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. 设计代码解释区块链的原理和方法

Demo的开发环境是：Mac + VS Code + Dotnet Core 3.1.2。

环境：

$ dotnet --info

.NET Core SDK (reflecting any global.json):

Version: 3.1.201

Commit: b1768b4ae7

Runtime Environment:

OS Name: Mac OS X

OS Version: 10.15

OS Platform: Darwin

RID: osx.10.15-x64

Base Path: /usr/local/share/dotnet/sdk/3.1.201/

Host (useful for support):

Version: 3.1.3

Commit: 4a9f85e9f8

.NET Core SDKs installed:

3.1.201 [/usr/local/share/dotnet/sdk]

.NET Core runtimes installed:

Microsoft.AspNetCore.App 3.1.3 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.App]

Microsoft.NETCore.App 3.1.3 [/usr/local/share/dotnet/shared/Microsoft.NETCore.App]

1. 在这个环境下建立工程
2. 创建Solution

% dotnet new sln -o demo

The template "Solution File" was created successfully.

1. 用Console创建工程

% cd demo

% dotnet new webapi -o demo

The template "Console Application" was created successfully.

Processing post-creation actions...

Running 'dotnet restore' on demo/demo.csproj...

Determining projects to restore...

Restored demo/demo.csproj (in 170 ms).

Restore succeeded.

基础工程搭建成功

1. 创建区块Model

在工程下面，创建一个目录Models，并在目录下建立类Block.cs。

public class Block

{

public DateTimeOffset time\_stamp { get; set; }

public object data { get; set; }

public string pre\_hash { get; set; }

public string hash { get; set; }

public string nonce { get; set; }

}

1. 创建链

有了区块model，创建链很简单。创建一个BlockChains类，并在里面用SortedList建立一个链。

public class BlockChains

{

private static SortedList<int, Block> \_block\_chains = new SortedList<int, Block>();

}

1. 往链中增加区块

private static string \_hash\_zero = "Initialize\_Hash\_By\_WangPlus";

public bool addBlockData(object data)

{

Block new\_block = new Block()

{

time\_stamp = DateTimeOffset.Now,

data = data,

nonce = $"{\_random.Next(9999):D4}",

};

new\_block.pre\_hash = \_block\_chains.Count <= 0 ? \_hash\_zero : \_block\_chains.Last().Value.hash;

new\_block.hash = calculateHash(new\_block);

\_block\_chains.Add(\_block\_chains.Count + 1, new\_block);

return true;

}

private string calculateHash(Block block)

{

if (block == null)

return string.Empty;

string data\_json = JsonConvert.SerializeObject(block.data, Formatting.None);

string block\_string = $"{block.time\_stamp.Ticks.ToString()}|{block.pre\_hash}|{data\_json}|{block.nonce}";

var block\_hash = new SHA256Managed().ComputeHash(Encoding.UTF8.GetBytes(block\_string));

return Convert.ToBase64String(block\_hash);

}

在这个实现的方法中，

第一个区块需要特殊处理，因为他的pre\_hash不存在，所以我们给了一个默认的串。

计算hash时，我们把区块的time\_stamp、pre\_hash、data、nonce全都包含在里面了。

1. 验证区块

也是一个方法，加在BlockChains中：

public bool isBlockValid(int index)

{

if (index <= 0 || index > \_block\_chains.Count)

return false;

if ((index > 1 && \_block\_chains[index].pre\_hash != \_block\_chains[index - 1].hash) || (index == 1 && \_block\_chains[index].pre\_hash != \_hash\_zero))

return false;

if (\_block\_chains[index].hash != calculateHash(index))

return false;

return true;

}

private string calculateHash(int index)

{

return calculateHash(\_block\_chains[index]);

}

验证区块的部分，只做了简单验证：验证当前区块和前一个区块的hash是否匹配。