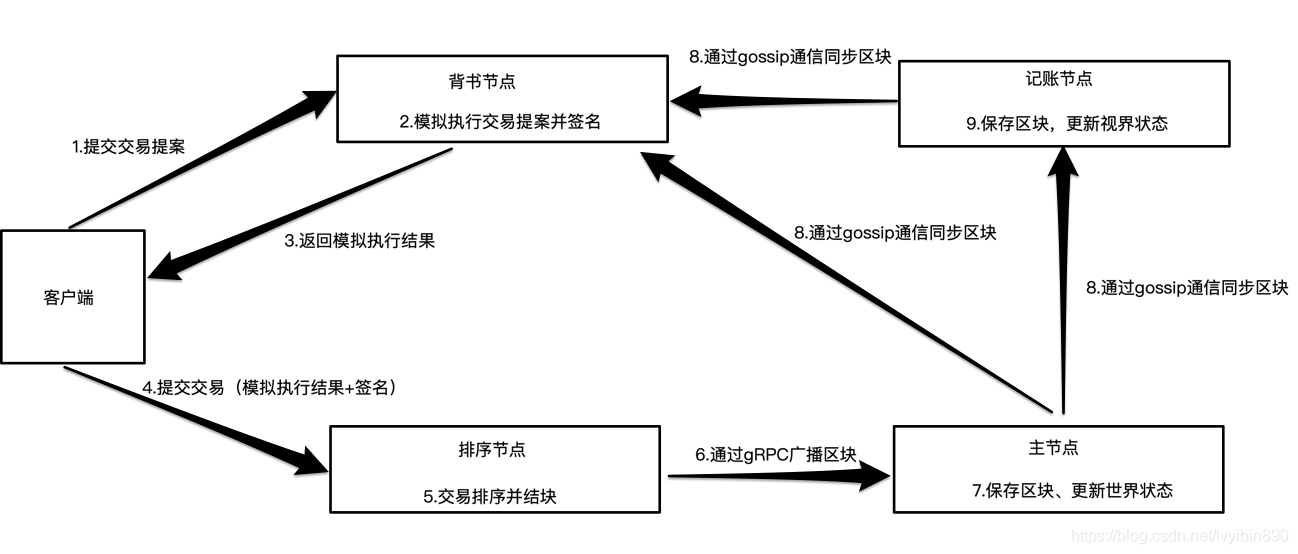
**分析Fabric源代码**

Fabric交易的整体流程图：

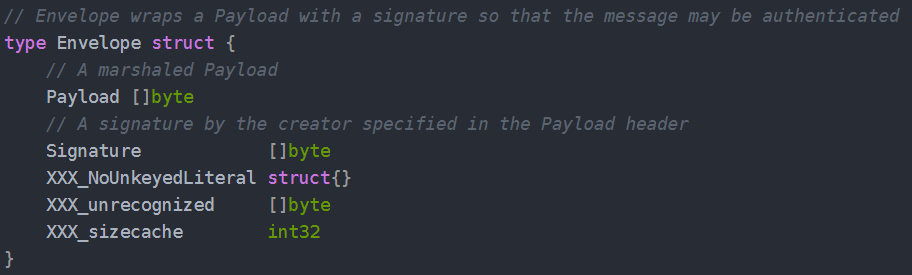


Fabric交易流程的步骤：

1. 客户端节点构造交提案，选择背书节点提交。
2. 背书节点模式执行交易提案并生成签名。
3. 背书节点将结果返回给客户端。
4. 客户端提交交易。
5. 排序节点对交易排序并做成区块。
6. 排序节点将区块打包并广播区块给组织中的主节点。
7. 主节点保存区块，更新世界状态。
8. 主节点在组织内部通过gossip传递消息同步区块。
9. 记账节点保存区块，更新世界状态。

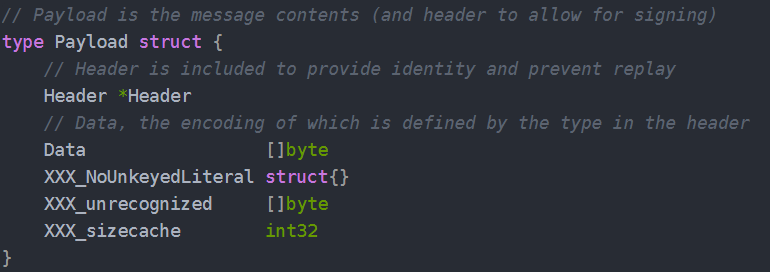
**数据结构与分析**

交易的结构原型是从Envelope的结构体开始，然后封装其他结构体比如Payload和Signature。

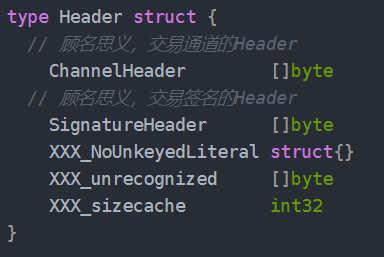


从它的命名可以知道，交易就像是一个信封，在排序节点的时候不会去拆开信封来看，作用就只是排序而不关注交易的数据。而当交易完成被打包成区块分发到各个记账节点时，记账节点才会去打开这个信封查看交易数据同时还需要验证才行。Envelope结构体还有Payload和Signature字段，交易的关键信息都存储在Payload中，Signature则是存储签名信息。往下看Payload的结构体。

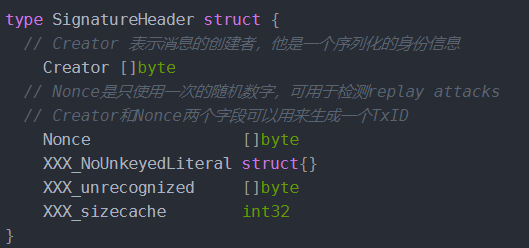
Payload结构体有两个字段，分别是Header（Payload的头部字段信息）和Data（Payload的交易序列化的信息数据）



