实验二 区块链共识协议

龚小航 PB18151866

【实验目的及要求】

在 LAB1 的基础上，完成打包区块时的 pow 工作量证明。补充 proofofwork.go 文件，计算合适的随机数 nonce 使区块 hash 必须满足规定的条件才能获得打包区块的写入权限。同时验证 pow 结果的正确性，最后用代码实现 sha256 散列算法。

【实验原理】

LAB1 简单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。这样，我们就可以简单地进行区块的写入和具体的操作了。LAB1 实现的区块链添加区块是相对比较容易的，但是，对于真正的区块链来说添加一个区块需要所有节点达成共识，所以是一个相当复杂的工作。在本次实验中，我们就要进行对于共识部分的补充，来保证区块链的安全性和一致性。

在本实验中，对区块的定义如下所示：

```
// Block keeps block headers
type Block struct {
    Timestamp     int64
    Data          [][]byte
    PrevBlockHash []byte
    Hash          []byte
    Nonce         int
}
```

其中 TimeStamp 表示整个区块的时间戳；Data 二维数组存储了当前区块的数据，包括了多笔交易信息；Prevblockhash 是上一个区块的哈希值，可以从当前区块链的状态中得到；Hash 字段则是本区块对应的哈希值，也是本次实验需要生成的域；Nonce 字段则表示难度随机数。

Pow 工作过程大致如下：将当前区块的信息作如下操作：

sha256(PrevBlockHash ⊕ 当前区块 Merkel 根 hash ⊕ Timestamp ⊕ 目标难度 ⊕ Nonce)

若上述结果满足目标要求则输出当前的 Nonce 以及上述散列结果；若不满足则改变随机数的取值，每次+1 重复计算暴力枚举。若 Nonce 非常大且仍未找到可行解，则反馈给生成新块函数，重新打包区块获取新的时间戳和交易信息再次调用工作量证明。

【实验平台】

GoIDE：goland2021.1.1x64

在线平台：中国科大计算实训平台 training.ustc.edu.cn

【实验步骤】

1、填写 run()函数：

按照实验原理阐述的 pow 工作机制与过程，计算下式

sha256(PrevBlockHash ⊕ 当前区块 Merkel 根 hash ⊕ Timestamp ⊕ 目标难度 ⊕ Nonce)

得到的结果与目标值相比较即可，不满足时改变 Nonce 再重做。最终返回满足条件的 Nonce 值与上述结果，或是 Nonce 返回-1，即没有找到可行解，重新打包区块再做。

```
// Run performs a proof-of-work
// implement
func (pow *ProofOfWork) Run() (int, []byte) {
    nonce := 0

    var buf bytes.Buffer
        buf.Write(pow.block.PrevBlockHash)
        buf.Write(pow.block.HashData())
        buf.Write(Int64ToHex(pow.block.Timestamp))
        buf.Write(Int64ToHex(targetBits))
      tempsum := buf.Bytes()//将上述每次不改变的值的连接用变量记录
    for {
        buf.Reset()
        buf.Write(tempsum)
        buf.Write(Int64ToHex(int64(nonce)))
        hash := sha256.Sum256(buf.Bytes())
        //计算出了哈希值，将其与设定的目标值相比较即可
            tempcmp := new(big.Int)//大数运算包，利用比较函数比较目标值与运算出的哈希值
            tempcmp.SetBytes(hash[:])
        if(tempcmp.Cmp(pow.target) == -1){ //满足条件，直接输出
            pow.block.Hash = hash[:]
            break;
        } else {
            if(nonce < maxNonce){
                nonce = nonce + 1
            } else{ //一直未找到,需要在生成块函数中重新生成时间戳和交易信息，再运行工作量证明
                nonce = -1
                return nonce, hash[:]
            }
        }
    }
    return nonce, pow.block.Hash
}
```

2、验证 pow 结果的正确性：

在 run()函数运行成功之后，当前新块得到了 hash 域与 Nonce 值。再次按照 pow 工作过程检测生成的 Nonce 是否能让新块散列值满足要求。若满足则允许当前节点 打包区块；若不满足则生成失败，当前节点没有权限打包区块。检验函数能检测是否真的付出了工作量，提交的 Nonce 是否真的满足要求。

实现如下所示：

```
// Validate validates block's PoW
// implement
func (pow *ProofOfWork) Validate() bool {
    var buf bytes.Buffer
    buf.Write(pow.block.PrevBlockHash)
    buf.Write(pow.block.HashData())
    buf.Write(Int64ToHex(pow.block.Timestamp))
    buf.Write(Int64ToHex(targetBits))
    buf.Write(Int64ToHex(int64(pow.block.Nonce)))

    hash := sha256.Sum256(buf.Bytes())
    hash_bigint := new(big.Int)
    hash_bigint.SetBytes(hash[:])
    if(hash_bigint.Cmp(pow.target) == -1) { //验证成功
        return true
    } else {
        return false
    }
}
```

3. 实现 sha256 散列算法

哈希散列函数应该满足以下几条性质：

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)

