## POW算法实验指导书

## 实验课时

4课时

## 实验内容

理解POW算法的工作原理和流程，能够使用编程语言实现POW算法

## 实验环境

GO开发环境：go1.18.1

IDE开发工具：vscode

操作系统：windows 11

## 算法描述

POW（Proof of Work，工作量证明）是一种用于比特币和其他加密货币的共识算法。它的核心思想是通过利用HASH运算的复杂度进行CPU运算实现工作量确定，当然也可以利用卷积求导、大质数分解这些复杂的运算来达到证明的目的。通过这种方式，可以保证节点没有欺骗行为，从而确保整个网络的安全性。

在POW算法中，节点需要进行一定量的计算，才能获得一个区块的记账权。这个计算过程需要大量的计算资源，因此一般只能由拥有专业设备的矿工来完成。每次计算都会依据区块头中的数据生成一组哈希值，如果这组哈希值符合一定规则（即满足指定的复杂度要求），那么这个节点就可以获得一个新区块的记账权。

## 实验过程

POW算法的流程如下：

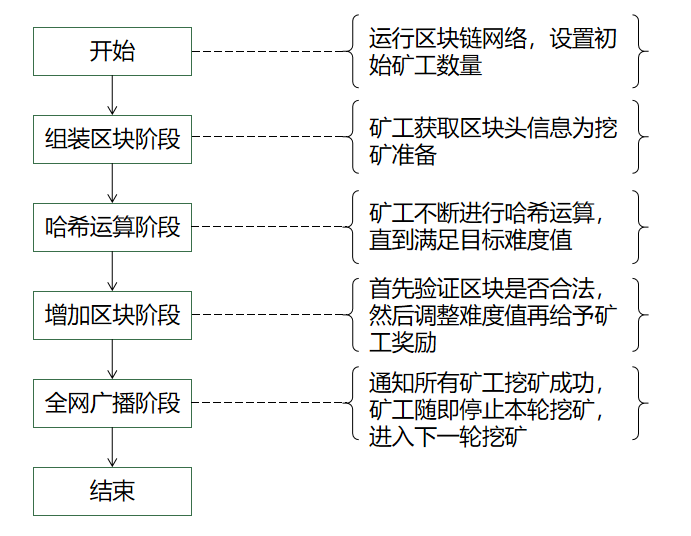


图1：Pow算法流程图

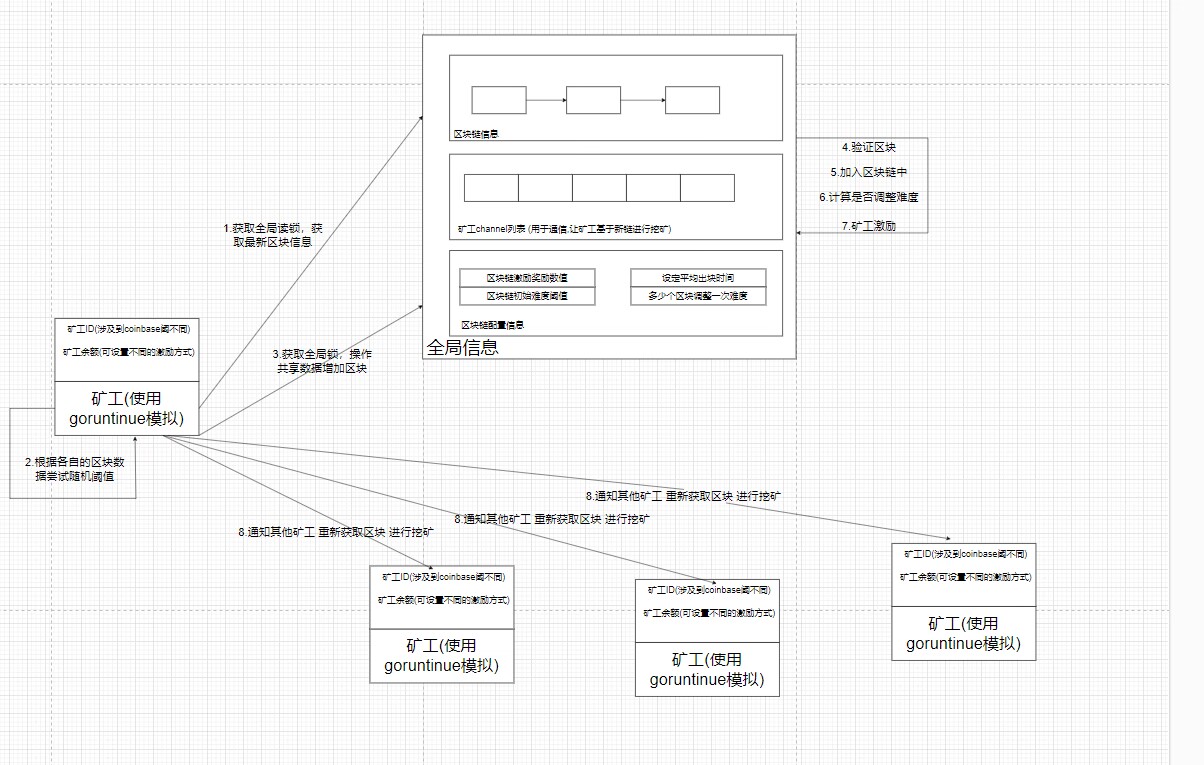


图2：矿工结构图

数据结构如下：

var (

    //Nonce循环上限

    maxNonce = math.MaxInt64

)

// Block 自定义区块结构

type Block struct {

    \*BlockWithoutProof

    Proof

}

//区块的证明信息

type Proof struct {

    //实际的时间戳 由于比特币在挖矿中不光要变动nonce值 也要变动时间戳

    ActualTimestamp int64 `json:"actualTimestamp"`

    //随机值

    Nonce int64 `json:"nonce"`

    //当前块哈希

    hash []byte

    // 转换成十六进制可读

    HashHex string `json:"hashHex"`

}

//不带证明信息的区块

type BlockWithoutProof struct {

    // 挖矿成功矿工

    CoinBase int64 `json:"coinBase"`

    //时间戳

    timestamp int64

    //数据域

    data []byte

    //前一块hash

    prevBlockHash []byte

    //前一块hash

    PrevBlockHashHex string `json:"prevBlockHashHex"`