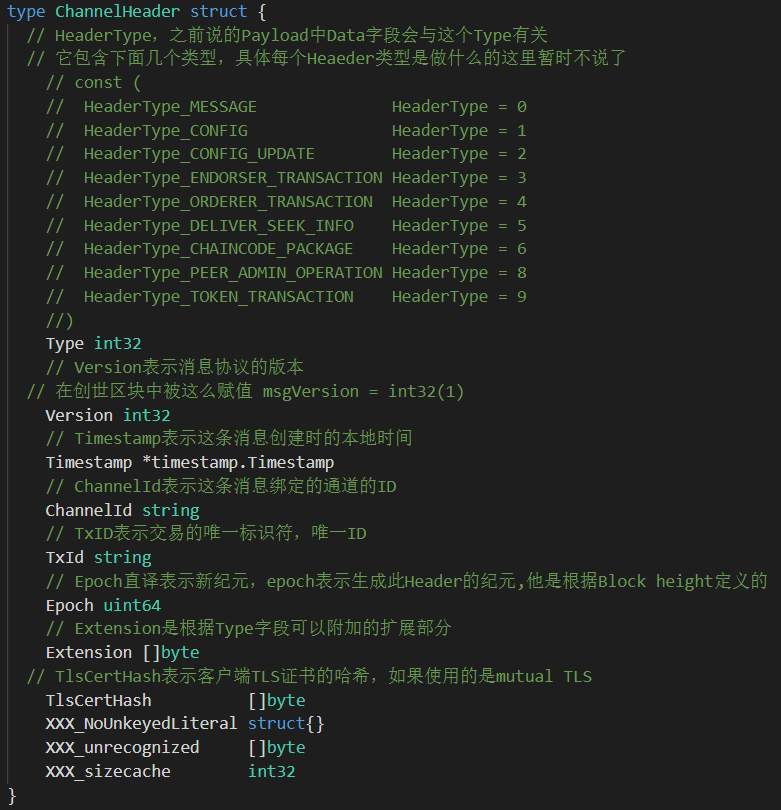
Header字段的结构体又包含两个字段的序列化对象，有ChannelHeader和SignatureHeader。



SignatureHeader结构体又有两个字段，是Creator（消息的创建者，一个序列化的身份信息），Nonce（随机数），这两个字段可以生成一个TxID。



ChannelHeader的结构体有以下的字段，有Type（Header的type），Version（消息协议的版本），Timestamp（消息创建的时间戳），ChannelID（消息绑定的通道ID），TxID（消息的唯一标识符），Epoch（Header的纪元）和Extension（根据Type字段可以附加的扩展部分）。



说完Header的部分，接下来到Payload的Data部分，Data是一个具体的反序列化对象，根据ChannelHeader中的Type不同而不同，就是不同的交易类型会导致Data的元数据不同。当Payload.Data赋值的时候，可能是不同的结构体对象序列化之后的字节切片。但是在验证交易需要获取交易信息的时候，又被反序列化成了一个统一的结构对象。

由一个GetTransaction来看，可以看到获取是从Payload.Data中反序列化成了一个Transaction对象，所以可以按照Transaction对象来分析

|  |
| --- |
| // Transaction  tx, err := utils.GetTransaction(payload.Data)  // GetTransaction Get Transaction from bytes  func GetTransaction(txBytes []byte) (\*peer.Transaction, error) {      tx := &peer.Transaction{}      err := proto.Unmarshal(txBytes, tx)      return tx, errors.Wrap(err, "error unmarshaling Transaction")  } |

接下来介绍Transaction的结构体，包括一个或多个TransactionAction，每个TransactionAction都将一个提案绑定到多个actions里，Transaction是要被发送到排序节点的最终结果。

* Transaction是原子性的，要么提交所有actions，要么都不提交。
* 当Transaction包含一个以上的Headers是，那么每个Header.Creator字段必须意义。
* 一个客户端可以自由发布多个独立提案，每个提案都包含他们的Header和Payload。
* 每个提案都被独立背书，生成一个action，每个背书者都有签名。
* 任意数量的独立提案可能包含在一个Transaction中以确保是原子性的。

|  |
| --- |
| type Transaction struct {      // Actions是TransactionAction的数组    // 为了每个Transaction容纳多个acitons，必须要是数组      Actions              []\*TransactionAction      XXX\_NoUnkeyedLiteral struct{}      XXX\_unrecognized     []byte      XXX\_sizecache        int32  } |

TransactionAction结构体，其Payload字段是根据Header中的Type字段来定义的，当Type是ChainCode时，它就是ChainCodeActionPayload的序列化；

|  |
| --- |
| // TransactionAction 绑定一个提案到它的aciton.  // header中的type字段决定了应用于账本的action的type  type TransactionAction struct {      // 交易提案action的Header，Proposal的header      Header []byte      // 交易提案action的主要信息    // 当type是chaincode时，它是ChaincodeActionPayload的序列化      Payload              []byte      XXX\_NoUnkeyedLiteral struct{}      XXX\_unrecognized     []byte      XXX\_sizecache        int32  } |

ChainCodeActionPayload的结构体可分成两个模块，字段ChaincodeProposalPayload类型的序列化信息，包含了原始调用函数的参数信息，还有Action列表中的ChaincodeEndorsedAction类型。

|  |
| --- |
| // ChaincodeActionPayload是当Header的type设置为CHAINCODE时用于赋值给TransactionAction的Payload的  type ChaincodeActionPayload struct {      // 这个字段是 ChaincodeProposalPayload 类型的序列化信息        ChaincodeProposalPayload []byte      // 应用于账本的actions的列表        Action               \*ChaincodeEndorsedAction        XXX\_NoUnkeyedLiteral struct{}        XXX\_unrecognized     []byte        XXX\_sizecache        int32    } |

以下是ChaincodeProposalPayload类型的结构体，是提案的主要信息，包含两个字段，字段Input包含了链码调用的一些参数，TransientMap则是包含数据比如密码材料，可以用来实现某种形式的应用级机密性。

|  |
| --- |
| // ChaincodeProposalPayload是提案的具体信息，当Header的类型是CHAINCODE时  // 它包含了这次调用的参数  type ChaincodeProposalPayload struct {      // 包含了这次调用的参数        Input []byte      // TransientMap包含一些数据(例如密码材料)，可以用来实现某种形式的应用级机密性      // 这个字段的内容应该总是从交易中被省略，从分类账中被排除。        TransientMap         map[string][]byte        XXX\_NoUnkeyedLiteral struct{}        XXX\_unrecognized     []byte        XXX\_sizecache        int32    } |

以下是ChaincodeEndorsedAction的结构体，包含了一个指定的提案(Proposal)背书信息，主要有两个字段，为ProposalResponsePayload提案的响应信息，是一个序列化，还有Endorsements提案的背书，基本上是背书者在ProposalResponsePayload上的签名。

|  |
| --- |
| // ChaincodeEndorsedAction 包含了一个指定提案的背书信息  type ChaincodeEndorsedAction struct {      // 提案响应信息        ProposalResponsePayload []byte      // 提案的背书，基本上是背书者在proposalResponsePayload上的签名        Endorsements         []\*Endorsement        XXX\_NoUnkeyedLiteral struct{}        XXX\_unrecognized     []byte        XXX\_sizecache        int32         } |

