|  |  |
| --- | --- |
| 文件路径 | peer/main.go/ 97-101行 |
| 功能描述 | Peer节点的入口文件 |
| 核心代码 | **mainCmd.AddCommand(version.Cmd())**  **mainCmd.AddCommand(node.Cmd())**  **mainCmd.AddCommand(chaincode.Cmd(nil))**  **mainCmd.AddCommand(clilogging.Cmd(nil))**  **mainCmd.AddCommand(channel.Cmd(nil))** |
| 说明 | 上面四行代码为cli提供了命令行，分别对应version命令（显示当前版本信息），node命令(提供 start 和 state 两个选项，分别对应启动Peer节点和显示节点信息),chaincode命令（提供create|fetch|join|list|update选项）, 信道命令，打印Level相关信息…… |
| 其他 | 重点关注 chaincode和channel相关命令，前者在执行chaincode的时候（也就是进行一次transaction）是会写入账本，后者在创建新的channel的时候会创建账本区块链的初始块 |

## create channel下一步 先分析channel 命令的create选项，观察创建channel时order节点是怎么创建初始块的。

https://zhuanlan.zhihu.com/p/26033176

|  |  |
| --- | --- |
| peer/channel/channel.go/ 70行 | channelCmd.AddCommand(**createCmd(cf)**) |
|  |  |
| peer/channel/create.go/ 200行 | func create(cmd \*cobra.Command, args []string, cf \*ChannelCmdFactory) |
|  |  |
| peer/channel/create.go/175行 | executeCreate(cf \*ChannelCmdFactory) |
| 到此为止已经解析了用户的CLI指令 |  |
|  |  |
| peer/channel/create.go/145行 | sendCreateChainTransaction(cf \*ChannelCmdFactory) |
| createChannelFromDefaults(cf) | 从默认设置创建channel transaction |
| createChannelFromConfigTx | 从用户设置创建channel transaction |
| sanityCheckAndSignConfigTx | 检查创建结果 |
| broadcastClient.Send(chCrtEnv) | 广播生成的transaction |
|  |  |
| peer/channel/create.go/72行 | createChannelFromDefaults(cf) |
| GetDefaultSigningIdentity() | 获取当前SMP提供的的身份信息 |
| MakeChainCreationTransaction (**chainID……**) | 通过chainID来调用chaincode创建channel,并发送transaction通知orderer节点创建channel |
|  |  |
| common/configtx /template.go/246行 | **func** MakeChainCreationTransaction(channelID ,signer msp.SigningIdentity) |
| 生成的transaction主要包含两个内容   1. Payload: paylBytes：包括channel相关信息，和签名相关信息。 2. Signature: sig：对payload的签名。 | **返回值：**&cb.Envelope{Payload: paylBytes, Signature: sig}，之后交由sendCreateChainTransaction函数将这个数据结构广播给order，由order写入账本 |
| . |  |
| peer/common/ordererclient.go/91行 | broadcastClient.Send(chCrtEnv) |
| err := s.client.Send(env) | s.client对应的是ab.AtomicBroadcast\_BroadcastClient 其中ab是位于order节点的代码目录里面。 |
| ab.AtomicBroadcast\_BroadcastClient |  |
| ab "github.com/hyperledger/fabric/protos/orderer" |  |
|  |  |
|  |  |
|  |  |
| 以下为Order节点的处理流程 |  |
|  |  |
| Order/main.go | ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server) |
| grpcServer是监听服务， server是处理的服务器，类型是ab.AtomicBroadcastServer于peer节点发送广播的结构体，ab.AtomicBroadcast\_BroadcastClient相对应 | 如果接受到的消息类型HeaderType\_CONFIG\_UPDATE，（这个类型的消息会定义当前channel中的基本配置，比如版本，channel中的设置能否被修改）则执行创建channel的相关操作。  首先对transaction进行检验\_, filterErr := support.Filters().Apply(msg)。只接受通过已经存在的chaincode生成的transaction。  之后将transaction放入order节点的队列中。if !support.Enqueue(msg)；代码位于/order/common/broadcast.go 106行 |
| 之后调用ConfigUpdateProcessor处理队列中的CONFIG\_UPDATE类型的transaction  /order/common/broadcast.go 35行 |  |
|  |  |
|  |  |

Q:1 广播的发送和接受是怎么实现的？

2 无法定位order节点对于create channel是怎么处理的。

|  |
| --- |
| **func** makeConfigMessage(chainID string) \*cb.Envelope {      payload := &cb.Payload{          Data: utils.MarshalOrPanic(&cb.ConfigEnvelope{}),          Header: &cb.Header{              ChannelHeader: utils.MarshalOrPanic(&cb.ChannelHeader{                  ChannelId: chainID,                  Type: int32(cb.HeaderType\_CONFIG\_UPDATE),              }),          },      }  **return** &cb.Envelope{          Payload: utils.MarshalOrPanic(payload),      }  }  **func** makeMessage(chainID string, data []byte) \*cb.Envelope {      payload := &cb.Payload{          Data: data,          Header: &cb.Header{              ChannelHeader: utils.MarshalOrPanic(&cb.ChannelHeader{                  ChannelId: chainID,              }),          },      }  **return** &cb.Envelope{          Payload: utils.MarshalOrPanic(payload),      }  }  表中是一个普通Message 和一个ConfigMessage的例子，后者是在创建channel时生成的，规定了当前channel中所有区块构建的格式，是当前channel账本中的一个记录。 |

## 先分析一下order节点是怎么处理账本的

在fabric1.0里面，存在有三种不同类型的账本结构，分别是File ledger、RAM ledger、JSON ledger，其中第一类为稳定版本，后两类为beta测试版本，可以在编译order库之前通过修改ORDERER\_GENERAL\_LEDGERTYPE环境变量的值来选择不同的账本结构（包括区块链，世界状态，History记录）。

|  |  |
| --- | --- |
| File ledger | 账本以文件形式存储，世界状态和history记录以轻量级数据库存储（K-V）形式 |
| RAM ledger | 存储在内存。块，但是不接受掉电 |
| JSON ledger | 以JSON文件形式存储，，世界状态和history记录以轻量级数据库存储（K-V）形式 |

目的是为了提供不同梯度的崩溃容忍度。

现在以File ledger为例进行分析。

|  |  |
| --- | --- |
| /order/ledger/ledger.go | 提供处理ledger的各种接口函数，函数的实现在相应的上级目录里面，例如对应File ledger的接口实现就在/order/ledger/file文件夹里面 |

主要分析func (flf \*fileLedgerFactory) GetOrCreate(chainID string) (ledger.ReadWriter, error)，