    //目标阈值

    TargetBit float64 `json:"targetBit"`

}

//矿工结构

type Miner struct {

    //矿工ID

    Id int64 `json:"id"`

    //矿工账户余额

    Balance uint `json:"balance"`

    //当前矿工正在挖的区块

    blockchain \*Blockchain

    // 用于通知 当接收到新区块的时候  不应该从原有的链继续往后挖

    waitForSignal chan interface{} `json:"-"`

}

// Blockchain 区块链数据，因为是模拟，所以我们假设所有节点共享一条区块链数据，且所有节点共享所有矿工信息

type Blockchain struct {

    // 区块链配置信息

    config BlockchainConfig

    // 当前难度

    currentDifficulty float64

    // 区块列表

    blocks []Block

    // 矿工列表

    miners []Miner

    // 互斥锁 防止发生读写异常

    mutex \*sync.RWMutex

}

//区块链配置信息

type BlockchainConfig struct {

    MinerCount                  int     // 矿工个数

    OutBlockTime                uint    // 平均出块时间

    InitialDifficulty           float64 // 初始难度

    ModifyDifficultyBlockNumber uint    // 每多少个区块修改一次难度阈值

    BookkeepingIncentives       uint    // 记账奖励

}

type BlockchainInfo struct {

    Blocks []\*Block `json:"blocks"` // 区块列表

    Miners []\*Miner `json:"miners"` // 矿工列表

}

#### 初始化阶段

此部分进行了矿工的初始化，以及创世区块的生成

func main() {

    var count int

    fmt.Printf("请输入初始矿工数量：")

    fmt.Scanf("%d", &count)

    time.Sleep(10000)

    fmt.Printf("开始挖矿")