以下是ProposalResponsePayload的结构体，包含提案的对应哈希值(ProposalHash)字段，还有根据Type字段可以附加的扩展部分——Extension字段。

|  |
| --- |
| type ProposalResponsePayload struct {      // 提案相应的哈希        ProposalHash []byte      // 根据Type字段可以附加的扩展部分        Extension            []byte        XXX\_NoUnkeyedLiteral struct{}        XXX\_unrecognized     []byte        XXX\_sizecache        int32    } |

对于Chaincode，它是ChaincodeAction的序列化信息，ChaincodeAction包含了链码执行生成的Events、Results（读写集）、Response（响应）、ChaincodeID（链码ID）、TokenExpectation。

|  |
| --- |
| // ChaincodeAction包含了链码执行生成的events的actions  type ChaincodeAction struct {      // 包含了链码执行的读写集        Results []byte      // 包含了链码调用生成的events        Events []byte      // 包含了链码调用生成的Response        Response \*Response      // 链码ID        ChaincodeId \*ChaincodeID      // 包含了链码调用生成的token expectation        TokenExpectation     \*token.TokenExpectation        XXX\_NoUnkeyedLiteral struct{}        XXX\_unrecognized     []byte        XXX\_sizecache        int32    } |

最后是Endorsement结构体，主要包含了一些背书者的身份信息以及签名，有Endorse（背书者）字段和Signature（签名）字段。

|  |
| --- |
| type Endorsement struct {      // 背书者本身的身份信息      Endorser []byte    // 背书签名      Signature            []byte      XXX\_NoUnkeyedLiteral struct{}      XXX\_unrecognized     []byte      XXX\_sizecache        int32  } |

**背书节点的交易背书主要流程（过程）：**

1. 检查Proposal
2. 为交易创建模拟器，调用模拟器进行模拟执行交易，生成执行结果
3. 背书模块对执行结果和Proposal身份信息背书，生成背书响应发送给客户端（Client）。

Peer接收Proposal并处理流程：

|  |
| --- |
| func (e \*Endorser) ProcessProposal(ctx context.Context, signedProp \*pb.SignedProposal) (\*pb.ProposalResponse, error) {      ...      // 0 -- check and validate      // 这里有相当多的工作量      vr, err := e.preProcess(signedProp)      if err != nil {          resp := vr.resp          return resp, err      }      prop, hdrExt, chainID, txid := vr.prop, vr.hdrExt, vr.chainID, vr.txid      // 获取指定账本模拟器      // obtaining once the tx simulator for this proposal. This will be nil      // for chainless proposals      // Also obtain a history query executor for history queries, since tx simulator does not cover history      var txsim ledger.TxSimulator      var historyQueryExecutor ledger.HistoryQueryExecutor      if acquireTxSimulator(chainID, vr.hdrExt.ChaincodeId) {          if txsim, err = e.s.GetTxSimulator(chainID, txid); err != nil {              return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, nil          }          ...          defer txsim.Done()          if historyQueryExecutor, err = e.s.GetHistoryQueryExecutor(chainID); err != nil {              return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, nil          }      }      txParams := &ccprovider.TransactionParams{          ChannelID:            chainID,          TxID:                 txid,          SignedProp:           signedProp,          Proposal:             prop,          TXSimulator:          txsim,          HistoryQueryExecutor: historyQueryExecutor,      }      // this could be a request to a chainless SysCC      // 模拟执行交易，失败则返回背书失败的响应      // 1 -- simulate      cd, res, simulationResult, ccevent, err := e.SimulateProposal(txParams, hdrExt.ChaincodeId)      if err != nil {          return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, nil      }      if res != nil {          if res.Status >= shim.ERROR {              endorserLogger.Errorf("[%s][%s] simulateProposal() resulted in chaincode %s response status %d for txid: %s", chainID, shorttxid(txid), hdrExt.ChaincodeId, res.Status, txid)              var cceventBytes []byte              if ccevent != nil {                  cceventBytes, err = putils.GetBytesChaincodeEvent(ccevent)                  if err != nil {                      return nil, errors.Wrap(err, "failed to marshal event bytes")                  }              }              pResp, err := putils.CreateProposalResponseFailure(prop.Header, prop.Payload, res, simulationResult, cceventBytes, hdrExt.ChaincodeId, hdrExt.PayloadVisibility)              if err != nil {                  return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, nil              }              return pResp, nil          }      }      // 对模拟执行的结果进行签名背书      // 2 -- endorse and get a marshalled ProposalResponse message      var pResp \*pb.ProposalResponse      // TODO till we implement global ESCC, CSCC for system chaincodes      // chainless proposals (such as CSCC) don't have to be endorsed      if chainID == "" {          pResp = &pb.ProposalResponse{Response: res}      } else {          // Note: To endorseProposal(), we pass the released txsim. Hence, an error would occur if we try to use this txsim          pResp, err = e.endorseProposal(ctx, chainID, txid, signedProp, prop, res, simulationResult, ccevent, hdrExt.PayloadVisibility, hdrExt.ChaincodeId, txsim, cd)          ...      }      // Set the proposal response payload - it      // contains the "return value" from the      // chaincode invocation      pResp.Response = res      // total failed proposals = ProposalsReceived-SuccessfulProposals      e.Metrics.SuccessfulProposals.Add(1)      success = true      return pResp, nil  } |

