答案：

实验答案：

* 1. go语言，并实现了一个非常简单的区块链demo。

1. 创建区块

package core

import (

"time"

)

type Block struct {

Timestamp int64 //时间戳

Data []byte //区块信息

PrevBlockHash []byte //前一个区块哈希值

Hash []byte //自身哈希值

Nonce int //工作量证明

}

//创建新区块（传入data信息以及上一个区块的哈希值）

func Newblocks (Data string,PrevBlockHash []byte) \*Block{

block := &Block{time.Now().Unix(),[]byte(Data),PrevBlockHash,[]byte{},0}

pow := NewProfowork(block)

n,h := pow.Run() //工作量证明run方法

block.Hash = h

block.Nonce = n

return block

}

//创世纪块（第一个块）

func NewGenesisBlock() (\*Block) {

return Newblocks("Genesis Block",[]byte{})

}

1. 生产区块链

package core

type BlockChain struct{

Blocks []\*Block

}

//新区块加入区块链

func (b \*BlockChain) AddBlock(data string){

preB := b.Blocks[len(b.Blocks)-1]

newB := Newblocks(data,preB.Hash)

b.Blocks = append(b.Blocks,newB)

}

//创世区块链（唯创世块的区块链）

func NewBlockChain() \*BlockChain{

Newb := NewGenesisBlock()

a := []\*Block{Newb}

return &BlockChain{a}

}

1. 工作量证明

package core

import(

"crypto/sha256" //哈希算法包

"log"

"bytes"

"math"

"math/big" //大整数（哈希值很大）

"encoding/binary"

)

var MaxNonce = math.MaxInt64 //最多挖矿math.MaxInt64 次

const Targetbits = 20 //挖矿难度系数 越大越难

//工作量结构体

type Profofwork struct{

Block \*Block

Target \*big.Int

}

// 区块以及目标值

func NewProfowork(b \*Block) \*Profofwork{

target := big.NewInt(1)

target.Lsh(target,uint(256-Targetbits)) //移位运算（256-20）位

pow := &Profofwork{b,target}

return pow

}

// 数据整合成字节换算哈希值

func (p \*Profofwork)prepareDate(n int) []byte{

data := bytes.Join([][]byte{

p.Block.PrevBlockHash,

p.Block.Data,

IntToHex(p.Block.Timestamp),

IntToHex(int64(Targetbits)),

IntToHex(int64(n)),

},[]byte{},)

return data

}

// 寻找有效的哈希值（挖矿）

func (p \*Profofwork) Run() (int,[]byte){

var Hashbig big.Int

var hash [32]byte

var nonce = 0

for nonce < MaxNonce{

data := p.prepareDate(nonce)

hash = sha256.Sum256(data) //哈希算法生成哈希

Hashbig.SetBytes(hash[:]) //哈希值转为大整数

//与所移位运算后的大整数比较 小于才算有效哈希 返回哈希值以及循环次数 退出“挖矿”

if Hashbig.Cmp(p.Target) == -1{

break

}else{

nonce++

}

}

return nonce,hash[:]

}

// 校验区块有效与否

func (p \*Profofwork) Isvalidata() bool{

var Hashbig big.Int

//把记录的循环次数传参

data := p.prepareDate(p.Block.Nonce)

hash := sha256.Sum256(data)

Hashbig.SetBytes(hash[:])

is := Hashbig.Cmp(p.Target)== -1

return is

}

//将一个 int64 转化为一个字节数组（byte array）

func IntToHex(num int64) []byte {

buff:=new(bytes.Buffer)

err:=binary.Write(buff, binary.BigEndian, num)

if err !=nil{

log.Panic(err)

}

return buff.Bytes()

}

1. main.go

package main

import (

"core"

"fmt"

)

func main() {

//生成创世纪区块链

bc := core.NewBlockChain()

//🉐️区块

bc.AddBlock("send 1 to yoyo")

bc.AddBlock("send 12 to lala")

for \_, block := range bc.Blocks {

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

fmt.Printf("Data:%s\n", block.Data)

fmt.Printf("Hash:%x\n", block.Hash)

//校验区块

pow := core.NewProfowork(block)

fmt.Printf("pow:%t\n", pow.Isvalidata())

}

}

1.2设计一个很简单的， PoS 算法机制的代码实现，单纯地根据持币数量来做股权分配。

首先我们使用一个候选区块数组来保存，每一个 节点 广播过来的和自己当前节点生成的区块对象：

candidateBlocks [ ]Blocks 候选区块数组

每个区块结构体里面有一个变量 是用来记录生成这个区块的节点地址的

type Block struct {

Timestamp string // 时间戳，代表该区块的生成时间

Hash string // 这个区块的 hash 值

PrevHash string // 这个区块的 上一个 区块的 hash 值

NodeAddress string // 生成这个区块的 节点地址

Data string // 区块携带的数据

}

然后有一个 子线程，专门负责遍历 候选区块数组，来根据区块里面的节点地址 获取 它的代币数量，然后分配股权：

stakeRecord []string // 数组

for block ~ candidateBlocks {

coinNum = getCoinBalance(block.NodeAddress) // 获取代币数量

for i ~ coinNum { // 币有多少，就循环添加多少次

if stakeRecord.contains(block.NodeAddress) { // 是否以及包含了

break // 包含的就不再重复添加

}

stakeRecord = append(block.NodeAddress) // 添加

}

}

然后从 stakeRecord 中 选出一个竞选胜利者。这个概率就和上面的 coinNum 有关，越大就越有机会。

index = randInt() // 得出一个整形随机数

winner = stakeRecord[index] // 取出胜利者节点的地址

最后，我们就能取出这个 winner 所生成的区块来进行公链的接入，然后广播出去：

for block ~ candidateBlocks {

if block.NodeAddress == winner {

// 添加

}

}

// 广播出去

...