实验二：使用Go语言构造区块链

一、 实验概述

区块链是21世纪最具先进性的代表性技术之一。目前， 它仍然处于并将长期处于不断成长的时期，而且区块链技术还有很多潜在的力量没有完全展露出来。 从本质上来讲，区块链可以类比作一个分布式数据库。不过， 区块链与传统的分布式数据库的差别在于，区块链的数据库是公开的，具有大量的冗余备份。 也就是说，每个使用它的人， 都将在自己的机器上拥有一个完整的副本。 而向数据库中添加新的记录， 必须经过其他“矿工”的同意才可以。 除此以外， 也是因为区块链的兴起，才使得加密货币和智能合约这一新兴技术成为可能。本实验将在Go语言的环境下，实现一个简化版的区块链。

二、 预备知识

工作量证明：在区块链系统中，其中的一个核心关键点就是，一个人想要将数据放入到区块链中，必须经过一系列困难的工作。区块链的奖励也由此而来，正是由于完成了这种困难的工作，才使得区块链的安全性和一致性得到了保证，而完成这个工作的人，也会获得相应奖励（这也就是通过挖矿获得token的过程）。

这个机制称为激励机制，没有人愿意义务劳动。在区块链中，是通过矿工（网络中的实际参与人）连续不停的工作，进而支撑起整个系统网络。矿工不断地向区块链中加入新块，然后获得相应的奖励。在这种机制的作用下，新生成的区块能够被安全地加入到区块链中，它维护了整个区块链数据库的稳定性。值得注意的是，完成了这个工作的人必须要证明这一点，即他必须要证明他的确完成了这些工作。

整个机制，就叫做工作量证明（Proof-of-Work）。事实上，要想快速的完成工作，是一件非常不容易的事情，因为这需要大量的计算能力：即便是一台高性能计算机，也无法在短时间内快速完成。另外，随着时间不断增长，这件工作会变得越来越困难，以保持平均每一块10分钟的生成速度。在比特币系统中，这个工作就是要设法找到一个块的哈希，同时这个哈希满足了一些必要条件。而这个哈希，也就充当了证明的角色。因此，快速寻求到有效的哈希值，就是矿工实际在做的事情。

BoltDB: Bolt 是一个 Go编写的kv数据库，这一项目启发自 Howard Chu 的 LMDB。它旨在为那些不需要维护一个庞大的，像 Postgres 和 MySQL 这样的，完整的数据库服务器的项目，提供一个简单、快速和可靠的kv数据库。并且由于 Bolt 主要目的在于提

供一些底层基础的数据库功能，因此简洁便成为其关键所在。

Bolt 使用键值存储，这意味着它没有像 SQL RDBMS （MySQL，PostgreSQL 等等）的表，也就是意味着他没有行和列。相反，数据在SQL中被存储为键值对的性质（key-value pair，就像 Golang 的 map）。键值对被存储在若干个bucket 中，这是为了将相似的键值对进行分组（类似 RDBMS 中的表格）。因此，为了获取一个值，你需要知道一个 bucket 和他的key值。

需要注意的一个事情是，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）），我们需要对它们进行序列化， 也就说，实现一个从 Gostruct 转换到一个 bytearray 的机制，同时还可以从一个 bytearray 再转

换回 Go struct。虽然我们将会使用 https://golang.org/pkg/encoding/gob/ 来完成这一目标，但实际上也可以选择使用 JSON,XML,Protocol Buffers 等等。之所以选择使用 encoding/gob,是因为它很简单，而且是 Go 标准库的一部分。

三、 实验准备

实验系统：Win10，Linux，Mac OS

均可要求环境：Go 1.13.15或更高

注：线上实验环境以Linux系统及VScode为基础，使用前需要预配置实验环境。

（1）

四、 实验内容

实验1：构建区块

区块链的基本构成单位是区块，而区块又分为区块头和区块体两部分，我们所实现的简单区块链自然也是将以下字段分为这两部分：

字段	解释	数据类型
Time	当前时间戳，也就是区块创建的时间	int64
PrevHash	前一个块的哈希，即父哈希	[]byte
Hash	当前块的哈希	[]byte
Data	区块存储的实际有效信息，也就是交易	[]byte

