一、代码部分

main.go

package main

import (

    "crypto/sha256"

    "encoding/hex"

    "encoding/json"

    "io"

    "log"

    "net/http"

    "os"

    "time"

    "github.com/davecgh/go-spew/spew"

    "github.com/gorilla/mux"

    "github.com/joho/godotenv"

)

//数据模型

//定义一个结构体Struct，代表每一个块的数据模型

type Block struct {

    Index     int    //索引，表示是这个块在整个链中的位置

    Timestamp string //时间戳，块生成时的时间

    //DATA

    BPM       int    //一分钟心跳的频率，脉搏

    Hash      string //哈希，这个块通过 SHA256 算法生成的散列值

    PrevHash  string //代表前一个块的 SHA256 散列值

}

//定义一个结构表示整个链，最简单的表示形式就是一个 Block 的 slice：

var Blockchain []Block //区块链

//定义hash算法 SHA256 hasing

/\*我们使用散列算法（SHA256）来确定和维护链中块和块正确的顺序，

确保每一个块的 PrevHash 值等于前一个块中的 Hash 值，

这样就以正确的块顺序构建出链。\*/

func calculateHash(block Block) string {

    record := string(block.Index) + block.Timestamp + string(block.BPM) + block.PrevHash

    h := sha256.New()       //创建一个Hash对象

    h.Write([]byte(record)) //h.Write写入需要哈希的内容

    hashed := h.Sum(nil)    //h.Sum添加额外的[]byte到当前的哈希中，一般不是经常需要这个操作

    return hex.EncodeToString(hashed)

}

/\*这个 calculateHash 函数接受一个块，

通过块中的 Index，Timestamp，BPM，以及 PrevHash 值

来计算出 SHA256 散列值。\*/

//接下来我们就能便携一个生成块的函数：生成块block

func generateBlock(oldBlock Block, BPM int) (Block, error) {

    var newBlock Block

    t := time.Now()

    newBlock.Index = oldBlock.Index + 1

    newBlock.Timestamp = t.String()

    newBlock.BPM = BPM

    newBlock.PrevHash = oldBlock.Hash

    newBlock.Hash = calculateHash(newBlock)

    return newBlock, nil //此处error为后期完善复杂业务时留用，（可以省略）

}

/\*其中，Index 是从给定的前一块的 Index 递增得出，

时间戳是直接通过 time.Now() 函数来获得的，

Hash 值通过前面的 calculateHash 函数计算得出，

PrevHash 则是给定的前一个块的 Hash 值。\*/

//验证块，查看是否被篡改

func isBlockValid(newBlock Block, oldBlock Block) bool {

    if newBlock.Index != oldBlock.Index+1 {

        return false

    }

    if newBlock.PrevHash != oldBlock.Hash {

        return false

    }

    if calculateHash(newBlock) != newBlock.Hash {

        return false

    }

    return true

}

/\*检查 Index 来看这个块是否正确得递增，

检查 PrevHash 与前一个块的 Hash 是否一致，

再来通过 calculateHash 检查当前块的 Hash 值是否正确。\*/

//判断链长最长的作为正确的链进行覆盖（作为主链）

func replaceChain(newBlocks []Block) {

    if len(newBlocks) > len(Blockchain) {

        Blockchain = newBlocks

    }

}

/\*帮我们将本地的过期的链切换成最新的链：\*/

//Web服务

//借助 Gorilla/mux 包，我们先写一个函数来初始化我们的 web 服务：

func run() error {

    mux := makeMuxRouter()

    httpAddr := os.Getenv("PORT")

    log.Println("Listening on ",httpAddr )

