**HyperLedger Fabric部署与链码解读**

目录

HyperLedger Fabric部署与链码解读 1

1.Fabric简介 3

2.Fabric中的基本概念 3

3. Fabric整体架构概览 4

3.1 Fabric底层服务模块 4

3.2 Fabric应用层开发 5

3.3 Fabric各节点功能 6

4. Fabric组网通信 7

4.1消息结构与访问策略 7

4.1.1消息协议结构 7

4.1.2配置管理结构 7

4.1.3访问控制之策略管理 7

4.2 Gossip协议 8

4.2.1 Gossip数据分发 8

4.2.2 Gossip成员认证与身份管理 8

4.2.3 Gossip对多通道支持 8

5. Fabric账本存储 8

6. fabric的共识机制 8

7. Fabric文件组织 9

7.1常用包介绍 9

7.2 源码相关工具，一些辅助代码包 11

7.3 安装部署 11

8. Fabric安装部署实践 11

8.1 Fabric环境准备 11

8.2 Fabric代码下载和镜像下载 14

8.3 Fabric组网部署及过程解读 14

9. Fabric智能合约解读 21

9.1 智能合约特点 21

9.2 智能合约的一次完整过程 21

9.3 Fabric链码生命周期 21

9.3.1 应用链码的生命周期 21

9.3.2 系统链码 25

9.3.2.1 系统链码生命周期LSCC 25

9.3.2.2 系统链码配置管理CSCC 25

9.3.2.3 系统链码查询(QSCC)，交易背书(ESCC)，及验证(VSCC) 25

9.4智能合约的链码实现 25

9.4.1实现链码所需接口 26

9.4.2 实现链码结构 26

9.5案例解读 26

9.5.1 链码初始化证书与初始区块生成 27

9.5.2 链码通道生成 28

9.5.3 生成各组织内的锚节点配置文件 28

9.5.4 节点初始化完成后加入通道 28

9.5.6 节点的链码安装 28

9.5.7 节点的链码初始化 28

9.5.8 节点的更新交易，查询交易，删除交易操作 28

**1.Fabric简介**

Fabric是超级账本中的一个项目，用以推进区块链技术。和其他区块链类似，它也有一个账本，使用智能合约，且是一个参与者可以分别管理自身交易的系统。它是一个联盟链。

Fabric与其他区块链系统最大的不同在于它是隐私的、许可的网络。相对于像其他区块链那样通过“工作量证明”来验证身份（允许任何人加入网络），Fabric的成员通过会员注册服务提供商来加入网络。

Fabric提供了多种可插拔的选择，账本数据可以以多种形式存储，共识机制可切换，且支持不同的会员服务。Fabric还提供了创建通道的能力，允许一组参与者创建一个单独的账本交易。这对于一些不想让其竞争对手（同为参与者）知道其每一笔交易的参与者来说尤其重要，如果两个参与者间创建了一个通道，那么其他参与者不会拿到此通道的数据。

**2.Fabric中的基本概念**

**Fabric的总账（Ledger）**包括两部分：世界状态与交易日志。Fabric的每个参与者都持有一份账本副本。世界状态部分描述了账本在某个时间点的状态。它是账本的数据库。交易日志部分则记录了导致当前世界状态的所有交易，是世界状态的历史记录。账本是世界状态与交易日志的结合。

**智能合约（链码Chaincode）**：Fabric智能合约写在链码中，且当外部应用与账本互动时被调用。大多数情况中，链码只与账本的数据库部分互动（查询等），即世界状态，而不是交易日志。链码可以被多种语言编写。

**共识（consensus）**：交易必须按照它们发生的顺序写入账本，即使它们可能发生在网络中不同的参与者身上。为了做到这一点，必须建立交易的排序服务，且有一套抵制错误交易（恶意）写入账本的方法。

**Anchor（锚点）**：一般指作为刚启动时候的初始联络元素或与其它结构的沟通元素。如刚加入一个 channel 的节点，需要通过某个锚节点来快速获取 channel 内的情况（如其它节点的存在信息）。

**Block（区块）**：代表一批得到确认的交易信息的整体，准备被共识加入到区块链中。

**Blockchain（区块链）**：由多个区块链接而成的链表结构，除了初始区块，每个区块头部都包括前继区块内容的 hash 值。

**Channel（通道）**：Fabric 网络上的私有隔离。通道中的 chaincode 和交易只有加入该通道的节点可见。同一个节点可以加入多个通道，并为每个通道内容维护一个账本。

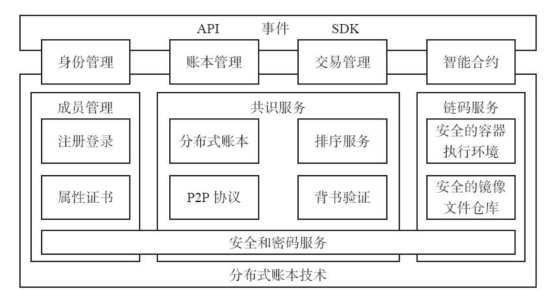
**Endorser（推荐节点或背书节点）**：1.0 架构中一种 peer 节点角色，负责检验某个交易是否合法，是否愿意为之背书、签名。

**Transaction（交易）**：执行账本上的某个函数调用或者部署 chaincode。调用的具体函数在 chaincode 中实现。

**Member（成员）**：代表某个具体的实体身份，在网络中有用自己的根证书。节点和应用都必须属于某个成员身份。同一个成员可以在同一个通道中拥有多个 peer 节点，其中一个为 leader 节点，代表成员与排序节点进行交互，并分发排序后的区块给属于同一成员的其它节点。

**MSP（Member Service Provider，成员服务提供者）**：抽象的实现成员服务（身份验证，证书管理等）的组件，实现对不同类型的成员服务的可拔插支持。

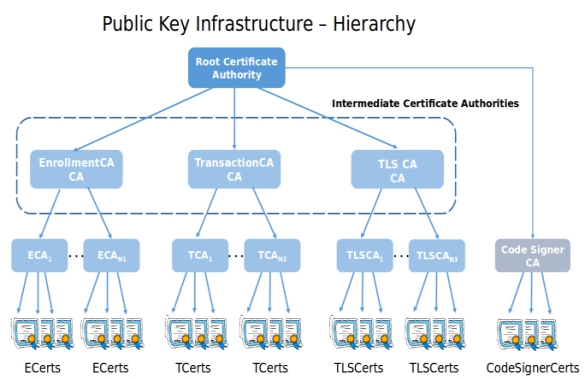
**3. Fabric整体架构概览**



**3.1 Fabric底层服务模块**

(1) 成员管理MSP（Membership Service Provider）

MSP对成员管理进行了抽象， 每个MSP都会建立一套根信任证书体系， 利用PKI（Public Key Infrastructure） 对成员身份进行认证， 验证成员用户提交请求的签名。 结合Fabric-CA或者第三方CA系统， 提供成员注册功能， 并对成员身份证书进行管理， 例如证书新增和撤销[有证书撤销列表]。 注册的证书分为注册证书（ECert） 、 交易证书（TCert） 和TLS证书（TLS Cert） ， 它们分别用于用户身份、 交易签名和TLS传输。



