区块链 实验二

PB20000096 潘廷岳

实验目的

- 实现区块链上的POW证明算法
    - proofofwork.go
    - Run() pow计算部分
    - Validate() pow结果的验证工作
    - 完成比较targetBits的修改带来的计算次数上的变化情况，从5-15的变化
- 理解区块链上的难度调整的作用

实验介绍 & 原理

之前的实验中，我们简单构建了一个区块链的数据结构以及对应持久化操作的数据库。这样，我们就可以简单地进行区块的写入和具体的操作了。同时，我们也了解到了一些区块链的基本特点：每个区块都是连接在上一个区块上的。同时，我们目前的区块链有这样的特点：添加区块是相对比较容易的。但是，对于真正的区块链我们都知道：添加一个区块需要所有节点达成共识，所以是一个相当复杂的工作。在本次实验中，我们就要进行对于共识部分的补充，来保证区块链的安全性和一致性。

实验涉及知识板块：

- 区块链共识协议
- 工作量证明（POW）
- 区块链哈希

区块链共识协议

区块链共识的关键思想就是为了结点通过一些复杂的计算操作来获取写入区块的权利。这样的复杂工作量是为了保证区块链的安全性和一致性。如果是对应比特币、以太坊等公有链的架构，对于写入的区块会得到相应的奖励（俗称挖矿）。

根据比特币的白皮书,共识部分是为了决定谁可以写入区块的问题，区块链的决定是通过最长链来表示的，这个是因为最长的区块对应有最大的工作量投入在其中。相应地，为了保证区块链的出块保持在一个相对比较稳定的值，对应地，对进行区块链共识难度的调整来保证出块速度大致保持一致。对应比特币来说，写入区块的节点还对应会获得奖励。

工作量证明（POW）

工作量的证明机制，简单来说就是通过提交一个容易检测，但是难以计算的结果，来证明节点做过一定量的工作。对应的算法需要有两个特点：计算是一件复杂的事情，但是证明结果的正确与否是相对简单的。对应地行为，可以类比生活中考驾照、获取毕业证等。

工作量证明由Cynthia Dwork 和Moni Naor 1993年在学术论文中首次提出。而工作量证明（POW）这个名词，则是在1999年 Markus Jakobsson 和Ari Juels的文章中才被真正提出。在发明之初，POW主要是为了抵抗邮件的拒绝服务攻击和垃圾邮件网关滥用，用来进行垃圾邮件的过滤使用。POW要求发起者进行一定量的运算，消耗计算机一定的时间。

区块链哈希

SHA256算法的哈希函数，具有区块链上哈希函数的一些基本特点：

1. 原始数据不能直接通过哈希值来还原，哈希值是没法解密的。
2. 特定数据有唯一确定的哈希值，并且这个哈希值很难出现两个输入对应相同哈希输出的情况。
3. 修改输入数据一比特的数据，会导致结果完全不同。
4. 没有除了穷举以外的办法来确定哈希值的范围。

比特币采用了哈希现金(hashcash)的工作量证明机制，也就是之前说过的用在垃圾邮件过滤时使用的方法，对应流程如下：

1. 本次实验需要首先构建当前区块头，区块头包含**上一个区块哈希值(32位)，当前区块数据对应哈希（32位，即区块数据的merkle根），时间戳，区块难度，计数器(nonce)。**通过计算当前区块头的哈希值来求解难题。
2. 添加计数器，作为随机数。计算器从0开始基础，每个回合**+1**
3. 对于上述的数据来进行一个哈希的操作。
4. 判断结果是否满足计算的条件：
    1. 如果符合，则得到了满足结果。
    2. 如果没有符合，从2开始重新直接2、3、4步骤。

从中也可以看出，这是一个"非常暴力"的算法。这也是为什么这个算法需要指数级的时间。

实验内容

```
targetBits = 5 //难度值
```

在本次实验中，我们选用了一个固定的难度值来进行计算。5的难度值意味着我们需要获取一个**1<<(256-targetBits)小的数，这里为1<<(256-5)。在本实验中，为了模拟出块的难度值调整，每出一个块，对应targetBits难度增加1。**（在代码测试时，可以修改Block.NewBlock，来保持困难度不改变）

```
type ProofOfWork struct {
    block  *Block
}

type Block struct {
    Timestamp     int64
    Data          [][]byte
    PrevBlockHash []byte
    Hash          []byte
    Bits          uint  //记录区块的难度值
    Nonce         int
}
```

