实验一 区块链编写

介绍

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。

在这个实验中，我们将编写一个简化版的区块链，并且会将数据写入数据库。

区块链概念介绍

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。

在本次实验中，我们会使用一个简化版本的区块结构，大致的内容如下：

```
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 //当前区块哈希值 指数据对应的HASH值
    Nonce         int //随机数
}
```

在这些信息中，`Timestamp`代表了整个区块对应的时间戳，`Data`当前区块存储的数据。Data对应区块链中的交易`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。

数据库

在本次实验中，我们选取了BoltDB的数据库。这是一个简单的，轻量级的集成在Go语言上的数据库。他和通常使用的关系型数据库（MySQL,PostgreSQL等）不同的是，它是一个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。所以，为了存储一个数据，我们需要知道key和bucket。在我们区块链的实验中，我们是希望通过数据库来进行对于区块的存储操作。

在本次使用中，我们可以通过encoding/gob 来进行数据的序列化和反序列化。

数据结构

在比特币代码中，区块主要存储的是两种数据：

1. 区块信息，存储对应每个区块的元数据内容。
2. 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

在我们本次实验中，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

其余信息对于本次实验作用不大。对于数据结构感兴趣的同学，可以查看比特币代码的解析

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。

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 {
...
})
```

例如，所给代码中，区块链的创建代码如下：

```
    err = db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))

        if b == nil {
            fmt.Println("No existing blockchain found. Creating a new one...")
            genesis := NewGenesisBlock()

            b, err := tx.CreateBucket([]byte(blocksBucket))
            if err != nil {
                log.Panic(err)
            }

            err = b.Put(genesis.Hash, genesis.Serialize())
            if err != nil {
                log.Panic(err)
            }

            err = b.Put([]byte("l"), genesis.Hash)
            if err != nil {
                log.Panic(err)
            }
            tip = genesis.Hash
        } else {
            tip = b.Get([]byte("l"))
        }

        return nil
    })
```

其中，我们通过`l`读取的是上一个区块的信息，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，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

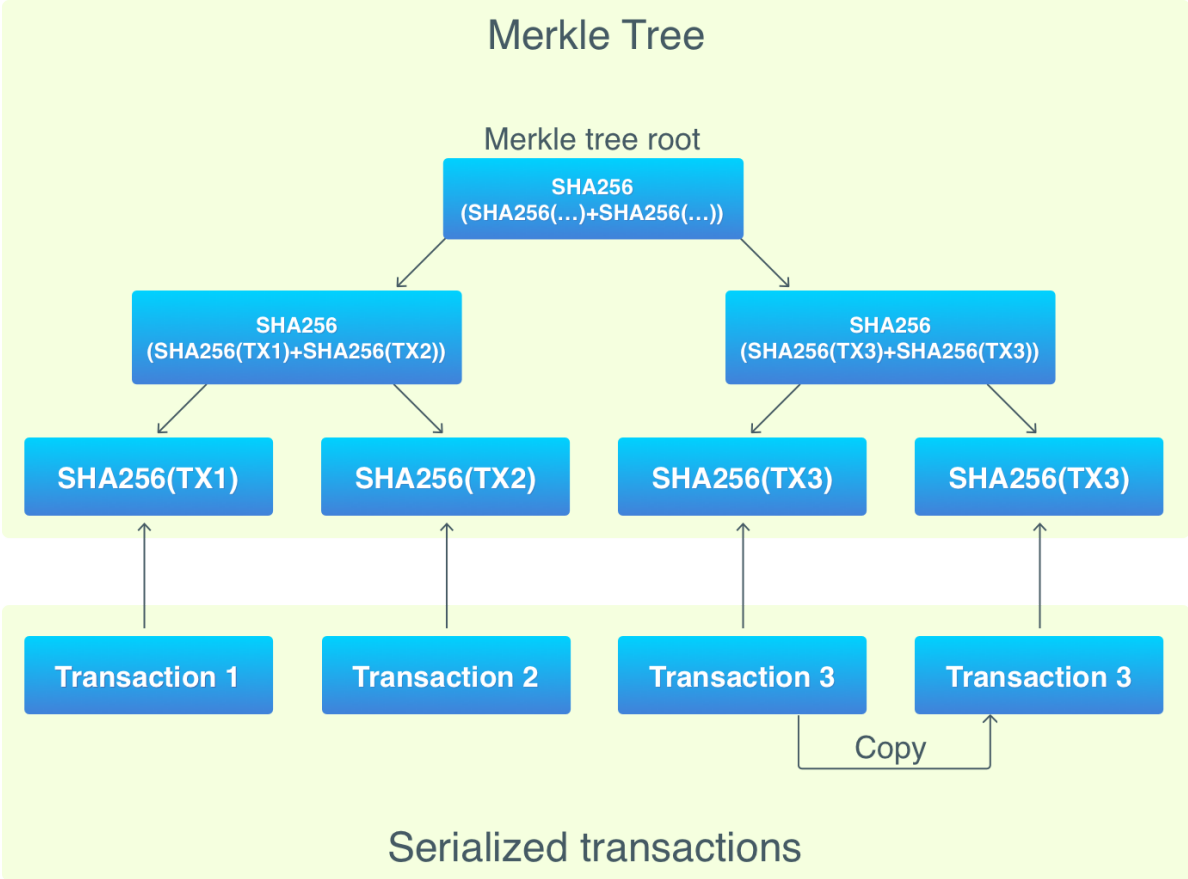
在比特币的白皮书中，是通过**SPV**（Simplified Payment Verification）的方式来进行交易认证的。通过这个机制，我们可以让多个轻节点依赖一个全节点来运行。

在Merkle树结构中，我们需要对每一个区块进行节点建立，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。

这样做的好处是，在我们进行对于特定交易认证的时候，我们不需要下载区块中包含的所有交易，我们，我们只需要验证对应的Merkle根节点和对应的路径。简单的Merkle树示例可以参考图片

Merkle tree的原理部分可以参考资料



基本操作

`go run .`运行项目中的main程序。

`go test`运行项目中的_test文件，这里对应MerkleTree验证部分

目录结构

```
- template
    - block.go // 区块相关代码
    - blockchain.go // 区块链操作相关代码
    - main.go //主程序，为了支持命令行操作
    - merkle_tree.go //merkle树相关代码
    - merkle_tree_test.go //merkle树验证部分相关代码
    - untils.go //简便操作代码，本次实验可以不适用
    - proofofwork.go //POW验证相关代码，本次实验可以不使用
    - blockchain.db //区块链数据
    - go.mod //go模块管理
```

完成部分

blockchain.go

addblock

添加区块

Merkle_tree.go

NewMerkleTree

创建merkle树

bonus

merkle树验证

结果检验

可以通过`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

可以通过`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。

参考资料

比特币白皮书

比特币代码

Merkle Tree

Go语言指南

Go圣经

Go进阶

官方文档