# 目录



# 一. flag基本使用

通常我们在写命令行程序（工具、server）时，对命令参数进行解析是常见的需求。各种语言一般都会提供解析命令行参数的方法或库，以方便程序员使用。在 go 标准库中提供了一个包：`flag`，方便进行命令行解析。

## 1.导入flag包

```
import (
    "flag"
)
```

## 2.使用示例

```
//定义一个字符串flag,flag名为printchain,默认值为:hello BTC world,参数说明: 输出所有的区块信息
flagString := flag.String("printchain", "hello BTC world", "输出所有的区块信息")
//定义一个整型flag,flag名为:number ,默认值为:6 ,参数说明:输入一个整数
flagInt := flag.Int("number", 6, "输出一个整数...")
//定义一个布尔类型的flag,flag名为:open,默认值:false,参数说明:判断真假
flagBool := flag.Bool("open", false, "判断真假...")
//解析flag
flag.Parse()
//输入参数后的值
fmt.Printf("%s\n", *flagString)
fmt.Printf("%d\n", *flagInt)
fmt.Printf("%t\n", *flagBool)
```

### 执行结果

```
//编译main.go
go build main.go
```

### 2.1 默认执行

```
./main  //执行./main，输出所有flag的默认值
hello BTC world

6

false
```

### 2.2 .输入一个未定义的flag

```
./main -xx //xx并未定义，输出错误信息以及所有flag使用方法(通过flag.Parse()生效)
flag provided but not defined: -xx

Usage of ./main:

  -number int

        输出一个整数... (default 6)

  -open

        判断真假...

  -printchain string

        输出所有的区块信息 (default "hello BTC world")
```

### 2.3 输入已定义的flag

```
./main  -printchain //仅仅传入flag,提示flag需要一个参数值

flag needs an argument: -printchain
Usage of ./main:
  -number int
        输出一个整数... (default 6)
  -open
        判断真假...
  -printchain string
        输出所有的区块信息 (default "hello BTC world")
```

### 2.4 输入已定义的flag以及对应类型的值

```
./main  -printchain "hello bruce" //传入flag=printchain,value="hello bruce"
hello bruce
6
false
./main  -number  88 //传入flag=number,value=88
hello BTC world
88
false
./main  -open  true //传入flag=open,value=true (默认为false,传入-open即为true)
hello BTC world
6
true
```

# 二.os.Args基本使用

os包提供了一些与操作系统交互的函数和变量，并且go对其做了一些封装。程序的命令行参数可以从os包的Args变量获取；os包外部使用os.Args访问该变量。

Go言里也采用左闭右开形式, 即，区间包括第一个索引元素，不包括最后一个, 因为这样可以简化逻辑。os.Args的第一个元素，os.Args[0], 是命令本身的名字；其它的元素则是程序启动时传给它的参数。

## 1.导入os包

```
import os
```

## 2. 使用示例

```
//实例化os.Args
args := os.Args
//打印args切片所有内容
fmt.Printf("%v\n",args)
//打印args切片的第二个参数
fmt.Printf("%v\n",args[1])
//打印args切片的第三个参数
fmt.Printf("%v\n",args[2])
```

### 传入参数

```
./main arg1 arg2  //执行main程序，传入两个参数arg
[./main arg1 arg2] //args
arg1              //args[1]
arg2              //args[2]
```

# 三.flag与os.Args组合使用

## 1.创建flag对象

```
//通过flag.NewFlagSet实例化flag对象
addBlockCmd := flag.NewFlagSet("addBlock", flag.ExitOnError)
printChainCmd := flag.NewFlagSet("printChain", flag.ExitOnError)
```

## 2.拼接flag

flag名称：data

默认值:bruce

使用说明:交易数据

使用方法： ./main addBlock -data

```
flagAddBlockData := addBlockCmd.String("data", "bruce", "交易数据")
```

## 3.判断flag

```
//判断os.Args中第二个flag的值
switch os.Args[1] {
case "addBlock":
   err := addBlockCmd.Parse(os.Args[2:])
   if err != nil {
      log.Panic(err)
   }
case "printChain":
   err := printChainCmd.Parse(os.Args[2:])
   if err != nil {
      log.Panic(err)
   }
default:
   printUsage()
   os.Exit(1)
```

```
}
```

## 4.解析flag输出信息

```
if addBlockCmd.Parsed() {
   if *flagAddBlockData == "" {
      printUsage()
      os.Exit(1)
   }
   fmt.Println(*flagAddBlockData)
}

if printChainCmd.Parsed() {
   fmt.Println("测试输出区块信息...")
}
```

