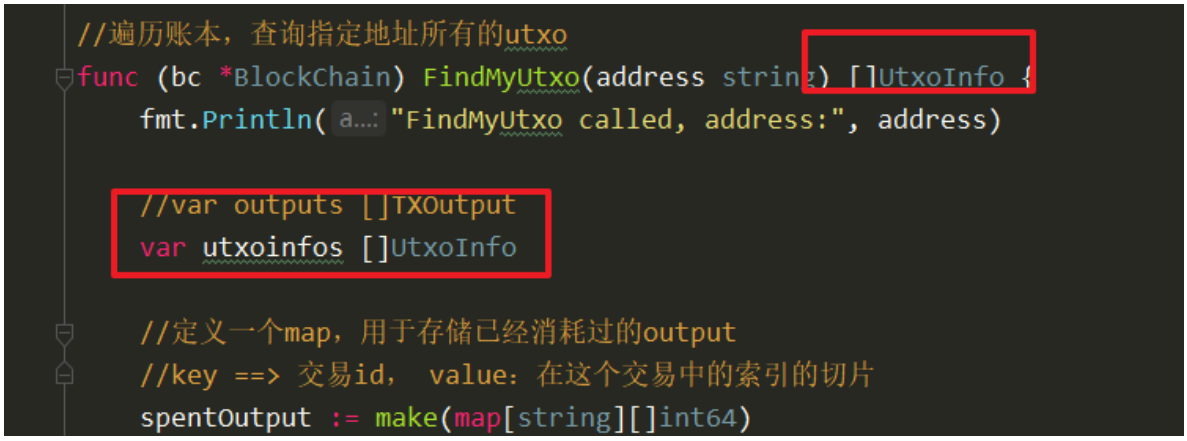
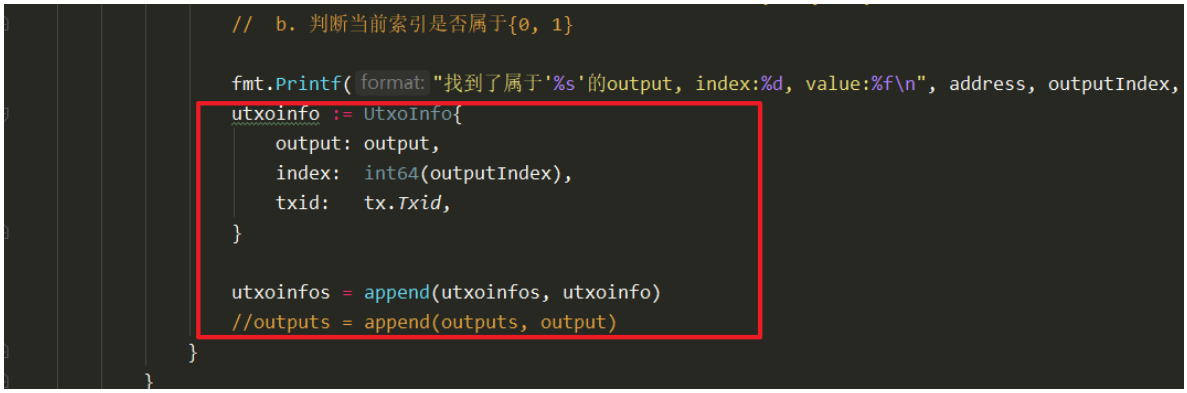
1. 使用UTXOInfo改写程序

```
//定义一个结构，同时包含output和它的位置信息
type UtxoInfo struct {
    output TXOutput
    index  int64
    txid   []byte
}
```





通过返回值，返回数据结构

FindNeedUtxoInfo

FindNeedUtxoInfo(付款人，支付的金额）[ ]UtxoInfo

1. 找到满足转账需求所需要的钱，找到后立刻返回
2. 如果没有找到，则返回空切片已经0，创建交易前，会校验返回的金额是否满足转账需求，如果不满足则创建交易失败

```go
func (bc *BlockChain) FindNeedUtxoInfo(address string, amount float64)
([]UtxoInfo, float64) {
    fmt.Printf("FindNeedUtxoInfo called, address :%s, amount:%f\n", address,
amount)

    //1. 遍历账本，找到所有address（付款人）的utxo集合
    utxoinfos := bc.FindMyUtxo(address)

    //返还的utxoinfo里面包含金额
    var retValue float64
    var retUtxoInfo []UtxoInfo

    //2. 筛选出满足条件的数量即可，不要全部返还
    for _, utxoinfo := range utxoinfos {
        retUtxoInfo = append(retUtxoInfo, utxoinfo)
        retValue += utxoinfo.output.Value

        if retValue >= amount {
            //满足转账需求，直接返回
            break
        }
    }

    return retUtxoInfo, retValue
}
```

