基于共识算法和区块链模拟实现超级账本

实验语言：GO

实验环境：GoLand 2022.1； go 1.13.4.widows-amd64.msi； curl-7.83.1

实验人：自动化96 薛荣坤 2196113513

实验中使用的开源包：

http://github.com/davecgh/go-spew/spew； http://github.com/gorilla/mux； http://github.com/joho/godotenv；

实验中使用的工具包：

"crypto/sha256""encoding/hex""encoding/json""fmt""io""log""net/http""os""strconv""strings""sync""time”

实验中的网络端口：http://localhost:8080

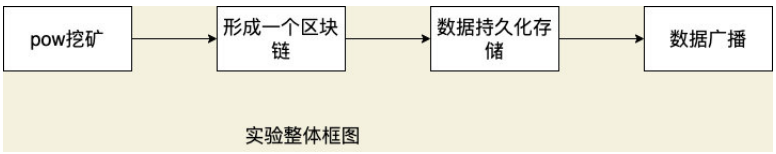
目录：

实验简述

1.基于pow算法挖矿并实现区块链的基本功能

2.基于LevleDB实现数据的KV存储

3.



实验背景

P2P

对等网络，即对等计算机网络，是一种在对等者之间分配任务和工作负载的分布式应用架构,网络的参与者共享他们所拥有的一部分硬件资源（处理能力、存储能力、网络连接能力、打印机等），这些共享资源通过网络提供服务和内容，能被其它对等节点直接访问而无需经过中间实体。在此网络中的参与者既是资源、服务和内容的提供者，又是资源、服务和内容的获取者。

比特币

与大多数货币不同，比特币不依靠特定货币机构发行，它依据特定算法，通过大量的计算产生，比特币经济使用整个P2P网络中众多节点构成的分布式数据库来确认并记录所有的交易行为，并使用密码学的设计来确保货币流通各个环节的安全性。P2P的去中心化特性与算法本身可以确保无法通过大量制造比特币来人为操控币值。基于密码学的设计可以使比特币只能被真实的拥有者转移或支付。

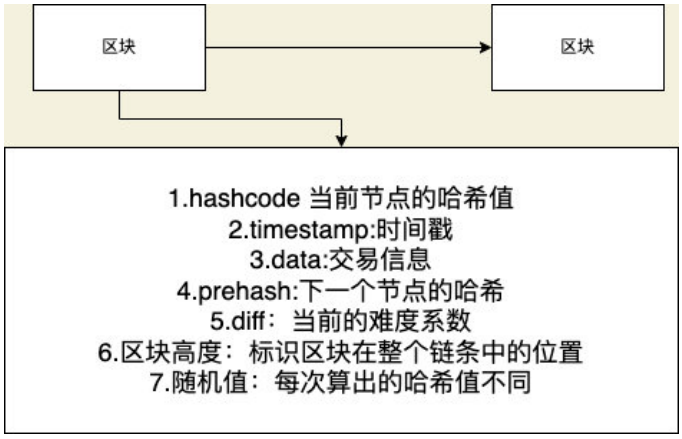
区块链

区块链，就是一个又一个区块组成的链条。每一个区块中保存了一定的信息，它们按照各自产生的时间顺序连接成链条。这个链条被保存在所有的服务器中，只要整个系统中有一台服务器可以工作，整条区块链就是安全的。这些服务器在区块链系统中被称为节点，它们为整个区块链系统提供存储空间和算力支持。如果要修改区块链中的信息，必须征得半数以上节点的同意并修改所有节点中的信息，而这些节点通常掌握在不同的主体手中。