(2) 共识服务

在分布式节点环境下， 要实现同一个链上不同节点区块的一致性， 同时要确保区块里的交易有效和有序。 共识机制由3个阶段完成： 客户端向背书节点提交提案进行签名背书， 客户端将背书后的交易提交给排序服务节点进行交易排序， 生成区块和排序服务， 之后广播给记账节点验证交易后写入本地账本。 网络节点的P2P协议采用的是基于Gossip的数据分发， 以同一组织为传播范围来同步数据， 提升网络传输的效率。

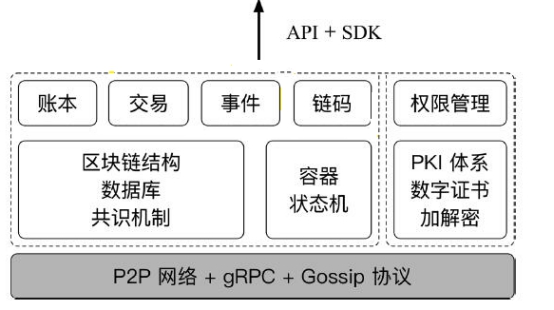
(3) 链码服务

智能合约的实现依赖于安全的执行环境， 确保安全的执行过程和用户数据的隔离。 Hyperledger Fabric采用Docker管理普通的链码， 提供安全的沙箱环境和镜像文件仓库。 其好处是容易支持多种语言的链码， 扩展性很好。

(4) 安全和密码服务

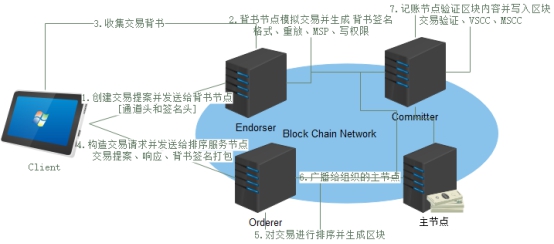
Hyperledger Fabric 1.0专门定义了一个BCCSP（BlockChain Cryptographic Service Provider） ， 使其实现密钥生成、 哈希运算、 签名验签、 加密解密等基础功能。 BCCSP是一个抽象的接口， 默认是软实现的国标算法。

**3.2 Fabric应用层开发**



Fabric为应用提供了gRPC API，以及封装API的SDK供应用调用。应用可以通过SDK访问Fabric网络中的多种资源，包括账本、交易、链码、事件、权限管理等。应用开发者只需要跟这些资源打交道即可，无需关心如何实现。其中，账本是最核心的结构，负责记录应用信息，应用则通过发起交易来向账本中记录数据。交易执行的逻辑通过链码来承载。整个网络运行中发生的事件可以被应用访问，以触发外部流程甚至其他系统。权限管理则负责整个过程中的访问控制。账本和交易进一步地依赖核心的区块链结构、数据库、共识机制等技术；链码则依赖容器、状态机等技术；权限管理利用了已有的PKI体系、数字证书、加解密算法等诸多安全技术。底层由多个节点组成P2P网络，通过gRPC通道进行交互，利用Gossip协议进行同步。层次化结构提高了架构的可扩展和可插拔性，方便开发者以模块为单位进行开发。

**3.3 Fabric各节点功能**



**v 客户端(App)**：客户端应用使用SDK来跟Fabric网络打交道。首先，客户端从CA获取合法的身份证书来加入网络内的应用通道。发起正式交易前，需要先构造交易提案(Proposal)提交给Endorser进行背书(通过EndorserClient提供的ProcessProposal())；客户端收集到足够(背书策略决定)的背书支持后可以利用背书构造一个合法的交易请求，发给Orderer进行排序(通过BroadcastClient提供的Send()处理。客户端还可以通过事件机制来监听网络中消息，获知交易是否被成功接收。命令行客户端的主要实现代码在peer/chaincode目录下。

**v Endorser节点**：主要提供ProcessProposal()方法供客户端调用，完成对交易提案的背书(目前主要是签名)处理。收到来自客户端的交易提案后，首先进行合法性和ACL权限检查，检查通过则模拟运行交易，对交易导致的状态变化(以读写集形式记录，包括所读状态的键和版本，所写状态的键值)进行背书并返回结果给客户端。注意网络中可以只有部分节点担任Endorser角色。主要代码在core/endorser目录下。

**v Orderer节点**：仅负责排序。为网络中所有合法交易进行全局排序，并将一批排序后的交易组合生成区块结构。Orderer一般不需要跟账本和交易内容直接打交道。主要实现代码在orderer目录下。对外主要提供Broadcast()和Deliver()两个RPC方法。

**v Committer节点**：负责维护区块链和账本结构(包括状态DB、历史DB、索引DB等)。该节点会定期地从Orderer获取排序后的批量交易区块结构，对这些交易进行落盘前的最终检查(包括交易消息结构、签名完整性、是否重复、读写集合版本是否匹配等)。检查通过后执行合法的交易，将结果写入账本，同时构造新的区块，更新区块中BlockMetadata记录交易是否合法等信息。同一个物理节点可以仅作为Committer角色运行，也可以同时担任Endorser和Committer这两种角色。主要实现代码在core/committer目录下；

**v CA**：负责网络中所有证书的管理(分发、撤销等)，实现标准的PKI架构。主要代码在单独的fabric-ca项目中。CA在签发证书后，自身不参与网络中的交易过程。

|  |
| --- |
| **注：**交易验证：格式、签名、内容篡改、节点是否在通道  VSCC：链码名称、版本、更新  MSCC：根据交易位图判断是否有读写权限 |

**4. Fabric组网通信**

**4.1消息结构与访问策略**

**4.1.1消息协议结构**

信封消息认证内容中最基本的单元，包括负载消息和签名，满足两个条件，peer节点才会接受一个Envelope:

1. 消息中指定的时期信息[区块高度定义]是当前窗口期

2. 该负载在该周期内只看到一次[没有重放]—由nonce确定

**4.1.2配置管理结构**

Envelope.data 中包含CONFIGURATION\_TRANSACTION  为配置消息

Envelope.data 中包含ENDORSER\_TRANSACTION  为背书消息

**4.1.3访问控制之策略管理**

策略分为两类：SignaturePolicy 签名验证策略；ImplicitMetaPolicy 隐含元策略；

交易背书策略：对交易进行背书的规则，与channel和chaincode相关，在chaincode 初始化的时候指定。只有通过背书策略的交易才是有效的。可以在部署链码的时候通过-P参数指定。

链码实例化策略：链码实例化策略是用来验证是否有权限进行链码实例化和链码升级的。链码实例化策略是在对链码打包和签名的时候指定的，如果没有指定实例化策

略，默认是通道的管理员才能实例化。

通道管理策略：指定了通道的读取、写入、管理权限。

**4.2 Gossip协议**

**4.2.1 Gossip数据分发**

在背书节点模拟执行签名的结果经过排序后，会广播给其它所有节点，广播时使用的是原子广播服务，即逻辑上所有节点接受到的消息顺序是相同的的，相同序号的内容也是相同的。fabric中采用基于Gossip协议实现P2P数据广播，Gossip模块负责连接排序服务和peer节点，实现从单个源点到所有节点高效的数据分发，达到不同节点的状态同步。利用Gossip协议可以处理拜占庭问题，动态节点增加和网络分区等问题，账本信息，状态信息，成员信息都会通过Gossip协议进行分发。

**4.2.2 Gossip成员认证与身份管理**

在gossip网络中，每个节点都有超记账本网络认可的MSP颁发的证书，从证书中计算hash值导出的一个标识符称为节点的PKI-ID。身份管理模块管理节点标识符和证书之间的映射，在内部维护了以PKI-ID为键，证书为值的映射表pkiID2cert，通过PKI-ID获取节点的证书，也可通过从证书导出的标识符和PKI-ID是否匹配，来验证证书是否有效。

**4.2.3 Gossip对多通道支持**

创建通道的目的是为了限制信息传播的范围，它是和某一个账本相关联的。客户端SDK通过发送一个CONFIGURATION的交易背书请求来创建一个通道，然后通过排序服务广播给其它节点。创建通道的请求中包含组成通道的组织列表，定义了那些组织及其成员可以加入通道。通道创建好后，SDK通知组织内节点加入通道，节点的Gossip模块通过广播消息确定自己加入通道中。通过Gossip协议转发给其它节点的消息带有节点的PKI-ID、节点的签名，没有节点的签名不能转发。

**5. Fabric账本存储**

fabric的数据存储主要就是账本数据，以二进制文件的形式存储，包括一下元素：账本编号，账本数据，区块索引，状态数据，历史数据。每个peer会维护4个DB，idStore，stateDB，versionedDB，blockDB。

**账本的存储管理包括**：提交区块到账本（AddBlock）、获取区块链信息（GetBlockchainInfo）、获取区块数据（RetrieveBlocks）、关闭区块存储（Shutdown）。

**账本的索引管理**：用于跟踪区块和确定交易保存在那个文件中。可进行的索引操作有：根据哈希值获取区块、根据区块编号获取区块、根据交易编号获取交易、 根据区块编号和交易编号获取交易、根据交易编号获取区块、根据交易编号获取交易验证码。

**状态数据**：记录的是交易的执行结果，最新的状态代表了通道（Channel）上所有键的最新值，又称为世界状态。状态数据库是区块链交易日志中的索引视图，可随时根据区块链重新生成。

**历史数据**：记录了每个状态数据的历史信息，每一个历史信息用一个四元组表示，namespace[channelID]，writeKey，blockNo，tranNo。

**6. fabric的共识机制**

fabric的共识发生在orderer进行排序时，分为3个阶段：交易背书，交易排序，交易验证。

当前实现主要有solo和Kafka两种机制。Kafka中使用交易在消息流中的有序来确定排序。

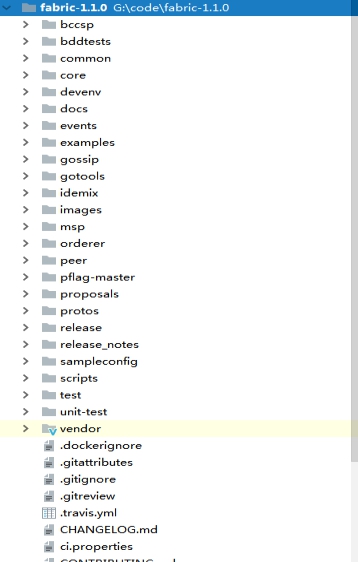
主要通过在configtx.yaml中的orderer.Type来进行配置。每一个通道对应Kafka中的一个topic，排序时Kafka会分为两种角色，接受交易的生产者和处理交易的消费者。基于Kafka的排序服务节点也分为两种角色：作为服务端，对记账节点和应用程序进行排序后的原子广播服务；对于内部kafka集群作为客户端，接受记账节点和应用程序的交易请求并转发给kafka集群，然后将经过Kafka集群排序后的交易打包成区块广播给记账节点。

作为接受交易的生产者，会进行对消息信封和通道的检验，并判断是配置更新还是普通交易，通过检验后将交易信息提交对Kafka集群对应的主题和分区。

作为接受交易的消费者，会先通过chainID获取配置区块等信息启动协程，然后对信息大小进行判断，确信信息是否需要分割，然后打包生成新的区块，生成新区块后对消息进行判断，确定是配置交易还是普通交易，若为配置交易则要更新通道并重启链码，最后将区块保存到账本中，并通过广播更新到其他节点。

