目录

一.区块序列化

本章节讨论如何将区块对象序列化，以便存储至数据库中。

1.序列化概念

互联通讯的双方需要采用约定的协议，序列化和反序列化属于通讯协议的一部分。通讯协议往往采用分层模型，不同模型每层的功能定义以及颗粒度不同，例如：TCP/IP协议是一个四层协议，而OSI模型却是七层协议模型。在OSI七层协议模型中展现层（Presentation Layer）的主要功能是把应用层的对象转换成一段连续的二进制串，或者反过来，把二进制串转换成应用层的对象--这两个功能就是序列化和反序列化。一般而言，TCP/IP协议的应用层对应与OSI七层协议模型的应用层，展示层和会话层，所以序列化协议属于TCP/IP协议应用层的一部分。本文对序列化协议的讲解主要基于OSI七层协议模型。

序列化： 将数据结构或对象转换成二进制串的过程。 反序列化：将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程。

2.对象转换

这个概念是从Java序列化中抽取而来，同样适用于本文的理解

不同的计算机语言中，数据结构，对象以及二进制串的表示方式并不相同。

数据结构和对象：对于类似Java这种完全面向对象的语言，工程师所操作的一切都是对象（Object），来自于类的实例化。在Java语言中最接近数据结构的概念，就是POJO（Plain Old Java Object）或者Javabean－－那些只有setter/getter方法的类。而在C二进制串：序列化所生成的二进制串指的是存储在内存中的一块数据。C语言的字符串可以直接被传输层使用，因为其本质上就是以'0'结尾的存储在内存中的二进制串。在Java语言里面，二进制串的概念容易和String混淆。实际上String 是Java的一等公民，是一种特殊对象（Object）。对于跨语言间的通讯，序列化后的数据当然不能是某种语言的特殊数据类型。二进制串在Java里面所指的是byte[]，byte是Java的8中原生数据类型之一（Primitive data types）。

在Golang中，我们同样是将区块对象转换成字节数组([]byte)进行序列化并进行数据存储。

3. Gob编码工具

gob是Golang包自带的一个数据结构序列化的编码/解码工具。编码使用Encoder，解码使用Decoder。一种典型的应用场景就是RPC(remote procedure calls)。

gob由发送端使用Encoder对数据结构进行编码。在接收端收到消息之后，接收端使用Decoder将序列化的数据变化成本地变量。

gob包是golang提供的“私有”的编解码方式，官方文档中也提及其它的效率会比json，xml等更高。因此在两个Go 服务之间的相互通信建议不要再使用json传递了，完全可以直接使用gob来进行数据传递。

```
import (
   "encoding/gob"
)
```

4.序列化实现

4.1 区块定义

```
//定义区块
type Block struct {
   //1.区块高度，也就是区块的编号，第几个区块
   Height int64
   //2.上一个区块的Hash值
   PreBlockHash []byte
   //3.交易数据（最终都属于transaction 事务）
   Data []byte
   //4.创建时间的时间戳
   TimeStamp int64
   //5.当前区块的Hash值
   Hash []byte
   //6.Nonce 随机数，用于验证工作量证明
   Nonce int64
}
```

4.2 序列化区块

```
// 定义Block的方法Serialize(),将区块序列化成字节数组
func (block *Block) Serialize() []byte {
//1.定义result的字节buffer，用于存储序列化后的区块
   var result bytes.Buffer
//2.初始化序列化对象encoder
   encoder := gob.NewEncoder(&result)
//3.通过Encode()方法对区块进行序列化
   err:=encoder.Encode(block)
   if err !=nil {
```

```
        log.Panic(err)
    }
//4.返回result的字节数组
    return result.Bytes()
}
```

4.3 反序列化字节数组

```
//定义函数DeserializeBlock()，传入参数为字节数组，返回值为Block
func DeserializeBlock(blockBytes []byte) *Block {
//1.定义一个Block指针对象
    var block Block
//2.初始化反序列化对象decoder
    decoder := gob.NewDecoder(bytes.NewReader(blockBytes))
//3.通过Decode()进行反序列化
    err := decoder.Decode(&block)

    if err !=nil {
        log.Panic(err)
    }
//4.返回block对象
    return &block
}
```

