Peer 启动后会在后台执行 gossip 服务，包括若干 goroutine，实现位于 gossip/state/state.go#NewGossipStateProvider(chainID string, services \*ServicesMediator, ledger ledgerResources) GossipStateProvider 方法。

其中一个协程专门负责处理收到的区块信息。

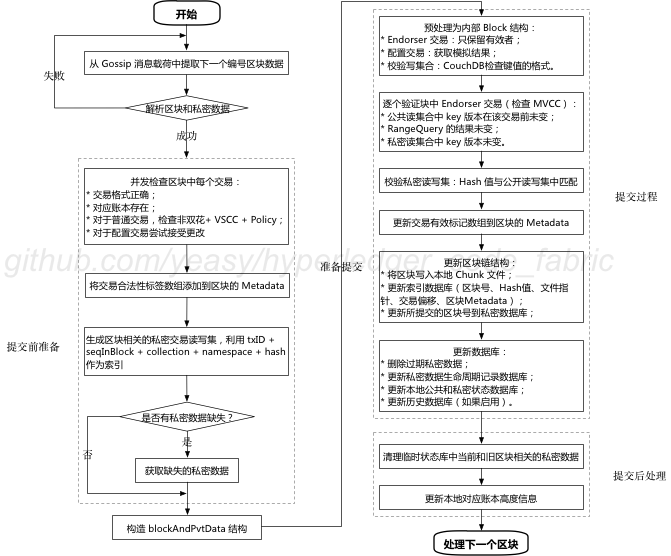
1. // Deliver in order messages into the incoming channel
2. go s.deliverPayloads()

deliverPayloads() 方法实现位于同一个文件的 GossipStateProviderImpl 结构下，其主要过程为循环从收到的 Gossip 消息载荷缓冲区按序拿到封装消息，解析后进行处理。核心代码逻辑如下：

1. // gossip/state/state.go#GossipStateProviderImpl.deliverPayloads()
2. for {
3. select {
4. case <-s.payloads.Ready(): // 等待消息
5. // 依次处理收到的消息
6. for payload := s.payloads.Pop(); payload != nil; payload = s.payloads.Pop() {
7. rawBlock := &common.Block{}
8. // 从载荷数据中尝试解析区块结构，失败则尝试下个消息
9. if err := pb.Unmarshal(payload.Data, rawBlock); err != nil {
10. logger.Errorf("Error getting block with seqNum = %d due to (%+v)...dropping block", payload.SeqNum, errors.WithStack(err))
11. continue
12. }
13. // 检查区块结构是否完整，失败则尝试下个消息
14. if rawBlock.Data == nil || rawBlock.Header == nil {
15. logger.Errorf("Block with claimed sequence %d has no header (%v) or data (%v)",
16. payload.SeqNum, rawBlock.Header, rawBlock.Data)
17. continue
18. }
19. // 从载荷中解析私密数据，失败则尝试下个消息
20. var p util.PvtDataCollections
21. if payload.PrivateData != nil {
22. err := p.Unmarshal(payload.PrivateData)
23. if err != nil {
24. logger.Errorf("Wasn't able to unmarshal private data for block seqNum = %d due to (%+v)...dropping block", payload.SeqNum, errors.WithStack(err))
25. continue
26. }
27. }
28. // 核心部分：提交区块到本地账本
29. if err := s.commitBlock(rawBlock, p); err != nil {
30. if executionErr, isExecutionErr := err.(\*vsccErrors.VSCCExecutionFailureError); isExecutionErr {
31. logger.Errorf("Failed executing VSCC due to %v. Aborting chain processing", executionErr)
32. return
33. }
34. logger.Panicf("Cannot commit block to the ledger due to %+v", errors.WithStack(err))
35. }
36. }
37. case <-s.stopCh: // 停止处理消息
38. s.stopCh <- struct{}{}
39. logger.Debug("State provider has been stopped, finishing to push new blocks.")
40. return
41. }
42. }

