# Fabric 1.0——chancode执行流程源码分析

## 总体流程

1. 客户端的Peer节点创建一个交易请求（请求执行chaincode），然后把交易请求与相关信息发送给Endorser节点。
2. Endorser节点接受交易请求并判断交易请求的和发行后，会在在本地模拟执行chaincode，并将结果返回个Peer节点。（这个过程就是我们常说的“背书”）
3. Peer节点等待Endorser节点的返回结果，当接受到了指定数量的Endorser节点返回值，如果交易是不需要写入账本（query指令）则直接返回结果，否则（Invoke指令），启动本地的Broadcast客户端向Orderer节点发送Broadcast广播，请求Orderer节点进行共识写账本操作。
4. Orderer节点通过系统设定的共识算法（SOLO模式、kafka模式）进行共识并将结果写入本地账本，完成后向Peer节点发送信号，提醒Peer节点ordering操作结果。
5. peer 在Ordering结束之后会执行 commiting ,即写入stateDB操作。
6. 如果Peer节点希望同步账本，则会启动本地的Deliver客户端向Orderer节点发送Seek\_INFO类型指令，请求相应的区块数据。
7. Orderer节点接受到了Peer节点的Seek\_INFO，执行本地的账本查询功能比返回相应的Block。并通过调用Deliver服务端将Block发送给Peer节点。
8. Peer节点调用本地Deliver客户端区块数据，并写入本地账本。

## 代码分析

现在已Query/Invoke指令为例对代码进行分析，其中Query指令只涉及账本查询的chaincode执行，不会写入新的区块，因此只涉及上述1-3步。Invoke指令则会涉及上述1-7的所有步骤。

### 步骤一：创建交易请求，并发送给Endorser节点



图 0—1 函数调用关系图

Cilent想要发起交易，入口函数是 ChaincodeQuery/ChaincodeInvoke 。

|  |  |
| --- | --- |
| 路径 | /fabric/peer/common.go |
| 函数原型 | ChaincodeQuery/ChaincodeInvoke |
| 输入 | * cobra.Command：用户输入的cmd指令，例如Query指令，Invoke指令 * []string：用输入的指令参数，比如Query指令就有对应的 –n –w 参数 * \*ChaincodeCmdFactory：包含执行chaincode所需的结构体，包括EndorserClient，负责向Endorser节点发送请求；BroadcastClient，负责向order节点发送请求；Signer，负责为请求签名。 |
| 输出 | Error：用来判断操作成果/失败 |

这两个函数都会调用同一个函数chaincodeInvokeOrQuery，这个函数负责具体的代码，其代码实现是通过重载自身，调用chaincodeInvokeOrQuery’。

|  |  |
| --- | --- |
| 路径 | /fabric/peer/chaincode/common |
| 函数原型 | chaincodeInvokeOrQuery |
| 输入 | * pb.ChaincodeSpec：用户输入的指令，包含了指令类型和相关参数 * cID string：当前请求所执行的channelID * invoke bool：如果是invoke指令则为真，否则为假 * msp.SigningIdentity：从ChaincodeCmdFactory提取出来，负责签名 * pb.EndorserClient：负责向Endorser节点发送请求 * common.BroadcastClient：负责向order节点发送请求 |
| 输出 | pb.ProposalResponse 背书节点执行chaincode的返回结果 |

其中pb.ProposalResponse是关键结构体，包含了背书节点的执行结果。

|  |  |
| --- | --- |
| 路径 | /fabric/protos/peer/proposal\_response.proto |
| 结构体定义 | message ProposalResponse {  int32 version = 1;  google.protobuf.Timestamp timestamp = 2;  Response response = 4;  bytes payload = 5;  Endorsement endorsement = 6;  } |
| 说明 | * version：当前数据包的版本 * timestamp：时戳 * response：用来判断背书是否成功 * payload： 背书的有效负载 * endorsement：endorsermen对payload的签名 |

在函数chaincodeInvokeOrQuery中首先调用CreateProposalFromCIS生成发送给Endorser的交易请求，随后对请求进行签名，并调用endorserClient.ProcessProposal将签名后的SignedProposal结构体发送给Endorser背书。

|  |  |
| --- | --- |
| 路径 | /fabric/protos/peer/proputils.go |
| 函数原型 | CreateProposalFromCIS |
| 输入 | * common.HeaderType：Proposal的类型，在这里是ENDORSER\_TRANSACTION，代表通过SDK生成的发送给Endorser节点的请求。 * chainID string：channelID * peer.ChaincodeInvocationSpec：提供chaincode的执行空间。包含<Type,ChaincodeId,input>分别对应 chaincode的语言（比如JAVA,GO）；chaincodeID；指令输入参数 * Creator：用于签名的结构体 |
| 输出 | peer.Proposal：发送给Endorser节点的请求 |