preProcess检查Proposal和获取信息

|  |
| --- |
| // preProcess checks the tx proposal headers, uniqueness and ACL  // 检查proposal、ACL  func (e \*Endorser) preProcess(signedProp \*pb.SignedProposal) (\*validateResult, error) {      // 包含proposal、header、chainID、txid等信息      vr := &validateResult{}      // at first, we check whether the message is valid      // 检查proposal，并获取各种需要的信息      prop, hdr, hdrExt, err := validation.ValidateProposalMessage(signedProp)      if err != nil {          e.Metrics.ProposalValidationFailed.Add(1)          vr.resp = &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}          return vr, err      }      // 获取Header中的2个Header      chdr, err := putils.UnmarshalChannelHeader(hdr.ChannelHeader)      if err != nil {          vr.resp = &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}          return vr, err      }      shdr, err := putils.GetSignatureHeader(hdr.SignatureHeader)      if err != nil {          vr.resp = &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}          return vr, err      }      // 检查是否调用了不可外部（用户）的系统链码      // 先找到链码实例，然后调用链码的方法判断本身是否可调用      // block invocations to security-sensitive system chaincodes      if e.s.IsSysCCAndNotInvokableExternal(hdrExt.ChaincodeId.Name) {          endorserLogger.Errorf("Error: an attempt was made by %#v to invoke system chaincode %s", shdr.Creator, hdrExt.ChaincodeId.Name)          err = errors.Errorf("chaincode %s cannot be invoked through a proposal", hdrExt.ChaincodeId.Name)          vr.resp = &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}          return vr, err      }      chainID := chdr.ChannelId      txid := chdr.TxId      endorserLogger.Debugf("[%s][%s] processing txid: %s", chainID, shorttxid(txid), txid)      if chainID != "" {          // labels that provide context for failure metrics          meterLabels := []string{              "channel", chainID,              "chaincode", hdrExt.ChaincodeId.Name + ":" + hdrExt.ChaincodeId.Version,          }          // 检查交易是否已上链          // Here we handle uniqueness check and ACLs for proposals targeting a chain          // Notice that ValidateProposalMessage has already verified that TxID is computed properly          if \_, err = e.s.GetTransactionByID(chainID, txid); err == nil {              // increment failure due to duplicate transactions. Useful for catching replay attacks in              // addition to benign retries              e.Metrics.DuplicateTxsFailure.With(meterLabels...).Add(1)              err = errors.Errorf("duplicate transaction found [%s]. Creator [%x]", txid, shdr.Creator)              vr.resp = &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}              return vr, err          }          // 用户链码检查ACL          // check ACL only for application chaincodes; ACLs          // for system chaincodes are checked elsewhere          if !e.s.IsSysCC(hdrExt.ChaincodeId.Name) {              // check that the proposal complies with the Channel's writers              if err = e.s.CheckACL(signedProp, chdr, shdr, hdrExt); err != nil {                  e.Metrics.ProposalACLCheckFailed.With(meterLabels...).Add(1)                  vr.resp = &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}                  return vr, err              }          }      } else {          // chainless proposals do not/cannot affect ledger and cannot be submitted as transactions          // ignore uniqueness checks; also, chainless proposals are not validated using the policies          // of the chain since by definition there is no chain; they are validated against the local          // MSP of the peer instead by the call to ValidateProposalMessage above      }      // 保存提取的各信息      vr.prop, vr.hdrExt, vr.chainID, vr.txid = prop, hdrExt, chainID, txid      return vr, nil  }  // ValidateProposalMessage checks the validity of a SignedProposal message  // this function returns Header and ChaincodeHeaderExtension messages since they  // have been unmarshalled and validated  func ValidateProposalMessage(signedProp \*pb.SignedProposal) (\*pb.Proposal, \*common.Header, \*pb.ChaincodeHeaderExtension, error) {      if signedProp == nil {          return nil, nil, nil, errors.New("nil arguments")      }      putilsLogger.Debugf("ValidateProposalMessage starts for signed proposal %p", signedProp)      // extract the Proposal message from signedProp      prop, err := utils.GetProposal(signedProp.ProposalBytes)      if err != nil {          return nil, nil, nil, err      }      // 1) look at the ProposalHeader      hdr, err := utils.GetHeader(prop.Header)      if err != nil {          return nil, nil, nil, err      }      // validate the header      chdr, shdr, err := validateCommonHeader(hdr)      if err != nil {          return nil, nil, nil, err      }      // 从SignatureHeader交易客户端的签名      // validate the signature      err = checkSignatureFromCreator(shdr.Creator, signedProp.Signature, signedProp.ProposalBytes, chdr.ChannelId)      if err != nil {          // log the exact message on the peer but return a generic error message to          // avoid malicious users scanning for channels          putilsLogger.Warningf("channel [%s]: %s", chdr.ChannelId, err)          sId := &msp.SerializedIdentity{}          err := proto.Unmarshal(shdr.Creator, sId)          if err != nil {              // log the error here as well but still only return the generic error              err = errors.Wrap(err, "could not deserialize a SerializedIdentity")              putilsLogger.Warningf("channel [%s]: %s", chdr.ChannelId, err)          }          return nil, nil, nil, errors.Errorf("access denied: channel [%s] creator org [%s]", chdr.ChannelId, sId.Mspid)      }      // 检查txid的计算是否符合规则      // Verify that the transaction ID has been computed properly.      // This check is needed to ensure that the lookup into the ledger      // for the same TxID catches duplicates.      err = utils.CheckTxID(          chdr.TxId,          shdr.Nonce,          shdr.Creator)      if err != nil {          return nil, nil, nil, err      }      // 依据不同的proposal类型对proposal分别进行检查      // continue the validation in a way that depends on the type specified in the header      switch common.HeaderType(chdr.Type) {      case common.HeaderType\_CONFIG:          //which the types are different the validation is the same          //viz, validate a proposal to a chaincode. If we need other          //special validation for confguration, we would have to implement          //special validation          fallthrough      case common.HeaderType\_ENDORSER\_TRANSACTION:          // 主要是提取ChaincodeHeaderExtension          // validation of the proposal message knowing it's of type CHAINCODE          chaincodeHdrExt, err := validateChaincodeProposalMessage(prop, hdr)          if err != nil {              return nil, nil, nil, err          }          return prop, hdr, chaincodeHdrExt, err      default:          //NOTE : we proably need a case          return nil, nil, nil, errors.Errorf("unsupported proposal type %d", common.HeaderType(chdr.Type))      }  } |

