# Fabric安装环境构建：

<https://www.cnblogs.com/llongst/p/9482038.html> 参考博客

## 安装docker

**1卸载原有的docker,**

yum remove docker

yum remove docker docker-common

yum remove docker docker-selinux

yum remove docker docker-engine

**2安装docker ce**

yum install -y yum-utils device-mapper-persistent-data lvm2

#yum-utils提供yum-config-manager工具，设备映射存储驱动需要device-mapper-persistent-data和lvm2工具

yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo #设置稳定存储库

# yum-config-manager --enable docker-ce-edge #开启edge，待理解

# yum-config-manager --enable docker-ce-test #开启test，待理解

yum install docker-ce

docker –version #安装顺利将看到ce版本

systemctl start docker

systemctl enable docker

docker images

docker kill $(docker ps -a -q) 杀死所有正在运行的容器

docker rm $(docker ps -a -q) 删除所有已经停止的容器

docker rmi $(docker images -q) 删除所有镜像  
docker rmi -f $(docker images –q 强制删除所有镜像

**3安装docker-compose**

curl -L https://github.com/docker/compose/releases/download/1.15.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose #下载最1.15.0版本docker-compose

chmod +x /usr/local/bin/docker-compose #赋予执行权限

cp /usr/local/bin/docker-compose /usr/bin #

docker-compose –version #安装顺利将看到版本

## go语言安装：

1. 参照Go官网，找到linux版本下载路径，执行以下操作下载最新版Go语言包

wget https://studygolang.com/dl/golang/go1.10.3.linux-amd64.tar.gz #1.10.3版本视情况而定

tar -xzf go1.8.3.linux-amd64.tar.gz –C /usr/local #解压到指定目录

vim /etc/profile #修改profile文件，最后添加如下两行

export PATH=$PATH:/usr/local/go/bin

export GOPATH=/opt/gopath

source profile #修改后执行

echo $PATH #正常会多个/usr/local/go/bin

echo $GOPATH #正常会显示/opt/gopath

## 安装gcc

[root@orderer ~]#yum -y install gcc #如果二进制文件要编译就需要安装gcc

[root@orderer ~]# go version #查看go版本

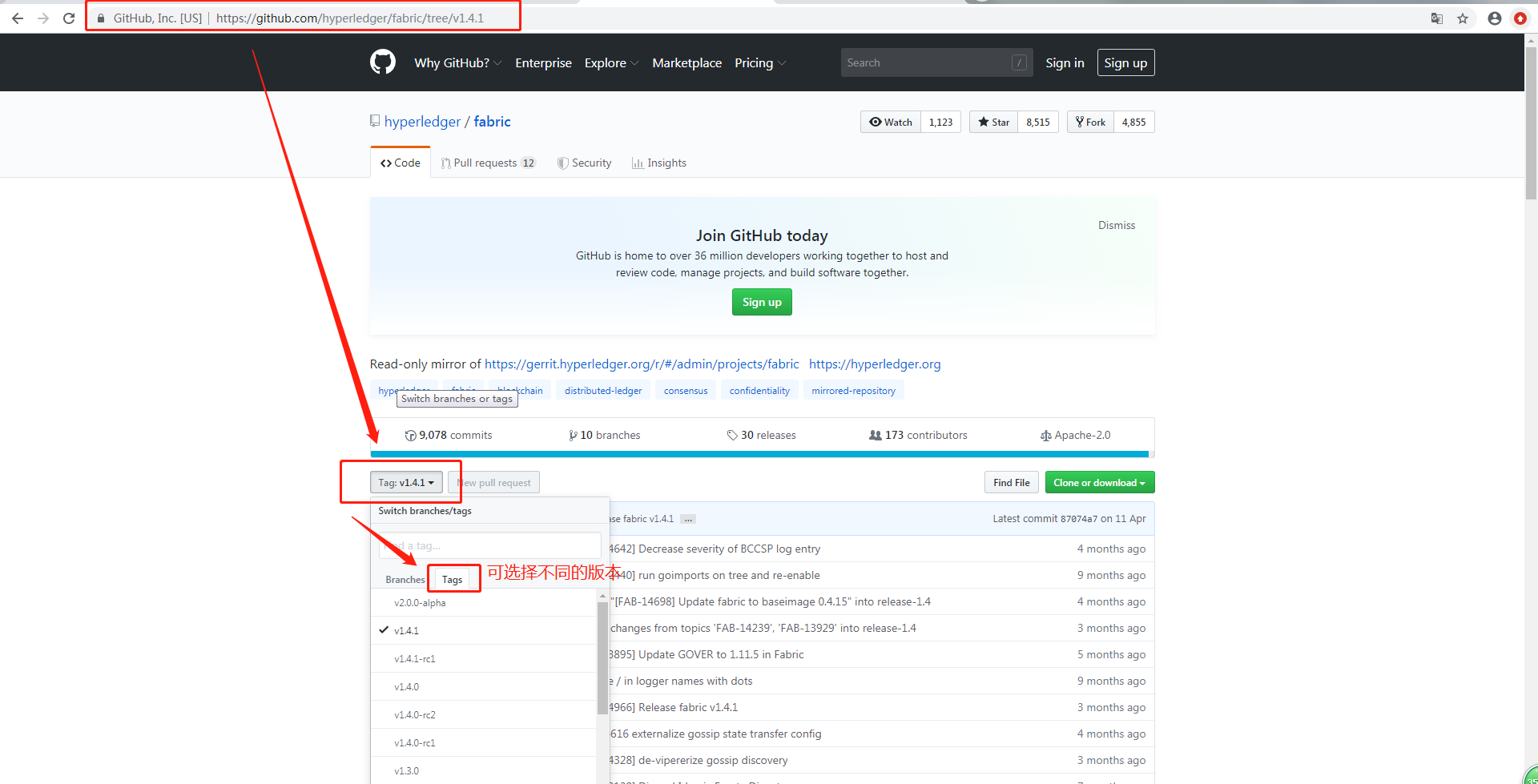
go version go1.12.5 linux/amd64

# 下载fabric源码与docker相关镜像：

<https://github.com/hyperledger/fabric/tree/v1.0.0>

<https://github.com/hyperledger/fabric/tree/v1.1.0>

<https://github.com/hyperledger/fabric/tree/v1.4.1> #修改版本号即可



方式一：一般不成功

go get github.com/hyperledger/fabric

方式二：git clone

yum install git

git clone https://github.com/hyperledger/fabric/ #git clone下来最新的fabric源码（本次为1.4.1版本）

# cd fabric #待验证

# git checkout v1.4.0 #待验证

.

下载相关镜像：

1. 可根据git 下来的 /fabric/scripts/bootstrap.sh 脚本文件下载相关镜像，
2. bootstrap.sh脚本是个自动化下载相关版本二进制文件和镜像文件的脚本。

配置加速镜像

复制代码

# mkdir -p /etc/docker

# tee /etc/docker/daemon.json <<-'EOF'

{

"registry-mirrors": ["https://8w1wqmsz.mirror.aliyuncs.com"]

}

EOF

# systemctl daemon-reload

# systemctl restart docker

docker镜像：

docker pull hyperledger/fabric-ca:1.4.1 #先下载镜像，以ca为例

docker tag hyperledger/fabric-ca:1.4.1 hyperledger/fabric-ca:latest #再将下载的改TAG（标签）

[root@fabric1 fabric-samples]# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

hyperledger/fabric-ca 1.4.1 3a1799cda5d7 2 months ago 252MB

hyperledger/fabric-ca latest 3a1799cda5d7 2 months ago 252MB

hyperledger/fabric-tools 1.4.1 432c24764fbb 2 months ago 1.55GB

hyperledger/fabric-tools latest 432c24764fbb 2 months ago 1.55GB

hyperledger/fabric-ccenv 1.4.1 d7433c4b2a1c 2 months ago 1.43GB

hyperledger/fabric-ccenv latest d7433c4b2a1c 2 months ago 1.43GB

hyperledger/fabric-orderer 1.4.1 ec4ca236d3d4 2 months ago 173MB

hyperledger/fabric-orderer latest ec4ca236d3d4 2 months ago 173MB

hyperledger/fabric-peer 1.4.1 a1e3874f338b 2 months ago 178MB

hyperledger/fabric-peer latest a1e3874f338b 2 months ago 178MB

hyperledger/fabric-zookeeper 0.4.15 20c6045930c8 3 months ago 1.43GB

hyperledger/fabric-zookeeper latest 20c6045930c8 3 months ago 1.43GB

hyperledger/fabric-kafka 0.4.15 b4ab82bbaf2f 3 months ago 1.44GB

hyperledger/fabric-kafka latest b4ab82bbaf2f 3 months ago 1.44GB

hyperledger/fabric-couchdb 0.4.15 8de128a55539 3 months ago 1.5GB

hyperledger/fabric-couchdb latest 8de128a55539 3 months ago 1.5GB

hyperledger/fabric-baseimage amd64-0.4.15 c4c532c23a50 3 months ago 1.39GB

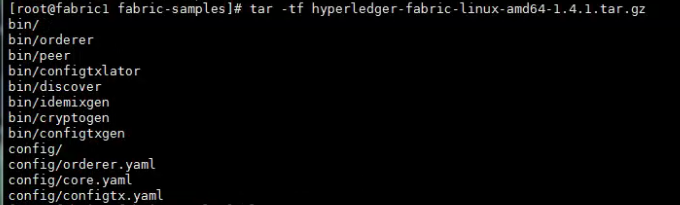
hyperledger/fabric-baseos 0.4.15 9d6ec11c60ff 3 months ago 145MB

hyperledger/fabric-baseos latest 9d6ec11c60ff 3 months ago 145MB

## bin目录二进制文件获取之下载：

在以下的两个压缩包内：下载较慢，最好是用现成的

wget https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/fabric-ca/hyperledger-fabric-ca/linux-amd64-1.4.0/hyperledger-fabric-ca-linux-amd64-1.4.1.tar.gz



wget https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/fabric-ca/hyperledger-fabric-ca/linux-amd64-1.4.1/hyperledger-fabric-ca-linux-amd64-1.4.1.tar.gz



[root@fabric1 bin]# pwd

/opt/gopath/src/github.com/hyperledger/fabric/scripts/fabric-samples/bin

[root@fabric1 bin]# ls #二进制文件获取之前的内容

orderer

[root@fabric1 bin]# ls #二进制将两个下载下来的包的内容解压到此目录中，

bin config configtxgen configtxlator cryptogen discover fabric-ca-client idemixgen orderer peer



