

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 区块链与数字货币 |
| 姓 名： | 庄毅非 |
| 学 院： | 计算机科学与技术学院 |
| 系： | 软件工程 |
| 专 业： | 软工2002 |
| 学 号： | 3200105872 |
| 指导教师： | 杨小虎 |

2022年 10月 29日

# 目标

分析 Fabric v2.× 源代码中相关数据结构，说明 Fabric 区块链中的参与主体、区块、交易、世界状态等数据的组成要素及这些数据的相互问关系。

# 源代码分析

以下分析基于 fabric v2.4.7，由于 fabric 使用 Protobuf 进行对象的序列化和反序列化，所以结构声明中有 XXX\_NoUnkeyedLiteral、XXX\_unrecognized、XXX\_sizecache 三个无效字段，以下分析时将其删除。

## **Block 分析（在 block data 部分的分析中含有 Transaction 结构分析）**

Block 主要由 blockHeader、transactionData 和 Metadata 三部分组成，具体结构如下[[1]](#_参考文献)。

## image-20221028192724091

在fabric 源代码中可以找到相关的结构声明语句，其中对于 Block 结构体进行声明的 go 源代码如下

|  |
| --- |
| type Block struct {  Header \*BlockHeader `protobuf:"bytes,1,opt,name=header,proto3" json:"header,omitempty"`  Data \*BlockData `protobuf:"bytes,2,opt,name=data,proto3" json:"data,omitempty"`  Metadata \*BlockMetadata `protobuf:"bytes,3,opt,name=metadata,proto3" json:"metadata,omitempty"` } |

可以看到，在 fabric 中，block 主要由 BlockHeader、BlockData 和 BlockMetadata三部分组成，这三部分组成了 fabric 区块链部分的主体。

### **2.1.1 BlockHeader**

首先，我们先观察 Header 的结构。

|  |
| --- |
| type BlockHeader struct {  Number uint64 `protobuf:"varint,1,opt,name=number,proto3" json:"number,omitempty"`  PreviousHash []byte `protobuf:"bytes,2,opt,name=previous\_hash,json=previousHash,proto3" json:"previous\_hash,omitempty"`  DataHash []byte `protobuf:"bytes,3,opt,name=data\_hash,json=dataHash,proto3" json:"data\_hash,omitempty"` } |

这些属性都在 order 节点创建新区块的时候被赋值，具体代码如下。

|  |
| --- |
| func (bg \*blkGenerator) nextBlock() \*common.Block {  ...  // 以下三行对header中的属性进行赋值  block.Header.DataHash = protoutil.BlockDataHash(block.Data)  block.Header.Number = bg.blockNum  block.Header.PreviousHash = bg.previousBlockHash  ...  bg.blockNum++  bg.previousBlockHash = protoutil.BlockHeaderHash(block.Header)  return block } |

可以看到，在生成新区块的时候，新区块的头中的 Number 被赋值为其在 fabric chain 中区块的下标，能够表示这是区块链中的第几个区块； DataHash 则是对区块本身的 Data 部分进行了 Hash 操作所得到的结果；最后对 previousHash 进行了赋值，将这个属性保存为前一个区块的 hash 值，这样就实现了当前区块和前一个区块之间的数据绑定，保证存储的交易数据不会被篡改，值得注意的是，这里执行的 hash 函数的输入中并不包含元数据，这是因为元数据是在创建区块的时候一并生成的。

值得一提的是，在 nextBlock 函数的最后，bg 还对 blockNum 进行了自增操作，将 previousHash的值修改为对当前 Header 进行 Hash 的结果，为下一次产生新区块进行数据的更新。

### **2.1.2 BlockMetaData**

接下来，我们分析 BlockMetaData 的结构。

|  |
| --- |
| type BlockMetadata struct {  Metadata [][]byte `protobuf:"bytes,1,rep,name=metadata,proto3" json:"metadata,omitempty"` } |