## 5.相关函数

### 5.1 打印使用说明

```
func printUsage() {
   fmt.Println("Usage:")
   fmt.Println("\taddblock -data DATA -- 交易数据")
   fmt.Println("\tprintChain --输出区块信息")
}
```

### 5.2 判断是否输入flag

```
func isValid() {
    //如果args切片长度小于2(即没有跟参数，则打印使用方法并退出程序)
   if len(os.Args) < 2 {
      printUsage()
      os.Exit(1)
   }
}
```

## 6.测试命令行

### 6.1 无参数

```
./main
Usage:
addBlock -data DATA -- 交易数据:
printChain --输出区块信息
```

### 6.2 无效参数

```
./main xxx
Usage:
addBlock -data DATA -- 交易数据:
printChain --输出区块信息
```

### 6.3 参数默认值

```
./main  addBlock //输出flag=data的默认值bruce
bruce
```

```
./main   printChain //输出解析到printChain后打印的信息
测试输出区块信息...
```

### 6.4 指定参数

```
./main  addBlock -data "brucefeng" //输出flag=data的输入值brucefeng
brucefeng
```

# 四.通过命令行添加/查询区块

## 1.实现目标

```
Usage:
        createblockchain -data DATA --交易数据
        addBlock -data DATA -- 交易数据
        printChain --输出区块信息
```

## 2.定义命令行属性与方法 CLI.go

### 2.1 导入相关包

```
import (
   "fmt"
   "os"
   "flag"
   "log"
)
```

### 2.2 定义结构体

```
type CLI struct{}
```

### 2.3 打印帮助提示

```
func printUsage() {
   fmt.Println("Usage:")
   fmt.Println("\tcreateblockchain -data DATA --交易数据")
   fmt.Println("\taddBlock -data DATA -- 交易数据")
   fmt.Println("\tprintChain --输出区块信息")
}
```

### 2.4 判断参数输入是否合法

```
func isValid() {
   if len(os.Args) < 2 {
      printUsage()
      os.Exit(1)
   }
}
```

### 2.5 定义创世区块的方法

```
func (cli *CLI) createGenenisBlockChain(data string) {

   CreateBlockChainWithGenesisBlock(data)

}
```

### 2.6 定义普通区块的方法

```
func (cli *CLI) addBlock(data string) {
//判断数据库是否存在
   if !DBExists() {
      fmt.Println("当前不存在区块链，请先创建创世区块")
      os.Exit(1)
   }
//通过BlockChainObject()函数获取一个block
   blockchain := BlockChainObject()
   defer blockchain.DB.Close()
    //通过blockchain的AddBlockChain方法添加区块
   blockchain.AddBlockChain(data)
}
```

### 2.7 定义遍历区块的方法

```
func (cli *CLI) printChain() {
//判断数据库是否存在
   if !DBExists() {
      fmt.Println("当前不存在区块链")
      os.Exit(1)

   }

   blockchain := BlockChainObject()
   defer blockchain.DB.Close()
   blockchain.PrintChain()
}
```

### 2.8 定义命令行参数方法集合

```
func (cli *CLI) Run() {
//1.判断命令行参数是否合法
   isValid()
//2.通过flag.NewFlagSet实例化三个flag对象
    //创建创世区块
   createBlockCmd := flag.NewFlagSet("createblockchain", flag.ExitOnError)
    //创建普通区块
   addBlockCmd := flag.NewFlagSet("addBlock", flag.ExitOnError)
    //遍历打印区块信息
   printChainCmd := flag.NewFlagSet("printChain", flag.ExitOnError)

//3.定义命令行对象的相关属性
   flagCreateBlockChainWithData := createBlockCmd.String("data", "", "交易数
据")
   flagAddBlockData := addBlockCmd.String("data", "", "交易数据")

//4.根据参数值决定进行解析的信息
```

```
    switch os.Args[1] {
   case "createblockchain":
       err := createBlockCmd.Parse(os.Args[2:])
       if err != nil {
          log.Panic(err)
       }
    case "addBlock":
       err := addBlockCmd.Parse(os.Args[2:])
       if err != nil {
          log.Panic(err)
       }
    case "printChain":
       err := printChainCmd.Parse(os.Args[2:])
       if err != nil {
          log.Panic(err)
       }

    default:
       printUsage()
       os.Exit(1)

    }

//5. 判断是否解析成功 Parsed reports whether f.Parse has been called.
     //5.1 如果createBlockCmd解析成功，输出的DATA的值作为创建创世区块的传入参数
  if createBlockCmd.Parsed() {
       if *flagCreateBlockChainWithData == "" {
          fmt.Println("创世区块交易数据不能为空")
          printUsage()
          os.Exit(1)
       }
      //调用createGenenisBlockChain()方法创建创世区块
       cli.createGenenisBlockChain(*flatCreateBlockChainWithData)

    }
     //5.2 如果addBlockCmd解析成功，输出的DATA的值作为创建新区块的传入参数
    if addBlockCmd.Parsed() {
       if *flagAddBlockData == "" {
          fmt.Println("创建区块交易数据不能为空")
          printUsage()
          os.Exit(1)
       }
        //调用addBlock方法创建区块
       cli.addBlock(*flagAddBlockData)
    }
     //5.3 如果printChainCmd解析成功，调用遍历区块链的方法
    if printChainCmd.Parsed() {
       cli.printChain()
    }
```

```
}
```