    //新建区块链网络

    work := pow.NewBlockChainNetWork(pow.BlockchainConfig{

        //矿工数量

        MinerCount: count,

        //平均出块时间

        OutBlockTime: 10,

        //初始难道值

        InitialDifficulty: 20,

        //每多少个区块修改一次难度值

        ModifyDifficultyBlockNumber: 10,

        //每次记账奖励

        BookkeepingIncentives: 20,

    })

    //运行区块链网络

    work.RunBlockChainNetWork()

    //启动web服务

    web.RunRouter(work)

}

//新建一个区块链网络

func NewBlockChainNetWork(blockchainConfig BlockchainConfig) \*Blockchain {

    b := &Blockchain{

        blocks:            nil,

        miners:            nil,

        config:            blockchainConfig,

        mutex:             &sync.RWMutex{},

        currentDifficulty: blockchainConfig.InitialDifficulty,

    }

    b.blocks = append(b.blocks, \*GenerateGenesisBlock([]byte("")))

    //新建矿工

    for i := 0; i < blockchainConfig.MinerCount; i++ {

        miner := Miner{

            Id:            int64(i),

            Balance:       0,

            blockchain:    b,

            waitForSignal: make(chan interface{}, 1),

        }

        b.miners = append(b.miners, miner)

    }

    return b

}

//生成创世区块

func GenerateGenesisBlock(data []byte) \*Block {

    b := &Block{BlockWithoutProof: &BlockWithoutProof{}}

    b.ActualTimestamp = time.Now().Unix()

    b.data = data

    return b

}

// 运行区块链网络

func (b \*Blockchain) RunBlockChainNetWork() {

    for \_, m := range b.miners {

        go m.run()

    }

}

#### 区块组装阶段

所有矿工同时获取了区块数据并将其组装为当前区块头部分，接下来哈希运算准备

//挖矿逻辑

func (m Miner) run() {

    count := 0

    //死循环

    for ; ; count++ {

        //根据全局信息组装去了

        blockWithoutProof := m.blockchain.assembleNewBlock(m.Id, []byte(fmt.Sprintf("模拟区块数据:%d:%d", m.Id, count)))

        block, finish := blockWithoutProof.Mine(m.waitForSignal)

        if !finish {

            //如果不满足条件则计数器增加继续计算hash并判断

            continue

        } else {

            //如果条件满足则增加区块

            m.blockchain.AddBlock(block, m.waitForSignal)

        }

    }

}

// 根据全局信息组装区块

func (b \*Blockchain) assembleNewBlock(coinBase int64, data []byte) BlockWithoutProof {

    b.mutex.RLock()

    defer b.mutex.RUnlock()

    proof := BlockWithoutProof{

        CoinBase:         coinBase,

        timestamp:        time.Now().Unix(),

        data:             data,

        prevBlockHash:    b.blocks[len(b.blocks)-1].hash,

        TargetBit:        b.currentDifficulty,

        PrevBlockHashHex: b.blocks[len(b.blocks)-1].HashHex,

    }

    return proof

}

#### 哈希运算阶段

若矿工在哈希运算过程中收到其他矿工挖矿成功消息则停止挖矿

// Mine 挖矿函数

func (b \*BlockWithoutProof) Mine(waitForSignal chan interface{}) (\*Block, bool) {

    //target为最终难度值

    target := big.NewInt(1)

    //target为1向左位移256-24（挖矿难度）

    target.Lsh(target, uint(256-b.TargetBit))

    var hashInt big.Int

    var hash [32]byte

    nonce := 0

    for nonce != maxNonce {

        //  判断一下是否别的矿工已经计算出来结果了 模拟 一旦收到其他矿工 的交易，立即停止计算

        select {

        case \_ = <-waitForSignal:

            return nil, false

        default:

            //准备数据整理为哈希

            data := b.prepareData(int64(nonce))

