**Fabric交易源码分析**

付嘉欣 3190105562

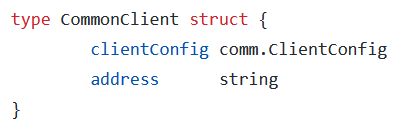
在用git工具将<https://github.com/hyperledger/fabric.git>克隆到本地时，可能由于项目较大的原因，总是出现各种各样的问题，所以索性在Github上面分析，截图来自于github，而非本地项目。

Fabric的项目代码个人感觉比较分散，没有像以太坊源码那样不同文件各司其职（也可能是我对源码还不够熟悉的缘故），所以基于交易流程分析。

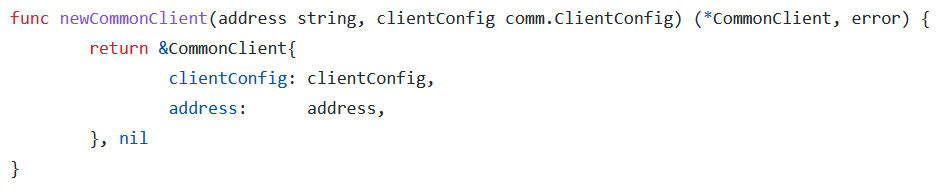
Fabric交易从产生到记入账本大致分为四个部分：（1）客户端向背书节点发送交易提案请求（2）背书节点对交易提案进行签名背书并将结果返回（3）客户端向排序服务提交交易（4）排序服务节点生成区块。我根据这四个部分的顺序依次进行分析。

1. 客户端向背书节点发送交易提案请求

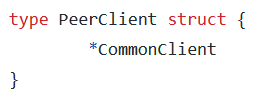
首先要得到一个Endorser客户端。internal/peer/common/common.go中这样定义一个普通客户端结构体：



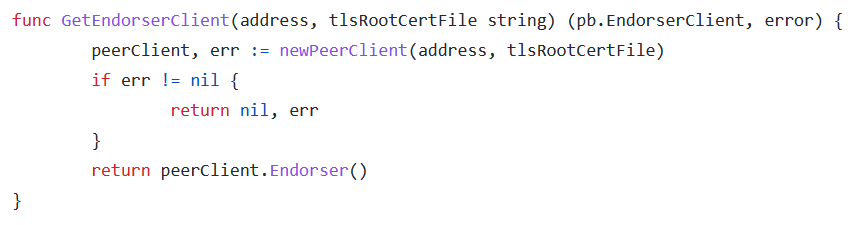
newCommonClient函数用来返回一个根据地址和配置参数创建的普通客户端：



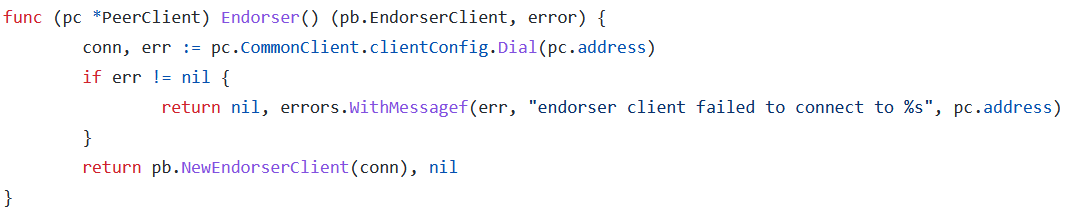
在internal/peer/common/peerclient.go中定义了PeerClient结构体，可以看到其实际结构与CommonClient一致：

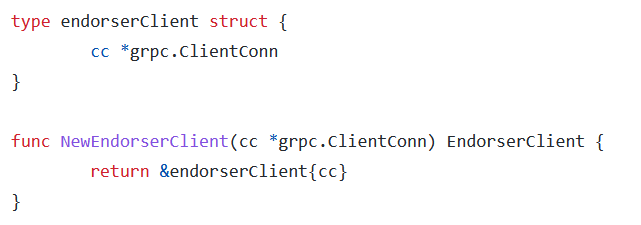


peerclient.go文件中定义的GetEndorserClient函数调用newPeerClient函数得到一个新的客户端peerClient，并且根据该客户端中的Endorser方法返回了一个Endorser客户端：

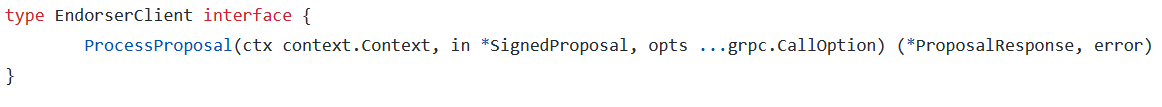


Endorser方法又调用了处于vendor/github.com/fabric-protos-go/peer/peer.pb.go文件中的NewEndorserClient函数，它最终返回了一个ClinetConn连接，意味着一个Endorser客户端建立成功：

