区块链 实验一

PB20000096 潘廷岳

Part_1 实验内容

①完成SHA256算法实现

②实现Merkle树的构建

③搭建简单的区块链结构

Part_2 实验过程

本次实验，主要完成四部分：

- sha256.go/mySha256 编写自己的sha256函数
- Merkle_tree.go/NewMerkleTree merkle树的构建
- Merkle_tree.go/NewMerkleNode merkle树节点的构建
- 使用blockchain.go/addblock 添加区块

完成这四部分，能够实现对区块的简单运行；而对于区块的工作验证等交由后面的实验完成。

Part_3 代码展示与说明

①sha256

调用此函数，传入需hash的消息，返回sha256算法的hash结果

```
//对消息的补充
var msg_comp []byte = msg_complement(message);

//对消息块的拓展
var msg_chunks [][64] uint32 = Msg_chunk_extend( Msg_div(msg_comp) )
msg_chunk_num := len(msg_chunks)

//按照公式，对message进行hash
for i := 0; i < msg_chunk_num; i += 1 {
var a, b, c, d, e, f, g, h uint32 = h0, h1, h2, h3, h4, h5, h6, h7

    for j := 0; j < 64; j += 1 {
        s1 := rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25)
        ch := (e & f) ^ ((^e) & g)
        t1 := h + s1 + ch + k[j] + msg_chunks[i][j]
        s0 := rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22)
        maj := (a & b) ^ (a & c) ^ (b & c)
        t2 := s0 + maj

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

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

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

sha256data := [32]byte{}

//对消息拼接，输出结果
sha256data = append_to_bytes([8] uint32 {h0, h1, h2, h3, h4, h5, h6, h7}) //No Problem
return sha256data
```

主要流程：

Ⅰ）对消息进行补充，按要求补充至长度为 512 的整数倍

Ⅱ）划分消息块并进行块拓展（ w[16] -> w[64] ）

Ⅲ）按照公式，对message进行hash

Ⅳ）对 h0 ~ h7 进行拼接并返回拼接结果

对应功能函数介绍：

Ⅰ）消息扩充函数

```
func msg_complement(message []byte) []byte{

    msg_len := len(message)
    msg_len_rmd := (msg_len * 8) % 512
    msg_extend_len := 0
    if msg_len_rmd < 448 {
        msg_extend_len = 448 - msg_len_rmd
    } else {
        msg_extend_len = (512 - msg_len_rmd) + 448
    }  //计算需要拓展的位数

    msg_extend_len /= 8;
    var byte_extend []byte = make([]byte, msg_extend_len + 8)
    byte_extend[0] = 0x80

    for i := 1 ; i < msg_extend_len ; i++ {
        byte_extend[i] = 0x00 // + 8'b00000000
    }

    msg_len_64 := uint64 (msg_len << 3) // 消息长度64位存储,按bit计算长度
    for i := 7 ; i >= 0 ; i-- {
        byte_extend[msg_extend_len + i] = uint8(msg_len_64)
        msg_len_64 >>= 8
    } //附加长度

    return append(message, byte_extend...)
}
```

Ⅱ）消息块划分函数

```
func Msg_div(message []byte) [][16] uint32{
    msg_len := len(message)
    chunk_link := make([][16]uint32 , msg_len >> 6)
    var chunk_num int = 0
    for i := 0 ; i < msg_len - 1; i += 64 {
        var subchunk_num int = 0
```

```
        for j := i ; j < i + 64 ; j += 4 {
            chunk_link [chunk_num][subchunk_num] += uint32(message[j]) << 24
            chunk_link [chunk_num][subchunk_num] += uint32(message[j + 1]) << 16
            chunk_link [chunk_num][subchunk_num] += uint32(message[j + 2]) << 8
            chunk_link [chunk_num][subchunk_num] += uint32(message[j + 3])
            subchunk_num += 1
        } //每块大小为 64 * 8 位

        chunk_num += 1
    }

    return chunk_link
}
```

Ⅲ）块拓展函数

```
func Msg_chunk_extend(msg_chunk [][16] uint32) [][64] uint32{
    msg_chunk_sum := len(msg_chunk)

    w := make([][64]uint32 , msg_chunk_sum)


    var s0,s1 uint32
    for i := 0 ; i < msg_chunk_sum ; i += 1 {
        for j := 0 ; j < 64 ; j += 1 {
            if j < 16 {
                w[i][j] = msg_chunk[i][j]
            } else {
                s0 = rightrotate(w[i][j - 15] , 7) ^ rightrotate(w[i][j - 15] , 18) ^ (w[i][j - 15] >> 3)
                s1 = rightrotate(w[i][j - 2] , 17) ^ rightrotate(w[i][j - 2] , 19) ^ (w[i][j - 2] >> 10)
                w[i][j] = w[i][j - 16] + s0 + w[i][j - 7] + s1
            }
        }
    }
    return w
}
```

