|  |  |
| --- | --- |
| 班级 |  |
| 姓名 |  |
| 学号 |  |

**一、实验名称**

基于 PoS 共识机制的区块链实验

**二、实验内容**

本实验旨在让学生通过构建一个基于 PoS 共识机制的模拟区块链系统，了解区块链的基本原理和 PoS 共识机制的实现过程。具体内容包括：

理解 PoS 共识机制的原理和实现方式

实现一个简单的 PoS 共识算法

构建一个基于 PoS 共识机制的区块链系统

**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1.20，不建议go版本低于1.13）

具备基本的命令行和编程能力的计算机操作系统

**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**

**数据结构：**

//Block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，时间戳，交易信息

type Block struct {

Hash []byte

PrevHash []byte

Height int64

Dif int64

MinerAddr string

Reward Coin

Timestamp int64

tradeData string

}

// Coin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

}

// Miner结构体，包含miner的地址和持币数量，以及记录的币龄

type Miner struct {

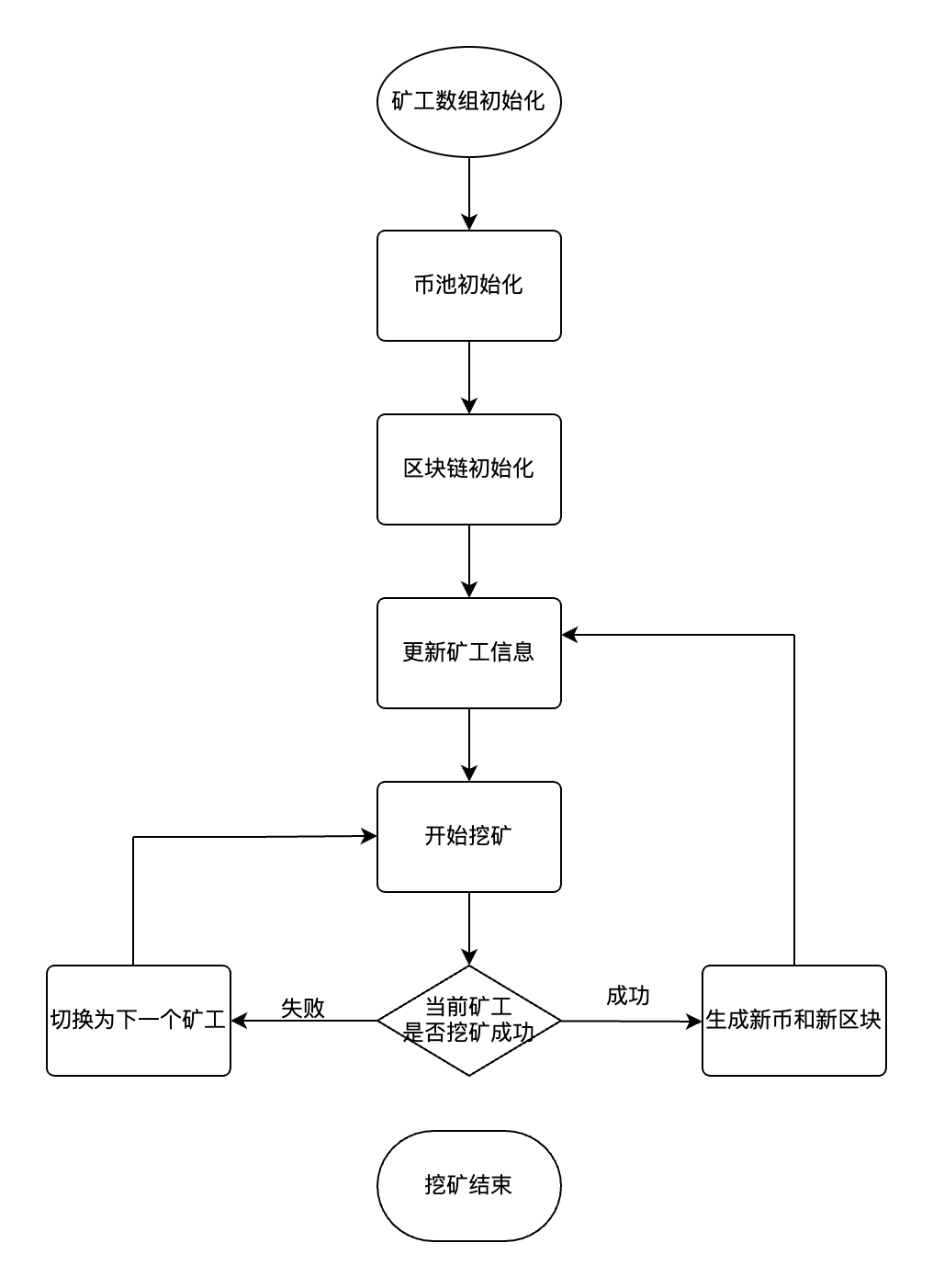
addr []byte

num int64

coinAge int64

}

**实验流程：**



**函数方法：**

// 创建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

// 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，返回一个Miners数组

func InitMiners() []Miner

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

func AddMiner(miner Miner, Miners \*[]Miner)

