

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：

```
1 type Block struct {
2     Timestamp    int64 // 时间戳
3     Data          [][]byte //数据
4     PrevBlockHash []byte //前一个区块对应哈希
5     Hash          []byte //当前区块数据对应哈希值
6     Nonce         int //随机数
7 }
```

`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：

```
1 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：

```
1 err = db.Update(func(tx *bolt.Tx) error {
2     ...
3 })
```

对于只读操作的格式如下：

```
1 err = db.View(func(tx *bolt.Tx) error {
2     ...
3 })
```

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

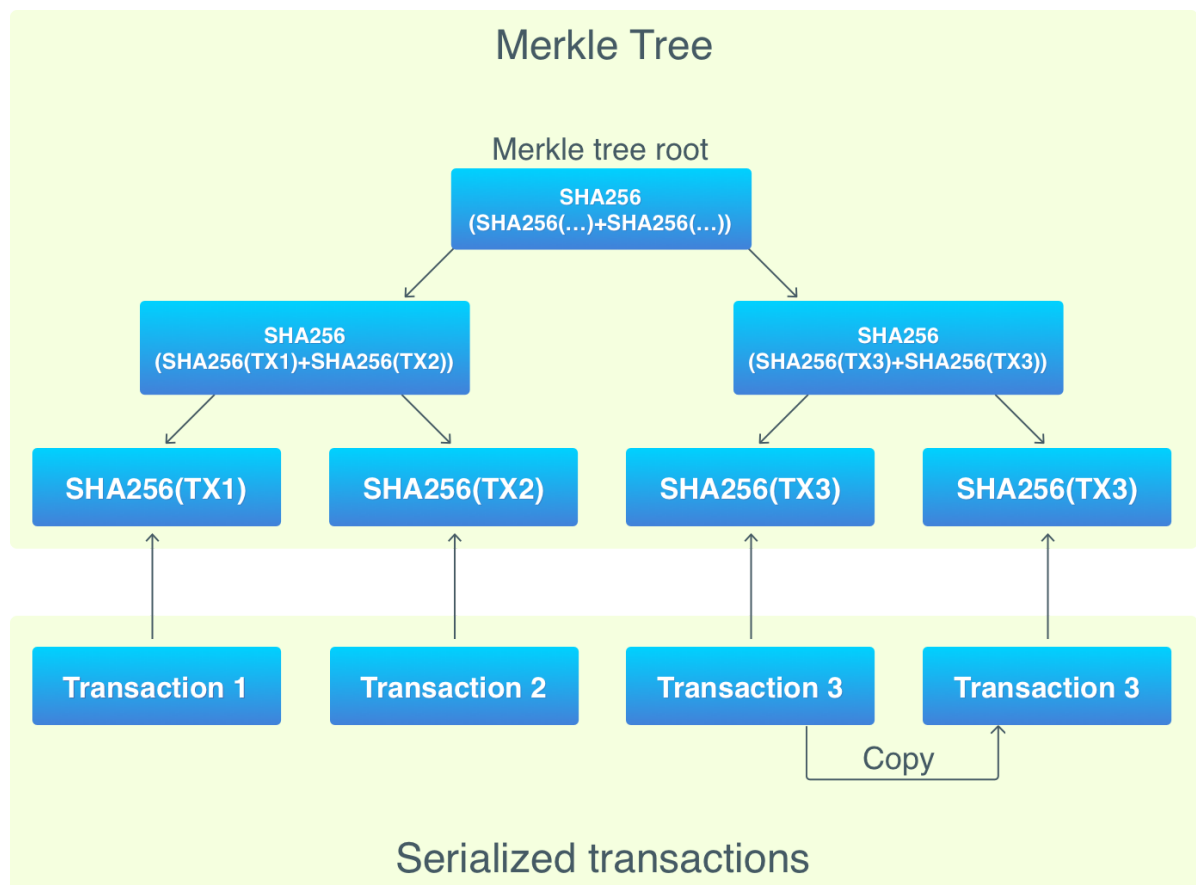
```
1 type Blockchain struct {
2     tip []byte
3     db  *bolt.DB
4 }
```

`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

```
1 func (bc *Blockchain) AddBlock(data []string) {
2     block := NewBlock(data, bc.Iterator().currentHash)
3     err := bc.db.Update(func(tx *bolt.Tx) error {
4         b := tx.Bucket([]byte(blocksBucket))
5         blockInDb := b.Get(block.Hash)
6
7         if blockInDb != nil {
8             return nil
9         }
10
11         blockData := block.Serialize()
12         err := b.Put(block.Hash, blockData)
13         if err != nil {
14             log.Panic(err)
15         }
16         return nil
17     })
18     if err != nil {
19         log.Panic(err)
20     }
21 }
```

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

```

1 func NewMerkleNode(left, right *MerkleNode, data []byte) *MerkleNode {
2     mNode := MerkleNode{}
3     if left == nil && right == nil {
4         hash := sha256.Sum256(data)
5         mNode.Data = hash[:]
6     } else {
7         prevHashes := append(left.Data, right.Data...)
8         hash := sha256.Sum256(prevHashes)
9         mNode.Data = hash[:]
10    }
11    mNode.Left = left
12    mNode.Right = right
13    return &mNode
14 }
```

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

```

1 func NewMerkleTree(data [][]byte) *MerkleTree {
2     var nodes []MerkleNode
3
4     if len(data) % 2 != 0 {
5         data = append(data, data[len(data) - 1])
6     }
7
8     for _, dataitem := range data {
9         node := NewMerkleNode(nil, nil, dataitem)
10        nodes = append(nodes, *node)
11    }
12    for i := 0; i < len(data)/2; i++ {
13        var newNodes []MerkleNode
14
15        for j := 0; j < len(nodes); j += 2 {
```