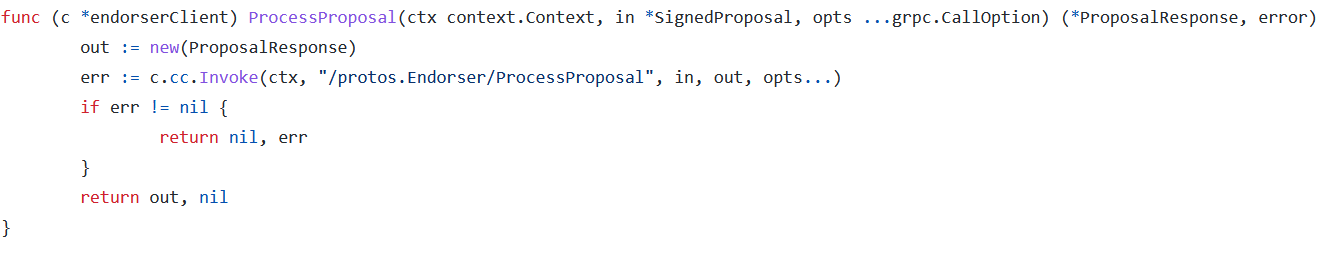




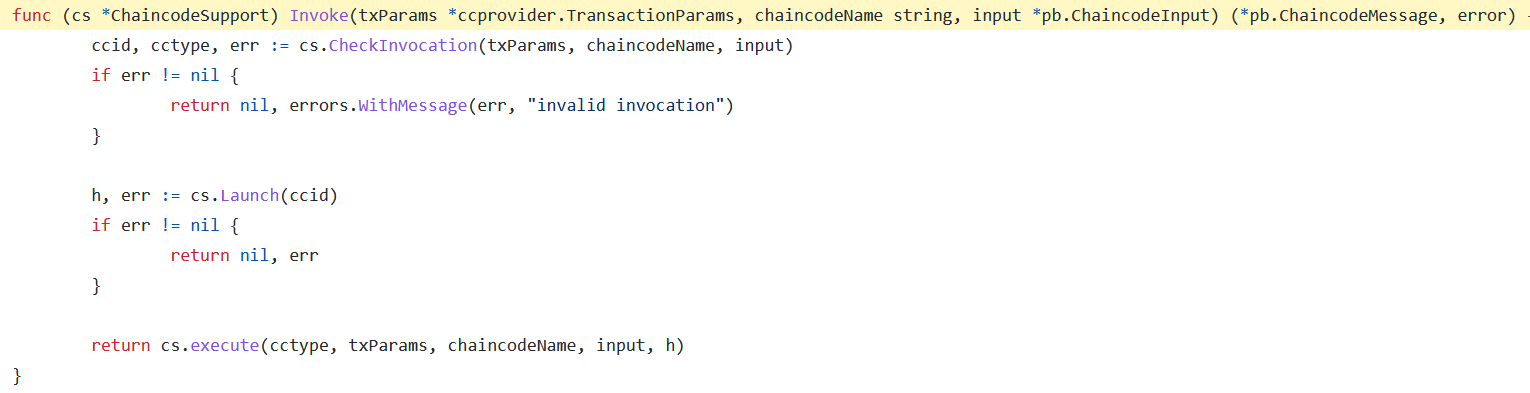
EndorserClient是为了背书服务而定义的客户端接口。可以看到peer.pb.go文件中有关于EndorserClient接口的定义：



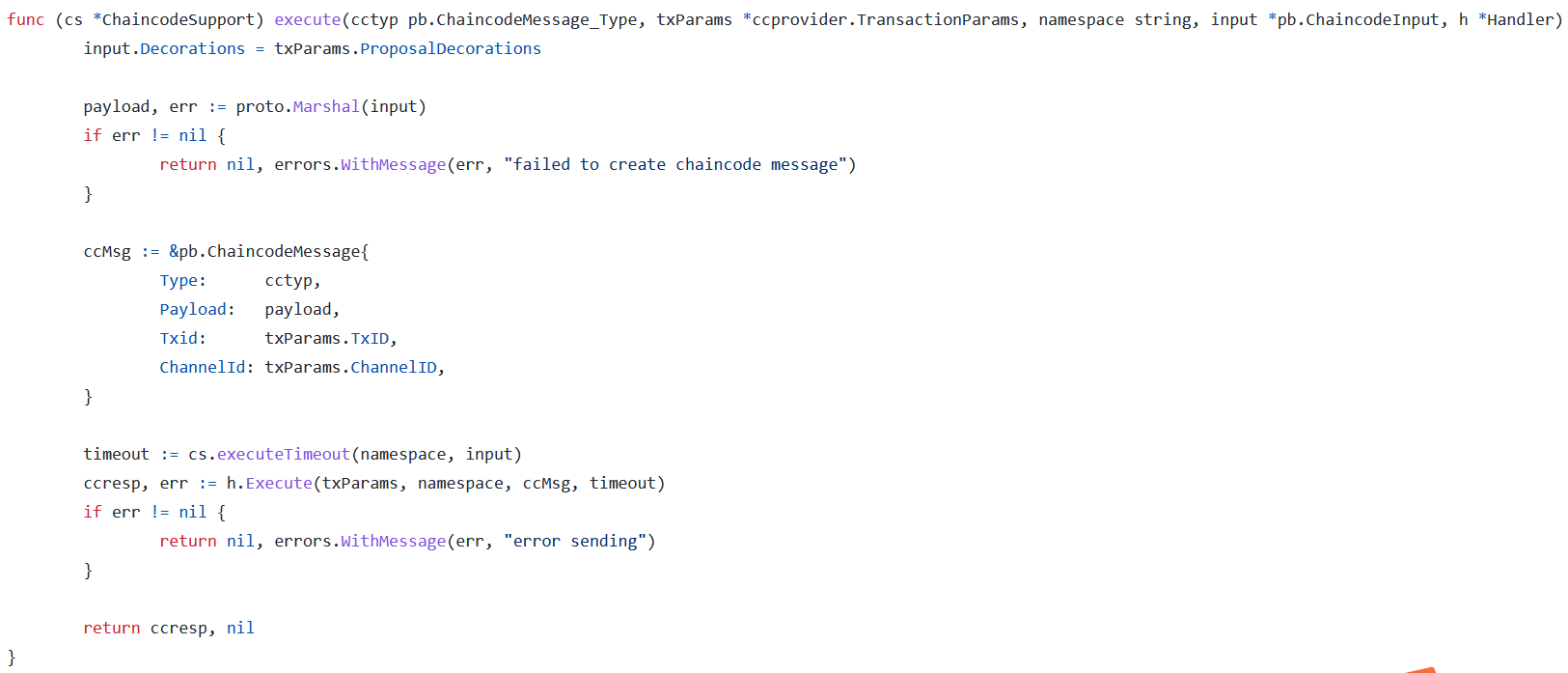
这个接口调用了ProcessProposal函数，用来执行交易中的智能合约。ProcessProposal函数新建了一个响应体后，将交易参数ctx，智能合约名称和合约输入作为参数传递给core/chaincode/chaincode\_support.go文件中的函数Invoke：



