lab1-区块链编写

姓名：裴启智

学号：PB18111793

实验目的及要求

- 编写一个简化版的区块链，并且会将数据写入数据库，实现数据的持久化
- 具体而言，需要在现有框架下补充 blockchain.go 的 addblock 函数和 Merkle_tree.go 的 NewMerkleTree函数
- Bonus：完成 merkle 树的验证，即校验某个区块中是否存在特定的交易
- 本人在完成时同时完成了实验二部分，结果可能和单纯完成实验一稍有不同，比如 addblock 添加的Hash值

实验原理

这里主要是结合文档并分析实验框架中已经给出的部分代码

区块

- 区块是区块链中重要的组成部分，在区块链中信息通常是在区块中进行存储的。例如，比特币中会在区
  块中存储交易信息。同时，一个区块还包含有版本号，时间戳，前一个区块哈希指等信息
- 本次实验中使用的简化版区块结构

```
// Block 由区块头和交易两部分构成
// Timestamp     : 当前时间戳，也就是区块创建的时间
// PrevBlockHash : 前一个区块对应的哈希
// Hash          : 当前区块对应的哈希
// Data          : 区块实际存储的信息，也就是比特币的交易
// Nonce         : 区块对应的随机数，用于工作量证明时的验证
type Block struct {
    Timestamp     int64
    Data          [][]byte
    PrevBlockHash []byte
    Hash          []byte
    Nonce         int
}
```

- 在区块中的Hash值通常采用SHA-256的方式来进行加密，在Go语言中，我们可以调用函数 sha256.Sum256 来对于[]byte的数据进行加密工作

```
// 根据区块的数据建立Merkle树,返回Merkle根的数据
func (b *Block) HashData() []byte {
    mTree := NewMerkleTree(b.Data)

    return mTree.RootNode.Data
}
```

- 创建新区块

```
// 创建新区块，需要运行工作量证明找到有效哈希
// NewBlock creates and returns Block
func NewBlock(datas []string, prevBlockHash []byte) *Block {
    // 类型为byte的二维切片
    blockData := [][]byte{}
    // 将datas中的数据读入blockData
    for _, data := range datas {
        blockData = append(blockData,[]byte(data))
    }
    // 建立区块
    block := &Block{time.Now().Unix(), blockData, prevBlockHash, []byte{},
0}
    pow := NewProofOfWork(block)
    nonce, hash := pow.Run()

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

    return block
}
```

对比比特币

- 完整的比特币区块头结构

Field	Purpose	Updated when...	Size (Bytes)
Version	Block version number	You upgrade the software and it specifies a new version	4
hashPrevBlock	256-bit hash of the previous block header	A new block comes in	32
hashMerkleRoot	256-bit hash based on all of the transactions in the block	A transaction is accepted	32
Time	Current timestamp as seconds since 1970-01-01T00:00 UTC	Every few seconds	4
Bits	Current target in compact format	The difficulty is adjusted	4
Nonce	32-bit number (starts at 0)	A hash is tried (increments)	4

- 比特币的 golang 实现 btcd 的 BlockHeader 定义:

```
// BlockHeader defines information about a block and is used in the bitcoin
// block (MsgBlock) and headers (MsgHeaders) messages.
type BlockHeader struct {
    // Version of the block.  This is not the same as the protocol version.
    Version int32

    // Hash of the previous block in the block chain.
    PrevBlock chainhash.Hash

    // Merkle tree reference to hash of all transactions for the block.
    MerkleRoot chainhash.Hash

    // Time the block was created.  This is, unfortunately, encoded as a
    // uint32 on the wire and therefore is limited to 2106.
    Timestamp time.Time

    // Difficulty target for the block.
    Bits uint32

    // Nonce used to generate the block.
    Nonce uint32
}
```

- 完整的比特币区块结构

Field	Description	Size
Magic no	value always 0xD9B4BEF9	4 bytes
Blocksize	number of bytes following up to end of block	4 bytes
Blockheader	consists of 6 items	80 bytes
Transaction counter	positive integer VI = VarInt	1 - 9 bytes
transactions	the (non empty) list of transactions	-many transactions

