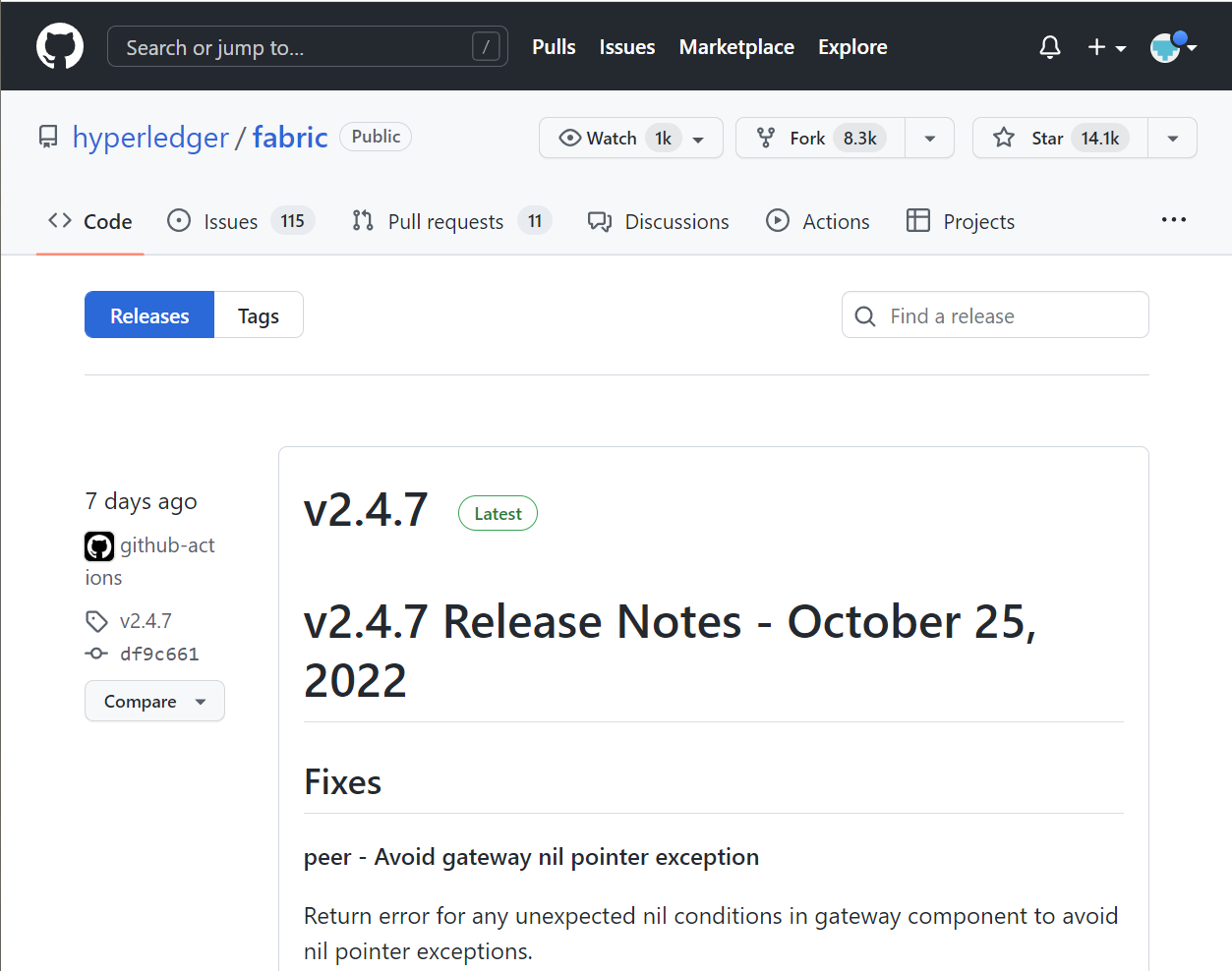
Homework 3

张天涵 3200105746

1. 前言

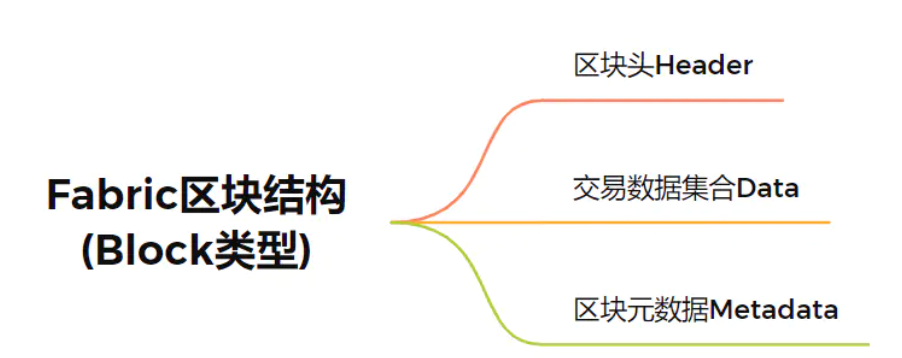
本次作业所参考的源代码是hyperledger在2022.10.25发行的最新版本代码——fabric v2.4.7，基于以上代码进行数据结构的分析，具有一定的时效性。【鉴于篇幅所限，本文代码引用均隐藏了一定量的细节】