`ProofOfWork`是一个区块的指针,对应我们在区块中记录加上了**Bits**，记录当前区块计算的难度。 为了进行区块上的操作，我们需要使用`big.Int`来得到一个大数操作，对应难度就是之前提到的`1<<(256-targetBits)`。

在这个实验中，我们还需要注意到的是`第一个区块对应的hash`是一个为空的值。在这个实验中，可以使用`"crypto/sha256`来进行哈希函数的操作。对于*int*转*byte*的操作可以使用`utils.go`里的`IntToHex`函数来实现

实验平台

- Windows10
- vscode
    - go version: go1.17.7 windows/amd64

代码展示

pow.Validate()

- 按照pow共识算法，对pow结果的合法性进行验证
- 区块头依照以下顺序构建：
    - 上一个区块哈希值(32位)，
    - 当前区块数据对应哈希（32位，即区块数据的merkle根），
    - 时间戳，
    - 区块难度，
    - 计数器(nonce)
- 使用 GetBase4Nonce() 进行区块头计算
- 代码如下：

```
//pow.Validate()

func (pow *ProofOfWork) Validate() bool {
  //获取区块难度
    Get_dif := IntToHex(int64(pow.block.Bits))

  //获取区块数据hash
    BlockData := bytes.Join(pow.block.Data, nil)
    BlockDataHash := mySha256(BlockData)

    var intHash big.Int
    var hash [32]byte

  //计算区块难度对应阈值
    intTarget := big.NewInt(1)
    intTarget.Lsh(intTarget, uint(256-pow.block.Bits))

  //获取区块头hash
    data := pow.block.GetBase4Nonce(pow.block.Nonce, BlockDataHash[:], Get_dif)
    hash = mySha256(data)
    intHash.SetBytes(hash[:])

  //比较区块头hash结果与阈值的大小，返回合法判定
```

```
    if intHash.Cmp(intTarget) == -1 {
        pow.block.Hash = hash[:]
        return true
    } else {
        return false
    }
}
```

```
//GetBase4Nonce()

func (block *Block) GetBase4Nonce(nonce int, BlockDataHash, dif []byte) []byte {

  //按要求计算并返回区块头
    data := bytes.Join([][]byte{
        block.PrevBlockHash,
        BlockDataHash[:],
        IntToHex(block.Timestamp),
        dif,
        IntToHex(int64(nonce)),
    },
        []byte{},
    )
    return data
}
```

pow.Run()

- 按照pow共识算法进行工作，进行pow的计算并找到符合要求的解
- 从 0 到 maxNonce 枚举随机数nonce，并调用Validate对合法性进行验证，直到找到合法解。
- 其计算次数应当与区块难度呈正相关
- 代码如下：

```
//pow.Run()

var (
    maxNonce = math.MaxInt64
)

func (pow *ProofOfWork) Run() (int, []byte) {
    nonce := 0

    for nonce < maxNonce {
        pow.block.Nonce = nonce
        if pow.Validate() {
            break
        } else {
```

```
                nonce++
            }
        }
            // 逐个枚举nonce并进行合法性判定

        return nonce, pow.block.Hash
    }
```

实验结果

- 删除 blockchain.db 数据库文件
- 依次交替输入addblock [str] & printchain指令

创世区块 & 区块一

```
PS C:\Users\pty\blockchain-lab\lab2> cd .\template\
PS C:\Users\pty\blockchain-lab\lab2\template> go run .
No existing blockchain found. Creating a new one...
chaincode > addblock aaa
add Success
chaincode > printchain
Prev. hash: 048ef24369c979b82d72663385664536aab7872845cf37896c7639a43b493b10
Data: [aaa]
Hash: 0293c342cce6f11371f5a8e3bca0a31200e57e4ccd5c8bcc85e43862df285a54
TargetBits 6
Nonce 22
PoW: true

Prev. hash:
Data: [Genesis Block]
Hash: 048ef24369c979b82d72663385664536aab7872845cf37896c7639a43b493b10
TargetBits 5
Nonce 5
PoW: true
```