数据库

本次实验中通过数据库进行区块的存储

BoltDB

- 一个简单的，轻量级的集成在Go语言上的数据库，不需要运行服务器
- BoltDB是一个K-V数据库。所以，数据是以键值对的形式进行存储的
- 在BoltDB上对应操作是存储在bucket中的，这是为了将相似的键值对进行分组（类似 RDBMS 中的表格）
- 为了获取一个值，需要知道一个 bucket 和一个键（key）
- Bolt 数据库没有数据类型：键和值都是字节数组（byte array）。鉴于需要在里面存储 Go 的结构（准确来说，也就是存储**Block（块）**），需要对它们进行序列化和反序列化（通过 encoding/gob ）
    - **序列化**

```
// 将区块数据序列化为一个字节数组
// Serialize serializes the block
func (b *Block) Serialize() []byte {
    // 存储序列化后的数据
    var result bytes.Buffer
    encoder := gob.NewEncoder(&result)
    // 对 Block 进行编码
    err := encoder.Encode(b)
    if err != nil {
        log.Panic(err)
    }

    // 编码的结果作为字节数组返回
    return result.Bytes()
}
```

    - **反序列化**

```
// 将字节数组反序列化为一个Block
// 注意和 Serialize 不同的是，DeserializeBlock 是一个函数 而非 方法
// DeserializeBlock deserializes a block
func DeserializeBlock(d []byte) *Block {
    var block Block
```

```
    decoder := gob.NewDecoder(bytes.NewReader(d))
    err := decoder.Decode(&block)
    if err != nil {
        log.Panic(err)
    }

    return &block
}
```

对比比特币中的数据库存储

在 **blocks** 中，**key -> value** 为：

key	value
`b` + 32 字节的 block hash	block index record
`f` + 4 字节的 file number	file information record
`l` + 4 字节的 file number	the last block file number used
`R` + 1 字节的 boolean	是否正在 reindex
`F` + 1 字节的 flag name length + flag name string	1 byte boolean: various flags that can be on or off
`t` + 32 字节的 transaction hash	transaction index record

在 **chainstate**，**key -> value** 为：

key	value
`c` + 32 字节的 transaction hash	unspent transaction output record for that transaction
`B`	32 字节的 block hash: the block hash up to which the database represents the unspent transaction outputs

数据结构

- 在比特币代码中，区块主要存储的是两种数据：
    - 区块信息，存储对应每个区块的元数据内容。
    - 区块链的世界状态，存储链的状态，当前未花费的交易输出还有一些元数据
- 本次实验中，区块链需要存储的信息在k-v数据库中，存储的键值对如下：
    - 32 字节的 block-hash -> block 结构
    - `l` -> 链中最后一个块的 hash

