LAB2实验报告

【实验名称】区块链编写

姓名 李泓民 学号 PB18071495

【实验目的及要求】

在实验一构建了一个区块链的数据结构以及对应持久化操作的数据库，在这个基础上对于共识部分进行补充，保证区块的一致性和安全性。添加一个区块需要所有节点达成共识，计算一个难以计算的hash值，解决出数学问题。

【实验原理】

区块链共识的关键思想就是为了结点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。

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。主要概念就是：**通过工作以获得指定成果，用成果来证明曾经付出的努力**

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。

在比特币中，使用非对称密码解决了数字货币的所有权问题，用区块时间戳解决了交易的存在性问题，用分布式账本解决了剔除第三方结构后交易的验证问题，剩下需要解决的问题是双重支付，这要求所有节点账本统一，而真正的平等又必须赋予人人都有记账的权利，记账是一件简单的事情，每个人都可以做，显然最终会存在众多大同小异的账本，但我们只需要其中的一个账本，POW选出了有用的那个账本。

POW给记账加入成本，总账本由各个分页按照时间先后排序，给每个账本分页设立一个评判标准，以区分账本分页是否合格，这给记账增加了难度，同时给每个账本分页加入一个随机元素，用以调节记账难度以保证一定时间段内只有一个人生成合格的账本分页。增加的成本就是工作量，合格的账本分页就是工作量证明，这个计算使用了hash函数。

哈希函数

哈希函数是输入数据进行一种函数计算，获取一个独特的表示。哈希函数需要满足如下的性质：

1. 可以接受任意大小的输入
2. 输出是固定长度的
3. 计算哈希的过程相对是比较简单的，时间都在O(n)

对于区块链的哈希函数，也需要满足一定优秀的性质：

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。

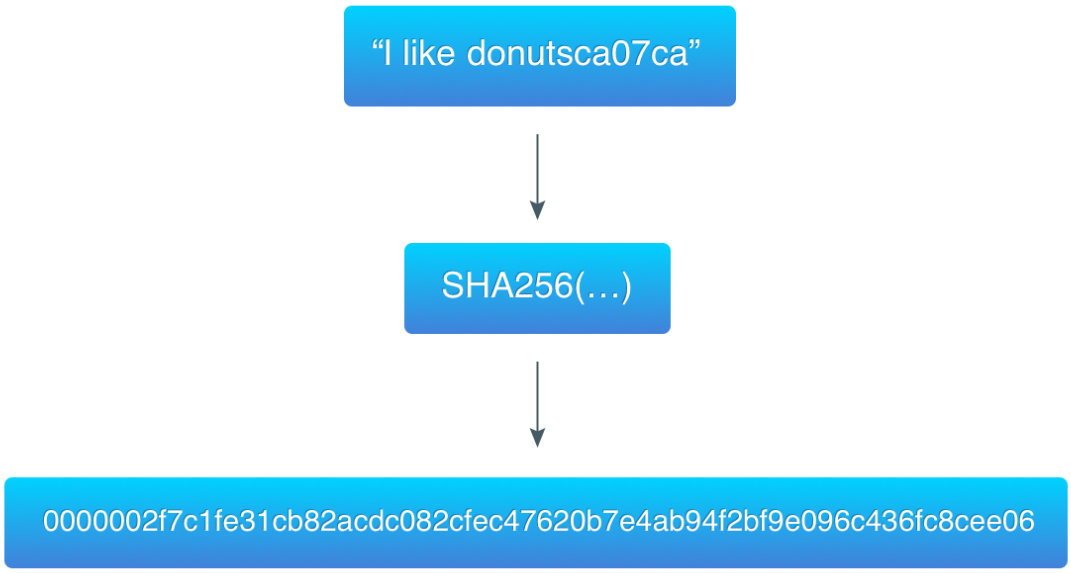
区块链哈希

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. 从区块链中获取一些公开的数据，对应本次实验我们需要获取**上一个区块哈希值(32位)，当前区块数据对应哈希（32位），时间戳，区块难度，随机数。**对应数据直接进行合并的操作来进行合并。
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。

从中也可以开出，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



【实验平台】

在本地使用vscode进行代码编辑，在学校提供的试验平台进行代码正确性验证

【实验步骤】

proofofwork.go

```
const targetBits = 10 //难度值
```

选用一个固定的难度值来进行计算。10位的难度值意味着需要获取一个**1<<(255-10)**小的数。

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

`ProofOfWork`是一个区块的指针和一个目标值，我们使用了`big.Int`来得到一个大端的数据，对应难度就是之前提到的`1<<(255-targetBits)`。

在这个实验中，我们还需要注意到的是`第一个区块对应的hash`是一个为空的值。在这个实验中，可以使用`"crypto/sha256`来进行哈希函数的操作。对于*int*转*byte*的操作可以使用`utiles.go`里的`IntToHex`函数来实现