其中 Timestamp，PrevHash, Hash属于区块头，而Data则属于区块体。完整的比特币的区块头（block header）结构可参考https://en.bitcoin.it/wiki/Block_hashing_algorithm，目前仅供了解：

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-bithashbasedonallof the transactionsinthe 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)	Ahashistried(increments)	4

而我们的Data实际是区块链中的”交易（Transaction)”字段，目前暂时不会涉及太过复杂的结构，只需要知道是一串字符信息即可。

而Hash字段，则表示当前区块的哈希，它是由工作量证明算法计算得来，是区块链安全性的基石，这一部分内容也将在后续实验环节中进行学习。

练习1：下载实验附件并解压，将ex2.1中的blockchain_demo复制到/home/coder/project/

目录下，完成以下步骤：

将Block类中元素补充完整；

将Block.CalHash()函数补全，实现对Block的Hash计算。其中 Hash =SHA256(PrevHash + Time+ Data)。

如能够编译输出类似如下结果，说明实验正确。（检查点1）

```
PrevHash:
Time: 2019-09-04 14:34:36
Data: Genesis Block
Hash:
0e36aec43e3fcc7468edb35b6633971b5dd72b87ce540c8f32dd3d1dc9364ccb
Time using:  15.9588ms
```

成功执行后将关键代码与编译结果截图保存至实验报告，并与实验代码一起提交至本系统。

实验2：实现一条链

有了区块，下面将实现区块链。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。

为了实现这样一件事情，首先要新创建一个blockchain.go文件，在其中定义blockchain类，这里我们使用 array 来存储有序的区块：

```
1. type Blockchain struct {
2.     blocks []*Block
3. }
```

可以看到很简单区块链就是由一个block数组组成的。那么对于一条区块链来说，首先需要能够添加区块，因此要实现添加区块的方法：

```
1. func (bc *Blockchain) AddBlock(data string) {...}
```

这里存在一个问题，添加区块的前提是要有一个已有的块，但是在初始状态下链时空的没有块。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，通常叫做创世区块（genesis block）。因此需要实现一个方法来创建创世区块：

```
1. func GenesisBlock() *Block {...}
```

最后，实现构造函数来生成一个有创世区块的区块链：

```
1. func NewBlockchain() *Blockchain {
2. return &Blockchain{[]*Block{GenesisBlock()}}
3. }
```

练习2：将“exp2.2”文件夹下的blockchain.go文件复制至/home/coder/project/blockchain_demo目录下(使用copy命令,使用格式为 copy <src path> <dest path>)，并补全以下代码：

添加区块函数Blockchain.NewBlock()

创世区块生成函数GenesisBlock()

修改main()函数为以下代码：

```
1. func main() {
2.     bc := NewBlockchain()
3.
```

```
    bc.AddBlock("Send 1 BTC to Ivan")
    bc.AddBlock("Send 2 more BTC to Ivan")

    for _, block := range bc.blocks {
        fmt.Printf("PrevHash: %x\n", block.PrevBlockHash)
        fmt.Printf("Data: %s\n", block.Data)
            fmt.Printf("Hash: %x\n", block.Hash)
            fmt.Println()
        }
    }
```

（检查点2）

成功执行后将关键代码与编译结果截图保存至实验报告，并与实验代码一起提交至本系统。

实验3：添加工作量证明模块

我们创建了一个非常简单的区块链原型：它仅仅是一个数组构成的一系列区块，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。在我们的区块链中，加入新的块非常简单，也很快，但是在真实的区块链中，加入新的块需要很多工作：你必须要经过十分繁重的计算（即工作量证明PoW，或通俗来说：挖矿），来获得添加一个新块的权力。

这就是区块链和比特币的其中一个核心：要想加入新的区块，必须先完成一些非常困难的工作。本节实验将实现这样一个事情。

在工作量证明中，有一个全局的难度值difficulty，用来限制平均挖矿时间。在本例程中，difficulty比特数，例如当difficulty=20时，那么要求矿工需要不断运行工作量证明算法，直到找到一个输入，使该算法输出的hash值的前20比特，也就是16进制的前5位必须全为0。

```
1. const difficulty = 20
```

