lab2-区块链共识协议

实验目的及要求

- 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
- 完成`proofofwork.go`
  - Run() pow计算部分
  - Validate() pow结果的验证工作
- 完成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。

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。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）。
- 在这种机制的作用下，新生成的区块能够被安全地加入到区块链中，它维护了整个区块链数据库的稳定性。值得注意的是，完成了这个工作的人必须要证明这一点，即他必须要证明他的确完成了这些工作。
- 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

- 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。
- **在比特币中，这个工作就是找到一个块的哈希**，同时这个哈希满足了一些必要条件。这个哈希，也就充当了证明的角色。因此，寻求证明（寻找有效哈希），就是矿工实际要做的事情。
- POW要求发起者进行一定量的运算，消耗计算机一定的时间。

哈希函数

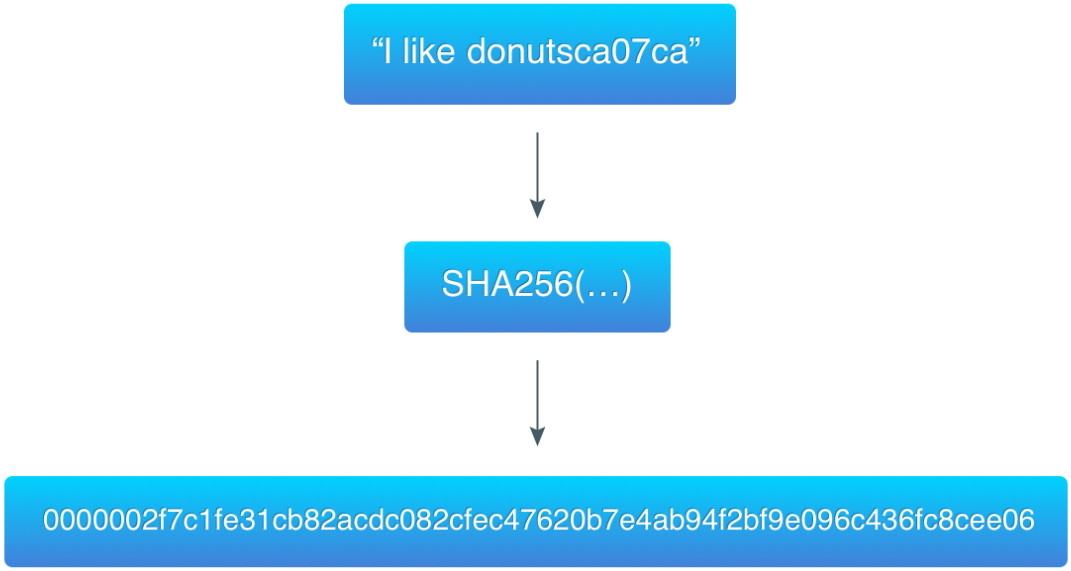
- 哈希函数是输入数据进行一种函数计算，获取一个独特的表示。哈希函数需要满足如下的性质：
    1. 可以接受任意大小的输入
    2. 输出是固定长度的
    3. 计算哈希的过程相对是比较简单的，时间都在O(n)
- 哈希函数被广泛用于检测数据的一致性。软件提供者常常在除了提供软件包以外，还会发布校验和。当下载完一个文件以后，可以用哈希函数对下载好的文件计算一个哈希，并与作者提供的哈希进行比较，以此来保证文件下载的完整性。
- 对于区块链的哈希函数，也需要满足一定优秀的性质：
    1. 原始数据不能直接通过哈希值来还原，哈希值是没法解密的。
    2. 特定数据有唯一确定的哈希值，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

- 在区块链中，哈希被用于保证一个块的一致性。哈希算法的输入数据包含了前一个块的哈希，因此使得不太可能（或者，至少很困难）去修改链中的一个块：因为如果一个人想要修改前面一个块的哈希，那么他必须要重新计算这个块以及后面所有块的哈希。
- 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