**7. Fabric文件组织**

**7.1常用包介绍**



bccsp 包：实现对加解密算法和机制的支持。

common 包：一些通用的模块；

core 包：大部分核心实现代码都在本包下。其它包的代码封装上层接口，最终调用本包内代码；

events 包：支持 event 框架；

examples 包：包括一些示例的 chaincode 代码；

gossip 包：实现 gossip 协议；

msp 包：Member Service Provider 包；整体结构

order 包：order 服务相关的入口和框架代码；

peer 包：peer 的入口和框架代码；

protos 包：包括各种协议和消息的 protobuf 定义文件和生成的 go 文件。

**7.2 源码相关工具，一些辅助代码包**

bddtests：测试包，含有大量 bdd 测试用例；

gotools：golang 开发相关工具安装；

vendor 包：管理依赖；

**7.3 安装部署**

busybox：busybox 环境，精简的 linux；

devenv：配置开发环境；

images：镜像生成模板等。

scripts：各种安装配置脚本；

**8. Fabric安装部署实践**

**8.1 Fabric环境准备**

|  |  |
| --- | --- |
| 环境 | 所需版本 |
| ubuntu | >=14 |
| docker | >=1.10 |

安装ssh   sudo apt-get install ssh

更改源   sudo vim /etc/apt/sources.list

:%s/us./cn./g

sudo apt-get update

安装python  sudo apt-get install python3.5

sudo apt-get install python-pip

sudo -H python -m pip install --upgrade pip

安装代理网络 sudo pip install shadowsocks

touch shadowsocks.json

{

"server" : "45.78.13.106",

"server\_port" : 44,

"localPort" : 1080,

"password" : "xxxxx",

"method" : "rc4-md5",

"remarks" : ""

}

后台运行sslocal

sudo mv shadowsocks.json /etc/

nohup sslocal -c /etc/shadowsocks.json &

sudo apt-get install polipo

#sudo vim /etc/polipo/config

    socksParentProxy = "127.0.0.1:1080"

    socksProxyType = socks5

    chunkHighMark = 50331648

    objectHighMark = 16384

    serverMaxSlots = 64

    serverSlots = 16

    serverSlots1 = 32

    proxyAddress = "0.0.0.0"

proxyPort = 8123

# sudo /etc/init.d/polipo restart

配置http代理

# export http\_proxy='http://127.0.0.1:8123'

# export https\_proxy='http://127.0.0.1:8123'

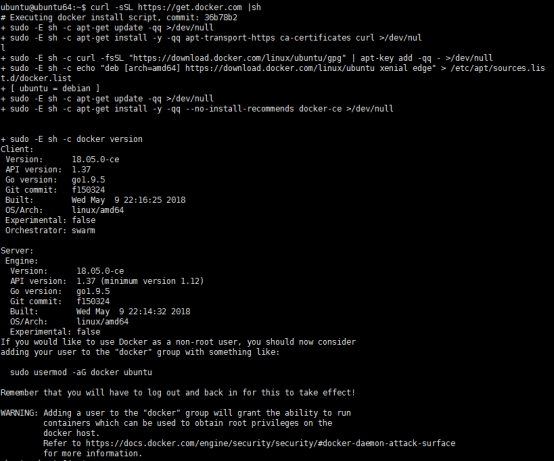
验证

# wget www.google.com

安装Docker

curl -sSL <https://get.docker.com> | sh

**完成后如下图所示：版本为18.05**



DAOCloud加速器

curl -sSL https://get.daocloud.io/daotools/set\_mirror.sh | sh -s http://85f9f519.m.daocloud.io





sudo vim /etc/default/docker

DOCKER\_OPTS="-s=aufs -r=true --api-cors-header='\*' -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --registry-mirror=curl -sSL https://get.daocloud.io/daotools/set\_mirror.sh | sh -s http://85f9f519.m.daocloud.io"

加入用户到docker中

sudo usermod -aG docker fabric

启动docker服务

service docker restart

下载docker-compose

sudo curl -L https://github.com/docker/compose/releases/download/1.17.1/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

安装环境所需软件

go安装

[sudo apt-get install golang   |  卸载go  sudo apt-get purge golang-go]

#wget https://storage.googleapis.com/golang/go1.10.1.linux-amd64.tar.gz

#sudo tar -xzf go1.10.1.linux-amd64.tar.gz -C /usr/local

设置环境变量

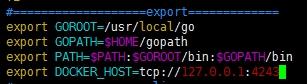
vim  ~/.bashrc

export GOROOT=/usr/local/go

export GOPATH=$HOME/gopath

export PATH=$PATH:$GOROOT/bin:$GOPATH/bin

export DOCKER\_HOST=tcp://127.0.0.1:4243



source .bashrc

**8.2 Fabric代码下载和镜像下载**

下载fabric代码

mkdir -p $GOPATH/src/github.com/hyperledger

cd $GOPATH/src/github.com/hyperledger

git clone https://github.com/hyperledger/fabric.git

下载fabric-ca代码

cd $GOPATH/src/github.com/hyperledger

git clone https://github.com/hyperledger/fabric-ca.git

下载案例快速搭建

cd  $GOPATH/src/github.com/hyperledger

git  clone https://github.com/hyperledger/fabric-samples.git

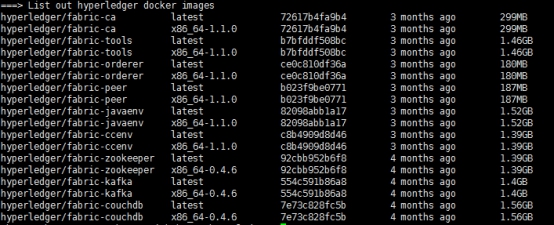
cd  fabric-samples/first-network

使用curl命令进行下载，之前确保外网可访问

sudo  curl **-**sSL https:**//**goo**.**gl**/**6wtTN5 **|** bash **-**s 1.1**.**0

***注：如果上述操作有问题，可使用脚本直接下载镜像***

下载后如图所示



镜像下载完成后，会产生bin文件，下面有工具，将此工具的绝对路径添加到环境变量中

export PATH=下载位置的绝对路径/bin:$PATH

[export PATH=/home/ubuntu/gopath/src/github.com/hyperledger/fabric-samples/bin:$PATH]

**8.3 Fabric组网部署及过程解读**

执行脚本./byfn.sh  up会根据Docker Compose配置文件docker-compose-cli.yaml启动超级账本网络， 还会执行scripts/script.sh脚本安装和实例化链码， 并且执行简单链码调用和查询操作。

执行script.sh脚本时，会执行：创建通道，加入通道，更新通道信息，安装链码，实例化链码，链码调用，链码查询

创建通道

byfn.sh  generate  [根据crypto-config.yaml生成peer,order的MSP证书]

                  [根据configtx.yaml生成创世区块]

**a. 使用cryptogen工具生成证书**

命令：

|  |
| --- |
| # cryptogen generate --config=./crypto-config.yaml org1.example.com org2.example.com |

