实验二 基于比特币区块链的简单搭建（下）

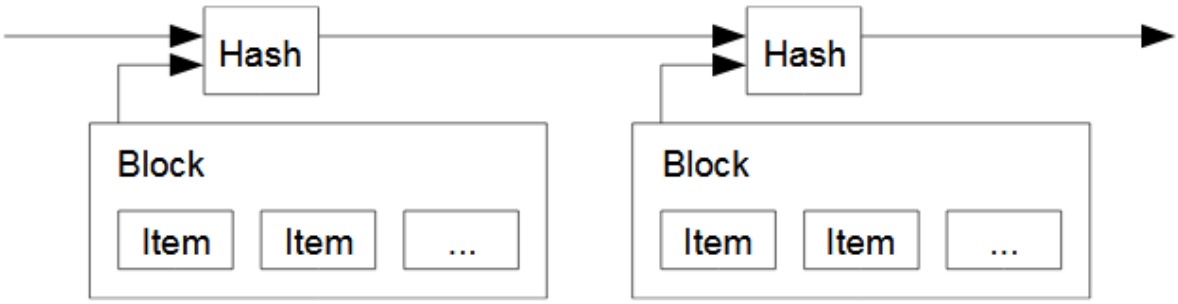
实验目的

- 进一步理解区块链上的数据结构
- 实现基于的POW共识算法出块
- 实现比特币上的账户创建和查询
- 理解UTXO的基本使用方法
- 实现区块链与数据库的交互

实验介绍

区块链

区块链是通过链连接的区块的方式连接的数字账本，是一个不断增加的分布式账本。在链的层面，我们对应就是对一个个区块的数据进行的操作，来保证他们的串联成全序关系。



例如在我们的代码中，`NewGenesisBlock`代表了创建一个创世区块的意思。`addBlock`代表了添加单个区块。

因为我们在实验中使用了区块链，对应区块链的结构

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

`tip`代表了最新区块的哈希值，`db`表示了数据库的连接

区块链共识协议

区块链共识的关键思想就是为了矿工通过一些复杂的计算操作来获取写入区块的权利。这样的复杂工作量是为了保证区块链的安全性和一致性。如果是对应比特币、以太坊等公有链的架构，对于写入的区块会得到相应的奖励（俗称挖矿）。

根据比特币的白皮书,共识部分是为了决定谁可以写入区块的问题，区块链的决定是通过最长链来表示的，这个是因为最长的区块对应有最大的工作量投入在其中。相应地，为了保证区块链的出块保持在一个相对比较稳定的值，对应地，对进行区块链共识难度的调整来保证出块速度大致保持一致。对应比特币来说，写入区块的节点还对应会获得奖励。

工作量证明（POW）

工作量的证明机制，简单来说就是通过提交一个容易检测，但是难以计算的结果，来证明节点做过一定量的工作。对应的算法需要有两个特点：计算是一件复杂的事情，但是证明结果的正确与否是相对简单的。对应地行为，可以类比生活中考驾照、获取毕业证等。

工作量证明由Cynthia Dwork 和Moni Naor 1993年在学术论文中首次提出。而工作量证明（POW）这个名词，则是在1999年 Markus Jakobsson 和Ari Juels的文章中才被真正提出。在发明之初，POW主要是为了抵抗邮件的拒绝服务攻击和垃圾邮件网关滥用，用来进行垃圾邮件的过滤使用。POW要求发起者进行一定量的运算，消耗计算机一定的时间。

区块链哈希

在上个实验中，我们已经实现了一个SHA256算法的哈希函数，它具有区块链上哈希函数的一些基本特点：

1. 原始数据不能直接通过哈希值来还原，哈希值是没法解密的。
2. 特定数据有唯一确定的哈希值，并且这个哈希值很难出现两个输入对应相同哈希输出的情况。
3. 修改输入数据一比特的数据，会导致结果完全不同。
4. 没有除了穷举以外的办法来确定哈希值的范围。