在源代码中，我们可以发现 BlockMetadata 实际上就是一个二维数组，用来存储包括产生区块的节点的签名等信息，具体包含内容在 commom.go 文件中可以找到。

|  |
| --- |
| const (  BlockMetadataIndex\_SIGNATURES BlockMetadataIndex = 0  BlockMetadataIndex\_LAST\_CONFIG BlockMetadataIndex = 1 *// Deprecated: Do not use.* BlockMetadataIndex\_TRANSACTIONS\_FILTER BlockMetadataIndex = 2  BlockMetadataIndex\_ORDERER BlockMetadataIndex = 3 *// Deprecated: Do not use.* BlockMetadataIndex\_COMMIT\_HASH BlockMetadataIndex = 4 )  var BlockMetadataIndex\_name = map[int32]string{  0: "SIGNATURES",  1: "LAST\_CONFIG",  2: "TRANSACTIONS\_FILTER",  3: "ORDERER",  4: "COMMIT\_HASH", } |

应该注意的是，在 2.3.0 版本的 Fabric 中，BlockMetadataIndex\_LAST\_CONFIG 和BlockMetadataIndex\_ORDERER 已经被弃用，还在使用的只有 BlockMetadataIndex\_SIGNATURES，BlockMetadataIndex\_TRANSACTIONS\_FILTER，BlockMetadataIndex\_COMMIT\_HASH，接下来将对这三者的含义进行解释。

#### **2.1.2.1 BlockMetadataIndex\_SIGNATURES**

对该属性赋值的代码如下。

|  |
| --- |
| *// Block constructs and returns a genesis block for a given channel ID.* func (f \*factory) Block(channelID string) \*cb.Block {  ...  // 这里对两个属性进行赋值  block.Metadata.Metadata[cb.BlockMetadataIndex\_LAST\_CONFIG] = protoutil.MarshalOrPanic(&cb.Metadata{  Value: protoutil.MarshalOrPanic(&cb.LastConfig{Index: 0}),  })  block.Metadata.Metadata[cb.BlockMetadataIndex\_SIGNATURES] = protoutil.MarshalOrPanic(&cb.Metadata{  Value: protoutil.MarshalOrPanic(&cb.OrdererBlockMetadata{  LastConfig: &cb.LastConfig{Index: 0},  }),  })  return block } |

可以看到，这里存储了 Metadata[BlockMetadataIndex\_SISNATURES] ，这是区块创建者对区块进行的签名。

#### **2.1.2.2 BlockMetadataIndex\_COMMIT\_HASH**

这个属性在创建区块的时候产生，包括对提交区块（txValidation Code 长度、txValidationCode列表、所有写操作的排序字节组、上个区块的提交 Hash 值）的 SHA256 Hash 值。在 Peer 节点提交时候添加。

#### 2.1.2.3 BlockMetadataIndex\_TRANSACTIONS\_FILTER

对这个属性的赋值同样发生在创建新区块的时候，具体操作如下。

|  |
| --- |
| *// 这里的txsfltr存储的是本区块包含的所有交易的状态（即，是否有效）* txsfltr := txflags.NewWithValues(len(block.Data.Data), peer.TxValidationCode\_VALID) *// 将交易状态进行存储* block.Metadata.Metadata[common.BlockMetadataIndex\_TRANSACTIONS\_FILTER] = txsfltr |

### 2.1.3 BlockData

最后，我们对BlockData数据进行分析，这也是区块中比较核心的一部分，其中含有交易的列表。BlockData结构的声明如下。

|  |
| --- |
| type BlockData struct {  Data [][]byte `protobuf:"bytes,1,rep,name=data,proto3" json:"data,omitempty"` } |