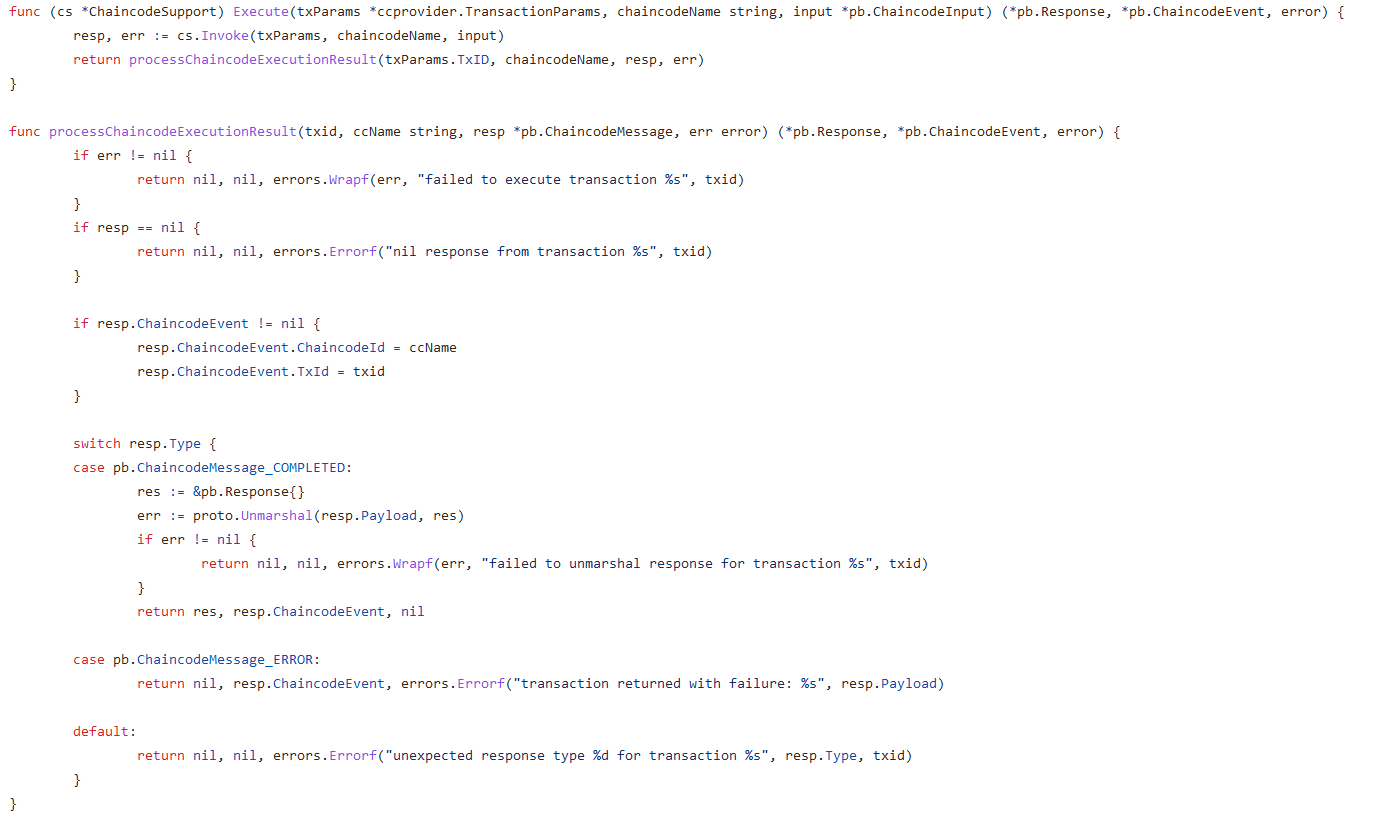
Invoke函数则调用CheckInvocation函数得到了智能合约的ID和类型，并在检查了各参数的合法性之后将execute函数执行的结果（即正确或错误信息）返回到ProcessProposal函数中定义的err变量：



而execute函数则主要对智能合约相关信息（Type, Payload, Txid, ChannelId）做了说明并将环境参数、合约名称、合约相关信息作为参数传递给Execute函数使之调用智能合约，并传递了timeout参数作为执行是否超时的接口：

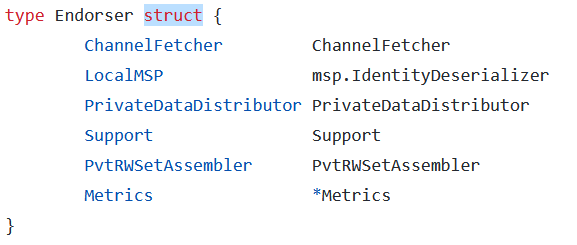


Execute函数调用了processChaincodeExecutionResult函数并返回最初的响应体，此时Invoke调用结束，ProcessProposal中的err变量得到了智能合约的执行结果，ProcessProposal返回结果并结束。



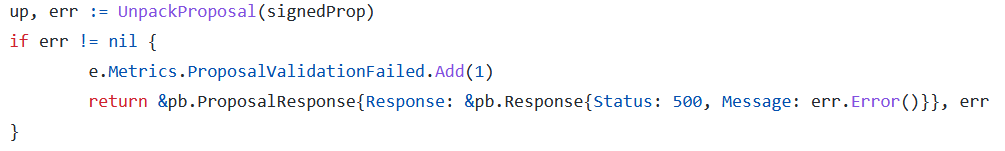
1. 背书节点对交易提案进行签名背书并将结果返回

与EndorserClient类似地，在core/endorser/endorser.go文件中有Endorser结构体的定义：

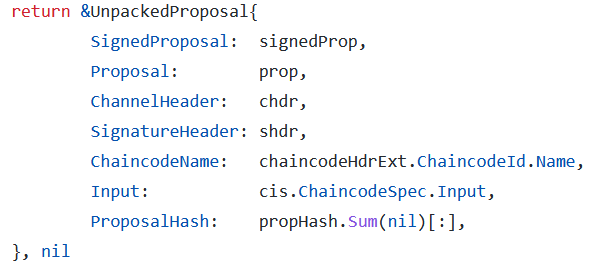


在该文件中存在ProcessProposal函数，作为Endorser.go文件中最重要的接口。由于ProcessProposal函数代码量比较大，所以将其划分成不同的部分进行分析。

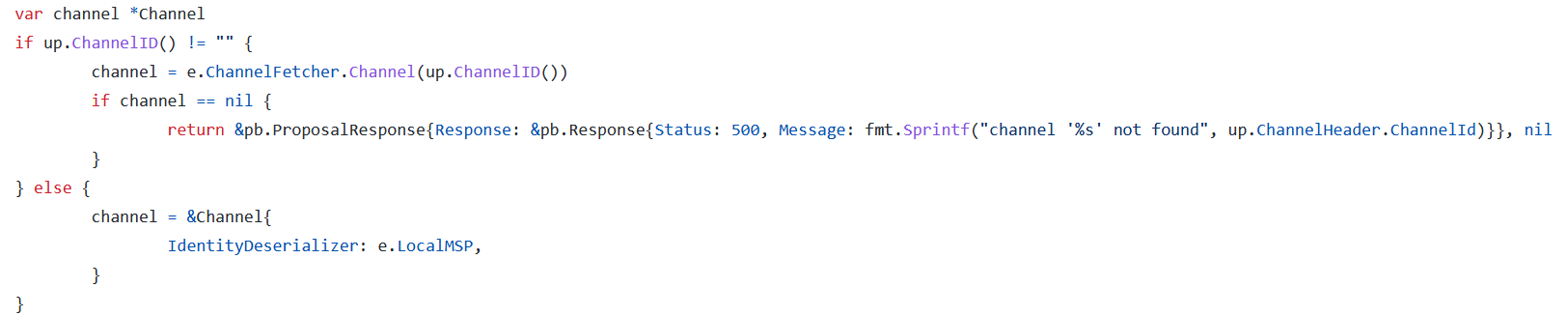
首先，调用了core/endorser/msgvalidation.go文件中的UnpackProposal函数检查能否将传进来的交易提案参数singnedProp进行解包：