二.BoltDB数据库使用

1.BoltDB简介

Bolt是一个纯粹Key/Value模型的程序。该项目的目标是为不需要完整数据库服务器（如Postgres或MySQL）的项目提供一个简单，快速，可靠的数据库。

BoltDB只需要将其链接到你的应用程序代码中即可使用BoltDB提供的API来高效的存取数据。而且BoltDB支持完全可序列化的ACID事务，让应用程序可以更简单的处理复杂操作。

其源码地址为:https://github.com/boltdb/bolt

2.BoltDB特性

BoltDB设计源于LMDB，具有以下特点：

- 使用Go语言编写
- 不需要服务器即可运行
- 支持数据结构
- 直接使用API存取数据，没有查询语句；

- 支持完全可序列化的ACID事务，这个特性比LevelDB强；
- 数据保存在内存映射的文件里。没有wal、线程压缩和垃圾回收；
- 通过COW技术，可实现无锁的读写并发，但是无法实现无锁的写写并发，这就注定了读性能超高，但写性能一般，适合与读多写少的场景。

BoltDB是一个Key/Value（键/值）存储，这意味着没有像SQL RDBMS（MySQL，PostgreSQL等）中的表，没有行，没有列。相反，数据作为键值对存储（如在Golang Maps中）。键值对存储在Buckets中，它们旨在对相似的对进行分组（这与RDBMS中的表类似）。因此，为了获得Value(值)，需要知道该Value所在的桶和钥匙。

3.BoltDB简单使用

```
//通过go get下载并import
import "github.com/boltdb/bolt"
```

3.1 打开或创建数据库

```
db, err := bolt.Open("my.db", 0600, nil)
if err != nil {
   log.Fatal(err)
}
defer db.Close()
```

- 执行注意点

如果通过goland程序运行创建的my.db会保存在$GOPATH /src/Project目录下 如果通过go build main.go ; ./main 执行生成的my.db，会保存在当前目录$GOPATH /src/Project/package下

3.2 数据库操作

3.2.1 创建数据库表与数据写入操作

```
//1. 调用Update方法进行数据的写入
err = db.Update(func(tx *bolt.Tx) error {
//2.通过CreateBucket()方法创建BlockBucket(表)，初次使用创建
   b, err := tx.CreateBucket([]byte("BlockBucket"))
   if err != nil {
      return fmt.Errorf("Create bucket :%s", err)
   }

//3.通过Put()方法往表里面存储一条数据(key,value)，注意类型必须为[]byte
   if b != nil {
      err := b.Put([]byte("l"), []byte("Send $100 TO Bruce"))
      if err != nil {
         log.Panic("数据存储失败..")
      }
```

```
    }

    return nil
})

//数据Update失败，退出程序
if err != nil {
    log.Panic(err)
}
```

3.2.2 数据写入

```
//1.打开数据库
db, err := bolt.Open("my.db", 0600, nil)
if err != nil {
    log.Fatal(err)
}
defer db.Close()

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

//2.通过Bucket()方法打开BlockBucket表
    b := tx.Bucket([]byte("BlockBucket"))
//3.通过Put()方法往表里面存储数据
    if b != nil {
        err := b.Put([]byte("l"), []byte("Send $200 TO Fengyingcong"))
        err =  b.Put([]byte("ll"), []byte("Send $100 TO Bruce"))
        if err != nil {
            log.Panic("数据存储失败..")
        }
    }

    return nil
})
//更新失败
if err != nil {
    log.Panic(err)
}
```

3.2.3 数据读取

```
//1.打开数据库
db, err := bolt.Open("my.db", 0600, nil)
if err != nil {
    log.Fatal(err)
}
defer db.Close()
```

```
//2.通过View方法获取数据
err = db.View(func(tx *bolt.Tx) error {

//3.打开BlockBucket表，获取表对象

   b := tx.Bucket([]byte("BlockBucket"))

//4.Get()方法通过key读取value
   if b != nil {
      data := b.Get([]byte("l"))
      fmt.Printf("%s\n", data)
      data = b.Get([]byte("ll"))
      fmt.Printf("%s\n", data)
   }

   return nil
})

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

三.通过BoltDB存储区块

该代码包含对BoltDB的数据库创建，表创建，区块添加，区块查询操作