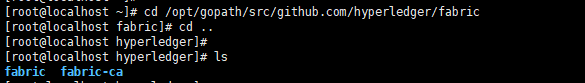
实际开发中，这些二进制执行文件可以使用fabric源码编译生成，我们也许并不需要自己手动去编译生成这些文件，可以直接执行bootstrap.sh这个脚本帮我们下载好。bootstrap.sh下载相关二进制文件与镜像非常的慢。作为学习研究，我们还是有必要知道这些二进制文件是怎么来的。

下面介绍如何编译fabric和fabric-ca源码

## bin目录二进制文件获取之源码编译：

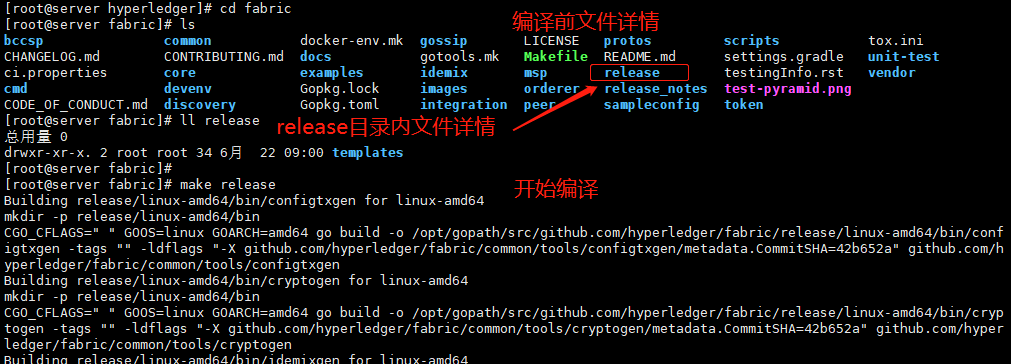
**编译fabric源码：**编译之前请确保系统中已经安装了Go语言运行环境和正确的配置了环境变量。$ mkdir -p $GOPATH/src/github.com/hyperledger #创建目录，前面GOPATH定义的路径是：/opt/gopath

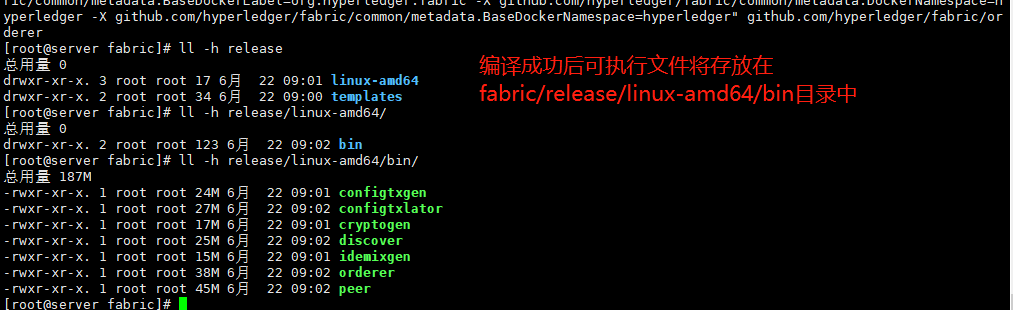
$ cd $GOPATH/src/github.com/hyperledger #将fabric源码放入此路径中



$ cd fabric #进入fabric目录准备编译

$ make release #编译命令





$ cd $GOPATH/src/github.com/hyperledger/fabric/release/linux-amd64/bin/ #编译后二进制文件所在路径

**编译fabric-ca源码**

$ git clone https://github.com/hyperledger/fabric-ca.git #下载fabric-ca源码



# 按照官网建立第一个fabric案例网络

下载fabric-samples（fabric案例）源码：

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

[root@server ~]# mkdir -p /opt/gopath/src/github.com/hyperledger/ #GOPATH路径中，有就忽略此步

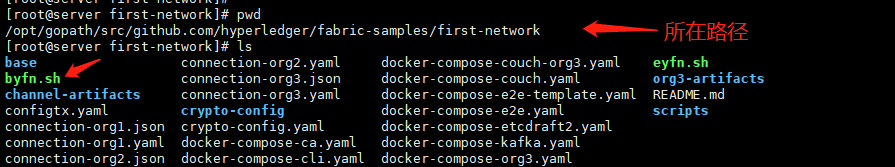
[root@server ~]# cp -r fabric-samples /opt/gopath/src/github.com/hyperledger/ #将fabric-samples拷贝此路径

[root@server ~]# cd /opt/gopath/src/github.com/hyperledger/fabric-samples/ #进入fabric-samples目录

[root@server fabric-samples]# ls bin/ #确认有如下二进制文件，如无此目录，则拷贝过来

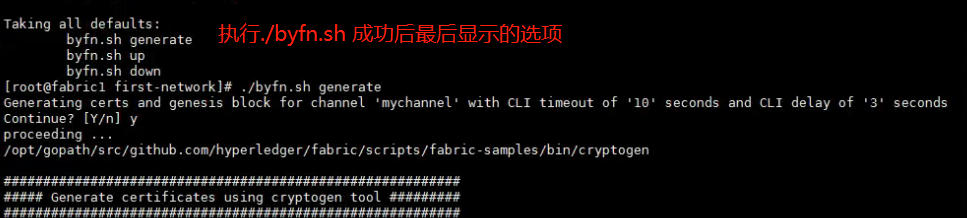
configtxgen configtxlator cryptogen discover fabric-ca-client get-docker-images.sh idemixgen orderer peer

byfn.sh利用这些Docker镜像快速引导Hyperledger Fabric网络，该网络默认由四个代表两个不同组织的对等体和一个orderer节点组成。它还将启动一个容器来运行脚本执行，该脚本执行将对等点连接到通道，部署链代码并根据部署的链代码驱动事务执行。

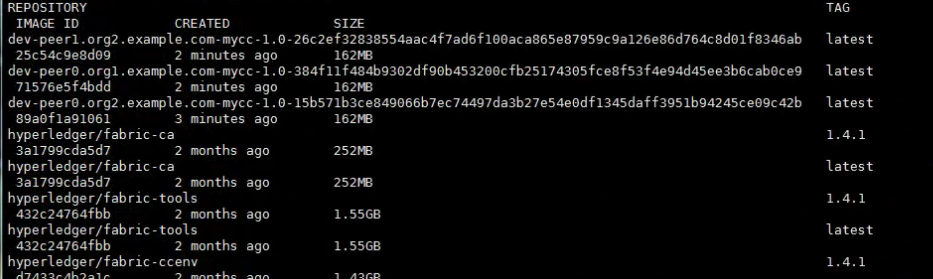


[root@fabric1 first-network]# ./byfn.sh #执行脚本得到提示

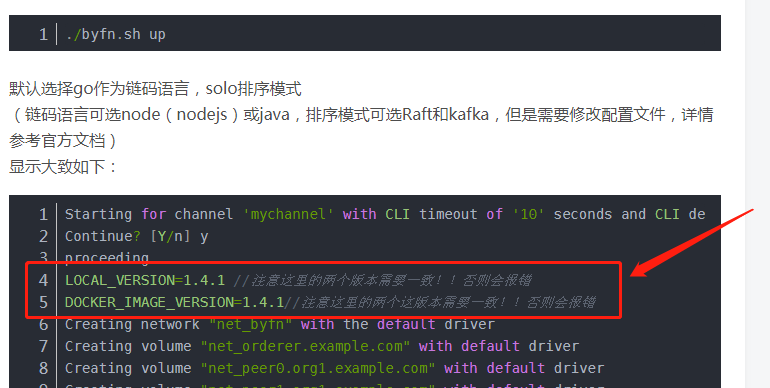
oot@fabric1 first-network]# ./byfn.sh generate #generate(生成)



[root@fabric1 first-network]# docker images #将创建一些docker镜像



[root@fabric1 first-network]# ./byfn.sh up #启动网络



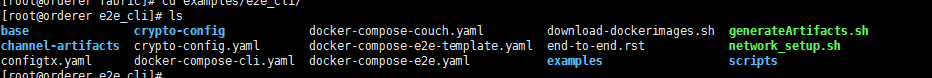
[root@fabric1 first-network]# ./byfn.sh down #停止网络

[root@fabric1 first-network]# docker images #新创建的会被删除



# E2E案例实施之1.0.0

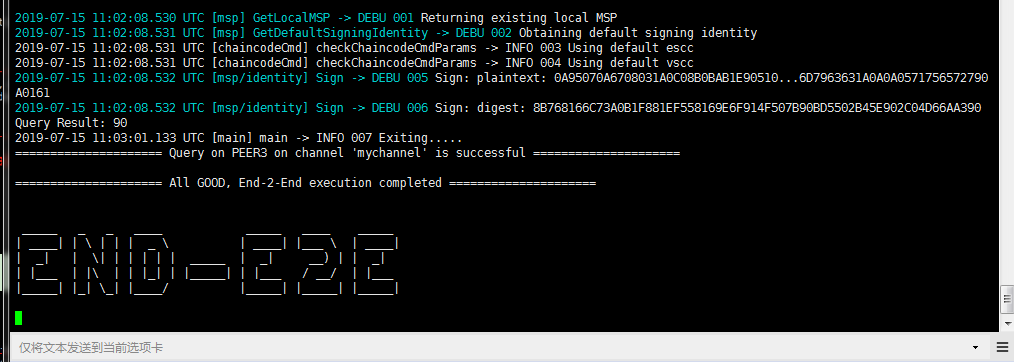
[root@orderer fabric]# cd examples/e2e\_cli/



