区块链实验一实验报告

PB19071405 王昊元

2022 年 04 月 10 日

1 实验目的

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

2 具体实现

1. *SHA256* 算法
   算法中涉及到几种运算符，如 `rightRotate`、`Sigma0`、`ch`、`maj` 等，通过新定义的函数实现，下面仅展示核心函数 `mySha256` 相关代码，该函数按照官方算法实现。

```go
func mySha256(message []byte) [32]byte {
    //前八个素数平方根的小数部分的前面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)

    //自然数中前面64个素数的立方根的小数部分的前32位
    k := [64]uint32{
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0
            x59f111f1, 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}

    sha256data := [32]byte{}

    // l + 1 + m = 448 mod 512
    l := len(message) * 8
    m := (447 - l % 512) % 512
    code := make([]byte, (l + 1 + m + 64) / 8)
    copy(code[0: l / 8], message)
    code[l / 8] = byte(1 << 7)
    // code[l + 1: l + 1 + m] = byte(0)
    binary.BigEndian.PutUint64(code[(l + 1 + m + 64) / 8 - 8: (l + 1 + m + 64) / 8], uint64(l))
    N := (l + 1 + m + 64) / 8 / 64
    w := [64]uint32{}
    for n := 0; n < N; n++ {
        // compute w
        for i := 0; i < 16; i++ {
            w[i] = binary.BigEndian.Uint32(code[i * 4 + n * 64: (i + 1) * 4 + n * 64])
        }
        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
        for i := 0; i < 64; i++ {
            t1 := h + Sigma1(e) + ch(e, f, g) + k[i] + w[i]
            t2 := Sigma0(a) + maj(a, b, c)
            h = g
            g = f
            f = e
            e = d + t1
            d = c
            c = b
            b = a
            a = t1 + t2
        }
        h0 = a + h0
        h1 = b + h1
        h2 = c + h2
        h3 = d + h3
        h4 = e + h4
        h5 = f + h5
        h6 = g + h6
        h7 = h + h7
    }
    h := []uint32{h0, h1, h2, h3, h4, h5, h6, h7}
    for i := 0; i < 8; i++ {
        binary.BigEndian.PutUint32(sha256data[i * 4: (i + 1) * 4], h[i])
    }

    return sha256data
}
```

2. 构建 *Merkle* 树

构建 Merkle 树分为两部分，一部分为构建 Merkle 结点，一部分为构建 Merkle 树。首先实现新建 Merkle 结点，分为两种情况，一种是边缘的 Merkle 结点，即左右子孩子均为 nil 的结点，另一种为中间 Merkle 结点，即左右子孩子均为 Merkle 结点的结点。具体代码如下：

```
func NewMerkleNode(left, right *MerkleNode, data []byte) *MerkleNode {
        node := MerkleNode{}
        if left == nil && right == nil {
                hash := mySha256(data)
                node.Data = hash[:]
        } else {
                // ... means two params
                prevHashes := append(left.Data, right.Data...)
                hash := mySha256(prevHashes)
                node.Data = hash[:]
        }
        node.Left = left
        node.Right = right

        return &node
}
```

对于构建 Merkle 树，先对每个原始数据构建一个对应的结点，然后利用这些结点进行 Merkle 树的构建。在顶层 Merkle 结点数大于 1 时，分为两步进行构建，第一步为确保该层 Merkle 结点数为偶数，第二步再利用相邻两个 Merkle 结点的数据构建内部 Merkle 结点。具体代码如下：

```
func NewMerkleTree(data [][]byte) *MerkleTree {
    var nodes []MerkleNode
    // make merkle node for every data
    for _, dataItem := range data {
        node := NewMerkleNode(nil, nil, dataItem)
        nodes = append(nodes, *node)
    }
    // merge all nodes to a tree, merkle tree
    for len(nodes) > 1 {
```

```
        // make the number of nodes even
        if len(nodes) % 2 == 1 {
            nodes = append(nodes, nodes[len(nodes) - 1])
        }
        var tmpNodes []MerkleNode
        for i := 0; i < len(nodes); i += 2 {
            node := NewMerkleNode(&nodes[i], &nodes[i + 1], nil)
            tmpNodes = append(tmpNodes, *node)
        }
        nodes = tmpNodes
    }
    mTree := MerkleTree{&nodes[0]}

    return &mTree
}
```

3. 添加区块添加区块的本质是更新数据库和区块链的内容，即将新的区块信息添加到数据库中，并将区块链中的当前区块更新。具体实现如下：

```
func (bc *Blockchain) AddBlock(data []string) {
    newBlock := NewBlock(data, bc.tip)
    bc.tip = newBlock.Hash
    err := bc.db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        err := b.Put(newBlock.Hash, newBlock.Serialize())
        if err != nil {
            log.Panic(err)
        }
        err = b.Put([]byte("l"), newBlock.Hash)
        if err != nil {
            log.Panic(err)
        }
        return nil
    })
    if err != nil {
        log.Panic(err)
    }
    return
```

```
20 }
```

3 结果及分析

mySha256 函数测试结果如下：

图 1: mySha256 函数测试结果

构建 Merkle 树部分测试结果如下：

图 2: 构建 Merkle 树部分测试结果

添加区块部分测试

图 3: 区块测试结果

根据图3的结果可以发现，数据库的数据被正常修改（因为第一次 `print` 时就有了多的信息，说明之前退出后修改成功保存），也可以看到可以正常添加区块。
但是可以发现，每次添加区块，哈希值仅仅是在原来的哈希值后添加了 `0x31`，经过查看框架代码，可以发现 `NewBlock` 函数中对哈希值的处理如下，

```
hash := append(prevBlockHash, '1')
```

而字符 `1` 的 `ascii` 码刚好为 `0x31`。由于实验没有要求对此部分进行实现，考虑到助教可能出去其他考虑特意如此实现，故没有修改。

4 实验总结

4.1 收获

- 通过本次试验，我更加深入地理解了 `SHA256`、`Merkle` 树和区块链的基本概念，在自己具体通过代码实现部分功能后，不再像以前一样简单的了解。
- 在完成实验的同时，简单学习了 `Go` 这门语言，了解了基本语法和代码格式等。

4.2 建议

- 可以将 `SHA256` 的实验部分单独抽出来作为一次实验，可以早一点发布，可以让没有学习或了解过 `Go` 语言的同学提前学习或熟悉。