Run（）pow计算部分

从区块链中获取一些公开的数据，即**上一个区块哈希值(32位)，当前区块数据对应哈希（32位），时间戳，区块难度，随机数**，对应数据直接进行合并的操作来进行合并。合并时把不是byte的数据转换为byte类型，一起记录在data之中。对于Run函数的修改：

```
func (pow *ProofOfWork) Run() (int, []byte) {
    nonce := 0
    var hashInt big.Int
    var hash [32]byte
    for nonce < maxNonce {
        data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
            pow.block.HashData(),//pow.block.Hash
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
        )

        hash = sha256.Sum256(data)
        hashInt.SetBytes(hash[:])

        if hashInt.Cmp(pow.target) == -1 {
            pow.block.Hash = hash[:]
            break
        } else {
            if nonce < maxNonce{
                nonce++
            } else {
                nonce = -1
                return nonce,hash[:]
            }
        }
    }

    return nonce, pow.block.Hash
}
```

这里我遇到过一个问题，debug许久才找出来。。。在使用当前区块数据对应哈希（32位）时，先是使用了`pow.block.Hash`作为当前数据块hash，没有考虑到在建立新块时还没写入相应的hash，需要直接调用生成merkle树的函数生成hash，否则是空值。所以应该使用`pow.block.HashData()`，addblock时计算pow就应该要有相应的hash值。

Validate() pow结果的验证工作

通过在pow中保存的nonce直接计算hash，并与target比较，如果相等则证明工作是有效的，返回true，否则返回false：

```
func (pow *ProofOfWork) Validate() bool {
```

```
    var hashInt big.Int

    data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
            pow.block.HashData(),//pow.block.Hash
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
    )
    hash := sha256.Sum256(data)
    hashInt.SetBytes(hash[:])

    if hashInt.Cmp(pow.target) == -1{
        return true
    } else {
        return false
    }
}
```

bonus

> 自己编写一个满足区块链要求的哈希函数（如sha256，sha3），并说明其满足区块链哈希函数的性质。

由于我觉得自己没有能力遍新的hash函数,于是照着sha256的计算过程把sha256重新实现了一次

整个实现过程如下：

```
package main

import (
    "encoding/binary"
    "fmt"
    "bytes"
    "log"
)

func main(){
    var str string = "abcd"//测试的输入
    var data []byte = []byte(str)
    fmt.Printf("%x", Sha256Compute(data))
    return
}
func IntToHex(num int64) []byte {
    buff := new(bytes.Buffer)
    err := binary.Write(buff, binary.BigEndian, num)
    if err != nil {
        log.Panic(err)
    }
    return buff.Bytes()
}
func Sha256Compute(message []byte) [32]byte {
```

```
    //初始哈希值为前8个质数(2到19)的平方根的小数部分的前32位
    h0 := uint32(0x6a09e667)
    h1 := uint32(0xbb67ae85)
    h2 := uint32(0x3c6ef372)
    h3 := uint32(0xa54ff53a)
    h4 := uint32(0x510e527f)
    h5 := uint32(0x9b05688c)
    h6 := uint32(0x1f83d9ab)
    h7 := uint32(0x5be0cd19)

    //计算过程当中用到的常数,即前64个质数(2到311)的立方根小数部分的前32位:
    k := [64]uint32{
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
0x59f111f1, 0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74,
0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f,
0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3,
0xd5a79147, 0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354,
0x766a0abb, 0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,
0xd6990624, 0xf40e3585, 0x106aa070,
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3,
0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa,
0xa4506ceb, 0xbef9a3f7, 0xc67178f2
    }
//前期处理
    tobecompute := append(message, 0x80)
    if len(tobecompute) % 64 < 56 {
        suffix := make([]byte, 56 - (len(tobecompute) % 64))
        tobecompute = append(tobecompute, suffix...)
    } else {
        suffix := make([]byte, 64 + 56 - (len(tobecompute) % 64))
        tobecompute = append(tobecompute, suffix...)
    }
    msgLen := len(message) * 8
    bs := make([]byte, 8)
    binary.BigEndian.PutUint64(bs, uint64(msgLen))
    tobecompute = append(tobecompute, bs...)

    slices := [][]byte{};

    for i := 0; i < len(tobecompute) / 64; i++ {
        slices = append(slices, tobecompute[i * 64: i * 64 + 63])
    }

    //主循环
    for _, chunk := range slices {
        w := []uint32{}

        for i := 0; i < 16; i++ {
            w = append(w, binary.BigEndian.Uint32(chunk[i * 4:i * 4 + 4]))
        }
        w = append(w, make([]uint32, 48)...)

