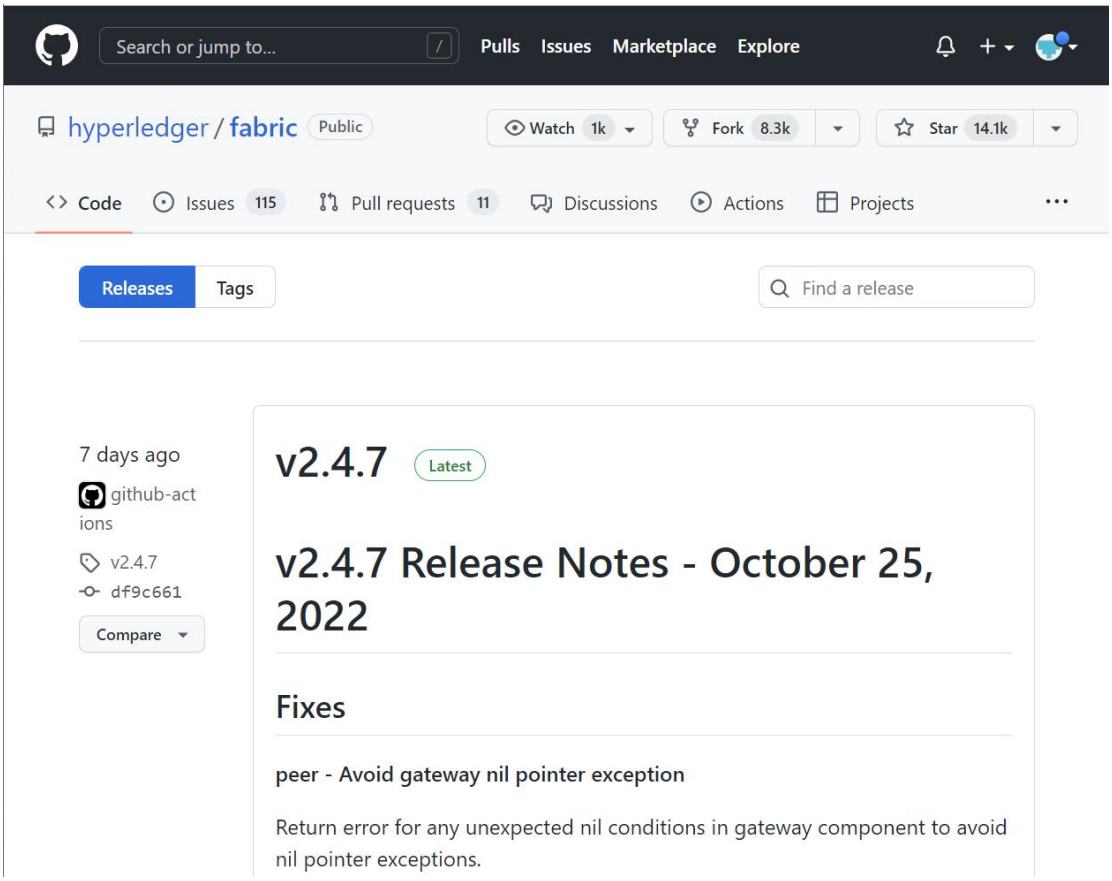
Homework 3

张天涵 3200105746

一． 前言

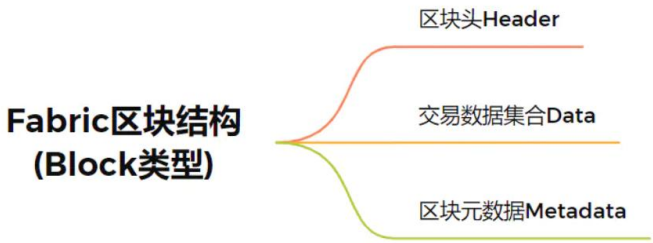
本次作业所参考的源代码是 hyperledger 在 2022.10.25 发行的最新版本代码——fabric v2.4.7，基于以上代码进行数据结构的分析，具有一定的时效性。【鉴于篇幅所限，本文代码引用均隐藏了一定量的细节】