```
//1.创建一个区块对象block
block := BLC.NewBlock("Send $500 to Tom", 1, []byte{0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0})

//2. 打印区块对象相关信息
fmt.Printf("区块的Hash信息为:\t%x\n", block.Hash)
fmt.Printf("区块的数据信息为:\t%v\n", string(block.Data))
fmt.Printf("区块的随机数为:\t%d\n", block.Nonce)

//3. 打开数据库
db, err := bolt.Open("my.db", 0600, nil)
if err != nil {
   log.Fatal(err)
}
defer db.Close()

//4. 更新数据
err = db.Update(func(tx *bolt.Tx) error {

//4.1 打开BlockBucket表对象
   b := tx.Bucket([]byte("blocks"))
//4.2 如果表对象不存在，创建表对象
```

```
    if b == nil {
        b, err = tx.CreateBucket([]byte("blocks"))
        if err != nil {
            log.Panic("Block Table Create Failed")
        }
    }

    //4.3 往表里面存储一条数据(key,value)
    err = b.Put([]byte("l"), block.Serialize())
    if err != nil {
        log.Panic("数据存储失败..")
    }
    return nil
})

//更新失败,返回错误
if err != nil {
    log.Panic("数据更新失败")
}

//5. 查看数据
err = db.View(func(tx *bolt.Tx) error {

//5.1打开BlockBucket表对象
    b := tx.Bucket([]byte("blocks"))

    if b != nil {
        //5.2 取出key=“l”对应的value
        blockData := b.Get([]byte("l"))
        //5.3反序列化
        block := BLC.DeserializeBlock(blockData)
 //6. 打印区块对象相关信息
     fmt.Printf("区块的Hash信息为:\t%x\n", block.Hash)
     fmt.Printf("区块的数据信息为:\t%v\n", string(block.Data))
     fmt.Printf("区块的随机数为:\t%d\n", block.Nonce)
    }

    return nil
})
//数据查看失败
if err != nil {
    log.Panic("数据更新失败")
}
```

四.区块链基于BoltDB存储区块

1. 定义区块属性与方法 Block.go

1.1 导入相关库

```
import (
    "time"
    "bytes"
    "encoding/gob" //编码解码库
    "log"
)
```

1.2 定义区块属性

```
//定义区块属性
type Block struct {
    //1.区块高度，也就是区块的编号，第几个区块
    Height int64
    //2.上一个区块的Hash值
    PreBlockHash []byte
    //3.交易数据（最终都属于transaction 事务）
    Data []byte
    //4.创建时间的时间戳
    TimeStamp int64
    //5.当前区块的Hash值
    Hash []byte
    //6.Nonce 随机数，用于验证工作量证明
    Nonce int64
}
```

1.3 序列化区块

```
// 序列化区块（Block类对象转成字节数组）
func (block *Block) Serialize() []byte {
    var result bytes.Buffer

    encoder := gob.NewEncoder(&result)
    err:=encoder.Encode(block)
    if err !=nil {
        log.Panic(err)
    }
    return result.Bytes()

}
```

1.4 反序列化

```
//反序列化
func DeserializeBlock(blockBytes []byte) *Block {
   var block Block

   decoder := gob.NewDecoder(bytes.NewReader(blockBytes))

   err := decoder.Decode(&block)

   if err !=nil {
      log.Panic(err)
   }
   return &block
}
```

1.5 创建区块

```
//传入参数data,height,PreBlockHash,返回值*Block
func NewBlock(data string, height int64, PreBlockHash []byte) *Block {
   //根据传入参数创建区块
   block := &Block{
      height,
      PreBlockHash,
      []byte(data),
      time.Now().Unix(),
      nil,
      0,
   }
   //调用工作量证明的方法，并且返回有效的Hash和Nonce值
   //创建pow对象
   pow := NewProofOfWork(block)
    //通过Run()方法进行挖矿验证
   hash, nonce := pow.Run(height)
//将Nonce,Hash赋值给类对象属性
   block.Hash = hash[:]
   block.Nonce = nonce
   return block