难度控制

本次实验中，我们选用了一个固定的难度值来进行计算。10位的难度值意味着我们需要获取一个**1<<(255-10)**小的数。

```
const targetBits = 10 //难度值
```

这里的 10 指的是算出来的哈希前 10 位必须是 0，这一点从最后的输出可以看出来

Pow结构

- ```
  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)`。
  - 需要将哈希值域目标进行比较：先把哈希转换成一个大整数，然后检测它是否小于目标。
- `第一个区块对应的hash`是一个为空的值。
- 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（注意 Go 语言的跨文件调用不需要 import，只需在命名时大写开头即可）
- 创建新区块
  ```

```go
// NewProofOfWork builds and returns a ProofOfWork
func NewProofOfWork(b *Block) *ProofOfWork {
    // 初始化为1
    target := big.NewInt(1)
    // 256 bits,对应 32 bytes
    // 左移 256 - targetBits 位
    target.Lsh(target, uint(256-targetBits))

    pow := &ProofOfWork{b, target}

    return pow
}
```

实验平台

- Windows 10 Professional
- GoLand
    - go version：go1.16.3 windows/amd64

实验步骤

数据准备

- 定义 ProofOfWork 的方法 prepareData 方法（仅在该文件内使用，所以 p 小写），用来将**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，传入随机数**转化为 1 维切片（通过 byte.Join 方法）

```go
// 定义结构体 prepareData 的方法(只在该文件内使用)
func (pow *ProofOfWork) prepareData (nonce int) []byte {
    // Join 结果得到1维切片
    data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
            pow.block.HashData(),
            // 大写字母开头的变量 or方法是暴露给其他包用的
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
    )
    return data
}
```

Run

- 通过一个 for 循环来进行工作量证明
    - `maxNonce` 对这个循环进行了限制（等于 `math.MaxInt64`），避免 `nonce` 可能出现的溢出
    - 循环主要做以下工作
        - 调用 `prepareData` 方法准备数据
        - 用 `crypto/sha256` 中的 SHA-256 算法对数据进行哈希

- 将哈希值转化为一个大整数 hashInt
- 比较大整数和目标值

```
// 寻找有效哈希
// Run performs a proof-of-work
// implement
func (pow *ProofOfWork) Run() (int, []byte) {
    nonce := 0
    var hashInt big.Int
    var hash [32]byte

    fmt.Printf("Mining the block, containing data :\"%s\"\n",
pow.block.Data)
    // 类似 While 循环, nonce < maxNonce 避免溢出
    for nonce < maxNonce {
        // 准备数据
        data := pow.prepareData(nonce)
        // 计算Hash值
        hash = sha256.Sum256(data)
        // SetBytes将hash解释为大端无符号整数的字节，并将hashInt设置为该值
        // 这是为了格式统一,便于后续比较
        hashInt.SetBytes(hash[:])
        // 如果 hashInt 小于目标值
        if hashInt.Cmp(pow.target) == -1 {
            // \r 为回车符
            fmt.Printf("\r%x", hash)
            break
        } else {
            nonce++
        }
    }
    fmt.Print("\n\n")
    // 修改，因为是通过调用 Run 函数后通过返回值来设置 Block 的hash
    // 所以这里没必要返回 pow.block.Hash
    return nonce, hash[:]
}
```

Validate

- 对得到的哈希值进行验证，主要是判断是否小于目标值。也就是说得到的哈希值前 10bits 是否都是0

```
// 只要哈希小于目标就是有效工作量
// Validate validates block's PoW
// implement
func (pow *ProofOfWork) Validate() bool {
    var hashInt big.Int
    // 准备数据
    data := pow.prepareData(pow.block.Nonce)
    // 调用 SHA-256 加密算法
    hash := sha256.Sum256(data)
    // 转化为大整数
    hashInt.SetBytes(hash[:])
    // 验证是否小于目标值，Cmp == -1
```

```
    isValid := (hashInt.Cmp(pow.target) == -1)
    return isValid
}
```