区块二

```
chaincode > addblock bbb
add Success
chaincode > printchain
Prev. hash: 0293c342cce6f11371f5a8e3bca0a31200e57e4ccd5c8bcc85e43862df285a54
Data: [bbb]
Hash: 00c8ff73c4eafad6aafc5229ac7c7d1cf8666e4868e0fb194469263fa94e977c
TargetBits 7
Nonce 40
PoW: true

Prev. hash: 048ef24369c979b82d72663385664536aab7872845cf37896c7639a43b493b10
Data: [aaa]
Hash: 0293c342cce6f11371f5a8e3bca0a31200e57e4ccd5c8bcc85e43862df285a54
TargetBits 6
Nonce 22
PoW: true

Prev. hash:
Data: [Genesis Block]
Hash: 048ef24369c979b82d72663385664536aab7872845cf37896c7639a43b493b10
TargetBits 5
Nonce 5
PoW: true
```

区块三 & close

```
chaincode > addblock ccc
add Success
chaincode > printchain
Prev. hash: 00c8ff73c4eafad6aafc5229ac7c7d1cf8666e4868e0fb194469263fa94e977c
Data: [ccc]
Hash: 0029ce8f6a0b636f316f272918f156b9b8e8fe596cccdfc7195bf5141d53fbaf
TargetBits 8
Nonce 60
PoW: true

Prev. hash: 0293c342cce6f11371f5a8e3bca0a31200e57e4ccd5c8bcc85e43862df285a54
Data: [bbb]
Hash: 00c8ff73c4eafad6aafc5229ac7c7d1cf8666e4868e0fb194469263fa94e977c
TargetBits 7
Nonce 40
PoW: true

Prev. hash: 048ef24369c979b82d72663385664536aab7872845cf37896c7639a43b493b10
Data: [aaa]
Hash: 0293c342cce6f11371f5a8e3bca0a31200e57e4ccd5c8bcc85e43862df285a54
TargetBits 6
Nonce 22
PoW: true

Prev. hash:
Data: [Genesis Block]
Hash: 048ef24369c979b82d72663385664536aab7872845cf37896c7639a43b493b10
TargetBits 5
Nonce 5
PoW: true

chaincode > close
close.
```

- 可以初步看到，随着区块长度增大，区块难度及Nonce的值也随之增大。

观察TargetBits修改带来的次数改变

- 为了更好观察区块难度对计算次数Nonce的影响，按照如下思路设计验证程序：
  - TargetBits从 5~15 变化，每个不同的 TargetBits 生成数据完全一样的M个区块，并对这M个区块的 Nonce 取平均值作为该难度下计算次数的值。
- 代码段如下所示：

①命令行操作设定

- chaincode > TestNonce [TestSum_M]
- TestSum_M: 每个难度下计算的总次数

```
//在main.go中插入代码，以支持命令行操作
    {
        Name:    "TestNonce",
        Aliases: []string{"T"},
        Usage:   "TestNonce TestSum_M    - test the influence of TargetBits to
Nonce",
        Action: func(c *cli.Context) error {
            Sum_M := c.Args()
            bc.TestNonce(Sum_M)
            fmt.Println("Couting...")
            return nil
        },
    },
```

```
PS C:\Users\pty\blockchain-lab\lab2\template> go run .
chaincode > template
NAME:
   chaincode template - options for task templates

USAGE:
   chaincode template command [command options] [arguments...]

COMMANDS:
   addblock, a    addblock BLOCK_DATA    - add a block to the blockchain
   printchain, p  printchain
   TestNonce, T   TestNonce TestSum_M    - test the influence of TargetBits to Nonce

OPTIONS:
   --help, -h  show help

chaincode >
```

②bc.TestNonce()

```
func (bc *Blockchain) TestNoce(Sum_M []string) {
    fmt.Println("M = ", Sum_M)
    var Str_M string = Sum_M[0]
```

```
    var test_str []string = make([]string, 2, 4)
    test_str[0] = string("aaa")
    Aver_Nonce := float32(0) //每个难度下区块随机数平均值

    err := bc.db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte(blocksBucket))
        PreHash := b.Get([]byte("l"))

        if M, err := strconv.Atoi(Str_M); err == nil {
            for targetBits = 5; targetBits <= 15; targetBits += 3 {
                Aver_Nonce = 0
                for i := 1; i <= M; i++ {
                    NewBlock := NewBlock(test_str, PreHash)
                    Aver_Nonce += float32(NewBlock.Nonce)
                }
                //创建M个相同TargetBits的块，取Nonce平均

                fmt.Println("targetBits = ", targetBits)
                fmt.Println("Aver_Nonce = ", Aver_Nonce/float32(M))
                fmt.Println("")
            }
            //对每个TargetBits下生成的区块进行Nonce取平均
        }
        return nil
    })
    if err != nil {
        log.Panic(err)
    }

}
```