Ⅳ）hash结果拼接函数

```
func append_to_bytes (msg_uint32 [8]uint32) [32]byte {
    msg_bytes := [32]byte{}

    for i := 0; i < 8; i += 1 {
        msg_bytes [i << 2] = byte((msg_uint32[i] >> 24) & uint32(0xff))
        msg_bytes [(i << 2) + 1] = byte((msg_uint32[i] >> 16) & uint32(0xff))
        msg_bytes [(i << 2) + 2] = byte((msg_uint32[i] >> 8) & uint32(0xff))
        msg_bytes [(i << 2) + 3] = byte(msg_uint32[i] & uint32(0xff))
    }

    return msg_bytes
}
```

Ⅴ）位右旋函数

```
func rightrotate(msg uint32 , rt_bytes uint32) uint32{
        return (msg >> rt_bytes) | (msg << (32 - rt_bytes))
}
```

②Merkel树

结构定义

```
// MerkleTree represent a Merkle tree
type MerkleTree struct {
    RootNode *MerkleNode
}

// MerkleNode represent a Merkle tree node
type MerkleNode struct {
    Left  *MerkleNode
    Right *MerkleNode
    Data  []byte
}
```

新建Merkel节点

```
func NewMerkleNode(left, right *MerkleNode, data []byte) *MerkleNode {
    node := MerkleNode{}

    if left == nil && right == nil {
        hash_256 := mySha256(data)
        node.Data = hash_256[:]  //如果是叶子，则将消息hash结果作为Data属性的值
        } else {
            Offs_data := append(left.Data, right.Data...)
            hash_256 := mySha256(Offs_data)
            node.Data = append(node.Data,hash_256[:]...)
        } //否则将两个子节点的.Data进行拼接，并以拼接后结果的hash值作为当前节点的Data属性的值

    node.Left = left
    node.Right = right
    return &node
}
```

新建Merkel树

这里采用了逐层向上构建树的方法，最终得到Merkel根节点并将其返回

```
func NewMerkleTree(data [][]byte) *MerkleTree { //ToDo

    //var node = MerkleNode{nil, nil, data[0]}

    var node_que []MerkleNode //MerkleNode节点队列

    if len(data) & 1 != 0 {
        data = append(data, data[len(data) - 1]) } // 保证叶结点偶数个

        for _, data_travel := range data {
            node := NewMerkleNode(nil, nil, data_travel) //data => TreeNode
            node_que = append(node_que, *node)
        }

        data_len := len(data)
        for i := 0; i < data_len ; i++ { //调整大小
            var next_node_que []MerkleNode

            for j := 0; j < len(node_que); j += 2 {
                node := NewMerkleNode(&node_que[j], &node_que[j+1], nil)
                next_node_que = append(next_node_que, *node)
                }// 逐层构建Merkle树

                node_que = next_node_que
```

```
                if len(node_que) == 1 {
                    break
                } //只剩根节点，跳出循环
            }

    var mTree = MerkleTree{&node_que[0]} //节点队列首个，必为根节点
    return &mTree
}
```

③数据库管理

添加新区块，并与数据库进行信息交互

```
func (bc *Blockchain) AddBlock(data []string) {
    var PreHash []byte // 获取上一区块内容

    err := bc.db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        PreHash = b.Get([]byte("l")) // 访问数据库，获取上一个块的哈希
        return nil
    })
    if err != nil {
        log.Panic(err)
    } //错误中断语句

    newBlock := NewBlock(data, PreHash) // 添加区块
    err = bc.db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        err := b.Put(newBlock.Hash, newBlock.Serialize())  //Hash -> key,序列化后的块信息 -> value，载入数据库
        if err != nil {
            log.Panic(err)
        }

        err = b.Put([]byte("l"), newBlock.Hash) //维护字段[l],存储最新hash值
        if err != nil {
            log.Panic(err)
        }

        bc.tip = newBlock.Hash //更新tip
        return nil
    })
}
```

④addblock指令

Ⅰ）处理addblock指令：

```
    Name:    "addblock",
    Aliases: []string{"a"},
    Usage:   "addblock BLOCK_DATA    - add a block to the blockchain",
    Action: func(c *cli.Context) error {
        data := c.Args()
        bc.AddBlock(data) //生成新的区块，并放入数据库中
        fmt.Println("add Success")
        return nil
    },
```

Ⅱ）AddBlock函数

该函数将通过NewBlock(data, PreHash)生成的新区块存于数据库中，并更新链信息。具体代码见上文。

Ⅲ）NewBlock函数

该函数依据传入的信息及上一区块的hash，生成新区块并返回给调用者。

```
func NewBlock(datas []string, prevBlockHash []byte) *Block {
    blockData := [][]byte{}
    for _, data := range datas {
        blockData = append(blockData, []byte(data))
    }

    block := &Block{time.Now().Unix(), blockData, prevBlockHash, []byte{}, 0}
    pow := NewProofOfWork(block)
    nonce, _ := pow.Run()

    hash := append(prevBlockHash, '1')
    //可以发现，这里对于当前区块hash的计算没有使用到datas【】，故推断本次实验结果应当不随信息改变而改变

    block.Hash = hash[:]
    block.Nonce = nonce

    return block
}
```