1. 数据结构以及相关联系

2.1 block 区块

相关注解已在代码区域写下



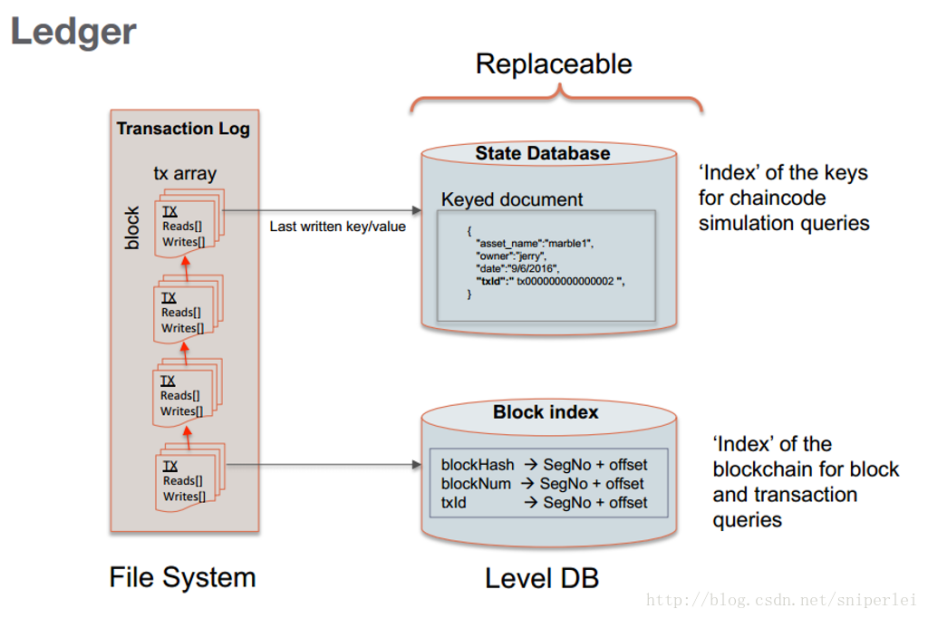
1. type Block struct {
2. Header \*BlockHeader   //区块头
3. Data \*BlockData     //区块数据 一个有序的交易列表。区块数据是在排序服务创建区块时被写入
4. Metadata \*BlockMetadata //元数据 区块被写入的时间等信息 与区块一起形成
5. }
6. type Metadata struct {
7. Value= []byte //
8. Signatures  []\*MetadataSignature
9. }
10. type Header struct {
11. ChannelHeader []byte
12. SignatureHeader  []byte
13. }
14. type BlockHeader struct {
15. Number uint64  //区块编号
16. PreviousHash []byte  //前一个区块头的哈希
17. DataHash []byte // 当前区块数据的哈希 不包括元数据
18. }

2.2 peer节点

1. type Peer struct {
2. ServerConfig             comm.ServerConfig
3. CredentialSupport        \*comm.CredentialSupport
4. StoreProvider            transientstore.StoreProvider
5. GossipService            \*gossipservice.GossipService
6. LedgerMgr                \*ledgermgmt.LedgerMgr
7. OrdererEndpointOverrides map[string]\*orderers.Endpoint
8. CryptoProvider           bccsp.BCCSP
9. validationWorkersSemaphore semaphore.Semaphore
10. server             \*comm.GRPCServer
11. pluginMapper       plugin.Mapper
12. channelInitializer func(cid string)
13. mutex    sync.RWMutex
14. channels map[string]\*Channel
15. configCallbacks []channelconfig.BundleActor
16. }

其中comm.ServerConfig是一些所有参与主体都具有的常规配置，比如id，network，address，等等配置，在之后不再赘述.CredentialSupport是认证服务，主要是身份验证等功能相关，LedgerMgr使用要创建本地的账本数据库对象以及相关服务，GossipService是用gossip方式寻找节点并达成共识的一种途径。 OrdererEndpointOverrides储存了与之相关的orderer节点，在之后的过程中产生联系。

Ledger在此处指的是账本数据，由Orderer 节点创建，然后由Orderer节点发送给每一个Peer，每一个Peer维护一个Ledger的副本。



2.3 endorser节点（背书节点）

1. type Endorser struct {
2. ChannelFetcher         ChannelFetcher
3. LocalMSP               msp.IdentityDeserializer
4. PrivateDataDistributor PrivateDataDistributor
5. Support                Support
6. PvtRWSetAssembler      PvtRWSetAssembler
7. Metrics                \*Metrics
8. }