实验第一部分-基于pow算法挖矿并实现区块链的基本功能

实验原理

区块有两种，一个是普通区块，一个就是创世区块。创世区块就是一项区块链项目中的第一个区块。

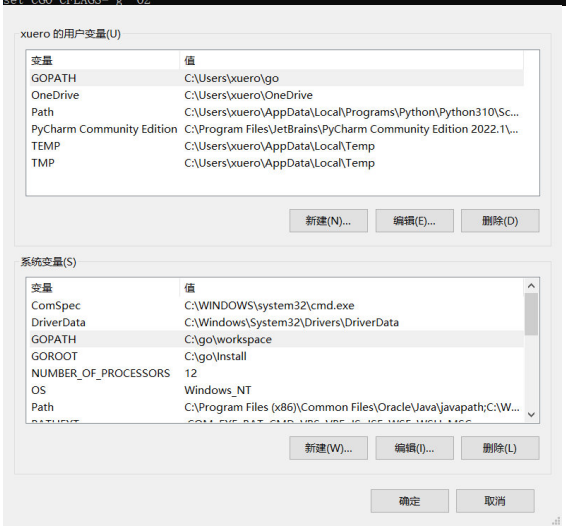


实验内容

配置环境

```
Microsoft Windows [版本 10.0.19044.1766]
(c) Microsoft Corporation。保留所有权利。

C:\Users\xuero>go env
set GO111MODULE=
set GOARCH=amd64
set GOBIN=
set GOCACHE=C:\Users\xuero\AppData\Local\go-build
set GOENV=C:\Users\xuero\AppData\Roaming\go\env
set GOEXE=.exe
set GOFLAGS=
set GOHOSTARCH=amd64
set GOHOSTOS=windows
set GONOPROXY=
set GONOSUMDB=
set GOOS=windows
set GOPATH=C:\go\workspace
set GOPRIVATE=
set GOPROXY=https://proxy.golang.org,direct
set GOROOT=C:\go\Install
set GOSUMDB=sum.golang.org
set GOTMPDIR=
set GOTOOLDIR=C:\go\Install\pkg\tool\windows_amd64
set GCCGO=gccgo
set AR=ar
set CC=gcc
set CXX=g++
set CGO_ENABLED=1
set GOMOD=
set CGO_CFLAGS=-g -O2
```



声明变量

```go
//区块结构体
type Block struct
{
    PreHash   string
    Hashcode  string
    TimeStamp string
    Diff      int
    Value     string
    Index     int
    Nonce     int
}

//区块链链表
type lian_my struct {
    NextNode *lian_my
    Data     *BLOCK.Block
}

```

这一步用于根据比特币定义我们的哈希值以及新的链表

• 方法

- 通过自带的数据包生成每一个区块的哈希值

```
1   func GenerationHashValue(block Block) string {
2      var hashdata = strconv.Itoa(block.Index) + strconv.Itoa(block.Nonce) +
    strconv.Itoa(block.Diff) + block.TimeStamp
3      var hashmy = sha256.New()
4      hashmy.Write([]byte(hashdata))
5      hashed := hashmy.Sum(nil)
6      return hex.EncodeToString(hashed)
7   }
8
```

- 生成之后的区块

```
1    func GenerateNextBlock(data string, oldBlock Block) Block {
2       var newBlock Block
3       newBlock.TimeStamp = time.Now().String()
4       newBlock.Diff = 3
5       newBlock.Index = oldBlock.Nonce + 1
6       newBlock.Value = data
7       newBlock.PreHash = oldBlock.Hashcode
8       newBlock.Nonce = 0
9       newBlock.Hashcode = pow(newBlock.Diff, &newBlock)
10      return newBlock
11   }
```

- 初始区块声明

```go
func GenerateFirstBlock(data string) Block {
    var chuangshi Block
    chuangshi.PreHash = "0"
    chuangshi.TimeStamp = time.Now().String()
    chuangshi.Diff = 3
    chuangshi.Value = data
    chuangshi.Index = 1
    chuangshi.Nonce = 0
    chuangshi.Hashcode = GenerationHashValue(chuangshi)
    return chuangshi
}
```

pow算法

```go
//pow工作量证明
func pow(diff int, block *Block) string {
    for {
        hashmy := GenerationHashValue(*block)
        if strings.HasPrefix(hashmy, strings.Repeat("0", diff)) {
            return hashmy
            fmt.Println("挖到了一个区块")
        } else {
            block.Nonce++
        }
    }
}
```

函数传入指针，这个相当于是在不断的挖矿，for是一个死循环，用block的信息生成一个哈希值，判断这一次的哈希值是否前面的0满足diff，这里可以直接使用hasprefix（）函数来进行判断，相当的方便。

区块链记录

```go
//创建头节点，保存创世区块
func CreaterHeader(data *BLOCK.Block) *lian_my {
    headerNode := new(lian_my)
    headerNode.NextNode = nil
    headerNode.Data = data
    return headerNode
}

//添加下一个结点
func Addnode(data *BLOCK.Block, node *lian_my) *lian_my {
```

```
    var newNode *lian_my = new(lian_my)
    newNode.NextNode = nil
    newNode.Data = data
    node.NextNode = newNode
    return newNode
}

//展示当前所有节点
func ShowNodes(node *lian_my) {
    n := node
    for {
        if n.NextNode == nil {
            fmt.Println(n.Data)
            break
        } else {
            fmt.Println(n.Data)
            n = n.NextNode
        }
    }
}
```

