内部资料

注意保密

Internal Document

Confidential

Fabric开发手册

**编 号： V0.8**

**版 本： 初稿**

**拟制人： 李涛**

**文件修订记录**

**版本修改记录** （A --- 增加 M --- 修改 D --- 删除）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **当前版本** | **日期** | **修改位置**  **（图、表、章节）** | **A/M/D** | **简单描述**  **（原因和修改简介）** | **修改人** | **评审人** |
| 初稿 | 2018-4-29 | ALL | A | 创建 | 李涛 |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

目录

[Fabric开发手册 1](#_Toc513455740)

[1. fabric源码解析1——线头 15](#_Toc513455741)

[Getting Started 15](#_Toc513455742)

[1.1. 找到线头 15](#_Toc513455743)

[1.2. 源码中的简拼 20](#_Toc513455744)

[1.3. 源码中的惯例 23](#_Toc513455745)

[1.4. 源码目录的基本结构 24](#_Toc513455746)

[2. fabric源码解析2——peer命令结构 25](#_Toc513455747)

[2.1. peer目录结构 25](#_Toc513455748)

[2.2. 第三方包 26](#_Toc513455749)

[2.3. peer命令结构解析 29](#_Toc513455750)

[2.4. 子命令结构解析 30](#_Toc513455751)

[2.5. peer命令结构 31](#_Toc513455752)

[3. fabric源码解析3——日志系统 32](#_Toc513455753)

[3.1. go-logging简介 32](#_Toc513455754)

[3.2. flogging 35](#_Toc513455755)

[3.2.1. init函数 35](#_Toc513455756)

[3.2.2. MustGetLogger函数 36](#_Toc513455757)

[3.2.3. 其他函数 36](#_Toc513455758)

[4. fabric源码解析4——配置系统 36](#_Toc513455759)

[4.1. viper简介 37](#_Toc513455760)

[4.2. viper搜索路径和文件 38](#_Toc513455761)

[4.3. InitViper 40](#_Toc513455762)

[4.4. 安全文件配置 42](#_Toc513455763)

[4.5. 命令选项配置 43](#_Toc513455764)

[4.6. 环境变量配置 43](#_Toc513455765)

[5. fabric源码分析5–kvledger的初始化 45](#_Toc513455766)

[5.1. 账本简单介绍 45](#_Toc513455767)

[5.1.1. 账本源码目录 45](#_Toc513455768)

[5.1.2. 账本数据库 46](#_Toc513455769)

[5.1.3. leveldb的基本操作 47](#_Toc513455770)

[5.2. 账本对象 47](#_Toc513455771)

[5.3. 块数据库存储服务对象 51](#_Toc513455772)

[5.3.1. 块索引配置indexConfig 54](#_Toc513455773)

[5.3.2. 块存储配置conf 54](#_Toc513455774)

[5.3.3. leveldb数据库存储服务对象blockStoreProvider 55](#_Toc513455775)

[5.4. 账本服务对象的目录结构 56](#_Toc513455776)

[6. fabric源码解析6–GRPC服务 58](#_Toc513455777)

[6.1. GRPC简介 58](#_Toc513455778)

[6.2. fabric中的grpc服务接口和实例 63](#_Toc513455779)

[6.2.1. TLS安全配置项 63](#_Toc513455780)

[6.2.2. GRPCServer接口 66](#_Toc513455781)

[6.2.3. GRPCServer实现实例 69](#_Toc513455782)

[6.3. peer node start启动的grpc服务 72](#_Toc513455783)

[6.4. peerServer 72](#_Toc513455784)

[6.4.1. 创建peerServer 72](#_Toc513455785)

[6.4.2. 注册服务 74](#_Toc513455786)

[6.5. globalEventsServer 76](#_Toc513455787)

[6.5.1. 创建globalEventsServer 78](#_Toc513455788)

[6.5.2. 注册事件服务 79](#_Toc513455789)

[6.5.3. 启动事件服务 82](#_Toc513455790)

[6.6. 事件实际处理者eventProcessor 82](#_Toc513455791)

[6.6.1. 事件类型 84](#_Toc513455792)

[6.6.2. start函数 84](#_Toc513455793)

[7. fabric源码解析7——peer的ChaincodeSupport服务 86](#_Toc513455794)

[7.1. 综述 86](#_Toc513455795)

[7.2. FSM 90](#_Toc513455796)

[7.3. 结构概览： 96](#_Toc513455797)

[7.4. Register 96](#_Toc513455798)

[7.4.1. Handler 97](#_Toc513455799)

[7.4.2. processStream 98](#_Toc513455800)

[7.4.3. HandleMessage 99](#_Toc513455801)

[7.4.4. serialSend或serialSendAsync 106](#_Toc513455802)

[7.5. 小结 106](#_Toc513455803)

[8. fabric源码解析8——peer的System Chaincode 106](#_Toc513455804)

[8.1. 综述 106](#_Toc513455805)

[8.2. 预定义和注册 108](#_Toc513455806)

[8.3. 释义 113](#_Toc513455807)

[9. fabric源码解析9——文档翻译之MSP 114](#_Toc513455808)

[9.1. 安全和会员服务 115](#_Toc513455809)

[9.2. Membership Service Providers (MSP) 115](#_Toc513455810)

[9.2.1. MSP配置 116](#_Toc513455811)

[9.2.2. 如何生成MSP证书和它们的签名匙 119](#_Toc513455812)

[9.2.3. MSP setup on the peer & orderer side 119](#_Toc513455813)

[9.2.4. Channel MSP setup 119](#_Toc513455814)

[9.2.5. Best Practices 120](#_Toc513455815)

[10. fabric源码解析10——文档翻译之Architecture 124](#_Toc513455816)

[10.1. 构架释义 124](#_Toc513455817)

[10.1.1. 系统构架 125](#_Toc513455818)

[10.1.2. 结点（Nodes） 129](#_Toc513455819)

[10.1.3. 交易背书的基础工作流 137](#_Toc513455820)

[10.1.4. 背书策略 141](#_Toc513455821)

[10.1.5. (v1后续版本). 验证过的账本和PeerLedger验证节点(正在修改) 145](#_Toc513455822)

[10.2. 交易流 147](#_Toc513455823)

[10.3. Hyperledger Fabric SDKs 153](#_Toc513455824)

[11. fabric源码解析11——peer的Admin和Endorser服务 154](#_Toc513455825)

[11.1. Admin 155](#_Toc513455826)

[11.2. Endorser 161](#_Toc513455827)

[11.2.1. 背书服务原型和实现 162](#_Toc513455828)

[11.3. 频道中的策略检查器 180](#_Toc513455829)

[12. fabric源码解析12——peer的MSP服务 189](#_Toc513455830)

[12.1. 背书检验 192](#_Toc513455831)

[12.1.1. Identity 198](#_Toc513455832)

[12.1.2. SigningIdentity 203](#_Toc513455833)

[12.2. MSP和MSPManager接口 206](#_Toc513455834)

[12.2.1. MSP 206](#_Toc513455835)

[12.2.2. MSPManager 211](#_Toc513455836)

[12.3. mgmt 213](#_Toc513455837)

[13. fabric源码解析13——peer的BCCSP服务 217](#_Toc513455838)

[13.1. 加密的一般话题 217](#_Toc513455839)

[13.2. BCCSP服务结构 218](#_Toc513455840)

[13.3. bccsp中的接口和选项 219](#_Toc513455841)

[13.3.1. 接口 219](#_Toc513455842)

[13.3.2. 选项 223](#_Toc513455843)

[13.4. SW实现方式 229](#_Toc513455844)

[13.5. pkcs11实现方式 236](#_Toc513455845)

[13.6. BCCSP工厂 241](#_Toc513455846)

[14. fabric源码解析14——peer的gossip服务之初始化 245](#_Toc513455847)

[14.1. gossip数据传播协议 246](#_Toc513455848)

[14.1.1. gossip协议 247](#_Toc513455849)

[14.1.2. gossip消息传送 248](#_Toc513455850)

[14.2. gossip源码结构 249](#_Toc513455851)

[14.3. gossip服务的初始化 253](#_Toc513455852)

[14.4. gossip服务初始化的前提元素 259](#_Toc513455853)

[14.4.1. 签名者serializedIdentity 259](#_Toc513455854)

[14.4.2. 消息加密者服务messageCryptoService 260](#_Toc513455855)

[14.4.3. 安全顾问secAdv 260](#_Toc513455856)

[14.4.4. 其他元素 261](#_Toc513455857)

[14.5. 消息类型分类 261](#_Toc513455858)

[14.5.1. 辅助类消息 261](#_Toc513455859)

[14.5.2. pull机制消息 262](#_Toc513455860)

[14.5.3. state消息 262](#_Toc513455861)

[14.5.4. 关系消息 263](#_Toc513455862)

[14.6. gossip服务中主要模块的初始化 265](#_Toc513455863)

[14.6.1. discovery模块 266](#_Toc513455864)

[14.6.2. emitter模块 267](#_Toc513455865)

[14.6.3. comm模块 268](#_Toc513455866)

[14.6.4. idMapper模块 271](#_Toc513455867)

[14.6.5. Mediator模块 272](#_Toc513455868)

[14.6.6. certStore模块 275](#_Toc513455869)

[14.6.7. chanState模块 275](#_Toc513455870)

[14.6.8. filter模块 280](#_Toc513455871)

[14.7. gossip服务初始化中所起的goroutine 281](#_Toc513455872)

[14.7.1. discovery模块goroutine 282](#_Toc513455873)

[14.7.2. start函数goroutine 284](#_Toc513455874)

[14.7.3. 身份验证终止goroutine 285](#_Toc513455875)

[14.8. gossip服务器的初始化 287](#_Toc513455876)

[14.8.1. election模块 289](#_Toc513455877)

[14.8.2. state模块 294](#_Toc513455878)

[14.9. gossip服务的停止 300](#_Toc513455879)

[15. fabric源码解析15——peer的gossip服务之一条消息的散播旅行 300](#_Toc513455880)

[15.1. 旅行的前提 300](#_Toc513455881)

[15.2. 消息从何而来 302](#_Toc513455882)

[15.2.1. deliverservice模块的初始化 302](#_Toc513455883)

[15.2.2. deliverservice模块的启动 303](#_Toc513455884)

[15.3. 消息如何散播 304](#_Toc513455885)

[15.3.1. 散播过程中如何选择散播结点 310](#_Toc513455886)

[15.4. 消息去往何方 314](#_Toc513455887)

[15.5. 旅行足迹图 314](#_Toc513455888)

[16. fabric源码解析16——peer的gossip服务之测试 317](#_Toc513455889)

[16.1. 彻底一点儿 317](#_Toc513455890)

[16.2. fabric test 318](#_Toc513455891)

[16.3. gossip test 319](#_Toc513455892)

[16.3.1. 测试函数TestDissemination 320](#_Toc513455893)

[16.3.2. 设置打印点 323](#_Toc513455894)

[16.3.3. 测试并分析打印信息 330](#_Toc513455895)

[17. fabric源码解析17——peer的chaincode之元数据 333](#_Toc513455896)

[17.1. chaincode涉及的主要目录 334](#_Toc513455897)

[17.2. chaincode的元数据 334](#_Toc513455898)

[17.3. chaincode的元工具 340](#_Toc513455899)

[17.4. 前奏 343](#_Toc513455900)

[18. fabric源码解析18——SCC的安装和部署 343](#_Toc513455901)

[18.1. 概述 344](#_Toc513455902)

[18.2. 安装 346](#_Toc513455903)

[18.3. 部署 347](#_Toc513455904)

[18.3.1. Launch 349](#_Toc513455905)

[18.3.2. Execute 359](#_Toc513455906)

[18.4. 部署后的状态 362](#_Toc513455907)

[18.5. 遗留问题 363](#_Toc513455908)

[19. fabric源码解析19——ACC的安装 363](#_Toc513455909)

[19.1. 概述 363](#_Toc513455910)

[19.2. 生成签名申请包 365](#_Toc513455911)

[19.3. 处理安装申请 366](#_Toc513455912)

[19.3.1. 执行申请 367](#_Toc513455913)

[19.4. 安装后的状态 374](#_Toc513455914)

[20. fabric源码解析20——ACC的部署 374](#_Toc513455915)

[20.1. 概述 374](#_Toc513455916)

[20.2. 起点 376](#_Toc513455917)

[20.3. 部署 376](#_Toc513455918)

[20.4. 广播 396](#_Toc513455919)

[20.5. 部署后的状态 397](#_Toc513455920)

[20.6. 后记闲言 398](#_Toc513455921)

[21. fabric源码解析21——撇开的一笔 398](#_Toc513455922)

[21.1. 概述 398](#_Toc513455923)

[22. fabric源码解析22——Orderer服务的初始化 399](#_Toc513455924)

[22.1. 概述 399](#_Toc513455925)

[22.2. 目录和图片 400](#_Toc513455926)

[22.3. kingpin 402](#_Toc513455927)

[22.4. 模块 402](#_Toc513455928)

[22.5. 配置 403](#_Toc513455929)

[22.6. 模块的初始化 405](#_Toc513455930)

[22.6.1. Server 405](#_Toc513455931)

[22.6.2. ledger 412](#_Toc513455932)

[22.6.3. kafka 414](#_Toc513455933)

[23. fabric源码解析23——Orderer服务 422](#_Toc513455934)

[23.1. 建立连接 422](#_Toc513455935)

[23.2. Broadcast 425](#_Toc513455936)

[23.3. Orderer 426](#_Toc513455937)

[23.4. Deliver 432](#_Toc513455938)

[23.5. 拓展：orderer实现的共识机制 437](#_Toc513455939)

[23.5.1. 问题A 440](#_Toc513455940)

[23.5.2. 问题B 441](#_Toc513455941)

[23.5.3. 思考 442](#_Toc513455942)

[24. fabric源码解析24——ledger之idStore和BlockStore 443](#_Toc513455943)

[24.1. 概述 443](#_Toc513455944)

[24.2. 涉及的目录 444](#_Toc513455945)

[24.3. peer结点中的leveldb 445](#_Toc513455946)

[24.4. peer结点中的账本 445](#_Toc513455947)

[24.4.1. 创建 446](#_Toc513455948)

[24.4.2. 使用 447](#_Toc513455949)

[24.5. idStore 448](#_Toc513455950)

[24.5.1. 存储账本ID 449](#_Toc513455951)

[24.5.2. ConstructionFlag 449](#_Toc513455952)

[24.6. BlockStore 452](#_Toc513455953)

[24.6.1. blockfile 453](#_Toc513455954)

[24.6.2. block信息数据库 454](#_Toc513455955)

[24.6.3. block迭代器 458](#_Toc513455956)

[24.6.4. 创建-增-查 459](#_Toc513455957)

[25. fabric源码解析25——ledger之VersionedDB和HistoryDB 465](#_Toc513455958)

[25.1. 续接前文 465](#_Toc513455959)

[25.2. VersionedDB 465](#_Toc513455960)

[25.2.1. 交易模拟器/交易查询器 469](#_Toc513455961)

[25.2.2. 验证器 477](#_Toc513455962)

[25.2.3. state数据库 489](#_Toc513455963)

[25.2.4. 重启恢复 490](#_Toc513455964)

[25.3. HistoryDB 493](#_Toc513455965)

[26. fabric源码解析26——验证体系 498](#_Toc513455966)

[26.1. 概述 498](#_Toc513455967)

[27. fabric源码解析27——Channel 499](#_Toc513455968)

[27.1. 概述 499](#_Toc513455969)

[27.2. 目录 500](#_Toc513455970)

[27.3. 配置文件 501](#_Toc513455971)

[27.4. 命令 502](#_Toc513455972)

[27.5. 结语 513](#_Toc513455973)

[28. fabric源码解析28——官网案例部署 514](#_Toc513455974)

[28.1. 概述 514](#_Toc513455975)

[28.2. 下载部署环境 514](#_Toc513455976)

[28.3. 编译peer、orderer、configtxgen等程序 517](#_Toc513455977)

[28.4. 部署 521](#_Toc513455978)

[28.4.1. Crypto Generator 522](#_Toc513455979)

[28.4.2. Configuration Transaction Generator 526](#_Toc513455980)

[28.4.3. networkUp 528](#_Toc513455981)

[28.5. 运行容器+区域链操作 529](#_Toc513455982)

# fabric源码解析1——线头

## Getting Started

简单提一下Fabric说明文档中的Getting Started部分。说明文档下载地址在 [http://hyperledger-fabric.readthedocs.io/en/latest/](http://hyperledger-fabric.readthedocs.io/en/latest/" \t "_blank) 。其中网址中有en的，就说明各种语言版本的说明文档可能在开发中，希望能尽快出中文版的，毕竟在有些概念上，个人在英语上的理解还是有一点困难。Getting Started主要引导你做的以下事情：

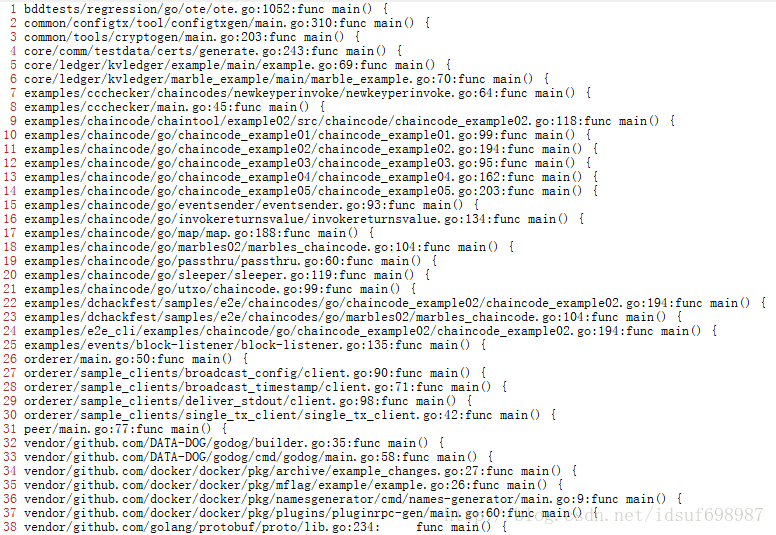
* 根据Prerequisites的要求部署环境，安装docker，go等。
* 下载要操作体验的单独的一个小例子及所需的镜像，解压后是一个release目录，以下的步骤都是操作这个小例子中的文件，脚本或程序。
* 用configtxgen和cryptogen生成用于区域链的配置文件，gensis块文件，交易配置文件。这两个工具也可手工生成。
* 用shell脚本运行docker镜像。
* 执行peer命令，部署区域链，执行一个实际交易，把a的10块钱转给b，然后查询一下。

## 找到线头

入手一个陌生的项目，我这水平的只能从main函数开始了。关于这点也想了很久，也咨询过一些人，从main函数入手算是最快也最容易的方法了。幸而go语言传承了C的不少特点，而我们还有grep这样搜索的利器。在Fabric根目录执行如下命令——递归搜索所有go文件中的main函数，并将结果导入func\_main.txt文件：

grep "func main" \* -r -n --include=\*.go > func\_main.txt

打开func\_main.txt文件，会发现大约七八十条记录，但是很有规律，如下：



可以看到，结果只集中在bddtests,common,core,examples,orderer,peer,vendor这几个目录。

秉承学习的姿态，可以将结果分为三类：可以直接排除掉的结果，可辅助研究源码的结果，源码本身的结果。

首先可以直接排除掉vendor中的所有结果，因为vendor目录下都是go语言使用的第三方库的代码。其次目录中有test，example，sample字眼的，都属于测试和示例范畴的代码，可以参考辅助我们研究源码，因为有时候看不懂源码时，看看例子是怎么用的，可能就知道这个函数是要干什么或者要怎么使用了。最后是源码本身，也是我们要研究的线头，经过以上两步的排除，剩下的，也就是源码：

* 2 common/configtx/tool/configtxgen/main.go:310:func mainc(){
* 3 common/tools/cryptogen/main.go:203:func main() {
* 26 orderer/main.go:50:func main() {
* 31 peer/main.go:77:func main() {

操作过Fabric文档中Getting Started的话，很容易就能推测出2，3行这两个main是用于生成configtxgen和cryptogen这两个工具。31行的main函数是用于生成peer程序，因为操作交易时在命令行使用到了peer …。Getting Started中虽然没有明显使用到关于orderer的操作，但既然被单独赋予了main函数，就说明其是自成一体，自成一个服务，最终生成会一个orderer程序。

这样看来，研究整个项目源码所要扯出的线头是不是一目了然了？

验证线头

从 **Getting Started** 入手，因为能作为一个项目的Getting Started，肯定是属于麻雀虽小五脏俱全的。

1. 我们知道，Getting Started用脚本启动了一些容器，并将这些容器作为分布的客户端进行区域链的部署，传递消息和交易等操作。启动容器操作的是 **./network\_setup.sh up** 命令，也就是使用的network\_setup.sh脚本，查看该文件，可以知道该命令最终执行的是该脚本中的networkUp函数。理解这一步，需要你知道基本的脚本知识。
2. networkUp函数主要运行了generateCfgTrx.sh脚本（进行了两个工具的生成进而使用两个工具生成区域链的必要的配置文件）和 **docker-compose -f $COMPOSE\_FILE up -d**命令，其中docker-compose就是根据指定yaml配置文件启动相应容器的命令。该配置文件为docker-compose-no-tls.yaml（$COMPOSE\_FILE的值），这又将我们的目光转移到了docker-compose-no-tls.yaml配置文件。理解这一步，需要你知道基本的docker容器操作和docker-compose命令。
3. 从yaml配置文件中我们可以看出，docker-compose命令启动了**orderer0,peer0,peer1,peer2,peer3,cli**六个容器服务。也就是说，这六个容器中所运行的服务覆盖了我们所要研究的源码中所会生成的主要服务，也都是最重要的核心服务。理解这一步，以及下面的每一步，需要你理解yaml配置文件的基本配置项。
4. orderer0容器的**command**选项为**orderer**，即启动orderer容器之时默认就启动了**orderer程序**，与我们牵扯出的orderer目录下的main函数所生成的**orderer程序**就对应上了。这也说明，启动容器的时候，orderer程序已经存在与该容器之中。
5. 每个peer容器没有command选项，但有**extends**选项，其加载的是peer-base目录下的**peer-base-no-tls.yaml** 配置文件。打开该文件，我们看到其加载的peer-base服务时所默认执行的**command**为 **peer node start –peer-defaultchain=false** ，即执行的是**peer命令**，这就和我们牵扯出的peer的main函数所生成的**peer程序**对应上了。这也说明，启动容器的时候，peer程序已经存在与该容器之中。
6. cli容器服务的**command**选项是 **/bin/bash -c ‘./scripts/script.sh ${CHANNEL\_NAME}; ‘** ，即启动容器时默认执行scripts目录下的**script.sh**脚本。打开script.sh脚本，我们发现最终所执行的命令都是**peer channel…**与**peer chaincode…**， 即执行的是**peer命令**，这也和我们牵扯出的peer的main函数所生成的peer程序对应上了。
7. 根据每个容器的**depends\_on**选项（依赖关系），我们甚至可以推测出在区域链中，哪些命令和服务是必须先于一些命令服务运行起来的。orderer容器先起，各个peer之间其实可以没有先后顺序但是都必须后与orderer容器，最后是cli容器。这说明，orderer服务必须先于peer服务，peer node start命令必须先于peer channel或peer chaincode命令。

>   
最终，以上的这些指向将我们引向了 **orderer和peer的main函数** ，也就是研究整个项目源码的线头。接下来的源码分析也是先将从peer的main函数这个线头开始。

## 源码中的简拼

能被Linux Foundation支持的项目，自然差不了。代码瞄上几眼就知道是很优秀的代码。代码中很多地方都是英语的全拼，有时候不需要看ReadMe.md或注释都能知道这个目录或函数是干嘛用的。但有些目录，文件名也有用首字母的，知道这些简拼对于顺利阅读代码也是很有帮助的。以下是我在阅读代码中收集的简拼，各位可以看一下，翻译不到之处还请指正。

MSP：Membership service provider 会员服务提供者   
BCCSP：blockchain（前两个字母BC） cryptographic service provider 区域链加密服务提供者   
ab：atomic broadcast原子（操作）广播   
lscc：lifecycle(L) system(S) chaincode（CC）生命周期系统链码   
Spec：Specification，规格标准，详细说明   
KV：key-value 键-值   
CDS：ChaincodeDeploymentSpec   
CIS：ChaincodeInvocationSpec   
mgmt：management   
SW：software-based   
AB：AtomicBroadcast   
GB：genesis block，创世纪的block，也就是区域链中的第一个块   
CC或cc：chaincode   
SCC或scc：system chaincode   
cscc：configer system chaincode   
lscc：lifecycle system chaincode   
escc：endorser system chaincode   
vscc：validator system chaincode   
qscc：querier system chaincode   
alg：algorithm 算法   
mcs：mspMessageCryptoService   
mock：假装，学样子，模仿的意思，基本上是服务于xxx\_test.go的，即用于测试的   
Gossip：一种使分布结点达到状态最终一致的算法   
attr：attribute   
FsBlockStore：file system block store   
vdb：versioned database 也就是状态数据库   
RTEnv：runtime environment运行环境   
pkcs11：pcks#11，一种公匙加密标准，有一套叫做Cryptoki的接口，是一组平台设备无关的API   
MCS：mspMessageCryptoService，消息加密服务   
sa：SecurityAdvisor   
impl：implement，好多处XXX.go和XXXimpl.go是对应的，前者是用于接口或者定义的，后者是实现该接口或定义的   
FSM：finite state machine 有限状态机   
FS：filesystem 文件系统   
blk：block   
cli：command line interface 命令行界面   
CFG：FABRIC\_CFG\_PATH中的，应该是config的意思   
mgr：manager   
cpinfo：checkpoint information，检查点信息   
DevMode：development mode，开发模式   
Reg：register，注册，登记   
hdr：header   
impl：implement   
oid：ObjectIdentifier，对象标识符   
ou或OU：organizational unit   
CRL：certificate revocation list，废除证书列表   
prop：proposal，申请，交易所发送的申请   
ACL：Access Control List，访问控制列表   
rwset：read/write set，读写集   
tx，Tx：transaction，交易   
CSP：cryptographic service provider，BCCSP的后三个字母，加密服务提供者   
opt：option，选项   
opts：options，多个选项   
SKI：当前证书标识，所谓标识，一般是对公匙进行一下hash   
AKI：签署方的SKI，也就是签署方的公匙标识   
HSM：Hardware Security Modules   
ks：KeyStore，Key存储，这个key指的是用于签名的公匙私匙   
oid：OBJECT IDENTIFIER，对象身份标识

## 源码中的惯例

源码中有很多惯例，也是大型项目（这就跟语言无关了）中需要有的很优秀的习惯。这一点也是值得学习的。目前个人发现的惯例有以下几点，对于阅读源码也是很有帮助的：

* common目录是其所在的层级中的公用的代码。A/common，则说明该common中的代码在A范围中公用，A/B/C/common，则说明该common中的代码在C目录中公用。
* mock目录是用于方便go测试文件（即众多的XXX\_test.go）中进行测试所需要的模拟数据/环境等。研究源码的初始阶段可忽略该类目录。
* XXX.go与XXXimpl.go是定义与实现的配套代码。
* 同一事务分别存在与不同主题下。如protos目录下的peer与core目录下的peer都是peer相关的代码，但是相关主题的代码却分开放置。
* no-tls标有no-tls的与说明相关代码未使用安全传输协议（TLS）。
* util文件夹，一般都是该层级或该主题源码中具有辅助性，工具性的代码。

## 源码目录的基本结构

其实关于源码目录的基本结构应该是自己在探索源码的过程中自己自然而然的明白的。在此只做记录使用，不建议各位看这部分，而是建议各位在自己阅读源码过程中自己不断的翻目录，自己慢慢弄懂。最开始研究源码，需要关注的也就是如下这些目录了。

* bcssp 加密服务代码目录
* common 全局公用代码目录
* core 核心功能代码目录
* docs 以.rst文件为核心，可编译生成文档。说明文档的目录
* events 事件代码目录，用于生产和消费信息
* examples 示例目录
* gossip 本意是绯闻的意思，是一种可达到去中心化，有一定容错能力且可达到最终一致的传播算法
* msp 会员服务代码目录
* orderer 就理解成orderer目录就好，orderer也算是区域链中的专用名词，用于消息的订阅与分发处理
* protos 原型目录，定义个各种原型和生成的对应的XXX.pb.go源码
* vendor 原意是商贩，在此就是存放go中使用的全部的各种第三方包

# fabric源码解析2——peer命令结构

## peer目录结构

peer目录结构自身十分清晰，一个main.go文件，其余文件夹除common,gossip外均为子命令集合，有chaincode，channel，clilogging，node，version五个，各司其职，供main.go整合使用。子命令文件夹中，与文件夹名称相同的.go文件为主要源码文件，其余的均为按功能划分的动作命令源码。以node目录为例，node自身作为根命令的一个子命令，在node.go中实现，而node这个命令自身又有start，status，stop这三个动作去执行不同的任务，分别在对应start.go，status.go，stop.go中实现。注意，start，status，stop其实也是子命令，是node这个子命令的子命令，因为他们是在命令层级中最终去干活的底层的人，我觉得用动作去形容他们更贴切一些。

* chaincode
* channel
* clilogging
* common
* gossip
* node   
  + node.go
  + start.go
  + status.go
  + stop.go
* version
* main.go

## 第三方包

在Getting Started中，无论是在启动peer容器时默认执行的命令，还是手工执行交易时在终端窗口所输入的命令，都类有类似的格式，如peer channel…，peer node…，peer chaincode…，这种 **命令+子命令+选项** 的风格，让人在感觉上毫无违和感。peer命令主要依赖第三方包github.com/spf13/cobra，由其组成基本的peer命令架构。所以在此简单介绍一下cobra。

cobra既是一个用于生成命令行程序的库，也是用来生成程序和命令文件的程序（即在命令行用cobra命令进行一系列操作，格式化生成一些使用cobra框架的源代码文件，用户可在此基础上进行编程）。目前，peer源码只将cobra当作一个库进行使用。cobra基本用法如下：

* 创建一个（根）命令对象，其原型为Command，每个命令都是一个Command对象实例。创建命令对象其实就是填充Command中的成员的过程罢了。需要注意的是，Command中的成员还有很多，其中有一批字段名为\*Run，\*RunE形式的成员，其作用与Run类似，区别在于运行的时间有先有后，是否被子命令继承，是否返回错误。

type Command struct {

Use string //命令名称字段，如你在命令行敲的是peer ...，则该字段值就是"peer"

Short string //短说明字段

Long string //详细说明字段

Run func(cmd \*Command, args []string) //该命令所执行的函数

...

}

RootCmd := &cobra.Command{...}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 如果有需要，对命令的添加flag，这一点可以简单的理解为命令选项。

RootCmd.PersistentFlags().BoolVarP(&Verbose, "verbose", "v", false, "verbose output")

RootCmd.Flags().StringVarP(&Source, "source", "s", "", "Source directory to read from")

* 1
* 2
* 如果有需要，对根命令添加子命令，子命令与根命令本质是一样的，只是人为的进行级别上的区分。

RootCmd.AddCommand(versionCmd)

* 1
* 运行命令。

RootCmd.Execute()

* 1

由于文章重点在peer，所以在此只做简单介绍，更详细的使用方法，可在go doc或github.com/spf13/cobra上学习。其实阅读fabric源码过程中有一个感觉，就是项目的大神们选用的第三方库，一般都是既满足需求，又比较容易学习和上手。

## peer命令结构解析

我们现在正式从peer/main.go文件开始解析源码，本文旨在解析peer的命令结构，因此只会涉及相关的源码，其他部分将会在其他主题文章中对应分析。如果你对cobra的用法稍微熟悉后，很容易就可以看懂main函数的构建。peer目录下的子命令的源码结构非常类似，也基本逃不出上文介绍的关于cobra的基本操作。

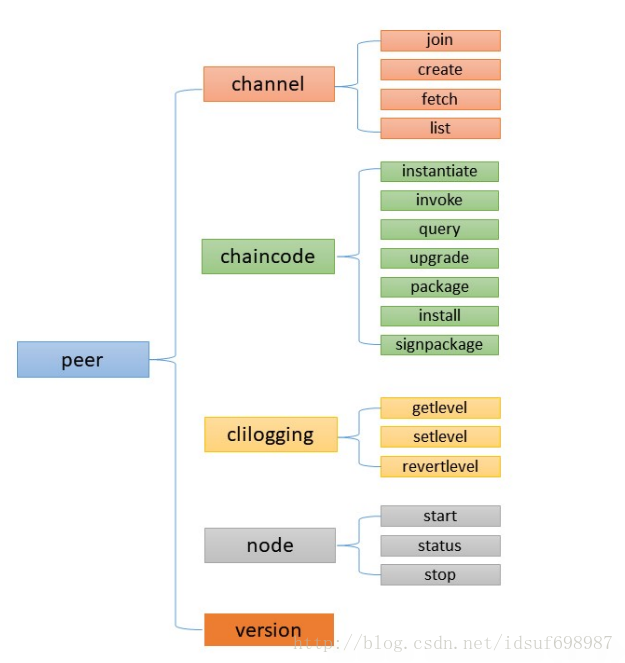
* 首先定义了一个**mainCmd**命令，**var mainCmd = &cobra.Command{…}**，该命令填充了Use，PersistentPreRunE和Run成员。Use如我们预见的那样被赋值为**peer**，PersistentPreRunE先于Run执行，都被赋值了一个匿名函数。因为mainCmd只单纯作为根命令，不实现由子命令实现的具体的交易事务，因此实现的只是PersistentPreRunE指定的检查、初始化日志系统并缓存配置的功能，和Run指定的版本打印、命令帮助功能
* 生成mainCmd对象的命令行标识对象**mainFlags**，**mainFlags := mainCmd.PersistentFlags()**，也就是peer命令的选项，并对该标识对像进行了维护，增加了version，logging\_level两个选项。这也对应了其在自身对象中设置PersistentPreRunE和Run的功能。
* 添加子命令，**mainCmd.AddCommand(…)**。添加的命令有**version.Cmd()，node.Cmd()，chaincode.Cmd(nil)，clilogging.Cmd(nil)，channel.Cmd(nil)**五个。Cmd()是每个子命令文件中暴露出的函数，各自整合了各自的动作命令。
* 启动根命令，**mainCmd.Execute()**。启动了根命令，也就启动了其下的所有命令。

## 子命令结构解析

以node为例，其实读懂了peer命令，其余的子命令类推即可。在此还是啰嗦两句吧。上文已经说了子命令的源码结构是极其相似的，这里只以node为例。

* 在node.go中，首先定义了一个node命令对象，**var nodeCmd = &cobra.Command{…}**
* 在Cmd函数中，添加了startCmd()，statusCmd()，stopCmd()三个函数返回的start，status，stop子命令（动作命令），分别实现在start.go，status.go，stop.go。这三个命令的源码结构也是基本一致，在此仅以start和start.go为例。
* 在start.go中，首先定义了一个start命令对象，**var nodeStartCmd = &cobra.Command{…}**，其中对RunE成员赋值了一个匿名函数，函数体中执行了**serve**函数，这也是该命令最终会调用的函数。serve函数是一个非常重要，非常复杂的函数。记不记得在上篇介绍Fabric项目[线头](http://blog.csdn.net/idsuf698987/article/details/74912362)的文章提到过的，在每个peer容器启动后默认执行的就是**peer node start –peer-defaultchain=false**命令，在此处就对接上了，该命令最终调用执行的就是serve函数，同时也就是说，serve函数会做了很多很多的准备工作。

## peer命令结构



# fabric源码解析3——日志系统

这里所说的日志，是指程序运行过程中打印到终端或日志文件，记录程序运行过程的日志，而非涉及到fabric记录交易数据，账本数据一类的交易日志。其实日志系统机制对于源码研究可以忽略，且相对简单，但是一方面日志系统对于程序运行和调试必不可少，fabric如何形成这个日志系统可以学习学习，日志所打印的内容对阅读源码也是一种提示；另一方面不弄清楚这个日志，在阅读源码的过程中这些到处出现的打印日志总像围绕在源码上面的蚊子一样，闹心。

fabric的日志系统主要使用了第三方包go-logging，可在github.com/op/go-logging下载。很少一部分使用了go语言标准库中的log。在此基础上fabric自己封装出来了**flogging**，这个**f**，应该代表fabric吧，意思是说这是fabric的logging。代码集中在fabric/common/flogging目录下，供项目全局使用。

## go-logging简介

简单点儿，logging其实就是封装了各种打印格式，包括消息层级上的，如DEBU、消息、注意、警告、错误，包括消息颜色上的，如消息是正常的绿色、错误则是醒目的红色。

基本用法如下：

//创建一个名字为examplename的日志对象log

var log = logging.MustGetLogger("examplename")

//创建一个日志输出格式对象format，也就是用什么格式输出

var format = logging.MustStringFormatter(

`%{color}%{time:15:04:05.000} %{shortfunc} ▶ %{level:.4s} %{id:03x}%{color:reset} %{message}`,

)

//创建一个日志输出对象backend，也就是日志要打印到哪儿，在此是标准错误输出

backend := logging.NewLogBackend(os.Stderr, "", 0)

//将输出格式与输出对象绑定

backendFormatter := logging.NewBackendFormatter(backend, format)

//将绑定了格式的输出对象设置为日志的输出对象

//这样log打印每一句话都会按格式输出到backendFormatter所代表的对象里，在此即是标准错误输出

logging.SetBackend(backendFormatter)

//log打印依据Info信息

log.Info("info")

//log打印一句Error信息

log.Error("err")

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

更多详细的用法，请在go doc或库中自行学习。

## flogging

在flogging目录下有两个文件，**grpclogger.go**和**logging.go**。

**grpclogger.go**用于设置grpc的日志，因为grpc默认只是用go语言的标准日志接口，因此将logging封装成go语言的标准日志形式的结构**type grpclogger struct {logger \*logging.Logger}**，然后通过**initgrpclogger()**生成对象供grpc使用，从而实现让grpc也使用flogging的效果。

**logging.go**文件中，自带一个名为flogging的日志记录者**logger**，同时规定了默认的日志格式，日志等级，用**defaultFormat**，**defaultLevel**常量表示。默认的输出端**defaultOutput**，并有用于存放所有fabric模块日志的级别映射**modules map[string]string**，从类型上看其存储的日志级别都字符串化了。最后还有一个存放每个peer启动之时的日志级别的映射**peerStartModules map[string]string**，由在每个peer启动完成之时调用SetPeerStartupModulesMap()初始化，并可通过调用RevertToPeerStartupLevels()恢复初始值。

### init函数

**init()**函数通过调用**Reset()**函数等初始化了一系列默认值，如默认的输出端被设置成标准错处输出，默认输出级别这设置成info级别。最后调用initgrpclogger()初始化了grpc的日志对象。

### MustGetLogger函数

在许多各级不同模块的源码中，在全局的开始处都有类似一句这样的调用，如在fabric/peer/main.go中：var logger = flogging.MustGetLogger("main")。这就是调用MustGetLogger函数生成一个名为指定了名字的日志对象，用以记录该模块的日志，并用安全的方式（用锁的方式）将该对象记录日志的级别备案到modules中。MustGetLogger函数内部依然用的是go-logging库的相应函数**logging.MustGetLogger()**生成的日志对象。

### 其他函数

在logging.go中的其他函数，基本都是了封装go-logging库函数，供fabric全局使用。如**SetModuleLevel函数**，调用来调用去，其实就是封装了go-logging库中的**logging.SetLevel()**函数，以达到符合fabric自己使用要求的目的，可谓万变不离其宗。

# fabric源码解析4——配置系统

fabric的配置系统是程序原始数据的来源之一，虽然简单却很重要。在阅读源码过程中对于具象化程序也很有帮助。在分析peer的具体交易工作之前，我们可以先分析一下fabric的配置系统。我们还将我们的目光聚焦在/fabric/peer/main.go的main函数中，除了一系列mainCmd的命令操作，还有viper进行的一系列配置操作，并通过err := common.InitConfig(cmdRoot)进行了配置的初始化。

fabric索取配置的途径有：环境变量，命令行参数，各种格式的配置文件。其中以配置文件为主，环境变量和命令行参数辅助，三者可以相互作用。主要的配置文件有core.yaml，orderer.yaml等，在/fabric/sampleconfig中有示例。主要使用的配置代码集中在/fabric/core/config下。

## viper简介

fabric的配置系统主要运用第三方包viper，可在github.com/spf13/viper下载。viper可以对系统环境变量，yaml/json等格式的配置文件甚至是远程配置进行读取和设置，并可以在不重启服务的情况下动态设置新的配置项的值并使之实时生效，是一个专门处理配置的解决方案。 而且，viper，眼镜蛇，称自己与cobra（见peer命令结构一文）是companion，足见使用viper的理由。

viper的基础用法如下：

//设置一个要读取的配置文件名（不包含后缀），一个viper只支持一个文件名

viper.SetConfigName("config")

//设置一个搜索配置文件的路径，viper的搜索路径可以有多个

viper.AddConfigPath("/etc/appname/")

viper.AddConfigPath(".")

//读取配置文件

viper.ReadInConfig()

//获取其中一个name项的值

viper.Get("name")

//将name的值设置为Bill

viper.Set("name", "Bill")

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

## viper搜索路径和文件

peer命令对core.yaml的引入也是通过viper，具体过程如下：

* /fabric/peer/main.go中定义const cmdRoot = "core"。
* main函数中调用err := common.InitConfig(cmdRoot)，该参数一路向下传递。
* InitConfig函数在/fabric/peer/common/common.go中定义，其中调用了config.InitViper(nil, cmdRoot)和viper.ReadInConfig()。
* InitViper在/fabric/core/config/config.go中定义，接收cmdRoot作为参数，最终调用了**viper.SetConfigName()**，也即将core设置为了配置文件名。
* common.InitConfig(cmdRoot)中的**viper.ReadInConfig()**则读取了该配置文件

orderer命令则使用orderer.yaml配置文件，由viper引入，具体过程如下：

* 在/fabric/orderer/main.go中main函数调用了config.Load()。
* Load在/fabric/orderer/localconfig/config.go中定义。该文件中定义了Prefix = "ORDERER"和configName string，并在**init**初始化函数中将**configName**赋值为strings.ToLower(Prefix)，即**orderer**，也即所用的配置文件名为orderer。Load函数新建了一个用于orderer自己的viper，并调用了cf.InitViper(config, configName)，其中config参数为新建的用于orderer自身的viper，configName为配置文件名orderer
* InitViper在/fabric/core/config/config.go中定义，最终调用了**viper.SetConfigName()**，也即将**orderer**设置为了配置文件名
* Load随后调用了**config.ReadInConfig()**，读取了配置文件

## InitViper

上述步骤中peer和orderer在初始化配置文件时，最终都将调用的终点指向了/fabric/core/config/config.go中定义**InitViper()**。下面集中分析InitViper。

* 首先判断环境变量**FABRIC\_CFG\_PATH**是否有值，如果有值，则是手工定义了FABRIC的配置文件所在路径。参考Getting Started中关于手工设置export FABRIC\_CFG\_PATH=$PWD（当前目录）。
* 若没有定义该环境变量的值，则用代码添加三个路径作为搜索配置文件的路径：**当前工作目录，$GOPATH/src/github.com/hyperledger/fabric/sampleconfig，/etc/hyperledger/fabric**。
  + 当前工作目录，调用addConfigPath(v, “./”)添加，其内部调用的是viper.AddConfigPath()。
  + $GOPATH/src/github.com/hyperledger/fabric/sampleconfig，通过调用AddDevConfigPath(v)添加。
    - AddDevConfigPath首先调用GetDevConfigDir()，读取GOPATH路径下是否存在src/github.com/hyperledger/fabric/sampleconfig目录，若存在，则通过filepath.Join拼接GOPATH路径下是否存在src/github.com/hyperledger/fabric/sampleconfig目录，若存在，则通过filepath.Join拼接GOPATH和src/github.com/hyperledger/fabric/sampleconfig，形成完整的路径并返回。
    - AddDevConfigPath接着调用addConfigPath函数，其内部调用的是viper.AddConfigPath()。
  + /etc/hyperledger/fabric。定义了OfficialPath = “/etc/hyperledger/fabric”常量，如果该路径存在，则调用addConfigPath加入该路径。
* 调用**SetConfigName()**设置配置文件名，所指的的配置文件名**configName**是由参数传递进来的。

由经由InitViper，形成了以下viper配置：

搜索路径（二选一）

* FABRIC\_CFG\_PATH指定的路径
* ./，$GOPATH/src/github.com/hyperledger/fabric/sampleconfig，/etc/hyperledger/fabric

搜索的配置文件名

* core —— 核心配置，供各个模块使用
* orderer —— orderer配置，orderer使用

另外注意InitViper的第一个参数，**v \*viper.Viper**。在**InitViper**函数中，无论是添加搜索路径（使用的是addConfigPath函数），还是设置要搜索的配置文件名（viper自身的SetConfigName函数），都分为全局的viper和特定的viper（也就是参数**v**）。最终由**viper**.AddConfigPath或**viper**.SetConfigName完成的，则是全局的，由**v**.AddConfigPath或**v**.SetConfigName完成的，则是特定的。这样就可以很方便的初始化需要单独使用viper的模块，如orderer就是单独使用一条毒蛇，其在/fabric/orderer/localconfig/config.go中的Load函数中，config := viper.New()新养了一条自己的蛇，然后将此蛇通过参数传给InitViper，cf.InitViper(config, configName)。

## 安全文件配置

安全配置相关的代码在/fabric/peer/main.go中没有体现，而是在/fabric/peer/node/start.go中的serve函数中才初次出现。若grpc服务中使用了TLS网络，则需要.key，.crt，.ca文件配套文件。在此简略介绍，关于TLS，将会在将来专门的文章中详细介绍。

在/fabric/peer/node/start.go中的serve函数中secureConfig, err := peer.GetSecureConfig()获取安全配置。

使用的安全配置结构为/fabric/core/comm/server.go中定义的SecureServerConfig，用于一个grpc服务端实例。   
.key，.crt，.ca文件所在的目录都是在core.yaml中定义都的tls文件夹中，当使用TLS网络时，会读取这些文件的数据到SecureServerConfig对象中。在GetSecureConfig()函数中，使用ioutil.ReadFile读取由/fabric/core/config/config.go中定义的config.GetPath(“…”)函数获取的tls路径下的相应文件。如/etc/hyperledger/fabric/tls/server.crt。

## 命令选项配置

以peer start命令为例，在/fabric/peer/node/start.go中**startCmd()**函数中，flags.BoolVarP(&chaincodeDevMode, "peer-chaincodedev", "", false,"Whether peer in chaincode development mode")设置了peer start命令的选项之一为peer-chaincodedev，用于赋值文件中的全局变量**chaincodeDevMode**，该变量指定了**chaincode的模式**。chaincode的模式在core.yaml中也有定义，chaincode.mode的值为**net**，为默认选项。而当执行peer start 命令时指定了选项peer-chaincodedev=true，也即将chaincodeDevMode赋值为true，在**serve()**函数中，就会使用viper.Set("chaincode.mode", chaincode.DevModeUserRunsChaincode)将chaincode的模式值设置成了**dev**。

## 环境变量配置

在fabric目前的阶段，各个peer都是在容器中运行的，因此环境变量指的是各个容器中的环境变量。在各个容器的启动脚本中对容器的一些环境变量也进行了设置。在peer-base-no-tls.yaml（见源码解析1——线头）中，如peer容器中，设置了如下环境变量：

- CORE\_PEER\_ADDRESSAUTODETECT=true

- CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock

- CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=e2ecli\_default

- CORE\_LOGGING\_LEVEL=ERROR

...

* 1
* 2
* 3
* 4
* 5

在fabric/peer/main.go中的开始，即对容器的环境变量进行了获取并设置：

//设置了环境变量前置，在此也就是peer

viper.SetEnvPrefix(cmdRoot)

//将环境变量加载进来了

viper.AutomaticEnv()

replacer := strings.NewReplacer(".", "\_")

//将环境变量中的\_换成.，这样就和yaml文件的配置相匹配了。

//因为viper读取yaml文件所形成的配置项就是按层级并以.分隔的格式，如peer.address

viper.SetEnvKeyReplacer(replacer)

# fabric源码分析5–kvledger的初始化

前两篇文章借由/fabric/peer/main.go这个线头，简单分析了fabric的配置和日志系统。虽然还有一部分可说的内容，如common.InitCrypto()调用，但现在暂且按下main.go不表，而把目光投到**/fabric/peer/node/start.go**中的**serve()**函数。具体原因参见peer命令结构一文的最后。

## 账本简单介绍

ledger，中文就是账本，账册的意思。先来讨论Fabric账册的原因来自于peer node start所执行的函数serve第一句代码即为ledgermgmt.Initialize()，字面上看属于对账册进行初始化操作，换句话说，对账本相关环境的初始化，是其他peer命令执行的基础之一。

### 账本源码目录

* common/ledger
* core/ledger

也就是说Fabric把关于ledger的源码把其他模块用得到的，有共享属性的部分放到了common目录下，核心的代码放到了core目录下。

### 账本数据库

Fabric中的ledger说白了就是一系列**数据库存储操作**。对应所选用的数据库，主要有两种:**goleveldb**和**couchDB**。在core.yaml配置文件中ledger区域中，stateDatabase选项即为指定所选用的数据库，默认选用goleveldb。本文章着意于ledger的操作，因此下文也只以goleveldb为例进行介绍。

>

goleveldb：主要使用了第三方库**leveldb**。下载地址github.com/syndtr/goleveldb/leveldb。leveldb是一个典型的key-value数据库，也是谷歌自己做做出来的，有多种语言版本，在此自然是使用goleveldb。把Fabric的ledger也称为kvledger的原因，这很好的体现了账本的特性，数据的操作都是基于**键-值**。关于goleveldb数据库定义，操作的代码，集中在common/ledger/util/leveldbhelper目录下。

>

couchDB：在Fabric用go语言手工实现的一个数据库，源码集中在core/ledger/util/couchdb下。couchDB在Fabric目前只用于版本数据库所使用的两个方案中的一个。

### leveldb的基本操作

1. 打开数据库，db, err:=leveldb.OpenFile("./db", nil)。作用就是在当前目录下创建一个db文件夹作为数据库的目录。
2. 存储键值，db.Put([]byte("key1"),[]byte("value1"),nil)。作用就是在数据库在数据库中存储 **键值对 key1-value1**。leveldb数据库中对键值的操作都是byte格式化的数据。
3. 获取键值对，data,\_ := db.Get([]byte("key1"),nil)。获取key1对应的值。
4. 遍历数据库，iter := db.NewIterator(nil, nil)，for iter.Next(){ fmt.Printf("key=%s,value=%s\n",iter.Key(),iter.Value()) }，iter.Release()。作用就是建立迭代器iter，然后依次遍历数据库中所有的数据并打印键和值，最后释放迭代器iter。
5. 关闭数据库，db.Close()。

## 账本对象

Fabric到处都是接口，各个层级的代码编写风格和习惯很一致，甚至使用的函数名，对象名都有大量雷同的，因而在有些源码处十分的绕，其实阅读源代码我们应该遵循以下2条原则：

1. 无论（概念上或形式上）多么复杂的对象，其本质也不过是一个结构体和挂载到该结构体一些操作函数而已。
2. 无论对象的初始化多么复杂，其本质也不过是声明后填充该对象中的各个字段的过程而已。挂载到该对象的函数无论多么复杂，也不过是对该对象中的成员所承载的数据进行增、删、改、查操作而已。

因此，追溯并查看ledgermgmt.Initialize()函数，在core/ledger/ledgermgmt/ledger\_mgmt.go中，其直接once.Do了initialize()函数。在initialize()函数中，除了日志操作和锁保护（一般在初始化的函数中，都会进行锁保护，各位可自行参看其他源码）外，所做的只是初始化了三个对象：**openedLedgers**，**ledgerProvider**和**initialized**，该三个对象均为文件中的全局变量：

initialized = true

openedLedgers = make(map[string]ledger.PeerLedger)

provider, err := kvledger.NewProvider()

ledgerProvider = provider

* 1
* 2
* 3
* 4

其中initialized和openedLedgers分别赋了初值和分配了内存，openedLedgers应该是存放peer的账本映射的，initialized则是是否初始化的标识，两者并未有进一步操作，可暂时搁置一旁。而ledgerProvider则被赋于由kvledger.NewProvider()函数返回的值，从其名字我们就可以知道该对象是一个账本服务提供者，因此我们的目光停留在此即可。其原型为ledger.PeerLedgerProvider接口，在core/ledger/ledger\_interface.go中定义：

type PeerLedgerProvider interface {

Create(genesisBlock \*common.Block) (PeerLedger, error)

Open(ledgerID string) (PeerLedger, error)

Exists(ledgerID string) (bool, error)

List() ([]string, error)

Close()

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

根据Fabric的惯例，有Provider字样的对象，或大或小，都是某一主题模块服务的提供者，提供该主题模块的一系列操作服务。而接口类型的Provider对象，则在具体实现上则会分为多种具体的Provider以供使用（同时也留下了扩展空间）。ledger的Provider亦然如此，kvledger.NewProvider()函数返回的就是PeerLedgerProvider接口的一个具体实现，kv ledger provider，即**键值账本提供者**，在core/ledger/kvledger/kv\_ledger\_provider.go中定义：

type Provider struct {

idStore \*idStore //ledgerID数据库

blockStoreProvider blkstorage.BlockStoreProvider //block数据库存储服务对象

vdbProvider statedb.VersionedDBProvider //状态数据库存储服务对象

historydbProvider historydb.HistoryDBProvider //历史数据库存储服务对象

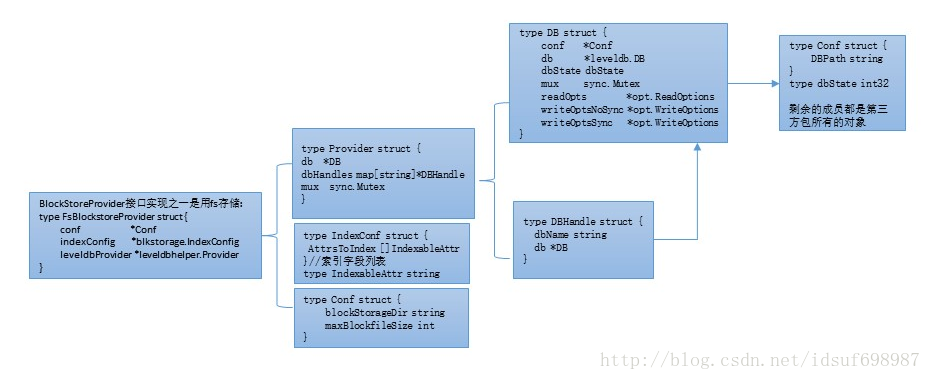
}

* 1
* 2
* 3
* 4
* 5
* 6

还是根据Fabric的惯例，在每个定义对象结构的文件里，通常都会有一个专门用于生成该对象的函数，kvledger.NewProvider()既是用于生成键值账本服务提供者的函数。通过追溯，我们会发现，这个对象中的四个成员对象其实就是四个数据库，分别用于存储不同的数据，也是账册存储所需要的。而kvledger.NewProvider()所做的，就是分别按照配置生成这四个数据库对象，这也符合和上文我们所说的两个原则。四个数据库除了idStore和blockStoreProvider有自己特殊的配置外，都共同使用的leveldb数据库存储服务提供者，后文将以块数据库服务对象为例，详细介绍。

## 块数据库存储服务对象

以block数据库存储服务对象blockStoreProvider的结构为例，其代码集中在commom/ledger/blkstorage下（本节中除介绍leveldb数据库存储服务对象外，所示路径皆以此路径为基准）。blockStoreProvider的原型BlockStoreProvider依然是个接口，在blockstorage.go中定义此接口，具体实现为用文件系统存储，即fsblkstorage/fs\_blockstore\_provider.go中定义的FsBlockstoreProvider，对象结构如下：



也就是说，与块数据存储服务对象blockStoreProvider最终对接的是三个成员，其中两个配置项成员**conf**和**indexConfig**，是相较于其他数据库服务对象所独有的，一个leveldb数据库存储服务提供者**leveldbProvider**，则和其他数据库服务对象一样。而专门用于初始化FsBlockstoreProvider的函数即为fsblkstorage.NewProvider()。

在kvledger.NewProvider()中，以下三句代码用于初始化blockStoreProvider对象：

attrsToIndex := []blkstorage.IndexableAttr{

blkstorage.IndexableAttrBlockHash,

blkstorage.IndexableAttrBlockNum,

blkstorage.IndexableAttrTxID,

blkstorage.IndexableAttrBlockNumTranNum,

blkstorage.IndexableAttrBlockTxID,

blkstorage.IndexableAttrTxValidationCode,

}

indexConfig := &blkstorage.IndexConfig{AttrsToIndex: attrsToIndex}

//传递两个配置项，并在内部使用专用函数生成一个leveldb数据库对象，最终生成块数据存储服务对象

blockStoreProvider := fsblkstorage.NewProvider(

fsblkstorage.NewConf(ledgerconfig.GetBlockStorePath(),

ledgerconfig.GetMaxBlockfileSize()),

indexConfig)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

### 块索引配置indexConfig

用于存储块索引字段值，这点blockstorage.go中进行了硬编码。可以将其想象成数据库表中准备为哪些字段建立索引，因而在此记录一下。

### 块存储配置conf

该配置对象在fsblkstorage/config中定义，两个字段**blockStorageDir**和**maxBlockfileSize**指定了块数据库存储服务对象所使用的路径和存储文件的大小。在**fsblkstorage.NewProvider()**中，传入的config是用NewConf(ledgerconfig.GetBlockStorePath(),ledgerconfig.GetMaxBlockfileSize())进行创建的，而NewConf又使用了ledgerconfig下的函数分别获取了路径和大小。而通过追溯ledgerconfig，发现其最终形成的路径值为/var/hyperledger/production/ledgersData/chains，大小为64M（关于路径和配置，将在以配置和路径为主题的文章中详细介绍）。

### leveldb数据库存储服务对象blockStoreProvider

实际的，最终操作数据库数据的对象，账本所使用的四个数据库服务对象均使用此数据库对象对数据进行操作。在common/ledger/util/leveldbhelper/leveldb\_provider.go中定义：

type Provider struct {

db \*DB

dbHandles map[string]\*DBHandle

mux sync.Mutex

}

* 1
* 2
* 3
* 4
* 5

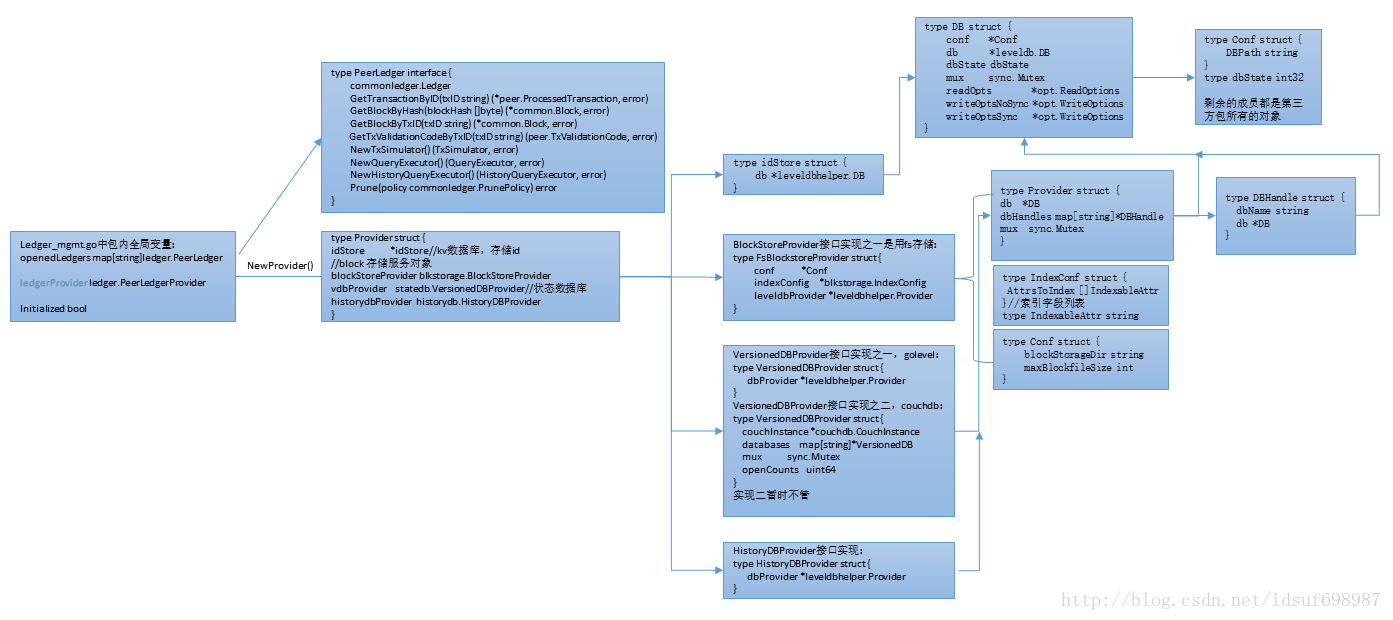
leveldb数据库存储服务对象包含了封装leveldb数据库对象的**db**，和一个数据库映射**dbHandles**，和一把保护锁**mux**。因为数据库是对数据的读写操作，因此有一把保护锁很正常也很应该。其处于同一个文件中专用的初始化函数leveldbhelper.NewProvider()，在fsblkstorage.NewProvider()函数中即有体现：p := leveldbhelper.NewProvider(&leveldbhelper.Conf{DBPath: conf.getIndexDir()})

在此仍需关注的是**DB**结构中自带一个leveldbhelper.Conf配置选项，定义了leveldb数据库所在的目录。在leveldbhelper.NewProvider()初始化的过程中，使用了上层对象-**块数据库存储服务对象**中的conf所挂载的函数getIndexDir()，在common/ledger/blkstorage/fsblkstorage/config.go中定义，获取了一个路径，最终被初始化在chains/index下。

## 账本服务对象的目录结构

回到**kvledger.NewProvider()**函数中，其他几个数据库的初始化过程和**块数据库存储服务对象**类似，但更简单一些，基本都只是用专用函数初始化了一个leveldb数据库存储服务对象。至此，整个账本服务对象初始化完毕。以下列出账本服务对象整个对象的结构和所形成的目录结构，具象化一下。

**对象结构**：



**目录结构**：

>

* /var/hyperledger/production(core.yaml定义的flieSystemPath的值)   
  + ledgersData //账本目录   
    - ledgerProvider //ledgerID数据库目录
    - chains //block块存储数据库目录   
      * index //block索引数据库目录
      * chains   
        + 账本ID1
        + 账本ID2
        + …
    - stateLeveldb //状态数据库目录
    - historyLeveldb //历史数据库目录

在kvledger.NewProvider()函数中接近结尾的地方，有一句代码需要注意：provider.recoverUnderConstructionLedger()，该句调用了账本服务对象的一个函数，主要是用于恢复处理一些之前账本初始化失败的操作，从中牵扯出了一大堆函数调用。但对这些操作的理解需要建立在理解账本操作的基础之上，因此在此埋下伏笔，之后的文章中将回过头来详细介绍此句。

# fabric源码解析6–GRPC服务

## GRPC简介

GRPC是由自谷歌开发的一项多语言开源的RPC技术，在fabric用于实现客户端与服务器端的远程调用。比如chaincode，客户定义了一项rpc服务并相应生成了客户端代码和服务端代码，在此基础上进行业务逻辑上的开发后，分别运行服务端代码和客户端代码，实现客户端调用服务器端服务的目的。由于gprc相对来说还是很复杂的，所有还请自行学习。在[gprc官网](https://blog.csdn.net/idsuf698987/article/details/grpc.io/)非常详细的资料和例子。

极其粗线条的写一下grpc的用法，为后文伏笔：

**1. XXX.proto文件中定义一个rpc服务**

service Events {

rpc Chat(stream SignedEvent) returns (stream Event) {}

}

* 1
* 2
* 3

**2. 命令行使用protoc生成对应的XXX.pb.go源码**

在XXX.pb.go中，**Client API for Events service**处为供**客户端**使用的**接口定义、接口实例、接口实例的初始化函数**。**Server API for Events service**处为供**服务端**使用的**接口定义**，**注册函数**。

如果其中某一端或同时两端为流式RPC，在有流的一端，会专门为其流生成接口定义、接口实例。可以直接使用生成的实例，也可以自己实现接口，自定义实例。接口定义的主要方法就是**Send**和**Recv**。

protoc --go\_out=plugins=grpc:. XXX.proto

**3. 编写客户端代码**

//注意，由于目前我们关注的是peer node start，而其启动的基本都是后台服务端的服务，

//因此本文中不涉及客户端的代码。

//填充grpc网络链接连接选项

var opts []grpc.DialOption

opts = append(opts, grpc.WithInsecure())

//创建连接服务器端的grpc连接对象

conn, err := grpc.Dial("0.0.0.0:7051", opts...)

defer conn.Close()

//使用连接对象做参数，利用XXX.pb.go中的初始化函数创建grpc客户端对象

client := NewEventsClient(conn)

//调用服务

client.Chat(...)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

**4. 编写服务端代码**

//定义一个监听对象，即服务器监听的地址

lis,err := net.Listen("tcp",":7051")

//创建grpc服务器选项并填充

var serverOpts []grpc.ServerOption

//创建标准的grpc服务器对象

server = grpc.NewServer(serverOpts...)

//创建服务端对象，根据XXX.pb.go中生成的接口定义，自己实现服务端接口

type eventSever{...}

func (e \*eventSever)Chat(...){...}

es := new(eventServer)

//使用XXX.pb.go中的注册函数注册服务，注册到grpc服务器对象

RegisterEventsServer(server,es)

//根据监听对象启动grpc服务器对象

server.Serve(lis)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

## fabric中的grpc服务接口和实例

在**/fabric/core/comm/server.go**中，定义了安全配置项，GPRCServer的接口、实现和初始化函数。默认情况下fabric中是不使用tls的。

### TLS安全配置项

type SecureServerConfig struct {

//Whether or not to use TLS for communication

UseTLS bool

//PEM-encoded X509 public key to be used by the server for TLS communication

//在core.yaml中指定，读取的tls目录下server.cert文件数据存储于此

ServerCertificate []byte

//PEM-encoded private key to be used by the server for TLS communication

//在core.yaml中指定，读取的tls目录下server.key文件数据存储于此

ServerKey []byte

//Set of PEM-encoded X509 certificate authorities to optionally send

//as part of the server handshake

//在core.yaml中指定，读取的tls目录下ca.crt文件数据存储于此

ServerRootCAs [][]byte

//Whether or not TLS client must present certificates for authentication

RequireClientCert bool

//Set of PEM-encoded X509 certificate authorities to use when verifying

//client certificates

ClientRootCAs [][]byte

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

### GRPCServer接口

type GRPCServer interface {

//返回GRPCServer监听的地址

Address() string

//启动下层的grpc.Server

Start() error

//停止下层的grpc.Server

Stop()

//返回GRPCServer实例对象

Server() \*grpc.Server

//返回GRPCServer使用的网络监听实例对象

Listener() net.Listener

//返回grpc.Server使用的Certificate

ServerCertificate() tls.Certificate

//标识GRPCServer实例是否使用TLS

TLSEnabled() bool

//增加PEM-encoded X509 certificate authorities到

//用于验证客户端certificates的authorities列表

AppendClientRootCAs(clientRoots [][]byte) error

//用于验证客户端certificates的authorities列表中

//删除PEM-encoded X509 certificate authorities

RemoveClientRootCAs(clientRoots [][]byte) error

//基于一个PEM-encoded X509 certificate authorities列表

//设置用于验证客户端certificates的authorities列表

SetClientRootCAs(clientRoots [][]byte) error

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

### GRPCServer实现实例

type grpcServerImpl struct {

//server指定的监听地址，地址格式：hostname:port

address string

//监听address的监听对象，用于处理网络请求

listener net.Listener

//标准的grpc服务器，通过此对象进行各种grpc服务操作

server \*grpc.Server

//Certificate presented by the server for TLS communication

serverCertificate tls.Certificate

//Key used by the server for TLS communication

serverKeyPEM []byte

//List of certificate authorities to optionally pass to the client during

//the TLS handshake

serverRootCAs []tls.Certificate

//lock to protect concurrent access to append / remove

lock \*sync.Mutex

//Set of PEM-encoded X509 certificate authorities used to populate

//the tlsConfig.ClientCAs indexed by subject

clientRootCAs map[string]\*x509.Certificate

//TLS configuration used by the grpc server

tlsConfig \*tls.Config

//Is TLS enabled?

tlsEnabled bool

}

同文件中的NewGRPCServerFromListener函数是grpcServerImpl的初始化函数，其中tls相关代码使用到了crypto下的tls、x509工具库。

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

## peer node start启动的grpc服务

在**start.go**中**serve函数**中，创建的GRPCServer服务对象有两个：

* peerServer
* globalEventsServer

peer服务器**peerServer**，在**/fabric/core/peer/peer.go**中定义。事件服务器**globalEventsServer**，是一个全局单例，在**/fabric/events/producer/producer.go**中定义。

## peerServer

### 创建peerServer

追溯serve函数中peerServer对象的创建，代码最终都使用了**/fabric/core/comm/server.go**中的**NewGRPCServerFromListener**函数创建了一个**grpcServerImpl**实例对象，对象中的。

//在serve函数中

peerServer, err := peer.CreatePeerServer(listenAddr, secureConfig)

//在CreatePeerServer函数中

peerServer, err = comm.NewGRPCServer(listenAddress, secureConfig)

//在NewGRPCServer函数中

lis, err := net.Listen("tcp", address)

NewGRPCServerFromListener(lis, secureConfig)

//在NewGRPCServerFromListener函数中，最终建立grpc标准服务器并返回grpcServerImpl

grpcServer.server = grpc.NewServer(serverOpts...)

return grpcServer

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

### 注册服务

注册**ChaincodeSupport服务**

这是我们第一次遇到ChaincodeSupport这个对象，分为**ChaincodeSupport服务原型**和对应定义的**ChaincodeSupport对象**。从名称上就可以看出，是对fabric的chaincode提供一系列。自然，peer的grpc服务中关于chaincode的操作需要这种支持服务。而**ChaincodeSupport对象**本身比较复杂，在/fabric/core/chaincode/chaincode\_support.go中定义，提供了一干配置值成员和chaincode的运行环境，也实现了很多接口，如该处提到的服务所需的Register函数。因在此侧重与grpc服务，因此不展开详述。

**服务原型定义：**

ChaincodeSupport服务原型在**/fabric/protos/peer/chaincode\_shim.proto**中定义，相应生成**chaincode\_shim.pb.go**源码，在此只展示其中生成的服务端定义。

//服务原型   
service ChaincodeSupport {   
rpc Register(stream ChaincodeMessage) returns (stream ChaincodeMessage) {}   
}   
//生成服务端的接口和注册函数   
type ChaincodeSupportServer interface {   
Register(ChaincodeSupport\_RegisterServer) error   
}   
func RegisterChaincodeSupportServer(s \*grpc.Server, srv ChaincodeSupportServer){   
s.RegisterService(&\_ChaincodeSupport\_serviceDesc, srv)   
}   
//生成的服务端流的接口定义、接口实例   
type ChaincodeSupport\_RegisterServer interface {   
Send(\*ChaincodeMessage) error   
Recv() (\*ChaincodeMessage, error)   
grpc.ServerStream   
}//接口   
type chaincodeSupportRegisterServer struct {   
grpc.ServerStream   
}//接口实口

**注册服务：**

在serve函数中使用registerChaincodeSupport(peerServer.Server())完成注册。在该函数中：

//创建了一个ChaincodeSupport对象，基本都是读取配置值填充成员   
//ChaincodeSupport对象实现了生成的服务端接口ChaincodeSupportServer中的Register方法   
ccSrv := chaincode.NewChaincodeSupport(...)   
//利用生成的注册函数，完成注册   
pb.RegisterChaincodeSupportServer(grpcServer, ccSrv)

在registerChaincodeSupport中还有一句scc.RegisterSysCCs()实现了系统链的注册，将在系统链主题文章中进行详述。

**Register实现：**

将在ChaincodeSupport主题文章中详述。

**注册的其他服务**

关于peerServer所注册的服务，还有**AdminServer**、**EndorserServer**、**GossipService**，注册的方式和注册**ChaincodeSupport**服务一样，毕竟都是用的gprc，还是万变不离其宗的。各个服务到底负责什么，会做什么，将会在相应主题文章中详述。

## globalEventsServer

//在/fabric/events/producer/producer.go中定义

//全局单例

var globalEventsServer \*EventsServer

//定义和Chat实现

type EventsServer struct {

}

func (p \*EventsServer) Chat(stream pb.Events\_ChatServer) error {...}

//初始化函数

func NewEventsServer(bufferSize uint, timeout int) \*EventsServer {...}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

事件服务器这个全局单例没有任何成员，只有一个专用初始化函数**NewEventsServer**，一个**Chat**实现。这一切看上去非常简单，只是在专用初始化函数中的一句initializeEvents(bufferSize, timeout)又牵扯出了一段文字。原来globalEventsServer自己只是一个事件服务器的代表，实际做事情的是initializeEvents(bufferSize, timeout)初始化并运行的**eventProcessor对象**，下文细说。

在serve函数中，使用ehubGrpcServer, err := createEventHubServer(secureConfig)完成了对事件服务器的创建和注册，ehubGrpcServer承接的就是globalEventsServer这个全局单例。

### 创建globalEventsServer

//在createEventHubServer中

//创建grpcServerImpl对象，其中包含了grpc标准服务器

lis, err = net.Listen("tcp", viper.GetString("peer.events.address"))

grpcServer, err := comm.NewGRPCServerFromListener(lis, secureConfig)

//创建事件服务器，NewEventsServer返回的就是globalEventsServer

ehServer := producer.NewEventsServer(

uint(viper.GetInt("peer.events.buffersize")),

viper.GetInt("peer.events.timeout"))

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

### 注册事件服务

**事件服务原型定义：**

事件服务原型在**/fabric/protos/peer/events.proto**中定义，相应生成**events.pb.go**源码，在此只展示其中生成的服务端定义。

//服务原型   
service Events {   
rpc Chat(stream SignedEvent) returns (stream Event) {}   
}   
//生成服务端的接口和注册函数   
type EventsServer interface {   
Chat(Events\_ChatServer) error   
}   
func RegisterEventsServer(s \*grpc.Server, srv EventsServer) {   
s.RegisterService(&\_Events\_serviceDesc, srv)   
}   
//生成的服务端流的接口定义、接口实例   
type Events\_ChatServer interface {   
Send(\*Event) error   
Recv() (\*SignedEvent, error)   
grpc.ServerStream   
}//接口   
type eventsChatServer struct {   
grpc.ServerStream   
}//接口实例

**注册服务：**

//还是在createEventHubServer中   
//ehServer对象实现了生成的服务端接口EventsServer的Chat方法   
//利用生成的注册函数，完成注册   
pb.RegisterEventsServer(grpcServer.Server(), ehServer)

**Chat实现：**

Chat的操作很清晰，循环的接收数据然后处理数据，即处理客户端的Chat调用，这也自然而然的是grpc服务端所要做的。

* handler, err := newEventHandler(stream)，根据服务端流接口stream创建一个handler，用于处理接收的客户端发送的**SignedEvent**类型数据。handler于后文详述。
* in, err := stream.Recv()，接收**SignedEvent**类型的数据，这也是gprc双向流的标准用法。
* err = handler.HandleMessage(in)，使用handler处理数据，HandleMessage是实际的数据处理函数。

在HandleMessage函数中，客户端发送签名过的SignedEvent类型数据，检查有效性后，若是注册或注销事件，则注册或注销，并返回Event类型数据；若是其他类型的事件，则打印一条错误消息后返回。

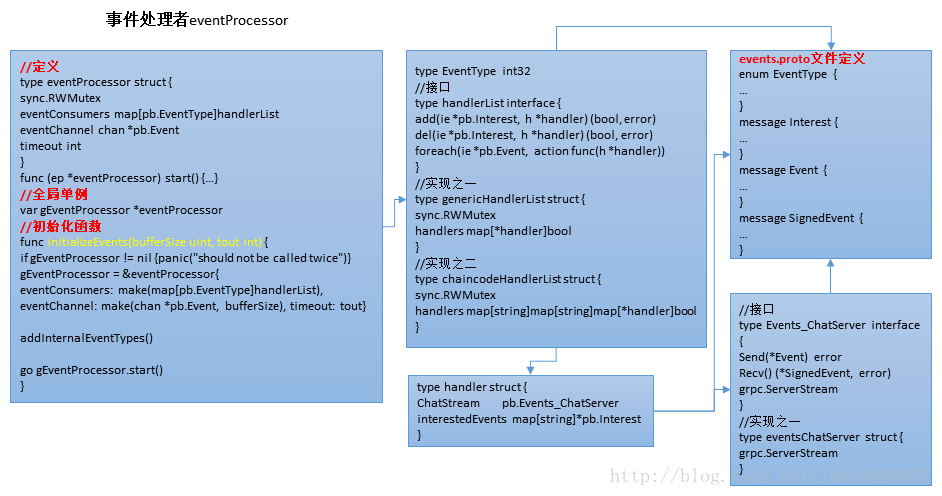
* evt, err := validateEventMessage(msg)，利用local MSP验证数据的有效性。关于local MSP将在对应主题文章中详述。
* switch evt.Event.(type) {...}，判断事件类型，并对注册事件或注销事件进行注册或注销。
* if err := d.ChatStream.Send(evt);err != nil{...}，若是注册事件或注销事件，执行相应操作之后返回Event数据给客户端，该数据是在验证函数validateEventMessage中获取的。

### 启动事件服务

在serve函数中靠后的地方，if ehubGrpcServer != nil {go ehubGrpcServer.Start()}将该服务的gprc服务端启动起来了。Start内部调用了grpc服务器启动的标准函数server.Serve(lis)。

## 事件实际处理者eventProcessor

事件处理者也是一个全局单例，接收不同类型的事件进行处理。在**/fabric/events/events.go**中定义，结构如下：



在eventProcessor的成员中：

**eventConsumers**

按照事件类型分类的事件处理链条，处理链**handlerList**接口有两种具体实现：一般处理链**genericHandlerList**和chaincode专用处理链**chaincodeHandlerList**。在此以一般处理链为例，其实现了对**handlers**三个操作：**add，del，foreach**。其中遍历操作foreach则对handlers中的每个**handler**执行了由**参数**指定的动作。这里的handlers映射了handler与bool值，bool值应该是起到类似于**开关**的作用。

**handler**在**/fabric/events/handle.go**中定义，其成员**ChatStream**是一个events.pb.go中生成的Events\_ChatServer类型的gprc流接口，用于发送流数据。handler挂载了一系列操作函数，如**register**，**HandleMessage**，**SendMessage**，**Stop**。

**eventChannel**

带缓存且专门处理**Event类型**数据的事件频道，所有的事件都是通过此频道分发出去的。缓存大小由core.yaml定义为100，由initializeEvents的参数带进来并设置。Event类型在由**events.proto**中定义并对应生成**events.pb.go**中定义。

**timeout**

频道eventChannel若满时等待的时间，在core.yaml中设置并有释义。

我们将从其初始化函数，也就是上文提到的**initializeEvents**入手，分析事件处理者eventProcessor。

### 事件类型

initializeEvents的前两句很容易理解，if gEventProcessor != nil{...}只为保证gEventProcessor的单例性质，gEventProcessor = &eventProcessor{...}则为gEventProcessor创建了对象实例，分配了内存空间。而addInternalEventTypes()，实质4次调用了AddEventType，则为添加内部事件类型并相应的分配了这些类型各自的处理链handlerList。

添加的已知的事件类型由/fabric/protos/peer/events.proto中定义，对应生成的events.pb.go中的四种：

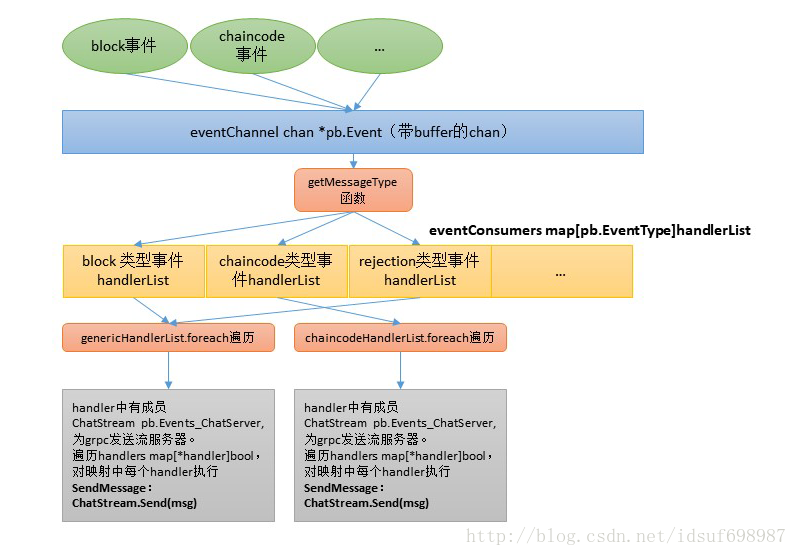
* EventType\_BLOCK - 块事件，对应genericHandlerList
* EventType\_CHAINCODE - chaincode事件，对应chaincodeHandlerList
* EventType\_REGISTER - addInternalEventTypes中有但是AddEventType未做处理
* EventType\_REJECTION - 拒绝事件，对应genericHandlerList

### start函数

initializeEvents最后一句就是go gEventProcessor.start()，就是另起一个goroutine运行全局单例gEventProcessor的start函数。start函数是一个死循环，不断从eventChannel中接收数据Event类型数据并处理。过程如下：

* e := <-ep.eventChannel，获取一个事件**e**。
* eType := getMessageType(e)，获取事件e的类型**eType**。
* if hl, \_ = ep.eventConsumers[eType]; hl == nil {...}，根据eType获取该事件类型的处理链**hl**，同时判断该类型是否存在，若不存在则会被忽略本次事件而continue继续处理下一个事件。
* hl.foreach(...)，调用hl的**foreach函数**，foreach遍历了hl.handlers中的每个handler，并对每个handler执行第二个参数指定的动作。该动作为func(h \*handler){if e.Event != nil{h.SendMessage(e)}}，即调用每个handler的**SendMessage**发送事件**e**。SendMessage则是使用handler中自有的**ChatStream**这个生成的grpc流服务接口去发送事件e：err := d.ChatStream.Send(msg)。

流程图如下：



# fabric源码解析7——peer的ChaincodeSupport服务

## 综述

fabirc源码解析6中讲述了peer结点如何创建和注册grpc服务，接下来的几篇文章将对peer注册的各个服务进行详述。该篇讲述ChaincodeSupport服务，ChaincodeSupport服务为每个peer提供了chaincode操作的支持。registerChaincodeSupport(peerServer.Server())一句，位于**/fabric/peer/node/start.go**文件中的**serve**函数中，给peerServer注册了**ChaincodeSupport**服务。

ChaincodeSupport的服务原型和生成的go定义在**/fabric/protos/peer/**下的**chaincode\_shim.proto**和**chaincode\_shim.pb.go**中，核心的实现代码在**/fabric/core/chaincode/chaincode\_support.go**中。主要的定义的是一个rpc Register(stream ChaincodeMessage) returns (stream ChaincodeMessage) {}服务。该服务实现客户端和服务器端**ChaincodeMessage**类型流数据的交换。用于**服务端**流数据交换的grpc流服务接口象为**/fabric/protos/peer/chaincode\_shim.pb.go**中的**ChaincodeSupport\_RegisterServer**，在**/fabric/core/container/ccintf/ccintf.go**中有对应用于容器内部间的流接口**ChaincodeStream**。

ChaincodeSupport的服务是一个**全局单例**，该单例对象定义在chaincode\_support.go中，var theChaincodeSupport \*ChaincodeSupport。ChaincodeSupport对象自身存储一系列配置值，而接收和处理客户端ChaincodeMessage类型消息的任务其实是委托给了一个个Handler对象。

//生成的收发的数据类型

type ChaincodeMessage struct {

Type ChaincodeMessage\_Type

Timestamp \*google\_protobuf1.Timestamp

Payload []byte

Txid string

Proposal \*SignedProposal

ChaincodeEvent \*ChaincodeEvent

}

//proto中ChaincodeSupport服务原型

service ChaincodeSupport {

rpc Register(stream ChaincodeMessage) returns (stream ChaincodeMessage) {}

}

//生成的服务端流接口

type ChaincodeSupport\_RegisterServer interface {

Send(\*ChaincodeMessage) error

Recv() (\*ChaincodeMessage, error)

grpc.ClientStream

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

## FSM

FSM是finite state machine的缩写，有限状态机，是ChaincodeSupport服务使用到的一个第三方库，在github.com/looplab/fsm可以下载。FSM将一个事物从状态A向状态B的转化看作一个事件，并可以设置在进入/离开某个状态时自动调用的时机函数。每个**状态事件、状态、时机函数**都用**字符串关键字**表示。在此简单介绍一下用法：

//创建一个状态机

//三个参数：1.默认状态 2.定义状态事件 3.定义状态转变时调用的函数

fsm := fsm.NewFSM(

"green",

fsm.Events{

//状态事件的名称 该事件的起始状态Src 该事件的结束状态Dst

//即：状态事件warn（警告事件）表示事物的状态从状态green到状态yellow

{Name: "warn", Src: []string{"green"}, Dst: "yellow"},

{Name: "panic", Src: []string{"yellow"}, Dst: "red"},

{Name: "calm", Src: []string{"red"}, Dst: "yellow"},

},

//状态事件调用函数，在此称为 时机函数。关键字用'\_'隔开，格式是："调用时机\_事件或状态"

//before表示在该事件或状态发生之前调用该函数，如"before\_warn"，表示在warn

//这个状态事件发生前调用这个函数。"before\_yellow"表示进入yellow状态之前调用

//该函数。

//依此类推，after表示在...之后，enter表示在进入...之时，leave表示在离开...