可以看到，BlockData实际上就是一个二维数组，这个二维数组实际上是使用Envelope结构序列化得到的。具体序列化代码如下。

|  |
| --- |
| *// Block constructs and returns a genesis block for a given channel ID.* func (f \*factory) Block(channelID string) \*cb.Block {  // 创建payload所需的两个header  payloadChannelHeader := protoutil.MakeChannelHeader(cb.HeaderType\_CONFIG, msgVersion, channelID, epoch)  payloadSignatureHeader := protoutil.MakeSignatureHeader(nil, protoutil.CreateNonceOrPanic())  protoutil.SetTxID(payloadChannelHeader, payloadSignatureHeader)  // 创建payload header  payloadHeader := protoutil.MakePayloadHeader(payloadChannelHeader, payloadSignatureHeader)  // 创建payload  payload := &cb.Payload{Header: payloadHeader, Data: protoutil.MarshalOrPanic(&cb.ConfigEnvelope{Config: &cb.Config{ChannelGroup: f.channelGroup}})}  // 将payload封装到envelope中  envelope := &cb.Envelope{Payload: protoutil.MarshalOrPanic(payload), Signature: nil}   block := protoutil.NewBlock(0, nil)  // 将envelope序列化为字节流，存储到对应的blockdata中  block.Data = &cb.BlockData{Data: [][]byte{protoutil.MarshalOrPanic(envelope)}}  ... } |

这里的envelope本身也是由payload和signature进行序列化得到的，结构定义如下。

|  |
| --- |
| *// Envelope wraps a Payload with a signature so that the message may be authenticated* type Envelope struct {  *// A marshaled Payload* Payload []byte `protobuf:"bytes,1,opt,name=payload,proto3" json:"payload,omitempty"`  *// A signature by the creator specified in the Payload header* Signature []byte `protobuf:"bytes,2,opt,name=signature,proto3" json:"signature,omitempty"` } |

其中signature顾名思义，就是对payload数据的签名，用来对经过网络传输得到的区块进行有效性验证。对payload属性来说，其存储了具体的交易数据，其结构声明如下。

|  |
| --- |
| *// Payload is the message contents (and header to allow for signing)* type Payload struct {  *// Header is included to provide identity and prevent replay* Header \*Header `protobuf:"bytes,1,opt,name=header,proto3" json:"header,omitempty"`  *// Data, the encoding of which is defined by the type in the header* Data []byte `protobuf:"bytes,2,opt,name=data,proto3" json:"data,omitempty"` } |

接下来对其中的Header结构和Data结构进行分析。

#### 2.1.3.1 Header

header的结构声明如下。

|  |
| --- |
| type Header struct {  ChannelHeader []byte `protobuf:"bytes,1,opt,name=channel\_header,json=channelHeader,proto3" json:"channel\_header,omitempty"`  SignatureHeader []byte `protobuf:"bytes,2,opt,name=signature\_header,json=signatureHeader,proto3" json:"signature\_header,omitempty"` } |

通过Header结构的定义，我们能够发现其由ChannelHeader和SignatureHeader两部分组成，其中ChannelHeader结构的声明如下。

|  |
| --- |
| type ChannelHeader struct {  Type int32 `protobuf:"varint,1,opt,name=type,proto3" json:"type,omitempty"`  *// 存储的是信息协议的版本号* Version int32 `protobuf:"varint,2,opt,name=version,proto3" json:"version,omitempty"`  *// 存储的是发送时间* Timestamp \*timestamp.Timestamp `protobuf:"bytes,3,opt,name=timestamp,proto3" json:"timestamp,omitempty"`  *// ChannelID表示这个信息要发送的目标channelid，由于frabic具有隐私保护功能，其他channel的节点不会收到这个消息* ChannelId string `protobuf:"bytes,4,opt,name=channel\_id,json=channelId,proto3" json:"channel\_id,omitempty"`  *// TxID存储的是一个全剧唯一性的ID，这是一个更高层指定的id，在传输给endoser进行背书的时候，endoser会检查这个id的唯一性，在传输给commiter进行提交的时候，这个id的唯一性也会被检查，最终这个id会被存储在账本中* TxId string `protobuf:"bytes,5,opt,name=tx\_id,json=txId,proto3" json:"tx\_id,omitempty"` *// Epoch字段指定了本消息的时间窗口，如果目的节点收到本消息的时间并不在这个时间窗口中，或者本消息在这个时间窗口中出现了多次（这是为了预防replay攻击），那么这个交易都会被拒绝*  Epoch uint64 `protobuf:"varint,6,opt,name=epoch,proto3" json:"epoch,omitempty"`  *// extension用来存储上述不同type可能所具有的独有数据*  Extension []byte `protobuf:"bytes,7,opt,name=extension,proto3" json:"extension,omitempty"`  *// tlsCertHash指定了tls证书，只有在使用tls通信的时候使用。*  TlsCertHash []byte `protobuf:"bytes,8,opt,name=tls\_cert\_hash,json=tlsCertHash,proto3" json:"tls\_cert\_hash,omitempty"` } |