### 2.9 代码整合

```
package BLC

import (
   "fmt"
   "os"
   "flag"
   "log"
)

type CLI struct{}

func printUsage() {
   fmt.Println("Usage:")
   fmt.Println("\tcreateblockchain -data DATA --交易数据")
   fmt.Println("\taddBlock -data DATA -- 交易数据")
   fmt.Println("\tprintChain --输出区块信息")
}

func isValid() {
   if len(os.Args) < 2 {
      printUsage()
      os.Exit(1)
   }
}

func (cli *CLI) addBlock(data string) {

   if !DBExists() {
      fmt.Println("数据库不存在")
      os.Exit(1)

   }

   blockchain := BlockChainObject()
   defer blockchain.DB.Close()
   blockchain.AddBlockChain(data)

}

func (cli *CLI) printChain() {
   if !DBExists() {
```

```
        fmt.Println("数据库不存在")
        os.Exit(1)

    }

    blockchain := BlockChainObject()
    defer blockchain.DB.Close()
    blockchain.PrintChain()
}

func (cli *CLI) createGenenisBlockChain(data string) {

    CreateBlockChainWithGenesisBlock(data)

}

func (cli *CLI) Run() {

    isValid()

    addBlockCmd := flag.NewFlagSet("addBlock", flag.ExitOnError)
    printChainCmd := flag.NewFlagSet("printChain", flag.ExitOnError)
    createBlockCmd := flag.NewFlagSet("createblockchain", flag.ExitOnError)

    flagAddBlockData := addBlockCmd.String("data", "", "交易数据")
    flagCreateBlockChainWithData := createBlockCmd.String("data", "", "交易数
据")
    switch os.Args[1] {
    case "addBlock":
        err := addBlockCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    case "printChain":
        err := printChainCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    case "createblockchain":
        err := createBlockCmd.Parse(os.Args[2:])
        if err != nil {
            log.Panic(err)
        }
    default:
        printUsage()
        os.Exit(1)

    }
```

```
    if addBlockCmd.Parsed() {
        if *flagAddBlockData == "" {
            printUsage()
            os.Exit(1)
        }

        cli.addBlock(*flagAddBlockData)
    }

    if printChainCmd.Parsed() {
        cli.printChain()
    }

    if createBlockCmd.Parsed() {
        if *flagCreateBlockChainWithData == "" {
            fmt.Println("交易数据不能为空")
            printUsage()
            os.Exit(1)
        }

        cli.createGenenisBlockChain(*flagCreateBlockChainWithData)

    }

}
```

# 3.生成区块链方法改造 BlockChain.go

## 3.1 定义判断数据库是否存在的方法

```
func DBExists() bool {
    if _, err := os.Stat(dbName); os.IsNotExist(err) {
        return false
    }
    return true
}
```

## 3.2 定义返回BlockChain对象

```
func BlockChainObject() *BlockChain {
//定义tip用于存储从数据库中获取到的最新区块的Hash值
    var tip []byte
 //打开数据库
    db, err := bolt.Open(dbName, 0600, nil)
    if err != nil {
        log.Fatal(err)
    }
```

```
    //通过key="l"获取value值(最新区块的Hash值)
    err = db.View(func(tx *bolt.Tx) error {
        //获取表对象
        b := tx.Bucket([]byte(blockTableName))
        if b != nil {
            //获取最新区块的Hash值
            tip = b.Get([]byte("l"))

        }
        return nil
    })
    //返回保存最新区块Hash信息的BlockChain对象
    return &BlockChain{tip, db}
}
```