背书节点模拟执行交易

获取模拟器

|  |
| --- |
| // Support contains functions that the endorser requires to execute its tasks  type Support interface {      crypto.SignerSupport      // IsSysCCAndNotInvokableExternal returns true if the supplied chaincode is      // ia system chaincode and it NOT invokable      IsSysCCAndNotInvokableExternal(name string) bool      // GetTxSimulator returns the transaction simulator for the specified ledger      // a client may obtain more than one such simulator; they are made unique      // by way of the supplied txid      GetTxSimulator(ledgername string, txid string) (ledger.TxSimulator, error)  }  // GetTxSimulator returns the transaction simulator for the specified ledger  // a client may obtain more than one such simulator; they are made unique  // by way of the supplied txid  func (s \*SupportImpl) GetTxSimulator(ledgername string, txid string) (ledger.TxSimulator, error) {      // 使用账本和txid创建模拟器，每个交易有单独的模拟器      lgr := s.Peer.GetLedger(ledgername)      if lgr == nil {          return nil, errors.Errorf("Channel does not exist: %s", ledgername)      }      return lgr.NewTxSimulator(txid)  }  // NewTxSimulator returns new `ledger.TxSimulator`  func (l \*kvLedger) NewTxSimulator(txid string) (ledger.TxSimulator, error) {      return l.txtmgmt.NewTxSimulator(txid)  }  // NewTxSimulator implements method in interface `txmgmt.TxMgr`  func (txmgr \*LockBasedTxMgr) NewTxSimulator(txid string) (ledger.TxSimulator, error) {      logger.Debugf("constructing new tx simulator")      s, err := newLockBasedTxSimulator(txmgr, txid)      if err != nil {          return nil, err      }      txmgr.commitRWLock.RLock()      return s, nil  }  // 就2项重要的：查询执行器、读写集构建器  // LockBasedTxSimulator is a transaction simulator used in `LockBasedTxMgr`  type lockBasedTxSimulator struct {      lockBasedQueryExecutor      rwsetBuilder              \*rwsetutil.RWSetBuilder      writePerformed            bool      pvtdataQueriesPerformed   bool      simulationResultsComputed bool      paginatedQueriesPerformed bool  }  func newLockBasedTxSimulator(txmgr \*LockBasedTxMgr, txid string) (\*lockBasedTxSimulator, error) {      // 创建读写集构建器，能帮助构建读写集      rwsetBuilder := rwsetutil.NewRWSetBuilder()      helper := newQueryHelper(txmgr, rwsetBuilder)      logger.Debugf("constructing new tx simulator txid = [%s]", txid)      return &lockBasedTxSimulator{lockBasedQueryExecutor{helper, txid}, rwsetBuilder, false, false, false, false}, nil  }  // LockBasedQueryExecutor is a query executor used in `LockBasedTxMgr`  // "只读"，不包含写相关的操作  type lockBasedQueryExecutor struct {      helper \*queryHelper      txid   string  } |

执行Endorser的部分

|  |
| --- |
| if acquireTxSimulator(chainID, vr.hdrExt.ChaincodeId) {      if txsim, err = e.s.GetTxSimulator(chainID, txid); err != nil {          return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, nil      }      defer txsim.Done()      // 历史查询器      if historyQueryExecutor, err = e.s.GetHistoryQueryExecutor(chainID); err != nil {          return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, nil      }  }  txParams := &ccprovider.TransactionParams{      ChannelID:            chainID,      TxID:                 txid,      SignedProp:           signedProp,      Proposal:             prop,      TXSimulator:          txsim,    // 模拟器在此      HistoryQueryExecutor: historyQueryExecutor,  }  // this could be a request to a chainless SysCC  // TODO: if the proposal has an extension, it will be of type ChaincodeAction;  //       if it's present it means that no simulation is to be performed because  //       we're trying to emulate a submitting peer. On the other hand, we need  //       to validate the supplied action before endorsing it  // 模拟执行交易，失败则返回背书失败的响应  // 1 -- simulate  cd, res, simulationResult, ccevent, err := e.SimulateProposal(txParams, hdrExt.ChaincodeId) |

|  |
| --- |
| // SimulateProposal simulates the proposal by calling the chaincode  func (e \*Endorser) SimulateProposal(txParams \*ccprovider.TransactionParams, cid \*pb.ChaincodeID) (ccprovider.ChaincodeDefinition, \*pb.Response, []byte, \*pb.ChaincodeEvent, error) {      endorserLogger.Debugf("[%s][%s] Entry chaincode: %s", txParams.ChannelID, shorttxid(txParams.TxID), cid)      defer endorserLogger.Debugf("[%s][%s] Exit", txParams.ChannelID, shorttxid(txParams.TxID))      // we do expect the payload to be a ChaincodeInvocationSpec      // if we are supporting other payloads in future, this be glaringly point      // as something that should change      // 根据Proposal生成Invoke需要的信息      cis, err := putils.GetChaincodeInvocationSpec(txParams.Proposal)      if err != nil {          return nil, nil, nil, nil, err      }      // 链码的元数据      var cdLedger ccprovider.ChaincodeDefinition      var version string      // 设置version      if !e.s.IsSysCC(cid.Name) {          // 根据要调用的链码名称，从lscc获取链码的元数据          cdLedger, err = e.s.GetChaincodeDefinition(cid.Name, txParams.TXSimulator)          if err != nil {              return nil, nil, nil, nil, errors.WithMessage(err, fmt.Sprintf("make sure the chaincode %s has been successfully instantiated and try again", cid.Name))          }          version = cdLedger.CCVersion()          // 实际被打桩了，无实现          err = e.s.CheckInstantiationPolicy(cid.Name, version, cdLedger)          if err != nil {              return nil, nil, nil, nil, err          }      } else {          // scc版本是固定的"latest"          version = util.GetSysCCVersion()      }      // ---3. execute the proposal and get simulation results      var simResult \*ledger.TxSimulationResults      var pubSimResBytes []byte      var res \*pb.Response      var ccevent \*pb.ChaincodeEvent      // 模拟执行，执行结果保存在模拟器      res, ccevent, err = e.callChaincode(txParams, version, cis.ChaincodeSpec.Input, cid)      if err != nil {          endorserLogger.Errorf("[%s][%s] failed to invoke chaincode %s, error: %+v", txParams.ChannelID, shorttxid(txParams.TxID), cid, err)          return nil, nil, nil, nil, err      }      if txParams.TXSimulator != nil {          // 通过模拟器获取模拟执行结果，包含公开和私密数据2份读写集          if simResult, err = txParams.TXSimulator.GetTxSimulationResults(); err != nil {              txParams.TXSimulator.Done()              return nil, nil, nil, nil, err          }          // 存在私密数据          if simResult.PvtSimulationResults != nil {              if cid.Name == "lscc" {                  // TODO: remove once we can store collection configuration outside of LSCC                  txParams.TXSimulator.Done()                  return nil, nil, nil, nil, errors.New("Private data is forbidden to be used in instantiate")              }              // 获取要通过Gossip传播的私密数据              pvtDataWithConfig, err := e.AssemblePvtRWSet(simResult.PvtSimulationResults, txParams.TXSimulator)              // To read collection config need to read collection updates before              // releasing the lock, hence txParams.TXSimulator.Done()  moved down here              txParams.TXSimulator.Done()              if err != nil {                  return nil, nil, nil, nil, errors.WithMessage(err, "failed to obtain collections config")              }              endorsedAt, err := e.s.GetLedgerHeight(txParams.ChannelID)              if err != nil {                  return nil, nil, nil, nil, errors.WithMessage(err, fmt.Sprint("failed to obtain ledger height for channel", txParams.ChannelID))              }              // Add ledger height at which transaction was endorsed,              // `endorsedAt` is obtained from the block storage and at times this could be 'endorsement Height + 1'.              // However, since we use this height only to select the configuration (3rd parameter in distributePrivateData) and              // manage transient store purge for orphaned private writesets (4th parameter in distributePrivateData), this works for now.              // Ideally, ledger should add support in the simulator as a first class function `GetHeight()`.              pvtDataWithConfig.EndorsedAt = endorsedAt              // 把私密数据同通道id、交易id和区块高度发出去，代表私密数据所属的区块和交易              if err := e.distributePrivateData(txParams.ChannelID, txParams.TxID, pvtDataWithConfig, endorsedAt); err != nil {                  return nil, nil, nil, nil, err              }          }          // 交易模拟完成，释放模拟器占用的资源          txParams.TXSimulator.Done()          // 获取模拟执行的公开结果          if pubSimResBytes, err = simResult.GetPubSimulationBytes(); err != nil {              return nil, nil, nil, nil, err          }      }      // 返回链码元数据、模拟执行结果、交易执行产生的事件      return cdLedger, res, pubSimResBytes, ccevent, nil  } |