通过制定的chainID获取相应的账本，如果没有，则创建账本。

|  |  |
| --- | --- |
| /order/ledger/file/factory.go 33 行 | func (flf \*fileLedgerFactory) GetOrCreate(chainID string) (ledger.ReadWriter, error) |
| 这个借口的实现类型是一个fileLedgerFactory类型的指针，这个指针保存有当前order节点中所有的ledger记录，其中ledger的使用一个map结构体来存储的，map的key值是chainID，对应的value值是一个ledger.ReadWriter结构体，这个结构体的定义在/order//ledger/ledger.go。这个结构体可以提供对当前Ledger的查找，添加block等操作。 | 需要提出的是从代码上来看应该是一个chaincode对应一个ledger |
| 如果通过当前chainID没有查找到对应的ledger，则创建一个ledger.ReadWriter，创建ledger.ReadWriter的语句 | ledger = &fileLedger{blockStore: blockStore, signal: make(**chan** **struct**{})}  flf.ledgers[key] = ledger  其中fileLedger是核心结构体，定义在  /order/ledger/file/impl.go 35行，由 blkstorage.BlockStore结构体+ **chan** **struct**{}结构体组成 |
|  |  |
|  |  |
| blkstorage.**BlockStore定义在下面** |  |
| /common/ledger/blkstorage/blockstorge.go 64 行 | 提供了BlockStore的接口定义，这些接口的实现在上层目录/common/ledger/blkstorage/ fsblkstorage。主要分析一下AddBlock(block \*common.Block)  和RetrieveBlocks(startNum uint64) (ledger.ResultsIterator, error)。它的主要作用是维护和检索区块信息，比如上面两个接口分别实现了将block连接到Ledger上和通过ledger上面block的编号来检索区块。 |
|  |  |
|  |  |
| Addblock的实现如下 |  |
| /common/ledger/blkstorage/ fsblkstorage/block\_mgr.go/ 244 行 | **func** (mgr \*blockfileMgr) addBlock(block \*common.Block) error |
| 由blockfileMgr类型来实现这个接口，首先，函数会检查block的序列号是否是当前ledger的序列号+1(通过函数mgr.getBlockchainInfo().Height)，之后对block数据进行一系列的格式化（通过函数serializeBlock(block)），之后计算一些block的相关信息，比如block的大小（blockBytesLen := len(blockBytes)），从而判断当前文件的空闲空间能否装下当前Blcok（**if** currentOffset+totalBytesToAppend > mgr.conf.maxBlockfileSize）。并且更新一些相关信息，比如ledger的长度（lastBlockNumber: block.Header.Number），当前存储文件后缀编号（latestFileChunkSuffixNum: currentCPInfo.latestFileChunkSuffixNum,），block指针（blockFLP := &fileLocPointer{fileSuffixNum: newCPInfo.latestFileChunkSuffixNum}） |  |
|  |  |
|  |  |
|  |  |

Q: 1 在修改ledger的过程中好像并没有修改包括，世界状态，History记录，只对区块链进行了修改，怀疑这些代码存放在cli客户端这边？？？

2 在这次读写中，没有看到创建block的代码，block是直接作为函数的输入参数进来的，block的数据结构是怎么创建的的可能还需要往前面看（从服务端接受到客户端的广播开始）

block的结构定义存放在github.com/hyperledger/fabric/protos/common/block处，

3 注意 在这里的ledger实际上都是ledger.ReadWriter结构体，这个结构体并没有存放Ledger的具体数据，他只是提供了对Ledger进行维护和检索的函数接口。

## 下一步先读清楚block的结构体

|  |  |
| --- | --- |
| /fabric/protos/common/block.go | 这个文件定义了block的初始化函数和相应的接口函数，block结构体的定义在/fabric/protos/common/common.pb.go文件中 |
| **type** Block **struct** {      Header \*BlockHeader      Data \*BlockData      Metadata \*BlockMetadata  }block结构体分为三个部分， | type BlockHeader struct {  Number uint64  PreviousHash []byte ` "`  DataHash []byte `  }块头由三部分组成，分别是当前块的序号，前一个区块块头的hash，当前区块数据的hash。每一个块头都用指针连接了起来 |
|  | **type** BlockData **struct** {      Data [][]byte  }  块数据，使用一个二维byte数组组织起来，存放有当前chaincode的执行结果 |
|  | **type** BlockMetadata **struct** {      Metadata [][]byte  }  此部分数据表示一些与调用相关的属性值 |

## 几个关键的文件

|  |  |
| --- | --- |
| /fabric/protos/common/common.pb.go | 这个文件Protobuf包生成的文件，Protobuf是go语言提供的第三方库，用来生成编码后的数据包，并提供相应的编码，解码的功能使用 proto.Marshal 函数进行编码，使用 proto.Unmarshal 函数进行解码。他生成的文件后缀名就是.pb.go。上面这个文件中保存了fabric中绝大部分数据结构的定义，包括block结构，节点交互数据包的payload结构等。 |
| /fabric/protos/common/ | 这个文件夹里面定义了网络中传输的所有包的结构体。 |
| /fabric/protos文件夹下面所有以.proto后缀名结尾的文件都是网络包结构的定义文件，相应的.pb.go都是调用golang的第三方库grpc生成的协议文件，这些协议的具体使用方法可以参考grpc的使用方法，  附相关链接：  <http://www.cnblogs.com/andyidea/p/6526825.html> |  |

银链科技。。。

IBM

## 现在，结合cli端的输入指令分析用户是怎么实现通过制定chanID 实现transaction的

|  |  |
| --- | --- |
| 文件路径 | peer/main.go/ 97-101行 |
| 功能描述 | Peer节点的入口文件 |
| 核心代码 | **mainCmd.AddCommand(version.Cmd())**  **mainCmd.AddCommand(node.Cmd())**  **mainCmd.AddCommand(chaincode.Cmd(nil))**  **mainCmd.AddCommand(clilogging.Cmd(nil))**  **mainCmd.AddCommand(channel.Cmd(nil))** |
| 说明 | 上面四行代码为cli提供了命令行，分别对应version命令（显示当前版本信息），node命令(提供 start 和 state 两个选项，分别对应启动Peer节点和显示节点信息),chaincode命令（提供create|fetch|join|list|update选项）, 信道命令，打印Level相关信息…… |
| 其他 | 重点关注 chaincode和channel相关命令，前者在执行chaincode的时候（也就是进行一次transaction）是会写入账本，后者在创建新的channel的时候会创建账本区块链的初始块 |
| 指令的实现是调用了go语言的cobra包实现的，如果需要进行55二次开发，新增命令可以考虑从这里入手，上述5中命令的具体实现在均在peer/的子目录下的相应文件夹中 |  |