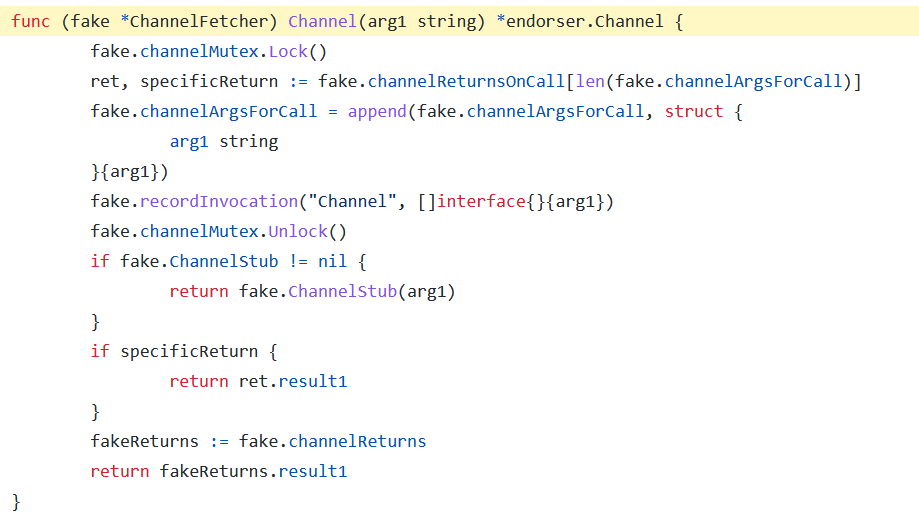
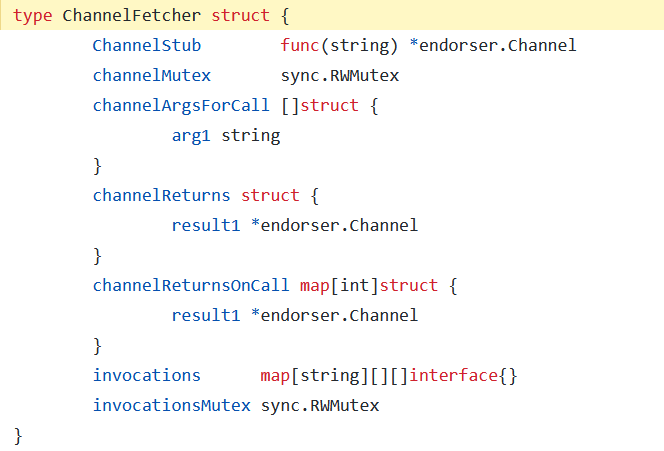
UnpackProposal函数对交易提案进行了大量合法性检查，这里不一一列举。如果通过了合法性检查，函数返回一个解包后的提案；如果交易提案不合法，函数最后该函数返回错误信息：



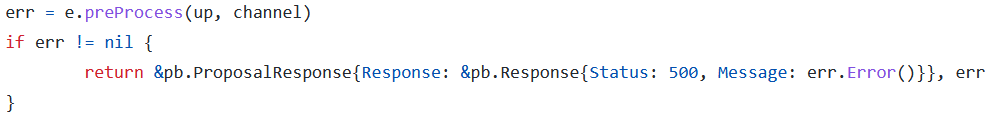
之后，ProcessProposal调用了Channel函数对解包后的交易提案的ChannelID进行检查，返回错误信息或绑定本地频道号：



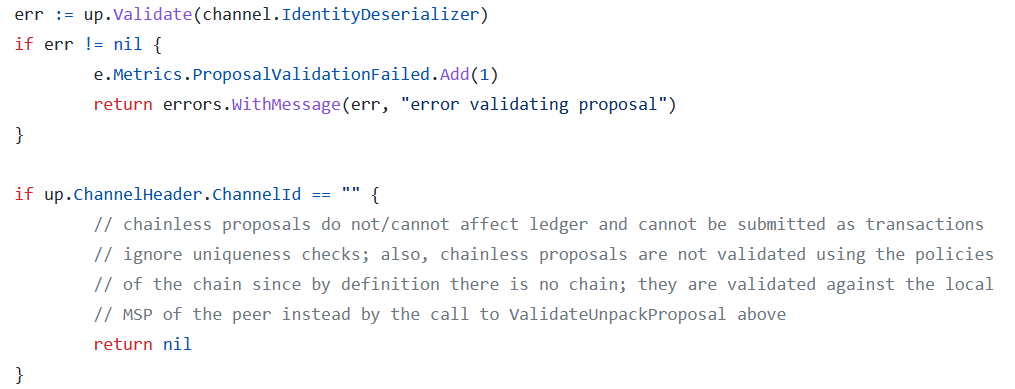
以下是Channel函数代码以及验证频道号所使用的变量ChannelFetcher结构体定义：

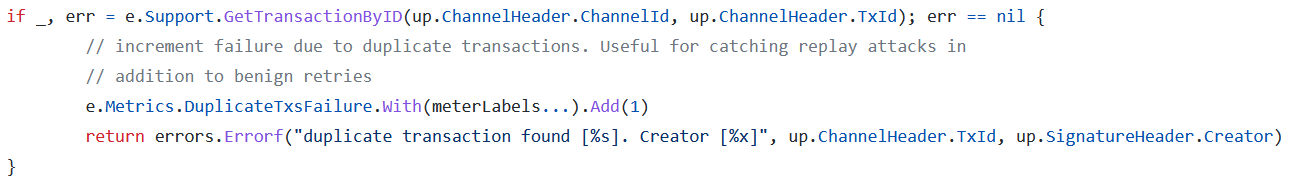
 

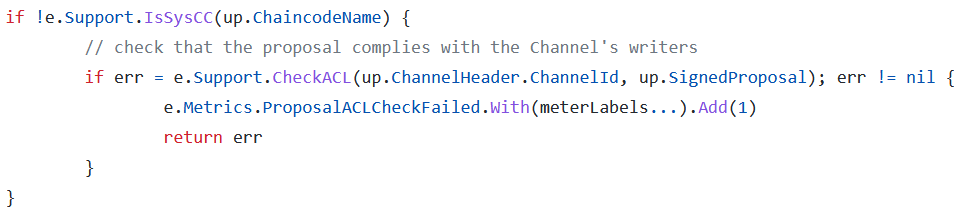
验证通过后，ProcessProposal函数进行了预执行，调用了preProcess函数：