### 3.3 创建带有创世区块的区块链

```
func CreateBlockChainWithGenesisBlock(data string) {

    //判断数据库是否存在
    if DBExists() {
        fmt.Println("创世区块已经存在")
        os.Exit(1)
    }

    //打开数据库
    db, err := bolt.Open(dbName, 0600, nil)
    if err != nil {
        log.Fatal(err)
    }

    err = db.Update(func(tx *bolt.Tx) error {
        //创建数据库表
        b, err := tx.CreateBucket([]byte(blockTableName))
        if err != nil {
            log.Panic(err)
        }

        if b != nil {
            //创建创世区块
            genesisBlock := CreateGenesisBlock(data)
            //将创世区块存储至表中
            err := b.Put(genesisBlock.Hash, genesisBlock.Serialize())
            if err != nil {
                log.Panic(err)
            }

            //存储最新的区块链的hash
```

```
            err = b.Put([]byte("l"), genesisBlock.Hash)
            if err != nil {
                log.Panic(err)
            }
        }

        return nil
    })
    if err != nil {
        log.Fatal(err)
    }

}
```

### 3.4 创建添加区块的方法

```
func (blc *BlockChain) AddBlockChain(data string) {

    err := blc.DB.Update(func(tx *bolt.Tx) error {
        //1.获取表
        b := tx.Bucket([]byte(blockTableName))
        //2.创建新区块
        if b != nil {
            //获取最新区块
            byteBytes := b.Get(blc.Tip)
            //反序列化
            block := DeserializeBlock(byteBytes)

            //3. 将区块序列化并且存储到数据库中
            newBlock := NewBlock(data, block.Height+1, block.Hash)
            err := b.Put(newBlock.Hash, newBlock.Serialize())
            if err != nil {
                log.Panic(err)
            }

            //4.更新数据库中"l"对应的Hash
            err = b.Put([]byte("l"), newBlock.Hash)
            if err != nil {
                log.Panic(err)
            }
            //5. 更新blockchain的Tip
            blc.Tip = newBlock.Hash
        }

        return nil

    })
```

```
    if err != nil {
        log.Panic(err)
    }

}
```

### 3.5 遍历区块链的方法

```
func (blc *BlockChain) PrintChain() {

    blockchainIterator := blc.Iterator()

    for {
        block := blockchainIterator.Next()

        fmt.Printf("Height:%d\n", block.Height)
        fmt.Printf("PreBlockHash:%x\n", block.PreBlockHash)
        fmt.Printf("Data:%s\n", block.Data)
        fmt.Printf("TimeStamp:%s\n", time.Unix(block.TimeStamp,
0).Format("2006-01-02 03:04:05 PM"))
        fmt.Printf("Hash:%x\n", block.Hash)
        fmt.Printf("Nonce:%d\n", block.Nonce)

        var hashInt big.Int
        hashInt.SetBytes(block.PreBlockHash)

        if big.NewInt(0).Cmp(&hashInt) == 0 {

            break
        }

    }

}
```