//之时。

fsm.Callbacks{

//fsm内定义的状态事件函数，关键字指定的是XXX\_event和XXX\_state

//表示任一的状态或状态事件

"before\_event": func(e \*fsm.Event) {

fmt.Println("before\_event")

},

"leave\_state": func(e \*fsm.Event) {

fmt.Println("leave\_state")

},

//根据自定义状态或事件所定义的状态事件函数

"before\_yellow": func(e \*fsm.Event) {

fmt.Println("before\_yellow")

},

"before\_warn": func(e \*fsm.Event) {

fmt.Println("before\_warn")

},

},

)

//打印当前状态，输出是默认状态green

fmt.Println(fsm.Current())

//触发warn状态事件，状态将会从green转变到yellow

//同时触发"before\_warn"、"before\_yellow"、"before\_event"、"leave\_state"函数

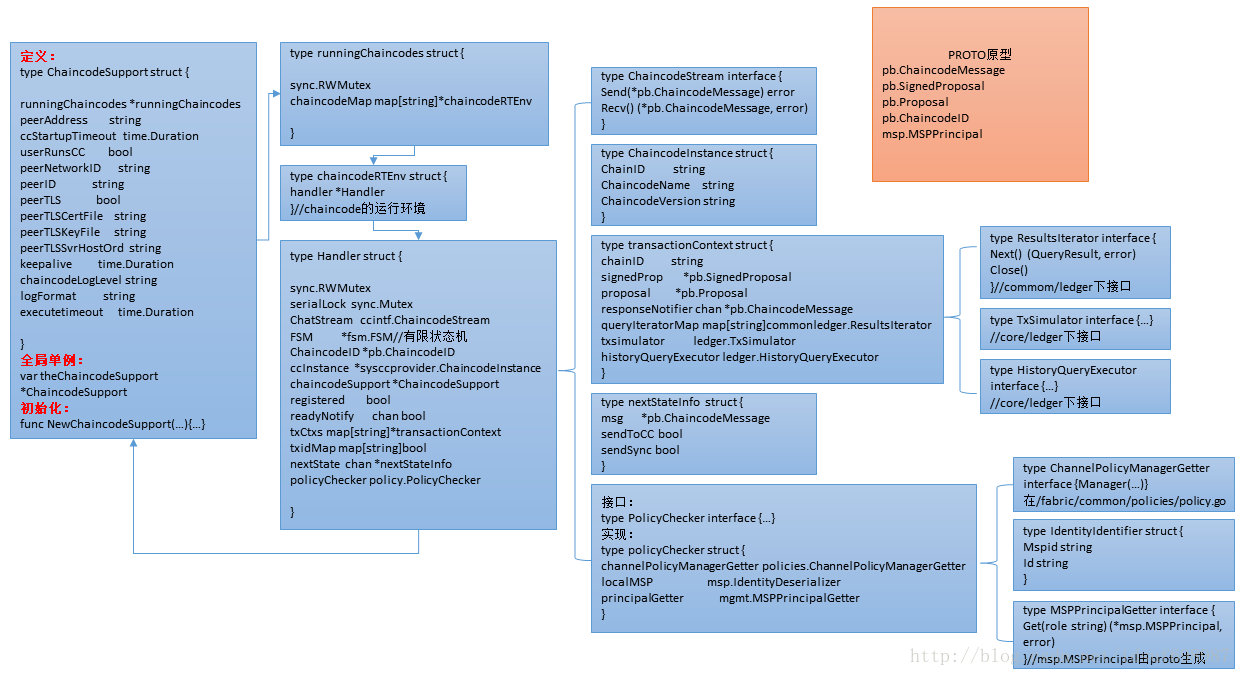
fsm.Event("warn")

//打印当前状态，输出状态是yellow

fmt.Println(fsm.Current())

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42

## 结构概览：



任何项目中，服务是以所能提供的操作为中心的，ChaincodeSupport服务的操作（即可被外部调用的函数）有**Launch，Register，Execute，HandleChaincodeStream，Stop**。

## Register

追溯ChaincodeSupport对象挂载的**Register**函数，最终调用的是/fabric/core/chaincode/handler.go中的**HandleChaincodeStream**函数。在HandleChaincodeStream函数中：

handler := newChaincodeSupportHandler(chaincodeSupport, stream)

handler.processStream()

* 1
* 2

创建了一个Handler，然后调用Handler的processStream函数对客户端发送的流数据进行了处理。这两个函数都在同文件中实现。newChaincodeSupportHandler函数所传入的两个参数值得注意，一个是**chaincodeSupport**，一个是**stream**。前者是**Register服务**所在的**ChaincodeSupport对象自身**，赋值给了Hanlder对象成员**chaincodeSupport**，为的是让Handler对象处理接收数据时能够使用ChaincodeSupport对象的服务；后者是**Register服务**的grpc流接口，赋值给了Handler对象成员**ChatStream**，为的是Handler能够从客户端接收到数据。后文还会提到这点。

### Handler

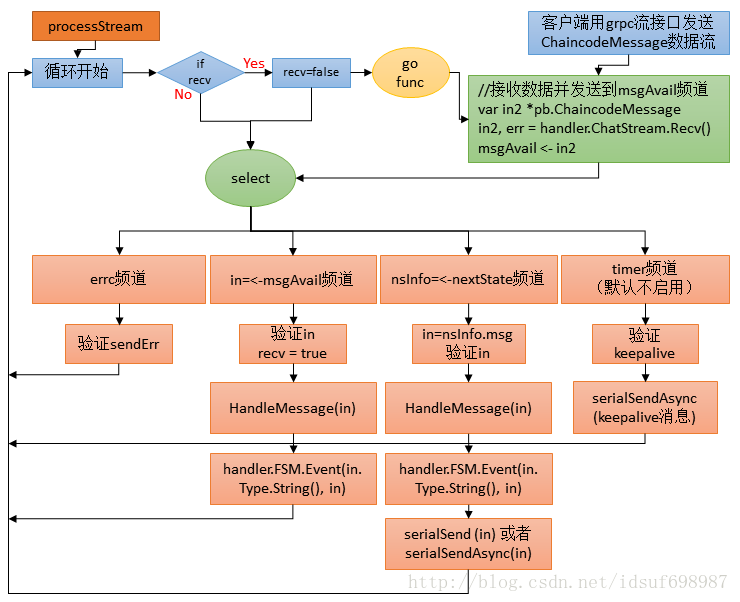
newChaincodeSupportHandler创建并初始化了一个Handler，初始化的成员有：   
\* ChatStream - grpc流服务接口，是用Register函数传进来的。   
\* chaincodeSupport - chaincodeSupport自身。   
\* nextState - 状态通道。   
\* FSM - 状态机，参看上文。   
\* policyChecker - 策略检查器，将在对应主题文章中详述。

### processStream

processStream用**recv标识**、**| errc | msgAvil | nextState | keepalivetimer |**四个频道、**select**三者相互配合，形成了对客户端消息的接收控制。然后调用**HandleMessage、serialSend、serialSendAsync**处理接收到的消息。

* errc - 错误频道
* msgAvil - ChaincodeMessage频道
* nextState - 包含ChaincodeMessage的频道
* keepalivetime - 心跳频道

流程如下：



### HandleMessage

HandleMessage处理ChaincodeMessage数据的方式完全是由Handler中的**状态机FSM**驱动的。在newChaincodeSupportHandler有大段代码是初始化其状态机的：

v.FSM = fsm.NewFSM(createdstate,fsm.Events{...},fsm.Callbacks{...})

* 1

状态机**FSM**所注册的**状态事件**有：

///fabric/protos/peer/chaincode\_shim.pb.go中定义

//REGISTER即pb.ChaincodeMessage\_REGISTER.String()对应的字符串值，下同

//REGISTER事件表示从状态createdstate到状态establishedstate，下略。

REGISTER Src: []string{createdstate}, Dst: establishedstate

READY

PUT\_STATE

DEL\_STATE

INVOKE\_CHAI

COMPLETED

GET\_STATE

GET\_STATE\_B

GET\_QUERY\_R

GET\_HISTORY

QUERY\_STATE

QUERY\_STATE

ERROR

RESPONSE

INIT

TRANSACTION

RESPONSE

INIT

TRANSACTION

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

状态机**FSM**所涉及的**事件状态**有：

//在/fabric/core/chaincode/handler.go中以常量的形式定义

createdstate = "created"

establishedstate = "established"

readystate = "ready"

endstate = "end"

* 1
* 2
* 3
* 4
* 5

状态机**FSM** 状态事件所调用的**时机函数**为：

//在REGISTER事件发生之前调用beforeRegisterEvent，下同。

"before\_REGISTER" : beforeRegisterEvent

"before\_COMPLETED" : beforeCompletedEvent

"after\_GET\_STATE" : afterGetState

"after\_GET\_STATE\_BY\_RANGE" : afterGetStateByRange

"after\_GET\_QUERY\_RESULT" : afterGetQueryResult

"after\_GET\_HISTORY\_FOR\_KEY" : afterGetHistoryForKey

"after\_QUERY\_STATE\_NEXT" : afterQueryStateNext

"after\_QUERY\_STATE\_CLOSE" : afterQueryStateClose

"after\_PUT\_STATE" : enterBusyState

"after\_DEL\_STATE" : enterBusyState

"after\_INVOKE\_CHAINCODE" : enterBusyState

//表示在进入established状态之时调用enterEstablishedState，下同。

"enter\_established" : enterEstablishedState

"enter\_ready" : enterReadyState

"enter\_end" : enterEndState

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

在**HandleMessage**函数中，对传入的数据msg简单验证后，eventErr := handler.FSM.Event(msg.Type.String(), msg)触发了状态机的**状态事件**，进而触发了对应的**时机函数**。

以**“REGISTER类型的ChaincodeMessage”**为例。客户端通过grpc发送REGISTER类型的ChaincodeMessage信息，服务端通过msgAvil频道接收后传入HandlerMessage函数，状态机对应执行**REGISTER状态事件**，从状态**createdstate**向状态**establishedstate**转变，同时在转变之前自动触发**beforeRegisterEvent时机函数**完成注册。当状态进入**establishedstate**时，又接着触发了**“enter\_established”**所对应的**enterEstablishedState时机函数**去通知客户端注册已经正确完成。

在beforeRegisterEvent函数中，err = handler.chaincodeSupport.registerHandler(handler)完成了注册，使用的是前文所提到的在创建Handler时传入进来的ChaincodeSupport对象的registerHandler函数。所谓的注册，也不过是将Handler对象赋值给ChaincodeSupport对象中的runningChaincodes中的chaincodeMap映射：chainID作key，以Handler对象为成员handler值的chaincodeRTEnv对象作value。

### serialSend或serialSendAsync

都是使用Handler中grpc服务端流接口**ChatStream**成员发送ChaincodeMessage消息的函数，两者都将**应答ChaincodeMessage信息**发送给客户端，也都实现了将所发送的ChaincodeMessage信息**串行化**的目的。区别在于serialSend是**阻塞**发送，而serialSendAsync是利用新启goroutine进行**非阻塞**发送，且这些非阻塞的goroutine中任何一个发生发送消息的错误，都会利用errc频道将错误发送给processStream函数。

## 小结

不同类型的ChaincodeMessage的消息，能够触发状态机不同的状态事件，处理数据，完成Chaincode上的操作。有关其他类型事件以及具体的实现，在此不再赘述。强调一句，**ChaincodeSupport服务是以状态机驱动的为chaincode提供支持的一项服务**。

# fabric源码解析8——peer的System Chaincode

## 综述

关于System Chaincode，下文中以**系统链码**称之。这是个人翻译，依据是Chaincode本质是注册存储到链上的一段逻辑代码，因此个人习惯称Chaincode为**链码**，fabric文档中也变相的称为智能合约。但是，因为Chaincode是专用名词，个人觉得不翻译而直接使用是最好的。

start.go中的**serve**函数里，在为peerServer注册ChaincodeSupport服务的函数registerChaincodeSupport(peerServer.Server())中，实现了注册System Chaincode：scc.RegisterSysCCs()。

**系统链码**的核心代码在**/fabric/core/common/sysccprovider**和**/fabric/core/scc**下，scc也就是System Chaincode的缩写。系统链码分为五种：**cscc,escc,lscc,qscc,vscc**，均为各个系统链码的缩写。系统链码均实现了**/fabric/core/chaincode/shim/interfaces.go**中定义的**Chaincode接口**，从此点就可以看出，系统链码也属于Chaincode，只不过作用稍微特殊一点：

* cscc：configuration system chaincode
* lscc：lifecycle system chaincode
* escc：endorser system chaincode
* vscc：validator system chaincode
* qscc：querier system chaincode

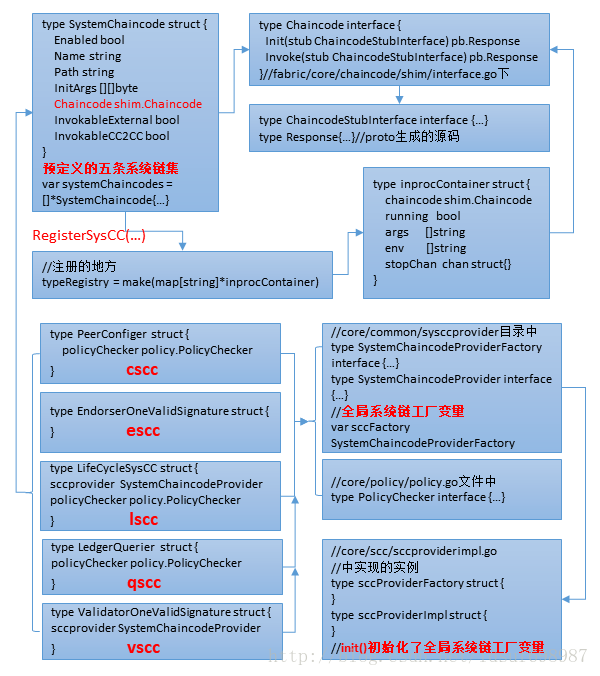
**sysccprovider**目录下的文件有：

* sysccprovider.go - 定义系统链码服务提供者接口

**scc**目录下的文件有：

* sysccapi.go - 系统链码的各种api操作
* importsysccs.go - 导入五种预定义的系统链码
* sccproviderimpl.go - 定义了系统链码服务提供者的具体实现和其操作

**结构图**：



## 预定义和注册

在**/fabric/core/scc/importsysccs.go**中：

//预定义的五个系统链码存放到数组中

var systemChaincodes = []\*SystemChaincode{

{

Enabled: true,

Name: "cscc",

Path: "github.com/hyperledger/fabric/core/scc/cscc",

InitArgs: [][]byte{[]byte("")},

Chaincode: &cscc.PeerConfiger{},

InvokableExternal: true, // cscc is invoked to join a channel

},{...}，{...}，{...}，{...}，

}

//注册五个系统链码

func RegisterSysCCs() {

for \_, sysCC := range systemChaincodes {

RegisterSysCC(sysCC)

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

RegisterSysCCs遍历systemChaincodes中所有的系统链码，并依次调用**RegisterSysCC**进行注册。RegisterSysCC在/fabric/core/scc/sysccapi.go中定义：

//系统链码开启且处于白名单中

if !syscc.Enabled || !isWhitelisted(syscc) {...}

//最终将系统链码注册到系统中

err := inproccontroller.Register(syscc.Path, syscc.Chaincode)

* 1
* 2
* 3
* 4

**inproccontroller.Register**定义在**/fabric/core/container/inproccontroller/inproccontroller.go**：

//存放安装的chaincode，以chaincode所在的path为key

typeRegistry = make(map[string]\*inprocContainer)

//注册到typeRegistry

func Register(path string, cc shim.Chaincode) error {

tmp := typeRegistry[path]

if tmp != nil {

return SysCCRegisteredErr(path)

}

typeRegistry[path] = &inprocContainer{chaincode: cc}

return nil

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

Register函数以系统链码**Path**成员值为key，包含系统链码的**inprocContainer**对象为value，将系统链码放入**typeRegistry**映射中。至此，系统链码注册完毕。

## 释义

以下文字翻译自[Fabric 文档](http://hyperledger-fabric.readthedocs.io/en/latest/)中关于系统链码（System Chaincode）的部分。

系统链码与一般chaincode一样，有相同的编程模型，比不过系统链码是运行在peer程序中，即其是peer程序的一部分，而一般的chaincode是单独运行在一个容器中的。因此，系统链码是内建在peer程序中且不遵循一般chaincode那样的生命周期。特别的，install，instantiate和upgrade操作也不应用于系统链码。

系统链码区别与一般的chaincode的目的是缩短grpc在peer结点与chaincode之间通信的时间消耗（因为peer结点在一个容器，chaincode是单独的一个容器），并权衡管理上的灵活性。比如，一个系统链码可以仅通过升级peer程序的二进制包来得到升级。系统链码可以用预定义的元素注册并编译到peer程序中，而且不需要有类似于背书策略或背书功能等这样的冗杂的功能。

系统链码被用在fabric中，去操纵整个系统的配置表现，这样的话可以随时把系统改变到合适的状态。

当前存在的系统链码名单：

* LSCC Lifecycle system chaincode，处理生命周期请求。我理解的生命周期请求应该指的是一个chaincode的安装，实例化，升级，卸载等对其生命周期起关键作用的一系列操作请求。
* CSCC Configuration system chaincode，处理在peer程序端的频道配置。
* QSCC Query system chaincode，提供账本查询接口，如获取块和交易信息。
* ESCC Endorsement system chaincode，通过对交易申请的应答信息进行签名，来提供背书功能。
* VSCC Validation system chaincode，处理交易校验，包括检查背书策略和版本在并发时的控制。

在修改或替换系统链码时必须注意，特别是LSCC，ESCC和VSCC，因为它们处于重要的交易环节中。以vscc为例，因为区域链中的交易数据都是持久性的，因此当vscc在提交一个block到账本中之前先验证该块，这不值什么，重要的是在同频道中的所有peer必须计算出相同的证书（由验证输出的证书）以避免账本产生冲突。因此特别需要注意的是vscc被修改或替换时，要避免和以前所产生的交易数据产生冲突。

# fabric源码解析9——文档翻译之MSP

只看代码而不读文档，无论是对自己的理解，还是对文章的释明，都容易使之存在困惑之处。因此在此先写fabric官方文档中几个相关主题的译文，作为自己相关主题文章的预热。水平有限，尽量直译，生硬之处很多，错误之处估计也不少而不自知。译文中括号内若是中文，则为解释，若为英文，则是专用名词。

个人觉得fabric的官方文档不好：

文档用语不简。作为说明性文档，个人认为语言应该尽量简明短小，以陈述句为主。而现在的官方文档中，充斥着大量长句，定状语句和指代名词。如果英语水平不高，比如我，读起来相当费劲且容易理解错。在此吐槽一下。

以下是Fabric文档中与MSP相关的主题译文：

## 安全和会员服务

Fabric加固了一个所有参与者都有一个明确身份的交易网络，并以此网络为基础。公匙基础设施（PKI，Public Key Infrastructure）被用来生成加密证书（cryptographic certificates），这些被试用于组织、网络的构成、终端用户或客户端的应用。结果，数据权限控制可以在广域网络（broader network）和层级频道（channel levels）被操作和支配。这个fabric的“授权”的概念，与channel的存在与能力相配对，帮助实现了隐私和机密为主要考虑的地址方案（address scenarios）

看Membership Service Providers (MSP)主题去更好的理解加密操作，签名，校验，认证（cryptographic implementations, and the sign, verify, authenticate）在fabric中的应用。

## Membership Service Providers (MSP)

该文档着力于提供关于MSP的安装，最好的实践的细节。

MSP是一个着眼于提供一个会员关系操作构架的抽象，作为系统的一个组成部分。

特别的，MSP抽象了所有加密机制和协议，隐藏了存在的争论（issuing）、验证证书和用户认证。一个MSP可能定义它自己的一套身份标识，验证规则，签名的生成和认证方法。

fabric区域链网络可能会被一个或多个MSP管理，他们提供了模块化的会员操作和在不同的会员关系标准和构架间协同工作的能力。

下面的文章我们详细的介绍MSP的操作体系，并讨论最好的MSP的相关用途的练习。

### MSP配置

为了启动一个MSP，它的配置文件必须在每个peer和orderer（去使能peer并排序签名）的本地中被指定，开启频道上的peer，orderer，客户端的身份认证，所有会员各自的签名认证。

首先，MSP需要指定一个名字在网络中代表自己，如msp1,org2等。这个名字在其会员关系规则中代表一个channel中的一个共同体、组织或者组织分部。也被引用作为MSP对象的ID（MSP Identifier）。每个MSP的ID是唯一的。例如，若检测出来存在两个相同ID的MSP实例存在于系统初创的channel，orderer将会启动失败。

MSP的默认操作的情况下，一个参数集合需要被指定（也就是一个MSP需要哪些成员），用于身份认证（identity (certificate) validation）和签名认证（signature verification）。这些参数由RFC5280指定，包括：

* 一个自己签名出的证书（self-signed）列表，用于作为信任机制的根证书列表（root of trust，这个根证书应该是：把自己签出来的证书分发给他人，他人拿着证书来核对时，在根证书列表中查询核对，如果存在证书存在在这个列表中，则说明这个人是受信任的）。该列表可以用root CAs表示。
* 一个X.509证书列表，用于代表（作为）中间人-CAs证书这个用于证书验证（certificate validation）的提供者（即所说的CA证书）。这些CA证书应该是在root of trust中被鉴定过的。中间人CA证书是可选的（参数）。该列表可以用intermediate CAs表示。
* 一个X.509证书列表，该列表中的证书包含可验证的证书路径，而这些路径对应指向root of trust中的证书。所以该证书列表代表着MSP的管理员的角色。这些证书的拥有者被允许发起请求改变该MSP的配置（即改变root CAs，中间人CAs）。该列表可以用administrator CAs表示。
* 一个组织单位列表，MSP有效的成员应该包含在这些组织单位的X.509证书中。这是一个可选配置参数，当多个组织使用同一个root CAs，intermediate CAs并为组织成员保留了一个OU字段时，才会使用这个列表。该列表可以OU List表示。
* 一个废除证书列表，每一个证书都可以对应到root CAs或intermediate CAs（废除的证书也是从这里面来的，而不是凭空产生的），这是一个可选参数。该列表可以用CRLs表示。
* 一个用于TLS的X.509根证书列表（self-signed (X.509) certificates，TLS root of trust）。
* 一个用于TLS的X.509中间人证书列表。这是一个可选参数。

对于一个MSP对象实例来说，有效的身份需要满足以下条件：   
\* X.509格式的证书，并包含可验证的证书所在路径，该路径对应到root CAs。   
\* 不包括在任何一个CRL中。   
\* 在X.509格式的证书结构体中的OU字段里，存在一个或多个属于OU List的组织单位。

更多的MSP身份的有效性的信息，请阅读MSP Identity Validity Rules。

除了验证相关参数，对于MSP来说，使结点能够去签名或授权，你需要指定：

* 用于结点签名的签名匙（signing key），当前只支持ECDSA keys。
* 结点的X.509证书，该证书必须是在MSP的一些列参数中（root CAs，中间人CAs中）的有效的证书。

需要重点注意的是，MSP中的身份信息永不消失，只能被加入到CRLs中。此外，当前不支持强制废除TLS的证书。

### 如何生成MSP证书和它们的签名匙

为了生成X.509证书去填MSP的配置，应用我们可以使用openssl。我们强调一下，fabric不支持证书中包含RSA keys。

替代的，我们可以使用cryptogen tool，在Getting Started中有详述。

Hyperledger Fabric CA这个项目也可以用来生成MSP配置所需的keys和证书。

### MSP setup on the peer & orderer side

为了建立本地MSP（local MSP），无论是peer还是orderer上的，管理者都要创建一个目录，如$MY\_PATH/mspconfig，包含六个子目录和一个文件：   
1. admincerts目录，包含pem文件，每个pem文件对应一个administrator证书，即administrator CAs。   
2. cacerts目录，包含每个的pem对应一个root证书，即root CAs。   
3. （可选）intermediatecerts目录，对应中间人CA。   
4. （可选）config.yaml文件，包含关于组织单位OU的信息。会定义如

### Channel MSP setup

在系统初创的时候，出现在网络中所有MSP的验证元素（即各种证书，配置）都需要被指定过（即必须已经存在），并被包含到系统channel的创世纪块（genesis block）中。回忆一下，MSP验证元素由MSP身份标识（MSP identifier），root CAs，intermediate CAs，admin CAs，OU List，CRLs。在orderer进行体系解析时，系统的创世纪块被用于orderer，据此orderer被允许去认证channel的创建请求（等于说orderer有认证的材料了，如果创建channel的请求合法才通过该请求）。如果创世纪块包含了两个相同身份标识的MSP，orderer将拒绝此块，自然的，整个网络的引导建立也会失败。

对于应用（application）的channel，其验证的各个组成部分，也就是（只能是）管理该channel的MSP，必须存在于channel的创世纪块中。我们强调，在channel创建一个或多个peer加入该channel之前，保证包含在该channel的创世纪块（或者是最新的配置块）中的MSP配置信息的正确性是应用的责任。

当在configtxgen工具（configtxgen tool）帮助下引导启动一个channel时，通过将MSP的验证元素（如root CAs，中间人CAs等）放置到MSP专用的配置目录，并在configtx.yaml文件中设置相应的选项，我们可以使用configtxgen工具配置该channel的MSP。

重新配置一个channel上的MSP，包括该MSP相关的废止证书列表（CRLs）的公告，通过该MSP中的Admin CAs中的一员（即拥有Admin CA证书的结点，也叫做admin结点）创建一个配置更新对象（config\_update object），可以实现。如此之后，被admin管理的客户端应用将宣布这个针对该MSP所在的channel的更新。

### Best Practices

在这个部分，我们将详述MSP配置在一般情况下的最好的实践（best practices）。

**1）组织单位和MSP之间的映射**

我们要求MSP与组织（organizations）之间是一对一的映射。如果选择一个不同的映射类型，需要考虑如下内容：

* 一个组织雇佣多种（个）MSP。这对应了一个组织包含多个不同部门，出于独立管理或隐私（权限）的原因，每个部门有一个MSP代表的情况。在这种情况中，一个peer只能被一个MSP所拥有（代表一个层级上的一个部门），也不能从同一组织中其他的MSP中识别出其他peer的身份。这样做的意义在于，一个结点可能通过gossip只与从属于该结点的部门（子部门）分享组织范围内的数据，而不是与组织内的所有成员分享。
* 多个组织使用一个MSP。这对应了多个组织以类似的关系被管理所组成一个的协作整体（consortium，如多个学校组织联合起来成立一个学校联盟）的情况。这里我们需要知道的是，一个组织中的多个peer将传递组织范围内的消息给同属与同一个MSP下的peer，而且会忽略这些peer是否是属于自己的同一个组织。这是一个MSP定义的颗粒度（granularity）和peer结点配置的局限。（最后这句话不太理解）

**2）一个组织有不同的部门（称为组织单位-organizational units），针对其中的一个想给予多个不同channel访问权限**

两种方法：

* 定义一个MSP，将所有组织的成员纳入到该MSP的成员关系（membership）中（即将所有成员的证书之类的数据纳入到该MSP）。该MSP的配置将由root CAs，intermediate CAs和admin CA组成；会员身份将包括所有的OU。策略（Policies）被定义为获取指定OU的所有成员，该策略可以用于组成一个channel的读写（read/write）策略，或者一个chaincode的背书（endorsement）策略。该方法的一个限制就是gossip peer会把那些在其本地MSP中的peer（即gossip结点的本地的MSP中所有成员）当作是同一个组织内的成员，结果也自然会向这些结点传播组织范围内的数据（比如他们的状态）。
* 为每一个部门定义一个MSP。这将涉及到每一个部门的指定，即每一个部门都需要指定root CAs， intermediate CAs和admin Certs，这样的话，在各个MSP之间没有重叠的证书路径。这就是说，比如，不同的intermediate CA对应不同的子部门。这里的缺点是要管理多个MSP而不是仅仅的一个，但这规避了上一种方法所出现的问题。我们也可以通过使用MSP的OU扩展为每一个部门定义一个MSP。

**3）从同一个组织中的众多peer中分离客户端**

在很多情况下，可以从身份自身获取身份的类型是需要的（如背书被担保是由peer这种类型的结点产生，而不是客户端或扮演orderer的结点）

以下是对该需求的有限的支持。

一种实现这种分离的方法是为每一类结点都创建一个单独的intermediate CA —— 一个客户端的，一个peer/orderer的；配置两个不同的MSP —— 一个客户端的，一个peer/orderer的。该组织应该（可以）访问的频道，需要同时包含这两个MSP，而背书策略（定义）则只使用代表peer结点的MSP。最终，将会把这个组织映射到两个MSP实例中，自然而然的peer与client可以相互作用（交流）了。

Gossip不会被大幅影响，因为同一个组织的所有peer依然属于同一个MSP。peer们结点可以限制某些系统chaincode的运行到本地的基于策略的MSP。例如，peer将只会执行“joinChannel”请求，如果该请求被客户端类型的MSP（终点用户位于请求的起源处）的admin签名的话。我们可以绕开这点的前后矛盾，如果我们接受，对于peer/orderer的MSP来说，客户端能成为该MSP成员的只能是该MSP的admin。

此种方法另一点需要考虑是，基于请求源头在它本地MSP中所处的地位，peer授权事件注册请求。很明显，因为请求源头是客户端，相较于被请求的peer，请求源头总是注定属于不同的MSP，并且被请求的peer将拒绝该请求。

**4）Admin和其证书**

MSP的admin证书是独一无二的，与该MSP中root CAs或者intermediate CAs中的证书都不同。这是一种通用的实践：将管理成员关系的责任从证书的验证讨论中脱离出来。

**5）intermediate CA黑名单**

正如前文提到的，重新配置一个MSP通过重新配置机制（对本地MSP实例手工重配，或者通过构建一个channel中MSP实例的一个配置升级消息）实现。很明显，这里有两种方法去确保一个intermediate CA被拉入黑名单：

1. 重新配置一个不再包含黑名单证书的MSP。对于本地配置，这意味着从intermediatecert CAs目录中移除黑名单证书。
2. 重新配置MSP的CRL，将黑名单证书从root CAs中加入到CRL中（也就是拉黑）。

当前的MSP操作，我们仅支持第一种方法，这种方法更简单，也不需要拉黑不信任的证书。

**6）CAs和TLS CAs**

MSP的root CAs与TLS的root CAs（相对于中间人CA来说）需要定义在不同的目录中。这样是为了避免在两种不同类型的证书之间产生混淆。关于这点来说，不禁止，但是最好在产品中避免。

# fabric源码解析10——文档翻译之Architecture

## 构架释义

**Fabric**的构架提供了以下优势：

* Chaincode信任机制（trust）灵活性。Fabric的构架将对于chaincode的信任假设（trust assumptions）从对于ordering的信任假设中分离了出来。换句话说，ordering服务（the ordering service）可能由一个结点集合（orderers）提供并容忍其中的一些结点失败或犯错（即容错性），背书者（endorser）对于每个chaincode来说可能不同。
* 可扩展性。由于负责特殊chaincode的背书者正交于（are orthogonal to）orderers，系统可以比所有功能都由同一个结点实现有更好的规模（指的是效率，交易吞吐量更高）。特别的，这样将导致，如果有一些chaincode指定了互斥的背书者，需要在chaincode与背书者之间引入分区（partitioning）以隔开两者，和允许并行执行chaincode（背书）。除此之外，某些潜在的非常消耗的chaincode的执行，被从ordering服务的关键路径删除了。
* 机密性。fabric构架帮助了那些有交易内容/状态等方面的机密性需求的chaincode的部署。
* 共识（Consensus）模块。该构架模块化并允许热插拔的共识（如，ordering服务）操作。

第一部分：**Fabric v1相关的架构的元素**

1.系统（System）构架

2.交易背书的基本工作流

3.背书策略

第二部分：**后续版本的构架元素**

4.账本检查点（Ledger checkpointing）（正在修改中）

### 系统构架

区域链（blockchain）是一个由多个结点相互通信所组成的分布式系统。区域链运行的被称作链码（chaincode）的程序，持有状态（state）和账本数据（ledger data），执行交易。chaincode是核心元素，因为交易是在chaincode上的一系列调用操作。交易必须被背书（be endorsed），只有背书过的交易才可能被提交并作用于状态。区域链中可能存在一个或多个特殊的chaincode用于管理功能和参数，被成为系统链码（system chaincodes）。

#### 交易

交易可以是两种类型：

* **Deploy交易**。该交易创建一个新的chaincode并把program作为参数。当一个部署交易运行成功，说明这个新的chaincode已经被“安装”到blockchain上了。
* **Invoke交易**。该交易涉及到一个chaincode和一个自身提供的function，当交易运行成功，该chaincode会执行指定的function-可能会修改相应的state并返回输出结果。

后文描述的，部署交易其实是特殊情况的Invoke交易。该交易创建了一个新的chaincode，对应的是在一条system chaincode上执行一次Invoke交易。

注意：文档在此假设无论是Deploy交易还是Invoke交易，前提都是已经部署好了chaincode。该文档未描述的部分有：a）查询交易的最佳方法（包含在v1版本中） b）对交叉链的支持（v1之后版本的特征）。

#### 区域链数据结构

**1.2.1.状态**

区域链最新的状态（或者简单点儿，状态）是按照版本化的“键/值”这样的键值对（key-value pair）格式存储（KVS），键是名称，值则是任意的点（blob）。这些条目（账目）由运行在区域链上的chaincode通过**获取**（get）或**存入**（put）这样的**键值存储操作**（KVS-operation）来维护。状态是永久性存储的，对状态的修改会被记录。注意状态的格式采用版本化的KVS（这里版本化的意思是说，键值对中的值是用来存储版本号的），一种实现可能就是使用现行的KVSs，也可能是关系型数据库管理系统（RDBMS）或者其他解决方案。

更正式的来说，状态s被模型化为**K->(V x N)**的映射元素，其中：

K 是一个**键**集合

V 是一个**值**集合

N 是一个无限版本号序列集合。该集合的内射函数（Injective function）“next：N->N”（内射函数，在go中可以把它想象成一个集合结构体的挂载函数func (\*s set)next(now n) n{…}，其中set是这个集合结构体，n是版本号类型）使用一个元素N去返回N的下一个版本号。

V和N都包含一个特殊的元素“\bot”，该元素是N的最小值（即最小的版本号）。初始时所有的键被映射成“（\bot,\bot）”。比如对于“s(k)=(v,ver)”，我们可以用“s(k).value”表示v，用“s(k).version”表示ver。（解释一下：即s这个状态k的映射是（v，ver），可以把（v,ver）想象成一个有两个成员的结构体。v应该是和k对应的，同一个事物，由k和v对应，k不变，v也不变，ver则是独立的，这点可以从下面的put函数看出来。可以暂时这么理解，一个状态映射中，kv组合算是标识着一个独立的对象，在该对象下进行着各种版本的变化）。

KVS操作如下一样被模式化：

* **put(k,v)**，这其中，k属于K这个集合，v属于V这个集合。该函数将当前区域链的状态s改变为s’。如此的话（可推算出等式），只要k’不等于k，则有s’(k)=(v,next(s(k).version))，s’(k’)=s(k’)。（即状态s’映射的k的值为（v和下一个版本号），状态s’的k’映射与s的k’映射的值相等，s’表示现在的状态，s表示过去的状态，同时也说明所谓世界状态，前后是连贯和记录的，你可以从s’找回过去s时的状态，也可以从过去s获取现在s’的状态）。
* **get(k)**，返回s(k)。即返回k所映射的状态。

State由peer结点操作维护，而不是客户端或orderer。

**状态**分区（这个分区是分割的意思，参考糗百的分割，一般在此区域是对同一主题的一些解释性说明或版本差异）。在KVS中的键值可以被认为是chaincode的名称，也就是说特定的chaincode所进行的交易只能修改属于他的键值。原则上，任何chaincode都可以读去属于其他chaincode的键值。对于交叉链交易（cross-chaincode transactions），修改状态的权限属于两个或更多的chaincode，这是v1后续版本的特性。

**1.2.2.账本**

账本（Ledger）提供了一个可验证的历史记录，该记录保存了系统运行期间所有成功或失败的**状态**的改变，这里的成功和失败也就是交易（transactions）的成功和失败。

账本作为一个全部按顺序排列的由交易产生的块的哈希链（hashchain），由ordering服务（参看1.3.3部分）构建。该哈希链在账本中把块（block）全部依序排列，每个块包含一个全部被依序排列的交易的数组。所有的交易之间，都必须遵循这种全部的依序排列。

所有的peer结点都保存着账本，一部分orderer结点也可以选择性的保存。我们在一个orderer结点的上下文（context）中查阅账本相当于查阅**orderer类账本**（**OrdererLedger**），在一个peer结点的上下文中查阅账本相当于查阅**peer类账本**（**PeerLedger**）。PeerLedger不同于OrdererLedger之处在于，peer结点维护一个位掩码（bitmask）并以此分辨有效的交易和无效的交易（更详细的说明请参考XX部分）。

peer结点可以像在XX（v1后续版本特性）部分所描述的那样修改PeerLedger。orderer为了容错性和PeerLedger的效率而维护OrderLedger，并可以在任何时间修改OrderLedger，提供的ordering服务的属性也被相应维护。

账本允许peer重新实现所有交易历史，重构状态。因此，在1.2.1部分描述的状态是一个可选择的数据结构。

### 结点（Nodes）

结点（Node）是区域链交流的实体。一个Node仅仅是一个**逻辑功能**（logical function），从这种意义上说多种不同类型的node可以在同一个实体服务器上。结点问题上需要在意的是，如何在信任领域（trust domains）组织这些结点，如何将这些结点与**逻辑实体**（logical entities）联合起来并由这些逻辑实体控制这些结点。

（这里的逻辑功能和逻辑实体的关系，如ordering服务，它是一项服务，也即一项逻辑上的功能，可以称之为**节点**，因为它属于整体功能中的一部分，整体服务过程中的一节，因此称为**节点**。而实实在在编译所成并运行在系统中的orderer程序，称为**结点**，它实现了或者说被设计成用于ordering服务，是承载ordering服务的实体，因此称之为**结点**。逻辑功能与逻辑实体可以一一对应，也可以根据实际需要一对多，多对多。而在faric中，都是一对一的关系，所以一般不用区分这两者）

这里有三种类型的结点：

1. 客户端（**Client**）或提交客户端（**submitting-client**）：一个客户端，该客户端提交（submits）真实的**交易召唤**（transaction-invocation）给背书者，广播**交易请求**（transaction-proposals）给ordering服务。（这里，背书前的交易称为invocation，是背书后的交易称为proposals，invocation是为了proposal而去“召唤”各个背书者进行背书）
2. **Peer**：一个结点，该结点提交（commits）交易，维护世界状态和一个账本的拷贝（参看1.2）。此外，peer可以当一个特殊的背书者的角色。（这里的提交与客户端的submitting要进行区别，submitting类似与提交初稿供大家审阅修改，而commit则类似于将审阅后的定稿提交，用于发行）。
3. ordering服务结点（**Ordering-service-node**）或者**orderer**：一个结点，该结点运行着信息传送服务，提供消息分发的担保，如原子性，全部排序广播。

这些类型的结点在下文有更详细的解释：

**1.3.1.客户端**

客户端代表着一个扮演了一半的终端用户（end-user）角色的实体。它必须连接到一个peer，才能在区域链上通信（communicate）。它可以根据它的选择连接到任何一个peer上。客户端创建并因而调用（invoke）交易。

如在章节2所详述的，客户端既与peer通信，也与ordering服务通信。

**1.3.2.peer**

一个peer从ordering服务接收排序后的**块**形式的**状态升级信息**，然后维护相应的世界状态和账本。

peer可以额外的扮演一个背书peer（endorsing peer）的角色，或者说是背书者（endorser）。背书者这个特殊的功能存在于特别的chaincode上（这里的特别就是说的该chaincode需要背书），并在提交交易之前对交易进行背书中发挥作用。每个chaincode可以指定一个背书策略（endorsement policy），该策略可能涉及到一个需要进行背书的peer的集合（即定义了需要哪些peer背书后，该交易才能提交，类似于现实中的一个文件必须有哪些人签过字之后才能生效或者往上层提交一样）。背书策略定义了一个有效的交易背书所具有的必要和满足的条件（典型的就是一个背书者的签名集合），如下文章节2和3描述的那样。在安装新的chaincode的部署交易（deploy transaction）这种特殊情形中，（部署）背书策略指定的像系统chaincode（system chaincode）的背书策略一样。

**1.3.3.orderer**

orderer结点形成了排序服务（the ordering service），即，一个提供分发保证的通信fabric。排序服务可以以不同的方式被实施：从一个中心化的服务（在开发和测试中使用）到面向不同的网络、结点错误模型的分布式协议。

ordering服务为客户端和peer提供了一个共享的通信频道（channel），为包含交易的消息（messages）提供广播服务。客户端连接到该频道后可以在频道上广播消息，之后消息会被分发给所有的peer。这个频道支持对所有的消息进行原子分发，即，消息通信是在完全排序分发（total-order delivery）和可靠性的条件下进行的。换句话说，这个频道把同一消息按统一的逻辑顺序输送到所有连接到该频道的peer。这个原子通信担保（atomic communication guarantee）又被称作完全排序广播（total-order broadcast），原子广播（atomic broadcast），或者分布式系统共识机制（consensus in the context of distributed systems）。对于区域链状态来说，被分发的消息就成为了候选交易（供各个peer进行背书，签名等操作）。

**分割（ordering服务频道）**。ordering服务可能支持多个频道，类似于一个**发行-订阅消息系统**（publish/subscribe messaging system）。客户端可以连接到给定的频道，之后可以发送消息和获取消息。频道可以被认为是分隔物（partitions），连接到一个频道的客户端不会意识到其他频道的存在，但是客户端可以连接多个频道。尽管一些ordering服务实现，包括fabric，支持多频道，但为了呈现的简洁，接下来的文章中，我们假定ordering服务由单独的一个服务于一个频道/话题组成。

**ordering服务API**。peer通过ordering服务提供的接口连接到由ordering服务所提供的频道。ordering服务接口由两个基本的操作（更通用的异步事件）组成：

（**TODO** 为获取客户端或peer指定的的序列号（范围）的对应的块增加接口）

* **broadcast(blob)**：客户端调用此函数，广播一条自定义的消息blob在频道中散播。在拜占庭容错系统的上下文中当发送一个请求到服务器时，这个操作也被称为request(blob)。
* **deliver(seqno, prevhash, blob)**：ordering服务在peer上调用此函数，分发一个消息blob，对应为该消息的指定了一个非负整数序列号seqno，和最近的上一个已经被分发的blob的哈希值prevhash。换句话说，这是一个来自ordering服务的输出事件。有时deliver()在发布-订阅系统中也被称为notify()，在拜占庭容错系统（BFT systems）中也被称为commit()。（这里说的在peer上调用，意思不是说ordering服务在peer上，而是上文所说的，peer通过ordering服务的接口连接ordering服务提供的频道后进行通信，因此这里说在peer上调用此函数，意思是说peer调用此ordering服务函数）。

**账本和块的结构**。账本（看1.2.2部分）包含ordering服务输出的所有数据。从表面上看，它是一个deliver(seqno, prevhash, blob)事件的队列，根据计算prevhash，如上文所述的那样，该队列形成了一个哈希链（hash chain）。

大部分时间里，出于效率原因，ordering服务会将多个blob消息组成一组，然后在一个单独的deliver事件中输出块，以此替代单个blob一个交易一个交易地输出的方式。这种情况下，ordering服务必须在每一个块中强制并输送确定顺序的blob信息。blob在块中的顺序的数字可以通过ordering服务的操作被动态的确定。

下文中，为了呈现的简单，我们定义ordering服务的功能并在一个deliver事件对应一个blob消息的基础上解释交易背书（章节2）的工作流。通过上文提到的在一个块内的对blob消息确定性的排序，这些很容易被扩展到块上面，假设一个针对块的deliver事件，该块对应的是一个由多个单独的针对块中一个个blob消息的deliver事件组成的队列。

**ordering服务属性**

ordering服务担保规定了如何广播一条信息和分发信息之间的关系。这些担保描述如下：

1. **安全（一致性保证）**：只要每个peer连接到共享频道足够长的时间（他们可以连接断开或被销毁，但是会重启并重新连接），这些peer将看到完全一样的一系列被分发的(seqno, prevhash, blob)消息。这意味着输出（即deliver()事件）在所有peer和所依据的次序号码上都是以同样的顺序产生的，对于同一个次序号码的消息则必持有同样的内容（指blob和prevhash）。注意这个顺序只是逻辑上的顺序，一个在某一peer上调用的deliver(seqno, prevhash, blob)不要求与在另一个peer上输出同样消息的所调用的deliver(seqno, prevhash, blob)有任何实时关系（即同样一份信息传给所有peer，但是不要求同时传达）。换句话说，给定一个明确的seqno，不存在两个正确的peer（这里的正确是指频道中peer正常连接且身份合法）被分发了不同的prevhash或blob值。此外，只有一些客户端（或peer）真正调用了broadcast(blob)，最好是每个blob只广播发送一次，值blob才会被分发。更深的讲，deliver()事件包含了上一次deliver()事件所生数据（prevhash）的加密哈希值（cryptographic hash）。当ordering服务执行原子广播，prevhash是对一个从次序号码是seqno-1的deliver()事件而来的元素加密后的hash值。这由从头到尾的deliver()事件建立了一条hash链，这条链用于帮助检查ordering服务输出的完整性，像稍后的章节4和5讨论的那样。对于第一个deliver()事件这样的特殊情况，prvhash有一个默认值。
2. **活性（Liveness）（分发保证）**：ordering服务的活性保证由ordering服务实施方案来指定。正确的担保可能依靠网络和错误结点模型。首要的就是，如果正在提交交易的客户端（the submitting client）没有失败，ordering服务应该保证每个连接到服务的正确的peer最终分发每一个由客户端提交的交易（这里的peer分发，指的是peer也可以调用broadcast()事件，通过ordering服务向频道中的其他peer进行广播消息）。

总结一下，ordering服务保证如下特性：

* **一致性**（Agreement）。对于在正确peer中的任意两个事件**deliver(seqno, prevhash0, blob0)**和**deliver(seqno, prevhash1, blob1)**，**seqno**相同，则**prevhash0==prevhash1**且**blob0==blob1**。
* **哈希链的完整性**（Hashchain integrity）。对于在正确peer中的任意两个事件**deliver(seqno-1, prevhash0, blob0)**和**deliver(seqno, prevhash, blob)**，**prevhash = HASH(seqno-1||prevhash0||blob0)**。
* **禁止跳跃**（No skipping）。如果ordering服务在一个正确的peer结点p上输出**deliver(seqno, prevhash, blob)**，seqno>0，则p肯定在此之前已经被投递了一个事件**deliver(seqno-1, prevhash0, blob0)**。
* **禁止创建**（No creation）。任一在一个正确peer上的事件**deliver(seqno, prevhash, blob)**必须以某些（可能很少）peer触发**broadcast(blob)**事件为先导。
* **禁止重复**（No duplication）（可选，但是最好有）。当两个事件**deliver(seqno0, prevhash0, blob)**和**deliver(seqno1, prevhash1, blob’)**发生在正确的peer中且**blob == blob’**，则**seqno0==seqno1**和**prevhash0==prevhash1**，则会产生两个（重复的）事件**broadcast(blob)**和**broadcast(blob’)**。
* **活性**（Liveness）。如果一个正确的客户端调用了broadcast(blob)事件，然后每个正确的peer都会**最终**（eventually）放出一个事件deliver(\*, \*, blob)，其中\*代表任意值。

### 交易背书的基础工作流

接下来我们要为一个交易描绘出高层次的请求工作流。

注意：留意这条协议：不要假定所有的交易都是确定性的，即，允许不确定的交易。

#### 客户端创建一个交易并发送该交易到交易所选择的背书结点

为了请求一个交易（invocation），客户端发送一个**PROPOSE**消息到消息所选的背书peer点集合（中的每一位）（可能不是同时-参看章节2.1.2和2.3）。客户端通过peer使用一个给定chaincodID的背书peer点集合，即依次从背书策略（参看章节3）知道整个背书集合（意思是说，peer可以接触背书策略，而背书策略规定了背书者集合，所以说客户端通过peer把PROPOSE发送到该发送的peer中）。例如，交易可能会被发送到全部的给定chaincodeID中的背书者。同时也就是说，一些背书者可能掉线或者反对并拒绝为该交易背书。正在提交交易的客户端则是尝试着用有效的背书者去满足背书策略的表达。

下面我们先详述PROPOSE消息的格式，然后讨论在客户端和背书者之间相互作用的可能的例证。

**2.1.1.PROPOSE消息格式**

PROPOSE消息的格式为**\**

#### 背书peer模拟执行一个交易并进行背书签名

从客户端接收一个\

#### 客户端（The submitting client）收集交易的背书并通过ordering服务广播该背书

客户端（在发出交易后）会一直等待，直到它收集到“足够（enough）”的消息和在**(TRANSACTION-ENDORSED, tid, *,*)**语句上的签名，以得出该交易被背书的结论。如章节2.1.2中讨论的那样，这可能与各个背书者之间触发多次交互的“往返旅行”。

关于“足够”的确切定义，由chaincode的背书策略所决定（参看章节3）。如果满足了背书策略，则就说该交易已经被背书，注意此时该交易还没有提交（committed）。从那些构建了交易是被背书的背书者中而来的签名消息**TRANSACTION-ENDORSED**的集合（简单说就是，来自于那些认可交易的背书者的签名所形成的集合），被称为一个背书（endorsement，这里是名词），并用背书表示。

如果客户端没有成功收集到交易申请的背书（即收到的背书者的签名不满足背书策略），它会用一个选项（option）抛弃这个交易并稍后再试。

对于拥有有效背书的交易，我们现在开始使用ordering服务。客户端使用**broadcast(blob)**触发ordering服务，这里**blob=endorsement**。如果客户端没有直接触发ordering服务的能力，它可以通过选择某个peer去代理它的broadcast。该peer必须是被客户端信任的，信任这个peer不会从背书中删除任何信息或者做其他可能使交易被视为无效的事情。注意，然而，一个代理peer可能不会捏造一个有效背书（同时也就是说可能捏造）。

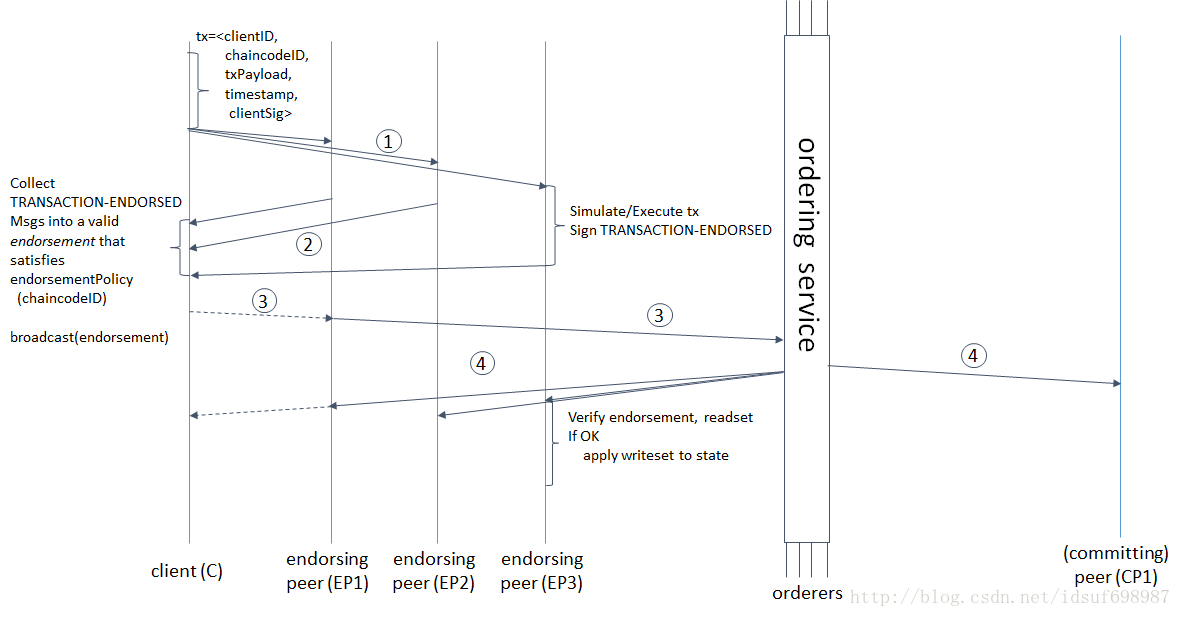
#### ordering服务分发一个交易到各个peer

当一个事件**deliver(seqno, prevhash, blob)**发生时，一个peer必定已经用一个比seqno小的次序数字为blobs（之前的blob或者说其他的blob）申请了所有状态更新，一个peer做了如下的事：

* 根据chaincode（blob.tran-proposal.chaincodeID指定）的策略检查**blob.endorsement**是否有效。
* 典型的情形是，与此同时也确认**依赖**（blob.endorsement.tran-proposal.readset，**读集合**）（dependencies，参看章节2.2，也就是版本依赖）仍没有被违背。更复杂的使用情况，背书中的tran-proposal字段可能不一样，在这种情况下，背书策略（参看章节3）规定了状态如何演化。根据为状态更新选择的一致性（或着说“隔离度保证”，isolation guarantee）属性，版本依赖的校验可以用不同的方法实现。可串行性（Serializability）是一个默认的隔离度保证，除非chaincode背书策略指定了一个不同的。要求在readset中与每个key相关联的版本值等于在状态中这些key对应的版本值，并且拒绝那些不满足该要求的交易，这样则能提供所谓的可串行性。
* 如果这些检查全部通过，这个交易则被视为有效的或可提交的（committed）。这种情形下，peer用PeerLedger的位掩码（bitmask）中的1标记这个交易，应用**blob.endorsement.tran-proposal.writeset（写集合）**到**区域链状态**（blockchain state）（如果交易申请是相同的，背书策略逻辑的其他方面定义了相应的使用blob.endorsement的功能）。
* 如果对blob.endorsement的背书策略校验失败，这个交易则是无效的且peer使用PeerLedger的位掩码中的0标记这个交易。重点留意无效的交易是不改变区域链状态的。

注意，用一个给定的次序号处理一个deliver事件（块）后，所有（正确）的peer会拥有相同的状态，（对于**交易背书的基础工作流**来说）这样就足够了。这就是说，在ordering服务的保证下，所有正确的peer将接收一个deliver(seqno, prevhash, blob)事件的相同次序（也即一个相同次序的deliver事件集）。因为背书策略的测定和readset中的版本依赖的测定是确定的，所以无论包含在一个blob中的交易是否有效，所有正确的peer也将得出相同的结论。自此，所有的peer提交并应用同样顺序的交易，并用同样的方法升级他们的状态。

**交易流程图（通用情况下的方式）：**



*流程图的①②③④与2.1，2.2，2.3，2.4对应。*

### 背书策略

#### 背书策略详细说明

一个背书策略（endorsement policy），是一个如何背书一个交易的条件。区域链的peer拥有一个预定义的背书策略集合，该背书策略与安装指定chaincode的部署交易（deploy transaction）相关。背书策略可以被参数化，这些参数可以通过一个部署交易来指定。

为了保证区域链和安全（security）特性，背书策略集合应该是一个经过验证的由有限的功能（函数）集合组成的策略集合，目的在于确保边界的执行时间（bounded execution time，即可终止），确定性（determinism），执行（performance）和安全担保（security guarantees）。

背书策略的动态添加（如，通过部署交易在chaincode之上部署时间条件）在有限的策略评定时间，决策，执行和安全担保这些项上是非常敏感的。因此，不允许背书策略的动态添加，但是在将来能被支持。

#### 交易评定（Transaction evaluation）对照背书策略

只有当一个交易已经依据指定的策略被背书，该交易才会被声明为有效的。对于某chaincode的一个请求交易必须先包含一个满足该chaincode策略的背书，否则这个交易不会被提交（committed）。这个发生在客户端和背书peer之间的交互过程中，如章节2解释的那样。

正常的，背书策略是在背书上的声明，潜在的更深的意思就是评定为TRUE或FALSE的状态。对于部署交易，背书根据一个系统范围的策略被包含（例如，从系统的chaincode）。

一个背书策略声明要引用确定的变量。潜在的它可能引用：

1. 与这个chaincode相关的key或者身份（identities）（key或身份存在于chaincode中的原始数据），如，一个背书者的集合。
2. 更丰富的chaincode原始数据（metadata，一般指配置文件中配置的值或功能源代码之类的）。
3. endorsement和endorsement.tran-proposal的元素。
4. 更多潜在的。

以上列表按表现力和复杂性的递增排列，也就是说，只引用一个结点的key或身份对于所能支持的策略来说，是相对简单容易些的。

**背书策略声明的评定必须是确定性的**。一个背书应该由每一个peer在本地评定，如此一个peer就不需要再与其他peer交互了，而且所有正确的peer都使用同样的方法评定背书策略。

#### 背书策略示例

声明可能包含逻辑表达式和TRUE或FALSE评定。典型的，条件将使用在由chaincode对应的背书peer讨论的交易请求上的**数字签名**。

设定chaincode指定的背书者集合**E = {Alice, Bob, Charlie, Dave, Eve, Frank, George}**。则一些示例策略如下：

* 一个来自所有E成员针对同一个交易请求（tran-proposal）的有效签名。
* 一个来自单独一个E成员的签名。
* 来自根据条件**(Alice OR Bob) AND (any two of:Charlie, Dave, Eve, Frank, George)**指定的背书peer对同一个交易请求的有效签名。（这里的条件表达式说明：必须有Alic或Bob签名，且同时有四个人中任意两个人的签名）。
* 7个背书者中的任意5个对同一交易请求的有效签名。（一般情况下，chaincode中有**n**个背书者，**n>3f**，则只要n个背书者中的任意**2f+1**个，或者任何一组多于**(n+f)/2**的背书者同意，则为有效签名）。
* 设定这里把一种赌注或权重的分配给每个背书者，像**{Alice=49, Bob=15, Charlie=15, Dave=10, Eve=7, Frank=3, George=1}**，这里筹码总数是100：策略要求一个拥有占大头儿的赌注集合（即，所有人的赌注加起来严格的大于50）的有效签名。例如**{Alice, X}**，其中X是除George外的任何人，或者**{everyone together except Alice}**。等等。
* 上一个分配赌注的例子的条件可以是静态的（在chaincode中的原始数据中修改），也可以是动态的（即，依赖于chaincode的状态并在运行期间修改）。
* **tran-proposal1**上**(Alice OR Bob)**的有效签名和**tran-proposal2**上**(any two of: Charlie, Dave, Eve, Frank, George)**的有效签名，这里tran-proposal1和tran-proposal2仅在指定的背书者和状态升级这两点上不一样。

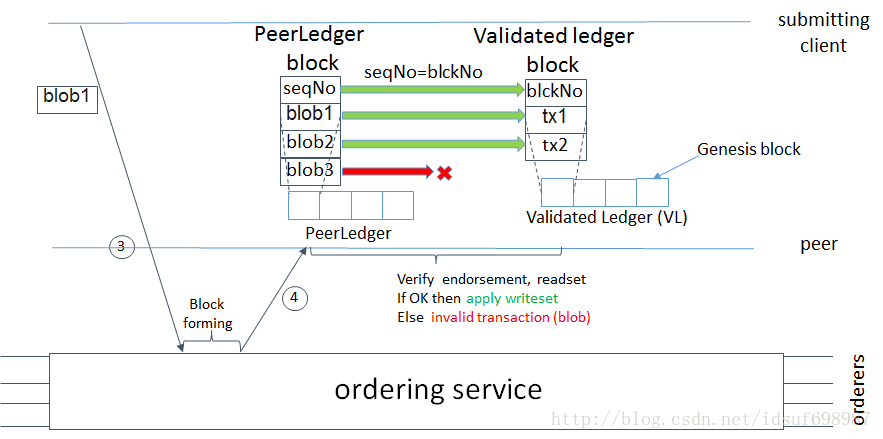
以上这些策略的有用程度取决于具体的应用、所期望的对抗坏结点解决方案的弹性、和其他多种多样的属性。（这里的坏结点指一个因连接等原因导致连接失败或者故意在背书交易的事情上搞破坏的结点）。

### (v1后续版本). 验证过的账本和PeerLedger验证节点(正在修改)

#### 验证过的账本(VLedger)

为了维护一个只包含有效和可提交（committed）交易（例如，这样的特征出现在比特币中）的账本的抽象层，各个peer可能除状态（state）和账本（Ledger）外，还维护一个**有效账本**（**VLedger**）。这是一个由过滤掉账本中的无效交易而来的哈希链。也就是说，Ledger是用于记录所有的交易，无论该交易是否有效，只要发生就记录，而VLedger只用于记录发生的有效的交易。

VLedger块（这里称为vblock）的构建过程如下。因为一个PeerLedger的block中可能包含无效交易（即，一个交易拥有无效的背书或无效的版本依赖），这样的交易会在从block变为vblock之前被peer过滤掉。每个peer通过自身做这些工作（即，通过使用与PeerLedger相关的位掩码）。一个vblock被定义为没有无效交易的block，这些无效交易已经被删除。这样的话vblock在大小上天性就是动态的，也可能为空。vblock的构件图如下：



通过每一个peer，一块块vblock被连接在一起，从而形成一个哈希链。更详细点儿，这个验证过的账本中的每一块vblock都包含：

* 前一块vblock的哈希值
* vblock的号码
* 一个所有由peer提交的有效交易按序排列的列表（即，一个相应块中的有效交易列表）。
* 得出这个vblock所对应的block的哈希值

所有这些信息由一个peer连在一起并计算哈希值，产出在VLedger中的vblock的哈希值。

#### PeerLedger验证节点

账本包含一些可能永远不需要被记录的无效的交易。然而，peer不能简单的丢弃PeerLedger的block，也因此，一旦peer要建立对应的vblock，它们是去修剪Peerledger。也就是说，在这种情况下，如果一个新的peer加入了网络，其他peer不能传送被丢弃的block给这个新的peer，也不能让这个新的peer信服这些peer的vblock的有效性。

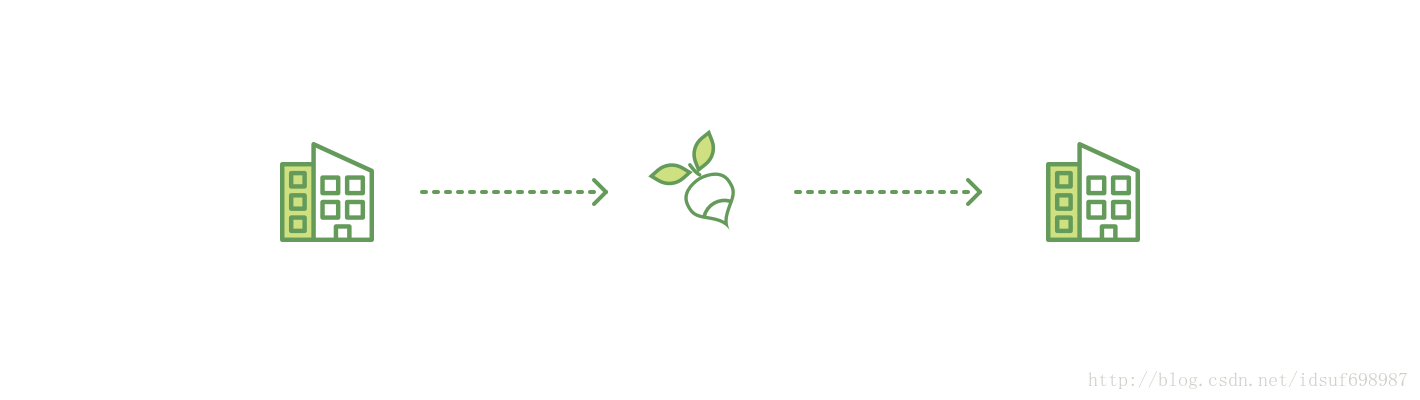
为了帮助修剪PeerLedger，该文档描述了一个**验证节点机制**（checkpointing mechanism）。这个机制建立了peer网络之间的vblock的校验，并允许检查过的vblock替换被丢弃的block。这样，依次的，减少了内存空间占用，因为不用在去存储无效的交易了。同时也减少了为一个新加入网络的peer重构状态的工作量（因为新加入的peer在通过重演PeerLedger重构状态时不需要去验证个别的交易，而只需重演包含在VLedger中的状态。这里的重演指的是，在为新peer建立state时，需要根据PeerLedger的状态，也就是需要从头到尾过一遍PeerLedger，所以原文用了replay）。

**4.2.1.验证节点协议**

验证结点周期性的由peer的每个**CHK块**执行，这里CHK是一个可配置的参数。为了初始化一个验证节点，peer广播（即，gossip）一条消息**\**

## 交易流

这部分文档描述了发生在一个标准的财产交换期间的交易机制（transactional mechanics）。“剧本”包含两个客户端，A和B，A买萝卜，B卖萝卜。两者在网络上都有一个peer，通过网络A和B发送交易并与账本交互。



**假设**

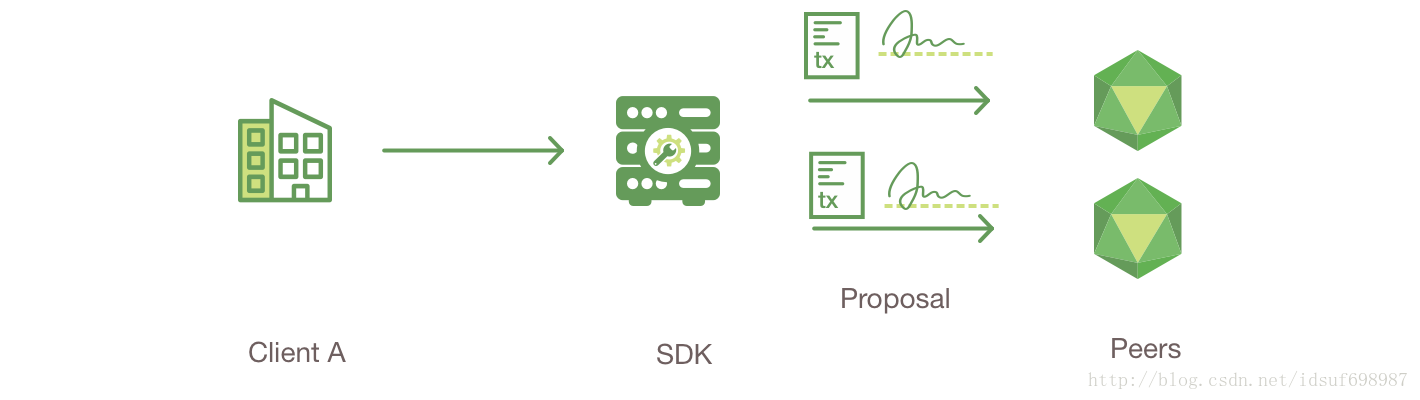
这个流程假设一个channel已经建立并运行。应用的使用者已经注册，已把组织的信任授权（CA）编入清单中并收到必要的用于验证的加密原材料。

一个chaincode（包含一个表示萝卜市场的初始化状态的键值对集合）被安装到两个peer上并在channel上实例化。该chaincode包含的逻辑定义了一个交易指令集合和双方都同意的一个萝卜的价格。一个背书策略也被设置到这个chaincode中，该策略需要满足peerA和peerB同时背书两者之间的任何交易。

**1.客户端A初始化一个交易**

会发生什么呢？客户端A发送一个购买萝卜的请求。该**请求**（request）把peerA和peerB当作目标，各自对应客户端A和客户端B。背书策略规定了任何交易必须得到两个peer都要签名，因此该请求发送到peerA和peerB。

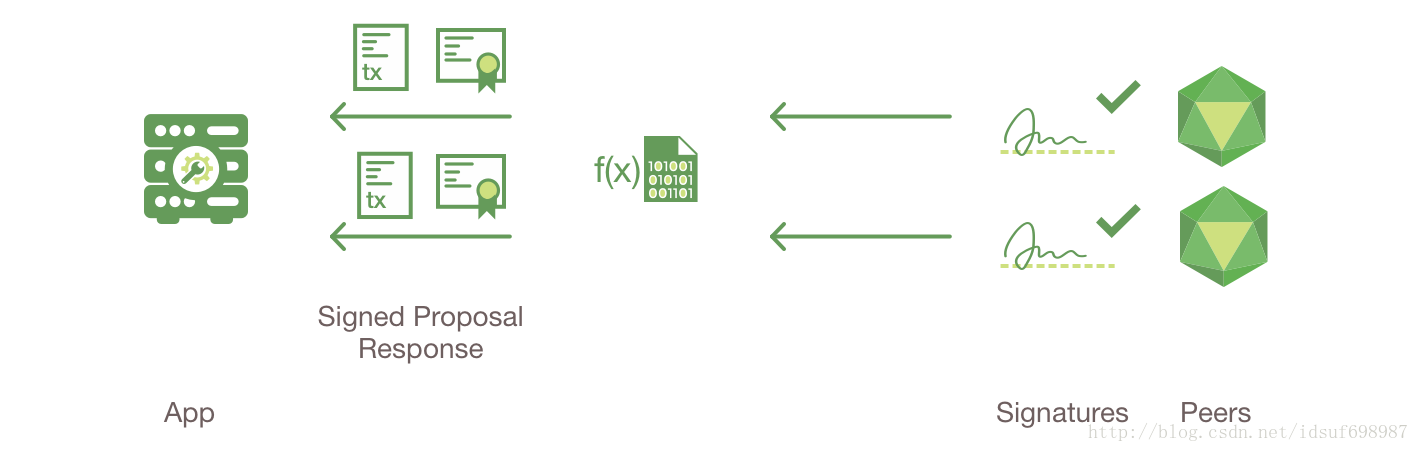
接着，一个**交易申请**（transaction proposal）被构建。一个撬动（leveraging）了所支持的SDK（Node, Java, Python）的应用程序（application）利用有效的API生成一个交易申请。该申请是一个触发一个chaincode功能（函数）的请求，如此就可以从账本中读取数据或把数据写到账本中（即，写一个针对财产的新的键值对）。**SDK**作为一个垫片（**shim**，相当于一个中间支撑件）把这个交易申请打包成合适的所设计的形式（在grpc上的protocol buffer格式），并使用用户的加密证书（cryptographic credentials）为该交易申请生成一个唯一的**签名**（signature）。



