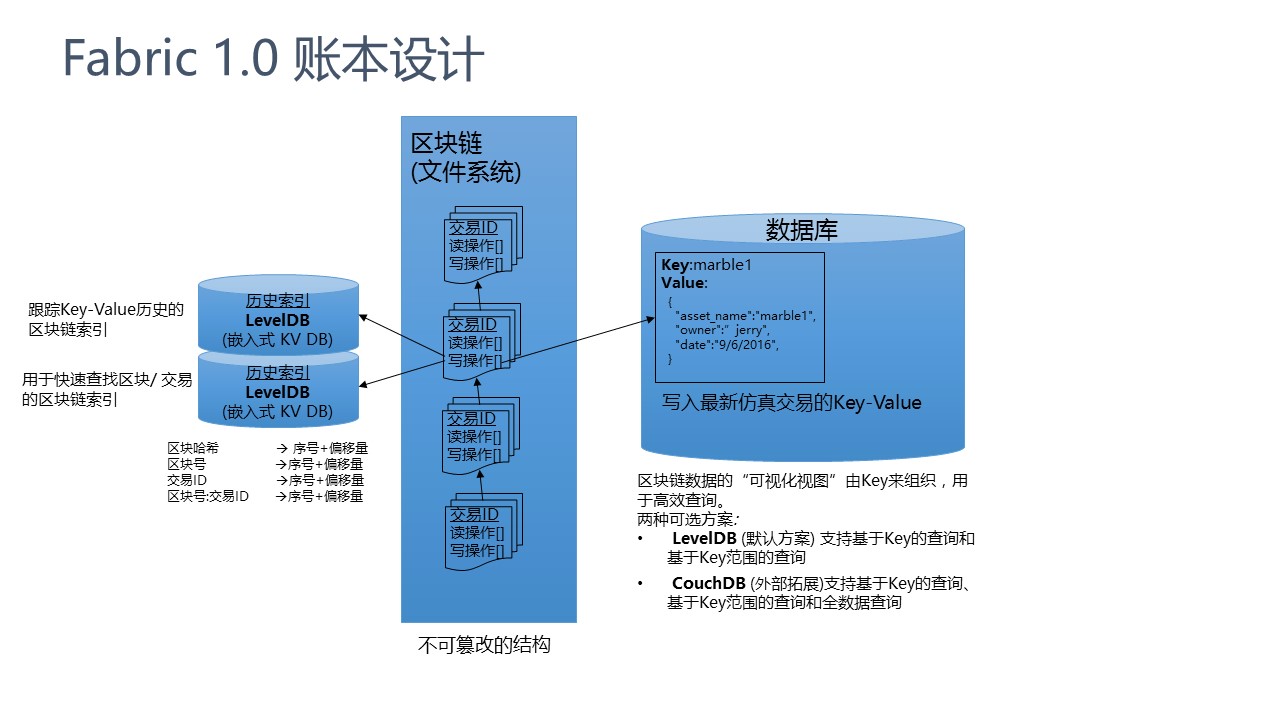
# Fabric 1.0账本结构分析

在fabric1.0的源码中，定义了三种不同类型的Ledger结构体，分别是：

* PeerLedger：Peer节点保存的本地账本记录，包括了区块链，全局状态（World State），历史索引（History Index）。账本中同时包含了合法交易与非法交易记录。
* ValidatedLedger:是由Peer节点保存的最终本地账本记录，在fabric 1.0版本中被实现。表示从PeerLedger中过滤出非法交易后的“最终账本记录”。
* OrdererLedger：Order节点保存的账本记录，结构体定义中只包含了区块链。账本中同时包含了合法交易与非法交易。

注：存储在Peer上的Ledger和存储在Orderer上的Ledger另有一个不同地方在于Peer上的Ledger在本地维护一个位掩码（比特位）用来区分有效交易和无效交易。

## Peer节点Ledger结构分析

图1 Peer节点账本结构的示意图

上图是Peer节点账本结构的示意图，其中：

1. 区块链结构不可篡改，并以文件的形式存储在本地，每一个区块代表多个交易记录，每个交易记录可以是合法的也可以是非法的，交易记录的合法性通过Ledger本地维护的掩码进行判别。
2. 历史索引使用leveldb存储，存储形式为key-value形式。保存了历史交易记录，便于用户进行检索
3. World state 默认使用leveldb存储，存储形式为key-value形式。保存当前时间用户的资产状况。