其中peer.Proposal是关键结构体，是发送给背书节点的请求

|  |  |
| --- | --- |
| 路径 | /fabric/protos/peer/proposal.proto |
| 结构体定义 | message Proposal {  bytes header = 1;  bytes payload = 2;  bytes extension = 3;  } |
| 说明 | * header：头部，包含< ChannelHeader , SignatureHeader > ，前者主要包含了TransactionID，时戳，channelID,后者包含了签名的相关信息。 * payload：请求的有效负载，主要包含 chaincode的语言（比如JAVA,GO）；chaincodeID；指令输入参数 * extension：**暂不明（可选参数）** |

endorserClient.ProcessProposal会将propose进行加密并发送给Endorser节点。

### 步骤二：Endorser节点接受proposal，本地执行chaincode。如果需要（invoke指令），向orderer发送Boradcast报文请求将交易写入账本



图 0—2函数调用关系图

Endorser节点接受proposal的入口函数是ProcessProposal()。

|  |  |
| --- | --- |
| 路径 | Fabric/core/endorser/endorser.go |
| 函数原型 | ProcessProposal() |
| 输入 | * context.Context：Proposal的类型，在这里是ENDORSER\_TRANSACTION，代表通过SDK生成的发送给Endorser节点的请求。 * pb.SignedProposal：加密后的proposal 主要包括签名后的数据SignedProposal . proposal\_bytes；签名SignedProposal . signature； |
| 输出 | pb.ProposalResponse：返回个peer节点的背书结果 |

首选调用函数simulateProposal()进行chaincode的模拟执行，如果执行成功，则调用endorseProposal()进行背书，并将编码好的proposalResponse返回给Peer节点。

其中是返回值pb.ProposalResponse=<version , response , payload, endorsement >。重点payload和response的生成。**这部分还有点模糊，需要进一步读下去。相关代码如下：**

|  |
| --- |
| func (e \*Endorser) ProcessProposal(ctx context.Context, signedProp \*pb.SignedProposal) (\*pb.ProposalResponse, error) {  **//判断当前的proposal是否合法**  prop, hdr, hdrExt, err := validation.ValidateProposalMessage(signedProp)    ……  **//首先 模拟执行chaincode并返回执行结果(res)**  cd, res, simulationResult, ccevent, err := e.simulateProposal(ctx, chainID, txid, signedProp, prop, hdrExt.ChaincodeId, txsim)  ……  **//2 –执行背书（将执行结果编码成proposalResponse，并进行签名）** pResp, err = e.endorseProposal(ctx, chainID, txid, signedProp, prop, res, simulationResult, ccevent, hdrExt.PayloadVisibility, hdrExt.ChaincodeId, txsim, cd)  ……  pResp.Response.Payload = res.Payload  return pResp, nil  } |

在返回了ProposalResponse之后，会返回函数ChaincodeInvokeOrQuery。该函数会判断当前指令的类型，如果是invoke，调用函数：

env, err := putils.CreateSignedTx(prop, signer, proposalResp) 将Proposal转换成发送给orderer节点的envelop结构体，并调用common.BroadcastClient 结构体将envelop发送给orderer。

相应代码如下：

|  |
| --- |
| if invoke {  if proposalResp != nil {  if proposalResp.Response.Status >= shim.ERROR {  return proposalResp, nil  }  // assemble a signed transaction (it's an Envelope message)  **env, err := putils.CreateSignedTx(prop, signer, proposalResp)**  if err != nil {  return proposalResp, fmt.Errorf("Could not assemble transaction, err %s", err)  }  // send the envelope for ordering  **if err = bc.Send(env);** err != nil {  return proposalResp, fmt.Errorf("Error sending transaction %s: %s", funcName, err)  }  }  } |

这里的env是一个关键的数据结构，它是一个common.Envelope结构体，是peer节点与order节点进行交互的数据结构。

|  |  |
| --- | --- |
| 路径 | /fabric/protos/common/common.proto |
| 结构体定义 | message Envelope {  // A marshaled Payload  bytes payload = 1;  // A signature by the creator specified in the Payload header  bytes signature = 2;  } |
| 说明 | 主要内容是proposalResponse进行编码后的结果，其内容是在ProposalResponse内容基础上加上签名。 |