现在开始分析chaincode类的指令

|  |  |
| --- | --- |
| peer/chaincode/chaincode.go/ 46行 | Cmd(cf \*ChaincodeCmdFactory) \*cobra.Command |
| 在这里定义了chaincode类命令的几种类型，其中负责调用chaincode来查询transaction对应的方法是**queryCmd(cf)，其中输入cf是一个**ChaincodeCmdFactory类型的对象，这个对象的定义在文件peer/chaincode/common.go/229行处，里面包含了背书节点客户端pb.EndorserClient，MSP的签名类msp.SigningIdentity，负责广播的客户端  common.BroadcastClient。 | chaincodeCmd.AddCommand(installCmd(cf))  chaincodeCmd.AddCommand(instantiateCmd(cf))  chaincodeCmd.AddCommand(invokeCmd(cf))  chaincodeCmd.AddCommand(packageCmd(cf, nil))  chaincodeCmd.AddCommand(**queryCmd(cf)**)  chaincodeCmd.AddCommand(signpackageCmd(cf))  chaincodeCmd.AddCommand(upgradeCmd(cf)) |
|  |  |
|  |  |
| peer/chaincode/chaincode.go/invoke.go 28 行 | **invokeCmd(cf)的实现** |
| Cli端输入peer chaincode invoke  就会调用querCmd(cf)函数，这个函数的输入有三个，分别是用户的cli输入参数，一个string数组（来源暂时未定），一个上面提到的ChaincodeCmdFactory结构体，在对输入进行检查之后，调用函数chaincodeInvokeOrQuery(cmd, args, true, cf)进行处理 |  |
|  |  |
|  |  |
| peer/chaincode/chaincode/common.go 99 行 | chaincodeInvokeOrQuery(cmd, args, true, cf)的实现 |
| 函数首先会根据cli输入的参数调用函数getChaincodeSpec(cmd)获取指定的chaincode。之后调用重载函数ChaincodeInvokeOrQuery()，参数输入在邮编，这个函数会将用户请求发送给endorser节点，并有endorser节点模拟执行，如果此时是query型的命令，此时就可以返回结果了。否则还需要与Order节点通信进行共识+写Block | func ChaincodeInvokeOrQuery(      spec \*pb.ChaincodeSpec, //chaincode      cID string, //chainid      invoke bool, //是invoke操作函数query操作      signer msp.SigningIdentity,      endorserClient pb.EndorserClient,      bc common.BroadcastClient, //ChaincodeCmdFactory结构体中的三个成员变量  ) (\*pb.ProposalResponse, error)//返回值  这个函数的实现在peer/chaincode/chaincode/common.go 287行 |
| **这是个重点函数，它封装了query和invoke指令的具体实现。 返回的结构体**proposalResp就是最终的query/invoke结果 |  |
|  |  |
| peer/chaincode/chaincode.go/common.go 287行 | ChaincodeInvokeOrQuery() |
| 这个函数会按顺序做下面几个工作：   1. 格式化chaincode对象，调用函数ChaincodeInvocationSpec 2. 检查BroadcastClient 3. 使用msp对初始化身份信息 4. 判断cli调用的功能是Invoke还是query 5. 调用函数CreateProposalFromCIS产生要发送的编码后的数据包 6. 调用endorserClient.ProcessProposal发送给背书节点执行chaincode，并返回结果proposalResp这个结构体是一个数组，里面每一个元素都是一个背书节点的执行结果 7. 如果是invoke操作还需要向Order节点发送数据包进行写账本操作，调用CreateSignedTx函数生成发送给order节点的envelope,并调用send方法进行发送 | 需要重点分析的几个函数：   1. CreateProposalFromCIS()函数是怎么生成数据包的 2. endorserClient.ProcessProposal背书节点是怎么执行chaincode并返回结果的 3. invoke操作的CreateSignedTx和send方法是怎么实现发送的，同时order节点又是怎么处理这些请求的？ |
|  |  |
|  |  |
| 由于我们主要关心的是账本结构，这里姑且先分析客户端是怎么根据背书节点的返回信息生成发给order节点用来修改账本的结构体，也就是函数putils.CreateSignedTx(prop, signer, proposalResp)，第一个参数是上文第一步ChaincodeInvocationSpec格式化后的还没签名的chaincode对象，第二个参数是执行签名的结构体，第三个参数是上文由背书节点返回的chaincode执行结果。 | fabric/protos/utils/txutils.gp文件的111行，函数原型如下  **func** CreateSignedTx  (proposal \*peer.Proposal, signer msp.SigningIdentity, resps ...\*peer.ProposalResponse) (\*common.Envelope, error) 返回值是一个envelope结构体，这个结构体的实现在/fabric/protos/common/common.pb.go文件中，描述上=一个签过名了的payload |
| 这个函数会按顺序做如下几个操作：   1. 检查背书节点的结果是否为空**if** len(resps) == 0 2. 解码prop（头部，payLoad） 3. 检查prop头部的签名和singer的签名是否一致 4. 判断背书节点的执行结果是否一致， 5. 填写背书，生成的结构体为endorsements，这个结构体的说明在/protos/peers/proposal\_response.pb.go处。是一个背书节点对chaincode执行结果的签名 6. ….. 7. 生成一个envelope结构体   envelope结构体，这个结构体的实现在/fabric/protos/common/common.pb.go文件中，描述上=一个签过名了的payload | **之后就会调用**bc.Send(env)函数发送这个Envelope，其中bc 是一个common.BroadcastClient结构体，这个结构体的实现在fabric/peer/common/orderclient.go里面  实现流程如下： 1. 定义了一个BroadcastClient接口，包含send()和close()函数。   1. 使用 一个broadcastClient类来实现接口的函数，braodcast调用了ab.AtomicBroadcast\_BroadcastClient.send来实现发送，并返回发送的序号，其中返回也是调用ab.AtomicBroadcast\_BroadcastClient.recv来实现的。   AtomicBroadcast\_BroadcastClient 的实现在fabric/protos/orderer/ab.pb.go中，其函数的实现时调用了grpc.ClientStream的实现。 |
|  |  |
|  |  |
| /fabric/order/main.go 61行 | 现在开始考虑order节点是怎么接收广播并进行处理的，入口文件在左侧。 |
| 首先考虑order节点是怎么接受来自peer节点的广播数据包的，查看该文件的引用目录，发现有  Cb fabric/protos/common"  Ab hyperledger/fabric/protos/orderer"  两个文件，根据《几个关键文件中》protos中定义了大量的数据结构和网络结构。  order作为订阅广播的人，在Main函数处编写了server端，查看Main函数 | Main函数做了一下几个操作：   1. 调用golang的kingpin包（golang中的命令行解析插件） 如果用户输入start则启动Order 节点 如果用户输入version 则返回版本信息。 2. 重点查看start命令下的处理：主要工作时启动一系列的服务  * conf := config.Load()读取配置文件 * initializeLoggingLevel(conf) 设置日志级别 * grpcServer := initializeGrpcServer(conf)启动RPC服务器 * initializeLocalMsp(conf)启动MSP * signer := localmsp.NewSigner()为MSP注册signer * initializeMultiChainManager(conf, signer)启动共识服务方式（支持solo和kafka两种方式） * **server := NewServer(manager, signer)** * **ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server)** |
| 最后两个语句用于注册广播的server端，也就是说，这个server负责处理peer节点处发送过来的数据包，因为我们主要跟踪账本结构，接下来主要分析这两个语句。 | 其中**grpcServer.Server()对应监听服务，server对应相应的处理操作。** |
|  |  |
| /fabric/ orderer/server.go 61行 | 首先分析NewServer(ml multichain.Manager, signer crypto.LocalSigner) ab.AtomicBroadcastServer  输入值是multichain.Manager,（排序服务）+crypto.LocalSigner(签名服务)。  返回值是ab.AtomicBroadcastServer  它的定义在/fabric/protos/orderer/ab.pb.go文件中 |
| NewServer构建了一个server结构体（这个结构体是接口ab.AtomicBroadcastServer  的一个实现）， | 里面定义了两个handel,分别是deliver.NewHandlerImpl  broadcast.NewHandlerImpl，其实现分别在  fabric/orderer/common/broadcast"  fabric/orderer/common/deliver" |
|  |  |
|  |  |
| /fabric/ orderer/ab.pb.go 538行 | 分析函数RegisterAtomicBroadcastServer  (s \*grpc.Server, srv AtomicBroadcastServer) |
| 这个函数直接调用了  s.RegisterService(&\_AtomicBroadcast\_serviceDesc, srv) |  |
|  |  |
|  |  |
| 其中负责处理Broadcast的类就是newserver中的server结构体。 | 它的结构是**type** server **struct** {      bh broadcast.Handler      dh deliver.Handler  }+type AtomicBroadcastServer interface {  Broadcast(AtomicBroadcast\_BroadcastServer) error  Deliver(AtomicBroadcast\_DeliverServer) error  }  Handler和相应的方法一一对应，将handLer与相应方法结合在一起的代码在/fabric/ protos/orderer/ab.pb.gp文件里面  客户端通过Broadcast向orderer发送消息，orderer通过Broadcast获取客户端的消息并将其储存（写账本操作）。客户端通过Deliver接口从Order获取数据 |
| **func** (s \*server) Broadcast(srv ab.AtomicBroadcast\_BroadcastServer)  和**func** (s \*server) Deliver(srv ab.AtomicBroadcast\_DeliverServer)的实现在文件/fabric/ orderer/server.go 73 和85 行 |  |
|  |  |

