实验一 区块链编写

龚小航 PB18151866

【实验目的及要求】

在给出的代码框架下，完成简化的区块链编写。需要正确实现新块的加入，默克尔树的创建以及利用 SPV 路径检验在某个轻节点中验证一笔交易是否存在。

【实验原理】

区块链（是借由密码学串接并保护内容的串连文字记录（又称区块）。每一个区块包含了前一个区块的加密散列、相应时间戳记以及交易资料（通常用默克尔树（Merkle tree）算法计算的散列值表示），这样的设计使得区块内容具有难以篡改的特性。用区块链技术所串接的分布式账本能让两方有效记录交易，且可永久查验此交易。

目前区块链技术最大的应用是数字货币，例如比特币的发明。因为支付的本质是“将账户 A 中减少的金额增加到账户 B 中”。如果人们有一本公共账簿，记录了所有的账户至今为止的所有交易，那么对于任何一个账户，人们都可以计算出它当前拥有的金额数量。而区块链恰恰是用于实现这个目的的公共账簿，其保存了全部交易记录。在比特币体系中，比特币地址相当于账户，比特币数量相当于金额。

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，而 Nonce 字段则表示难度随机数。

代表当前区块链状态的结构如下所示：

```
// Blockchain keeps a sequence of Blocks
type Blockchain struct {
    tip []byte
    db  *bolt.DB
}
```

其中 tip 字段代表了当前最新区块的哈希值，当创建新块时，父块哈希值可以从此处取出使用。而 db 是全局数据库连接，在将新区块连接至链上时，使用此连接可以将数据存入数据库中。

【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 AddBlock 函数：

参考生成创世区块的代码，先利用函数 NewBlock 建立一个新的区块，再利用传入参数区块链指针 bc 得到父区块哈希值，最后将其插入数据库中并更新区块链状态即可。

```
// AddBlock saves provided data as a block in the blockchain
// implement
func (bc *Blockchain) AddBlock(data []string) {
    nb := NewBlock(data,bc.tip)  //nb:newblock
    err := bc.db.Update(func(tx *bolt.Tx) error {//读写修改数据库
        b := tx.Bucket([]byte(blocksBucket))  //打开 bucket，全局只用了一个 bucket
        err := b.Put(nb.Hash, nb.Serialize()) //系列化数据，存入数据库
        if err != nil {
            log.Panic(err)
        }
        err = b.Put([]byte("l"), nb.Hash)   //维护 L 的信息
        if err != nil {
            log.Panic(err)
        }
        bc.tip = nb.Hash
        return nil
    })
    if err != nil {
        log.Panic(err)
    }
}
```