### 步骤三：orderer节点接受Broadcast类型的广播，并通过peer节点发送的envelop数据写账本



图 0—3函数调用关系图

Orderer节点处理boradcast广播的入口函数是Handle()函数，。

|  |  |
| --- | --- |
| 路径 | Fabric/orderer/common/broadcast/broadcast.go |
| 函数原型 | Handle() |
| 输入 | * ab.AtomicBroadcast\_BroadcastServer：broadcast的server端结构体，其底层是通过Golang的grpc实现的。用于作为处理broadcast信号的服务端。这个server的启动函数在Fabric/orderer/main.go |
| 输出 | error：处理结果 |

这个函数首先调用server的Recv()函数接受由Peer发送的Envelope结构体。之后对Envelope进行解码。最后将这个交易内容放入缓冲队列，等待共识写入Block并向peer节点返回处理成功的Broadcast消息。

PS:这个函数同时也会处理peer节点发送的创建channel的请求，此时envelope的头部会以HeaderType\_CONFIG\_UPDATE 类型进行标识，调用相关模块进行处理：

|  |
| --- |
| **func** (bh \*handlerImpl) Handle(srv ab.AtomicBroadcast\_BroadcastServer) error {  for {  msg, err := srv.Recv() **//接收数据包**  ……  payload, err := utils.UnmarshalPayload(msg.Payload)  **//解码**  …….  chdr, err := utils.UnmarshalChannelHeader(payload.Header.ChannelHeader)**//头部解码**  ……  chdr.Type == int32(cb.HeaderType\_CONFIG\_UPDATE){ **//处理创建channel请求**  msg, err = bh.sm.Process(msg)  }    support, ok := bh.sm.GetChain(chdr.ChannelId) **//获取相应的channel**  **if** !support.Enqueue(msg){} **//将消息插入队列**  ……  err = srv.Send(&ab.BroadcastResponse{Status: cb.Status\_SUCCESS}) **//返回广播** |

也就是说，每个transaction的数据内容实际上就是ProposalResponse里面的内容。



图 0—4 函数调用关系图

当order节点接受到了足够的transaction记录之后，他会调用相应的共识模块进行写账本操作，比如，如果采用solo的共识方式写账本的话，它的入口函数就是main函数。如下：

|  |  |
| --- | --- |
| 路径 | Fabric/orderer/solo/consensus.go |
| 函数原型 | main() |
| 输入 | 无 |
| 输出 | 无 |

这个函数首先会调用ch.support.BlockCutter().Ordered(msg)对队列中的交易记录进行排序，随后执行block := ch.support.CreateNextBlock(batch)创建新的block，并通过ch.support.WriteBlock(block, committers, nil)将交易记录写入block中。代码如下：

|  |
| --- |
| func (ch \*chain) main() {  var timer <-chan time.Time  for {  select {  case msg := <-ch.sendChan:  /**/对缓冲队列中的交易记录排序**  batches, committers, ok := ch.support.BlockCutter().Ordered(msg)  ……  batch, committers := ch.support.BlockCutter().Cut()  ……  **//创建新的Block**  block := ch.support.CreateNextBlock(batch)  **//写block**  ch.support.WriteBlock(block, committers, nil)  }  }  } |

其中，Block是这部分代码的核心结构体，他是fabric 1.0中保存区块的结构体，它的定义如下：

|  |  |
| --- | --- |
| 路径 | /fabric/protos/common/common.proto |
| 结构体定义 | message Block {  BlockHeader header = 1;  BlockData data = 2;  BlockMetadata metadata = 3;  } |
| 说明 | blockHeader=< number, previous\_hash, data\_hash>分别表示区块号，前一区块hash，当前区块hash  BlockData里面包含的transaction的内容。  BlockMetadata是一个数组结构，数组长度为4  BlockMetadata[0,1,2,3]=< 块签名，区块链的config区块的区块号，用来过滤非法交易的过滤器，共识相关的配置数据> |