### Peer节点PeerLedger代码分析

#### Peer节点PeerLedger代码接口



图2 Peer节点PeerLedger代码示意图

上图是Peer节点PeerLedger代码的示意图，其中：

PeerLedger是账本的interface定义，代码位置在Fabric/Core/Ledger/Ledger\_interface.go。他为外界提供了一系列的接口定义，代码如下。

|  |
| --- |
| interface定义 |
| **type** PeerLedger **interface** {      commonLedger.Ledger      GetTransactionByID(txID string) (\*Peer.ProcessedTransaction, error)      GetBlockByHash(blockHash []byte) (\*common.Block, error)      GetBlockByTxID(txID string) (\*common.Block, error)      GetTxValidationCodeByTxID(txID string) (Peer.TxValidationCode, error)      NewTxSimulator() (TxSimulator, error)      NewQueryExecutor() (QueryExecutor, error)(HistoryQueryExecutor, error)   Prune(policy commonLedger.PrunePolicy) error  } |

|  |  |
| --- | --- |
| **接口定义** | **功能说明** |
| commonLedger.Ledger | 这个结构体是ValidatedLedger，PeerLedger OrdererLedger 共有的interface，其中定义了一系列对所有种类的Ledger都适用的方法，其实现代码在:  Fabric/Common/Ledger/Ledger\_interface.go。 |
|  |  |
| 下面对commonLedger.Ledger结构体提供的方法进行说明 |  |
| GetBlockchainInfo() (\*common.BlockchainInfo, error) | 返回的当前区块链的基本信息。 |
|  |  |
| GetBlockByNumber(blockNumber uint64) (\*common.Block, error) | 根据区块号（其实也就是区块所在的高度）返回指定区块 |
|  |  |
| GetBlocksIterator(startBlockNumber uint64) (ResultsIterator, error) | 根据区块号（其实也就是区块所在的高度）返回指定区块的迭代器（指针） |
|  |  |
| Close() | 关闭当前Ledger句柄 |
|  |  |
| Commit(block \*common.Block) | 将新的区块加入区块链中 |
|  |  |
| 下面开始是PeerLedger自身扩展的方法 |  |
| GetTransactionByID(txID tring) (\*Peer.ProcessedTransaction, error) | 输入交易ID，返回相应的交易结构体 |
|  |  |
| GetBlockByHash(blockHash []byte) (\*common.Block, error) | 输入区块的hash值，返回区块结构体 |
|  |  |
| GetBlockByTxID(txID string) (\*common.Block, error) | 输入交易ID，返回包含指定交易的区块结构体 |
|  |  |
| GetTxValidationCodeByTxID(txID string) (Peer.TxValidationCode, error) | 输入交易ID，返回当前交易的有效位编码值（比特位） |
|  |  |
| NewTxSimulator() (TxSimulator, error) | 返回交易模拟器，用于背书节点模拟执行chaincode |
|  |  |
| NewQueryExecutor() (QueryExecutor, error) | 返回针对world state查询的查询执行器，用于Peer节点对Ledger进行查询操作（在这里是对world state 进行查询） |
|  |  |
| NewHistoryQueryExecutor() (HistoryQueryExecutor, error) | 返回针对历史索引查询的查询执行器，用于Peer节点对Ledger进行查询操作 |
|  |  |
| Prune(policy commonLedger.PrunePolicy) error | 根据输入的策略，返回指定的区块，但是在1.0的代码中，这个函数还没有实现。 |
|  |  |
|  |  |

#### Peer节点PeerLedger方法实现

KvLedger是账本interface结构体PeerLedger的实现，代码位置在Fabric/Core/Ledger/kvLedger/kv\_Ledger.go。它的代码如下：

|  |
| --- |
| type kvLedger struct {      LedgerID string      blockStore blkstorage.BlockStore      txtmgmt txmgr.TxMgr      historyDB historydb.HistoryDB  } |

KvLedge主要由三个结构体组成，分别是：

* **LedgerID**：String 类型变量，标识了当前区块链的ID。
* **blockStore**：区块生成器，负责维护区块，有关区块的生成，检索，查询，本质上都是调用这个结构体实现的。
* **txtmgmt**：交易管理器，负责生成world state 查询器，交易模拟器，检验区块中的交易合法性，修改world state等。
* **historyDB**：历史索引管理器，提供链接history数据库的句柄，负责修改History 数据库，生成 历史索引查询器等。