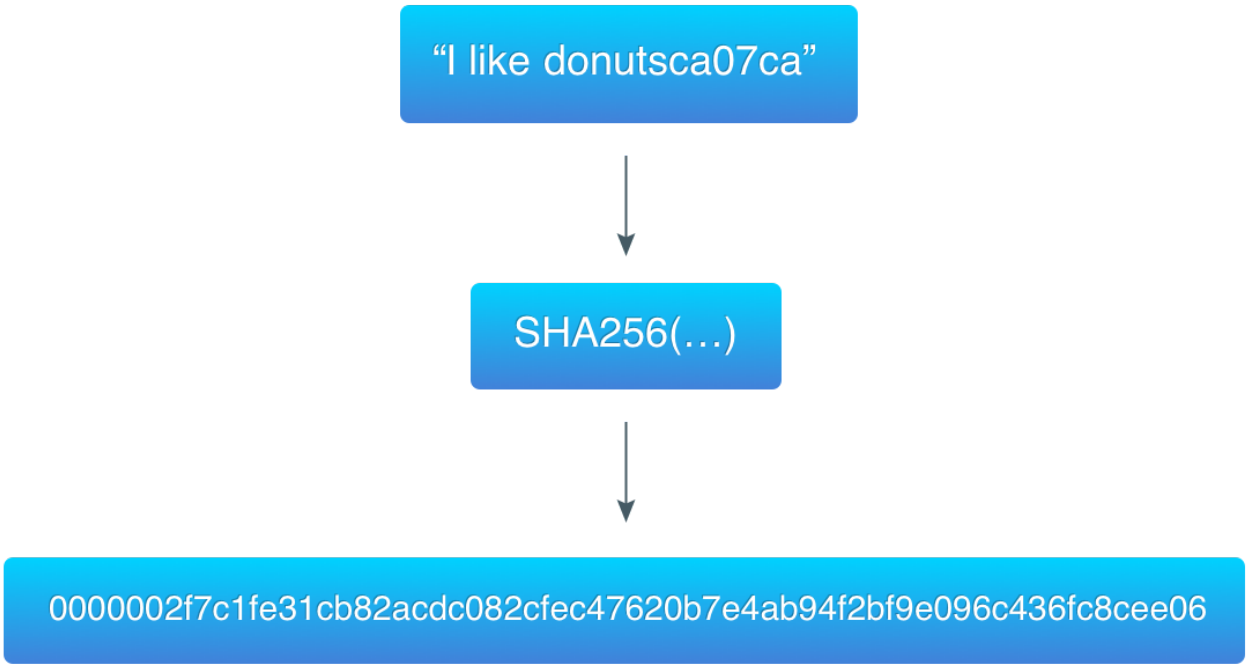
在接下来的实验中，我们会通过sha256算法来实现一个简单的工作量证明。

比特币采用了哈希现金(hashcash)的工作量证明机制，也就是之前说过的用在垃圾邮件过滤时使用的方法，对应流程如下：

1. 本次实验我们需要首先构建当前区块头，区块头包含**版本号，上一个区块哈希值(32位)，当前区块数据对应哈希（32位，即区块数据的merkle根），时间戳，区块难度，计数器(nonce)**。通过计算当前区块头的哈希值来求解难题。
2. 添加计数器，作为随机数。计算器从0开始基础，每个回合**+1**
3. 对于上述的数据来进行一个哈希的操作。
4. 判断结果是否满足计算的条件：
    1. 如果符合，则得到了满足结果。
    2. 如果没有符合，从2开始重新直接2、3、4步骤。

从中也可以看出，这是一个"非常暴力"的算法。这也是为什么这个算法需要指数级的时间。

这里举一个简单的例子，对应数据为`I like donuts`，`ca07ca`是对应的前一个区块哈希值



在本次实验中，我们选用了一个固定的难度值`targetBits //难度值`来进行计算。难度值意味着我们需要获取一个**1<<(256-targetBits)**小的数。在代码测试时，可以修改Block.NewBlock，来保持困难度不改变)。 计算哈说数据的内容可以通过区块序列化来获得`func (b *Block) Serialize() []byte`