实验结果

- 运行`go run .`，仍然使用实验1中创建区块的过程

```
chaincode > printchain
Prev. hash: 0008ceedf3e3b2899a1878c1ed4edfef2223b8601edddf370c26e9cdbb280140
Data: [123]
Hash: 003c51d201486c03cd70f62b7f3a9bdc5f875bd2a84758814699219cdbb9af6b
PoW: true

Prev. hash: 0027d55ca56d3c367deebb7c8cf0a7a74298b04b843b18dab8de2283706ca8f1
Data: [pqz]
Hash: 0008ceedf3e3b2899a1878c1ed4edfef2223b8601edddf370c26e9cdbb280140
PoW: true

Prev. hash:
Data: [Genesis Block]
Hash: 0027d55ca56d3c367deebb7c8cf0a7a74298b04b843b18dab8de2283706ca8f1
PoW: true
```

- 可以看到每个区块都经过了工作量证明，即每个区块的 Hash 值的前 10 bits（图中是以 byte 为单位的） 都是0，对应的 PoW 为 true，

Bonus-SHA256

- SHA-256 算法输入报文的最大长度不超过2^64 bit，输入按512-bit 分组进行处理，产生的输出是一个256-bit 的报文摘要。
- SHA256算法主流程分三大模块：
  - 常量的初始化
  - 信息预处理
  - 使用到的逻辑运算。

满足区块链哈希函数的性质

- 对于区块链的哈希函数，需要满足一定优秀的性质：
  - 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
    - **单向性**：如果在输入序列中嵌入密码，那么任何人在不知道密码的情况下都不能产生正确的散列值，从而保证了其安全性。
    - SHA 将输入流按照每块 512 位进行分块，并产生 160 位的被称为信息认证代码或信息摘要的输出。
  - 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。
    - **Hash碰撞**：一个哈希位有 0 和 1 两个可能值，则每一个独立的哈希值通过位的可能值的数量对于 SHA-256，有 2^{256} 组合，这是一个庞大的数值。哈希值越大，碰撞的机率就越小。
    - 每个散列算法，包括安全算法，都会发生碰撞，而 SHA-1 的大小结构发生碰撞的机率比较大，所以 SHA-1 被认为是不安全的。
  - 修改输入数据一比特的数据，会导致结果完全不同。

        - 通过 SHA-256 的算法可知，数据经过了多轮迭代、非线性及其他各种复杂运算，所以修改输入数据一比特的数据会导致结果完全不同
    - 没有除了穷举以外的办法来确定哈希值的范围

常量的初始化

- **SHA256 中用到两种常量**
    - 这些常量的作用是和数据源进行计算，增加数据的加密性
    - 如果常量是一些如 1、2、3 之类的整数，没什么加密可言了，所以需要这些常量很复杂，生成的规则是：对自然数中前 8 个（或 64 个）质数（2、3、5、7、11、13、17、19）的平方根的小数部分取前 32 bit（在后面的映射的过程中会用到这些常量）。
    - 8 个哈希初值=>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
    - 64 个哈希常量=>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、59、61、67、71、73、79、83、89、97...）的立方根的小数部分取前 32 bit