[root@orderer e2e\_cli]# ./network\_setup.sh up 启动

[root@orderer e2e\_cli]# ./network\_setup.sh up 停止

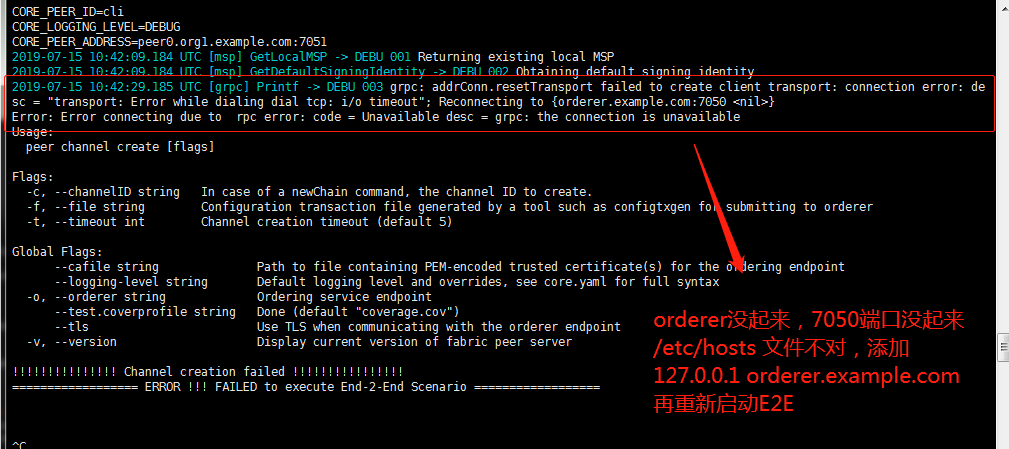
启动成功如下：



错误一：



错误二：



# E2E案例之多节点部署：

Orderer orderer.example.cn 192.168.10.221

Sp0 peer0.org1.example.cn 192.168.10.222

Sp1 peer1.org1.example.cn 192.168.10.223

Sp2 peer0.org2.example.cn 192.168.10.224

Sp3 peer1.org2.example.cn 192.168.10.225

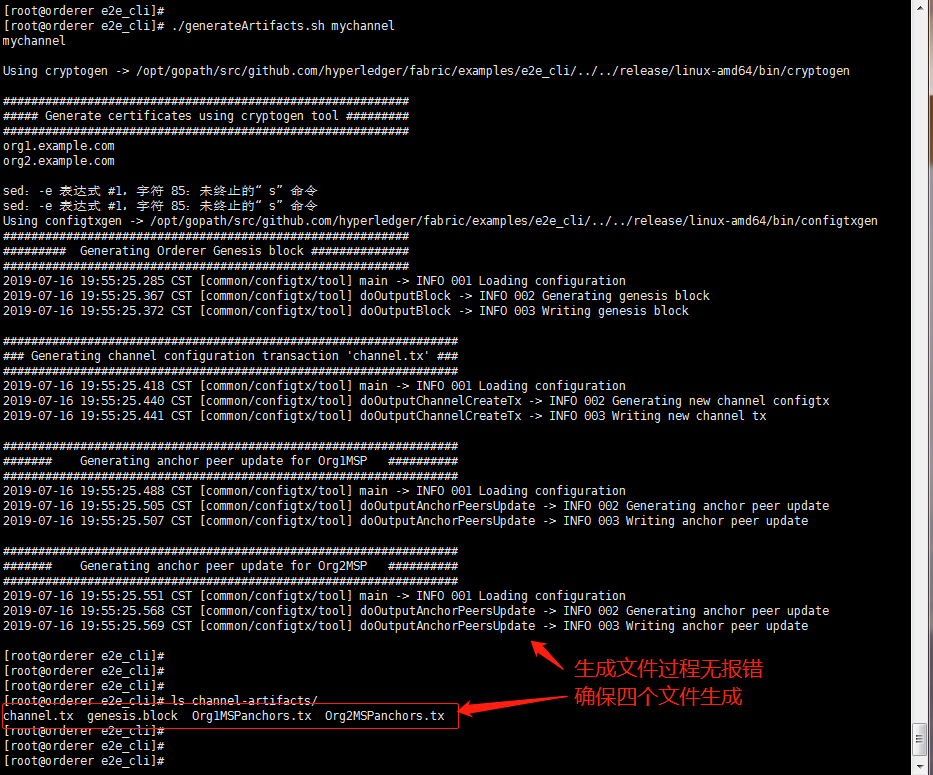
## 生成公私钥、证书、创世区块等(在orderer上)

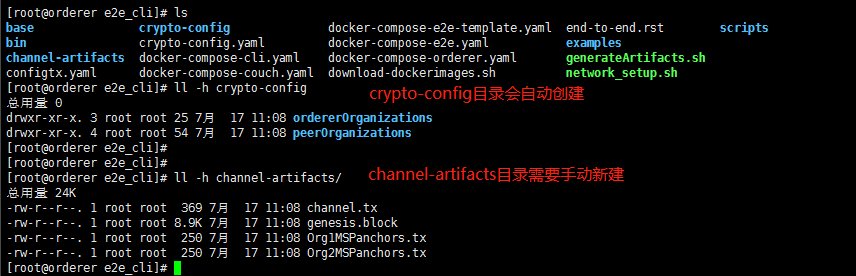
[root@orderer e2e\_cli]# ls generateArtifacts.sh

generateArtifacts.sh

[root@orderer e2e\_cli]# mkdir channel-artifacts

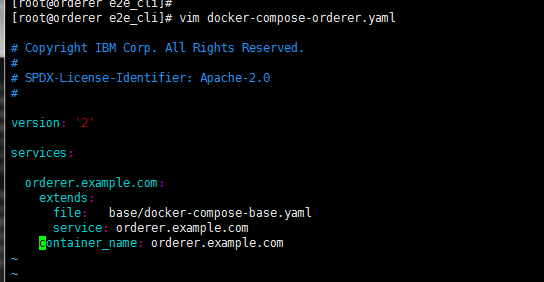
[root@orderer e2e\_cli]# ./generateArtifacts.sh mychannel #生成公私钥、证书、创世区块等





## 创建修改orderer文件

[root@orderer e2e\_cli]# cp docker-compose-cli.yaml docker-compose-orderer.yaml



docker-compose-orderer.yaml 文件是定义了镜像docker配置文件

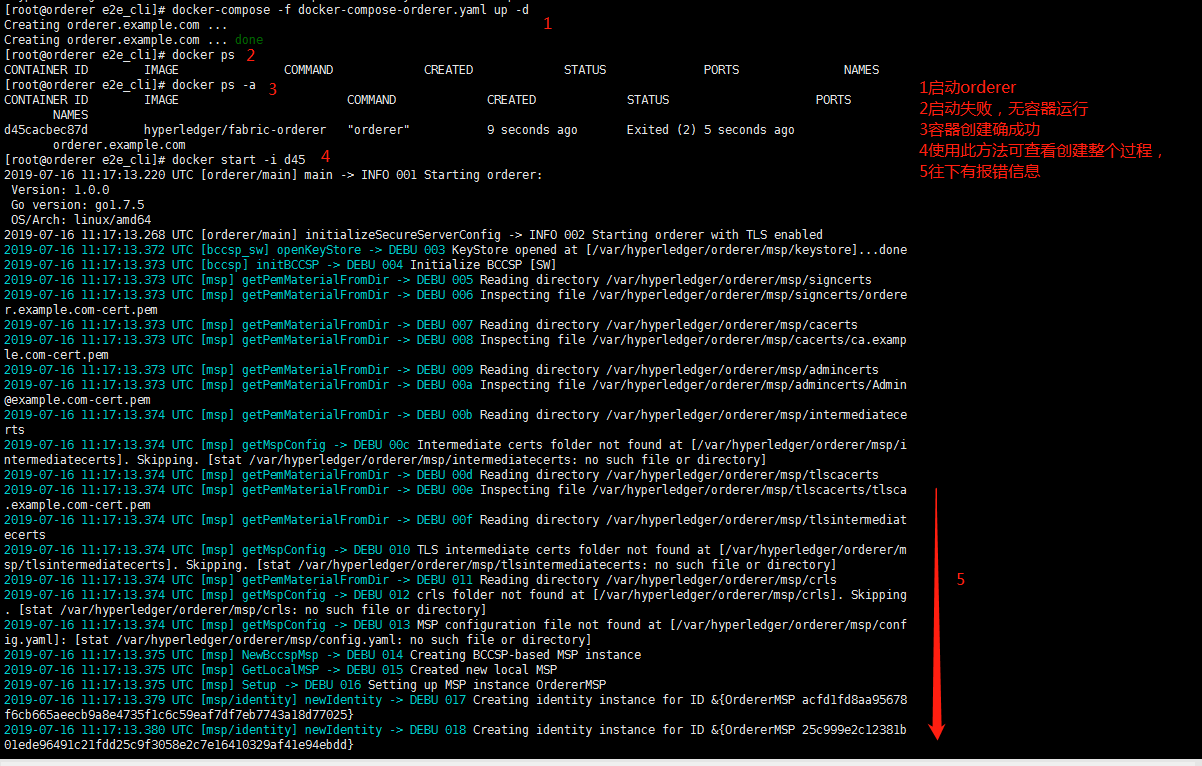


[root@orderer e2e\_cli]# docker-compose -f docker-compose-orderer.yaml up -d #启动orderere

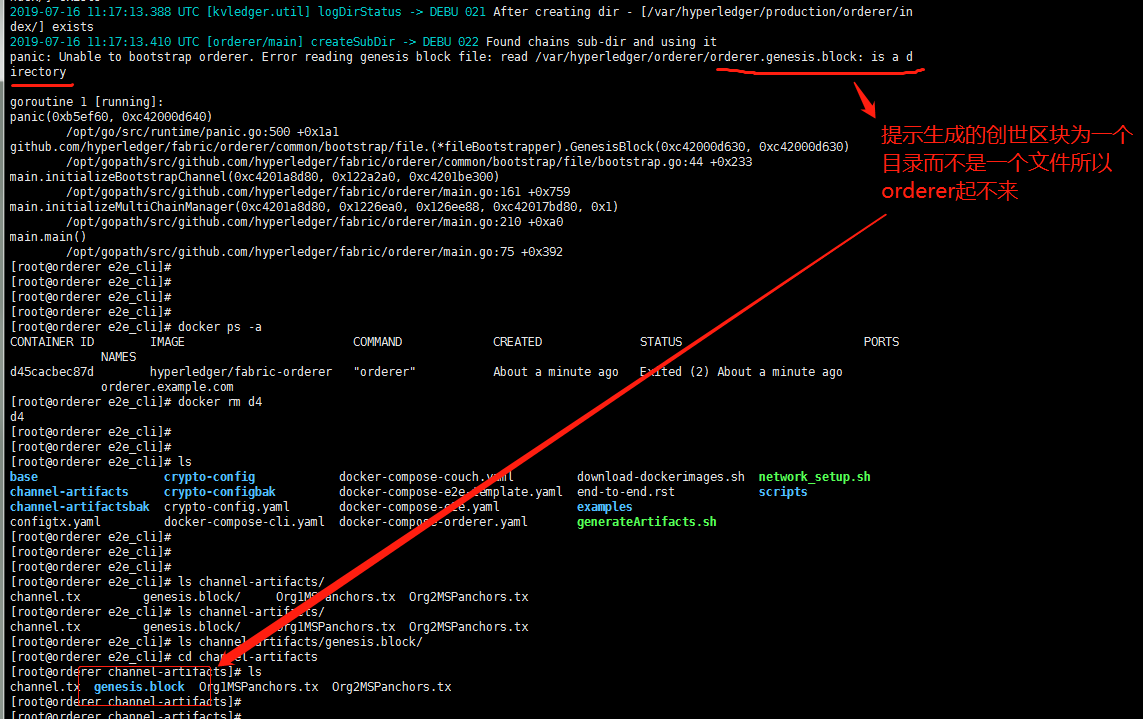


## 问题：

如果orderer起不来，则可以借鉴如下情况：



报错如下



解决办法，删掉这个目录，重新创建，成为目录原因暂时未知

# E2E案例实施之1.4.0

Fabric官方提供了实现点对点的Fabric网络示例，该网络有两个组织（organizations），一个组织有两种节点（Peer）,通过Kafka方式实现排序（Orderer）服务。