2. 创建普通交易

分析：

1. 参数：
    1. 付款人
    2. 收款人
    3. 付款金额
    4. bc *Blockchain
2. 逻辑分析：
    1. 找到付款人能够支配的合理的钱，返回金额和utxoinfo
    2. 判断返回金额是否满足转账条件，如果不满足，创建交易失败。
    3. 拼接一个新的交易
        1. 拼装inputs
            1. 遍历返回的utxonifo切片，逐个转成input结构
        2. 拼装outputs
            1. 拼装一个属于收款人的output
            2. 判断一下是否需要找零，如果有，拼装一个属于付款方output
        3. 设置交易id
        4. 返回
    4.

代码实现：

```
//普通交易
func NewTransaction(from, to string, amount float64, bc *BlockChain)
(*Transaction, error) {

    //1. 1. 找到付款人能够支配的合理的钱，返回金额和utxoinfo
    utxoinfos, value := bc.FindNeedUtxoInfo(from, amount)

    //2. 判断返回金额是否满足转账条件，如果不满足，创建交易失败。
    if value < amount {
        return nil, errors.New("付款人金额不足!")
    }

    //3. 拼接一个新的交易
    var inputs []TXInput
    var outputs []TXOutput

    //1. 拼装inputs
    for _, utxoinfo := range utxoinfos {
        input := TXInput{
            TXID:      utxoinfo.txid,
            Index:     utxoinfo.index,
            ScriptSig: from,
        }

        inputs = append(inputs, input)
    }
    //1. 遍历返回的utxonifo切片，逐个转成input结构
    //2. 拼装outputs
    //1. 拼装一个属于收款人的output
    output := TXOutput{
        LockScript: to,
        Value:      amount,
    }
    outputs = append(outputs, output)

    //2. 判断一下是否需要找零，如果有，拼装一个属于付款方output
    if value > amount {
        //找零
        output1 := TXOutput{
            LockScript: from,
            Value:      value - amount,
        }

        outputs = append(outputs, output1)
    }

    tx := Transaction{
        TxInputs:  inputs,
        TXOutputs: outputs,
        TimeStamp: time.Now().Unix(),
    }
```

```
    //3. 设置交易id
    tx.SetTxId()

    //4. 返回
    return &tx, nil
}
```