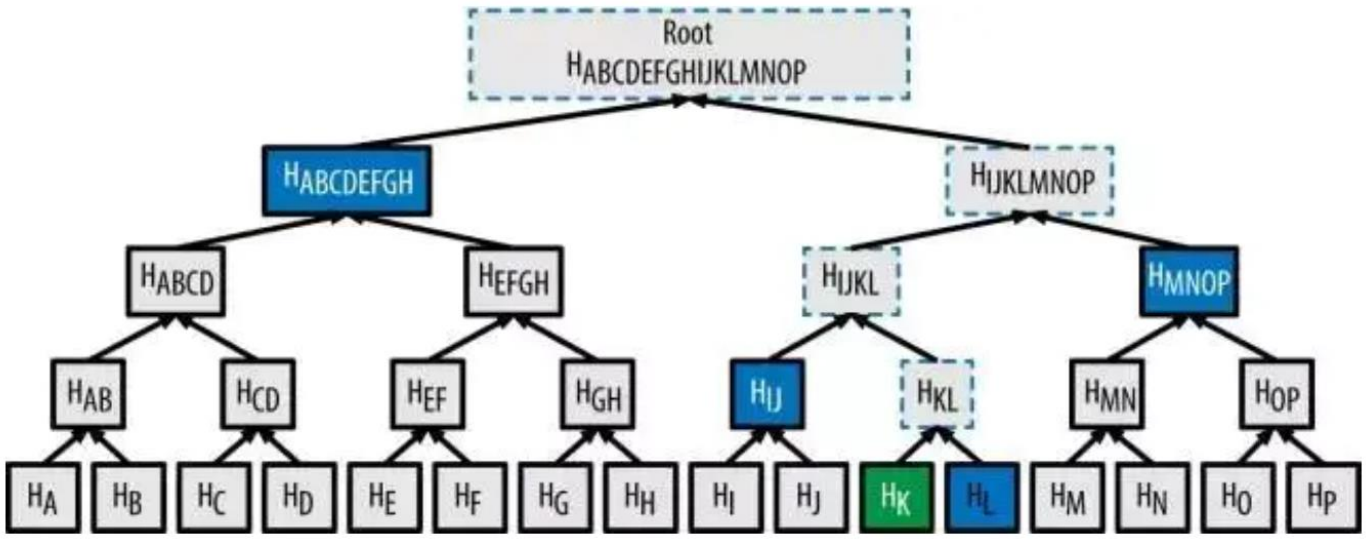
2、填写 NewMerkleTree 函数：

利用双向链表包，将其作为队列使用。先将所有交易的原始数据做哈希运算，填入叶子节点的结构体中，并插入队列同时保证每一次计算时队列中初始都有偶数个元素。层层向上计算，当 Q 中仅剩一个元素时即为 merkle 树的根节点。在这个计算的过程中，由于每个节点都是带指针的结构体，因此树结构也随之建立了起来。

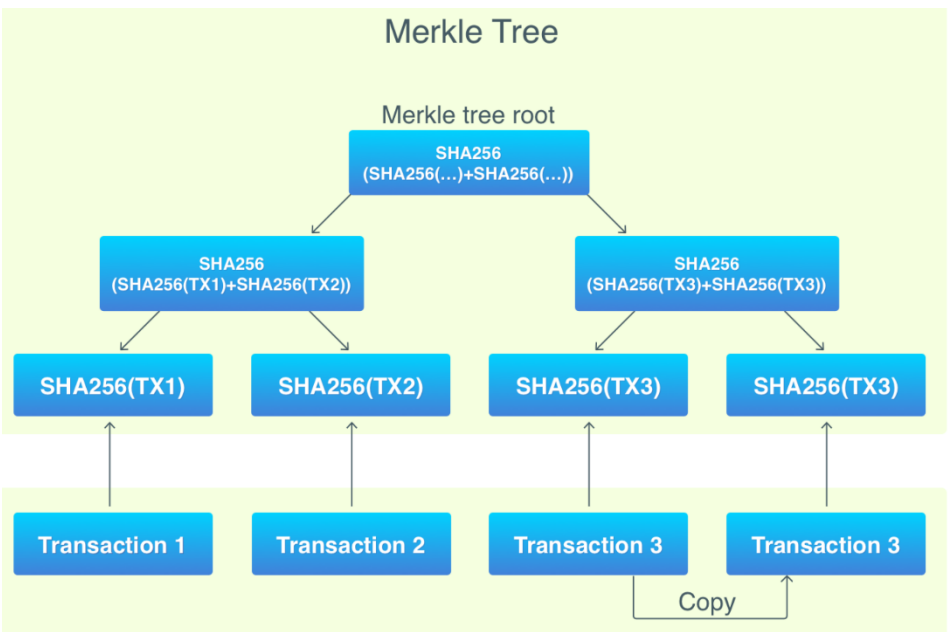
```
// NewMerkleTree creates a new Merkle tree from a sequence of
data
// implement
func NewMerkleTree(data [][]byte) *MerkleTree {
    len_data := len(data)
    Q := list.New()       //双向链表包
    Q.Init()
        //fmt.Print("1\n")
    for i:=0;i < len_data;i++{
        var temp MerkleNode
        temp.Right = nil
        temp.Left = nil
            a := sha256.Sum256(data[i])
        temp.Data = a[:]//a仅用作类型转换
        Q.PushBack(temp)
    }
        //fmt.Print("2\n")
        if (len_data % 2 == 1) {
            var temp MerkleNode
            temp.Right = nil
            temp.Left = nil
        a := sha256.Sum256(data[len_data-1])
        temp.Data = a[:]//a仅用作类型转换
        Q.PushBack(temp)
    }                   //将原始数据做哈希存入队列

    for Q.Len() != 1{
        len_data = Q.Len() //更新长度
        if len_data % 2 == 1 {
            Q.PushBack(Q.Back().Value.(MerkleNode))
            len_data = Q.Len()
        }
        for i:=1;i<=len_data/2;i++ {//取出两个元素，求父节点，放回队尾
            var father MerkleNode
            op1_temp := Q.Front()
            op1_value := op1_temp.Value.(MerkleNode)
            Q.Remove(op1_temp)
            op2_temp := Q.Front()
            op2_value := op2_temp.Value.(MerkleNode)
            Q.Remove(op2_temp)

            father.Left = &op1_value
            father.Right = &op2_value
            var buffer bytes.Buffer
//Buffer是一个实现了读写方法的可变大小的字节缓冲，用于实现两个哈希值的连接
            buffer.Write(op1_value.Data)
            buffer.Write(op2_value.Data)
            op_1add2 := buffer.Bytes()  //得到了b1+b2的结果
                a := sha256.Sum256(op_1add2)
            father.Data = a[:]
            Q.PushBack(father)
            //fmt.Println(Q.Len())
        }
    }

    a := Q.Front().Value.(MerkleNode)
    var mTree = MerkleTree{&a}
    return &mTree
    //var node = MerkleNode{nil,nil,data[0]}
    //return &mTree
}
```