整体逻辑

s.commitBlock(rawBlock, p) 是对区块进行处理和提交的核心逻辑，主要包括提交前准备、提交过程和提交后处理三部分，如下图所示。

[](https://github.com/yeasy/hyperledger_code_fabric/blob/master/process/_images/peer_commit.png)

下面分别进行介绍三个阶段的实现过程。

提交前准备

主要完成对区块中交易格式的检查和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

对区块格式的检查主要在 core/committer/txvalidator/validator.go#TxValidator.Validate(block \*common.Block) error 方法中完成，包括检查交易格式、对应账本是否存在、是否双花、满足 VSCC 和 Policy 等。核心逻辑如下。

1. // core/committer/txvalidator/validator.go#TxValidator.Validate(block \*common.Block) error
2. // 并发验证交易有效性
3. go func() {
4. for tIdx, d := range block.Data.Data {
5. // ensure that we don't have too many concurrent validation workers
6. v.Support.Acquire(context.Background(), 1)
8. go func(index int, data []byte) {
9. defer v.Support.Release(1)
11. v.validateTx(&blockValidationRequest{
12. d: data,
13. block: block,
14. tIdx: index,
15. }, results)
16. }(tIdx, d)
17. }
18. }()

21. // 处理检查结果
22. for i := 0; i < len(block.Data.Data); i++ {
23. res := <-results
24. if res.err != nil {
25. if err == nil || res.tIdx < errPos {
26. err = res.err
27. errPos = res.tIdx
28. }
29. } else { // 设置有效标记，记录链码名称，更新链码信息
30. txsfltr.SetFlag(res.tIdx, res.validationCode)
31. if res.validationCode == peer.TxValidationCode\_VALID {
32. if res.txsChaincodeName != nil {
33. txsChaincodeNames[res.tIdx] = res.txsChaincodeName
34. }
35. if res.txsUpgradedChaincode != nil {
36. txsUpgradedChaincodes[res.tIdx] = res.txsUpgradedChaincode
37. }
38. txidArray[res.tIdx] = res.txid
39. }
40. }
41. }
42. // 标记双花交易
43. if v.Support.Capabilities().ForbidDuplicateTXIdInBlock() {
44. markTXIdDuplicates(txidArray, txsfltr)
45. }
46. // 防止多次升级操作
47. v.invalidTXsForUpgradeCC(txsChaincodeNames, txsUpgradedChaincodes, txsfltr)
48. // 确认所有交易都完成检查
49. err = v.allValidated(txsfltr, block)
50. if err != nil {
51. return err
52. }
53. // 更新交易有效标签到元数据
54. utils.InitBlockMetadata(block)
55. block.Metadata.Metadata[common.BlockMetadataIndex\_TRANSACTIONS\_FILTER] = txsflt

获取缺失的私密数据

首先根据已有的私密数据计算区块中交易关联的读写集信息。如果仍有缺失，则尝试从其它节点获取。

1. // gossip/privdata/coordinator.go#coordinator.StoreBlock(block \*common.Block, privateDataSets util.PvtDataCollections) error
2. // 利用已有的私密数据计算读写集
3. ownedRWsets, err := computeOwnedRWsets(block, privateDataSets)
4. privateInfo, err := c.listMissingPrivateData(block, ownedRWsets)
6. // 获取缺失私密数据
7. for len(privateInfo.missingKeys) > 0 && time.Now().Before(limit) {
8. c.fetchFromPeers(block.Header.Number, ownedRWsets, privateInfo)
9. if len(privateInfo.missingKeys) == 0 {
10. break
11. }
12. time.Sleep(pullRetrySleepInterval)
13. }