End-2-End案例的运行需要“cryptogen”和“configtxgen”两个工具，用于Fabric网络所需的数字证书验证和访问控制功能。

cryptogen：生成用于识别和验证网络中各种组件的x509证书。

configtxgen：生成用于通道和区块所需要配置文件。

1、两个工具可以通过如下命令方式生成，在该End-2-End例子中已集成到generateArtifacts.sh这个文件（执行这个脚本文件亦是编译了fabric源码，从而在releas产生linux-amd64/bin目录，此目录中生成了二进制可执行文件），运行后自动生成，无需手动命令操作,两个文件生成到$GOPATH/src/github.com/hyperledger /release/linux-amd64/bin这个目录。

# cd $GOPATH/src/github.com/hyperledger/fabric

# make release #编译faric，注意需要安装gcc

# cd $GOPATH/src/github.com/hyperledger/release/linux-amd64/bin

2、二进制文件还可以通过以下网址直接访问下载（下载比较慢需要长时间），下载的就是二进制执行文件，地址：https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/fabric/hyperledger-fabric/，本书使用V1.4版本，对应文件为hyperledger-fabric-linux-amd64-1.4.0.tar.gz，下载完成后解压，获取bin目录。

1. 拷贝e2e\_cli源文件

由于Fabric 1.4开始删除了End-2-End案例，需要手动拷贝e2e\_cli目录到$GOPATH/src/github.com/hyperledger/fabric/examples目录下。End-2-End案例源文件访问https://github.com/dragon-lin/bookfile网页中的“书籍对应源码/第七章 官方End-2-End运行”目录中获取。

[root@fabric1 ~]# cd bookfile/书籍对应源码/第七章\ 官方End-2-End运行/

[root@fabric1 第七章 官方End-2-End运行]# ls

e2e\_cli #将此目录全部拷贝到/opt/gopath/src/github.com/hyperledger/fabric/examples/目录下

chmod -R 777 $GOPATH/src/github.com/hyperledger/fabric/examples/e2e\_cli #赋予权限

执行

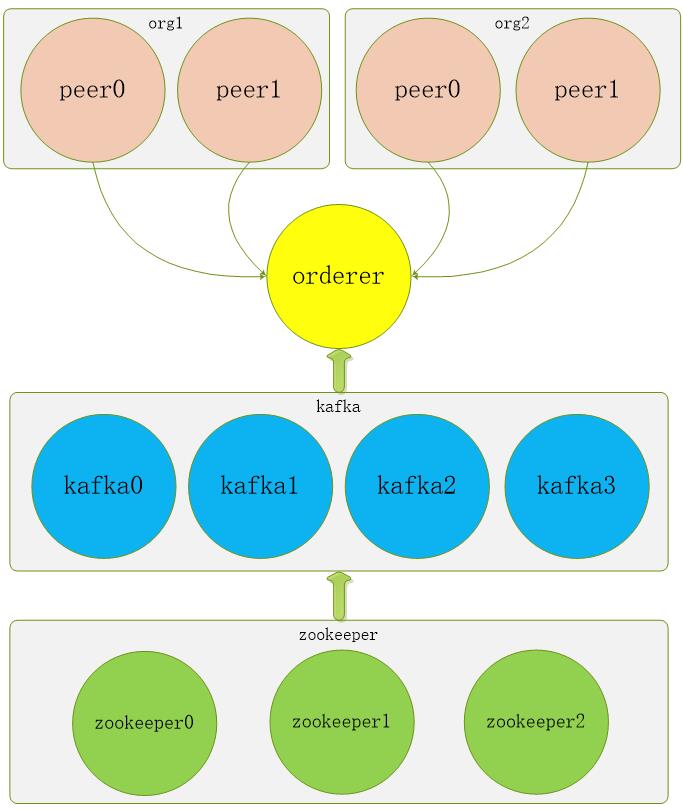
[root@fabric1 e2e\_cli]# ./generateArtifacts.sh #执行这个编译了fabric源码，在release中产生二进制文件

[root@fabric1 e2e\_cli]# ./network\_setup.sh up #启动e2e案例

[root@fabric1 e2e\_cli]# ./network\_setup.sh down #停止e2e案例

## End-2-End案例分析

案例架构：End-2-End案例由3个zookeeper、4个kafka和1个orderer实现排序,包括两个组织，分别为Org1和Org2，每个组织中有两个节点，分别为peer0和peer1,结构图如下所示：



所有的配置都在docker-compose-cli.yaml文件里，配置顺序分别为zookeeper、kafka、orderer、peer和cli，先运行zookeeper集群、再运行kafka集群，最后运行orderer和peer，必须按照以上运行顺序；实现的功能集中写在script.sh文件里，自动运行全部功能，直到显示成功，具体功能如下：

1. 验证排序（orderer）服务是否可用，函数：checkOSNAvailability

2. 创建通道，函数：createChannel

3. 加入通道，函数：checkOSNAvailability

4. 更新组织1的锚节点，函数：updateAnchorPeers

5. 更新组织2的锚节点，函数：updateAnchorPeers

6. 在组织1的节点0上安装智能合约，函数：installChaincode

7. 在组织2的节点0上安装智能合约，函数：installChaincode

8. 在组织2的节点0上实例化智能合约，函数：instantiateChaincode

9. 在组织1的节点0上查询智能合约，函数：chaincodeQuery

10. 从组织1的节点0向组织2的节点0转移数据10的交易，函数：chaincodeInvoke

11. 在组织2的节点1上安装智能合约，函数：installChaincode

12. 在组织2的节点1上查询智能合约，函数：chaincodeQuery

## E2E文件结构

End-2-End案例的全部文件在fabric/examples/e2e\_cli目录下，文件结构如下所示：

* base:目录，存放配置提炼的公有部分，有两个文件，分别为docker-compose-base.yaml和peer-base.yaml
* crypto-config.yaml：生成的公私钥和证书的配置文件
* docker-compose-e2e-template.yaml:
* end-to-end.rst:
* scripts：目录，只有一个script.sh文件，该文件是案例的运行功能的集合，运行后会自动执行全部功能，直到完成
* channel-artifacts：目录，存放生成的通道和创世纪块等文件，包括有channel.tx、genesis.block、Org1MSPanchors.tx和Org2MSPanchors.tx
* docker-compose-cli.yaml：Fabric网络Docker运行配置文件
* docker-compose-e2e.yaml:
* generateArtifacts.sh：生成公私钥和证书的执行文件
* configtx.yaml：通道配置文件
* docker-compose-couch.yaml:
* download-dockerimages.sh: 下载Fabric镜像执行文件
* network\_setup.sh: 案例运行的入口文件

## 执行流程：

Fabric基础环境搭建完成后，End-2-End案例的运行先从network\_setup.sh文件执行，执行过程中调用generateArtifacts.sh生成公私钥和证书等文件，再根据docker-compose-cli.yaml的配置内容通过docker运行zookeeper、kafka、orderer、peer和cli,最后在cli中运行script.sh文件，批量执行创建通道、加入通道、安装智能合约、实例化智能合约、执行交易和执行查询等功能，以上过程在没有错误的情况下，自动执行逐行执行，直到提示END-E2E表示成功。

流程说明：

1. 在e2e\_cli目录执行network\_setup.sh up表示开始执行，network\_setup.sh down表示结束执行；

2. 执行network\_setup.sh up后先判断是否存在crypto-config目录，如果不存在，则调用generateArtifacts.sh文件生成公私钥和证书；否则通过命令docker-compose -f $COMPOSE\_FILE up -d开始启动Fabric网络；

3. Fabric网络启动成功后，自动执行script.sh文件，按照代码顺序，分别执行如下代码：

1) 显示start-e2e：显示将开始执行案例；

2) checkOSNAvailability：执行peer channel fetch 0 0\_block.pb -o orderer.example.com:7050 -c "$ORDERER\_SYSCHAN\_ID" --tls --cafile $ORDERER\_CA >&log.txt命令，验证排序（orderer）服务是否可用；

3) createChannel：执行peer channel create -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/channel.tx --tls --cafile $ORDERER\_CA >&log.txt命令创建通道；

4) joinChannel：执行peer channel join -b $CHANNEL\_NAME.block >&log.txt命令四个peer节点加入到通道中；

5) updateAnchorPeers 0 1：执行peer channel update -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/${CORE\_PEER\_LOCALMSPID}anchors.tx --tls --cafile $ORDERER\_CA >&log.txt命令更新组织1的锚节点0；

6) updateAnchorPeers 0 2：执行peer channel update -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/${CORE\_PEER\_LOCALMSPID}anchors.tx --tls --cafile $ORDERER\_CA >&log.txt命令更新组织2的锚节点0；

7) installChaincode 0 1：执行peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/chaincode/go/example02/cmd >&log.txt命令在组织1的节点0上安装智能合约；

8) installChaincode 0 2：执行peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/chaincode/go/example02/cmd >&log.txt命令在组织2的节点0上安装智能合约；

9) instantiateChaincode 0 2：执行peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile $ORDERER\_CA -C $CHANNEL\_NAME -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.peer')" >&log.txt在组织2的节点0上实例化智能合约，初始化a值为100和b值为200；

10) chaincodeQuery 0 1 100：执行peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}' >&log.txt命令在组织1的节点0上查询a值，并判断是否为100；