下面对部分关键函数的实现代码进行分析，

|  |
| --- |
| func (l \*kvLedger) GetBlockByNumber(blockNumber uint64) (\*common.Block, error) |

代码如下：

|  |
| --- |
| func (l \*kvLedger) GetBlockByNumber(blockNumber uint64) (\*common.Block, error) {      return l.blockStore.RetrieveBlockByNumber(blockNumber)  } |

kvLedger会通过他的成员变量blockStore来实现功能，blockStore是blkstorage.BlockStore结构体，其定义的路径如下

路径:/fabric/common/Ledger/blkstorge/fsblkstorage/fs\_blockstore.go

相应的RetrieveBlockByNumber()方法代码如下：

|  |
| --- |
| func (store \*fsBlockStore) RetrieveBlockByNumber(blockNum uint64) (\*common.Block, error) {      return store.fileMgr.retrieveBlockByNumber(blockNum)  } |

fsBlockStore是blockStore的具体实现结构，他会调用他的成员变量fileMgr来实现函数功能，这个fileMgr是一个blockfileMgr结构体，其定义也在

/fabric/common/Ledger/blkstorge/fsblkstorage/blockstore\_mgr.go

retrieveBlockByNumber(blockNum)的实现代码如下：

|  |
| --- |
| func (mgr \*blockfileMgr) retrieveBlockByNumber(blockNum uint64) (\*common.Block, error) {      logger.Debugf("retrieveBlockByNumber() - blockNum = [%d]", blockNum)      // interpret math.MaxUint64 as a request for last block      if blockNum == math.MaxUint64 {          blockNum = mgr.getBlockchainInfo().Height - 1      }      loc, err := mgr.index.getBlockLocByBlockNum(blockNum)      if err != nil {          return nil, err      }      return mgr.fetchBlock(loc)  } |

如果当前输入的blockNumber是math.MaxUint64则返回最后一个block， mgr.index.getBlockLocByBlockNum(blockNum)返回blockNumber对应的block存储位置。返回值是一个fileLocPointer的文件指针。

mgr.fetchBlock(loc)根据存储位置返回block

### Peer节点ValidatedLedger代码分析

#### Peer节点ValidatedLedger代码接口



图3 Peer节点ValidatedLedger代码示意图

上图是Peer节点ValidatedLedger的示意图，它的实现比较简单，由一个commonLedger.Ledger组成

代码位置在Fabric/Core/Ledger/Ledger\_interface.go。他为外界提供了一系列的接口定义，代码如下。

|  |
| --- |
| interface定义 |
| type ValidatedLedger interface {      commonLedger.Ledger  } |

|  |  |
| --- | --- |
| 接口定义 | 功能描述 |
| commonLedger.Ledger | 这个结构体是ValidatedLedger，PeerLedger OrdererLedger 共有的interface，其中定义了一系列对所有种类的Ledger都适用的方法，其实现代码在:  Fabric/Common/Ledger/Ledger\_interface.go。 |
|  |  |
| 下面对commonLedger.Ledger结构体提供的方法进行描述 |  |
| GetBlockchainInfo() (\*common.BlockchainInfo, error) | 返回的当前区块链的基本信息。 |
|  |  |
| GetBlockByNumber(blockNumber uint64) (\*common.Block, error) | 根据区块号（其实也就是区块所在的高度）返回指定区块 |
|  |  |
| GetBlocksIterator(startBlockNumber uint64) (ResultsIterator, error) | 根据区块号（其实也就是区块所在的高度）返回指定区块的迭代器（可以看做指针） |
|  |  |
| Close() | 关闭当前Ledger句柄 |
|  |  |
| Commit(block \*common.Block) | 将新的区块加入区块链中 |

### Peer节点ValidatedLedger接口实现

ValidatedLedger的接口实现与PeerLedger相同，唯一的区别在于ValidatedLedger中删除了不合法的交易信息。

### 与Peer节点Ledger相关的数据结构

## Order节点Ledger结构分析



上图是Order节点账本结构的示意图，其中：

1. 区块链结构不可篡改， Order节点的Ledger结构体有三种实现方式，分别是File类型，JSON类型和RAM类型。其中File类型的Ledger为IBM提供的稳定版本，其余两种存储方式的Ledger为测试版本。
2. File类型的Ledger：File类型的Ledger将block存储在文件系统中。 磁盘上的块位置可以在LevelDB数据库中建立索引。 这是OrderLedger的默认配置选项。
3. RAM类型的Ledger：RAM类型的Ledger将block存储在内存中，优点是查询速度快。
4. JSON类型的Ledger：File类型的Ledger将block存储在文件系统的JSON编码文件中，优点是部署简单