**2.背书peer确认签名&执行交易**

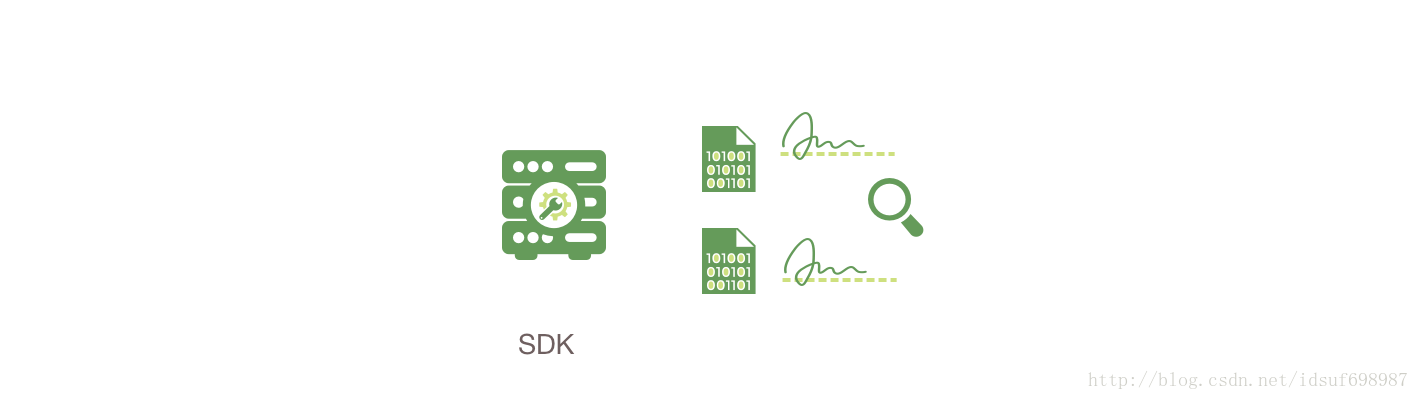
背书peer确认的有：（1）交易申请被完好的格式化，（2）之前没有被客户端提交（submitted）过（这是为了replay-attack防御），（3）签名是有效的（使用MSP），（4）提交者（在这个例子中是客户端A）在这个channel上被适当的授权，从而可以执行所计划的操作（即，每个背书peer确保提交者（submitter）满足这个channel的写者策略（Writers policy））。背书peer把这个交易申请的**输入**（inputs）作为被调用的（invoked）chaincode函数的参数。接着chaincode被执行，依靠当前**状态数据库**（state database）产生**交易结果**（transaction results），该交易结果包含一个应答值（response value），读集合（read set），写集合（write set）。至此，还没有对账本进行更新。这些值，与背书peer的签名和一个**YES/NO**背书语句一起形成一个集合，作为一个**申请应答**（proposal response）被传送回SDK，该SDK解析这些数据（payload），供应用程序消费（consume）。

{MSP是一个peer的组成部分，该部分确认从客户端到来的交易请求并给交易结果签名（背书）。写者策略在channel创建的时候被定义，决定了哪些peer被提名可以提交一个交易给channel}



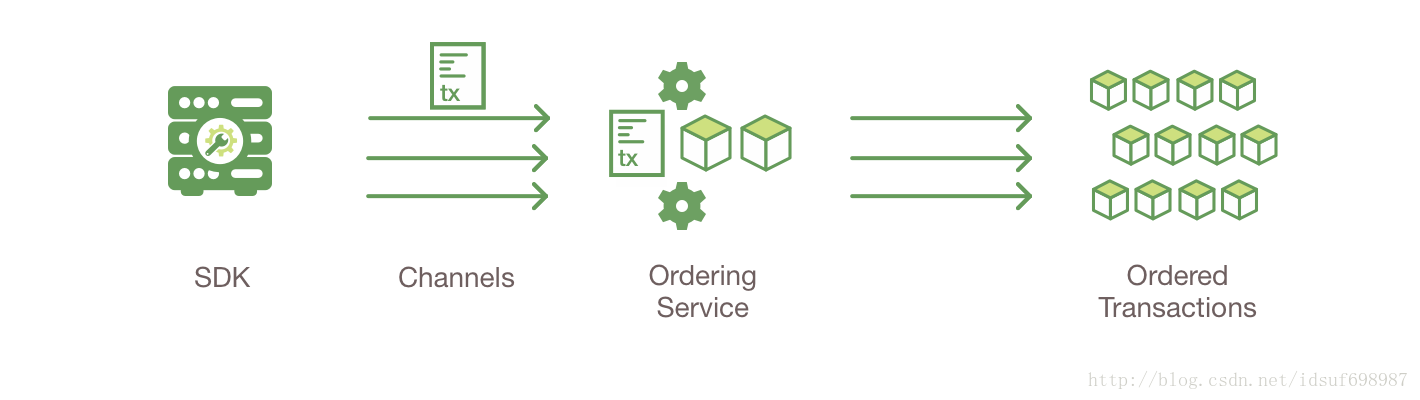
**3.检查申请应答**

应用程序确认背书peer的签名并对比申请应答（链接到包含申请应答的代表的词汇表，每个申请应答都有一个名字的）去决定申请应答是否一样（这里的一样应该指的是，是否是交易申请对应的那个应答），规定的背书策略是否完全被填写（即，peerA和peerB是否都进行了背书）。这里的设计是，即便一个应用程序选择不去检查应答或者在其他方面转发一个未背书的交易，这个检查也会在提交验证阶段（commit validation phase，即下文的第5步）被peer强制执行。



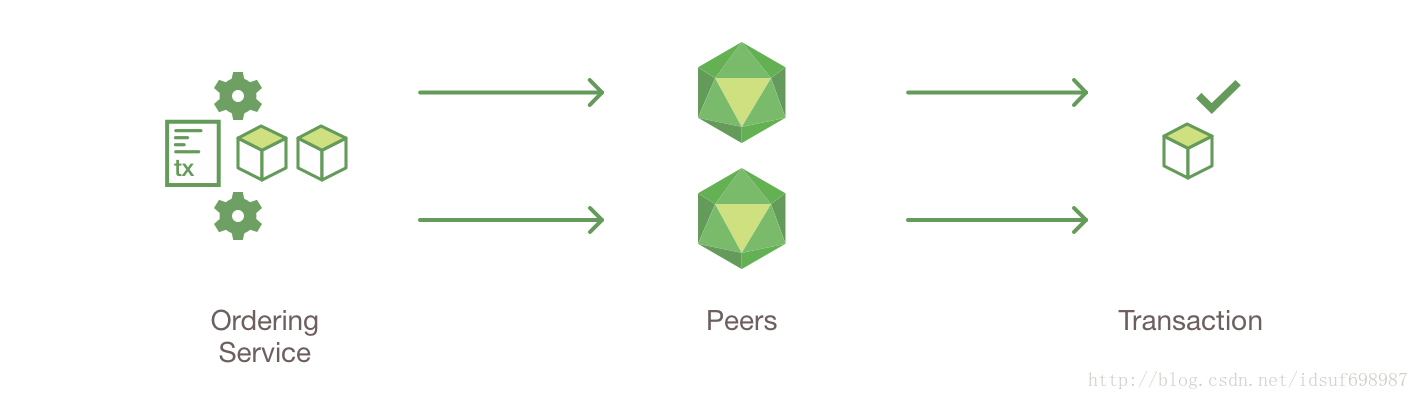
**4.客户端将背书组装进一个交易**

应用程序将交易请求和应答组装进一个**交易消息**（transaction message），并将此交易消息**广播**（broadcast）到ordering服务（Ordering Service）。这个广播交易包含了读写集合，所有背书peer的签名和频道ID（Channel ID）。ordering服务不需要去检查交易的整个内容用以执行内容的操作，而是简单的从网络中所有的channel接收交易，分频道按时间的前后顺序排序这些交易，创建每个channel的交易块（即包含这些交易的块）。



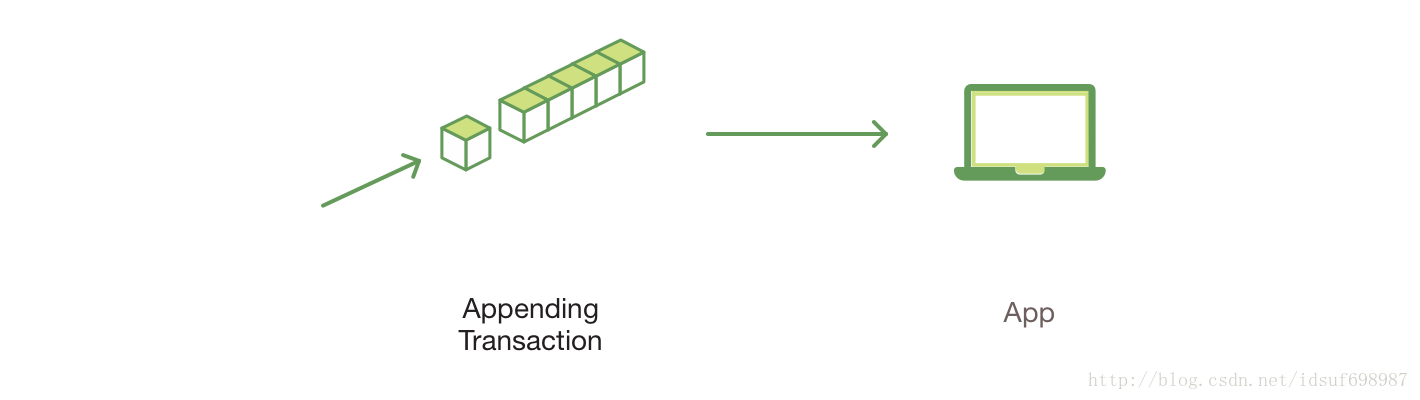
**5.交易被校验和提交**

这些包含交易的块（block）被分发到channel中所有的peer。每个peer都会验证块中的交易，以确保背书策略被完全填写，确保从交易执行生成读集合（第2步）起到现在，账本状态对于读集合来说没有任何变化（即根据读集合去验证账本状态是否发生变化）。在块中的交易被标记为有效（valid）或者无效（invalid）。



**6.账本更新**

每个peer把块（block）添加到频道的链（channel’s chain）上，对于每一个有效的交易，其中的写集合被提交到当前的状态数据库（state database，也就是账本）。然后，无论交易是否成功，一个事件被触发，把这个交易（invocation）已经被不可改变的添加到了链上的事实，和该交易是有效还是无效的“通知书”通知给客户端应用。



注意：对照泳道图（即上面每个步骤中所画的UML活动图）可以更好的理解服务端的流和protobuffers。

## Hyperledger Fabric SDKs

…

Bringing up a Kafka-based Ordering Service

Caveat emptor

…

Big picture

…

Steps

…

Additional considerations

…

Supported Kafka versions and upgrading

…

Debugging

…

Example

…

频道

…

账本

…

链

…

状态数据库

…

交易流

…

状态数据库选项

# fabric源码解析11——peer的Admin和Endorser服务

继续start.go的**serve**函数，**peerServer**对象在ChaincodeSupport服务之后，又注册了Admin，Endorser服务：

pb.RegisterAdminServer(peerServer.Server(), core.NewAdminServer())

serverEndorser := endorser.NewEndorserServer()

pb.RegisterEndorserServer(peerServer.Server(), serverEndorser)

* 1
* 2
* 3

其注册方式与注册ChaincodeSupport服务一致，在fabirc源码解析7中已经详述，所以本文只将目光锁定在Admin和Endorser服务本身。

## Admin

**Admin服务**实现对**服务器**、**模块日志级别**的获取和控制。服务原型定义在**/fabric/protos/peer/admin.proto**以及对应生成的**admin.pb.go**文件。核心代码在**/fabric/core/admin.go**中，该文件中实现了admin.pb.go中的Admin服务端服务。

//proto定义的服务原型和定义的服务状态

//在/fabric/protos/peer/admin.proto中定义

service Admin {

rpc GetStatus(google.protobuf.Empty) returns (ServerStatus) {}

rpc StartServer(google.protobuf.Empty) returns (ServerStatus) {}

rpc StopServer(google.protobuf.Empty) returns (ServerStatus) {}

...

}

//proto使用枚举定义的服务状态

message ServerStatus {

enum StatusCode {

UNDEFINED = 0;

STARTED = 1;

STOPPED = 2;

PAUSED = 3;

ERROR = 4;

UNKNOWN = 5;

}//状态

StatusCode status = 1;

}

//proto生成的服务端接口和注册函数

//在/fabric/protos/peer/admin.pb.go中定义

type AdminServer interface {

GetStatus(context.Context, \*google\_protobuf.Empty) (\*ServerStatus, error)

StartServer(context.Context, \*google\_protobuf.Empty) (\*ServerStatus, error)

...

}

func RegisterAdminServer(s \*grpc.Server, srv AdminServer) {

s.RegisterService(&\_Admin\_serviceDesc, srv)

}

//核心代码实现Admin服务

//在/fabric/core/admin.go中定义

type ServerAdmin struct {

}

//服务器的开启、停止、获取状态函数

func (\*ServerAdmin) StartServer(context.Context, \*empty.Empty) (\*pb.ServerStatus, error) {

status := &pb.ServerStatus{Status: pb.ServerStatus\_STARTED}

log.Debugf("returning status: %s", status

return status, nil

}

func (\*ServerAdmin) StartServer(...)...{...}

func (\*ServerAdmin) GetStatus(...)...{...}

func (\*ServerAdmin) StartServer(...)...{...}

//模块日志的获取、设置、恢复函数

func (\*ServerAdmin) GetModuleLogLevel(...)...{...}

func (\*ServerAdmin) SetModuleLogLevel(...)...{...}

func (\*ServerAdmin) RevertLogLevels(...)...{...}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48

在此对Admin的StartServer和StopServer操作有一些疑问，字面上看来其是启动和停止服务，但是实际操作中，如StartServer函数，其只是返回了一个状态为STARTED的状态对象，并无其他操作。既然是返回状态，也有可能是其他处需要此状态，但是通过grep搜索StartServer也并未发现有按期望的调用。在此歇笔，若后文其他模块发现这两个操作用法的意图，则会补明。【？？？】

Admin服务关于模块日志的函数中，涉及到了flogging，在fabirc源码解析3中已经详述，可参看之后再追溯Admin的这三个函数。若对fabric的日志系统有所了解，则此处的三个函数的作用很好理解。以SetModuleLogLevel为例，作用就是从客户端获取requst中包含的模块关键字和要设置的日志级别，然后在flogging中进行设置。

## Endorser

**Endorser**，个人习惯直译成**背书者**。类似于对支票的背书，表示对一种行为或权利的认可。放在fabric中，就是一个主体接收到其他某一peer点发送的申请消息，通过检查该申请消息中的签名的方式，向发送者表示支持和认可，这样该请求或申请才有可能被最终提交，使之作用于系统。这里说有可能，是指一个申请可能还需要满足一些条件，比如必须得到一定比例数量的背书，才算得到整个系统的认同。而这些条件，就是指的**背书策略**。

背书者Endorser在一个交易流中充当的作用如下：

1. 客户端发送一个背书申请（**SignedProposal**）到Endorser。
2. Endorser对申请进行背书，发送一个申请应答（**ProposalResponse**）到客户端。
3. 客户端将申请应答中的背书组装到一个交易请求（**SignedTransaction**）中。

在此简单提一下交易（**Transaction**）的概念，之后还会在主题文章中详述。交易是一个更大的概念，fabric的任何操作，如chaincode操作，甚至是配置系统的操作，都被定义为一项交易。而背书只是组成交易请求数据的一个准备环节而已，交易请求数据准备就绪后，会被发送到orderer。需要注意的是，一般交易和chaincode交易所进行的背书过程一致，但是背书过程所交流的数据中包含的内容项有所区别。具体可参看/fabric/protos/peer/proposal.proto和proposal\_response.proto中的注释和数据原型定义。

### 背书服务原型和实现

Endorser服务的原型定义在**/fabric/protos/peer/peer.proto**以及对应生成的**peer.pb.go**中，核心代码实现在**/fabric/core/endorser**下。

//proto中定义的服务原型

//在/fabric/protos/peer/peer.proto中定义

service Endorser {

rpc ProcessProposal(SignedProposal) returns (ProposalResponse) {}

}

//proto生成的服务端接口和注册函数

//在/fabric/protos/peer/peer.pb.go中定义

type EndorserServer interface {

ProcessProposal(context.Context, \*SignedProposal) (\*ProposalResponse, error)

}

func RegisterEndorserServer(s \*grpc.Server, srv EndorserServer) {

s.RegisterService(&\_Endorser\_serviceDesc, srv)

}

//核心代码实现Endorser服务

type Endorser struct {

policyChecker policy.PolicyChecker

}

//Endorser专用初始化函数

func NewEndorserServer() pb.EndorserServer {

e := new(Endorser)

e.policyChecker = policy.NewPolicyChecker(

peer.NewChannelPolicyManagerGetter(),

mgmt.GetLocalMSP(),

mgmt.NewLocalMSPPrincipalGetter(),

)

return e

}

//实现ProcessProposal服务

func (e \*Endorser) ProcessProposal(...){...}

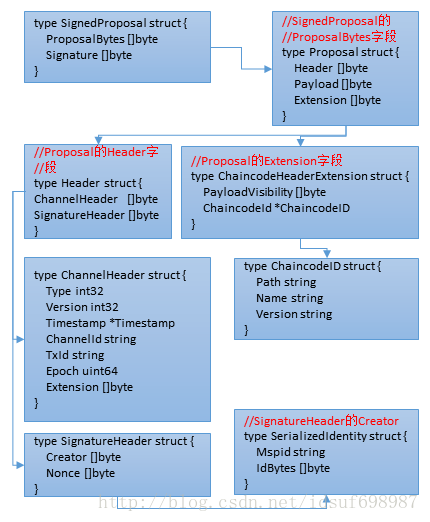
func (e \*Endorser) endorseProposal(...){...}

...

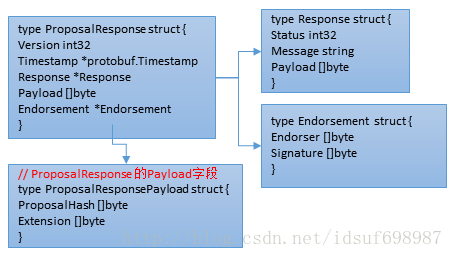
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31

Endorser服务的核心实现中，只有一个核心函数**ProcessProposal**，endorser.go中其余的函数都是相互配合供ProcessProposal调用，处理客户端发来的**SignedProposal**数据，返回**ProposalResponse**数据，完成最终的任务。

**SignedProposal**数据结构如下：



**ProposalResponse**数据结构如下：



按图索骥，我们接下来顺着ProcessProposal函数来看看Endorser函数到底是如何处理从客户端传来的SignedProposal数据，然后包装何种数据到ProposalResponse并将之返回。

//ProcessProposal函数中调用

prop, hdr, hdrExt, err := validation.ValidateProposalMessage(signedProp)

* 1
* 2

首先，**第一步**，ProcessProposal函数使用**ValidateProposalMessage**对所接收的**signedProp**数据进行了验证，并返回signedProp中的一些字段Unmarshal过后的数据。这个函数还是比较绕的，而且写在此处不是太顺理成章，因为涉及到数据验证，所以最起码要知道接收的都有哪些数据或哪种数据是合法的，你才能验证，而这又涉及到了客户端的操作，我们还没讲到。在将来讲到客户端相关操作的时候，可以于此对看。

验证过程涉及到了**MSP**。MSP将在相关主题文章中详述，在此撇开一笔简单介绍：**MSP**又是一个比较大的概念，是**Membership Service Provider**的缩写，个人习惯直译为**成员关系服务提供者**。类似于一个运行的fabric系统网络中，有众多的参与者，而MSP就是为了管理这些参与者，辨识验证哪些人有资格，哪些人没资格，既维护某一个参与者的权限，也维护参与者之间的关系。

回到本文主线，**ValidateProposalMessage**在/fabric/core/common/validation/msgvalidation.go中定义，调用了同文件中的辅助验证函数和/fabric/protos/utils下的工具函数（putils.XXX一类的函数，下文用putils表示）对SignedProposal按结构进行逐步验证。/fabric/protos/utils下的工具函数基本都是对protos中所定义的数据原型的**Unmarshal**操作，如proputils.go中的**GetProposal**函数，就是尝试着将传入的[]byte格式的数据Unmarshal成/fabric/protos/peer/proposal.pb.go中定义的**Proposal**结构数据。

//ValidateProposalMessage函数中调用

//与ValidateProposalMessage函数在同文件中

chdr, shdr, err := validateCommonHeader(hdr)

err = checkSignatureFromCreator(shdr.Creator, ...)

err = utils.CheckProposalTxID(...)

switch common.HeaderType(chdr.Type) {

case ...

case common.HeaderType\_ENDORSER\_TRANSACTION:

chaincodeHdrExt, err := validateChaincodeProposalMessage(prop, hdr)

default:

...

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

参看结构图，**validateCommonHeader**函数验证了SignedProposal中的ProposalBytes中的Header，主要验证的SignatureHeader中的Creator不为空、Nonce不为空且存在，ChannelHeader中的Type必须是ENDORSER\_TRANSACTION、CONFIG\_UPDATE、CONFIG三者之一，并顺道返回了Unmarshal出来的对应的结构体数据对象。

**checkSignatureFromCreator**函数利用validateCommonHeader顺道返回的结构体数据，验证了SignedProposal中的证书、背书策略，发起者的身份等的有效性，是核心验证函数。这里面大有文章，也使用到了MSP，进而使用到了bccsp，但也正因这样，将此细节放入MSP服务主题文章中详述。

**CheckProposalTxID**函数将SignatureHeader中由Nonce和Creator联合生成的Hash字符串值与ChannelHeader中的TxId做对比，这也同时说明TxId就是Nonce和Creator联合生成的Hash值。这里的Nonce是为了防止replay attack，在加密技术中经常用到。

**validateChaincodeProposalMessage**函数简单验证了Proposal中的Extension对应的结构体ChaincodeHeaderExtension中的PayloadVisibility是否为空。该字段控制着Proposal的payload在最终的交易和在ledger中能用到的范围。至此，对接收的SignedProposal对象**signedProp**验证完毕，并返回SignedProposal中的Proposal，Proposal中的Header和ChaincodeHeaderExtension字段，分别为**prop，hdr，hdrExt**。对SignedProposal的验证内容，在/fabric/protos/peer/proposal.proto中定义SignedProposal原型时，就注释了4点，可以阅读参考。

//ProcessProposal函数中调用

chdr, err := putils.UnmarshalChannelHeader(hdr.ChannelHeader)

shdr, err := putils.GetSignatureHeader(hdr.SignatureHeader)

if syscc.IsSysCCAndNotInvokableExternal(hdrExt.ChaincodeId.Name){...}

* 1
* 2
* 3
* 4

紧接着，**第二步**，在ProcessProposal函数中，因为validation.ValidateProposalMessage没有返回，所有再次使用putils下的函数获取Header下的ChannelHeader和SignatureHeader，分别为**chdr，shdr**。然后用函数**IsSysCCAndNotInvokableExternal**验证目前所处理的SignedProposal所涉及的chaincode的ID是否是系统chaincode，若是，是否能被外部调用。如果该chaincodeID是系统chaincode且不能被外部调用，则返回true，则进入if分支，返回相应错误。这表明，发送SignedProposal所在的chaincode，若是系统chaincode时，要保证该系统chaincode允许被外界调用。在系统chaincode中，InvokableExternal字段设定了是否可以被外界调用。

//频道ID

chainID := chdr.ChannelId

//交易ID

txid := chdr.TxId

if chainID != "" {

lgr := peer.GetLedger(chainID)

if \_, err := lgr.GetTransactionByID(txid); err == nil { return }

if !syscc.IsSysCC(hdrExt.ChaincodeId.Name) {

if err = e.checkACL(signedProp, chdr, shdr, hdrExt);

...

}

}else{

//do nothing

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

**第三步**，ValidateProposalMessage的检查中并没有检查频道ID是否为空，当频道ID不为空时，程序根据频道ID调用GetLedger获取peer本地的账本PeerLedger，然后根据交易ID调用账本对象自身函数查看该交易ID是否已经存在于账本中，即交易的唯一性检查。接下来进行策略检查，当chaincode\*\*不是**系统chaincode时，会调用背书者成员policyChecker的函数checkACL，对背书者接收的SignedProposal是否符合所依赖频道的**写者策略\*\*（writers policy of the chain）进行检查（系统chaincode的检查在其他地方）。成员policyChecker在背书者创建的时候被赋予为policy.NewPolicyChecker(...)的值，详细的过程写到下文的背书策略中。如果频道ID为空，则什么都不做（当前版本是这样），注释是这么说的：因为交易忽略了唯一性检查，没有频道ID的proposal不会影响到ledger，也不会被提交（submitted）。没有频道ID的proposal是对照peer的本地MSP验证有效的，而不是通过调用ValidateProposalMessage函数。

//交易模拟器接口

var txsim ledger.TxSimulator

//账本历史查询接口

var historyQueryExecutor ledger.HistoryQueryExecutor

if chainID != "" {

//交易模拟器

if txsim, err = e.getTxSimulator(chainID); err != nil {...}

//账本历史查询器

if historyQueryExecutor, err = e.getHistoryQueryExecutor(chainID); err != nil {...}

ctx = context.WithValue(ctx, chaincode.HistoryQueryExecutorKey, historyQueryExecutor)

defer txsim.Done()

}

//模拟交易

cd, res, simulationResult, ccevent, err := e.simulateProposal(...)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

**第四步**，当频道ID不为空时，背书者对象使用自身的函数e.getTxSimulator(chainID)和e.getHistoryQueryExecutor(chainID)，根据频道ID分别获取了交易模拟对象和账本历史查询对象，进行**模拟（simulate）交易**。从这里可以看出，频道ID **chainID** 也被作为了这个频道的账本的名称，因为peer的账本是存在在/fabric/core/peer/peer.go中的chains映射中的，映射的key就是频道ID。

和上一步类似，也是通过频道ID获取账本对象，然后使用账本对象的接口NewTxSimulator得到**交易模拟器**。交易模拟器定义在/fabric/core/ledger/kvledger/txmgmt/txmgr/lockbasedtxmgr/lockbased\_tx\_simulator.go中，隶属于同目录下lockbased\_txmgr.go中的交易管理者LockBasedTxMgr。使用账本对象的接口NewHistoryQueryExecutor得到**账本历史查询器**。历史查询器定义在fabric/core/ledger/kvledger/history/historydb/historyleveldb/historyleveldb\_query\_executer.go中，隶属于同目录下historyleveldb.go中的账本历史数据库historyDB。而且，程序中调用了context.WithValue将账本历史查询器对象添加到了context中（关于context，还请自行学习此标准库的用法）。

得到了模拟所需的对象（交易模拟器和账本历史查询器）后，背书者使用自己的函数simulateProposal模拟交易。这里的模拟，不是说chaincode没有真正的执行，而应该理解是为对现实交易的模拟的意思，即，chaincode中所谓的智能合约的部分被确确实实的执行并产生了相应的结果集合，只不过这个过程是用数字化模拟出来的。simulateProposal中使用了背书者对象的checkEsccAndVscc、getCDSFromLSCC、callChaincode三个内调函数和交易模拟器的GetTxSimulationResults完成模拟任务。

（1）checkEsccAndVscc在目前版本里直接返回nil了，自带TODO标签，在此不延伸。（2）getCDSFromLSCC，若前面获得的交易模拟器不为空，则将其也加入了context。然后开始调用chaincode关于执行的代码，这里是从LSCC中获取指定名字的chaincode的数据。chaincode关于执行的代码在/fabric/core/chaincode中chaincodeexec.go和exectransaction.go中，最终完成核心任务的是exectransaction.go中的Execute函数，其中使用了ChaincodeSupport服务（服务支持各个peer之间的通信交流，这也就是所谓垫片的地位，核心逻辑由主题代码实现，而与各个peer之间通信去实现主题代码的功能，则用该服务支撑）。关于chaincode执行相关的代码，在此只是简单涉及，之后会在专题文章中详述。（3）callChaincode，真正执行了chaincode并返回HTTP状态应答和执行的chaincode事件。这里也是调用chaincode关于执行的代码，HTTP状态应答原型定义在/fabric/protos/commom/common.proto中。（4）GetTxSimulationResults，在lockbased\_tx\_simulator.go中定义，获取执行chaincode的读写集合。

至此，模拟交易函数simulateProposal执行完毕，返回chaincode数据、执行chaincode的应答信息、模拟结果集合、chaincode的执行事件，供下一步使用。这一步中，要注意当频道ID为空时，交易模拟器即为空，之后的相关操作也会有所区别。

var pResp \*pb.ProposalResponse

if chainID == "" {

pResp = &pb.ProposalResponse{Response: res}

}else{

pResp, err = e.endorseProposal(...)

}

pResp.Response.Payload = res.Payload

* 1
* 2
* 3
* 4
* 5
* 6
* 7

最后，**第五步**，背书者对象使用自身函数endorseProposal对模拟交易进行背书，并得到交易申请应答数据pResp \*pb.ProposalResponse。当频道ID为空时，简单的将上一步所得的应答信息赋予Response即返回，当频道ID不为空时，则使用上一步所返回的数据进行交易的背书。endorseProposal中主要调用的也是背书者的内调函数callChaincode，其之上是为了它准备数据，其之后是根据背书返回的数组组装申请应答信息ProposalResponse。模拟交易和背书都调用了callChaincode，而实现了不同的功能，主要起分别作用的是传入其的倒数第三个参数，该参数是chaincode的执行详细说明书ChaincodeInvocationSpec，说明书不同的内容能指导chaincode关于执行的代码实现不同的功能。如背书功能，使用的说明书中ChaincodeID指定的就是系统chaincode中用于背书的**escc**，而chaincode关于执行chaincode的代码也就根据所给定的ChaincodeID找到指定的chaincode执行。

Endorser对象的ProcessProposal进行的五步，可参看**fabirc源码解析10——文档翻译之Architecture**中的章节2部分。

## 频道中的策略检查器

fabirc源码解析7中的Handler对象和fabirc源码解析8中的lscc，qscc对象中，与此篇文章中的背书者Endorser都存在一个相同的成员policyChecker policy.PolicyChecker，且在初始化该对象的时候，都使用了同样的代码：

policyChecker = policy.NewPolicyChecker(

peer.NewChannelPolicyManagerGetter(),

mgmt.GetLocalMSP(),

mgmt.NewLocalMSPPrincipalGetter(),

)

* 1
* 2
* 3
* 4
* 5

这就是所谓的**策略检查器**，而我们通过探寻这个检查器检查了什么，怎么检查的，可以顺带了解一下频道中的各种策略。

**策略**相关的代码集中在**/fabric/core/policy和policyprovider**，**/fabric/common/cauthdsl和policies**，**/fabric/protos/common/policies.proto和对应生成的policies.pb.go**：

* /fabric/protos/common/policies.proto：ImplicitMetaPolicy的原型定义。
* /fabric/common/policies：定义了Policy接口、Manager接口和其实现ManagerImpl，定义了频道策略管理者获取器ChannelPolicyManagerGetter接口（其实现在/fabric/core/peer/peer.go中的channelPolicyManagerGetter）。对应定义了ImplicitMetaPolicy（也是一种类型的策略）。
* fabric/common/cauthdsl：在policy.go中实现了Policy接口，定义和实现了**策略对象提供者provider**，在policyparser.go中实现了原始字符串策略（如”OR(‘A.member’, AND(‘B.member’, ‘C.member’))”）的解析（FromString函数），cauthdsl.go中实现了生成指定策略的评估函数，即策略的Evaluate接口（compile函数），cauthdsl\_builder.go中则定义了用于生成各种所需结构的函数。
* /fabric/core/policy/policy.go：定义了PolicyChecker的接口和其实现policyChecker，定义了策略检查器工厂PolicyCheckerFactory接口。
* /fabric/core/policyprovider/provider.go：实现了PolicyCheckerFactory接口defaultFactory并初始化了一个实例对象。

策略接口定义如下：

//策略接口，在/fabric/common/policies/policy.go中

type Policy interface {

//对比SignedData中的签名是否满足SignedData中策略

Evaluate(signatureSet []\*cb.SignedData) error

}

//错误或拒绝情况下的策略实现

type rejectPolicy string

func (rp rejectPolicy) Evaluate(signedData []\*cb.SignedData) error {

return fmt.Errorf("No such policy type: %s", rp)

}

//策略的实现1，在fabric/common/cauthdsl/policy.go中

type policy struct {

evaluator func([]\*cb.SignedData, []bool) bool

}

func (p \*policy) Evaluate(signatureSet []\*cb.SignedData) error { ... }

//策略的实现2，在/fabric/common/policiesimplicitmeta.go中

type implicitMetaPolicy struct {

conf \*cb.ImplicitMetaPolicy

threshold int

subPolicies []Policy

}

func (imp \*implicitMetaPolicy) initialize(config \*policyConfig) { ... }

func (imp \*implicitMetaPolicy) Evaluate(...) error { ... }

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

回到**策略检查器**身上，策略检查器的定义如下：

//策略检查器接口

type PolicyChecker interface {

CheckPolicy(...) error

CheckPolicyBySignedData(...) error

CheckPolicyNoChannel(...) error

}

//策略检查器实现

type policyChecker struct {

//频道策略管理者获取器接口，在/fabric/peer/peer.go中实现

channelPolicyManagerGetter policies.ChannelPolicyManagerGetter

//本地MSP或MSPManager，在/fabric/msp中定义和实现

localMSP msp.IdentityDeserializer

//MSP主角获取器，在/fabric/msp/mgmt/principal.go中定义和实现

principalGetter mgmt.MSPPrincipalGetter

}

func (p \*policyChecker) CheckPolicy(...){ ... }

func (p \*policyChecker) CheckPolicyNoChannel(...){ ... }

func (p \*policyChecker) CheckPolicyBySignedData(){

...

//获取策略管理者

policyManager, \_ := p.channelPolicyManagerGetter.Manager(channelID)

//根据策略名获取策略对象，在/fabric/common/policies/policy.go中定义和实现

policy, \_ := policyManager.GetPolicy(policyName)

//评定策略

err := policy.Evaluate(sd)

...

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

首先是**从哪里获取策略**的问题，变相的我们就可以知道频道的策略都存储在哪儿了。在策略检查器的三个接口存在内部相互调用，最典型的就是CheckPolicyBySignedData接口，该接口中用三句代码完成了策略评定，如上代码中注释的那样。据此就可以很容易看出策略其实是存储在**策略管理者对象policyManager**中的**config**成员中，策略对象的名字是其所在的路径字符串，这些路径字符串在/fabric/common/policies/policy.go中开始的部分用常量定义，进而在此我们就可以看到都有哪些类型的策略：ChannelReaders、ChannelWriters、ChannelApplicationReaders等，对应的就是频道的读者策略，频道的写者策略，频道应用的读者策略等。根据GetPolicy的过程，我们甚至可以看出策略管理者是怎么编排其所管理的策略对象的，管理者与管理者存在父子关系，呈现的管理形式也和目录结构类似。

其次是**如何验证策略**的问题，不同的策略有不同的验证方法，所以**策略检查器policyChecker**有三个接口。一般的，直接调用所获取的**策略对象**的接口Evaluate进行评定，而在CheckPolicyNoChannel中则使用了**MSP对象**（即MSP或MSPManager）的Verify接口进行评定。

最后是**验证了什么**的问题，根据背书者Endorser使用policyChecker的地方，在checkACL内调函数中。而checkACL是在进行模拟交易simulateProposal之前被调用的（在ProcessProposal函数中），这也就是说policyChecker在背书过程中所做的就是检查一下客户端发来的请求数据是否合法，也就是检查的客户端的签名，也就是说是根据频道的写者策略判断客户端在该频道中是否有权利发送请求。因此，不要把这个策略检查器与验证是否满足背书策略的那个检查（这个是发生在模拟交易后的，由Evaluate完成）混淆。

# fabric源码解析12——peer的MSP服务

**MSP**是**Membership Service Provider**的缩写，个人习惯直译为**成员关系服务提供者**。作用类似于，在一个运行的fabric系统网络中有众多的参与者，MSP就是为了管理这些参与者，辨识验证哪些人有资格，哪些人没资格，既维护某一个参与者的权限，也维护参与者之间的关系。关于MSP更专业的概念，请参阅Fabric文档。

这里说句题外话，最近这几篇文章其实感觉挺别扭的，因为有些模块是相互牵连的，放在一起说吧文章过于复杂，主题不明，分开说又感觉讲不深，形不成体系，比如MSP和BCCSP之间就是这样。之后在start.go完结之后，会有chaincode安装之类操作性较强的文章，在这些文章中，以具体的示例，通过原始数据在系统中被各个服务之间接收、加工、传送，以此来形成比较系统认识。

MSP的核心代码在**/fabric/msp**中，相关代码分布在**/fabric/common/config/msp、/fabric/protos/msp、/fabric/sampleconfig/msp**。主要目录结构如下：

* msp   
  + msp.go - 定义MSP，MSPManager，Identity，SigningIdentity等主要接口
  + mspimpl.go - 实现MSP接口，结构为bccspmsp
  + mspmgrimpl.go - 实现MSPManager接口，结构为mspManagerImpl
  + identities.go - 实现Identity，SigningIdentity接口
  + configbuilder.go - 提供读取证书文件并将其组装成MSP等接口需要的数据结构，转换配置结构体（由FactoryOpts->MSPConfig）等工具函数
  + mgmt - msp的管理代码目录   
    - mgmt.go - 主要文件，localMsp和mspMap都在这个文件，还有多个管理函数
* common   
  + config   
    - msp/config.go - 实现了MSPManager接口，结构为MSPConfigHandler，是用来配置MSP的，用于configtx工具
* protos   
  + msp - 原始的msp配置（MSPConfig）、定义，和对应生成的.go文件
* sampleconfig   
  + msp - msp所需的证书文件等的一个实例

**结构图**：



## 背书检验

fabirc源码解析11所涉及到了Endorser服务使用到了MSP对接收的SignedProposal消息进行检验，在此将详解检验过程。

函数追溯过程：ProcessProposal -> validation.ValidateProposalMessage -> checkSignatureFromCreator。在checkSignatureFromCreator函数中，进行了如下验证：

//接收的参数有：

creatorBytes - SignatureHeader中的Creator

sig - SignedProposal中的Signature

msg - SignedProposal中的ProposalBytes

ChainID - ChannelHeader中的ChannelId

//根据ChainID获取实现IdentityDeserializer对象

mspObj := mspmgmt.GetIdentityDeserializer(ChainID)

//利用获取的对象，根据二进制的creatorBytes数据获取creator对象

creator, err := mspObj.DeserializeIdentity(creatorBytes)

//使用creator的方法进行验证

err = creator.Validate()

err = creator.Verify(msg, sig)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

**GetIdentityDeserializer**在**mgmt.go**中定义，返回的是一个**IdentityDeserializer**接口对象，实际上最终返回的是**bccspmsp**对象或**mspManagerImpl/MSPConfigHandler**对象，因为IdentityDeserializer接口从定义处的注释就可以看出其作用注意在粘合MSP和MSPManager这两个接口，这两个接口都要求实现了**IdentityDeserializer**接口。对应的，当返回的是bccspmsp对象时，是通过**GetLocalMSP**函数，其实质返回的是**localMsp**变量；当返回的是mspManagerImpl/MSPConfigHandler对象时，是通过**GetManagerForChain**函数，其实质返回的是**mspMap**映射的ChainID的值。

这里要另外说的是**MSPConfigHandler**对象，在/fabric/common/config/msp/config.go中定义，该对象将MSPManager接口作为了成员，自然也应实现了IdentityDeserializer接口，但是冲突的是，搜索源码，并没有发现其实现MSPManager接口的代码，想必是让用户自己实现或在具体使用的时候再赋予具体的MSPManager实现对象。GetManagerForChain函数中，若MSPConfigHandler的成员MSPManager为空，则直接返回nil，进而checkSignatureFromCreator函数也会直接返回。在一些test代码中，如mgmt\_test.go中，实例化MSPConfigHandler对象时，其MSPManager成员直接初始化为系统实现的mspManagerImpl或直接给nil。在此将此对象先搁置，等之后的代码中遇到可解明之处了再补。

因此，**mspObj**是bccspmsp对象或mspManagerImpl对象，在此分别用A和B表示。两种对象实现IdentityDeserializer接口的DeserializeIdentity方法略有所不同，原因在于其自身结构的差异，但**殊途同归**。A是一个MSP，而B实现MSPManager，自然肩负这管理MSP的任务，因此其成员中有一个MSP的映射**mspsMap**，算是一堆MSP。

1. A和B都将接收的**creatorBytes**参数Unmarshal成**SerializedIdentity**结构对象，该对象定义在fabric/protos/msp/identities.pb.go中，且其成员**Mspid**和**IdBytes**将用于A和B的后续验证。
2. A验证**Mspid**与自身存储的**name**是否一致后，将**IdBytes**传入内调函数**deserializeIdentityInternal**，以进行进一步的验证；B验证以**Mspid**为key值的映射是否存在于自身的**mspsMap**中后，用switch判断映射的MSP对象的类型，若是系统实现的bccspmsp，则与A一样去调用**deserializeIdentityInternal**，若是用户自己实现的MSP，则再去调用用户实现的DeserializeIdentity方法，我们撇下用户自己实现的MSP的这种情况。
3. deserializeIdentityInternal函数的操作涉及到了**mspObj对象**中的bccsp bccsp.BCCSP成员，该成员是一个**bccsp服务对象**，而bccsp是fabric整个系统的加密服务的提供者，将在主题文章中详述。deserializeIdentityInternal函数接收**IdBytes**，并根据**IdBytes**调用一系列bccsp所提供的函数，如GetHashOpt、Hash、KeyImport，生成**identity对象**所需的数据，最终生成**identity对象**并返回。**identity对象**实现了Identity接口，在msp/identities.go中定义，代表一个身份对象，也提供了验证能力（函数）。这里的身份是用来表明一个会员的身份的，包含了这个会员的名称、使用的证书、加密算法、公匙和MSP对象自身等信息，验证能力指的是其提供的**Validate**和**Verify**两个函数。
4. 使用**identity对象**，也就是creator接收到的值，进行验证和确认。追溯**Validate**和**Verify**两个函数，都在/fabric/msp/identities.go中定义，我们会发现最终使用的是其所包含的MSP对象的Validate和MSP对象所包含的bccsp对象的Verify函数。这里和第3步一样，使用了bccsp服务，在此不再赘述。identity validation（与certificate validation是一个意思，即证书和身份的概念是重叠的，证书就代表着身份）和signature verification这两个短语在fabric文档中是专用的，分别对应两个函数Validate和Verify，也就是说，前者验证的是身份，后者确认的是签名。

了解一个服务模块，基本上问题还是集中在两点：

* **服务模块自身能做什么**。
* **系统如何使用这个模块**。这里的如何包含了何时使用和如何使用两个意思。

我们接下来从这个思路入手，了解MSP服务。这里先进行一个打比方的描述和证书的解释：

**打个比方：**

一个集团化的大型公司，可能有一个集团总部，集团下分若干个子集团，分别涉及不同商业领域（如建设集团，商业发展集团，投资集团等子集团），每个集团下在全国各地有若干个分公司，分公司下有部门，部分中有职员，职员有管理者和普通职员之分。\*\*

**关于证书（fabric里面所用的都是x509证书）：**

* 证书本身是承载公匙的容器，里面最主要的就是公匙，和一些认证信息。比较证书的时候，自然比较公匙。
* SKI是当前证书的标识符，所谓标识符，一般是对公匙进行hash，得到的一段字符标识。
* SKI是当前证书的标识符，AKI是签署方的SKI，也就是签署方的公匙标识符。
* 证书是一级级信任的，比如某个CA的密匙签署了你的密匙，生成了你的证书，也就是该CA认证了你的证书，也就是你的证书就是该CA证书的下一级证书。原来的那个CA证书自己的标识符也是SKI，而对于你这个证书而言，CA那个证书的标识符是你的签署方标识符，也就是AKI。即上一级的SKI变为下一级的AKI。
* x509是证书的一种类型，具体可参看/fabric/sampleconfig下的admincerts，cacerts中的文件。## Identity和SigningIdentity接口

### Identity

type identity struct {

//实例的身份标识，如MSPID等

id \*IdentityIdentifier

//实例所对应的x509证书，代表着身份

cert \*x509.Certificate

//实例的身份公匙

pk bccsp.Key

//“拥有”此实例的MSP实例

msp \*bccspmsp

}

//调用msp的SatisfiesPrincipal接口检查身份实例是否与principal中所描述的那种类型匹配

//如果匹配，则返回nil

func (id \*identity) SatisfiesPrincipal(principal \*msp.MSPPrincipal)error{

return id.msp.SatisfiesPrincipal(id, principal)

}

//调用msp的Validate接口验证这个身份实例

func (id \*identity) Validate() error {

return id.msp.Validate(id)

}

//调用msp的成员bccsp的Verify接口确认签名sig与消息msg是否匹配，也即验证签名的有效性

func (id \*identity) Verify(msg []byte, sig []byte) error {

...

digest, err := id.msp.bccsp.Hash(msg, hashOpt)

...

valid, err := id.msp.bccsp.Verify(id.pk, sig, digest, nil)

...

}

//调用msp的SerializedIdentity接口把此身份实例转为byte形式

func (id \*identity) Serialize() ([]byte, error) {

...

sId := &msp.SerializedIdentity{Mspid: id.id.Mspid, IdBytes: pemBytes}

idBytes, err := proto.Marshal(sId)

...

}

//未列出的接口实现要么没用（不实现，只是直接返回错误或打印一句话），要么是GetXXX系列

...

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37

在此省略Identity接口中**GetXXX**相关的实现，这些都是获取身份信息的，算不上什么功能性接口，而相较于获取身份信息这种比较简单的实现，我们更想知道身份中的数据是如何形成的和这些数据可以做什么。**Identity**是系统成员**身份**的代表，而其所包含的**x509证书**就可以在系统中代表着这个身份，由MSP“拥有”和管理的。Identity其实不真正实现什么功能，只是比较本质的起到代表一个成员的作用（就足够了），这点可以从SatisfiesPrincipal，Validate，Verify，Serialize这样的功能性接口的实现看出：这些实现都是调用管理这个身份实例的**msp**的接口实现的，而身份实例自身只提供身份数据。**身份**在系统中，可以代表一个peer结点的身份，可以代表一个组织的身份，等身份性的对象。还有一点，如果一个身份中包含所属的组织信息（同时也就是说身份可能没有组织信息），该组织信息是被包含在x509证书中的。

从SatisfiesPrincipal这个接口实现，可以引出身份的**类型**之分。类型之分的原始定义在**/fabric/protos/common**中的**msp\_principal.proto**和对应生成**msp\_principal.pb.go**。结构体为**MSPPrincipal**，有以下三种类型的身份：

* **ROLE**，表示一个角色，有**成员MEMBER**和**管理者ADMIN**之分。类似于一个公司中，普通职员和经理。
* **ORGANIZATION\_UNIT**，表示**组织单位**。类似于一个公司中的，一个部门。
* **IDENTITY**，就表示一个普通的**身份**。是与ORGANIZATION\_UNIT相对的，类似于一个公司中，一个个体。

### SigningIdentity

type signingidentity struct {

//签名者身份身份实例

identity

//crypto库中的“专用签名笔”

signer crypto.Signer

}

//在msg上进行签名

func (id \*signingidentity) Sign(msg []byte) ([]byte, error) {

//获取用于签名的hash选项（用哪种hash技术进行签名）

hashOpt, err := id.getHashOpt(id.msp.cryptoConfig.SignatureHashFamily)

...

//获取哈希值

digest, err := id.msp.bccsp.Hash(msg, hashOpt)

...

//使用专用签名笔进行签名

return id.signer.Sign(rand.Reader, digest, nil)

}

//其他实现都是未实质实现（只是返回一个错误）

...

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

签名者身份是一个拿有“**专用签名笔**”的身份。自然，也用来进行签名。在MSP管理的所有身份对象中，可以指定一些成员用来对消息进行签名。在Sign实现中，使用的是“专用签名笔”crypto标准库里的Signer对象，同时用了MSP的一些功能作为签名的辅助。

**可以把Identity想象成一个部门或一个个体职员，当为一个部门时，其为组织类型，当为一个个体时，又有管理者和普通职员之分。**

## MSP和MSPManager接口

### MSP

type bccspmsp struct {

//root CAs

rootCerts []Identity

//intermediate CAs

intermediateCerts []Identity

//签名身份

signer SigningIdentity

//管理员身份列表

admins []Identity

//加密算法

bccsp bccsp.BCCSP

//MSP的名字

name string

//验证选项

opts \*x509.VerifyOptions

//废除CAs

CRL []\*pkix.CertificateList

//组织列表

ouIdentifiers map[string][][]byte

//加密选项

cryptoConfig \*m.FabricCryptoConfig

}

//根据配置信息conf1建立MSP对象，即利用conf1填充这个bccspmsp实例中的字段

func (msp \*bccspmsp) Setup(conf1 \*m.MSPConfig) error { ... }

//验证给定的身份id是否有效

func (msp \*bccspmsp) Validate(id Identity) error { ... }

//验证给定的身份id是否与所给的principal中所描述的类型相匹配

func (msp \*bccspmsp) SatisfiesPrincipal(id Identity, principal \*m.MSPPrincipal) error{ ... }

//其余都为GetXXX系列和辅助性内调函数

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

关于**bccspmsp**的结构体中的字段，可以对照**fabirc源码解析9——文档翻译之MSP**的**MSP配置**章节所讲的一个MSP需要指定哪些数据，就很容易理解了。这里起名为bccspmsp，指的是该实现使用了bccsp作为其加密技术的提供者，正如其有一个成员bccsp一样。由对Identity接口所讲的那样，MSP实质上承载着身份验证和管理的任务。**可以把MSP想象成一个子公司，每个其所管理的成员要么是公司的一个个体，要么是公司的一个部门，个体要么是经理，要么是普通职员，而且这些主体都用一个通用的身份接口Identity表示**。

**Setup**：Setup的过程会如你想象的那样，就是对参数**conf1**中所携带的数据进行择选，然后填充bccspmsp的各个字段，这其中用到了bccsp的加密技术，如果某些所需的数据conf1中没有，则置默认值。这也就是说，**conf1**中有我们需要的数据，那么这个conf1是怎么形成的呢，它的数据是从哪儿来的？如果不想搜索源代码以查看这个接口在何时何处被调用（因为既然要调用，肯定会准备好数据，依此可以寻找conf1是怎么形成的），那么可以直接看test或mock文件，这里，比较简单直接的且能说明问题的就是/fabric/msp/mspwithintermediatecas\_test.go这个测试文件，里面简单查看Setup调用之前的conf数据组装，就可以知道conf是怎么来的了。

**Validate**：验证身份有效，主要验证（满足）三点：

* 身份中所携带的x509证书存在于bccsmsp的rootCerts或intermediateCerts之中。这需要你比较熟悉x509证书Certificate的方法Verify的用法（追溯Validate函数可以找到），原型为func (c \*Certificate) Verify(opts VerifyOptions) (chains [][]\*Certificate, err error)，这里使用到的参数opts就是bccsmsp成员**opts**，这个成员相当与一个证书地图，即指示函数去哪些证书池（CertPool）中与c这个证书进行对比。opts在bccsmsp的**Setup**实现中被初始化为包含rootCerts和intermediateCerts两个证书池。
* 身份中所携带的x509证书不在CRL中。这里的对比不是直接用证书本身对比，因为CRL所对应的结构体是pkix.CertificateList。
* 身份中所携带的组织信息有效。

**SatisfiesPrincipal**：验证给定的身份id是否与所给的principal中所描述的类型相匹配。过程就是根据身份的类型所进行的一个switch-case分支判断的过程，这个判断就是分别抽取id和principal中对应的字段信息进行对比。

### MSPManager

type mspManagerImpl struct {

//一个MSP的映射，包含所有建立的MSP，添加的MSP

mspsMap map[string]MSP

//是否正常启用的标识

up bool

}

//根据所给的msps填充该实例中的mspsMap

func (mgr \*mspManagerImpl) Setup(msps []MSP) error { ... }

//根据所给的byte格式的身份，返回其对应的Identity结构体

func (mgr \*mspManagerImpl) DeserializeIdentity(serializedID []byte) (Identity, error) { ... }

//GetXXX系列

func (mgr \*mspManagerImpl) GetMSPs() (map[string]MSP, error) {

return mgr.mspsMap, nil

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

**可以把MSPManager想象成一个子集团，其管理集团旗下的各个子公司，而且这些子公司主体都用一个MSP接口表示**。

## mgmt

//本地MSP实例对象

var localMsp msp.MSP

//本地存储的按chaincodeID为key的MSPManager映射

var mspMap map[string]msp.MSPManager = make(map[string]msp.MSPManager)

//从目录dir中加载本地MSP

func LoadLocalMsp(dir string, bccspConfig \*factory.FactoryOpts, mspID string) error {

...

//调用configbuilder.go中的GetLocalMspConfig将FactoryOpts转为MSPConfig

conf, err := msp.GetLocalMspConfig(dir, bccspConfig, mspID)

...

//根据conf调用localMsp的Setup填充localMsp

return GetLocalMSP().Setup(conf)

}

//获取指定chaincodeID的MSPManager，如果没有，则创建

func GetManagerForChain(chainID string) msp.MSPManager { ... }

//其他公共函数，GetXXX系列和临时替代者XXXSetMSPManager

...

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

mgmt就是management的缩写，也就是管理的意思。这里的本地指代的是peer结点，而mgmt也不是一个具体的对象。可以说真正经常供peer其他模块调用和使用的函数是这里提供的函数，也即mgmt才是对外提供MSP服务的窗口。MSP数据存储的主体就是**localMsp**和**mspMap**。

* **localMsp**，本地MSP，管理了所有本地的身份。**LoadLocalMsp**是给localMsp读入MSP数据的，按部就班，攒成配置选项，然后调用Setup填充localMsp。GetXXX系列中与localMsp相关的则是向外界提供这个本地的MSP。在**fabric源码分析5–kvledger的初始化**中开篇所遗留的关于**InitCrypto**的问题，在此就可以解释一下，InitCrypto在/fabric/peer/main.go中main函数中被调用，在/fabric/peer/common/common.go中定义。而InitCrypto中就调用了**LoadLocalMsp**，也就初始化了peer结点本地MSP。在/fabric/orderer/main.go中，也调用了**LoadLocalMsp**，也就初始化了orderer结点的本地MSP。
* **mspMap**是一个以chaincodeID为key的映射，可以把每个chaincode当作是一个不同的商业领域，而把mspMap看作是按chaincode分的每个chaincode上的子集团的集合，这些子集团在各自的商业领域管理这自己的子公司（MSP）。**GetManagerForChain**函数既是数据赋予者也是数据给予者，当所给定的chaincodeID不存在时，会自动创建一个MSPManager对象，当所给的chaincodeID存在时，则返回对应的MSPManager。如在/fabric/core/scc/lscc/lscc.go中的checkInstantiationPolicy函数中调用了**GetManagerForChain**，获取了指定chainName的chaincode的MSPManager以供使用。

**可以把mgmt想象成一个集团化公司的总部，管理各个子集团（MSPManager）并统一向外界提供自身所管理的成员数据和信息，这也就是所谓的MSP服务。**

# fabric源码解析13——peer的BCCSP服务

## 加密的一般话题

加密涉及到了内容挺复杂的，是一门专业性很强的学科。笔者没有专门学过，在此只是略讲一些bccsp服务所涉及到的皮毛：

* RSA - 一种非对称的加密算法，用于加密。有几种族簇，如RSA1024，RSA2048等。
* AES - 一种块加密算法，用于加密成块的大量数据。有几种族簇，如AES128，AES192等。
* ECDSA - 一种椭圆曲线签名，用于签名。有几种族簇，如ECDSAP256，ECDSAP384等。
* Hash - 哈希，有几种族簇，如SHA256，SHA3\_256等。
* HMAC - 密匙相关的哈希运算消息认证码。
* x509 - 证书的一种，可参看文章12中对证书的解释。
* PKCS#11 - 一套标准安全接口，可与安全硬件相关，以上的这些东西，可以找它来建立，读取，写入，修改，删除等操作进行管理。

fabric所用到的这些技术的常量名称在/fabric/bccsp/opts.go中开始的部分定义，如ECDSA支持ECDSAP256，ECDSAP384等几种类型。

## BCCSP服务结构

**BCCSP**，是**blockchain cryptographic service provider**的缩写，个人译作**区域链加密服务提供者**，为fabric项目提供各种加密技术，签名技术，工具的性质很强，MSP服务模块中就使用到了BCCSP。这里需要说明的一点是，工具性强，也就说明了，如果不是想专门学这一领域，其实不用太在乎其实现的细节，只要用就行了。BCCSP服务的代码集中在**/fabric/bccsp**中，**目录结构**如下：

* mocks - 模拟代码文件夹，可以参看之帮助理解bccsp服务
* **signer** - 实现的是crypto标准库的Signer接口，可参看文章12中MSP服务实现中带“专用签名笔”的身份signingidentity，该目录的签名接口是专用于向外界提供签名对象的功能的。
* **factory** - bccsp服务工厂
* **pkcs11** - **bccsp服务实现之一**：HSM基础的bccsp（the hsm-based BCCSP implementation）
* **sw** - **bccsp服务实现之二**：软件基础的bccsp（the software-based implementation of the BCCSP）
* utils - bccsp服务工具函数
* bccsp.go - 定义了BCCSP，Key接口，众多BCCSP接口所使用到的**选项**接口，如Key，KeyGenOpts，KeyDerivOpts等
* keystore.go - 定义了key的管理存储接口，如果生成的key不是暂时的，则存储在该接口的实现对象中，如果是暂时性的，则不存储
* XXXopts.go - XXX表示该目录下的各种值，bccsp服务实现所使用到的各种技术的**选项**实现

从以上可以看出bccsp服务有两种实现：**pkcs11和sw**。简单明了的解释的话（虽不太精准），就是pckcs11是硬件基础的加密服务实现，sw是软件基础的加密服务实现。这个硬件基础的实现以 <https://github.com/miekg/pkcs11> 这个库为基础，而HSM是Hardware Security Modules，即硬件安全模块的缩写。相对应的两种bccsp服务实现，这里有两种工厂，两种工厂为其他使用bccsp服务的模块提供了窗口函数（就是给其他模块提供窗口的函数，这些函数一般统一管理自己服务的功能模块，供外界调用，如后文所讲的InitFactories函数）。所有产生的bccsp实例存储在factory/factory.go中所定义的全局变量中。

## bccsp中的接口和选项

### 接口

//在fabric/bccsp/bccsp.go中定义

type BCCSP interface {

//根据key生成选项opts生成一个key

//与key有关的选项opts选项要适合原始的key（与“证书是一级一级的认证”对看）

KeyGen(opts KeyGenOpts) (k Key, err error)

//根据key获取选项opts从k中重新获取一个key

KeyDeriv(k Key, opts KeyDerivOpts) (dk Key, err error)

//根据key导入选项opts从一个key原始的数据中导入一个key

KeyImport(raw interface{}, opts KeyImportOpts) (k Key, err error)

//根据SKI返回与该接口实例有联系的key

GetKey(ski []byte) (k Key, err error)

//根据哈希选项opts哈希一个消息msg，如果opts为空，则使用默认选项

Hash(msg []byte, opts HashOpts) (hash []byte, err error)

//根据哈希选项opts获取hash.Hash实例，如果opts为空，则使用默认选项

GetHash(opts HashOpts) (h hash.Hash, err error)

//根据签名者选项opts，使用k对digest进行签名，注意如果需要对一个特别大的消息的hash值

//进行签名，调用者则负责对该特别大的消息进行hash后将其作为digest传入

Sign(k Key, digest []byte, opts SignerOpts) (signature []byte, err error)

//根据鉴定者选项opts，通过对比k和digest，鉴定签名

Verify(k Key, signature, digest []byte, opts SignerOpts) (valid bool, err error)

//根据加密者选项opts，使用k加密plaintext

Encrypt(k Key, plaintext []byte, opts EncrypterOpts) (ciphertext []byte, err error)

//根据解密者选项opts，使用k对ciphertext进行解密

Decrypt(k Key, ciphertext []byte, opts DecrypterOpts) (plaintext []byte, err error)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

### 选项

bccsp文件夹中任何带opt字眼的文件，都是和选项有关的源码。关于对象的配套选项，我们在讲MSP服务的时候就见识过。**根据一个选项的不同配置，对象主体可以得到不同的数据或进行不同的操作**，这也是一种比较值得学习的**语言上的组织技巧**。尤其是在bccsp这种涉及的技术比较多，而每个对象自身又分为好多类的情况。在此以**哈希选项HashOpts**和**key导入选项KeyImportOpts**作为例子进行说明：

//在/fabric/bccsp/bccsp.go中定义

//哈希选项接口

type HashOpts interface {

Algorithm() string //获取hash算法字符串标识，如"SHA256"，"SHA3\_256"

}

//在/fabric/bccsp/hashopts.go中定义

//哈希选项实现之一，SHA256选项

type SHA256Opts struct {

}

func (opts \*SHA256Opts) Algorithm() string {

return SHA256

}

//哈希选项实现之二，SHA384选项

type SHA384Opts struct {

}

func (opts \*SHA384Opts) Algorithm() string {

return SHA384

}

--------------------------------------------------

//在/fabric/bccsp/bccsp.go中定义

//key导入选项接口

type KeyImportOpts interface {

Algorithm() string //返回key导入算法字符串标识

Ephemeral() bool //如果生成的key是短暂的（ephemeral），返回true，否则返回false

}

//在/fabric/bccsp/opts.go中定义

//key导入选项接口实现之一，ECDSA公匙的导入选项

type ECDSAPKIXPublicKeyImportOpts struct {

Temporary bool

}

func (opts \*ECDSAPKIXPublicKeyImportOpts) Algorithm() string {

return ECDSA

}

func (opts \*ECDSAPKIXPublicKeyImportOpts) Ephemeral() bool {

return opts.Temporary

}

//key导入选项接口实现之二，ECDSA私匙的导入选项

type ECDSAPrivateKeyImportOpts struct {

Temporary bool

}

func (opts \*ECDSAPrivateKeyImportOpts) Algorithm() string {

return ECDSA

}

func (opts \*ECDSAPrivateKeyImportOpts) Ephemeral() bool {

return opts.Temporary

}

//比较特殊的，比如签名选项接口SignerOpts

//由于因为使用的标准库，因此使用到此选项时多赋值为nil，bccsp源码中未实现

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48

## SW实现方式

BCCSP的SoftWare（SW）实现形式是默认的形式，这点仅从/fabric/bccsp/factory/opts.go中工厂的默认选项DefaultOpts和核心配置文档中关于bccsp的配置就可以看出来。主要使用的包是标准库**hash**和**crypto**（包括其中的各种包，如**aes**，**rsa**，**ecdsa**，**sha256**，**elliptic**，**x509**等）。

**目录结构**：

* /fabric/bccsp/sw   
  + impl.go - bccsp的sw实现**impl**
  + internals.go - 签名者、鉴定者、加密者、解密者接口定义
  + conf.go - bccsp的sw实现的配置定义
  + ———————————————————————–
  + aes.go - aes类型的生成key函数、加密者/解密者实现
  + ecdsa.go - ecdsa类型的签名者、公匙/私匙鉴定者实现
  + rsa.go - rsa类型的签名者、公匙/私匙鉴定者实现
  + ———————————————————————–
  + aeskey.go - aes类型的Key接口实现
  + ecdsakey.go - ecdsa类型的Key接口实现
  + rsakey.go - rsa类型的Key接口实现
  + ———————————————————————–
  + dummyks.go - dummy类型的KeyStore接口实现dummyKeyStore，当生成的key是短暂的，则说明这些key不会保存到文件中，而是保存到内存中，系统一关闭，这些key就消失了
  + fileks.go - file类型的KeyStore接口实现fileBasedKeyStore，当生成的key不是短暂的，则说明这些key在导入时，会存储在文件中，即便系统关闭，这些key也不会消失

**BCCSP接口实现**：

//在/fabric/bccsp/sw/impl.go中定义

//SW bccsp的实例结构体

type impl struct {

conf \*config //bccsp实例的配置

ks bccsp.KeyStore //key存储系统对象，存储和获取Key对象

encryptors map[reflect.Type]Encryptor //加密者映射

decryptors map[reflect.Type]Decryptor //解密者映射

signers map[reflect.Type]Signer //签名者映射，Key实现的类型作为映射的键

verifiers map[reflect.Type]Verifier //鉴定者映射，Key实现的类型作为映射的键

}

//专用生成函数

func New(...) (bccsp.BCCSP, error) { ... }

//接口实现

func (csp \*impl) KeyGen(opts bccsp.KeyGenOpts) (k bccsp.Key, ...) { ... }

...

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

粗线条上看，由于impl对象和各种操作选项的存在，绝大部分接口的实现都是以**switch-case**为主干，根据选项的类型或配置，分情况完成功能，且每个分支的操作基本都很类似，如KeyGen。接下来一一介绍：

1. **KeyGen** - 根据key生成选项不同，生成三种系列的key：**ECDSA，AES，RSA**。ECDSA使用库ecdsa的GenerateKey函数，AES使用自定义的GetRandomBytes函数（在aes.go中实现，包装了rand.Read），RSA使用库rsa的GenerateKey函数。每个系列又根据具体参数的不同生成不同“尺寸”的key，具体的细节略过。最后返回不同的Key实现对象，如ecdsaPrivateKey，aesPrivateKey等（分别定义在同目录中的XXXkey.go中）。这里要注意的是，当前版本中用于签名的key，只支持ECDSA系列的key，这是官方文档中所说的，但是从实现上看签名的支持不止一种，又但是，bccsp本质上无论实现多上中key，也只有被调用者决定使用哪一种。而MSP模块是bccsp的使用者之一，应该是在它这个地方只认ECDSA的key。
2. **KeyDeriv** - 根据key获取选项opts从k中重新获取一个key，这里的重新获取可以理解为把k中的内容重新打乱再生成一个key。处理两种key类型：**ecdsaXXXKey（XXX代表Public和Private），aesPrivateKey**。最后返回打乱后重新生成的两种类型的key：**reRandomizedKey**和**hmacedKey**。对于重新生成的key，如果选项中指定的不是暂时性的key，则会在ks中存储。
3. **KeyImport** - 从原始的数据raw中取出选项opts指定的key，如果不是临时性的，则在ks中存储，最后返回key。这里的原始，指的是[]byte或者含有key的数据（如证书数据里的key）。raw是一个空接口，也就是说可以接收任何形式的原始数据。为了得到key，对原始数据raw的转化或抽取，一部分使用了utils下的工具函数utils.Clone和utils.DERToPublicKey，如AES256，ECDSAPrivateKey等类型的key；一部分直接用go语言的断言raw.(\*Key类型)，如ECDSAGoPublicKey，RSAGoPublicKey等类型的key。
4. **Hash** - 根据哈希选项opts对一个消息msg进行哈希，返回该msg的哈希值。如果opts为空，则使用默认选项。这个比较好理解，种类支持SHA2，SHA3哈希家族。
5. **Sign** - 根据签名者选项，使用k对digest进行**签名**，这里的签名者选项在当前版本里没什么用处，调用者都给的是nil。**签名**涉及到了**Key接口**和**签名者Signer**在SW bccsp中的实现。Key接口有三种实现：ecdsakey.go中的ecdsaPublicKey/ecdsaPrivateKey，rsakey.go中的rsaPublicKey/rsaPrivateKey，aeskey.go中的aesPrivateKey，这里签名（自然）使用的都是私匙。签名者接口Signer定义在同目录下的internals.go中，有两种类型的实现：ecdsa.go中的ecdsaSigner和rsa.go中的rsaSigner。参看SW bccsp专用生成函数New的signers赋值部分，可知用到了两种对应类型的Key和Signer：**ecdsaPrivateKey - ecdsaSigner**和**rsaPrivateKey - rsaSigner**。这里**签名**的实现是，从该bccsp实例的签名者集合成员signers获取类型为reflect.TypeOf(k)的签名者signer，然后直接调用signer的接口Sign，追溯，ecdsaSigner使用ecdsa库的Sign函数，rsaSigner使用rsa库PrivateKey结构体的Sign函数。
6. **Verify** - 根据鉴定者选项opts，通过对比k和digest，**鉴定**签名。**鉴定**涉及到了**Key接口**和**鉴定者接口Verifier**在SW bccsp中的实现。Key接口实现如上描述。鉴定者接口Verifier定义在同目录下的internals.go中，有两种类型的实现：ecdsa.go中的ecdsaPublicKeyKeyVerifier/ecdsaPrivateKeyVerifier和rsa.go中的rsaPublicKeyKeyVerifier/rsaPrivateKeyVerifier。参看SW bccsp专用生成函数New的verifiers赋值部分，可知所有鉴定者（与对应的Key接口实现）都有用到。这里**鉴定**的实现是，从该bccsp实例的鉴定者集合成员verifiers获取类型为reflect.TypeOf(k)的鉴定者verifier，然后直接调用verifier的接口Verify，追溯，ecdsaXXXKeyVerifier使用ecdsa库的Verify函数，rsaXXXKeyVerifier使用rsa库的VerifyPSS函数（这里XXX表示PublicKey或Private）。
7. **Encrypt** - 根据加密者选项opts，使用k\*\*加密**plaintext。**加密**涉及到了\*\*Key接口**和**加密者接口Encryptor**在SW bccsp中的实现。Key接口实现如上描述。加密者接口Encryptor定义在同目录下的internals.go中，在aes.go中实现（只能是aes，因为只有aes是用来加密的）：aescbcpkcs7Encryptor。参看SW bccsp专用生成函数New的encryptors赋值部分，只有**aesPrivateKey - aescbcpkcs7Encryptor**被使用。这里**加密**的实现是，从该bccsp实例的加密者集合成员encryptors获取类型为reflect.TypeOf(k)的加密者encryptor，然后直接调用encryptor的接口Encrypt，追溯，aescbcpkcs7Encryptor使用了aes库的加密流程进行加密。
8. **Decrypt** - **解密**与加密类似，过程类似，也是只有aes配套实现，最终使用aes库解密流程进行解密。
9. **GetXXX** - GetXXX系列接口，获取实例对象中的数据，略。

在fabirc源码解析12——peer的MSP服务文中背书检验部分的第3、4点，涉及到了包含在msp中的bccsp对象成员，使用了bccsp对象的GetHashOpt、Hash、KeyImport、Verify等接口用以生成identities对象或identities自身一些接口实现。在此可以对看。

## pkcs11实现方式

BCCSP的pkcs11实现形式主要使用到的库与sw实现如出一辙，但外加一个**github.com/miekg/pkcs11**库，最好参看其文档以熟悉pkcs11的简要操作。pkcs11（PKCS，Public-Key Cryptography Standards）是一套非常通用的接口标准，可以说这里是用pkcs11实现了bccsp的功能，也为fabric支持热插拔和个人安全硬件模块提供了服务。这点可以从bccsp的pkcs11的实现实例的专用生成函数New（参看下文）中所调用的loadLib函数可以看出来：loadLib加载了一个系统中的动态库，能加载系统的动态库，就可以和驱动、热插拔、连接电脑的字符设备联系在一起。比如将来，开发出了一款在区域链上类似于现在网上银行所用的U盾之类的个人身份或安全交易硬件模块或芯片，这些硬件模块或芯片只需要也遵循pkcs11，fabric即可对此进行支持和扩展。

对于pkcs11的所提供的接口，在此提供两个文档地址，读者可以稍作了解： <https://www.ibm.com/developerworks/cn/security/s-pkcs/> ， <http://docs.oracle.com/cd/E19253-01/819-7056/6n91eac56/index.html#chapter2-9> 。这些文档与fabric和区域链无关，但是因为pkcs11是通用的接口，所以有一定参考价值。pkcs11库中的解释相对过于简单。

**目录结构**：

* /fabric/bccsp/pkcs11   
  + impl.go - bccsp的pkcs11实现**impl**
  + conf.go - 定义了bccsp服务的配置和PKCS11Opts、FileKeystoreOpts、DummyKeystoreOpts选项
  + **pkcs11.go** - 以**pkcs11库**为基础，包装各种pkcs11功能，实现了impl基于pkcs11的内调函数，和一些bccsp服务使用到的独立的内调函数
  + ———————————————————————–
  + aes.go - 实现aes类型的生成key、加密、解密函数
  + ecdsa.go - 实现impl在ecdsa技术下的签名函数signECDSA和鉴定函数verifyECDSA
  + ———————————————————————–
  + aeskey.go - 实现aes类型的Key接口，只实现私匙aesPrivateKey
  + ecdsakey.go - 实现ecdsa类型的Key接口，实现了公匙ecdsaPublicKey，私匙ecdsaPrivateKey
  + rsakey.go - 实现了rsa类型的Key接口，实现了公匙rsaPublicKey，私匙rsaPrivateKey
  + ———————————————————————–
  + dummyks.go - dummy类型的KeyStore接口实现DummyKeyStore
  + fileks.go - file类型的KeyStore接口实现FileBasedKeyStore

**BCCSP接口实现**：

type impl struct {

conf \*config //配置

ks bccsp.KeyStore //key存储系统对象，存储和获取Key对象

ctx \*pkcs11.Ctx //标准库的pkcs11上下文

sessions chan pkcs11.SessionHandle //实质是uint，会话标识符频道，默认10个缓存

slot uint //安全硬件外设连接插槽标识号

lib string //加载库所在路径

noPrivImport bool //禁止导入私匙标识

softVerify bool //使用软件方式鉴定签名标识

}

//专用生成函数

func New(opts PKCS11Opts, keyStore bccsp.KeyStore) (bccsp.BCCSP, error) { ... }

//接口实现

func (csp \*impl) KeyGen(opts bccsp.KeyGenOpts) (k bccsp.Key, err error) { ... }

...