### 3.6 代码整合

```
package BLC

import (
    "github.com/boltdb/bolt"
    "log"
    "math/big"
    "fmt"
    "time"
    "os"
```

```
)

const dbName = "blockchain.db"  //数据库名
const blockTableName = "blocks" //表名
type BlockChain struct {
   Tip []byte   //区块链里面最后一个区块的Hash
   DB  *bolt.DB //数据库
}

//迭代器
func (blockchain *BlockChain) Iterator() *BlockChainIterator {
   return &BlockChainIterator{blockchain.Tip, blockchain.DB}

}

//判断数据库是否存在
func DBExists() bool {
   if _, err := os.Stat(dbName); os.IsNotExist(err) {
      return false
   }
   return true
}

func (blc *BlockChain) PrintChain() {

   blockchainIterator := blc.Iterator()

   for {
      block := blockchainIterator.Next()

      fmt.Printf("Height:%d\n", block.Height)
      fmt.Printf("PreBlockHash:%x\n", block.PreBlockHash)
      fmt.Printf("Data:%s\n", block.Data)
      fmt.Printf("TimeStamp:%s\n", time.Unix(block.TimeStamp,
0).Format("2006-01-02 03:04:05 PM"))
      fmt.Printf("Hash:%x\n", block.Hash)
      fmt.Printf("Nonce:%d\n", block.Nonce)

      var hashInt big.Int
      hashInt.SetBytes(block.PreBlockHash)

      if big.NewInt(0).Cmp(&hashInt) == 0 {

         break
      }

   }

}
```

```
//
func (blc *BlockChain) AddBlockChain(data string) {

   err := blc.DB.Update(func(tx *bolt.Tx) error {
      //1.获取表
      b := tx.Bucket([]byte(blockTableName))
      //2.创建新区块
      if b != nil {
         //获取最新区块
         byteBytes := b.Get(blc.Tip)
         //反序列化
         block := DeserializeBlock(byteBytes)

         //3. 将区块序列化并且存储到数据库中
         newBlock := NewBlock(data, block.Height+1, block.Hash)
         err := b.Put(newBlock.Hash, newBlock.Serialize())
         if err != nil {
            log.Panic(err)
         }

         //4.更新数据库中"l"对应的Hash
         err = b.Put([]byte("l"), newBlock.Hash)
         if err != nil {
            log.Panic(err)
         }
         //5. 更新blockchain的Tip
         blc.Tip = newBlock.Hash
      }

      return nil

   })
   if err != nil {
      log.Panic(err)
   }

}

//1.创建带有创世区块的区块链
func CreateBlockChainWithGenesisBlock(data string) {

   //判断数据库是否存在
   if DBExists() {
      fmt.Println("创世区块已经存在")
      os.Exit(1)
   }

   //打开数据库
```

```
    db, err := bolt.Open(dbName, 0600, nil)
    if err != nil {
        log.Fatal(err)
    }

    err = db.Update(func(tx *bolt.Tx) error {
        //创建数据库表
        b, err := tx.CreateBucket([]byte(blockTableName))
        if err != nil {
            log.Panic(err)
        }

        if b != nil {
            //创建创世区块
            genesisBlock := CreateGenesisBlock(data)
            //将创世区块存储至表中
            err := b.Put(genesisBlock.Hash, genesisBlock.Serialize())
            if err != nil {
                log.Panic(err)
            }

            //存储最新的区块链的hash
            err = b.Put([]byte("l"), genesisBlock.Hash)
            if err != nil {
                log.Panic(err)
            }
        }

        return nil
    })
    if err != nil {
        log.Fatal(err)
    }

}

//返回BlockChain对象
func BlockChainObject() *BlockChain {

    var tip []byte

    //打开数据库
    db, err := bolt.Open(dbName, 0600, nil)
    if err != nil {
        log.Fatal(err)
    }

    err = db.View(func(tx *bolt.Tx) error {
```

```
        b := tx.Bucket([]byte(blockTableName))

        if b != nil {
            //读取最新区块的Hash
            tip = b.Get([]byte("l"))

        }
        return nil

    })

    return &BlockChain{tip, db}
}
```

## 4. 未改造代码块

- Block.go
- BlockChainItertor.go
- help.go
- ProofOfWork.go

> 以上未改造代码可以参考:【Golang区块链开发003】区块序列化存储

```
https://mp.weixin.qq.com/s?__biz=MzA4Mzg5NTEyOA==&mid=2651781600&idx=1&sn=ce72dd45079759dd004be83ed3f9f90c&chksm=84152fd7b362a6c18d1b752b8ffc14d07a057dd4f263c14c3058a822a7b74f7ae7bcb9eb1cd2#rd
```

# 五.测试代码与测试结果

## 1. 测试代码main.go

```
func main() {
    //初始化CLI对象
    cli := BLC.CLI{}
    fmt.Println("====返回结果======")
    //执行Run命令调用对应参数对应的方法
    cli.Run()
}
```

## 2.测试结果

### 2.1 无参数执行

```
./main //直接执行
====返回结果======
Usage:
        createblockchain -data DATA --交易数据
        addBlock -data DATA -- 交易数据
        printChain --输出区块信息
```

## 2.2 首次遍历区块链

```
 ./main  printChain
====返回结果======
当前不存在区块链
```

## 2.3 首次添加普通区块

```
./main addBlock 或者 ./main  addBlock -data "bruce"
====返回结果======
当前不存在区块链，请先创建创世区块
```

## 2.4 创建创世区块

```
./main createblockchain --data "Create Genenis Block 20180708"
====返回结果======
第1个区块，挖矿成
功:0000d5b050b448b6db0d273b9320036f93b7cb2e8f8005c1e6192dc91b4a3381
2018-07-08 22:03:23.939771259 +0800 CST m=+0.087935424
```