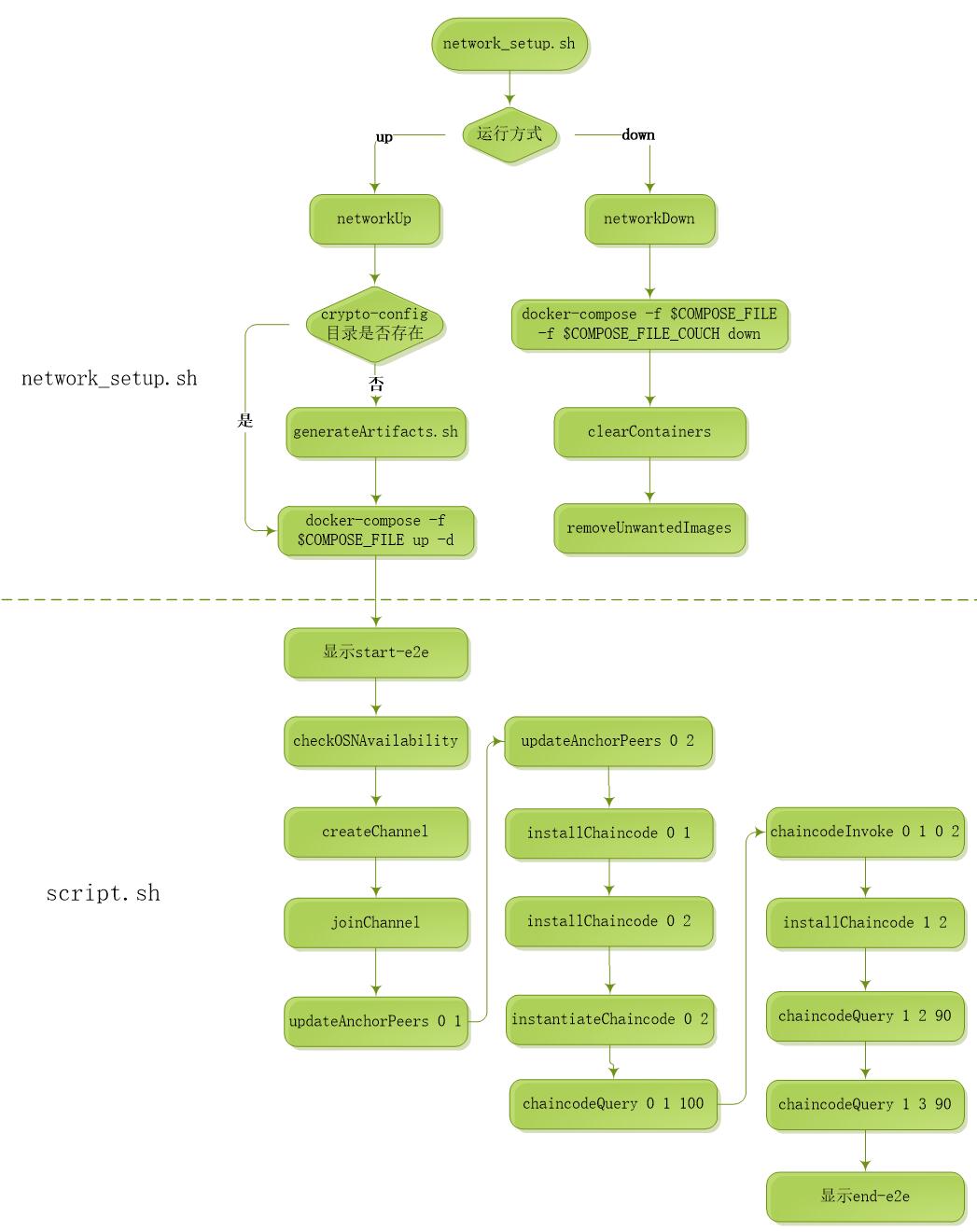
11) chaincodeInvoke 0 1 0 2：执行peer chaincode invoke -o orderer.example.com:7050 --tls --cafile $ORDERER\_CA -C $CHANNEL\_NAME -n mycc $PEER\_CONN\_PARMS -c '{"Args":["invoke","a","b","10"]}' >&log.txt命令从a值中转称10到a值中；

12) installChaincode 1 2：执行peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/chaincode/go/example02/cmd >&log.txt命令在组织2的节点1上安装智能合约；

13) chaincodeQuery 1 2 90：执行peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}' >&log.txt命令在组织2的节点1上查询a值，并判断是否为90；

14) chaincodeQuery 1 3 90：执行peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}' >&log.txt命令在组织3的节点1上查询a值，并判断是否为90；

15) 显示end-e2e：以上代码执行没有出现错误，则显示end-e2e表示成功执行；



# Helloworld案例简介

参考网站： https://www.cnblogs.com/llongst/p/9543574.html

通过执行官方End-2-End案例，初始了解Fabric网络的运行流程及yaml配置，官方End-2-End案例把执行过程集成，通过一条命令即可完成全部操作，对于初学者只能了解Fabric网络搭建是否成功，对于Fabric网络的执行细节还是迷惑。

为了能让初学者全面了解Fabric网络的执行细节，本章通过手动方式搭建一个orderer、一个组织和一个peer的SOLO排序的Fabric网络，把配置独立出来，形成orderer和peer配置等单个yaml文件，通过手动执行orderer和peer搭建Fabric网络。

编写最简单的智能合约，初始化时在区块中存储Hello world字符串，然后通过智能合约可以查询出Hello world字符串，初步了解智能合约编写。

## Helloworld链码编写

Helloworld链码实现Init和Invoke两个接口，通过stub.PutState和stub.GetState保存和获取链值对数据。

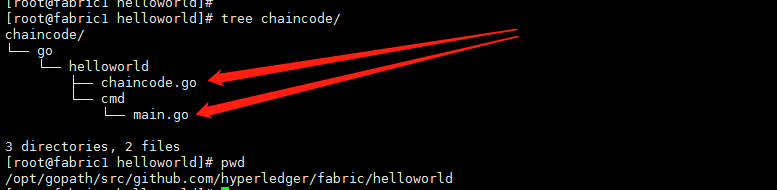
Init(stub shim.ChaincodeStubInterface)：用于智能合约初始化及升级初始化,实现初始化时保存链值对；

Invoke(stub shim.ChaincodeStubInterface)：是节点（peer）调用链码的入口函数，实现对账本进行保存和获取链值对；

　　实现两个文件,分别chaincode.go和cmd/ main.go，main.go是主入口函数

## Helloword具体操作步骤如下：

1、准备helloworld链码备用，拷贝编写好的智能合约文件到helloworld/chaincode/go/helloworld目录下。具体代码参考网站或者样机的代码，文件结构如下，helloword文件自行创建



1. 获取“cryptogen”和“configtxgen”工具

第一种方法：直接网上下载，链接如下

https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/fabric/hyperledger-fabric/linux-amd64-1.2.0 解压至helloword目录，有bin目录（执行文件）与config目录（配置模板文件）（本次实使用第一种方法）

第二种方法：编译源码，将$GOPATH/src/github.com/hyperledger/fabric/release/linux-amd64/bin/ 目录拷贝至helloword目录内

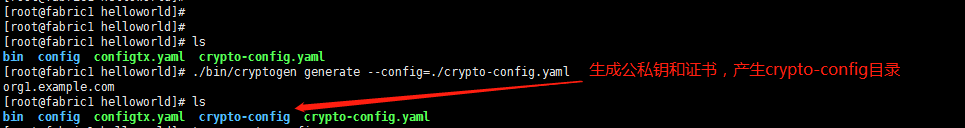
1. 准备生成证书和区块配置文件

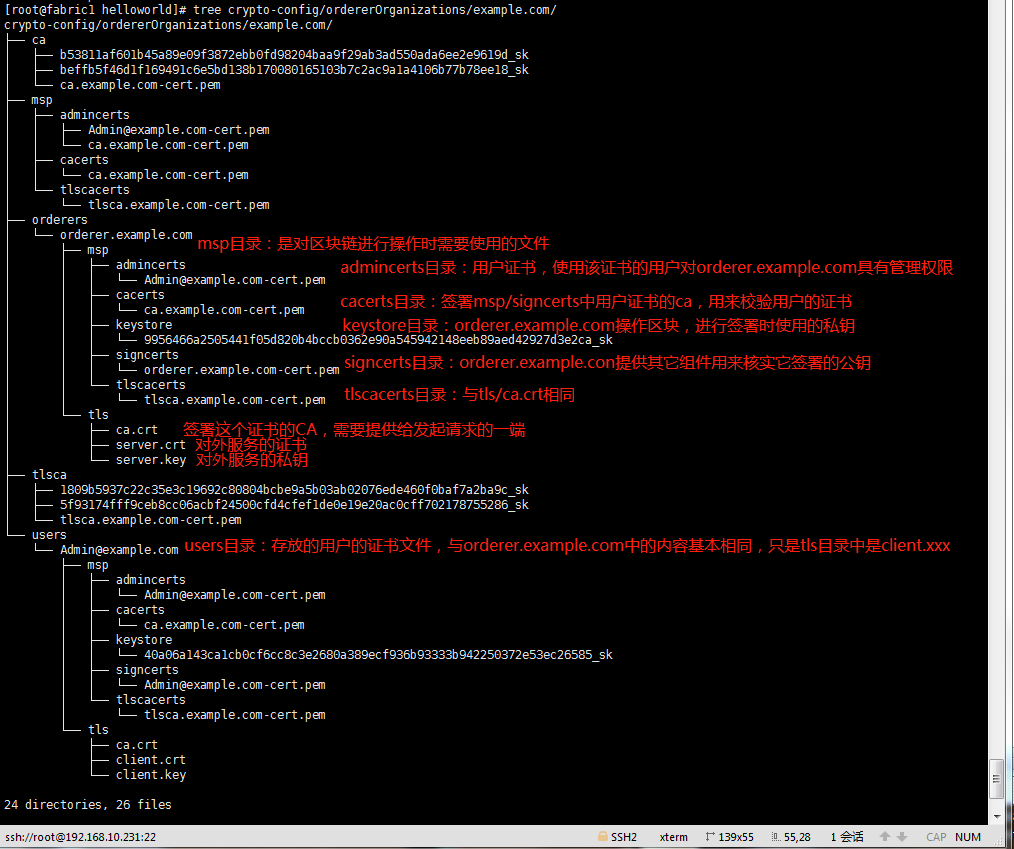
配置crypto-config.yaml和configtx.yaml文件，拷贝到helloworld目录下。然后按照参考网站修改

crypto-config.yaml文件：生成公私钥和证书的配置文件 configtx.yaml文件：通道配置文件

1. 生成公私钥和证书

./bin/cryptogen generate --config=./crypto-config.yaml #会多了crypto-config目录





certs目录中的文件留着备用。

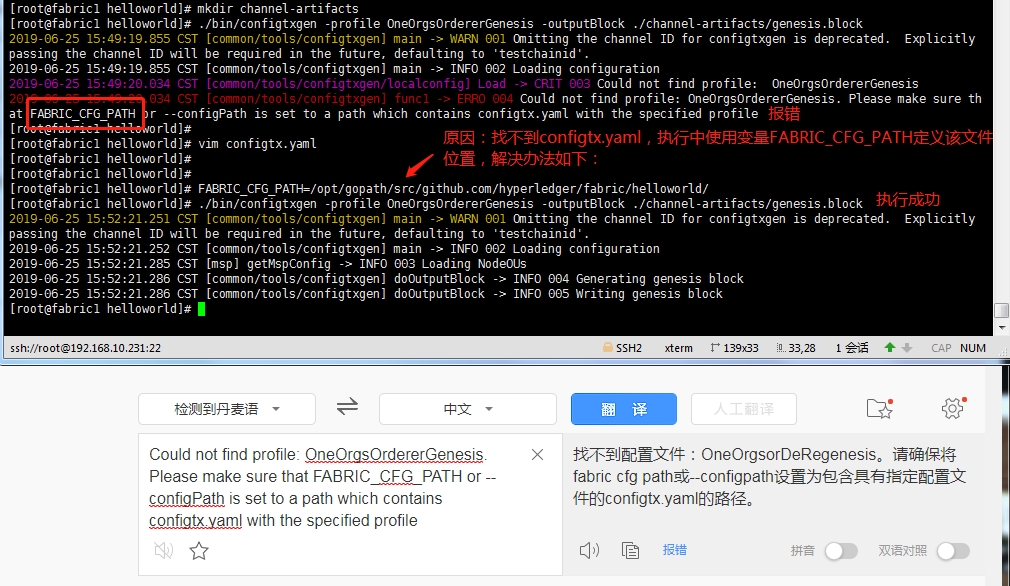
1. 生成创世区块

# mkdir channel-artifacts #创建通道目录

# ./bin/configtxgen -profile OneOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block

# ls channel-artifacts/

genesis.block #创世区块文件



1. 生成通道（channel）配置区块

./bin/configtxgen -profile OneOrgsChannel -outputCreateChannelTx ./channel-artifacts/mychannel.tx -channelID mychannel

