**POS算法实验指导书**

**一、实验课时**

2课时

**二、实验内容**

理解 PoS 共识机制的工作原理和流程；

能够实现一个简单的 PoS 共识算法；

**三、实验环境**

本实验需要使用以下工具和环境：

Go 语言开发环境（示例代码的go版本为1.17.13，不建议go版本低于1.17）

IDE工具：vscode / goland

操作系统：windows / mac os

**四、算法描述**

PoS（Proof of Stake）共识机制是一种基于代币持有量来确定区块链网络共识的算法。相比于 PoW（Proof of Work）共识机制，PoS 具有更高的效率和更低的能源消耗。使用Pos机制的区块链共识主要流程如下：

1、初始状态下，网络中的每个节点都需要拥有一定数量的数字资产作为抵押。这些数字资产将被用作随机选择记账节点的依据。

2、当需要生成一个新的区块时，网络会根据参与者的抵押数量来选择出共识节点（也称为验证节点或记账人），并由共识节点来完成区块的验证和打包。

3、其他节点将对新区块进行验证，确保其中的交易合法且不与其他已经存在的交易冲突。

4、共识节点会获得一定的奖励作为激励，而抵押的代币则会被锁定一段时间（称为冻结期），以确保共识节点不会恶意攻击网络。

PoS共识算法实现：

PoS 共识算法中引入了“币龄”的概念，相关定义如下：

币龄（coinAge）是指持币数量（coins）与持币时间 （holdTime）的乘积： coinAge=coins × holdTime

PoS共识算法给定一个全网统一的难度值 D，以及新打包进区块的元数据

tradeData，寻找满足条件的记数器 timeCounter， 使得：

SHA256(SHA256(tradeData|timeCounter))<D×coinAge，找到则挖矿成功。

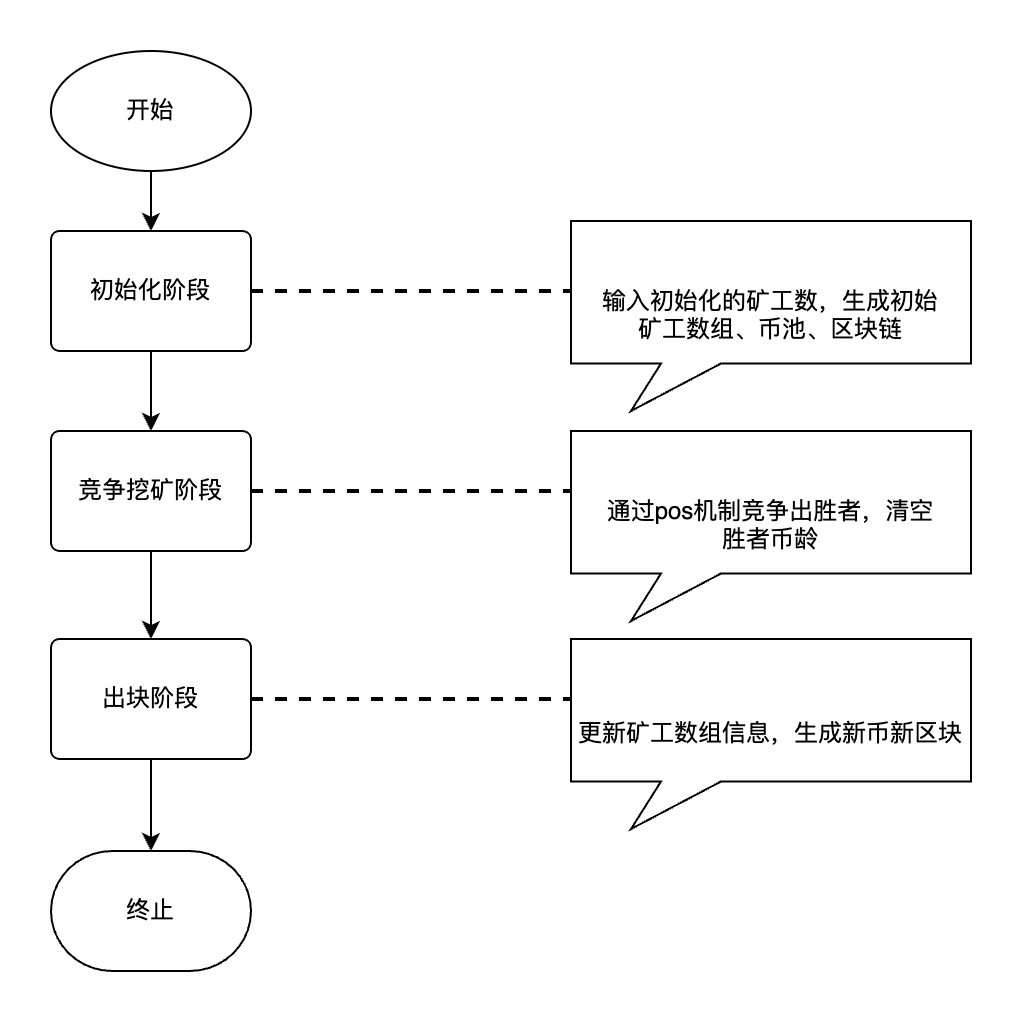
PoS 共识算法的记账规则与 Pow 共识算法基本相似，但 PoS共识算法不需要矿