|  |  |
| --- | --- |
| /fabric/ orderer/server.go 73 行 | **func** (s \*server) Broadcast(srv ab.AtomicBroadcast\_BroadcastServer)  现在分析Order节点是怎么根据用户的输入处理Block的。 |
| Server类的Broadcast(srv ab.AtomicBroadcast\_BroadcastServer)调用了  s.bh.Handle(srv)来实现功能，这个s.bh是  server类中的bh变量这个变量是一个broadcast.Handler结构体，它的handle函数的实现在fabric/orderer/common/broadcast" 76行 | **func** (bh \*handlerImpl) Handle(srv ab.AtomicBroadcast\_BroadcastServer) error  这个函数是server处理broadcast的具体实现函数，下面对他进行一步步分析。   * msg, err := srv.Recv()获取这个消息中的Envelope结构体 * payload, err := utils.UnmarshalPayload(msg.Payload)对Envelope结构体进行解码。 * chdr, err := utils.UnmarshalChannelHeader(payload.Header.ChannelHeader)解码payload中的channel头部信息 * **if** chdr.Type == int32(cb.HeaderType\_CONFIG\_UPDATE)如果接收到的头部标识是CONFIG\_UPDATE，则代表这个transaction请求时创建channel的请求-》执行创建channel的操作（这一部分需要进一步分析） * support, ok := bh.sm.GetChain(chdr.ChannelId)根据channelID获取相应的区块链 * \_, filterErr := support.Filters().Apply(msg) support.Filters()返回一个rules []Rule结构体rule结构体中一个实现函数apply()这个函数实现代码在fabric/orderer/filter/ filter.go 66行 这个语句会将不符合规则的transaction进行过滤 * **if** !support.Enqueue(msg) （这一步还没有分析清楚，没有看到它具体的实现函数） * err = srv.Send(&ab.BroadcastResponse{Status: cb.Status\_SUCCESS}) 这个语句会调用ab.AtomicBroadcast\_BroadcastServer类的send函数，这个函数会流式的向客户端返回服务端的处理信息，即处理成功/失败 |
| 这边可以分析一下Envelope  Payload 的结构体是什么样子的 | 还是没有看到server是怎么将用户的transaction转换成Block的  Broadcast.go文件中所有的接口函数实现都没有看到 |
|  |  |
| 考虑main.go中的语句  **server := NewServer(manager, signer) 之前已经分析了这里的server应该是处理广播的具体操作函数，从初始化的命名可以看出，主要负责处理block的应该是这里的manager。**  **分析一下它的定义** | manager := initializeMultiChainManager(conf, signer) |
|  |  |
| fabric/orderer/main.go 208行 | initializeMultiChainManager(conf \*config.TopLevel, signer crypto.LocalSigner) multichain.Manager   * lf, \_ := createLedgerFactory(conf) 创建管理账本的结构体 * len(lf.ChainIDs()) == 0判断是否存在channel，如果不存在则调用initializeBootstrapChannel(conf, lf)创建初始的system channel * consenters["solo"] = solo.New()根据配置创建账本的顺序梳理模式（solo 还是kafka） * **return** multichain.NewManagerImpl(lf, consenters, signer) 返回 manager |
| 其中createLedgerFactory(conf)返回的结构体的定义在  /fabric/orderer/ledger/file  /fabric/orderer/ledger/json  /fabric/orderer/ledger/ram文件夹里面factory结构体 | Manager 结构体的定义在文件/fabric/orderer/multichain/manager.go中，  // Manager coordinates the creation and access of chains  **type** Manager **interface** {      // GetChain retrieves the chain support for a chain (and whether it exists)      GetChain(chainID string) (ChainSupport, bool)      // SystemChannelID returns the channel ID for the system channel      SystemChannelID() string      // NewChannelConfig returns a bare bones configuration ready for channel      // creation request to be applied on top of it（**实际上就是返回了configTX**）      NewChannelConfig(envConfigUpdate \*cb.Envelope) (configtxapi.Manager, error)  } |
|  |  |
|  |  |
|  |  |
| /fabric/orderer/common /broadcast/broadcast.go文件是处理broadcast广播的操作函数，其中将Broadcast的内容加到chain中的代码是149行的  support.Enqueue(msg)。。其中这里的support的创建时/fabric/orderer/main.go函数中的  manager := initializeMultiChainManager(conf, signer)的manager结构体调用了它的getchain函数生成的，相应的support的Enqueue函数定义在 |  |
|  |  |
| **/fabric/orderer/server.go 73行是接受客户端Boradcast广播的入口函数。之后调用**  **/fabric/orderer/common/broadcast.go文件中的handle函数。** | 这个函数会调用**/fabric/orderer/multichain/chainsupport.go215行的Enqueue函数将msg加入chain**  **这里的chainsupport里面提供了N多接口。chainsupport：主要负责chain的操作接口，包括新增签名，数据块写入等。** |
| 这里的Enqueue函数系统会根据选择的背书策略（solo，kafka，sbft<注意，这个背书方式在代码中还没有实现，可能会在后期版本中增加>）调用相应代码里面的Enqueue实现，比如  **/fabric/orderer/solo/consensus.go** | 以order 背书方式为例，他会直接将当前的客户端的envelope类型的变量写入chain结构体，并在它的main函数中进行相应的调用，将多个envelope对应的transaction组合成block  **/fabric/orderer/solo/consensus.go 87行**  **关键语句**   * batch, committers := ch.support.BlockCutter().Cut() 调用 blockcutter处理transaction。 * block := ch.support.CreateNextBlock(batch) 生成Block * ch.support.WriteBlock(block, committers, nil) 写block |
| 其中CreateNextBlock(batch)的实现在  **fabric/orderer/ledger/util.go 50行.这个函数会根据输入的envelope值生成 block.** | 其中关键语句  **首先调用ledger的reader结构体获取当前最新的block信息**   * it, \_ := rl.Iterator(&ab.SeekPosition{   Type: &ab.SeekPosition\_Newest{  &ab.SeekNewest{},  },  写新block的blocknumber+hash+block内容   * **nextBlockNumber=block.Header.Number+1** * **previousBlockHash = block.Header.Hash()** * data := &cb.BlockData{   Data: make([][]byte, len(messages)),  }   * 然后根据这些信息创建新的block   block := **cb.NewBlock(nextBlockNumber, previousBlockHash)** |
|  |  |
| 之后  ch.support.WriteBlock(block, committers, nil)语句会将新的block写入相应的ledger比如是（file类型的还是json类型的） | 其中support是**multichain.ConsenterSupport**类型的结构体。函数实现在  **fabric/orderer/multichain/chainsupport.go 274行**  主要是添加一些相关的配置信息。  关键语句：  err := cs.ledger.Append(block)。这个语句会根据当前的ledger类型（file类型的还是json类型的）调用  **fabric/orderer/ledger/file ，fabric/orderer/ledger/json fabric/orderer/ledger/ram中的append函数进行处理.**  **blkstorage.BlockStore.AddBlock(block)** blkstorage.BlockStore.AddBlock(block) |
|  |  |
|  |  |
| 另外，order节点对创建channel的transaction处理也在**fabric/orderer/common/broadcast.go中，处理的消息类型是**CONFIG\_UPDATE 创建 channel的函数是process()<109行> 它的实现代码还没找到? | 处理完成之后会给客户端返回处理结果，返回类型是  BroadcastResponse ，他是一个status类型数据，定义如下：  /protos/common.proto中  const (  Status\_UNKNOWN Status = 0  Status\_SUCCESS Status = 200  Status\_BAD\_REQUEST Status = 400  Status\_FORBIDDEN Status = 403  Status\_NOT\_FOUND Status = 404  Status\_REQUEST\_ENTITY\_TOO\_LARGE Status = 413  Status\_INTERNAL\_SERVER\_ERROR Status = 500  Status\_SERVICE\_UNAVAILABLE Status = 503  ) |
|  |  |
|  |  |