}
```

1.6 创建创世区块

```
//单独为创世区块定义的函数
func CreateGenesisBlock(data string) *Block {

   return NewBlock(data, 1, []byte{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0})

}
```

1.7 代码整合

```
package BLC

import (
   "time"
   "bytes"
   "encoding/gob"
   "log"
)

//定义区块
type Block struct {
   //1.区块高度，也就是区块的编号，第几个区块
   Height int64
   //2.上一个区块的Hash值
   PreBlockHash []byte
   //3.交易数据（最终都属于transaction 事务）
   Data []byte
   //4.创建时间的时间戳
   TimeStamp int64
   //5.当前区块的Hash值
   Hash []byte
   //6.Nonce 随机数，用于验证工作量证明
   Nonce int64
}

// 序列化区块（Block类对象转成字节数组）
func (block *Block) Serialize() []byte {
   var result bytes.Buffer

   encoder := gob.NewEncoder(&result)
   err:=encoder.Encode(block)
   if err !=nil {
      log.Panic(err)
   }
   return result.Bytes()

}
```

```
//反序列化，字节数组==>区块
func DeserializeBlock(blockBytes []byte) *Block {
   var block Block

   decoder := gob.NewDecoder(bytes.NewReader(blockBytes))

   err := decoder.Decode(&block)

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


//1. 创建新的区块
func NewBlock(data string, height int64, PreBlockHash []byte) *Block {
   //创建区块
   block := &Block{
      height,
      PreBlockHash,
      []byte(data),
      time.Now().Unix(),
      nil,
      0,
   }

   //创建pow对象
   pow := NewProofOfWork(block)
   //调用工作量证明的方法，并且返回有效的Hash和Nonce值
   hash, nonce := pow.Run(height)

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

}

//2.生成创世区块
func CreateGenesisBlock(data string) *Block {

   return NewBlock(data, 1, []byte{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0})

}
```

2.定义POW属性与方法ProofOfWork.go

2.1 导入相关库

```
import (
   "math/big" //导入big包
   "bytes"
   "crypto/sha256" //导入sha256密码库
   "fmt"
   "time"
)
```

- **关于big包的说明**

使用Go语言中的float64类型进行浮点运算，返回结果将精确到15位，足以满足大多数的任务。当对超出int64或者uint64 类型这样的大数进行计算时，如果对精度没有要求，float32 或者 float64 可以胜任，但如果对精度有严格要求的时候，我们不能使用浮点数，在内存中它们只能被近似的表示。

对于整数的高精度计算 Go 语言中提供了 big 包。其中包含了 math 包：有用来表示大整数的 big.Int 和表示大有理数的 big.Rat 类型（可以表示为 2/5 或 3.1416 这样的分数，而不是无理数或 π）。这些类型可以实现任意位类型的数字，只要内存足够大。缺点是更大的内存和处理开销使它们使用起来要比内置的数字类型慢很多。

大的整型数字是通过 big.NewInt(n) 来构造的，其中 n 为 int64 类型整数。而大有理数是用过 big.NewRat(N,D) 方法构造。N（分子）和 D（分母）都是 int64 型整数。因为 Go 语言不支持运算符重载，所以所有大数字类型都有像是 Add() 和 Mul() 这样的方法。它们作用于作为 receiver 的整数和有理数，大多数情况下它们修改 receiver 并以 receiver 作为返回结果。因为没有必要创建 big.Int 类型的临时变量来存放中间结果，所以这样的运算可通过内存链式存储。

2.2 定义POW结构体属性

```
type ProofOfWork struct {
   Block  *Block   //当前要验证的区块
   target *big.Int //大数存储，区块难度
}
```

2.3 定义常量 TargetBit

```
//256位Hash里面至少要有16个零0000 0000 0000 0000
const TargetBit = 16
```

2.4 拼接区块属性

```
func (pow *ProofOfWork) prePareData(nonce int) []byte {

   data := bytes.Join(
      [][]byte{
         pow.Block.PreBlockHash,
```

```
            pow.Block.Data,
            IntToHex(pow.Block.TimeStamp),
            IntToHex(int64(TargetBit)),
            IntToHex(int64(nonce)),
            IntToHex(int64(pow.Block.Height)),
        },
        []byte{},
    )
    return data
}
```

2.5 创建工作量证明对象