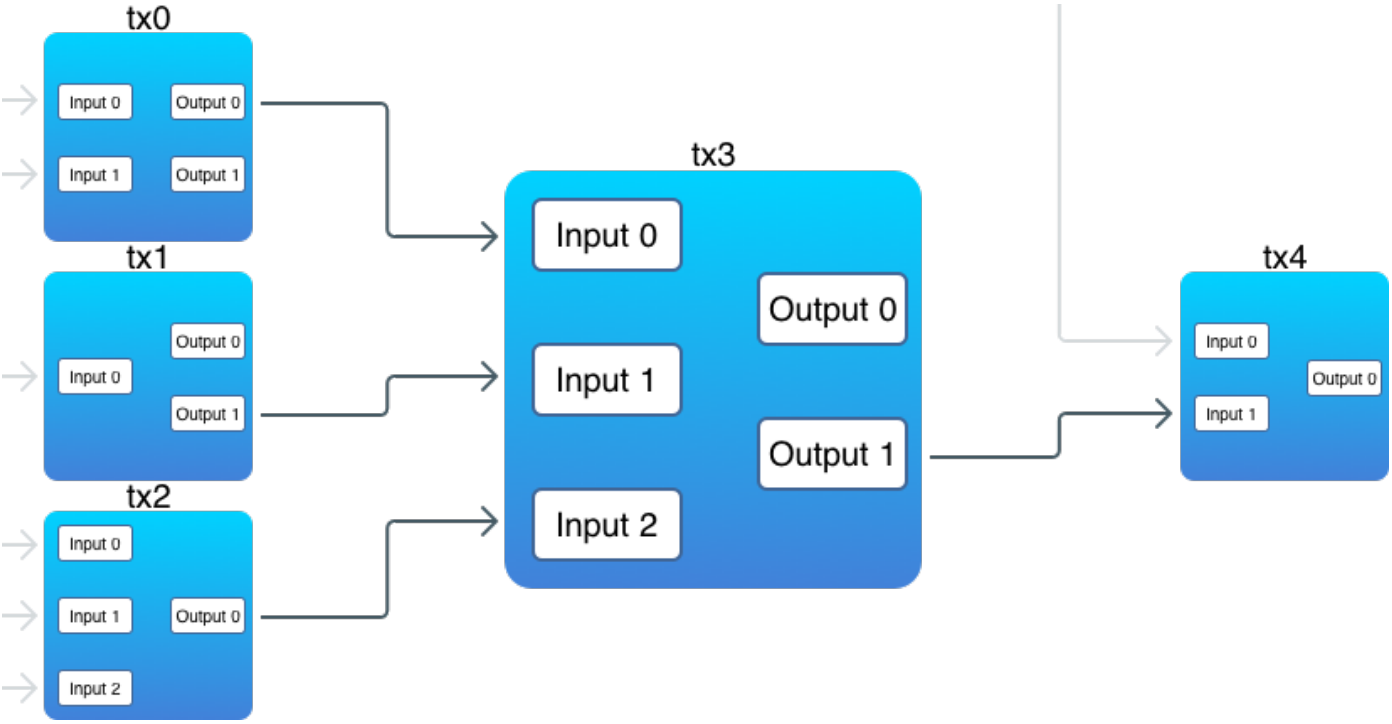
```
type ProofOfWork struct {
  block  *Block
  target *big.Int
}

type Block struct {
  Header *BlkHeader
  Body   *BlkBody
}

type BlkHeader struct {
  Version       int64
  PrevBlockHash []byte
  MerkleRoot    []byte
  Timestamp     int64
  Bits          int64
  Nonce         int64
}

type BlkBody struct {
  Transactions Transactions
}
```