## 另外一个和账本结构相关的重要流程是deliver类型的消息，这种消息是客户端向order节点查询消息的广播类型。当order节点完成了背书之后，会向所有节点广播ProposalResponse通知节点将新的block写入账本并更新ledger。

|  |  |
| --- | --- |
| 首先，deliver在order节点处理的入口函数在**fabric/orderer/common/deliver.go中的hanle函数里面<72行>** | * 首先 检查接受报文的合法性72-102行 * GetChain(chdr.ChannelId)获取chain * lastConfigSequence := chain.Sequence()获取.. * … * 最终如果成功返回，会返回一个成功的status和一个block |
|  |
|  |  |
| Deliver在peer节点的定义在  **fabric/peer/channel/deliverClient.go** |  |
|  |  |

## 另外一个比较重要的结构体是世界状态，它记录了当前区块链的最新状态（拥有资产数量等）

|  |
| --- |
| 1. NewServer的实现   **func** NewServer(ml multichain.Manager, signer crypto.LocalSigner) ab.AtomicBroadcastServer {      s := &server{          dh: deliver.NewHandlerImpl(deliverSupport{Manager: ml}),          bh: broadcast.NewHandlerImpl(broadcastSupport{              Manager: ml,              ConfigUpdateProcessor: configupdate.New(ml.SystemChannelID(), configUpdateSupport{Manager: ml}, signer),          }),      }  **return**  }   1. bh broadcast.Handler 数据类型的构造函数   **func** NewHandlerImpl(sm SupportManager) Handler {  **return** &handlerImpl{          sm: sm,      }  }输入是一个SupportManager类型的数据，数据类型的构造函数   1. SupportManager需要对外提供两个函数   GetChain(chainID string) (Support, bool)  Process(envConfigUpdate \*cb.Envelope) (\*cb.Envelope, error)   1. 在这里SupportManager通过子类broadcastSupport来实现，其构造函数为**type** broadcastSupport **struct** {       multichain.Manager      broadcast.ConfigUpdateProcessor  }  也就是说，标红的两个函数的实现实际上是由  Manager: ml, ConfigUpdateProcessor。实现了 |

## 考虑怎么开发监控程序?

|  |  |
| --- | --- |
| Fabric的文件在peer/main.go/ | 这里面应该有客户端连接peer节点的函数 |
| Kafka 和solo是 fabric的两种排序方式 |  |

Core kedger kv\_ledger????

**shim**

shim 包可以让 chaincode 代码跟 ledger 进行交互。

比较重要的 ChaincodeStub 结构，提供了用户 chaincode 代码跟 ledger 进行交互的一系列方法，例如 PutState 与 GetState 来写入和查询链上键值对的状态。

用户链码 --> ChaincodeStub --> Handler -----ChaincodeMessage -----> Peer

**orderer**

在 fabric 1.0 架构中，共识功能被抽取出来，作为单独的 fabric-orderer 模块来实现，完成核心的排序功能。最核心的功能是实现从客户端过来的 broadcast 请求，和从 orderer 发送到 peer 节点的 deliver 接口。同时，orderer 需要支持多 channel 的维护。

目前，orderer 模块支持三种排序类型：

* Solo：单节点的排序功能，试验性质，不具备可扩展性和容错；
* Kafka：基于 Kafka 集群的排序实现，支持可持久化和可扩展性；
* BFT：支持 BFT 容错的排序实现，尚未完成。

账本记录上支持两种类型：

* Ram：存放近期若干（默认为 1000 个）区块到内存中。
* File：存放区块记录到文件系统，默认是临时目录下的 hyperledger-fabric-ordererledger 文件。

[sampleconfig](https://github.com/yeasy/hyperledger_code_fabric/blob/master/sampleconfig/README.md) 目录下有示例的配置文件 [orderer.yaml](https://github.com/yeasy/hyperledger_code_fabric/blob/master/sampleconfig/orderer_yaml.md)。

## peerledger的结构定义在core/ledger/kvledger处

## ledger结构体的代码实现，Ledger在peer节点和order节点的实现不太一样，peer节点的Ledger由三部分组成：以文件形式组织的区块链、存储在数据库中的state，和存储在数据库中的history。Order节点的ledger只有block链，但是他的存储方式有三种，分别是file、ram、json类型。代码如下：