信息预处理

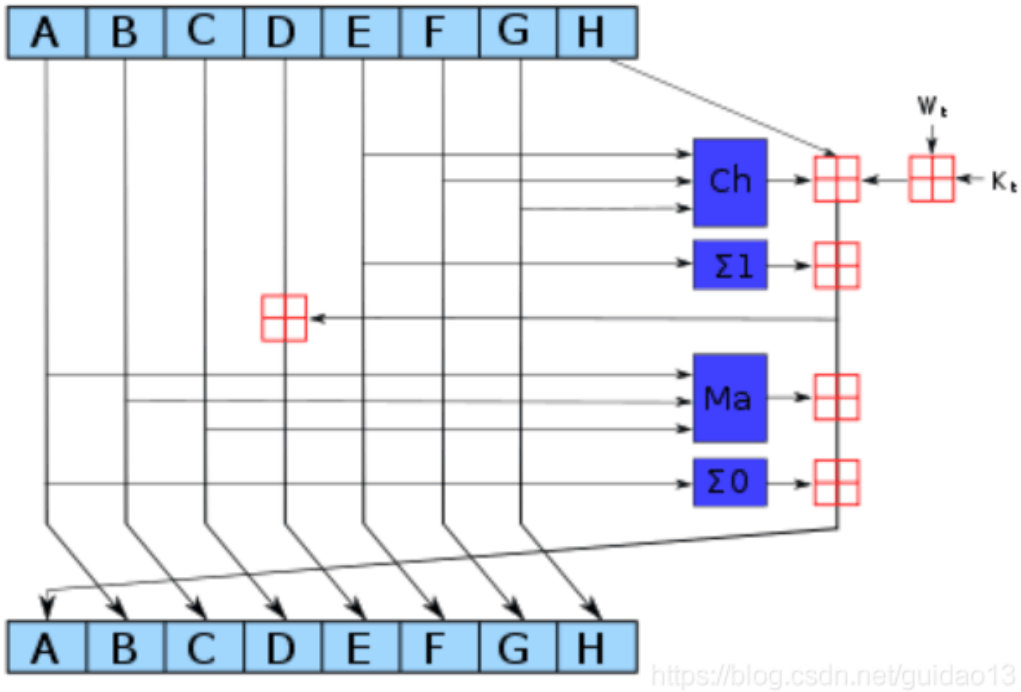
- 目的：让整个消息满足指定的结构，从而处理起来可以统一化、格式化
- **附加填充比特**：对报文进行填充使报文长度与448 模512 同余（长度=448 mod 512），填充的比特数范围是1 到512，填充比特串的最高位为1，其余位为0。
    - 即先在报文后面加一个 1，再加多个0，直到长度满足 length mod 512=448
    - 为什么是448？因为448+64=512（第二步会加上一个64bit的原始报文的长度信息）
    - 需要注意即使长度已经满足对 512 取模后余数是 448，补位也必须要进行，这时要填充 512 个比特。所以，填充是至少补一位，最多补 512 位。
- **附加长度值：** 将用64-bit 表示的初始报文（填充前）的位长度附加在上一步的结果后（低位字节优先）。得到512bits的报文
- **初始化缓存**：使用一个256-bit 的缓存来存放该散列函数的中间及最终结果。 该缓存表示为（每个为32bits，共8个）

```
const (
    init0     = 0x6A09E667
    init1     = 0xBB67AE85
    init2     = 0x3C6EF372
    init3     = 0xA54FF53A
    init4     = 0x510E527F
    init5     = 0x9B05688C
    init6     = 0x1F83D9AB
    init7     = 0x5BE0CD19
)
```

逻辑运算

- **处理512bits（16 个字）报文分组序列**：该算法使用了六种基本逻辑函数，由64步迭代运算组成。每步都以256-bit 缓存值*ABCDEFGH* 作为输入，然后更新缓存内容。
    - 六种基本逻辑函数
        - ∧ → 按位“与”
        - ¬ → 按位“补”
        - ⊕ → 按位“异或”
        - Sn → 右移 n 个 bit