二． 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



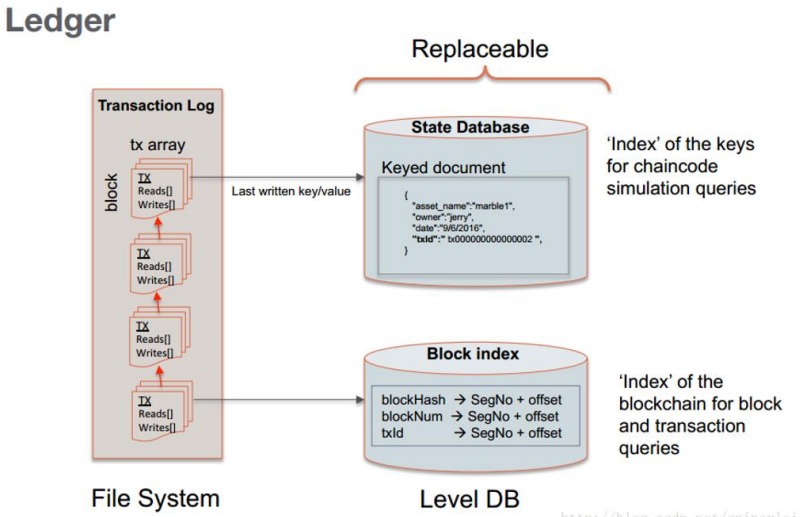

```
type Block struct {
 Header *BlockHeader   //区块头
 Data *BlockData     //区块数据 一个有序的交易列表。区块数据是在排序服务创建区块时被写入
 Metadata *BlockMetadata //元数据 区块被写入的时间等信息 与区块一起形成
}
type Metadata struct {
 Value= []byte //
 Signatures  []*MetadataSignature
}
type Header struct {
 ChannelHeader []byte
 SignatureHeader  []byte
}
type BlockHeader struct {
 Number uint64  //区块编号
 PreviousHash []byte  //前一个区块头的哈希
 DataHash []byte // 当前区块数据的哈希 不包括元数据
}
```

2.2 peer 节点

```
type Peer struct {
 ServerConfig              comm.ServerConfig
 CredentialSupport         *comm.CredentialSupport
 StoreProvider             transientstore.StoreProvider
 GossipService             *gossipservice.GossipService
 LedgerMgr                 *ledgermgmt.LedgerMgr
 OrdererEndpointOverrides map[string]*orderers.Endpoint
 CryptoProvider            bccsp.BCCSP
 validationWorkersSemaphore semaphore.Semaphore
 server              *comm.GRPCServer
 pluginMapper        plugin.Mapper
 channelInitializer func(cid string)
```

```
    mutex     sync.RWMutex
    channels map[string]*Channel
    configCallbacks []channelconfig.BundleActor
  }
```

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）

```
  type Endorser struct {
    ChannelFetcher         ChannelFetcher
    LocalMSP               msp.IdentityDeserializer
    PrivateDataDistributor PrivateDataDistributor
    Support                Support
    PvtRWSetAssembler      PvtRWSetAssembler
    Metrics                *Metrics
  }
```

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

```
  func (e *Endorser) callChaincode(txParams *ccprovider.TransactionParams, input *pb.ChaincodeInput, chaincodeName string) (*pb.Response, *pb.ChaincodeEvent, error) {
```

```
    }(time.Now())
  func (e *Endorser) simulateProposal(txParams *ccprovider.Transaction
Params, chaincodeName string, chaincodeInput *pb.ChaincodeInput) (*pb.R
esponse, []byte, *pb.ChaincodeEvent, *pb.ChaincodeInterest, error) {

  }
  func (e *Endorser) preProcess(up *UnpackedProposal, channel *Channel)
 error {
    err := up.Validate(channel.IdentityDeserializer)
  }
  func (e *Endorser) ProcessProposal(ctx context.Context, signedProp *
pb.SignedProposal) (*pb.ProposalResponse, error) {
  }
```

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

```
  type SignedProposal struct {
   // The bytes of Proposal
   ProposalBytes []byte
   Signature  []byte
  }
```

ProposalResponse

```
  type ProposalResponse struct {
   Version int32
   Timestamp *timestamp.Timestamp
   Response *Response  Payload []byte
   Endorsement *Endorsement
   Interest *ChaincodeInterest

  }
```

ProposalResponsePayload

```
  type ProposalResponsePayload struct {
   ProposalHash []byte
```

```
    Extension []byte
}
```

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。

```
type Consenter interface {
    Order(env *cb.Envelope, configSeq uint64) error
    Configure(config *cb.Envelope, configSeq uint64) error
    WaitReady() error
}
```

```
func (bh *Handler) ProcessMessage(msg *cb.Envelope, addr string) (resp *ab.BroadcastResponse) {
    tracker := &MetricsTracker{
     ChannelID: "unknown",
     TxType:    "unknown",
     Metrics:   bh.Metrics,
    }
    logger.Debugf("[channel: %s] Broadcast has successfully enqueued message of type %s from %s", chdr.ChannelId, cb.HeaderType_name[chdr.Type], addr)
    return &ab.BroadcastResponse{Status: cb.Status_SUCCESS}
}
```

