在传统银行中，每一个账户都会有一个对应的数据表，

里面有一个字段记录A的总额，当A进行转账时，只需要修改这个字段的数据即可。

在比特币中，A=》B转账时，并不是修改一个字段，因为比特币的交易数据一旦写入账本之后，是不可以被修改的。

比特币是通过找零机制来完成转账功能的：

A总共有500个比特币（假设是一笔交易的金额）

A=》B 转账100

比特币会将500个比特币全部花费掉，转100个给B，转400个给A，比特币账本中，A原来的500个就作废了。

此时，我们没有修改原来交易中的数据（没有修改，但是系统中会有方式标识出这500个已经被消耗了）

同时，转账完成。

张三：1000元总额。

由10张100元面值组成

1. 工作400（区块10==》第 1条交易）
2. 理财300（区块20 ==》 第三条交易）
3. 副业300（区块50 ==》 第20条交易）

在比特币账本中，有三笔交易记录着张三的钱。

在比特币中，是没有一个字段来存储某个地址的总额的。

比特币通过遍历整个账本，去查询所有自己的私钥能够支配的零散的钱

进行交易时，比特系统会自动找到一个合理的钱

比特币中，接收转账使用的是地址。

A=》B转账,

需要A的私钥对A能够支配的钱进行`解锁`

需要使用B的`公钥的哈希`进行`锁定`

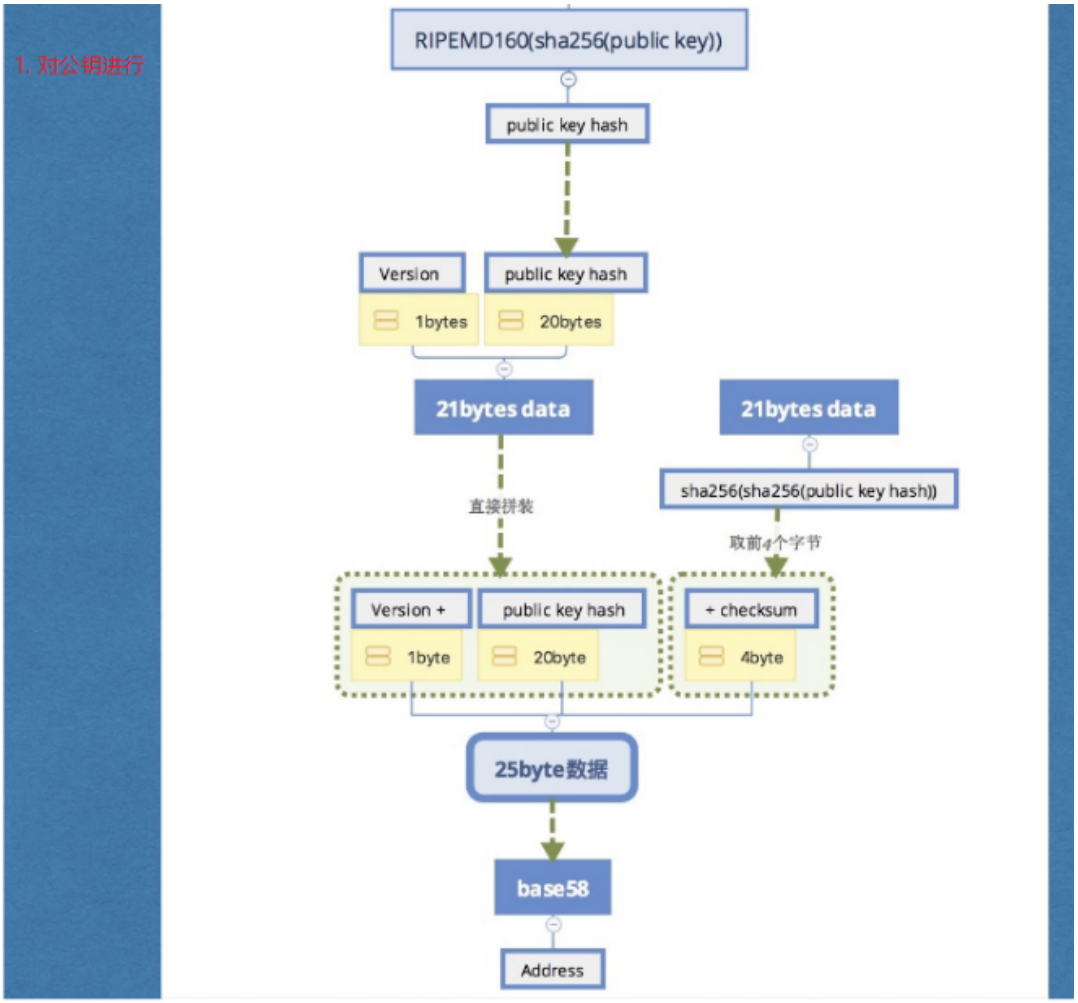
B: 私钥=》公钥=》公钥哈希《=》地址

地址

在线生成地址

BitCoinAddress(生成真实可用的地址)