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

BCCSP的pkcs11实现的骨架在impl.go中与sw的实现基本一致，只是追溯到最终实现的语句时，sw实现是使用crypto库下的各个包进行签名，加密，密匙导入等，**而pkcsll则用pkcs11包对数据进行了多一层的处理，使用pkcs11提供的上下文（pkcs11.Ctx）和会话（SessionHandle）之上对签名，密匙，加密等进行管理，这也是pkcs11.go文件的作用**。两者最大的不同是一个面向软件，一个面向硬件，pkcs11自身又非常的冗杂，因此在此只讲pkcs11中与安全硬件模块建立连接的**loadLib**函数。

**loadLib函数**在pkcs11.go中定义，供专用生成函数New使用。为建立与安全硬件模块的通信，进行了如下步骤：

1. 根据所给的系统动态库路径lib加载动态库（如openCryptoki的动态库），调用pkcs11.New(lib)建立pkcs11实例ctx。ctx相当于fabric与安全硬件模块通信的桥梁：**bccsp<–>ctx<–>驱动lib<–>安全硬件模块**，只要驱动lib是按照pkcs11标准开发。
2. ctx.Initialize()进行初始化。
3. 从ctx.GetSlotList(true)返回的列表中获取由label指定的插槽标识slot（这里的槽可以简单的理解为电脑主机上供安全硬件模块插入的槽，如USB插口，可能不止一个，每一个在系统内核中都有名字和标识号）。
4. 尝试10次调用ctx.OpenSession打开一个会话session（会话就是通过通信路径与安全硬件模块建立连接，可以简单的理解为pkcs11的chan）。
5. 登陆会话ctx.Login。
6. 返回ctx，slot，会话对象session，用于赋值给impl实例成员ctx，slot，把session发送到sessions里。

关于pkcs11，还有一点可说的，就是SoftHSM库，它是一个模拟硬件实现的pkcsll，对应到的系统动态库可参看impl.go中FindPKCS11Lib测试函数中所涉及的，如Linux下的libsofthsm2.so。现阶段是没有安全硬件模块可以配合测试的，所以只有使用SoftHSM模拟测试，将libsofthsm2.so导入pkcs11对象。

## BCCSP工厂

对应两种bccsp实现，这里也有两种bccsp工厂：**pkcs11factory.go**和**swfactory.go**。fabric中某一模块一旦涉及工厂factory，则说明该模块基本就是由工厂提供“窗口函数”，供其他模块调用。这里以swfactory为例进行讲解。

**目录结构**：

* /fabric/bccsp/factory/   
  + factory.go - 声明了默认bccsp实例**defaultBCCSP**，bccsp实例存储映射**bccspMap**等全局变量和这些变量的获取函数**GetXXX**，定义bccsp工厂接口。
  + nopkcs11.go/pkcs11.go - 定义了两种版本的工厂选项**FactoryOpts**，初始化工厂函数InitFactories和获取指定bccsp实例函数GetBCCSPFromOpts。nopkcs11是默认版本，可条件编译指定使用哪种版本（编译时加入nopkcs11或!nopkcs11选项）。两种版本的差异集中在是否使用pkcs11上。
  + opts.go - 定义了默认的工厂选项**DefaultOpts**。
  + pkcs11factory.go - pkcs11类型的bccsp工厂实现**PKCS11Factory**。
  + swfactory.go - sw类型的bccsp工厂实现**SWFactory**。还定义了sw版本的bccsp选项。

**swfactory接口和实现**：

//在factory.go中定义

//接口

type BCCSPFactory interface {

//返回工厂的名字

Name() string

//返回符合工厂选项opts的bccsp实例

Get(opts \*FactoryOpts) (bccsp.BCCSP, error)

}

//在swfactory.go中定义

//实现

type SWFactory struct{}

func (f \*SWFactory) Name() string {

return SoftwareBasedFactoryName

}

func (f \*SWFactory) Get(config \*FactoryOpts) (bccsp.BCCSP, error) { ... }

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

实现的代码本身比较简单，Get最终是调用的sw的专用生成函数New来生成符合opts的bccsp实例。Name则是直接返回一个”SW”常量。

在每个chaincode例子中，如/fabric/examples/e2e\_cli/examples/chaincode/go/chaincode\_example02/chaincode\_example02.go，都使用了chaincode垫片shim中的Start函数。不知道在fabirc源码解析7——peer的ChaincodeSupport服务文中是否说过，在此说一下，chaincode的“垫片”shim核心代码集中在/fabric/core/chaincode/shim中，该垫片所“承垫”的是与各个结点通信的任务，也即ChaincodeSupport服务。chaincode形成的通信的信息，通过shim分发到各个结点，然后shim负责从各个结点收集信息，汇总返回给chaincode，完成chaincode的功能。其中shim的Start函数就是用来启动一个chaincode，定义在/fabric/core/chaincode/shim/chaincode.go中。在Start的函数中，就调用了err := factory.InitFactories(&factory.DefaultOpts)来初始化一个默认的bccsp工厂，在此可以知道，这里使用的默认工厂选项（参看opts.go），也就是使用的swfactory。

# fabric源码解析14——peer的gossip服务之初始化

自《fabirc源码解析11——peer的Admin和Endorser服务》之后，饶了一个圈，去讲了peer的msp和bccsp服务，现在我们再回到peer的**start.go**中的serve函数，继续往下讲。在11中讲了Admin和Endorser两个服务的注册，接着，serve开始初始化**gossip服务**：service.InitGossipService(...)。

gossip本意是绯闻，流言蜚语，闲谈聊天的意思，上大学的时候看过一部美剧叫《gossip girls》。而在这里，gossip代表了一种**可最终达到一致**的算法，其灵感来源于办公室八卦：当一个八卦在办公室出现时，在一定阶段内通过散播（dissemination），所有人最终都会知道这个八卦。这样就比较容易理解了，比如peer经过背书签名，将一个有效的交易最终提交。这份交易的写集合是A减100，B加100，因为网络中所有的结点都存有一份账本，因此该交易提交后，在有限的时间内，每个**分布**在网络中的结点中的账本都会应用这个交易，将自己的账本中的A减去100，B加上100。或者，有新结点加入网络中后，经过一定的时间，该新结点也会存储和其他结点一样的账本数据。这里需要注意是，最终一致的另外的含义就是，**不保证同时达到一致**，也就是在某一指定的时刻，每个结点的账本（也就是状态）不保证一致。同时，gossip不要求节点知道所有其他节点，因此具有**去中心化**的特性，节点之间完全对等，不需要任何的中心节点，这点也是区域链的显著特征。gossip是区域链相当核心的模块了，值得好好学习一下。

gossip中有三种基本的操作，如下：

* push - A节点将数据(key,value,version)及对应的版本号推送给B节点，B节点更新A中比自己新的数据
* pull - A仅将数据key,version推送给B，B将本地比A新的数据（Key,value,version）推送给A，A更新本地
* push/pull - 与pull类似，只是多了一步，A再将本地比B新的数据推送给B，B更新本地

## gossip数据传播协议

***该章节内容翻译子官方文档***。

Fabric通过划分各个执行交易的（背书和提交）peer和ordering结点之间的工作负载来优化区域链网络的执行，安全和可测量性。网络操作的解耦要求一个安全的，可信赖的，可测量的数据传播协议，以保证数据的完整性和机密性。为了达到这些要求，Fabric实现了一个gossip数据传播协议。

### gossip协议

peer结点“撬动”gossip以可测量的方式去**广播**（broadcast）账本和频道数据。gossip消息传送是是持续的，而且在频道中的每个peer不间断的从其它的peer那里接收当前的和一贯的（也就是格式等前后一致）账本数据（ledger data）。每个传播的消息都被签名过，因此“拜占庭的参与者”发送虚假的消息会很容易被识别，把消息发到消息不想到达的目标的分发行为会被阻止。peer会被延迟，网络参与者或者其他造成block丢失的原因所影响，但这些丢失block的peer最终将通过联系持有这些丢失block的peer异步更新到当前账本状态。

以gossip为基础的数据传播协议在Fabric网络上执行三个基础的功能：

1. 通过持续性的识别有效成员peer和检测那些已经下线的peer，管理peer的发现（discovery）和频道成员关系。
2. 在频道上所有的peer之间传播账本数据。任何持有与频道其他peer结点不同步的数据的peer识别丢失的block并通过拷贝正确的数据来同步自身。
3. 通过允许账本数据以peer点对peer点（peer-to-peer）状态传输更新的方式，提高新加入网络的peer结点的同步速度。

以gossip为基础的广播操作是通过peer从频道中其他peer中接收消息，然后把这些消息传送到频道上一定数量随机选择的peer结点，这个数量是可配置常量。peer结点也能运用一个pull机制，而不是等待一个消息的投递。这个循环重复着，伴随着频道成员关系的结果，账本和状态信息持续保持最新且同步。对于新block的传播，在频道中的领导peer（the leader peer）从ordering服务pull数据并初始化gossip到各个peer的传播。

### gossip消息传送

在线的peer结点通过持续的广播“alive”信息来（向leader或其他结点）指示其自身的有效性，每条消息中都包含PKI\_ID（the public key infrastructure ID）和发送者的签名。每个peer结点也通过收集这些“alive”消息，来维护自身的频道成员关系（channel membership）。如果没有任何一个peer接收到某一特定的peer的“alive”消息，则这个“dead”peer最终会从频道成员关系中被清除。因为“alive”消息都是加密签名了的，所以恶意的peer会因缺少由root CA认证的签名匙（signing key）而不可能冒充其他正确的peer。

除了自动传输接收的消息（即散播dissemination），一个状态调节进程（state reconciliation process）会通过每个频道上的众多peer结点来与世界状态（world state）同步。每个peer持续性的从频道上的其他peer那里pull来block数据，目的在于，如果（通过与自己的block数据对比）存在差异则修复自身的状态。因为**固定的连接**（fixed connectivity）不被要求去维护以gossip为基础的数据散播，因此这个进程会可靠的提供私密的和完整的数据到共享的账本，同时包括了对错误结点的容错度。（这里的**固定的连接**应该这么理解：在网络中没有发生变化的结点集合，比如A，B，C，D四个结点一直没有发生变化，因而四个结点之间的关系也不会发生变化，因此这四个结点之间就不需要去进行gossip散播消息数据。比如一个新加入的E点是与D点发生关系，则只需要D去向E散播消息，而A，B，C，D四者之间仍是不需要互相进行gossip散播的。）

因为多个频道之间是被相互隔离的，所有在一个频道上的peer点不能向其他频道发送消息或分享信息。虽然任一peer都可以属于多个频道，但是依照应用的**消息线路选择策略**（message routing policies），分配的消息传送禁止把block数据散播到其他频道的peer结点，这里的消息线路选择策略是以peer的频道订阅为基础的。（关于频道订阅，参考出版-订阅消息系统，即一个peer能够接收一个频道中的消息，必须先订阅这个频道的消息。）

注意：

1. 点对点（point-to-point）消息的安全性由peer的TLS层来处理，不需要签名。peer结点凭借它们自身的证书获得认证，这些证书由一个CA分配。虽然TLS证书也被使用，但是在gossip层是该peer点的证书被验证授权（而不是TLS的证书）。账本的block由ordering服务签名，然后投递到频道中的leader peer。
2. 认证是由peer的MSP对象管理的。当一个peer第一次连接到频道上，TLS会话（session）同成员身份绑定。这主要是使用在网络和频道中的成员关系去认证每个与新的peer发生的连接。

## gossip源码结构

* /fabric/gossip/   
  + api - 消息加密服务接口，与peer衔接的接口   
    - crypto.go - MessageCryptoService接口定义，用于消息加密
    - channel.go - SecurityAdvisor接口定义，用于安全辅助
  + comm - Comm模块，供gsssip调用，实现点对点的通信
  + common - 公用函数，定义，结构体
  + filter - 过滤模块，用于选择一个信息是否应该发送给一个网络成员
  + discovery - 发现模块，用于发现网络中的有效结点，供gossip调用
  + identity - 身份映射模块，用于PKI-ID与identity之间的映射，供service调用
  + election - 选举模块，用于选举领导peer，供service调用
  + state - state模块，供service调用
  + util - 公用工具文件夹，提供工具函数
  + gossip - 定义了gossip接口，实现goosip服务，供integration调用   
    - algo - 算法，PullEngine对象，供pull调用
    - pull - Mediator对象，供channel调用
    - msgstore - MessageStore对象，供channel调用
    - channel - GossipChannel对象，供channelState实例管理
    - chanstate.go - channelState对象，管理GossipChannel，供gossip实例调用
    - gossip.go - 接口定义
    - gossipimpl.go - 接口实现
    - …
  + integration - 整合实现在获取配置基础上的生成连入grpc的gossip实例的功能，供service调用
  + service - gossip服务器，封装了gossip服务，状态，分发模块等，与核心代码衔接
* —————————————————————–
* /fabric/peer/gossip/   
  + mcs.go - MessageCryptoService接口实现
  + sa.go - SecurityAdvisor接口的实现
* —————————————————————–
* /fabric/protos/gossip/ - 原型   
  + message.pb.go/message.proto - 定义gossip处理的消息的原型，grpc服务原型
  + extensions.go - gossip原型方法的拓展，用于辅助gossip服务处理消息
* —————————————————————–
* /fabric/core/deliverservice/ - 与gossip服务衔接的核心分发消息的代码

由上述结构可知：

1. gossip实例的专用生成函数供integration调用，integration供service调用，service供peer node start命令调用。
2. peer程序 –> proto实现的MessageCryptoService接口 –> gossip –> service模块实例 –> fabric核心代码 –> fabric其他模块。
3. fabric的gossip服务有多个主体和模块组成，如MessageCryptoService，discovery，filter，Gossip，GossipService。
4. deliverservice中是与gossip服务相关的核心代码，提供核心的消息分发客户端等供ordering等服务，从而也就衔接了gossip服务与ordering等服务。但是消息的散播，最终还是回归到gossip下，使用gossip服务的功能代码。

fabric中不直接使用**gossip服务**（/fabric/gossip/gossip/gossip\_impl.go中定义的gossipServiceImpl），而是使用**gossip服务器**（/fabric/gossip/service/gossip\_service.go中定义的gossipServiceImpl），注意gossip服务和gossip服务器是两个不同的对象，后者管理并拥有前者，并为peer提供调用的窗口。在peer node start命令的serve函数中，service.InitGossipService(...)初始化了gossip服务器，初始化后这个实例被存储在/fabric/gossip/service/gossip\_service.go中的**gossipServiceInstance**变量中。同时，gossip服务器管理着一个gossip服务作为其成员，此成员也被初始化。接下来分别详述gossip服务和gossip服务器两个对象的初始化。

## gossip服务的初始化

//gossip接口，在fabric/gossip/gossip/gossip.go中定义

type Gossip interface {

// 发送一个消息到远程的各个peer

Send(msg \*proto.GossipMessage, peers ...\*comm.RemotePeer)

//返回所有被认为是活的网络成员

Peers() []discovery.NetworkMember

//返回所有被认为是活的且订阅了ChainID频道的网络成员

PeersOfChannel(common.ChainID) []discovery.NetworkMember

//更新自身发现层的metadata

UpdateMetadata(metadata []byte)

//更新自身metadata，该metadata是peer出版给其他peer的自身频道相关的状态数据

UpdateChannelMetadata(metadata []byte, chainID common.ChainID)

发送一个消息给网络中的其他peer

Gossip(msg \*proto.GossipMessage)

//返回一个针对由其他匹配一个明确断言的结点发送的消息的专用只读频道

Accept(...) (<-chan \*proto.GossipMessage, <-chan proto.ReceivedMessage)

//使一个绯闻实例加入频道

JoinChan(joinMsg api.JoinChannelMessage, chainID common.ChainID)

//验证可疑peer点的身份，若发现该可疑peer是无效的，则关闭与此peer的连接

SuspectPeers(s api.PeerSuspector)

//停止这个gossip实例

Stop()

}

//gossip实例，在/fabric/gossip/gossip/gossipimpl.go中定义

type gossipServiceImpl struct {

selfIdentity api.PeerIdentityType //自身身份标识

includeIdentityPeriod time.Time

certStore \*certStore //certStore模块

idMapper identity.Mapper //idMapper模块

presumedDead chan common.PKIidType

disc discovery.Discovery //discovery模块

comm comm.Comm //comm模块

incTime time.Time

selfOrg api.OrgIdentityType //自身组织标识

\*comm.ChannelDeMultiplexer //多路分配器

logger \*logging.Logger

stopSignal \*sync.WaitGroup //等待组

conf \*Config //配置

toDieChan chan struct{} //停止通道

stopFlag int32 //停止标识

emitter batchingEmitter //emitter模块

discAdapter \*discoveryAdapter //discovery适配器

secAdvisor api.SecurityAdvisor //安全辅助

chanState \*channelState //chanState模块

disSecAdap \*discoverySecurityAdapter//discovery安全辅助适配器

mcs api.MessageCryptoService //消息加密服务

stateInfoMsgStore msgstore.MessageStore //消息存储模块

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48

## gossip服务初始化的前提元素

这些前提元素都是在peer start的serve函数中，既供gossip服务使用，也供gossip服务器使用，在此一并列出：

### 签名者serializedIdentity

在fabirc源码解析12——peer的MSP服务中最后部分提及的**localMsp**的初始化（自然也包括了其成员signer的初始化），而serializedIdentity,err = mgmt.GetLocalSigningIdentityOrPanic().Serialize()则是获取localMsp中的签名者signer。localMsp对象的初始化的数据来源于MSPConfig配置数据，该MSPConfig是随着InitCrypto->LoadLocalMsp->**GetLocalMSP().Setup(conf)**一步步包装出来的，最后形成**conf**传入Setup进行初始化。使用到的是core.yaml配置文件中的BCCSP，mspConfigPath，localMspId项，分别作为bccsp服务的配置，msp配置文件（指各种代表身份的证书路径等）路径，localMsp的ID。其中mspConfigPath中就包含用于初始化signer的身份证书。这个签名者，一方面提供了签名的能力，另一方面代表着peer的身份。

### 消息加密者服务messageCryptoService

MessageCryptoService接口实例，接口在/fabric/gossip/api/crypto.go，实现在/fabric/peer/gossip/mcs.go。该实例是gossip模块与peer的加密层之间的“合约”，被gossip模块用于确认和认证远程peer身份和这些peer发送的数据（加密解密），也被用于验证由ordering服务发来的block。本身包含频道策略管理者获取器，本地签名者，反序列化工具管理者，三者分别用peer.NewChannelPolicyManagerGetter()，localmsp.NewSigner()，mgmt.NewDeserializersManager()初始化，分别用于获取指定频道ID的策略管理者，本地签名者（假设本地MSP已被初始化）进行签名，提供各种反序列化peer发送过来的序列化数据的工具，分别在/fabric/core/peer/peer.go，/fabric/common/localmsp/signer.go，/fabric/msp/mgmt/deserializer.go中实现。该元素生成后，通过参数渗透到了gossip服务自身的多个模块中，如成员idMapper，disSecAdap。

### 安全顾问secAdv

SecurityAdvisor接口实例，接口在/fabric/gossip/api/channel.go，实现在/fabric/peer/gossip/sa.go中。为了系统安全，让MSP能够更新是十分基础的，通过由ordering服务分配的配置交易，可以更新频道的MSP。安全顾问接口就是为了这点，提供安全和身份相关（如身份验证识别等）的能力。该元素也和messageCryptoService和idMapper模块（参看下文章节3.4）一样，生成后通过参数渗透到了gossip服务自身的多个模块中。

### 其他元素

bootstrap，gossip服务引导的地址。secureDialOpts，grpc服务拨号的标准选项。peerEndpoint.Address，使用的是core.yaml中的peer.address项，peer本地的IP地址。peerServer，peer结点的grpc服务器对象。

在InitGossipService中，除了用上述传递进来的参数用于给自身成员赋值，也使用其他模块中的函数给剩余的成员赋值，如使用integration.NewGossipComponent给重要的成员gossipSvc赋值，而这个函数是专门用来整合gossip实例和gossip在core.yaml中的配置的函数。

## 消息类型分类

说到底，gossip服务是处理消息的，每种类型的消息有不同的用途，gossip服务也使用不同模块处理不同类型的消息。gossip服务所处理的消息类型原型在/fabric/protos/gossip/message.proto中定义，对应生成message.pb.go。gossip中传播（发送）的消息以**GossipMessage**形式传递，而不同类型的具体的消息数据存放在GossipMessage中的**Content成员**中。

### 辅助类消息

之所以叫辅助类消息，是这类消息不承担具体传送传播数据的任务，而是辅助性的：

* Empty - 空消息，用于结点间的Ping（来测试结点是否连通）和测试。
* ConnEstablish - 用于gossip之间的握手，即任何时候一个peer想与另一个peer连接通信，都需要先发送这个消息以证明其身份。

### pull机制消息

用于pull机制中处理的消息类型。所谓pull机制，就是一个结点主动向其他结点索要自己需要的消息（数据），pull进（即拉进，拽进的意思）自己的结点中处理或存储，整个pull过程需要四步，可参看下文章节3.5。从**内容**上可以分为未定义消息，块消息，身份消息（即pull进来的可以是以块数据为内容的消息，也可以是身份数据为内容的消息，下述4种消息中都有一个成员MsgType来表明这个消息中所携带的数据内容），但从**pull步骤**上分为四种：

* GossipHello - hello消息
* DataDigest - 消息摘要
* DataRequest - 摘要请求
* DataUpdate - 摘要应答

### state消息

和**状态**有关的消息。这里的状态指的是chain的数据状态，如一个结点所存储的块数据是否一致，可以说这里所谓的**状态**接近**数据**的意思。

* DataMessage - 数据消息
* StateInfo - 状态消息
* StateInfoSnapshot - 状态快照消息（即一组StateInfo消息）
* StateInfoPullRequest - 状态请求消息（索要StateInfoSnapshot消息）
* RemoteStateRequest - 远程单点状态请求消息
* RemoteStateResponse - 远程单点状态应答消息

### 关系消息

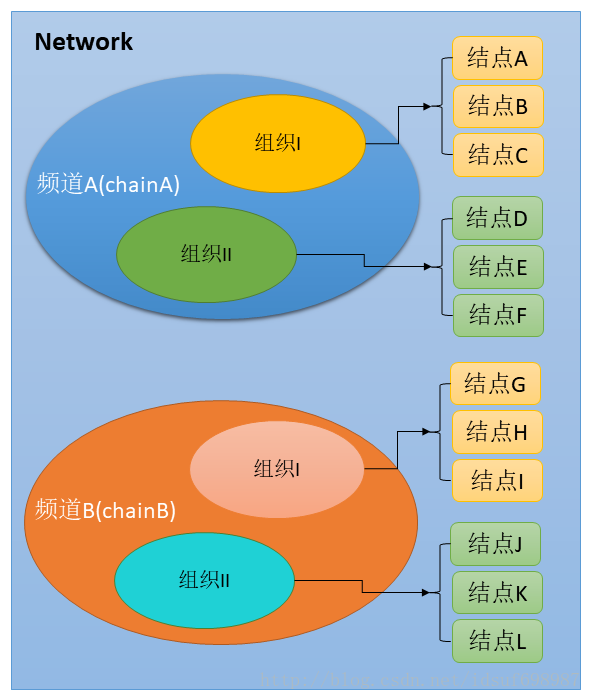
与频道成员身份，关系，存续相关的消息类型。

* AliveMessage - alive消息
* MembershipRequest - 成员关系请求消息
* MembershipResponse -成员关系应答消息
* LeadershipMessage - leader消息
* PeerIdentity - 身份消息

以上只是为了方便理解和辨识，所以依照消息的作用和处理所依赖的模块的不同进行人为的分类。在实际的代码中，模块处理的消息类型有所交叉，如关系消息，前3种主要由discovery模块处理，PeerIdentity由certStore模块处理，LeadershipMessage却又由chanState模块处理。而且，消息的分类其实有多个标准或维度，以上是按照GossipMessage中的**Content**所包含的消息类型进行分类的，也可以按GossipMessage中**Tag成员**对消息进行分类，Tag成员的值决定了一个消息可以在哪种范围内传播，有四种值：

* UNDEFINED - 未定义
* EMPTY - 空，对应上文章节2.1中的EMPTY消息
* ORG\_ONLY - 只在同一组织内传播
* CHAN\_ONLY - 只在同一频道内传播
* CHAN\_AND\_ORG - 同时在同一频道和同一组织内传播
* CHAN\_OR\_ORG - 既可以在同一频道内，也可以在统一组织内传播

图示举例CHAN和ORG范围如下，一个网络中，有A，B两个（或更多）频道（也可以说是chainA，chainB），每个频道包含两个（或更多）组织：组织I，组织II。每个组织I又各包含三个（或更多）结点。



## gossip服务中主要模块的初始化

gossip服务的实现个人感觉还是挺复杂的，这里面牵涉到了利用grpc通信，chan通道控制，类“出版-订阅”消息系统，各个模块，模块的“适配器”，这些元素相互配合，共同实现gossip服务。该章节文章中目录除绝对路径外，都以**/fabric/gossip/**为基础。gossip服务的各个模块都作为成员存在于gossipServiceImpl实例中，并在gossipServiceImpl专用初始化函数NewGossipService中初始化。

在初始化的过程中，要注意“倒钩”的使用。这也算fabric中常用的一招了，宿主将**倒钩**以参数的形式传入一个成员对象，成员对象在实现功能A的时候调用了这个倒钩，然后宿主使用这个成员对象的A功能时，实质还是使用自身的功能，但又可以让成员对象发挥一些作用来帮助宿主，这些作用一般都是宿主自身无暇顾及或者不屑于干的“小事儿”。

### discovery模块

在NewGossipService中使用g.disc = discovery.NewDiscoveryService(...)初始化**discovery模块**，代码集中在discovery目录。可以简单的将discovery模块当作一个**扫描器**，扫描并辨别管理频道或网络中存在的活着的（alive）结点和死掉的（即不再存在于网络或频道中，掉线）结点。为了实现这样的功能，discovery模块使用handleMsgFromComm处理三类信息（对应三个if分支，很清晰）：alive消息**AliveMessage**，成员关系请求消息**MembershipRequest**，成员关系应答消息**MembershipResponse**。discovery模块中有一个来自gossip实例的“适配器”discoveryAdapter（在gossip/gossip\_impl.go中定义）的成员，这个适配器成员封装了**comm模块**（参看下文章节2.3），使得discovery模块利用这个适配器可以处理这三类消息。

discovery模块判断一个结点是否活着的机制是：根据最后一次收到一个结点的alive消息的时间与当前时间的**差值**，当这个差值大于某个值，则判定该结点已经死掉。因此discovery模块判定的死掉的结点，只是这个模块**自认为**这个结点死掉了，也有可能这个结点其实并没有死，只是alive消息没有及时发送而已。discovery实例中有成员aliveLastTS，该映射即存储了每个peer结点最后一次所发送的alive消息的时间。对于判定死掉的peer点，discovery则将该peer点死掉的时间存储到deadLastTS成员中，并定期发送复活请求去尝试复活这些死掉的结点。另外，相应的，discovery把活着的和死掉的peer点信息分别存储在成员aliveMembership，deadMembership中。

### emitter模块

**emitter模块**的原型是batchingEmitter，在gossip/batcher.go中定义，可以把emitter模块理解为**批量发射器**，即在gossip服务传播的过程中，为了提高效率，不是来一条信息就传播一条，而是累计到一定条数，然后一批一批的发送。一批消息的个数由batchingEmitter的成员burstSize决定，gossip服务中默认的是10条为一批，存储在成员buff中。成员iterations指定了每条消息发送的次数，默认是1次。成员delay指定了每批消息发送的时间间隔，默认是10ms。在NewGossipService中使用g.emitter = newBatchingEmitter(...)对此模块进行初始化。

emitter模块在发送信息过程中，自身主要起到一个控制批次的作用，而发送的功能还是使用其宿主gossip实例投放的“倒钩”cb，这个emitter成员在初始化时被赋值为g.sendGossipBatch，即gossip实例的函数sendGossipBatch。

### comm模块

**comm模块**代码集中在comm目录中，实现了结点与结点之间的通信，即实现了**grpc通信服务**。比较特殊的是comm既是grpc的客户端，也是服务器端。首先comm实现了/fabric/protos/gossip/中定义的rpc服务**GossipStream**和**Ping**。其次，comm也实现了用于客户端发送消息的功能函数**Send**。comm模块的专用初始化函数有两个：NewCommInstanceWithServer和NewCommInstance，前者是comm自己创建一个grpc.Server，后者是由参数传入一个grpc.Server作为自己的自己的服务器成员**gSrv**，gossip服务默认使用的是后者，传入的是peer node start命令中serve函数中初始化的peer结点的peerServer。在专用初始化函数NewCommInstance中用proto.RegisterGossipServer(s, commInst.(\*commImpl))完成了comm模块**服务器的grpc注册**。comm模块中包含两个主要的子模块：connStore模块和msgPublisher模块，都在NewCommInstance中初始化。

#### connStore模块

**connStore模块**原型为connectionStore，在comm/conn.go中定义，可以将其理解为peer点间的**连接管理器**，管理的**连接**原型是connection，也在同一文件中定义。所有的connection都存储在connectionStore的成员pki2Conn中。既然是管理连接，自然的，主要的方法有创建connection，获取connection。而创建连接的方法又是comm模块所放的“倒钩”：在connectionStore的专用生成函数newConnStore中，用于创建新connection的成员**connFactory**被赋值为comm实例本身，而comm实例所实现的createConnection函数就是用于生成一个\*定制的新的\*\*connection。

connection代表了一个peer与另一个peer之间的**grpc连接**，这个连接可能是客户端流连接，也可能是服务端流连接，具体要看connection专用生成函数newConnection中对成员clientStream赋值，还是对成员serverStream赋值。因为connection封装了grpc的流接口，因此在gossip服务散播消息的过程中，消息经comm模块，最终是使用使用serviceConnection来**收发**和**处理**消息。**收发消息**方面，收消息用readFromStream，发消息用writeToStream，connection的send函数也是用于发送消息，但是这个函数只是辅助writeToStream的：send将消息直接丢到成员outBuff中，outBuff是一个通道，所发送的消息经这个通道到达writeToStream中，进而使用stream.Send发送消息（即用grpc的流对象发送消息）。**处理消息**方面（即，处理接收到的消息），使用connection成员handler，这又是comm模块放的“倒钩”：在定制connection的createConnection函数（参看上一段）中，handler被赋值为使用msgPublisher模块的DeMultiplex函数（参看下文章节2.3.2）去处理消息，这里将接收到的消息封装进ReceivedMessageImpl对象（接口原型在/fabric/protos/gossip/extensions.go中定义，在comm/msg.go中实现），同时也封装了所建的connection信息，以供之后向发送者回复消息所用。最后需要注意的是，connection成员pkiID代表的是**对方peer点**的信息，一个peer点的客户端流connection和服务端流connection是两个单独的连接存在于connStore中并接受其管理。

#### msgPublisher模块

**msgPublisher模块**原型为ChannelDeMultiplexer，在comm/demux.go中定义，可以将其理解为是一个以频道chan为基础的类似“出版-订阅”的模块，所有的频道channel存储在成员channels中。主要有两个方法：添加chan函数AddChannel，广播消息函数DeMultiplex。AddChannel很容易理解，DeMultiplex做的是遍历channels，按每个channel所定的条件（这个条件就是消息选择器）发送消息。

频道channel定义在同一个文件中，成员包含一个go语言的chan和一个消息选择器。这个消息选择器的原型为common.MessageAcceptor，用于甄选消息，即msgPublisher在通过成员channels中的一个个channel广播消息时，channel中的消息选择器会判断是否对这个消息感兴趣，如果感兴趣，则通过这个channel发送消息，如果不感兴趣，则忽略这个消息。这个消息选择器是gossip实例通过comm模块所放的“倒钩”：在gossip/gossip\_impl.go中的start函数中，首先，调用incMsgs := g.comm.Accept(msgSelector),既新建了一个channel并将消息选择器赋值为msgSelector，又把这个新的channel注册到msgPublisher中的channels并返回这个频道（即incMsgs）。接着，调用go g.acceptMessages(incMsgs)，新启一个goroutine接收incMsgs这个频道所进来的消息（acceptMessages就是gossip服务接收消息的函数，参看下文章节3.2，go g.acceptMessages）。对照章节2.3.1中connection中处理接收到的消息的handler所描述的：使用msgPublisher模块的DeMultiplex函数去处理消息，即遍历所有频道，把消息传递给消息选择器感兴趣的频道，最后又通过频道把消息交给了gossip服务的acceptMessages函数去处理。

### idMapper模块

**idMapper模块**原型为identityMapperImpl，在identity/identity.go中定义，可以将其理解为一个网络中peer点的身份管理者。在gossip服务中，peer点有两种表示自身身份的类型：PeerIdentityType和PKIidType。PeerIdentityType是peer点的所持有的证书，值来自于serializedIdentity（参看上文章节1.1），PKIidType可以简单的理解为是从PeerIdentityType中摘取出一部分摘要，然后把摘要经哈希处理后生成的叫做PKI-ID的身份，具体可参看/fabric/peer/gossip/mcs.go中messageCryptoService的实现中的GetPKIidOfCert函数。idMapper模块所管理的就是PKIidType与PeerIdentityType之间的映射，即成员pkiID2Cert，同时也就为gossip服务中身份和消息的认证，签名提供了基础。如idMapper被封装在gossip实例成员discoverySecurityAdapter（也是一种安全辅助适配器，在gossip/gossip\_impl.go中定义）中，该成员实现了ValidateAliveMsg函数。在初始化discovery模块时，成员crypt被赋值为discoverySecurityAdapter，在discovery模块处理接收到的alive消息时（即handleMsgFromComm，参看上文章节2.1），就调用了crypt的ValidateAliveMsg函数，即提供了验证接收到的alive消息（和消息中包含的身份）的功能。

idMapper还被封装在其他模块中，如comm模块，certStore模块，但是包括gossip实例中的idMapper在内，这些idMappper均来自于gossip服务器。gossip服务器实例初始化时，创建了一个idMapper对象，并将这个对象以参数的形式传入gossip服务实例，进而传入gossip服务的各个模块。因此，在各个模块中对idMappper中数据的增减，都会体现到gossip服务器实例中的那个成员idMapper，而gossip服务器实例对成员idMapper的操作，也会渗透体现到各个模块中的idMapper。

### Mediator模块

**Mediator模块**原型为pullMediatorImpl，在gossip/pullstore.go中定义，可以将其理解为一个调解媒介，在每个peer结点中，有些存储的信息（如成员身份，block等信息）是存在呼应关系的，比如网络中有一个C结点，那么A和B存储的成员信息中肯定都应该有C的身份信息，但是有时候由于一些原因会造成B中有C而A中没有C的差异情况，这个时候可以通过Mediator模块调解“矛盾”，通过A向B索要关于C的身份信息而弥补差异。该模块所处理的是**pull机制消息**，身份消息由certStore模块（参看下文章节2.6）处理，块消息由chanState模块处理（参看下文章节2.7）；hello消息，消息摘要，摘要请求，摘要应答分别对应Mediator模块处理消息的函数HandleMessage中的四个if分支。

Mediator模块是certStore模块和chanState模块的成员，在初始化这两个宿主时，Mediator模块同时也被初始化。以certStore模块为例，在初始化certStore模块时，在gossip/gossip\_impl.go中，调用createCertStorePuller()创建了一个pullMediatorImpl实例：创建了配置项conf和适配器adapter后传入pullMediatorImpl的专用生成函数NewPullMediator中。注意这里的conf中的MsgType被赋值为**PullMsgType\_IDENTITY\_MSG**，即身份消息类型，也即表明这里所生成的Mediator对象是用于pull进身份类型消息的，适配器adapter的作用是封装一些gossip模块和处理身份消息的函数供Mediator模块使用。在NewPullMediator中，又调用了p.engine = algo.NewPullEngineWithFilter(...)生成了一个Mediator模块的一个重要成员**engine**。

engine是Mediator模块的**pull引擎**，周期性的做了具体的pull工作，原型为PullEngine，在gossip/algo/pull.go中定义。这个引擎伴随着engine的初始化启动起来：在NewPullEngineWithFilter(…)中，新启了一个goroutine周期性的运行engine.initiatePull()，周期间隔为4s，由core.yaml中的peer.gossip.pullInterval项指定。这里要注意，engine初始化时，把宿主pullMediatorImpl作为自己成员PullAdapter的值，以便利用宿主pullMediatorImpl所包含的适配器（即上一段所提的adapter）的能力发送消息（如下文所提的Hello()函数）。PullAdpter也是engine的一个适配器，在Mediator中，包括其他模块中，都有较多的适配器，适配器的作用就如它的名字，起到适配的作用，比如这里的Mediator模块，certStore模块也想用它，chanState模块也想用它，但是在实现细节上和所处理的消息类型上都有区别，因此Mediator的实现是固定的，Mediator拥有一个适配器成员（虽然是成员，但是我觉得一个对象的适配器和这个对象应该是平级的关系），通过这个适配器，既可以实现Mediator使用certStore，chanState两个不同模块各自所拥有的功能，也可以实现certStore，chanState两个不同模块使用Mediator处理不同数据的功能。

engine引擎的执行有四步（假设有A，B两个结点，请对看gossip/algo/pull.go中开始处的注释+码图）：

1. A的engine调用Hello()发送**hello消息**给B，B的engine接收到了hello消息后调用OnHello()处理。
2. B在OnHello()中处理**hello消息**后调用SendDigest()返回**消息摘要**给A，A接收**消息摘要**调用OnDigest()处理。
3. A发送hello消息后等待一段时间接着调用processIncomingDigests()，进而调用SendReq()发送**摘要请求**给B，B接收后调用OnReq()处理。
4. B在OnReq()中处理消息后调用SendRes()返回**摘要应答**给A，A接收消息调用OnRes()处理。
5. 在3中A调用SendReq()发送**摘要请求**后等待一段时间接着调用endPull()结束整个pull过程，等待下一次调用initiatePull()。

engine引擎执行的过程中有两个关键词，一个是**摘要**，一个是**NONCE**。**摘要**指从消息中择取出的足以代表一条消息的关键信息，**身份消息**的摘要即为**PKI-ID**，**块消息**的摘要则为**块序号**，在接收到**摘要应答**后，调用OnRes()将摘要最终存储到engine成员state中，这些摘要来自于Mediator模块处理消息的函数HandleMessage中处理**摘要应答**的分支中itemIDs[i] = p.IdExtractor(msg)，即使用的是Mediator模块的适配器中封装的IdExtractor函数，也因此，由于certStore模块和chanState模块所实现的适配器中的IdExtractor不一样，所有才能从消息中分离出两种不同的摘要。**NONCE**是通信安全中的一个概念，指的是用一次即废弃的一个整数，目的在于防止replay attack，但是在这里只是简单使用NONCE一次性的性质，在engine引擎周期性执行pull的过程中，每个周期中的步骤之间不产生交叉混淆。

### certStore模块

**certStore模块**原型为certStore，在gossip/certstore.go中定义，可以将其理解为一个身份存储器，辅助性的维护idMapper模块，即通过处理身份消息，进而维护idMapper模块中存储的身份数据。certStore模块自身不提供什么功能，主要是提供窗口供gossip服务实例调用，其实现的功能基本依赖其封装的其他模块，如验证身份消息的函数validateIdentityMsg使用MessageCryptoService，返回废弃结点的函数listRevokedPeers使用idMapper，处理消息handleMessage则使用Mediator模块。certStore模块只处理pull机制消息中以身份数据为内容的消息，这点可以从gossip/gossip\_impl.go中handleMessage函数的if msg.IsPullMsg() && msg.GetPullMsgType() == proto.PullMsgType\_IDENTITY\_MSG分支看出。

### chanState模块

**chanState模块**原型为channelState，在gossip/chanstate.go中定义，和certStore模块一样，chanState本身作为管理角色并向gossip服务实例提供操作窗口，同时为管理的channel模块提供适配器gossipAdapterImpl，而任务性、功能性部分都由其管理的channel模块实现。这里的状态可以理解为数据的意思，即chanState模块主要按频道处理数据类型的消息（参看上文章节2.3）。chanState模块将管理的channel对象存储在成员channels中，映射的key为字符串格式的chainID，即频道的ID。chanState模块的专用生成函数是在gossip/gossip\_impl.go中的newChannelState(...)，而且既然是管理channel模块的，自然有**增加**，**获取**功能：**增加**为joinChannel函数，**获取**为lookupChannelForMsg函数。

channel模块原型为gossipChannel，在gossip/channel/channel.go中定义，是具体处理chanState模块任务的对象。channel模块中的成员Adapter原型为gossipAdapterImpl，即由chanState提供的适配器，适配器中封装了gossip服务实例和discovery模块，为channel提供了收发消息的能力。成员blockMsgStore，stateInfoMsgStore，leaderMsgStore三个存储成员为channel提供了存储能力。成员blocksPuller为channel提供了pull块消息的能力（参看上文章节3.5）。在channel专用生成函数中NewGossipChannel中，启动了两个go gc.periodicalInvocation(...)goroutine，周期性的执行gc.publishStateInfo和gc.requestStateInfo函数：

* **gc.publishStateInfo** - 周期性的传播频道的状态消息，该状态消息即为channel的成员stateInfoMsg，代表着频道当前的数据状态。周期间隔为4s，由core.yaml的peer.gossip.publishStateInfoInterval指定。这个散播的过程由成员shouldGossipStateInfo的值和函数UpdateStateInfo相互配合，每隔4s调用publishStateInfo函数，在其中检查shouldGossipStateInfo的值，如果是0，则表示频道当前的stateInfoMsg没有更新（之前的已经散播过），不用散播。当调用UpdateStateInfo函数更新stateInfoMsg后，也将shouldGossipStateInfo的值赋为1，当下一次调用publishStateInfo函数检查shouldGossipStateInfo的值时，发现为1，则调用适配器Adapter的gc.Gossip(stateInfoMsg)将stateInfoMsg散播出去，并将shouldGossipStateInfo的值赋为0，等待下次UpdateStateInfo函数的更新。
* **gc.requestStateInfo** - 周期性的发送StateInfoPullRequest类型消息，向远程peer结点索要StateInfoSnapshot消息。在requestStateInfo中，每隔4s（由core.yaml中的peer.gossip.requestStateInfoInterval指定），由filter模块随机筛选出来（参看下文章节3.8）3个（由core.yaml中的peer.gossip.pullPeerNum指定）peer点，然后调用适配器Adapter的gc.Send(req, endpoints...)向这些peer点发送StateInfoPullRequest消息。等这些peer点返回StateInfoSnapshot消息时，则放入HandleMessage函数中处理。

channel模块最重要的一个函数就是处理接收消息的HandleMessage(msg proto.ReceivedMessage)，其余的基本都是配合该函数处理消息：要么是用于验证消息和消息发送者身份，要么是生成回复消息，要么是处理指定类型消息。HandleMessage函数十分清晰，一个个if分支对应处理**6种类型**的消息：DataMessage，StateInfo，StateInfoSnapshot，StateInfoPullRequest，pull机制消息中以块数据为内容的消息，LeadershipMessage。**基本的过程**就是验证消息和消息的发送者，如果验证后消息合法，则该存储的存储，该相应回复的回复，该转发的转发。**如**if m.IsStateInfoPullRequestMsg()分支是处理StateInfoPullRequest类型消息，处理过程就是调用createStateInfoSnapshot函数生成此结点的StateInfoSnapshot消息然后使用ReceivedMessage的接口函数msg.Respond原路回复。在createStateInfoSnapshot函数中还体现了一个**传播策略问题**，即传播范围选择问题（对看上文对于Tag成员的图例解释），对于发送StateInfoPullRequest类型消息的结点A，接收到该请求的结点B需要判断A是否与自己在同一个组织内，如果A是同一个组织内的结点，则B会把存储的StateInfo消息都回给A，如果A不是同一个组织的，则B只把存储的**非组织内消息**或者**既是组织内也是频道内的消息**回给A。**再如**if m.IsDataMsg() || m.IsStateInfoMsg()分支是处理DataMessage，StateInfo消息，处理过程就是将数据存储到blockMsgStore或stateInfoMsgStore中，然后继续调用gc.Gossip，gc.DeMultiplex，gc.blocksPuller.Add转发，广播，pull进此消息。

#### msgstore模块

**msgstore模块**为消息存储模块，为每一个channel模块提供**存储功能**，原型有**两种**：（1）**MessageStore**，在goosip/msgstore/msgs.go中定义，用于存储3种类型消息：DataMessage、pull机制消息中以块数据为内容的消息、LeadershipMessage，channel中的**成员blockMsgStore**就用于存储前两种，**成员leaderMsgStore**存储后一种。（2）**stateInfoCache**，在gossip/channel/channel.go中定义，用于存储StateInfo类型消息，channel中的成员**stateInfoMsgStore**就用于存储这种消息。

MessageStore把消息存储在成员messages中，撇开存储中的增加和获取的功能不讲，MessageStore在存储消息的过程中有一个消息**有效期**的概念：在MessageStore专用生成函数NewMessageStoreExpirable中，go store.expirationRoutine()新启了一个goroutine周期性的调用expireMessages()来检查messages中的消息并废止（即用以新换旧的方式删除）已经失效的消息。blockMsgStore的周期间隔为4s，leaderMsgStore的周期间隔为1s，分别由core.yaml中的peer.gossip.pullInterval\*100/100，peer.gossip.election.leaderAliveThreshold\*10/100计算指定（见MessageStore的成员msgTTL和函数expirationCheckInterval()）。关于消息的有效性，涉及到两点，即有效性的**判断标准**和**判断函数**，判断函数中体现判断标准。MessageStore的**成员pol**就是判断函数，原型MessageReplacingPolicy在common/common.go中定义，在/fabric/protos/gossip/extensions.go中实现，为msgComparator对象的invalidationPolicy(this,that)函数，这个函数接收两个参数：新消息this和已存在的消息that，返回三种验证结果（在common/common.go中定义）：MessageNoAction，MessageInvalidates，MessageInvalidated。MessageNoAction表示this与that相安无事，MessageInvalidates表示this使that无效，MessageInvalidated表示this被that无效（即that使this无效）。这三个结果的名字也起的挺有意思的，NoAction，主动动词Invalidates，被动动词Invalidated，巧妙的表示出了谁是无效的。在invalidationPolicy(this,that)函数中，每个if分支中判断一类消息，每种类型的消息的判断标准不尽相同，如if thisMsg.IsDataMsg() && thatMsg.IsDataMsg()分支调用dataInvalidationPolicy函数判断**DataMessage类型的this与that**，判断标准是：依据this的消息序号**this\_N**和that的消息序号**that\_N**，如果**this\_N == that\_N**，说明已经存储过一个同样的消息，则this无效；如果**this\_N-that\_N的值的绝对值 <= 100**，则this与that相安无事；如果**this\_N-that\_N的值的绝对值 > 100且this\_N > that\_N**，则that无效；其余情况this无效。这里为什么是100，涉及到ordering服务，将在ordering服务主题文章中详述。

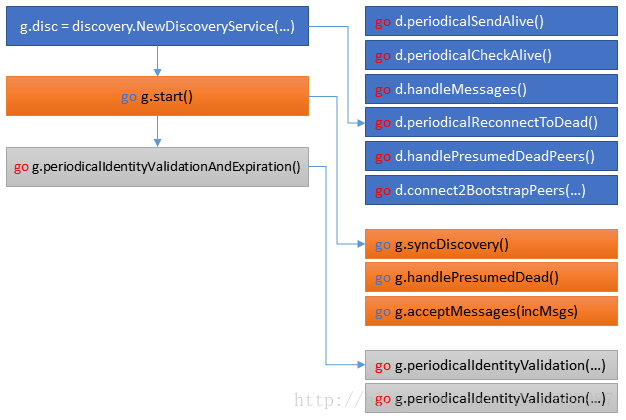
stateInfoCache封装了MembershipStore和MessageStore（参见上一段），StateInfo消息在这两个成员中各存一份，MembershipStore存储所有接收到的消息，MessageStore则只存储有效期内的消息。MembershipStore在util/msgs.go中定义，正常的进行增，删，获取。

### filter模块

**filter模块**相对简单，只实现了几个筛选函数和筛选策略原型，在filter/filter.go中定义，可以将其理解为一个成员过滤器，即按照给定的条件筛选出符合条件的成员。基础过滤策略的原型，其实就是定义的筛选函数，为type RoutingFilter func(discovery.NetworkMember) bool。筛选函数为func SelectPeers(...)[]\*comm.RemotePeer，接收三个参数：筛选个数k，成员集合peerPool，筛选策略filter，所做的就是在peerPool中随机筛选出k个符合filter的成员，放入数组中返回（如果筛选出来的成员个数<=k，则将筛选出了的成员全部返回，这时候可能返回的不足的成员不足k个）。辅助筛选函数CombineRoutingFilters(...)，接收多个RoutingFilter，然后通过for循环返回一个“合并”的RoutingFilter，比如现在要筛选出满足条件1，条件2的结点，相应实现的RoutingFilter是R1，R2，将R1和R2传入CombineRoutingFilters(...)，返回一个合并的RoutingFilter为R，R在循环中依次使用R1，R2判断传入的结点是否满足条件，然后给出“综合”结果。

## gossip服务初始化中所起的goroutine

gossip服务用于传播消息，但是这只是gossip服务明面上的功能，为了实现（或者说维护）这个功能，gossip服务**初始时**需要在背后启动一系列goroutine做一些工作，有些是准备性的工作，有些是周期性的工作，有些是循环性（即在服务存在期间，不间断）工作。**初始时**，指的就是一个gossip服务实例创建的时候，即在gossip/gossip\_impl.go中的gossip实例专用生成函数NewGossipService执行时。



### discovery模块goroutine

在创建discovery实例时，启动了6个goroutine，用于服务discovery模块的扫描处理工作。这6个goroutine所做的工作可与上文章节2.1内容对看。

* **go d.periodicalSendAlive()** - 周期性的发送alive消息，以告之网络中的其他结点：我还没死。间隔时间默认为5s，由core.ymal中的peer.gossip.aliveTimeInterval指定。
* **go d.periodicalCheckAlive()** - 周期性的检查aliveLastTS中的时间戳，判断结点是死是活，然后相应调整deadLastTS，aliveLastTS，aliveMembership，deadMembership中的信息。
* **go d.handleMessages()** - 循环处理来自discoveryAdapter成员**incChan频道**的消息，这个通道是gossip实例专门用来处理GossipMessage\_AliveMsg，GossipMessage\_MemReq，GossipMessage\_MemRes三类消息。incChan频道的消息又来源自gossip专用于处理接收消息的handleMessage函数（参看下文章节3.2，go g.acceptMessages）的if selectOnlyDiscoveryMessages(m)分支，这个分支中调用了g.forwardDiscoveryMsg(m)，将接收到的消息m发送到了incChan频道。
* **go d.periodicalReconnectToDead()** - 周期性的查新尝试连接死掉的结点，通过尝试向死掉的结点发送GossipMessage\_MemReq类型的消息来实现。
* **go d.handlePresumedDeadPeers()** - 循环处理来自discoveryAdapter成员**presumedDead频道**的消息，presumedDead频道在newDiscoveryAdapter()中初始化时即被赋值为gossip实例的成员g.presumedDead，而g.presumedDead是gossip专用于处理被认为是死掉的peer结点的频道（参看下文章节3.2，go g.handlePresumedDead）。
* **go d.connect2BootstrapPeers(bootstrapPeers)** - 这个属于准备性工作，整体只调用一次，即在discovery模块初始化时，向bootstrapPeers发送GossipMessage\_MemReq类型消息，索要成员关系信息。对于新加入的peer结点X实现的gossip服务，无论是散播消息，还是其他工作，都是基于现有网络成员信息的，也就是说我要先知道有哪些结点，才能与这些结点通信，收发消息。bootstrapPeers就是X指定的要去索要成员信息的地址列表，由core.yaml中的peer.gossip.bootstrap指定，默认只有127.0.0.1:7051一个值。这个过程最多会尝试索要120次，每隔25秒索要一次，分别由discovery/discovery\_impl.go中的maxConnectionAttempts常量和core.yaml中的peer.gossip.reconnectInterval指定，但一旦获取到了成员信息（即成员aliveLastTS中存在信息）则停止这个goroutine。

### start函数goroutine

go g.start()新启了一个goroutine去开始gossip实例的服务。在start函数中分别启动了三个goroutine：

* **go g.syncDiscovery()** - 周期性的在discovery模块的成员aliveMembership中随机选择n个peer结点，向其发送GossipMessage\_MemReq类型消息，索要成员关系信息，即在网络存在的结点之间进行成员关系信息的同步。时间间隔默认为4秒，n默认为3，分别由core.yaml中的peer.gossip.pullInterval，peer.gossip.pullPeerNum指定。这个goroutine主要调用了discovery/discovery\_impl.go中定义的InitiateSync函数，其中所选出的peer点的随机性由定义在util/misc.go中的GetRandomIndices函数实现，该函数接收两个参数c1和c2，随机选出c1个小于c2的数。选出peer后，即依次发送GossipMessage\_MemReq请求消息索要成员关系信息。
* **go g.handlePresumedDead()** - 循环接收从comm模块的deadEndpoints频道来的消息，然后转发到gossip实例的presumedDead频道（参看上文章节3.1，go d.handlePresumedDeadPeers）。即，这个goroutine专门用gossip实例来转发被认为死掉了的结点的身份信息，从comm模块接收后经适配器discoveryAdapter转交给discovery模块来处理。
* **go g.acceptMessages(incMsgs)** - 循环接收从msgPublisher模块订阅的频道incMsgs来的消息，交由g.handleMessage(msg)处理。可与上文章节2.3.2对看。这个goroutine就是用于gossip实例处理接受到的消息的。

### 身份验证终止goroutine

连续启动两个goroutine，用A和B表示，都执行g.periodicalIdentityValidation(...)函数。该函数接收两个参数：**猜想函数**（suspectFunc），**周期间隔**。也因这两个参数，A和B有两点区别：（1）根据**猜想函数**判断是否使用MessageCryptoService（参看上文章节1.2）去验证一个身份是否被废止。（2）周期间隔不同。A和B的**任务**是：周期性的检查idMapper模块中所存储的身份，将**无效身份**删除（包括idMapper模块中和Mediator模块中的），并停止这些无效身份所代表的peer结点的连接。这里的**无效身份**指**长时间未使用**（即不活动的，Inactivity）或**死亡**（即无效，被废止，被终止，Expiration）的身份。

* **go g.periodicalIdentityValidation(…)** - A周期间隔为24小时，由identityExpirationCheckInterval常量指定，猜想函数是永远返回true。B周期间隔为10分钟，由identityInactivityCheckInterval常量指定，猜想函数是永远返回false。A和B周期性的调用g.SuspectPeers(suspectFunc)，追溯SuspectPeers这个函数，经certStore模块（gossip/certstore.go中的listRevokedPeers函数），最终调用了idMapper模块，即identity/identity.go中的validateIdentities函数来获取**无效身份列表**：遍历idMapper模块成员pkiID2Cert中存储的所有身份，如果一个身份的最后访问时间（由该身份的成员lastAccessTime记录）距当前时间的时间差 >= 1小时，则说明该身份长时间未活动，这个1小时由同文件中的identityUsageThreshold常量指定；如果这个时间差小于1个小时，且**猜想函数**返回为true（只有A符合），则进一步使用MessageCryptoService验证这个身份，若验证失败，则说明该身份无效，被废或被终止。在遍历pkiID2Cert中存储所有的身份结束后，将无效身份列表revokedIds原路返回，在certStore模块中将这些无效的身份从Mediator模块中删除，最后在SuspectPeers函数中把这些无效身份所代表的peer结点的连接关闭。

## gossip服务器的初始化

**gossip服务器**原型为gossipServiceImpl，在service/gossip\_service.go中定义，同目录中还有一个eventer.go定义了gossip服务器配置升级相关的代码。gossip服务器管理的主要有：一个gossip服务实例gossipSvc，各个频道的状态模块chains和选举模块leaderElection，核心数据分发客户端deliveryService等等。除了gossipSvc提供的消息散播功能，直接管理的election模块与state模块（在下文详述），gossip服务器主要向外界提供两个功能（其余功能较简单或已在其他模块讲述，此处略）：配置初始化或更新，频道初始化。

（1）**配置更新** - gossip服务器也是需要通过配置而被控制的，除了像core.yaml这样的静态配置文件，还需要可以动态升级gossip服务器的配置。这样的实现涉及到gossip服务器的NewConfigEventer()接口，configUpdated()函数和eventer.go文件。NewConfigEventer()返回一个**可更新gossip服务器配置**的对象，这个对象在eventer.go中实现为**configEventer**。configEventer封装了存储最新配置的**成员lastConfig**和接收配置的**成员receiver**（即谁接收配置并应用到自己身上）。这里receiver有一个configUpdated(config)接口，就是让接收者用来升级自身的配置的，由gossip服务器实现，即在NewConfigEventer()中初始化configEventer对象实例时，receiver被赋值为gossip服务器实例本身。configEventer对象有一个ProcessConfigUpdate(config)接口实现，将传入的最新的配置数据config转换成适当的形式存储在lastConfig中，最后让receiver，也即gossip服务器，调用自身的configUpdated(config)升级自己的配置。追溯configUpdated，最终调用了channel模块的ConfigureChannel()将配置数据存储在对应的channel实例里面，channel实例再据此配置数据来验证结点和结点传送来的消息等，如IsOrgInChannel这个验证函数在HandleMessage这个处理消息函数中被调用。如此，调用者从gossip服务器实例的NewConfigEventer()获取该gossip服务器的配置对象，然后调用这个对象的ProcessConfigUpdate()传入配置数据，即可完成对gossip服务器的配置更新。这里补充两点，第一，配置数据有哪些：lastConfig的原型为configStore，封装了成员anchorPeers和成员orgMap，前者存储了频道中所有组织中的所有结点的anchor（锚点，即包含一个结点的地址，端口），后者存储了频道中所有组织的config.ApplicationOrg类型的配置，这个类型是configtx工具使用的配置对象，在eventer.go中实现为appGrp，包含了一个组织的名字，MSP服务ID，组织内所有结点的anchor。也即，配置的中数据项就包含这些而已。第二，ProcessConfigUpdate()所接收的配置config的数据有哪些：config原型是eventer.go中的Config，这个是一个兼容接口，/fabirc/common/config/application.go中的**ApplicationConfig**和/fabric/common/configtx/manager.go中的**configManager**各实现了Config的一部分，而/fabirc/core/peer/peer.go中的chainSupport则综合了两者，最终传入ProcessConfigUpdate()的其实也就是chainSupport实例，这点可以从peer.go的createChain函数中看出来，fabric中只有这个函数中有调用ProcessConfigUpdate()。

（2）**初始化频道** - 这里的初始化频道主要指为gossip服务器初始化一个所在频道的state模块，然后根据配置决定是使用election模块还是直接使用分发客户端deliveryService的服务。这个过程体现在InitializeChannel()函数，state模块的初始化参看下文，这里主要说根据配置二选一的问题。gossip服务器分发数据的工作有两种模式，一种随着结点在网络中的变动，**动态的**选举leader后，由选举出leader结点启动deliveryService服务开始负责分发数据，这就要启用electioin模块；另一种是直接有配置文件**静态的**的指定某一个peer作为leader，由其直接启动deliveryService服务开始负责分发数据。两者由core.yaml中的peer.gossip.useLeaderElection和peer.gossip.orgLeader指定，这两项是互斥的，最多只能有一项是true值。即：若使用election模块动态选举leader，所有结点的useLeaderElection必须为true，orgLeader必须为false；若直接静态指定leader，所有结点的useLeaderElection必须为false，leader结点的orgLeader必须为true，其他结点的orgLeader必须为false。fabric中默认使用第二种，即指定的leader结点直接调用StartDeliverForChannel()以启动deliveryService服务。election模块的初始化参看下文，在每次选举后每个结点根据自身的角色通过onStatusChangeFactory()函数进行开启或停止自己的deliveryService服务。

### election模块

**election模块**原型为leaderElectionSvcImpl，在election/election.go中定义，相关代码都在election目录中，可以将其理解为一个leadership管理模块，leadership可以翻译成领导关系，引申一下还包含了谁是领导，谁是群众，如何选出领导的意思。这里我们可以学习一下大神如何写The Algorithm in pseudo code的，即用“半code半语言”的方式描述一下算法，在election.go的开头注释中，详细描述了整个election模块的算法过程，十分清晰准确。election模块把网络中的所有结点分为leader（领导者）和follower（追随者，也可以说成群众）两种角色，处理的消息类型为LeadershipMessage（参看上文章节2.4），但LeadershipMessage消息成员IsDeclaration将消息分为proposal（申请）和declaration（声明）两种，proposal用于一个结点向其他结点**自荐**成为leader，declaration用于告知结点谁是leader。election模块有两大功能分区：（1）选举的算法，选举状态，领导者和群众的管理，（2）辅助选举而需要进行消息的收发，election模块自身专注于（1），而把（2）交给了一个gossip适配器，即成员adapter，adapter的接口原型为LeaderElectionAdapter，在election/adapter.go中实现。其实在gossip服务器的service/gossip\_service.go中，有一个专门生成election模块的函数newLeaderElectionComponent，分别调用了适配器专用生成函数NewAdapter和election模块专用生成函数NewLeaderElectionService，在这里，NewAdapter(g,...)传入了gossip服务器实例自身，自然这个适配器拥有收发消息的能力。

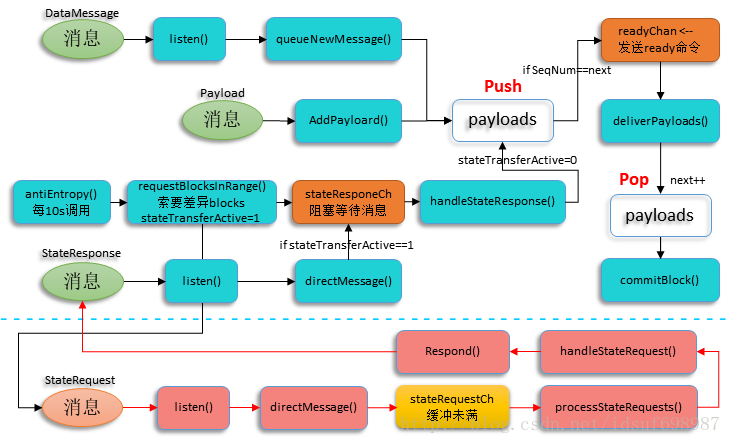
一个election模块初始化时，在专用生成函数NewLeaderElectionService中，初始化leaderElectionSvcImpl对象后，新启了一个goroutine执行start()函数开始运行这个election模块的服务，共做了三件事：

* **go le.handleMessages()** - 开启处理消息服务。首先，由适配器adapter获取接收消息的频道，msgChan := le.adapter.Accept()，msgChan是专门处理election模块中特定消息类型msgImpl的，所以Accept()中新启了一个goroutine去把gossip服务中获取的proto.GossipMessage类型的数据转成msgImpl类型的消息并转发到了msgChan中。然后，在for循环中持续接收msgChan中到来的一条条消息，交由handleMessage函数处理，处理的逻辑是：（1）如果消息是proposal，则记录到election模块成员proposals中，即把所有其他结点送来的想要成为leader的“自荐信”先放起来。（2）如果消息是declaration，则把已经存在leader的标识leaderExists置为1，然后若此时模块正在休眠且interruptChan是空的，则向interruptChan发空命令唤醒模块，若自己当前是leader而declaration中的新leader又不是自己，则调用le.stopBeingLeader()停止当leader。这里的interruptChan可以理解为模块的**中断休眠**频道，模块成员sleeping标识着模块当前是否在休眠，而waitForInterrupt(time)函数即通过select{...}让模块**进入休眠**，等待time时间后或declaration消息一旦到来，再置sleeping为false，即唤醒模块。
* **waitForMembershipStabilization** - 等待**成员关系**固定化。这里的**成员关系**指的是网络中存在的所有活着的结点的数量，固化指成员关系是**稳定的**。这个等待的过程是：确定当前时间为N后开始判断，使用适配器获取当前成员数量viewSize（追溯适配器可知，获取成员数量使用的还是discovery模块），然后进入for循环不断地判断成员关系是否稳定，判断的标准是，每隔一秒重新获取最新的成员数量newSize，若**newSize == viewSize**，则称当前成员关系是稳定的，否则，**更新viewSize**为最新发现的成员数量，即newSize = viewSize，进行下一轮判断。若整个判断过程超过N+15s的时间点（15s是由core.yaml中的peer.gossip.election.startupGracePeriod指定），或leader已经出现（即接收到declaration消息），则停止判断。这里需要理解，在一个新初始化的chain中，新的结点陆续加入到网络中，被gossip服务的discovery模块发现并记录，这个过程还是很快的，可能几毫秒短的时间内就会发现若干个新结点，所以在判断成员关系是否稳定时，若在等待1s后新获取的成员数量还是等于原来的成员数量，那么election模块就认为在1s“这么长”的一段时间内都没有新结点加入，则就“自认为”当前网络中所有活的结点都已经被发现了，即成员关系固定了。另外，这个等待成员关系固定化的过程是一个辅助election模块进行选举leader的，不能无限期的等待下去而耽误了选举的正事儿，所有给规定了15s，要是15s还没固定，那就不等了。自然的，要是这期间已经有了leader，那也就没有选举的必要了，没选举的必要，更没再等待固定化的必要了，所以这种情况下会停止判断。
* **go le.run()** - 在上两步的基础上，这一步进行election模块所应该做的本职的正事儿。（1）如果现在还没有leader，则调用le.leaderElection()进行选举，这里与其说是选举，不如说是进行自荐，在leaderElection()中，调用le.propose()向其他结点传播自己的“自荐信”（即包含自己身份信息的proposal消息，其他结点收到后会在它们的handleMessage中存储这封信）。然后调用waitForInterrupt(time)让模块进入休眠，这里的time为5s，由core.yaml中peer.gossip.election.leaderElectionDuration指定，在这个休眠的过程中既接收其他结点发来的它们的“自荐信”并存储在proposals中，也等待可能出现的declaration消息（参看上文第一步）。在模块被唤醒后，判断此时是否有leader存在，若存在（说明在休眠期间接收到了declaration消息），则自己放弃成为leader，若不存在，则拿自己的身份与proposals中已经收到“自荐信”中的其他结点的身份一一进行对比，看看自己是否比他们都更有资格当leader。这里还是比较有趣的，也算是无厘头的，这里所说的身份是一个结点的PKI-ID，而判断谁更有资格当leader的标准是bytes.Compare(peerID(id), le.id)，即谁的身份的二进制值更小，谁更有资格。嗯，怎么说呢，这样的标准好像没什么现实中的意义，不过gossip里面当leader是干更多活的，也没有什么其他特权和优待，所以呢，随便选选就好啦。若proposals中所有的身份都没有自己的身份小，那么自己就当仁不让当leader，通过调用le.beLeader()，真正成为一名leader：把已经有leader和自己是leader的标识，即成员leaderExists和isLeader，都置为1，然后开始调用le.callback(true)这个“倒钩”函数做leader在gossip中该做的事情。（2）选举后，如果自己是leader，则执行le.leader()，如果自己是follower，则执行le.follower()，都是自己所成为的角色该做的事情。leader要做的事儿：循环地每休眠5s后（因为自身是leader，所有不可能再从其他结点中接收到declaration消息而使休眠中断）调用le.adapter.Gossip(...)向其他结点传播发送declaration消息来宣告自己是leader；follower要做的事儿：清空“信箱”proposals以备下一轮选举，置leaderExists为0（用以等接收到leader发来的declaration消息后再置为1），然后用select{...}等待10s，这个10s由core.yaml中的peer.gossip.election.leaderAliveThreshold指定，10s结束后如果还没收到leader发来的declaration消息，即leaderExists还为0，则该结点将再发起新一轮的选举。这里仍然是，follower认为10s这么长的时间足够leader发送一条declaration消息给自己，若10s过后还没有收到，则follower“自认为”这个leader已死，则自己将发起新一轮的选举。（3）第（1）和第（2）在run()中循环进行的。

当一个结点调用le.beLeader()真正成为一名leader时（参看上文第三步），调用了le.callback(true)，当一个结点调用le.stopBeingLeader()停止作leader时，调用了le.callback(false)，两者只是参数不同。“倒钩”函数callback由election模块初始化时由参数传入，具体的赋给的值是service/gossip\_service.go中的onStatusChangeFactory(...)返回的函数func(isLeader bool) {...}，即根据是否是leader而分别执行g.deliveryService.StartDeliverForChannel和g.deliveryService.StopDeliverForChannel，从这点就可以看出如上文所说的，leader多干了DeliverForChannel向其他结点分发数据相关的一些事情，这个涉及到ordering服务的数据分发客户端，会在相关主题文章中详述。

### state模块

**state模块**原型为GossipStateProviderImpl，在state/state.go中定义，相关代码都在state目录中，可以将其理解为gossip传播和提交状态消息的管理模块，这里的状态的意思与账本数据，block数据的意思（参看上文章节2.3）一致。state模块也封装了一个适配器，election模块类似，也是被赋值为gossip服务器实例，在此不讲，而把笔墨集中在消息处理流程上。在state模块中，消息处理流程如下图：



参看流程图，蓝色虚线以上，以黑色箭头为流的为结点A的state模块**state\_A**，蓝色虚线以下，以红色箭头为流的为结点B的state模块**state\_B**。state模块处理三种消息类型：**DataMessage，StateResponse，StateRequest**。在state模块专用初始化函数NewGossipStateProvider中，初始化GossipStateProviderImpl对象后新启动了4个goroutine来开始运行state模块的服务。这些均在流程图中有所反映：

（1）payloads既是一个**存储器**，也是一个**消息排序输出器**，原型为PayloadsBufferImpl，在state/payloads\_buffer.go中定义。state\_A把接收的所有消息Push进payloads，一旦所接收的消息的序号**SeqNum == next**（next指所期望的下一个消息序号，消息序号即是一个频道的状态高度，也即一个频道的block高度），则向readyChan通道**发送ready命令**以指示所期望的消息已经收到，即当前阶段的数据**已经准备好了**。这里有一个**前提Q**是：ordering服务所输出的消息都是**排序过并依次**输出给state模块的。比如，在Push()函数中，若next为2，则说明序号为1的消息之前肯定已经被接收并处理过，当前所想要的是接收序列号为2的消息，**在前提Q下**，state模块再收到消息的序号也只可能是1或2，因为ordering服务可能因某些原因重发已经发过的消息1，但绝不可能跳过序号为2的消息而去发序号为3的消息。因此，若Push进的消息的序号小于2，则直接返回，若等于2，则**存储**进**成员buf**后**发送ready命令**。此外，Pop()函数从payloads中弹出一个消息给state模块处理，同时将next\*\*增1\*\*。比如一旦序号为2的消息被弹出交由state模块处理，则next变为3，即此时序号为2的消息已在处理且payloads所期望的消息变为了序号为3的消息。如此，由前提Q，Push()，Pop()三者相互配合，实现payloads的**消息排序输出功能**。

（2）启动的listen()循环接收来自gossipChan，commChan通道的消息，接收后分别交由queueNewMessage()和directMessage()处理。两个通道均在state模块初始化时由gossip服务的Accept()函数生成，且均在msgPublisher模块（参看上文章节3.3.2）中完成订阅，即gossipChan接收出版的标准的GossipMessage类型消息，实质上只处理DataMessage类型的消息，commChan则接收由comm模块转发来的ReceivedMessage类型的消息，实质上只处理StateRequest，StateResponse两种类型的消息。当state\_A接收到DataMessage消息时直接Push进payloads；state\_B在初始化时（也）启动了processStateRequests()循环监听stateRequestCh通道，当state\_B接收到来自state\_A的StateRequest消息时，说明A在向B索要\*\*一定序号范围内的\*\*block数据，state\_B则将StateRequest消息经由stateRequestCh通道发给handleStateRequest()函数，handleStateRequest()函数将自身存储的这些数据封装成StateResponse类型的消息，然后直接使用从A处接收的StateRequest消息中所携带的A的地址信息和方法Respond()把StateResponse消息回复给state\_A。

（3）state\_A在初始化时（也）启动了antiEntropy()这个**反熵函数**，每间隔10s执行一次去尝试调用requestBlocksInRange()，这个10s由state/state.go中的defAntiEntropyInterval常量指定。反熵的本意就是降低一个事物的絮乱程度，在这里所做的就是尝试降低自己结点与其他结点之间状态的差异，即所存储的数据高度的差异。由于消息在网络中传播快慢有所不同，自然，每个结点间当前所存储处理的数据也不同，可能state\_B已经在处理完了序列号为10的消息，而state\_A才刚刚处理完序列号为7的消息。这时10s的周期到了，state\_A调用committer.LedgerHeight()获取当前自身账本中的block高度current（假设为7），然后调用maxAvailableLedgerHeight()获取频道中其他结点中最高的高度max（假设为10），一看，current与max差了8-10三个数据块，则调用requestBlocksInRange()来弥补这个差异，差异弥补了，也就降低了整个频道状态的絮乱程度，也即反熵。在requestBlocksInRange()函数中，会尝试3次分批索要。3次由defAntiEntropyMaxRetries常量指定，分批则是在差异较大时，如几十块数据时，以10块（由defAntiEntropyBatchSize常量指定）为一批，一次次的索要，当然这里我们只差3块，只索要一次即可。索要时，先将stateTransferActive置为1以表明自己发生了**索要行为**，然后调用stateRequestMessage()生成StateRequest消息**gossipMsg**，再然后调用selectPeerToRequestFrom()随机从满足条件（即拥有8-10这三块数据）的结点中选出一个索要对象**peer**（假设peer为结点B），接着向调用gossip.Send(gossipMsg, peer)向state\_B发送StateRequest消息后，进入stateResponseCh通道阻塞，以\*\*等待\*\*state\_B的回复（对看上一点（2））。当state\_A的listen()监听到state\_B回复的StateResponse消息时，把消息交给directMessage()分流，此时若stateTransferActive==1，说明自己索要过StateResponse消息且当前正在等待这个回复（否则就说明state\_A之前不存在索要行为也没有在等，也即当前state\_A与其他结点状态暂时不存在差异，即便别人主动给state\_A发了，state\_A也不会处理），则再将StateResponse消息分流至stateResponseCh通道，此时阻塞等待结束，StateResponse消息被交由给handleStateResponse()处理，处理的过程无非就是把收到的消息中包含的自己索要的自己没有的块数据Push到payloads中以供进一步的处理（对看下一点（4））。antiEntropy()解决的是防止一个结点永远也跟不上其他结点的节奏或越来越落后于其他结点的状态。比如state\_A所在环境导致处理消息的效率本身就很慢，state\_B所在环境导致处理消息的效率较快，两者的环境一直处于不变的状态，则若没有这个反熵函数，则state\_A与state\_B的差距会越来越大。

（4）state\_A在初始化时（也）启动了deliverPayloads()，deliverPayloads()函数循环等待来自readyChan通道的**ready命令**。在以上（2）（3）两点state\_A所接收的包含数据的消息，最后都Push进payloads中，当payloads处于准备就绪的状态时，会发送**ready命令**（参看第（1）点），deliverPayloads()函数一旦收到这个命令，就会将payloads中所存储的所有消息（一般只有一条）Pop出来，Unmarshal()后将数据封装成common.Block类型的rawBlock，最后调用commitBlock(rawBlock)，一方面将数据提交至自己的核心账册：committer.Commit(block)，另一方面更新自己的状态（高度）：UpdateChannelMetadata(...)。这里的committer是核心代码中的数据提交模块，作用就是将数据提交至账本记录下来，将在相关主题文章中详述。

AddPayloard()也可以向payloads中Push消息，但这个函数是经由gossip服务器的AddPayloard()供外界手工调用以加入一条消息的。关于state模块初始化时有两个**遗留问题**，还是比较关键的，自己没有想明白，若有大神知道，还望赐教：

1. 调用committer.LedgerHeight()获取当前自己结点的账本状况高度height，并将height作为next初始化了payloards，按理说从Ledger中获取的序号height，说明序号为height的消息已经处理并提交至账本了，为什么还要作为next的值，即payloards初始化时期望接收的是序号为height的消息，但这个消息不是已经处理过并提交至账本中了么，那此时state模块初始化后，ordering服务再发序号为height的消息给state模块吗？这点可能要在探究ordering服务的时候寻找相应的答案了。
2. 在GossipStateProviderImpl对象初始化后，就调用了UpdateChannelMetadata(...)升级了自己的状态，但这里用的高度是height-1，这是为什么，是否和第1点相关？

## gossip服务的停止

gossip服务，包括它控制的各个模块的停止，都是通过一个**停止通道**来实现。以gossip服务对象gossipServiceImpl为例，它的**停止通道**是成员**toDieChan**。在停止服务函数Stop()中，按顺序依次停止各个模块中，g.toDieChan <- struct{}{}语句就是向停止通道发送一个**空命令**，而正在服务的periodicalIdentityValidation，handlePresumedDead，acceptMessages，Accept函数中所起的goroutine，select接收到toDieChan频道发来的停止服务的**空命令**，随机return退出循环而结束goroutine，即停止了服务。gossip服务中的其他模块也遵循这种停止自身服务的模式。

# fabric源码解析15——peer的gossip服务之一条消息的散播旅行

## 旅行的前提

要理清一条消息在频道中的各个结点中如何散播的，重点在于三点：

* **消息从何而来**。
* **消息如何散播**。
* **消息去往何方**。

需要说明的是，首先，这里消息指的是**ordering服务**经**deliverservice服务**发送给**gossip服务器**后在众结点间散播的**block块消息**（即把block作为payload封装到DataMessage类型的GossipMessage），而gossip服务器中处理的其他类型的消息，如state消息，关系消息等，都具有辅助性质，即都是为了**block块消息能够在众结点间散播并达到最终一致**所服务的。其次，**预设的情景**是：fabric网络中存在一个ID为**chainID**的频道，该频道包含一个名为**orgID**的组织和一个名为**ordererID**的ordering服务结点，orgID组织中包含ID为**nodeA，nodeB，nodeC，nodeD**四个结点，每个结点的gossip服务器都已初始化，**nodeA**被指定或暂时选举为**leader**。留意这些预设对象的名字，下文中将直接使用。再者，消息传播基于的**grpc服务**在下文所述的deliverservice模块初始化中应该还尚未与服务器端建立连接，这里只是假设已连接。最后，在讲述消息散播的过程中，重点是**消息如何散播**，由于这个过程还是比较曲折的，为了不冗杂且把注意力放在流程展现上，所以描述中文字不可能过细，函数所在文件也不可能一一列出，所涉及的函数基本集中在如下目录中。后文中若非绝对路径，则均以下列所提到的路径为基础，读者若找不到函数所在文件或文件所在目录，可与文章《fabirc源码解析14》中相应模块对看，或自行用grep，locate命令搜索。

* /fabric/core/deliverservice
* /fabric/gossip
* /fabric/protos/gossip

## 消息从何而来

一块块block消息由ordering服务序列化后，使用deliverservice服务的分发客户端发送给gossip服务器。即，消息直接来自于deliverservice模块。这里假设deliverservice模块会从ordering服务依次接收序号在11-20之间共10块的block，在传播过程中，这些数据在各个模块间传送，会变化或被封装成不同的消息类型，但均用**M11，M12，…，M20**表示。

### deliverservice模块的初始化

deliverservice模块代码集中在deliverservice目录下，原型为deliverServiceImpl，在deliveryclient.go中定义，利用**成员blockProviders**提供对每一个频道**BlocksProvider对象**的管理。BlocksProvider对象利用grpc客户端从ordererID结点出接收消息后使用gossip服务器开始传送消息，原型为blocksProviderImpl，在blocksprovider/blocksprovider.go中定义。

因为只有leader结点才会启动该模块，因此以nodeA结点为例。在start.go的serve中，InitGossipService()实例化了gossip服务器，但此时并未初始化deliverservice模块，直到后文的Initialize()才间接在/fabric/core/peer/peer.go的createChain()中调用service.GetGossipService().InitializeChannel()将deliverservice模块初始化：在InitializeChannel()中，调用g.deliveryFactory.Service(...)将gossip服务器实例和指定的ordererID的IP地址等封入**配置Config**后传入deliverservice模块中，然后调用StartDeliverForChannel()启动了模块。

### deliverservice模块的启动

在StartDeliverForChannel()中，步骤如下：

1. 在blockProviders中若属于chainID的BlocksProvider存在，则表明当前nodeA的deliverservice模块已经在运行（对看StopDeliverForChannel()，一旦nodeA停止模块，则把BlocksProvider从blockProviders中删除），则直接返回。
2. 若不存在，调用newClient新建一个grpc客户端client。这个客户端原型为broadcastClient，在client.go中定义，实现的是/fabirc/protos/orderer/ab.pb.go中定义的AtomicBroadcast\_DeliverClient这个**Deliver服务的grpc流客户端接口**。这里不直接使用ab.pb.go中的atomicBroadcastDeliverClient，自然是因为这个自动生成的结构不能满足功能的需要。而且可以臆测一下，Deliver服务的grpc流服务端应该实现在ordering服务中。
3. 调用NewBlocksProvider(...)新建一个属于chainID的BlocksProvider对象，并把2中新建的client、deliverservice模块Config中的gossip服务器，签名对象CryptoSvc传给这个对象的各个成员，然后把这个对象放入blockProviders中。
4. 新启一个goroutine，go d.blockProviders[chainID].DeliverBlocks()，执行3中新建的BlocksProvider对象的DeliverBlocks()服务。
5. DeliverBlocks()就是实际办事儿的函数，在这个函数中，循环使用client客户端从ordering结点接收11块block消息，msg, err := b.client.Recv()，然后使用switch-case根据消息的类型分别处理每块消息。这里只关注DeliverResponse\_Block类型的消息，即我们要传播的原始的block块数据，在此分支中，先调用createPayload()将block数据包装成可存储在本地账本的**payload**，再调用createGossipMsg()将payload包装成可用于传播的DataMessage类型的GossipMessage消息**gossipMsg**（gossipMsg的Tag值为**CHAN\_AND\_ORG**），然后调用gossip服务器的AddPayload()直接**将payload存储在本地的账本中并更新chanState模块中对应chainID的channel对象的状态**（此状态指channel实例成员stateInfoMsg，包含当前从payload抽取的高度和时间戳），再调用gossip服务器的Gossip()**将gossipMsg散播出去**。而gossipMsg被传播到其他结点，比如nodeB后，目的也是把gossipMsg中包含的payload抽取出来后存储到nodeB本地的账本中。