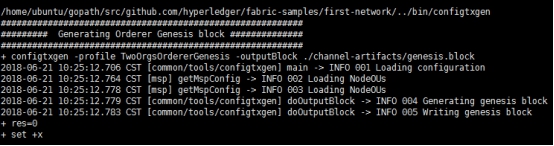
**b. 生成排序节点的初始区块**

执行命令：

|  |
| --- |
| # configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.blcok  -profile 是根据configtx.yaml中的Profiles配置项描述如何生成组织：默认kafka  -outputBlock 指定创世区块的目录文件和名称 |

此处代码调用过程：

common/tools/configtxgen/main.go:main()->msp/configbuilder.go：getMspConfig()->common/tools/configtxgen/main.go:doOutputBlock()



注：生成的初始区块中的信息包括：

排序节点信息和排序节点的MSP信息[管理员证书，根证书，TLS证书]

组织信息和组织的MSP信息[管理员证书，根证书，TLS证书]

共识的算法类型

区块配置信息

访问控制策略

通过使用工具configtxgen可以查看genession.block的信息

genession.block解析

data,header,metedate等信息，data中存有channel\_group，其中定义了组织及组织的策略信息

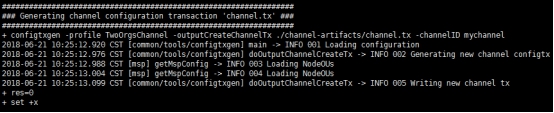
|  |
| --- |
| "Org1MSP": {  "mod\_policy": "Admins",  "policies": {  "Admins": {  "mod\_policy": "Admins",  "policy": {  "identities": [{  "principal": {  "msp\_identifier": "Org1MSP",  "role": "ADMIN"  },  "principal\_classification": "ROLE"  }],  "rule": {  "n\_out\_of": {"n": 1,"rules": [{"signed\_by": 0  "**Readers**": {  "mod\_policy": "Admins","policy": {"value": {"identities": [{  "principal": {"msp\_identifier": "Org1MSP","role": "MEMBER"},  "principal\_classification": "ROLE"  }],  "rule": { //NOutOf指的是满足m个条件中的n个就表示满足策略（m≥n）  "n\_out\_of": {"n": 1,"rules": [{"signed\_by": 0},  "**Writers**": {},  "values": {  "MSP": {  "mod\_policy": "Admins",  "crypto\_config": {  "identity\_identifier\_hash\_function": "SHA256",  "signature\_hash\_family": "SHA2"},  "name": "Org1MSP",  "root\_certs": [],  "tls\_root\_certs": []  "ConsensusType": {  "mod\_policy": "Admins",  "value": {  "type": "solo"}, |

**c. 生成通道交易配置文件**

执行命令：

|  |
| --- |
| #configtxgen  -profile TwoOrgsChannel  -outputCreateChannelTx  ./channel-artifacts/channel.tx -channelID mychannel |

此处代码调用过程：common/tools/configtxgen/main.go:main()->common/tools/configtxgen/main.go:doOutputChannelCreateTx()->msp/configbuilder.go:getMspConfig()->common/tools/configtxgen/main.go:doOutputChannelCreateTx()

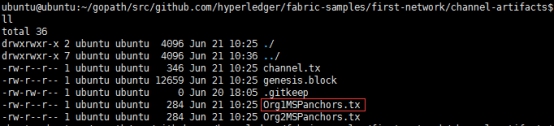


**d. 生成Org1MSP/Org2MSP锚节点配置**

命令：

|  |
| --- |
| #configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx -channelID mychannel -asOrg Org1MSP  #configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx -channelID mychannel -asOrg Org2MSP -outputAnchorPeersUpdate:创建在configtx.yaml中指定的AnchorPeers |

此处代码调用过程：common/tools/configtxgen/main.go:main()->doOutputAnchorPeersUpdate()



生成的channel.tx和Org1MSPanchors.tx的结构类似

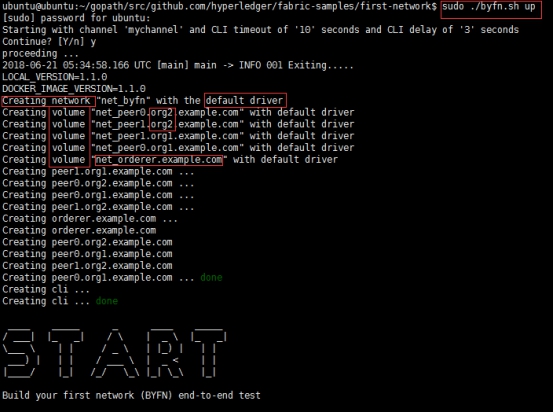
|  |
| --- |
| {  "payload": {  "data": {  "config\_update": {  "channel\_id": "onechannel",  "read\_set": {  "groups": {"Application": {"groups": {"Org2MSP": {"mod\_policy": "",  "policies": {  Admins": {"mod\_policy": "","version": "0"},"  Readers": {"mod\_policy": "","version": "0"},"  Writers": {mod\_policy": "","version": "0"}},"mod\_policy": "Admins",},  "type": 0,  "write\_set": {  "AnchorPeers": {  "mod\_policy": "Admins",  "anchor\_peers": [{  "host":"peer0.org2.example.com","port": 7051  },  "MSP": {"mod\_policy": "",},  "header": {  "channel\_header": {  "channel\_id": "onechannel",  } |

**e. 启动网络**

启动网络时，由于docker权限的影响，所以此时运行时

sudo ./byfn.sh up.

开始执行脚本后，首先创建网络 net\_byfn



后续过程都基于fabric-1.1.0\examples\e2e\_cli\scripts\script.sh基本调用

**【安装过程中可以通过docker ps -a 查看容器中程序运行；安装chaincode后，通过docker image查看链码生成的镜像】**

**f. 启动完成后开始创建channel**

条件：初始化时生成的根证书，证书，私钥，MSP等信息，及通道配置交易文件创建通道

执行命令：

|  |
| --- |
| #peer channel create -o orderer.example.com:7050 -c mychannel -f ./channel-artifacts/channel.tx --tls true --cafile /opt/gopath.../tlsca.example.com-cert.pem |