UTXO：未花费交易输出

1. 在比特币系统中，手续费没有单独的字段来定义。每一笔交易的总inputs（15）与总outputs（14 + 0.5）的差值就是手续费。

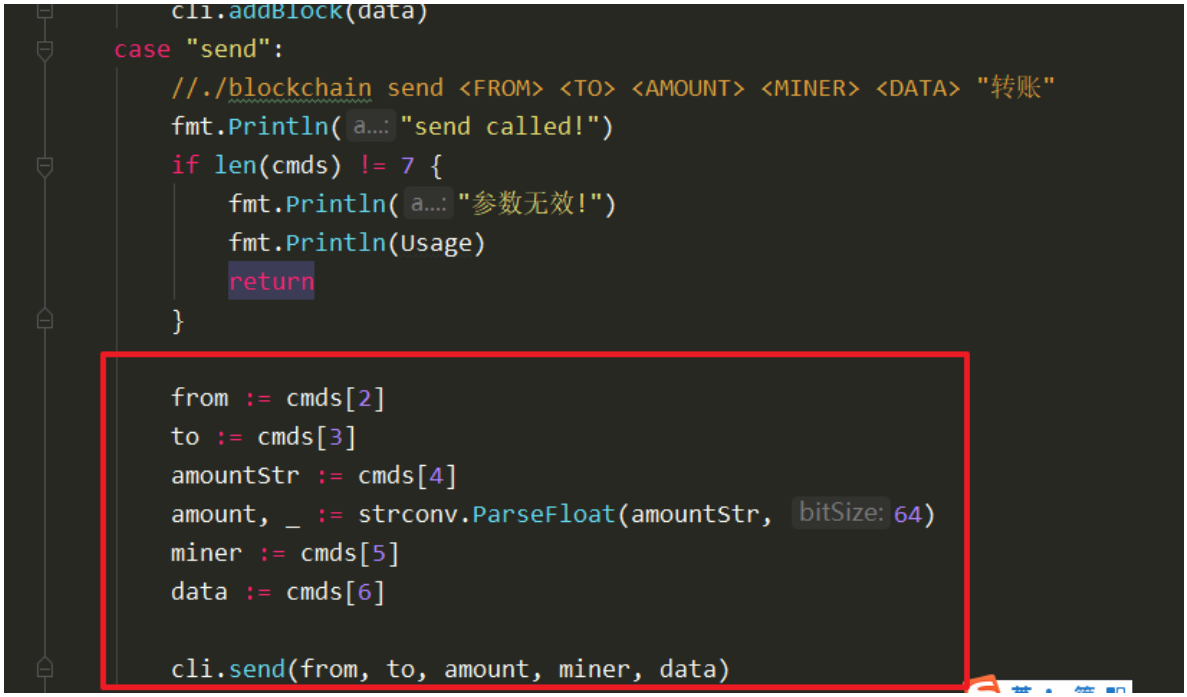
2. UTXO是最小的单位，任意面值的，每次使用时，必须一次用完，生成新的UTXO

3. send命令实现

在cli.go中增加send命令

```
const Usage = `
    ./blockchain addBlock <data>    "区块数据"
    ./blockchain print    "打印区块"
    ./blockchain getBalance <地址>  "获取某个地址的余额"
    ./blockchain send <FROM> <TO> <AMOUNT> <MINER> <DATA> "转账"
`

//持续解析命令的方法
func (cli *CLI) Run() {
```

解析响应的参数：



```
        cli.addBlock(data)
    case "send":
        //./blockchain send <FROM> <TO> <AMOUNT> <MINER> <DATA> "转账"
        fmt.Println( a...: "send called!")
        if len(cmds) != 7 {
            fmt.Println( a...: "参数无效!")
            fmt.Println(Usage)
            return
        }

        from := cmds[2]
        to := cmds[3]
        amountStr := cmds[4]
        amount, _ := strconv.ParseFloat(amountStr, bitSize: 64)
        miner := cmds[5]
        data := cmds[6]

        cli.send(from, to, amount, miner, data)
```

实现转账命令：

```
func (cli *CLI) send(from, to string, amount float64, miner, data string) {
    fmt.Printf("'%s'向'%s转账:'%f', miner:%s, data:%s\n", from, to, amount,
miner, data)

    //输入数据的有效性会进行校验
    //TODO

    //创建挖矿交易
    coninbaseTx := NewCoinbaseTx(miner, data)
    txs := []*Transaction{coninbaseTx}

    //一个区块只添加一笔有效的普通交易
    tx, err := NewTransaction(from, to, amount, cli.bc)
    if err != nil {
        fmt.Println("err:", err)
    } else {
        fmt.Printf("发现有效的交易，准备添加到区块, txid:%x\n", tx.Txid)
        txs = append(txs, tx)
    }

    //创建区块，添加到区块链
    cli.bc.AddBlock(txs)
}
```

更新getBalance函数

```go
func (cli *CLI) getBalance(address string) {
    //utxos := cli.bc.FindMyUtxo(address)
    utxoinfos := cli.bc.FindMyUtxo(address)
    var total float64

    for _, utxoinfo := range utxoinfos {
        total += utxoinfo.output.Value
    }

    fmt.Printf( format: "'%s'的比特币余额为:%f\n", address, total)
}
```