# ls channel-artifacts/

genesis.block mychannel.tx # 产生mychannel.tx文件，为通道配置区块文件

1. 准备docker配置文件

配置docker-orderer.yaml和 docker-peer.yaml文件，拷贝到helloworld目录下，按照参考网站修改

9.3.9 启动Fabric网络

# docker-compose -f docker-orderer.yaml up -d #启动orderer

# docker-compose -f docker-peer.yaml up –d #启动peer

# docker exec -it cli bash #启动进入，cli容器

# 多节点部署准备：

<https://www.cnblogs.com/llongst/p/9571363.html> 整个部署参考网站

架构图：

- "orderer.example.com:192.168.10.231"

- "peer0.org1.example.com:192.168.10.232"

- "peer1.org1.example.com:192.168.10.233"

- "peer0.org2.example.com:192.168.10.234"

- "peer1.org2.example.com:192.168.10.235"

部署orderer.example.com

地址:192.168.10.231

# cd /opt/gopath/src/github.com/hyperledger/fabric

# mkdir multipeer

# cd multipeer

防火墙配置：

[root@peer0 multipeer]# firewall-cmd --get-default-zone

[root@peer0 multipeer]# firewall-cmd --zone=public --add-port=7051/tcp --permanent

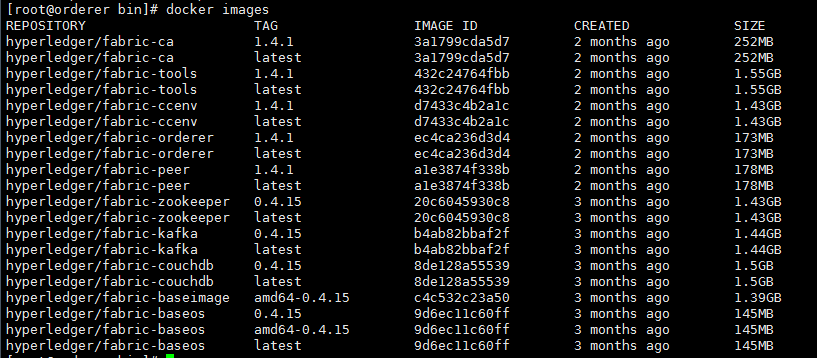
[root@peer0 multipeer]# firewall-cmd --zone=public --add-port=7052/tcp --permanent

[root@peer0 multipeer]# firewall-cmd --zone=public --add-port=7053/tcp --permanent

[root@peer0 multipeer]# firewall-cmd --reload

[root@peer0 multipeer]# docker exec -it cli bash

Docker文件：

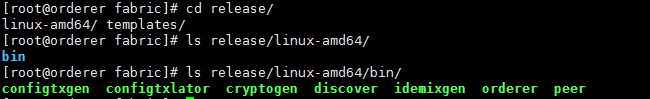


## Orderer部署

### 获取生成工具（二进制可执行文件）

在fabric源码目录下，make release编译，编译成功后将会在release目录内产生linux-adm64/bin目录，二进制可执行文件在此目录中

[root@orderer fabric]# make release

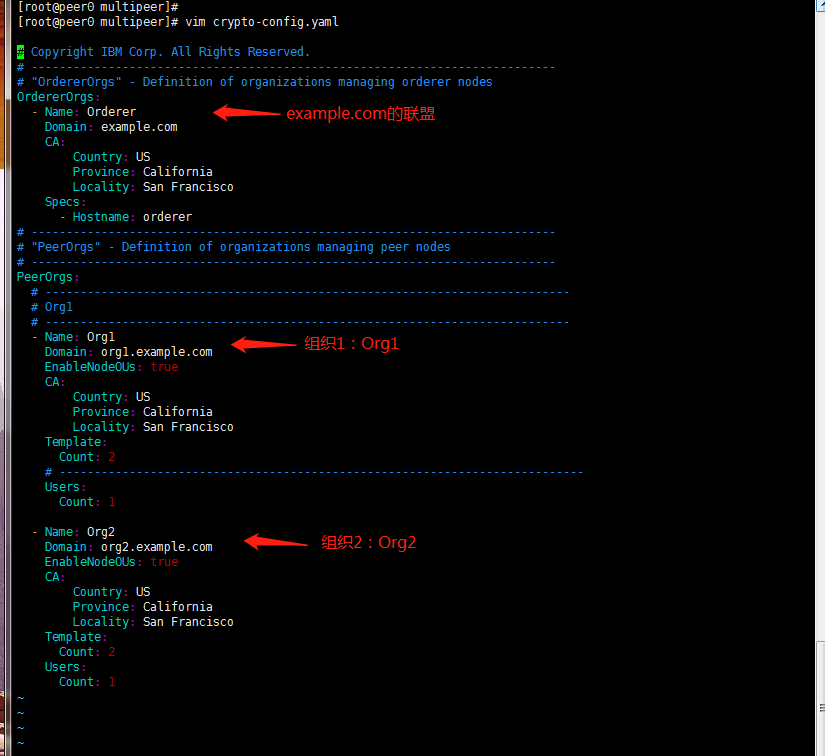


[root@orderer fabric]# cp -r release/linux-amd64/bin multipeer/ #将bin目录全部拷贝到multipeer路径下

### 准备生成证书和区块配置文件

配置crypto-config.yaml(生成公私钥和证书的文件)和configtx.yaml(生成创世区块和生产通道配置区块的文件)，拷贝到multipeer目录下。

[root@peer0 multipeer]# vim crypto-config.yaml



生成公私钥和证书：

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

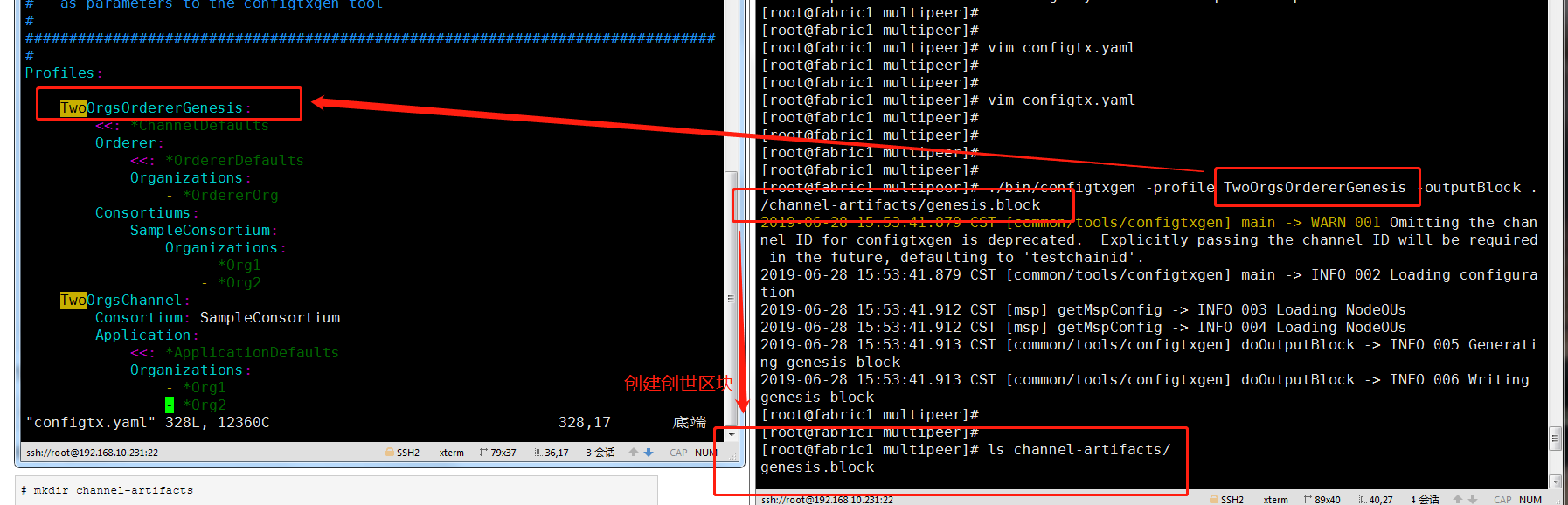
#multipeer目录下会多了crypto-config目录，具体可参考helloworld配置