工枚举所有的随机数 Nonce，而是在 1s内只允许一次哈希值，大大减轻了计算工作

量，从而减缓了算力竞争带来的资源消耗。

1. **实验过程**

**实验流程：**

****

**图1:pos算法流程图**

**1、初始化**

//全局变量

const (

dif = 2

INT64\_MAX = math.MaxInt64

MaxProbably = 255

MinProbably = 235

)

// 创建一种名为Miner的结构体，包含miner的地址和持币数量，以及记录的币龄

type Miner struct {

addr []byte

num int64

coinAge int64

}

// 初始化Miner的函数，默认addr为用sha256方法对字符串miner和现在的时间拼接后的字符串处理后的结果,num为0，coinAge为0  
func createMiner() \*Miner {  
 temp := sha256.Sum256([]byte("miner" + time.Now().String()))  
 miner := Miner{  
 addr: temp[:],  
 num: 0,  
 coinAge: 0,  
 }  
 return &miner  
}

// 初始化Miners数组的函数，调用AddMiner函数，生成一个Miner，然后将其添加到Miners数组中

func InitMiners() []Miner {

miner := createMiner()

Miners := []Miner{\*miner}

return Miners

}

// 传入一个Miner和Miners数组，将miner添加到Miners数组中

func AddMiner(miner Miner, Miners \*[]Miner) {  
 \*Miners = append(\*Miners, miner)  
}

//添加矿工

func AddMiners() {  
 var MinerNum int  
 fmt.Print("请输入创建矿工的数量：")  
 fmt.Scanf("%d",&MinerNum)  
 for i := 0; i < MinerNum; i++ {  
 AddMiner(\*createMiner(),&Miners)  
 }  
}

// 创建一种名为Coin的结构体，包含币的数量，矿工序号，币的时间戳

type Coin struct {

Num int64

MinerIndex int64

Time int64

}

func NewCoin(MinerIndex int64, Miners []Miner) Coin {  
 n, \_ := rand.Int(rand.Reader, big.NewInt(4))  
 coin := Coin{  
 Num: 1 + n.Int64(),  
 MinerIndex: MinerIndex,  
 Time: time.Now().Unix(),  
 }  
 Miners[MinerIndex].num += coin.Num  
 return coin  
}

// 初始化coins数组函数，调用NewCoin函数，生成一个coin，然后将其添加到coins数组中

func InitCoins(Miners []Miner) []Coin {

coin := NewCoin(0, Miners)

Coins := []Coin{coin}

return Coins

}

// 创建一种名为Block的结构体,包含区块哈希，前区块哈希，区块号，难度值，矿工地址，奖励币数，时间戳

type Block struct {

Hash []byte

PrevHash []byte

Height int64

Dif int64

MinerAddr string

Reward Coin

Timestamp int64

tradeData string

}

// 生成创世区块，默认难度值为1，矿工地址为矿工数组0  
func GenesisBlock(Miners []Miner, Coins []Coin) Block {  
 temp := sha256.Sum256([]byte("Genesis Block"))  
 genesisBlock := Block{  
 Hash: temp[:],  
 tradeData: "Genesis Block",  
 PrevHash: []byte(""),  
 Height: 1,  
 Dif: 0,  
 MinerAddr: string(Miners[0].addr),  
 Reward: Coins[0],  
 Timestamp: time.Now().Unix(),  
 }  
 return genesisBlock  
}