`ProofOfWork`是一个区块的指针,对应我们在区块中记录加上了**Bits**，记录当前区块计算的难度。 为了进行区块上的操作，我们需要使用`big.Int`来得到一个大数操作，对应难度就是之前提到的`1<<(256-targetBits)`。

在本次实验中，我们需要完成`func (pow *ProofOfWork) Validate() bool`,`func (pow *ProofOfWork) Run() (int64, []byte)`，对应区块链上的验证和挖矿相关的操作。此外，`func (bc *Blockchain) MineBlock(transactions []*Transaction) *Block`会调用`ProofOfWork.Run`来生成一个合法的区块，进而写入区块链。

UTXO

UTXO是Unspent Transaction Outputs的缩写，中文翻译是没有花掉的交易输出，实际可以理解为在一次转账时剩余没有转出的资金。UTXO的交易模型上，用户通过使用未使用的交易输出（UTXO）来执行一笔交易。



在UTXO中，交易的转账方需要通过签名来证明自己是UTXO的合法使用者，并且通过输出脚本来限制收款方是谁。在比特币中，通过执行`Script`脚本来限制交易的接收方和验证方。在一笔UTXO的交易中，**每个输入都指向之前一些输出，每个输出中存储了具体的交易金额数量。**在UTXO中一个显著的特点就是单个输出是不可分的，如果只需要部分输出，可以生成一笔UTXO交易，把金额分为两个不同的部分。

```
type Transaction struct {
  ID   []byte
  Vin  []TXInput
  Vout []TXOutput
}

type TXInput struct {
  Txid      []byte
  Vout      int
  Signature []byte
  PubKey    []byte
}
```

```
type TXOutput struct {
  Value      int
  PubKeyHash []byte
}
```

在`TXOutput`中，`Value`字段对应是存储的金额大小，锁定脚本是通过`PubKeyHash`来规定的，对应是用户的公钥地址的哈希值。

在`TXInput`中，`Txid`字段存储了对应交易的哈希值，`Vout`字段存储了上一笔交易的索引，`Signature`存储了交易的签名，`PubKey`存储了公钥。

在本次实验中，需要学习并构建UTXO来进行交易，熟悉了解UTXO上交易的基本格式和一般的使用方法

在UTXO交易中，由于都是先有输出再有输入，所以当我们需要进行UTXO查询的时候可以通过从后往前遍历的方式来进行实现：首先先查看保存对应有输入，然后去查看哪些输出时判断是否已经被花费。

在本次实验中，我们需要完成`func NewUTXOTransaction(from, to []byte, amount int, UTXOSet *UTXOSet) *Transaction`，通过输入输出的地址和对应金额来构建UTXO的交易

UTXO池

在比特币的设计中，所有的交易都是放在UTXO池中，这样的好处是为了快速的得到某个UTXO是否当前可用。这也引入了另外问题：如何判断哪些UTXO是属于我的？我当前的余额是多少？

这个时候，我们就需要通过公私钥的唯一标识来算出我们当前的余额了。**在构建UTXO的交易时，我们需要通过UTXO池来需要当前属于自己的UTXO，然后获得相对应的账户金额，然后再生成对应的输出。**在这个实验中，我们会使用`utxo_set.go`中具体实现utxo池`FindUnspentOutputs`函数来查询用户当前未未花费的UTXO金额和对应的一个map映射关系，`FindUTXO`和`FindUnspentOutputs`的功能上大抵相似，但是是通过[]TXOutput的方式输出的,`Reindex`来查询持久化存储当前未花费的UTXO（实现类似cache的操作）