其中channelfetcher用来部署本节点所属的链，其余分别执行相关服务。

1. func (e \*Endorser) callChaincode(txParams \*ccprovider.TransactionParams, input \*pb.ChaincodeInput, chaincodeName string) (\*pb.Response, \*pb.ChaincodeEvent, error) {
2. }(time.Now())
3. func (e \*Endorser) simulateProposal(txParams \*ccprovider.TransactionParams, chaincodeName string, chaincodeInput \*pb.ChaincodeInput) (\*pb.Response, []byte, \*pb.ChaincodeEvent, \*pb.ChaincodeInterest, error) {
4. }
5. func (e \*Endorser) preProcess(up \*UnpackedProposal, channel \*Channel) error {
6. err := up.Validate(channel.IdentityDeserializer)
7. }
8. func (e \*Endorser) ProcessProposal(ctx context.Context, signedProp \*pb.SignedProposal) (\*pb.ProposalResponse, error) {
9. }

在peer启动时会创建Endorser背书服务器，并注册到gRPC服务器对外提供服务，客户端发送一个signalproposal到endorser节点之后，首先模拟处理提案，之后调用preProcess方法去检查和检验提案的合法性，验证交易提案格式是否正确，交易的唯一性，验证是否满足对应通道的访问控制策略，验证客户端签名是否有效，验证请求者在通道内是否具有相应的权限。这之后执行背书操作，调用ProcessProposal方法，传回一个ProcessProposal给peer节点，以上是主要功能，在之后还会调用一些函数，比如

func (e \*Endorser) ProcessProposalSuccessfullyOrError(up\*UnpackedProposal) (\*pb.ProposalResponse, error)等对此次背书行为做出补充说明和信息传递。

在此补充说明一些proposal的数据结构

SingnedProposal

1. type SignedProposal struct {
2. *// The bytes of Proposal*
3. ProposalBytes []byte
4. Signature  []byte
5. }

ProposalResponse

1. type ProposalResponse struct {
2. Version int32
3. Timestamp \*timestamp.Timestamp
4. Response \*Response  Payload []byte
5. Endorsement \*Endorsement
6. Interest \*ChaincodeInterest
8. }

ProposalResponsePayload

1. type ProposalResponsePayload struct {
2. ProposalHash []byte
3. Extension []byte
4. }

值得注意的是，ProposalResponsePayload是客户以及背书节点之间的桥梁，包含状态变化以及事件的哈希值。

2.3 orderer节点 对交易排序

客户端收到消息和签名之后会广播给排序节点，排序节点对交易排序并打包成区块，排序服务之中有重要功能模块，broadcast和deliver。Broadcast的主要功能是接收来自客户端的交易请求，对客户端发送过来的数据格式进行校验，同时也会对客户端的访问权限进行检查，然后再尝试将请求打包给共识组件进行排序。如下述两个函数接口，分别是定义初始化broadcast相关实例，之后对传来的信息进行加工，打包请求发送给共识组件并进行排序。

1. type Consenter interface {
2. Order(env \*cb.Envelope, configSeq uint64) error
3. Configure(config \*cb.Envelope, configSeq uint64) error
4. WaitReady() error
5. }
6. func (bh \*Handler) ProcessMessage(msg \*cb.Envelope, addr string) (resp \*ab.BroadcastResponse) {
7. tracker := &MetricsTracker{
8. ChannelID: "unknown",
9. TxType:    "unknown",
10. Metrics:   bh.Metrics,
11. }
12. logger.Debugf("[channel: %s] Broadcast has successfully enqueued message of type %s from %s", chdr.ChannelId, cb.HeaderType\_name[chdr.Type], addr)
13. return &ab.BroadcastResponse{Status: cb.Status\_SUCCESS}
14. }

Broadcast和deliver在server.go文件之中启动，启动服务的代码数据结构如下：两者的功能：broadcast把client发来的需要排序的信息接受，而deliver的作用是把排序完的信息打包发回给client。这两个模块实现了和client的信息发送接收，而共识机制进行对信息的排序。

1. func (s \*server) Broadcast(srv ab.AtomicBroadcast\_BroadcastServer) error {
2. logger.Debugf("Starting new Broadcast handler")
3. return s.bh.Handle(&broadcastMsgTracer{
4. },
5. })
6. }
7. func (s \*server) Deliver(srv ab.AtomicBroadcast\_DeliverServer) error {
8. logger.Debugf("Starting new Deliver handler")
9. policyChecker := func(env \*cb.Envelope, channelID string) error {
10. return sf.Apply(env)
11. }
12. return s.dh.Handle(srv.Context(), deliverServer)
13. }