编译测试：

```
duke@DUKEDU51C6 MINGW64 /c/goprojects/src/go5期/03-比特币/v4-b-Transaction-send
$ ./blockchain send 中本聪 班长 2.5 lily 你是个好人
lastHash : 0000d8cd00ca717a195c248c65ea14dc0802ffaaaa9a8f5338fd1944a9489e99
CLI Run called!
send called!
'中本聪'向'班长转账:'2.500000', miner:lily, data:你是个好人
FindNeedUtxoInfo called, address :中本聪, amount:2.500000
FindMyUtxo called, address: 中本聪
找到了属于'中本聪'的output, index:0, value:12.500000
发现有效的交易，准备添加到区块，txid:3de976def1fa047c916d8a6b80986d116ef120785e46a0d5fb8464354223b7c1
AddBlock called!
挖矿成功，当前哈希值为:000090de4b77b262379540f866d38a4c81a2d31eeccabeabdd35654231445676, nonce: 287
```

查询三个人的金额：

1. 中本聪：10
2. 班长：2.5
3. lily：12.5

4. 优化程序

- IsFileExist
- IsCoinbase
- HashTransaction

判断是否为挖矿交易：

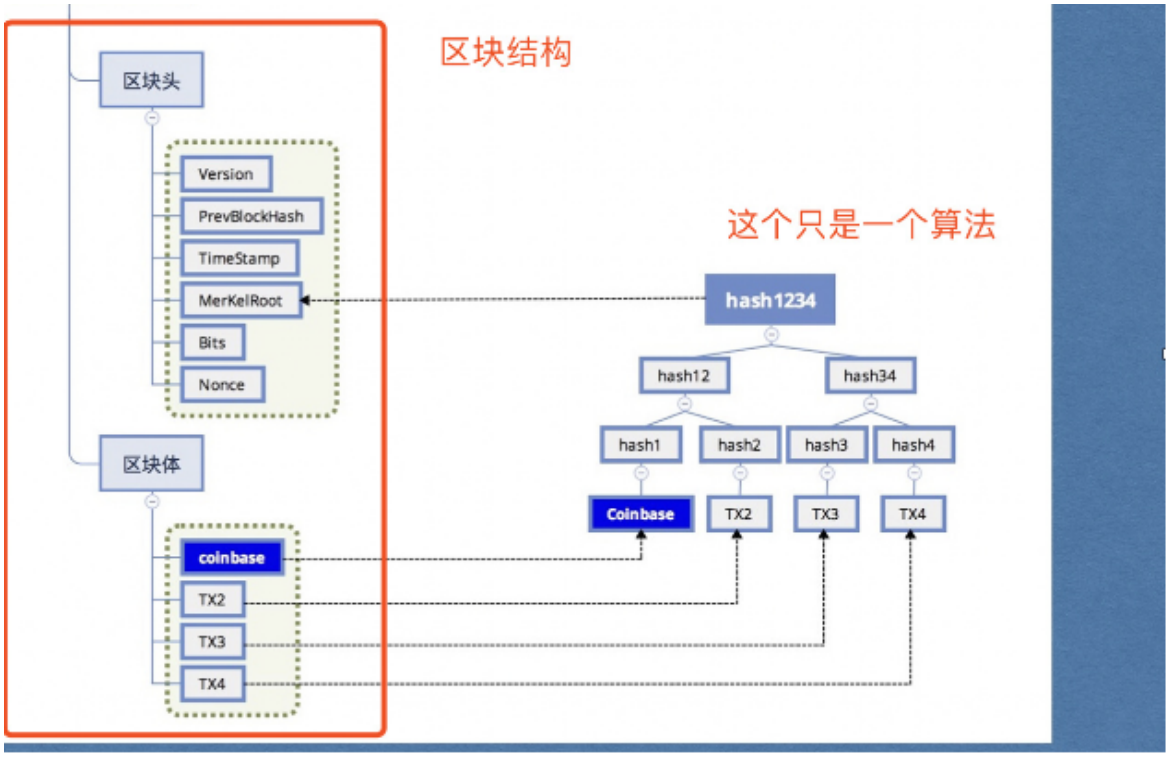
transaction.go

```go
//判断一个交易是否为挖矿交易
func (tx *Transaction) isCoinbaseTx() bool {
    input := tx.TxInputs[0]
    if len(tx.TxInputs) == 1 && input.TXID == nil && input.Index == -1 {
        return true
    }

    return false
}
```