数据库操作

- 持久化操作
    1. 打开一个数据库文件
    2. 检查文件里面是否已经存储了一个区块链
    3. 如果已经存储了一个区块链：

1. 创建一个新的 `Blockchain` 实例
2. 设置 `Blockchain` 实例的 tip 为数据库中存储的最后一个块的哈希

4. 如果没有区块链：

1. 创建创世块
2. 存储到数据库
3. 将创世块哈希保存为最后一个块的哈希
4. 创建一个新的 `Blockchain` 实例，初始时 tip 指向创世块

- 创建数据库连接实例

```
db, err := bolt.Open(dbFile, 0600, nil)
```

- Go 关键词 defer 在当前函数返回前执行传入的函数，在这里用来数据库的连接断开。

```
defer bc.db.Close()
```

- 在BoltDB中，对于数据库的操作是通过 `bolt.Tx` 来执行的，对应有两种交易模式**只读操作和读写操作**

  对于读写操作的格式如下：

```
err = db.Update(func(tx *bolt.Tx) error {
...
})
```

  对于只读操作的格式如下：

```
err = db.View(func(tx *bolt.Tx) error {
...
})
```

- 区块链的创建

```
err = db.Update(func(tx *bolt.Tx) error {
        // 获取存储区块的Bucket
        b := tx.Bucket([]byte(blocksBucket))

        // 如果不存在
        if b == nil {
            fmt.Println("No existing blockchain found. Creating a new
one...")
            // 生成创始区块
            genesis := NewGenesisBlock()
            // 创建 bucket
            b, err := tx.CreateBucket([]byte(blocksBucket))

            if err != nil {
                log.Panic(err)
            }
            // 将创世区块保存到 bucket 中
            err = b.Put(genesis.Hash, genesis.Serialize())
            if err != nil {
                log.Panic(err)
            }
```

```
            // 更新 l 键以存储区块链中最后一个块的哈希
            err = b.Put([]byte("l"), genesis.Hash)
            if err != nil {
                log.Panic(err)
            }
            // 设置 tip 指向创世区块
            tip = genesis.Hash
        } else {
            // 如果存在,从中读取 l 键,并设置 tip 指向最后一个块的哈希
            tip = b.Get([]byte("l"))
        }
        // 如果存在 error
        return nil
    })
```

通过`l`读取上一个区块的信息，所以我们在添加一个新的区块之后，需要维护`l`字段对应的内容。

区块链

- 通过链的方式来对于区块数据进行存储的模式，即区块链
- 本质上，区块链就是一个有着特定结构的数据库，是一个有序，每一个块都连接到前一个块的链表。也就是说，区块按照插入的顺序进行存储，每个块都与前一个块相连。这样的结构，能够让我们快速地获取链上的最新块，并且高效地通过哈希来检索一个块。
- 区块链结构

```
// Blockchain keeps a sequence of Blocks
// tip指的是存储最后一个块的哈希
// 在链的末端可能出现短暂分叉的情况，所以选择 tip 其实也就是选择了哪条链
// db 存储数据库连接
type Blockchain struct {
    tip []byte
    db  *bolt.DB
}
```

`tip`代表了最新区块的哈希值，`db`表示了数据库的连接

- 创建区块链的方式

```
// 仅存储区块链的 tip 和数据库连接
// 这样一旦打开数据库连接,区块链就可以运行
bc := Blockchain{tip, db}
```

- 初始状态下，区块链是空的。所以，在任何一个区块链中，都必须至少有一个块。这个块，也就是链中的第一个块，通常叫做创世块（**genesis block**）。通过如下方法来创建创世区块：

```
// 创建创世区块Genesis Block
// NewGenesisBlock creates and returns genesis Block
func NewGenesisBlock() *Block {
    return NewBlock([]string{"Genesis Block"}, []byte{})
}
```

区块链迭代器

- 使用 BlockchainIterator 对bucket 里面的所有 key 进行迭代。我们不需要将所有块都加载到内存中，而是从磁盘中一个一个读取

```
// 迭代器的初始状态为 tip,所以区块将从尾到头进行迭代
// 选择一个 tip 实际上就是给一个链投票
// Iterator ...
func (bc *Blockchain) Iterator() *BlockchainIterator {
    bci := &BlockchainIterator{bc.tip, bc.db}

    return bci
}
```

- 迭代器的初始状态为链中的 tip，因此区块将从尾到头（创世块为头）进行获取。
- 实际上，**选择一个 tip 就是意味着给一条链“投票”。**
- 一条链可能有多个分支，最长的那条链会被认为是主分支。在获得一个 tip （可以是链中的任意一个块）之后，就可以重新构造整条链，找到它的长度和需要构建它的工作。这同样也意味着，一个 tip 也就是区块链的一种标识符。
- `BlockchainIterator` 的方法 `Next`：返回链中的下一个块。