// 生成区块函数，传入参数为矿工序号，矿工数组，Coin,tradeData,区块数组,新区块的Hash是tradeData的sha256的运算结果，PrevHash是上一个区块的哈希，区块号是上一个区块的区块号加1，难度值是上一个区块的难度值，矿工地址是矿工数组中对应序号的地址，奖励币数是Coin，时间戳是当前时间戳，将新生成的区块添加到区块数组中

func GenerateBlock(MinerNum int, Miners []Miner, coin Coin, tradeData string, bc \*[]Block) {

var newBlock Block

temp := sha256.Sum256([]byte(tradeData))

newBlock.Hash = temp[:]

newBlock.PrevHash = (\*bc)[len(\*bc)-1].Hash

newBlock.Height = (\*bc)[len(\*bc)-1].Height + 1

newBlock.Dif = (\*bc)[len(\*bc)-1].Dif

newBlock.MinerAddr = string(Miners[MinerNum].addr)

newBlock.Reward = coin

newBlock.Timestamp = time.Now().Unix()

newBlock.tradeData = tradeData

\*bc = append(\*bc, newBlock)

}

// 初始化函数，生成创世区块，并添加到区块链中

func InitBlockChain(Miners []Miner, Coins []Coin) []Block {

var bc []Block

bc = append(bc, GenesisBlock(Miners, Coins))

return bc

}

**2、更新矿工信息**

// 更新Miners数组函数，传入Coins数组和Miners数组，遍历Coins数组，将Coins数组中的币的矿工序号与Miners数组中的矿工序号相同的矿工的币龄加上（现在的时间-Coin的时间戳）\*Coin的数量

func UpdateMiners(Coins \*[]Coin, Miners \*[]Miner) []Miner {

for i := 0; i < len(\*Coins); i++ {

index := (\*Coins)[i].MinerIndex

(\*Miners)[index].coinAge += (time.Now().Unix() - (\*Coins)[i].Time) \* (\*Coins)[i].Num

(\*Coins)[i].Time = time.Now().Unix()

}

return \*Miners

}

**3、pos挖矿**

type MinerTime struct {

minerIndex int

totalTime int64

}

var start int64

var end int64

func AddMinerData(minerDatas \*[]MinerTime,minerData \*MinerTime) {

\*minerDatas = append(\*minerDatas, \*minerData)

}

// 函数名：Pos,传入Miners数组，当前难度值Dif和一个string类型变量tradeData，内设一个int变量timeCounter, 从0递增到Intmax，

//hash值为SHA256(SHA256(tradeData|timeCounter)),循环内遍历Miners数组，目标值target=Dif乘当前Miner的币龄，

//要求hash小于target，返回满足要求的第一个Miner的序号并清空这个Miner的币龄，一旦满足要求则退出整个循环

func Pos(Miners Miner, Dif int64, tradeData string) bool {

var timeCounter int

var realDif int64

realDif = int64(MinProbably)

if realDif +Dif\*Miners.coinAge > int64(MaxProbably) {

realDif = MaxProbably

} else {

realDif += Dif \* Miners.coinAge

}

target := big.NewInt(1)

// 数据长度为8位

//需求：需要满足前两位为0，才能解决问题

//1 \* 2 << (8-2) = 64

// 0100 0000

// 00xx xxxx

// 32 \* 8

target.Lsh(target, uint(realDif))

for timeCounter = 0; timeCounter < INT64\_MAX; timeCounter++ {

hash := sha256.Sum256([]byte(tradeData + string(timeCounter)))

hash = sha256.Sum256(hash[:])

var hashInt big.Int

hashInt.SetBytes(hash[:])

if hashInt.Cmp(target) == -1 {

return true

}

}

return false

}

func CorrectMiner(Miners \*[]Miner, Dif int64, tradeData string) int{

var minTime int64 = INT64\_MAX

var correctMiner int

var MinerData []MinerTime

for i := 0; i < len(\*Miners); i++ {

start = time.Now().UnixNano()

//最小持币量为2才能挖矿

time.Sleep(1)

if (\*Miners)[i].num >= 2 {

success := Pos((\*Miners)[i], Dif, tradeData)

if success ==true {

end = time.Now().UnixNano()

MinerDataDemo := MinerTime{

minerIndex: i,

totalTime: end - start,

}

AddMinerData(&MinerData, &MinerDataDemo)

}

}

}

if MinerData != nil {

fmt.Println(MinerData)

for j, \_ := range MinerData{

if MinerData[j].totalTime < minTime {

minTime = MinerData[j].totalTime

correctMiner = MinerData[j].minerIndex

}

}

(\*Miners)[correctMiner].coinAge = 0

return correctMiner

}

return -1

}

**4、生成新币**

// 传入新coin和coins数组，将其添加到coins数组中并保存，无返回值

func AddCoin(coin Coin, Coins \*[]Coin) {

\*Coins = append(\*Coins, coin)