### Order节点Ledger代码分析

#### Order节点Ledger代码接口



上图是Order节点OrderLedger代码的示意图，其中：

OrderLedgee的interface定义是通过一系列的句柄实现的，代码位置在Fabric/Order/Ledger/Ledger.go。他为外界提供了一系列的接口定义，代码如下：

其中，由Ledger实现的方法主要是ReadWriter结构体提供的方法。但是这些方法在实际调用过程中，是通过ReadWriter调用的。

|  |
| --- |
| interface定义 |
| type Factory interface {    GetOrCreate(chainID string) (ReadWriter, error)  ChainIDs() []string  Close()  }  type Iterator interface {  Next() (\*cb.Block, cb.Status)  ReadyChan() <-chan struct{}  }  type Reader interface {  Iterator(startType \*ab.SeekPosition) (Iterator, uint64)  Height() uint64  }  type Writer interface {    Append(block \*cb.Block) error  }  type ReadWriter interface {  Reader  Writer  } |

|  |  |
| --- | --- |
| **接口定义** | **功能说明** |
| Append(block \*cb.Block) error | 将新的block写入账本 |
|  |  |
| Iterator(startType \*ab.SeekPosition) (Iterator, uint64) | 根据Peer节点发送的seekINFO信号返回相应的Ledger迭代器（指针） |
|  |  |
| Height() uint64 | 返回当前的区块链高度 |
|  |  |
|  |  |
| 以下是factory结构体的接口说明 |  |
| GetOrCreate(chainID string) (ReadWriter, error) | 根据输入的ReadWriter结构体所标识的区块链返回对应的区块链ID。如果对应的区块链不存在，则创建相应的区块链并返回ID。 |
|  |  |
| ChainIDs() []string | 返回当前 factory结构体所对应的区块链ID |
|  |  |
| Close() | 关闭factory句柄 |
|  |  |
|  |  |
| 以下是factory结构体的接口说明 |  |
| Next() (\*cb.Block, cb.Status) | 返回当前指针的下一个区块，并返回执行结果状态码（比如当前区块是最新的，则返回error） |
|  |  |
| ReadyChan() <-chan struct{} | 返回一个通信管道，具体作用暂不明。 |

#### Order节点Ledger接口实现

考虑到当前只有file类型的Ledger会在实际工作中使用，因此主要分析FileLedger的代码实现。

代码位置在Fabric/Order/Ledger/file/imp.go。它的代码如下：

|  |
| --- |
| type fileLedger struct {      blockStore blkstorage.BlockStore      signal chan struct{}  } |

FileLedger主要由两个结构体组成，分别是：

* **blockStore**：区块生成器，负责维护区块，有关区块的生成，检索，查询，本质上都是调用这个结构体实现的。
* **signal**：golang中的channel，可以把它看成一个管道，通过它并发核心单元就可以发送或者接收数据进行通讯

注：由于各个函数的实现代码文字量较大，故在此不赘述，本质上来说都是调用fileLedger的成员变量进行一系列处理。

下面对部分关键函数的实现代码进行分析，

|  |
| --- |
| func (fl \*fileLedger) Height() uint64 |

代码如下：

|  |
| --- |
| func (fl \*fileLedger) Height() uint64 {      info, err := fl.blockStore.GetBlockchainInfo()      if err != nil {          logger.Panic(err)      }      return info.Height  } |

fileLedger会通过他的成员变量blockStore来实现功能，blockStore是blkstorage.BlockStore结构体，其定义的路径如下

路径:/fabric/common/Ledger/blkstorge/fsblkstorage/fs\_blockstore.go

相应的GetBlockchainInfo()方法代码如下：

|  |
| --- |
| func (store \*fsBlockStore) GetBlockchainInfo() (\*common.BlockchainInfo, error) {      return store.fileMgr.getBlockchainInfo(), nil  } |

fsBlockStore是blockStore的具体实现结构，他会调用他的成员变量fileMgr来实现函数功能，这个fileMgr是一个blockfileMgr结构体，其定义也在

