区块链 hyperledger 开发实战学习

对应的学习博客:

网络环境配置: https://blog.csdn.net/Box_clf/article/details/82534469

运行测试用例: https://blog.csdn.net/Box_clf/article/details/82683417

常见问题解决: https://blog.csdn.net/Box_clf/article/details/82588062

对应的 SDK 学习项目地址: (选择分支 drug-trace-system-for-local)

https://github.com/kvenLin/drug-trace-system.git

项目学习网络完整环境地址:

https://github.com/kvenLin/fabric1.0.git

1.区块链基础

1.1 理论概述

1.区块链的类型

公有链:任何人都可以加入,以太坊

联盟链:所有节点不是随意加入退出的,必须要有准入控制

2.区块链特点

去中心化,

去信任化,

数据共享,

不可篡改:不同于不可修改,指发起一笔交易后不能单方面撤销交易,如果需要撤销之前的交易必须再花几笔交易告诉全网我需要撤销之前的交易.

3.区块链平台

比特币

以太坊(区块链 2.0,将智能合约引入区块链中)　　注:区块链 1.0 是数字货币

EOS(区块链 3.0),企业级区块链操作系统

超级账本(Fabric,hyperledger 中最主要的一个项目)

4.应用场景

去信任(去中介)

价值转移(非拷贝)

数据共享

1.2 区块链技术

Hyperledger Fabric ---> gRPC 对等服务

密码学—哈希(HASH)

将指定长度的数据转换成固定的哈希值

MD5,SHA1,SHA2(SHA2-256);MD5 和 SHA1 已经被证明不安全,可进行反推

正向快速,逆向困难

输入敏感

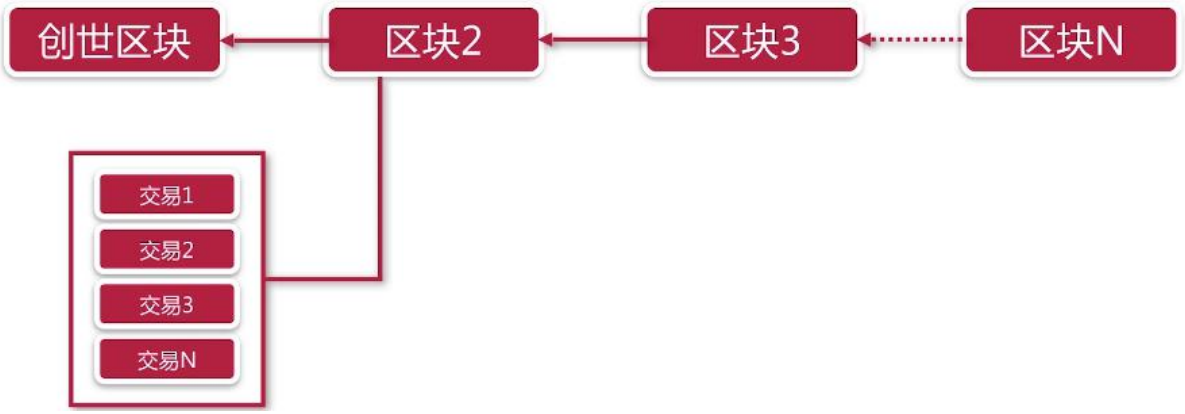
抗碰撞

加解密算法

对称加密:密钥相同,易泄漏(AES,DES 等)

非对称加密:公钥/私钥,效率低(RSA,椭圆曲线算法,区块链中就应用的该方法)

账本结构



一系列有序的不可篡改的状态转移记录日志(交易)

区块头包含所有交易的 hash,同时也包含前一个区块的 hash 这样账本中的所有交易就被有序的存储,

有序的理解:区块有序,区块里面的交易是有序的.

如果本地节点的 hash 值和别的节点中记录的 hash 值不一样就会被认为是不一样的,将被排除在网络之外

共识算法(分布式一致性算法)

强一致性

最终一致性

CAP 原理:一个分布式系统不可能同时满足一致性,可用性,分区容忍性,只能三者取其二

ACID 原理:原子性,一致性,隔离性,持久性

现有共识算法系列:

(一般的共识)Paxos:分布式系统中只有故障节点(宕机,网络中断)而没有恶意节点(故意制造错误信息的节点)

(区块链的共识)拜占庭容错:能容纳故障节点也能容纳恶意节点(参考拜占庭将军问题)

联盟链和公有链的不同在共识算法的选择上导致的

共识算法（分布式一致性算法）

	PoW	DPoS/PoS	PBFT	Raft
场景	公链	公链	联盟链	联盟链
去中心化	完全	完全	多中心	多中心
响应时间	10分钟	1分钟内	秒级	秒级
容错	50%	50%	33%	50%

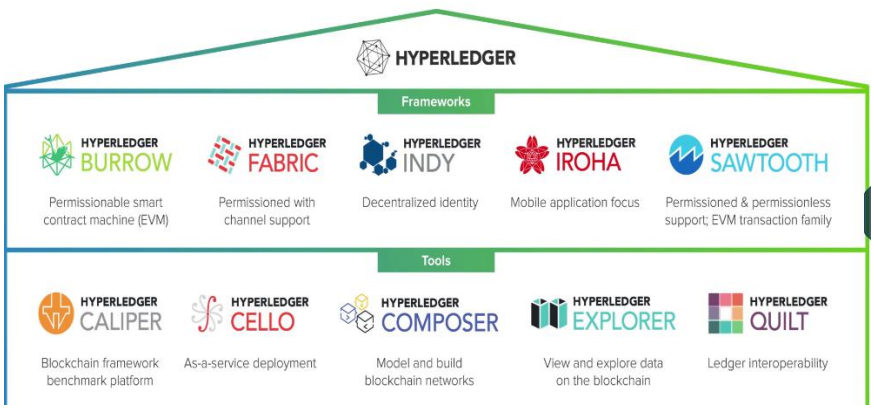
待读书籍

拓展阅读

- 《比特币白皮书一种点对点的电子现金系统》 中本聪
- 《区块链技术指南》 杨宝华yeasy
- Hyperledger Fabric官方文档

2.Hyperledger

开源项目



2.1 Fabric

项目简介

IBM 发起,Linux 基金会托管

企业级联盟链基础设施

可插拔的共识机制(solo,kafka…)

多链多通道隔离

注:solo 模式中只有一个排序服务的节点,整个网络的交易顺序就是节点收到消息的顺序,

而在 kafka 模式中,使用 kafka 集群对交易进行排序,kafka 消息队列系统,保证消息的一致性

智能合约

区块链 2.0:以太坊

合约协议的数字化代码表达(将平常的合同用代码的形式表达)

分布式有限状态机

执行环境安全隔离,不受第三方干扰(EVM,Docker)

链码

Fabric 应用层基石(中间件)

编程接口

Init()　链码初始化操作

Invoke()　链码交互的入口,所有链码的业务逻辑都是通过这个接口进行调用

环境准备

Fabric 的 docker 镜像

源码库版本切换到--> release1.0

cryptogen(生成相关的证书),configtxgen(生成创世区块和通道配置)工具编译

下载官方例子 fabric-samples:https://github.com/hyperledger/fabric-samples

第一个 Fabric 网络

1.byfn.sh -m generate

2.byfn.sh -m up

3.byfn.sh -m down

2.2Fabirc 源码项目

准备工作

1.使用 git 下载 Fabric 源码

(注:源码都下载在$GOPATH/src/github.com/hyperledger/下)

```
hk@chailinfeng:~/go/src/github.com/hyperledger$ ls
fabric  fabric-samples
```

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

2.切换到版本 release1.0

git checkout release-1.0

3.进入指定目录:

cd $GOPATH/src/github.com/hyperledger/fabric/common/configtx/tool/configtxgen/

4.安装 configtxgen 工具:

go install --tags=nopkcs11

5.同 3.4 步操作,进入指定目录进行安装工具

cd common/tools/cryptogen/

go install --tags=nopkcs11

此时两个工具都被安装到了 GOPATH 的 bin 目录下

```
hk@chailinfeng:~/go/bin$ ls
configtxgen  cryptogen
```

搭建 samples 网络

1.进入 samples 的项目目录

cd $GOPATH/src/github.com/hyperledger/fabric-samples/

2.cd fist-network

3.查看 byfn 的帮助: ./byfn.sh -h

```
Typically, one would first generate the required certificates and
genesis block, then bring up the network. e.g.:

        byfn.sh -m generate -c mychannel
        byfn.sh -m up -c mychannel -s couchdb
        byfn.sh -m up -c mychannel -s couchdb -i 1.0.6
        byfn.sh -m down -c mychannel

Taking all defaults:
        byfn.sh -m generate
        byfn.sh -m up
        byfn.sh -m down
```

4. 创建一个网络通道名为 imooc:　　./byfn.sh -m generate -c imooc

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric-samples/first-network$ ./byfn.sh -m generate -c imooc

Generating certs and genesis block for with channel 'imooc' and CLI timeout of '10'
Continue (y/n)? y
proceeding ...
/home/hk/go/src/github.com/hyperledger/fabric-samples/first-network/../bin/cryptogen

############################################################
##### Generate certificates using cryptogen tool #########
############################################################
org1.example.com
org2.example.com
```

5.启动网络:　　./byfn.sh -m up -c imooc

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric-samples/first-network$ ./byfn.sh -m up -c imooc
Starting with channel 'imooc' and CLI timeout of '10'
Continue (y/n)? y
proceeding ...
Creating network "net_byfn" with the default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating volume "net_peer1.org2.example.com" with default driver
Creating volume "net_peer1.org1.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_orderer.example.com" with default driver
Creating peer0.org2.example.com
```

如果出现下图则说明启动成功:

```
18-09-17 06:34:53.040 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using defa
18-09-17 06:34:53.041 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A90070A6208
18-09-17 06:34:53.041 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: A520947E19BBC76
ery Result: 90
18-09-17 06:35:32.691 UTC [main] main -> INFO 007 Exiting.....
=================== Query on PEER3 on channel 'imooc' is successful ===================

======= All GOOD, BYFN execution completed ===========


 ____    _   _   ____
|___ \  | \ | | |  _ \
  _|    |  \| | | | | |
 |___  | |\  | | |_| |
 |____| |_| \_| |____/
```

搭建 samples 错误解决

错误 1:

fabric-simples 提示 cryptogen tool not found.