### 生成创世区块

# mkdir channel-artifacts #创建channel-artifacts目录，创世区块与通道配置区块将在此目录中

# ./bin/configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block

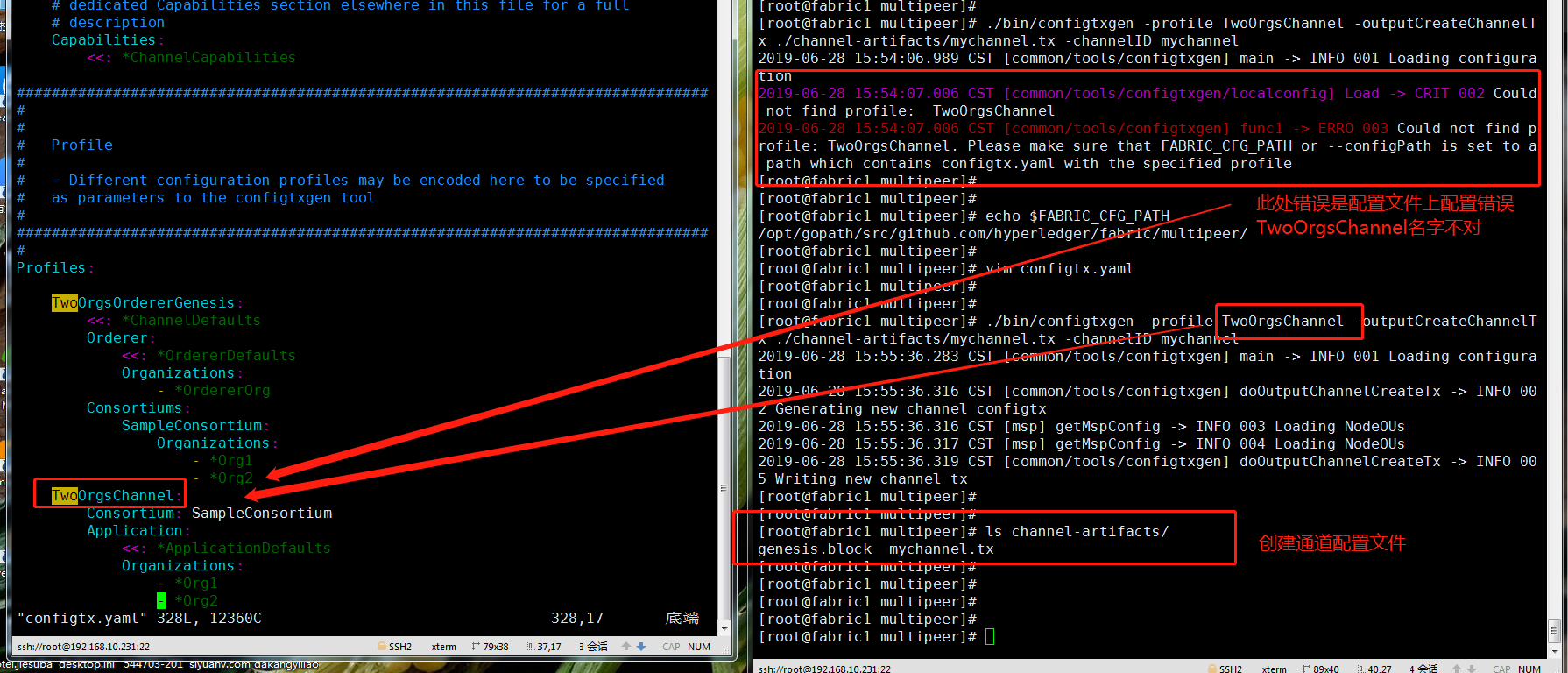
生成创世区块如下图



### 生成通道配置区块

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

生成通道配置区块如下图



### 拷贝生成文件到其它电脑

# cd ..

# scp -r multipeer root@192.168.10.232:/opt/gopath/src/github.com/hyperledger/fabric

# scp -r multipeer root@192.168.10.233:/opt/gopath/src/github.com/hyperledger/fabric

# scp -r multipeer root@192.168.10.234:/opt/gopath/src/github.com/hyperledger/fabric

# scp -r multipeer root@192.168.10.235:/opt/gopath/src/github.com/hyperledger/fabric

### 准备docker配置文件

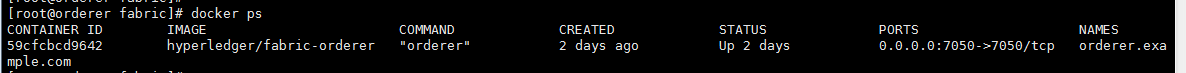
配置docker-compose-orderer.yaml文件，拷贝到multipeer目录下。

docker-compose-orderer.yaml文件：定义了启动的docker容器的配置，用哪个镜像文件启动什么样的容器，

### 启动Fabric网络

[root@orderer fabric]# docker-compose -f docker-compose-orderer.yaml up –d #

[root@orderer fabric]# docker ps



## Peer0.org1.example.com部署

192.168.10.232

### 准备docker配置文件

配置docker-compose-peer.yaml文件，拷贝到multipeer目录下。

docker-compose-peer.yaml：

### 部署智能合约

智能合约放在：multipeer/chaincode/go/example02目录下，本案例是用源码的：examples/chaincode/go/example02

将源码的：examples/chaincode/go/example02 拷贝到multipeer/chaincode/go/example02目录下

### 启动fabric网络

# cd $GOPATH/src/github.com/hyperledger/fabric/multipeer

# docker-compose -f docker-compose-peer.yaml up –d #1启动peer

# docker exec -it cli bash #2启动进入cli容器

#### 创建channel（mychannel.block）

#ORDERER\_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem

# peer channel create -o orderer.example.com:7050 -c mychannel -f ./channel-artifacts/mychannel.tx --tls --cafile $ORDERER\_CA #3 创建Channel：容器中产生mychannel.block

#### Peer加入Channel

# peer channel join -b mychannel.block

#### 退出容器保存mychannel.block到本机

# exit #退出容器

# docker cp xxxxxxxx:/opt/gopath/src/github.com/hyperledger/fabric/peer/mychannel.block /opt/gopath/src/github.com/hyperledger/fabric/multipeer #命令的xxxxxxxx为cli容器的ID

### mychannel.block拷贝到其他主机：

# scp mychannel.block root@192.168.10.233:/opt/gopath/src/github.com/hyperledger/fabric/multipeer

# scp mychannel.block root@192.168.10.234:/opt/gopath/src/github.com/hyperledger/fabric/multipeer

# scp mychannel.block root@192.168.10.235:/opt/gopath/src/github.com/hyperledger/fabric/multipeer

### 安装与运行智能合约

智能合约在“部署智能合约”那一步骤已经将合约放在./multipeer/chaincode/go/example02/

# docker exec -it cli bash

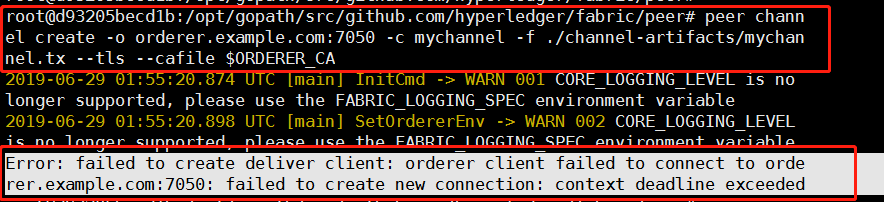
# peer chaincode install -n mycc -p github.com/hyperledger/fabric/multipeer/chaincode/go/example02/ -v 1.0

### 实例化合约失败

### 问题汇总：

问题1

无法连接



Fabric1.4用的是1.2的二进制文件，，，解决办法：使用直接编译源码获得的bin目录里面的二进制执行文件

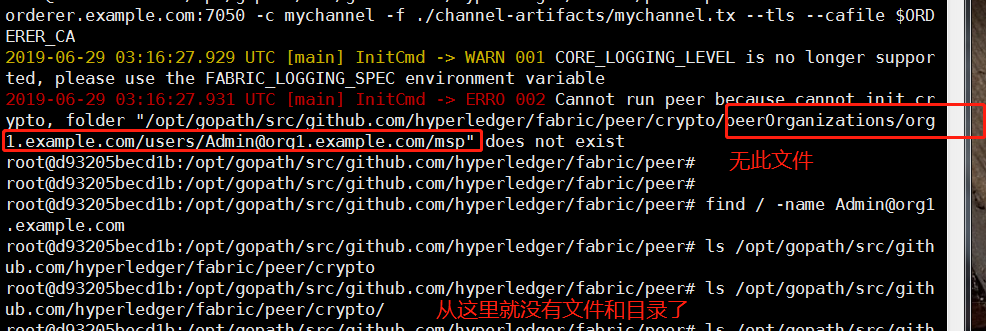
make release

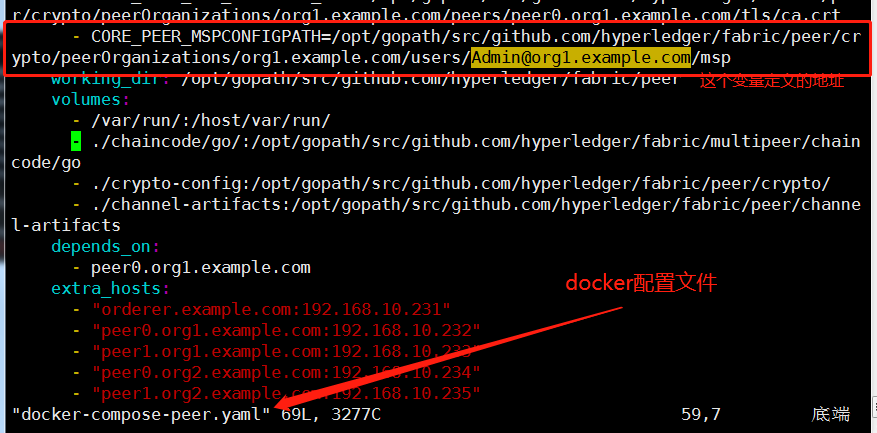
问题2

（已解决，，原因：docker镜像是1.4.1，fabric源码包是1.4.0，将fabric源码包换成1.4.1即可解决）

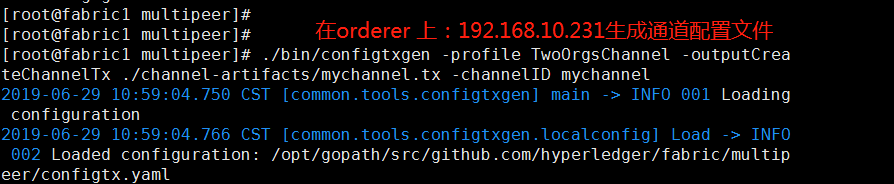
创建channl时

#peer channel create -o orderer.example.com:7050 -c mychannel -f ./channel-artifacts/mychannel.tx --tls --cafile $ORDERER\_CA