```
func NewProofOfWork(block *Block) *ProofOfWork {
    //1.创建初始值为1的target
    target := big.NewInt(1)
    //2.左移256-TargetBit
    target = target.Lsh(target, 256-TargetBit)
    return &ProofOfWork{block, target}
}
```

2.6 通过POW类对象Run方法挖矿

```
func (proofOfWork *ProofOfWork) Run(num int64) ([]byte, int64) {
//初始化随机数nonce为0
    nonce := 0
//存储新生成的hash值
    var hashInt big.Int
//存储hash值
    var hash [32]byte

    for {
        //1. 将Block的属性拼接成字节数组，注意，参数为Nonce
        databytes := proofOfWork.prePareData(nonce)
        //2.将拼接后的字节数组生成Hash
        hash = sha256.Sum256(databytes)
        //3. 将hash存储至hashInt
        hashInt.SetBytes(hash[:])
        //4.判断hashInt是否小于Block里面的Target
        // Cmp compares x and y and returns:
        //
        //   -1 if x <  y
        //    0 if x == y
        //   +1 if x >  y
        //此处需要满足hashInt(y)小于设置的target(x)即 x > y,则挖矿成功
        if proofOfWork.target.Cmp(&hashInt) == 1 {
```

```
            fmt.Printf("第%d个区块，挖矿成功:%x\n", num, hash)
            fmt.Println(time.Now())
            time.Sleep(time.Second * 2)
            //挖矿成功，退出循环
            break
        }
        nonce ++
    }

    return hash[:], int64(nonce)

}
```

2.7 代码整合

```
package BLC

import (
    "math/big"
    "bytes"
    "crypto/sha256"
    "fmt"
    "time"
)

type ProofOfWork struct {
    Block  *Block   //当前要验证的区块
    target *big.Int //大数存储，区块难度
}

//数据拼接，返回字节数组
func (pow *ProofOfWork) prePareData(nonce int) []byte {

    data := bytes.Join(
        [][]byte{
            pow.Block.PreBlockHash,
            pow.Block.Data,
            IntToHex(pow.Block.TimeStamp),
            IntToHex(int64(TargetBit)),
            IntToHex(int64(nonce)),
            IntToHex(int64(pow.Block.Height)),
        },
        []byte{},
    )
    return data
}

//256位Hash里面至少要有16个零0000 0000 0000 0000
```

```
const TargetBit = 16
//判断挖矿得到的区块是否有效
func (proofOfWork *ProofOfWork) IsValid() bool {
   //1.proofOfWork.Block.Hash
   //2.proofOfWork.Target
   var hashInt big.Int

   hashInt.SetBytes(proofOfWork.Block.Hash)

   if proofOfWork.target.Cmp(&hashInt) == 1 {
      return true
   }
   return false
}

//创建新的工作量证明对象
func NewProofOfWork(block *Block) *ProofOfWork {
   /*1.创建初始值为1的target
   0000 0001
   8 - 2
   */

   target := big.NewInt(1)

   //2.左移256-targetBit
   target = target.Lsh(target, 256-TargetBit)

   return &ProofOfWork{block, target}
}


func (proofOfWork *ProofOfWork) Run(num int64) ([]byte, int64) {


   nonce := 0
   var hashInt big.Int //存储新生成的hash值
   var hash [32]byte

   for {
      //1. 将Block的属性拼接成字节数组
      databytes := proofOfWork.prePareData(nonce)

      //2.生成Hash
      hash = sha256.Sum256(databytes)
      //fmt.Printf("挖矿中..%x\n", hash)
      //3. 将hash存储至hashInt
      hashInt.SetBytes(hash[:])

      //4.判断hashInt是否小于Block里面的target
```

```
        // Cmp compares x and y and returns:
        //
        //   -1 if x <  y
        //    0 if x == y
        //   +1 if x >  y
        //需要hashInt(y)小于设置的target(x)
        if proofOfWork.target.Cmp(&hashInt) == 1 {
            //fmt.Println("挖矿成功", hashInt)
            fmt.Printf("第%d个区块，挖矿成功:%x\n", num, hash)
            fmt.Println(time.Now())
            time.Sleep(time.Second * 2)
            break