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric-samples/first-network$ ./byfn.sh -m generate -c imooc
Generating certs and genesis block for with channel 'imooc' and CLI timeout of '10'
Continue (y/n)? y
proceeding ...
cryptogen tool not found. exiting
```

解决方案:说明之前生成的两个工具没有生效,需要手动创建文件夹添加工具

1.如果在 fabric-samples 文件夹下没有 bin 文件夹则手动添加 bin 文件夹

2.切换到$GOPATH/src/github.com/hyperledger/fabric/build/bin/目录下,即将 fabric 源码项目下 bin 文件夹里面的工具拷贝到 samples 的 bin 中

3.再次使用

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric-samples/first-network$ ./byfn.sh -m generate imooc
Generating certs and genesis block for with channel 'mychannel' and CLI timeout of '10'
Continue (y/n)? y
proceeding ...
/home/hk/go/src/github.com/hyperledger/fabric-samples/first-network/../bin/cryptogen

############################################################
##### Generate certificates using cryptogen tool #########
############################################################
org1.example.com
org2.example.com
```

错误 2:

经常启动出现下面的提示 BAD_REQUEST

```
018-09-17 06:00:30.315 UTC [msp] GetDefaultSigningIdentity -> DEBU 00d Obtaining default signing identity
018-09-17 06:00:30.315 UTC [msp/identity] Sign -> DEBU 00e Sign: plaintext: 0ABF060A1108021A0608FE85FDDC0522...0136AF68817D2F5E4EDFE51EB740992F
018-09-17 06:00:30.315 UTC [msp/identity] Sign -> DEBU 00f Sign: digest: 9C1979F6F10050F52FB7C3D753F162276CE03C2A648C74414674C47417F3AFEC
rror: Got unexpected status: BAD_REQUEST
sage:
 peer channel create [flags]

lags:
 -c, --channelID string   In case of a newChain command, the channel ID to create.
 -f, --file string        Configuration transaction file generated by a tool such as configtxgen for submitting to orderer
 -t, --timeout int        Channel creation timeout (default 5)

lobal Flags:
     --cafile string              Path to file containing PEM-encoded trusted certificate(s) for the ordering endpoint
     --logging-level string       Default logging level and overrides, see core.yaml for full syntax
 -o, --orderer string             Ordering service endpoint
     --test.coverprofile string   Done (default "coverage.cov")
     --tls                        Use TLS when communicating with the orderer endpoint
 -v, --version                    Display current version of fabric peer server

!!!!!!!!!!!!!! Channel creation failed !!!!!!!!!!!!!!!!
======== ERROR !!! FAILED to execute End-2-End Scenario ===========
```

为避免重复创建镜像容器而导致错误可以在每次启动前执行下面的命令进行清理:

./byfn.sh -m down

samples 启动日志讲解

出现第一个 start 表示 fabric 网络已经启动完成,后续执行 scripts 目录下的脚本

```
Creating network "net_byfn" with the default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating volume "net_peer1.org2.example.com" with default driver
Creating volume "net_peer1.org1.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_orderer.example.com" with default driver
Creating peer0.org1.example.com
Creating peer1.org1.example.com
Creating peer1.org2.example.com
Creating peer0.org2.example.com
Creating orderer.example.com
Creating cli

 ____    _____      _      ____    _____
/ ___|  |_   _|    / \    |  _ \  |_   _|
\___ \    | |     / _ \   | |_) |   | |
 ___) |   | |    / ___ \  |  _ <    | |
|____/    |_|   /_/   \_\ |_| \_\   |_|
```

脚本内容包含:创建一些通道,加入通道,更新通道锚节点的信息,以及安装链码,实例化链码,做交互

2.3 imooc 中 Fabric 项目源码获取

1.首先从该从该网盘中获取源码压缩文件

https://pan.baidu.com/s/1IUE-iDip88gDeck-6-Raag

2.解压文件后得到源码文件夹

3.在 home 路径下创建新的 GOPATH 项目路径:

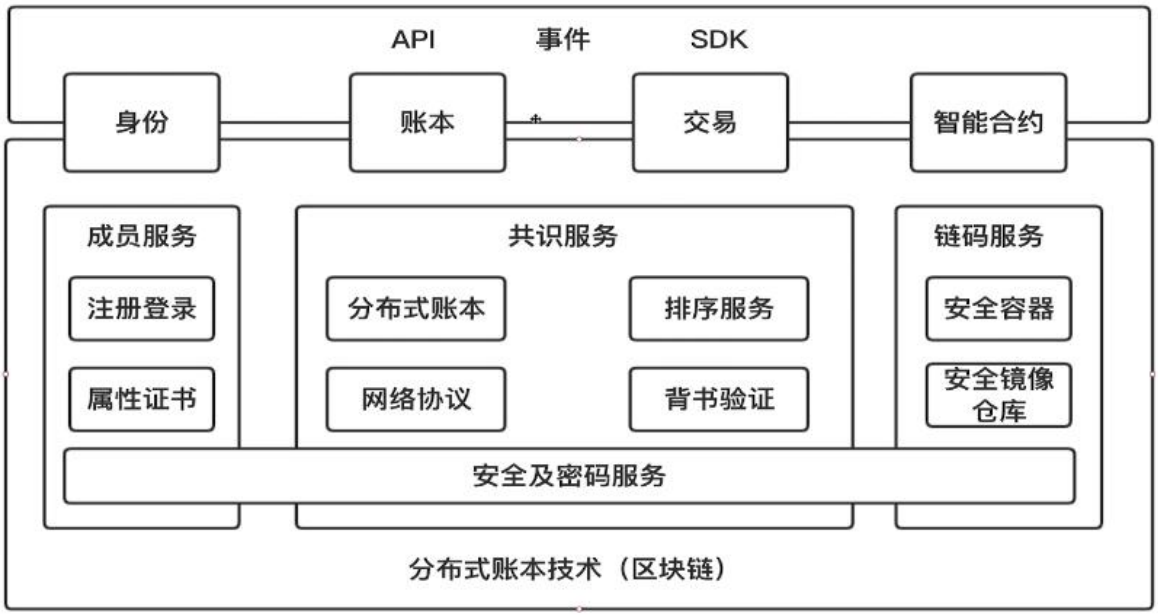
mkdir -p imooc/go/src/github.com/hyperledger/fabric/

更改 GOPATH 环境变量:

export GOPATH=$HOME/imooc/go/

4.解压后的源码 coding-268 复制到 imooc/go/src/github.com/hyperledger/fabric/下

3.系统架构



3.1 系统服务

交易具体流程:

1.客户端节点发起向背书节点发起一个交易提案

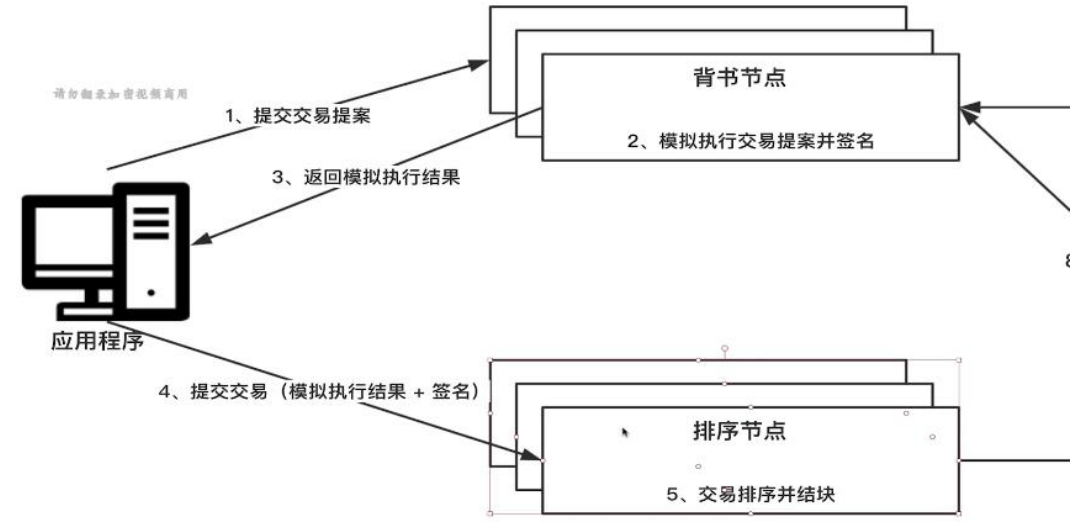
2.随后背书节点模拟交易(不会产生任何持久化的操作)返回给客户端结果以及签名

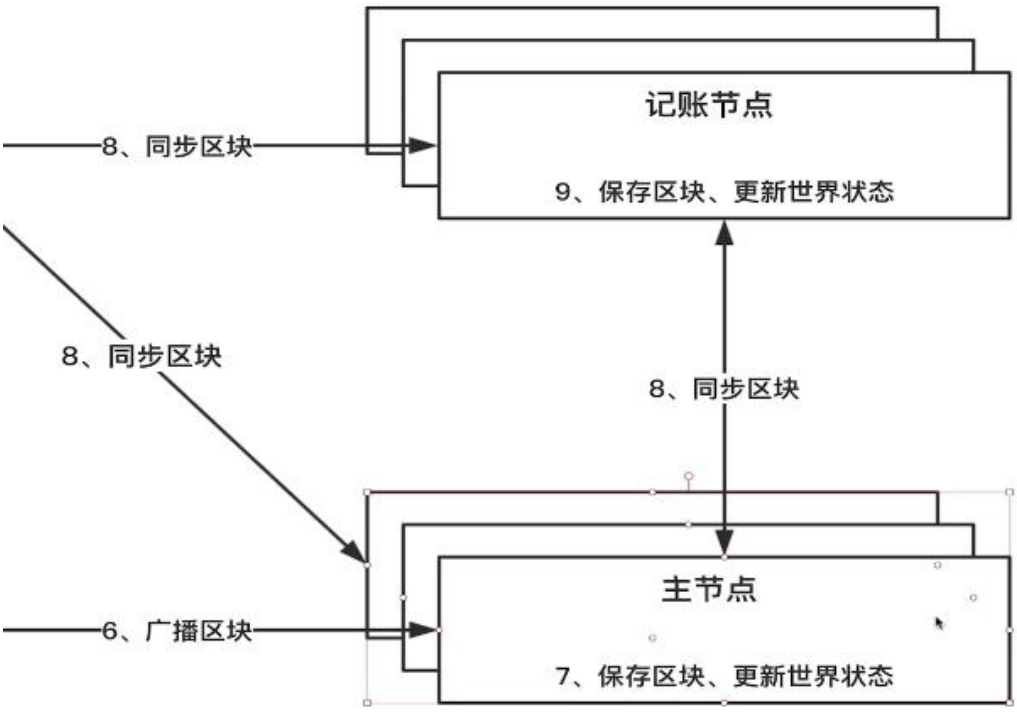
3.随后客户端将背书后的交易交给排序节点进行排序

4.随后由排序节点生成区块然后向全网进行广播

5.计算节点收到这些广播以后先验证交易的正确性,如果都验证通过了就存入本地的账本中

这中间的过程用到了两种通讯协议





排序节点与组织的锚节点使用的是 gRPC 进行通讯

而在组织内部使用的是 Gossip 协议进行区块扩散(提高了区块的传输效率)

3.2 整体架构

网络拓扑结构

客户端节点(应用程序/SDK/命令工具)　一个应用程序不可或缺的节点

介于应用程序和底层之间,两者交互的媒介节点,不能独立存在,必须与 Orderer 节点和 Peer 节点建立连接才能发挥作用

Peer 节点(包含:Anchor/Endorser/Committer)

Anchor 为锚节点/组节点:一个组织内部只有一个锚节点,是一个组织内部唯一与 Orderer 进行通信的节点

Endorser 为背书节点:(背书可以简单了理解为担保的意思),不是一个固定的节点类型,是与智能合约进行绑定的;节点被安装到区块链的时候都会设置专有的背书策略,同时指定智能合约的交易由哪些背书节点背书以后才是有效的(例:如果智能合约要求至少两个背书节点进行背书,则交易提案就必须发送给两给背书节点进行背书).也就是说只有在背书节点上才能运行智能合约.