callChaincode调用chaincode模块模拟执行交易，获取交易执行的公开和私密数据读写集，以及交易执行产生的事件，并把结果返回给上层进行背书。在前面的流程中，还未区分系统链码SCC和用户链码UCC，SCC和UCC会通过Execute函数被传递给chaincode模块执行交易。如果是调用lscc部署或升级UCC，则会调用ExecuteLegacyInit执行链码容器的初始化。最后返回链码模拟执行结果和事件。

|  |
| --- |
| // call specified chaincode (system or user)  func (e \*Endorser) callChaincode(txParams \*ccprovider.TransactionParams, version string, input \*pb.ChaincodeInput, cid \*pb.ChaincodeID) (\*pb.Response, \*pb.ChaincodeEvent, error) {      ...      // scc也在这执行      // is this a system chaincode      res, ccevent, err = e.s.Execute(txParams, txParams.ChannelID, cid.Name, version, txParams.TxID, txParams.SignedProp, txParams.Proposal, input)      if err != nil {          return nil, nil, err      }      ...      // 如果是调用lscc部署或升级链码，会走这段流程      if cid.Name == "lscc" && len(input.Args) >= 3 && (string(input.Args[0]) == "deploy" || string(input.Args[0]) == "upgrade") {          userCDS, err := putils.GetChaincodeDeploymentSpec(input.Args[2], e.PlatformRegistry)          ...          // 进行链码容器初始化，最后会调用链码的Init的函数          \_, \_, err = e.s.ExecuteLegacyInit(txParams, txParams.ChannelID, cds.ChaincodeSpec.ChaincodeId.Name, cds.ChaincodeSpec.ChaincodeId.Version, txParams.TxID, txParams.SignedProp, txParams.Proposal, cds)          ...      }      // ----- END -------      return res, ccevent, err  } |

Execute函数调用

|  |
| --- |
| // Execute a proposal and return the chaincode response  func (s \*SupportImpl) Execute(txParams \*ccprovider.TransactionParams, cid, name, version, txid string, signedProp \*pb.SignedProposal, prop \*pb.Proposal, input \*pb.ChaincodeInput) (\*pb.Response, \*pb.ChaincodeEvent, error) {      cccid := &ccprovider.CCContext{          Name:    name,          Version: version,      }      // decorate the chaincode input      decorators := library.InitRegistry(library.Config{}).Lookup(library.Decoration).([]decoration.Decorator)      input.Decorations = make(map[string][]byte)      input = decoration.Apply(prop, input, decorators...)      txParams.ProposalDecorations = input.Decorations      return s.ChaincodeSupport.Execute(txParams, cccid, input)  } |

Chaincode部分

|  |
| --- |
| func (cs \*ChaincodeSupport) Execute(txParams \*ccprovider.TransactionParams, cccid \*ccprovider.CCContext, input \*pb.ChaincodeInput) (\*pb.Response, \*pb.ChaincodeEvent, error) {      // Invoke得到ChaincodeMessage      resp, err := cs.Invoke(txParams, cccid, input)      // 根据ChaincodeMessage得到Response和事件      return processChaincodeExecutionResult(txParams.TxID, cccid.Name, resp, err)  }  func (cs \*ChaincodeSupport) Invoke(txParams \*ccprovider.TransactionParams, cccid \*ccprovider.CCContext, input \*pb.ChaincodeInput) (\*pb.ChaincodeMessage, error) {      h, err := cs.Launch(txParams.ChannelID, cccid.Name, cccid.Version, txParams.TXSimulator)      if err != nil {          return nil, err      }      // 执行调用链码的交易（和链码之间的消息为ChaincodeMessage\_TRANSACTION）      cctype := pb.ChaincodeMessage\_TRANSACTION      return cs.execute(cctype, txParams, cccid, input, h)  } |

获取链码执行环境，调用Launch函数可获取链码执行环境，即用户链码容器。如果已实例化的链码在当前背书节点上，链码容器未启动，则启动链码容器，Launch会返回一个跟链码容器交互的Handler。

|  |
| --- |
| // Launch starts executing chaincode if it is not already running. This method  // blocks until the peer side handler gets into ready state or encounters a fatal  // error. If the chaincode is already running, it simply returns.  func (cs \*ChaincodeSupport) Launch(chainID, chaincodeName, chaincodeVersion string, qe ledger.QueryExecutor) (\*Handler, error) {      cname := chaincodeName + ":" + chaincodeVersion      if h := cs.HandlerRegistry.Handler(cname); h != nil {          return h, nil      }      // 启动链码容器 ...      h := cs.HandlerRegistry.Handler(cname)      if h == nil {          return nil, errors.Wrapf(err, "[channel %s] claimed to start chaincode container for %s but could not find handler", chainID, cname)      }      return h, nil  } |