|  |  |
| --- | --- |
| 首先我们分析peer节点处的Ledger，他的定义代码位置在：  Fabric/Core/ledger/ledger\_interface.go 44行  它的实现代码位置在：  Fabric/Core/ledger/kvledger/kvledger.go 40行 |  |
| Kvledger定义如下：  type kvLedger struct {  ledgerID string  blockStore blkstorage.BlockStore  txtmgmt txmgr.TxMgr  historyDB historydb.HistoryDB  }  从上到下的几个参数分别是：  1 ledger的id->对应的也就是genesisblock中的id，也就是chanID。  2．Block链的结构体，在peer节点中目前只实现了以file形式存储的blcok链，也就是说order节点的ledger的file类型的实现形式是一致的。它的具体实现在  Fabric/common/ledger/blkstorage文件夹中，  其中 Fabric/common/ledger/blkstorage/blockstorge.go中是它的接口定义；  Fabric/common/ledger/blkstorage/fsblkstorage/fs\_blockstore.go是它的具体实现。  3 world state的存储结构体，保存有当前用户的资产状况world state默认使用levelDB进行存储的key-value键值，  它的具体实现在  Fabric/Core/ledger/kvledger/txmgmt/txmgr文件夹中，  其中 Fabric/Core/ledger/kvledger/txmgmt/txmgr/txmgr.go中是它的接口定义；  Fabric/Core/ledger/kvledger/txmgmt/txmgr/lockbased\_txmgr.go是它的具体实现。  4. history database 的结构体，保存有历史记录，与world state 一样。默认也是用LevelDB进行存储的。它的具体实现在  Fabric/Core/ledger/kvledger/history/historydb文件夹中，  其中 Fabric/Core/ledger/kvledger/history/historydb/historydb.go中是它的接口定义；  Fabric/Core/ledger/kvledger/history/historydb/historyleveldb/historyleveldb.go是它的具体实现,目前只有针对leveldb的实现 | 对于kvledger.go，他可以为上层提供以下几个功能。  GetBlockchainInfo() (\*common.BlockchainInfo, error)  返回区块链的基本信息  GetBlockByNumber(blockNumber uint64)  通过给定的block编号返回指定block。注：如果输入值是math.MaxUint64则返回最新的block。  GetBlocksIterator(startBlockNumber uint64) (ResultsIterator, error)  根据Block编号返回相应的位置的迭代器（指针）  Close()  关闭账本<>  Commit(block \*common.Block) error  将区块写入账本-是上层的writeblock函数的底层实现   * **以上5个方法对于order节点和peer节点的ledger都是公用的，下面是peer节点的Ledger独有的方法。**   GetTransactionByID(txID string) (\*peer.ProcessedTransaction, error)  根据交易ID获取交易信息。  GetBlockByHash(blockHash []byte) (\*common.Block, error)  通过blockhash值返回block  GetBlockByTxID(txID string) (\*common.Block, error)  //通过交易ID返回指定的Block  GetTxValidationCodeByTxID(txID string) (peer.TxValidationCode, error)  返回指定transaction的错误代码。这个错误代码标识了这个transaction是否和法并且表明了非法的原因，其定义如下：  enum TxValidationCode {  VALID = 0;  NIL\_ENVELOPE = 1;  BAD\_PAYLOAD = 2;  BAD\_COMMON\_HEADER = 3;  BAD\_CREATOR\_SIGNATURE = 4;  INVALID\_ENDORSER\_TRANSACTION = 5;  INVALID\_CONFIG\_TRANSACTION = 6;  UNSUPPORTED\_TX\_PAYLOAD = 7;  BAD\_PROPOSAL\_TXID = 8;  DUPLICATE\_TXID = 9;  ENDORSEMENT\_POLICY\_FAILURE = 10;  MVCC\_READ\_CONFLICT = 11;  PHANTOM\_READ\_CONFLICT = 12;  UNKNOWN\_TX\_TYPE = 13;  TARGET\_CHAIN\_NOT\_FOUND = 14;  MARSHAL\_TX\_ERROR = 15;  NIL\_TXACTION = 16;  EXPIRED\_CHAINCODE = 17;  CHAINCODE\_VERSION\_CONFLICT = 18;  BAD\_HEADER\_EXTENSION = 19;  BAD\_CHANNEL\_HEADER = 20;  BAD\_RESPONSE\_PAYLOAD = 21;  BAD\_RWSET = 22;  ILLEGAL\_WRITESET = 23;  INVALID\_OTHER\_REASON = 255;  }  NewTxSimulator() (TxSimulator, error)  返回一个新的`ledger.TxSimulator`结构体。它的具体实现是：  type lockBasedTxSimulator struct {  lockBasedQueryExecutor  rwsetBuilder \*rwsetutil.RWSetBuilder  }  其中lockBasedQueryExecutor是查询执行器，可以为用户提供一系列的查询服务。  RWSetBuilder 是读写集执行器，用来创建读写集  NewQueryExecutor() (QueryExecutor, error)  返回一个lockBasedQueryExecutor结构体。  它是查询执行器，可以为用户提供一系列的查询服务。也就是上一个函数中返回结构体重的第一个成员  NewHistoryQueryExecutor()(HistoryQueryExecutor, error)  返回一个针对historyDB的查询执行器  Prune(policy commonledger.PrunePolicy) error  根据给定policy返回block/transaction。。但是代码里面还没有实现。。。 |
| 另外，在peer节点里面，使用过一个叫PeerLedgerProvider的结构来对ledger进行管理的，这个结构体的接口实现在  /fabric/core/ledger/ledger\_interface。 提供了一些简单的接口函数  type PeerLedgerProvider interface {  Create(genesisBlock\*common.Block) (PeerLedger, error)  创建Legder  Open(ledgerID string) (PeerLedger, error)  根据id返回指定Ledger  Exists(ledgerID string) (bool, error)  判断指定Id的ledger是否存在  List() ([]string, error)  显示当前节点中所有ledger的id  Close()  关闭当前Provider  }  它的实现在  /fabric/core/ledger/kvledger/kv\_ledger\_provider.go |  |
|  |  |

有几个概念不清楚：

1 .什么叫history DB 里面的namespace又是什么?

2 什么叫query

3 什么叫读写集

4 txID又是什么？

|  |  |
| --- | --- |
| 然后我们开始分析order节点的ledger账本结构。它的定义文件在：  /fabric/order/ledger/file/impl.go  /fabric/order/ledger/json/impl.go  /fabric/order/ledger/ram/impl.go | 以file类型为例  type fileLedger struct {  blockStore blkstorage.BlockStore  signal chan struct{}  }  这个函数为外界提供了一些接口  **func** (fl \*fileLedger) Iterator(startPosition \*ab.SeekPosition) (ledger.Iterator, uint64)  根据peer节点发送的seekINFO信息返回对应位置的迭代器  这里的seekinfo消息是当peer节点请求order节点更新自己的chain时发送的消息，比如当invoke指令完成后，peer节点会进行commiter，这里就会发送seekinfo向order节点请求新的Block。Order节点这时候根据grpc服务返回一个DeliverResponse结构体。  它们的定义在/fabric/protos/order/ab.proto。如下：  message SeekInfo {  enum SeekBehavior {  BLOCK\_UNTIL\_READY = 0;  FAIL\_IF\_NOT\_READY = 1;  }  SeekPosition start = 1; // The position to start the deliver from  SeekPosition stop = 2; // The position to stop the deliver  SeekBehavior behavior = 3; // The behavior when a missing block is encountered  }  message DeliverResponse {  oneof Type {  common.Status status = 1;  common.Block block = 2;  }  }  **func** (fl \*fileLedger) Height() uint64  返回链的高度  **func** (fl \*fileLedger) Append(block \*cb.Block)  增加新的区块  但是在实际使用的时候往往是通过创建ReadWriter来控制order 节点的ledger。  Readwriter的定义在  fabric/order/ledger/ledger.go  它是作为ledgerfactory的一个成员被使用的。例如对于file类型的来说，factory的定义在fabric/order/ledger/file/factory.go  type fileLedgerFactory struct {  blkstorageProvider blkstorage.BlockStoreProvider  ledgers map[string]ledger.ReadWriter  mutex sync.Mutex // 互斥锁  } |
|  |  |
|  |  |
|  |  |

