱包对象

```
func (ws *Wallets) SaveFile() {
    //1.将ws对象的数据--->byte[]
    var buf bytes.Buffer
    //序列化的过程中：被序列化的对象中包含了接口，那么该接口需要注册
```

```
    gob.Register(elliptic.P256()) //Curve
    encoder := gob.NewEncoder(&buf)
    err := encoder.Encode(ws)
    if err != nil {
        log.Panic(err)
    }
    wsBytes := buf.Bytes()

    //2.将数据存储到文件中
    err = ioutil.WriteFile(walletsFile, wsBytes, 0644)
    if err != nil {
        log.Panic(err)
    }
}
```

6.获取钱包集合

此处我们提供一个函数，用户获取钱包集合

- 读取本地的钱包文件，如果文件存在，直接获取
- 如果文件不存在，创建并返回一个空的钱包对象

```
func GetWallets()  *Wallets {
    //钱包文件不存在
    if _, err := os.Stat(walletsFile); os.IsNotExist(err) {
        fmt.Println("区块链钱包不存在")
        //创建钱包集合
        wallets := &Wallets{}
        wallets.WalletMap = make(map[string]*Wallet)
        return wallets
    }

    //钱包文件存在
    //读取本地的钱包文件中的数据
    wsBytes, err := ioutil.ReadFile(walletsFile)
    if err != nil {
        log.Panic(err)
    }
    gob.Register(elliptic.P256()) //Curve
    //将数据反序列化变成钱包集合对象
    var wallets Wallets
    reader := bytes.NewReader(wsBytes)
    decoder := gob.NewDecoder(reader)
    err = decoder.Decode(&wallets)
    if err != nil {
        log.Panic(err)
    }
    return  &wallets
}
```

7.命令行中调用

7.1 创建钱包

回到上一章节(二.实现命令行功能-2.1创建钱包)的命令行功能

```
func (cli *CLI) GetAddressLists() {
    fmt.Println("钱包地址列表为:")
        //获取钱包的集合，遍历，依次输出
    _, wallets := GetWallets() //获取钱包集合对象
    for address, _ := range wallets.WalletMap {
        fmt.Printf("\t%s\n", address)
    }
}
```

此时进行编译运行

```
$ go build -o mybtc main.go
```

```
$ ./mybtc createwallet //创建第一个钱包
$ ./mybtc createwallet //创建第二个钱包
$ ./mybtc createwallet //创建第三个钱包
```

返回的结果:

```
创建的钱包地址：14A1b3Lp3hL5B7vZvT2UWk1W78m2Kh8MUB
创建的钱包地址：1G3SkYAJdWy5pd1hFpcciUoJi8zy8PdV11
创建的钱包地址：1AA2fyYdXCQMwLMu5NBvq7Fb9UiHqg2cQV
```

7.2 获取钱包地址

回到上一章节(二.实现命令行功能-2.2 获取钱包地址)的命令行功能

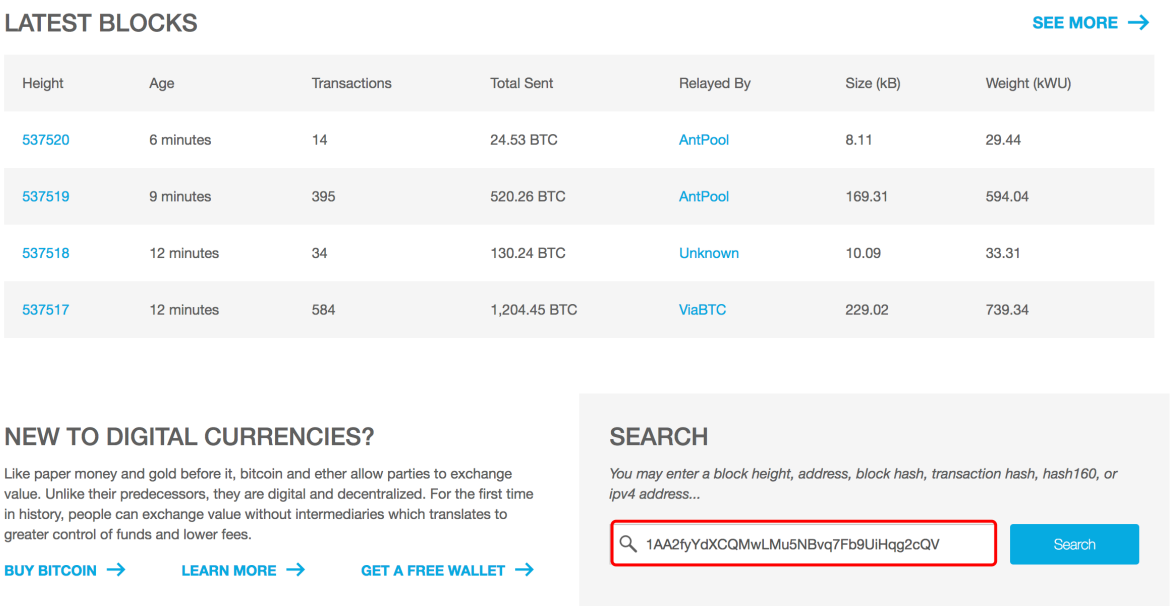
```
func (cli *CLI) GetAddressLists() {
    fmt.Println("钱包地址列表为:")
    //获取钱包的集合，遍历，依次输出
    wallets := GetWallets()
    for address, _ := range wallets.WalletMap {

        fmt.Printf("\t%s\n", address)
    }
}
```

```
$ ./mybtc getaddresslists
```