- **调用方法运行结果**

- 初始化第一个区块

```
GOROOT=C:\go\install #gosetup
GOPATH=C:\go\workspace\workspace #gosetup
C:\go\install\bin\go.exe build -o C:\Users\xuero\AppData\Local\Temp\GoLand\___go_build_mypow.exe mypow #gosetup
C:\Users\xuero\AppData\Local\Temp\GoLand\___go_build_mypow.exe
{0 4d1e3f5bea8b7d0421bd7349a87f73dbe085d1729776551a1c8b50702cc2b4fe 2022-06-23 2
2:28:34.9792346 +0800 CST m=+0.005982601 4 创世 1 0}
```

- 进行挖矿

```
{3 1 0 0 06eab553d843a716433c0107a88d56fe50a42ec2b95c0fd848b16c011d1af9fb 2022-06-24 09:22:43.5729843 +0800 CST m=+0.005985001 好家伙}
挖到了一个区块
{3 1 7465 06eab553d843a716433c0107a88d56fe50a42ec2b95c0fd848b16c011d1af9fb 00015f2109a03c09af79033c01b57630d6800f7b722014247fba1106b8655aa2 2022-06-24 09:22:43.5889832 +0800 CS
0.021983901 十分可爱}
挖到了一个区块
{3 7466 9604 00015f2109a03c09af79033c01b57630d6800f7b722014247fba1106b8655aa2 000ddb9034e969184fc8a8f91c7ee3246d71787211cf93e7ed22a085e891e470 2022-06-24 09:22:43.5959222 +0800
m=+0.028922901 记录数据}
```

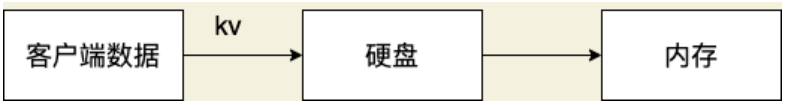
- 广播基本的区块链

```
打印当前链表
挖到了一个区块
挖到了一个区块
&{3 1 0 0 969dc37939c254be44eca4c426180d4825fc6cdd96ee2bb3661e5e30373d89dc 2022-06-24 09:35:0
&{3 5557 2617 000319b7a7011ed55f8381ea482068dd98c2eb39b689024fcba42835742e79a0 00054c44fb10fb
=+0.023943101 5ETH}
```

实验 第二部分-模拟实现基于LevelDB的KV存储

实验原理

leveldb是谷歌两位工程师使用C++实现的k-v存储系统，我们这里希望使用go进行复现， 属于按照key和value存储，用户也是可以重写排序函数，包含了最基本的数据操作接口，put,get,delay.同时多次操作可以认为是一次原子操作，可以用于支持事务。



实验代码里的方法及对象

结构体初始化

DB结构体

```
type DB struct {
   path string
   data map[string][]byte
}
```

这里path(字符串类型)用于存储连接的地址，data（byte类型）用于存储kv的键值

迭代器接口和结构体

```
type Iterator interface {
   Next() bool
   Key() []byte
   Value() []byte
}
type myIterator struct {
   data   []Pair
   index  int
   length int
}
```

NEXT判断是否下一个有值， key和value用于遍历键值

键值的结构体

```
type Pair struct {
   Key   []byte
   Value []byte
}
```

初始化结构体

```
func NewDefaultIterator(data map[string][]byte) *myIterator {
   my := new(myIterator)
   my.index = -1
   my.length = len(data)
   for k, v := range data {
      p := Pair{[]byte(k),
         v,
      }
      my.data = append(self.data, p)
   }
   return my
}
```

初始化迭代器的默认值并进行遍历

迭代器基本功能的实现

```
func (self *myIterator) Next() bool {
   if self.index < self.length-1 {
      self.index++
      return true
   }
   return false
}

func (self *myIterator) Key() []byte {
   if self.index == -1 || self.index >= self.length {
      panic(fmt.Errorf("INDEXOUTOFBOUNDERROR"))
   }
   return self.data[self.index].Key
}

func (self *myIterator) Value() []byte {
```

```
    if self.index >= self.length {
        panic(fmt.Errorf("INDEXOUTOFBOUNDERROR"))
    }
    return self.data[self.index].Value
}
```

实现下一个是否存在，以及返回value 和key

• 模拟功能实现

- 模拟连接

```
func New(path string) (*DB, error) {
    my := DB{
        path: path,
        data: make(map[string][]byte),
    }
    return &my, nil
}
```