## 消息如何散播

1. 序列号在11-20共10个gossipMsg进入nodeA的gossip服务器的Gossip(gossipMsg)中，先把gossipMsg包装成SignedGossipMessage类型的**sMsg**，然后使用MessageCryptoService对sMsg签名，使sMsg包含了nodeA的身份信息。
2. 检查gossipMsg的Tag，if msg.IsChannelRestricted()，若gossipMsg指定可以在频道范围内散播，则nodeA在chanState模块中使用chainID对应的channel对象的AddToMsgStore(sMsg)函数，将sMsg在blockMsgStore存储一份，也在blocksPuller中的itemID2Msg以PKI-ID为key存储了sMsg，并把sMsg的序号在engine中的state中存储一份。这里提一句，在blocksPuller中存sMsg和sMsg的序号是为了pull消息的过程中使用，但是在blockMsgStore存储一份是为了什么目前笔者还没搞清楚，因为事实上只见往blockMsgStore中存了但是压根没找有在哪里使用。
3. g.emitter.Add(sMsg)，将sMsg包装成batchedMessage后添加到emitter模块的buff中，准备发送。我们知道emitter是一批批发送的，指定一批大小的burstSize的值默认为10，当序号为20的sMsg被Add进emitter后，就会触发emitter模块的emit()函数。emit()将现存的消息，即序列号为11-20的消息，抽取出data放入“发射数组”msgs2beEmitted，然后先调用倒钩发送函数cb(msgs2beEmitted)，再调用decrementCounters()清除emitter模块中剩余发送次数为0的消息，因为将sMsg包装成batchedMessage时所给的发送次数默认为1，因此这发送过的10条block消息都会从buff中删除。这里需要说明的是，在同一时段，emitter模块不一定只收到block块数据，也会收到其他类型的、其他频道ID的消息（比如channel模块会通过它的适配器往emitter中Add消息），而因为我们的关注点在block块消息，所以我们这里只是假设emitter只是清一色的接收到了11-20这10个属于chainID的block块消息。
4. 倒钩发送函数cb即为gossip服务的sendGossipBatch(...)，简单的将接收的10条消息数组重新置换成SignedGossipMessage格式的消息数组msgs2Gossip，然后直接调用gossipBatch(msgs2Gossip)发送消息数组。
5. gossipBatch(msgs[])可处理多种类型数据，这里我们还是只关注block块消息。调用partitionMessages(isABlock, msgs)函数，将消息数组中的DataMessage类型SignedGossipMessage消息过滤出来（对看第3点）放入blocks，然后将**blocks**和一个**过滤器filter**传入gossipInChan(...)，这里的这个过滤器filter将在下文单独谈论。
6. 在gossipInChan(...)中，（1）调用extractChannels(messages)将消息中所有的频道ID抽取出来放入totalChannels，然后用第一层for循环遍历totalChannels，针对每个不同频道将属于各自频道的消息在频道内传播，这里10条消息中包含频道消息都是chainID，因此第一层for循环唯一一次循环就是针对chainID的。（2）在第一层循环for中，再次用partitionMessages来过滤选出属于chainID的消息放入messagesOfChannel（这里是10条消息都是属于chainID的），这个是最终要发送的**消息清单**。（3）接着获取chanState模块管理的对应chainID的channel模块gc供下两步使用。（4）再利用discovery模块获取当前chainID频道中所有活着结点membership（这里就是nodeB，nodeC，nodeD三个结点）。（5）接着调用过滤器filter模块的SelectPeers(...)从membership中随机筛选出3个结点集合peers2Send，这里原代码的逻辑会把nodeB，nodeC，nodeD三个结点都会被筛选出来，但为了体现gossip散播的过程，虽然三个结点都满足条件但**只随机选出一个结点**，假设为nodeB，这个是最终要发送的**结点清单**。清单的原型是RemotePeer，包含一个结点的Endpoint和PKIID（6）使用第二层for循环遍历**消息清单**中的所有消息（即序列号11-20的block块消息），依次调用comm模块的g.comm.Send(msg, peers2Send...)，向**结点清单**发送M11，M12，…，M20。
7. comm模块就是nodeA向其他结点传播消息的grpc通信的模块了，既是grpc流客户端，也是grpc流服务端。站在nodeA的角度，在发送消息时，使用的是流客户端。在Send()中，for循环遍历**结点清单**中的每个结点，这里只有nodeB，针对nodeB启动一个goroutine来调用c.sendToEndpoint(peer, msg)。
8. 在c.sendToEndpoint(peer, msg)中，参数peer值为nodeB时，nodeA先调用connStore模块的connStore.getConnection(peer)获取当前nodeB的grpc连接conn，当这个连接不存在时会进行创建，创建的时候会同时运行这个连接的读写函数。conn即nodeA与nodeB间的连接，然后调用conn的conn.send(msg, disConnectOnErr)将消息封装成msgSending类型后经conn的发送通道outBuff从nodeA发给nodeB。经过第6步中（6）的第二层循环，将M11，M12，…，M20最终都会发送给nodeB。**以下步骤只有M11为例**。
9. nodeB的comm模块作为grpc服务器端，在GossipStream()函数中接收到nodeA发来的M11。也即**从这一步起，就要站在nodeB的角度来看代码**。nodeB此刻作为服务器角色，调用connStore.onConnected()获取服务端流与nodeA的连接conn后，启动了conn的serviceConnection()，即新启了goroutine来用readFromStream接收nodeA发来的M11，然后抽取M11中的SignedGossipMessage类型内容作为msg后经由msgChan发给conn.handler(msg)这个conn中的“倒钩”成员来处理。
10. conn的handler()是在上一步的GossipStream()中获取conn后被赋值的，该“倒钩”成员所做的是：把接收的SignedGossipMessage类型的M11封装成ReceivedMessageImpl消息，交由msgPublisher模块的DeMultiplex()进行**出版**。
11. 在nodeB初始化自己的gossip服务对象实例时，已经在gossip/gossip\_impl.go的start()中调用incMsgs := g.comm.Accept(msgSelector)在msgPublisher模块中**注册订阅**了专用于接收ReceivedMessageImpl类型消息的频道incMsgs，然后又调用了go g.acceptMessages(incMsgs)新启了一个goroutine来接收incMsgs这个通道的消息。第10步中DeMultiplex()出版的M11会通过incMsgs这个通道发送到了acceptMessages(incMsgs)中，一旦收到M11，会交由g.handleMessage(msg)处理。
12. 在g.handleMessage(m)中，先重新抽取ReceivedMessageImpl类型的M11中的SignedGossipMessage作为消息msg进行一系列if判断，调用g.chanState.lookupChannelForMsg(m)获取chanState模块中chainID对应的channel对象gc，经过一系列判断，会调用gc.HandleMessage(m)对接收的原消息M11进行处理。
13. ReceivedMessageImpl格式的M11再次进入了channel模块，在HandleMessage()中，先从ReceivedMessageImpl类型的M11中抽取出SignedGossipMessage作为消息m，在m是DataMessage的前提下，m会进入if m.IsDataMsg()分支，进行如下处理：（1）gc.blockMsgStore.Add()，将M11存储到blockMsgStore中，如果添加成功，则继续，否则直接退出。（2）gc.Gossip()，从nodeB出发，从第1步开始，继续在网络中散播M11。（3）gc.DeMultiplex(m)，出版M11，供本地订阅这接收。（4）blocksPuller.Add()，添加到blocksPuller，供bocksPuller在pull机制中运作（blocksPuller中有了M11，就不必再向其他结点索要了）。
14. state模块在初始化时，就向msgPublisher模块**注册订阅**了专用于接收以DataMessage为Content的SignedGossipMessage消息的通道gossipChan，并启动一个goroutine来执行listen()，接收gossipChan中来的消息。因此当第13步（3）中出版M11时，state模块会通过listen()从gossipChan通道中接收到M11并执行go s.queueNewMessage(msg)处理。
15. 在queueNewMessage(msg)中，先M11中抽取出payload，然后Push进payload。然后会被payloads弹出来后交由commitBlock，同样，也是提交到nodeB自身的账本中后，更新nodeB自身的channel对象的stateInfoMsg，再次触发nodeB的channel的publishStateInfo，向其他结点，包括nodeA，发送StateInfo消息（用于向这些结点报备自己的身份，可结合下文理解报备的意思）。

### 散播过程中如何选择散播结点

我们做过比喻，gossip算法类似于办公室八卦和疫情传染，因此一个结点向**另一些**结点散播消息，这个**另一些**在选择上有两个特征：（1）**数量不定**。（2）**随机并就近选择**。

gossip在选择结点时使用的是filter/filter.go，即filter模块。主力函数是SelectPeers，辅助函数是CombineRoutingFilters，和util下的GetRandomIndices。而上述的两个特征，会由SelectPeers和GetRandomIndices体现。

传播过程第5步中所提及的过滤器filter，指下面这个调用g.gossipInChan的第二个参数：

//在gossip/gossip\_impl.go中的gossipBatch函数中

g.gossipInChan(

blocks,

func(gc channel.GossipChannel) filter.RoutingFilter {

return filter.CombineRoutingFilters(

gc.EligibleForChannel,

gc.IsMemberInChan,

g.isInMyorg)

}

)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

这个过滤器filter是在gossipInChan中散播block时，作为所调用的filter.SelectPeers的**第3个参数**，被用于筛选适合的结点集合peers2Send。上面代码中展示的筛选条件很清晰，有三条：即一个NetworkMember形式的结点身份传入gc.EligibleForChannel，gc.IsMemberInChan，g.isInMyorg验证后都返回为true，那么这个结点才会被选中。撇开后两个条件不谈，单说gc.EligibleForChannel，该条件验证了一个结点的PKIID是否存在于nodeA的channel对象（指chanState模块所管理的对应chainID的channel对象，下同）成员stateInfoMsgStore中。这个成员存储结点间发送的StateInfo消息和消息中携带的结点身份信息。也就是说，nodeA此刻传播block时，只有stateInfoMsgStore中存在的结点才会被筛选出来。而一个结点的身份信息想出现在nodeA的stateInfoMsgStore中，必须在nodeA开始筛选结点之前，就把自己的StateInfo消息通过publishStateInfo发送给nodeA（publishStateInfo是channel模块周期性执行的函数），而publishStateInfo也是像上述过程那样**一步步散播StateInfo消息**。说到底，散播使用的是comm模块实现的**grpc网络传输服务**，而只有与nodeA近的结点，才能更快的通过网络将自己的StateInfo消息散播给nodeA，如此的话，当nodeA在散播block时，能筛选出的都是**离自己特别近**或者**传输效率特别高**的结点。另外，stateInfoMsgStore存储的身份会定时的清理，实际上是每隔400s，即清理MessageStore，也用callback同步清理MembershipStore，这点可以从NewMessageStoreExpirable的实现看出。这么做是因为，一个结点，如nodeA，若不实施定时清除其他结点发来的StateInfo消息，经过一段时间后，nodeA也会接收到距离较远的结点发来的StateInfo消息并记录在stateInfoMsgStore中，这样当nodeA筛选结点发送block时，这些较远的结点也有可能被选中，这就违法了就近的原则。

每个结点，如nodeB，其state模块在初始化时都会调用UpdateChannelMetadata来更新（虚假更新，因为给的StateInfo消息中的高度是现有高度-1）一下自己channel对象成员stateInfoMsg，并置shouldGossipStateInfo为1，目的就在于驱动publishStateInfo传播一次这条虚假的stateInfoMsg，以让距自己比较近的结点，如nodeA的stateInfoMsgStore存储到自己的身份信息，进而在nodeA传播block时自己能在传播的范围之内。

*关于虚假更新，这个对看文章14中在讲state模块处所遗留的高度-1的疑问，这里解释为：可能就是进行一个虚假更新，只是为了让距自己比较近的结点存储到自己的身份信息，在之后传播消息的时候能算自己一份。其实只要这个虚假更新所更新的高度比现有高度低就行，即-2，-3其实也行，但是因为一个结点最低的高度就是1（即genesis块），也因为高度没有负数一说，所有这里给的是-1。不过这个好像也不太对，因为既然是虚假更新，那为什么不直接给当前的高度呢？*

以上就是gossip算法**就近**特征的叙述。

**随机**特征比较好理解，由GetRandomIndices实现，在filter.SelectPeers中被调用。

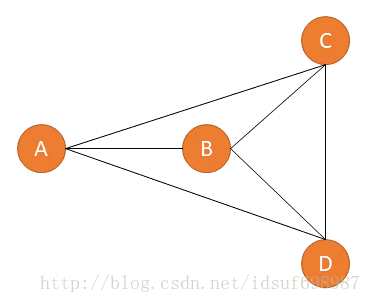
**数量不定**特征在gossip中不明显，因为指定数量的是filter.SelectPeers的**第1个参数**，而这个参数在实现中又是由配置指定的。但是当符合条件的结点少于指定的个数时，则数量不定，比如配置指定的是每次向10个结点散播，但筛选出来的只有5个，那只能向这5个结点散播，如果筛选出来8个，则只向这8个结点散播。

state模块订阅了DataMessage类型数据，而且是gossipMessage或signedgossipmessage类型的

## 消息去往何方

通过上述所讲的散播步骤的第14、15步，可知，消息在散播进一个结点后，一方面会存储到自己的结点的账本和一些本地模块中，另一方面会继续在网络中散播。至于散播何时停止，且往下看。

## 旅行足迹图



如图，A，B，C，D四个结点，属于同一频道同一组织。这里更粗线条的模拟一下传播的过程，初始条件：

* A-B，B-C，B-D，C-D均为一个单位距离，A-C，A-D均为两个单位距离。
* 四个结点处理消息效率一致，每100s，才能向一个单位距离范围的结点传播一条消息。
* 每隔400s，每个结点的stateInfoMsgStore清理一次名单。对于存在时间>=400s的名单，将废止。
* 只传播一条消息M，是高度为2的block。时间用Tn表示，如T100表示时间在100s时，T110表示时间在110s时。每个结点发送自身的状态信息用S（i，h，t）表示，i代表身份，h代表当前账本的高度，t代表时间戳。每个结点的账本高度当前均为1。每个结点需花费10s，才能将M存储到自身的账本中。
* 此刻从0s开始计时，A为leader，开始接收deliverservice来的消息并开始传播。此前已花费了100s时间供四个结点同时初始化，每个结点此刻中只存在一个单位距离范围的结点名单，即：A中有S（B，1，0）；B中有S（A，1，0），S（C，1，0），S（D，1，0）；C中有S（B，1，0），S（D，1，0）；D中有S（B，1，0），S（C，1，0）。
  1. T10时，A将M提交到自己的账本中，发送了S（A，2，10）给B，随即将M传播给B。
  2. T110时，B收到S（A，2，10）并更新自己的名单，同时收到M，立即传播给A，C，D。T120时，B将M提交到自己的账本中，向A，C，D发送S（B，2，120）。
  3. T200时，在T0前100s就开始传播S的A-C，A-D两条2个单位距离线路的双方也都收到了各自的S，即A收到S（C，1，-100），S（D，1，-100）；C收到S（A，1，-100）；D收到S（A，1，-100），此刻每个结点的名单中都包含另外三个结点。T210时，A，C，D收到B发送的M，A已有M，未作进一步处理，C和D立即向其余三个结点传播M。T220时，C和D将M提交到自己的账本中，向其余三个结点分别发送了S（C，2，220），S（D，2，220），同时，A，C，D也收到B发送来的S（B，2，120）。 T220时，名单情况为：A中有S（B，2，120），S（C，1，-100），S（D，1，-100）；B中有S（A，2，10），S（C，1，0），S（D，1，0）；C中有S（A，1，-100），S（B，2，120），S（D，1，0）；D中有S（A，1，-100），S（B，2，120），S（C，1，0）。
  4. T310时，A，B，D收到C发来的M，因为都有了，未作进一步处理；A，B，C都收到D发来的M，因为都有了，未作进一步处理。T320时，A，B，D收到C发来的S（C，2，220）；A，B，C收到D发来的S（D，2，220），各自更新自己的名单。T320时，名单情况为：A中有S（B，2，120），S（C，2，220），S（D，2，220）；B中有S（A，2，10），S（C，2，220），S（D，2，220）；C中有S（A，1，-100），S（B，2，120），S（D，2，220）；D中有S（A，1，-100），S（B，2，120），S（C，2，220）。
  5. T400时，每个结点的stateInfoMsgStore清理一次名单，C，D中的S（A，1，-100）将被清除。至此M在四个结点间散播过程终止。

从这里可以看出用文字描述散播过程还是相当费力的，本想画一个类似state模块那样的通信流程图的，但发现很难画的清晰。上述过程还是最简单最简单的情况，但是也是可以看出，对stateInfoMsgStore清理名单的周期进行一定量的设置，每个结点所持有的散播名单都会保持在一定范围内。而且目前总感觉自己对gossip这个模块还隔着一层，还有一些深层的东西没挖掘，理解出来。由于字符化的表达过多，要是有错误，还请指出。

# fabric源码解析16——peer的gossip服务之测试

## 彻底一点儿

原本打算在《fabric源码分析15》中将gossip服务的主题文章就此结束，毕竟前前后后有一个月了，但是确实仍留有一种不透彻的感觉。最近突然意识到，这种感觉是只看代码却没有测试验证想法所产生的。在《fabric源码分析1》中，就已经把fabric源码分为三类，其中一类就是辅助研究源码的，主要指项目中的test文件，一直没有好好利用。这里需要理解的是，test与源码之间相互关联，要看懂test，最起码对源码有一定理解的基础，否则连test在测什么内容都不知道。这篇文章就借着上篇文章不透彻的困扰，一方面介绍一下如何使用fabric中的test文件，方便读者自行测试自己感到困惑的地方；另一方面佐证对gossip源码的分析，驱除困扰。这里有一些前情概要：

* 这里叙述的操作系统是Ubuntu16.04，64位。
* 姑且假设各位和笔者一样，属于go菜鸟（如果大神觉得下文某些地方太小白的话）。各位的go编译已经部署到位。
* 仍以block消息的传播为例。若涉及的步骤或模块不清楚，请参看《fabric源码分析14》和《fabric源码分析15》。

## fabric test

1. fabric源码使用到的第三方库不用担心，都在源码目录的vendor中。可以将vendor目录下的所有文件直接复制到**GOPATH/src**下。查看自己系统的GOPATH路径，可以执行go env查看。
2. fabric源码中import进自身源码的路径均如github.com/hyperledger/fabric/...，因此要在GOPATH/src/github.com中新建hyperledger文件夹，然后将fabric项目整个文件夹以fabric的名字放入hyperledger中。
3. fabric在编译过程中（test也是先编译然后在执行test）可能会需要一些系统/usr/下的头文件或库，基本上都是vendor下go的第三方库所用到的。这个顺序就是，fabric用go标准库和vendor，vendor用go标准库和另外一些第三方库。各位的系统中并不一定会有vendor使用的一些第三方的库，在编译过程中若出现这样的问题，就需要安装了。这个问题就需要根据错误提示自行安装了，基本上在联网状态下使用apt install xxx就可以搞定安装问题。
4. go test命令，这里我们主要用到的是**单测某个测试函数**或**单测某个测试文件**。单测某个测试函数，如/fabric/gossip/gossip/gossip\_test.go中的TestDissemination，使用的命令是go test -test.run TestDissemination；单测某个测试文件，如/fabric/gossip/gossip/gossip\_test.go，使用的命令是go test gossip\_test.go XXX，其中XXX指gossip\_test.go测试代码直接用到的各个源码文件。我们这里只使用单测一个测试函数的命令。

## gossip test

测试的目的在于验证我们研究的**gossip传播消息的过程**，最基本的方法就是在适当的地方设置**打印点**打印输出信息。为了佐证，输出的信息基本上要围绕两点：（1）**谁把哪些块消息在什么时候发送给了谁**。（2）**谁在什么时候接收了来自谁的哪些块消息**。

测试的难点在于，gossip服务是一个由多个模块相互配合完成各个结点间的消息传播的，各类模块和类型消息可能对打印点形成干扰。而且除了正常的消息散播，还有pull机制等非散播方式的消息传输的影响。应该在何处收集信息来打印还是需要注意的。还有，项目中所给的测试函数的本意不一定是我们想测试的结果，或者为了方便，需要我们进行稍微的修改。接下来以gossip\_test.go中的TestDissemination测试函数为例，进行消息散播的测试。

### 测试函数TestDissemination

TestDissemination从名字上就可以看出来，是专用于测试消息散播的测试函数（以下的代码片段都引自该函数中），所做的是：

1. 分为上下两部分，上部分主要用于测试传播DataMessage类型的GossipMessage，下部分主要用于LeadershipMessage类型的GossipMessage，中间还夹杂着对最后一个结点lastPeer的UpdateChannelMetadata的测试。这里我们只以DataMessage为例，也即可以指我们所说的block消息。所以可以将//Sending leadership messages与t.Log("Stopping peers")之间关于测试LeadershipMessage消息的代码注释掉。
2. 创建了10个gossip服务实例，一个角色为bootstrap的gossip服务实例，因为是测试，所以就以gossip服务实例代表结点。fabric中，bootstrap角色本是用于新加入频道的结点向bootstrap结点直接索要数据以快速与其他结点达到状态一致的情景的，这里却类似于DeliverService的作用，即作为gossip所散播的消息的来源和起点。而且，这里为了简便和容易分析，我们将**结点数**，**发送的消息条数**都改为4个，即n := 10改为n := 4，msgsCount2Send := 10改为msgsCount2Send := 4，相应的，boot := newGossipInstance(portPrefix, 0, 100)中的100要改为16（100=10x10，16=4x4），这个值指定的是msgstore模块最多存储的block消息数。
3. 在for i := 1; i <= n; i++ { pI :=... }的**每次循环中**，创建一个gossip服务实例pI，然后调用pI.Accept从pI中**订阅一个专用于接收DataMessage消息的通道acceptChan**，并新启一个goroutine，不断接收从acceptChan发来的消息，每接收一条消息，就将pI的**接收消息数量槽**（参看下文第7步）的值+1，一旦pI收满4条消息，立即wg.Done()并结束这个goroutine。这样，我们有了名为**p0-p4**五个结点，其中p0为bootstrap结点。对应使用的地址都是本机，端口为3610-3614。
4. waitUntilOrFail(...,predfunc)函数用于等待，最多等待timeout（全局变量，180s），每隔3s执行一下predfunc，若该函数返回true则立即结束等待。checkPeersMembership(t, peers, n)则是检查每个结点所存储的成员关系是否已经处于**饱和**（discovery模块要干的事情）。这里所说的**饱和**指除自己外其余4个结点（包含bootstrap）的身份信息都已存在。有了这些**总名单**，gossip服务才能从中甄选一些进行散播消息。注意，这里测试中进行了成员关系饱和的等待，而实际操作中，一个结点的消息的散播不会等到成员关系饱和才开始。
5. 待各个结点成员关系饱和后，for i := 1; i <= msgsCount2Send; i++ { boot.Gossip(...) }，用bootstrap结点boot依次散播4条DataMessage消息。这里的DataMessage由createDataMsg生成，只标记了消息的Channel，Tag和SeqNum。第一条消息则SeqNum为1，第二条消息则SeqNum为2，…。
6. waitUntilOrFailBlocking(t, wg.Wait)，等待wg.Wait完成。第3步中每个结点新启的goroutine结束后，即每个结点接收到了4条消息，都会执行wg.Done()。等4个结点都Done过，wg.Wait就完成了，也就结束等待了。所以，这一步是在等待消息的完全传播，即等待每个结点都接收到4条消息。
7. **接收消息数量槽**指的是receivedMessages数组，容量是n，4个结点各占一个，用来表示每个结点接收到的消息数量。assert.Equal(t, msgsCount2Send, receivedMessages[i])就是用来验证在第6步的等待后，每个结点是否都接收到了4条消息，验证的方法就是看看每个结点的**接收消息数量槽**的值是否等于4。
8. 对看/fabric/gossip/gossip/channel/channel.go中HandleMessage的if m.IsDataMsg() || m.IsStateInfoMsg()分支，这里是每个结点**最终处理接收到的DataMessage消息的地方**，补充说三点。**（1）**每个结点只有调用gc.DeMultiplex(m)将接收到的DataMessage出版出来，上文第3步订阅DataMessage消息的acceptChan才会收到消息。而gc.DeMultiplex(m)在if added { ... }分支中，即只有added = gc.blockMsgStore.Add()添加消息成功，才会进一步把消息出版。而blockMsgStore（msgstore模块）的存储机制上，当添加一条DataMessage时会验证该条消息是否已经存储，若已经存储则返回false。因此，当成功添加4条消息，说明添加了4条不同的消息，而bootstrap结点又只散播了SeqNum1-SeqNum4，即一个结点成功添加4条消息就是添加的SeqNum1-SeqNum4这4条消息，然后会在每次成功添加后出版这4条消息。如此，一个结点的acceptChan收到4次消息时（此时该节点接收消息数量槽的值为4），证明这个结点已经接收到了SeqNum1-SeqNum4这4条消息。**（2）**同样的，在添加成功的情况下，一个结点才会进一步调用gc.Gossip()将收到的4条消息进一步向其他结点散播。**（3）**第6步中，每隔3秒检查一下每个结点是否接收到4条消息，一旦接收到，等待立即结束，接着就会调用waitUntilOrFailBlocking(t, stop)开始停止每个gossip服务实例（关于测试leadership的代码已经注释掉），由于都是本机上的实例，这个操作是非常快的。而我们要测试查看的是每个结点的散播情况，所以这里会有这么一种情况：当最后一个结点刚刚接收完第4条消息时还没来得及进一步gc.Gossip()或没有到达我们的设置的出口打印点（参看下文），就进行完了第6步中的检查并并把这个结点给停掉了，因此我们看到的散播过程也不会完整。所以，这里我们在停止结点前等待3秒（足够了），即在t.Log("Stopping peers")之后紧接着插入一句time.Sleep(time.Second\*3)。

### 设置打印点

知道了TestDissemination所做的事情并做了适当的修改后，回归到我们打印点输出信息所要围绕的两点内容。这两点内容可以定位在一个结点**发送消息的出口**和**接收消息的入口**，在这两个口处设置打印点最容易收集消息，收集的消息也最能表明和佐证gosisp散播消息的过程和方向。发送消息的出口为gossip/gossip\_impl.go中的gossipInChan，接收消息的入口为gossip/channel/channel.go中HandleMessage的if m.IsDataMsg() || m.IsStateInfoMsg()分支。

**1.发送消息出口**打印点代码：

//行首有#的为原始代码，可据此定位。其余的即为自己写的打印点代码。

str := fmt.Sprintf("\033[41;32m%s have ",g.conf.ID)

for \_,mem := range membership {

str += fmt.Sprintf(" p%s,",mem.Endpoint[11:])

}

str = str[:len(str)-1]

str += "\033[0m"

strstr := fmt.Sprintf("\033[41;32m%s select ",g.conf.ID)

for \_,mm := range peers2Send {

strstr += fmt.Sprintf("p%s,",mm.PKIID[13:])

}

strstr += "send "

msgnum := 0

# // Send the messages to the remote peers

# for \_, msg := range messagesOfChannel {

# g.comm.Send(msg, peers2Send...)

mmm := msg.GetDataMsg()

if mmm != nil && mmm.Payload != nil {

strstr += fmt.Sprintf("SeqNum %d,",mmm.Payload.SeqNum)

msgnum++

}

# }

if msgnum != 0 {

strstr += fmt.Sprintf("time:%v\033[0m",time.Now().UnixNano())

fmt.Println(str)

fmt.Println(strstr)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

发送消息出口这个打印点做了3件事：（1）将一个结点有哪些成员名单记录到str中，形成类似**“p0 have p0 have p4, p2, p3, p1”**的字符串。由于测试代码中执行了checkPeersMembership等待，所以每个节点都会有其余4个结点的信息。（2）将一个结点把哪些块消息在什么时候（以纳秒计）发送给了谁记录在strstr中，形成类似**“p0 select p1,p4,p2,send SeqNum 1,SeqNum 2,SeqNum 3,SeqNum 4,time:1505618519599724473”**的字符串。（3）若msgnum>0，则说明发送了>=1条的DataMessage消息，则同时打印str和strstr。这里还需要解释的是，这个出口是很多类型消息的出口，因此4条消息有可能不会在一批消息中发送，可能emitter模块发来的一批消息中只有2条是DataMessage类型的消息。代码中的\033[41;32m和\033[0m是带颜色输出格式符号，下同。

**2.接收消息入口**打印点代码：

//行首有#的为自己写的打印点代码，其余为原始代码，可据此定位。

if m.IsDataMsg() || m.IsStateInfoMsg() {

...

if added {

# if m.IsDataMsg() {

# strstr := fmt.Sprintf("\033[42;31m%s recive SeqNum %d from p%s,time:%v\033[0m",

# gc.GetConf().ID,

# m.GetDataMsg().Payload.SeqNum,

# msg.GetConnectionInfo().ID[13:],

# time.Now().UnixNano())

# fmt.Println(strstr)

# }

// Forward the message

gc.Gossip(msg.GetGossipMessage())

// DeMultiplex to local subscribers

gc.DeMultiplex(m)

...

}

}

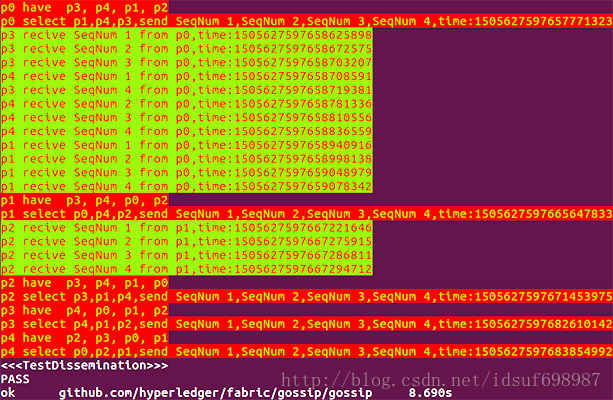
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

接收消息入口这个打印点只做了1件事：将一个结点在什么时候接收了来自谁的哪条消息记录到strstr中，形成类似**“p3 recive SeqNum 1 from p0,time:1505627597658625898”**的字符串并打印出来。这里还需要解释的是，应该在if added分支中（说明消息添加成功了，同时也说明这是第一次收到这条消息，重复收到同一条消息的话进不了这个分支，也不会有进一步操作），并判断接收的是DataMessage类型的消息时才打印信息。

如果时间更长，消息更多，结点更多且每个结点处理消息的效率差异更大，则有可能需要在if m.IsPullMsg() && m.GetPullMsgType()分支中设置打印点，以查看一个结点是否通过pull机制向其他结点索要block消息（因为这样的话这个结点正常的gossip散播步骤可能缺失一部分）。更复杂的，msgstore模块还会自动清理自认为过期信息，等等，这些都会影响上面所设置的打印点的输出。

### 测试并分析打印信息

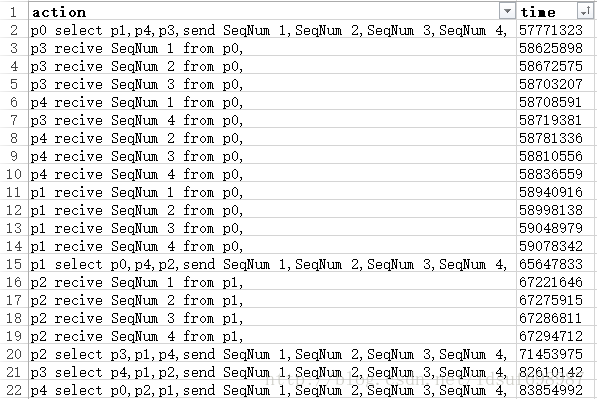
接下来我们就可以执行测试了，在终端中切换到gossip\_test.go所在的目录中，执行go test -test.run TestDissemination命令，输出的原始信息如下（成功的情况下，因每个结点的散播都是同时进行的，散播过程也是随机的，输出终端又只有一个，所以每次散播的顺序基本上不会相同）：



**红底绿字**的为**发送消息出口**打印点输出的信息，**绿底红字**的为**接收消息入口**打印点输出的信息（大红配大绿，最炫民族风）。从上面的信息可以看出：

* 每个结点都有其余4个结点的信息。
* 每个结点都会随机选择3个结点散播消息。这次测试中每个结点只散播了1次，SeqNum1-SeqNum4没有出现分批散播的情况。有时候一个结点因其他类型的消息侵占了emitter模块一批缓存的大部分空间，会把SeqNum1-SeqNum4分两次或更多次散播。
* 从上到下，从左到右，时间递增。从左到右的时间递增由gossip散播中循环的方法，打印点代码的方法和测试函数散播消息的循环的方法共同保证。

利用excel表对原始输出信息进行整理，把每个结点have的身份信息行删除，把时间点共有的前缀15056275976删除后按升序排列，整理如下：



以p3结点为例，在第3，4，5，7行，p3先后接收了来自p0的SeqNum1，SeqNum2，SeqNum3，SeqNum4消息，并依次调用了gc.Gossip()向emitter模块的缓存中添加消息，以求**继续散播**这4条消息。到了第21行的时间点，emitter模块由于攒足了一批消息（测试函数中设置的是一批20条，如果设置小点儿，应该会看到更零散的send发送信息），正好这4条DataMessage消息都在这一批消息中，所以p3随机选择了p4，p1，p2三个结点继续散播这4条消息。由于p4，p1，p2三个结点在21行之前均接收了4条消息，所以p3这次继续散播的消息即便到达了这三个结点中，这三个结点也不会added成功，也就不会再继续散播，而从p3来的这次继续散播也就到此为止了（第15，20，22行三个结点散播消息的动作不是因为p3的这次继续散播）。

其余的结点的接收和继续散播遵循同样的规则，均能相互映证。因而，测试对应验证了《fabric源码分析15》中所述的内容。

# fabric源码解析17——peer的chaincode之元数据

gossip初始化后，在start.go的serve函数中，执行了initSysCCs()和peer.Initialize(...)对**系统链码**（system chaincode）进行了初始化和部署。其实用户自定义的chaincode与系统链码只是人为上的角色的区分，在原理，实现，部署等方面并无二致。借此，展开对chaincode的讨论。之前的文章《fabric源码分析7》，《fabric源码分析8》都有涉及peer的chaincode，但是都不够深入，在此再次探讨，重点在于理清。由于chaincode是fabric项目中的中心概念之一，十分庞杂，因此分篇讨论。该篇文章旨在于认识和了解，主要讨论chaincode所涉及的各种承载chaincode数据的结构体和元工具，这些结构体和工具统称为chaincode的元数据。

在此和之后的文章，用户实现应用的chaincode（application chaincode）用**ACC**表示，系统chaincode（system chaincode）用**SCC**表示。对于chaincode的元数据描述过程中，可能会涉及到一些chaincode在具体操作时的结构上的概念，如chaincode的状态机，shim端等，这些暂不用在意，与之后的几篇同同讲chaincode的文章对看即可。同样，按照惯例，若读者找不到所述对象，文件或函数，请自行在chaincode所涉及的目录中利用grep命令等进行搜索。

## chaincode涉及的主要目录

* core/common
* core/scc
* core/chaincode
* peer/chaincode
* examples/chaincode/go
* protos/peer/chaincode相关的原型

## chaincode的元数据

1. **命令行Flag** - 在peer/chaincode/chaincode.go中，在init()中调用resetFlags()初始化了一众Flag，并把Flag的值存储在文件中定义的变量中，如chaincodeLang，chaincodeCtorJSON，chaincodePath，chaincodeName等，并在chaincode的每个子命令初始化时调用attachFlags添加子命令想要的Flag。
2. **policy** - 较复杂的Flag之一，chaincode中用到策略之一。chaincode部署时需要签名，该策略指定了对chaincode签名的时候都要有谁的签名才能生效。**策略原型字符串**是如"OR(AND('A.member', 'B.member'), OR('C.admin', 'D.member'))"这样的**嵌套结构**，OR(X,Y)表示X，Y二者取其一，AND(X,Y)表示X，Y二者都取，是组成嵌套结构的基础单位。X，Y是如A.member，C.admin这样的ORG.ROLE格式的组织成员，ORG表示一个组织MSP的ID，ROLE表示该MSP管理的成员角色（admin或memeber）。最终由common/cauthdsl/policyparser.go中的FromString()将策略原型字符串解析（主要使用了第三方库**govaluate**）存储在一个SignaturePolicyEnvelope（在protos/common/policies.pb.go中定义）对象中，该对象中的成员SignaturePolicy是一个**递归结构**，对应就可以存储嵌套结构的策略原型字符串：SignaturePolicy的存储SignaturePolicy\_SignedBy和SignaturePolicy\_NOutOf\_两种类型的对象，SignaturePolicy\_NOutOf\_是递归结构的，有成员SignaturePolicy和成员N，成员SignaturePolicy就是用于嵌套下一层策略的，成员N表示当前嵌套层的策略是AND还是OR，1表示OR，2表示AND。最终policy会在调用protos/utils/commonutils.go中的proto.Marshal()之后存储在chaincode.go中的变量policyMarhsalled中。
3. **chaincodeCtorJSON** - 较复杂的Flag之一，指定chaincode要运行的函数和函数的参数，原型为JSON格式的字符串，如{ "Function":"func", "Args":["param"] }。可以直接调用json.Unmarshal将原型字符串存入一个ChaincodeInput对象中，作为ChaincodeSpec的Input。
4. **chaincode说明书** - ChaincodeSpec，在protos/peer/chaincode.pb.go中定义，描述说明chaincode的结构体，简称为**CS**。成员有：Type指定chaincode的语言类型，当前版本的Fabric只支持go语言的chaincode；ChaincodeId指定了chaincode的路径，名称，版本；Input存储指定的chaincode的运行的函数和函数的参数。这些数据都来自于Flag（参照上述1-3点）。
5. **chaincode部署说明书** - ChaincodeDeploymentSpec，在protos/peer/chaincode.pb.go中定义，描述说明一个chaincode该如何部署，简称为**CDS**。成员有：ChaincodeSpec指定了第4点所说的chaincode说明书；EffectiveDate记录了chaincode何时有效的时间戳，即在chaincode开始运行时记录下时间点；CodePackage存储了一个.tar.gz压缩包的二进制数据，这个压缩包包含chaincode源码以及源码所依赖第三方库；ExecEnv标识chaincode的运行的环境，表明是运行在docker容器中还是操作系统之中。其中CodePackage经core/container/vm.go中的GetChaincodePackageBytes()，选择go平台的goPlatform对象后，最终调用core/chaincode/platforms/golang/platform.go中的GetDeploymentPayload，根据chaincode说明书中记录的chaincode信息，生成压缩包。这个过程的复杂之处主要在于收集chaincode源码所依赖的第三方库（这些依赖的第三方库应该放在GOPATH/src下），在搜集过程中，排除掉了GOROOT和fabric项目已经提供的库，也排除掉了chaincode源码目录路径中所有的vendor目录中的库，剩下的所依赖的第三方库都被重新映射到chaincode源码目录下的vendor目录。最终，将源码文件和所依赖的库都打进一个.tar.gz压缩包中返回。如chaincode源码的路径是GOPATH/src/hyperledger/fabric/examples/chaincode/go/chaincode\_example02/，假设该chaincode源码依赖第三方库github.com/jmoiron/sqlx（放在GOPATH/src/下，实际上不依赖），则生成的压缩包中的路径为src/hyperledger/fabric/examples/chaincode/go/chaincode\_example02/，chaincode\_example02目录下包含chaincode所有源码和一个vendor目录，vendor目录中包含路径github.com/jmoiron/sqlx/，sqlx目录中包含sqlx库全部的文件。
6. **ccpackfile包文件** - 一种由chaincode命令的package或signpackage子命令生成的chaincode二进制部署包。前者是由CDS对象生成，后者是由Envelope对象（包含签名过的CDS）生成。将这两者形成的ccpackfile使用ioutil.ReadFile被读入一个buf中后，可以使用CDSPackage对象或SignedCDSPackage对象的InitFromBuffer先将buf中的数据转入对象，然后调用对象的GetPackageObject从对象中抽取部署数据将其尝试转化为CDS或Envelope供部署时使用。这里和下文，我们将CDS和用于部署的Envelope都称作**部署数据**。
7. **CDSPackage/SignedCDSPackage对象** - 在core/common/ccprovider/下定义，也是一种存储chaincode部署说明书数据的对象，但同时也提供了一系列接口供各种情况下调用，所以更像是扮演了一种存储CDS数据的中间角色，以供把CDS数据转化为其他所需的格式。
8. **ChaincodeInvocationSpec** - chaincode调用说明书，简称为**CIS**，主要存储了一份chaincode说明书，只不过这份说明书可能是关于某一个系统链码的说明书，Input中存储的函数变为**install/upgrade**等对ACC进行操作的函数，而用户的部署数据（CDS或Envelope）则变为该函数的参数。比如用户要安装一个自己的chaincode，生成的调用说明书中，会使用到一份系统链lscc的说明书，而对于lscc说明书中的输入Input来说，操作的函数为install，函数的参数为用户chaincode的部署数据。
9. **Proposal/SignedProposal** - Proposal封装了chaincode的数据，如部署包等，作为一个申请消息，让结点签名后，会同签名数据一同放入SignedProposal，而SignedProposal就是chaincode可以通过grpc与结点进行通信的数据包结构，即ACC向结点发送执行交易的消息，都是以SignedProposal消息的形式发送出去的。
10. **CCContext** - 字面意思就是chaincode context，即链码内容。也是描述chaincode自身信息的一种载体，记录了chaincode的一些关键信息，如成员ChainID指定了所代表的chaincode所在的链的ID，成员Name指定了所代表的chaincode的名字，成员Syscc指定了所代表的chaincode是否是SCC。
11. **transactionContext** - 交易上下文，在交易中产生，以交易id为key记录在Handler的一个map中，随用随删。比如一个部署交易中，在Handler处理这个交易期间，会产生一个这样的交易上下文，存储关于这个交易的一些信息，供处理交易的主体使用。待该次部署交易完成，则会将其删除。
12. **Chaincode消息** - ChaincodeMessage，链对象的服务端和shim端进行消息交互的主要的消息载体，在protos/peer/chaincode\_shim.pb.go中定义。成员Type指定了消息的类型，有ChaincodeMessage\_REGISTER，ChaincodeMessage\_INIT等。成员Txid指定了该消息所在的交易编号（每个chaincode执行的交易都会有一个编号，每个交易会使用到多个类型的ChaincodeMessage在chaincode的服务端和shim端进行交互）。成员Timestamp自然是时间戳。成员Payload，Proposal是消息承载的chaincode数据，如源码包，执行的参数等等（只要是[]byte格式的，那其承载的数据就自由的多），可能有，也可能是空。成员ChaincodeEvent是chaincode在Init或Invoke时**给回**的所要触发的事件，比如，chaincode部署的时候，部署完毕后，shim可能给回服务端一个Event让服务端去触发，做一些必要的其他事情，同样，可能有，也可能是空。
13. **CCPackage/SignedCDSPackage** - 分别定义在core/common/ccprovider/下的cdspackage.go和sigcdspackage.go中，两者都实现ccprovider.go中定义的CCPackage接口，分别可以从一个Marshal过的CDS或Envelope抽取出其中所承载的关于ACC的数据来初始化自身，以方便SCC进行检查ACC的数据是否符合要求，最终也是以这种两类结构写入文件系统以保存ACC的。
14. **ChaincodeData** - 也是承载ACC数据的，用于ACC在系统中部署时，最终向账本提交安装的数据形式。
15. **StartImageReq** - 容器相关的数据结构，在core/container/controller.go中定义。包含了部署一个chaincode时启动一个容器所需要知道的如环境变量，网络ID，结点ID，部署包，建立函数，执行函数等等数据。相应的，有StopImageReq，DestroyImageReq两个结构体。
16. **ChaincodeStub** - 相当重要的一个结构体，接口定义在core/chaincode/shim/interfaces.go，实现在chaincode.go中。但是这个可以自己实现的，参见core/chaincode/shim/mockstub.go和mockstub\_test.go。这个结构体是链码执行Init，Invoke两个接口时直接使用的数据，也即一个交易到底要做什么的信息都会封装到这个结构里。

## chaincode的元工具

1. **Signer** - 签名者，由peer/chaincode/common.go中的InitCmdFactory初始化，使用的是一个结点的本地MSP服务对象bccspmsp中的signer，即一个SigningIdentity，可参看《fabric源码分析12》。签名者的作用就是调用CreateInstallProposalFromCDS和GetSignedProposal对chaincode的部署数据签名（CDS或Envelope），形成一个已签名的申请SignedProposal。
2. **EndorserClient** - 背书客户端，由peer/chaincode/common.go中的InitCmdFactory初始化。在《fabric源码分析11》中所提及的Endorser服务，所述的即为背书的服务端，该服务端是会随着一个结点的start.go的serve函数运行起来的。背书客户端的作用是调用ProcessProposal将签名者生成的SignedProposal发送给背书服务端**执行交易后背书**，获得一个申请应答ProposalResponse。ACC的交易，都是通过它来发起的。
3. **BroadcastClient** - 一个连接orderer服务的广播客户端。chaincode用其来发送交易的结果数据给orderer结点，供其进一步处理。
4. **ccprovider** - chaincode提供者，接口在core/common/ccprovider/ccprovider.go中的ChaincodeProvider，唯一的实现是core/chaincode/ccproviderimpl.go中的ccProviderImpl（ccprovider.go中初始化的也是这个实例，参看ccproviderimpl.go的init()）。这个工具相当于一个封装层，置于chaincode与各种结构的chaincode数据之间。比较明显的就是这个接口的方法中，很多参数都是interface{}类型。也即，各种不同目的各种形式的chaincode数据，要经过它，去往各个适合其该去的地方去做事情。
5. **ChaincodeSupport** - chaincode支持者，是一个全局单例，在core/chaincode/chaincode\_support.go中实现和定义。为整个chaincode的框架提供支持，更精确的说，是提供chaincode整个服务端的框架支持。相当于一个大杂烩，关于chaincode用到的，需要的，涉及到的，都可能在这里存储并引导执行chaincode的动作。但他也仅仅是这样，并不真正掺和chaincode的执行，也就是说，对于ChaincodeSupport来说，你chaincode需要的，我尽力给你，比如一个账本对象，但给你之后你怎么用你自己处理或账本对象自身如何处理交易，我就不管了，再比如，你要部署，我把你交给属于你的Handler之后我就坐等结果了，也不管。再比如，你需要何种类型的容器，我根据你的需要返回给你inproc容器或Docker容器对象，然后让容器对象自己跟你接触和交互，我也不管。
6. **container** - 容器，即chaincode最终要寄居的地方。有两种，inproc容器，docker容器。
7. **Handler** - 分为服务端的Handler和客户端的Handler，各自封装了一个FSM状态机，chaincode的交易就是由这两个Handler的状态机驱动的，相当于chaincode交易的控制器和发动机。
8. **txsimulator** - 交易模拟器，模拟交易发生的事情，其实就是在内存（典型的就是map）中记录交易产生的数据。在一个chaincode交易进行后，会将数据由交易模拟器记录一份，然后再把数据交给orderer服务处理（最终提交至账本）。类似的还有historyQueryExecutor这类的查询工具。

## 前奏

下一篇正式地开讲chaincode，由于chaincode比较庞杂，若文章混乱，模糊，错误之处，还请见谅。未定位的函数，源码文件，请参照本文罗列的chaincode相关目录自行使用grep，locate命令搜索，定位的函数或源码文件，均以这些目录为基础。

从上面就可以看出（上文罗列的只是些较关键重要的对象），承载chaincode数据的结构变化相当多，SCC还稍微好些，ACC确实相当复杂，而且某些字段会影响到逻辑判断，需要十分耐心。比较好的方法是**尽量用具体的例子和数据填充**，即**实例化原则**，这样有助于具象化，因此笔者在研究时自己画了数据组装图，分享在 <http://pan.baidu.com/s/1bpu91uf> ，密码s51j，根据这些数据组装的过程走，边看代码，不容易产生混乱，后文讲解过程中也会以依照这些图和图中的数据。图本身比较杂乱，不过在这种情况下也很难做到美观了，可以自行下载，文中就不再贴图。

# fabric源码解析18——SCC的安装和部署

SCC的注册和部署由于集成于peer内部，直接与用户交互之处甚少，因此也最简单。在此将注册和部署合为一篇。首先要理解的是，SCC和ACC本质是一样的，但ACC是处理用户的数据并根据用户指定的操作进行执行的，而SCC是处理ACC自身的，也就是说，对于SCC来说，ACC就是它要处理的数据。具体负责ACC的安装，部署等，就算是ACC要查个数据，也要经过SCC，让SCC去查，查出来后由SCC给ACC。之所以区分SCC和ACC，按照官方文档的解释，就是这样做可以使项目源码的弹性十足。先讲SCC是最合适的，一是SCC相对于ACC比较简单，二是SCC的机制也是ACC的机制，SCC代码走的线路，基本也是ACC代码要走的线路，熟悉了SCC，再入手ACC要顺利得多。很自然的，由于ACC要与开发者和使用者进行交互，因此要更繁杂一些。

## 概述

* SCC的Register相当于ACC的Install，SCC的Deploy相当于ACC的instantiate。这里和之后讲ACC，统一用Install和Deploy来描述两个动作。只有这两个动作执行完毕，一个chaincode才算真正可以使用。
* SCC启动占用的是**inproc容器**，可以当作就是内存里概念上的容器，在core/container/inproccontroller下实现，ACC启动占用的是**docker容器**，在core/container/dockercontroller下实现。
* chaincode接口是定义在**core/chaincode/shim/interfaces.go中的Chaincode**，只有两个接口：**Init(stub)和Invoke(stub)**，统一用同文件中的**ChaincodeStubInterface**接口实例作为唯一的参数。ChaincodeStubInterface接口唯一的实现是在同目录下的chaincode.go中的ChaincodeStub。
* 每一个chaincode都会起两个Handler，每一个Handler都是一个**以状态机（FSM）驱动的通信机器**，在驱动的过程中执行具体的状态事件，完成一个chaincode所需做的事情。一个可以称为服务端Handler（相当于服务端）实现在core/chaincode/handler.go中，另一个可以称为shim（shim本身有垫片的意思，可以理解为其是项目源码与开发者之间贴合的垫片）端Handler（相当于客户端），实现在core/chaincode/shim/handler.go中。后续文中统一以**ServerHandler**和**ShimHandler**区分。对状态机，即第三方库github.com/looplab/fsm的概念和操作不太熟悉的，建议复习学习一下（可参考《fabirc源码解析7》），这是你能看懂这部分源码的前提。若粗略的分一下的话，core/chaincode下的为chaincode服务端的代码，主要用于处理chaincode的请求；core/chaincode/shim下的为chaincode客户端的代码，用于定义供开发者使用的接口和客户端提交申请。
* 一个chaincode实质还是一个结构体对象，该结构体实现了Chaincode接口。SCC如core/scc/lscc/lscc.go中的LifeCycleSysCC，ACC如examples/chaincode/go/chaincode\_example01/chaincode\_example01.go中的SimpleChaincode。
* **SCC部署所涉及的图为SCC-Deploy-DataConstruct.PNG**。所有的SCC对象和原始数据定义在importsysccs.go中，下文仅以**lscc**这个SCC作为例子进行分析，lscc的chaincode对象LifeCycleSysCC具体实现在core/scc/lscc中。因chaincode自身已经比较复杂，因此与chaincode相关却不属chaincode主题的对象，文中将简单略过，待在之后相关主题文章中详述。文章开头部分可能讲的尽量详细，后边与前文重叠的部分将不再详述，如两个Handler之间的数据流转，第一次涉及到会比较详细的讲，之后就将粗笔述之。

## 安装

1. 在peer/node/start.go的serve函数中，调用了registerChaincodeSupport，这其中执行了scc.RegisterSysCCs()，针对每一个SCC依次执行core/scc/sysccapi.go中实现的RegisterSysCC(syscc)，进行注册（安装）。自然，也包括lscc。
2. if !syscc.Enabled || !isWhitelisted(syscc)，判断lscc是使能的且处于白名单之中，即lscc的Enabled要为true，且在配置文件core.yaml的chaincode.system项中，lscc要配置为enable（或yes/true）。如此，lscc的注册才能继续。
3. inproccontroller.Register(lscc.Path,lscc.Chaincode)，在core/container/inproccontroller定义，注册（即安装）lscc。上文说过，SCC启动占用的是inproc容器，这里Register所做的事情就是把lscc的Chaincode成员（lscc实际的chaincode对象LifeCycleSysCC）包装进一个inprocContainer容器，然后以lscc.Path为key放进typeRegistry这个map中，这就算注册（安装）完毕了。从这一点就可以体会，相较于ACC的安装启动一个docker容器，然后将ACC数据放入docker容器内指定的目录，inproc容器，只是一个叫法上的容器。typeRegistry这个map的作用也仅仅是存储注册的SCC了。

## 部署

1. 在peer/node/start.go的serve函数中，调用了initSysCCs()，这其中执行了scc.DeploySysCCs("")，针对每一个SCC依次执行core/scc/sysccapi.go中实现的deploySysCC(""，syscc)，进行部署。这里需要留意第一个参数，是空”“，这个值是赋值给**chainID**的，chainID的值对之后对每一个chaincode的一系列操作有很大的影响，自然，也包括lscc。这里提前说一下，同chainID，之后会遇到channelID，channel，chain这样的字眼，其实指的都是同一个东西，都是**链**的概念，一条链有一套属于自己的管理范围，工具，账本等，只有一个chaincode属于这条链，才能使用这条链的工具，账本，并受其管理，这个chaincode才能在链上起作用。SCC不属于任何一个链，它属于peer结点，也有权限，它只是处理ACC的请求，ACC在请求中提供chainID，SCC根据这个chainID从相应链上取出ACC所需要的工具或数据返回给ACC使用即可。
2. if !syscc.Enabled || !isWhitelisted(syscc)，如同安装一样，检测lscc是否使能且处于白名单之中。
3. ctxt := context.Background()，if chainID != "" {...}，声明了一个Context，这也是一个之后一路都在使用的变量（对于Context不熟的需要自行学习了）**ctxt**。然后if判断chainID是否为空，若不是空的，if分支中所做的事情就是查看一下这条链上的账本和账本交易模拟器是否正常。换句话说，若chainID不为空，即指定了lscc部署到哪条链上，那肯定是之后部署lscc的时候会用到这条链的Ledger和Ledger的TxSimulator（否则不会为了不做无用功而提前检查）。这也是一点可说的，算是一个小技巧：在追溯比较冗长的代码过程中，前期函数中使用的if判断，尤其是函数中开头处的if判断，也有很好的间接提示作用，在后期的函数中可以做到相互映证，来判断自己追溯路线的是否正确。比如这里的if，若chainID不为空，后期追溯的过程中我们发现并没有使用Ledger和TxSimulator，那基本就可以判断你追溯错了。这一小技巧在后文还会提及。当然，这里我们还是遵循尽量实例化的原则，chainID为空，因此该if分支不会被执行。
4. spec := &pb.ChaincodeSpec{...}，chaincodeDeploymentSpec, err := buildSysCC(ctxt, spec)，cccid := ccprov.GetCCContext(...)，根据原始的lscc数据，一路封装最后得到lscc的部署包CDS，CCContext，并将这些数据连同ctxt传入下一步准备执行部署。
5. ccprov.ExecuteWithErrorFilter(...)是chaincode执行交易的一个路线之一（另一个路线是core/chaincode/chaincodeexec.go中的ExecuteChaincode()，ACC走的是这条线），经ccprovider导航，执行调用到core/chaincode/exectransaction.go中的ExecuteWithErrorFilter(...)，lscc的数据也跟着到此，进而直接调用通文件中的Execute(...)，部署动作算是正式拉开序幕。需要说明的是，不单是lscc的部署，所有的交易，无论是ACC的安装部署，查询，Invoke等，最终都会到Execute(...)这里来开始，这里算是**交易产生和最终结束的地方**。同时也因此，要处理的情况更多，各种判断也会更多，而同样，遵循实例化原则，不会进入执行的if分支则不讨论。
6. Execute(...)接收的还是在第4步中所封装的lscc数据，利用这些数据，主要执行了两步：theChaincodeSupport.Launch()和theChaincodeSupport.Execute()。在Execute执行完后，会对交易返回的应答消息resp进行判断，若成功，将返回resp和resp中的“倒钩事件”（这个后文再提及）。下文将分别讲述Launch和Execute。

### Launch

1. 根据lscc封装的数据，cds不为空，cID = cds.ChaincodeSpec.ChaincodeId，cMsg = cds.ChaincodeSpec.Input，canName := cccid.GetCanonicalName()，从封装的数据中抽取出目前需要的数据，然后首先调用chaincodeSupport.chaincodeHasBeenLaunched(canName)进行if判断查看lscc是否已经被Launch过。这个if判断已经可以看出Launch这个函数最终要做的事情了：将lscc以canName（就是chaincode名字+’:’+版本号，即lscc:1.0.0，这算是chaincode内部认可的权威名字）为key，将lscc的对应的ServerHandler放入这个map中。这一点还是上文所说的技巧之一。当然，这里lscc是第一次部署，其数据不会存在于chaincodeMap中，所返回的chrte也自然为空。
2. if cds == nil { ... }，该分支不会进入，但是在ACC安装部署时会进入，这里不讨论。
3. if (!....userRunsCC || cds.ExecEnv == ..) && (...) { ... }，对照lscc封装的数据会发现会，该分支可以进入。分支中，判断if cds.CodePackage == nil会进入，但是if !(....userRunsCC || cds.ExecEnv == ....SYSTEM)不会进入。直接走到了builder := func() ...，定义了建立Docker容器的对象builder，并将Builder作为参数之一传入chaincodeSupport.launchAndWaitForRegister(...)，**真正开始Launch的工作**。注意的是，这个builder对于SCC来说，在之后不会使用，对于ACC来说才会用到。这里罗列一下进入launchAndWaitForRegister的参数的值：context，cccid，cds仍是上文第3，4步封装的数据（参看组装图，仍是进入Execute时的数据），builder是此步产生的，但之后不会再用。
4. launchAndWaitForRegister(...)从名字可以看出，是一个等待函数，即一直要等到Register完毕之后才会返回。这里的Register是指的lscc服务端的Handler的状态，即要等到lscc的ServerHandler处于REGISTER状态才会返回，对应该函数结尾处的select-case等待，顺理成章。在函数的开头，再次调用了chaincodeSupport.chaincodeHasBeenLaunched(canName)查看lscc是否已经被Launch过。然后开始准备数据，形成ipCtxt，vmtype，sir（包含CCID，Env，Args等数据），传入container.VMCProcess()（core/container/controller.go中定义），在其执行成功的情况下进入等待。这里需要注意几点要记清，后文要对应提及：**（1）**被包装进sir中的preLaunchFunc函数，负责将lscc的ServerHandler生成并放入chaincodeMap中。**（2）\*\*notfy是ServerHandler进入REGISTER状态的通知通道，在preLaunchFunc中被赋值，所赋的即是lscc的ServerHandler实例中的readyNotify通道。而select-case等待的就是这个notfy通道的通知，也即lscc的ServerHandler实例中的readyNotify通道的通知。**（3）\*\*ipcCtxt依然出自上文第3步创建的ctxt变量，一路跟着函数传递到这一步，这里又往里新加了ChaincodeSupport这个对象。
5. 在VMCProcess()中，首先获取了lscc使用的inproc类型容器对象v，然后新起了一个goroutine，非阻塞的在锁保护下将v传入StartImageReq（即上一步生成传入到这里的sir）的do函数，然后使用select-case等待do函数的执行结束。
6. do函数所做的，就是将StartImageReq自身已经初始化的每一个成员字段作为参数（第4步），依旧还有ctxt，传入lscc对应的inproc虚拟机对象v的Start()函数。这个Start()定义在core/container/inproccontroller/inproccontroller.go中，v的原型为InprocVM（Docker容器的虚拟机对象原型为DockerVM，在dockercontroller.go中）。
7. Start()函数中，ipctemplate := typeRegistry[path]，ipc, err := vm.getInstance(...)，所做的是先从上文lscc注册（安装）进typeRegistry的map（对看上文安装一节第3步）中取出lscc的chaincode实例，作为新生成的inprocContainer容器的成员，然后将这个属于lscc的容器放入instRegistry这个map中。if ipc.running则检查lscc对应的容器ipc是否已经处于运行状态，若重复部署lscc且能运行到这里，那这里就会返回错误。ccSupport, ok := ctxt.Value(...)则是从ctxt中取出之前放进去的ChaincodeSupport（对看第4步（3）），接着执行了prelaunchFunc()（对看第4步（1），（2）），最后ipc.running = true，置lscc的inprocContainer为运行状态，至此，**lscc就算是部署到位了**，之后再对lscc进行chaincodeHasBeenLaunched(canName)检查返回都会是true。但是虽然lscc已经部署到了容器中，但还是有一些关于lscc的两个Handler的初始化工作要做，Start()函数最后的最后，新起的goroutine中去执行了ipc.launchInProc，即是为此。新起了这个goroutine之后，Start()返回，一路返回，结束了第5步中VMCProcess()的等待，使得第4步的launchAndWaitForRegister(...)继续向下执行进入等待。标记一下，执行到这里，Launch的旅途中只剩下这一个select-case等待处。
8. launchInProc()函数层次分明，三段式，起了两个goroutine，一个去启动lscc的ShimHandler，一个去启动lscc的ServerHandler，最后使用select-case等待两个Handler初始化完成。从select-case等待结束的条件上看，两个goroutine任何一个结束（无论成功与否），都会导致函数结束返回。这里需要说明的是，由于SCC是运行在inproc容器中的，其实就是运行在内存中的，所以SCC的ServerHandler与ShimHandler之间直接使用的是go中的chan进行交互，即peerRcvCCSend和ccRcvPeerSend，并在core/container/inproccontroller/inprocstream.go中定义了inProcStream来模拟grpc流（只是模拟），实现了Send()和Recv()，这两个函数使用上述两个chan为两端的Handler收发消息（ServerHandler用peerRcvCCSend收，用ccRcvPeerSend发，ShimHandler则相反）。在此罗列一下传入用于启动ShimHandler的函数shim.StartInProc()的参数：env，args来自于第4步包含在sir中的数据，ipc.chaincode则为第7步lscc所部署的自身的chaincode对象实例。
9. 第一个goroutine，调用了shim.StartInProc(...)，在/core/chaincode/shim/chaincode.go中定义。在这个函数中，for \_, v := range env首先从env中获取lscc的权威名字赋于chaincodename（即lscc:1.0.0），stream := newInProcStream(recv, send)再利用第8步提到的两个chan生成模拟收发流，最后调用chatWithPeer(...)传入chaincodename，stream，chaincode（即lscc的chaincode实例），**运行ShimHandler**。在chatWithPeer(...)中：**（1）**handler := newChaincodeHandler(stream, cc)使用stream和chaincode创建了一个ShimHandler实例handler。**（2）**handler.serialSend(...)使用handler向ServerHandler发送一个ChaincodeMessage\_REGISTER类型的ChaincodeMessage，即要服务端进入REGISTER状态的消息。注意，由于第8步是连起两个goroutine去运行两个Handler，且工作量接近，因此两个Handler运行起来的时间不会差太多，由于是通过chan发送的，即便此时ServerHandler没有运行起来，ShimHandler也会进行阻塞等待。**（3）**发送完毕之后，新启一个goroutine在for循环中持续接收来自ServerHandler和自身状态机的消息，然后chatWithPeer利用<-waitc等待该goroutine结束。在这个for循环中，接收三个路径来的消息：一个是stream接收来自ServerHandler的消息后转发给msgAvail的消息，一个是自身状态机handler.nextState的状态消息，一个是stream发送消息失败的errc的消息。除errc通道接收到失败消息直接返回外，另外两个路径来的消息都会交予ShimHandler处理，即handler.handleMessage(in)。倘若是handler.nextState来的状态消息，还要再向ServerHandler发送，即else if nsInfo != nil && nsInfo.sendToCC分支中的内容。每次接收处理完一次消息后，for循环将重置一些变量，再次进入等待接收三路消息的状态。
10. 第二个goroutine，newInProcStream(...)先使用第8步提到的两个chan生成了模拟收发流，再使用这个流调用ccSupport.HandleChaincodeStream(...)，进而调用HandleChaincodeStream(...)（core/chaincode/handler.go中定义）创建了一个ServerHandler实例（如果你够细心，或者跟着步骤对看至此仍没有糊涂，你就会产生一个问题，即在前文的第7步，所执行的prelaunchFunc()函数，已经针对lscc在chaincodeMap中注册了一个Handler（也是ServerHandler），那为何这里又创建一个ServerHandler实例呢？这点下文会提及）。随即，执行了handler.processStream()，**运行ServerHandler**。processStream()函数的结构与第9步（3）所述ShimHandler的for循环结构十分类似，不再累述。
11. 从步骤中抽出来闲述一笔。两端的Handler的初始状态都为createdstate（创建状态）。ServerHandler端的代码是这么安排的：接收和处理（回复）消息都在core/chaincode/handler.go中进行，接收用handler.processStream()，处理（回复）用同文件下的各个ServerHandler的各个方法函数。ShimHandler端的代码则是这么安排的：接收消息在core/chaincode/shim/chaincode.go中进行，处理（回复）消息则在同目录下的handler.go中进行。接收用chaincode.go中的chatWithPeer，处理（回复）用handler.go中ShimHandler的各个方法函数。
12. 继续，第9步ShimHandler在启动前就向ServerHandler发送了一条ChaincodeMessage\_REGISTER类型的ChaincodeMessage消息，ServerHandler在启动后，于processStream()中接收到该条消息，交由handler.HandleMessage(in)处理，一看这条消息是REGISTER类型，直接就触发ServerHandler的状态机状态变为establishedstate（建立状态），同时触发了beforeRegisterEvent，enterEstablishedState两个状态机的事件函数，且前者一定先执行完毕后才会执行后者。**（1）**beforeRegisterEvent主要做了两件事：第一，handler.chaincodeSupport.registerHandler(handler)注册lscc当前的这个ServerHandler。第二，handler.serialSend(...)向ShimHandler回复ChaincodeMessage\_REGISTERED类型消息，注意这里加了ED，即表示注册过的意思。第二件事很好理解，而第一件事，就是第4步所提及的**等待Register完毕**所要等的事情，也对应第10步所留下的疑问。该函数所做的就是先将之前processStream()预注册到chaincodeMap中的Handler的readyNotify通道赋值给当前的ServerHandler的readyNotify通道，然后用当前的ServerHandler替换这个预注册的Handler，之后将ServerHandler的registered置为true，给txCtxs，txidMap分配内存，这就是Register所要做的事情。至于为什么用替换的方法，应该是为了在第4步留下预注册的Handler的readyNotify通道，使得第4步的launchAndWaitForRegister可以接收到ServerHandler中的通知（目前只能想到这个原因，至于是否还有更深层的原因，还请大神指点）。**（2）**enterEstablishedState主要做的通知的事情（此时经过（1）的执行，ServerHandlerd的readyNotify不可能为空），就是调用handler.notifyDuringStartup(true)向ServerHandler的readyNotify通道发送一个true值。由于（1）中所做的替换的事情，发送后，步骤7中的标记的还在select-case等待的launchAndWaitForRegister(...)会收到这个true值，立即结束等待并返回值为空的err，即表明ServerHandler已经Register成功，也即表明ServerHandler现在已经处于可以正常运行，收发处理消息的状态之中。
13. ShimHandler收到第12步（1）发送来的ChaincodeMessage\_REGISTERED类型消息，交由handler.handleMessage(in)处理，ShimHandler的状态机状态变为established，同时触发了beforeRegistered状态事件。beforeRegistered函数没有继续做任何事情。
14. 第12步（2）中enterEstablishedState执行之后，第4步launchAndWaitForRegister(...)成功返回，Launch()将于第1，2，3，4步后继续执行，调用了chaincodeSupport.sendReady(...)，目的在于由ServerHandler发起让自身和ShimHandler的状态机都进入ready状态，没有做多余的事情。在sendReady(...)中：（1）if chrte, ok = chaincodeSupport.chaincodeHasBeenLaunched(...)，判断lscc于此时是否被Launch过，注意这里用的是!ok，即此时lscc必须已经在chaincodeMap中被Launch了，否则将错误返回。（2）notfy, err = chrte.handler.ready(...)让lscc的ServerHandler发起进入ready状态的动作。（3）select-case等待notfy的通知，即等待两端Handler都进入ready状态的通知。
15. 在第14步（2）的handler.ready(...)函数中，txctx, funcErr := handler.createTxContext(...)创建了一个交易上下文对象transactionContext，将这个对象存储在lscc的ServerHandler的txCtxs这个map中，以txid（交易ID）为key。这里岔开说一下txid，之前步骤中一直没说。txid是部署之初生成的txid，部署也是一个交易，而每个交易都会有唯一的txid，整个交易的过程也都使用这个txid，以此来避免交易的重复和区分不同交易。这里所创造的以txid为key的transactionContext，会在ServerHandler进入ready状态之后从txCtxs中被删除。继续，生成了一个ChaincodeMessage\_READY类型的ChaincodeMessage消息，然后调用handler.triggerNextStateSync(ccMsg)，把消息发送到handler.nextState后handler.ready(...)立即将txid对应的transactionContext的responseNotifier通道返回，赋值给第14步（2）中的notfy，并进入（3）中的等待。
16. processStream()从handler.nextState中接收到ChaincodeMessage\_READY类型的消息，做了两件事：**（1）**交由handler.HandleMessage(in)处理，这个lscc的ServerHandler一看消息类型，状态机状态立刻由establishedstate变为readystate，同时触发enterReadyState状态事件函数，该事件函数只是通过notify(msg)，将消息转发给取出的之前第15步存放在txCtxs中txid对应的transactionContext的responseNotifier通道（这个通道经过第15步已经赋值给了第14步（2）中的notfy），而现在第14步（3）中等待的notfy收到这条消息后，结束等待，然后调用handler.deleteTxContext(cccid.TxID)将txid对应的transactionContext从txCtxs中删除后，第14步开始的chaincodeSupport.sendReady(...)结束返回，继而Launch()结束。至此，lscc的ServerHandler终于全部Launch结束。**（2）**由于handler.nextState路径接收到的消息，因此最后的if nsInfo != nil && nsInfo.sendToCC分支可以进入，也就可以将ChaincodeMessage\_READY消息异步发给了ShimHandler。
17. ShimHandler的chatWithPeer()收到来自ServerHandler的ChaincodeMessage\_READY消息，交由handler.handleMessage(in)处理，ShimHandler的状态机简单的将状态从established变为ready后，不会触发其他动作。至此，ShimHandler端的工作在Launch阶段也彻底结束。
18. 题外话：步骤的线条已经尽量的细了，不过还是有很多省略之处，这还仅仅是开头的一个Launch。经过这么一遭，各位可能对chaincode的交易的整个过程和冗杂程度稍有了了解，ACC的部署过程只会比这个更复杂。步骤都是在函数间跳来跳去，比较关键：第一，还是要清楚函数传入的数据是什么，函数返回的数据是什么。第二，熟悉一个消息在两个Handler之间是怎么流转和触发状态事件的。

### Execute

将目光重新定位到core/chaincode/exectransaction.go中的Execute(...)函数，重新查看进入Execute(...)的数据图SCC-Deploy-DataConstruct.PNG。在执行完毕theChaincodeSupport.Launch(...)后，继续。ccMsg, err = createCCMessage(...)根据Launch返回的CDS.CS.ChaincodeInput（实际为空）和之前的cctyp生成一个ChaincodeMessage\_INIT类型的ChaincodeMessage传入theChaincodeSupport.Execute(...)（下文若非特指，凡提到的Execute函数均指此函数）。在此罗列一下进入Execute的参数：ctxt和cccid仍是lscc部署之初生成的，ccMsg是当前生成的，executetimeout是ChaincodeSupport设置的超时时间。

1. if chrte, ok := chaincodeSupport.chaincodeHasBeenLaunched(canName)，再次检查lscc:1.0.0是否被Launch过，同时若已被Launch也返回了lscc对应的ServerHandler。到这一步，lscc:1.0.0必定已被Launch。
2. notfy, err = chrte.handler.sendExecuteMessage(...)，利用lscc的ServerHandler，目的在于运行ChainMessage中所携带的数据指向的动作，并返回通知通道给notfy。当顺利返回后，Execute(...)函数将进入常规的select-case等待（由此可知sendExecuteMessage中的具体的任务依旧会是异步执行）。
3. 在sendExecuteMessage函数中，handler.createTxContext(...)，依然用txid为key创建了一个交易上下文对象transactionContext（这个对象会在Execute结束前调用chrte.handler.deleteTxContext(msg.Txid)被删除），然后调用handler.triggerNextState(msg, true)将ChaincodeMessage\_INIT类型的消息发送到handler.nextState通道，然后将交易上下文对象的responseNotifier通道返回给Execute函数的notfy，并让Execute函数进入等待。
4. ServerHandler对于handler.nextState通道来的ChaincodeMessage\_INIT类型消息不会触发任何状态的改变和事件函数。只会把该消息原封不动的发给ShimHandler。ShimHandler收到此消息后，将触发beforeInit事件函数。beforeInit事件函数中实际执行任务的又是handleInit(msg)函数。
5. handleInit(msg)函数也是一个比较典型的处理流程（handleTransaction函数也是这种流程，ACC中会讲到）。函数整体只新起一个goroutine，在goroutine中，在defer中执行最后要发送的消息nextStateMsg（可能是正常的消息，可能是错误消息），然后定义了一个errFun函数，一旦检测有错，则返回一个ChaincodeMessage\_ERROR类型的ChaincodeMessage消息（即错误消息）给nextStateMsg，随即return，也就触发了defer。若一路都没有发生错误，则将一个ChaincodeMessage\_COMPLETED类型的ChaincodeMessage赋值给nextStateMsg，函数结束，也会触发defer。handleInit(msg)函数主要做了三件事：**（1）**stub := new(ChaincodeStub)，stub.init(...)组装ChaincodeStub（对于SCC来说，这个数据没用，但ACC会用到）。**（2）**handler.cc.Init(stub)，调用lscc的chaincode实例的Init接口初始化这个chaincode，也即ChaincodeMessage\_INIT类型的消息就是指示lscc做这个动作的。\*\*（3）\*\*defer发送ChaincodeMessage\_COMPLETED类型消息，当然这是在之前的步骤都成功的情况下。注意这里返回的ChaincodeMessage中给ChaincodeEvent成员赋值为stub.chaincodeEvent，这就是部署之初所提到的“倒钩事件”，这个“倒钩事件”会随着ChaincodeMessage一路返回到peer结点最核心的地方。但是目前SCC和ACC都没有明显的使用这一点，也算是留下的一个可升级的地方（看大神的代码就是精妙啊）。
6. 第5步（2）处调用的是lscc的chaincode实例的Init接口，定义在core/scc/lscc/lscc.go中，所做的事情相当简单（ACC会稍微麻烦点儿），就是给lscc的两个成员赋值，一个是系统链提供者实例，一个是策略检查器。然后就返回成功标识shim.Success(nil)。
7. 第6步执行完毕后，第5步的handleInit(msg)也会随之结束触发defer而再向ServerHandler发送ChaincodeMessage\_COMPLETED类型消息，ServerHandler接收到此消息后状态机也会无动于衷，只是在handler.HandleMessage(in)中会通过handler.notify(msg)将消息再转发给txid对应的transactionContext的responseNotifier通道。对看第2，3步，由此，第2步Execute(...)函数的等待结束，将收到的ChaincodeMessage\_COMPLETED类型消息resp返回。

将目光重新定位到core/chaincode/exectransaction.go中的Execute(...)函数，当theChaincodeSupport.Execute(...)执行完毕返回ChaincodeMessage\_COMPLETED消息，经过一些列的if判断，会进入if resp.Type == pb.ChaincodeMessage\_COMPLETED分支并成功返回。至此Execute(...)函数执行完毕，转回同文件中的ExecuteWithErrorFilter(...)，亦是成功返回。再一路回溯，一直回到最初的core/scc/sysccapi.go中的deploySysCC(...)，也是就此成功返回。就此，**整个lscc的部署过程结束**。

## 部署后的状态

1. theChaincodeSupport中的runningChaincodes.chaincodeMap中存在这以lscc:1.0.0为key的lscc的ServerHandler实例。
2. lscc的ServerHandler和ShimHandler均处于ready状态，且接收发送消息的goroutine都在运行当中。
3. 其他的SCC均与lscc状态相同。

## 遗留问题

* 版本数据问题：关于版本数据问题，笔者还未搞清楚，但无论对于SCC还是ACC来说，这都是个比较关键的数据，交易的一路都在使用。SCC所用到的版本数据是在deploySysCC中组装CCContext时，调用version := util.GetSysCCVersion()得到的，这个版本数据来自于common/metadata/metadata.go中的Version变量。然而这个变量整个项目中没有找到给其赋值的地方，按照注释的说法是来自于fabric的Makefile文件中的GO\_LDFLAGS，查看Makefile文件确有版本号相关的数据，如BASE\_VERSION = 1.0.0。但是笔者对Makefile不算精通，不清楚如何在编译项目时是如何将Makefile文件中的版本数据赋值给metadata.go中的Version变量的。还请精通Makefile文件的大神若也在研究fabric，予以指点。