Committer 为记账节点:所有的 Peer 节点都是记账节点,主要功能验证从 Orderer 接收的区块,验证区块的有效性以及交易的有效性,验证完以后记录到本地的账本中.交易有效则更改区块链上的状态数据.

Orderer 排序节点(solo,kafka)

功能:

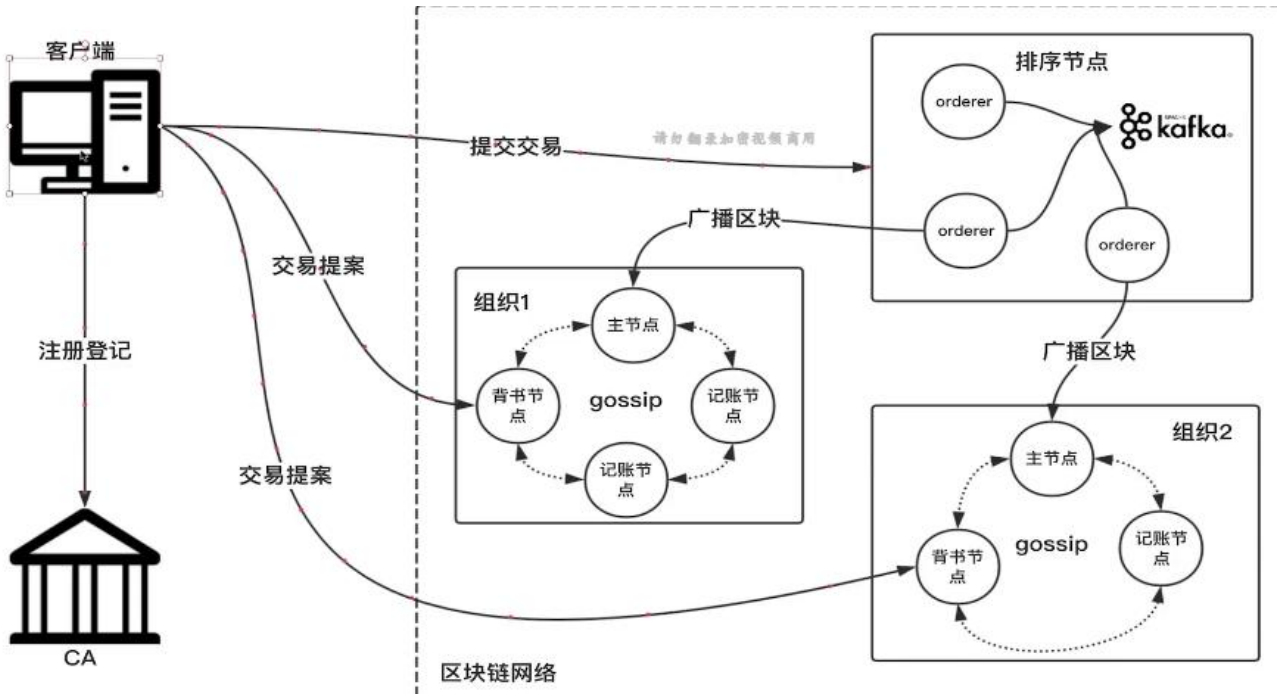
1.从全网中接收交易然后按照一定的规则进行排序

2.将排序号的交易按照固定的时间间隔打包成区块分发给其他组织的组节点(Anchor)

CA 节点(可选)

身份认证:证书的颁发机构,CA 节点接收客户端的注册申请,返回注册密码用于用户的登录,以便获取身份证书.

拓扑图示例:



4.共识机制

4.1 所谓共识

交易背书(模拟@Endorser)

交易排序(排序@Orderer)

交易验证(验证@Committer)

Orderer 功能:

交易排序,区块分发,多通道数据隔离

交易排序

目的:保证交易系统交易顺序的一致性(有限状态机)

Solo:单节点排序,所见即所得

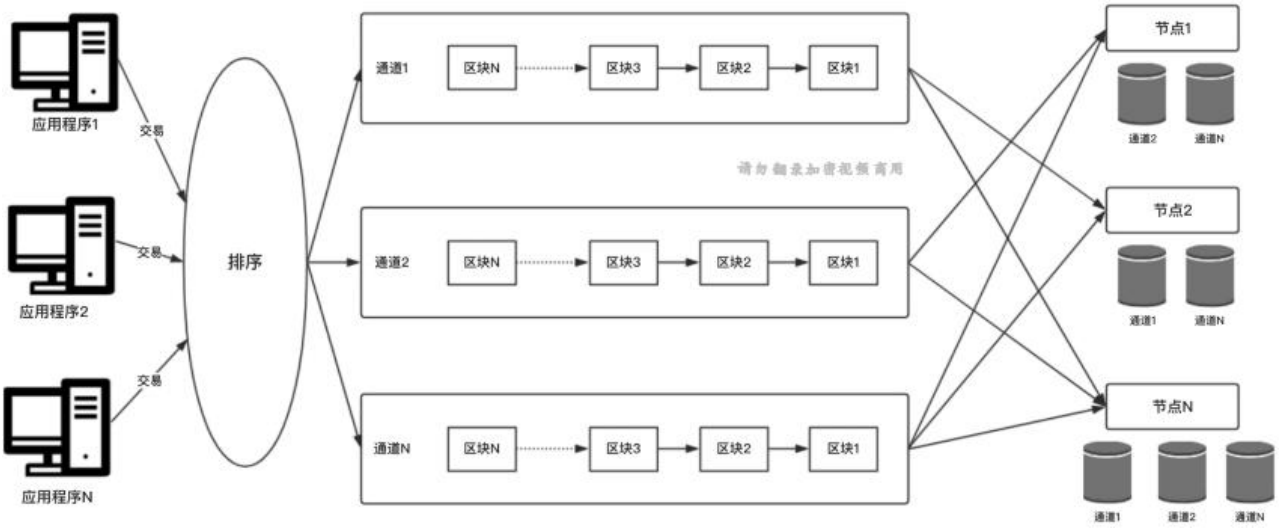
Kafka:外置消息队列保证一致性

区块分发

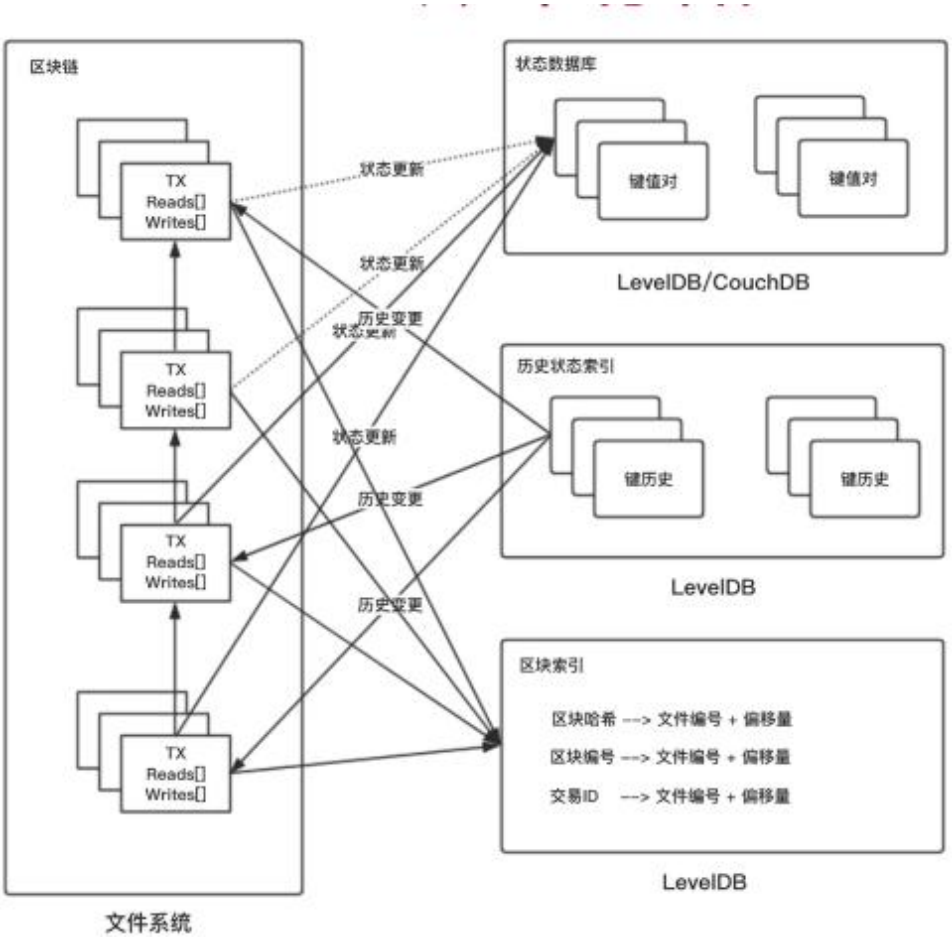
中间状态区块(非落盘区块)

有效交易&无效交易

多通道



5.账本存储



5.1 交易流程

注:后续可以重点理解源码中交易读写集的校验

交易模拟-----> 读写集(RWSet)

交易排序

交易验证-----> 状态更新

5.2 交易读写集的理解

```
// 1. 验证状态更新集中是否对key进行了更改
if updates.Exists(ns, kvRead.Key) {
    return false, nil
}

// 2. 读取世界状态
versionedValue, err := v.db.GetState(ns, kvRead.Key)
if err != nil {
    return false, err
}
var committedVersion *version.Height
if versionedValue != nil {
    committedVersion = versionedValue.Version
}

// 3. 验证世界状态version == 读集中的version
if !version.AreSame(committedVersion, rwsetutil.NewVersion(kvRead.Version)) {
    logger.Debugf( format: "Version mismatch for key [%s:%s]. Committed version =
        ns, kvRead.Key, committedVersion, kvRead.Version)
    return false, nil
```