此处代码调用过程：

peer/channel/channel.go的InitCmdFactory(),初始化Endorser和Orderer节点

channel创建完成后，Org1的peer1、peer2与Org2的peer1、peer2加入mychannel

**g. 条件：利用上面的根证书、证书、私钥、MSP等信息，加入channel**

调用命令：

|  |
| --- |
| #peer channel join -b mychannel.block |

此处代码调用过程：

peer/channel/join.go的executeJoin(cf \*ChannelCmdFactory)方法

**h. 更新Org1和Org2的anchor peer**

调用命令：

|  |
| --- |
| #peer channel update -o org1.example.com:7050 -c mychannel -f ./channel-artifacts/Org1MSPanchor.tx --tls true --cafile /opt/.../tlsca.example.com-cert.pem |

此处代码调用过程：peer/main.go:channel.Cmd()->peer/channel/channel.go:updateCmd()->update()

--tls true ：开启TLS验证； --cafile ：指定证书为orderer的tlsca证书

**i. 给Org1、Org2的peer0安装链码**

调用命令：

|  |
| --- |
| #peer chaincode install -n mycc -v 1.0 -l golang -p github.com/chaincode/chaincode\_example02/go |

此处代码调用过程：peer/chaincode/chaincode.go:Cmd()->installCmd()->genChaincodeDeploymentSpec()->getChaincodeSpec()->checkChaincodeCmdParams()

**j. emcc和vscc没有定义，使用默认配置进行安装**

对Org2的peer0进行链码初始化

调用命令：

|  |
| --- |
| #peer chaincode instantiate -o orderer.example.com:7050 --tls true --cafile /opt/.../tlsca.example.com-cert.pem -C mychannel -n mycc -l golang -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P 'OR ('\''Org1MSP.peer'\'','\''Org2MSP.peer'\'')' |

此处代码调用过程：peer/chaincode/chaincode.go:Cmd()->instantiateCmd()->genChaincodeDeploymentSpec()->getChaincodeSpec()->checkChaincodeCmdParams()

**k.emcc和vscc没有定义，使用默认配置进行初始化**

查询Org1的peer0信息

通过初始化Org2的peer0，查询Org1的peer0获取链码结果

调用命令：

|  |
| --- |
| #peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]} |

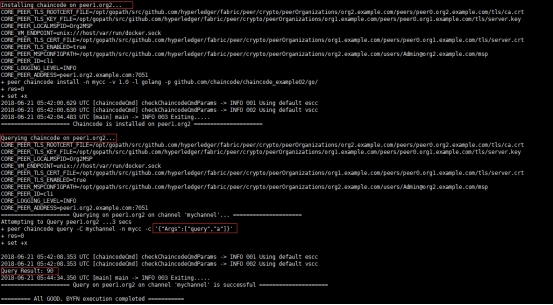
l.进行交易

调用命令：

|  |
| --- |
| #peer chaincode invoke -o orderer.example.com:7050 --tls true --cafile /opt/.../tlsca.example.com-cert.pem -C mychannel -n mycc -c '{"Args":["invoke","a","b","10"]}' |

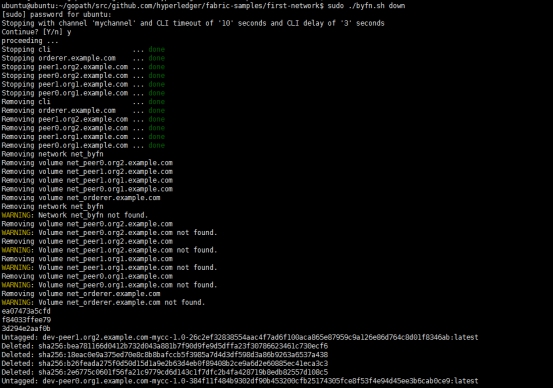
m.网络测试

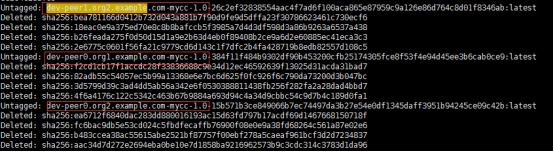
通过安装Org2.peer1链码，在Org2.peer1上查询a的值，得到正确的结果，说明安装链码时交易信息自动同步，网络正常



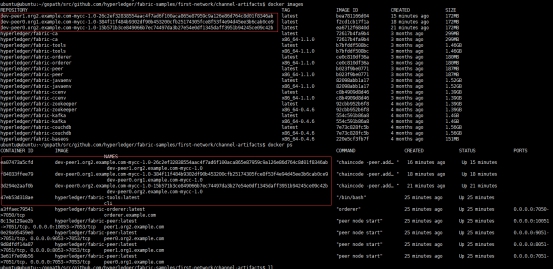
n.关闭网络

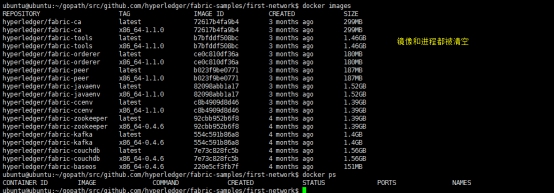
可以看到关闭peer节点，排序服务节点容器，链码容器，删除自动生成的镜像文件。





运行./byfn.sh up 后的docker显示信息与./byfn.sh down 后的docker显示信息





**9. Fabric智能合约解读**

**9.1 智能合约特点**

I. 无状态，事件驱动的代码。

II. 智能合约可以操作账本状态，账本状态记录着与业务相关重要数据。

III. 应用程序通过向区块链网络发送交易来调用智能合约。

IV. 多个智能合约时，应用程序通过名称、版本号来确定调用哪个智能合约。

**9.2 智能合约的一次完整过程**

I. 从CA获取合法的身份证书

II. 构造合法的交易Proposal后提交Endorser进行背书

III. 收集足够多的背书后，构造合法交易请求发送给order节点进行排序

IV. 监听事件，确保交易已经写入账本

**9.3 Fabric链码生命周期**

**9.3.1 应用链码的生命周期**

应用链码的生命周期：打包、安装、实例化、升级

a. 打包

a.1链码创建:

#peer chaincode package -n mycc -p github.com/hyperledger/.../chaincode\_example02 -v 0 -s -S -i "AND('OrgA.admin')" ccpack.out

注：s：多人创建  -i创建策略[OrgA.admin创建]

链码创建使用签名策略：版本策略、角色策略、主体策略[SignaturePolicyEnvelope]

签名策略[多人签名、一人签名]==背书策略

a.2 链码打包签名

#peer chaincode signpackage ccpack.out signedccpack.out

签名后包含有本地MSP的签名

b. 安装

#peer chaincode install -n mycc -v 1.0 -l golang -p github.com/chaincode/chaincode\_example02/go

