# 2019-04-29工作记录-比特币公链节点间通信机制分析

现在已经编写模拟比特币公链的节点间交互代码，是时候进行一番总结分析了，本文就是在代码基础上的总结，记录了节点类型、节点启动、节点网络连接、节点命令种类、节点命令交互，完整的区块同步流程。

## 0 准备工作

标准的通信需要多台电脑，我们一般是在一台电脑上完成开发工作的，就需要把项目代码复制另一份，启动两个Go工程，同时运行两个程序。一个程序当做公共节点P，另一个程序当做普通节点N。当然这两个程序使用相同的IP，分别使用不同的端口，可从系统环境变量确定端口号。

## 1 节点类型

在这个模拟公链上可以分为公共节点和普通节点。

公共节点特点是长期运行，一般具备完整的区块数据，区块链高度最高。允许有多个公共节点，就形成了公共节点列表。

普通节点特点是区块高度落后，需要向公共节点申请下载缺少的最新区块数据。

## 2 节点启动过程

节点启动前需要先设置使用的端口号，通过系统环境变量的方法设置。

Windows系统的Go IDE命令行终端下：

set 查看所有环境变量

set NODE\_ID =3000 设置环境变量NODE\_ID，代表本终端程序使用的端口号3000

要求公共节点P设置成3000，普通节点N设置成3001。

程序启动时时如何确定自己的身份呢？是如何知道该向哪个公共节点查询区块高度呢？其实程序一启动就读取环境变量NODE\_ID，生成网络地址（IP+port）,与预置的公共节点列表中的地址比较，相同说明自己是公共节点，不同说明是普通节点。

读取环境变量的方法：

nodeID := os.Getenv(**"NODE\_ID"**)  
**if** nodeID==**""**{  
 startNodeCmd.Usage()  
 os.Exit(1)  
}

程序入口：

cli.go startNode()

*//启动节点***func** (cli \*CLI) startNode(nodeID string, minnerAddress string) {  
 fmt.Printf(**"Starting node: port=%s\n"**,nodeID)  
  
 **if** len(minnerAddress)>0{  
 **if** IsValidAdress([]byte(minnerAddress)){  
 fmt.Println(**"minner address is ok "**,minnerAddress)  
 }**else**{  
 log.Panic(**"Error: minner address"**)  
 }  
 }  
  
 StartServer(nodeID,minnerAddress, cli.bc)  
}

## 3节点网络连接

节点启动就要与其它节点发送、接收命令，因此少不了TCP通信流程。为了让公共节点能同时与多个普通节点进行通信，使用Go语言的协程分离处理客户端连接上的命令事件。



核心代码：

*//启动节点的服务器程序, 参数nodeID就是端口号***func** StartServer(nodeID, minerAddrerss string, bc \*BlockChain){  
 nodeAddress = fmt.Sprintf(**"localhost:%s"**,nodeID)  
  
 listen,err := net.Listen(**"tcp"**,nodeAddress)  
 checkErr(err)  
 **defer** listen.Close()  
  
 *//如果本程序监听的IP:port不是公共节点，就向公共节点发送自己的版本信息* **if** nodeAddress != knownNodes[0]{  
 sendVersion(knownNodes[0],bc)  
 }  
  
 **for**{  
 conn, err := listen.Accept()  
 **if** err != nil{  
 fmt.Println(err)  
 **continue** }  
 **go** HandleConnection(conn,bc) *//每收到连接就启动一个协程进行处理* }  
}

//处理其他节点的请求命令

func HandleConnection(conn net.Conn, bc \*BlockChain) {

request,err := ioutil.ReadAll(conn) //读取全部数据

checkErr(err)

//解析命令

cmd := bytesToCmd(request[:cmdLength])

fmt.Println("HandleConnection(): cmd=",cmd)

switch cmd{

case "version":

handleVersion(request,bc)

case "getblocks":

handleGetBlocks(request,bc)

case "inv":

handleInv(request,bc)

case "getdata":

handleGetData(request,bc)

case "blockdata":

handleBlockData(request,bc)

}

}

## 4节点命令种类

|  |  |  |
| --- | --- | --- |
| 命令 | 发送方功能说明 | 接收方功能说明 |
| **version** | 通知其他节点，宣告本节点的版本信息：区块高度，本节点IP地址。用于检查本节点是否区块高度落后。 | 用于检查本节点是否区块高度落后。落后就发送getblocks，领先就发送自身的version命令 |
| **getblocks** | 请求对方的区块hash列表 | 读取本节点的区块hash列表，发送inv命令 |
| **inv** | 请求对方节点的区块hash列表 | 保存区块hash列表，请求最新的区块数据，发送getdata命令 |
| **getdata** | 请求指定区块Hash的数据 | 发送blockdata命令 |
| **blockdata** | 请求区块数据 | 接收区块数据，保存到数据库，更新区块链尾部指针，更新UTXO |

## 5 节点命令交互

一般我们相信普通节点区块高度落后公共节点，因此让普通节点N先向公共节点P发送自己的版本命令。完整的同步区块流程如下：



构建网络区块链中，两个节点之间的实际交互：

a.有A，B两节点，B发现A节点，并向A发送版本信息(版本号、区块高度、B节点地址)

b.A节点获取到B节点的版本信息，会进行区块高度的比较，如果A节点区块高度大于B节点区块高度，那么A节点向B节点发起获取区块信息请求。如果A节点区块高度大于B节点区块高度，那么A节点向B节点发送版本信息。

c.B节点接收到A节点发送的获取区块信息请求。会将自己区块链中所有区块的hash值，返回给A节点。

d.A节点接收到B节点的信息后，会向B节点发送获取区块数据请求(包含B节点最新区块的hash值)。然后将B节点的信息，去除掉最新区块hash值后的结果赋值给一全局变量blockInTransit。

e.B节点接收到A节点发送的请求数据信息后，根据返回过来的最新区块hash值，获取区块。发送给A节点。

f.A节点接收到新区块后，将新区块以区块hash，区块序列化值，键值对形式存入数据库，并更新区块链ID。如果blockInTransit的长度大于0，那么继续讲最新hash值(blockInTransit[0])发送给B，请求获得最新区块,并将最新hash值提出出blockInTransit。如果blockInTransit的长度小于0，更新UTXO数据库桶数据。