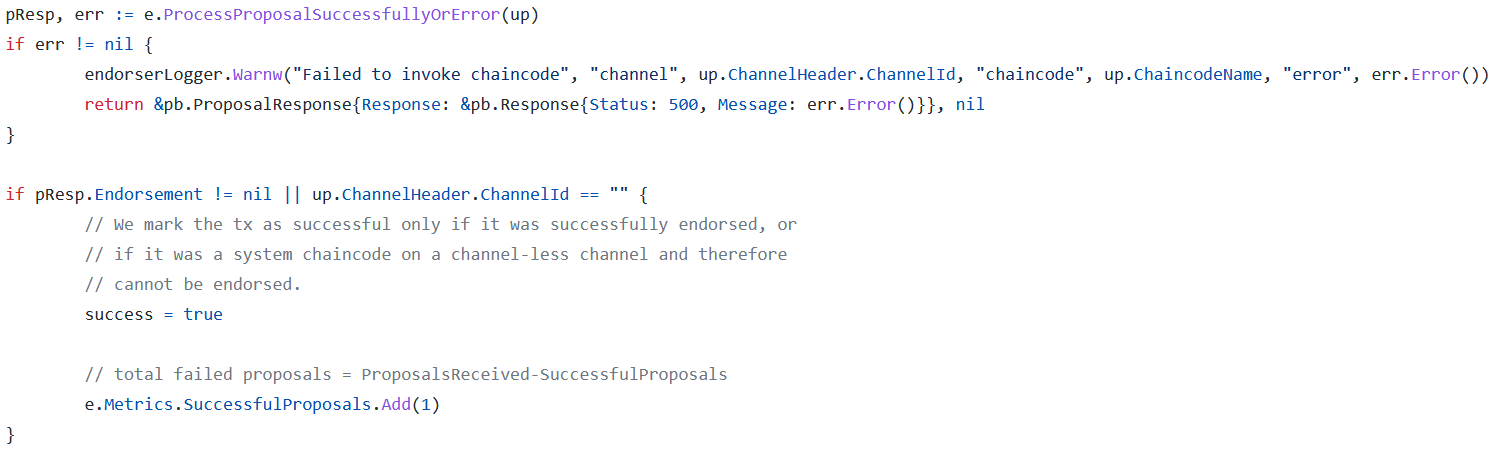
preProcess函数主要进行了了tx交易头检查（消息是否有效）、唯一性检查和应用智能合约的通道策略检查，若检查通过则返回一个空值，否则返回错误信息：



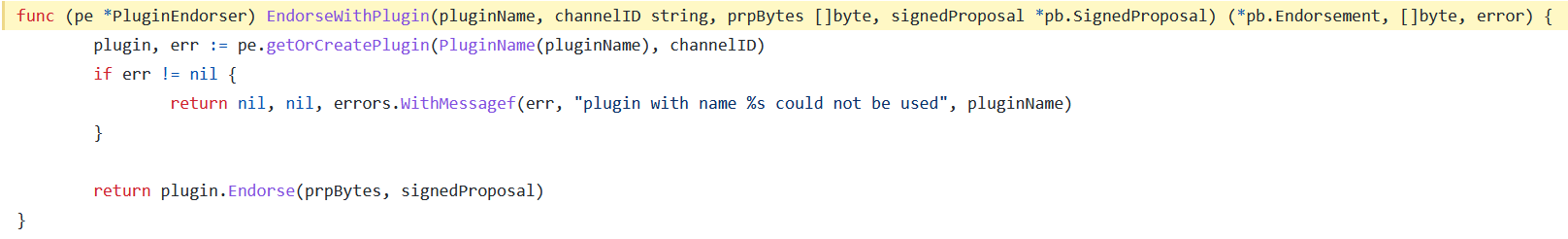




检查工作全部完成后，ProcessProposal函数将解包后的提案变量up传递给ProcessProposalSuccessfullyOrError函数，使其最终执行提案：



ProcessProposalSuccessfullyOrError函数实际上也进行了一系列的错误判断并调用simulateProposal函数对提案做了模拟执行，如果上述工作都没有出错，那么则调用core/endorser/plugin\_endorser.go文件中的EndorseWithPlugin函数执行背书操作：



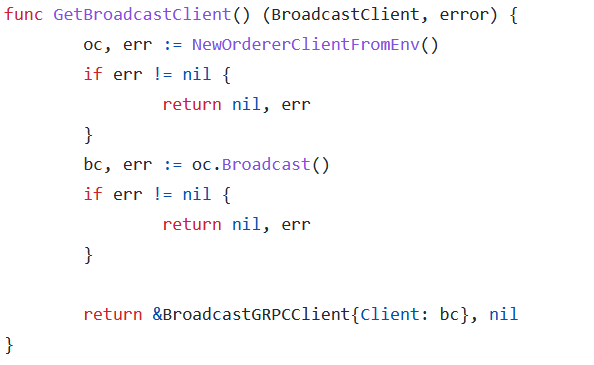
首先，EndorserWithPlugin函数调用getOrCreatePlugin函数得到了插件，并执行了插件下的Endorse，将结果返回并储存在err中。 ProcessProposalSuccessfullyOrError函数根据err结果进行了最后一次错误判断后，将simulateProposal执行的结果res和ccInterest以及EndorserWithPlugin的执行结果endorsement和mPrpBytes注入到变量ProposalResponse中并将其返回给ProcessProposal函数：



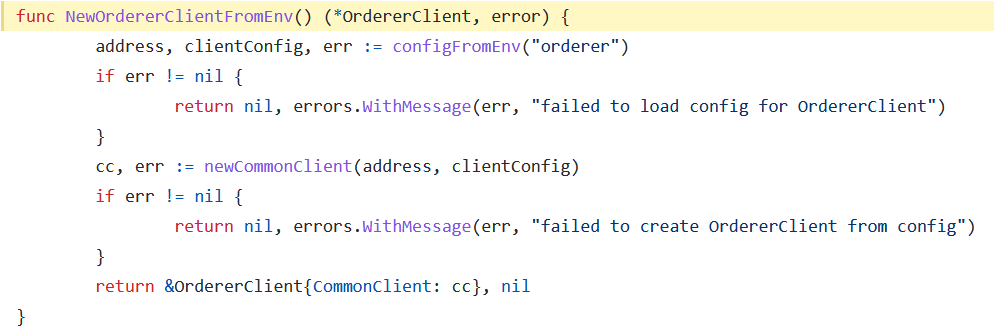
ProcessProposal函数用临时变量pResp接收ProposalResponse，再将结果提交给Endorser客户端。至此，经背书节点之手的提案签名完成。