可以看到，ChannelHeader中存储了几个描述Channel的关键属性，其中Type描述了PayLoad包含的信息种类，这些种类在一个enum中声明，通过查阅文档，可以发现type可以取如下几个值。

| **取值** | **含义** |
| --- | --- |
| MESSAGE | 不透明的消息类型 |
| CONFIG | 对本Channel进行配置 |
| CONFIG\_UPDATE | 更新本Channel的配置 |
| ENDORSER\_TRANSACTION | Client通过sdk向endoser提交提案 |
| ORDERER\_TRANSACTION | order节点保留使用 |
| DELIVER\_SEEK\_INFO | 用作deliver api信息查询 |
| CHAINCODE\_PACKAGE | 说明是对链码的打包 |
| PEER\_RESOURCE\_UPDATE | 对peer资源进行更新 |

SignatureHeader的结构如下。

|  |
| --- |
| type SignatureHeader struct {  *// Creator of the message, a marshaled msp.SerializedIdentity* Creator []byte `protobuf:"bytes,1,opt,name=creator,proto3" json:"creator,omitempty"`  *// Arbitrary number that may only be used once. Can be used to detect replay attacks.* Nonce []byte `protobuf:"bytes,2,opt,name=nonce,proto3" json:"nonce,omitempty"` } |

其中Nonce是一个随机生成的数字，用来抵御replay攻击；Creator由signingidentity序列化得到，signingidentity的结构声明如下

|  |
| --- |
| type signingidentity struct {identitysigner crypto.Signer } |

这里的identity存储了包括公钥在内的身份证明信息，而这里的Signer则是一个对接口的声明，用来向上层提供签名和提供公钥服务。

#### 2.1.3.2 Data

这个字段存储了交易的主要信息，具体来说，这是由Transaction结构体序列化得到的，Transaction结构体的声明如下。

|  |
| --- |
| type Transaction struct {Actions []\*TransactionAction `protobuf:"bytes,1,rep,name=actions,proto3" json:"actions,omitempty"` } |

其具有如下的特性：

1. 每一个transaction都是原子性的，即要么所有提案全部通过，要么所有提案全部不通过，并且一个transaction中的所有提案都必须来自同一个创建者。
2. 每一个transaction中可以有若干个TransactionAction对象，每一个TransactionAction实际上都对应一个提案，这个提案会被发送给特定的背书节点供其检查和签名。
3. 我们能够发现，Transaction中存储的实际上是一个TransactionAction列表，TransactionAction的结构声明如下。

|  |
| --- |
| type TransactionAction struct {  *// The header of the proposal action, which is the proposal header* Header []byte `protobuf:"bytes,1,opt,name=header,proto3" json:"header,omitempty"`  *// The payload of the action as defined by the type in the header For  // chaincode, it's the bytes of ChaincodeActionPayload* Payload []byte `protobuf:"bytes,2,opt,name=payload,proto3" json:"payload,omitempty"` } |

通过注释可以发现，这里的属性header指定了提案的类型，而属性payload实际上是这个交易对应的一个ChaincodeActionPayload对象的一个序列化结果，ChaincodeActionPayload的结构声明如下

|  |
| --- |
| type ChaincodeActionPayload struct {ChaincodeProposalPayload []byte `protobuf:"bytes,1,opt,name=chaincode\_proposal\_payload,json=chaincodeProposalPayload,proto3" json:"chaincode\_proposal\_payload,omitempty"`  *// The list of actions to apply to the ledger* Action \*ChaincodeEndorsedAction `protobuf:"bytes,2,opt,name=action,proto3" json:"action,omitempty"` } |