简单说就是防止双花的机制

“双花”，即双重支付，指的是在数字货币系统中，由于数据的可复制性，使得系统可能存在同一笔数字资产因不当操作被重复使用的情况。

当前的读集状态必须要与当前的世界状态以及当前未被持久化的前续交易执行后的状态保持一致,如果不一致这笔交易就是无效的

5.3 状态数据库

在 fabric 中现支持的有两种数据库引擎

1.levelDB

2.couchDB

区别:couchDB 支持模糊查询,而 levelDB 不支持

官方建议,开发前期采用 levelDB,但是存储的值都是 json 格式,后续切换到 couchDB 也不会有障碍

6.智能合约

1.区块链 2.0:以太坊进行引入智能合约

2.合约协议的数字化代码表达

3.分布式有限状态机

4.执行环境安全隔离,不受第三方干扰(以太坊的 EVM 虚拟机,fabric 使用 Docker)

badexample.go 简单 chaincode 代码(错误代码,只是为了后续的安装链码演示):

```
package main
import (
   "github.com/hyperledger/fabric/core/chaincode/shim"
   pb "github.com/hyperledger/fabric/protos/peer"
   "bytes"
   "strconv"
   "math/rand"
   "time"
   "fmt"
)
type BadExampleCC struct {}
//链码的初始化
func (c *BadExampleCC) Init(stub shim.ChaincodeStubInterface)
pb.Response {
   return shim.Success(nil)
}
func (c *BadExampleCC) Invoke(stub shim.ChaincodeStubInterface)
pb.Response{
   return
shim.Success(bytes.NewBufferString(strconv.Itoa(int(rand.Int63n(time.
Now().Unix()))))).Bytes())
}
func main() {
   //启动链码
   err := shim.Start(new(BadExampleCC))
   if err != nil {
      fmt.Printf("Error starting Simple chaincode: %s", err)
   }
}
```

6.1 链码

1.Fabric 应用层基石(中间件)

2.独立的 Docker 执行环境

3.背书节点 gRPC 连接(注:只有背书节点才会执行链码)

4.生命周期的管理

6.1.1 生命周期

1.打包

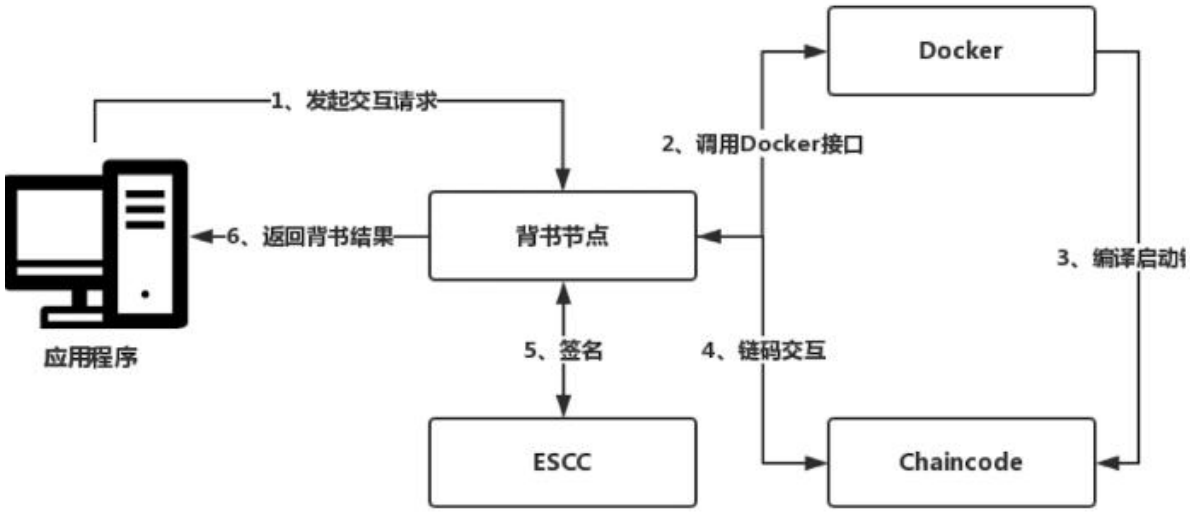
2.安装(将打包好的文件上传到背书节点)

3.实例化(执行 init 方法,在整个链码的生命周期中这个方法只会被执行一次)

4.升级

5.交互(查询,写入)

6.1.2 链码的交互流程



整个过程是 Fabric 交易中的交易模拟,这里只是将其细化

ESCC 是系统链码,主要来完一些系统流程,虽然是链码但是是运行在节点进程中的而不是以一个容器存在

6.1.3 系统链码

LSCC(Lifecycle System Chaincode)管理链码的生命周期,主要管理安装实例化和升级

CSCC(Configuration System Chaincode)管理某一条链的配置,允许新的节点加入某一条链

QSCC(Query System Chaincode)查询账本存储,区块索引外部服务

ESCC(Endorsement System Chaincode)主要是交易执行后的链码进行交易背书签名组装成客户端认识的交易背书结果

VSCC(Validation System Chaincode)主要使用做交易验证

6.2 链码的编程接口

Init()

进行链码的初始化操作

Invoke()

程序交互的入口,查询和写入

任何实现了这两个接口的代码都可以认为是链码

6.2.1 链码的 SDK 接口

参数解析

获取当前交易的信息

状态的操作

链码的互操作

事件送达

其他

6.2.2 链码编程的禁忌

链码的运行是在分布式,所节点的系统中隔离执行

在整个区块链网络中会对该交易执行很多次,执行的次数取决于背书策略的选择. 可以选择这条链上所有节点都执行,也可以选择只有某一个组织的某一个节点执行这样的坏处就是不能因为节点的不同而执行而产生不一样的结果,因为客户端会去比较从不同节点返回的交易模拟的结果,如果不一样就不会被发往排序节点进行排序.

随机函数

系统时间(因为是分布式系统,所以任何基于时间的执行结果都可能造成该交易无效)

不稳定的外部依赖

7.网络搭建

7.1 准备配置文件:

crypto-config.yaml:用于配置组织节点的个数,每一个组织里面用户的个数,后面会更具这些用户生成组织节点的证书

configtx.yaml:配置区块链联盟中的组织信息,配置组织的名字以及对应的证书的位置,peer 节点来说还有对外的组节点,对排序节点来说还有共识机制的选择,时间间隔以及每个块中能包含的交易的数量块的大小等信息

7.2 创世配置构造

MSP 证书:组织节点用户的证书

Orderer 创世区块

Channel 创世交易(区块):构建第一笔交易,这笔交易是一笔配置交易,包含了通道的初始化信息,交易会被单独包装成区块,就是这个通道的创世区块

组织主节点的交易

7.3 网络启动

1.创建 Channel

2.加入 Channel

3.安装 Chaincode

4.实例化 Chaincode

5.Chaincode 交互测试

7.4 具体搭建流程

前提环境变量配置:

添加 GOPATH 路径:

例: export GOPATH=/home/hk/go(根据自己的实际情况进行配置)

添加 FBRIC_CFG_PATH 路径:

export FBRIC_CFG_PATH=$GOPATH/src/github.com/hyperledger/fabric/imocc/deploy

1.生成证书

1.进入之前的 fabric 项目中并创建自己的项目文件夹

例:在 fabric 文件夹下创建文件夹 imocc

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric/imocc$ pwd
/home/hk/go/src/github.com/hyperledger/fabric/imocc
```

2.然后进入 imocc 后创建两个文件夹 chaincode 和 deploy

3.在 deploy 下创建 config 和 crypto-config 以及三个 yaml 文件

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric/imocc/deploy$ ls
config  configtx.yaml  crypto-config  crypto-config.yaml  docker-compose.yaml
```

4.然后进入 crypto-config.yaml 进行配置

首先使用 bash 命令进行清空 crypto-config 文件夹下的所有文件:

rm -rf crypto-config/*

```
#"OrdererOrgs" - 定义管理排序节点的组织
OrdererOrgs:
  - Name: Orderer #组织的名字
    Domain: imocc.com #域名
    Specs: #定义组织的节点信息
      - Hostname: orderer #单个orderer(solo),节点域名orderer.imocc.com
      # - Hostname: orderer1 - 如果是基于kafka 模式的可以继续加节点
#"PeerOrgs" - 定义管理peer 节点的组织
```

```
PeerOrgs:
  - Name: Org0
    Domain: org0.imocc.com
    Template: #使用模板定义peer 节点
      Count: 2 # peer0.org0.imocc.com & peer1.org0.imocc.com
      # Start: 5 - 指定index 计数从0 开始还是从定义的Start 开始
    Users: #定义这个组织有多少个用户,这个用户指的是除了Admin 之外的其他用户
      Count: 2 # Admin & User1 & User2,Admin 是内置自动生成的
  - Name: Org1
    Domain: org1.imocc.com
    Specs:
      - Hostname: peer0
      - Hostname: peer1
    Template: #又创建一个节点peer3
      Count: 1
      Start: 2
    Users:
      Count: 3 #Admin & User1 & User2 & User3
```

最后编写好 yaml 的配置在终端对应的路径下运行下面的命令:

cryptogen generate --config=./crypto-config.yaml

2.生成创世区块

1.清空 config 文件夹下的所有文件:

rm -fr config/*

2.编写 configtx.yaml

```
Profiles: #联盟的配置
  OneOrgOrdererGenesis: #一般联盟配置包含两个部分,系统链:全局的的组织信息
    Orderer:
      <<: *OrdererDefaults
      Organizations:
        - *OrdererOrg
    Consortiums:
      SampleConsortium:
        Organizations:
          - *Org0
          - *Org1
  TwoOrgChannel: #业务相关的联盟
    Consortium: SampleConsortium
    Application:
      <<: *ApplicationDefaults
      Organizations:
        - *Org0
        - *Org1
############################组织的相关配置#################################
Organizations:
```

```
    - &OrdererOrg
      Name: OrdererOrg
      ID: OrdererMSP
      MSPDir: crypto-config/ordererOrganizations/imocc.com/msp #证书的所在
位置,相对路径
    - &Org0
      Name: Org0MSP
      ID: Org0MSP
      MSPDir: crypto-config/peerOrganizations/org0.imocc.com/msp
      AnchorPeers: #主节点的配置
        - Host: peer0.org0.imocc.com
          Port: 7051
    - &Org1
        Name: Org1MSP
        ID: Org1MSP
        MSPDir: crypto-config/peerOrganizations/org1.imocc.com/msp
        AnchorPeers:
          - Host: peer0.org1.imocc.com
            Port: 7051
########################Orderer 组织的相关配置##############################
Orderer: &OrdererDefaults
    # Available types are "solo" and "kafka"
    Ordertype: solo #共识机制
    Addresses:
      - orderer.imocc.com:7050
    BatchTimeout: 2s #出块的时间间隔
    BatchSize:
      MaxMessageCount: 10
      AbsoluteMaxBytes: 99 MB
      PreferredMaxBytes: 512 KB
    Kafka:
      Brokers:
        - 127.0.0.1:9092
    Organizations:
############################应用配置#####################################
Application: &ApplicationDefaults
    Organizations:
```

3.最后在对应的路径执下行命令:

-profile:指定联盟配置,在上述的 yaml 文件中的配置

-outputBlock:指定输出的 block 放在什么地方

configtxgen -profile OneOrgOrdererGenesis -outputBlock ./config/genesis.block

在 config 文件夹下的就多了一个文件 genesis.block

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric/imooc/deploy/config$ ls
genesis.block
```

3.生成某一通道的创世交易

-profile:指定业务联盟,对应上述的 yaml 文件中的业务联盟的配置

-outputCreateChannelTx:输出一个创建 channel 交易的路径放在 config 目录下

-channelID:创建 channel 的名字为 mychannel

configtxgen -profile TwoOrgChannel -outputCreateChannelTx ./config/mychannel.tx -channelID mychannel

注意:channelID 不能使用大写来命名

生成 mychannel.tx 文件

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric/imooc/deploy/config$ ls
genesis.block  mychannel.tx
```

4.设置锚节点的配置

非必要操作,但是最好还是进行配置

-profile:指定业务联盟

-outputAnchorPeersUpdate :指定输出文件路径为 config 路径下

-channelID:针对的是 mychannel 的通道

-asOrg:主要指定的是针对哪一个组织去做锚节点的配置

configtxgen -profile TwoOrgChannel -outputAnchorPeersUpdate ./config/Org0MSPanchors.tx -channelID mychannel -asOrg Org0MSP

configtxgen -profile TwoOrgChannel -outputAnchorPeersUpdate ./config/Org1MSPanchors.tx -channelID mychannel -asOrg Org1MSP

即生成两个组织的锚节点交易配置文件:

Org0MSPanchors.tx Org1MSPanchors.tx

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric/imooc/deploy/config$ ls
genesis.block  mychannel.tx  Org0MSPanchors.tx  Org1MSPanchors.tx
```

5.通过 Docker Compose 来启动网络

1.配置 docker-compose.yaml 文件

```
version: '2'
services:
```

```
# 关注点
# 1. 如何注入系统配置到容器中 环境变量注入
# 2. 端口的映射关系
# 3. 文件的映射
  orderer.imocc.com:
    container_name: orderer.imocc.com
    image: hyperledger/fabric-orderer:x86_64-1.0.0
    environment:
      - ORDERER_GENERAL_LOGLEVEL=debug # general.loglevel: debug
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0 # 服务暴露地址
      - ORDERER_GENERAL_GENESISMETHOD=file # 注入创世区块
      -
ORDERER_GENERAL_GENESISFILE=/etc/hyperledger/config/genesis.block # 注
入创世区块,使用的是容器中的路径而不是本地的路径
      - ORDERER_GENERAL_LOCALMSPID=OrdererMSP # 证书相关
      - ORDERER_GENERAL_LOCALMSPDIR=/etc/hyperledger/orderer/msp # 证书
相关
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderer
    command: orderer
    ports:
      - 7050:7050 # 17050:7050,如果本机的端口7050 已经被占用,将本机的配置为
17050
    volumes:
        #- ./config/:/etc/hyperledger/config
        - ./config/genesis.block:/etc/hyperledger/config/genesis.block
#使用这种方式就没有引入过多的文件,看上去更加清晰

- ./crypto-config/ordererOrganizations/imocc.com/orderers/orderer.imoc
c.com/:/etc/hyperledger/orderer
  peer.base: # 因为peer 的公共参数较多,提取出来作为peer 的公共服务
    image: hyperledger/fabric-peer:x86_64-1.0.0
    environment: # 前缀:CORE
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock # docker 的服
务端注入
      - CORE_LOGGING_PEER=debug
      - CORE_CHAINCODE_LOGGING_LEVEL=DEBUG
      - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/peer/msp # msp 证书(节
点证书)
      - CORE_LEDGER_STATE_STATEDATABASE=goleveldb # 状态数据库的存储引擎(or
CouchDB)
      # # the following setting starts chaincode containers on the same
      # # bridge network as the peers
      # # https://docs.docker.com/compose/networking/
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=deploy_default #
chaincode 与peer 节点使用同一个网络,如果不设置该参数,链码可能会连不上peer 节点
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
    command: peer node start
##################对5 个peer 节点做不同的配置#################
  peer0.org0.imocc.com:
```

```
    extends:
      service: peer.base
    container_name: peer0.org0.imocc.com
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer0.org0.imocc.com
      - CORE_PEER_LOCALMSPID=Org0MSP
      - CORE_PEER_ADDRESS=peer0.org1.imocc.com:7051
    ports:
      - 7051:7051 # grpc 服务端口
      - 7053:7053 # eventhup 端口,主要做事件监听
    volumes:
      - /var/run/:/host/var/run/

- ./crypto-config/peerOrganizations/org0.imocc.com/peers/peer0.org0.im
occ.com:/etc/hyperledger/peer
    depends_on:
      - orderer.imocc.com
  peer1.org0.imocc.com:
    extends:
      service: peer.base
    container_name: peer1.org0.imocc.com
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer1.org0.imocc.com
      - CORE_PEER_LOCALMSPID=Org0MSP
      - CORE_PEER_ADDRESS=peer1.org0.imocc.com:7051
    ports:
      - 17051:7051
      - 17053:7053
    volumes:
      - /var/run/:/host/var/run/

- ./crypto-config/peerOrganizations/org0.imocc.com/peers/peer1.org0.im
occ.com:/etc/hyperledger/peer
    depends_on:
      - orderer.imocc.com
  peer0.org1.imocc.com:
    extends:
      service: peer.base
    container_name: peer0.org1.imocc.com
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer0.org1.imocc.com
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_ADDRESS=peer0.org1.imocc.com:7051
    ports:
      - 27051:7051
      - 27053:7053
    volumes:
      - /var/run/:/host/var/run/

- ./crypto-config/peerOrganizations/org1.imocc.com/peers/peer0.org1.im
occ.com:/etc/hyperledger/peer
```

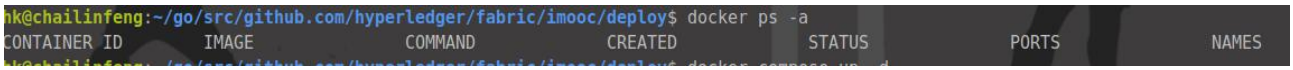
```
    depends_on:
      - orderer.imocc.com
  peer1.org1.imocc.com:
    extends:
      service: peer.base
    container_name: peer1.org1.imocc.com
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer1.org1.imocc.com
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_ADDRESS=peer1.org1.imocc.com:7051
    ports:
      - 37051:7051
      - 37053:7053
    volumes:
      - /var/run/:/host/var/run/

- ./crypto-config/peerOrganizations/org1.imocc.com/peers/peer1.org1.im
occ.com:/etc/hyperledger/peer
    depends_on:
      - orderer.imocc.com
  peer2.org1.imocc.com:
    extends:
      service: peer.base
    container_name: peer2.org1.imocc.com
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer2.org1.imocc.com
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_ADDRESS=peer2.org1.imocc.com:7051
    ports:
      - 47051:7051
      - 47053:7053
    volumes:
      - /var/run/:/host/var/run/

- ./crypto-config/peerOrganizations/org1.imocc.com/peers/peer2.org1.im
occ.com:/etc/hyperledger/peer
    depends_on:
      - orderer.imocc.com
  cli: # peer 节点客户端 交易都是从客户端发起 需要用到User 证书
    container_name: cli
    image: hyperledger/fabric-tools
    tty: true
    environment:
      - GOPATH=/opt/gopath
      - CORE_LOGGING_LEVEL=DEBUG
      - CORE_PEER_ID=cli
      - CORE_PEER_ADDRESS=peer0.org1.imocc.com:7051 #现在连接的是组织1 的
peer0 节点,如果需要连接到其他的节点需要进行修改
      - CORE_PEER_LOCALMSPID=Org1MSP
      -
CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/peer/users/Admin@org1.imocc.
com/msp
```

```
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/
    command: /bin/bash
    volumes:
      - ./../chaincode:/opt/gopath/src/github.com/chaincode # 链码路径注入
      - ./config:/etc/hyperledger/config
- ./crypto-config/peerOrganizations/org1.imocc.com/:/etc/hyperledger/peer
```

2.配置完成后先查看环境是否正常:

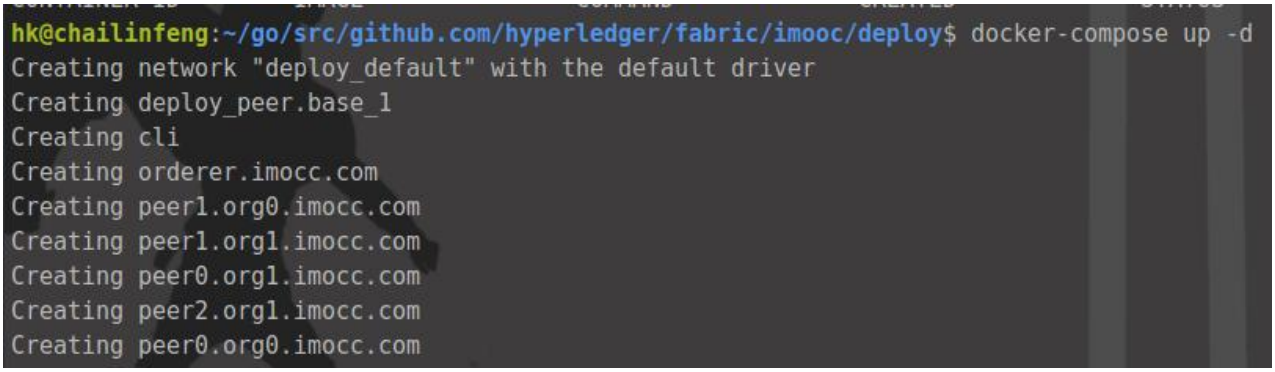
docker ps -a

显示没有已经创建过的容器说明环境正常.



3.使用命令进行启动:

docker-compose up -d (-d 表示后台启动运行,不显示 debug 的过程)



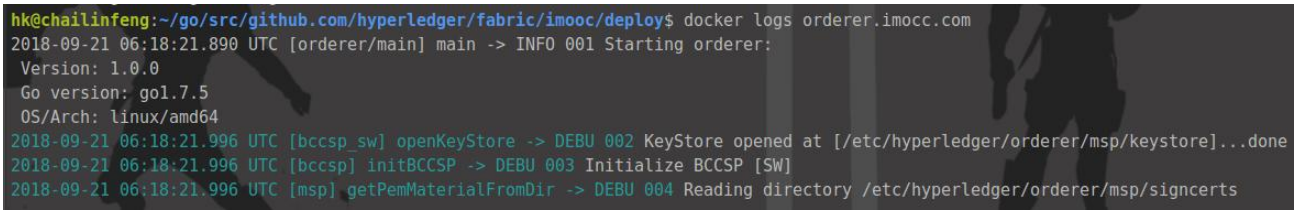
表示已经创建完成.

4.查看启动是否正常

查看容器:docker ps -a

查看日志:

orderer 节点的日志:docker logs orderer.imooc.com



如果日志最后打印的是 Beginning to serve requests 则表示启动正常

peer 节点的日志:docker logs peer0.org1.imocc.com

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric/imooc/deploy$ docker logs peer0.org1.imocc.com
2018-09-21 06:18:32.022 UTC [nodeCmd] serve -> INFO 001 Starting peer:
 Version: 1.0.0
 Go version: go1.7.5
 OS/Arch: linux/amd64
 Chaincode:
  Base Image Version: 0.3.1
  Base Docker Namespace: hyperledger
  Base Docker Label: org.hyperledger.fabric
  Docker Namespace: hyperledger

2018-09-21 06:18:32.022 UTC [ledgermgmt] initialize -> INFO 002 Initializing ledger mgmt
2018-09-21 06:18:32.022 UTC [kvledger] NewProvider -> INFO 003 Initializing ledger provider
2018-09-21 06:18:32.022 UTC [kvledger.util] CreateDirIfMissing -> DEBU 004 CreateDirIfMissing [/var/h
2018-09-21 06:18:32.061 UTC [kvledger.util] logDirStatus -> DEBU 005 Before creating dir - [/var/hype
2018-09-21 06:18:32.061 UTC [kvledger.util] logDirStatus -> DEBU 006 After creating dir - [/var/hyper
2018-09-21 06:18:32.234 UTC [kvledger.util] CreateDirIfMissing -> DEBU 007 CreateDirIfMissing [/var/h
2018-09-21 06:18:32.234 UTC [kvledger.util] logDirStatus -> DEBU 008 Before creating dir - [/var/hype
2018-09-21 06:18:32.234 UTC [kvledger.util] logDirStatus -> DEBU 009 After creating dir - [/var/hyper
```