因此为了实现这样一个功能，首先需要构建一个ProofOfWork类：

```
1. type ProofOfWork struct {
2.     block  *Block
3.     target *big.Int
4. }
5.
6. func NewProofOfWork(b *Block) *ProofOfWork {
7.     targetHash := big.NewInt(1)
8.     targetHash.Lsh(target, uint(256-difficulty))
9.
10.          return &ProofOfWork{b, targetHash}
11.      }
```

在此前区块的哈希是通过Hash = SHA256(PrevBlockHash + Timestamp + Data)这个算式计算的，但是根据工作量证明算法的要求，需要获得满足特定条件的哈希值（本实验中为哈希值前面有多少个0）。在比特币中，使用的是Hashcash算法，具体可分为以下步骤：

1. 选择部分公开数据（比特币中选择的是区块头；Hashcash最初被设计出来防垃圾邮件时，选择的是邮箱地址）
2. 在该公开数据后连接一个nonce，nonce的值为一计数器从0开始不断计数。
3. 将 data+nonce 用hash函数进行计算。
4. 检查hash值是否在给定范围内：

   如果在范围内，证明结束；

如果超出范围，增加nonce值，重复步骤 3-4

它要求证明者不断计算，直至得到某个nonce值，使得最终的hash运算结果落在该hash函数的值域的一小块范围内，是一个暴力算法。当这个范围划定的越小，难度就越高，因此需要更高的成本和更多的算力。

回到程序中，首先需要准备用来哈希的数据：

```
1. func (pow *ProofOfWork) dataPreprocess(nonce int) []byte {
2.     data := bytes.Join(
3.         [][]byte{
4.             pow.block.PrevBlockHash,
5.             pow.block.Data,
6.             IntToHex(pow.block.Timestamp),
7.             IntToHex(int64(targetBits)),
8.             IntToHex(int64(nonce)),
9.         },
10.                []byte{},
11.          )
12.
13.          return data
14.      }
```

这个部分比较直观：只需要将区块的一些数据与nonce进行合并形成一个大的byte数组。这里的nonce，就是上面 Hashcash 所提到的计数器，是一个密码学术语。

完成准备工作后，下面需要实现PoW算法的核心：

```
1. func (pow *ProofOfWork) Mine() (int, []byte) {
2.     var hashInt big.Int  //hash 的整形表示
3.     var hash [32]byte
4.     nonce := 0          //计数器
5.
6.     fmt.Printf("Mining the block containing \"%s\"\n", pow.block.D
ata)
7.     /*
8.     实现 Hashcash 算法：对 nonce 从 0 开始进行遍历，计算每一次哈希是否满足
条件
9.     可能会用到的包及函数：big.Int.Cmp(),big.Int.SetBytes()
10.          */
11.          fmt.Printf("\r%x", hash)
12.          fmt.Print("\n\n")
13.
14.          return nonce, hash[:]
15.      }
16.
17.      func (pow *ProofOfWork) Validate() bool {
18.          var hashInt big.Int
19.          bool isValid
20.
21.          //对给定哈希进行验证是否满足条件
22.
23.          return isValid
24.      }
```

其中Mine()函数用以进行PoW算法，Validate()函数用以进行对区块中的哈希值进行验证。

最后需要对Block类进行修改，将Nonce添加至Block类的结构中，并修改SetHash()函数使其调用ProofOfWork算法获得哈希值。

练习3：从ex2.3文件夹里面复制proofofWork.go和utils.go至/home/coder/project/blockchain_demo/目录下，并补全以下代码：