接下来对这两个属性进行分析：

##### 2.1.3.2.1 ChaincodeProposalPayload

如果客户端想要调用对应的chaincode的话，它就会将TransactionAction中的Header中的type设定为CHAINCODE，这样ChaincodeActionPayload就会携带调用参数和调用在这个提案上的若干个背书行为的列表。这里的ChaincodeProposalPayload属性是ChaincodeProposalPayload对象的一个序列化结果，ChaincodeProposalPayload类型的定义如下所示。

|  |
| --- |
| type ChaincodeProposalPayload struct {Input []byte `protobuf:"bytes,1,opt,name=input,proto3" json:"input,omitempty"`TransientMap map[string][]byte `protobuf:"bytes,2,rep,name=TransientMap,proto3" json:"TransientMap,omitempty" protobuf\_key:"bytes,1,opt,name=key,proto3" protobuf\_val:"bytes,2,opt,name=value,proto3"` } |

ChaincodeProposalPayload结构中比较重要的是这个Input属性，它是ChaincodeInvocationSpec的序列化结果，ChaincodeInvocationSpec声明如下。

|  |
| --- |
| type ChaincodeInvocationSpec struct {  ChaincodeSpec \*ChaincodeSpec `protobuf:"bytes,1,opt,name=chaincode\_spec,json=chaincodeSpec,proto3" json:"chaincode\_spec,omitempty"` } |

其中的ChaincodeSpec是一个ChaincodeSpec数组，我们可以看到ChaincodeSpec结构中存储了2个对chaincode调用过程至关重要的参数ChaincodeId和Input，以下主要对其进行分析。

|  |
| --- |
| type ChaincodeSpec struct {  Type ChaincodeSpec\_Type `protobuf:"varint,1,opt,name=type,proto3,enum=protos.ChaincodeSpec\_Type" json:"type,omitempty"`  ChaincodeId \*ChaincodeID `protobuf:"bytes,2,opt,name=chaincode\_id,json=chaincodeId,proto3" json:"chaincode\_id,omitempty"`  Input \*ChaincodeInput `protobuf:"bytes,3,opt,name=input,proto3" json:"input,omitempty"`  Timeout int32 `protobuf:"varint,4,opt,name=timeout,proto3" json:"timeout,omitempty"` } |

* ChaincodeId

这个属性用来表征客户端调用的是哪一个chaincode，其中含有PATH，Name和Version三个字段，其中PATH是在部署chaincode的时候使用的一个属性，Name是对应chaincode的hash结果，用来表示调用的是哪个chaincode，Version表示chaincode的版本。通过以上三个属性，fabric channel能够确定client调用的是哪一个chaincode。

* Input

这个属性存储了调用的函数和对应的参数的数组，初始化源代码如下。

|  |
| --- |
| func (c \*chaincodeInput) UnmarshalJSON(b []byte) error {  sa := struct {  Function string  Args []string  }{}  err := json.Unmarshal(b, &sa)  if err != nil {  return err  }  allArgs := sa.Args  if sa.Function != "" {  allArgs = append([]string{sa.Function}, sa.Args...)  }  c.Args = util.ToChaincodeArgs(allArgs...)  return nil } |

可以看到，实际上input数组存储的就是要调用的chaincode中的function名和function所需的参数表，peer通过检查这个属性确定使用什么参数调用对应的function。

##### 2.1.2.3.2 Action

这是一个ChaincodeEndorsedAction组成的列表，用来实现peer节点对提案的背书，ChaincodeEndorsedAction的结构声明如下。

|  |
| --- |
| type ChaincodeEndorsedAction struct {ProposalResponsePayload []byte `protobuf:"bytes,1,opt,name=proposal\_response\_payload,json=proposalResponsePayload,proto3" json:"proposal\_response\_payload,omitempty"`Endorsements []\*Endorsement `protobuf:"bytes,2,rep,name=endorsements,proto3" json:"endorsements,omitempty"` } |