        }

        nonce ++

    }

    return hash[:], int64(nonce)

}
```

3.区块链属性与方法 BlockChain.go

3.1 导入相关库

```
import (
    "github.com/boltdb/bolt" //导入bolt库，用于操作DB
    "log"
)
```

3.2 创建常量

```
const dbName = "blockchain.db"  //数据库名
const blockTableName = "blocks" //表名
```

3.3 创建区块链属性

```
type BlockChain struct {
    Tip []byte   //区块链里面最后一个区块的Hash
    DB  *bolt.DB //数据库
}
```

3.4 创建带有创世区块的区块链

```go
func CreateBlockChainWithGenesisBlock() *BlockChain {
    //1. 创建或者打开数据库
   db, err := bolt.Open(dbName, 0600, nil)
   if err != nil {
      log.Fatal(err)
   }
    //2.创建存储
   var blockHash []byte
    //3.更新数据
   err = db.Update(func(tx *bolt.Tx) error {
      //获取表
      b := tx.Bucket([]byte(blockTableName))
      if b == nil { //如果不存在block表，则进行创建
         b, err = tx.CreateBucket([]byte(blockTableName))
         if err != nil {
            log.Panic(err)
         }

      }

       //4.调用Block.go中的函数CreateGenesisBlock创建创世区块
      genesisBlock := CreateGenesisBlock("Genesis Data..")
       //5.将创世区块存储至表中(key=当前区块的Hash值,value=当前区块的序列化字节数组)
      err := b.Put(genesisBlock.Hash, genesisBlock.Serialize())
      if err != nil {
         log.Panic(err)
      }

       //6.存储最新的区块链的hash（key="l",value=当前区块的Hash值)
      err = b.Put([]byte("l"), genesisBlock.Hash)
      if err != nil {
         log.Panic(err)
      }
       //7.定义该区块链的Tip值为最新区块的Hash值
      blockHash = genesisBlock.Hash

      return nil
   })

    //返回区块链对象(Tip:blockHash,DB:进行Update操作的db对象)
   return &BlockChain{blockHash, db}

}
```

3.5 添加新区块至区块链中

```
func (blc *BlockChain) AddBlockChain(data string) {
    //获取db对象blc.DB
   err := blc.DB.Update(func(tx *bolt.Tx) error {
       //1.获取表
      b := tx.Bucket([]byte(blockTableName))
      //2.创建新区块
      if b != nil {
          //通过Key:blc.Tip获取Value(区块序列化字节数组)
         byteBytes := b.Get(blc.Tip)
          //反序列化出最新区块(上一个区块)对象
         block := DeserializeBlock(byteBytes)

         //3.通过NewBlock进行挖矿生成新区块newBlock
         newBlock := NewBlock(data, block.Height+1, block.Hash)
          //4.将最新区块序列化并且存储到数据库中(key=新区块的Hash值，value=新区块序列
化)
         err := b.Put(newBlock.Hash, newBlock.Serialize())
         if err != nil {
            log.Panic(err)
         }

/*5.更新数据库中"l"对应的Hash为新区块的Hash值
用途:便于通过该Hash值找到对应的Block序列化，从而找到上一个Block对象，为生成新区块函数
NewBlock提供高度Height与上一个区块的Hash值PreBlockHash
*/
         err = b.Put([]byte("l"), newBlock.Hash)
         if err != nil {
            log.Panic(err)
         }
         //6. 更新Tip值为新区块的Hash值
         blc.Tip = newBlock.Hash
      }

      return nil

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

}
```

3.6 代码整合

```
package BLC

import (
   "github.com/boltdb/bolt"
```

```
    "log"
)

const dbName = "blockchain.db"  //数据库名
const blockTableName = "blocks" //表名
type BlockChain struct {
    Tip []byte   //区块链里面最后一个区块的Hash
    DB  *bolt.DB //数据库
}

//1.创建带有创世区块的区块链
func CreateBlockChainWithGenesisBlock() *BlockChain {
    //创建或者打开数据库
    db, err := bolt.Open(dbName, 0600, nil)
    if err != nil {
        log.Fatal(err)
    }

    var blockHash []byte
    err = db.Update(func(tx *bolt.Tx) error {
        //获取表
        b := tx.Bucket([]byte(blockTableName))
        if b == nil {
            b, err = tx.CreateBucket([]byte(blockTableName))
            if err != nil {
                log.Panic(err)
            }