调用：

```go
        //遍历inputs，得到一个map
        if !tx.isCoinbaseTx() {
            //如果不是挖矿交易，才有必要遍历innputs
            for _, input := range tx.TxInputs {
                if input.ScriptSig == address {
                    spentKey := string(input.TXID) //这个input的来源
                    spentOutput[spentKey] = append(spentOutput[spentKey], input.Index)

                    //下面是错误的，无法将数据写到spentOutput中
                    //indexArray := spentOutput[spentKey]
                    //indexArray = append(indexArray, input.Index)
                }
            }
        }
    }
```

HashTransaction实现，模拟生成梅克尔根



在block.go中，增加函数：

```go
//模拟计算梅克尔根
func (b *Block) HashTransaction() {
    var info []byte
    for _, tx := range b.Transactions {
        info = append(info, tx.Txid...)
    }

    hash := sha256.Sum256(info)

    b.MerkleRoot = hash[:]
}
```

在创建block中调用：

```go
func NewBlock(txs []*Transaction, prevHash []byte) *Block {
    block := &Block{
        Version:       "0",
        PrevHash:      prevHash,
        TimeStamp:     time.Now().Unix(),
        Bits:          0,
        Nonce:         0,
        Transactions: txs,
    }

    //将梅克尔根赋值
    block.HashTransaction()

    //挖矿过程暂且省略
    //block.setHash()
    pow := NewProofOfWork(block)
    nonce, hash := pow.Run()
```

```
print called!
++++++++++++++++++++++
version::0
prevHash:
hash:000035bbeec670749769f5b38cb8169a56c7d53a32b4fcb92f7b86c5db923ebd
merkleRoot:10001a46ab8ad969a38e6e39a327af52889b49214b5d9edd987c9c19b0cf30e2
timeStamp:1572677633
bits:0
nonce:24142
isValid: true
```

5. 钱包相关

椭圆曲线介绍

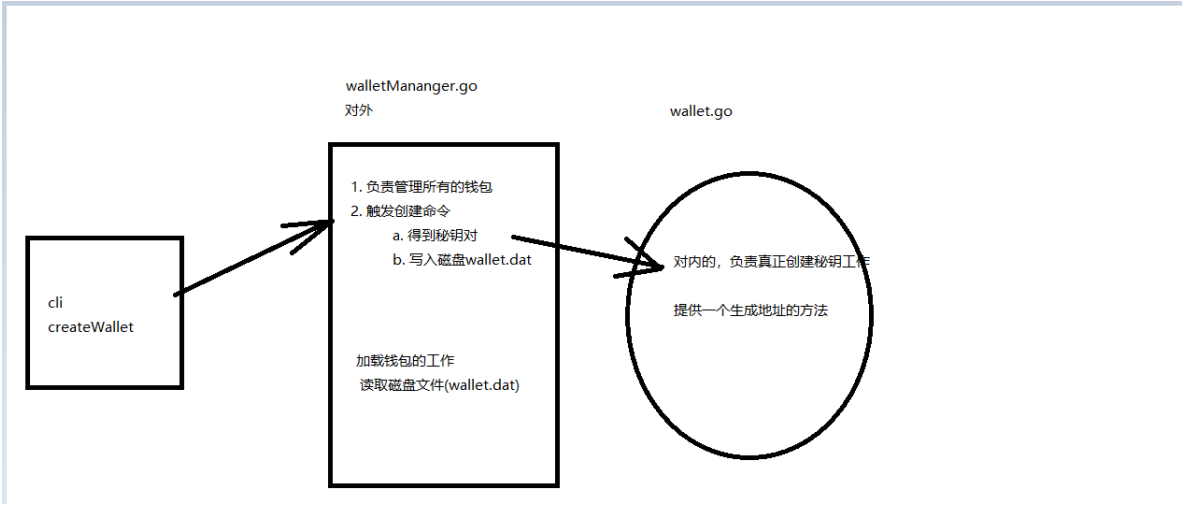
ecdsa介绍demo

```
package main

import (
    "crypto/ecdsa"
    "crypto/elliptic"
    "crypto/rand"
    "crypto/sha256"
    "fmt"
    "math/big"
)

func main() {
    //创建一条椭圆曲线
    curve := elliptic.P256()

    //1. 创建秘钥对
    privKey, err := ecdsa.GenerateKey(curve, rand.Reader)
    if err != nil {
        fmt.Println("生成私钥失败, err:", err)
        return
    }

    data := "hello world"

    hash := sha256.Sum256([]byte(data))

    //2. 使用私钥签名
    //r和s是数据签名
    r, s, err := ecdsa.Sign(rand.Reader, privKey, hash[:])
    fmt.Println("r len:", len(r.Bytes()))
    fmt.Println("s len:", len(s.Bytes()))

    //r与s的长度是相同的，我们将两者拼接到一起，进行传输
    //到对端，从中间分割，还原成big.Int类型即可
    signature := append(r.Bytes(), s.Bytes()...)

    //进行数据传输。。。。

    pubKey := privKey.PublicKey

    //3. 公钥验证签名

    //还原r, s
    var r1, s1 big.Int
    r1.SetBytes(signature[:len(signature)/2])
    s1.SetBytes(signature[len(signature)/2:])

    //func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool {
    //res := ecdsa.Verify(&pubKey, hash[:], &r1, &s1) ///正确的
    res := ecdsa.Verify(&pubKey, hash[:], &r1, &r1) //错误的
    fmt.Println("res :", res)
}
```

```
duke@DUKEDU51C6 MINGW64 /c/goprojects/src/go5期/03-比特币/test
$ go run ecdsaTest.go
r len: 32
s len: 32
res : true

duke@DUKEDU51C6 MINGW64 /c/goprojects/src/go5期/03-比特币/test
$ go run ecdsaTest.go
r len: 32
s len: 32
res : false

duke@DUKEDU51C6 MINGW64 /c/goprojects/src/go5期/03-比特币/test
```