1. 使用私钥PrivKey ==》获取公钥pubKey
2. 对公钥进行两次哈希处理：=》公钥哈希pubKeyHash
    1. sha256(pubKey)
    2. 第二次做ripemd160哈希处理，160位， 20字节
3. 再在公钥哈希前面加上一个字节的版本号。（当前网络版本号），得到payload，21字节
    1. 主网：0x00
    2. 测试：
4. 对拼好21字节数据做两次sha256,截取前4字节，得到checksum，即4字节的校验码。=》得到25字节的数据
5. 对25字节数据进行base58处理，得到地址



1私钥 =》1地址

1root privete =》 Son private =》 孙私钥

```
09:25:29    欢迎使用 Bitcoin Core 的 RPC 控制台。
            使用上下方向键浏览历史， 以及 Ctrl-L 清除屏幕。
            输入help命令显示可用命令信息。
            输入help-console来取得使用这个控制台的更多信息。

            警告： 已有骗子通过要求用户在此输入指令以盗取钱包。不要在没有完全理解命令规范时使用控制台。
09:26:07    dumpprivkey 2MsLzLtVyYaEUtNYageNNjzjgjoMGSyWiu5
09:26:07    cVqlWtwu3WHyuXUrmpBrB21dzLYYkyRJWVEuK99KsaCJb7upzvd8
09:26:25    dumpprivkey 2NEHg52QeTCrynDi2cUsJeFAwevST6h8AUw
09:26:25    cNgP6Dhs7TwAFoBbyTMnWVLcWJMBnoqu7emznFFtaJuTtNm7EPUz
```

两个脚本

交易输入：

1. 付款人的多笔钱的来源（100 + 50）
2. 一个或者多个

交易输出：

1. 一个或者多个

解锁脚本：

1. 需要提供付款方的私钥签名和公钥

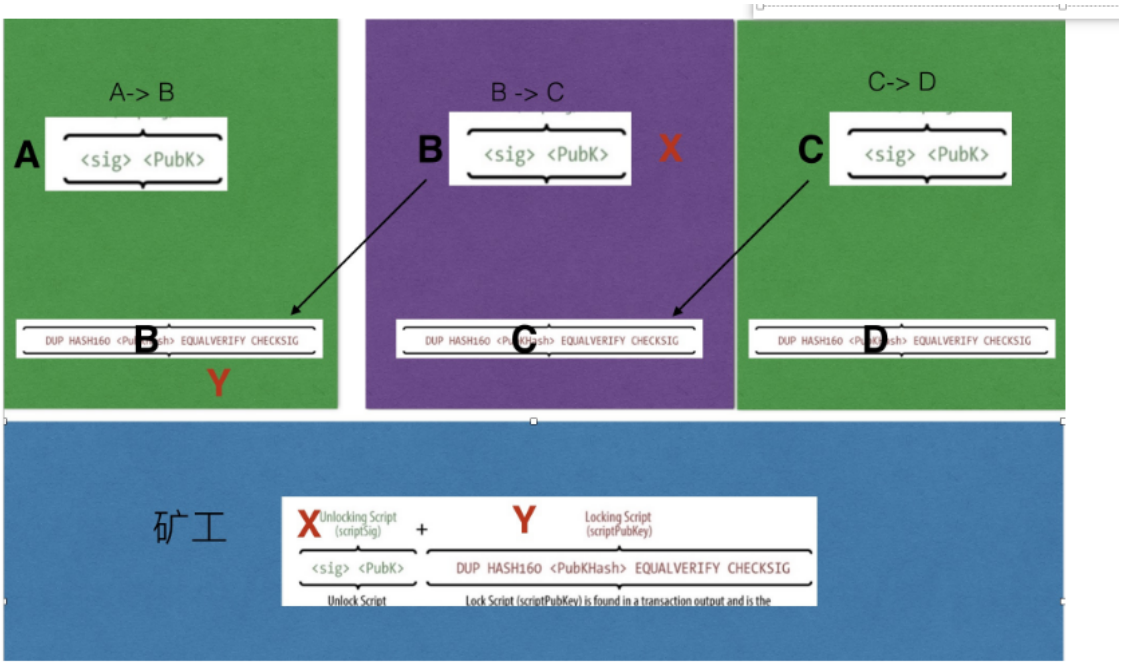
锁定脚本：

1. 收款方的地址（公钥哈希）

每一个input都会有一个解锁脚本：描述付款方可以使用这笔钱

麽易output都会有一个锁定脚本：描述收款方对这笔钱进行锁定。

解锁过程分析



想要花费output的公钥哈希

A：pubKeyHash

比较

A确实想花费A的钱

A：pubKey

A：Sig

校验签名与私钥是否匹配

A确实拥有A的私钥

交易结构

1. 交易输入：
    1. 所引用的output所在的交易id
    2. 所引用的output的索引值
    3. 解锁脚本：
        1. 私钥签名
        2. 公钥
2. 交易输出：
    1. 锁定脚本
    2. 接收金额
3. 时间戳：timestamp