            //计算哈希

            hash = sha256.Sum256(data)

            hashInt.SetBytes(hash[:])

            //按字节比较，hashInt cmp小于0代表找到目标Nonce

            if hashInt.Cmp(target) < 0 {

                block := &Block{

                    BlockWithoutProof: b,

                    Proof: Proof{

                        Nonce:   int64(nonce),

                        hash:    hash[:],

                        HashHex: hex.EncodeToString(hash[:]),

                    },

                }

                return block, true

            } else {

                nonce++

            }

        }

    }

    return nil, false

}

##### 其中涉及的方法：

// 准备数据 整理成待计算哈希

func (block \*BlockWithoutProof) prepareData(nonce int64) []byte {

    data := bytes.Join(

        [][]byte{

            int2Hex(block.CoinBase),

            block.prevBlockHash,

            block.data,

            int2Hex(block.timestamp),

            int2Hex(int64(block.TargetBit)),

            int2Hex(nonce),

        },

        []byte{},

    )

    return data

}

#### 增加区块阶段

矿工打包的区块得到验证，即可加入区块链并且获得奖励

// 增加一个区块到区块链

func (bc \*Blockchain) AddBlock(block \*Block, signal chan interface{}) {

    bc.mutex.Lock()

    defer bc.mutex.Unlock()

    block.ActualTimestamp = time.Now().Unix()

    //验证新区块

    if !bc.verifyNewBlock(block) {

        return

    }

    bc.blocks = append(bc.blocks, \*block)

    //根据挖矿难度调整难度值

    bc.adjustDifficulty()

    //给予挖矿矿工奖励

    bc.bookkeepingRewards(block.CoinBase)

    //通知所有矿工挖矿成功

    bc.notifyMiners(block.CoinBase)

  fmt.Printf(" %s: %d 节点挖出了一个新的区块 %s\n", time.Now(), block.CoinBase, block.HashHex)

}

##### 涉及到的方法：

//验证新区块

func (bc \*Blockchain) verifyNewBlock(block \*Block) bool {

    prevBlock := bc.blocks[len(bc.blocks)-1]

    // 新区块 一定要符合 当前难度值的 要求

    if uint64(block.TargetBit) != uint64(bc.currentDifficulty) {

        return false

    }

    // hash 链一定要符合

    if string(prevBlock.hash) != string(block.prevBlockHash) {

        return false

    }

    // 区块 本身需要符合规范

    if !block.Verify() {

        return false

    }

    return true

}

//根据挖矿的时间调整难度值

func (bc \*Blockchain) adjustDifficulty() {

    if uint(len(bc.blocks))%bc.config.ModifyDifficultyBlockNumber == 0 {

        block := bc.blocks[len(bc.blocks)-1]

        preDiff := bc.currentDifficulty

        actuallyTime := float64(block.ActualTimestamp - bc.blocks[uint(len(bc.blocks))-bc.config.ModifyDifficultyBlockNumber].ActualTimestamp)

        theoryTime := float64(bc.config.OutBlockTime \* bc.config.ModifyDifficultyBlockNumber)

        ratio := theoryTime / actuallyTime

        if ratio > 1.1 {

            ratio = 1.1

        } else if ratio < 0.5 {

            ratio = 0.5

        }

        bc.currentDifficulty = bc.currentDifficulty \* ratio

        fmt.Println("难度阈值改变 preDiff: ", preDiff, "nowDiff", bc.currentDifficulty)

    }

}

//给予挖矿成功的矿工奖励

func (bc \*Blockchain) bookkeepingRewards(coinBase int64) {

    bc.miners[coinBase].Balance += bc.config.BookkeepingIncentives

}

#### 全网广播阶段

成功挖矿的矿工通知其他所有矿工

//通知所有矿工挖矿成功 重置矿工的Block字段

func (bc \*Blockchain) notifyMiners(sponsor int64) {

    for i, miner := range bc.miners {

        if i != int(sponsor) {

            go func(signal chan interface{}) {

                signal <- struct{}{}

            }(miner.waitForSignal)