⑤printchain指令

Ⅰ）处理printchain指令

```
    Name:    "printchain",
    Aliases: []string{"p"},
    Usage:   "printchain",
    Action: func(c *cli.Context) error {
        bci := bc.Iterator()
        for {
            block := bci.Next() //

            fmt.Printf("Prev. hash: %x\n", block.PrevBlockHash)
            fmt.Printf("Data: %s\n", block.Data)
            fmt.Printf("Hash: %x\n", block.Hash)
            pow := NewProofOfWork(block)
            fmt.Printf("PoW: %s\n", strconv.FormatBool(pow.Validate()))
            fmt.Println()

            if len(block.PrevBlockHash) == 0 {
                break
            }
        }
        return nil
    },
```

Ⅱ）.Next方法的实现

```
func (i *BlockchainIterator) Next() *Block {
    var block *Block
```

```
    err := i.db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        encodedBlock := b.Get(i.currentHash)
        block = DeserializeBlock(encodedBlock)

        return nil
    })

    if err != nil {
        log.Panic(err)
    }

    i.currentHash = block.PrevBlockHash
    //每次将currentHash置为当前block的PreBlock's Hash,方便下一次直接访问PreBlock

    return block
}
```

不难看出，结合.Next方法，该指令作用是从区块链链尾（最新数据库信息存储）开始，一路追溯至链头，并将沿路的块信息输出。

Part_4 实验结果展示

①Sha256 运行测试

运行

```
go test -v sha256_test.go sha256.go
```

得到结果

```
PS C:\Users\pty\blockchain-lab\lab1\template> go test -v sha256_test.go sha256.go
=== RUN   TestSha256
--- PASS: TestSha256 (0.00s)
PASS
ok      command-line-arguments  0.692s
PS C:\Users\pty\blockchain-lab\lab1\template>
```

②MerkelTree 运行测试

运行

```
go test -v merkle_tree_test.go merkle_tree.go sha256.go
```

得到结果

```
PS C:\Users\pty\blockchain-lab\lab1\template> go test -v merkle_tree_test.go merkle_tree.go sha256.go
=== RUN   TestNewMerkleNode
--- PASS: TestNewMerkleNode (0.00s)
=== RUN   TestNewMerkleTree
--- PASS: TestNewMerkleTree (0.00s)
PASS
ok      command-line-arguments  0.709s
PS C:\Users\pty\blockchain-lab\lab1\template>
```

这里包含了MerkelTree 和 MerkelNode的测试结果，均通过

③addblock & printchain 使用测试

首先将 blockchain.db 数据库文件删除，然后依此运行以下指令：

```
> go run .
> addblock
> printchain
> addblock aaa
> printchain
```

得到以下结果：

```
PS C:\Users\pty\blockchain-lab\lab1\template> go run .
No existing blockchain found. Creating a new one...
chaincode > addblock
add Success
chaincode > printchain
Prev. hash: 31
Data: []
Hash: 3131
PoW: true

Prev. hash:
Data: [Genesis Block]
Hash: 31
PoW: true

chaincode > addblock aaa
add Success
chaincode > printchain
Prev. hash: 3131
Data: [aaa]
Hash: 313131
PoW: true

Prev. hash: 31
Data: []
Hash: 3131
PoW: true

Prev. hash:
Data: [Genesis Block]
Hash: 31
PoW: true

chaincode >
```

从结果看出，初始检测到区块链不存在，则调用 NewBlockchain() 函数创建新链；而后两次addblock，由于上文提到的 NewBlock() 函数并未考虑新块 data 对新块hash结果的影响，故hash结果不随 data 改变而改变，每次结果均在上次 hash 基础上于末尾连接上 0x31 。 printchain 也如预期，从链尾块开始，直至链头块，依次输出各区块信息。

Part_4 实验总结

- 为完成本次实验，笔者结合资料与助教提供的代码框架，进行了Go语言的初步学习。在学习过程中，发现Go语言对于变量的类型定义与匹配要求较严格，与C语言类似；而其对于变量的定义格式则为 变量名 类型 ，这又与C语言相反。同时，在学习过程中，笔者发现Go语言对于内存开销相对于C语言放开了限制，例如多值拷贝的实现。并且，Go语言只有一种参数传递规则，那就是值拷贝，这点又决定了其无法像 C 那样使用引用传递。
- 在实验过程中，笔者对sha256的实现、Merkel树的构建及区块链的基本工作流程、区块链的底层基础架构等有了一定的理解，这些都能帮助笔者更好地理解区块链结构。