1. 客户端向排序服务提交交易

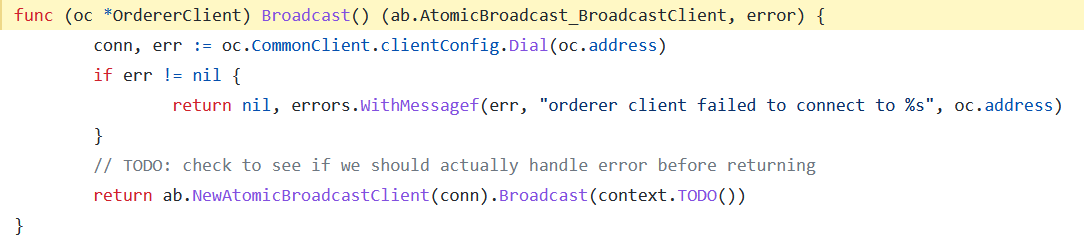
可以用internal/peer/common/broadcastclient.go文件中的 GetBroadcastClient函数来得到一个BroadcastGRPC客户端：



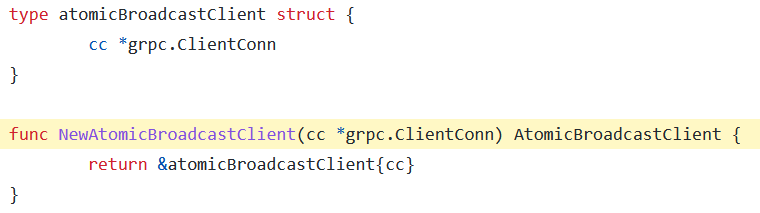
GetBroadcastClient函数首先调用了internal/peer/common/ordererclient.go文件中的NewOrdererClientFromEnv函数来创建一个排序服务客户端oc：



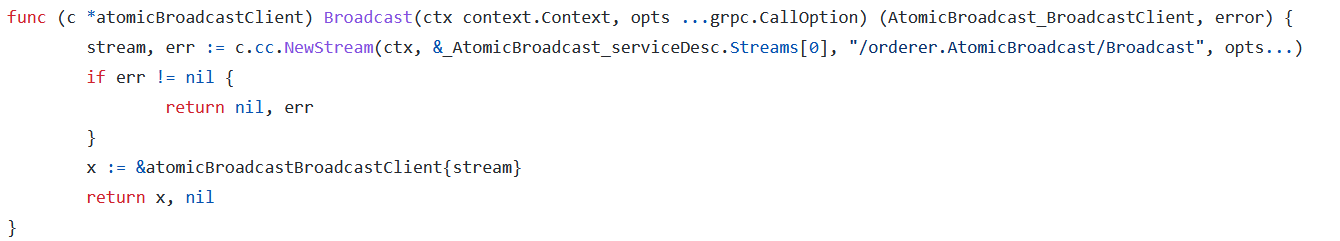
之后，GetBroadcastClient函数又使用了相同文件下的Broadcast方法：



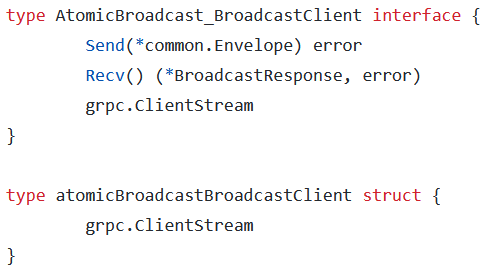
Broadcast函数调用了Dial方法来创建一个新的与oc地址连接的GRPC客户端连接，之后使用该连接作为参数，调用了vendor/github.com/hyperledger/fabric-protos-go/orderer/ab.pb.go文件中的NewAtomicBroadcastClient函数，返回一个AtomicBroadcast客户端，该客户端可以与排序服务节点连接：



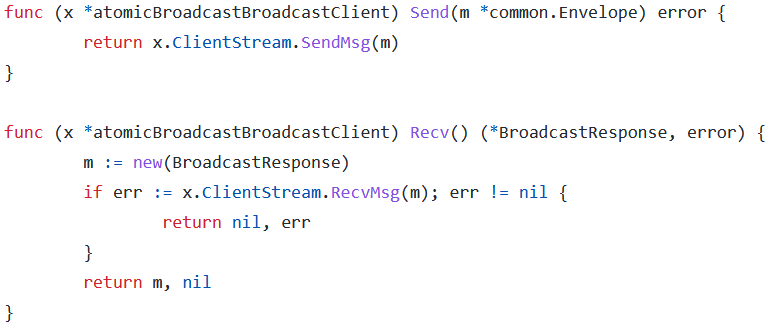
在ab.pb.go文件中的Broadcast函数调用NewStream函数，用来生成一个信息流：



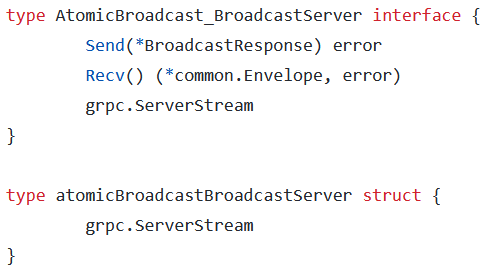
文件中也有BroadcastClient接口和结构体定义，其中有发送消息的Send函数和用于接收的Recv函数：



其中，Send函数调用SendMsg函数在ClientStream信息流上发送消息，Recv函数建立广播应答、从ClientStream接收消息并将应答返回：



以上是BroadcastClient接口实现，除了客户端，也要有服务端定义。下面是BroadcastServer的接口和结构体定义：



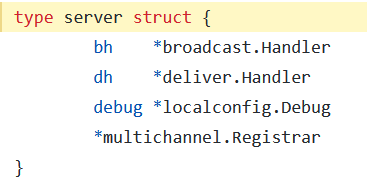
BroadcastServer也有两个方法Send和Recv，分别用来发送和接收消息;