- 模拟put,get

```
func (my *DB) Put(key []byte, value []byte) error {
    my.data[string(key)] = value
    return nil
}


func (my *DB) Get(key []byte) ([]byte, error) {
    if v, ok := my.data[string(key)]; ok {
        return v, nil//如果有则说明可以直接使用
    } else {//如果返回为空，则说明没有
        return nil,
        fmt.Errorf("Not Found")
    }
}
```

模拟 Del

```
func (self *DB) Del(key [](byte)) error {
   if _, ok := self.data[string(key)]; ok {
      delete(self.data, string(key))
      return nil
   } else {
      return fmt.Errorf("not Found")
   }
}
```

KB存储测试

迭代器测试

测试方法:

```
import (
   "fmt"
   "testing"
)

//迭代器测试
func TestNewDefaultIterator(t *testing.T) {
   data := make(map[string][]byte)
   data["K1"] = []byte("V1")
   data["K2"] = []byte("V2")
   data["K3"] = []byte("V3")
   data["K4"] = []byte("V4")
   my := NewDefaultIterator(data)
   if my.length != 3 {
      t.Fatal()
   }
   for iter.Next() {
      fmt.Printf("%s : %s\n", my.Key(), string(my.Value()))
   }
}
```

测试结果

```
=== RUN   TestNewDefaultIterator
K1 : V1
K2 : V2
K3 : V3
K4 : V4
--- PASS: TestNewDefaultIterator (0.00s)
PASS
```

DB测试

测试方法

```go
func Test_leveldb(t *testing.T) {
    db, err := New("")
    check(err)
    err = db.Put([]byte("k1"), []byte("v1"))
    check(err)
    err = db.Put([]byte("k4"), []byte("v4"))
    check(err)
    err = db.Put([]byte("k2"), []byte("v2"))
    check(err)
    err = db.Put([]byte("k1"), []byte("v1"))
    check(err)
    err = db.Put([]byte("k8"), []byte("v8"))
    check(err)
    v, err := db.Get([]byte("k8"))
    fmt.Printf("%s\n\n", v)

    if !bytes.Equal(v, []byte("v8")) {
        t.Fatal()
    }
    v, err = db.Get([]byte("k1"))
    if !bytes.Equal(v, []byte("v1")) {
        t.Fatal()
    }
    //err = db.Del([]byte("k1"))
    //check(err)
    iter := db.Iterator()
    for iter.Next() {
        fmt.Printf("%s - %s \n", string(iter.Key()), string(iter.Value()))
    }
}

func check(err error) {
    if err != nil {
        panic(err)
    }
}
```

测试结果

PUT和GET测试

```
k1 - v1
k4 - v4
k2 - v2
k8 - v8
```

DEL测试

```
k4 - v4
k2 - v2
k8 - v8
```

GET不存在元素时测试

```
--- FAIL: Test_leveldb (0.00s)
    leveldb_test.go:21:
```

实验第3部分-局域网广播和有效区块链

实验内容

1.可以在网络端POST指令访问，添加新的比特币交易区块。

2.可以在网络端GET指令访问，查看所有已经添加的交易信息。

实验原理

1.实验中使用的开源包：

http://github.com/davecgh/go-spew/spew ；

http://github.com/gorilla/mux ；

http://github.com/joho/godotenv ；

开源包功能：编写web程序的软件包，在控制台格式化输出结果，配置编写.env文件

实验方法

声明区块，区块链等基本（和前两部分一样）

注：基本代码和前两部分相同，但在下面处存在区别

```
var Blockchain []Block
var mutex = &sync.Mutex{}
```

上锁，并且有数组进行存储，同时在有效判别函数中，会抓取所有节点，寻找最长的链，来进行写入。但是在本次试验中因为我在本地并没有其他人的加入，所以我就用简单的python语言大概简述。

```
def resolve(self) ->bool:
  fujin=self.nodes
  new_chain = None
  max_length = len(self.chain)
for node in fujin:
   get new_length
    if new_legth>max_length.  and ishashvaild(self)
      max_length=length
      new_chain=chain
if new_chain:
    self.chain = new_chain
    return true
return false
```

互联网区块链方法

HTTP启动

```
//http启动
func run() error {
   mux := makeMuxRouter()
   httpAddr := os.Getenv("ADDR")
   log.Println("listening on", os.Getenv("ADDR"))
   s := &http.Server{
      Addr:           ":" + httpAddr,
      Handler:        mux,
      ReadTimeout:    10 * time.Second,
      WriteTimeout:   10 * time.Second,
```

```
        MaxHeaderBytes: 1 << 20,
    }
    if err := s.ListenAndServe(); err != nil {
        return err
    }
    return nil
}
```