ProofOfWork.Mine()函数

ProofOfWork.Validate()函数

修改Block类，使其哈希计算方法变为工作量证明算法在main函数中添加对区块哈希的PoW验证

回答问题：工作量证明中的difficulty值的大小会怎样影响PoW计算时间？

参考输出：

roofOfWork.Mine()函数

ProofOfWork.Validate()函数

```
Mining the block containing "Genesis Block"
000000fa153c2965b304041aab15b2ef59770d1064f68171a3afc55914437a63

Mining the block containing "Send 1 BTC to Ivan"
000000a99d1cc5de9f1b2a11ffce9dcca55216cc90bff76d9b118d96c27282c1

Mining the block containing "Send 2 more BTC to Ivan"
0000008bb704912ab654bbc0fb1be0499d4029e2a900b0efa7e26e3549f52ec0

PrevHash:
Data: Genesis Block
Hash: 000000fa153c2965b304041aab15b2ef59770d1064f68171a3afc55914437a63
PoW: true

PrevHash: 000000fa153c2965b304041aab15b2ef59770d1064f68171a3afc55914437a63
Data: Send 1 BTC to Ivan
Hash: 000000a99d1cc5de9f1b2a11ffce9dcca55216cc90bff76d9b118d96c27282c1
PoW: true

PrevHash: 000000a99d1cc5de9f1b2a11ffce9dcca55216cc90bff76d9b118d96c27282c1
Data: Send 2 more BTC to Ivan
Hash: 0000008bb704912ab654bbc0fb1be0499d4029e2a900b0efa7e26e3549f52ec0
PoW: true

Time using:  18.7657653s
```

（检查点3）

成功执行后将编译结果与补全的代码存至工作报告。

实验4：阅读代码：添加数据库

之前的代码中，使用Go语言的结构体变量来存储所有的区块信息，当程序执行时，这些变量都被临时保 存在内存中，这面临两个问题，首先是当程序运行结束时，内存被释放，目前保存的所有信息都将清空；其次是，真正的区块链其数据量是GB计算且不断增长的，不可能全部存在内存中。因此我们需要考 虑将区块链的数据持久化存储，而程序只提供写入和读取的功能。我们选用的是BoltDB这个纯键值数据库，主要优点是其基于Go语言实现，且是nosql型。

存储结构

首先需要明确数据库的结构。在 Bitcoin Core 中，是使用两个 “bucket” 来存储数据：

1. 第一个 bucket 是 **blocks**，它用来存储每个区块的元数据
2. 另一个 bucket 是 **chainstate**，用于存储状态信息，如未花费的交易输出等

其中，本实验仅用到了blocks这个bucket：

根据https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_%28ch_2%29:_Data_Storage **blocks** 中的键值对为：

key	value
'b' + 32-byte block hash	block index record
'f' + 4-byte file number	file information record
'l' -> 4-byte file number	the last block file number used
'R' -> 1-byte boolean	whether we're in the process of reindexing.
'F' + 1-byte flag name length + flag name string	1 byte boolean: various flags that can be on or off
't' + 32-byte transaction hash	transaction index record

而由于本实验实现的区块链功能较为基础，只用到了block hash：32字节 -> block，l -> 链顶区块的 hash这两对键值对。

序列化与反序列化

区块链信息在代码中是以结构体的形式存在的，但是存储到数据库则只能用[]byte形式。所以在存储和读取的过程中，序列化和反序列化是必不可少的。

在block类下添加以下代码：

```
func (b *Block) Serialize() []byte {
    var result bytes.Buffer
    encoder := gob.NewEncoder(&result)

    err := encoder.Encode(b)

    return result.Bytes()
}

  func DeserializeBlock(d []byte) *Block {
      var block Block

      decoder := gob.NewDecoder(bytes.NewReader(d))
      err := decoder.Decode(&block)

      return &block
  }
```

存入数据库

之前实现的NewBlockchain在调用时，会创建一个新的Blockchain实例，并执行GenesisBlock方法创建创世区块。而在加入数据库后，将会增加读取数据库的操作：

1. 打开目标数据库文件
2. 检查是否存在一个区块链
3. 如果存在：
    1. 对其创建Blockchain 实例
    2. 将 Blockchain 中的tip设置为从数据库key l读取到的最后一个区块hash
4. 如果不存在：
    1. 创建创世区块
    2. 存储至数据库
    3. 把key l对应的value设为创世区块的hash
    4. 将上述区块链创建 Blockchain 实例，设置tip为创世区块的hash