        }

    }

}

附：本实验采用web服务，若在挖矿过程中需增加矿工则访问localhost:8080/addMiner 增加矿工，亦可访问localhost:8080/getBlockChainInfo 获取到目前为止的打印信息

//运行web服务

//访问localhost:8080/addMiner  可以增加矿工

//访问localhost:8080/getBlockChainInfo  获取到目前为止打印的挖矿信息

func RunRouter(blockchain \*pow.Blockchain) {

    r := gin.Default()

    r.GET("/addMiner", addMiner(blockchain))

    r.GET("/getBlockChainInfo", getBlockChainInfo(blockchain))

    r.Run()

}

//增加矿工

func addMiner(blockchain \*pow.Blockchain) gin.HandlerFunc {

    return func(c \*gin.Context) {

        blockchain.IncreaseMiner()

        c.JSON(200, gin.H{

            "message": "增加成功",

        })

    }

}

//打印挖矿信息

func getBlockChainInfo(blockchain \*pow.Blockchain) gin.HandlerFunc {

    return func(c \*gin.Context) {

        blocks, miners := blockchain.GetBlockInfo()

        c.JSON(200, gin.H{

            "blocks": blocks,

            "miners": miners,

        })

    }

}

//增加矿工

func (bc \*Blockchain) IncreaseMiner() bool {

    bc.mutex.Lock()

    defer bc.mutex.Unlock()

    var miner = Miner{

        Id:            int64(len(bc.miners)),

        Balance:       0,

        blockchain:    bc,

        waitForSignal: make(chan interface{}, 1),

    }

    bc.miners = append(bc.miners, miner)

    go miner.run()

    return true

}

//获取区块信息

func (bc \*Blockchain) GetBlockInfo() ([]Block, []Miner) {

    bc.mutex.RLock()

    defer bc.mutex.RUnlock()

    blocks := make([]Block, len(bc.blocks))

    miners := make([]Miner, len(bc.miners))

    copy(blocks, bc.blocks)

    copy(miners, bc.miners)