对区块链数据的操作的初始化

```
func makeMuxRouter() http.Handler {
    muxRouter := mux.NewRouter()
    muxRouter.HandleFunc("/", handgetblockhain).Methods("GET")
    muxRouter.HandleFunc("/", handerwriteblock).Methods("POST")
    return muxRouter
}
```

- 查看互联网请求中的内容

```
func handgetblockhain(w http.ResponseWriter, r *http.Request) {
    bytes, err := json.MarshalIndent(Blockchain, "", "\t")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    io.WriteString(w, string(bytes))
}
```

验证当前区块是否加入

```go
func isblockivaild(newBlock, oldblock Block) bool {
    if oldblock.Index+1 != newBlock.Index {
        return false
    }
    if oldblock.HashCode != newBlock.PreHash {
        return false
    }
    if calculationHash(newBlock) != newBlock.HashCode {
        return false
    }
    return true
}
```

论证是否是旧的区块+1，以及哈希值是否对应

POST，挖到一个区块后进行广播

```go

//声明一个post类型的数据类型
type Message struct {
    BPM int
}

func handerwriteblock(writer http.ResponseWriter, request *http.Request) {
    writer.Header().Set("Content-Type", "application/json")
    var message Message
    decoder := json.NewDecoder(request.Body)
    if err := decoder.Decode(&message); err != nil {
        respondWithJson(writer, request, http.StatusNotFound, request.Body)
        return
    }
    defer request.Body.Close()

    //生成区块
    mutex.Lock()
    newblock := generateBlock(Blockchain[len(Blockchain)-1], message.BPM)
    mutex.Unlock()

    //判断是否合法
    if isblockivaild(newblock, Blockchain[len(Blockchain)-1]) {
        Blockchain = append(Blockchain, newblock)
        spew.Dump(Blockchain)
    }
    respondWithJson(writer, request, http.StatusCreated, newblock)
}
```

```
29
30        func respondWithJson(writer http.ResponseWriter, request *http.Request, code
    int, inter interface{}) {
31     writer.Header().Set("Content-Type", "application/json")
32     //格式化输出JSON
33     response, err := json.MarshalIndent(inter, "", "\t")
34     if err != nil {
35        writer.WriteHeader(http.StatusInternalServerError)
36        writer.Write([]byte("HTTP 500:Serve Error"))
37        return
38     }
39     writer.WriteHeader(code)
40     writer.Write(response)
41  }
42
```

- **实验结果**

- **测试主方法**

```
1   func main() {
2      err := godotenv.Load()
3      if err != nil {
4         log.Fatal(err)
5      }
6      go func() {
7         genesisblock := Block{}
8         genesisblock = Block{
9            0, time.Now().String(), 0, calculationHash(genesisblock),
10           "", differculty, 0}
11        mutex.Lock()
12        Blockchain = append(Blockchain, genesisblock)
13        mutex.Unlock()
14        spew.Dump(genesisblock)
15     }()
16     log.Fatal(run())
17  }
```

.env配置文件

```
ADDR= 8080
```

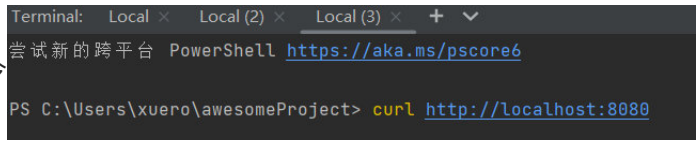
Curl测试命令事例

```
curl -H "Content-Type: application/json" -X POST -d"
{\"BPM\":10}"“http://localhost:8080”

curl http://localhost:8080
```