指定名称，版本号，路径

c. 实例化

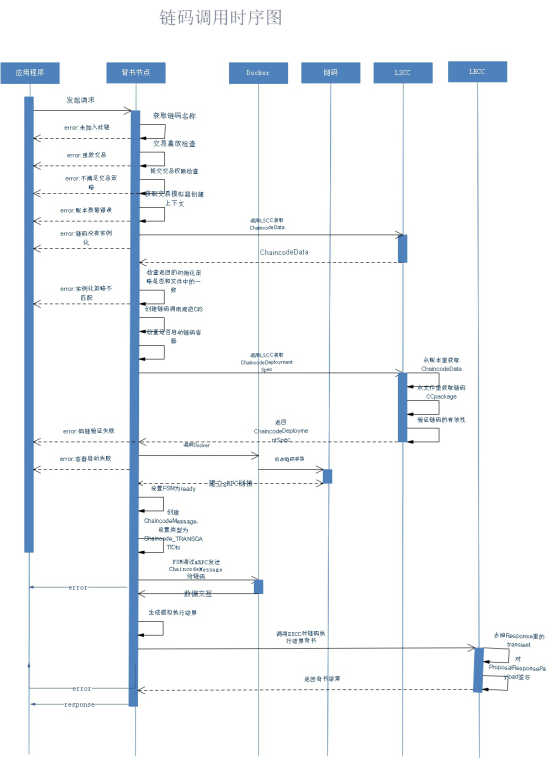
链码实例化的时候会进行实例化策略检查和通道写入策略见检查：实例化策略检查指实例化交易签名是否符合策略规则；通道写入策略指创建通道时根据configtx.yaml文件指定的规则

#peer chaincode instantiate -n ChaincodeName -v version -c '{"Args":["john","0"]}' -P "OR ('Org1.member','Org2.member')"

p：指定背书策略

d. 升级

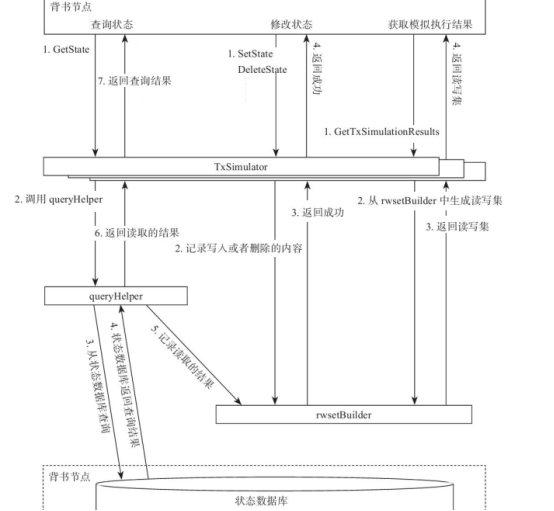
链码的升级和链码的初始化类似，绑定链码到通道上，执行初始化操作



代码调用

|  |
| --- |
| **func** (c \*endorserClient) ProcessProposal(ctx context.Context, in \*SignedProposal, opts ...grpc.CallOption) (\*ProposalResponse, error) {     out := new(ProposalResponse)  //gRPC调用时的path:/{package}.{ServiceName}/{RpcMethodName}     err := grpc.Invoke(ctx, **"/protos.Endorser/ProcessProposal"**, in, out, c.cc, opts...) |





背书节点侧的有限状态机记录不同交易号调用时的上下文信息，根据交易号从交易上下文映射表中获取交易的上下文信息，进而利用交易模拟生成器获取模拟结果。

|  |
| --- |
| **type** VM **interface** {     Deploy(ctxt context.Context, ccid ccintf.CCID, args []string, env []string, reader io.Reader) error     Start(ctxt context.Context, ccid ccintf.CCID, args []string, 。。。) error     Stop(ctxt context.Context, ccid ccintf.CCID, timeout uint, dontkill bool, dontremove bool) error     Destroy(ctxt context.Context, ccid ccintf.CCID, force bool, noprune bool) error     GetVMName(ccID ccintf.CCID, format **func**(string) (string, error)) (string, error)  } |

fabric中通过插件vender/fsouza/go-dockerclient模块与docker进行交互，当启动.byfn.sh,会调用script.sh，script.sh会对环境中的docker进程进行检查

具体交互过程所用命令如下：

|  |  |  |
| --- | --- | --- |
| **接口** | **go-dockerclient的接口** | **Docker Engine API** |
| Deploy | Client.BuildImage | POST/build |
| Destroy | Client.RemoveImageExtended | DELETE/images/:id |
| Start | Client.CreateContainer | POST/containers/create |
| Stop | Client.KillContainer | POST/containers/:id/kill |

**9.3.2 系统链码**

系统链码区别与用户链码：指的是Fabric Peer中负责系统配置、 背书、 验证等平台功能的逻辑， 运行在Peer进程内。

系统链码与用户链码的异同

|  |  |  |
| --- | --- | --- |
| **对比项** | **系统链码** | **普通链码** |
| 链码源码 | 无main() | 有main() |
| 运行空间 | 背书节点进程 | Docker进程 |
| 调用方式 | 网络+进程内部 | 网络 |
| 启动参数 | 内置 | 动态输入 |
| 通信方式 | Golang的通道机制 | 网络 |
| 数据存取 | Golang的通道+本地文件 | 网络 |
| 升级方式 | 和背书节点一起升级 | 单独升级 |
| 背书策略 | 无 | 有 |

内置的系统链码和属性列表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **链码名称** | **功能** | **是否可外部调用** | **是否可被其他链码调用** | **是否可被代替** |
| 生命周期系统链码LSCC | 管理部署在背书节点的链码 | 是 | 是 | 否 |
| 配置管理系统链码CSCC | 管理记账节点上的信息 | 是 | 否 | 否 |
| 查询系统链码QSCC | 查询记账节点上的账本数据 | 是 | 是 | 否 |
| 交易背书系统链码ESCC | 对交易结果进行结构转换和背书签名 | 否 | 否 | 是 |
| 验证系统链码VSCC | 记账前对交易和区块进行验证 | 否 | 否 | 是 |