其中的proposalResponsePayload由createProposalResponsePayload函数生成，这个函数主要功能就是将执行该chaincode所对应的读写集存储在ProposalResponsePayload的extension属性中，而这里的proposalHash通常是对整个提案进行hash运算的结果，引入目的是将response和其对应的提案进行关联。

|  |
| --- |
| type ProposalResponsePayload struct {ProposalHash []byte `protobuf:"bytes,1,opt,name=proposal\_hash,json=proposalHash,proto3" json:"proposal\_hash,omitempty"`Extension []byte `protobuf:"bytes,2,opt,name=extension,proto3" json:"extension,omitempty"` } |

而endoesement结构的定义如下，可以发现其中含有背书者的身份和对应的签名，用来实现对提案的签名功能。

|  |
| --- |
| type Endorsement struct {Endorser []byte `protobuf:"bytes,1,opt,name=endorser,proto3" json:"endorser,omitempty"`Signature []byte `protobuf:"bytes,2,opt,name=signature,proto3" json:"signature,omitempty"` } |

至此，我们完成了对Block和transaction结构的分析。

## 世界状态分析

世界状态相关的接口为为VersionedDB，声明如下。

|  |
| --- |
| *// VersionedDB lists methods that a db is supposed to implement* type VersionedDB interface {  // 通过给定的命名空间和key获取对应的valueGetState(namespace string, key string) (\*VersionedValue, error)  // 通过命名空间和key获取版本  GetVersion(namespace string, key string) (\*version.Height, error)  *// 获取给定命名空间下多个key对应的value列表*  GetStateMultipleKeys(namespace string, keys []string) ([]\*VersionedValue, error)  *// 获取能够遍历给定范围的迭代器* GetStateRangeScanIterator(namespace string, startKey string, endKey string) (ResultsIterator, error)  *// 获取能够遍历给定范围的迭代器，并且限制返回的结果数* GetStateRangeScanIteratorWithPagination(namespace string, startKey string, endKey string, pageSize int32) (QueryResultsIterator, error)  *// 执行查询指令，返回能够遍历结果列表的迭代器* ExecuteQuery(namespace, query string) (ResultsIterator, error)  *// 执行查询指令，返回能够遍历结果列表的迭代器，限制返回的结果数* ExecuteQueryWithPagination(namespace, query, bookmark string, pageSize int32) (QueryResultsIterator, error)  *// 执行对应的更新指令* ApplyUpdates(batch \*UpdateBatch, height \*version.Height) error  *// 获取使数据库保持一致性的最新的一个事务（这是对数据库来说的事务）* GetLatestSavePoint() (\*version.Height, error)  *// 测试对应的key和value是否被数据库支持* ValidateKeyValue(key string, value []byte) error  *// 测试数据库是否支持任意属性的序列化结果作为key* BytesKeySupported() bool  *// 获取能够遍历所有key-value对的迭代器* GetFullScanIterator(skipNamespace func(string) bool) (FullScanIterator, error)  *// 启动数据库* Open() error  *// 关闭数据库* Close() } |

|  |
| --- |
| // 用来遍历返回的结果  type ResultsIterator interface {  Next() (\*VersionedKV, error)  Close() }  // 使查询结果能够支持多种类型  type queryResult interface{}  *// 对namespace和key组合得到的复合键* type CompositeKey struct {  Namespace string  Key string }  *// 用来返回指定状态的查询结果*  type VersionedValue struct {  Value []byte  Metadata []byte  Version \*version.Height } |