}

1. **挖矿**

func Mine(Miners []Miner,Dif int64, tradeData string,BlockChain \*[]Block) {

fmt.Println("开始挖矿")

winnerIndex := CorrectMiner(&Miners , Dif, tradeData)

if winnerIndex == -1 {

panic("挖矿失败")

}

fmt.Println("挖矿成功")

fmt.Println("本轮获胜矿工:",winnerIndex)

AddCoin(NewCoin(int64(winnerIndex), Miners), &Coins)

GenerateBlock(winnerIndex, Miners, Coins[len(Coins)-1], tradeData, BlockChain)

time.Sleep(5\*time.Second)

UpdateMiners(&Coins, &Miners)

PrintMiners(Miners)

}

**6、打印本轮挖矿完后矿工信息**

//传入Miners数组，打印矿工数组每个矿工信息的函数

func PrintMiners(Miners []Miner) {

for i := 0; i <= len(Miners)-1; i++ {

fmt.Println("Miner", i, ":", hex.EncodeToString(Miners[i].addr), Miners[i].num, Miners[i].coinAge)

}

}

**7、是否继续挖矿**

func IsContinueMining() {

var isContinue string

for {

Mine(Miners, Dif, "New block",&BlockChain)

fmt.Println("是否继续挖矿?y or n")

fmt.Scanf("%s",&isContinue)

if isContinue == "y" {

continue

}else if isContinue == "n" {

fmt.Println("挖矿结束")

break

}else{

fmt.Println("输入错误")

continue

}

}

}

**7、挖矿结束**

**Note:每轮挖矿完延迟5s（time.Sleep(5\*time.Second)），用以计算币龄。**

1. **实验执行过程**

**执行的主程序：**

import (

"encoding/hex"

"fmt"

"time"

)

//创建币池数组Coins

var Coins []Coin

//调用InitBlockChain函数，生成一个区块数组

var BlockChain []Block

//默认难度值dif为1

var Dif int64 = 1

//创建矿工数组Miners

var Miners []Miner

func main() {

//默认难度值dif为1

//var Dif int64 = 1

//创建矿工数组Miners

//var Miners []Miner

Miners = InitMiners()

//添加矿工

AddMiners()

//创建币池数组Coins

//var Coins []Coin

//给矿工数组中的矿工添加币

Coins = InitCoins(Miners)

for i := 0; i < len(Miners); i++ {

AddCoin(NewCoin(int64(i), Miners), &Coins)

}

//调用InitBlockChain函数，生成一个区块数组

BlockChain = InitBlockChain(Miners, Coins)

//时间延迟，给出币龄

time.Sleep(5\*time.Second)

UpdateMiners(&Coins, &Miners)

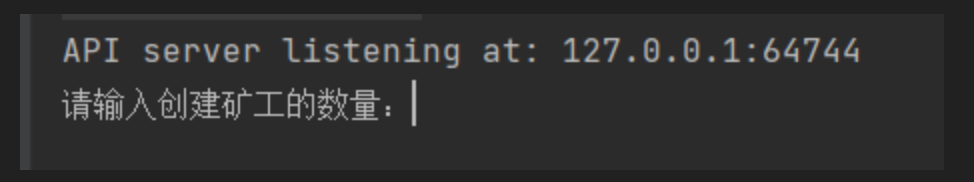
PrintMiners(Miners)