    return blocks, miners

}

### 程序执行过程：

1.首先在根目录运行：



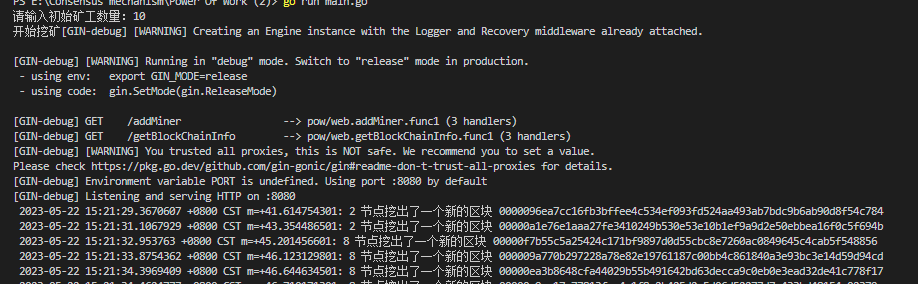
2.随后开始运行程序



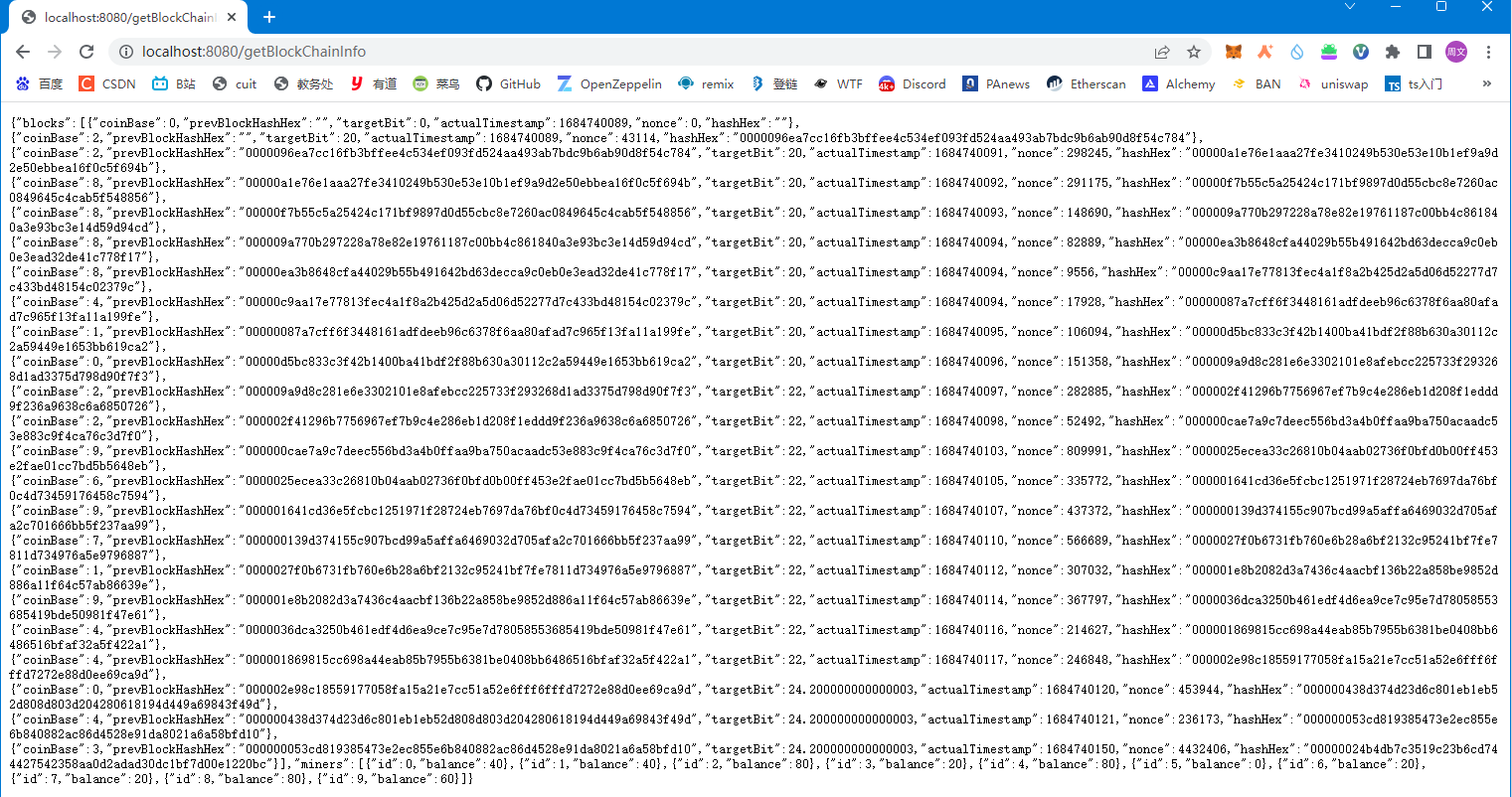
3.需要输入初始矿工数量，后面可以在web中增加矿工数量



4.随后可以看见已经开始挖矿



5.浏览器访问localhost:8080/getBlockChainInfo获取区块链信息

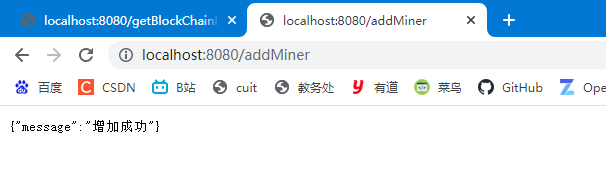


可以看见已经存在id为0-9的矿工节点，以及节点目前获得的挖矿奖励

分析每一条区块信息可以看见挖矿成功的矿工，前一区块的哈希，当前难度值，时间戳，随机数，交易哈希等信息。



6.访问localhost:8080/addMiner 增加矿工





可以看见目前存在id为0-10的矿工节点