返回的结果

```
钱包地址列表为：
        1AA2fyYdXCQMwLMu5NBvq7Fb9UiHqg2cQV
        14A1b3Lp3hL5B7vZvT2UWk1W78m2Kh8MUB
        1G3SkYAJdWy5pd1hFpcciUoJi8zy8PdV11
```

上面我们提到生成的比特币地址可以通过https://www.blockchain.com/explorer进行钱包地址查看余额，现在我们来进行简单的查看验证，查看该地址:**1AA2fyYdXCQMwLMu5NBvq7Fb9UiHqg2cQV**

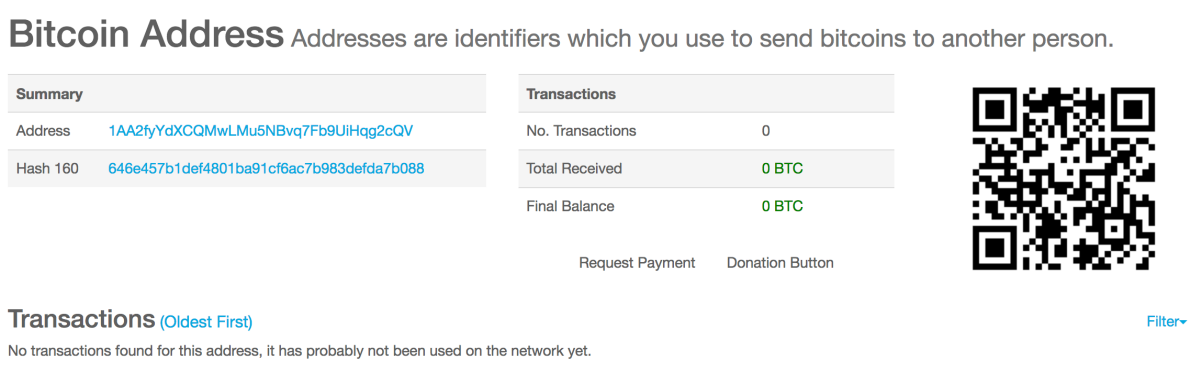


LATEST BLOCKS

SEE MORE →

Height	Age	Transactions	Total Sent	Relayed By	Size (kB)	Weight (kWU)
537520	6 minutes	14	24.53 BTC	AntPool	8.11	29.44
537519	9 minutes	395	520.26 BTC	AntPool	169.31	594.04
537518	12 minutes	34	130.24 BTC	Unknown	10.09	33.31
537517	12 minutes	584	1,204.45 BTC	ViaBTC	229.02	739.34

图 通过搜索框进行地址搜索



Bitcoin Address Addresses are identifiers which you use to send bitcoins to another person.

Summary	
Address	1AA2fyYdXCQMwLMu5NBvq7Fb9UiHqg2cQV
Hash 160	646e457b1def4801ba91cf6ac7b983defda7b088

Transactions	
No. Transactions	0
Total Received	0 BTC
Final Balance	0 BTC

Request Payment Donation Button

Transactions (Oldest First) Filter

No transactions found for this address, it has probably not been used on the network yet.

图 钱包地址详情

> 如果修改钱包地址的某个字符，如将随后的V改为X

1AA2fyYdXCQMwLMu5NBvq7Fb9UiHqg2cQV === >
1AA2fyYdXCQMwLMu5NBvq7Fb9UiHqg2cQ**X**

Oops! We couldn’t find what you are looking for.

Unrecognized search pattern. Please try searching for a transaction by entering a block height, address, block hash, transaction hash, hash 160 or ipv4 address. Or select from the general topics below.

Address / ip / SHA hash