构建 blockAndPvtData 结构

blockAndPvtData 结构用于后续的提交工作，因此，需要包括相关的区块和私密数据。

主要实现逻辑如下：

1. // gossip/privdata/coordinator.go#coordinator.StoreBlock(block \*common.Block, privateDataSets util.PvtDataCollections) error
2. // 填充私密读写集信息
3. for seqInBlock, nsRWS := range ownedRWsets.bySeqsInBlock() {
4. rwsets := nsRWS.toRWSet()
5. blockAndPvtData.BlockPvtData[seqInBlock] = &ledger.TxPvtData{
6. SeqInBlock: seqInBlock,
7. WriteSet: rwsets,
8. }
9. }
11. // 填充缺失私密数据信息
12. for missingRWS := range privateInfo.missingKeys {
13. blockAndPvtData.Missing = append(blockAndPvtData.Missing, ledger.MissingPrivateData{
14. TxId: missingRWS.txID,
15. Namespace: missingRWS.namespace,
16. Collection: missingRWS.collection,
17. SeqInBlock: int(missingRWS.seqInBlock),
18. })
19. }

提交过程

提交过程是核心过程，主要包括预处理、验证交易、更新本地区块链结构、更新本地数据库结构四个步骤。

预处理

预处理阶段负责构造一个有效的内部区块结构。包括：

* 处理 Endorser 交易：只保留有效的 Endorser 交易；
* 处理配置交易：获取配置更新的模拟结果，放入读写集；
* 校验写集合：如果状态数据库采用 CouchDB，要按照 CouchDB 约束检查键值的格式：Key 必须为非下划线开头的 UTF-8 字符串，Value 必须为合法的字典结构，且不包括下划线开头的键名。

核心实现代码位于 core/ledger/kvledger/txmgmt/validator/valimpl/helper.go#preprocessProtoBlock(txmgr txmgr.TxMgr, validateKVFunc func(key string, value []byte) error, block \*common.Block, doMVCCValidation bool) (\*valinternal.Block, error)，如下所示：

1. // core/ledger/kvledger/txmgmt/validator/valimpl/helper.go#preprocessProtoBlock(txmgr txmgr.TxMgr, validateKVFunc func(key string, value []byte) error, block \*common.Block, doMVCCValidation bool) (\*valinternal.Block, error)
2. // 处理 endorser 交易
3. if txType == common.HeaderType\_ENDORSER\_TRANSACTION {
4. // extract actions from the envelope message
5. respPayload, err := utils.GetActionFromEnvelope(envBytes)
6. if err != nil {
7. txsFilter.SetFlag(txIndex, peer.TxValidationCode\_NIL\_TXACTION)
8. continue
9. }
10. txRWSet = &rwsetutil.TxRwSet{}
11. if err = txRWSet.FromProtoBytes(respPayload.Results); err != nil {
12. txsFilter.SetFlag(txIndex, peer.TxValidationCode\_INVALID\_OTHER\_REASON)
13. continue
14. }
15. } else { // 处理配置更新交易
16. rwsetProto, err := processNonEndorserTx(env, chdr.TxId, txType, txmgr, !doMVCCValidation)
17. if \_, ok := err.(\*customtx.InvalidTxError); ok {
18. txsFilter.SetFlag(txIndex, peer.TxValidationCode\_INVALID\_OTHER\_REASON)
19. continue
20. }
21. if err != nil {
22. return nil, err
23. }
24. if rwsetProto != nil {
25. if txRWSet, err = rwsetutil.TxRwSetFromProtoMsg(rwsetProto); err != nil {
26. return nil, err
27. }
28. }
29. }
30. // 检查读写集是否符合数据库要求格式
31. if txRWSet != nil {
32. if err := validateWriteset(txRWSet, validateKVFunc); err != nil {
33. txsFilter.SetFlag(txIndex, peer.TxValidationCode\_INVALID\_WRITESET)
34. continue
35. }
36. b.Txs = append(b.Txs, &valinternal.Transaction{IndexInBlock: txIndex, ID: chdr.TxId, RWSet: txRWSet})
37. }

验证交易

接下来，对区块中交易进行 MVCC 检查，并校验私密读写集，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，满足下列条件者才认为有效：