之后介绍共识机制，与排序有关，共识机制：共识机制一共有三种：

1. Solo 已弃用：上课讲过主要是可以作为课堂练习等用途，无法处理实际生活中的情况。【但是在】
2. Kafka fabric-2.4.7\orderer\consensus\kafka

以下代码是processMessageToBlocks()，也就是kafka关于排序打包交易的一个主线循环，基本上是由不同的情况(select case)来对应不同的方法，完成排序打包。大的结构上属于for循环。

1. func (chain \*chainImpl) processMessagesToBlocks() ([]uint64, error) {
2. defer func() { *// When Halt() is called*
3. }()
4. for {
5. select {
6. case <-chain.haltChan:
7. case <-chain.errorChan: *// If already closed, don't do anything*
8. default:
9. }
10. select {
11. case <-chain.errorChan:
12. default:
13. }
14. case <-topicPartitionSubscriptionResumed:
15. case <-deliverSessionTimedOut:
16. case in, ok := <-chain.channelConsumer.Messages():
17. if !ok {
18. logger.Criticalf("[channel: %s] Kafka consumer closed.", chain.ChannelID())
19. return counts, nil
20. }
21. }
22. select {
23. case <-chain.errorChan:
24. default:
25. }   }
26. Raft fabric-2.4.7\orderer\consensus\etcdraft\chain.go 与kafka不相同的是，这个处理的函数存在在func (c \*Chain) run()之中，其中基本也是for循环搭配select语句完成对交易的排序打包。

2.4 Transaction 交易

以下为transaction的数据结构，文件位于fabric-2.4.7\vendor\github.com

\hyperledger\fabric-protos-go\peer\transaction.pb.go 之中

1. type Transaction struct {
2. Actions              []\*TransactionAction
3. }
4. type TransactionAction struct {
5. Header []byte
6. Payload []byte
7. }

交易分为交易行为数组，以及单个交易，交易行为是一个交易的数组，交易之中有交易头以及payload信息。payload字段包括chaincode\_proposal\_payload（背书提案时调用链码的信息）和action字段。如下述所示：

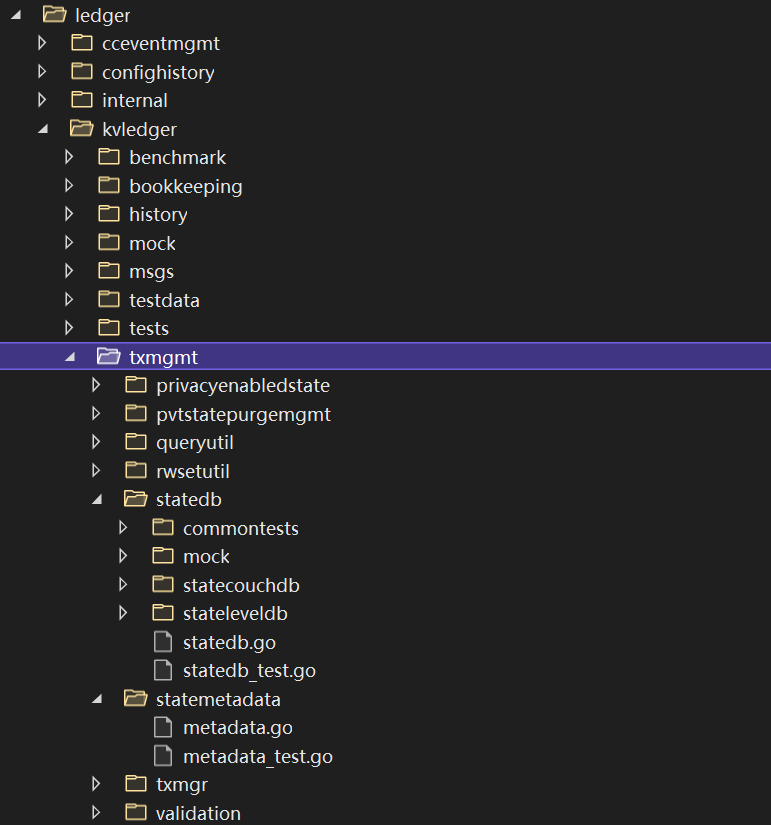
Payload：

1. type Payload struct {
2. Header \*Header
3. Data  []byte
4. }

2.5 world state世界状态

这里可以看到key-value关系以及更加详细的更新数据

1. *// SnapshotWriter generates two files, a data file and a metadata file.*
2. The datafile contains a series of tuples <key, dbValue>
3. *// and the metadata file contains a series of tuples*
4. <namesapce, number-of-tuples-in-the-data-file-that-belong-to-this-namespace>



在账本文件夹之中，部署了不同和数据库以及相关接口，世界状态数据库，源数据库，历史数据库等等，里面有不同的数据库提供者，初始化以及实例化调用的方法。