    - Rn → 循环右移 n 个 bit
- 在 SHA256 算法中的最小运算单元称为“字”（Word），一个字是 32 位（byte），就是 4 个字节（bit），256 个字节（bit）就是 64 个字（word）。
- **数据分解**：将原始数据分解成 512-bit 大小的块，例如，消息 M 可以被分解为 n 个块，于是整个算法需要做的就是完成 n 次迭代，n 次迭代的结果就是最终的哈希值，即 256bit 的数字摘要。
- 每次迭代进行的映射用 $Map(H_{i-1}) = H_i$表示
    - 摘要的初始值 H0，经过第一个映射后，得到 H1，即完成了第一次迭代，H1 经过第二次映射得到 H2，……，依次处理，最后得到 Hn，Hn 即为最终的 256-bit 消息摘要。
- 每步使用一个32-bit 常数值Kt 和一个32-bit Wt
- 映射 $Map(H_{i-1}) = H_i$包含了 64 次加密循环，即进行 64 次加密循环即可完成一次迭代
- 维基百科图解



- 说明：
    - 深蓝色方块是事先定义好的非线性函数，里面是逻辑运算。
    - ABCDEFGH 一开始分别是八个初始值，即初始化的 8 个常量。
    - Kt 是第 t 个密钥，密钥从 64 个常量里取。
    - Wt 是本区块产生第 t 个 word。
    - 原消息被切成固定长度的区块，对每一个区块，产生 n 个 word，透过重复运作循环 n 次对 ABCDEFGH 这八个工作区块循环加密。最后一次循环所产生的八段字符串合起来即是此区块对应到的散列字符串。
    - 若原消息包含数个区块，则最后还要将这些区块产生的散列字符串加以混合才能产生最后的散列字符串