//挖矿

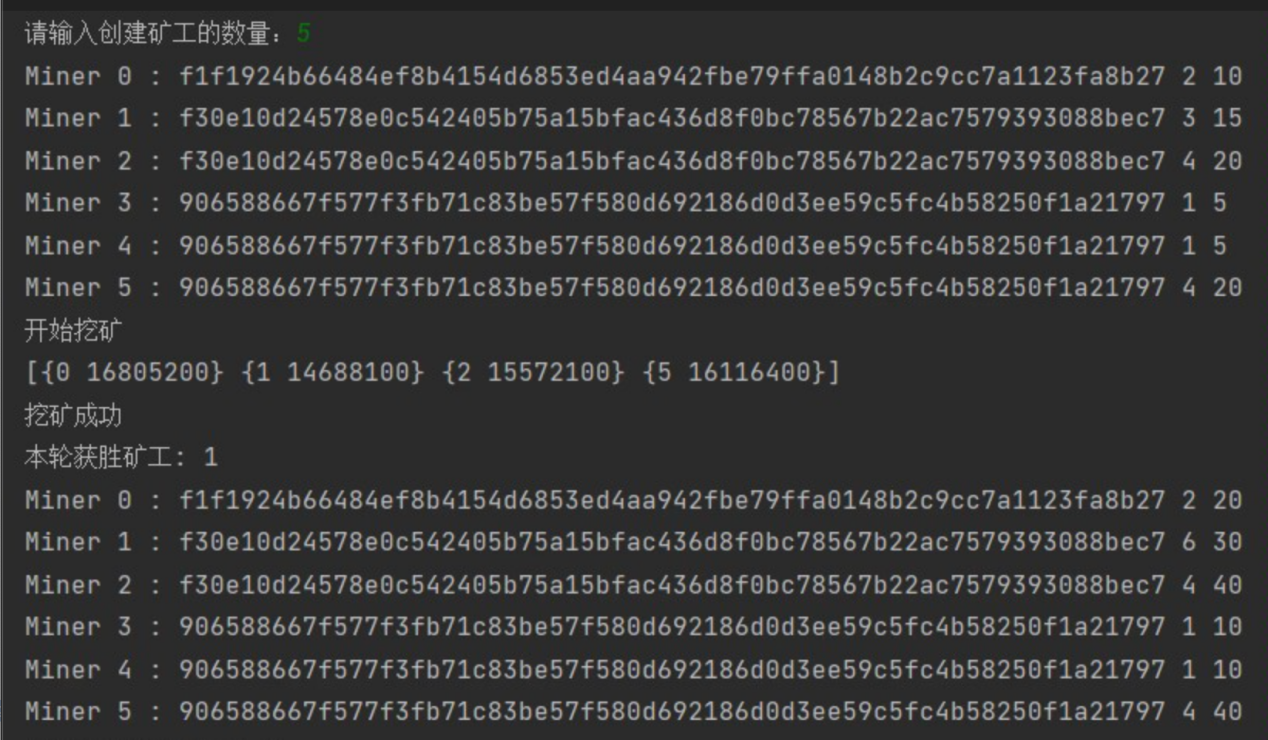
IsContinueMining()

}

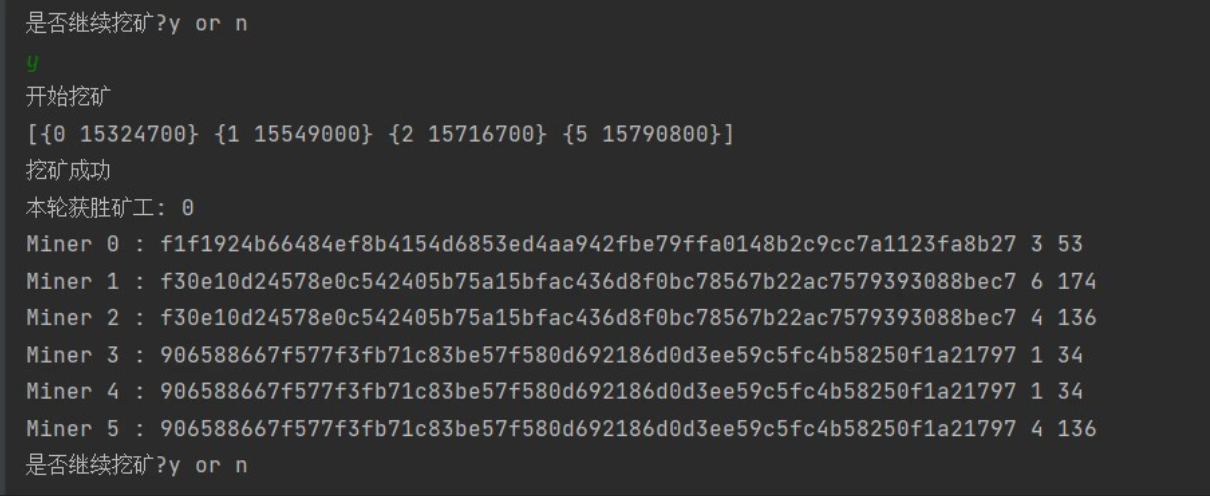
1. **输入初始矿工数**



1. **开始初始化并完成一轮挖矿，打印矿工信息**



1. **判断是否继续挖矿，输入y，开启下一轮挖矿**



1. **判断是否继续挖矿，输入n，挖矿结束**