数据结构

在比特币代码中，区块主要存储的是两种数据：

1. 区块信息，存储对应每个区块的元数据内容。
2. 区块链的世界状态，存储链的状态，当前未花费的交易输出还有一些元数据

在我们本次实验中，区块链需要存储的信息相对也进行了简化。例如k-v数据库中，存储数据如下：

1. b，存储了区块数据
2. l，存储了上一个区块信息

UTXO数据存储了所有的UTXO

数据库

在本次实验中，我们选取了BoltDB的数据库。这是一个简单的，轻量级的集成在Go语言上的数据库。他和通常使用的关系型数据库（MySQL,PostgreSQL等）不同的是，它是一个K-V数据库。所以，数据是以键值对的形式进行存储的。在BoltDB上对应操作是存储在bucket中的。所以，为了存储一个数据，我们需要知道key和bucket。在我们区块链的实验中，我们是希望通过数据库来进行对于区块的存储操作。

在本次使用中，我们可以通过encoding/gob 来进行数据的序列化和反序列化。

对于数据库的操作主要如下：

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

用来创建一个数据库连接的实例。Go 关键词`defer`在当前函数返回前执行传入的函数，在这里用来数据库的连接断开。

在BoltDB中，对于数据库的操作是通过`bolt.Tx`来执行的，对应有两种交易模式**只读操作和读写操作**

对于读写操作的格式如下：

```
err = db.Update(func(tx *bolt.Tx) error {
...
})
```

对于只读操作的格式如下：

```
err = db.View(func(tx *bolt.Tx) error {
...
})
```

例如，所给代码中，区块链的创建代码如下：

```
  err = db.Update(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte(blocksBucket))

    if b == nil {
      fmt.Println("No existing blockchain found. Creating a new one...")
      genesis := NewGenesisBlock()

      b, err := tx.CreateBucket([]byte(blocksBucket))
      if err != nil {
        log.Panic(err)
      }

      err = b.Put(genesis.Hash, genesis.Serialize())
      if err != nil {
        log.Panic(err)
      }

      err = b.Put([]byte("l"), genesis.Hash)
      if err != nil {
        log.Panic(err)
      }
      tip = genesis.Hash
    } else {
      tip = b.Get([]byte("l"))
    }
```

```
    return nil
  })
```

其中，我们通过`l`读取的是上一个区块的信息，所以我们在添加一个新的区块之后，需要维护`l`字段对应的内容。在查询UTXO时，需要迭代器读取UTXO池中相应需要的UTXO。`db.Update`的操作是通过`err := db.Update(func(tx *bolt.Tx) error{ ··· return nil })`函数中的具体事务来完成的，如果`return err`会回滚事务，如果`return nil`会对应提交事务。另外，`TX.Bucket`是对应存储健值对集合的桶，键值是唯一索引，对应这个实验中存在`blocksBucket`和`utxoBucket`两个桶。

未花费的UTXO池需要通过`tx.Bucket([]byte(utxoBucket))`来进行读取,需要通过迭代区块的UTXO来进行维护。

实验内容

目录结构

```
├── go.mod //go模块管理
├── merkle_tree.go //merkle树相关代码
├── merkle_tree_test.go //merkle树验证部分相关代码
├── proofofwork.go //POW验证相关代码，本次实验可以不使用
├── transaction.go //交易相关代码
├── util.go   //一些操作
├── utxo_set.go  //utxo结合相关代码，本次试验中可以不适用
└── wallet.go //wallet相关代码
├── TXInput.go  //交易输入相关代码
├── TXOutput.go //交易输出相关代码
├── block.go  //区块相关代码
├── blockchain.go  //区块链相关代码
```

UTXO池部分

```
func (u UTXOSet) FindUnspentOutputs(pubkeyHash []byte, amount int) (int, map[string]
[]int)
```

POW部分

```
func (pow *ProofOfWork) Validate() bool
func (pow *ProofOfWork) Run() (int64, []byte)
```

Blockchain部分

```
func (bc *Blockchain) MineBlock(transactions []*Transaction) *Block
func (bc *Blockchain) FindUTXO() map[string]TXOutputs
```

Transaction部分

```
func NewUTXOTransaction(from, to []byte, amount int, UTXOSet *UTXOSet) *Transaction
```

参考资料

比特币白皮书

比特币代码

区块链哈希算法

POW算法

哈希现金

UTXO