        }

        //创建创世区块
        genesisBlock := CreateGenesisBlock("Genesis Data..")
        //将创世区块存储至表中
        err := b.Put(genesisBlock.Hash, genesisBlock.Serialize())
        if err != nil {
            log.Panic(err)
        }

        //存储最新的区块链的hash
        err = b.Put([]byte("l"), genesisBlock.Hash)
        if err != nil {
            log.Panic(err)
        }

        blockHash = genesisBlock.Hash

        return nil
    })
```

```
    //返回区块链对象
    return &BlockChain{blockHash, db}

}

func (blc *BlockChain) AddBlockChain(data string) {

    err := blc.DB.Update(func(tx *bolt.Tx) error {
        //1.获取表
        b := tx.Bucket([]byte(blockTableName))
        //2.创建新区块
        if b != nil {
            //获取最新区块
            byteBytes := b.Get(blc.Tip)
            //反序列化
            block := DeserializeBlock(byteBytes)

            //3. 将区块序列化并且存储到数据库中
            newBlock := NewBlock(data, block.Height+1, block.Hash)
            err := b.Put(newBlock.Hash, newBlock.Serialize())
            if err != nil {
                log.Panic(err)
            }

            //4.更新数据库中"l"对应的Hash
            err = b.Put([]byte("l"), newBlock.Hash)
            if err != nil {
                log.Panic(err)
            }
            //5. 更新blockchain的Tip
            blc.Tip = newBlock.Hash
        }

        return nil

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

}
```

4.自定义帮助库:help.go

```
package BLC

import (
    "bytes"
```

```
    "encoding/binary"
    "log"
)

//将int64转换为字节数组
func IntToHex(num int64) []byte {
    buff := new(bytes.Buffer)
    err := binary.Write(buff, binary.BigEndian, num)
    if err != nil {
        log.Panic(err)
    }
    return buff.Bytes()
}
```

5. 测试代码 main.go

```
package main

import (
    "publicChain/BLC" //引用BLC包
    "fmt"
)

func main() {

    fmt.Println("开始挖矿")
    //创建创世区块
    blockChain := BLC.CreateBlockChainWithGenesisBlock()
    defer blockChain.DB.Close()

    //创建新的区块
    blockChain.AddBlockChain("Send $100 to Bruce")

    blockChain.AddBlockChain("Send $200 to Apple")

    blockChain.AddBlockChain("Send $300 to Alice")

    blockChain.AddBlockChain("Send $400 to Bob")

}
```

挖矿成功，数据已经存入blockchain.db中

五.遍历区块链区块信息

1.定义遍历函数

上述操作已经将区块挖出并且通过序列化存储至基于BoltDB的区块链中，本节将通过区块链对象中Tip的值逐步取出所有区块的相关信息，该方法**PrintChain()**集成在**BlockChain.go**中

```
func (blc *BlockChain) PrintChain() {
    //1.定义区块对象block
   var block *Block
    //2.定义当前区块Hash值对象currentHash
   var currentHash = blc.Tip
    //3.从数据库对象blc.DB中循环取值
   for {
      err := blc.DB.View(func(tx *bolt.Tx) error {
          //4. 打开表 blockTableName
         b := tx.Bucket([]byte(blockTableName))
         if b != nil {
             //5.通过获取Key:currentHash获得对应的Value:当前区块的序列化字节数组
            blockBytes := b.Get(currentHash)
             //6.反序列化字节数组，获取最新区块对象
            block = DeserializeBlock(blockBytes)
             //7. 打印获取到的最新区块对象的相关属性
            fmt.Printf("Height:%d\n", block.Height)
            fmt.Printf("PreBlockHash:%x\n", block.PreBlockHash)
            fmt.Printf("Data:%s\n", block.Data)
            fmt.Printf("TimeStamp:%s\n", time.Unix(block.TimeStamp,
0).Format("2006-01-02 03:04:05 PM"))
            fmt.Printf("Hash:%x\n", block.Hash)
            fmt.Printf("Nonce:%d\n", block.Nonce)
         }