交易结构定义

```
package main

import (
    "bytes"
    "crypto/sha256"
    "encoding/gob"
    "fmt"
    "time"
)

//交易结构
type Transaction struct {
    //交易ID
    Txid []byte

    //多个交易输入
    TxInputs []TXInput

    //多个交易输出
    TXOutputs []TXOutput

    //时间戳
    TimeStamp int64
}

//交易输入
type TXInput struct {
    //1. 所引用的output所在的交易id
    TXID []byte
    //2. 所引用的output的索引值
    Index int64
    //3. 解锁脚本：
    ScriptSig string //先使用string代替，后续会改成签名
    //1. 私钥签名
    //2. 公钥
}

//交易输出
type TXOutput struct {
    //1. 锁定脚本
    LockScript string
    //2. 转账金额
    Value float64
}
```

获取交易ID

```
//设置当前交易的id，使用交易本身的哈希值作为自己交易id
func (tx *Transaction) SetTxId() {
    var buff bytes.Buffer

    encoder := gob.NewEncoder(&buff)
    err := encoder.Encode(tx)
    if err != nil {
        fmt.Println("设置交易id失败，err:", err)
        return
    }

    hash := sha256.Sum256(buff.Bytes())

    tx.Txid = hash[:]
}
```

创建挖矿交易

```
const reward = 12.5

//挖矿交易
//没有引用的输入，只有输出，只有一个output
func NewCoinbaseTx(miner string, data string) *Transaction {
    intputs := []TXInput{{
        TXID:      nil,
        Index:     -1,
        ScriptSig: data,
    }}

    outputs := []TXOutput{{
        LockScript: miner,
        Value:      reward,
    }}

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

    //设置交易id
    tx.SetTxId()

    return tx
}

//普通交易
func NewTransaction() *Transaction {
    //TODO
    return nil
}
```

每一个交易都会产生交易输出output，这个output可以作为后续交易的输入。

如果这个output还没有被其他交易引用，那么它有一个专门的名字：未消费交易输出（Unspent TX Output），UTXO

一个地址的总额，就是遍历账本，查询这个地址锁定的UTXO所包含的Value总和。

使用Transaction改写程序

1.改写block字段

```
    Nonce int64

    //区块体，交易数据
    //Data []byte
    Transactions []*Transaction //多条交易
}

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

2.blockchain.go改写NewBlockchain函数

```
            return err
        }

        //写入创世块
        //创建一个挖矿交易，里面写入创世语
        coinbaseTx := NewCoinbaseTx( miner: "中本聪", genesisInfo)
        genesisBlock := NewBlock([]*Transaction{coinbaseTx}, prevHash: nil)

        //第一次：写入区块的数据
        _ = b.Put(genesisBlock.Hash, genesisBlock.Serialize() /*区块转换成字节流*/)
        //第二次：写入最后一个区块哈希
        _ = b.Put([]byte(lastBlockHashKey), genesisBlock.Hash)

        hash := b.Get([]byte(lastBlockHashKey))
        fmt.Printf( format: "lastHash : %x\n", hash)
```

3.AddBlock修改

```go
//添加区块的方法
func (bc *BlockChain) AddBlock(txs []*Transaction) {
    fmt.Println( a...: "AddBlock called!")
    //最后一个区块的哈希值
    lastHash := bc.tail

    //1. 创建新的区块
    newBlock := NewBlock(txs, lastHash)

    bc.db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blockBucket))
        if b == nil {
            log.Fatal( v...: "AddBlock是, bucket不应为空!")
        }
```

4.commandline.go

```go
func (cli *CLI) addBlock(data string) {
    //cli.bc.AddBlock(data) //TODO  <<===注释
}
```

```go
    fmt.Printf( format: "merkleRoot:%x\n", block.MerkleRoot)
    fmt.Printf( format: "timeStamp:%d\n", block.TimeStamp)
    fmt.Printf( format: "bits:%d\n", block.Bits)
    fmt.Printf( format: "nonce:%d\n", block.Nonce)
    pow := NewProofOfWork(block)
    fmt.Printf( format: "isValid: %v\n", pow.isValid())
    fmt.Printf( format: "data: %s\n", block.Transactions[0].TxInputs[0].ScriptSig)

    //判断是否已经是创世块
```

5.proofofwork.go

```go
func (pow *ProofOfWork) prepareData(nonce int64) []byte {
    b := pow.block

    tmp := [][]byte{
        []byte(b.Version),
        b.PrevHash,
        b.MerkleRoot,
        digi2byte(b.TimeStamp),
        digi2byte(b.Bits),
        digi2byte(nonce), //<<<<===== 注意修改, 要使用传递进来的nonce
        //b.Data, //TODO
    }
```

测试

获取挖矿人金额

遍历 交易(只有outputs)

获取余额

遍历inputs