模拟执行交易

调用execute封装出执行交易的消息，然后使用Handler执行交易。

|  |
| --- |
| // execute executes a transaction and waits for it to complete until a timeout value.  func (cs \*ChaincodeSupport) execute(cctyp pb.ChaincodeMessage\_Type, txParams \*ccprovider.TransactionParams, cccid \*ccprovider.CCContext, input \*pb.ChaincodeInput, h \*Handler) (\*pb.ChaincodeMessage, error) {      input.Decorations = txParams.ProposalDecorations      // 创建消息      ccMsg, err := createCCMessage(cctyp, txParams.ChannelID, txParams.TxID, input)      if err != nil {          return nil, errors.WithMessage(err, "failed to create chaincode message")      }      // 执行交易      ccresp, err := h.Execute(txParams, cccid, ccMsg, cs.ExecuteTimeout)      if err != nil {          return nil, errors.WithMessage(err, fmt.Sprintf("error sending"))      }      return ccresp, nil  } |

Handler的执行交易过程

由于链码容器在执行交易的时候会和Peer之间进行多次通信，进行数据的读写，上下文可以让数据读写获取到正确的信息，因此先创建交易执行的上下文Context，再由Handler把消息发送到链码容器，等待链码容器发来包含执行结果的消息。

|  |
| --- |
| func (h \*Handler) Execute(txParams \*ccprovider.TransactionParams, cccid \*ccprovider.CCContext, msg \*pb.ChaincodeMessage, timeout time.Duration) (\*pb.ChaincodeMessage, error) {      chaincodeLogger.Debugf("Entry")      defer chaincodeLogger.Debugf("Exit")      // 私密数据      txParams.CollectionStore = h.getCollectionStore(msg.ChannelId)      // 是否是执行链码初始化      txParams.IsInitTransaction = (msg.Type == pb.ChaincodeMessage\_INIT)      // 创建交易context      txctx, err := h.TXContexts.Create(txParams)      if err != nil {          return nil, err      }      // 退出时（执行交易完毕），释放交易上下文资源      defer h.TXContexts.Delete(msg.ChannelId, msg.Txid)      // proposal保存到msg      if err := h.setChaincodeProposal(txParams.SignedProp, txParams.Proposal, msg); err != nil {          return nil, err      }      // 向链码容器发送msg      h.serialSendAsync(msg)      // 等待链码容器响应，或者超时      var ccresp \*pb.ChaincodeMessage      select {      case ccresp = <-txctx.ResponseNotifier:          // response is sent to user or calling chaincode. ChaincodeMessage\_ERROR          // are typically treated as error      case <-time.After(timeout):          err = errors.New("timeout expired while executing transaction")          ccName := cccid.Name + ":" + cccid.Version          h.Metrics.ExecuteTimeouts.With(              "chaincode", ccName,          ).Add(1)      }      return ccresp, err  } |

处理链码容器模拟响应

链码容器执行的响应会往上传递，直到ChaincodeSupport.Execute函数，它调用processChaincodeExecutionResult函数响应把链码容器返回的响应转化为交易模拟执行的Response，而Response最后会返回到Endorser。

|  |
| --- |
| func processChaincodeExecutionResult(txid, ccName string, resp \*pb.ChaincodeMessage, err error) (\*pb.Response, \*pb.ChaincodeEvent, error) {      if err != nil {          return nil, nil, errors.Wrapf(err, "failed to execute transaction %s", txid)      }      if resp == nil {          return nil, nil, errors.Errorf("nil response from transaction %s", txid)      }      if resp.ChaincodeEvent != nil {          resp.ChaincodeEvent.ChaincodeId = ccName          resp.ChaincodeEvent.TxId = txid      }      switch resp.Type {      // 交易执行成功则提取Payload中保存的Response      case pb.ChaincodeMessage\_COMPLETED:          res := &pb.Response{}          err := proto.Unmarshal(resp.Payload, res)          if err != nil {              return nil, nil, errors.Wrapf(err, "failed to unmarshal response for transaction %s", txid)          }          return res, resp.ChaincodeEvent, nil      //  失败，则提取Payload中保存的错误信息      case pb.ChaincodeMessage\_ERROR:          return nil, resp.ChaincodeEvent, errors.Errorf("transaction returned with failure: %s", resp.Payload)      default:          return nil, nil, errors.Errorf("unexpected response type %d for transaction %s", resp.Type, txid)      }  } |

释放模拟器资源

在Endorser.SimulateProposal中，它获取了交易模拟执行器TXSimulator，这里可是有很多的资源，如果不及时释放，在高TPS下，Peer压力大，资源泄漏，性能降低等问题会暴露出来。txParams.TXSimulator.Done()用于释放资源，主要是释放查询操作相关的资源。

ESCC处理模拟执行结果

Endorser调用ESCC对结果进行背书，最终生成ProposalResponse

|  |
| --- |
| func (e \*Endorser) ProcessProposal(ctx context.Context, signedProp \*pb.SignedProposal) (\*pb.ProposalResponse, error) {      // Pre-process, simulate      if chainID == "" {          pResp = &pb.ProposalResponse{Response: res}      } else {          // Note: To endorseProposal(), we pass the released txsim. Hence, an error would occur if we try to use this txsim          pResp, err = e.endorseProposal(ctx, chainID, txid, signedProp, prop, res, simulationResult, ccevent, hdrExt.PayloadVisibility, hdrExt.ChaincodeId, txsim, cd)          // ...      }      // ...  } |

endorseProposal

背书链码实现了可插拔，可以使用不同的ESCC，系统链码和用户链码的背书过程是不同的。

|  |
| --- |
| // endorse the proposal by calling the ESCC  func (e \*Endorser) endorseProposal(\_ context.Context, chainID string, txid string, signedProp \*pb.SignedProposal, proposal \*pb.Proposal, response \*pb.Response, simRes []byte, event \*pb.ChaincodeEvent, visibility []byte, ccid \*pb.ChaincodeID, txsim ledger.TxSimulator, cd ccprovider.ChaincodeDefinition) (\*pb.ProposalResponse, error) {      endorserLogger.Debugf("[%s][%s] Entry chaincode: %s", chainID, shorttxid(txid), ccid)      defer endorserLogger.Debugf("[%s][%s] Exit", chainID, shorttxid(txid))      // 系统链码和用户链码使用不同的ESCC      isSysCC := cd == nil      // 1) extract the name of the escc that is requested to endorse this chaincode      var escc string      // ie, "lscc" or system chaincodes      if isSysCC {          escc = "escc"      } else {          escc = cd.Endorsement()      }      endorserLogger.Debugf("[%s][%s] escc for chaincode %s is %s", chainID, shorttxid(txid), ccid, escc)      // marshalling event bytes      var err error      var eventBytes []byte      if event != nil {          eventBytes, err = putils.GetBytesChaincodeEvent(event)          if err != nil {              return nil, errors.Wrap(err, "failed to marshal event bytes")          }      }      // set version of executing chaincode      if isSysCC {          // if we want to allow mixed fabric levels we should          // set syscc version to ""          ccid.Version = util.GetSysCCVersion()      } else {          ccid.Version = cd.CCVersion()      }      // 创建背书上下文信息      ctx := Context{          PluginName:     escc, // 插件名称          Channel:        chainID,          SignedProposal: signedProp,          ChaincodeID:    ccid,          Event:          eventBytes,          SimRes:         simRes,          Response:       response,          Visibility:     visibility,          Proposal:       proposal,          TxID:           txid,      }      // 调用插件背书      return e.s.EndorseWithPlugin(ctx)  } |