* 问题：seekINFO是什么

seekinfo消息是当peer节点请求order节点更新自己的chain时发送的消息，比如当invoke指令完成后，peer节点会进行commiter，这里就会发送seekinfo向order节点请求新的Block。Order节点这时候根据grpc服务返回一个DeliverResponse结构体。

Fabric/protos/peer.pb.go ProcessProposal() 函数 是commiter节点发送ProcessProposal给背书节点进行处理的函数。

Fabric/protos/peer.pb.go 98行

type EndorserServer interface {

ProcessProposal(context.Context, \*SignedProposal) (\*ProposalResponse, error)

}

这里定义了背书节点的接口！！！

他的实现在

Fabric/core/endorser/endorser.go!!!!!!! 卧槽！！找到了！！！！

## Fabric 对外提供的API接口定义在core/chaincode/shim/java文件夹

并且api底层最终还是调用peer文件夹中的代码

## 比如 对于 query 指令的数据流向。

首先调用/fabric/peer/chaincode/query.go

|  |
| --- |
| **func** chaincodeQuery(cmd \*cobra.Command, args []string, cf \*ChaincodeCmdFactory) error {  **var** err error  **if** cf == nil {          cf, err = InitCmdFactory(true, false)  **if** err != nil {  **return** err          }      }  **return** chaincodeInvokeOrQuery(cmd, args, false, cf)  } |
| cmd \*cobra.Command：表明可以用的指令，比如 query指令，invoke指令 |
| args []string： 指令相关的参数，比如当你使用query指令时，  args = []string{"-r", "-x", "-n", "example02", "-c", "{\"Args\": [\"query\",\"a\"]}"} |
| ChaincodeCmdFactory：包含签名者，与背书节点通信客户端，与order通信客户端。 |

接着调用/fabric/peer/chaincode/common.go

|  |
| --- |
| **func** chaincodeInvokeOrQuery(cmd \*cobra.Command, args []string, invoke bool, cf \*ChaincodeCmdFactory) (err error) |

进行具体操作

其关键函数是

|  |
| --- |
| proposalResp, err := ChaincodeInvokeOrQuery(  spec, 保存cmd指令的具结构体，通过cobra.Command生成  chainID,  invoke, bool  cf.Signer,  cf.EndorserClient,  cf.BroadcastClient) |
|  |
| message ChaincodeSpec {  enum Type {  UNDEFINED = 0;  GOLANG = 1;  NODE = 2;  CAR = 3;  JAVA = 4;  }  Type type = 1;  ChaincodeID chaincode\_id = 2;  ChaincodeInput input = 3;  int32 timeout = 4;  } 这个是 spec的结构体 |
|  |
|  |

上述函数的实现在：/fabric/peer/chaincode/common.go。

|  |
| --- |
| **func** ChaincodeInvokeOrQuery(      spec \*pb.ChaincodeSpec,      cID string,      invoke bool,      signer msp.SigningIdentity,      endorserClient pb.EndorserClient,      bc common.BroadcastClient,  ) (\*pb.ProposalResponse, error) |

这个函数的返回值是相应的chaincode执行结果，由背书节点返回。这个函数有两个分支，如果对应的cli端指令是quey，则只向endoser节点发送数据并接受endoser返回的执行结果。如果对应的cli端指令是invoke，则还要想order节点进行广播，提醒Order节点更新账本。其中发送给endorser的结构体是\*pb.SignedProposal。这个结构体是对pb.Proposal进行签名后的结构体，pb.Proposal定义在/fabric/protos/peer/proposal.proto

|  |
| --- |
| message Proposal {  // The header of the proposal. It is the bytes of the Header  bytes header = 1;  // The payload of the proposal as defined by the type in the proposal  // header.  bytes payload = 2;  // Optional extensions to the proposal. Its content depends on the Header's  // type field. For the type CHAINCODE, it might be the bytes of a  // ChaincodeAction message.  bytes extension = 3;  } |

发送给order节点的结构体是Envelope。定义文件是

/fabric/protos/common/common.proto

|  |
| --- |
| message Envelope {  // A marshaled Payload  bytes payload = 1;  // A signature by the creator specified in the Payload header  bytes signature = 2;  } |

chaincodeInvokeOrQuery()关键代码如下：

|  |
| --- |
| putils.CreateProposalFromCIS(pcommon.HeaderType\_ENDORSER\_TRANSACTION, cID, , creator) |

根据以 HeaderType\_ENDORSER\_TRAN的SACTION类型的头部（ENDORSER\_TRANSACTION）、指定的channelID（cID,）、指定的chaincode相关信息与用户加密ID的算法（invocation）、指定的签名结构体（creator）来生成要发送给endorser节点的Proposal。

|  |
| --- |
| invocation := &pb.ChaincodeInvocationSpec{ChaincodeSpec: spec} |

这条语句生成了invocation，它是一个ChaincodeInvocationSpec结构体，定义在：

/fabric/protos/peer/chaincode.proto 如下：

|  |
| --- |
| message ChaincodeInvocationSpec {  ChaincodeSpec chaincode\_spec = 1;  // This field can contain a user-specified ID generation algorithm  // If supplied, this will be used to generate a ID  // If not supplied (left empty), sha256base64 will be used  // The algorithm consists of two parts:  // 1, a hash function  // 2, a decoding used to decode user (string) input to bytes  // Currently, SHA256 with BASE64 is supported (e.g. idGenerationAlg='sha256base64')  string id\_generation\_alg = 2;  } |

第一个成员是pb.ChaincodeSpec，只是在后面加了一个标识用来表示当前加密用户ID的算法，默认使用的是sha256base64算法。

|  |
| --- |
| creator, err := signer.Serialize() |

这条语句生成了签名器（还是签名？？？？？）

CreateProposalFromCIS()的代码实现在/fabric/protos/utils/protoutils.go

首先，他会调用creator,生成本次transation的交易ID,代码如下

|  |
| --- |
| nonce, err := crypto.GetRandomNonce()  txid, err := ComputeProposalTxID(nonce, creator) |

然后调用

|  |
| --- |
| **func** CreateChaincodeProposalWithTxIDNonceAndTransient(txid string, typ common.HeaderType, chainID string, cis \*peer.ChaincodeInvocationSpec, nonce, creator []byte, transientMap **map**[string][]byte) |

这个函数首先根据chaincodeID生成proposal的头部

|  |
| --- |
| hdr := &common.Header{ChannelHeader: MarshalOrPanic(&common.ChannelHeader{  Type: int32(typ),  TxId: txid,  Timestamp: timestamp,  ChannelId: chainID,  Extension: ccHdrExtBytes,  Epoch: epoch}),  SignatureHeader: MarshalOrPanic(&common.SignatureHeader{Nonce: nonce, Creator: creator})}  hdrBytes, err := proto.Marshal(hdr) |