6. 钱包关系

私钥=》公钥=》地址



Wallet

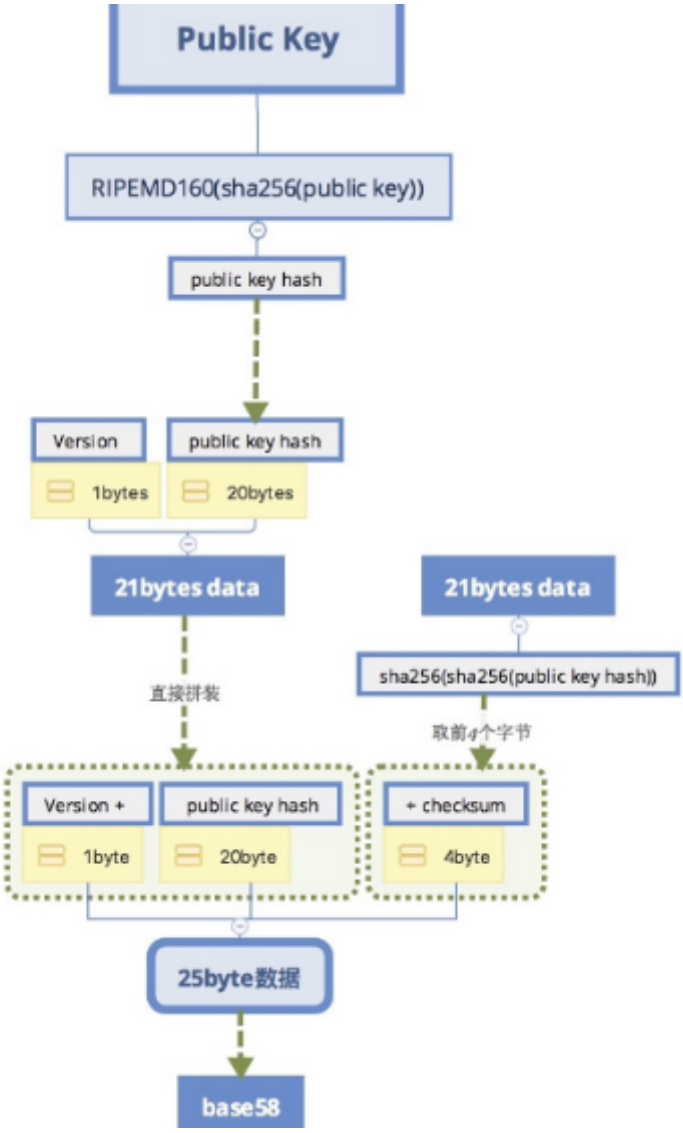
- 创建秘钥对
- 根据公钥生成地址

wallet.go文件，内容如下：

```
package main

import (
    "crypto/ecdsa"
    "crypto/elliptic"
    "crypto/rand"
    "fmt"
)
```

```go
type Wallet struct {
    PrivKey *ecdsa.PrivateKey
    PubKey  []byte //这不是原生的公钥，而是X，Y两个点的字节流拼成而成的
}

//创建一个秘钥对
func NewWallet() *Wallet {
    //私钥
    priKey, err := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
    if err != nil {
        fmt.Println("创建秘钥对失败，err:", err)
        return nil
    }
    //公钥
    pubKeyRaw := priKey.PublicKey
    x := pubKeyRaw.X
    y := pubKeyRaw.Y
    pubKey := append(x.Bytes(), y.Bytes()...)

    return &Wallet{
        PrivKey: priKey,
        PubKey:  pubKey,
    }
}
```

根据公钥生成地址




```go
func (w *Wallet) getAddress() string {
    //一、第一次哈希
    firstHash := sha256.Sum256(w.PubKey)
    //第二次哈希
    hasher := ripemd160.New()
    hasher.Write(firstHash[:])
    pubKeyHash := hasher.Sum(nil)

    //二、在前面添加1个字节的版本号
    payload := append([]byte{byte(00)}, pubKeyHash...)

    //三、做两次哈希运算，截取前四个字节，作为checksum，
    f1 := sha256.Sum256(payload)
    second := sha256.Sum256(f1[:])

    checksum := second[:4] //作闭右开

    //四、拼接25字节数据
    payload = append(payload, checksum...)

    //五、base58处理，得到地址
    address := base58.Encode(payload)
    return address
}
```

WalletManager

- 定义结构
    1. 定义一个map来管理所有的钱包
    2. key：地址
    3. value：wallet
- 创建结构
    1. 分配空间

```
package main

//- 定义结构
type WalletManager struct {
    //1. 定义一个map来管理所有的钱包
    //2. key：地址
    //3. value：wallet
    wallets map[string]*Wallet