# fabric源码解析19——ACC的安装

## 概述

* peer chaincode install命令执行安装命令，命令定义在peer/chaincode/install.go中，这也是安装的起点。另外需要注意的一点是，这个命令是在peer node start，peer channel create，peer channel join命令依次执行完毕之后所执行的，即执行install之时，peer结点的基本的模块（包括SCC）都已初始化完毕，channel也已经建立。
* 根据实例化的原则，从install\_test.go中提取了一句实际的install命令：peer chaincode install -n example02 -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode\_example02 -v anotherversion，本篇将以此句命令为例子。安装的ACC是example02，存在于-p指定的路径下，版本是anotherversion。-l没有给，自然取默认的值，为go语言。
* install安装使用的chaincode数据有两种形式，一种是CDS，一种是chaincode package/signpackage命令形成的ccpackfile包。因为我们没有涉及过这两个命令，因此这里我们只以前一种CDS数据包为例，叙述安装过程。
* ACC的安装涉及的图为**ACC-Install-DataConstuct.PNG**。下文中提及“图中”字眼，均指此图中。
* install能够识别的flag有-l，-c，-p，-n，-v。其中-c不常用，其指定的是ACC所要执行的函数和函数的参数，一般在具体执行的时候，如查询或转账的时候再给定。
* install最终所要做的事情，就是**将example02的源码包放入docker容器的安装目录中**。

## 生成签名申请包

以peer/chaincode/install.go的chaincodeInstall(...)为起点（设定ccpackfile==""），根据**ACC-Install-DataConstuct.PNG**，从左上角最原始的命令行数据开始，一路组装数据，至形成下面中路的SignedProposal，然后将SignedProposal通过cf.EndorserClient.ProcessProposal(...)提交至Endorser服务端。这个冗长的过程，不再详述。只提以下几点：

* chainID在SignedProposal中为空，即安装所要做的仅仅是把example02的压缩包放入容器的指定目录中而已，只有部署的时候才需要指定部署到哪条链上。
* CDS的CodePackage是example02的代码压缩包，包括源码和依赖的第三方库。因为SCC的一切都在项目编译时编进peer中了，所以SCC的CDS的这个字段就是空的，而ACC需要安装源码，所以这个字段在正常的情况下肯定不是空的。这个压缩包最终是在core/chaincode/platforms/golang/platform.go中的GetDeploymentPayload()打包的。该程序的复杂之处在于除了要打包example02源码，还要打包example02直接依赖的但go标准库未提供的第三方库，还要打包这些第三方库直接依赖但go标准库未提供的第三方库（即example02间接依赖但go标准库未提供第三方库）。这么做的目的就是让一个chaincode无论放到哪个容器里，不会因为缺少某个第三方库而编译失败。同时这个打包的过程也就要求我们在执行example02安装的时候，要实现把其依赖的第三方库事先放到GOPATH/src下。
* txid是在/protos/utils/proputils.go中的CreateChaincodeProposalWithTransient(...)中计算出来的，是哈希（peer结点的MSPID+证书元数据+随机数nonce）的值，可以就把它当作一个唯一的字符串，不用太过深究。

## 处理安装申请

1. Endorser服务端在core/endorser/endorser.go中的ProcessProposal(...)处，接收到来自Endorser客户端发送的**SignedProposal**和一个之后一路都会用到的Context上下文**ctxt**。
2. 在ProcessProposal(...)中，从开始至var txsim ledger.TxSimulator处，之上的代码全部是**一边解压抽取SignedProposal中的数据，一边验证这些数据**。至于抽取验证了哪些数据，根据代码对照图回溯，在此不做详述。
3. var txsim ledger.TxSimulator，var historyQueryExecutor ...，一个是交易模拟工具，一个是历史查询执行工具。由于是Install，且chainID为空，这两个值在之后都一直为空。if chainID != ""的分支也不会进入（当部署example02时，chainID不为空，则会进入此分支，根据chainID获取这两个工具，供之后部署使用）。
4. ProcessProposal(...)所做的主要的两件事就是：**（1）**e.simulateProposal(...)，模拟执行申请。**（2）**e.endorseProposal(...)，背书申请执行的结果。但是由于chainID为空，所以install命令不会执行此步（同样，部署时会用到），而是直接以ProposalResponse的形式返回（1）中执行的结果。下文将对（1）展开详述。
5. 在此撇开一笔说一下chaincode的交易使用的结果。该结果定义在core/chaincode/shim/response.go中，目前定义的还相当的简单（也为以后升级留了空间），只定义了三个：OK，ERRORTHRESHOLD，ERROR。其中ERRORTHRESHOLD算是错误标志线，值为400，即小于它的值，表示成功或者还能勉强接受且无伤大雅的异常，而大于等于它的值，则表示是不能接受的错误。

### 执行申请

1. 在此罗列一下传入e.simulateProposal(...)的参数：ctx为上文所述的上下文ctxt（至此未有更新）；chainID为空；txid为交易ID；signedProp是客户端发送来的原数据；prop是从signedProp中抽取出来的Proposal；hdrExt.ChaincodeId也是抽取出来的数据，只包含一个值为lscc的Name字段；txsim为空。
2. 在e.simulateProposal(...)中，前期又做了些简单的抽取和检查的事情：cis, err := putils.GetChaincodeInvocationSpec(prop)从prop中抽取出CIS。if err = e.disableJavaCCInst(cid, cis); err != nil通过判断example02的CDS.CS.Type来断定要安装的是否为Java源码，关于这点注释说的很清楚，当前版本不支持Java写的chaincode，但这部分在将来会被移除。if e.checkEsccAndVscc(prop); err != nil，escc和vscc对prop的检查，但是当前版本未作什么实际的检查，以后的版本可能会加入。if !syscc.IsSysCC(cid.Name){...}else{ version = util.GetSysCCVersion() }，由于cid.Name就是lscc，因此只会进入else分支，得到的为lscc的版本值为1.0.0。
3. 最后一步调用了e.callChaincode(...)，执行CIS指定的动作。上文所述的另一个函数endorseProposal()最后也是调用这个函数开始执行申请的任务的。也就是说，e.callChaincode(...)能实现什么效果，做什么事情，完全是传入的参数决定的，这个函数可以算是**实际开始执行申请的起点**。在此罗列一下进入e.callChaincode(...)的参数：ctx依旧为ctxt；chainID=空；version=1.0.0；txid为交易ID；signedProp/prop/cid/txsim不变；cis是第2步新抽取出的CIS。这些参数均在图中可以找到对应数据。
4. callChaincode()函数做了三件事：**（1）**cccid := ccprovider.NewCCContext(...)，根据传入的参数，创建一个CCContext对象供执行申请所用。**（2）**chaincode.ExecuteChaincode(...)，执行申请。**（3）**if cid.Name == "lscc" && len(cis.ChaincodeSpec.Input.Args) >= 3 && ...，这一步只有部署，升级的交易才会进入此分支，还记得上文的txsim和概述中提到的-c么，这里的分支就是执行这些对象所承载的任务的，在此不做讨论。下文将对（2）展开详述。
5. ExecuteChaincode(...)函数，在core/chaincode/chaincodeexec.go中定义，这里可以对看《fabric源码分析18》的部署章节第5步所述的线路问题。spec, err = createCIS(cccid.Name, args)又根据传入的参数新建了一个CIS，不过仔细看一下createCIS就可以知道，其新生成的CIS和第2步从prop中抽取出的CIS在内容上是完全一致的，因此我们完全可以将这个spec看作是图中的那个CIS。接着就调用了Execute(ctxt, cccid, spec)，对应到《fabric源码分析18》的部署章节的第6步，殊途同归，**ACC也就此进入了类似于SCC部署所述的机制和道路来进行example02的安装**。只不过，传入数据所承载的任务不同，执行的方向也会稍微有所不同。由于执行过程在《fabric源码分析18》详述过，因此下文将以粗线条叙述，重点在于提及不同之处。在此罗列一下进入Execute(...)的参数：ctxt依旧未有更新；cccid是第4步生成的，对应图中的CCContext；spec可以将其当作图中的CIS。
6. Execute(ctxt, cccid, spec)仍依次执行theChaincodeSupport.Launch(...)，theChaincodeSupport.Execute(...)，但这里spec是第5步生成的CIS，因此cctyp的值为ChaincodeMessage\_TRANSACTION，进而生成的供theChaincodeSupport.Execute(...)（下文若非特指，凡提到的Execute函数均指此函数）使用的ccMsg是ChaincodeMessage\_TRANSACTION类型的消息（对应图中的ChaincodeMessage）。依旧，分开讲解两个函数。

#### Launch

1. Launch(...)函数在ACC安装的情况下执行不了太久，canName := cccid.GetCanonicalName()等到的canName=lscc:1.0.0，此值会是程序进入if chrte, ok = chaincodeSupport.chaincodeHasBeenLaunched(canName); ok分支，进而进入if chrte.handler.isRunning()分支而返回。因为负责处理安装example02的lscc已经Launch过了，所以这样安排顺理成章。

#### Execute

1. 继续，在此罗列一下传入Execute(...)函数的参数：ctxt依旧位更新；cccid为图中的CCContext；ccMsg是图中的ChaincodeMessage；executetimeout为超时时间。
2. Execute(...)函数中，canName := cccid.GetCanonicalName()再次得到lscc:1.0.0，然后通过chrte, ok := chaincodeSupport.chaincodeHasBeenLaunched(canName)，获取了lscc在《fabric源码分析18》中已部署过的ServerHandler，再chrte.handler.sendExecuteMessage(...)开始使用该ServerHandler以触发状态机进入运行，最后进入select-case等待。在此罗列一下传入sendExecuteMessage(...)的参数：ctxt依旧未更新；cccid.ChainID为空；msg为图中的ChaincodeMessage；cccid.SignedProposal为图中的SignedProposal；cccid.Proposal为图中的Proposal。
3. 在sendExecuteMessage(...)中，通过调用handler.triggerNextState(msg, true)，ServerHandler将msg发给自身的handler.nextState通道以触发ServerHandler的状态机进入下一个状态。
4. ServerHandler的processStream()收到来自handler.nextState通道的msg，先交给handler.HandleMessage(in)处理，ServerHandler状态机无任何变化，然后handler.serialSendAsync(in, errc)给lscc的ShimHandler发送msg。
5. lscc的ShimHandler的chatWithPeer()收到msg，交由handler.handleMessage(in)处理，触发beforeTransaction事件函数，该事件函数主要调用同文件中的handleTransaction(...)函数。
6. handleTransaction(...)函数主要做的就是根据ShimHandler收到的msg生成并初始化一个ChaincodeStub，对应图中的ChaincodeStub，然后handler.cc.Invoke(stub)调用lscc的Invoke()方法对example02进行安装。
7. 在lscc的Invoke()中，args := stub.GetArgs()获取到的是CIS.CS.Input.Args。这个数组的值来自于protos/utils/proputils.go中createProposalFromCDS()中的case "install":中的ccinp。因此Invoke()中，function := string(args[0])得到function的值是"install"，switch function会进入case INSTALL:的分支。
8. case INSTALL:分支中，首先，lscc.policyChecker.CheckPolicyNoChannel(...)专门使用了检测未指定Channel的chaincode的函数来检查要安装的example02，这也侧面映证了上文**生成签名申请包**章节中对chainID的描述。接着，depSpec := args[1]取出来的就是example02的CDS，不过此时的CDS仍是被Marshal过的。最后，lscc.executeInstall(stub, depSpec)，调用lscc的函数，依据stub和example02的CDS，执行安装。
9. 在executeInstall(...)中，首先，ccpack,err := ccprovider.GetCCPackage(ccbytes)，根据example02被Marshal过的CDS创建一个CDSPackage（core/common/ccprovider/cdspackage.go中定义），对应图中的CDSPackage。其次，简单的验证了example02的Name和Version。最后，调用ccpack.PutChaincodeToFS()将example02源码写入文件系统。
10. 在PutChaincodeToFS(...)中，一系列if检查之后，先path := fmt.Sprintf("%s/%s.%s", chaincodeInstallPath, ccname, ccversion)整合出要写入的路径，即在chaincodeInstallPath目录下放入名为example02.anotherversion的文件。然后os.Stat(path)先查看这个文件名是否可用。最后ioutil.WriteFile(path, ccpack.buf, 0644)将CDSPackage中成员buf写入path指定的地方。这个buf，从图中就可知道，就是example02源码压缩包。至此，**example02的安装申请执行完毕**。由此，开始一路返回。
11. 一路返回至第6步ShimHandler的事件函数handleTransaction(...)中，handler.cc.Invoke(stub)返回，继续向下执行，将nextStateMsg赋值为ChaincodeMessage\_COMPLETED类型的消息，并执行defer中的handler.triggerNextState(nextStateMsg, send)将该消息发送给自己的状态机。ShimHandler只将ChaincodeMessage\_COMPLETED消息发送给ServerHandler之后就再无其他动作或变化。
12. ServerHandler收到ChaincodeMessage\_COMPLETED消息，通知仍处于等待之中的Execute(...)函数，然后等待结束，Execute(...)函数成功返回。

#### 一路返回

1. Execute(...)函数结束之后，就此一路返回，一直返回到core/endorser/endorser.go中的callChaincode(...)，chaincode.ExecuteChaincode(...)执行完毕，对应到上文**执行申请**章节的第4步的（2），由于这一步的（3）install申请不会执行，因此callChaincode(...)也就此结束。
2. 继续返回到simulateProposal，继续的代码中if txsim != nil分支不会进入，因此也是直接返回至ProcessProposal()。
3. 继续ProcessProposal()，将进入if res != nil分支，但无法进入if res.Status >= shim.ERROR分支。因此继续向下走，进入if chainID == ""分支对要返回给Endorser客户端的应答消息pResp赋值，最后返回pResp给Endorser客户端。
4. example02安装申请的起点，peer/chaincode/install.go中的chaincodeInstall(...)，所调用的install(...)中的cf.EndorserClient.ProcessProposal()，即是Endorser客户端，收到服务端发来的消息，返回后install(...)随之结束，进而chaincodeInstall(...)结束。至此，**整个example02的安装全部结束**。

## 安装后的状态

1. peer结点的chaincodeInstallPath目录下，会有一个名为example02.anotherversion的文件，该文件即为example02源码压缩包

# fabric源码解析20——ACC的部署

## 概述

* peer chaincode instantiate命令执行部署命令，命令定义在peer/chaincode/instantiate.go中，这也是部署的起点。另外需要注意的一点是，这个命令是在peer node start，peer channel create，peer channel join，peer chaincode install命令依次执行完毕之后所执行的，即执行instantiate之时，peer结点的基本的模块（包括SCC）都已初始化完毕，channel已经建立，ACC也已安装。
* 根据实例化的原则，从instantiate\_test.go中和官方文档中提取了整合了（尽量能用的flag都用上）一句实际的instantiate命令：peer chaincode instantiate -n example02 -v anotherversion -o orderer.example.com:7050 -C testchain -c '{"Args":["init","a", "100", "b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"，本篇将以此句命令为例子。-n指定部署的ACC是example02，-v指定版本是anotherversion，-o指定连接的orderer服务实例的端点是orderer.example.com:7050，-C指定要部署的链是testchain，-c指定执行的函数和函数的参数，-P指定策略。
* 同样，本文将重点放在不同于install之处。
* ACC的部署涉及的图为**ACC-Deploy-DataConstuct.PNG**。下文中提及“图中”字眼，均指此图中。
* instantiate能够识别的flag有-l，-c，-n，-v，-P，-E，-V，-C。相对于install，这里就指定了-c，要求部署的时候初始化a，b两个账户。
* ACC的部署是存储在docker容器中的，所启动的两端Handler通过grpc进行通信。
* instantiate最终要做三件事情，涉及三条链码：（1）lscc执行部署交易将example02的源码放入自己的写集。（2）example执行自身的部署交易，启动example02容器并与peer结点通过grpc通信进行初始化，最后将初始化的状态写入自己的写集。（3）获取lscc，example02的读写集，使用escc进行背书，然后将签名，读写集等部署产生的数据封装成Envelope，发送到orderer结点交由其处理。

## 起点

1. 起点在peer/chaincode/instantiate.go中的chaincodeDeploy(...)，主要做了两件事：**（1）**env, err := instantiate(cmd, cf)，与install命令执行的线路类似，对example02进行部署，并返回部署结果env。**（2）**cf.BroadcastClient.Send(env)，在部署成功的前提下，向各个结点广播部署结果env。下文将分别详述。

## 部署

1. instantiate()是部署的起点，同install一样，从图中左上角最原始的命令行数据开始，一路组装数据，至SignedProposal，依旧，可一边看图一边对看源码。这里需要注意的是，中途所组装的关于example02的CDS中，CodePackage为nil（这个很好理解，install已经将example02的源码放入指定目录了，部署的时候自然就不必再携带example02的源码数据）。chainID值为testchain，不再为空，即要把example02安装在testchain上。CIS.CS.Input.Args的值依旧来自于protos/utils/proputils.go的createProposalFromCDS()中的ccinp，不过不同于install，这次进入的是case "upgrade":分支（case "deploy"中执行了fallthrough），这点将直接影响lscc的Invoke所进入的分支，后文还将提到。
2. cf.EndorserClient.ProcessProposal(...)将组装好的SignedProposal，连同一个之后一路在用的Context上下文**ctxt**，一起发给Endorser服务端。
3. Endorser服务端在core/endorser/endorser.go中的ProcessProposal(...)接收到来自客户端的ctxt和SignedProposal。ProcessProposal(...)所做的事情依旧如install时所描述的那样，但不同与install之处在于，chainID := chdr.ChannelId获取的值为testchain，以至于之后三个if chainID != ""分支都会进入：**（1）**lgr := peer.GetLedger(chainID)，获取testchain的账本对象lgr，并调用lgr.GetTransactionByID(txid)对txid（交易ID）的唯一性进行检查（在这里是为了避免重复部署）。**（2）**调用txsim, err = e.getTxSimulator(chainID)和historyQueryExecutor, err = e.getHistoryQueryExecutor(chainID);，分别获取testchain的交易模拟工具和历史查询工具并赋值给txsim和historyQueryExecutor，同时将两个工具先后放入了ctxt中。**（3）**e.endorseProposal()将会执行，对example02的部署进行背书。在此罗列一下进入simulateProposal(...)的参数：ctxt被更新，此刻暂时只加入了历史查询工具；chainID/txid/signedProp/prop均未变；hdrExt.ChaincodeId对应图中的CIS.CS.ChaincodeId，只包含一个值为lscc的Name字段；txsim为testchain的交易模拟工具。
4. 在e.simulateProposal(...)中，不同于install所述之处在于，执行e.callChaincode(...)之后，会进入if txsim != nil分支，也会进入最后的if cid.Name == "lscc" && ...分支。在此罗列一下进入callChaincode(...)的参数：除了新抽取出来的cis，其余均未变。
5. 在callChaincode(...)中，会进入if txsim != nil分支，将第3步（2）中获取的交易模拟工具加入了ctxt。在此罗列一下进入chaincode.ExecuteChaincode(...)的参数：ctxt新加入了交易模拟工具；cccid对应图中的CCContext；cis.ChaincodeSpec.Input.Args对应图中的CIS.CS.Input.Args。
6. 在chaincode.ExecuteChaincode(...)中，依旧生成相当于图中CIS的对象，同ctxt，CCContext一同传入同文件中的Execute(...)函数，在这个函数中按部就班的开始Launch和Excute。
7. 在此省略与install的Launch/Excute章节相似的过程，一直到调用lscc的Invoke（core/lscc/lscc.go中），args := stub.GetArgs()获取得到的是第1步所提到的ccinp，function := string(args[0])得到的值是deploy，因而之后的switch-case会进入case DEPLOY:分支。在case DEPLOY:分支中，依次对args的每个值进行了检查，对一下参数进行了修补，如escc/vscc若为空，则给默认值，最后调用lscc.executeDeploy(...)，**开始部署example02**。在此罗列一下进入executeDeploy(...)的参数：stub为图中的ChaincodeStub；chainname值为testchain；depSpec为图中的CDS，但是此刻仍是被Marshal过的数据；policy为args的第3个参数，值为"OR ('Org1MSP.member','Org2MSP.member')"；escc/vscc由于命令行未指定，为空值。
8. 在executeDeploy(...)中，做了如下事情：**（1）**先检查了example02的名字，版本，是否可通过ACL（账户控制列表）。**（2）**调用lscc.getCCInstance(...)，查看example02是否已经存在于链上。**（3）**调用ccpack, err := ccprovider.GetChaincodeFromFS(...)将example02的源码读进一个CDSPackage对象ccpack，对应图中的CDSPackage。然后cd := ccpack.GetChaincodeData()，再由ccpack生成一个ChaincodeData数据对象，对应图中的ChaincodeData。**（4）**调用lscc.getInstantiationPolicy(chainname, ccpack)，lscc.checkInstantiationPolicy(...)分别获取并检查一个部署策略（不展开详述）。**（5）**调用lscc.createChaincode(stub, cd)，进而直接调用lscc.putChaincodeData(stub, cd)，传入图中的ChaincodeStub和ChaincodeData，**执行部署任务**。
9. 在putChaincodeData(stub, cd)中，简单的检查之后，就调用图中ChaincodeStub的函数stub.PutState(cd.Name, cdbytes)，以example02的名字为key，Marshal过的ChaincodeData数据为value，把这一对key-value放到账本中去。之后，图中又有没有地方了，数据组装就没有了。同时，从这里可以看出，链，其实就是peer中的账本。
10. stub.PutState(cd.Name, cdbytes)（core/chaincode/shim/chaincode.go中定义），调用了stub.handler.handlePutState(...)触发了lscc的ShimHandler的handlePutState()函数（stub的handler是在第7步省略的过程中，在core/chaincode/shim/handler.go中handleTransaction中生成ChaincodeStub时传入的lscc的ShimHandler实例）。在此罗列一下传入handlePutState()函数的参数：key，example02的名字；value，被Marshal过的图中的ChaincodeData；txid，交易ID。
11. handlePutState()中，所做的事情：**（1）**proto.Marshal(&pb.PutStateInfo{...})，首先将key和value封装，作为一个ChaincodeMessage\_PUT\_STATE类型的ChaincodeMessage消息的Payload，Txid依旧是txid。**（2）**handler.sendReceive(msg, respChan)，调用ShimHandler将ChaincodeMessage\_PUT\_STATE类型的消息异步发送给lscc的ServerHandler，然后进入select-case等待ServerHandler的回信。注意这里等待的respChan，是createChannel生成的，这个函数类似于ServerHandler的createTxContext，都是以txid为key在map中存储通知频道，防止交易重复，且随用随删。
12. lscc的ServerHandler收到ChaincodeMessage\_PUT\_STATE类型的消息，将只触发状态机的enterBusyState事件函数。该事件函数整个都是异步执行的。
13. enterBusyState函数一眼看上去很长很麻烦，所以先讲一下**函数的布局**：**（1）**使用defer作为最后发送消息的地方，发送的消息是triggerNextStateMsg。**（2）**如ShimHandler的handleInit函数一样，定义了一个errHandler函数，一旦检查有错误，即将triggerNextStateMsg赋值后返回，触发defer发送。若中途没有错误，则顺利到达函数的最后，给triggerNextStateMsg赋值一个正常的应答消息，然后随着函数的结束触发defer发送应答**（3）**中部的if大分支是函数的处理主体，分别处理ChaincodeMessage\_PUT\_STATE，ChaincodeMessage\_DEL\_STATE，ChaincodeMessage\_INVOKE\_CHAINCODE三类消息，对应执行不同动作，从名字基本就可以判断各是做什么的：放一个状态，删一个状态，调用chaincode（改一个状态），就是熟悉的**增删改**。接着，**函数的具体执行**：**（1）**handler.createTXIDEntry(msg.Txid)，这也是防止同一个交易重复执行的一招。**（2）**handler.isValidTxSim(msg.Txid...)，根据txid获取txContext，这个交易上下文transactionContext是在第7步省略的过程中，Excute(...)最初执行的sendExecuteMessage中的handler.createTxContext(...)创建的（对看《fabric源码分析18》章节Excute第3步），创建的同时，也将第3步，第5步更新到ctxt中的两个工具取出来后赋值给了txContext的txsimulator，historyQueryExecutor两个字段。这其中**交易模拟工具**在此之后将用到。**（3）**chaincodeID := handler.getCCRootName()取出来的是ServerHandler关于lscc的信息lscc:1.0.0。**（4）**只进入if msg.Type.String() == pb.ChaincodeMessage\_PUT\_STATE.String()分支（其他分支这里不讨论），txContext.txsimulator.SetState(...)，利用txContext中的**交易模拟工具**（回顾一下，这两个工具是在core/endorser/endorser.go中的ProcessProposal(...)中创建的，工具的原型是在core/ledger/kvledger/txmgmt/txmgr/lockbasedtxmgr/lockbased\_tx\_simulator.go中定义的lockBasedTxSimulator，SetState(...)函数也定义在同文件中），最终将key和value，连同处理example02的lscc:1.0.0一同写入到**写集**中。这里说的写集追踪一下就可以知道，其实只是个map，这个map的线路是：在core/ledger/kvledger/txmgmt/rwsetutil/rwset\_builder.go中的RWSetBuilder中的rwMap映射，这个map以lscc:1.0.0为键，映射一个nsRWs，而SetState(...)最终就是将key和value存储在这个nsRWs中的writeMap（写集）中。也就是说，对example02的部署目前并没有真正提交到账本（数据库）中，部署也是一个交易，自然也需要最终提交到账本中，只是目前还没到最终提交的时候。至此，**example02的chaincode完成了真正的部署**。**（5）**将交易的结果放入triggerNextStateMsg，然后触发defer，调用ServerHandler的handler.triggerNextState向ShimHandler发送携带部署结果res的ChaincodeMessage\_RESPONSE类型的消息（ServerHandler对这个消息会无动于衷）。
14. lscc的ShimHandler收到ChaincodeMessage\_RESPONSE类型的消息，触发状态机的afterResponse事件函数。该事件函数调用handler.sendChannel(msg)向第11步提到的respChan发送消息，第11步（2）的sendReceive(...)的等待结束。重新定位到core/chaincode/shim/handler.go中的handlePutState，sendReceive(...)结束返回后，自此开始一路返回。
15. 一路返回至core/lscc/lscc.go中的putChaincodeData(...)，对应第9步。继续返回，一直返回到同文件中的lscc的Invoke(stub)函数中的case DEPLOY:，lscc.executeDeploy(...)结束，整个Invoke(stub)函数的本次部署也执行完毕。返回的是return shim.Success(cdbytes)，cdbytes指的是图中Marshal过的ChaincodeData。
16. 继续返回，定位到core/chaincode/shim/handler.go中的handleTransaction函数，handler.cc.Invoke(stub)执行完毕。触发defer，向lscc的ServerHandler发送ChaincodeMessage\_COMPLETED类型的ChaincodeMessage消息，消息的Payload是第15步所返回的结果。
17. lscc的ServerHandler收到完成的消息，通知之后，core/chaincode/chaincode\_support.go中的Execute(...)等待结束，函数返回。继续一路返回，直至返回到core/endorser/endorser.go中的callChaincode中，chaincode.ExecuteChaincode(...)执行完毕（对应第5步），继续向下执行。
18. 进入if cid.Name == "lscc" && ...分支，对应第4步。cds, err = putils.GetChaincodeDeploymentSpec(...)从图中example02的CIS解压出图中的CDS，cccid = ccprovider.NewCCContext(...)并重新生成一个CCContext，最后将example02的CDS，CCContext和ctxt一同传入chaincode.Execute(...)，**第二次进入Launch-Execute过程**（这个过程图中没有体现）。这次进入没有通过core/chaincode/chaincodeexec.go中的ExecuteChaincode，而是直接调用了exectransaction.go中的Execute(...)。不同于第一次的是：**（1）**第一次传入的是lscc的CCContext和CIS，这次传入的是example02的CCContext和CDS。**（2）**第一次时Launch的lscc已经Launch过，中途就返回了，但example02没有Launch过，因此这次Launch将一直执行下去，建立example02的容器。**（3）**第一次传入的是lscc的CIS，因此生成的是ChaincodeMessage\_TRANSACTION消息，第二次传入的是example02的CDS，因此生成的是ChaincodeMessage\_INIT类型的消息。不同类型的消息将传入Execute。
19. 在Launch(...)中，这里提几个字段的值：（1）userRunsCC是在初始化ChaincodeSupport时写入的，引用的是core.yaml中chaincode的mode配置项是否是dev模式，这里默认是net，即userRunsCC的值为false。（2）example02也没有Launch过，因而chaincodeHasBeenLaunched(canName)获取的chrte为空。（3）example02的CDS的ExecEnv的值为默认的ChaincodeDeploymentSpec\_DOCKER。据此三点，可知整个Launch中只会进入if (!chaincodeSupport.userRunsCC ||...分支，接着进入if !(chaincodeSupport.userRunsCC ||...从文件目录中读取处install命令放入的example02的源码包数据，进而获取CDS，该CDS的CodePackage包含example02的源码数据，将供后文example02的docker容器的建立使用。builder = func() (io.Reader, error) { return platforms.GenerateDockerBuild(cds) }创建了一个builder函数。最后调用launchAndWaitForRegister(...)，开始对example的Launch。在此罗列一下传入的参数：context为ctxt；cccid是第18步生成的example02的CCContext；cds是图中的CDS；cLang是GO语言；builder是具体启动example02的容器的函数。
20. 对看《fabric源码分析18》Launch章节第4步，从此步起，开始了对example02的Launch过程，类似的步骤将省略。不同于SCC的Launch之处在于，builder将在部署中使用到，example02所启动的是docker容器DockerVM。一直追溯，将定位到core/container/dockercontroller/dockercontroller.go中的Start(...)函数，在此罗列一下传入的参数：ctxt，添加了ChaincodeSupport实例；其余参数都来自StartImageReq实例（在launchAndWaitForRegister中进行组装）的成员。
21. 在Start(...)函数中，启动了example02的docker容器：**（1）**imageID, err := vm.GetVMName(ccid)，根据ccid中保存的三个ID，组装一个example02的docker镜像ID，这个ID的规则是小写，字符范围在只有字母数字，-，.,\_之内，否则会用-替换，形式为%s-%s-%s。**（2）**client, err := vm.getClientFnc()，创建一个go-dockerclient的客户端对象，这个对象是实际进行docker容器的基础。**（3）**containerID := strings.Replace(imageID...)，根据镜像ID生成一个容器ID。**（4）**attachStdout := viper.GetBool("vm.docker.attachStdout")，获取一个配置项，这个配置项默认值是false，用于为调试目的而使能docker容器的标准输出和标准错误输出，这里使用默认值，即后边的if attachStdout分支（该分支起了两个goroutine分别接收容器的标准输出和标准错误输出）不会进入。**（5）**vm.stopInternal(ctxt,...)，根据example02的镜像ID，容器ID，尝试删除可能已经存在的同ID的镜像和容器，为之后的创建扫清障碍。**（6）**err = vm.createContainer(ctxt,...)，创建容器，罗列一下传入这个函数的参数：ctxt；example02的镜像ID，容器ID；供容器使用的参数args；要应用到容器里的环境变量env；不开启标准输出和错误输出的attachStdout。这个函数中主要做的就是首先根据现有数据的指向生成一个client认可且可以使用的docker容器配置对象copts，这个配置对象除了基本容器的基本信息外，还指定了一个配置函数，即getDockerHostConfig，然后调用client.CreateContainer(copts)创建容器。**（7）**进入if err != nil分支，事实上，（6）将执行失败，因为创建容器的基础是容器使用的镜像存在，而当第一次部署的时候，example02的镜像并不存在，因此将产生err == docker.ErrNoSuchImage的错误，在这个分支中，使用了builder先创建了example02的镜像，然后重新执行（6）创建容器**（8）**prelaunchFunc()预Launch一下example02，对看《fabric源码分析18》中Launch章节第7步。**（9）**client.StartContainer(containerID, nil)，启动example02的容器。这里还可以说一句，根据注释，通过配置对象创建容器的方式将在未来的版本中改变。这是docker自身的相关接口将在未来发生变化而产生的连锁反应。
22. 详解第21步**创建example02的镜像和启动容器的过程**。**首先概述一下**：**（1）**使用的是第三方库github.com/fsouza/go-dockerclient，读者可以自行对该库进行学习，这是能理解这一步内容的基础。**（2）**对看core/chaincode/platforms/util/utils.go中的DockerBuild注释，创建ACC的docker容器并不是简单的使用标准的docker build+Dockerfile的机制（因为这样产生的镜像有体积过大，有额外安全漏洞，运行笨拙等缺点），而是先积攒关于**example02的镜像数据（Dockerfile文件，peer结点的tls证书，编译后的可执行程序）**，然后创建一个相对轻量级的ACC容器（由此可以看出，对ACC的容器进行减负，主要是减去要为编译ACC而存在的部分，这部分通常使用较少，但占用的空间和资源又相对多）。**（3）**编译example02源码用到一个容器，这个容器将core.yaml中chaincode.builder项指定的**fabric-ccenv**作为启动镜像，该镜像由fabric项目提供，在Getting Started中下载镜像时会下载（这个容器有1G+，所以说上述的机制还是有必要的，不能为了最多M级别的源码一时的编译而一直运行一个G级别的容器），其实应该就是一个能编译example02的linux系统容器，ccenv，就是chaincode environment的缩写。粗略的过程就是先创建这个容器，然后把example02的源码上传到容器中，然后启动容器时执行go build…，然后再把编译好的可执行程序下载出来。**（4）**在core/chaincode/platforms下是平台相关的代码，用于生成支持的语言的ACC的镜像所需的数据包，platforms.go是总控文件，car、golang、java是平台相关的代码，这里只关注golang语言。**然后详述过程**：**（1）\*\*builder执行的是core/chaincode/platforms/platforms.go中的GenerateDockerBuild(...)，在这个函数中，先把example02容器通过tls连接peer结点的证书peer.crt放入inputFiles中，然后调用generateDockerfile来生成可用的Dockerfile文件（使用golang平台的GenerateDockerfile来创建了文件头，FROM命令指定example02最终使用\*\*fabric-baseos**镜像，由core.yaml中的chaincode.golang.runtime项指定，ADD将编译生成的example02的可执行程序压缩包binpackage.tar复制并解压到/usr/local/bin目录下，还定义了一些LABEL和环境变量等，该Dockerfile文件的范本也上传至网盘中），也将它放入了inputFiles中。**（2）**input, output := io.Pipe()生成了一个管道，连同go func(){...}中的gw，tw压缩对象，形成了**input<—>output<—gw<—tw**的数据流向管道，即向tw中写数据，最终会形成压缩包并流向**input**。**（3）**在新启的goroutine中，generateDockerBuild(...)汇总了example02镜像数据。先把证书和Dockerfile文件写入tw，然后调用golang的GenerateDockerBuild(cds, tw)，进而调用core/chaincode/platforms/util/utils.go中的DockerBuild，依据fabric-ccenv镜像创建了一个容器，创建该容器的选项DockerBuildOptions指定了三个值：Cmd指定了编译命令，将example02编译成名为chaincode的可执行程序并放入/chaincode/output；InputStream指定了输入流，该流为example02的CDS.CodePackage；OutputStream指定了容器的输出流，该流最后也通过调用cutil.WriteBytesToPackage写入tw，随后流向input。在DockerBuild中，所做的就是根据选项先检查fabric-ccenv是否存在，若不存在则尝试下载，然后创建、启动fabric-ccenv容器，然后等待编译完成，最后将编译好的chaincode从/chaincode/output/中下载到输出流OutputStream并删除fabric-ccenv容器。**（4）**异步执行（3）后直接将input返回。返回到第21步（7）处builder执行完毕将input返回给reader，对接上文，reader就是接收example镜像数据。然后通过调用vm.deployImage，把reader作为镜像的输入流（即上下文，可以理解为以此镜像使用Dockerfile运行容器时Dockerfile能使用的哪个范围下的数据），将example02的镜像部署。继续第21步的（7）向后执行。这里需说明的是，这里启动的容器是example02镜像的Dockerfile指定的fabric-baseos，且ADD命令会将binpackage.tar（即名为chaincode的example02的可执行程序的压缩包）复制并解压到/usr/local/bin目录下，再者createContainer创建该容器的时候，配置Config中Cmd的值（相当于Dockerfile中的CMD）是最初在core/chaincode/chaincode\_support.go中getArgsAndEnv(...)生成的args = []string{"chaincode", fmt.Sprintf("...}，所以当client.StartContainer(containerID, nil)启动fabric-baseos时（准确的说是exmaple02镜像，fabric-baseos只是其基础镜像）会执行example02的程序chaincode -peer.address=0.0.0.0:7051。这里要清晰的区分，执行client.StartContainer(containerID, nil)的是peer结点（这个结点可以宿存在主机中，也可以宿存在一个docker容器中），执行chaincode -peer.address=0.0.0.0:7051的是example02容器。
23. **创建example02的两个Handler**。在**新运行的example02容器**中执行example02的程序chaincode，参看源码examples/chaincode/go/chaincode\_example02/chaincode\_example02.go，执行的func main中直接调用了shim.Start(new(SimpleChaincode))，该函数在core/chaincode/shim/chaincode.go中定义，相当于部署SCC时调用的StartInProc（参看《fabric源码分析18》章节Launch第9步），旨在启动一个example02的ShimHandler并通过grpc主动发送一个ChaincodeMessage\_REGISTER类型消息给peer结点中example02的ServerHandler。传入的SimpleChaincode即为example02链码对象，相当于lscc的LifeCycleSysCC。具体的过程如下：**（1）**SetupChaincodeLogging()，设置viper在本容器内获取环境变量值的一些方法，如把前缀设置为CORE，把\_替换为.，这样viper.GetString("chaincode.id.name")就可以获取第22步最后启动example02容器时设置的Env中CORE\_CHAINCODE\_ID\_NAME=example02:antherversion的值，其次是获取其他的环境变量以设置日志输出级别等，这些环境变量均是最初在core/chaincode/chaincode\_support.go中getArgsAndEnv(...)生成的，一路被传至example02的容器配置中，对看第22步中的createContainer。**（2）**stream, err := streamGetter(chaincodename)，获取一个grpc流，这个流是连接peer结点的ChaincodeSupport客户端流。其中peer的地址是通过flag.StringVar(&peerAddress,"peer.address"...)获取的，对应第22步最后启动example02容器时执行的程序是chaincode -peer.address=0.0.0.0:7051，即通过flag给定了peer结点的地址，在此则通过flag获取这个地址。这一步执行过后，由于streamGetter中执行了chaincodeSupportClient.Register(...)   
    ，因此peer结点在core/chaincode/chaincode\_support.go中的gprc服务端的Register(...)函数将被调用，进而调用HandleChaincodeStream。HandleChaincodeStream新创建了属于example02的ServerHandler并调用handler.processStream()启动了循环接收ShimHandler消息的for循环。**（3）**chatWithPeer(chaincodename, stream, cc)，罗列一下传入该函数的参数：chaincodename值为example02:anotherversion；stream为连接peer结点的grpc客户端流；cc为example02链码对象SimpleChaincode自身。如同SCC的部署一样，通过chatWithPeer，先创建了属于example02的ShimHandler对象，将stream和cc赋值给ShimHandler相应成员，然后利用ShimeHandler向在peer结点中的ServerHandler发送了一条ChaincodeMessage\_REGISTER类型消息，最后启动了循环接收ServerHandler消息的进程。**（4）**（2）中的ServerHandler收到（3）中ShimHandler发送的ChaincodeMessage\_REGISTER消息，开始了注册的过程。这里的注册指的是用（2）新建的属于example02的ServerHandler把第21步（8）中prelaunchFunc()预Launch的Handler替换掉，这个过程省略，可对看《fabric源码解析18》章节Launch第12，13，14步。直到example02的ServerHandler和ShimHandler均达到ready状态。**至此，第二次进行的Launch-Execute，Launch部分执行完毕，开始返回**。中间过程省略，直接返回定位到对应第18步，core/chaincode/exectransaction.go的Execute(...)中，theChaincodeSupport.Launch(...)执行结束。
24. 继续执行theChaincodeSupport.Execute(...)，罗列一下传入的参数：ctxt，依旧包含这两个工具，交易模拟工具和历史查询工具；cccid，存放example02的数据和最初的部署申请数据，即图中的SignedProposal和Proposal；ccMsg，一个ChaincodeMessage\_INIT类型消息，Payload存放的是example02的CDS.CS.Input，即命令行-c指定的{"Args":["init","a", "100", "b","200"]，txid依旧是最初申请部署时的txid，Proposal在之后将被赋值为图中的SignedProposal；executetimeout超时时间。依旧对看《fabric源码解析18》章节Execute第1-5步，在此省略，直接定位到example02容器中运行的ShimHandler端core/chaincode/shim/handler.go的handleInit(msg)（ShimHandler接收到ServerHandler发来的ChaincodeMessage\_INIT消息，状态机触发beforeInit事件函数，进而调用handleInit(msg)），在这个函数中：**（1）**stub := new(ChaincodeStub)，stub.init(...)创建并根据收到的ChaincodeMessage\_INIT消息初始化了一个ChaincodeStub。**（2）**handler.cc.Init(stub)，调用了ShimHandler的cc（即example02的SimpleChaincode对象）的Init接口，可以定位到examples/chaincode/go/chaincode\_example02/chaincode\_example02.go中的Init(stub)。**（3）**在Init(stub)中，首先stub.GetFunctionAndParameters()获取了stub中的args中包含的函数和函数所用参数，即-c指定的函数init，参数a，100，b，200，分别看作a账户余额100，b账户余额200。然后stub.PutState(A, []byte(strconv.Itoa(Aval)))，stub.PutState(B, []byte(strconv.Itoa(Bval)))将两个账户的初始状态提交。下文只以A账户状态为例。
25. stub.PutState将触发ShimHandler的handler.handlePutState，之后的过程类似于第9-14步，只不过这时使用的ServerHandler和ShimHandler都是example02的，所提交的key是A的账户名，value是A的余额，最终也是将这一对key-value通过交易模拟工具在core/chaincode/handler.go的enterBusyState中提交到example02:anotherversion的写集（同第13步中的写集）中。然后返回到core/chaincode/shim/handler.go的handleInit(msg)中，随着handler.cc.Init(stub)的结束，触发defer发送ChaincodeMessage\_COMPLETED消息。ServerHandler收到后通知core/chaincode/chaincode\_support.go中的Execute(...)结束等待并返回，exectransaction.go中的Execute(...)也随之返回。**至此，第二次进行的Launch-Execute，Execute部分执行完毕，开始返回**。返回至core/endorser/endorser.go的callChaincode(...)，对应第18步。继续返回至simulateProposal()中，接着进入if txsim != nil分支执行了txsim.GetTxSimulationResults()，获取了交易的**读写集**（这里主要是写集中的数据，是由第13步中lscc的写集写入的example02的链码数据ChaincodeData，此步中example02的写集写入的A/B两个账户的状态数据，当前操作的读集里面没有数据），然后返回，至ProcessProposal(...)中，这里罗列一下e.simulateProposal(...)返回的数据：cd为空；res为lscc部署example02时的返回结果Response，成功的结果中包含example02的ChaincodeData；simulationResult，交易的读写集（即目前所进行的交易的结果）；ccevent为lscc部署example02时最终返回ChaincodeMessage\_COMPLETED消息时所携带的ChaincodeStub中定义的事件，这里为空（core/chaincode/shim/handler.go的handleTransaction中）。继续，由于chainID不为空，将继续执行e.endorseProposal(...)，在此罗列一下传入该函数的参数：ctxt；chainID，值为testchain；txid，交易ID；signedProp/prop对应图中的SignedProposal和Proposal；cd/res/res/simulationResult/ccevent均为上文返回的数据；hdrExt.PayloadVisibility为空；hdrExt.ChaincodeId只包含一个值为lscc的Name字段。
26. 在endorseProposal(...)中，主要做的就是生成一个供escc使用的CIS，然后通过再次调用callChaincode来执行背书。具体过程如下：**（1）**确定要使用的进行背书的SCC的名字escc和版本号。**（2）**根据参数和准备的数据，生成一个ecccis，这个作用类似于ExecuteChaincode中的createCIS和图中的CIS，罗列一下Args的值：【0】函数名，为空；【1】图中Proposal的Header；【2】图中Proposal的Payload；【3】[]byte格式的ccid，这个ccid的Name值为lscc，version值为1.0.0；【4】[]byte格式的包含example02的ChaincodeData的成功返回结果；【5】交易读写集数据；【6】事件，为空；【7】payload的权限控制，为空，当前版本对这个字段并没有使用。后边的背书过程中所用到的数据均来自于此。**（3）**调用callChaincode。
27. 在callChaincode中，这次只会执行一次Launch-Execute过程，且Launch会中途返回，因为escc已经被Launch过。中途的过程对看《fabric源码解析19》执行申请章节第5步之后，一直对看到到Execute章节的第6步，只不过这期间一直使用的是escc的两个Handler，最后调用到的是escc的Invoke()方法对example02的交易结果进行背书。直接定位到core/scc/escc/endorser\_onevalidsignature.go的Invoke，传入Invoke的stub是在core/chaincode/shim/handler.go的handleTransaction中生成的。
28. 在Invoke中，只做了两件事：**（1）**分别将携带的Args中的每个值都解压出来，在此可以与第26步（2）处的对看，看看都是哪些数据。**（2）**根据解压出来的Args中的值，**签名**并整理应答数据，最后返回这个数据，这里不再详述。自此一路返回，过程省略，直接定位到endorseProposal中的e.callChaincode(...)执行完毕，对应第26步的（3）。接着返回至ProcessProposal()，至此，ProcessProposal()全部执行完毕，返回的数据是protos/utils/txutils.go中CreateProposalResponse生成的ProposalResponse，且该ProposalResponse的Response.Payload被赋值为example02的ChaincodeData。这里详解一下ProposalResponse的构成：**（1）**成员Version，明确版本固定为1。**（2）**成员Endorsement，包含了结点的签名者Endorser和签名者的签名Signature。**（3）**成员Payload，[]byte格式的ProposalResponsePayload，包含[]byte格式的ChaincodeAction，哈希过的图中的Proposal。ChaincodeAction相当于在描述一个chaincode动作，即谁干了什么事儿，产生了什么后果，包含的有：ChaincodeId里的数据是lscc和1.0.0，Response数据是lscc部署example02是返回的res，其中Payload包含了example02的ChaincodeData，Results则包含了lscc部署完example02后两条链码的读写集，Events为空。**（4）**一个Success的Response，Payload为example02的ChaincodeData。这样的一个ProposalResponse被返回给peer/chaincode/instantiate.go的instantiate中，cf.EndorserClient.ProcessProposal(...)执行完毕。
29. instantiate继续执行，utils.CreateSignedTx，结合图中的Proposal，返回的ProposalResponse，生成“一封信”，即可供cf.BroadcastClient.Send(env)使用的common.Envelope。Envelope数据的封装过程是一个具体数据到被一层层包装进一个通用数据的过程，也需要耐心。

## 广播

1. 在peer/chaincode/instantiate.go的chaincodeDeploy中，env, err := instantiate(cmd, cf)返回的Envelope紧接着被cf.BroadcastClient.Send(env)发送进行广播。
2. peer命令的广播客户端为在peer/common/ordererclient.go中的broadcastClient，封装了一个grpc连接和一个AtomicBroadcast\_BroadcastClient客户端对象（这个对象其实是AtomicBroadcastClient客户端的一部分，即Broadcast流客户端，在protos/orderer/ab.pb.go中定义）。从所在的文件名就可知，这是连接orderer服务的客户端。ordering服务客户端的地址是命令行中-o指定的，若未指定，则是在peer/common/common.go中的GetOrdererEndpointOfChain，通过向cscc发送请求获取的（这里又是一大堆文字）。这个连接在instantiate命令执行之初就已经被建立。
3. cf.BroadcastClient.Send(env)将env发送到了orderer/server.go的Broadcast(...)处接收，自此数据交给了orderer服务中进行处理。由于涉及到orderer服务，这里只延续讲大概过程：即orderer服务将env数据排序后发送给peer结点的gossip服务开始散播，散播后，最终提交到网络中的每个结点的链（账本）上。这点在讲述orderer服务的时候会再次提及。

## 部署后的状态

* example02的对应图中的ChancodeData数据被放入交易模拟器的写集中。
* example02的镜像被创建，镜像的上下文包含Dockerfile文件，通过tls连接peer结点的证书，可执行程序压缩包。
* example02的容器被创建，单独运行ShimHandler，并通过grpc与peer结点中ServerHandler通信。
* example02的ServerHandler在peer结点的theChaincodeSupport.runningChaincodes.chaincodeMap中进行了注册。
* example02的ShimHandler和ServerHandler均处于ready状态。
* example02指定的{"Args":["init","a", "100", "b","200"]，即A账户余额100，B账户余额200，这两个状态被放入交易模拟器的写集中。
* example02部署的数据通过orderer服务排序后散播到网络的各个有效结点中并最终提交到各自的链（账本）上（这一点本不是peer部署example02本身做的事情，只是会促成的orderer服务和gossip服务要做的事情）。

## 后记闲言

ACC的部署至此结束，相当复杂，复杂处主要在于的是各层数据之间的包装、传递、解包验证和各自链码的两个状态机之间的缠斗，需要对照图中数据一点一点的磨。文中如有纰漏错误之处，可以留言指明。

# fabric源码解析21——撇开的一笔

## 概述

看fabric到现在快接近一年，逐渐形成系列，创造出来了一点儿东西，自己还是挺开心的。这期间也有不少有同样兴趣的朋友加过我，不过平时比较忙，交流不多。所以产生了建立一个群的想法，各位多交流。群号：655438051，欢迎各位加入。

讲一个撩妹的故事，是一个插曲，黑幽默一下。

在一个莫名的时代，年份被划分两种，一种年份叫做**木一年**，一种年份叫做**石页年**。**1石页年==1.25木一年**。有一个姑娘，自己也不是专业的作家，但是特别喜欢写小说，写写画画的，也会把自己的小作品放到网上，记录自己想写的东西，也为分享一下。

有一个高富帅叫桦沩，是个小说迷，一心想找一个女作家当女朋友，觉得当作家的女孩子特别有修养，特别有气质。有一天在网上闲逛无意中看到了那个姑娘的小说，一看写的还不错，字由心生，字里行间透露着不错的修养和气质，觉得会比较满足自己的要求，于是就寻找这个姑娘的联系方式，然后开始接触。不过稍稍接触了一点，就发现这姑娘没有自己想象中那样有蕙质兰心的文学气质啊，也不是个正牌的作家，按木一年算来，她连五年的经历都没有，他觉得我这高富帅这么好的条件，是有自己的标准的，完全可以找个更好的，于是很快就过客般的离开了那个姑娘。

之后故事就结束了，就是匆匆的一瞥，姑娘依旧自己的日常，没有再见过桦沩。人的一生，是宇宙时间之中小数点后无法想象的多的0之后的那个1。而这匆匆的一瞥，又成为她生命中小数点之后若干个零之后的那个1。

# fabric源码解析22——Orderer服务的初始化

## 概述

orderer服务是单独拥有说明文档的组件，这说明这个服务应该是相当独立的。orderer提供的服务既涉及到chaincode，也涉及到账本，因此这时讲最合适。先回顾一下下列内容：

* 《fabric源码解析1》的**验证线头**章节第3，4点，关于启动orderer服务容器的内容。现在版本中启动orderer服务容器的文件为examples/e2e\_cli/base/docker-compose-base.yaml。
* 《fabric源码解析14》的**gossip服务器的初始化**章节第（2）点关于初始化频道的内容，**election模块**章节关于分发数据来源的内容。

从以上回顾可以明确：**（1）** orderer服务是先于所有peer结点启动起来的单独的结点。**（2）** orderer服务与gossip服务密切相关。这里可以概述一下orderer所提供的服务，只做broadcast和deliver。orderer结点与各个peer结点通过grpc连接，orderer将所有peer结点通过broadcast客户端发来的消息（Envelope格式，比如peer部署后的数据）按照配置的大小依次封装到一个个block中并写入orderer自己的账本中，然后供各个peer结点的gossip服务通过deliver客户端来消费这个账本中的数据进行自身结点账本的同步。**数据流向：**   
peer结点->grpc->Server->broadcast/deliver->broadcastSupport/deliverSupport->multichain->chainSupport->kafka/solo->chain->BlockCutter->chainSupport.CreateNextBlock->chainSupport.WriteBlock->ledger账本->kafka->peer结点。

## 目录和图片

在本文即后续讲解orderer的文中所涉及的图片都上传到了网盘，<http://pan.baidu.com/s/1bpu91uf> ，密码s51j，涉及的目录如下：

* common/
* orderer/   
  + common - orderer服务公用代码，主要包含了切个块的工具blockcutter，具体的broadcast/deliver处理程序，各个过滤器（根据一些标准过滤要装进block的消息）。
  + configupdate - 升级配置代码。
  + localconfig - 本地orderer配置。
  + ledgger - orderer可使用的各类账本。
  + kafka - 卡夫卡客户端。
  + solo - solo服务端。
  + multichain - 总服务端，可将其视作一个总管，orderer直接使用此对象来控制各种服务。
  + server.go - orderer grpc服务端代码。
  + main.go - orderer初始化启动代码。
* protos/orderer
* sampleconfig/（配置文件）

## kingpin

orderer服务的命令行使用了第三方库kingpin，v2版本，地址为gopkg.in/alecthomas/kingpin.v2。这里orderer不再像peer那样使用cobra的原因就不得而知了。kingpin本身比较简单，而且orderer也只使用了kingpin最表面的一层。只有两个子命令，start和version，且没有任何flag或参数。version不用讲，start是默认命令，即《fabric源码解析1》中提及的docker-compose-base.yaml中启动orderer结点容器时所执行的命令。当前版本中，docker-compose-base.yaml在examples/e2e\_cli/base下，且执行的command是orderer，但是start是默认命令，所以执行orderer相当于执行了orderer start命令。

## 模块

* Server - 服务自身，包括broadcast和deliver。有以下三种类型的服务，分别用于不同情况下的部署：**（1）** Solo ordering service，Solo，针对开发测试环境的一种服务，特点是简单，不支持consensus，不够高效且不可测量。**（2）** Kafka-based ordering service，卡夫卡，一种经典的出版-订阅服务。orderer中使用了第三方库github.com/Shopify/sarama创建客户端，以连接kafka。在fabric Getting Started操作下载的镜像中，有一个kafka镜像以作为服务端处理消息（还有一个zookeeper镜像辅助kafka）。在产品级别的项目部署中，这种服务是当前最好的选择，有很高的吞吐量，非常的高效，但是不支持容错。理解orderer服务中的kafka客户端代码需要专门了解一些关于这个系统的概念，详见下文。**（3）** PBFT ordering service，拜占庭容错，当前处于开发之中，文档中的意思虽未明说，但意思明显是这种服务可以兼顾高效和容错。当前版本没有相关代码，不予讨论。这个模块是服务主体，包含多个子模块，如cutter，kafka，过滤器等。
* Ledger - 账本。orderer服务必须允许客户端能够在排序过的块流中进行查询，因此orderer服务需要一个支持账本。有以下三种类型的账本，分别用于不同情况下的部署：**（1）** File ledger，文本账本，针对产品级别的项目部署使用，默认选择此类账本（下文也只讨论这类账本）。账本直接存储在文件系统上，通过轻量级LevelDB数据库以Index进行索引，以供客户端高效的检索指定的块。**（2）** RAM ledger，内存账本，可配置用于存储的内存的大小，即一切数据都存于内存中，针对测试，不容错，重启后将重置。**（3）** JSON ledger，JSON账本，即用JSON文本存储账本数据，也是针对测试环境的，特点在于简单明了，且可以容错，重启后不会重置。
* Profiling - 监控orderer服务的模块，可供通过http（如浏览器）来监控orderer的情况。暂不讨论。

## 配置

配置是orderer服务启动过程中所必须用到的，也直接影响了orderer如何服务。主要涉及到orderer.yaml，configtx.yaml两个配置文件。

* orderer.yaml - 用于orderer服务自身。orderer.yaml的数据使用main.go中main函数中的conf := config.Load()被加载，形成了localconfig/config.go中的TopLevel对象conf（common/configtx/tool/localconfig/config.go中也有对应的一套TopLevel），过程稍显复杂，多个结构体一级一级的嵌套，但较方便的地方是取出来具体的某一项配置的值的变量与orderer.yaml中对应的配置key一样，如要取出orderer.yaml中GenesisFile项的值，则使用conf.General.GenesisFile即可，且config.go中有一个默认的TopLevel对象defaults作为默认配置，可做参考。
* configtx.yaml - 主要供orderer生成genesisblock作为orderer使用的链的第一块block。在main.go中initializeBootstrapChannel函数中生成genesisblock时，genesisBlock = provisional.New(genesisconfig.Load(conf.General.GenesisProfile)).GenesisBlock()（这个函数相当于Getting Started使用configtxgen手工生成的genesis.block，使用哪种方式生成gensisblock由orderer.yaml中的GenesisMethod项和启动orderer容器时所给的环境变量，即docker-compose-base.yaml中ORDERER\_GENERAL\_GENESISMETHOD项所决定，后者优先权应该大于前者），configtx.yaml被genesisconfig.Load(...)加载，形成了ConfigEnvelope被写入genesisblock中。这个过程也是相等复杂，经历了多成结构的嵌套，具体参看图片gensisblock.png。基本的数据结构嵌套变化流程是：common/configtx/tool/localconfig/config.go中的TopLevel –> provisional/provisional.go中的bootstrapper（ConfigGroup） –> common/configtx/template.go中的Template –> Envelope（ConfigEnvelope） –> Block（genesisblock）。而具体哪些数据写入了gensisblock，可以求助于test函数将这个ConfigEnvelope打印出来。multichain/manager.go中的multiLedger对象是orderer服务直接使用的总管，所以初始化这个对象可以执行到orderer服务的各个子模块，自然也包含生成gensisblock这一步，所以在manager\_test.go的TestManagerImpl函数中，有初始化这个对象，可以以此测试。打印点设置在common/genesis/genesis.go的Block(...)中的err = proto.Unmarshal(configEnv.ConfigUpdate, configUpdate)下面，使用fmt.Printf("\033[43;36mconfigUpdate:%+v\033[0m\n",configUpdate)以黄底绿字的形式打印。打印的数据已放入configupdate.txt稍作整理并上传至网盘。由打印的数据可知，写入gensisblock的配置数据主要是configtx.yaml中orderer部分的配置数据。

## 模块的初始化

### Server

* gprc服务 - orderer所基于的gprc服务AtomicBroadcast原型在protos/orderer/ab.proto中定义，对应生成的默认的客户端、服务端对象在ab.pb.go中，其中orderer所操作的服务端又单独实现在orderer/server.go中，peer点使用的客户端则基本沿用默认生成的客户端。server.go中的服务端对象实例server在main.go的main()中由server := NewServer(manager, signer)生成，使用ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server)进行了注册，随后grpcServer.Start()启动起来。server也只是一个空架子，不做实事儿，其两个服务Broadcast、Deliver直接对应交由两个成员handlerImpl（orderer/common/broadcast/broadcast.go）和deliverServer（orderer/common/deliver/deliver.go）的Handle处理。客户端的初始化，如peer chaincode和peer channel各自使用到的命令工厂ChaincodeCmdFactory/ChannelCmdFactory中使用到的BroadcastClient客户端，都会随着peer结点的初始化初始化。再如peer结点的gossip服务，所使用的Deliver客户端（core/deliverservice/deliverclient.go中的DefaultABCFactory）也是默认生成的客户端，会随着gossip模块的初始化而初始化。
* multiLedger总管 - 在orderer/multichain/manager.go中定义，是orderer直接使用的“总管家”，由其支配各个子模块。multiLedger包装chain集合，consenters集合，签名工具，账本生成工具，系统chain。在main.go的main()中，manager := initializeMultiChainManager(conf, signer)完成了multiLedger的初始化，初始化完成时，所有包含的子对象也相应的被初始化，所有orderer中现存的chain已经启动起来。这里的chain的概念比较模糊，既可以指账本本身，也可以指包含了账本的chainSupport，也可以指具体的处理消息的流程（如orderer/solo和orderer/kafka下各自实现的chain所执行的Enqueue）。而multiLedger自身也人如其名，multi+Ledger，成了多条链的“总管家”。
* chainSupport - 接口和实现均在orderer/multichain/chainsupport.go中定义，如同chaincode\_support对chaincode一样，属于尽一切努力对chain操作提供支持的对象。既包含账本本身，也包含了账本用到的各种工具对象，如分割工具cutter，过滤工具filter，签名工具signer，最新配置在chain中的位置信息（lastConfig的值代表当前链中最新配置所在的block的编号，lastConfigSeq的值则代表当前链中最新配置消息自身的编号）等。chainSupport也实现了一些用于支持各个工具的小接口。这样，一些流程切换，数据流向的转变的工作，都交给了chainSupport来做，chainSupport也做一些杂活。比如A工具操作中需要调用B工具，则A包含的往往不是B本身，而是chainSupport，借chainSupport来使用B。
* consenter - 分为solo和kafka两种类型，用于序列化生产（即各个peer点传送来的Envelope）出来的消息。solo在orderer/solo中实现，kafka在orderer/kafka中实现，两者也都是起到引导和配置的作用供chainSupport使用，真正干活的是solo/kafka封装进来的chain（orderer/solo/consensus.go）和chainImp（orderer/kafka/chain.go）。solo的chain是for循环 + select-case + chan组成的简单处理流程。kafka的chainImp则是一套使用第三方库github.com/Shopify/sarama实现的连接kafka服务端进行消息的订阅出版的方案，下文将详述。这里讲句题外话，solo和kafka都算不上真正的consenter，consenter意为同意者，与官方文档中的consensus mechanisms相对应，但是无论是solo和kafka都不具备容错性，因此也就谈不上是不是consenter了。这个模块之所以叫这个名字，估计是在等将来的SBFT。
* chain的启动 - 这里chain指的是处理消息的线程。在orderer/multichain/manager.go中，当NewManagerImpl初始化完毕一个chainSupport后，都会执行chain.start()启动背后的chain处理消息的流程。以kafka为例，chain.start()调用的是orderer/kafka/chain.go中chainImpl的Start()，进而go startThread(chain)启动了一个新线程，在startThread()中：（1）chain.producer, err = setupProducerForChannel(...)创建了kafka的生产者。（2）chain.parentConsumer, err = setupParentConsumerForChannel(...)创建了消费者。（3）chain.channelConsumer, err = setupChannelConsumerForChannel(...)分区消费者。（4）close(chain.startChan)，chain.errorChan = make(chan struct{})分别**开启**处理peer结点通过Broadcast，Deliver发来的请求的**开关**（只有开启orderer才会开始处理peer发来的请求，这点在后续文章中会详解）。（5）chain.processMessagesToBlocks()启动了for循环接收处理消息过程，该过程是用来**接收kafka服务端（在orderer中就是kafka容器）序列化后返回给kafka客户端（在orderer中就是sarama）的消息流，然后依次分类处理**。在processMessagesToBlocks()中：（1）正常的kafka生产出来的消息，会从chain.channelConsumer.Messages()这个通道中出来，根据不同的kafka消息类型，在switch-case中分别使用processConnect，processTimeToCut，KafkaMessage\_Regular函数进行处理。
* filter/committer - 过滤器/执行器，有几种，分散在orderer目录下，几种filter可以组装到一起形成一个过滤集合，orderer服务用此过滤集合（orderer/common/filter/filter.go中的RuleSet）中各个filter的Apply函数对收到的消息进行过滤，比如有些类型的消息的一些字段不能为空，大小必须在配置的范围之内等。和filter对应的是一个提交对象committer，即一个消息通过了filter的Apply，会返回一个committer，这个committer会在该消息写入账本之前执行Commit操作，做一些事情。同时，committer的Isolated的返回值标识着一条消息是否要单独成块，一些比较特殊的消息可能在不满足一定大小的情况下也要单独作为一个block，比如gensisblock。这么描述的话，一个个filter就有点儿模拟合同条款的味道了，一个filter集合，就是一个合同。而一个Envelope来到orderer中之后，会根据这个合同一条条的条款对照，如果符合了，提供执行一个对应的committer工具，以使Envelope能够拿到这个committer工具进一步做一些什么事情（这也是我们可以拓展的地方）。filter/committer接口在orderer/common/filter下定义，同时定义了三种基本的Apply过滤的结果：Accept表示接受，Reject表示拒绝，Forward表示当前filter不清楚接受还是拒绝而需要进一步验证。filter有如下类型：（1）emptyRejectRule，验证不能为空的filter，orderer/common/filter/filter.go中定义。（2）sigFilter，验证签名的filter，orderer/common/sigfilter/sigfilter.go中定义。这里会使用公共签名策略（common/policies中定义）来验证进入orderer的Envelope中所携带的签名是否满足要求。（3）sizeFilter，验证大小的filter，orderer/common/sigfilter/sigfilter.go中定义。在configtx.yaml中Orderer配置中有许多关于大小多少的配置项，将在这个filter中验证。（4）systemChainFilter，系统链filter（orderer中系统链的概念先不说），orderer/multichain/systemchain.go中定义，作用在于验证一个Envelope是否是一个用于升级orderer配置的ConfigUpdate类型消息，若是，则会提供一个包含configtx工具的committer，在该Envelope写入账本前，执行committer，新建一条新配置的链。（5）configtxfilter，orderer/common/configtxfilter/filter.go中定义，作用与systemChainFilter类似，检测Envelope是否是一个HeaderType\_CONFIG类型的消息，然后返回一个包含有configManager工具的committer供之后对消息做进一步操作。
* blockcutter - 块分割工具，用于分割block，具体为orderer/common/blockcutter/blockcutter.go中定义的receiver。类似于工厂流水线的自动打包机，一条一条消息数据在流水线上被传送到cutter处，cutter按照configtx.yaml中的配置（主要是大小限制），把一条条消息打包成一批（一箱）消息，同时返回整理好的这批消息对应的committer集合，至此cutter的任务完成。后续的，每一批消息被当作一个block，执行完对应的committer集合后，被写入账本。在configtx.yaml中关于Orderer的配置项中，BatchSize部分规定了block的大小：MaxMessageCount（10）指定了block中最多存储的消息数量，AbsoluteMaxBytes（10 MB）指定了block最大的字节数，PreferredMaxBytes（512 KB）指定了一条消息的最优的最大字节数（blockcutter处理消息的过程中会努力使每一批消息尽量保持在这个值上）。根据这三个值，cutter在工作时（具体指blockcutter.go中的Ordered函数）：（1）若一个Envelope的数据大小（Payload+签名）大于PreferredMaxBytes时，无论当前缓存如何，立即Cut；（2）若一个Envelope被要求单纯存储在一个block（即该消息对应的committer的Isolated()返回为true），要立即Cut；（3）若一个Envelope的大小加上blockcutter已有的消息的大小之和大于PreferredMaxBytes时，要立即Cut；（4）若blockcutter当前缓存的消息数量大于MaxMessageCount了，要立即Cut。（5）还有一个比较特殊的Cut，由configtx.yaml中BatchTimeout配置项（默认2s）控制，当时间超过此值，chain启动的处理消息的流程中主动触发的Cut（后续文章中会详解）。Cut所做的工作，就是将当前缓存的消息和committer返回供blockcutter与当前处理的Envelope打包成一批或两批消息，然后清空缓存信息。在上述需要Cut的情况中，只有（1）（2）会产生两批消息，且**先是**旧的消息（即blockcutter中之前缓存的消息）为一批，**后是**当前处理的最新的消息为一批。receiver中（如你设想的），filters是一个RuleSet，定义了过滤条件集合，均来自于orderer/multichain/chainsupport.go中的createStandardFilters或createSystemChainFilters（后者只是比前者多了一个systemChainFilter过滤对象，对看上一条（4）内容）；pendingBatch是一个Envelope数组，用来缓存Envelope消息；pendingBatchSizeBytes来记录这些缓存的消息的大小，pendingCommitters是一个Committer数组，用来缓存每个Envelope对应的committer。

### ledger

这里只讨论file形式的账本。orderer中使用的账本同peer结点中使用的一样，账本生成者BlockStoreProvider和账本自身BlockStore。只不过被稍微包装了一些，BlockStore被包装进了fileLedger（orderer/ledger/file/imp.go中），而fileLedger又是ReadWriter接口（orderer/ledger/ledger.go中）的实现。ReadWriter由Reader和Writer组成，也就是说，这可能是会拓展的部分，即将来在orderer中可能会实现只读账本，只写账本，读写账本的分类。fileLedger是同配置工具configResources一同封装到ledgerResources（orderer/multichain/manager.go）中，供chainSupport使用的。现阶段直接账本看作一个可写入可查询的数据库即可，详细的内部操作，将留在ledger主题文章中叙述。

与fileLedger相配合的是一个同文件中的fileLedgerIterator迭代器，由fileLedger的Iterator(startPosition)生成，传入一个链上（实际是账本中）查询block开始迭代的位置赋值给迭代器的成员blockNumber，比如startPosition为0（即SeekPosition\_Oldest），则说明从链的第一块block开始遍历迭代，再比如startPosition为SeekPosition\_Newest，则说明从链的现存最新的一块block开始遍历迭代，其余的值均算为SeekPosition\_Specified类型的位置，即任意指定链上的一个block块。任何查询fileLedger的行为，都是通过循环调用这个迭代器的Next()进行的，每调用一次，blockNumber自增1。fileLedgerIterator迭代器有一个特征是Next()会在迭代到还没有写入到账本block（即当前账本最新的block的接下来将要有的一个block）时阻塞等待，一直等待到该block被写入到账本中。这个阻塞控制涉及到两个chan，impl.go中的closedChan和fileLedger的成员signal。初始时closedChan为被close(closedChan)掉。下面详述这个阻塞控制：

* 当迭代器当前遍历到的block序列号**<=**账本上当前最新的block序列号时（即ledger.Height()-1），在Next()中，会进入if i.blockNumber < i.ledger.Height()分支（根据orderer对账本的操作，可知ledger.Height()返回的是链上将要添加的下一个block的序列号），查询后返回。而当迭代器遍历到ledger.Height()时，此时i.blockNumber == i.ledger.Height()，Next()会进入<-i.ledger.signal等待。当一个新的block要被写入账本，则会调用fileLedger的Append(block)，在Append(block)中，把block写入账本成功之后，会调用一下close(fl.signal)（此时Next()的等待会结束再次进入if i.blockNumber < i.ledger.Height()分支取block），然后紧接fl.signal = make(chan struct{})再给signal创建一个chan（此时若再调用Next()，仍会再次进入<-i.ledger.signal等待）。
* fileLedgerIterator迭代器的ReadyChan()根据当前迭代器的实际情况，即情况1：当i.blockNumber > i.ledger.Height()-1时，说明迭代器已经遍历到要查询账本的下一块还没有写入的block了，则返回signal；情况2：相反，则直接返回closedChan。这个控制会在Deliver服务处理客户端索要block时用到，即在orderer/common/deliver/deliver.go的Handle(...)中，正常的会进入if seekInfo.Behavior == ab.SeekInfo\_BLOCK\_UNTIL\_READY分支进行select-case选择，此时case <-erroredChan:除了关闭Deliver时会来消息，erroredChan不会再来消息，只会等待case <-cursor.ReadyChan():：若情况1，这里将等待signal，直到Append(block)中写入新的block后close(fl.signal)一下程序才会继续前进，在下文会调用到迭代器的cursor.Next()；若情况2，由于closedChan是关闭的，程序将直接继续前进。

### kafka

卡夫卡官方文档地址，<http://kafka.apache.org/documentation/> 。这里非常粗线条的概述一下orderer/kafka中涉及到的地方。

**1. kafka相关的概念**

* Producers/Consumers，生产者/消费者。最基本的概念，生产/消费消息的人（程序，结点）。在fabric中，生产者就是各个peer结点，如peer chaincode的一些操作生产出来一些消息发送到orderer，再由orderer送至kafka；消费者分为直接消费者和最终消费者，直接消费者就是orderer中的ledger，kafka出版的消息都被直接写入了orderer的ledger之中。最终消费者是各个peer结点，各个peer结点通过grpc再从orderer的ledger中取出写入的数据来同步自身的ledger。
* Topic，主题。生产者生产消息后，**按照主题发布消息**，kafka将处理的消息按内容主题进行分类，就犹如现实中出版的各种主题的报纸，如体育，娱乐，财经等，到了fabric中的orderer服务中，topic是按照chainID来分的，即一个chain对应一个主题，一条chian的orderer所生产的消息都会发布到同一个对应主题中。
* Partition，分区。每个Topic消息可以分为多个分区进行存储。kafka是基于文件存储的，因此可以直接把一个分区视作一个独立的文件（消息多，自然要分开存在多个文件中），接收到的消息直接会被顺序地追加到文件尾部。到了fabric中的orderer服务中，一个主题（即一个chain）只对应一个分区。
* offset，偏移。消息存储在每个分区中，都会有一个offset（类似C语言的文件指针的偏移）作为定位这条消息的唯一ID。
* replicas，备份。kafka可以设定每个分区的备份数量。
* Broker，代理（掮客）。是一个相对的概念，即当只有一个kafka的Server时，并没有代理这一说，但当有多个kafka服务形成集群（cluster）时，每个具体的kafka身份就变成了一个Broker。
* zookeeper，“动物园管理员”。kafka自身集中火力实现高吞吐量的核心目标，而zookeeper则辅助kafka来管理各个消费者和在各个broker之间共享信息，比如，保存每个消费者对分区消息消费的偏移量信息。
* Leader/Follower，领导者/跟随着。也是一个相对的概念，领导者主抓分区的读取和写入，Follower只跟随者就是了。如有两个broker，1和2，有5个分区A，B，C，D，E。broker1负责A，B，C分区的读取和写入，则对于A，B，C分区来说，broker1是领导者，broker2是追随者。broker2负责D，E分区的读取和写入，则对于D，E分区来说，broker2是领导者，broker1是追随者。这一点是不是挺熟悉的？gossip服务中的leader模块也是这么做的。

**2. sarama**

Sarama is an MIT-licensed Go client library for Apache Kafka，这是sarama库的README的第一句话，即是说，sarama是一个kafka的go客户端库。sarama使用的基本流程如下（所产生的对象均是客户端对象）：

//1.创建一个kafka的综合配置对象，该对象可以配置生产者，消费者，网络等等等等。

brokerConfig := sarama.NewConfig()

//2.对配置对象逐一赋值，细节不叙，这一点是比较杂的一点。

brokerConfig.Producer.XXX = ...

brokerConfig.Consumer.XXX = ...

brokerConfig.Net.XXX = ...

brokerConfig.Metadata.XXX = ...

...

//3.根据配置创建生产者对象，其中brokers为字符串数组类型的broker的地址列表，

//在生成具体的生成者对象时，该对象已经根据brokers连接了kafka服务端。

//sarama的生产者对象有SyncProducer，AsyncProducer两种，即同步，异步两种。

//同步表现在出版一条kafka消息后会一直阻塞到收到出版成功的回复，异步则不存在阻塞。

//同步Producer

producer := sarama.NewSyncProducer(brokers, brokerConfig)

//异步Producer

producer := sarama.NewAsyncProducer(brokers, brokerConfig)

//4.根据配置创建消费者和分区消费者，参数与生产者一致。

//消费者是负责生成和管理（多个）分区消费者的对象，分区消费者是真正订阅消费消息的对象。

consumer := sarama.NewConsumer(brokers, brokerConfig)

//参数依次是哪个主题，哪个分区，从哪个offset开始（消费）

partitionConsumer := consumer.ConsumePartition(topic,partition,startFrom)

//5.生产者出版kafka消息，分区消费者订阅消费kafka消息

producer.SendMessage(&sarama.ProducerMessage{...})

msg\_chan := partitionConsumer.Messages()

//从分区消费者chan中获取消息并使用

msg := <-msg\_chan

...