## 参与交易的其他类型的结构

### 2.3.1 PEER

|  |
| --- |
| type Peer struct {  // 存储使用的grpc服务器的配置信息  ServerConfig comm.ServerConfig  // 处理grpc服务器的鉴权工作  CredentialSupport \*comm.CredentialSupport  // 存储世界状态  StoreProvider transientstore.StoreProvider  // 节点之间使用gossip协议进行状态同步和分发  GossipService \*gossipservice.GossipService  // 管理账本  LedgerMgr \*ledgermgmt.LedgerMgr  // 存储所管理的终端的列表  OrdererEndpointOverrides map[string]\*orderers.Endpoint  // 提供所需的加密服务  CryptoProvider bccsp.BCCSP   *// 限制go语言协程个数，这是go语言处理并发请求的机制之一*  validationWorkersSemaphore semaphore.Semaphore   // 提供grpc服务的服务器信息  server \*comm.GRPCServer  // 管理各种插件  pluginMapper plugin.Mapper  // 进行channel初始化的函数  channelInitializer func(cid string)   *// channel是从channelID到channel本体的映射* mutex sync.RWMutex  channels map[string]\*Channel   configCallbacks []channelconfig.BundleActor } |

peer所定义的部分功能函数

|  |
| --- |
| // 更新信任的根证书分发中心，用来增加ca认证服务器  updateTrustedRoots(cm channelconfig.Resources)  // 创建channel  CreateChannel(cid string, cb \*common.Block, deployedCCInfoProvider ledger.DeployedChaincodeInfoProvider, legacyLifecycleValidation plugindispatcher.LifecycleResources, newLifecycleValidation plugindispatcher.CollectionAndLifecycleResources) error  // 获取channel信息  GetChannelConfig(cid string) channelconfig.Resources  // 获取账本  GetLedger(cid string) ledger.PeerLedger |

### 2.3.2 Channel

|  |
| --- |
| // 管理一个channel的结构  type Channel struct {  // channel 对应的账本  ledger ledger.PeerLedger  // channel存储机制  store \*transientstore.Store  // 提供密码学函数的成员变量  cryptoProvider bccsp.BCCSP  applyLock sync.Mutex  bundleSource \*channelconfig.BundleSource  lock sync.RWMutex  resources channelconfig.Resources  } |

### 2.3.3 ENDOSER

|  |
| --- |
| // Endorser provides the Endorser service ProcessProposal  type Endorser struct {  ChannelFetcher ChannelFetcher  LocalMSP msp.IdentityDeserializer  PrivateDataDistributor PrivateDataDistributor  Support Support  PvtRWSetAssembler PvtRWSetAssembler  Metrics \*Metrics  } |

在提供背书服务的时候，endoser节点会触发ProcessProposal方法处理对应的提案，这个函数接受一个签名提案，之后会返回一个对提案的响应。执行逻辑如下：

1. endoser会调用preprocess函数对提案进行预处理，对signedHeader和ChannelHeader两个header字段进行验证。对ChannelHeader来说，要求channelHeader非空，channelHeader.Type必须是ENDORSER\_TRANSACTION、CONFIG\_UPDATE、CONFIG、TOKEN\_TRANSACTION四种类型中的一种，并且channel.Epoch也非空（这是为了避免replay攻击）；也会检查SignedHeader，要求其非空，nonce非空并且creator非空，
2. endoser随后调用checkSignatureFromCreator函数对提案的签名进行验证，检查是否是对应的creator发出的提案。
3. endoser进入账本，查询是否已经有该txid对应的交易存在，避免replay攻击
4. 判断是否是不可调用的系统类型chaincode，如果是，那么返回错误信息，否则检查提案中的权限，判定器其能否调用对应的chaincode。
5. 如果上述流程都正常通过，那么preprocess函数正确返回，在主函数ProcessProposal中，endoser会调用SimulateProposal函数进行模拟提案执行，使用callChaincode获取提案执行结果（也就是所需的读写集），最后执行endoseProposal对提案签名，完成背书，将响应结构体ProposalResponse返回给client对应的peer。

# 参考文献

[1]: Zhou, Ence, Haoli Sun, Bingfeng Pi, Jun Sun, Kazuhiro Yamashita and Yoshihide Nomura. “Ledgerdata Refiner: A Powerful Ledger Data Query Platform for Hyperledger Fabric.” 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS) (2019): 433-440.