**9.3.2.1 系统链码生命周期LSCC**

LSCC的主要功能是管理部署在背书节点上的链码，并不是全生命周期管理。

链码的生命周期包括：链码的安装，部署，升级，查询

链码安装、部署、升级时的结构基本一致，都有signature，其中包含有creator对整个proposal的签名，能验证消息的完整性，起到防抵赖的作用。

**9.3.2.2 系统链码配置管理CSCC**

CSCC的主要功能管理记账节点上的配置信息。

节点加入区块时，会根据初始区块中的配置信息来判定是否可以加入，包括类型检查、配置检查和全县检查等。

通过CSCC可以查询通道的配置区块和可加入的通道链表

**9.3.2.3 系统链码查询(QSCC)，交易背书(ESCC)，及验证(VSCC)**

QSCC：提供查询记账节点的账本数据，包括区块，交易 ，区块链等信息。

ESCC：对交易进行的结果进行结构转换和签名背书。

VSCC：记账前对交易和区块进行验证。

**9.4智能合约的链码实现**

**9.4.1实现链码所需接口**

|  |
| --- |
| **type** Chaincode **interface** {  //链码收到实例化或升级的交易时调用  Init(stub ChaincodeStubInterface) pb.Response  //链码收到调用或查询的交易调用   Invoke(stub ChaincodeStubInterface) pb.Response  } |

**9.4.2 实现链码结构**

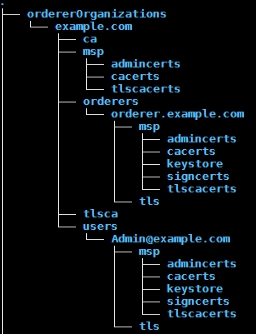
|  |
| --- |
| **import** (  **"../shim"**pb **"../../../protos/peer"   "fmt"**  )  *//声明一个计算结构体*  **type** Computer **struct**{}  *//为结构体添加Init方法*  **func** (c \*Computer) Init(stub shim.ChaincodeStubInterface) pb.Response {  *//链码初始化或升级时调用***return**  }  **func** (c \*Computer) Invoke(stub shim.ChaincodeStubInterface) pb.Response {  *//链码被调用和查询***return**  }  **func** main() {  *//調用shim的start方法*  err := shim.Start(new (Computer))  **if**(err == nil){         fmt.Println(err)     }  } |

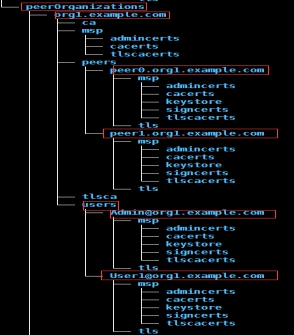
**9.5案例解读**

|  |
| --- |
| **type** SimpleChaincode **struct** {}  //定义结构体  **func** (t \*SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {    stub.GetFunctionAndParameters() //获取方法名和参数  stub.PutState(A, []byte(strconv.Itoa(Aval)))//初始化链    stub.PutState(B, []byte(strconv.Itoa(Bval)))//写入链码到账本  //调用  **func** (t \*SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {  stub.GetFunctionAndParameters()  **if** function == **"invoke"** {//A与B间的交易调用  **return** t.invoke(stub, args)     } **else if** function == **"delete"** {//删除某个交易  **return** t.delete(stub, args)     } **else if** function == **"query"** {//查询某个交易  **return** t.query(stub, args)     }  *// A,B 间进行交易*  **func** (t \*SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response {     stub.GetState(A) //获取A状态     stub.GetState(B) //获取B状态     Aval = Aval - X   Bval = Bval + X     stub.PutState(A, []byte(strconv.Itoa(Aval)))//设置A状态     stub.PutState(B, []byte(strconv.Itoa(Bval)))//设置B状态}  *// 从账本中删除某个交易*  **func** (t \*SimpleChaincode) delete(stub shim.ChaincodeStubInterface, args []string) pb.Response {stub.DelState(A)  **return** shim.Success(nil)}  *// 查询交易*  **func** (t \*SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string) pb.Response {    stub.GetState(A)}  **func** main() {   shim.Start(new(SimpleChaincode))//启动链码} |

**9.5.1 链码初始化证书与初始区块生成**

链码初始化的时候，会根据crypto-config.yaml配置文件获取CA和CSP等信息，根据信息创建FabricClient并设置加密套件cryptosuilt与存储KVStore，然后根据配置信息和组织信息创建FabricCAClient，使用注册证书和私钥信息发送请求给FabricCAClient进行注册，fabric-ca验证请求后返回用户注册的Secert给用户完成注册。用户使用Secert调用fabric-ca的Enroll生成注册证书和私钥。





证书生成后，使用证书和配置文件中的组织信息TwoOrgsOrdererGenesis生成初始区块。

**9.5.2 链码通道生成**

由configtxgen工具根据crypto-config.yaml中的TwoOrgsChannel及指定的通道名称，通道配置文件路径生成通道配置文件mychannel.tx。

**9.5.3 生成各组织内的锚节点配置文件**

由configtxgen工具根据crypto-config.yaml中的TwoOrgsChannel及指定的通道名称，及指定的组织生成锚节点的配置文件。

**9.5.4 节点初始化完成后加入通道**

应用程序调用GenesisBlock向order节点获取初始区块信息，使用初始区块信息构造加入通道请求，链码根据请求构造加入通道的交易背书提案，并使用用户信息进行签名生成signatureProposal，节点根据生成的signatureProposal调用CSCC进行相应的消息有效性检查和权限检查，检查通过后创建对应通道的本地账本，然后通过gossip服务从排序节点同步最新的区块数据。

**9.5.6 节点的链码安装**

加入通道后会对链码进行安装，安装的过程中指定链码名称，版本号，链码语言，路径等信息安装。

**9.5.7 节点的链码初始化**

链码安装完成后会对其进行初始化操作，初始化的时候通过构造包含ChaincodeDeploymentSpec的ChaincodeInvocationSpec信息，并使用通道的用户进行签名生成signatureProposal，然后使用创建的gRPC链接发送请求到背书节点，背书节点验证通过后调用LSCC的deploy操作，部署完成后将部署响应信息提交给排序服务节点生成最终的区块完成最终部署。

例子中的方法Init，需要执行初始A与B的值

**9.5.8 节点的更新交易，查询交易，删除交易操作**

链码的查询，更新，删除等操作和初始化过程类似，不同点：

1. 实例化链码交易调用的方法是：Init()；普通交易调用的是Invoke();

2. 普通的交易请求是不嵌套ChaincodeInvocationSpec请求，包含通道的名称，函数和参数等，如：{"Args":["query","a"]}/{"Args":["invoke","a","b","10"]}等。

3. 初始化请求由于要启动容器，所以执行时间长；普通交易调用由于容器已经开启，所以速度快。