3. SPV 验证某笔交易的存在性：

某个轻节点需要查询一笔交易的存在性，已知这笔交易数据的哈希值，从全节点得到计算出根值的所有需要数据以及它们的左/右方位。如下图，H_k 为待查询的交易的哈希值，所有蓝色节点的值以及左右信息都通过全节点传递给轻节点。通过层层计算哈希值即可得到算出的默克尔根。与全节点提供的根值相比即可知道这笔交易是否存在。另一方面，全节点提供的根值需要在轻节点本身的默克尔根域中存在，防止全节点欺骗。



检验代码如下所示，添加在 test 文件下，在 assert 风格下进行单元测试。测试图如下所示，检验交易 2 是否存在。



检验代码如下所示：

```
func SPVCheck(data []SPVnode,hash_to_check []byte)
[]byte{
    len_data := len(data)
    Q := list.New()       //双向链表包
    Q.Init()

    temp := SPVnode{
        hash_to_check,
        true, //任意值均可，待检验节点
    }

    Q.PushBack(temp)//先把带校验数据放入

    for i:=0;i < len_data;i++{
        Q.PushBack(data[i])
    }

    for Q.Len() != 1{
        var father SPVnode
        father.left_or_right = true//任意值
            //fmt.Println(Q.Len())
        op1_temp := Q.Front()
        op1_value := op1_temp.Value.(SPVnode)
        Q.Remove(op1_temp)
        op2_temp := Q.Front()
        op2_value := op2_temp.Value.(SPVnode)
        Q.Remove(op2_temp)

        if(op2_value.left_or_right == true){
                //fmt.Println("left")
            var buffer bytes.Buffer
        //Buffer 是一个实现了读写方法的可变大小的字节缓冲
            buffer.Write(op2_value.hash)
            buffer.Write(op1_value.hash)
            op_2add1 :=buffer.Bytes()//得到了 op2+op1 的结果
            a := sha256.Sum256(op_2add1)
            father.hash = a[:]
            Q.PushFront(father)

        } else{
                //fmt.Println("right")
            var buffer bytes.Buffer
        //Buffer 是一个实现了读写方法的可变大小的字节缓冲
            buffer.Write(op1_value.hash)
            buffer.Write(op2_value.hash)
            op_1add2 :=buffer.Bytes()//得到了 op1+op2 的结果
            a := sha256.Sum256(op_1add2)
            father.hash = a[:]
            Q.PushFront(father)
        }
    }

    return Q.Front().Value.(SPVnode).hash
}
```