    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

}

/\*再来定义不同 endpoint 以及对应的 handler。

例如，对“/”的 GET 请求我们可以查看整个链，

“/”的 POST 请求可以创建块。\*/

func makeMuxRouter() http.Handler {

    muxRouter := mux.NewRouter()

    muxRouter.HandleFunc("/", handleGetBlockchain).Methods("GET")

    muxRouter.HandleFunc("/", handleWriteBlock).Methods("POST")

    return muxRouter

}

//GET 请求的 handler：

func handleGetBlockchain(w http.ResponseWriter, r \*http.Request) {

    //json.MarshalIndent() - 格式化输出json

    bytes, err := json.MarshalIndent(Blockchain, "", "\t")

    if err != nil {

        http.Error(w, err.Error(), http.StatusInternalServerError)

        return

    }

    io.WriteString(w, string(bytes))

}

/\*在浏览器中访问 localhost:8080 或者 127.0.0.1:8080 来查看\*/

//POST 请求的 handler

/\*定义一下 POST 请求的 payload：\*/

type Message struct {

    BPM int

}

/\* handler 的实现：\*/

func handleWriteBlock(w http.ResponseWriter, r \*http.Request) {

    var m Message

    decoder := json.NewDecoder(r.Body)

    if err := decoder.Decode(&m); err != nil {

        respondWithJSON(w, r, http.StatusBadRequest, r.Body)

        return

    }

    defer r.Body.Close()

    newBlock, err := generateBlock(Blockchain[len(Blockchain)-1], m.BPM)

    if err != nil {

        respondWithJSON(w, r, http.StatusInternalServerError, m)

        return

    }

    if isBlockValid(newBlock, Blockchain[len(Blockchain)-1]) {

        newBlockchain := append(Blockchain, newBlock)

        replaceChain(newBlockchain)

        spew.Dump(Blockchain)

    }

    respondWithJSON(w, r, http.StatusCreated, newBlock)

}

//响应json格式化封装：返回客户端一个响应：

func respondWithJSON(w http.ResponseWriter, r \*http.Request, code int, payload interface{}) {

    reponse, err := json.MarshalIndent(payload, "", "\t")

    if err != nil {

        w.WriteHeader(http.StatusInternalServerError)         //响应错误状态码

        w.Write([]byte("HTTP 500: Internal Server Error"))    //响应错误内容

        return

    }

    w.WriteHeader(code)

    w.Write(reponse)

}

func main() {

    err := godotenv.Load()

    if err != nil {

        log.Fatal(err)

    }

    go func() {

        t := time.Now()

        genesisBlock := Block{0, t.String(), 0, "", ""}

        spew.Dump(genesisBlock)

        Blockchain = append(Blockchain, genesisBlock)

    }()

    log.Fatal(run())