## 2.5 遍历区块链

```
./main printChain
====返回结果======
Height:1
PreBlockHash:0000000000000000000000000000000000000000000000000000000000000000
00
Data:Create Genenis Block 20180708
TimeStamp:2018-07-08 10:03:23 PM
Hash:0000d5b050b448b6db0d273b9320036f93b7cb2e8f8005c1e6192dc91b4a3381
Nonce:61212
```

## 2.6 添加区块

```
./main  addBlock -data "bruce"
====返回结果======
第2个区块，挖矿成
功:000081872fe39a35be79e30f915c9716b9bbbae74f665fdd53f4ab57ae4b379d
2018-07-08 22:06:27.642330532 +0800 CST m=+0.055636890
```

## 2.7 遍历区块链

```
./main printChain
====返回结果======
Height:2
PreBlockHash:0000d5b050b448b6db0d273b9320036f93b7cb2e8f8005c1e6192dc91b4a33
81
Data:bruce
TimeStamp:2018-07-08 10:06:27 PM
Hash:000081872fe39a35be79e30f915c9716b9bbbae74f665fdd53f4ab57ae4b379d
Nonce:39906
Height:1
PreBlockHash:0000000000000000000000000000000000000000000000000000000000
00
Data:Create Genenis Block 20180708
TimeStamp:2018-07-08 10:03:23 PM
Hash:0000d5b050b448b6db0d273b9320036f93b7cb2e8f8005c1e6192dc91b4a3381
Nonce:61212
```

## 2.8 添加多个区块

```
./main  addBlock -data "bruce"
./main  addBlock  -data "send 100 BTC TO bruce"
......
第3个区块，挖矿成
功:0000565d60b6a26b8fc238bfeffb2cc1de20d28ca2a312bef9601997bb914fc3
2018-07-08 22:09:52.644353193 +0800 CST m=+0.032078853
第4个区块，挖矿成
功:0000dd1354cc868b96aeb18360b320b7a30ea0c00b27b79c32c0d9269ff2dbb3
2018-07-08 22:10:22.805477297 +0800 CST m=+0.210806453
```

## 2.9 遍历整个区块链

```
./main  printChain
====返回结果======
Height:6
PreBlockHash:000019ba65a19b26f81fae22025e963e68a6cf0983fb211aae208c42836825
2d
```

```
Data:bruce send 30 BTC TO jackma
TimeStamp:2018-07-08 10:12:57 PM
Hash:0000e3ba0e34ace19cba574deefdf5b75315b2ce10e488332952544e5a056571
Nonce:3502
Height:5
PreBlockHash:0000dd1354cc868b96aeb18360b320b7a30ea0c00b27b79c32c0d9269ff2db
b3
Data:bruce send 50 BTC TO ponyma
TimeStamp:2018-07-08 10:12:44 PM
Hash:000019ba65a19b26f81fae22025e963e68a6cf0983fb211aae208c428368252d
Nonce:80833
Height:4
PreBlockHash:0000565d60b6a26b8fc238bfeffb2cc1de20d28ca2a312bef9601997bb914f
c3
Data:send 100 BTC TO bruce
TimeStamp:2018-07-08 10:10:22 PM
Hash:0000dd1354cc868b96aeb18360b320b7a30ea0c00b27b79c32c0d9269ff2dbb3
Nonce:182779
Height:3
PreBlockHash:000081872fe39a35be79e30f915c9716b9bbbae74f665fdd53f4ab57ae4b37
9d
Data:bruce
TimeStamp:2018-07-08 10:09:52 PM
Hash:0000565d60b6a26b8fc238bfeffb2cc1de20d28ca2a312bef9601997bb914fc3
Nonce:21138
Height:2
PreBlockHash:0000d5b050b448b6db0d273b9320036f93b7cb2e8f8005c1e6192dc91b4a33
81
Data:bruce
TimeStamp:2018-07-08 10:06:27 PM
Hash:000081872fe39a35be79e30f915c9716b9bbbae74f665fdd53f4ab57ae4b379d
Nonce:39906
Height:1
PreBlockHash:0000000000000000000000000000000000000000000000000000000000000
00
Data:Create Genenis Block 20180708
TimeStamp:2018-07-08 10:03:23 PM
Hash:0000d5b050b448b6db0d273b9320036f93b7cb2e8f8005c1e6192dc91b4a3381
Nonce:61212
```

## 2.10 测试重新创建区块结构

```
./main  createblockchain -data "第二次创建创世区块"
====返回结果======
创世区块已经存在
```