其中typ是输入的proposal类型，这里是HeaderType\_ENDORSER\_TRAN。Txid是输入的transactionID，通过签名器creator生成。chainID是输入的channelID。

ccHdrExtBytes,是通过输入的peer.ChaincodeInvocationSpec生成的，其处理函数如下

|  |
| --- |
| ccHdrExt := &peer.ChaincodeHeaderExtension{ChaincodeId: cis.ChaincodeSpec.ChaincodeId}      ccHdrExtBytes, err := proto.Marshal(ccHdrExt) |

这个ChaincodeHeaderExtension的定义在

|  |
| --- |
| message ChaincodeHeaderExtension {  // The PayloadVisibility field controls to what extent the Proposal's payload  // (recall that for the type CHAINCODE, it is ChaincodeProposalPayload  // message) field will be visible in the final transaction and in the ledger.  // Ideally, it would be configurable, supporting at least 3 main visibility  // modes:  // 1. all bytes of the payload are visible;  // 2. only a hash of the payload is visible;  // 3. nothing is visible.  // Notice that the visibility function may be potentially part of the ESCC.  // In that case it overrides PayloadVisibility field. Finally notice that  // this field impacts the content of ProposalResponsePayload.proposalHash.  bytes payload\_visibility = 1;  // The ID of the chaincode to target.  ChaincodeID chaincode\_id = 2;  } |

包含两个部分，第一个是chaincodeID 第二个是chaincode的有效负载量。

然后会生成proposal的payload

|  |
| --- |
| ccPropPayload := &peer.ChaincodeProposalPayload{Input: cisBytes, TransientMap: transientMap} |

其中 &peer.ChaincodeProposalPayload的结构如下

|  |
| --- |
| message ChaincodeProposalPayload {      // Input contains the arguments for this invocation. If this invocation      // deploys a new chaincode, ESCC/VSCC are part of this field.     &nbspbytes input = 1;      // TransientMap contains data (e.g. cryptographic material) that might be used      // to implement some form of application-level confidentiality. The contents      // of this field are supposed to always be omitted from the transaction and      // excluded from the ledger.     &nbspmap<string, bytes> TransientMap = 2;  } |

也就是说，proposal的payload只包含了chaincode指令的相关参数 比如-c –N。

头部才包含了那些关键信息，比如proposal类型，transactionID，时戳，channelID。

完成了这一串逻辑之后，函数返回到了/fabric/peer/chaincode/query.go的ChaincodeInvokeOrQuery处，对Proposal进行加密。

然后调用endorserClient将数据发送给endorser节点，由endorser返回pb.ProposalResponse

Endorser处理这个Proposal的代码在Fabric/core/endorser/endorser.go的

ProcessProposal()函数，首先会检验当前Proposal的合法性（验证签名），返回三个值：  
proposal、proposal.head、proposal.head. Extension

随后，返回函数返回到了/fabric/peer/chaincode/query.go的ChaincodeInvokeOrQuery处，如果检测到是query指令，则返回相关信息，如果是invoke指令，还要将prorosalRsp指令包装成Envelope的数据结构并发送broadcast指令给order节点。

Envelope结果体的定义在 /fabric/protos/common/common.proto

|  |
| --- |
| message Envelope {  // A marshaled Payload  bytes payload = 1;  // A signature by the creator specified in the Payload header  bytes signature = 2;  } |

其中的重要结构体为payload，

当order节点接受到了commiter节点发送的envelope数据包后，会调用utils.UnmarshalPayload(Envelope.Payload)来序列化envelope的Payload成员，返回一个cb.Payload结构体，代码的位置在/fabric/protos/utils /commonutils.go。返回的payload结构体定义在/fabric/protos/common/common.go

如下：

|  |
| --- |
| message Payload {  // Header is included to provide identity and prevent replay  Header header = 1;  // Data, the encoding of which is defined by the type in the header  bytes data = 2;  } |

其中header成员变量包含

|  |
| --- |
| message Header {  bytes channel\_header = 1;  bytes signature\_header = 2;  } |

|  |
| --- |
| message SignatureHeader {  // Creator of the message, specified as a certificate chain  bytes creator = 1;  // Arbitrary number that may only be used once. Can be used to detect replay attacks.  bytes nonce = 2;  } |

|  |
| --- |
| message ChannelHeader {  int32 type = 1; // Header types 0-10000 are reserved and defined by HeaderType  // Version indicates message protocol version  int32 version = 2;  // Timestamp is the local time when the message was created  // by the sender  google.protobuf.Timestamp timestamp = 3;  // Identifier of the channel this message is bound for  string channel\_id = 4;  // An unique identifier that is used end-to-end.  // - set by higher layers such as end user or SDK  // - passed to the endorser (which will check for uniqueness)  // - as the header is passed along unchanged, it will be  // be retrieved by the committer (uniqueness check here as well)  // - to be stored in the ledger  string tx\_id = 5;  // The epoch in which this header was generated, where epoch is defined based on block height  // Epoch in which the response has been generated. This field identifies a  // logical window of time. A proposal response is accepted by a peer only if  // two conditions hold:  // 1. the epoch specified in the message is the current epoch  // 2. this message has been only seen once during this epoch (i.e. it hasn't  // been replayed)  uint64 epoch = 6;  // Extension that may be attached based on the header type  bytes extension = 7;  } |

到此位置，从用户输入->order节点写入账本的流程已经分析完毕了

## 下一步可以分析从order节点->peer节点写入账本的流程

Ps:完整的函数调用可以参考http://www.jianshu.com/p/7d39047684cc

当order节点成功将Block写入他本地的Ledger中之后，他会首先会向peer节点发送相应的提示信息。例如，当新的block成功写入后，他会发送广播Status\_SUCCESS类型的信息，调用函数如下：

|  |
| --- |
| /fabric/order/broadcast/broadcast.go |
| err = srv.Send(&ab.BroadcastResponse{Status: cb.Status\_SUCCESS}) |

当peer节点接受到相应的提示信息后，会向order节点发送seekInfo信息（peer节点接受提示信息并发送seekinfo的代码还没找到）。

之后，order节点的deliverServer监听到了peer节点发送的seekINfo请求之后，首先判断消息的合法性，如果不合法，则调用sendStatusReply(srv, cb.status)返回相应的提示信息。如果合法，则调用sendBlockReply(srv, block)+ sendStatusReply(srv, cb.Status\_SUCCESS)返回相应的status和对应的block

调用函数如下：

|  |
| --- |
| /fabric/orderer/deliver/deliver.go |
| Handle() |

当peer节点接受到从Order节点发来的deliver信息后，会调用自身的deliverclient进行处理，调用函数DeliverBlocks，同样先对接收到类型进行检查，如果正确则会先更新自己的world state，然后将Block发送给其他peer节点。

调用代码如下：

|  |
| --- |
| /fabric/core/deliverservice/blockprovider/blockprovider.go |
| DeliverBlocks() |

其中关键代码有：

|  |
| --- |
| // Add payload to local state payloads buffer              b.gossip.AddPayload(b.chainID, payload) |
| // Gossip messages with other nodes  b.gossip.Gossip(gossipMsg) |

之后，peer节点会调用相应，kv\_ledger即的账本结构写账本