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。

```
func (s *server) Broadcast(srv ab.AtomicBroadcast_BroadcastServer) error {
    logger.Debugf("Starting new Broadcast handler")
    return s.bh.Handle(&broadcastMsgTracer{
     },
    })
}
```

```
7.    func (s *server) Deliver(srv ab.AtomicBroadcast_DeliverServer) er
   ror {
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, erro
   r) {
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.", chai
   n.ChannelID())
19.       return counts, nil
20.      }
21.      }
22.      select {
23.      case <-chain.errorChan:
24.      default:
25.      }    }
```

3. 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

```
type Transaction struct {
 Actions              []*TransactionAction
}
```

```
type TransactionAction struct {
 Header []byte
 Payload []byte
}
```

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：

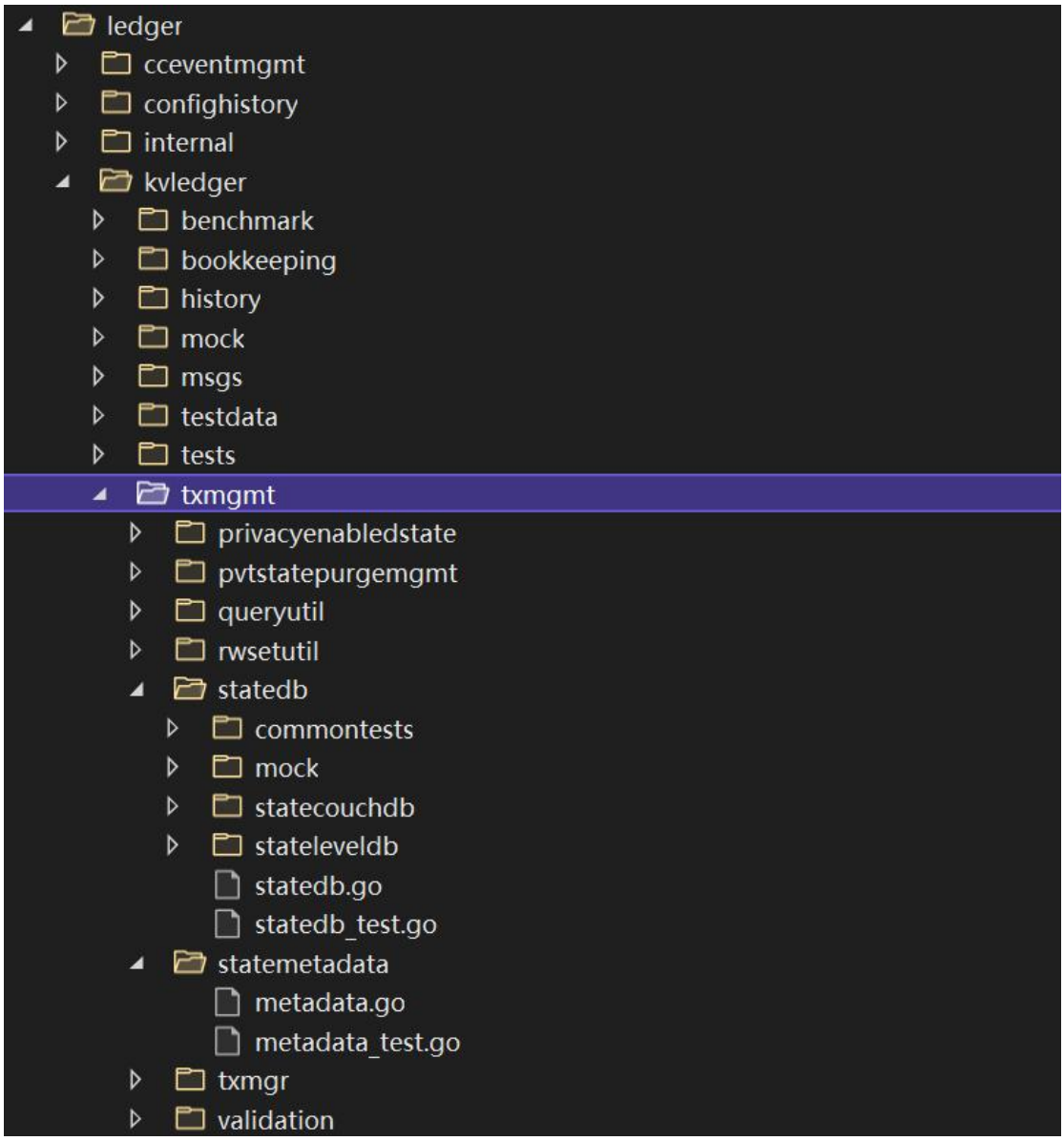
Payload:

```
type Payload struct {
 Header *Header
 Data  []byte
}
```

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

```
// SnapshotWriter generates two files, a data file and a metadata file.
The datafile contains a series of tuples <key, dbValue>
// and the metadata file contains a series of tuples
<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。