```

```
        //W消息区块处理
        for i := 16; i < 64; i++ {
            s0 := Loopright(w[i - 15], 7) ^ Loopright(w[i - 15], 18) ^ (w[i - 15] >> 3)
            s1 := Loopright(w[i - 2], 17) ^ Loopright(w[i - 2], 19) ^ (w[i - 2] >> 10)
            w[i] = w[i - 16] + s0 + w[i - 7] + s1
        }

        a := h0
        b := h1
        c := h2
        d := h3
        e := h4
        f := h5
        g := h6
        h := h7

        //在主循环中用压缩函数处理
        for i := 0; i < 64; i++ {
            S1 := Loopright(e, 6) ^ Loopright(e, 11) ^ Loopright(e, 25)
            ch := (e & f) ^ ((^e) & g)
            temp1 := h + S1 + ch + k[i] + w[i]
            S0 := Loopright(a, 2) ^ Loopright(a, 13) ^ Loopright(a, 22)
            maj := (a & b) ^ (a & c) ^ (b & c)
            temp2 := S0 + maj
            h = g
            g = f
            f = e
            e = d + temp1
            d = c
            c = b
            b = a
            a = temp1 + temp2
        }
        //将压缩后的尾端加到现有的hash值
        h0 = h0 + a
        h1 = h1 + b
        h2 = h2 + c
        h3 = h3 + d
        h4 = h4 + e
        h5 = h5 + f
        h6 = h6 + g
        h7 = h7 + h
    }
    hashedbytes := [][]byte{IntToByte(h0), IntToByte(h1), IntToByte(h2), IntToByte(h3), IntToByte(h4), IntToByte(h5), IntToByte(h6), IntToByte(h7)}
    hash := []byte{}
    hasharr := [32]byte{}
    for i := 0; i < 8; i ++ {
        hash = append(hash, hashedbytes[i]...)
    }
    copy(hasharr[:], hash[0:32])
    return hasharr
}

func IntToByte(i uint32) []byte {
    bs := make([]byte, 4)
```

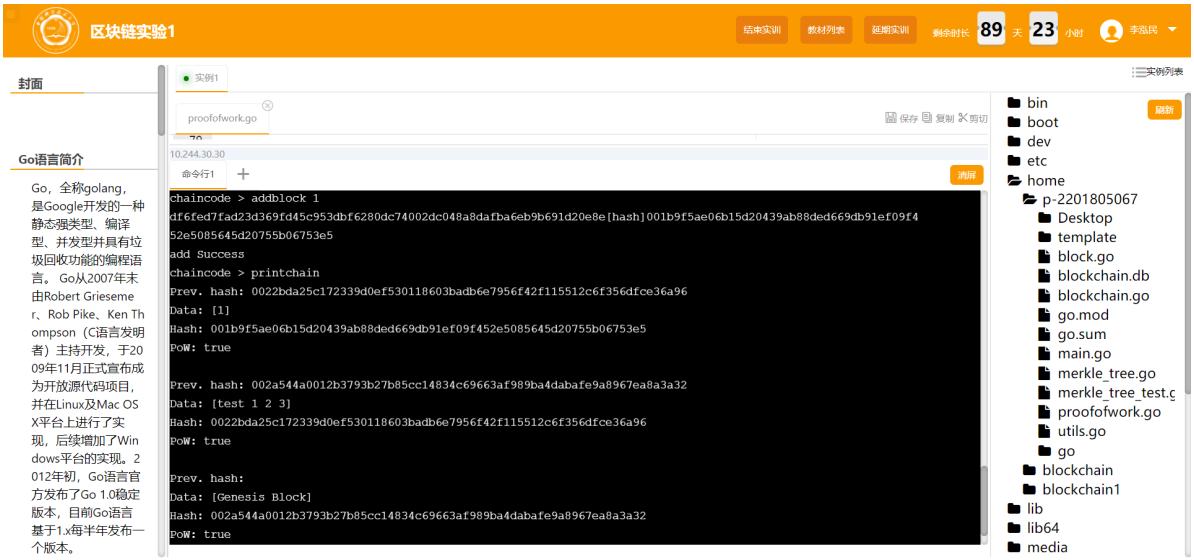
```
    binary.BigEndian.PutUint32(bs, i)
    return bs
}

//循环右移函数
func Loopright(n uint32, d uint) uint32 {
    return (n >> d) | (n << (32 - d))
}
```

实验的结果见后面。

【实验结果】

proofofwork.go

新加入一个block，



再往其中加入更多块：

```
2ea7f2c911e08a25e35fa3b3
dd Success
haincode > printchain
rev. hash: 001a214153aa30d0f7db234c7d0b89b2ef8175ca9165683cf5bb500700a6b2b4
ata: [1234]
ash: 003f6845e41bcb43fddf08c411704e0e9ea14d9b2ea7f2c911e08a25e35fa3b3
oW: true

rev. hash: 001b9f5ae06b15d20439ab88ded669db91ef09f452e5085645d20755b06753e5
ata: [123]
ash: 001a214153aa30d0f7db234c7d0b89b2ef8175ca9165683cf5bb500700a6b2b4
oW: true

rev. hash: 0022bda25c172339d0ef530118603badb6e7956f42f115512c6f356dfce36a96
ata: [1]
ash: 001b9f5ae06b15d20439ab88ded669db91ef09f452e5085645d20755b06753e5
oW: true

rev. hash: 002a544a0012b3793b27b85cc14834c69663af989ba4dabafe9a8967ea8a3a32
```

可以看出,根据计算得到的pow都是true，表示是经过共识得到的新块

bonus

对于abcd字符串进行hash，得到的结果和在其他网站得到的结果一样，可以看出hash运算是正确的。