背书插件实现以下的接口

|  |
| --- |
| // Plugin endorses a proposal response  type Plugin interface {      // Endorse signs the given payload(ProposalResponsePayload bytes), and optionally mutates it.      // Returns:      // The Endorsement: A signature over the payload, and an identity that is used to verify the signature      // The payload that was given as input (could be modified within this function)      // Or error on failure      Endorse(payload []byte, sp \*peer.SignedProposal) (\*peer.Endorsement, []byte, error)      // Init injects dependencies into the instance of the Plugin      Init(dependencies ...Dependency) error  } |

使用背书插件，需要获取插件实例，然后组装响应Payload，包含了交易执行的多种结果，然后对Payload以及签名的Proposal背书

|  |
| --- |
| // EndorseWithPlugin endorses the response with a plugin  func (pe \*PluginEndorser) EndorseWithPlugin(ctx Context) (\*pb.ProposalResponse, error) {      endorserLogger.Debug("Entering endorsement for", ctx)      if ctx.Response == nil {          return nil, errors.New("response is nil")      }      if ctx.Response.Status >= shim.ERRORTHRESHOLD {          return &pb.ProposalResponse{Response: ctx.Response}, nil      }      // 获取插件      plugin, err := pe.getOrCreatePlugin(PluginName(ctx.PluginName), ctx.Channel)      if err != nil {          endorserLogger.Warning("Endorsement with plugin for", ctx, " failed:", err)          return nil, errors.Errorf("plugin with name %s could not be used: %v", ctx.PluginName, err)      }      // 把模拟执行的信息组成生成背书响应Payload      prpBytes, err := proposalResponsePayloadFromContext(ctx)      if err != nil {          endorserLogger.Warning("Endorsement with plugin for", ctx, " failed:", err)          return nil, errors.Wrap(err, "failed assembling proposal response payload")      }      // 对Payload和签名的Proposal进行背书      endorsement, prpBytes, err := plugin.Endorse(prpBytes, ctx.SignedProposal)      if err != nil {          endorserLogger.Warning("Endorsement with plugin for", ctx, " failed:", err)          return nil, errors.WithStack(err)      }      resp := &pb.ProposalResponse{          Version:     1,          Endorsement: endorsement,          Payload:     prpBytes,          Response:    ctx.Response,      }      endorserLogger.Debug("Exiting", ctx)      return resp, nil  } |

系统提供的默认的背书插件如下，对交易执行结果和Proposal签名人信息进行签名。

|  |
| --- |
| // Endorse signs the given payload(ProposalResponsePayload bytes), and optionally mutates it.  // Returns:  // The Endorsement: A signature over the payload, and an identity that is used to verify the signature  // The payload that was given as input (could be modified within this function)  // Or error on failure  func (e \*DefaultEndorsement) Endorse(prpBytes []byte, sp \*peer.SignedProposal) (\*peer.Endorsement, []byte, error) {      // 提取Proposal的签名人      signer, err := e.SigningIdentityForRequest(sp)      if err != nil {          return nil, nil, errors.New(fmt.Sprintf("failed fetching signing identity: %v", err))      }      // 得到签名人身份      // serialize the signing identity      identityBytes, err := signer.Serialize()      if err != nil {          return nil, nil, errors.New(fmt.Sprintf("could not serialize the signing identity: %v", err))      }      // 对Payload和身份进行签名      // sign the concatenation of the proposal response and the serialized endorser identity with this endorser's key      signature, err := signer.Sign(append(prpBytes, identityBytes...))      if err != nil {          return nil, nil, errors.New(fmt.Sprintf("could not sign the proposal response payload: %v", err))      }      endorsement := &peer.Endorsement{Signature: signature, Endorser: identityBytes}      return endorsement, prpBytes, nil  } |

发送Response

最后调用ProcessProposal把ProposalResponse作为返回值，剩下的就交给gRPC，发送给请求放了。

**交易提交的过程**

Peer启动后在后台执行gossip服务。过程主要分三个阶段来实现。

1. 提交前准备

主要完成对区块中交易格式的检查、获取关联该区块但缺失的私密数据、最后构建blockAndPvtData结构。格式检查包括检查交易格式、账本是否存在、是否满足VSCC和Policy等。获取缺失的私密数据根据已有的私密数据计算区块中交易关联的读写集信息，若有缺失则尝试从其他节点获取。最后构建blockAndPvtData结构，用于后续的提交工作，需要包括相关的区块和私密数据。

1. 提交过程

过程包括预处理、验证交易、更新本地区块链结构、更新本地数据库结构。

* 预处理模块负责构造一个有效的内部区块结构包括处理Endorser交易（保留有效的Endorser交易）、处理配置交易（获取配置更新的模拟结果，放入写读集）、校验写集合。
* 验证交易是对区块中交易进行MVCC检查并校验私密读写集（再次检查哈希值是否匹配），更新区块元数据中的交易有效标记列表。MVCC检查需要逐个验证块中的Endorser交易，需要满足一些条件才被认为有效：公共读集合中key版本在该交易前未变、RangeQuery的结果未变、私密读集合中key版本未变。
* 更新本地区块链结构：将区块写入本地Chunk文件、更新索引数据库（区块号、哈希值、文件指针、交易偏移、区块元数据）、更新所提交的区块号到私密数据库。
* 更新本地数据库结构：删除过期私密数据、更新私密数据的生命周期记录数据库、更新本地公共状态数据库和私密状态数据库、若启用历史数据库则更新数据。

1. 提交后处理

提交后的处理阶段比较简易，只需清理本地的临时状态数据库和更新账本高度信息。清理工作包括区块关联的临时私密数据和旧区块关联的临时私密数据。

参考资料：

https://blog.csdn.net/lvyibin890/article/details/106236077

https://blog.csdn.net/m0\_43499523/article/details/104882106

\*大部分内容都是看参考资料写的，代码和图片也是从网页上截取的。\*

姓名：姚熙源

学号：3190300677