实验结果

发送指令

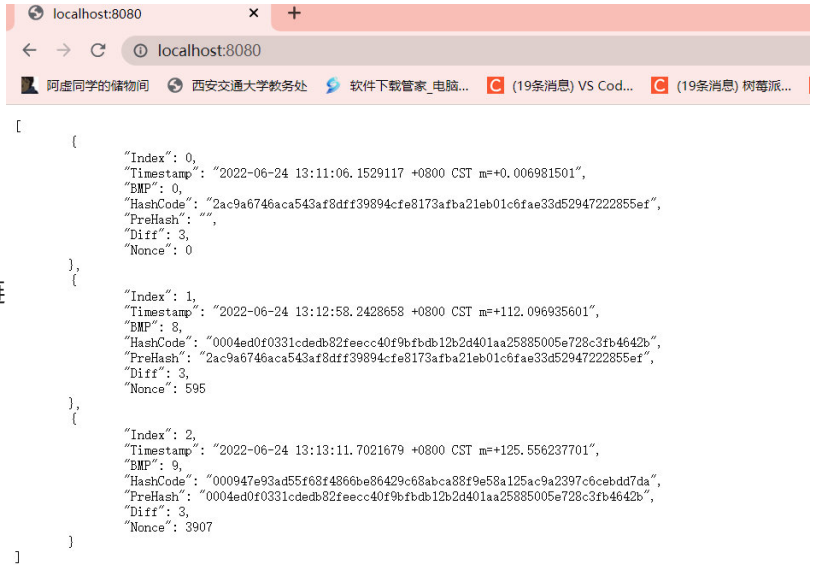


播在局域网中


```
[
        {
                "Index": 0,
                "Timestamp": "2022-06-24 13:11:06.1529117 +0800 CST m=+0.006981501",
                "BMP": 0,
                "HashCode": "2ac9a6746aca543af8dff39894cfe8173afba21eb01c6fae33d52947222855ef",
                "PreHash": "",
                "Diff": 3,
                "Nonce": 0
        }
]
```


发送被希望记录的交易信息

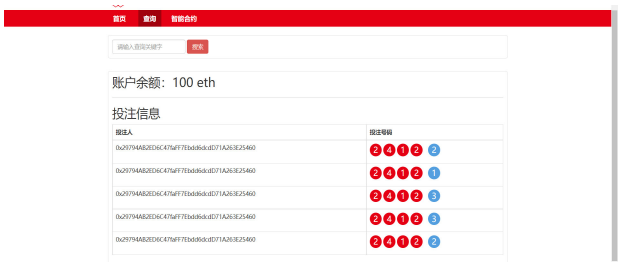
挖到矿后将其加入了区块链



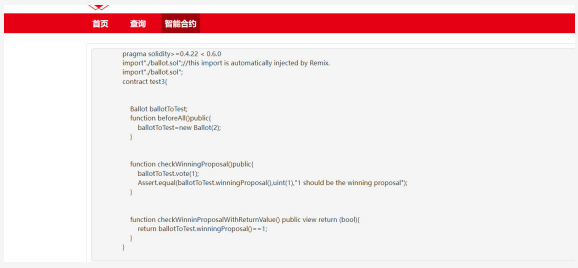
代码扩展部分

我个人是还想使用bee-go metaMast做一个使用区块链交易界面的，但是还是因为个人的问题，代码在golang的前端部分已经完善，但在remix智能公约部分陷进去，出不来了。所以我简要的附上几张图，工程文件里的view会附上我的html css文件。

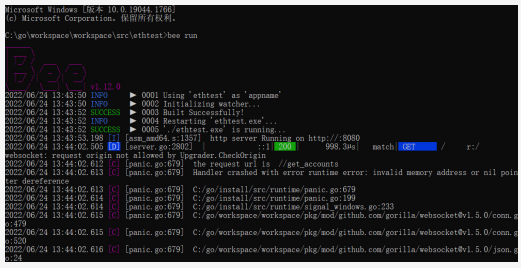
首页



查询



智能合约



beego

实验心得

在长达两个月的时间，我从GO语言的小白到可以逐步完善自己的想法，从只知道区块链可以炒股到逐渐对其中的原理有了更深的理解，尤其是在广播时候的代码，因为比特币公开的源代码是用c++编写的，所以用go浮现的时候就有了颇多困难。尤其是开发智能合约到最后还要学 *Solidity* 语言。

因为之前一直都在TI杯比赛中接触代码，但是比赛中大多数都是基于嵌入式环境的硬件系统，无论是ARDUINO,STM32还是ESP82266的开发虽然在调试上但是其中的编程语言都是基于C的，GO和css则完全是面向对象的，尤其是Golang的go get，每次下载一个包都是困难重重，无法cloning竟然还和内网有关。

但是面向对象的有趣在于可以大大减少程序员的调试难度，思路也更有逻辑性。暑假里 我一定要彻底完成我自己的以太坊彩票站。

也感谢沈老师和蔺老师在课堂上对于网络安全的讲解，让我认识到安全学科已经成为了新的行业热门，也了解到网络安全对于社会生产的重要性。

说明：

本实验中的代码均已开源在 gittee ；本说明文档首发于 个人博客