除Broadcast外，ab.pb.go也有Deliver方法，接口实现与Broadcast类似。

1. 排序服务节点生成区块

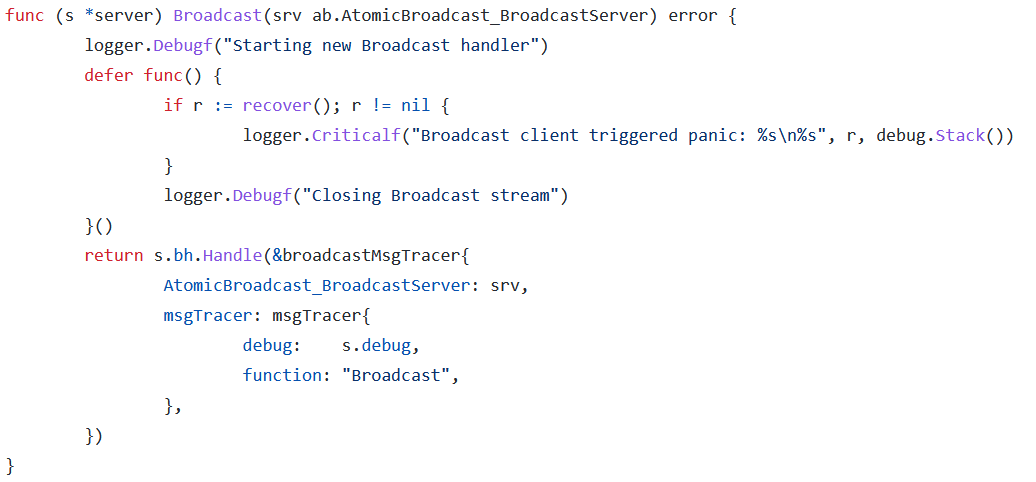
在orderer/common/server/server.go文件中定义了server结构体：



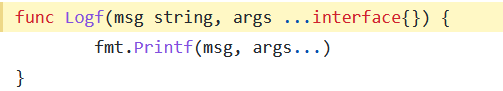
Server.go文件中的NewServer方法，可以根据广播标的和账本读者创建一个BroadcastServer：



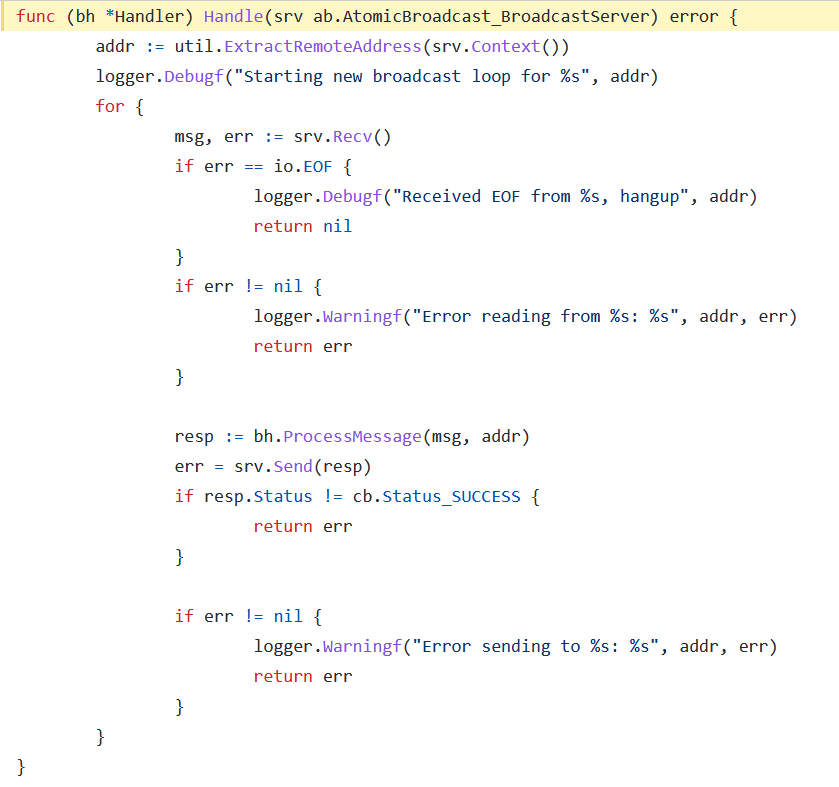
BroadcastServer可以使用定义好的Broadcast方法，从一个客户端以orderer/common/broadcast/broadcast.go文件中定义的Broadcast方式接收一串信息用于排序：