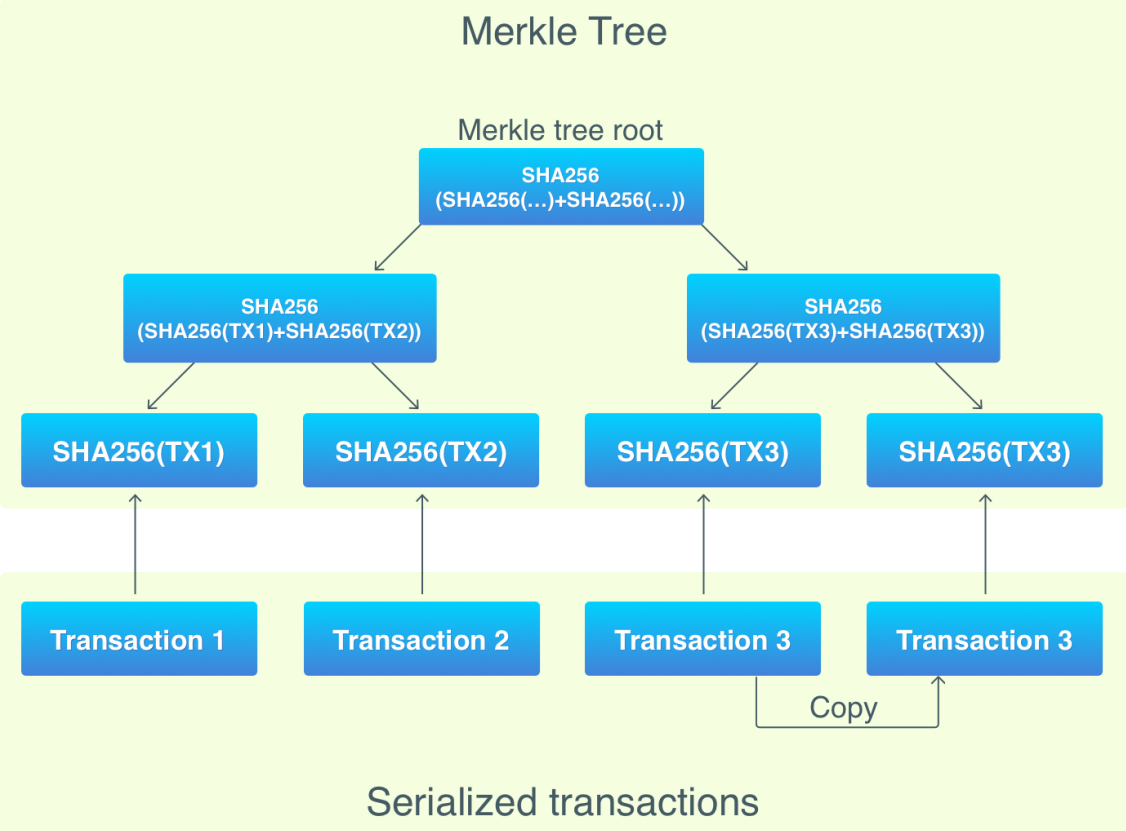
```
// Next returns next block starting from the tip
func (i *BlockchainIterator) Next() *Block {
    var block *Block
    // 只读事务,读取当前的区块
    err := i.db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        encodedBlock := b.Get(i.currentHash)
        block = DeserializeBlock(encodedBlock)

        return nil
    })

    if err != nil {
        log.Panic(err)
    }
    // 调整迭代器到前一个区块
    i.currentHash = block.PrevBlockHash

    return block
}
```

Merkle 树

- 完整的比特币数据库（也就是区块链）需要超过 140 Gb 的磁盘空间。因为比特币的去中心化特性，网络中的每个节点必须是独立，自给自足的，也就是每个节点必须存储一个区块链的完整副本。随着越来越多的人使用比特币，这条规则变得越来越难以遵守：因为不太可能每个人都去运行一个全节点。并且，由于节点是网络中的完全参与者，它们负有相关责任：节点必须验证交易和区块。另外，要想与其他节点交互和下载新块，也有一定的网络流量需求。
- 在比特币的白皮书中，是通过**SPV**（Simplified Payment Verification）的方式来进行交易认证的。SPV 是一个比特币轻节点，它不需要下载整个区块链，也**不需要验证区块和交易**。相反，它会在区块链查找交易（为了验证支付），并且需要连接到一个全节点来检索必要的数据。这个机制允许在仅运行一个全节点的情况下有多个轻钱包。通过这个机制，我们可以让多个轻节点依赖一个全