- **首先，将M设置成 10 （较小）**

```
PS C:\Users\pty\blockchain-lab\lab2\template> go run .
chaincode > TestNonce 10
M =  [10]
targetBits =  5
Aver_Nonce =  33

targetBits =  6
Aver_Nonce =  88

targetBits =  7
Aver_Nonce =  37

targetBits =  8
Aver_Nonce =  76

targetBits =  9
Aver_Nonce =  4

targetBits =  10
Aver_Nonce =  114

targetBits =  11
Aver_Nonce =  2879

targetBits =  12
Aver_Nonce =  11926

targetBits =  13
Aver_Nonce =  2512

targetBits =  14
Aver_Nonce =  18723

targetBits =  15
Aver_Nonce =  36307

Couting...
chaincode >
```

发现Nonce的平均并没有呈现递增趋势；推测是 M 过小

- **将 M 设置成500，再次运行**

```
chaincode > TestNonce 500
M =  [500]
targetBits =  5
Aver_Nonce =  21

targetBits =  6
Aver_Nonce =  23

targetBits =  7
Aver_Nonce =  360

targetBits =  8
Aver_Nonce =  258.084

targetBits =  9
Aver_Nonce =  101

targetBits =  10
Aver_Nonce =  2511.44

targetBits =  11
Aver_Nonce =  1654.598

targetBits =  12
Aver_Nonce =  1826.948

targetBits =  13
Aver_Nonce =  2405.724

targetBits =  14
Aver_Nonce =  2229.12

targetBits =  15
Aver_Nonce =  14787.184
```

发现结果大致出现了分级，但仍有部分Nonce出现异常

- **再次调大 M ,设置成10000**

```
PS C:\Users\pty\blockchain-lab\lab2\template> go run .
chaincode > TestNonce 10000
M =  [10000]
targetBits =  5
Aver_Nonce =  12.7201

targetBits =  6
Aver_Nonce =  77.0692

targetBits =  7
Aver_Nonce =  167.6111

targetBits =  8
Aver_Nonce =  96.0072

targetBits =  9
Aver_Nonce =  360.224

targetBits =  10
Aver_Nonce =  525.7401
```

发现结果基本符合递增，唯有targetBits = 8 略有异常

- **最终尝试跨targeiBits进行观察，如图所示结果**

```
PS C:\Users\pty\blockchain-lab\lab2> cd .\template\
PS C:\Users\pty\blockchain-lab\lab2\template> go run .
chaincode > TestNonce 50
M =  [50]
targetBits =  5
Aver_Nonce =  43

targetBits =  8
Aver_Nonce =  720

targetBits =  11
Aver_Nonce =  129

targetBits =  14
Aver_Nonce =  9639.22

Couting...
chaincode > TestNonce 10000
M =  [10000]
targetBits =  5
Aver_Nonce =  25.0441

targetBits =  8
Aver_Nonce =  150.174

targetBits =  11
Aver_Nonce =  470.9894

targetBits =  14
Aver_Nonce =  1949.0363

Couting...
chaincode >
```

可以发现，当M = 10000时，四个级别显示出了较好的分级结果，基本符合预测。

测试结果分析

由于Sha256的强大随机性，导致hash结果在被hash数差 1 位的情况下也会产生巨大偏差。故targetBits相差 1 时，在测试数据数远小于 2^256 时，Nonce的均值大小排序有可能出现倒置的情况。 因此，若想看出Nonce随着targetBits的变化规律，应当满足：

- 适当调大targetBits变化率
- 增大测试数据组数（至少在10000以上）

同时在测试过程中，笔者发现，当TargetBits超过20后，求解难度明显加大。故通过改变TargetBits的值，能够进行难度调整，以保证区块链的出块保持在一个相对比较稳定的值，从而保证交易速度的稳定性。

实验总结

- 对区块链共识机制有了更深入的理解

- 对pow共识算法的流程及实现较为熟悉
- 对go语言机制有了更进一步理解