// 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

// 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

// 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

// 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)

// 生成新coin函数，传入矿工序号，返回一个coin

func NewCoin(MinerIndex int64, Miners []Miner) 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

// 传入新coin和coins数组，将其添加到coins数组中并保存

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

// 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) int

//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)

}

}

//main.go中集成后的挖矿函数

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

fmt.Println("开始挖矿")

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

fmt.Println("挖矿成功")

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)

}

**关键函数：**

// 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，并更新Coin的时间为当前时间

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

}

// Pos竞争挖矿权函数,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) int {

var timeCounter int

target := big.NewInt(1)

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[:])

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

//最小持币量为3才能挖矿

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

continue

}

//模拟实验降低挖矿难度，提高挖矿速度

// 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(Dif)\*uint((\*Miners)[i].coinAge))

if hashInt.Cmp(target) < 0 {

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

return i

}

}

}

return -1

}

**完整代码参考**

**<https://github.com/supredu/pos-simulate-experiment>**

1. **实验结果**

**测试用例：**

// 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1

AddMiner(\*createMiner(), &Miners)

PrintMiners(Miners)

//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

//var BlockChain []Block

BlockChain = InitBlockChain(Miners, Coins)

fmt.Println("创建第二个区块")

GenerateBlock(0, Miners, Coins[0], "second block", &BlockChain)

fmt.Println("创建结束")

//时间延迟，给出币龄

time.Sleep(5 \* time.Second)

UpdateMiners(&Coins, &Miners)

//fmt.Println(hex.EncodeToString(Miners[0].addr))

//fmt.Println(Miners[0].num)

PrintMiners(Miners)

//挖矿

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

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

//打印区块

for i, block := range BlockChain {

prevBlockHash := hex.EncodeToString(block.PrevHash)

currentHash := hex.EncodeToString(block.Hash)

if i == 0 {

fmt.Printf("prevBlockHash: %s, currentHash : 0x%s \n", prevBlockHash, currentHash)

} else {

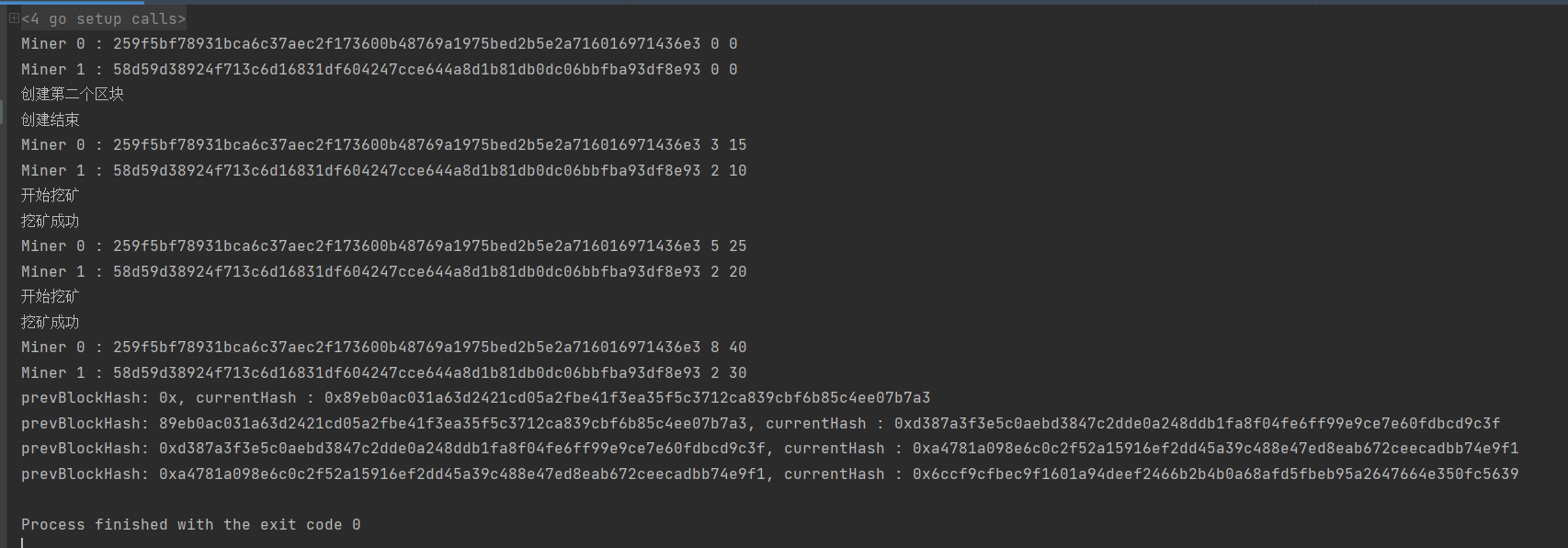
fmt.Printf("prevBlockHash: 0x%s, currentHash : 0x%s \n", prevBlockHash, currentHash)

}

}

}

**测试结果：**

****

**七、实验心得**