节点来运行。

- 在Merkle树结构中，我们需要对每一个区块进行节点建立，从叶子节点开始建立的。首先，对于叶子节点，我们会进行哈希加密（在比特币中采用了双重SHA加密哈希的方式）。如果结点个数为奇数，那么最后一个节点会把最后一个交易复制一份，来保证数量为偶数。
- 自底向上，我们会对于节点进行哈希合并的操作，这个操作会不停执行直到节点个数为1。根节点对应就是这个区块所有交易的一个表示，并且会在后续的POW中使用。
- 这样在我们进行对于特定交易认证的时候不需要下载区块中包含的所有交易，只需要验证对应的Merkle根节点和对应的路径。
- 简单的Merkle树示例可以参考图片



- Merkle 树的好处：一个节点可以在不下载整个块的情况下，验证是否包含某笔交易。并且这些只需要一个交易哈希，一个 Merkle 树根哈希和一个 Merkle 路径。
- 结构

```
// 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
}
```

- 创建新的 Merkle 树结点

```
// data为序列化后的交易
func NewMerkleNode(left *MerkleNode, right *MerkleNode, data []byte)
*MerkleNode {
```

```
    mNode := MerkleNode{}
    // 叶子结点，数据从外界传入
    if left == nil && right == nil {
        hash := sha256.Sum256(data)
        mNode.Data = hash[:]
    } else {
        // 内部结点，将子节点的数据连接后再哈希
        // 后面加三个点，这时append只支持两个参数，不支持任意个数的参数。
        prevHashes := append(left.Data, right.Data...)
        hash := sha256.Sum256(prevHashes)
        mNode.Data = hash[:]
    }
    mNode.Left = left
    mNode.Right = right

    return &mNode
}
```

实验平台

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

实验步骤

完成以下两个函数，具体解释参见注释

AddBlock

```
// 添加区块,并存储到数据库中
// AddBlock saves provided data as a block in the blockchain
// implement
func (bc *Blockchain) AddBlock(data []string) {
    var lHash []byte
    // 获取最后一个块的哈希用于生成新块的哈希,只读事务
    err := bc.db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        // 获取最后一个块的哈希
        lHash = b.Get([]byte("l"))
        return nil
    })

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

    // 用最后一个块的哈希产生新的块
    newBlock := NewBlock(data, lHash)
    // 读写事务，添加区块
    err = bc.db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        // key = newBlock.Hash, value = newBlock的序列化表示
        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)
        }
        // 设置 tip 指向新块的Hash值
        bc.tip = newBlock.Hash
        return nil
    })
}
```

NewMerkleTree

```
// NewMerkleTree creates a new Merkle tree from a sequence of data
// implement
func NewMerkleTree(data [][]byte) *MerkleTree {
    // var node = MerkleNode{nil,nil,data[0]}
    var nodes []MerkleNode
    // 保证叶子节点为偶数
    if len(data) % 2 != 0 {
        // 复制最后一份数据
        data = append(data, data[len(data) - 1])
    }

    // 将所有数据读入所有叶子结点
    for _, dataTemp := range data {
        node := NewMerkleNode(nil, nil, dataTemp)
        nodes = append(nodes, *node)
    }

    // 逐层合并，由下到上，生成内部结点
    for i := 0; i < len(data) / 2; i++ {
        var treeLevel []MerkleNode
        for j := 0; j < len(nodes); j += 2 {
            node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
            treeLevel = append(treeLevel, *node)
        }
        // 每轮循环中都对 nodes 赋值，最终nodes只含有 Merkle 树的根结点
        nodes = treeLevel
    }

    var mTree = MerkleTree{&nodes[0]}
    return &mTree
}
```

实验结果

- 首先清空之前的 `blockchain.db` 文件，从而演示新建一个区块链的过程
- 在代码目录运行 `go run .` 命令

```
D:\study\blockchain-lab\lab2\template>go run .
No existing blockchain found. Creating a new one...
Mining the block, containing data :"[Genesis Block]"
0027d55ca56d3c367deebb7c8cf0a7a74298b04b843b18dab8de2283706ca8f1
```