代码

```
package main
import(
    "fmt"
)

const Size = 32
```

```
const (
    chunk     = 64
    init0     = 0x6A09E667
    init1     = 0xBB67AE85
    init2     = 0x3C6EF372
    init3     = 0xA54FF53A
    init4     = 0x510E527F
    init5     = 0x9B05688C
    init6     = 0x1F83D9AB
    init7     = 0x5BE0CD19
)

var _K = []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,
}

// digest represents the partial evaluation of a checksum.
type digest struct {
    h      [8]uint32      // 8位 u32 数组
    x      [chunk]byte    // 64位字节数组
    nx     int
    len    uint64
}

func (d *digest) Reset() {

    d.h[0] = init0
    d.h[1] = init1
    d.h[2] = init2
    d.h[3] = init3
    d.h[4] = init4
    d.h[5] = init5
    d.h[6] = init6
    d.h[7] = init7
    d.nx = 0
    d.len = 0
}

func putUint32(x []byte, s uint32) {
    _ = x[3]
    x[0] = byte(s >> 24)
    x[1] = byte(s >> 16)
    x[2] = byte(s >> 8)
    x[3] = byte(s)
}

func putUint64(x []byte, s uint64) {
    _ = x[7]
    x[0] = byte(s >> 56)
    x[1] = byte(s >> 48)
    x[2] = byte(s >> 40)
    x[3] = byte(s >> 32)
```

```
    x[4] = byte(s >> 24)
    x[5] = byte(s >> 16)
    x[6] = byte(s >> 8)
    x[7] = byte(s)
}

func (d *digest) Write(p []byte) (nn int, err error) {
    nn = len(p)
    d.len += uint64(nn)
    if d.nx > 0 {
        n := copy(d.x[d.nx:], p)
        d.nx += n
        if d.nx == chunk {
            blockGeneric(d, d.x[:])
            d.nx = 0
        }
        p = p[n:]
    }
    if len(p) >= chunk {
        n := len(p) &^ (chunk - 1)
        blockGeneric(d, p[:n])
        p = p[n:]
    }
    if len(p) > 0 {
        // 将 p 复制到d.x[:]
        d.nx = copy(d.x[:], p)
    }
    return
}

func (d *digest) checkSum() [Size]byte {
    len := d.len
    // Padding. Add a 1 bit and 0 bits until 56 bytes mod 64.
    // 注意这里是以字节为单位
    var tmp [64]byte
    tmp[0] = 0x80
    if len % 64 < 56 {
        d.Write(tmp[0 : 56-len%64])
    } else {
        d.Write(tmp[0 : 64+56-len%64])
    }

    // Length from bytes to bits.
    len <<= 3
    putUint64(tmp[:], len)
    d.Write(tmp[0:8])

    if d.nx != 0 {
        panic("d.nx != 0")
    }

    var digest [Size]byte

    putUint32(digest[0:], d.h[0])
    putUint32(digest[4:], d.h[1])
    putUint32(digest[8:], d.h[2])
    putUint32(digest[12:], d.h[3])
    putUint32(digest[16:], d.h[4])
```

```
    putUint32(digest[20:], d.h[5])
    putUint32(digest[24:], d.h[6])
    putUint32(digest[28:], d.h[7])

    return digest
}

func blockGeneric(dig *digest, p []byte) {
    var w [64]uint32
    h0, h1, h2, h3, h4, h5, h6, h7 := dig.h[0], dig.h[1], dig.h[2], dig.h[3],
dig.h[4], dig.h[5], dig.h[6], dig.h[7]
    for len(p) >= chunk {

        for i := 0; i < 16; i++ {
            j := i * 4
            w[i] = uint32(p[j])<<24 | uint32(p[j+1])<<16 | uint32(p[j+2])<<8 |
uint32(p[j+3])
        }
        for i := 16; i < 64; i++ {
            v1 := w[i-2]
            t1 := (v1>>17 | v1<<(32-17)) ^ (v1>>19 | v1<<(32-19)) ^ (v1 >> 10)
            v2 := w[i-15]
            t2 := (v2>>7 | v2<<(32-7)) ^ (v2>>18 | v2<<(32-18)) ^ (v2 >> 3)
            w[i] = t1 + w[i-7] + t2 + w[i-16]
        }

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

        for i := 0; i < 64; i++ {
            t1 := h + ((e>>6 | e<<(32-6)) ^ (e>>11 | e<<(32-11)) ^ (e>>25 |
e<<(32-25))) + ((e & f) ^ (^e & g)) + _K[i] + w[i]

            t2 := ((a>>2 | a<<(32-2)) ^ (a>>13 | a<<(32-13)) ^ (a>>22 | a<<(32-
22))) + ((a & b) ^ (a & c) ^ (b & c))

            h = g
            g = f
            f = e
            e = d + t1
            d = c
            c = b
            b = a
            a = t1 + t2
        }

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

        p = p[chunk:]
    }
```

```
    dig.h[0], dig.h[1], dig.h[2], dig.h[3], dig.h[4], dig.h[5], dig.h[6],
dig.h[7] = h0, h1, h2, h3, h4, h5, h6, h7
}

func Sum256(data []byte) [Size]byte {
    var d digest
    d.Reset()
    d.Write(data)
    return d.checkSum()
}

func main() {

    sum := Sum256([]byte("hello world\n"))
    fmt.Printf("%x\n", sum)
}
```

结果

验证“hello world\n”哈希值

```
a948904f2f0f479b8f8197694b30184b0d2ed1c1cd2a1ec0fb85d299a192a447
```

和调用 crypto/sha256 的结果对比

```
package main

import (
    "crypto/sha256"
    "fmt"
)

func main(){
    s := "hello world\n"

    h := sha256.New()
    h.Write([]byte(s))
    bs := h.Sum(nil)

    fmt.Printf("origin: %s sha256 hash: %x\n", s, bs)
}
```

```
origin: hello world
 sha256 hash: a948904f2f0f479b8f8197694b30184b0d2ed1c1cd2a1ec0fb85d299a192a447
```

可以看出二者的结果是一致的

实验总结

- 通过本次实验，我对区块链的共识协议，尤其是工作量证明 Pow 有了更加深入的理解，节点通过自身算力，需要计算出一个小于目标值的哈希，以此来决定谁可以写入区块
- 对 Go 语言的特性有了更加深入的理解

参考

https://blog.csdn.net/luckydog612/article/details/80547758

https://github.com/liuchengxu/blockchain-tutorial/blob/master/content/SUMMARY.md