* 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。

实现在 core/ledger/kvledger/txmgmt/validator/statebasedval/state\_based\_validator.go#Validator.ValidateAndPrepareBatch(block \*valinternal.Block, doMVCCValidation bool) (\*valinternal.PubAndHashUpdates, error) 方法中，主要逻辑如下：

1. // core/ledger/kvledger/txmgmt/validator/statebasedval/state\_based\_validator.go#Validator.ValidateAndPrepareBatch(block \*valinternal.Block, doMVCCValidation bool) (\*valinternal.PubAndHashUpdates, error)
3. // 依次顺序检查每个交易
4. for \_, tx := range block.Txs {
5. var validationCode peer.TxValidationCode
6. var err error
7. // 检查 Endorser 交易
8. if validationCode, err = v.validateEndorserTX(tx.RWSet, doMVCCValidation, updates); err != nil {
9. return nil, err
10. }
12. tx.ValidationCode = validationCode
13. // 有效交易则将其读写集放到更新集合中
14. if validationCode == peer.TxValidationCode\_VALID {
15. committingTxHeight := version.NewHeight(block.Num, uint64(tx.IndexInBlock))
16. updates.ApplyWriteSet(tx.RWSet, committingTxHeight)
17. } else {
18. logger.Warningf("Block [%d] Transaction index [%d] TxId [%s] marked as invalid by state validator. Reason code [%s]",
19. block.Num, tx.IndexInBlock, tx.ID, validationCode.String())
20. }
21. }

对私密读写集的校验主要是再次检查 Hash 值是否匹配，实现在 core/ledger/kvledger/txmgmt/validator/valimpl/helper.go#validatePvtdata(tx \*valinternal.Transaction, pvtdata \*ledger.TxPvtData) error 方法中。

1. // core/ledger/kvledger/txmgmt/validator/valimpl/helper.go#validatePvtdata(tx \*valinternal.Transaction, pvtdata \*ledger.TxPvtData) error
2. for \_, nsPvtdata := range pvtdata.WriteSet.NsPvtRwset {
3. for \_, collPvtdata := range nsPvtdata.CollectionPvtRwset {
4. collPvtdataHash := util.ComputeHash(collPvtdata.Rwset)
5. hashInPubdata := tx.RetrieveHash(nsPvtdata.Namespace, collPvtdata.CollectionName)
6. // 重新计算私密数据 Hash 值，对比公共数据中的记录
7. if !bytes.Equal(collPvtdataHash, hashInPubdata) {
8. return &validator.ErrPvtdataHashMissmatch{
9. Msg: fmt.Sprintf(`Hash of pvt data for collection [%s:%s] does not match with the corresponding hash in the public data.
10. public hash = [%#v], pvt data hash = [%#v]`, nsPvtdata.Namespace, collPvtdata.CollectionName, hashInPubdata, collPvtdataHash),
11. }
12. }
13. }
14. }

最后，更新区块元数据中的交易有效标记列表，实现位于 core/ledger/kvledger/txmgmt/validator/valimpl/helper.go#postprocessProtoBlock(block \*common.Block, validatedBlock \*valinternal.Block) 方法，代码如下所示。

1. // core/ledger/kvledger/txmgmt/validator/valimpl/helper.go#postprocessProtoBlock(block \*common.Block, validatedBlock \*valinternal.Block)
2. func postprocessProtoBlock(block \*common.Block, validatedBlock \*valinternal.Block) {
3. txsFilter := util.TxValidationFlags(block.Metadata.Metadata[common.BlockMetadataIndex\_TRANSACTIONS\_FILTER])
4. for \_, tx := range validatedBlock.Txs {
5. txsFilter.SetFlag(tx.IndexInBlock, tx.ValidationCode)
6. }
7. block.Metadata.Metadata[common.BlockMetadataIndex\_TRANSACTIONS\_FILTER] = txsFilter
8. }