}

/\*genesisBlock （创世块）是 main 函数中最重要的部分，

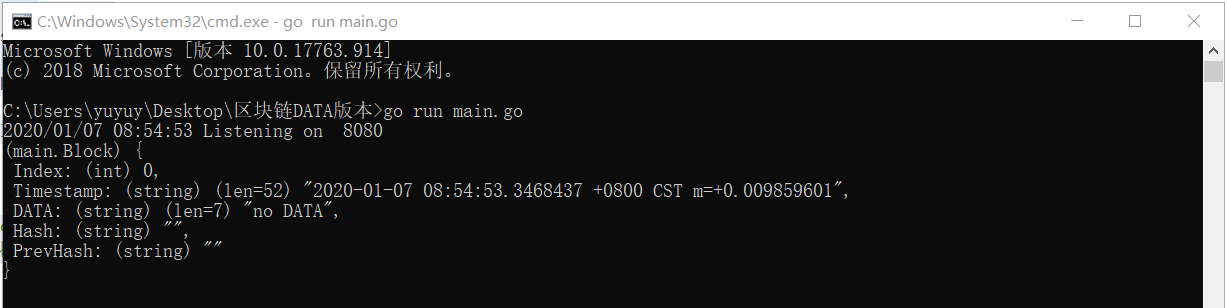
通过它来初始化区块链，毕竟第一个块的 PrevHash 是空的。\*/

.env文件

新建txt文档，输入PORT=8080然后保存。将其重新命名为.env.，于是命名后的文件就是.env。

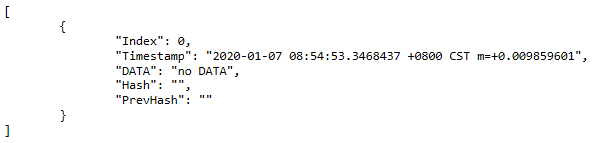
1. 结果验证：

首先在cmd中运行这个main.go。



发现已经产生了一个创世区块。

打开浏览器，输入localhost:8080，也会看到如下结果。



其中：

Index是块的号码；

Timestamp是时间戳，显示了程序运行时的时间；

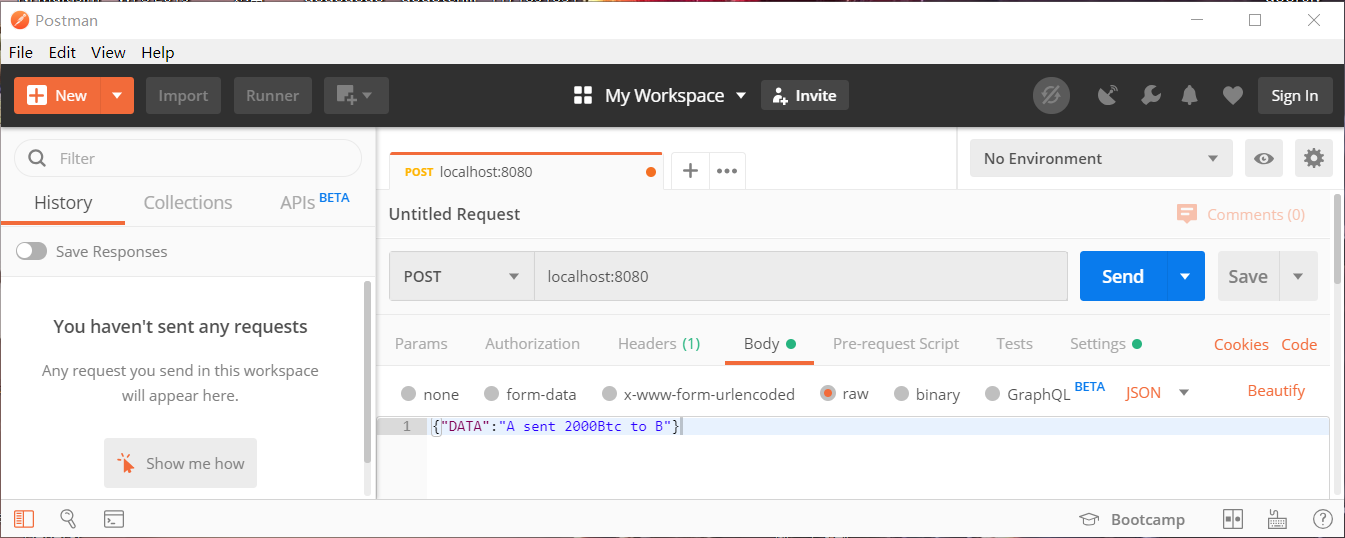
DATA应该表示交易信息，但创世区块由于没有交易信息，所以默认为“no DATA”。

