LAB1实验报告

【实验名称】区块链编写

姓名 李泓民 学号 PB18071495

【实验目的及要求】

学会简单的go使用语法，以及BoltDB数据库接口调用

编写一个简化版的区块链，并且会将数据写入数据库

实现merkle树建立，添加区块和merkle树的验证

【实验原理】

区块

实验中使用的区块结构如下：

```
type Block struct {
    Timestamp     int64  // 时间戳
    Data          [][]byte //数据
    PrevBlockHash []byte //前一个区块对应哈希
    Hash          []byte //当前区块数据对应哈希值
    Nonce         int //随机数
}
```

`Timestamp`代表了整个区块对应的时间戳，`Data`当前区块存储的数据。`PrevBlockHash`代表了前一个区块对应的哈希值。`Hash`代表了当前区块的哈希值。`Nonce`代表了这个区块对应的随机数。

在Go语言中，通过调用函数`sha256.Sum256`来对于*[]byte*的数据进行加密工作。

数据库

在本次实验中，我们选取了BoltDB的数据库。它是一个K-V数据库。

数据结构

在本次实验中，区块链需要存储的信息相对也进行了简化。例如k-v数据库中，存储数据如下：

1. b，存储了区块数据
2. l，存储了上一个区块信息

数据库操作

对于数据库的操作主要如下：

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

用来创建一个数据库连接的实例。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 语句会将函数推迟到外层函数返回之后执行。
>
> 推迟调用的函数其参数会立即求值，但直到外层函数返回前该函数都不会被调用。

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

通过链的方式来对于区块数据进行存储的模式，就是我们的区块链了。所以，在区块链层面，我们对应就是对一个个区块的数据进行的操作。

例如在我们的代码中，`NewGenesisBlock`代表了创建一个创世区块的意思。`addBlock`代表了添加单个区块。

因为我们在实验中使用了区块链，对应区块链的结构

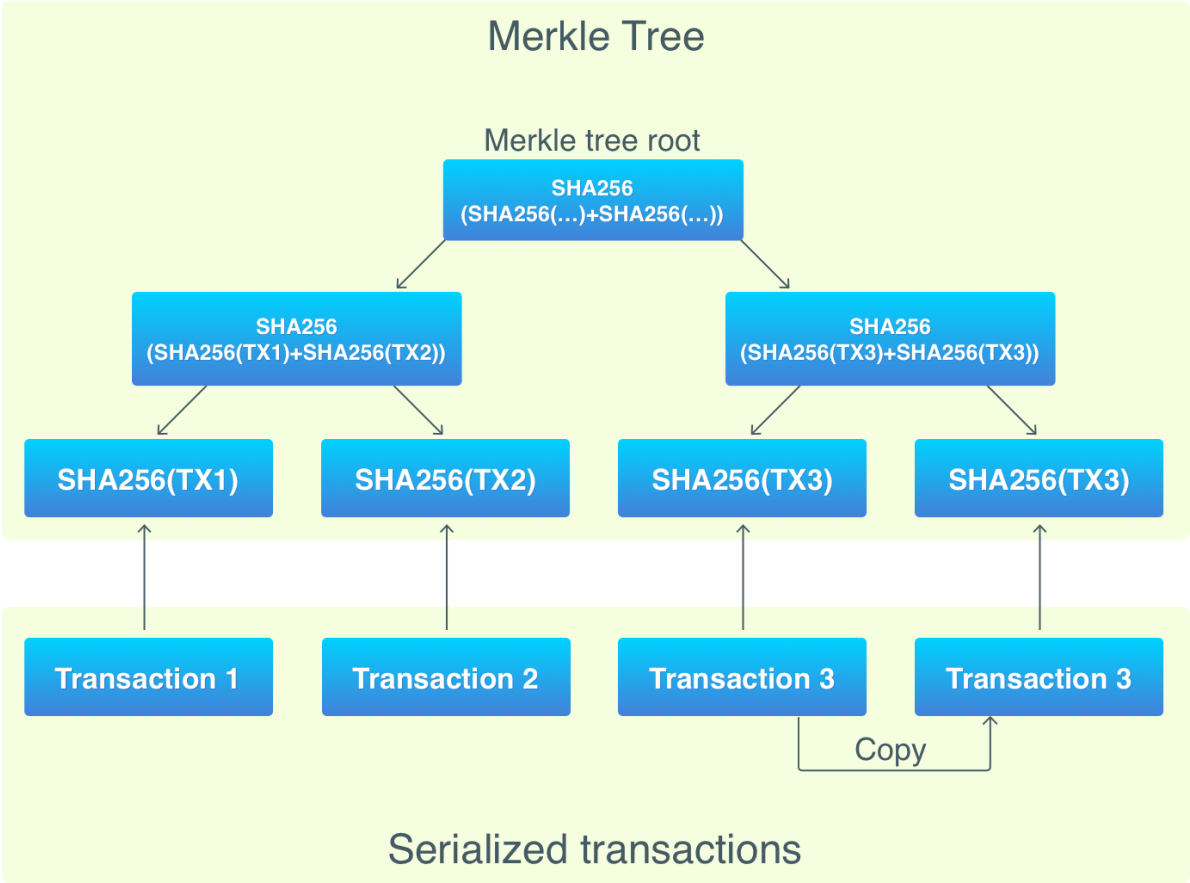
```
type Blockchain struct {
    tip []byte
    db  *bolt.DB
}
```