- 然后添加区块，`addblock pqz`

```
chaincode > addblock pqz
Mining the block, containing data :"[pqz]"
0008ceedf3e3b2899a1878c1ed4edfef2223b8601edddf370c26e9cdbb280140

add Success
```

- 再添加一个区块，`addblock 123`

```
chaincode > addblock 123
Mining the block, containing data :"[123]"
003c51d201486c03cd70f62b7f3a9bdc5f875bd2a84758814699219cdbb9af6b

add Success
```

- 查看当前区块链中的内容是否正确

```
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
```

- 验证 Merkle 建立是否正确：在代码目录运行`go test`命令

```
D:\study\blockchain-lab\lab2\template>go test
PASS
ok      chaincode       0.037s
```

Bonus

默克尔路径

- 默克尔树除了用来归纳交易并生成整个交易集合的默克尔根之外， 同时也提供了校验某个区块中是否存在特定的交易的一种高效的途径， 即**默克尔路径。**
- 默克尔路径是由从默克尔根到叶子节点所经过的节点（注意不包括叶子结点）组成的路径 。一般说来，在 N 个交易组成的区块中确认任一交易的算法复杂度（体现为默克尔路径长度） 仅为 $log_2 N$
- 如果交易哈希值的长度为 32 字节， 那么当区块大小由 16 笔交易增加至 4096 笔交易时， 默克尔路径长度增长极其缓慢， 仅仅从 128 字节增长到 384 字节。 因此，通过默克尔路径，基于 SPV 技术的轻节点只需很小的开销就可以快速定位一笔交易

实现

- go test 命令，会自动读取源码目录下面名为 *_test.go 的文件，生成并运行测试用的可执行文件。
- 在本次实验中，`go test`会自动测试`merkle_tree_test.go`下的两个函数

```
D:\study\blockchain-lab\lab2\template>go test -v
=== RUN   TestNewMerkleNode
--- PASS: TestNewMerkleNode (0.00s)
=== RUN   TestNewMerkleTree
--- PASS: TestNewMerkleTree (0.00s)
PASS
ok      chaincode       0.034s
```

- 故验证 Merkle 树中是否存在某笔交易只需在`merkle_tree_test.go`文件中添加一个函数即可

```go
// 验证该 Merkle 树上是否有 data[0] 对应的交易
func TestMerklePath(t *testing.T) {
    data := [][]byte{
        []byte("node1"),
        []byte("node2"),
        []byte("node3"),
    }

    mTree := NewMerkleTree(data)
    n1 := mTree.RootNode.Left.Left

    returnVal := MerklePath(n1, mTree.RootNode)

    assert.Equal(t, true, returnVal, "Merkle path : root hash is correct")
}
```

这里调用`blockchain.go`中的`MerklePath`函数，用来判断 n1 是否属于 mTree，如果 n1 属于 mTree，则 returnVal 为真，可以通过 Test 。反之为假

- 由于 Merkle 路径是自底向上的，故需要修改 Merkle 的数据结构，增加 Parent 指针用来对 Merkle 树进行自底向上的回溯

```
// 为了验证 Merkle 路径，需要从叶子结点向上查找，故修改数据结构
// MerkleNode represent a Merkle tree node
type MerkleNode struct {
    Left  *MerkleNode
    Right *MerkleNode
    Parent *MerkleNode
    Data  []byte
}
```

- `MerklePath`函数

```
func MerklePath(h *MerkleNode, root *MerkleNode) bool {
    inOrder(root)
    result := h.Data

    var tempHash [][]byte
    for getBrotherNode(h) != nil {
        tempHash = append(tempHash, getBrotherNode(h).Data)
        h = h.Parent
    }


    for i := 0; i < len(tempHash); i++ {
        hash := sha256.Sum256(append(result, tempHash[i]...))
        result = hash[:]
    }

    return bytes.Equal(result, root.Data)
}
```

该函数

- 首先调用 `inOrder` 函数完善树结构，填充 Merkle 树每个结点的 Parent 域
- `result := h.data`，记录要验证的交易的哈希
- 然后自底向上获得 Merkle 路径上的结点，存入 tempHash
- 自底向上计算，计算 Merkle 根的哈希

  ```
  for i := 0; i < len(tempHash); i++ {
      hash := sha256.Sum256(append(result, tempHash[i]...))
      result = hash[:]
  }
  ```