//6.生产者，消费者关闭，为防止leak，必须进行此步操作。

producer.Close()

consumer.Close()

partitionConsumer.Close()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38

**3. fabric中的kafka**

* fabric中使用kafka的基本目的是在多个peer结点同时向orderer服务发送消息时，能通过kafka形成一个串行的消息队列（队列中消息序号唯一，至此kafka的作用就算完成了），然后供cutter进行封装成一批批消息（即block），再将block写入orderer的ledger，也就形成了**块链**，最后供区域内各个peer结点中的leader获取orderer的ledger中的数据，然后由leader将数据在的区域中的各个peer结点间共享，形成了**区块链**。
* fabric中关于sarama用到的关于kafka的配置集中在orderer.yaml中的Kafka配置区域，configtx.yaml中的Orderer部分中的Kafka部分。
* sarama中主要的对象，生产者/消费者都封装在orderer/kafka/chain.go的chainImpl对象中，而综合配置则封装在管理chainImpl的在orderer/kafka/consenter.go的consenterImpl对象中。
* fabric中的kafka，具体来说是orderer中的kafka，是一个客户端，由sarama实现，通过网络连接的kafka服务端容器。sarama被包装进了orderer/kafka/chain.go中的chainImpl中使用。chainImpl成员中，producer即为生产者，parentConsumer为消费者（负责生成和管理分区消费者的），channelConsumer为分区消费者（实际消费消息的）。
* 官方文档的Bringing up a Kafka-based Ordering Service章节，介绍了如何启动一个以kafka服务为基础的orderer服务。还有就是，关于使用kafka，按官方文档的说法，将来肯定会被PBFT替换掉。在之后的orderer服务的讲述中，会把kafka服务端本身当作一个“暗盒”，即从消息进入到kafka服务端到从这里面出来的过程不会有所叙述，而还是把笔墨着于orderer服务自身。

# fabric源码解析23——Orderer服务

本篇文章分为下列四个部分：

1. peer结点与orderer结点间的grpc服务是如何建立的。
2. 消息从peer结点发给orderer服务的过程，核心是Broadcast服务。
3. 消息在orderer结点中被处理的过程。
4. peer结点从orderer结点索要服务的过程，核心是Deliver服务。

这里给的前提是，orderer服务已按照orderer.yaml/configtx.yaml的配置正常启动。讲解orderer服务可以按照服务分类来讲，也可以依据数据的流向按步骤讲。本文综合一下，既分服务，也按数据流向的步骤。在每个服务中按数据流向讲解，服务与服务之间的步骤前后衔接。

## 建立连接

grpc自然分为客户端和服务端，对于peer连接orderer来说，peer是客户端，orderer为服务端。

Broadcast服务主要集中在peer结点的命令中，如peer chaincode，peer channel命令。在peer/common/common.go中，GetBroadcastClientFnc的值为peer/common/ordererclient.go中的GetBroadcastClient函数，peer chaincode和peer channel命令中使用到的**命令工厂**中的broadcast客户端BroadcastClient（在peer/common/common.go中的ChaincodeCmdFactory中）均由此函数生成。同时，peer chaincode和peer channel命令执行时，都有一个Flag，-o，来指定所要连接的orderer服务结点的地址。当命令（第一次）执行时，peer结点会建立与orderer结点的Broadcast服务的grpc连接。

Deliver服务较复杂和绕一点，对象之间相互嵌套，主要集中在peer结点的gossip服务中，且包裹在core/deliverservice/blocksprovider/blocksprovider.go中的**blocksProviderImpl**中的**client**中的**createClient**，client的类型是core/deliverservice/client.go中的**broadcastClient**，createClient也不是Deliver服务本身，而是用于生成Deliver服务实例的，而core/deliverservice/deliverclient.go中的**deliverServiceImpl**按chainID存储不同的blocksProviderImpl（也就是说每个chain都会有一个自己的client），是gossip服务直接使用的deliver服务的对象。

1. 在peer/node/start.go的serve函数中，service.InitGossipService(...)初始化了gossip服务，将一个专门生成deliver服务对象的deliver工厂赋值给gossip服务的的成员deliveryFactory，这个deliver工厂的值是gossip/service/gossip\_service.go中的**deliveryFactoryImpl**。serve中随后的peer.Initialize(...)（core/peer/peer.go）调用了createChain，在createChain中调用了service.GetGossipService().InitializeChannel(...)（gossip/service/gossip\_service.go），传给InitializeChannel的最后一个参数ordererAddresses就是orderer的地址，对gossip服务的进一步进行了初始化。
2. 在InitializeChannel中，使用了第1点中deliveryFactoryImpl的Service函数，进而使用deliverclient.NewDeliverService(config)（core/deliverservice/deliverclient.go中实现）生成了一个Deliver服务实例**deliverServiceImpl**赋值给gossip服务中的成员**deliveryService**，所给的配置**config**中**Endpoints**，**ConnFactory**，**ABCFactory**被分别赋值为传进来的orderer的地址，core/deliverservice.go中的DefaultConnectionFactory，DefaultABCFactory。
3. 重回InitializeChannel中，无论是静态指定leader还是动态选举leader，最终都会调用deliveryService.StartDeliverForChannel(...)，即在core/deliverservice/deliverclient.go中，StartDeliverForChannel会使用newClient->NewBroadcastClient（core/deliverservice/client.go），创建一个client，再使用blocksprovider.NewBlocksProvider(...)创建一个包含client的blocksProviderImpl，随后go d.blockProviders[chainID].DeliverBlocks()**启动client的接收线程**。
4. 在NewBroadcastClient中，createClient就被赋值为上述config的ABCFactory，而生成的broadcastClient就是blocksProviderImpl中的成员client。也就是说，blocksProviderImpl中的createClient的值是core/deliverservice/deliverclient.go中的DefaultABCFactory，DefaultABCFactory即是使用了protos/orderer/ab.pb.go中生成的默认的AtomicBroadcastClient（自然包含默认的Deliver客户端）。**启动client的接收线程后**，在client接收消息时，即core/deliverservice/client.go中的Recv()，会执行try(...)，进而执行bc.doAction(action)，在doAction中会查看client是否已经和orderer通过grpc连接上，若未连接则会执行connect()进行grpc连接。
5. **总结一下：**当peer结点的容器启动起来的时候，peer结点的gossip服务随着peer node start命令启动，其中角色是leader的peer结点的gossip服务中使用到的deliver服务对象deliverServiceImpl也会使用默认生成的Deliver客户端通过grpc连接orderer结点并启动循环接收的线程。至于orderer如何不断的给peer结点中的leader发送block消息，将在下文Deliver服务中详述。

## Broadcast

可回看《fabric源码解析20》，以peer chaincode instantiate为例，假设名为peerA的结点将example02部署完毕后，生成了一个Envelope（参看图片EnvelopeSendToOrderer.png）发送给orderer（参看peer/chaincode/instantiate.go的chaincodeDeploy中的cf.BroadcastClient.Send(env)）。cf即为peer的命令工厂ChaincodeCmdFactory的实例，cf成员BroadcastClient即为使用/protos/orderer/ab.pb.go中默认生成的Broadcast服务客户端AtomicBroadcast\_BroadcastClient。

1. cf.BroadcastClient.Send(env)，通过grpc将消息以Envelope的形式发送到orderer结点中。

## Orderer

1. orderer/server.go中的Broadcast收到peer点发来的Envelope消息，直接交由成员bh的Handle处理。
2. bh原型为orderer/common/broadcast/broadcast.go中的handlerImpl，Handler函数也在这里实现，在Handler函数中，会在for中循环接收来自peer结点的消息：**（1）**msg, err := srv.Recv()接收消息。**（2）** Unmarshal并检查Envelope消息中的一些字段（如ChannelId），如果是HeaderType\_CONFIG\_UPDATE类型的消息，则会将消息经过bh.sm.Process(msg)，实际是调用orderer/configupdate/configupdate.go中的Process对消息进行进一步的加工，将消息加工成一个orderer可以处理的交易消息（加工成 创建新chain的配置 或 对已存在的chain更新配置）。**（3）**support, ok := bh.sm.GetChain(chdr.ChannelId)，获取消息对应ChannelId的链的chainSupport，然后support.Filters().Apply(msg)，使用该chainSupport的过滤器过滤该消息，以查看该消息是否达标，这里没有使用过滤器一并返回的committer集合，这是第一次过滤。**（4）**support.Enqueue(msg)，进而调用了support对应的chain的cs.chain.Enqueue(env)处理消息，这个chain是对应的consenter（这里只以kafka为例进行介绍）的HandleChain生成的（在创建chainSupport的时候，orderer/multichain/chainsupport.go的newChainSupport中，cs.chain, err = consenter.HandleChain(cs, metadata)）。
3. 消息进入orderer/kafka/chain.go的Enqueue(env)，在最外层的select-case中，如果startChan已经被close（对看《fabric源码解析22》chain的启动章节关于开关的文字），则说明orderer的kafka客户端部分（即orderer/kafka/chain.go的chainImpl）已经一切准备就绪，则会进入case <-chain.startChan:分支，否则进入default:分支，什么都不做（即不处理消息）而返回。在case <-chain.startChan:分支中，又是一个select-case，如果该chain没有停止，即haltChan没有被close，则会进入default:分支，消息也会在此进行处理。
4. 在这个default:分支中：**（1）**message := newProducerMessage(chain.channel, payload)把消息打包成kafka消息类型，也是使用的sarama预定义的ProducerMessage类型，Topic定义了消息的主题，Key和Value，key为分区号，value为Marshal过的peer点发来的原始的消息。**（2）**chain.producer.SendMessage(message)，使用sarama的生产者对象将kafka消息发送到kafka服务端。
5. kafka服务端（即kafka容器）这个“暗盒”接收到消息并**生产消息**。
6. 在之前chain启动起来的接收kafka服务端消息的线程中（对看《fabric源码解析22》chain的启动章节），即orderer/kafka/chain.go中的processMessagesToBlocks()，kafka服务端生产的消息从case in, ok := <-chain.channelConsumer.Messages()处**被消费出来**，得到ConsumerMessage类型的**消费消息**，并进入该分支。
7. 在case in, ok := <-chain.channelConsumer.Messages()分支中，首先出现的一个select-case是一个关于errorChan通道的开关（也对看《fabric源码解析22》chain的启动章节关于开关的文字），errorChan通道在chain初始化之初是关闭的（参看newChain(...)中的close(errorChan)），而后在startThread()中的chain.errorChan = make(chan struct{})中再度生成，算是再度开启。如果errorChan是关闭的，则在select-case中会进入case <-chain.errorChan:分支而返回，否则进入default:分支，程序继续。随后进入一个switch-case，根据解压消费消息所携带的数据的类型，进入不同分支，处理消息。正常的，数据会进入case \*ab.KafkaMessage\_Regular:分支，被processRegular(...)处理。这里注意两个数据：**（1）**一是传入processRegular(...)的消费消息的in.Offset，该值是消息在kafka服务端分区中的offset，在整个生产消费过程中是**序列化排序**依次分配给每个消息的，因此每个消息具有唯一的offset，也因此in.Offset可以代表每个具体的消费消息。在block的结构中，一方面，Metadata是一个数组，用来存储与block块有关的基础信息，其中下标BlockMetadataIndex\_ORDERER处存储的就是block中最后一条消息的**offset值+1**（也相当于下一个block中第一条消息的offset）。另一方面，block中实际存储数据的Data是一个[]byte数组，序列化过后的消息都被二进制化后依次存放在这个数组中，我们可以很容易的算出block中具体有多少条消息。因此通过**< offset+1**的条件，就可以准确定位block中的每条消息。**（2）**二是传入processRegular(...)的time计时器，这个计时器对应控制《fabric源码解析22》blockcutter章节描述的关于Cut操作的第（5）点情况。
8. 在processRegular()中，将原始数据（即peer结点发来的消息）从kafka类型消息中分解出来后，交由blockcutter模块的Ordered进行分块处理。
9. 对看《fabric源码解析22》blockcutter章节，消息进入orderer/common/blockcutter/blockcutter.go的Ordered(env)，先r.filters.Apply(msg)进行了第二次的过滤（对看第2步（3）的第一次过滤，这主要是防止从第2步到此步期间orderer的配置有所更新），然后就是一个根据配置信息进行分割成block的过程。这里假设此时满足生成一批消息的条件，该批消息和对应的committer集合一同被返回。
10. 重回processRegular()，当返回了一或二批消息，程序会进入for i, batch := range batches循环依次处理每批消息。在循环中：**（1）**计算当批消息的offset值（即最后一条消息的offset值+1），offset := receivedOffset - int64(len(batches)-i-1)，对看上文第7点对传入的消息的offset的描述和《fabric源码解析22》blockcutter章节所描述的批量消息是如何返回的，就可以明白为何这么计算。**（2）**block := support.CreateNextBlock(batch)，将当批消息打包成block，至此**消息正式成块**。**（3）**support.WriteBlock(block, committers[i], encodedLastOffsetPersisted)，将block对应的committer集合，然后将block写入账本。
11. 这里单独描述一下传入processRegular()中的time计时器和该计时器控制的主动Cut操作。调用processRegular(...)是chainImpl启动的processMessagesToBlocks，而消息来源于kafka服务器，倘若kafka服务器长时间没有消息被消费出来，要么是kafka服务器出了故障，要么是客户端没有新的生产消息发给kafka服务器。在这种情况下，blockcutter可能已经缓存了一些之前的消息，为了不使这部分消息丢失和及时记录到账本中（比如kafka处理的数据量很小或消息流很不稳定，一条消息后很长时间都不来下一条消息，会造成已缓存的数据不能及时记录到账本，有丢失的风险，也有交易已经产生了好长时间，但是包裹交易的消息一直悬空在blockcutter缓存中无法被消费，进而无法被记录和查询），configtx.yaml中配置的BatchTimeout规定了超时时间，默认2s。当下一条消息超过2s还没被消费出来，将会主动Cut操作将现有缓存打包成一个block写入账本。具体的操作是：**（1）** timer初始状态为nil，当消息进入processRegular()中，若缓存进blockcutter中，则会进入if ok && len(batches) == 0 && \*timer == nil分支，设timer计时器为2s后触发然后返回。**（2）**下一条消息若在2s之前再次进入processRegular()，若仍是被放入缓存，则依然会进入if ok && len(batches) == 0 && \*timer == nil重置timer为2s计时然后返回。**（3）**当timer没有及时被重置，即超过了2s，processMessagesToBlocks会进入case <-timer:分支，调用sendTimeToCut向kafka服务器发送一条TimeToCutMessage消息（包裹着chain.lastCutBlockNumber+1，即若主动Cut，会使用的block的序号），接着kafka服务器收到，然后被消费出来（这里存在一个**时间过程**），这条TimeToCutMessage消息会再次进入processMessagesToBlocks的case in, ok := <-chain.channelConsumer.Messages():分支，进而进入case \*ab.KafkaMessage\_TimeToCut:分支，调用processTimeToCut(...)处理这条消息，也就是主动Cut。**（4）**在processTimeToCut(...)中，倘若此时if ttcNumber == \*lastCutBlockNumber+1（ttcNumber即为第（3）点TimeToCutMessage消息包裹的block序列号），说明在第（3）点提到的时间过程中，chain.lastCutBlockNumber的值没变，也就是没有发生新的Cut，因为一旦有新的Cut，chain.lastCutBlockNumber就会增1。没有发生新的Cut，则说明blockcutter中现在缓存的数据依然是发送TimeToCutMessage消息时的数据，或者有新数据但是依旧没有达到Cut的条件。此时就会进行主动的Cut，blockcutter中现有的缓存被打包出来后清空，timer计时器也会重新的置为nil。另外，在processRegular(...)中，当常规的触发了Cut后blockcutter中将不存在缓存消息，之后会进入if len(batches) > 0分支，会将timer计时器置为nil，在processMessagesToBlocks中的for-select-case等待再次从kafka中消费出消息。
12. 在block := support.CreateNextBlock(batch)（参看图片EnvelopeToBlockToWriteToLedger.png）和support.WriteBlock(block, committers[i], encodedLastOffsetPersisted)，调用了support中的ledger进行创建block和写入block，在写入block之前会逐一执行该block对应的committer集合，这里不再详述。至此，orderer端处理peer结点发送来的消息的流程结束。

## Deliver

Deliver服务较Broadcast服务复杂，我们知道最终orderer中ledger账本的数据会通过Deliver服务推送leader结点，现在还存在一些的问题：是orderer主动向peer结点推送数据？还是peer结点主动向orderer索要数据，要多少给多少？如果是peer结点向orderer索要数据，是在何处且索要方式是什么，定时索要固定量，还是orderer端一有block产生就索要？

1. 既然Deliver服务是建立grpc之上，且orderer为Deliver的服务端，则基本上可以推断是peer结点向orderer索要block数据。gossip服务的索要行为的对象是core/deliverservice/requester.go中的blocksRequester，索要行为发生在RequestBlocks(...)函数中，该函数传入一个包裹了高度值的LedgerInfo，若该高度值为向orderer索要的开始的block的序列号，如果高度值>0，则调用seekLatestFromCommitter，否则调用seekOldest()，两个函数过程基本一致，先创建一个包装了SeekInfo信息（即索要的block的起止范围信息）的HeaderType\_CONFIG\_UPDATE类型的Envelope消息，然后b.client.Send(env)向orderer端的Deliver服务端索要block。注意，这两个函数索要的block范围的起点可能不一样，但是止点都一致：**math.MaxUint64**，即最大极限值，这相当于向orderer端索要现在以及将来所产生的所有block。
2. 索要全部的block的行为在何时发生呢。对看上文建立连接章节，**（1）**在core/deliverservice/deliverclient.go的newClient中，broadcastSetup := func(...中调用了RequestBlocks(...)。**（2）**broadcastSetup作为一个参数通过NewBroadcastClient赋值给了bClient（原型是core/deliverservice/client.go中的broadcastClient）的成员onConnect。**（3）**随后bClient通过newClient返回，在StartDeliverForChannel(...)中的blocksprovider.NewBlocksProvider(...)，bClient赋值给了对应chainID的**BlocksProvider**（core/deliverservice/blocksprovider/blocksprovider.go）的成员client。**（4）**接着go d.blockProviders[chainID].DeliverBlocks()，调用了b.client.Recv()，这个client，就是第（3）点提到的bClient。**（5）**Recv()在core/deliverservice/client.go中，调用了bc.try(...)，传给try(...)参数是一个执行接收动作的函数，即使用bc.BlocksDeliverer.Recv()接收消息（此时BlocksDeliverer还没有被赋值）。**（6）**在try(...)中，在for循环中不断尝试执行bc.doAction(action)，这个action是上一步传入的接收消息的动作。在doAction(action)中，会首先查看连接conn是否已经建立，此时没有建立，则会调用bc.connect()建立连接。随后resp, err := action()执行action动作。**（7）**对看建立连接章节Deliver部分，追溯一下可知，在connect()中，使用了conn, endpoint, err := bc.prod.NewConnection()建立了一个连接orderer结点的grpc连接，abc, err := bc.createClient(conn).Deliver(ctx)使用建立的grpc连接创建了一个Deliver服务客户端，随后调用了bc.afterConnect(conn, abc, cf)。**（8）**在afterConnect中，将创建与orderer结点的连接和Deliver客户端分别赋值给broadcastClient的conn（可使之后的所有doAction(...)中的connect()不再执行）和BlocksDeliverer（对看在（5）时BlocksDeliverer还没有被赋值），然后调用bc.onConnect(bc)，这个onConnect就是（1）（2）中的broadcastSetup，即在这里发生了索要行为，向orderer结点索要所有的block。**（9）**随后开始返回，重新返回到doAction中，与orderer结点的连接建立并发送了索要请求后，继续开始执行动作resp, err := action()，这个action即为开始接收orderer结点发来的block数据（对看（5））。**（10）**总结一下，以上步骤都是随者gossip服务运行起来，调用deliverServiceImpl对象的StartDeliverForChannel，所启动的go d.blockProviders[chainID].DeliverBlocks()中的for循环中持续调用的，其中建立连接和索要行为只会发生一次。
3. 第2点（8）中执行bc.onConnect(bc)后，即执行的是core/deliverservice/requester.go中的RequestBlocks后，orderer结点在orderer/common/deliver/deliver.go的Handle中的envelope, err := srv.Recv()处被接收，之后：**（1）**一系列从Envelope的解压收取数据的操作，然后chain, ok := ds.sm.GetChain(chdr.ChannelId)获取Envelope中对应chainID的chainSupport。**（2）**erroredChan := chain.Errored()，select {case <-erroredChan:...，这也是一个erroredChan控制的开关，对看上文Orderer章节第7点，用法是一样的。此刻erroredChan开关处于开启状态，程序将继续向下走。**（3）**sigfilter.New(...)，sf.Apply(envelope)，创建了一个签名过滤对象验证Envelope消息，不详述。**（4）**从Envelope解压出来携带的索要的block的起止信息SeekInfo，cursor, number := chain.Reader().Iterator(seekInfo.Start)，根据起止信息创建一个账本的查询迭代器cursor，这个迭代器是在orderer/ledger/file/impl.go中定义的fileLedgerIterator，存储了起点，有一个特性是Next()会在迭代到还没有写入到账本block（即当前账本最新的block的接下来将要有的一个block）时阻塞等待，一直等待到该block被写入到账本中。**（5）**在for循环中，block, status := cursor.Next()，不断使用迭代器cursor获取一个block，随后sendBlockReply(srv, block)向Deliver客户端回复该block。**（6）**向Deliver客户端回复一条block后，会进行if stopNum == block.Header.Number判断，若刚刚发送的block已经是客户端索要的最后一块block，则break跳出循环等待客户端下次的索要行为。但是前面已经提到，gossip服务索要的block的止点是**math.MaxUint64**，即stopNum的值是math.MaxUint64，所以向gossip服务发送block的for循环“永远”不会退出。
4. orderer的Deliver服务端向peer结点的Deliver客户端发送block，block进入第2点（9）的doAction中，resp, err := action()接收到block，继续返回至try(...)，再返回至Recv()，再返回core/deliverservice/blocksprovider/blocksprovider.go的DeliverBlocks()中的msg, err := b.client.Recv()，然后进入case \*orderer.DeliverResponse\_Block:分支：**（1）**gossipMsg := createGossipMsg(b.chainID, payload)将block包装成gossip服务能处理的消息类型。**（2）**b.gossip.AddPayload(b.chainID, payload)，将block添加到peer结点的gossip服务模块的本地，目的在于添加到本地的ledger中。再次强调，接收block的是peer结点中的leader，所以这里的gossip服务模块是leader的。**（3）**b.gossip.Gossip(gossipMsg)，leader的gossip服务模块向其他peer结点散播block。
5. 向Orderer的Deliver服务端索要block消息的，除了gossip服务，还有peer channel的几个子命令，但peer channel索要的都是一定量的block，即非持续性的，在此不做详述，流程与上述一致，只是Deliver服务端最后会进入if stopNum == block.Header.Number分支跳出回复block的for循环（对看第3点（5）的描述）。**总结一下**：peer结点中的leader的gossip服务起来之后就建立了与orderer的Deliver服务的连接（这期间peer结点只会在开始的时候向orderer的Deliver服务索要一次block），之后当orderer中的账本中存在block数据后，就开始主动的向leader的Deliver客户端发送block数据，这个推送行为将一直持续。leader的Deliver客户端收到block流之后会使用gossip服务向自身和其他peer结点散播这些block数据。另外，peer channel等命令也会向Deliver服务端请求某一段block数据，但该推送是非持续性的。

## 拓展：orderer实现的共识机制

这一部分是在群中交流之后追加的。分享一篇文章 A Kafka-based Ordering Service for Fabric，原文在docs.google上，鉴于地址可能会失效且FQ问题，在此不贴链接了，可自行搜索或加入群中，群中该文件有共享，该文章也是官方文档Bringing up a Kafka-based Ordering Service章节中的链接。文章讲述的内容大概为如何考虑问题并一步步设计解决问题实现以kafka为基础的orderer服务。这是个**自上而下**的过程。源码解析是一个**自下而上**的过程，从代码的实现去推敲作者的设计意图和所要解决的问题。理解了一个方向后再去从另一个方向去理解，感觉还是很受益的。在此感谢悠悠浮云的共享。同时，建议读一下官方文档的Bringing up a Kafka-based Ordering Service章节，指导了如何部署多个orderer结点的步骤（若觉与本文有矛盾之处，乃笔者未理解透彻之过，已官方文档为准）。

从文章中受益而来的，是关于orderer服务的源码解析中没有讲述到多个orderer结点的情形。即区域链中，服务于每个“区域”的一个链中peer结点的多个orderer结点共同连接一个kafka服务器。每个orderer结点的服务过程如文章之前描述的那样不变，多个orderer连接kafka服务器也没有其他更多的不同之处，只是接收的“生产点”和“消费点”多了而已。但是考虑多个orderer结点的情况，可以提高了各位审视区域链orderer服务的高度，也出现了一个此前忽略的重要的概念（可以说是精髓）：共识机制。这里的共识问题就是：kafka输出的消息被消费至多个orderer结点后产生的**block块序列最终一致**。

若同一个chain存在多个orderer结点服务于众peer结点，即“生产点”和“消费点”多了的情况，则在共识问题上主要需要解决下列两个问题：

* A. 每个orderer结点不能保证在同一时刻启动开始服务（或者有新的orderer加入），因此，先启动的orderer结点可能会先产生block块，但后启动的orderer与前启动的orderer间相差的block怎么弥补？
* B. 当更新orderer服务的配置，每个orderer结点不能保证同时更新配置，因此在某一时刻每个orderer的配置可能不同，又因为配置会影响到Cut操作，因此如何保证一个chain中的某个orderer配置更新后，其他的orderer结点得到更新但不影响共识？

这里有几个前提：

* fabric中以kafka为基础的orderer实现，从实现上看是每一条链一个分区（因为每个orderer/kafka/chain.go中的chainImpl都只有一个sarama.PartitionConsumer成员）。在同一条链上多个orderer结点共同连接一个kafka服务器，消费kafka服务器上的同一个分区（可直接当成同一个文件）中的消息。
* kafka同一分区生产存储的消息序列无论在哪一个orderer结点，被消费出来的序列都是一样的。这也是kafka服务自身实现的特性。也就是说，多个orderer结点从kafka服务器出得到的消息序列，无论各个orderer结点在消费消息的速率的快慢，最终得到的消息序列是一致的。再也就是说，无论各个orderer结点在生产消息的速率上的差异如何，只要将不同的消息成功写入kafka分区，各个orderer结点消费出来的消息序列依旧一致。
* 要达到每个orderer结点所产生的block序列一致，关键在于对Cut操作的控制。orderer服务中的Cut操作受configtx.yaml中的BatchTimeout/BatchSize两项配置的影响。**（1）**由BatchTimeout触发的Cut将以TimeToCutMessage消息的形式发送到kafka服务器的分区中，当一个结点接收到TimeToCutMessage消息时，无论该消息是否是自身发出的，只要TimeToCutMessage携带的BlockNumber值等于自身账本的下一块block的序列号（对看上文Orderer章节第11点（4）），则Cut。这相当于利用分区消息序列的唯一性设置了一个统一Cut的命令，任何一个orderer结点无论先后，只要读到这个消息就进行Cut，这样被Cut出来的block是一致的，只是在时间早晚罢了。**（2）**由BatchSize触发的Cut操作，只要每个orderer结点的该配置一致，则顺序消费同一个分区的消息，所产生的block块序列自然一致。

### 问题A

1. 在orderer/kafka/chain.go中，newChain(...)生成的chainImpl实例时，成员lastOffsetPersisted所赋的值为-3。startThread(...)中在生成分区消费对象，也是chainImpl的成员channelConsumer时，chain.channelConsumer, err = setupChannelConsumerForChannel(..., chain.lastOffsetPersisted+1)，最后一个参数指定的是channelConsumer从分区的何处开始消费（参看sarama库对ConsumePartition函数的说明），此时chain.lastOffsetPersisted+1值为-2。
2. sarama库中，提及了两个常量：OffsetNewest，OffsetOldest，前者指从分区的最新偏移处开始消费，后者指从分区最开始偏移处开始消费。而OffsetOldest的常量值就是-2，即第1点所提及的chainImpl的channelConsumer默认是从分区的最开始出开始消费的。
3. 因此，同一个chain有多个orderer结点时，每个orderer结点中都有自己的一个chainImpl对象，但一个chain对应一个分区，所有每个orderer的chainImpl中的channelConsumer都是消费同一个分区下的消息。无论每个orderer结点的chainImpl何时开始消费消息，其都是从分区的最开始处开始消费消息的，因此不会错过任何一个之前（在该orderer结点开始消费消息之前）其他orderer结点产生的消息。
4. 另外需要注意的是，kafka服务器自身对分区消息保留的时长或大小是可以设置的，分别由.server.properties文件中的log.retention.hours和log.retention.bytes指定，超过设置的时间或大小，分区中之前旧的消息会被清除。Getting Started中下载的kafka镜像所启动的kafka服务器的配置中，log.retention.hours值为默认的168（一周），log.retention.bytes值为-1，即不开启此阈值（可以docker run一下kafka镜像，无论是否启动，都会将配置打印出来）。因此，orderer结点最晚加入的时间也将受此限制。

### 问题B

1. 同理，如同在分区中插入TimeToCutMessage消息作为统一的Cut命令一样，一个orderer通过向分区中插入ConfigUpdate消息（这里ConfigUpdate消息算作使一条正常的交易消息），对各个orderer结点进行统一的配置升级。需要明确的是，这个ConfigUpdate消息其实是链中具有写权限的peer结点通过Broadcast服务发给orderer结点的。
2. ConfigUpdate消息途径orderer/common/broadcast/broadcast.go的Handle(...)时，会进入if chdr.Type == int32(cb.HeaderType\_CONFIG\_UPDATE)分支进行bh.sm.Process(msg)加工处理，使这个消息在发送到分区后再被各个orderer或前或后消费出来后到达orderer/kafka/chain.go的processRegular(...)中，ConfigUpdate消息进入orderer/common/blockcutter/blockcutter.go的Ordered(...)中执行了committer, err := r.filters.Apply(msg)而产生了消息对应的committer。这个committer的Commit()就是最终用来执行配置升级的。ConfigUpdate和对应的committer最终会被包裹在一批消息中返回出来，假设该批消息为batchA，对应的committer集合为committerA。在processRegular(...)中，block := support.CreateNextBlock(batch)将batchA打包成block，随后在执行support.WriteBlock(...)中被执行依次执行了committerA，其中自然包括ConfigUpdate对应的committer。也就是说，bathA生成的block是以旧的配置信息生成的最后一块block，之后再进行Cut，均是新的配置下所生成的block。对于任何一个orderer结点，均是这样。

### 思考

《fabric源码解析》中提及到，以kafka为基础的orderer服务的共识机制不具备容错性，这点也是官方文档中所述的。在此的体会是，这个容错性可能指两方面的容错性：（1）orderer服务自身的容错性（而不是kafka服务的容错性）。kafka服务自身具有一点的容错性，在集群中，可以容忍一定数量的代理失效。但是多个orderer自身不存在容错性，一个orderer结点发生故障，假设是名为orderer\_ERROR的结点，则orderer\_ERROR结点所服务的所有peer结点将无法再继续同orderer\_ERROR通信，也就自然无法再向orderer\_ERROR发送交易数据或获取block数据。（2）多个orderer结点消费顺序同一分区的消息，正常工作自然好，但倘或某一个orderer结点在消费的过程中由于某些原因丢失了已经消费出来的一个消息或者根本就没有消费到某一个消息呢？如此之后该orderer所产生的block将同其他orderer结点产生差异。而据此推测，PBFT ordering service将实现应该是：多个orderer的情况下，允许一定数量的orderer结点失效，而这些失效的orderer结点背后服务的所有peer结点的通信将被及时转移到依然有效的orderer结点上来，即这些peer结点及时重新连接仍然有效的orderer结点，从而继续工作。而且可能会增加检查各个orderer间所产生的block序列的行为。还有一点，就是orderer结点是否可以自由加入的问题，目前从代码，kafka服务的配置显示，orderer的结点是不能像peer结点那样自由加入的，但这样又有点悖于对于区域链这种分布式的松散和自由，比如你设计启动一个区域链，需要预先设计好orderer结点的数量。

# fabric源码解析24——ledger之idStore和BlockStore

## 概述

《fabric源码解析5》中对fabric中涉及的账本进行了初次描述，本文则着眼于更系统的描述账本在fabric中的使用。使用账本的有两处：peer结点和orderer结点。各自有自己的实现，但最最基础的底层操作还是对数据库的操作。

从数据库类型角度讲，分为**goleveldb**和**couchDB**两种，在core.yaml配置文件中ledger区域中，stateDatabase选项即为指定所选用的数据库，默认选用goleveldb。goleveldb实现在common/ledger/util下，couchDB实现在core/ledger/util下。这里的实现不是具体实现数据库，而是将涉及的第三方库的数据库对象包装进fabric自己使用的一些对象（如common/ledger/util/leveldbhelper/leveldb\_helper.go中的DB对象）之中。使用couchDB，可以支持更丰富的方式进行查询。

从数据库用途角度讲，有用于存储账本ID的idStore，用于存储Block的BlockStore，有用于存储链上交易当前最新状态的VersionedDB，有用于存储有效交易部分信息的HistoryDB。

从使用者角度讲，peer结点使用的账本的基本的结构为：数据库db被包含在一个账本实例ledger中，ledger可以被一个账本提供者ledger\_provider提供，ledger\_provider被包裹在一个账本管理者ledger\_mgmt中，ledger\_mgmt供peer点的其他模块直接使用。更详细的请参看图peer\_ledger.png。orderer结点使用账本的基本结构与peer结点类似，只是形成的结构有所出入而已，详情请参看图orderer\_ledger.png。下文只以peer结点使用账本为例详述。

从功能角度讲，对于存储block的数据库，由于区域链不允许对链上的数据进行修改和删除，即已发生的数据永久而不变的保留，因此数据库有的**增删改查**操作，到了fabric用于存储block的账本中只有**增查**两个操作，加上数据库的开启和关闭两个操作，剩余的就是fabric账本针对区域链而保有的一些辅助功能。

## 涉及的目录

* common/ledger
* core/ledger
* orderer/ledger
* protos/ledger

## peer结点中的leveldb

peer结点使用github.com/syndtr/goleveldb/leveldb这个leveldb这个库，是将leveldb的数据库对象和操作都封装进自己的对象中进行使用，在common/ledger/util/leveldbhelper/下，主要有三个对象：（1）DB，在leveldb\_helper.go中，是一个数据库对象，封装了配置conf、leveldb.DB、同步或非同步的读写选项。（2）DBHandle，在leveldb\_provider.go中，是一个数据库处理对象，直接封装了DB和该数据库名称，也是**直接由账本使用**的对象，有Put，Get，Delete这样的基础操作。（3）Iterator，在leveldb\_provider.go中，数据库迭代器，直接封装了leveldb的Iterator，也没有特别之处。

## peer结点中的账本

基础功能：创建，开启，查，增，关闭。

辅助功能：在各个账本章节中详述。

数据库：idStore，BlockStore，VersionedDB，HistoryDB。

peer使用的账本的直接对象是core/ledger/kvledger/kv\_ledger.go中的kvLedger对象，除了idStore被更高层的PeerLedgerProvider管理外，其余三个账本均由kvLedger持有管理。本文将对idStore和BlockStore详述，VersionedDB和HistoryDB则在下一篇文章中详述。

### 创建

1. peer channel join -b gensisblock.block命令，根据gensisblock，创建一个channel。该命令最终会触发以下过程。
2. Invoke(stub) -> joinChain(cid, block)（core/scc/cscc/configure.go），cid即为channel的ID（这个channel的ID在后文实际上就用作账本的ID了），block即为gensisblock。
3. peer.CreateChainFromBlock(block)（core/peer/peer.go） -> l, err = ledgermgmt.CreateLedger(cb)（core/ledger/ledgermgmt/ledger\_mgmt.go） -> ledgerProvider.Create(genesisBlock)（core/ledger/kvledger/kv\_ledger\_provider.go）-> …，根据gensisblock创建账本并保存到了ledger\_mgmt.go中的openedLedgers映射中。这一步账本被创建。
4. 还是在peer.CreateChainFromBlock(block)中，在创建完账本l后，会执行createChain(cid, l, cb) -> c := committer.NewLedgerCommitterReactive(ledger,...)，创建了一个\*\*账本提交者对象\*\*c，而生成的账本l被c持有。账本提交者对象为core/committer/committer\_impl.go中的LedgerCommitter，在peer中向账本提交数据或获取账本信息，均是通过该对象的Commit(block)接口提交给账本的。committer.go中接口的注释中的一句话可以诠释该提交者的目标：leave-everything-to-the-committer-for-now，即现在把所有的事情交给committer。
5. 还是在createChain(cid, l, cb)中，创建了账本提交者对象c后，会执行service.GetGossipService().InitializeChannel(...,c,...) -> state.NewGossipStateProvider(...,committer,...) -> …，c被传到gossip模块的state实例中，被state持有。
6. 这里还有一点可说的是，主要是针对第3点创建账本的，账本有创建和打开两个接口，分别对应core/ledger/kvledger/kv\_ledger\_provider.go中的Create(genesisBlock)，Open(ledgerID)两个实现。创建包含了打开，是在账本不存在时进行的操作，创建的时候会将gensisblock直接写入账本。打开则是在账本已经存在时进行的操作。一个账本存在的标准：idStore中存有该账本的ID 且 该账本的在建标识不存在。关于在建标识下文会详述。

### 使用

回看《fabric源码解析14》state模块章节。在peer结点接收由orderer结点发送来的block数据后（这个过程下文省略），会交由gossip模块中的state子模块向账本添加block。而state添加时，则是通过**账本提交者对象**向账本添加block的。具体过程如下，在gossip/state/state.go中：

1. … -> deliverPayloads() -> s.commitBlock(rawBlock) -> s.committer.Commit(block)，这里的committer即为上文第4、5点创建后被传入state模块并由state模块持有的\*\*账本提交者对象\*\*c。该文件其他函数中还有调用committer.LedgerHeight()，即利用账本提交者对象获取账本对象。
2. 当block一路向账本提交，会到达core/ledger/kvledger/kv\_ledger.go中的Commit(block)处，在这个函数中：
3. l.txtmgmt.ValidateAndPrepare(block, true)，验证并准备block数据，为向VersionedDB中写入做准备。
4. l.blockStore.AddBlock(block)，向BlockStore账本中写入block。
5. l.txtmgmt.Commit()，在第3步的准备下，这里将block数据写入VersionedDB。
6. l.historyDB.Commit(block)，如果if ledgerconfig.IsHistoryDBEnabled()，即HistoryDB配置使能，则也将block数据写入HistoryDB账本。
7. 这里需注意一下，虽说三个账本都是写入block数据，但是写入的数据各有不同，具体写何内容在下文各个账本章节中详述。

## idStore

idStore相对简单，对象为core/ledger/kvledger/kv\_ledger\_provider.go中的idStore，直接使用了leveldb。主要有两个功能：（1）存储创建的账本的ID。（2）使用一个**在建标识**（ConstructionFlag）来标记账本是否正确建立，类似于一个检查点。

### 存储账本ID

存储账本ID相当简单，只有增/查操作，增，以 ledgerKeyPrefix+账本ID 为key，以gensisblock为value，组成键值对进行存储。查，有ledgerIDExists，getAllLedgerIds两个接口，前者判断一个账本是否存在，后者获取所有建立成功的账本ID。

### ConstructionFlag

建立一个账本需要一定的时间，而若在建立账本过程中系统崩溃，则会出现账本部分建立的情况，leveldb中可能残留部分账本数据，磁盘中可能残留部分账本文件，这些需要清除或修复，否则可能会影响账本的再次建立（实际上不影响，但是可以做，具体参看kv\_ledger\_provider.go中的runCleanup(...)）。idStore中以underConstructionLedgerKey为key，账本ID为value，组成键值对进行存储，从而作为一个**在建标识**，标记这个账本正在创建当中。在kv\_ledger\_provider.go中：

1. Create(genesisBlock)创建账本时，在检查完该账本是否存在后，若账本不存在，则立即provider.idStore.setUnderConstructionFlag(ledgerID)，向idStore中添加在建标识，对当前创建的账本进行标记。
2. provider.openInternal(ledgerID)，ledger.Commit(genesisBlock)，先创建账本，这需要一些列的操作，然后再向账本中存储genesisBlock。
3. provider.idStore.createLedgerID(ledgerID, genesisBlock)，在这个函数中，batch.Put(key, val)向idStore中存储该账本ID，batch.Delete(underConstructionLedgerKey)从idStore中删除在建标识。最后s.db.WriteBatch(batch, true)同步的批量写入这两点改变，如此这般这个账本才算完整的建立起来。而在WriteBatch将在建标识真正从idStore中删除前任何时候系统发生崩溃，下次系统重启时读取idStore时，在建标识仍能被读取出来，因此在建标识可以标识账本只被部分在建的情况，此时需要执行recoverUnderConstructionLedger来恢复账本。

**账本恢复**

由ConstructionFlag标志标识着最新的账本是否成功建立，这里分为两种情况，即gensisBlock是否成功写入。在创建账本管理对象PeerLedgerProvider时，会主动尝试恢复一下账本（参看kv\_ledger\_provider.go中的NewProvider()）。在recoverUnderConstructionLedger()中：

1. ledgerID, err := provider.idStore.getUnderConstructionFlag()获取当前idStore中的在建标识，若不存在，则ledgerID == ""成立，说明当前不存在不完整在建的账本，程序返回。否则会继续执行，实际进行账本的恢复工作。
2. ledger, err := provider.openInternal(ledgerID)，再次创建VersionedDB，BlockStore，HistoryDB。bcInfo, err := ledger.GetBlockchainInfo()，从BlockStore中读取block账本信息（包括当前账本高度，当前账本最新存储的block的哈希值，上一块block的哈希值）。
3. switch bcInfo.Height { ... }，根据获取的账本高度，分两种情况：（1）case 0:说明gensisblock未完整写入，则执行provider.runCleanup(ledgerID)，provider.idStore.unsetUnderConstructionFlag()，即清理后删除在建标识。这说明两点，一是block序列号确实是从0开始的，二是在未写入gensisblock的情况下账本算作未建立。（2）case 1:说明gensisblock已写入，则执行genesisBlock, err := ledger.GetBlockByNumber(0)，provider.idStore.createLedgerID(ledgerID, genesisBlock)，获取gensisblock块，并据此在idStore中添加这个账本。
4. **这里需要注意的是**：这里的账本恢复是在账本级别的层面上进行的恢复，而每个账本在创建时进行的各种操作因系统崩溃而造成的部分写入的情况，都由各自账本自己负责恢复或清除（这将在讲其他账本时详述）。而且在此基础上可以说的是，恢复工作的重点是修复和同步数据，而不在于创建直接可以使用的账本对象（创建可以使用的账本对象由其他接口负责，如kv\_ledger\_provider.go中的Create(genesisBlock)，Open(ledgerID)），因此第2点所新创建的各个账本对象，只是为了让这些账本做一下自己所负责的恢复工作而已，这一点可以从第2点bcInfo, err := ledger.GetBlockchainInfo()之后就执行了ledger.Close()将创建的账本对象关闭的操作看出来。

## BlockStore

BlockStore是一个较底层的，基础的，公用的，存储block块数据对象的接口，具体实现为common/ledger/blkstorage/fsblkstorage/blockfile\_mgr.go中的blockfileMgr，统筹管理block的存储操作。**block的存储状态信息**和**block数据自身**是分开存储的：当前block块的存储状态存储在leveldb数据库中，block块数据自身则存储在称为blockfile的文本文件中。换句话说，leveldb用于存储当前账本的block数据的保存状态信息，这些信息主要指block的位置，长度信息，当前blockfile文件大小，index信息等，**主要是用以定位block在blockfile中的所在位置**，而blockfile文件保存实际的block数据。基本的操作方式也是根据leveldb中的信息去blockfile中对block进行读写操作，两者相互配合，共同保证block存储的完整性。另外有一个block迭代器对象，用于逐个读取block块数据的对象。

这里涉及到几个对象（以下所说的block均指账本中当前存在的最新的block，目录以common/ledger/blkstorage/fsblkstorage/为基准）：

### blockfile

存储block块数据的文件。具体实现为block\_stream.go中的blockStream，主要对blockfile进行读写操作。blockfile文件名以blockfile为前缀，以六位数字为后缀，依次增加，如blockfile\_000000，blockfile\_000001，…。

首先说一下block数据在blockfile中具体的存储方式，存储方式决定了如何具体的读取和写入block，所以应先讲。一个完整的**block数据包**包含两部分：block数据长度信息+block数据自身，这里分别用A和B表示，block数据包在blockfile中的偏移也是以A为开始算起的（该例子下文依旧使用）。如一块block数据为blockBytes，则其长度是一个uint64类型的整数为blockBytesLen = uint64(len(blockBytes))，为了节约空间和方便读取，会用blockBytesEncodedLen := proto.EncodeVarint(blockBytesLen)对这个整数编码形成一个变长的数据（原理是在大端系统中删除一个整数低位部分的0）。这里的blockBytesEncodedLen就是A，blockBytes就是B。然后A和B先后写入blockfile后，这块block数据算是写入完成。注意，A是代表了B的长度的数据，A本身也有个长度，即len(A)+len(B)才是这个**block数据包**的长度。其他的对象，比如保存点checkpointInfo，会在写入的时候记录下该块block在blockfile中的位置等信息。

从blockfile中读取B时，会直接在block数据包的开始处（也就是从A开始）直接读取8个字节（A的所在的位置由leveldb存储的checkpointInfo保存），然后调用length, n := proto.DecodeVarint(lenBytes)尝试解码A的实际值。这里直接读取8个字节，是假设这8个字节中一定会完整包含A，笔者实验了一下，对一个512万亿的数字进行EncodeVarint，得到的数据的长度也只有7位，也就是说，这里假设一个block的大小值被EncodeVarint后的数据一定<=8位。另外，如当A只有3位，读取的8位数据的剩余5位为B的数据时，DecodeVarint依然会解码出A来，此时返回的n为3，length则为A代表的实际值（也就是B的实际大小）。然后，根据解码出来的length，以及A所占的n个字节，则可以定位到B开始的地方，然后读取length个字节，也就完整的把B读取出来了。对于B是否完整，是通过比较当前blockfile的大小与**length+n+checkpointInfo记录的block数据包的偏移值**之和的值来判定的，若后者小于当前blockfile的实际大小值，则自然是完整的。

读取block数据包，是由block\_stream.go中的blockStream和blockfileStream对象完成的。其中能体现上述读取过程的，是两个对象的nextBlockBytesAndPlacementInfo接口。该接口每次从指定的文件中的指定位置开始，尝试读取一个block数据包。

### block信息数据库

存储当前账本中**block存储状态信息**的leveldb数据库，具体到blockfileMgr对象中的db，index两个成员（两者实际上使用的是**同一个** leveldb数据库）。主要存储两种数据：（1）**检查点**，checkpoint，具体为blockfile\_mgr.go中的checkpointInfo对象。**每在blockfile中存储一个block数据包，都会在leveldb中存储该block数据包对应的检查点**。检查点以blkMgrInfoKey为key（blockfile\_mgr.go中），checkpointInfo为value，组成**一个键值对**存储在leveldb中。latestFileChunkSuffixNum记录block所在的blockfile的文件名后缀，latestFileChunksize记录当前blockfile最新的大小，isChainEmpty标识当前账本是否为空，lastBlockNumber记录账本当前最新block的序列号。（2）**索引**，index，具体为blockindex.go中的blockIndex对象。**每在blockfile中存储一个block数据包，都会在leveldb中存储该block数据包对应的一批索引**。索引分别以block序列号，block哈希值，交易ID等为key，以账本中最新block的位置信息等为value，组成**一批键值对**，存储在leveldb中。这些索引项的key值预定义在common/ledger/blkstorage/blockstorage.go中，主要是为调用者提供多种的索引方式去在blockfile中定位block块数据，比如有的想用block的序列号去查找一个block数据，有的想使用block数据的哈希值去查找block数据，有的想使用交易ID去查找一个block中的具体交易的数据，等等。block数据包对应的每批索引中，包含了block块中每笔具体交易数据的索引，具体存储在blockIndex对象的txOffsets，这个成员在计算每笔交易在blockfile中的偏移位置时，随着添加A+B，总共经历了三轮，首先计算的是每笔交易在B中的相对偏移，然后计算的是每笔交易在A+B中的相对偏移，最后计算的是每笔交易在blockfile中的偏移。索引自身也有一个检查点indexcheckpoint，用于记录当前最新的block数据包的索引的存储情况。

在写block数据包时：当先后写入A和B后，**（1）**创建一个新的checkpointInfo对象newCPInfo，然后根据现有的checkpointInfo对象currentCPInfo和写入的block数据包填充newCPInfo，再非同步的调用db.Put(...)将新的checkpointInfo写入leveldb中，覆盖掉currentCPInfo。检查点的作用主要在于记录当前账本的保存状态，也就是说记录账本当前保存到哪儿了，保存到哪个block了。**（2）**调用indexBlock(...)，进而调用batch.Put(...)，以批量写入的形式将各个索引对应的信息写入leveldb中，因为每批索引所使用的key不会一样，因此不会发生覆盖。索引检查点会在每批的索引的最后写入，也即当索引检查点存在，则当批索引一定存在。索引的作用主要在于为调用者提供多种查找具体block或交易数据的方式。

在此**总结一下**，写入一个block，从前到后所涉及的要依次写入的每个数据：

1. block长度信息
2. block块数据
3. blkMgrInfoKey - checkpointInfo
4. blockHashIdxKeyPrefix+block哈希值 - block数据包位置信息
5. blockNumIdxKeyPrefix+blockID - block数据包位置信息
6. txIDIdxKeyPrefix+交易ID - 交易数据位置信息（每个block中所含的所有交易）
7. blockNumTranNumIdxKeyPrefix+blockID+交易Seq - 交易数据位置信息（每个block中所含的所有交易）
8. blockTxIDIdxKeyPrefix+交易ID - block数据包位置信息（每个block中所含的所有交易）
9. txValidationResultIdxKeyPrefix+交易ID - 交易验证结果（每个block中所含的所有交易）
10. indexCheckpointKey - block序列号

上述列表中，可划分出两个范畴，三个小整体：1和2存储到blockfile文件中，属于一个范畴，是一个小整体；3-9存储在leveldb中，均为键值对（key-value pair），属于一个范畴；3为检查点键值对，是一个小整体；4-10为索引键值对，10是索引检查点，是一个小整体。blockID指block的序列号，交易ID指交易的序列号，交易Seq指交易在block的Data数组中的下标序号。所有的key的…Prefix前缀均在blockindex.go中定义。3和10的每次更新均会覆盖旧的值。

block数据包或交易数据的**位置信息**是一个blockindex.go中的fileLocPointer对象，该对象中，fileSuffixNum记录数据所在的blockfile文件后缀，offset记录数据在blockfile文件中的偏移，也就是数据从哪个位置开始的，bytesLength记录数据的大小（这个字段，block数据包未使用，因为其长度存放在A中，没必要使用这个字段。而交易数据使用了这个字段，来记录每个交易数据的大小）。

同时，正因为在写入一个block时涉及到多个、多类数据，并先后写入，因此存在当系统崩溃时**数据部分写入和三个小整体数据不同步的情况**。也因此在重新启动系统建立新的blockfileMgr时，需要对部分写入的数据进行修补和对三个小整体的数据进行同步。同步时，遵循的一个隐含的规则是前面的数据未写入，则后边的数据肯定没有写入，存在的情况有：（1）1未完整写入，（2）1完整写入，2未完整写入，（3）1-2完整写入，3未完整写入，（4）1-3完整写入，4-10未批量写入。据此，同步的步骤如下：

1. 从leveldb中获取存有的检查点信息cpInfo（如果没有则说明是新建的账本）。
2. 根据cpInfo中存储的block数据包的位置信息，定位该block数据包在哪个blockfile中的哪个位置。然后以此为开始尝试读取一个完整的block数据包（读取方法在上文blockfile章节已经叙述过），对cpInfo进行更新同步。
3. 这里需要辨别的是（1）-（3）三种情况，这三种情况下，取出的cpInfo其实是上一个block数据包对应的cpInfo，定位读取的时候也会完整读取出上一个block数据包，然后再读取到此block时，当是情况（1）（2）时，由于当前block数据包数据未完整写入，因此会直接从上一个block数据包的末尾处截断blockfile文件，删除当前block数据包不完整的数据，此时也不需要再更新cpInfo；当是情况（3）时，由于当前block数据包数据已完整写入，因此会再读取当前block数据包，依据当前block数据包的信息，更新cpInfo和索引（索引会使用更新过后的cpInfo进行更新）。当是情况（4）时，取出的cpInfo为当前block数据包对应的cpInfo，cpInfo不用更新，而只需要对索引信息进行同步。索引同步时，会读取索引的检查点信息，该索引检查点信息必然是上一个block数据包的索引检查点，而由此获取的block数据包位置信息也是上一块的block，跳过该block后，就同cpInfo更新的方式类似。

### block迭代器

迭代器自身没有太多可说的，依旧是调用者给一个从何处开始的参数，然后调用Next()依次遍历。当所要读取的block序列号大于账本中最新的block序列号时，迭代器会等待，直到新的block块产生并成功写入blockfile文件中。

### 创建-增-查

这里在代码中追溯一下BlockStore的主要操作，细节部分（如偏移量如何计算确定等）未细说。以下涉及的代码的基准目录为common/ledger/blkstorage/fsblkstorage/，涉及到的函数若为表明所在位置，请自行搜索。

**创建**

1. blockfile\_mgr.go中，newBlockfileMgr(...)创建了一个新的block存储管理者，接收了配置，索引配置和一个leveldb数据库三个参数。其中配置中包含了block存储路径等信息。leveldb以参数的形式传入，给了更高层的管理对象，如fs\_blockstore.go中的fsBlockStore，fs\_blockstore\_provider.go中的FsBlockstoreProvider，更大的灵活性。
2. cpInfo, err := mgr.loadCurrentInfo()，即是从leveldb中取出最新的检查点数据。当if cpInfo == nil，则说明当前新建的block存储管理者是管理的一个比较新的账本，要么什么数据都未写入，要么写入了第一块block数据包的部分数据（参看上文数据部分写入的情况（1）-（3）），则可以将cpInfo更新为blockfile文件开始处（以供后续步骤进行尝试读取现有的block数据，然后更新cpInfo）。
3. syncCPInfoFromFS(rootDir, cpInfo)，即根据取出来的cpInfo同步更新的函数。在这个函数中，scanForLastCompleteBlock(...)是尝试读取最后blockfile最后一块block的函数，所给的三个参数分别是哪个目录，哪个文件（后缀），从文件里的哪个位置开始读，第三个参数赋的值为int64(cpInfo.latestFileChunksize)，即cpInfo所对应的block数据包写入后文件的大小，也就是从cpInfo所代表的block数据包末尾的下一个位置（即下一个block数据包开始的位置）开始尝试读取一个完整的block数据包，如此，至多读取出一个完整的block。读取的方法依然如上文所述，具体的实现是block\_stream.go中的nextBlockBytesAndPlacementInfo()。若读取了这个至多的一个新的完整的block数据包，则会更新cpInfo的latestFileChunksize（指向blockfile中最新的block数据包尾），lastBlockNumber（更新为最新block的序列号），isChainEmpty（更新为账本非空）三个字段。
4. currentFileWriter.truncateFile(cpInfo.latestFileChunksize)，根据更新过后的cpInfo所存储的最后一个block数据包的末尾位置，直接截断该blockfile文件。这比较好理解，更新过后的cpInfo的latestFileChunksize值之后的数据都肯定是不完整的，所以直接截断，清除不完整的数据。
5. mgr.syncIndex()，根据更新过后的cpInfo和自身的检查点信息，创建同步索引。在这个函数中，lastBlockIndexed, err = mgr.index.getLastBlockIndexed()即为获取索引的检查点，获取的是解码后的block序列号，也即表示序列号为lastBlockIndexed的block数据包的索引已经完整存储，同样，若未获取，则说明所处理的账本较新，还没有索引存储进leveldb中。flp, err = mgr.index.getBlockLocByBlockNum(lastBlockIndexed)是使用blockNumIdxKeyPrefix+blockID的索引方式获取序号为lastBlockIndexed的block数据包位置信息flp。又因为能成功获取索引的检查点，所以可知该block数据包一定是完整存储的，因此可以直接跳过这个block数据包而去尝试读取下一个block，这也是函数中skipFirstBlock变量所起到的作用。而一旦能成功读取下一个block数据包，则会调用info, err := extractSerializedBlockInfo(blockBytes)根据读取出来的block数据包创建新的索引信息，然后调用mgr.index.indexBlock(blockIdxInfo)将新的block数据包的索引写入leveldb中，从而完成索引的更新同步。

**增**

1. addBlock(block)对一个block数据块进行了添加。在这个函数中，if block.Header.Number != mgr.getBlockchainInfo().Height首先验证了添加block数据的序列号是否是下一个block应有的，即序列号是依次连续的。blockBytes, info, err := serializeBlock(block)将block数据串行化成可以直接写入blockfile文件中的数据，**这里的blockBytes即为B**，也在这里第一次计算了每笔交易数据在block块内的偏移位置（即以block.Data为开始每笔交易的偏移）。currentOffset := mgr.cpInfo.latestFileChunksize即为当前blockfile的大小，也就是准备开始写当前添加的block数据包的位置。blockBytesLen := len(blockBytes)，blockBytesEncodedLen := proto.EncodeVarint(uint64(blockBytesLen))压缩了block的大小值，**这里的blockBytesEncodedLen即为A**。totalBytesToAppend := blockBytesLen + len(blockBytesEncodedLen)则为block数据包的总大小。如果if currentOffset+totalBytesToAppend > mgr.conf.maxBlockfileSize，即当前文件的大小+要添加的block数据包的大小之和大于配置中规定的blockfile的大小限制，则会调用mgr.moveToNextFile()直接新启用下一个blockfile文件来存储这个新的block数据包。
2. mgr.currentFileWriter.append(blockBytesEncodedLen, false)，非同步的在第一步确定的blockfile文件和文件中的位置处的写入A，第二个参数给的是false，因此此时A只是在缓存中而已。mgr.currentFileWriter.append(blockBytes, true)，第二个参数是true，同步写入B，即连同A一同实际的写入blockfile磁盘文件中。需要注意的是，虽然这里A和B是一起写入的，但只是为了减少磁盘文件的操作，第二个参数为true，在append(...)函数（blockfile\_rw.go中）中会手动调用file.Sync()，即手工刷新缓存将缓存中的数据写入实际的磁盘文件，但在这个操作也不是事务性的，即在系统层面也是一个字符一个字符向磁盘文件中写的，因此如果在写的过程中发生系统崩溃，仍会出现A或B部分写入的情况。如果写入A，B的过程中出错，则mgr.currentFileWriter.truncateFile(mgr.cpInfo.latestFileChunksize)直接把文件从最开始写的地方截断，即将可能新写入的部分block数据包的数据删去，然后添加程序结束并返回一个错误。
3. newCPInfo := &checkpointInfo{...}，根据写入的block数据包和现有检查点的信息，创建一个新的检查点。然后mgr.saveCurrentInfo(newCPInfo, false)，非同步的将新的检查点写入leveldb数据库中。同样的，若是检查点添加错误，也是直接截断文件，删除写入的block数据包，将blockfile恢复如初，结束添加返回一个错误。
4. blockFLP := &fileLocPointer{...}，根据新的检查点newCPInfo和开始写入block数据包时的偏移信息，创建一个block数据包的位置信息blockFLP。for ... {txOffset.loc.offset += len(blockBytesEncodedLen)}，这里第二次计算block中包含的每笔交易的偏移，即加上了A所占的字节数，计算每个交易从A处开始算起的偏移量，也可以说是每笔交易在block数据包内的偏移。mgr.index.indexBlock(...)，将收集的索引使用到的关于新的block数据包的信息传入，创建新的一个block数据包的索引，在这个函数中，会逐个写入1-6的索引项，其中在添加索引3和4（上文数据6和7）这两个与交易有关的索引时，会调用txFlp := newFileLocationPointer(...)，第三次计算每个交易的偏移信息，即确定每笔交易在blockfile文件内的偏移，同时也会在txFlp中记录每笔交易的大小。在所有索引项都batch写入后，然后batch.Put(indexCheckpointKey, encodeBlockNum(blockIdxInfo.blockNum))写入索引的检查点，最后index.db.WriteBatch(batch, false)非同步的写入leveldb数据库中。这里的非同步和第3点写入leveldb时都是用了非同步写入，即执行后可能这些数据是写入leveldb的缓存中而非数据库中，这样可以提高写入的效率。
5. mgr.updateCheckpoint(newCPInfo)，这个主要更新了blockfileMgr对象保存的检查点cpInfo，然后调用了一个mgr.cpInfoCond.Broadcast()函数，这个函数主要使用sync.Cond的特性，服务于通知block迭代器的Next()可能存在的等待，让其继续执行。上文已经说过，迭代器的Next()在遍历到序列号大于账本当前已有的最新block数据包的序列号时，会进入等待，一直到新的block数据包成功存储到blockfile中后，所以这里就是在成功添加了一个block数据包之后进行的通知。mgr.updateBlockchainInfo(blockHash, block)，这个函数使用atomic.Value的特性，原子性的存储链（也就是账本）的信息，即一个BlockchainInfo对象，该对象Height保存了链的高度，CurrentBlockHash保存了当前链存储的最新的block的哈希值，PreviousBlockHash保存了上一个block的哈希值。

**查**

关于查询block数据很简单，不同索引所提供了不同的查询方式，均集中在blockfile\_mgr.go，是一系列retrieve\_\_By\_\_格式的函数，名字很好理解，第一个空是你要查的东西，第二个空是你根据什么查这个东西，即通过什么来检索什么，也即By什么来retrieve什么。这些函数主要通过leveldb中保存的block数据包的索引来获取block数据。

以上即是对idStore和BlockStore的详述，下篇文章将对更复杂的VersionedDB和HistoryDB进行详述。

# fabric源码解析25——ledger之VersionedDB和HistoryDB

## 续接前文

续接《fabric源码解析24》，本文讲述另外两个账本VersionedDB，HistoryDB。为了比较清楚的说明问题，本文将以peer invoke...命令调用examples/chaincode/go/下的chaincode\_example02/chaincode\_example02.go和map/map.go两个chaincode为例（假设两个chaincode已正确部署）。

## VersionedDB

VersionedDB账本可以叫做状态账本，这个状态（state），即是官方文档中所提及的“世界状态”，world state，具体即为由每个交易数据**最新生成**的一个关于交易读写集的**有效的**键值对。该数据库中有几个重要的对象，如交易模拟器、查询器、验证器、读写集等，其中这些对象所体现的一些概念和操作还是比较难理解的，笔者在这卡了将近两个星期，所以这里强烈建议先阅读一些官方文档的State Database和Read-Write set semantics两处的内容。

VersionedDB使用的state数据库有两个版本，即leveldb版本和couchdb版本，这里只讲leveldb版本，两者的主要区别在于：leveldb只支持基本的以key为基础的查询，而couchdb则支持更丰富的查询手段，即富查询（参看core/ledger/util/couchdb/couchdb\_test.go中的TestRichQuery），对于调用者来说也更接近事物逻辑（如可以根据某账户的特征，如颜色，尺寸等特征进行查询，只需要预先定义好）。state数据库也不是直接由账本使用的，而是被一个可称作**交易管理者**的对象TxMgr持有并管理，账本通过这个交易管理者向state数据库读写交易数据。同时，TxMgr对象也提供其他功能，比如提供交易模拟器、交易查询器、交易验证器，这些“器”将在读写交易数据的时候发挥作用。TxMgr的基础目录在core/ledger/kvledger/txmgmt/下（该章节以此目录为基准），具体的实现为txmgr/lockbasedtxmgr/lockbased\_txmgr.go中的LockBasedTxMgr。

在peer结点中使用VersionedDB的过程：用户使用peer结点发起一笔交易，如ACC的部署，是一个明显的会写入数据的交易，再如用户执行peer chaincode query ...，则是一个明显的会读出数据的交易。在交易过程中，这些读取和写入的交易数据会通过TxSimulator（交易模拟器）放入一个叫做**读写集**的容器，在交易返回时，再使用TxSimulator统一读取，然后将读取的数据放入ProposalResponse返回，返回后读集中的数据直接打印，写集中的数据被放入Envelope中再发送给orderer（请参看peer/chaincode/common.go中的ChaincodeInvokeOrQuery）。orderer结点依据Envelope生成block后，再返回给peer结点。peer结点接收后，经gossip模块接收并提交到账本对象后，账本对象添加block，会添加到VersionedDB账本中，VersionedDB又会将block中交易的数据最终添加到state数据库中。以执行chaincode\_example02.go的invoke(stub,args)（该函数简单的查询了A、B两个账户的余额，并由A向B转了一笔钱，即形成了A、B两个账户转账调整后各自新的余额值）为例，粗线条的步骤如下：

1. 交易发起端执行peer chaincode invoke -n chaincode\_example02 -c '{"Args":["invoke","a","b","10"]}'，通过peer/chaincode/invoke.go中的chaincodeInvoke(...)函数向endorser结点发送请求。假设该交易为Tx\_A。
2. core/endorser/endorser.go中，ProcessProposal -> txsim, err = e.getTxSimulator(chainID)，创建了一个交易模拟器。
3. e.simulateProposal(...,txsim) -> e.callChaincode(...,txsim)。
4. 经过一系列辗转，会在core/chaincode/shim/handler.go中，handleTransaction中调用res := handler.cc.Invoke(stub)（即chaincode\_example02.go中的Invoke） -> 根据第1步的参数，在Invoke()中进入if function == "invoke"分支从而调用invoke() -> 再经过一系列辗转，会在core/chaincode/handler.go中，enterBusyState，通过交易模拟器的txsimulator.GetState()、txsimulator.SetState(...)，将A、B账户的数据放入读集，把更改的A、B账户的数据放入了写集（其他交易会用到了其他的DeleteState()，GetStateRangeScanIterator()，ExecuteQuery()），这一步相当于模拟执行了交易。
5. 重回core/endorser/endorser.go中，simResult, err = txsim.GetTxSimulationResults() -> 返回至ProcessProposal()-> pResp, err = e.endorseProposal(...,simulationResult,...) -> e.callChaincode(...)，获取交易的写集，并进行了第二轮的callChaincode，一系列调用后会进入下一步。
6. core/scc/escc/endorser\_onevalidsignature.go中，Invoke(...) -> presp, err := utils.CreateProposalResponse(...,results,...)，results为交易Tx\_A的结果集，被放入ProposalResponse返回至第5步的endorseProposal，在返回给交易发起端。
7. peer/chaincode/common.go中，ChaincodeInvokeOrQuery() -> env, err := putils.CreateSignedTx(...)，把返回的ProposalResponse打包成Envelope -> err = bc.Send(env)，发送给orderer -> …，orderer处理block过程省略 -> orderer将形成的block发送给peer结点，再经gossip模块散播使用committer模块的过程，这里亦省略。
8. committer向账本提交block时，最终定位到core/ledger/kvledger/kv\_ledger.go中的Commit(block) -> err = l.txtmgmt.ValidateAndPrepare(block, true)，l.txtmgmt.Commit()即是使用交易管理者向state提交交易数据。ValidateAndPrepare做两件事，一是使用验证器验证交易的读写集，以确定交易的有效性，二是若交易有效，则将交易的写集中的数据放入数据升级包中，为下一步的l.txtmgmt.Commit()提交这批数据做准备。

### 交易模拟器/交易查询器

交易模拟器TxSimulator和交易查询器QueryExecutor的接口在core/ledger/ledger\_interface.go中定义，具体实现为txmgr/lockbasedtxmgr/下lockbased\_tx\_simulator.go中的lockBasedTxSimulator和lockbased\_query\_executer.go中的lockBasedQueryExecutor。通过账本kvLedger的NewTxSimulator()，NewQueryExecutor()接口可获取TxSimulator和NewQueryExecutor。两者均直接使用helper.go中的queryHelper实现了自身的查询方面的功能，这点从queryHelper这个名字可以看出来。TxSimulator实际上包含QueryExecutor，而QueryExecutor算是TxSimulator在查询功能上的增强和拓展（其中交易查询器的ExecuteQuery是富查询接口，因此只支持couchDB版本的VersionedDB），因此直接使用的一般是交易模拟器。除了查询，TxSimulator还提供状态的写入功能，写入分为增加和删除。因此，TxSimulator和QueryExecutor涉及的操作就有**查、增、删**。

参看txmgr/lockbasedtxmgr/lockbased\_tx\_simulator.go和helper.go，交易模拟器的读、写、删的操作步如下：

* GetState(ns, key) -> q.helper.getState(ns, key)，依据名字空间和key，从state数据库中获取一个状态，返回并写入读集（返回的是**值**，写入读集的是**值的版本**）。这里的**名字空间**，即是chaincodeID，也即对应一个chaincode的每一个交易使用一个读写集，下同。
* SetState(ns, key, value) -> s.rwsetBuilder.AddToWriteSet(ns, key, value)，将一个值写入写集。
* DeleteState(ns, key) -> SetState(ns, key, nil)，删除一个值，当给的key的value为nil时，即表示要将此key的值置为nil，也即要删除这个值。
* SetStateMultipleKeys(ns, kvs) -> for ... { SetState(...) }，一次性设置多个键值对，写入交易的写集。
* GetStateMultipleKeys(...) -> q.helper.getStateMultipleKeys(...)，依据一个名字空间和一批key，从state数据库中获取一批状态。
* GetStateRangeScanIterator(...) -> q.helper.getStateRangeScanIterator(...)，依据一个命名空间，开始的key，结束的key，从state数据库中获取一个指定范围的交易查询迭代器。
* ExecuteQuery(namespace,query) -> q.helper.executeQuery(namespace,query)，这个接口对于leveldb来说不支持。
* Done() -> q.helper.done()，交易查询器执行完毕。

**读写集**

TxSimulator之所以叫交易模拟器，就是因为在使用它处理交易的时候，所形成的交易数据并未真正的直接写入state数据库中，而是将交易查、增、删得到的数据暂时放入了一个叫做**读写集**的地方备用，因此是**模拟**交易。

一个chaincode的每笔交易都对应一个读写集，读写集实现为rwsetutil/rwset\_builder.go中的RWSetBuilder，成员只有一个rwMap map[string]\*nsRWs映射，即以chaincodeID为key，每个chaincode单独有一个nsRWs。参看读写集结构图片read\_write\_set.png，读写集存储三类数据：KVRead读值、KVWrite写值、RangeQueryInfo范围读值（三者均定义在protos/ledger/rwset/kvrwset/下），多个值各自形成readMap读集、writeMap写集、rangeQueriesMap和rangeQueriesKeys组成的范围读集。

**读值**：KVRead，每个读值只包含key和值的版本号，而不包含值本身。TxSimulator每次调用GetState()接口，都会将读取到的读值写入交易的读集。**写值**：KVWrite，每个写值包含key和value，还包含一个此写值是否为删除的标识IsDelete。TxSimulator每次调用SetState()、DeleteState()等接口，都会将一个写值写入交易的写集。**范围读值**：RangeQueryInfo，每个范围读值是一个范围内所有读值的集合，范围读值何时被写入将在下文提及。rangeQueriesMap以rangeQueryKey为map的key，RangeQueryInfo为map的value。rangeQueryKey中，startKey标识了范围从何开始，endKey标识了范围至何处结束（注意，范围读值中不包含endkey对应的值），itrExhausted标识了是否结束。RangeQueryInfo中，其余的成员同rangeQueryKey一样，而成员ReadsInfo存储两类数据：原始的范围内的多个读值或范围内的读值的哈希值。范围内的多个原始的读值很容易理解，而范围内读值的哈希值的生成则需要借助其他工具，且往下看。

当用户发起读取一定范围数据的交易时，不是直接将这个范围内所有读值集合打包返回给调用者，而是将一个包含了读值范围信息的的**迭代器**返回给调用者（这样可以避免数据量过大而导致的处理效率低下），然后调用者使用迭代器的Next()一个一个抽取读值。这里涉及到两个迭代器：（1）在交易模拟器的范围内，为txmgr/lockbasedtxmgr/helper.go中的resultsItr，由交易模拟器的GetStateRangeScanIterator(...)接口获取，每获取一个迭代器，都会把迭代器记录在queryHelper中的itrs，再次强调，TxSimulator包含QueryExecutor，而提供迭代器也是QueryExecutor对TxSimulator在查询功能上上主要的拓展之一。（2）在核心的chaincode处理范围内，为core/chaincode/shim/chaincode.go中的StateQueryIterator。该迭代器的使用依附于（1）中迭代器所查询出来的结果。

**迭代器**

resultsItr迭代器管理了一个工具，为rwsetutil/query\_results\_helper.go中的RangeQueryResultsHelper（亦是“人如其名”）。RangeQueryResultsHelper中的pendingResults用于暂时存储范围内的读值，merkleTree用于存储一棵默克尔树，该树存储pendingResults对应的哈希值，两者是随着数据的增加同步更新的（其实哈希值是否同步生成是由hashingEnabled决定的，而该值又由账本的配置决定的，该配置默认开启，由core/ledger/ledgerconfig/ledger\_config.go中的IsQueryReadsHashingEnabled()接口决定）。每次resultsItr迭代器被调用一次Next()，读取的读值都会添加到RangeQueryResultsHelper中的pendingResults和merkleTree中暂存。

这里跳开粗略的讲一下默克尔树，也叫哈希树。实现为query\_results\_helper.go中的merkleTree，由工具RangeQueryResultsHelper持有并管理。在merkleTree中，一个map[MerkleTreeLevel][]Hash格式的映射中实现了该树，树的每个结点保存的是哈希值。默克尔树中有两个比较重要的值：Level和maxDegree。Level标识树的层级，默认为1，当树中的Level1有maxDegree+1个结点时，则会进行一个**归并哈希值**的操作：如两个结点存有hash1，hash2，该操作是将hash1和hash2的值前后连成一个整体，然后对这个整体进行哈希，得到一个新的哈希值hash1-2，然后将hash1-2放入上一层（也就是Level2）里面。下面也是举例子说明默克尔树的操作，对看merkleTree的update(nextLeafLevelHash)函数：

//设maxDegree==2，则连续插入1-9共9个哈希值，即调用9次merkleTree的update接口

//[x]-{hashN}，x为Level值，hashN代表N的哈希值。则tree的变化为：

1.[1]-{hash1}

2.[1]-{hash1,hash2}

3.[2]-{hash1-3}，插入第3个时，因为len(currentLevelHashes) <= m.maxDegree不成立，所有会向下一层，也就是第2层归集哈希值，同时删除第1层的值

4.[1]-{hash4},[2]-{hash1-3}

5.[1]-{hash4,hash5},[2]-{hash1-3}

6.[2]-{hash1-3,hash4-6}，插入第hash6时，同第3步原因一样和操作一样

7.[1]-{hash7},[2]-{hash1-3,hash4-6}

8.[1]-{hash7,hash8},[2]-{hash1-3,hash4-6}

9.[3]-{hash1-9}，这一步插入hash9时，hash7-9向第2层归集，同时删除第1层的值，此时第2层也够了3个，此时会继续向第3层归集，归集第2

层hash1-3，hash4-6，hash7-9三个哈希值，形成第3层的hash1-9，同时会删除第2层的值

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

当工具RangeQueryResultsHelper工具执行Done()时，会将存储的pendingResults和merkleTree一同返回，两者实际上只会返回其中一个，其中一个必为nil。当pendingResults中的存储读值的数量小于等于maxDegree时，是不会触发归并哈希值的操作的，因此此时返回的merkleTree为nil。其他情况下，因为hashingEnabled默认为true，所有只会返回一个归并后的唯一的哈希值。这个哈希值就代表了一个resultsItr迭代器被调用Next()而被读出的所有读值。返回一个归并后的哈希值而不返回原始的范围读集，这样在效率上更高。另外从这一点还可以看出，返回范围读值，只可能是被用于验证的目的，原因有二：一是读值中本身就没有key对应的值，而只有版本号。二是这里可以只返回范围读集的哈希值。

以map.go为例，当调用Invoke(stub)中的case "keys":分支时，keysIter, err := stub.GetStateByRange(startKey, endKey)即获取了一个迭代器，然后在for keysIter.HasNext()中依次抽取值。keysIter即为StateQueryIterator，该迭代器中存储了执行范围查询操作后所查出来的读值集合：（1）在core/chaincode/handler.go的handleGetStateByRange(...)中，rangeIter, err := txContext.txsimulator.GetStateRangeScanIterator(...)，使用交易模拟器获取一个resultsItr，该迭代器会被放入queryHelper中的itrs。（2）payload, err = getQueryResponse(...)，，使用resultsItr这个迭代器将指定范围内的结果查询出来，放入payload。这里就有一个maxResultLimit的限制，即查询范围的起止最多是100个数据。同时也就是说，传入的起止的key所得到的查询结果，不一定是该范围内的全部数据。而指定范围内是否还有更多的数据，会使用payload中的HasMore变量进行标识。（3）payload最终被返回到core/chaincode/shim/chaincode.go中的handleGetStateByRange()，被放入StateQueryIterator这个迭代器中。

当交易模拟器将交易模拟完毕，即读写集填充完毕，在交易返回给调用者之前，会一次性将交易读写集中的数据进行抽取，放入到应答信息ProposalResponse中，即上文关于peer结点使用VersionedDB的过程中的第6步，也即调用TxSimulator的GetTxSimulationResults()接口：（1）s.Done()，只能执行一次。TxSimulator每次执行范围查询时，不会像读取单个值一样直接就把查询的数据放入读集中，而是只将迭代器存储在queryHelper中的itrs，这些迭代器即代表了交易中读取了哪些范围内的数据。等到此时执行Done时，才会依据这些迭代器，一个迭代器对应一个范围读值，将每个范围读取的所有读值填写范围读集中。（2）s.rwsetBuilder.GetTxReadWriteSet()...，该函数主要起到一个排序，然后换个包装的作用。先使用sort.Strings(keys)对读集、写集、范围读集按照key进行排序，然后将读写集从RWSetBuilder中换装到rwsetutil/rwset\_proto\_util.go中定义的TxRwSet里面。

### 验证器

state数据库是保存的最实时最新的状态，peer结点进行查询操作时，也是从该账本读取的数据。所以在向state数据库提交交易数据前要使用验证器进行验证（对看上文peer结点使用VersionedDB的过程的第8点）。验证的时候，无效的交易将被屏蔽掉而不会提交到state账本中。因此需要一个验证器，这个验证器在validator/statebasedval/state\_based\_validator.go中实现为Validator，其ValidateAndPrepareBatch(block,domvcc)接口负责实现验证功能，其中第2个参数domvcc决定了是否进行mvcc验证，该值给的为true。关于mvcc（多版本并发控制），各位可以自行搜索参看一下与数据库相关的mvcc的作用。

存在验证就存在对比，因为验证只能通过对比。**对比的内容**：key对应的版本号。**对比的双方**：甲方是block中携带的读集和范围读集，乙方是验证时现从state数据库里查询出的甲方对应的读值和范围读集+由甲方自带的有效交易的写集形成的升级数据包updates（参看下文）。范围读集中虽说可能是哈希值，但是这些哈希值也是在原始读值（包含key和版本）的基础上形成的哈希值，在验证的时候，再从state数据库中读取同样范围内的读值集合，按照同样的方法生成一个新的哈希值，如果两个哈希值不同，则间接说明了当前state数据库中这个范围内的值已经有所变动，即比较哈希值其实也是在比较key对应的版本号。如现在验证的block中一个Envelope中的读集中有两个读值<read key="K1", version="1">,<read key="K2", version="1">，有一个范围读值<queryinfo,startkey=10,endkey=20,true,hash10-20>（注意乙方的updates在这个例子里面没有体现，更好的例子请参看官方文档Example simulation and validation   
章节）。**验证的方法**只有一个：验证时，依次将K1，K2的version与现从state数据库中现读出K1，K2的版本号相比较，然后将hash10-20与现从state数据库中读出startkey-endkey间的读值并按照同样的归并哈希的方法生成的新哈希值hash10-20'进行比较。概括点说，就是拿之前模拟交易产生的读集中的版本号与state先有的相应的版本号进行比较。**验证的标准**只有一个：若以上**所有**的比较均相同，则判定此交易有效，任何一个比较不同，则判定交易无效。**验证的目的**只有一个：将有效交易的写集数据放入批量升级包中，准备更新state数据库。