`tip`代表了最新区块的哈希值，`db`表示了数据库的连接

Merkle树

在Merkle树结构中，我们需要对每一个区块进行节点建立，他是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。

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。



【实验平台】

在本地使用vscode进行代码编辑，在学校提供的试验平台进行代码正确性验证

【实验步骤】

blockchain.go

addblock添加区块

```go
func (bc *Blockchain) AddBlock(data []string) {
    block := NewBlock(data, bc.Iterator().currentHash)
    err := bc.db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        blockInDb := b.Get(block.Hash)

        if blockInDb != nil {
            return nil
        }

        blockData := block.Serialize()
        err := b.Put(block.Hash, blockData)
        if err != nil {
            log.Panic(err)
        }
        return nil
    })
    if err != nil {
        log.Panic(err)
    }

```

```
}
```

实现思路：

- 先循环到最后一个区块，其中使用了`block.go`中的`NewBlock`函数和`blockchain.go`中的`Iterator`函数
- 将当前块的PrevBlockHash设置为前一个块hash，也就连接上了当前块和整个链
- 参考`NewBlockchain`函数中对于数据库的使用，将这个区块的数据加入到链的尾端，调用put函数放入数据库
- 另外有相应的出错处理

Merkle_tree.go

NewMerkleTree创建merkle树

- 在助教给的代码框架下新加入了`func NewMerkleNode(left,right *MerkleNode, data []byte) *MerkleNode`函数，方便地新建merkle树节点

```go
func NewMerkleNode(left,right *MerkleNode, data []byte) *MerkleNode {
    mNode := MerkleNode{}
    if left == nil && right == nil {
        hash := sha256.Sum256(data)
        mNode.Data = hash[:]
    }else {
        prevHashes := append(left.Data,right.Data...)
        hash := sha256.Sum256(prevHashes)
        mNode.Data = hash[:]
    }
    mNode.Left = left
    mNode.Right = right
    return &mNode
}
```

根据传入的左右子树节点和data，分为两种情况：

  - 一种是叶子结点，左右子树为nil，则直接计算hash
  - 另一种是中间节点，按照自底向上，对于节点进行哈希合并，根据子树节点计算出hash

- 新建树的函数是`func NewMerkleTree(data [][]byte) *MerkleTree`，在前一个函数的基础上建立merkle树

```go
func NewMerkleTree(data [][]byte) *MerkleTree {
    var nodes []MerkleNode

    if len(data) % 2 != 0 {
        data = append(data, data[len(data) - 1])
    }

    for _, dataitem := range data {
        node := NewMerkleNode(nil, nil, dataitem)
        nodes = append(nodes, *node)
    }
    for i := 0; i<len(data)/2; i++ {
        var newNodes []MerkleNode

        for j := 0; j < len(nodes); j += 2 {
```

```
            node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
            len := len(newNodes)
            newNodes = append(newNodes , *node)
        }

        nodes = newNodes
    }

    mTree := MerkleTree{&nodes[0]}

    return &mTree
}
```

与实验原理中建立树的过程类似，步骤如下：

- 奇数个节点的话需要把最后一个节点复制一遍
- 然后先循环输入中的data，建立所有的叶子结点
- 根据叶子结点循环建立上层节点，每两个相邻的子节点组成上一层节点，由于数目每次缩减为1/2，最终成为根节点

bonus：merkle树验证

这里我理解的是根据已经建立的merkle树和要验证的交易返回相应的merkle路径和相应的index，其中index为0表示左子树，1表示右子树，如果已知merkle路径还要验证的话直接计算hash就可以了

- 需要修改数据结构：
  - import加入"bytes"，方便比较hash
  - MerkleTree加入Leafs，记录树里面的所有叶子结点，方便根据输入比较确定要验证的椰子节点是哪一个

```
type MerkleTree struct {
    RootNode *MerkleNode
    Leafs    []*MerkleNode
}
```

  - MerkleNode节点加入父节点指针

```
type MerkleNode struct {
    Left  *MerkleNode
    Right *MerkleNode
    Parent *MerkleNode
    Data  []byte
}
```

- 为了方便在建树过程中记录相应的父节点和所有的叶子结点

```
func NewMerkleTree(data [][]byte) *MerkleTree {
    var nodes []MerkleNode

    if len(data) % 2 != 0 {
        data = append(data, data[len(data) - 1])
    }

    for _, dataitem := range data {
```

```
        node := NewMerkleNode(nil, nil, dataitem)
        nodes = append(nodes, *node)
    }
    leafnodes := nodes
    for i := 0; i<len(data)/2; i++ {
        var newNodes []MerkleNode

        for j := 0; j < len(nodes); j += 2 {
            node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
            len := len(newNodes)
            nodes[j].Parent = node
            nodes[j+1].Parent = node
            newNodes = append(newNodes , *node)
        }

        nodes = newNodes
    }

    mTree := MerkleTree{&nodes[0],leafnodes}

    return &mTree
}
```