- 最后比较计算得到的Hash值 `result` 和实际 Merkle 根的Hash值`root.Data`，如果二者相同则返回 true，否则返回 false

其中用到了两个附加的函数

`getBrotherNode`函数用来得到传入节点的兄弟节点

```go
func getBrotherNode(node *MerkleNode) *MerkleNode {
    if node.Parent == nil {
        return nil
    } else if node == node.Parent.Left {
        return node.Parent.Right
    } else {
        return node.Parent.Left
    }
}
```

`inOrder`函数用来对 Merkle 树进行中序遍历，主要是为了将除了根结点外的所有结点连接到其双亲结点

```go
func inOrder(p *MerkleNode) {
    if p.Left != nil {
        inOrder(p.Left)
        p.Left.Parent = p
    }
    if p.Right != nil{
        inOrder(p.Left)
        p.Right.Parent = p
    }
}
```

结果验证

- 重新执行 `go test -v`

```
D:\study\blockchain-lab\lab2\template>go test -v
=== RUN   TestNewMerkleNode
--- PASS: TestNewMerkleNode (0.00s)
=== RUN   TestNewMerkleTree
--- PASS: TestNewMerkleTree (0.00s)
=== RUN   TestMerklePath
--- PASS: TestMerklePath (0.00s)
PASS
ok      chaincode       0.064s
```

可以看到通过了新添加的 Test，说明 n1 属于以 mTree 为根的 Merkle树

- 我们也可以修改 n1 结点不属于以 mTree 为根的 Merkle树

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

    data1 := [][]byte{
        []byte("node4"),
```

```
        []byte("node5"),
        []byte("node6"),
    }

    mTree := NewMerkleTree(data)
    mTree1 := NewMerkleTree(data1)

    n1 := mTree1.RootNode.Left.Left

    returnVal := MerklePath(n1, mTree.RootNode)

    assert.Equal(t, true, returnVal, "Merkle path : root hash is correct")
}
```

执行 `go test -v`，可以看到没有通过 test

```
D:\study\blockchain-lab\lab2\template>go test -v
=== RUN   TestNewMerkleNode
--- PASS: TestNewMerkleNode (0.00s)
=== RUN   TestNewMerkleTree
--- PASS: TestNewMerkleTree (0.00s)
=== RUN   TestMerklePath
    merkle_tree_test.go:101:
                Error Trace:    merkle_tree_test.go:101
                Error:          Not equal:
                                expected: true
                                actual  : false
                Test:           TestMerklePath
                Messages:       Merkle path : root hash is correct
--- FAIL: TestMerklePath (0.00s)
FAIL
exit status 1
FAIL    chaincode       0.041s
```

实验总结

- 为了完成区块链系列实验，花费了一定的时间来学习 Go 语言。在实验的过程中也花费了不少时间来理解助教给出的 Go 代码框架，对 Go 语言有了更加深入的理解。个人感觉 Go 语言既有 C 语言的一些特性，比如要求指明变量类型，也有一些 Python 的特性，特别是切片类型很像 Python 中的 list，同时 Go 语言还有一些自己的特性，比如声明变量的多种方式，以及声明的变量必须在后续被使用等等
- 在本次实验中仅仅实现了一个简单的区块链，每个块都与前一个块相关联。真实的区块链要比这复杂得多。
    - 在实验区块链中，加入新的块简单快速，但是在真实的区块链中，加入新的块需要很多工作
    - 必须要经过十分繁重的计算（工作量证明），来获得添加一个新块的权限，这在lab2中会加以实现
    - 区块链是一个分布式数据库，并且没有单一决策者。因此，要加入一个新块，必须要被网络的其他参与者确认和同意——共识（consensus）。
    - 当前区块链还没有任何的交易（Transaction）
- 在本次实验中我也对区块链具体是什么 有了更加深入的理解，了解了区块链底层的数据结构等等，学以致用