}

//- 创建结构
func NewWalletMananger() *WalletManager {
    //return &WalletManager{
    //  wallets: make(map[string]*Wallet),
    //}

    var wm WalletManager
    wm.wallets = make(map[string]*Wallet)

    //加载已经存在钱包，从wallet.dat
    //TODO

    return &wm
}
```

在WalletManager.go中添加创建钱包的命令，用于cli调用：

```
func (wm *WalletManager) createWallet() (string, error) {
    //调用wallet结构的创建方法
    w := NewWallet()

    if w == nil {
        return "", errors.New("创建钱包失败!")
    }

    //填充自己的wallets结构
    //TODO ， 放入map结构，存储到磁盘

    //返回地址
    address := w.getAddress()
```

```
    return address, nil
}
```

在cli中添加命令createWallet

```
    case "createWallet":
        fmt.Println( a...: "createWallet called!")
        cli.createWallet()
```

在commandline.go中调用：

```
func (cli *CLI) createWallet() {
    wm := NewWalletMananger()
    if wm == nil {
        fmt.Println("打开钱包失败!")
        return
    }

    address, err := wm.createWallet()
    if err != nil {
        fmt.Println("创建钱包失败:", err)
        return
    }

    fmt.Println("创建新地址成功:", address)
}
```

测试：

```
duke@DUKEDU51C6 MINGW64 /c/goprojects/src/go5期/03-比特币/v5-a-wallet
$ ./blockchain createWallet
lastHash : 0000ba1fc708480b074726c66bbb079d52056d286fbce03d9a569b5e0bf0046e
CLI Run called!
createWallet called!
创建新地址成功: 1EyPttndkLVJ4YyixV7nuhZVqo5yq4bVdD

duke@DUKEDU51C6 MINGW64 /c/goprojects/src/go5期/03-比特币/v5-a-wallet
$ ./blockchain createWallet
lastHash : 0000ba1fc708480b074726c66bbb079d52056d286fbce03d9a569b5e0bf0046e
CLI Run called!
createWallet called!
创建新地址成功: 1HipCURaXoVkQdjjCPfLTtSq3c5rhw5bh8
```