关于理解为何如此判定交易的有效性，有几点说明：

1. state是存储当前账本所有最新有效交易的世界状态的数据库。这里的关键词是**最新**、**有效**、**交易**。
2. 所有block均为串行化后的数据。state数据库中值的版本号由blockID+TxID组成，blockID为序列递增，TxID则为交易在block中一批交易中的相对位置（即交易在block.Data.Data这个数组中的下标值，亦是相对序列递增）。因此当一个新的有效交易的数据被提交至state数据库中进行更新，该值的版本号与原值的版本号必然不一样，也即更新state数据库的值，值的版本号必然变化。
3. 模拟交易读出的数据都会放入读集中。这里假定调用者在执行chaincode的模拟交易时，读取出来的数据会被使用。这个也很正常，既然读出来了，就有被使用的可能。比如chaincode\_example02.go中，即是先查出来A，B账户的余额数据，并在此基础上进行的数据的加减。因此如果读集的数据失效，在此基础上得出的写集中的数据也必然无效。
4. fabric对交易是**并发并行**处理的，且自模拟交易产生读写集 -> 数据返回到peer结点 -> 送到orderer -> 再返回peer的gossip -> 再准备提交至state数据库，这个过程会耗费一段时间，而这段时间内，原有交易所使用的读出数据有可能已经被其他稍早点儿的交易改变。比如Tx\_1查到的A的余额为100，B的余额为100，A把50元转给B，则形成的读集是A-v1，B-v1，写集是A-50，B-150。这个读写集在到达Validator验证之前，经历了一段时间。就在这一段时间内，稍早点的交易Tx\_0已经把A的20元转给了B，A的余额为80（假定版本变为v2），B的余额为120（假定版本变为v2）。如果此时依然判定Tx\_1有效，将Tx\_1的写集提交至state数据库，则将会覆盖Tx\_0的交易，此时A的余额仍然为50（实际应该是30），B的余额仍然为150（实际应该是170），这样肯定不行。
5. 由第4点，可能有人会推出另一个疑问：既然从Tx\_1在交易模拟器生成读写集数据到提交前验证之间会耗费一段时间，存在稍早的交易Tx\_0已经把state数据更改而使得Tx\_1无效的风险，那么Tx\_1在通过验证到真正提交到state数据库中依然存在一段时间，这期间也可能存在被稍早的交易Tx\_N将state数据更改的风险，这点如何保证呢？其实不存在Tx\_N这种情况，因为如第2点所说，读写集是由block带入的，而block均为串行化依次处理的。这样的话只会存在两种情况：（1）Tx\_0与Tx\_1不在同一个block中，则Tx\_0所在的block早于Tx\_1所在的block。此时一定是Tx\_0所在的block处理完毕之后，才会开始处理Tx\_1所在的block，也因此当处理到Tx\_1开始验证时，此时从state数据库中取出的版本号必定是Tx\_0提交过后的新的版本号v2，也因此Tx\_1（版本号为v1）会被判定为无效交易。（2）Tx\_0与Tx\_1在同一个block中，则Tx\_0所在的位置（即下标值）小于Tx\_1所在的位置。这种情况会先验证Tx\_0交易，如果Tx\_0有效，则会把Tx\_0的写集放入到一个updates升级包中（但此时这个升级包并未提交到state数据库中），接着会继续验证Tx\_1，会发现updates中存在与Tx\_1读集相同的key，也即说明在Tx\_1之前的Tx\_0已经在试图改变Tx\_1所使用到的数据，而Tx\_0又是有效且早于Tx\_1的，因此要判定Tx\_1为无效。

**验证过程**

验证过程还是很值得看的，因为验证的过程不仅可以了解数据的包装结构，还能很清楚的知道什么情况下是对的，什么情况下是错的，都需要哪些对象参与，等等，这些都对于整个业务逻辑和流程的理解都很有帮助。其实committer将block提交至账本处时已经使用自身的验证器执行了一轮验证（参看core/committer/txvalidator/validator.go中的Validate(block)），并把非法的无效交易进行记录，不过该轮验证主要集中在验证block的身份、签名、权限等内容。而这里所讲的是账本中对数据的mvcc等验证，以确保有效交易能正确存储到state数据库中，所以验证过程以validator/statebasedval/state\_based\_validator.go中的ValidateAndPrepareBatch(block,domvcc)为起点（domvcc默认为true），从函数名即可理解这个函数分为上下两部分：Validate和PrepareBatch，即验证和准备批量升级包updates。

1. updates := statedb.NewUpdateBatch()，准备updates用于存放批量升级包，这个updates就是上文提到的升级包，会以参数的形式传递到每个更具体的验证函数中去，作为乙方的一部分参与验证 -> txsFilter := util.TxValidationFlags(...)，从block中抽取出block元数据的BlockMetadataIndex\_TRANSACTIONS\_FILTER位的数据txsFilter，该数据标记了block中的每笔交易有效性（这里的有效性数据即是committer处验证的结果），若是无效交易，这里的验证将直接跳过。交易的有效值为在protos/peer/transaction.pb.go中定义的如TxValidationCode\_VALID的系列常量值。
2. for txIndex, envBytes := range block.Data.Data { ... }，依次抽取block中的每个Envelope，开始验证 -> 一系列解压抽取Envelope的工作后，if txType != common.HeaderType\_ENDORSER\_TRANSACTION，判定类型，若不是正常的交易数据，则直接跳过，state只存储交易数据 -> 最重要的一步，txRWSet, txResult, err := v.validateEndorserTX(env...)，验证交易，返回交易的读写集和验证结果 -> txsFilter.SetFlag(txIndex, txResult)，在txsFilter中存储当前交易的验证结果，if txRWSet != nil { ... }，如果读写集不为空，则说明当前交易有效，则committingTxHeight := version.NewHeight(...)生成**提交版本号**（写值自身是不带版本号的，而写值的向state数据提交的版本号就是在这生成的），addWriteSetToBatch(...)将有效交易的**写集和对应的提交版本号**放入批量升级包updates -> for循环结束，block.Metadata.Metadata[...\_TRANSACTIONS\_FILTER] = txsFilter替换block元数据的交易有效性值。
3. 第2步中的txRWSet, txResult, err := v.validateEndorserTX(env)处，就是一个分发任务依次验证的过程。自己txRWSet.FromProtoBytes(respPayload.Results)把需要验证的读写集抽取出来后，就把任务交给了v.validateTx(txRWSet, updates)，validateTx又分拆把验证读集的任务交给validateReadSet，把验证范围读集的任务交给validateRangeQueries。这些函数均是“人如其名”的存在。

**验证读集**

validateReadSet验证读集的方法就是循环的调用validateKVRead，一一验证每一个读值。在validateKVRead中，（1）if updates.Exists(ns, kvRead.Key)即是查看升级包updates中是否有与读值相同的key，若存在，则直接判定交易无效。（2）versionedValue, err := v.db.GetState(ns, kvRead.Key) -> committedVersion = versionedValue.Version，当下现从state数据库中查出读值key在state中的版本号committedVersion -> if !version.AreSame(...)，比较两个版本号是否一致，若不一致，则判定当前交易无效，直接返回false，否则返回true。

**验证范围读集**

validateRangeQueries验证范围读集的方法就是循环调用validateRangeQuery，一一验证每个范围读值。范围读值的验证由于是一系列的值，这一系列的值既要与state中现存的版本号比较，也要与updates中已存在的key比较，还牵扯到可能范围读值中的一个key在updates和state中同时存在（这是这个key就存在两个版本号了）而选哪一个版本号来和范围读值中这个key的版本比较的问题，所以就稍显麻烦。为了描述简洁，下文值提及的key，既代表key本身，也代表key对应的读值。

这里涉及到validator/statebasedval/下的两个工具：

* range\_query\_validator.go中的rangeQueryValidator，“人如其名”，范围查询验证器，有init接口用于初始化，validate接口用于验证。实现为两个版本，一个是用来验证原始范围读值的rangeQueryResultsValidator，一个是用来验证范围读值的归并哈希值的rangeQueryHashValidator，由于多数情况下范围读值中存储的是读值集合的归并哈希值，因此这里只讲后者。
* combined\_iterator.go中的combinedIterator，联合迭代器。这个联合迭代器是供rangeQueryValidator管理使用的。针对上述验证的麻烦之处，联合迭代器联合的就是以updates为数据源生成的**迭代器A**（参看statedb/statedb.go中的nsIterator）和以state数据库为数据源生成的**迭代器B**（参看statedb/stateleveldb/stateleveldb.go中的kvScanner）。当前从A中获取的值存储在updatesItem中，当前从B中获取的值存储在dbItem（类型为VersionedKV，包含key，值，版本号）。当执行联合迭代器的Next()获取一个查询的版本号时，联合迭代器会调用compareKeys()比较updatesItem和dbItem中的key：updatesItem的key更小，返回updatesItem，A前进一步；dbItem的key更小，返回dbItem，B前进一步；当两个key相等时返回updatesItem，A，B都前进一步。这里的前进一步，指的是迭代器执行一下Next()接口迭代到下一个值。

回到validateRangeQuery中。（1）includeEndKey := !rangeQueryInfo.ItrExhausted，记录下此范围读值是否包含endkey对应的读值。因为范围读值默认是不包含endkey对应的读值的，但是当这个范围读值没有被模拟器读尽时，比如key2-key10只读了前3个就返回了，此时rangeQueryInfo.ItrExhausted值为false时，则说明此时实际的endkey是key4而非key10，且此时范围读值应该包含key4的读值。相反，若ItrExhausted值为true时，则说明模拟器读尽了key2-key10间的8个key，由于范围读值默认的不包含endkey，所以此时范围读值中不包含key10对应的读值。（2）combinedItr, err := newCombinedIterator(...)，根据updates，startkey，endkey，includeEndKey，创建一个联合迭代器。（3）if rangeQueryInfo.GetReadsMerkleHashes() != nil成立（默认返回的范围读值中是归并后的哈希值） -> validator = &rangeQueryHashValidator{} -> validator.init(rangeQueryInfo, combinedItr)，根据范围读值rangeQueryInfo和生成的联合迭代器combinedItr，同时内建了一个上文提及的RangeQueryResultsHelper工具，创建一个用于验证范围读值的归并哈希值的验证器。（4）validator.validate()，使用验证器验证范围读值。粗略的验证过程就是使用RangeQueryResultsHelper工具重新一步步构建出一个与rangeQueryInfo范围相同的默克尔树并不断进行归并哈希的操作，将得到的归并哈希值与rangeQueryInfo携带的哈希值进行比较。

**举个例子**，当前交易Tx\_1验证的一个范围读值（startkey为2，endkey为6）中有Tx\_1\_key2，Tx\_1\_key4，Tx\_1\_key5共3个key的读值（归并形成的哈希值为Tx\_1\_hash2-6），A里有A\_key2的一个准备更新的值，B中在key2-key6这个范围内中B\_key2的值被稍早点儿的交易Tx\_0已改动过（A\_key2的版本一定比B\_key2更高，因为A是updates中的，又由于B\_key2是被Tx\_0改动的，因此B\_key2的版本号一定比Tx\_1\_key2的更高），B\_key3为Tx\_0新加入的，其余与Tx\_1相同，因此有B\_key2，B\_key3，B\_key4，B\_key5共4个key。则两种类型的范围验证器验证过程如下（参看validator/statebasedval/range\_query\_validator.go中两类验证器的validate()）：

* rangeQueryResultsValidator的验证过程：（1）验证Tx\_1\_key2时，通过比较，发现A和B中都存在key2，将会把A\_key2的值返回，A、B都向前前进一步并更新updatesItem（变为nil）和dbItem（变为B\_key3对应的值）。因为B\_key2肯定是之前的交易提交的旧数据，而A\_key2则是当前准备向state提交的更新数据，所以这里返回updatesItem。当比较Tx\_1\_key2与A\_key2时，会发现两者版本不同，此时将判定交易Tx\_1无效，对Tx\_1的验证也将结束。（2）假如能继续，下一个验证的是Tx\_1\_key4时，此时updatesItem为nil，dbItem为B\_key3对应的VersionedKV，则返回的是dbItem，此时比较Tx\_1\_key4与B\_key3时发现一个是key4，一个是key3，一个是key4，此时也将判定交易Tx\_1无效。因为无论这个key3是从updates冒出来的还是从state数据库中冒出来的，都说明key2-key5这个范围内出现了新的数据，而Tx\_1中的这个范围的读值将变得无效。因为既然这个范围读值被读出来，就有被当作一个整体使用的可能，比如key2-key5之间全部都是某一公司的财产，此时要读出该公司所有的财产总和并计算出其总和的20%让渡给B公司，模拟交易时得到的是有100万，而到验证前又有一笔10万的入账。若此时还依然以100万的20%进行让渡显然是不合适的。另外其实像突然冒出的key3这种情况，比较专业的说法是数据库方面的幻读（phantom read）现象，读者可以自行搜索，更全的例子，也可参看state\_based\_validator\_test.go中的TestPhantomValidation函数。
* rangeQueryHashValidator的验证过程：设Tx\_1的范围读值的maxDegree为2，则形成的归并哈希后的MaxLevel为2（参看上文归并默克尔树归并哈希的操作）。inMerkle是例子中Tx\_1的一个范围读值中的归并数据（包含Tx\_1\_hash2-6，maxDegree，MaxLevel等信息）。在for { ... }循环中，（1）result, err = itr.Next()，使用联合迭代器获取一个读值result，此时result是A\_key2的值，之后依次会是B\_key3，B\_key4 -> v.resultsHelper.AddResult(...)，使用RangeQueryResultsHelper工具将result依次传化为适当形式后添加到默克尔树中 -> merkle := v.resultsHelper.GetMerkleSummary()，获取新建的默克尔树中的归并数据merkle -> if merkle.MaxLevel < inMerkle.MaxLevel，当默克尔树添加入A\_key2，B\_key3时，因为没有触发归并哈希值操作，所以merkle.MaxLevel依然是默认的1，因此会在这里直接continue。（2）当加入B\_key4时，此时叶节点已经有3个值，大于maxDegree，则会触发归并哈希值的操作生成一个哈希值hash2-4，merkle.MaxLevel变为2，因此程序会继续向下执行 -> if lastMatchedIndex == len(merkle.MaxLevelHashes)-1是-1==0的比较，不成立 -> lastMatchedIndex++后变为0 -> !bytes.Equal(...)，此时merkle与inMerkle的MaxLevelHashes[0]出应该都是空的，因此相等，程序继续循环。（3）result, err = itr.Next()，此次获取的result是B\_key5 -> 过程如上，添加B\_key5并不会触发merkle新的归并哈希值操作，因此之后的几个if分支均不会进入而返回，程序会进入下一次循环 -> result, err = itr.Next()，此次获取的result为nil，因此进入if result == nil {...}分支 -> \_, merkle, err = v.resultsHelper.Done()执行最后一次归并哈希值的操作生成归并数据，此归并数据其实是包含了之前加入的A\_key2，B\_key3，B\_key4，B\_key5四个值的归并哈希值，而Tx\_1\_hash2-6是Tx\_1\_key2，Tx\_1\_key4，Tx\_1\_key5三个值的归并哈希值，如此equals := inMerkle.Equal(merkle)比较的话，自然equals为false，因此return equals, nil，判定Tx\_1交易无效。

回归到state\_based\_validator.go中的ValidateAndPrepareBatch(...)，当validateEndorserTX调用的v.validateTx(txRWSet, updates)返回为peer.TxValidationCode\_VALID，则说明经过上述验证读集、验证范围读集的过程，证明当前交易有效且可以写入state数据库，因此会把交易的读写集以不为nil的形式返回，至此ValidateAndPrepareBatch的Validate部分的工作算是完成。接着，在ValidateAndPrepareBatch(...)中会进入if txRWSet != nil {...}分支，完成PrepareBatch部分的工作。最后，将准备的升级包updates返回。该updates返回后会被存储给LockBasedTxMgr的batch成员，供下一步l.txtmgmt.Commit()向state数据提交这些升级数据的时候使用（参看peer结点中使用VersionedDB的过程第8点）。在l.txtmgmt.Commit()中（txmgr/lockbasedtxmgr/lockbased\_txmgr.go），txmgr.db.ApplyUpdates(...)即是state数据库使用升级包数据来真正升级状态的。这里传入了两个参数，一个是txmgr.batch，即升级数据包，另一个是以blockID+block最后一个交易的下标序号组成的版本号s\_version。

### state数据库

state数据库就是一个正常的leveldb数据库，实现为statedb/stateleveldb/stateleveldb.go中的versionedDB。提供基本的打开，关闭，查询，写入功能，此外还提供支持范围查询的leveldb数据库的迭代器，多值查询的额外功能。leveldb版本的state数据不支持富查询，即ExecuteQuery(...)接口直接返回错误。

**查**

GetState，GetStateMultipleKeys两个接口分别提供单值查询，多值查询。GetStateRangeScanIterator接口提供范围查询的迭代器。这些较为简单，不再赘述。

**写**

state数据库为LockBasedTxMgr配合处理block，因此只支持批量升级数据，即ApplyUpdates(batch,height)接口的功能。batch即为上文验证过程中准备的有效交易的升级数据包，height则为一个版本号（即上文的s\_version），该版本号不用于具体的某个状态，而是作为state数据库的一个叫做**保存点**的键值对的值（idStore，BlockStore中都有类似的保存点，也可以叫检查点）。具体的写入操作也是常规的leveldb数据库的批量写入操作，保存点会在每一批升级数据的最后加入，key为savePointKey，value为height。也就是说，当能从state中获取的保存点的height值，且height中的BlockNum为N，则说明当前state数据库已经完整保存了序号为N的block中的有效交易。

### 重启恢复

同idStore和BlockStore存储一样，VersionedDB同样存在宕机重启后残缺数据的清理和恢复问题。不同于idStore和BlockStore的在自身对象建立起来后即进行“自我修复”，VersionedDB需要在BlockStore自我修复完毕之后，**使用恢复后的BlockStore提供的数据**由kvLedger手动的恢复。为何使用BlockStore提供的数据来恢复呢？这点需要参看账本处理block的前后顺序：参看core/ledger/kvledger/kv\_ledger.go的Commit(block)，是先执行l.blockStore.AddBlock(block)把block提交至BlockStore中后，然后执行l.txtmgmt.Commit()将有效交易数据提交至state数据库，最后向HistoryDB账本提交（**这个提交顺序很重要**，对恢复各个账本时的逻辑有很大影响）。BlockStore亦用于自恢复能力，在newKVLedger(...)之时，传入的BlockStore对象其实是已经完成了自我修复。因此当宕机发生时，BlockStore中的数据可能比VersionedDB更新的内容。在newKVLedger(...)中，随后执行了l.recoverDBs()，来恢复VersionedDB和HistoryDB，HistoryDB在下文详述。

在recoverDBs()中，lastAvailableBlockNum是从BlockStore中获取的最新的有效的blockID，recoverables存放预计需要恢复的账本对象，recoverers存放真正需要恢复的账本对象：

1. info, \_ := l.blockStore.GetBlockchainInfo()，lastAvailableBlockNum := info.Height - 1，使用已恢复的BlockStore获取当前已经写入BlockStore的有效的blockID。
2. recoverables := []recoverable{...}，可能需要恢复的有VersionedDB和HistoryDB两个数据库。
3. for \_, recoverable := range recoverables {...}，依次调用recoverable.ShouldRecover(lastAvailableBlockNum)，使用最新有效的lastAvailableBlockNum，检测账本是否需要恢复。这里只看VersionedDB（HistoryDB的检测方式与之相同）。在txmgr/lockbasedtxmgr/lockbased\_txmgr.go的ShouldRecover中：savepoint, err := txmgr.GetLastSavepoint()获取了state数据库中的保存点 -> 然后将savepoint.BlockNum与lastAvailableBlockNum进行对比 -> 如果相等，则state数据库不需要恢复，如果不同（savepoint.BlockNum一定比lastAvailableBlockNum小），则说明state中未完整存储序号为lastAvailableBlockNum的block的所有有效交易，需要进行恢复 -> 当需要恢复时，将返回ture，savepoint.BlockNum + 1。true标识state需要恢复操作，savepoint.BlockNum + 1则标识需要从哪一块block进行恢复，即**恢复起点**。ShouldRecover结束后，返回的数据付给了recoverFlag，firstBlockNum，如果recoverFlag为true，则会将VersionedDB账本放入recoverers。这里假设VersionedDB确实需要恢复。
4. 接着，也就是for循环之后，有一系列的if分支。if len(recoverers) == 0，则说明没有确实需要恢复的账本，直接返回。if len(recoverers) == 1，则说明只有一个需要恢复的账本，此时只可能是HistoryDB，因为若VersionedDB需要恢复，则HistoryDB必也需要恢复，且VersionedDB的恢复起点一定 >= HistoryDB的恢复起点。因此这里只看两个账本都需要恢复的情况，设VersionedDB需要从10处开始恢复，HistoryDB需要从6处开始恢复，lastAvailableBlockNum值为15，则：（1）if [0]... > [1]...将成立，因此执行[0], [1] = [1], [0]置换，将lagger放入recoverers的0的位置，lagger的意思就是懒散的、落伍的人的意思，这里指恢复起点值更小的账本，且肯定是HistoryDB，而把恢复起点值更大的VersionedDB放到后边。（2）if [0]... != [1]...，置换后，0处的6肯定 != 1处的10，因此执行l.recommitLostBlocks(...)，单独向lagger提交6-9之间的block数据进行恢复。（3）l.recommitLostBlocks([1].., lastAvailableBlockNum,[0],[1])，接着，也是最后，向lagger和VersionedDB一同提交10-15之间的block数据进行恢复。当两个账本的恢复起点相等时，比如都为8，则（1）和（2）处的if分支都不会进入而直接进行至此步，一起向两个账本提交8-15之间的block数据进行恢复。这里需要说明一下当两个账本的恢复起点不一致时为什么要进行（1）和（2）的操作，又是换位置又是分段恢复的：因为block是存储在磁盘文件blockfile中的，因此换位置和分段恢复的操作，可以使得恢复过程中BlockStore读取blockfile中的block数据时**顺序读取一次**，这样效率更高。
5. 最后单独看一下recommitLostBlocks(...)，传入的参数分别为恢复起点firstBlockNum、恢复终点lastBlockNum、需要恢复的账本recoverables。for循环中，block, err = l.GetBlockByNumber(blockNumber)使用BlockStore从blockfile中获取指定序列号的block，然后调用账本的r.CommitLostBlock(block)，向账本提交block，进行恢复。这里只看VersionedDB。在txmgr/lockbasedtxmgr/lockbased\_txmgr.go的CommitLostBlock(block)中，和正常的向VersionedDB账本提交交易数据的过程如出一辙：txmgr.ValidateAndPrepare(block, false)准备升级包数据，注意这里第二个参数给的是false，即不再做mvcc验证，因为这里是恢复，所提交的block是从BlockStore中获取的，也肯定是之前已经验证过才会放入BlockStore的，所以这里不需要再重复进行验证 -> txmgr.Commit()将升级包数据真正提交到state数据库，完成VersionedDB的恢复工作。

## HistoryDB

HistoryDB也是一个标准的以leveldb数据库为依托的账本，实现在core/ledger/kvledger/history/historydb下（下文以此目录为基准）。比较特殊的地方是这个账本只存储block中有效交易相关的key，而不存储value（由于leveldb存储的键值对不允许nil，因此这里实际上value是有值的，只不过所有的值均为[]byte{}）。另外提供一个历史查询器HistoryDBQueryExecutor在交易的时候使用。其余的操作，如开闭，读写操作均与正常的leveldb类型账本并无二致，不再赘述。

HistoryDB是可配置是否使能的，在core/ledger/ledgerconfig/ledger\_config.go的IsHistoryDBEnabled()接口可以获取当前是否开启使用了HistoryDB。原始的配置则在core.yaml中的enableHistoryDatabase项，默认为true，即开启使用HistoryDB。

既然HistoryDB实际只存储key，那么也就是说，key自身既携带了索引信息又携带了我们需要的值信息，key其实就是HistoryDB所要存储等价于value的对象并供外界检索。这个key是一个组合key，对看historyleveldb/historyleveldb.go中的Commit(block)接口，当向HistoryDB提交一个block时，会筛选出block中的有效交易，并把这些交易的写集中的每个写值读取出来，以**命名空间ns + compositeKeySep + 写值key + compositeKeySep + block序列号 + 交易ID**的组合形式形成一个组合键compositeHistoryKey，然后dbBatch.Put(compositeHistoryKey, emptyValue)，将compositeHistoryKey与空值emptyValue作为一个键值对写入批量升级包dbBatch中。当block中所有有效交易均遍历完毕后，dbBatch.Put(savePointKey, height.ToBytes())以保存点封底，最后historyDB.db.WriteBatch(dbBatch, false)向HistoryDB提交数据。

组合键compositeHistoryKey，其中的compositeKeySep当作分隔符理解即可。前半部分**命名空间ns + compositeKeySep + 写值key + compositeKeySep**即为用于索引的信息，后半部分**block序列号 + 交易ID**即为值信息。

**历史查询器**

对HistoryDB的检索主要通过HistoryDB提供的一个HistoryDBQueryExecutor对象来实现。HistoryQueryExecutor在historyleveldb/historyleveldb\_query\_executer.go中实现，只提供GetHistoryForKey(...)一个接口，该接口根据提供的命名空间和写值key，返回一个迭代器historyScanner（内部封装了levedb数据库的Iterator）。是迭代器必定涉及起点和止点，historyScanner的起点是**命名空间ns + compositeKeySep + 写值key + compositeKeySep**的组合键，止点是**命名空间ns + compositeKeySep + 写值key + compositeKeySep + 0xff**的组合键。对比起点和止点，止点多了一个0xff，相当于一个字符的极限值，也因此这个范围查询的是所有以**命名空间ns + compositeKeySep + 写值key + compositeKeySep**为开头的key值。比如起点是[]byte("A")，止点是append([]byte("A"),[]byte{0xff}...)，则这个范围查询的是所有以字符A开头的key。又因为有HistoryDB存储的格式为前提，假设这里的ns为“chaincode\_example02”，写值key为“A账户”，则这个起止范围相当于在查询**所有chaincode\_example02链码上改动过A账户的“ blockID+有效交易ID ”的信息**。而有了**blockID+有效交易ID**这两个信息，自然可以通过BlockStore定位查询出原交易的所有信息，如改动时间，改动值，是否是删除操作等。

参看historyleveldb\_query\_executer.go中historyScanner的Next()接口：

1. if !scanner.dbItr.Next()，因为HistoryDB不存储value，因此这里leveldb的迭代器dbItr的Next()操作不为获取value，而只是让迭代器向前走一步，同时作是否还有下一个值的判断。
2. historyKey := scanner.dbItr.Key()，获取leveldb迭代器当前值的key，这也是我们实质想获取的内容，这个key就是由**命名空间ns + compositeKeySep + 写值key + compositeKeySep + block序列号 + 交易ID**组成的组合键。
3. blockNum, bytesConsumed := util...，tranNum, \_ := util...，从组合键中分别分解出其中携带的blockID和交易ID。
4. tranEnvelope, err := scanner.blockStore...，使用BlockStore，根据blockID和交易ID，获取原始的交易信息。
5. queryResult, err := getKeyModificationFromTran(...)，根据获取的原始交易信息，进行整理，返回当次Next()的单个查询结果，该结果里面包含改动时间，改动值，是否是删除操作等信息。

**使用**

HistoryDB账本被kvLedger使用过程和VersionedDB在过程上就是一先一后的区别，不用过多赘述。被chaincode使用主要是通过出现在交易中的HistoryDBQueryExecutor进行查询，而查询的过程与VersionedDB的交易模拟器的范围查询过程颇为类似（至于HistoryDBQueryExecutor为何会出现在交易中，请参看core/endorser/endorser.go的ProcessProposal(...)中的ctx = context.WithValue(...)处），以map.go为例，：

1. peer chaincode invoke...，在peer结点执行map的Invoke命令，中间过程省略，直接定位到下一步。
2. 在map.go的Invoke(stub)的case "history":分支中，keysIter, err := stub.GetHistoryForKey(key)，调用ChaincodeStubInterface的此接口，最终获取一个迭代器，该迭代器以HistoryDBQueryExecutor获取的范围数据为数据源，返回查询结果。
3. 第1步调用后，会在ShimHandler和ServerHandler间辗转，然后在core/chaincode/handler.go中调用handleGetHistoryForKey(...)，在这个函数中，historyIter, err := txContext.historyQueryExecutor.GetHistoryForKey(chaincodeID, getHistoryForKey.Key)即是使用HistoryDBQueryExecutor获取一个historyScanner，然后在getQueryResponse(...)中依次调用historyScanner的Next()获取查询结果（即上文第5步查询到的内容）。查询结果最终放入了ChaincodeMessage的Payload返回。
4. 携带查询结果的ChaincodeMessage返回至core/chaincode/shim/chaincode.go中的GetHistoryForKey(...)中，也即第1步所调用处，然后把结果集放在了CommonIterator迭代器中，上面再套上HistoryQueryIterator返回给chaincode，亦也就是第1步获取的keysIter。
5. 获取keysIter后，就可以在for keysIter.HasNext() { response, iterErr := keysIter.Next()... }进行使用，以完成chaincode在该功能上自身想完成的任务。

因此，从使用上讲，HistoryDB算是一个辅助性的数据库，辅助其他数据库，辅助交易的进行。另外，拓展一下的话，HistoryDB的组合键的组合形式可以针对自身业务的需求进行设计，这样也可实现类似couchDB数据库那样的富查询。

在此总结一下吧。账本本身算是一个存储系统，或基于文件，或基于数据库，或两者相互配合，最终是用于存储交易的数据的，相当于fabric的“硬盘”。关键在于理解每个账本存储了什么，如何存储的，如何恢复的，向外界提供哪些功能。若读者读至此处，建议在回头看一遍《fabric源码解析24》的开篇之处。文中若有纰漏之处，还请留言指正。

# fabric源码解析26——验证体系

## 概述

这篇主题文章暂时没有完备，遂暂时跳过

# fabric源码解析27——Channel

## 概述

| **动作** | **释义** |
| --- | --- |
| create | 在orderer结点内部创建一个channel。如，peer channel create -o orderer.example.com:7050 -c mychannel -f ./channel.tx |
| join | 加入一个channel。如，peer channel join -b mychannel.block |
| update | 升级channel的某一组织的配置。如，peer channel update -o orderer.example.com:7050 -c mychannel -f ./Org1MSPanchors.tx |
| list | 列出当前系统中已经存在的所有的由peer结点创建的channel。 |
| fetch | 获取channel中的newest，oldest块数据或当前最新的配置数据。如，peer channel fetch config config\_block.pb -o orderer.example.com:7050 -c mychannel |

Channel在fabric中是一个相当重要的概念，可译作频道。对于channel的理解，不妨想象一下电视节目的频道和“我和你不在一个频道”这句话。Channel本身存在于orderer结点内部，但需要通过peer结点使用peer channel ...命令进行维护。一个peer结点要想与另一个peer结点发生交易，最基本的前提就是两个结点必须同时处在同一个Channel中，block账本与channel也是一对一的关系，即一个channel一个账本。Channel的基本动作如下：

*表27-1（本文只讲解create，join，update三个动作）*

chaincode分为SCC和ACC，这里Channel也分为system channel和application channel。system channel是随着orderer结点运行之时根据genesis.block创建的，而通过peer channel ...维护的channel，均为application channel。对application channel发起维护命令的peer结点必须是所提交的配置文件（channel.tx，mychannel.block，Org1MSPanchors.tx）中所配置的组织中的一员，如peer0执行create，则peer0必须是channel.tx中所配置的组织中的一员，而channel.tx对应生成的block块数据，就相当于application channel中的genesis.block。这里所说的属于组织的一员，其实质是指该peer结点要持有该组织所颁发的证书。

## 目录

* common   
  + config - 配置数据处理
  + configtx - 配置交易处理
* orderer
* peer   
  + channel - peer channel命令源码

## 配置文件

在表27-1的释义中，create/join/update三个动作都使用到了配置文件（数据）：

* channel.tx - application channel的创建配置文件，供peer channel create使用，由configtxgen工具根据指定的profile从configtx.yaml中读取配置数据。这里的profile指的是configtx.yaml中Profiles项下定义的某一个配置项。该文件主要规定了application channel中包含哪些组织，最终会被用作channel的配置原型，通过填补，作为channel账本的genesis块。例子：configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel.tx -channelID mychannel，指定了要生成的application channel的ID为mychannel，并从configtx.yaml的Profiles中选取TwoOrgsChannel项作为channel的具体配置，并把生成配置数据导入到./channel.tx文件中。channel.tx中存储的是二进制格式的Envelope，这点可以参看common/configtx/tool/configtxgen/main.go中的doOutputChannelCreateTx。
* mychannel.block - application channel的genesis.block，供join使用，由peer channel create动作生成。上文中说channel.tx只是配置原型，在create的过程中，会根据system channel中的配置，对channel.tx中的数据进行详细的填补，如填补orderer的配置（毕竟application channel是存在于orderer结点中的，不能对orderer的“规矩”一无所知），填补application channel所包含的组织的详细配置。填补了这些东西，最终生成一个block，作为application channel的genesis块被保存入账本。要想join一个application channel，就需要先获取这个channel的genesis块。
* Org1MSPanchors.tx - application channel的某组织升级配置文件，供peer channel update使用，由configtxgen工具根据指定的组织ID从configtx.yaml中指定的profile项中生成。channel的配置中，基本项目之一就是频道所包含的组织了，而update命令就是升级channel配置中的某个组织的配置。例子：configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./Org1MSPanchors.tx -channelID mychannel -asOrg Org1MSP，指定从configtx.yaml的Profiles下的TwoOrgsChannel项中获取组织ID为Org1MSP的组织的配置数据，用于升级频道ID为mychannel的application channel，并把获取生成的配置数据导入到./Org1MSPanchors.tx文件中。同样，Org1MSPanchors.tx存储的是二进制的Envelope，可参看common/configtx/tool/configtxgen/main.go中的doOutputAnchorPeersUpdate。

## 命令

| **create/update** |
| --- |

create和update的操作从字面上讲一个是创建，一个是升级，在fabric都被处理成升级操作，create作为从无到有的升级操作，update作为从旧到新的升级操作。在peer/channel/create.go和update.go中：

1. 如果**把create的执行过程分为三步**，那么update只是create的第一步而已。在create.go的createCmd -> create -> executeCreate(cf)中，过程可以简单的分为：（1）sendCreateChainTransaction(cf)，向orderer发送创建application channel的配置数据。（2）getGenesisBlock(cf)，多次尝试从orderer获取生成的application channel的genesis块，即此时application channel已经建立，上一步发送的配置数据经过填补已经作为第一块block被保存到频道的账本中。（3）ioutil.WriteFile(file, b, 0644)，将获取的genesis块写入的文件中，供peer channel join命令使用。在update.go的update(...)中，完成的任务与executeCreate中的sendCreateChainTransaction(cf)一致，只不过发送的配置数据是用于更新application channel中的某个组织的配置数据。以下将主要以create的执行过程为例，辅以update的不同之处进行讲述。
2. **开始第一步**。在sendCreateChainTransaction中，（1）根据命令行指定的channelTxFile，也即channel.tx文件路径，chCrtEnv, err = createChannelFromConfigTx(channelTxFile)读取配置数据到chCrtEnv中。（2）sanityCheckAndSignConfigTx(chCrtEnv)，对看函数名，理性的检查并对chCrtEnv进行签名，sanity-头脑清晰，心智健全的意思，所以这里的意思就是只要你没疯，就在将配置数据发往orderer之前检查一下。但同时又说明，这样的检查不是非要不可，因为channel.tx本身是configtxgen工具生成的，只要工具规规矩矩不搞鬼，那这里不检查也可以。后边的步骤中还会有多次这样的sanity类型的检查。签名的话就是正常的签名，哪个peer结点发起的peer channel create，在chCrtEnv中留下签名即可。（3）broadcastClient.Send(chCrtEnv)，向orderer结点发送chCrtEnv。
3. 在orderer/server.go的Broadcast(...)中的s.bh.Handle(srv) -> orderer/common/broadcast/broadcast.go的Handle(...)中的msg, err := srv.Recv()，orderer接收到第2点中peer发来的原始的“配置信”msg，接着开始从msg中抽取数据。因为是“配置信”，因此会进入if chdr.Type == int32(cb.HeaderType\_CONFIG\_UPDATE)分支，在此分支中，msg, err = bh.sm.Process(msg)对原始的“配置信”msg进行了重新整理：过滤重复的配置，若是peer channel create的话，过滤后的配置还会被装进一个以system channel的ID为外衣的“配置信”中，形成新的“配置信”msg。
4. bh.sm.Process(msg)最终调用的是orderer/configupdate/configupdate.go的Process(...)，在此函数中，support, ok := p.manager.GetChain(channelID)即获取了所需要的资源，也验证了“配置信”所对应的频道是否存在，若存在，则说明此“配置信”是用于update已有频道的配置的，则进入if ok分支执行p.existingChannelConfig(...)，否则，则说明此“配置信”是用于create新的频道的，则跳过if ok分支去执行p.newChannelConfig(...)。这里p.existingChannelConfig(...)依然算是雷同于p.newChannelConfig(...)中的一部分。
5. 在p.newChannelConfig(...)中，**（1）** ctxm, err := p.manager.NewChannelConfig(envConfigUpdate)这一步比较有意思，在orderer/multichain/manager.go的NewChannelConfig，相当于费了九牛二虎之力验证原配置数据，并填补了一些system channel的配置，又到common下的config和configtx遛了一大圈生成了一个内含application channel新的配置的configManager（这个过程相当复杂费劲），但是这个configManager仅仅停留在当前函数中且并没有返回供调用者继续使用，而且envConfigUpdate虽然是以指针的形式传进去的，但是在NewChannelConfig并没有改变envConfigUpdate中的值。而下文newChannelConfigEnv, err := ctxm.ProposeConfigUpdate(envConfigUpdate)又直接使用原envConfigUpdate来生成application channel的频道“配置信”。如此种种会感觉NewChannelConfig做了很多无用功，但这里就属于上文提及的类似于sanity类型的验证，因为后续步骤在真正创建application channel时也会经历NewChannelConfig和创建频道所使用的configManager，所以这里相当于事前先创建一下，若有问题趁早发现趁早返回。**（2）** newChannelConfigEnv, err := ctxm.ProposeConfigUpdate(envConfigUpdate)，根据原有的配置数据，利用configManager的ProposeConfigUpdate功能，通过对比“配置信”中读集和写集，将已有且版本相同的配置项过滤掉，生成要创建的application channel的频道配置数据newChannelConfigEnv。这里的读集和写集可以对看生成channel.tx的函数doOutputChannelCreateTx。peer channel update所调用的existingChannelConfig(...)中执行的是support.ProposeConfigUpdate(...)与这里的ctxm.ProposeConfigUpdate(...)实际是一样的，都是configManager的ProposeConfigUpdate，只不过existingChannelConfig(...)中使用的是已存在的application channel的configManager（包含在chainSupport->ledgerResources中）的ProposeConfigUpdate。这个已存在的configManager正好也证明了（1）所述的后文在创建频道时还会创建频道所使用的configManager。**（3）** newChannelEnvConfig, err := utils.CreateSignedEnvelope(...)，对新生成的频道的“配置信”进行签名，类型为HeaderType\_CONFIG。peer channel update所调用的existingChannelConfig(...)同样执行了这一步，且就此返回该“配置信”。**（4）** p.proposeNewChannelToSystemChannel(...)，将签名过的“配置信”装到一个以system channel为频道ID的“配置信”，且类型变为HeaderType\_ORDERER\_TRANSACTION，然后返回新的“配置信”。这一点是peer channel create独有的，因为创建application channel要使用system channel的chainSupport对象。
6. 重回orderer/configupdate/configupdate.go的Process(...)中，继续，对原始的“配置信”处理之后，support, ok := bh.sm.GetChain(chdr.ChannelId)，根据频道ID获取chainSupport对象，peer channel create获取的是system channel的chainSupport，peer channel update获取的是对应application channel的chainSupport。
7. \_, filterErr := support.Filters().Apply(msg)，由第6点获取的chainSupport获取频道的过滤器集合并对“配置信”进行Apply()过滤。只讲peer channel create，获取的是system channel的过滤器集，具体为orderer启动时，在orderer/multichain/chainsupport.go的createSystemChainFilters(...)生成。依次进行非空、大小、签名的检查后，最后调用systemChainFilter的Apply()，即orderer/multichain/systemchain.go的Apply(env)。在systemChainFilter的Apply(env)中，一系列抽取“配置信”并检查之后，**（1）** scf.authorizeAndInspect(configTx)对“配置信”整理和检查。这里需要明确一点，从peer发送到orderer接收至此，“配置信”均只有配置项而没有具体的配置值，是不完整的。而直到这一步才对“配置信”进行配置值的填充。在authorizeAndInspect(...)中，我们如愿的看到了NewChannelConfig，configtx.NewManagerImpl(...)的身影，即根据system channel的配置填充“配置信”对应配置项的值和新建application channel所使用的configManager。这也同时证明了第5点所说的有关sanity类型检验的说法所言非虚。**（2）** return filter.Accept, &systemChainCommitter{...}，返回Accept动作和包含了完整的“配置信”的systemChainCommitter。这里注意，这一步主要目的有两个：验证和填充“配置信”，system channel的过滤器集合Apply()后返回的执行器集合被\_省略，是因为后文会再次生成，在block写入账本之前调用执行器。
8. support.Enqueue(msg)，把“配置信”作为一条消息发送给kafka（或solo），具体是由orderer/kafka/chain.go中chainImpl的Enqueue(...)发给kafka的（又由于support是system channel的chainSupport，因此这里的chainImpl是system channel对应的对象，其成员support也是system channel的chainSupport），经过kafka暗盒的排序处理，配置信被包裹在一条KafkaMessage\_Regular消息，在orderer/kafka/chain.go的processMessagesToBlocks()中被接收，并进入case \*ab.KafkaMessage\_Regular:分支，交由processRegular(...)处理。在processRegular(...)中，将“配置信”抽取出来后：**（1）** batches, committers, ok := support.BlockCutter().Ordered(env)，由“配置信”生成block，由于是“配置信”，因此会单独作为一个block，且support是system channel的chainSupport，获取的执行器集合committers也是system channel的过滤器集合Apply(env)之后返回的，即第7点末尾中提及的被\_省略掉的过滤器集合，在这里又被重新生成。执行集合中只包含第7点（2）中所述的systemChainCommitter。**（2）** for i, batch := range batches，在循环中依次处理每批消息，将每批消息打包成block，然后support.WriteBlock(block, committers[i],...)在账本中写入block。**（3）** support.WriteBlock(...)执行的是orderer/multichain/chainsupport.go的WriteBlock(...)，在这个函数中，首先在for \_, committer := range committers循环中依次执行committer.Commit()，即将执行器集合兑现，这里因为只有一个systemChainCommitter，因此执行的是orderer/multichain/systemchain.go的Commit() -> scc.filter.cc.newChain(scc.configTx)，最终执行的是orderer/multichain/manager.go的multiLedger的newChain(...)，在这里，**正式的创建了application channel**。可以看到，**创建application channel的最终效果就是：创建了频道的账本（且写入“配置信”作为genesis块）并在orderer的multiLedger对象成员chains中添加一个chainSupport对象并启动了相应的服务**。而上文提及的peer channel update使用到的chainSupport对象，也就是这里创建的。**（4）**还是在WriteBlock(...)中，兑现执行器集合后，cs.ledger.Append(block)也会把application channel的genesis块写入system channel的账本。执行至此，peer channel create在orderer端的工作基本结束。
9. 重新返回到orderer/common/broadcast/broadcast.go的Handle(...)中，定位到support.Enqueue(msg)之后，之后的srv.Send(...\_SUCCESS)就是orderer结点处理完毕创建新的application channel的工作后，向发起peer channel create动作的peer结点返回成功的应答。这里注意一下peer结点接收这个应答的地方是在broadcastClient的Send(...)接口内部（参看peer/commono/ordererclient.go的Send(...)实现中的getAck()），也即当peer/channel/create.go的sendCreateChainTransaction中执行完broadcastClient.Send(chCrtEnv)，在Send(...)内部已经接收到了来自orderer结点的应答信息。至此，**第一步结束**。对于peer channel update动作来说，至此整个动作结束。
10. **开始第二步**。重新返回peer/channel/create.go的executeCreate(cf)中：block, err = getGenesisBlock(cf)，在一定时限内（默认5s，可由peer channel create命令行的-t指定），利用deliver服务每隔200毫秒向orderer结点的指定的application channel索要一次序号为0的block，即上文第2-8所创建的application channel的genesis块，直到成功获取或超时。这一步不展开细讲。
11. **开始第三步**。b, err := proto.Marshal(block) -> ioutil.WriteFile(file, b, 0644)，将第二步获取的block以application channel ID+.block的格式写入文件，供peer channel join动作使用。至此，整个peer channel create动作执行完毕。

| **join** |
| --- |

peer channel join动作看上去很像是peer要加入channel，而channel又在orderer结点中，所以顺气自然的就会认为peer需要发送一些自己的数据到channel，然后channel接收这些数据后添加到自身的对象中。其实join的动作完全是**peer结点本地化自身数据和服务，以达到和对应存在于orderer结点的application channel的数据和服务相配套的一个过程**。数据主要指账本，服务主要指gossip等模块的服务。这里假设peer channel create创建的是一个ID是mychannel的application channel，对应生成的即为mychannel.block文件。在peer/channel/join.go中：

1. joinCmd(cf) -> join(...) -> executeJoin(cf)中，**（1）** spec, err := getJoinCCSpec()创建了一个关于cscc的ChaincodeSpec格式的“说明书”，其中的ChaincodeInput作为scc执行的输入参数，指定了两项：动作cscc.JoinChain、从mychannel.block中读取的mychannel的genesis块数据。**（2）** invocation := &pb.ChaincodeInvocationSpec{...} -> prop, \_, err = putils.CreateProposalFromCIS(...) -> signedProp, err = putils.GetSignedProposal(...)，包装+签名，形成一个背书申请。**（3）** cf.EndorserClient.ProcessProposal(...)，通过背书客户端，发起背书请求。
2. 背书过程省略。可参看peer chaincode ...的背书过程，如《fabric源码解析18-20》。最终直接定位到core/scc/cscc/configure.go。
3. 在configure.go的Invoke(stub)中，根据第1点（1）中所述的“说明书”，将进入case JoinChain:分支：**（1）** block, err := utils.GetBlockFromBlockBytes(args[1])从第二个参数中获取mychannel的genesis块，然后是一系列的检查验证，这里略过不讲。**（2）** joinChain(cid, block)，最后执行join的具体动作。
4. 在joinChain(cid, block)中：**（1）** peer.CreateChainFromBlock(block)（core/peer/peer.go），创建peer结点本地的针对mychannel的链（账本）和gossip服务。这个系列之前关于chaincode，gossip主题的文章中涉及使用的账本，gossip服务，均是在此生成的。**（2）** peer.InitChain(chainID)（core/peer/peer.go），初始化peer结点本地的链。**（3）** producer.SendProducerBlockEvent(block)，创建事件服务，此为监控服务，在此不述。
5. 详解第4点的（1），在CreateChainFromBlock(block)中：**（1）** l, err = ledgermgmt.CreateLedger(cb)，根据mychannel的genesis块，创建peer本地专供mychannel使用的账本。**（2）** createChain(cid, l, cb)，创建针对mychannel使用的链对象，在这个函数中，着重看一下service.GetGossipService().InitializeChannel(...)，该句初始化了peer结点本地gossip模块中专供mychannel使用的gossip服务，并在此gossip服务与orderer结点之间建立了服务连接，由此peer结点就可以源源不断的从orderer结点中的mychannel频道索要block数据块并添加到本地的用于mychannel的账本。
6. 详解第4点的（2），在InitChain(chainID)中：只执行了chainInitializer(cid)，而chainInitializer这个函数变量，是在peer结点启动起来的时候被赋值的，即在peer/node/start.go的serve中执行peer.Initialize(...)时被赋值的，这里给chainInitializer所赋的值是func(cid string){ scc.DeploySysCCs(cid) }，即部署了指定频道ID的system chaincode，也即初始化peer结点的mychannel的链，其实就是在mychannel链上部署的scc。如此，peer chaincode ...动作在mychannel上执行的过程中所使用到的scc也就绪了。这里所部署的针对mychannel的scc要与peer结点启动时部署的scc（peer/node/start.go的serve中执行的initSysCCs()）做区分，即如peer chaincode ...命令若是没有指定频道ID，默认使用的是peer结点启动时部署的scc，否则使用的是具体的针对某一频道ID的链上的scc。
7. 重回core/scc/cscc/configure.go的joinChain(cid, block)中，当第4点执行完毕，则背书过程基本结束，开始一路返回，过程省略，可以直接定位回到peer/channel/join.go的executeJoin(cf)中，在接收到背书的返回结果proposalResp后，整个join动作也宣告完成。

## 结语

以上即为Channel关于create/update，join的执行过程。所述过程线条较粗，尤其是create执行过程中涉及到比较多且复杂的对配置数据的各种变形和处理（主要集中在common/config，common/configtx两个目录中）都进行了省略（若细讲会扰乱主线），读者可以自行深研。其实关于Channel的文章，应该放在类似chaincode、gossip、orderer等主题文章之前的，但笔者能力所限，一直没太吃透，以致拖延至此，所以也建议读者读完此篇文章后，可以重看一下前面的文章。

# fabric源码解析28——官网案例部署

## 概述

从这篇文章开始，系列文章进入实际操作阶段。根据实际的操作，也可以细细的体会操作与代码实现间的相互呼应。官网案例的部署多是一键操作，但为了说明问题，文中尽量手工一步步的实现官网案例的部署。

## 下载部署环境

查看官方文档的Getting Started部分，Install Prerequisites部分讲的是安装案例部署时系统所必须有的软件和环境，就是安装一些软件而已，这里不做说明。其余需要下载的有：1.fabric-samples，包含要部署的chaincode例子。2.运行的docker镜像、原始程序（peer/orderer/configtxgen等）。

官方文档给出了下载第1点内容（即fabric-samples）的一键式傻瓜操作：git clone -b master https://github.com/hyperledger/fabric-samples.git，如果安装了Prerequisites部分的git，则直接执行此命令即可，若没有，也可手动的用浏览器打开 <https://github.com/hyperledger/fabric-samples.git> 这个github网址，手工下载即可。下载后这里形成一个fabric-samples文件夹。

官方文档给出了下载第2点内容（即docker镜像、预编译程序）的一键式的傻瓜式操作：curl -sSL https://goo.gl/6wtTN5 | bash -s 1.1.0-preview，这个命令其实就是定位到 <https://goo.gl/6wtTN5> ，这个网址指向的是一个脚本，然后使用-s指定版本号为1.1.0-preview，并使用bash执行这个脚本。因为墙的存在， <https://goo.gl/6wtTN5> ，这个网址一般打不开，且因版本间的官方文档的变化，每个版本的网址的6wtTN5部分可能都不一样。其实，而这个脚本本身是存在于源码目录中的，我们可以手工下载。另外，下文在部署的时候若要顺利自动执行脚本，对某些文件所在路径是有要求的，因此这里的下载的第2点内容均要下载到fabric-samples文件夹中（主要是bin目录要放到这个文件夹中）。

在github（github没有被墙）上直接搜索fabric，然后找hyperledger/fabric，点击进去，自行选择版本，然后找script目录下的bootstrap-x.x.x.sh，这个文件其实就是上述网址指向的脚本，x代表版本数字，这里以1.0.4版本为例。该脚本文件各版本间框架基本保持不变，只是有若干变量值不同而已，如版本号VERSION。打开，可以尝试直接执行，但是笔者在试的时候，总会出错，比如下载docker镜像说找不到。因此可以根据这个脚本自己手工的下载。该脚本里面有两个比较重要的变量，一个是版本号VERSION/CA\_VERSION，一个是适用的系统ARCH/MARCH。VERSION是手工指定的，如脚本里的VERSION=${1:-1.0.4}，即版本号为1.0.4，自己可以根据自己的版本进行指定，如改为1.0.0（但官网上相应版本的资源是否还存在则不确定），也即下文命令中的凡${VERSION}处都直接替换成1.0.4。CA\_VERSION的值同VERSION。ARCH的值的确定方法脚本里也直接给出了，ARCH=$(echo "$(uname...))，自行将echo "$(uname...)放入自己的系统的命令行指向一下就能得到具体的值，比如在笔者这里输出的为linux-amd64，即下文命令中所有${ARCH}处都直接替换成linux-amd64即可。MARCH同ARCH一样的操作方法，笔者系统的MARCH为x86\_64。如此，变量值确定了，就可以手工执行脚本中的命令了。

脚本主要执行了三项任务，均可手工执行：

* curl https://nexus.hyperledger.org/....tar.gz | tar xz下载与系统适用的二进制程序压缩包并解压，如peer，orderer，configtxgen，configtxlator和脚本。这里可以替换掉命令中的变量值之后，将https://nexus.hyperledger.org/....tar.gz这个网址（其实已经是一个指向tar.gz压缩包文件的下载地址了）直接粘贴到下载工具中自行下载，然后自己解压即可。这里形成了一个bin目录，把这个bin目录放入fabric-samples文件夹下即可。这些都是编译好的现成的程序，而关于如何手工编译这些程序，则在下文详述。
* dockerFabricPull ${FABRIC\_TAG}，下载所有与hyperledger/fabric有关的docker镜像。在这个函数中，循环执行了docker pull hyperledger/fabric-$IMAGES:$FABRIC\_TAG和docker tag hyperledger/fabric...，即先下载镜像，后将镜像改为指定的标签，改标签这一步可做可不做。$IMAGES变量分别是for IMAGES in peer orderer couchdb ccenv javaenv kafka zookeeper tools;处指定的这些值，也即要循环下载peer、orderer、couchdb…这些镜像，$FABRIC\_TAG则是$MARCH-$VERSION的组合，在笔者的系统中即为x86\_64-1.0.4，也即手动逐条执行docker pull hyperledger/fabric-peer:x86\_64-1.0.4，docker pull hyperledger/fabric-orderer:x86\_64-1.0.4…即可。这里需要说一句的是笔者遇到的情况，同样的docker pull命令，有时有执行失败的说没有指定的镜像的错误，各位若发生类似错误，可以多试几次。
* dockerCaPull ${CA\_TAG}，下载CA镜像，原理与上一步相同。

如此，即可手工下载官网案例部署所需的程序、脚本、镜像。需要强调的是，因为fabric项目正在开发过程中，因此版本号也在不断的变，有些版本对应的资源可能在调整中无法下载，可能已经删除等等，因此版本号的值尽量选择比较新的，或fabric在github中有所呈现的版本号。

## 编译peer、orderer、configtxgen等程序

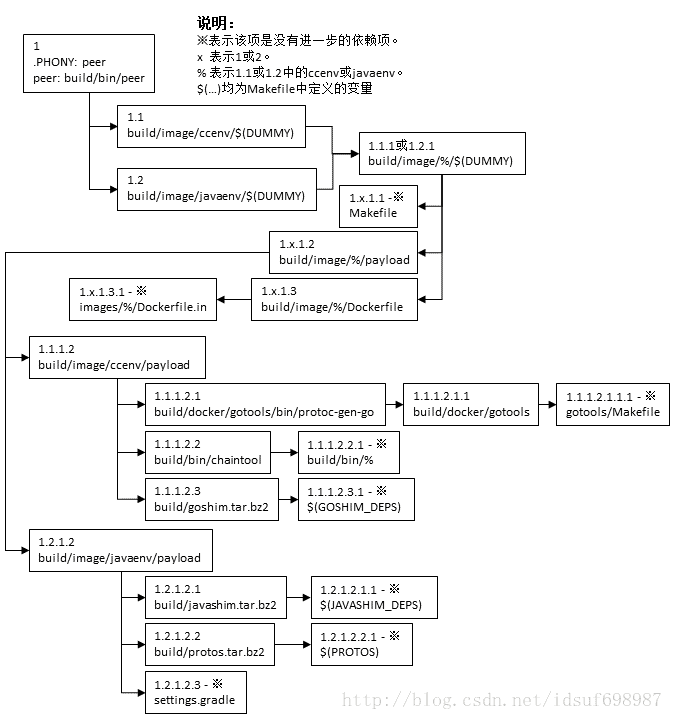
| **文件** | **说明** |
| --- | --- |
| Makefile | 主编译文件，make执行的主体。 |
| docker-env.mk | Makefile直接包含进的一个子makefile文件，用于定义编译期间使用到的关于docker一系列数据。 |
| scripts/ | 目录，包含Makefile编译时调用到的一些shell脚本。 |
| images/ | 目录，包含Makefile编译时使用到的一些用于生成Dockerfile文件的Dockerfile.in文件，  Dockerfile文件则是用于生成各个docker镜像的模板文件。 |
| gotools/ | 目录，包含一个子Makefile文件，该子Makefile文件用于安装Makefile编译时使用到的各种第三方的go工具，  如ginkgo，gocov，govendor等等。 |

fabric源码涉及的技术较多 + 编译期间会存在墙的问题 + 编译的环节很多 + 每个人的系统具体情况都一样，基于此，不讲具体的编译步骤，而是直接讲项目的Makefile构成，至于具体的编译过程，若读懂Makefile，则一切都不再是问题。这里假设各位对Makefile有最简单的了解，比如知道Makefile的基本书写规则以及make依据判定是否重新编译的规则（推荐《跟我一起写Makefile》这本书）。

*表28-1 Makefile相关目录文件*

首先说明的一点，不要被fabric冗长的看似复杂的Makefile糊弄了，其实一切都是万变不离其宗，依然遵循target ... : prerequisites ...这个基本规则。Makefile文件开头的注释中，罗列了所有的target，下文以编译peer结点为例，即make peer命令，其余target的形式与peer如出一辙。在具体编译peer的过程中，target的依赖项可以是另一个子target，而子target又可以有子target，因此整个过程是一个树形发散的过程，因此为了标记清楚编译执行的顺序路径，这里用xxx.xxx.xxx的形式标记每个target，如1表示顶层的target，1.1就表示1下面的第一个子target，1.2就表示1下面的第二个子target，1.1.1则表示1.1下面的第一个子target，如此类推。

先整体说一下Makefile，以include docker-env.mk为线，可将Makefile分为上下两部分：**（1）**上面部分，无论是=，?=，+=，:=，均是对一个下文会使用到的变量的赋值而已，其中个中差别，其自行搜索。include docker-env.mk即在Makefile中包含了docker-env.mk文件，从名字上即可看出该文件定义了一系列运行docker命令时的环境变量数据，其中对.dummy-xxx文件的设计初衷有大段解释，大概意思就是基于docker容器编译输出的文件并不算是标准的系统文件（主要在文件时间上会和实际系统中参与编译的文件不符），因此需要对容器输出的文件创建对应的.dummy-xxx文件，让.dummy-xxx文件真正参与make的编译。另外，在Makefile中到处使用的$DUMMY变量也在docker-env.mk中定义。**（2）**下面部分，遵循target ... : prerequisites ...，是makefile的执行主体。下面make peer的执行路径进行图示：



Makefile真正执行的一条条命令基本上均集中在末端项（即图中带※符号的框所代表的项），具体所做的事情请自行查看学习，不再详述。这些命令根据依赖的前后顺序，因此执行也会有先后顺序，如1.1肯定比1.2先执行，1.1.5也肯定比1.2.1先执行，以此类推。

## 部署

切换到fabric-samples/first-network目录下，傻瓜式脚本自动的操作（由于可能存在系统权限问题，最好以sudo执行所有的脚本、命令）：

1. 执行byfn.sh脚本，生成启动运行所需的必要配置或数据，如channel.tx、gensisblock等。sudo ./first-network/byfn.sh -m generate，这里可以加入一些flag：（1）-c channelDIY，指定channel的ID为channelDIY，若不指定默认为mychannel。（2）-t 10000，指定CLI容器自动退出的时间，若不指定默认为20000秒。（3）-s couchdb，指定使用的数据库为couchdb，若不指定默认为goleveldb。其他的参数可参看byfn.sh中的printHelp()函数中的打印（即执行./first-network/byfn.sh --help显示的内容）。
2. 执行byfn.sh脚本，启动运行环境。sudo ./byfn.sh -m up，第一次执行这一步的时候，需要确保自己的电脑是联网状态，因为启动过程中仍然会下载一些东西，如fabric-ca、fabric-baseos镜像。但是之后再次执行时，由于该下载的都已经下载过了，因此电脑是断网状态也无所谓。

首先关注byfn这个名字，是Build your first network的首字母缩写，寓意很清晰。byfn.sh脚本具体都做了哪些工作，分析这个脚本即可，然后我们也可以顺着这个脚本手工执行。byfn.sh开篇处有两个export，指定了脚本执行过程中寻找程序文件的路径（./和../bin），也指定了fabric的配置路径FABRIC\_CFG\_PATH的值（./）。文件分为3部分。首先，第1部分是定义各种操作的函数体，如networkUp、networkDown、generateCerts等。第2部分从while getopts "h?m:c:t:d:f:s:" opt; do循环开始，分析执行该脚本时携带的各种参数，如上文第1点分析的那些参数。第3部分处于脚本最底部，从if [ "${MODE}" == "up" ]; then开始，根据第2步所确定的MODE值，在每个if分支中执行up、down还是generate操作，这里只看generate操作对应的generateCerts、replacePrivateKey、generateChannelArtifacts三个函数和up操作对应的networkUp函数。

### Crypto Generator

generateCerts函数主要使用了cryptogen工具。在检查cryptogen工具和crypto-config是否存在后（在脚本开始时export指定的路径中寻找），执行了cryptogen generate --config=./crypto-config.yaml，即使用bin/cryptogen工具用于使用first-network/crypto-config.yaml配置文件生成fabric网络拓扑结构和各个组织的根证书。这个网络拓扑结构，指的是你这个区域链初始化时有哪些组织，组织的域名是什么，组织下有多少实体成员。并相应的生成每个组织及组织下实体所使用的身份证书、tls证书。最终在first-network下生成了一个**crypto-config**文件夹。

**crypto-config.yaml**

以组织为单位，如OrdererOrgs、PeerOrgs。在每个组织下可配置一个个实体，这个实体的意思同网络中的一个入口，或者说一个结点的意思，如OrdererOrgs下配置多个orderer结点：

* Name - 组织名
* Domain - 域名
* Specs - 详细说明，更细致的对每个实体名字的配置   
  + Hostname - 主机名，必填（以Hostname.Domain格式形成实体名）
  + CommonName - 主机通用名，可选，若定义，将覆盖上面Hostname.Domain格式的实体名
* Template - 模板，算是批量的对多个实体的名字进行配置   
  + Count - 实体的数量，如5
  + Start - 从哪个数开始，若不定义，则默认从0开始
  + Hostname - 主机名，若这里定义了，Count和Start定义将无效而只配置此名
* User - 普通用户，该配置区域只针对PeerOrgs组织   
  + Count - 普通用户的数量，即Specs配置实体数+Template配置实体数-Admin个数

Specs下可定义多个实体，每个实体都可以包含Hostname、CommonName一对儿字符串，CommonName是可选的。若未定义CommonName，则Hostname.Domain将该实体证书的CN值，若定义了CommonName，则CommonName将是该实体证书的CN值。Template中会从Start开始，定义Count个ordererN.Domain格式名字的实体，Start <= N < Count+Start。Template与Specs不互斥，可同时定义，所定义的实体均会被配置，但是注意两个区域所配置形成的实体名不要重复，否则会形成覆盖。User中，Admin实体数不允许配置，默认为1个，普通用户的数量也可不遵循上述公式。

**crypto-config文件夹**

crypto-config文件夹就是对之后运行起来的整个区域链的网络拓扑结构很好的描述：crypto-config有两个文件夹，表示有两类组织，orderer组织ordererOrganizations和peer组织peerOrganizations。以peerOrganizations为例，又包含两个文件夹，表示该类组织包含两个组织，org1.example.com和org2.example.com。以org1.example.com为例，包含组织根证书文件夹ca、组织中所有的MSP身份文件夹msp、组织中所有的TLS网络证书文件夹tlsca、组织中所有成员角色的MSP身份文件夹users、组织包含的实体结点文件夹peers。在peers中，包含两个文件夹，代表两个实体结点，peer0.org1.example.com和peer1.org1.example.com。以peer0.org1.example.com为例，包含结点MSP身份文件夹、结点网络证书文件夹tls。如此，一级一级的形成管理、对应关系，比如peer组织中的org1中，实体成员peer0与peer1的MSP身份文件夹中的管理员证书应该一样，且应该等于org1的msp/admincerts下的证书。再比如peer0的根证书是msp/cacerts/ca.org1.example.com-cert.pem，从名字上即可看出peer0使用的是org1组织的证书，如此的话，与peer0同级的peer1使用的也应该是所属组织org1的证书作为自己的根证书，而实际情况确实如此。再比如，peer0属于org1组织，则peer0的自有的签名证书msp/signcerts/peer0.org1.example.com-cert.pem理应是由org1这个组织所认证的证书，即由org1.example.com/ca/ca.org1.example.com-cert.pem所认证签发，对于此点我们可以自己写一个小程序验证一下，即分别读取两个证书，然后调用peer0证书的Verify()函数验证，源码参看verify\_cert.go。或者你若是在Ubuntu环境下，可以找到该证书，直接点开peer0的证书，Ubuntu中由解析证书的软件，在Issuer Name（发行者）栏中可以清楚的看到组织O为org1.example.com，CN（通用名称）为ca.org1.example.com，即org1的证书。事实证明peer0的证书确实是org1的证书签发的。另外，组织、实体结点等的命名均是xxx.xxx.xxx格式，也形成了**前面的在后面的范围之内**的意思。

replacePrivateKey函数“人如其名”，所做的主要是在generateCerts生成各个实体的证书和私匙后，替换部署的fabric-ca0、fabric-ca1容器所使用的配置文件中的指定的容器所使用的证书对应的私匙文件名。这一点也很好理解：fabric-ca容器是用于给新加入某一组织的peer结点签发认证的证书的，因此ca容器首先得自己持有组织的根证书和根证书的私匙（才能进行签发新证书的操作）。根证书的文件名根据cryptogen程序可预先得知，但是对应生成的私匙（\*\_sk）文件名则事先是不知道的，因此在docker-compose-e2e-template.yaml（带有template字样，表明是一个样板文件）中预先是用CA1\_PRIVATE\_KEY、CA2\_PRIVATE\_KEY这样的字段临时代替了私匙文件名，也因此需要在生成org1和org2两个组织的证书的私匙后用私匙文件名替换掉这两个字段。具体的操作是：先复制了一份docker-compose-e2e-template.yaml模板文件为docker-compose-e2e.yaml，在相应目录中执行PRIV\_KEY=$(ls \*\_sk)获取私匙文件名，然后使用sed $OPTS "s/...命令将docker-compose-e2e.yaml中的CA1\_PRIVATE\_KEY、CA2\_PRIVATE\_KEY字段分别替换成org1、org2组织的根证书对应的私匙文件名。

### Configuration Transaction Generator

generateChannelArtifacts函数主要使用了configtxgen工具。函数所做的事情和configtxgen工具的介绍可以参看脚本中该函数上面的注释。在此简单说一下，configtxgen使用configtx.yaml配置生成部署的网络实体所需的“手工艺品”，这些生成的文件均放入了**channel-artifacts**文件夹内：

* genesis.block - 不必多说，orderer服务启动必备的。特别说一下，每个组织的根证书都包含在gensis.block中。
* channel.tx - fabric的channel的配置。该配置在orderer服务启动创建channel时配置channel。这里就已经形成了channel的读写策略（即哪些实体可以读，哪些实体可以写，下同）。
* Org1MSPanchors.tx和Org2MSPanchors.tx - 用于指定在channel中，当前peer组织中都有哪些peer结点。这里就已经形成了组织的读写策略。

configtx.yaml包含部署网络实体的定义，主要分为三部分：Profiles，直接供configtxgen使用的配置部分，可以通过-profile命令行参数指定配置这里的哪一项；Organizations，定义了三个成员（组织），orderer组织OrdererOrg、两个peer组织Org1和Org2（每个组织管理维护两个peer结点），供Profiles使用；其余的Orderer和Application，具体的定义，供Profiles和Organizations两部分使用。这里可以留意的是每个组织都配置了MSPDir项，即每个组织的MSP所在的路径，这正是为了configtxgen能够找到每个组织的根证书，找到组织的根证书，就可以对实体身份或实体交易的数字签名进行验证。

configtx.yaml配置文件（的写法）与configtxgen的实现紧密相关，参看源码common/configtx/tool/configtxgen/main.go：（1）configtxgen默认就是读取configtx.yaml作为配置文件的，在main中，config := genesisconfig.Load(profile)默认就读取了当前文件夹或FABRIC\_CFG\_PATH或fabric/sampleconfig/（参看byfn.sh开始处的两个export，三个路径依次搜查指定的配置文件，先搜查到即停止）下的名为configtx的文件（也就是configtx.yaml）的profile配置项（profile由-profile参数指定）。（2）在main中，所有的flag.StringVar(...)即为configtxgen可以解析的命令行参数，其中就有outputBlock、outputCreateChannelTx、outputAnchorPeersUpdate，即是用于生成上面所说的“手工艺品”。比如，执行configtxgen时给的了-outputBlock参数，则在main的中会进入if outputBlock != ""分支，从而执行doOutputBlock(config, channelID, outputBlock)来生成gensisblock。另外还有两个用于检查参数，inspectBlock、inspectChannelCreateTx，分别用于检查生成的gensis.block和channel.tx文件是否符合要求。（3）对于生成的gensisblock、channel.tx都包含什么数据，建议执行同目录下的main\_test.go中的TestBlockFlags，TestConfigTxFlags两个测试函数，在测试过程中读取的是sampleconfig下的configtx.yaml中的配置，并会将生成的gensis.block和channel.tx以JSON格式打印出来。

重看generateChannelArtifacts函数，在检查configtxgen工具是否存在后，开始执行任务：**（1）**configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block，生成gensisblock。-profile指定使用configtx.yaml中的TwoOrgsOrdererGenesis项配置去生成gensisblock，-outputBlock则指定了生成的gensisblock放到哪里和名字。**（2）**configtxgen -profile TwoOrgsChannel -outputCreateChannelTx...，使用configtx.yaml中的TwoOrgsChannel生成channel.tx。**（3）**执行了两个configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate...命令，使用configtx.yaml中的TwoOrgsChannel配置生成org1、org2的配置文件Org1MSPanchors.tx和Org2MSPanchors.tx。

### networkUp

networkUp函数真正启动了部署网络。函数所做的很简单（前提是熟悉docker相关操作），if [ ! -d "crypto-config" ]; then检查crypto-config文件夹是否存在，也即变相的检查上文讲述过的generateCerts、replacePrivateKey、generateChannelArtifacts三个函数的准备工作是否已经做完。然后根据是否使用couchdb数据库，执行对应if分支中的... docker-compose -f ...命令，其实就是除了-f $COMPOSE\_FILE外是否多添加一个-f $COMPOSE\_FILE\_COUCH参数，即多指定一个docker-compose-couch.yaml文件以启动couchdb容器。-f $COMPOSE\_FILE默认指定的是docker-compose-cli.yaml，也可在byfn.sh的命令行参数中以-f指定。docker-compose -f命令所做的就是根据指定的docker-compose-cli.yaml文件自动部署文件中定义的容器。

## 运行容器+区域链操作

关于docker-compose-cli.yaml，共在byfn网络中在services下设置了6个服务：一个orderer容器，四个peer容器，一个cli容器。其中orderer和peer容器都具体定义在base/docker-compose-base.yaml中，cli则在当前文件中具体定义。容器的每个字段的定义没有冗余的地方，每个字段都很重要，具体在用到的时候会回过头来提及。docker-compose-couch.yaml中，除了为每个peer结点容器都对应启动一个couchdb容器外，还对每个peer容器增加了一些环境变量。一旦networkUp函数执行成功，所有的容器，容器的设置，都会被应用和启动。

在启动各个容器时，docker-compose-cli.yaml中每个服务下的command字段都给定了默认执行的命令。orderer容器默认执行的是orderer（也即orderer start命令），4个peer容器各自执行的是peer node start，最后启动的cli容器，在working\_dir目录下默认执行了/bin/bash -c './scripts/script.sh ${CHANNEL\_NAME} ${DELAY}; sleep $TIMEOUT'，即scripts下的script.sh脚本，但是当script.sh脚本运行结束之后，cli容器即会自动停止并退出。script.sh的执行就是各位在终端上看到的以大大的START开始，并以大大的END结束的过程。这个过程主要执行了几种区域链操作，如创建channel、加入channel、升级组织配置、安装部署chaincode、调用chaincode，算是cli变相的先测试一下这个部署的区域链网络的基本操作，同时也把peer1.org1和peer1.org2两个结点加入各自组织（回忆一下，生成gensis.block、channel.tx所使用的configtx.yaml中，org1，org2组织下均只包含了一个peer0结点）的状态留在了还在运行的网络里面（一个orderer结点容器，4个peer结点容器仍在运行）。由此我们可以知道，当你在终端上看到大大的START时，示例中的6个容器已经启动，整个实例的网络实际上已经部署完毕，再执行script.sh脚本只是为了在这个区域链网络中测试各种操作，如果想手工自己操作，在docker-compose-cli.yaml中将cli服务中的command字段注释掉即可，甚至不启动cli容器也可以，转而在各个结点的容器中执行操作。这里需要注意的是，script.sh脚本是在cli容器内运行的，也就是说脚本所使用的环境变量、路径等元素的值，均为cli容器中的值。而命令中script.sh后边的三个变量${CHANNEL\_NAME}、${DELAY}、$TIMEOUT则来自于byfn.sh中networkUp中执行的... docker-compose -f ...命令。由于script.sh脚本进行了区域链网络的基本操作，因此我们可以细细分析，从中学习，若产生问题，可以适当的修改脚本，验证猜想从而解除疑问。

再回头分析一下docker-compose-cli.yaml中对cli容器的设置：container\_name指定了容器名字为cli，image指定了cli以下载的hyperledger/fabric-tools为镜像，tty也设置为true（即开启了控制台），environment下设置了cli启动时所设置的环境变量，working\_dir指定了cli的基准工作目录，command指定了cli启动时运行的命令，volumes指定了宿主主机与cli之间挂接的目录，depends\_on指定了cli所依赖的容器（即必须这些容器正确启动之后cli才能启动，也即cli是最后启动的），networks指定了cli所使用的网络。

在script.sh中，从echo "Creating channel..."处开始执行命令，均为调用脚本上面已定义好的函数，这些函数有：

* setGlobals() - 公用函数，根据所给的参数来指定配置针对某一peer结点的环境变量以执行其他操作。这里预设的是给一个参数，0对应peer0.org1，1对应peer1.org1，2对应peer0.org2，3对应peer1.org2。比如当调用setGlobals 0，表示针对peer0.org1设置cli容器的CORE\_PEER\_LOCALMSPID（peer结点所在组织的MSPID）、CORE\_PEER\_TLS\_ROOTCERT\_FILE（peer结点的TLS证书所在路径）、CORE\_PEER\_MSPCONFIGPATH（peer结点的MSP配置所在路径）、CORE\_PEER\_ADDRESS（peer结点的容器通信地址）等环境变量。调用者需要这些环境变量的值继续执行其他操作。
* createChannel() - setGlobals 0设置了关于peer0.org1的环境变量，然后因为CORE\_PEER\_TLS\_ENABLED的值默认是true（启用TLS连接），因此执行了peer channel create...-o...-c...-f...--tls...命令，从而建立了一个与orderer结点相连的Application Channel。channel的名为$CHANNEL\_NAME指定，该变量来自于cli容器执行script.sh时所给的第1个参数，这个第1个参数又来自执行byfn.sh时所给的-c参数值。若不指定，默认值是mychannel，这里假设使用的是默认值。创建mychannel后，会在cli容器内的working\_dir下生成一个mychannel.block，供之后要加入mychannel的结点使用。mychannel.block中包含了channel.tx中的配置信息。
* joinChannel() - 循环0-3四个值，分别代表4个peer结点，在循环中依次调用setGlobals $ch，joinWithRetry $ch将每个peer结点加入上一步所创建的mychannel中。joinWithRetry中具体执行的命令为peer channel join -b $CHANNEL\_NAME.block，其中$CHANNEL\_NAME.block即是createChannel生成的mychannel.block。
* updateAnchorPeers() - 使用之前生成的Org1MSPanchors.tx、Org2MSPanchors.tx升级mychannel中**现有组织**的信息（参看configtx.yaml中的生成升级包所使用的TwoOrgsChannel下包含的组织Org1、Org2的信息，如MSPDir等）。使用的命令为peer channel update -o ... -c ... -f...。Org1MSPanchors.tx、Org2MSPanchors.tx就算是针对组织的升级包，可以升级组织的配置信息，又由于生成的Org1MSPanchors.tx、Org2MSPanchors.tx与创建mychannel的channel.tx所使用的组织信息一致，所以这里的升级实际上不会对现有部署的网络产生任何影响，这里只是为了测试peer channel update这项功能而已（也即在script.sh中将updateAnchorPeers 0和updateAnchorPeers 2注释掉，也没有关系）。
* installChaincode() - 使用指定的结点安装一个chaincode，经过cli的路径映射，安装的chaincode实际上是下载fabric-sample/chaincode/chaincode\_example02。具体执行的命令为peer chaincode install -n mycc -v 1.0 ...，分别在peer0.org1和peer0.org2两个结点上安装。
* instantiateChaincode() - 从指定结点部署安装的chaincode\_example02，这里是从peer0.org2结点部署。具体执行的命令是peer chaincode instantiate -o ... --tls true --cafile...-C...-n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"，-c指定了初始化的参数，初始化了账户a的余额为100，账户b的余额为200。-P指定了chaincode的背书策略，因为是OR，所以Org1或Org2下的任一成员背书交易即可，这个意思就是Org1或Org2组织下的所有peer结点都能执行这个部署的chaincode。
* chaincodeQuery() - 以第一个参数$1指定结点身份执行查询，对比查询的结果是否a账户的余额是否等于第二个参数$2。查询过程会在$TIMEOUT内每隔$DELAY秒尝试查询一次，每次执行的命令是peer chaincode query -C ... -n mycc -c '{"Args":["query","a"]}'。
* chaincodeInvoke() - 从指定结点执行chaincode\_example02。具体执行的命令是peer chaincode invoke -o ... --tls true --cafile ... -C ... -n mycc -c '{"Args":["invoke","a","b","10"]}'，所做的就是将a账户的10元转账给b账户（chaincode\_example02源码中所实现的）。