```

16         node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
17         len := len(newNodes)
18         newNodes = append(newNodes, *node)
19     }
20
21     nodes = newNodes
22 }
23
24 mTree := MerkleTree{&nodes[0]}
25
26 return &mTree
27 }

```

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

```

1 type MerkleTree struct {
2     RootNode *MerkleNode
3     Leafs    []*MerkleNode
4 }

```

- MerkleNode节点加入父节点指针

```

1 type MerkleNode struct {
2     Left  *MerkleNode
3     Right *MerkleNode
4     Parent *MerkleNode
5     Data []byte
6 }

```

- 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

```

1 func NewMerkleTree(data [][]byte) *MerkleTree {
2     var nodes []MerkleNode
3
4     if len(data) % 2 != 0 {
5         data = append(data, data[len(data) - 1])
6     }
7
8     for _, dataitem := range data {

```

```

9      node := NewMerkleNode(nil, nil, dataitem)
10     nodes = append(nodes, *node)
11 }
12 leafnodes := nodes
13 for i := 0; i < len(data)/2; i++ {
14     var newNodes []MerkleNode
15
16     for j := 0; j < len(nodes); j += 2 {
17         node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
18         len := len(newNodes)
19         nodes[j].Parent = node
20         nodes[j+1].Parent = node
21         newNodes = append(newNodes, *node)
22     }
23
24     nodes = newNodes
25 }
26
27 mTree := MerkleTree{&nodes[0], leafnodes}
28
29 return &mTree
30 }

```

- 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

```

1 func (m *MerkleTree) GetMerklePath(data []byte) ([]MerkleNode, []int64)
2 {
3     //找到要验证的节点
4     for _, current := range m.Leafs {
5         if bytes.Equal(data, current.Data){
6             currentParent := current.Parent
7             var merklePath []MerkleNode
8             var index []int64
9             for currentParent != nil {
10                 if bytes.Equal(currentParent.Left.Data, current.Data) {
11                     merklePath = append(merklePath, currentParent.Right)
12                     index = append(index, 1) // add right leaf
13                 } else {
14                     merklePath = append(merklePath, currentParent.Left)
15                     index = append(index, 0) // add left leaf
16                 }
17                 current = currentParent
18                 currentParent = currentParent.Parent
19             }
20             return merklePath, index
21         }else{
22             return nil, nil
23         }
24     }
25     return nil, nil
26 }

```

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

```

1 func (m *MerkleTree)Verify(path []MerkleNode, index []int64, data
  []byte) bool{
2     prenodehash := data[:]
3     for i := 0;i<len(path);i++){
4         if index[i] == 0{
5             var buf bytes.Buffer
6             buf.Write(path[i].Data)
7             buf.Write(prenodehash)
8             data := buf.Bytes()
9             prenodehash := sha256.Sum256(data)
10        }else{
11            var buf bytes.Buffer
12            buf.Write(prenodehash)
13            buf.Write(path[i].Data)
14            data := buf.Bytes()
15            prenodehash := sha256.Sum256(data)
16        }
17    }
18    if m.RootNode.Data == prenodehash{
19        return true
20    }
21    return false
22 }
23 }

```

思路其实和建树差不多，需要注意的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：

The screenshot displays a web-based experiment platform for blockchain. The top navigation bar is orange and includes a logo, the title '区块链实验1' (Blockchain Experiment 1), and buttons for '结束实验' (End Experiment), '教材列表' (Textbook List), '延期实验' (Extend Experiment), '剩余时长' (Remaining Time: 89天 23小时), and a user profile '李冠民'.

The main interface is divided into three sections:

- 封面 (Cover):** Contains a 'Go语言简介' (Go Language Introduction) section. It describes Go as a statically typed, compiled language developed by Google, known for its garbage collection and performance. It mentions its release in 2009 and its use in Linux and macOS.
- Code Editor:** The central area shows a Go file named 'blockchain.go'. The code defines a 'Blockchain' struct with a 'tip' and a 'db', and a 'BlockchainIterator' struct. It includes a 'Verify' function that iterates through the path to verify a data block. The code is highlighted with syntax coloring.
- Terminal:** At the bottom, a terminal window shows the execution of 'go test'. The output indicates that the tests passed ('PASS') and shows the execution time as 0.023s.
- File Explorer:** On the right side, a file explorer shows the project structure, including directories like 'bin', 'boot', 'dev', 'etc', 'home', and files like 'blockchain.db', 'blockchain.go', 'go.mod', 'go.sum', 'main.go', 'merkle_tree.go', 'merkle_tree_test.go', 'proofofwork.go', 'utils.go', and 'go'.

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

10.244.30.23

命令行1 + 清屏

```
p-2201805067@46fc2d9303a9:~$ go run .
chaincode > addblock 1234
[DATA] [1234]
add Success
chaincode > printchain
Prev. hash: 000c80c5f7ca766d63329b76e9d2b57b9c79d7f1806518009a53090b066b007e
Data: [cyhtest2]
Hash: 002f188f4f4017b1c3dbd28025b2ace9a4da904b149b56ff723b3a2cd7c8b617
PoW: true

Prev. hash: 0022bda25c172339d0ef530118603badb6e7956f42f115512c6f356dfce36a96
Data: [cyhtest1]
Hash: 000c80c5f7ca766d63329b76e9d2b57b9c79d7f1806518009a53090b066b007e
PoW: true

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