         return nil
      })

      if err != nil {
         log.Panic("数据库查询失败", err)
      }
       //8.定义存储最新区块对应的上一个区块Hash值
      var hashInt big.Int
      hashInt.SetBytes(block.PreBlockHash)
       //9.判断该区块是否为创世区块，如果是，则退出循环遍历
      if big.NewInt(0).Cmp(&hashInt) == 0 {
         fmt.Println("该区块为创世区块，退出程序")
         break
      }
      /*10.如果该区块不是创世区块，则将用于表数据查询的key的值currentHash的值用上一个
区块Hash赋值，继续进行取值循环，直至找到创世区块
      */
      currentHash = block.PreBlockHash
      fmt.Println()
```

```
    }

}
```

2.测试代码main.go

- 新增blockChain.PrintChain()

```
package main

import (
    "publicChain/part17-数据迭代/BLC"
    "fmt"
)

func main() {

    fmt.Println("开始挖矿")
    //创建创世区块
    blockChain := BLC.CreateBlockChainWithGenesisBlock()
    defer  blockChain.DB.Close()

    ////创建新的区块
    blockChain.AddBlockChain("Send $100 to Bruce")

    blockChain.AddBlockChain("Send $200 to Apple")

    blockChain.AddBlockChain("Send $300 to Alice")

    blockChain.AddBlockChain("Send $400 to Bob")

    blockChain.PrintChain()
}
```

通过遍历函数，打印出从最后一个区块至创世区块的所有信息。

六.通过迭代器遍历区块信息

1.迭代器属性与方法 BlockChainIterator.go

1.1 导入相关库

```
import (
   "github.com/boltdb/bolt" //导入BoltDB库
   "log"
)
```

1.2 定义属性

```
type BlockChainIterator struct {
   CurrentHash []byte   // 保存当前的区块Hash值
   DB          *bolt.DB //DB对象
}
```

1.3 迭代方法

```
func (blockchainIterator *BlockChainIterator) Next() *Block {
    //1.定义Block对象block
   var block *Block
    //2.操作DB对象blockchainIterator.DB
   err := blockchainIterator.DB.View(func(tx *bolt.Tx) error {
    //3.打开表对象blockTableName
      b := tx.Bucket([]byte(blockTableName))

      if b != nil {
          //Get()方法通过Key:当前区块的Hash值获取当前区块的序列化信息
         currentBlockBytes := b.Get(blockchainIterator.CurrentHash)
         //反序列化出当前的区块
         block = DeserializeBlock(currentBlockBytes)
         //更新迭代器里面的CurrentHash
         blockchainIterator.CurrentHash = block.PreBlockHash
      }
      return nil

   })

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

2.修改遍历方法 BlockChain.go

2.1 添加 Iterator() 方法

```
func (blockchain *BlockChain) Iterator() *BlockChainIterator {
   return &BlockChainIterator{blockchain.Tip, blockchain.DB}
}
```

2.2 修改PrintChain()方法

```
func (blc *BlockChain) PrintChain() {
    //创建迭代类对象blockchainIterator
   blockchainIterator := blc.Iterator()

   for {
       //通过Next()方法获取当前区块，并更新区块链对象保存的Hash值为上一个区块的Hash值
      block := blockchainIterator.Next()

      fmt.Printf("Height:%d\n", block.Height)
      fmt.Printf("PreBlockHash:%x\n", block.PreBlockHash)
      fmt.Printf("Data:%s\n", block.Data)
      fmt.Printf("TimeStamp:%s\n",
time.Unix(block.TimeStamp,0).Format("2006-01-02 03:04:05 PM"))
      fmt.Printf("Hash:%x\n", block.Hash)
      fmt.Printf("Nonce:%d\n", block.Nonce)

      var hashInt big.Int
      hashInt.SetBytes(block.PreBlockHash)

      if big.NewInt(0).Cmp(&hashInt) == 0 {
         break
      }

   }

}
```

通过迭代器的方式其实与直接遍历区块信息的结果没有任何差别，只是对数据操作的代码进行了封装，也能从一定程度上对程序进行了简单的解耦合设计，推荐使用。

冯颖聪 20180708