作为区块链的哈希函数还需要满足以下几个性质：

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碰撞。

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。

Sha256 散列算法接收任意长度的数据输入，返回一个 256bit 长的结果，且相似的输入可能得到完全不同的输出，符合区块蓝的哈希函数性质。

```go
package main

import (
    "crypto/sha256"
    "encoding/binary"
    "fmt"
)
func main() {
    a := Sha256([]byte("node1"))//ca12f31b8cbf5f29e268ea64c20a37f3d50b539d891db0c3ebc7c0f66b1fb98a
    aa := sha256.Sum256([]byte("node1"))
    fmt.Println(a)
    fmt.Println(aa)
}



func Sha256(message []byte) [32]byte {
    //初始哈希值,对自然数中前 8 个质数（2,3,5,7,11,13,17,19）的平方根的小数部分取前 32bit 而来
    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
    //这些常量是对自然数中前 64 个质数(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53…)的立方根的小数部分取前 32bit 而来。
    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}
    //信息预处理，附加填充比特
    padded := append(message, 0x80)
    if len(padded) % 64 <= 56 {
        suffix := make([]byte, 56 - (len(padded) % 64))
        padded = append(padded, suffix...)
    } else {
        suffix := make([]byte, 64 + 56 - (len(padded) % 64))
        padded = append(padded, suffix...)
    }
    msgLen := len(message) * 8
    bs := make([]byte, 8)
    binary.BigEndian.PutUint64(bs, uint64(msgLen))
    padded = append(padded, bs...)

    broken := [][]byte{};

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

    //主循环
    for _, chunk := range broken {
        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 := rightRotate(w[i - 15], 7) ^ rightRotate(w[i - 15], 18) ^ (w[i - 15] >> 3)
            s1 := rightRotate(w[i - 2], 17) ^ rightRotate(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

        //应用 SHA256 压缩函数更新 a,b,...,h
        for i := 0; i < 64; i++ {
            S1 := rightRotate(e, 6) ^ rightRotate(e, 11) ^ rightRotate(e, 25)
            ch := (e & f) ^ ((^e) & g)
            temp1 := h + S1 + ch + k[i] + w[i]
            S0 := rightRotate(a, 2) ^ rightRotate(a, 13) ^ rightRotate(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
        }

        h0 = h0 + a
        h1 = h1 + b
        h2 = h2 + c
        h3 = h3 + d
        h4 = h4 + e
        h5 = h5 + f
        h6 = h6 + g
        h7 = h7 + h
    }
    hashBytes := [][]byte{iToB(h0), iToB(h1), iToB(h2), iToB(h3), iToB(h4), iToB(h5), iToB(h6), iToB(h7)}
    hash := []byte{}
    hashArray := [32]byte{}
    for i := 0; i < 8; i ++ {
        hash = append(hash, hashBytes[i]...)
    }
    copy(hashArray[:], hash[0:32])
    return hashArray
}
//类型转换，uint32 转为 []byte
func iToB(i uint32) []byte {
    bs := make([]byte, 4)
    binary.BigEndian.PutUint32(bs, i)
    return bs
}

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

【实验结果】

添加两个区块之后得到的结果，此时 target = 10

```
终端:  Local  +
chaincode > addblock testpow1
add Success
chaincode > addblock pow2
add Success
chaincode > printchain
Prev. hash: 0010eebe38aef6f966d95f86e668a5c38bec712c4ad012e796e81c94116c46f2
Data: [pow2]
Hash: 0001b2346fb07ab7c4715004b827d3a4e367ec6ab8bdf108e135f293b65c9ae7
PoW: true

Prev. hash: 0022bda25c172339d0ef530118603badb6e7956f42f115512c6f356dfce36a96
Data: [testpow1]
Hash: 0010eebe38aef6f966d95f86e668a5c38bec712c4ad012e796e81c94116c46f2
PoW: true

Prev. hash: 002a544a0012b3793b27b85cc14834c69663af989ba4dabafe9a8967ea8a3a32
Data: [test 1 2 3]
Hash: 0022bda25c172339d0ef530118603badb6e7956f42f115512c6f356dfce36a96
PoW: true

Prev. hash:
Data: [Genesis Block]
Hash: 002a544a0012b3793b27b85cc14834c69663af989ba4dabafe9a8967ea8a3a32
PoW: true

chaincode >
结构
收藏夹
▶ 运行  TODO  问题  终端
```

添加两个区块之后得到的结果，此时 target = 24

```
终端:  Local  +
chaincode > addblock pow24_1
add Success
chaincode > addblock pow24_2
add Success
chaincode > printchain
Prev. hash: 000000fa3d1ef0e55114ed6b519d6b6b511adc18a00e0c10971a3c30937c1b31
Data: [pow24_2]
Hash: 0000001030383afcefd6d9ce69ee5f83845a53cd72cca5116872f90f7294dc7f
PoW: true

Prev. hash: 0022bda25c172339d0ef530118603badb6e7956f42f115512c6f356dfce36a96
Data: [pow24_1]
Hash: 000000fa3d1ef0e55114ed6b519d6b6b511adc18a00e0c10971a3c30937c1b31
PoW: true

Prev. hash: 002a544a0012b3793b27b85cc14834c69663af989ba4dabafe9a8967ea8a3a32
Data: [test 1 2 3]
Hash: 0022bda25c172339d0ef530118603badb6e7956f42f115512c6f356dfce36a96
PoW: false

Prev. hash:
Data: [Genesis Block]
Hash: 002a544a0012b3793b27b85cc14834c69663af989ba4dabafe9a8967ea8a3a32
PoW: false

chaincode >
结构
收藏夹
▶ 运行  TODO  问题  终端
```

Sha256 验证程序的运行结果如下所示：

```
运行:  go build scratch.go
<4 go 设置调用>
[202 18 243 27 140 191 95 41 226 104 234 100 194 10 55 243 213 11 83 157 137 29 176 195 235 199 192 246 107 31 185 138]
[202 18 243 27 140 191 95 41 226 104 234 100 194 10 55 243 213 11 83 157 137 29 176 195 235 199 192 246 107 31 185 138]

进程 已完成，退出代码为 0
```

各验证结果均符合预期。