### 步骤四： Orderer节点接受Peer节点的Seek\_Info请求发送相应的block帮助peer节点更新本地账本记录



图 0—5函数调用关系图

Order节点处理peer节点发送的Seek\_Info请求的入口函数是deliverServer. Handle()函数如下：

|  |  |
| --- | --- |
| 路径 | /Fabric/orderer/deliver/deliver.go |
| 函数原型 | Handle() |
| 输入 | * AtomicBroadcast\_DeliverServer：deliver服务的server端结构体，其底层是通过Golang的grpc实现的。用于作为处理deliver信号的服务端。 |
| 输出 | Error：操作结果 |

order节点的deliverServer接受到了peer节点发送的seek\_Info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)返回相应的错误提示信息。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。代码如下：

|  |
| --- |
| func (ds \*deliverServer) Handle(srv ab.AtomicBroadcast\_DeliverServer) error {  **//接受Peer节点的seek\_Info请求**  envelope, err := srv.Recv()；  ……  **//数据解码**  payload, err := utils.UnmarshalPayload(envelope.Payload)  ……  **//头部解码**  chdr, err := utils.UnmarshalChannelHeader(payload.Header.ChannelHeader)  **//获得指定的channel对应的Ledger结构**  chain, ok := ds.sm.GetChain(chdr.ChannelId)  ……  **//与身份验证相关的操作**  sf := sigfilter.New(policies.ChannelReaders, chain.PolicyManager())  result, \_ := sf.Apply(envelope)  err = proto.Unmarshal(payload.Data, seekInfo)  **//获取Seek\_Info指定的第一个Block**  cursor, number := chain.Reader().Iterator(seekInfo.Start)      **//顺序获取Seek\_Info指定的Block**  block, status := cursor.Next()  **//成功返回Peer节点的Seek\_Info请求**  **if** err := sendBlockReply(srv, block);  ……  **if** err := sendStatusReply(srv, cb.Status\_SUCCESS); |

其中sendBlockReply()和sendStatusReply返回的结构体都是ab.DeliverResponse。相关代码如下：

|  |  |
| --- | --- |
| 路径 | /fabric/protos/order/ab.proto |
| 结构体定义 | message DeliverResponse {  oneof Type {  common.Status status = 1;  common.Block block = 2;  }  } |
| 说明 | common.Status表明了当前Orderer节点的处理结果，如果处理成功返回Status\_SUCCESS。  common.Block是Orderer节点成功返回的block结构体。  这两个内容对于一个DeliverResponse只存在一个。 |

在Peer处会有相应的deliverclient结构体接受从Order节点发来的deliver信息，其入口函数是DeliverBlocks()。代码如下：

|  |  |
| --- | --- |
| 路径 | /fabric/core/deliverservice/blockprovider/blockprovider.go |
| 函数原型 | DeliverBlocks() |
| 输入 | * AtomicBroadcast\_DeliverServer：deliver服务的server端结构体，其底层是通过Golang的grpc实现的。用于作为处理deliver信号的服务端。 |
| 输出 | Error：操作结果 |



图 0—6函数调用关系图

Peer先对接收到类型进行检查，如果正确则会执行更新world state，写入Ledger，更新historyDB等操作。然后将Block发送给其他peer节点。具体代码如下：

|  |
| --- |
| func (b \*blocksProviderImpl) DeliverBlocks() {  **//接受Orderer节点发送的DeliverResponse**  msg, err := b.client.Recv()  ……  **//针对DeliverResponse的两种类型分别进行处理**  switch t := msg.Type.(type) {  **//如果是返回的response中是status类型变量，返回相应的提示信息**  case \*orderer.DeliverResponse\_Status:  ……  **//如果是返回的response中是block类型变量，更新相应数据结构并向其他Peer节点广播信息**  case \*orderer.DeliverResponse\_Block:{  **//区块号**  seqNum := t.Block.Header.Number  ……  **//区块结构体**  marshaledBlock, err := proto.Marshal(t.Block)；  ……  **//将这个block结构体进行编码，便于后期通过Gossip在peer节点之间同步这个Block**  payload := createPayload(seqNum, marshaledBlock);  …….  gossipMsg := createGossipMsg(b.chainID, payload)  **//将当前Block形成的payload加入本地的gossip payload中，从而更新本地的账本相关信息**  b.gossip.AddPayload(b.chainID, payload)  //发送gossip信息  b.gossip.Gossip(gossipMsg)    } |

其中gossip负责将Block写入Peer节点的本地账本中，通过gossip.AddPayload将block将入本节点的账本中，同时通过gossip.Gossip(gossipMsg)将Block同步到其他peer节点的账本中。