因此我们需要对Blockchain中的NewBlockchain和NewBlock函数进行大幅修改，新的代码可以在4.5/blockchain.go中找到。

检查区块链

很显然将区块链存入数据库中之后紧接着面临的问题即是如何查看链上的区块信息。接下来我们将通过BoltDB对bucket中存储的数据按key进行迭代，通常情况下区块链的数据大至数GB，因此不可能一次全部读取进内存中，我们将实现一个迭代器BlockchainIterator，来依次读取它们。通过Blockchain的Iterator()方法可以创建一个迭代器，迭代器会连接到数据库db上，并将当前迭代到的区块的Hash值作为标记记录：

```
type BlockchainIterator struct {
    currentHash []byte
    db          *bolt.DB
```

```
}

func (bc *Blockchain) Iterator() *BlockchainIterator {
    bci := &BlockchainIterator{bc.tip, bc.db}
    return bci
}
```

BlockchainIterator 会不断执行Next()，返回下一个区块：

```
func (i *BlockchainIterator) Next() *Block {
    var block *Block

    err := i.db.View(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        encodedBlock := b.Get(i.currentHash)
        block = DeserializeBlock(encodedBlock)

        return nil
          })

          i.currentHash = block.PrevBlockHash

          return block
      }
```

该部分代码可以在4.5/blockchain_interator.go中找到。

最后，需要重新修改main函数：

```
func main() {
    t := time.Now()

    bc := NewBlockchain()

    bc.AddBlock("Send 1 BTC to Ivan")
    bc.AddBlock("Send 2 more BTC to Ivan")

    bci := bc.Iterator()
          for {
              block := bci.Next()

              fmt.Printf("Prev. hash: %x\n", block.PrevBlockHash)

              fmt.Printf("Data: %s\n", block.Data)
              fmt.Printf("Hash: %x\n", block.Hash)
              pow := NewProofOfWork(block)
              fmt.Printf("PoW: %s\n", strconv.FormatBool(pow.Valid
ate()))
              fmt.Println()

              if len(block.PrevBlockHash) == 0 {
                  break
              }
          }

          fmt.Println("Time using: ", time.Since(t))
      }
```

练习4：从”附件/4.5“文件夹下将里面文件复制至工作目录，并将github.com文件夹复制至$GOPATH/src/目录下。将程序调通后，阅读代码回答以下问题：

- 为什么需要在block类中添加Serialize()和DeserializeBlock()两个函数？他们主要做了什么？描述一下NewBlockchain()和NewBlock()的执行逻辑。
- Blockchain类中的tip变量是做什么用的？
- 迭代器Interator是如何工作使得我们能够从数据库中遍历出区块信息的？

本节无代码部分，课上不检查，请直接写进实验报告。

五、拓展实验：添加命令行接口

本实验为拓展实验，不强制要求完成，完成的同学可以获得额外加分。

目前我们只是在 main 函数中简单执行了 NewBlockchain 和 bc.NewBlock 。很明显这并不是像一个完整的程序，我们希望的是能够拥有这样的效果：

```
$ ./blockchain_go listblocks
No existing blockchain found. Creating a new one...
Mining the block containing "Genesis Block"
000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b

Prev. hash:
Data: Genesis Block
Hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
PoW: true

$ ./blockchain_go newblock -data "Send 1 BTC to Ivan"
Mining the block containing "Send 1 BTC to Ivan"
000000d7b0c76e1001cdc1fc866b95a481d23f3027d86901eaeb77ae6d002b13

Success!

$ ./blockchain_go listblocks
Prev. hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
Data: Send 1 BTC to Ivan
Hash: 000000d7b0c76e1001cdc1fc866b95a481d23f3027d86901eaeb77ae6d002b13
PoW: true

Prev. hash:
Data: Genesis Block
Hash: 000000edc4a82659cebf087adee1ea353bd57fcd59927662cd5ff1c4f618109b
PoW: true
```

有意向的同学可以自由发挥。

可能用到的工具：命令行参数相关（os.Args, 标准库flag中的NewFlagSet等）

六、参考文献

[1] https://github.com/Jeiwan/blockchain_go，访问时间：2021.10.19

[2] https://code.visualstudio.com/，访问时间：2021.10.19

[3] https://en.bitcoin.it/wiki/Block_hashing_algorithm，访问时间：2021.10.19

[4] https://bitcoin.org/bitcoin.pdf，访问时间：2021.10.19

[5] https://golang.org/pkg/encoding/gob/ ，访问时间：2021.10.19

[6]https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_%28ch_2%29:_Data_Storage，访问时间：2021.10.19

[7] https://golang.org/dl/，访问时间：2021.10.19