Hash则是本区块的hash256的值，创世区块并没有；

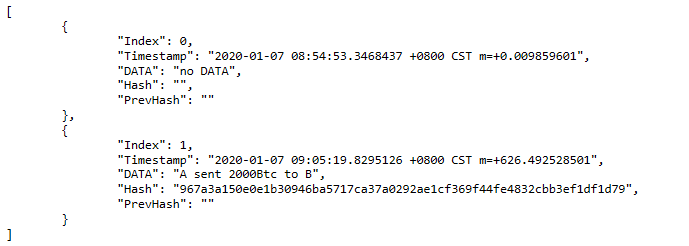
Prevhash则表示上一个区块的hash256的值，再创世区块上也没有。

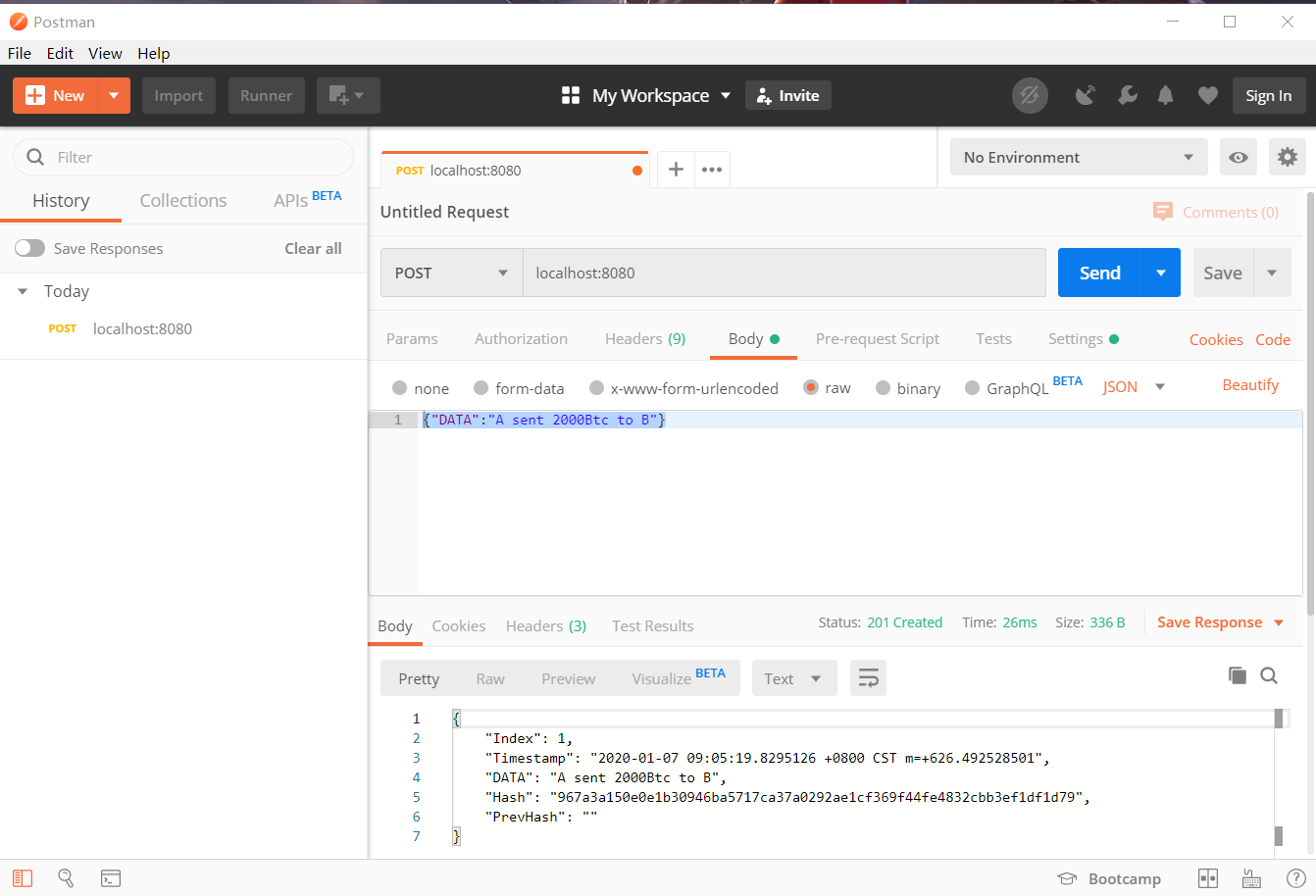
下面我们开始制造一个区块链：

我们用Postman这个软件来测试，即用此软件发送data。



输入：{"DATA":"A sent 2000Btc to B"}，然后send：浏览器上会产生一个新的区块（要刷新一下页面）：





多次“交易”会形成一个区块链：



以上就是实现了一个简易的本地的区块链的过程。