另一方面，SPV 检验函数如下所示，返回计算出的默克尔根值：

```
type SPVnode struct{
    hash          []byte
    left_or_right  bool   //左侧 true 右侧 false
}

func TestSPV(t *testing.T) {
    data := [][]byte{
        []byte("node1"),
        []byte("node2"),
        []byte("node3"),
    }

    var hash1 []byte
    var hash3 []byte
    var hash3_3 []byte

    mTree := NewMerkleTree(data)

        a1 := sha256.Sum256(data[0])
    hash1 = a1[:]
        a3 := sha256.Sum256(data[2])
    hash3 = a3[:]
    var buffer bytes.Buffer
    //Buffer 是一个实现了读写方法的可变大小的字节缓冲
    buffer.Write(hash3)
    buffer.Write(hash3)
    op_2add1 :=buffer.Bytes()  //得到了 op2+op1 的结果
    a := sha256.Sum256(op_2add1)
    hash3_3 = a[:]
    testdata := []SPVnode{
    //校验交易 2，给出信息交易 1 的哈希，以及交易 3 交易 3 哈希和的哈希
        SPVnode{
            hash1,
            true,
        }, //node1 hash
        SPVnode{
            hash3_3,
            false,
        },
    }

        a2 := sha256.Sum256(data[1])
    hash_to_check := a2[:]
    SPVroot := SPVCheck(testdata,hash_to_check)
    assert.Equal(t, mTree.RootNode.Data, SPVroot , "SPV OK")
}
```

【实验结果】

添加一个区块之后得到的结果：

```
终端:  Local  +
chaincode > addblock xyzab
add Success
chaincode > printchain
Prev. hash: 3afdc96b35e60a6c3d98fc06ca8647ad5a106c862503cb64f982d260928c7285
Data: [xyzab]
Hash: 14800107916b9718dade0403b469c75fa138d3b4db7dd49c8e90d5b0de32b463
PoW: true

Prev. hash: 0022bda25c172339d0ef530118603badb6e7956f42f115512c6f356dfce36a96
Data: [hhhhh]
Hash: 3afdc96b35e60a6c3d98fc06ca8647ad5a106c862503cb64f982d260928c7285
PoW: true

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

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

chaincode >
```

Merkle 树验证结果：

```
D:\OneDrive - mail.ustc.edu.cn\桌面\BC_LAB\lab1\template>go test
PASS
ok      chaincode       0.038s
```

SPV 路径验证存在的交易：（验证交易 2 是否存在）

```
D:\OneDrive - mail.ustc.edu.cn\桌面\BC_LAB\lab1\template>go test
PASS
ok      chaincode       0.302s
```

SPV 路径验证一个不存在的交易：

```
D:\OneDrive - mail.ustc.edu.cn\桌面\BC_LAB\lab1\template>go test
--- FAIL: TestSPV (0.00s)
    merkle_tree_test.go:126:
                Error Trace:    merkle_tree_test.go:126
                Error:          Not equal:
                                expected: []byte{0x4e, 0x3e, 0x44, 0xe5, 0x59, 0x26, 0x33, 0xa, 0xb6, 0xc3, 0x18, 0x92, 0xf9, 0x80, 0xf8, 0xbf, 0xd1, 0xa6, 0xe9, 0x10, 0xff, 0x1e, 0xbc
, 0x3f, 0x77, 0x82, 0x11, 0x37, 0x7f, 0x35, 0x22, 0x7e}
                                actual  : []byte{0x8f, 0x9e, 0x7d, 0x82, 0x33, 0x61, 0x3, 0x22, 0x95, 0xf8, 0x5c, 0x80, 0x4f, 0x85, 0x81, 0x55, 0x2, 0xa1, 0xb4, 0x2f, 0x18, 0xb2, 0xf3,
 0x99, 0xb2, 0xd4, 0xb3, 0xb6, 0xe7, 0x12, 0xca, 0x5c}

                                Diff:
                                --- Expected
                                +++ Actual
                                @@ -1,4 +1,4 @@
                                 ([]uint8) (len=32) {
                                - 00000000  4e 3e 44 e5 59 26 33 0a  b6 c3 18 92 f9 80 f8 bf  |N>D.Y&3.........|
                                - 00000010  d1 a6 e9 10 ff 1e bc 3f  77 82 11 37 7f 35 22 7e  |.......?w..7.5"~|
                                + 00000000  8f 9e 7d 82 33 61 03 22  95 f8 5c 80 4f 85 81 55  |..}.3a."..\.O..U|
                                + 00000010  02 a1 b4 2f 18 b2 f3 99  b2 d4 b3 b6 e7 12 ca 5c  |.../...........\|
                                 }
                Test:           TestSPV
                Messages:       Not EXISTS
```

各验证结果均符合预期。