/fabric/common/Ledger/blkstorge/fsblkstorage/blockstore\_mgr.go

getBlockchainInfo()的实现代码如下：

|  |
| --- |
| func (mgr \*blockfileMgr) getBlockchainInfo() \*common.BlockchainInfo {      return mgr.bcInfo.Load().(\*common.BlockchainInfo)  } |

bcInfo是blockfileMgr的成员变量。最终目的是返回common.BlockchainInfo

## 与账本结构有关的几个结构体

在Ledger提供的方法中，有返回的一系列特定结构体，现逐条对各个结构体的定义进行描述。

|  |
| --- |
| 调用接口 |
| PeerLedger.GetTransactionByID(txID string) (\***peer.ProcessedTransaction**, error) |
|  |
| GetBlockByHash(blockHash []byte) (\***common.Block**, error) |
|  |
| GetTxValidationCodeByTxID(txID string) (**peer.TxValidationCode**, error) |
|  |
| NewTxSimulator() (**TxSimulator**, error) |
|  |
| NewQueryExecutor() (**QueryExecutor**, error) |
|  |
| NewHistoryQueryExecutor() (**HistoryQueryExecutor**, error) |
|  |
| GetBlockchainInfo() (**\*common.BlockchainInfo**, error) |
|  |
| GetBlocksIterator(startBlockNumber uint64) (**ResultsIterator**, error) |
|  |
| 以下是orderLedger提供的方法 |
| Iterator(startType \***ab.SeekPosition**) (**Iterator**, uint64) |
|  |
| Next() (\*cb.Block, **cb.Status**) |

从下文开始对上述标粗结构体分条描述

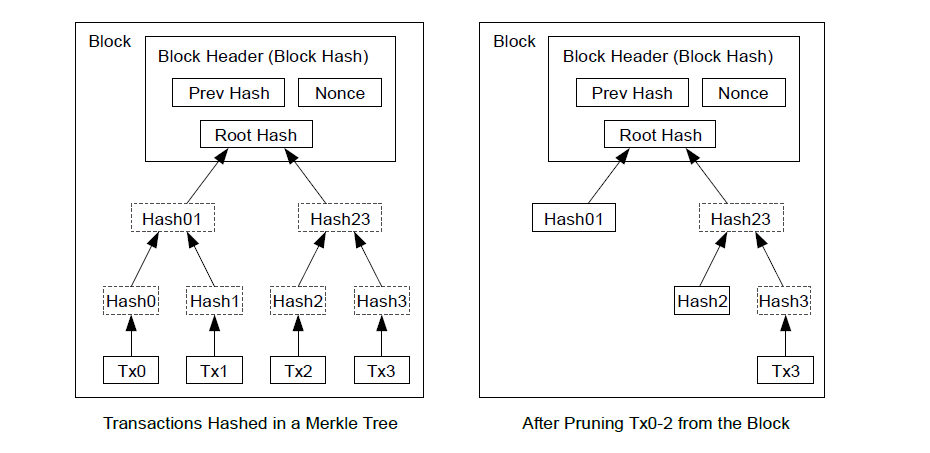
|  |  |
| --- | --- |
| 名称 | peer.ProcessedTransaction |
| 源码路径 | /fabric/protos/peer/transaction.go |
| 定义结构体 | message ProcessedTransaction {  common.Envelope transactionEnvelope = 1;  int32 validationCode = 2;  } |

common.Envelope是不同节点之间传输的的报文结构体。它的定义在：/fabric/protos/common/common.proto。Envelope=<signature,payload>，

### block结构体

|  |  |
| --- | --- |
| 名称 | Common. Block |
| 源码路径 | /fabric/protos/common/common.proto |
| 定义结构体 | message Block {  BlockHeader header = 1;  BlockData data = 2;  BlockMetadata metadata = 3;  } |

* Block有三个结构体组成，其中BlockHeader保存有当前Block的块标号、当前Block的Hash值和前一个block的Hash值。如下图



* BlockData保存有当前时间段内的所有交易记录。其中每一个元素都是一个Tx结构体。
* BlockMetadata 保存有与当前Block相关的信息，它是一个数组存储体，其中各索引下的数据意思如 BlockMetadataIndex所述。如下

|  |
| --- |
| enum BlockMetadataIndex {  SIGNATURES = 0;  LAST\_CONFIG = 1;  TRANSACTIONS\_FILTER = 2;  ORDERER = 3;  } |

即 BlockMetadata[0]为签名数据；BlockMetadata[1]为当前区块链中保存区块链配 置信息的Block编号；BlockMetadata[2]为当前区块链过滤非法交易的条件； BlockMetadata[3]为与共识相关的数据结构。