如果都没有异常说明网络已经正常启动了,后续还需要进入 client 进行初始化操作

6.初始化操作

1.进入 client 中: docker exec -it cli bash

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric/imooc/deploy$ docker exec -it cli bash
root@73d5a71d1239:/opt/gopath/src/github.com/hyperledger/fabric#
```

2.首先对 channel 进行操作:

第一步:

查看一些 peer 操作的帮助命令:

```
root@fb577552f31d:/opt/gopath/src/github.com/hyperledger/fabric# peer
Usage:
  peer [flags]
  peer [command]

Available Commands:
  chaincode   Operate a chaincode: install|instantiate|invoke|package|query|signpackage|upgrade.
  channel     Operate a channel: create|fetch|join|list|update.
  logging     Log levels: getlevel|setlevel|revertlevels.
  node        Operate a peer node: start|status.
  version     Print fabric peer version.
```

查看一些 channel 操作的帮助: peer channel -h

```
root@73d5a71d1239:/opt/gopath/src/github.com/hyperledger/fabric# peer channel -h
Operate a channel: create|fetch|join|list|update.

Usage:
  peer channel [command]

Available Commands:
  create      Create a channel
  fetch       Fetch a block
  join        Joins the peer to a chain.
  list        List of channels peer has joined.
  update      Send a configtx update.
```

第二步:

使用命令 peer channel list:

```
root@73d5a71d1239:/opt/gopath/src/github.com/hyperledger/fabric# peer channel list
2018-09-21 06:34:40.944 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-09-21 06:34:40.944 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-09-21 06:34:41.030 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and orderer connections initial
2018-09-21 06:34:41.035 UTC [msp/identity] Sign -> DEBU 004 Sign: plaintext: 0A81070A5B08031A0B0881A292DD05
2018-09-21 06:34:41.035 UTC [msp/identity] Sign -> DEBU 005 Sign: digest: D8BA842DB8121795887A44137BA8606FF
2018-09-21 06:34:41.108 UTC [channelCmd] list -> INFO 006 Channels peers has joined to:
2018-09-21 06:34:41.108 UTC [main] main -> INFO 007 Exiting.....
```

显示当前节点已经加入的通道为空

第三步:

接下来创建一个通道:

-o 表示需要与哪一个 peer 进行通信

-c 建立的通道名叫什么

-f 使用的是哪一个通道的创世交易,即指定交易方式

peer channel create -o orderer.imocc.com:7050 -c mychannel -f /etc/hyperledger/config/mychannel.tx

```
root@73d5a71d1239:/opt/gopath/src/github.com/hyperledger/fabric# peer channel create -o orderer.imocc.com:7050 -c mychannel -f /etc/hyperledger/config/mychannel.tx
2018-09-21 06:40:14.347 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-09-21 06:40:14.347 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-09-21 06:40:14.349 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and orderer connections initialized
2018-09-21 06:40:14.365 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing local MSP
2018-09-21 06:40:14.365 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining default signing identity
2018-09-21 06:40:14.365 UTC [msp] GetLocalMSP -> DEBU 006 Returning existing local MSP
2018-09-21 06:40:14.365 UTC [msp] GetDefaultSigningIdentity -> DEBU 007 Obtaining default signing identity
2018-09-21 06:40:14.365 UTC [msp/identity] Sign -> DEBU 008 Sign: plaintext: 0A84060A074F7267314D535012F8052D...53616D706C65436F6E736F727469756D
```

在次使用使用第二步的命令发现这时的 peer 节点加入的 channel 还是空,因为我们只是创建了当前的 channel 并没有让加入该通道

第四步:

加入通道:

使用 ls 可以看到当前路径之前下,在使用创建 channel 的命令后生成了一个通道的文件

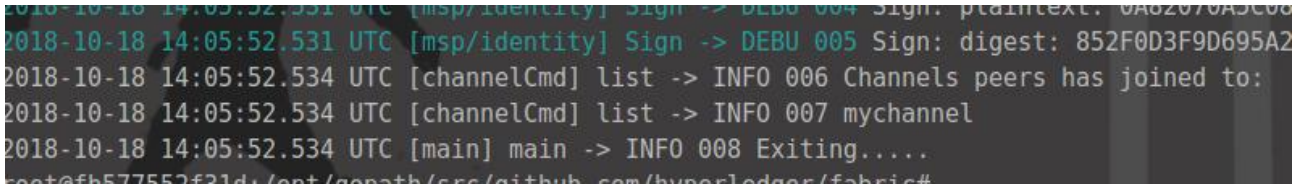
```
root@fb577552f31d:/opt/gopath/src/github.com/hyperledger/fabric# ls
mychannel.block
```

-b 指定的是这个通道的创世区块

peer channel join -b mychannel.block

这时候再执行第二步的操作就会发现已经加入了该通道:

peer channel list



第五步:

设置 channel 的主节点:

peer channel update -o orderer.imocc.com:7050 -c mychannel -f /etc/hyperledger/config/Org1MSPanchors.tx

为什么使用 org1 的 tx 交易?

如下图:

因为在配置中 cli 连接的是 org1 的 peer0 这个节点

如果配置中连接的是 org0 的节点,则这里就必须使用 org0 的交易

设置完成后的输出结果:

```
root@73d5a71d1239:/opt/gopath/src/github.com/hyperledger/fabric# peer channel update -o orderer.imocc.com:7050 -c mychannel -f /etc/hyperledger/config/Org1MSPanchors.tx
2018-09-21 06:49:17.479 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-09-21 06:49:17.479 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-09-21 06:49:17.480 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and orderer connections initialized
2018-09-21 06:49:17.534 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing local MSP
2018-09-21 06:49:17.534 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining default signing identity
2018-09-21 06:49:17.534 UTC [msp] GetLocalMSP -> DEBU 006 Returning existing local MSP
2018-09-21 06:49:17.534 UTC [msp] GetDefaultSigningIdentity -> DEBU 007 Obtaining default signing identity
2018-09-21 06:49:17.534 UTC [msp/identity] Sign -> DEBU 008 Sign: plaintext: 0A84060A074F7267314D535012F8052D...2A0641646D696E732A0641646D696E73
2018-09-21 06:49:17.534 UTC [msp/identity] Sign -> DEBU 009 Sign: digest: 6C9CECC91267A721164532CB307F22911C6C56B012B71DCB1A8E9CF9246DA241
2018-09-21 06:49:17.535 UTC [msp] GetLocalMSP -> DEBU 00a Returning existing local MSP
2018-09-21 06:49:17.535 UTC [msp] GetDefaultSigningIdentity -> DEBU 00b Obtaining default signing identity
2018-09-21 06:49:17.535 UTC [msp] GetLocalMSP -> DEBU 00c Returning existing local MSP
2018-09-21 06:49:17.535 UTC [msp] GetDefaultSigningIdentity -> DEBU 00d Obtaining default signing identity
2018-09-21 06:49:17.535 UTC [msp/identity] Sign -> DEBU 00e Sign: plaintext: 0ABB060A1508021A0608EDA892DD0522...3754968DE32D08131FBBFD6F7C99AEFF
2018-09-21 06:49:17.535 UTC [msp/identity] Sign -> DEBU 00f Sign: digest: 0D8DD280932E64D4F1B57082D9C7939A706C45EF3228E847A58139275074CE31
```

注:上述命令只能设置一次,因为同一笔交易只能被执行一次,再次执行上面的命令就会报出 BAD_REQUEST 的错误提示

7.安装链码

安装 badexample.go 示例代码:

-n 指定链码的名字

-v 指定参数

-l 指定当前链码是由哪门语言编写

-p 指定链码的路径(docker 的文件映射,将链码映射到了 cli 的路径下)

peer chaincode install -n badexample -v 1.0.0 -l golang -p github.com/chaincode/badexample

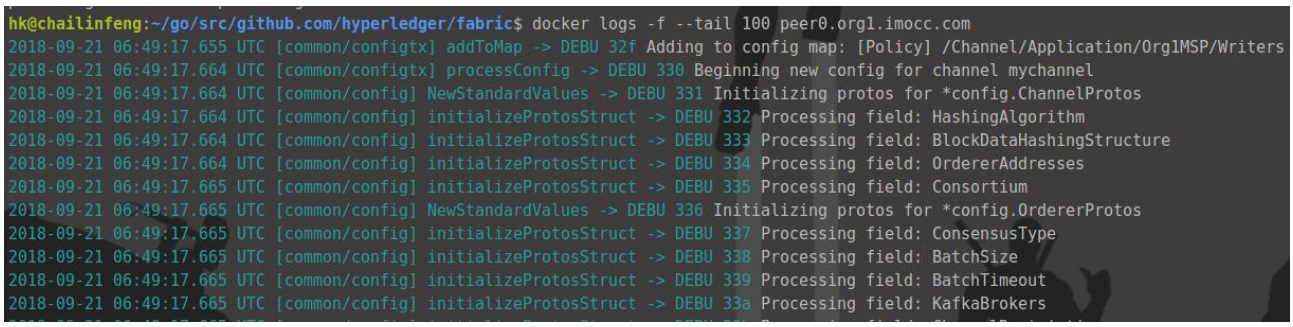
因为之前的 docker-compose.yaml 中配置文件链码映射的路径:

```
volumes:
  - ./../chaincode:/opt/gopath/src/github.com/chaincode # 链码路径注入
  - ./config:/etc/hyperledger/config
```

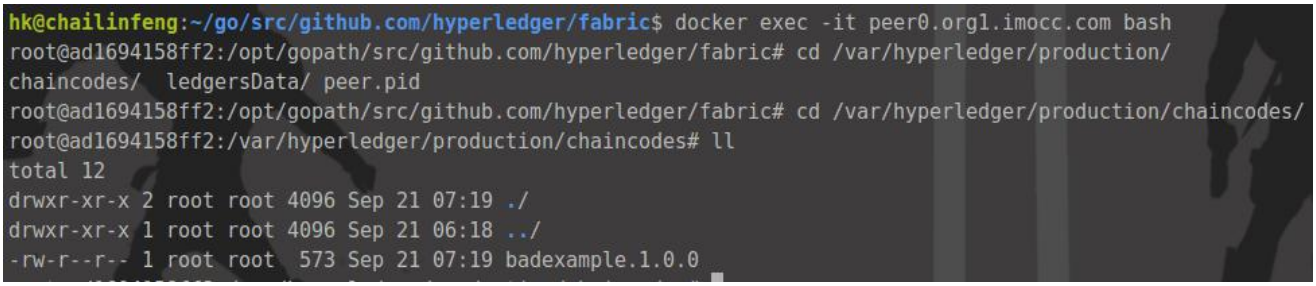
```
root@73d5a71d1239:/opt/gopath/src/github.com/hyperledger/fabric# peer chaincode install -n badexample -v 1.0.0 -l golang -p github.com/chaincode/badexample
2018-09-21 07:19:31.525 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-09-21 07:19:31.525 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-09-21 07:19:31.525 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-09-21 07:19:31.525 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-09-21 07:19:33.056 UTC [golang-platform] getCodeFromFS -> DEBU 005 getCodeFromFS github.com/chaincode/badexample
2018-09-21 07:19:34.422 UTC [golang-platform] func1 -> DEBU 006 Discarding GOROOT package bytes
2018-09-21 07:19:34.422 UTC [golang-platform] func1 -> DEBU 007 Discarding GOROOT package fmt
2018-09-21 07:19:34.422 UTC [golang-platform] func1 -> DEBU 008 Discarding provided package github.com/hyperledger/fabric/core/chaincode/shim
2018-09-21 07:19:34.422 UTC [golang-platform] func1 -> DEBU 009 Discarding provided package github.com/hyperledger/fabric/protos/peer
2018-09-21 07:19:34.422 UTC [golang-platform] func1 -> DEBU 00a Discarding GOROOT package math/rand
2018-09-21 07:19:34.422 UTC [golang-platform] func1 -> DEBU 00b Discarding GOROOT package strconv
2018-09-21 07:19:34.422 UTC [golang-platform] func1 -> DEBU 00c Discarding GOROOT package time
2018-09-21 07:19:34.423 UTC [golang-platform] GetDeploymentPayload -> DEBU 00d done
2018-09-21 07:19:34.430 UTC [msp/identity] Sign -> DEBU 00e Sign: plaintext: 0A82070A5C08031A0C0886B792DD0510...DFE06B000000FFFF74B16B6C000A0000
2018-09-21 07:19:34.430 UTC [msp/identity] Sign -> DEBU 00f Sign: digest: C4AA50306BFB085E0113B8AF0CF6845C5A5A44D39D0241D26DFCE0FDA41B4580
2018-09-21 07:19:34.448 UTC [chaincodeCmd] install -> DEBU 010 Installed remotely response:<status:200 payload:"OK" >
2018-09-21 07:19:34.448 UTC [main] main -> INFO 011 Exiting.....
```

安装成功后,另开一个终端,相同路径

使用 docker logs -f --tail 100 peer0.org1.imocc.com 实时查看 peer 节点的日志可以看到安装链码的日志



也可以使用 docker exec -it peer0.org1.imocc.com bash 进入 peer 节点,然后 cd /var/hyperledger/production/chaincode/查看是否有之前安装的对应的链码



8.实例化链码

-o 指定向哪个 order 进行通信

-C 指定通道的名字

-n 链码的名字

-l 链码的编写语言

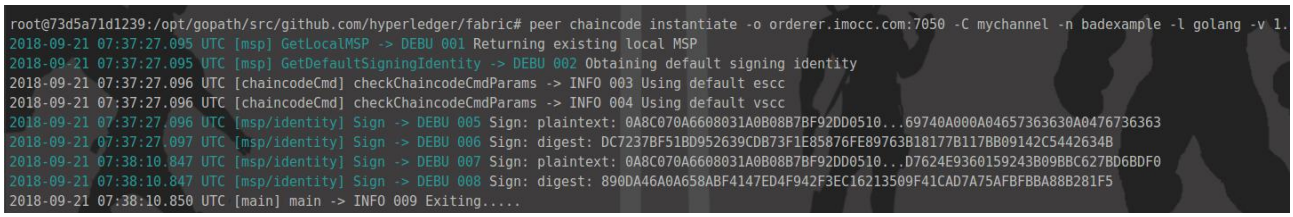
-v 链码的版本

-c 指定初始化的参数

peer chaincode instantiate -o orderer.imocc.com:7050 -C mychannel -n badexample -l golang -v 1.0.0 -c '{"Args":["init"]}'

注:过程可能会有点久,因为需要 pull 镜像并且编译镜像

注:这里可能会经常的犯的一个错误,如果开启了开发者模式则可能在实例化时提示找不到对应链码



成功之后可以去本地使用 docker images 看一下是否有对应的镜像:

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric$ docker images
REPOSITORY                                    TAG                 IMAGE ID
dev-peer0.org1.imocc.com-badexample-1.0.0     latest              24a42e4766e
aberic/fabric-edge                            1.0-RC6             f660ba0dc1a
centos                                        7.2.1511            ddc0fb7d7a7
centos                                        latest              5182e96772b
hyperledger/fabric-ca                         x86_64-1.0.1        5f30bda5f7e
```

查看容器的日志:

docker logs -f dev-peer0.org1.imocc.com-badexample-1.0.0

```
hk@chailinfeng:~/go/src/github.com/hyperledger/fabric$ docker logs -f dev-peer0.org1.imocc.com-badexample-1.0.0
2018-09-21 07:38:10.742 UTC [bccsp] initBCCSP -> DEBU 001 Initialize BCCSP [SW]
^C
```

9.与链码进行交互

进行查询操作

peer chaincode query -C mychannel -n badexample -c '{"Args":[]}'

Query Result 中返回值得到之前链码编写的随机数

```
root@73d5a71d1239:/opt/gopath/src/github.com/hyperledger/fabric# peer chaincode query -C mychannel -n badexam
2018-09-21 07:45:44.077 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-09-21 07:45:44.077 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-09-21 07:45:44.077 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-09-21 07:45:44.077 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-09-21 07:45:44.077 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A92070A6C08031A0B08A8C392DD0510
2018-09-21 07:45:44.077 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 0FBBDC941E5CE004D3D5212339BEC283FE5
Query Result: 721547517
```

8.案例实战

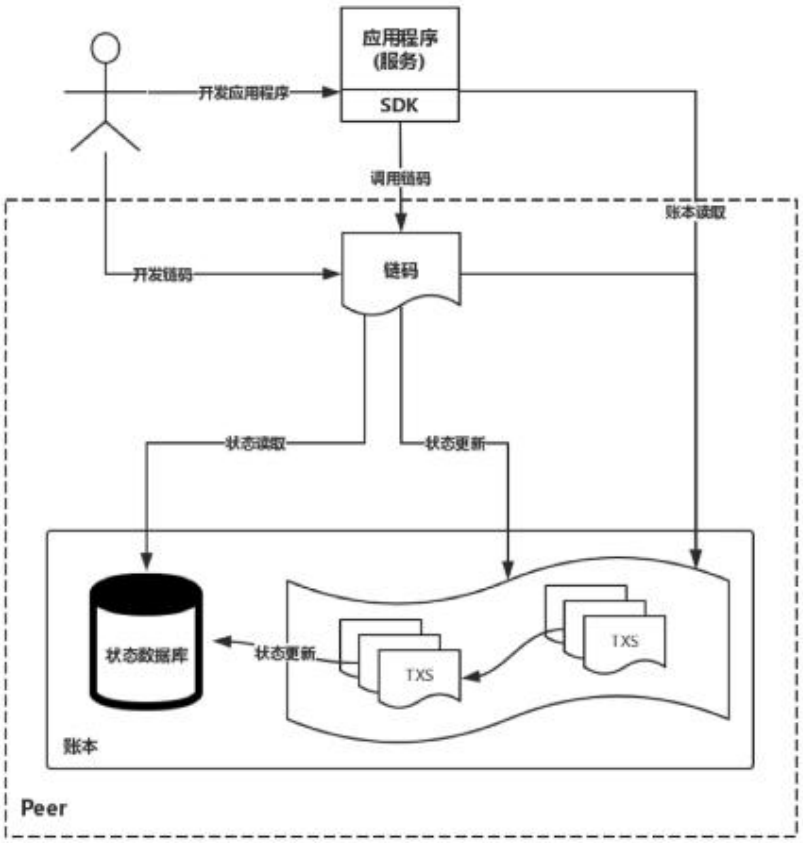
源码获取到 2.3 章节的链接获取源码

8.1 应用的开发流程

1.需求整理(数据上链,交互方法)

2.链码编写

3.链码的交互



链码实现:参考 2.3 节的源码

搭建网络的步骤参考第 7 章

主要使用的命令:

1.生成创世交易通道

configtxgen -profile TwoOrgChannel -outputCreateChannelTx ./config/assetsexchange.tx -channelID assetsexchange

2.创建通道

peer channel create -o orderer.imocc.com:7050 -c assetschannel -f /etc/hyperledger/config/assetschannel.tx

3.加入通道

peer channel join -b assetschannel.block

4.链码安装

peer chaincode install -n assets -v 1.0.0 -l golang -p github.com/chaincode/assetsExchange

5.链码实例化

不加背书策略

peer chaincode instantiate -o orderer.imocc.com:7050 -C assetschannel -n assets -l golang -v 1.0.0 -c '{"Args":["init"]}'

加入背书策略的

-P : policy 指定策略类型

OR('org0MSP.member','org1MSP.admin') : 一笔交易要想有效必须要有 org0 的某一个用户的签名或者说是 org1 的 admin 的签名,只有当交易是有效的才能被发往 order 节点进行排序

peer chaincode instantiate -o orderer.imocc.com:7050 -C assetschannel -n assets -l golang -v 1.0.0 -c '{"Args":["init"]}' -P "OR('org0MSP.member','org1MSP.admin')"

链码的交互进入 cli:

进入 cli 中: docker exec -it cli bash

链码交互:

peer chaincode invoke -C assetschannel -n assets -c '{"Args":["userRegister", "user1", "user1"]}'

peer chaincode invoke -C assetschannel -n assets -c '{"Args":["assetEnroll", "asset1", "asset1", "metadata", "user1"]}'

peer chaincode invoke -C assetschannel -n assets -c '{"Args":["userRegister", "user2", "user2"]}'

peer chaincode invoke -C assetschannel -n assets -c '{"Args":["assetExchange", "user1", "asset1", "user2"]}'

peer chaincode invoke -C assetschannel -n assets -c '{"Args":["userDestroy", "user1"]}'

链码查询:

peer chaincode query -C assetschannel -n assets -c '{"Args":["queryUser", "user1"]}'

peer chaincode query -C assetschannel -n assets -c '{"Args":["queryAsset", "asset1"]}'

peer chaincode query -C assetschannel -n assets -c '{"Args":["queryUser", "user2"]}'

peer chaincode query -C assetschannel -n assets -c '{"Args":["queryAssetHistory", "asset1"]}'

peer chaincode query -C assetschannel -n assets -c '{"Args":["queryAssetHistory", "asset1", "all"]}'

链码升级:

peer chaincode install -n assets -v 1.0.1 -l golang -p github.com/chaincode/assetsExchange

peer chaincode upgrade -C assetschannel -n assets -v 1.0.1 -c '{"Args":["init"]}'

peer chaincode upgrade -C assetschannel -n assets -v 1.0.1 -P "OR('org1MSP.admin')" -c '{"Args":["init"]}'

注:每次修改链码后需要重新安装链码或升级操作才能使链码生效

链码升级只需要对-v 参数版本号进行更改

8.2 链码调试

特殊配置:调试有两种配置方法

1. peer node start --peer-chaincodedev=true

即在 docker-compose 文件中进行修改配置

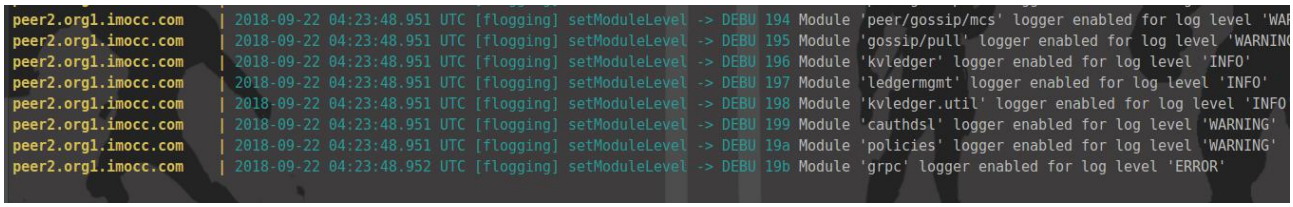
2.CORE_CHAINCODE_MODE=dev

打开终端 1

更改配置后再重新启动 docker-compose

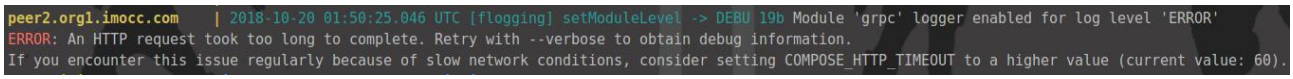
docker-compose down(避免环境冲突,每次启动前建议进行关闭操作)
因为这里需要查看日志进行调试,所以在启动命令后不需要-d(表示后台运行)参数

docker-compose up



启动容器后就进入一个等待的状态

注意:这里可能过一段时间后会出现 COMPOSE_HTTP_TIMEOUT to a higher value (current value: 60)的错误提示



这时候就需要更改一下环境变量 COMPOSE_HTTP_TIMEOUT 的值:

```
export COMPOSE_HTTP_TIMEOUT=12000
```

配置时间可以自己按需要设置

打开终端 2

在 chaincode 目录下的终端使用下面命令来启动链码:

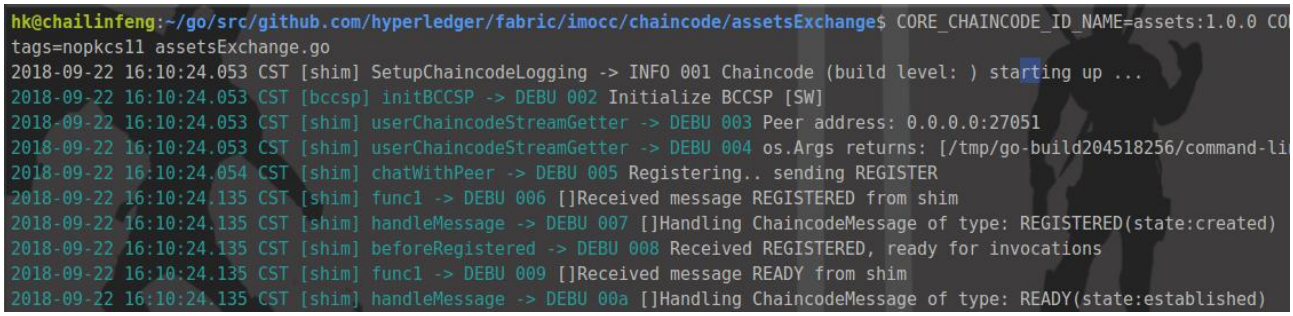
CORE_CHAINCODE_ID_NAME : 指定启动的链码(注:对应的链码名称后必须跟上版本号)

CORE_PEER_ADDRESS : 指定连接的 ip 和端口(注:这里的 0.0.0.0 和 127.0.0.1 大致没有什么区别,0.0.0.0 才是真正表示的本网络中的本机,而 127.0.0.1 只是环回地址并不表示本机)

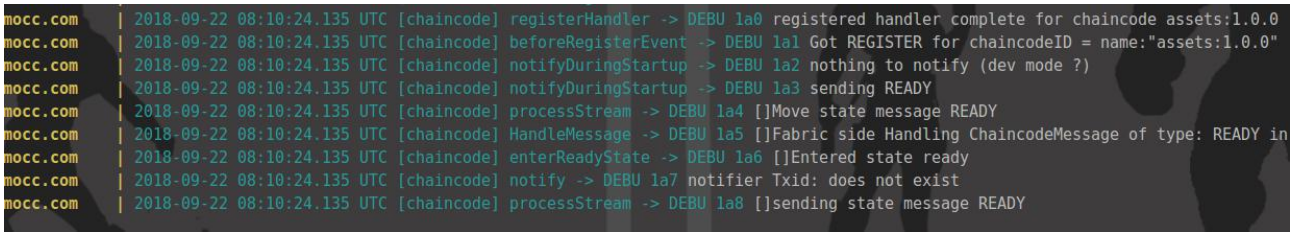
CORE_CHAINCODE_LOGGING_LEVEL : 表示启动的日志级别(方便调试建议使用 DEBUG, 如果不指定该环境变量是不会产生日志信息)

CORE_CHAINCODE_ID_NAME=assets:1.0.0 CORE_PEER_ADDRESS=0.0.0.0:27051 CORE_CHAINCODE_LOGGING_LEVEL=DEBUG go run -tags=nopkcs11 assetsExchange.go

启动后链码处于 READY 的状态:



再次打开之前的终端窗口可以看到 assets:1.0.0 链码已经注册的日志信息:



同样处于一个 READY 的状态.

打开终端 3

配置好上面的环境后就是 cli 端的操作:

如果对命令的意思不是很理解建议回到第 7 章学习

1.进入 cli : docker exec -it cli bash

2.创建通道:peer channel create -o orderer.imocc.com:7050 -c assetschannel -f /etc/hyperledger/config/assetschannel.tx

3.加入通道:peer channel join -b assetschannel.block

4.安装链码:peer chaincode install -n assets -v 1.0.0 -l golang -p github.com/chaincode/assetsExchange

5.实例化链码:peer chaincode instantiate -o orderer.imocc.com:7050 -C assetschannel -n assets -l golang -v 1.0.0 -c '{"Args":["init"]}'

完成以上操作后就是就可以直接在 cli 端进行链码的交互了,和之前的操作没有什么大的区别.

交互的命令参考上一个小结

DEV 模式测试

测试在链码中用户注册时输出两句话:

```
//写入状态数据库
if err := stub.PutState(constructUserKey(id),userBytes);err != nil{
    return shim.Error(fmt.Sprintf( format: "put user error %s",err))
}

fmt.Println( a: "user successful registered!")
fmt.Println( a: "hahaha im testing registered dev")
//成功返回
return shim.Success(userBytes)
```

最后

这里唯一的好处就是在更改链码内容后不需要进行升级操作,而是直接在**终端 2** 使用 CTRL+c 关闭启动的链码然后再次使用启动命令启动就可以了,所有日志信息都可以在**终端 1** 和**终端 2** 下输出

```
2018-10-20 10:12:19.077 CST [shim] sendChannel -> DEBU 02f [c47db8e1]after send
2018-10-20 10:12:19.077 CST [shim] afterResponse -> DEBU 030 [c47db8e1]Received RESPONSE, communicated (state:ready)
2018-10-20 10:12:19.077 CST [shim] handlePutState -> DEBU 031 [c47db8e1]Received RESPONSE. Successfully updated state
user successful registered!
hahaha im testing registered dev
2018-10-20 10:12:19.077 CST [shim] func1 -> DEBU 032 [c47db8e1]Transaction completed. Sending COMPLETED
```

终端 2 输出结果:

9.外部服务

9.1 应用简介和 SDK 选择

SDK 学习主要结合文档最开始的 demo 进行编码学习。下面为 SDK 的一些简单介绍。

决定于应用场景 终端用户

1. 智能硬件 socket/tcp 太阳能发电
2. 游戏、电商、社交 web/app http
3. 企业内部 rpc(grpc)

如何选择 SDK 以及对应的成熟度

现在最为成熟的 sdk 是 nodejs 而 go 语言只是和开发合约并适合开发外部服务,所以成熟度最低

1. Nodejs 4
2. Java 3
3. golang 1

构造交易,发送交易,数据查询

9.2 SDK 的模块

所有的 sdk 模块都一样,主要分为一下三个模块

区块链管理

1. 通道创建和加入
2. 链码的安装,实例化,升级等
3. 一般用到这个模块的主要有两类,admin | 云服务提供商

数据查询

1. 区块查询
2. 交易查询
3. 主要用于区块浏览器

区块链交互

发起交易 invoke | query

事件监听

业务事件　SendEvent

系统事件　block/transaction

9.3 开发外部服务

clone SDK 的源码到$GOPATH/src/github.com/hyperledger/下

切换到对应 fabric 版本的 sdk

编写 sdk 中的配置文件 yaml

9.3.1 主要流程

1. 使用 configtxgen 生成创世交易 tx 文件

2. 将 tx 文件存放至指定路径

3. sdk 调用指定方法导入 tx 文件创建通道