接下来，需要更新本地的账本结构，包括区块链结构和相关的本地数据库。

更新本地区块链结构

入口在 core/ledger/ledgerstorage/store.go#Store.CommitWithPvtData(blockAndPvtdata \*ledger.BlockAndPvtData) error 方法中，主要包括如下步骤：

* 将区块写入本地 Chunk 文件；
* 更新索引数据库（区块号、Hash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；

区块写入 Chunk 文件主要实现在 common/ledger/blkstorage/fsblkstorage/blockfile\_mgr.go#blockfileMgr.addBlock(block \*common.Block) error 方法中，主要逻辑如下所示：

1. // common/ledger/blkstorage/fsblkstorage/blockfile\_mgr.go#blockfileMgr.addBlock(block \*common.Block) error
2. // 计算长度信息
3. blockBytesLen := len(blockBytes)
4. blockBytesEncodedLen := proto.EncodeVarint(uint64(blockBytesLen))
5. totalBytesToAppend := blockBytesLen + len(blockBytesEncodedLen)
7. // 添加长度信息到 Chunk 文件
8. err = mgr.currentFileWriter.append(blockBytesEncodedLen, false)
10. // 更新 checkpoint 信息
11. newCPInfo := &checkpointInfo{
12. latestFileChunkSuffixNum: currentCPInfo.latestFileChunkSuffixNum,
13. latestFileChunksize: currentCPInfo.latestFileChunksize + totalBytesToAppend,
14. isChainEmpty: false,
15. lastBlockNumber: block.Header.Number}
16. mgr.saveCurrentInfo(newCPInfo, false);
18. // 更新区块在文件中索引位置和交易偏移量
19. blockFLP := &fileLocPointer{fileSuffixNum: newCPInfo.latestFileChunkSuffixNum}
20. blockFLP.offset = currentOffset
21. for \_, txOffset := range txOffsets {
22. txOffset.loc.offset += len(blockBytesEncodedLen)
23. }
25. // 更新索引数据库
26. mgr.index.indexBlock(&blockIdxInfo{
27. blockNum: block.Header.Number, blockHash: blockHash,
28. flp: blockFLP, txOffsets: txOffsets, metadata: block.Metadata})
30. // 更新 checkpoint 信息和区块链信息
31. mgr.updateCheckpoint(newCPInfo)
32. mgr.updateBlockchainInfo(blockHash, block)

更新本地数据库结构

更新数据库是提交交易的最后一步，主要包括如下步骤：

* 删除过期私密数据；
* 更新私密数据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；
* 如果启用了历史数据库，更新数据。

实现代码在 core/ledger/kvledger/txmgmt/txmgr/lockbasedtxmgr/lockbased\_txmgr.go#LockBasedTxMgr.Commit() error 方法中，主要逻辑如下。

1. // core/ledger/kvledger/txmgmt/txmgr/lockbasedtxmgr/lockbased\_txmgr.go#LockBasedTxMgr.Commit() error
3. // 准备过期的私密键值清理
4. txmgr.pvtdataPurgeMgr.PrepareForExpiringKeys(txmgr.current.blockNum())
6. // 更新私密数据生命周期记录数据库，这里记录了每个私密键值的存活期限
7. if err := txmgr.pvtdataPurgeMgr.DeleteExpiredAndUpdateBookkeeping(
8. txmgr.current.batch.PvtUpdates, txmgr.current.batch.HashUpdates); err != nil{
9. return err
10. }
12. // 更新本地公共状态数据库和私密状态数据库
13. if err := txmgr.db.ApplyPrivacyAwareUpdates(txmgr.current.batch, commitHeight); err != nil {
14. return err
15. }
17. // 如果启用了历史数据库，更新数据
18. if ledgerconfig.IsHistoryDBEnabled() {
19. if err := l.historyDB.Commit(block); err != nil {
20. panic(errors.WithMessage(err, "Error during commit to history db"))
21. }
22. }

提交后处理

提交后的处理比较简单，包括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。