- 最后是实现函数，根据已经建立的merkle树和要验证的交易返回相应的merkle路径和相应的index

```
func (m *MerkleTree) GetMerklePath(data []byte) ([]MerkleNode, []int64)
{
    //找到要验证的节点
    for _, current := range m.Leafs {
        if bytes.Equal(data, current.Data){
            currentParent := current.Parent
            var merklePath []MerkleNode
            var index []int64
            for currentParent != nil {
                if bytes.Equal(currentParent.Left.Data, current.Data) {
                    merklePath = append(merklePath, currentParent.Right)
                    index = append(index, 1) // add right leaf
                } else {
                    merklePath = append(merklePath, currentParent.Left)
                    index = append(index, 0) // add left leaf
                }
                current = currentParent
                currentParent = currentParent.Parent
            }
            return merklePath, index
        }else{
            return nil, nil
        }
    }
    return nil, nil
}
```

思路是：

  - 先在叶子节点中找到要验证的交易，不断根据父节点指针向上，记录其兄弟节点，直到父节点指针为nil，即根节点，将路径记录在merklePath和index中
- 在已知路径和index之后验证根hash是否正确

```go
func (m *MerkleTree)Verify(path []MerkleNode, index []int64, data []byte) bool{
    prenodehash := data[:]
    for i := 0;i<len(path);i++{
        if index[i] == 0{
            var buf bytes.Buffer
            buf.Write(path[i].Data)
            buf.Write(prenodehash)
            data := buf.Bytes()
            prenodehash := sha256.Sum256(data)
        }else{
            var buf bytes.Buffer
            buf.Write(prenodehash)
            buf.Write(path[i].Data)
            data := buf.Bytes()
            prenodehash := sha256.Sum256(data)
        }
    }
    if m.RootNode.Data = prenodehash{
        return true
    }
    return false

}
```

思路其实和建树差不多，需要注意是左子树还是右子树（由index指出），遍历的时候由于有相应的节点数目，循环其实要简单一点，遍历每个节点依次计算出hash即可

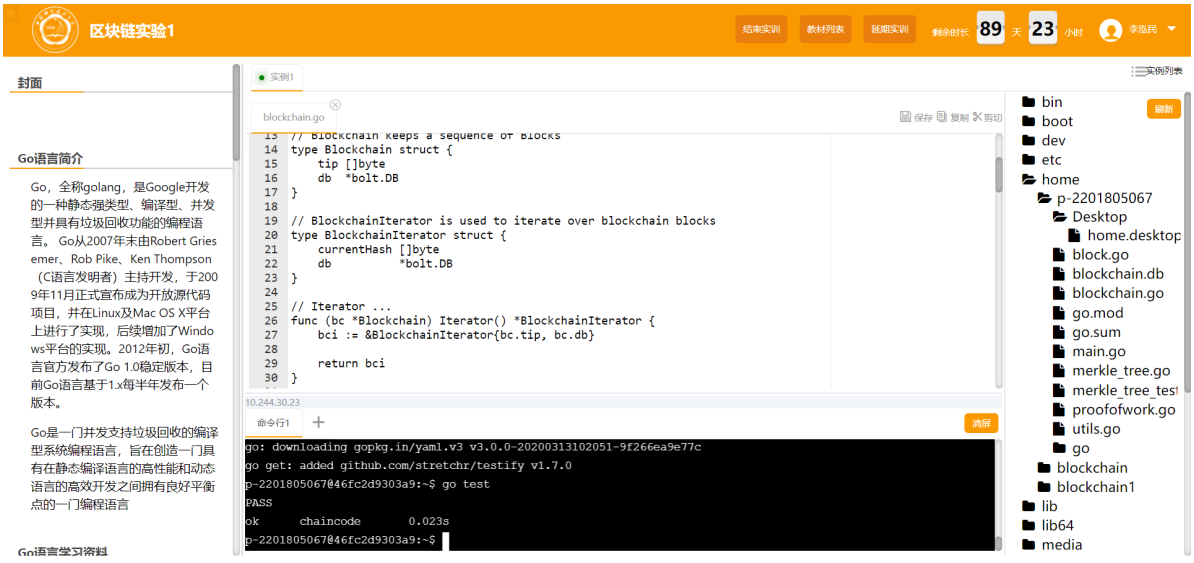
【实验结果】

创建merkle树

在实验平台运行

通过`go test`验证Merkle树建立相关代码是否正确，如果结果为`PASS`，则说明Merkle树建立正确

图中显示正确：



添加区块

在实验平台运行

通过`go run .`来运行区块，`addblock`指令添加区块，`printchain`指令查看区块内容是否正确

图中添加1234内容的块，显示已经加到了区块链上