Broadcast函数中调用了几次Debugf函数，用来在调试时在日志中输出信息（Debugf函数调用了Logf函数）：

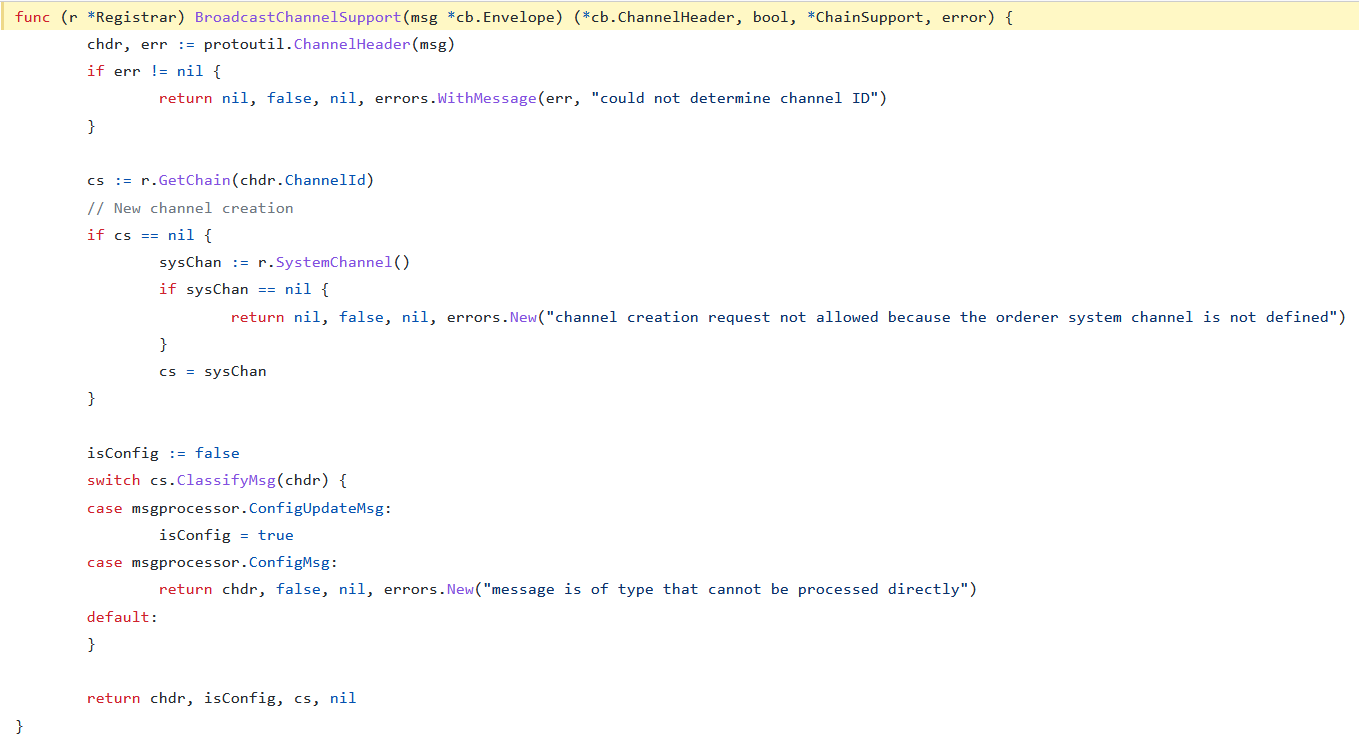


最终，Broadcast函数调用Handle函数将结果返回，以下是Handle函数的定义：

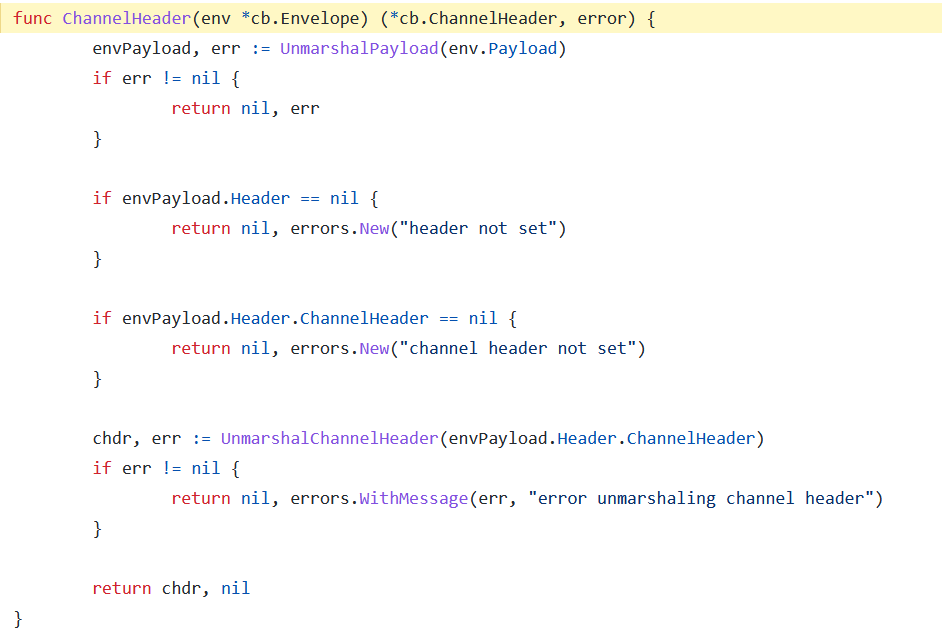


Handle函数循环调用了Recv函数，将接收到的消息存入msg中，在确认接收无误后，调用了ProcessMessage函数对消息进行处理。

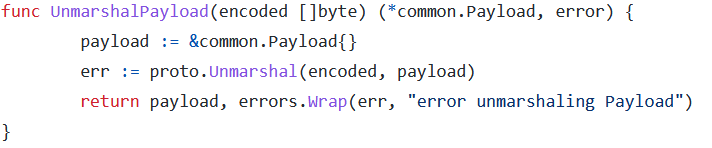
ProcessMessage函数首先调用了orderer/common/server/server.go文件中的BroadcastChannelSupport方法，此方法又调用了orderer/common/multichannel/registrar.go文件中的同名方法，返回一个频道头：

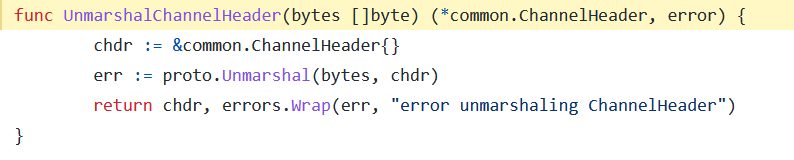


BoradcastChannelSupport函数将msg参数传递到protoutil/commonutils.go文件中的ChannelHeader方法：

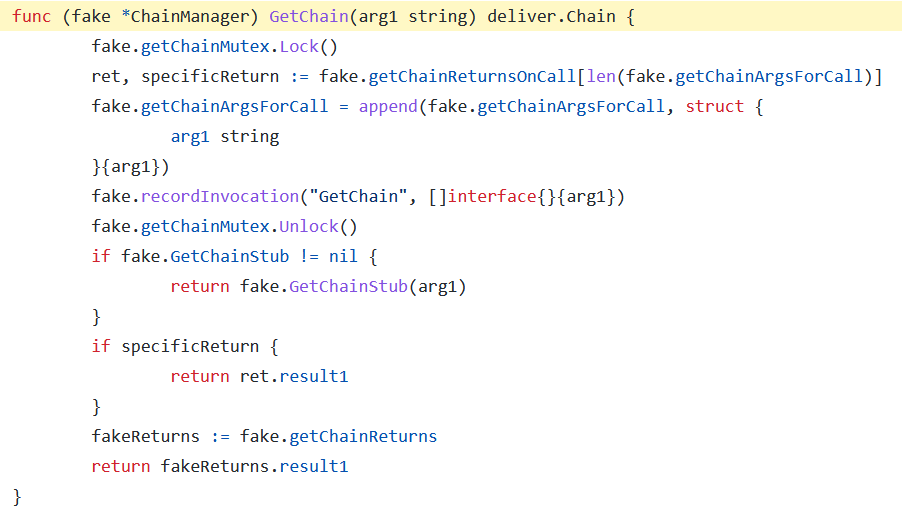


ChannelHeader方法首先调用protoutil/unmarshalers.go文件中的UnmarshalPayload方法，将消息中的Payload参数解码，在进行一系列错误判断之后又调用了unmarshalers.go文件中的UnmarshalChannelHeader方法，对频道头进行解码，检查无误后将频道头返回：





BroadcastChannelSupport函数将解码后的频道头存入临时变量chdr中，再将chdr中的ChannelId成员变量作为参数传递给common/deliver/mock/chain\_manager.go文件中的GetChain函数中，得到频道存入临时变量cs中：



在进行了错误判断后，以chdr作为参数调用链中定义的方法ClassifyMsg（位于orderer/common/broadcast/mock/channel\_support.go）得到该频道中链的配置信息，用来对不同的消息进行分类：



最后将频道头、配置信息、频道和错误信息返回给ProcessMessage函数并分别存储于临时变量chdr, isConfig, processor和err中，进行各种错误判断。如果上述步骤中均未出错，则调用orderer/common/broadcast/mock/channel\_support.go文件中的ProcessNormalMsg函数将结果返回到configSeq和err中：



如果没有出错，则调用channel\_support.go文件中的Order方法，在完成了复杂的验证工作后实现了一条消息的入队：



至此，排序服务节点生成区块，将一条消息加到链上的工作也已经完成。

以上就是Fabric交易从产生到记入账本的全过程。