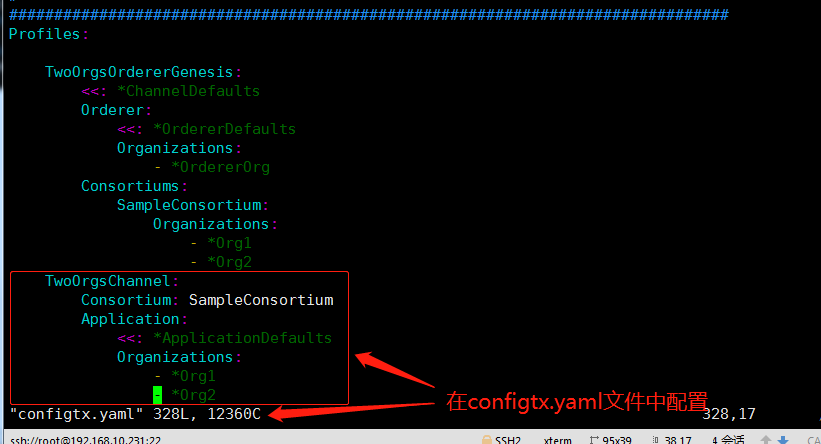




./channel-artifacts/mychannel.tx文件来源：在orderer上

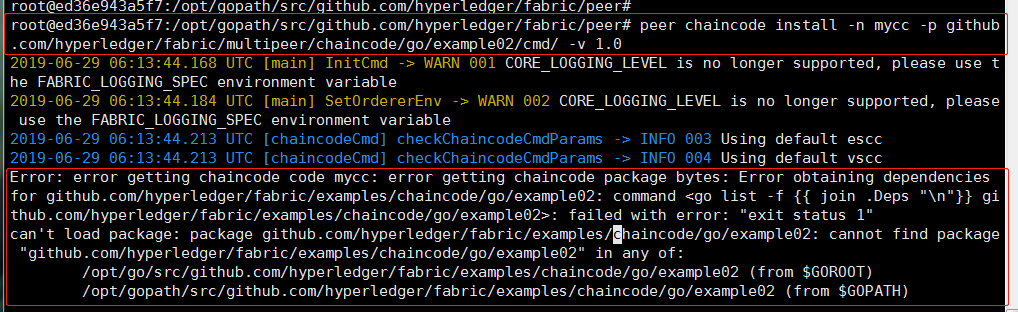


TwoOrgsChannel 所在

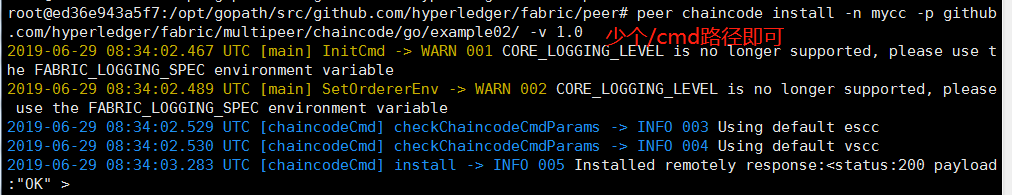


问题3

安装智能合约时候出错：

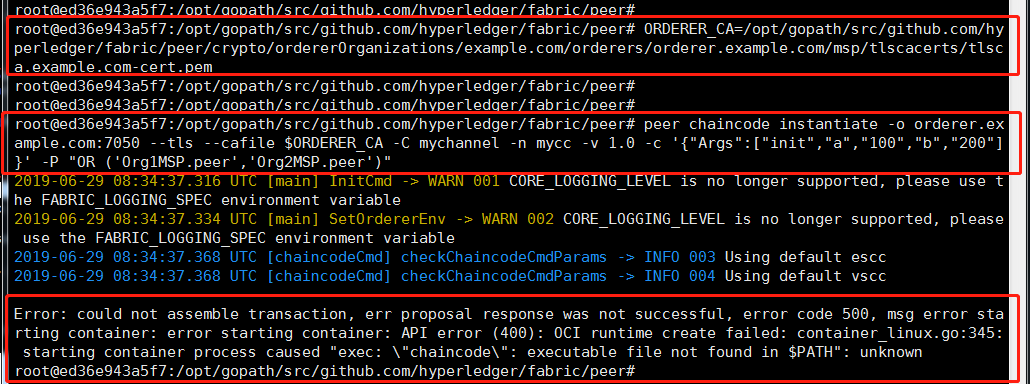


解决如下：



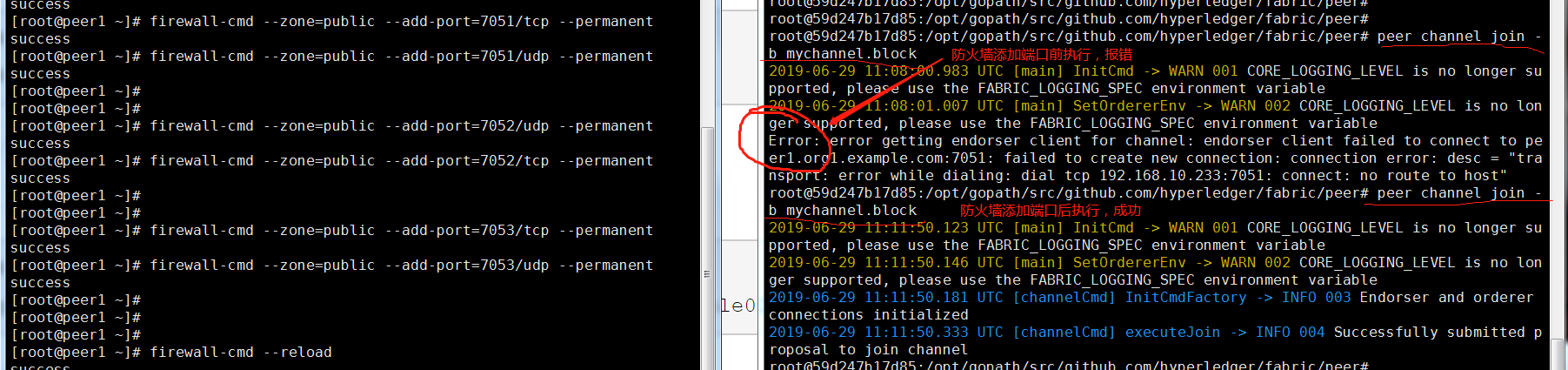
问题4：(未解决)

实例化智能合约失败：



将fabirc-ccenv、fabric-baseos、fabric-javaenv三个镜像提前下载好以后，实例化成功。

问题三：防火墙设置



## peer1.org1.example.com部署

192.168.10.233

### 准备docker配置文件

配置docker-compose-peer.yaml文件，拷贝到multipeer目录下，可以peer0.org1.example.com的为模板，将peer0改为peer1

[root@peer1 multipeer]# sed -i '1,65s/peer0.org1.example.com/peer1.org1.example.com/' docker-compose-peer.yaml

[root@peer1 multipeer]# vim docker-compose-peer.yaml

------------peer1.org1.example.com的docker-compose-peer.yaml配置内容如下-------------------

#docker-peer.yaml：

version: '2'

- CORE\_PEER\_ADDRESS=peer1.org1.example.com:7051

- CORE\_PEER\_CHAINCODEADDRESS=peer1.org1.example.com:7052

- CORE\_PEER\_CHAINCODELISTENADDRESS=0.0.0.0:7052

- CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT=peer1.org1.example.com:7051

- CORE\_PEER\_LOCALMSPID=Org1MSP

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

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

#- CORE\_LOGGING\_LEVEL=ERROR

- CORE\_LOGGING\_LEVEL=DEBUG

- CORE\_PEER\_TLS\_ENABLED=true

- CORE\_PEER\_GOSSIP\_USELEADERELECTION=true

- CORE\_PEER\_GOSSIP\_ORGLEADER=false

- CORE\_PEER\_PROFILE\_ENABLED=true

- CORE\_PEER\_TLS\_CERT\_FILE=/etc/hyperledger/fabric/tls/server.crt

#docker-peer.yaml：

version: '2'

- CORE\_PEER\_ADDRESS=peer1.org1.example.com:7051

- CORE\_PEER\_CHAINCODEADDRESS=peer1.org1.example.com:7052

- CORE\_PEER\_CHAINCODELISTENADDRESS=0.0.0.0:7052

- CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT=peer1.org1.example.com:7051

- CORE\_PEER\_LOCALMSPID=Org1MSP

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

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

#- CORE\_LOGGING\_LEVEL=ERROR

- CORE\_LOGGING\_LEVEL=DEBUG

- CORE\_PEER\_TLS\_ENABLED=true

- CORE\_PEER\_GOSSIP\_USELEADERELECTION=true

- CORE\_PEER\_GOSSIP\_ORGLEADER=false

- CORE\_PEER\_PROFILE\_ENABLED=true

- CORE\_PEER\_TLS\_CERT\_FILE=/etc/hyperledger/fabric/tls/server.crt

- CORE\_PEER\_TLS\_KEY\_FILE=/etc/hyperledger/fabric/tls/server.key

- CORE\_PEER\_TLS\_ROOTCERT\_FILE=/etc/hyperledger/fabric/tls/ca.crt

working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer

command: peer node start

volumes:

- /var/run/:/host/var/run/

- ./crypto-config/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/msp:/etc/hyperledger/fabric/msp

- ./crypto-config/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls:/etc/hyperledger/fabric/tls

ports:

- 7051:7051

- 7052:7052

- 7053:7053

extra\_hosts:

- "orderer.example.com:192.168.10.231"

cli:

container\_name: cli

image: hyperledger/fabric-tools

tty: true

environment:

- GOPATH=/opt/gopath

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

- CORE\_LOGGING\_LEVEL=DEBUG

- CORE\_PEER\_ID=cli

- CORE\_PEER\_ADDRESS=peer1.org1.example.com:7051

- CORE\_PEER\_LOCALMSPID=Org1MSP

- CORE\_PEER\_TLS\_ENABLED=true

- CORE\_PEER\_TLS\_CERT\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/server.crt

- CORE\_PEER\_TLS\_KEY\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/server.key

- CORE\_PEER\_TLS\_ROOTCERT\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/ca.crt

- CORE\_PEER\_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp

working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer

volumes:

- /var/run/:/host/var/run/

- ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/multipeer/chaincode/go

- ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/

- ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/channel-artifacts

depends\_on:

- peer1.org1.example.com

extra\_hosts:

- "orderer.example.com:192.168.10.231"

- "peer0.org1.example.com:192.168.10.232"

- "peer1.org1.example.com:192.168.10.233"

- "peer0.org2.example.com:192.168.10.234"

- "peer1.org2.example.com:192.168.10.235"

-------------peer1.org1.example.com的docker-compose-peer.yaml配置结束-------------------

### 部署智能合约

智能合约放在：multipeer/chaincode/go/example02目录下，本案例是用源码的：examples/chaincode/go/example02

将源码的：examples/chaincode/go/example02 拷贝到multipeer/chaincode/go/example02目录下

### 启动fabric网络

# cd $GOPATH/src/github.com/hyperledger/fabric/multipeer

# docker-compose -f docker-compose-peer.yaml up –d #1启动peer

# docker exec -it cli bash #2启动进入cli容器

#### 部署mychannel.block到cli容器中

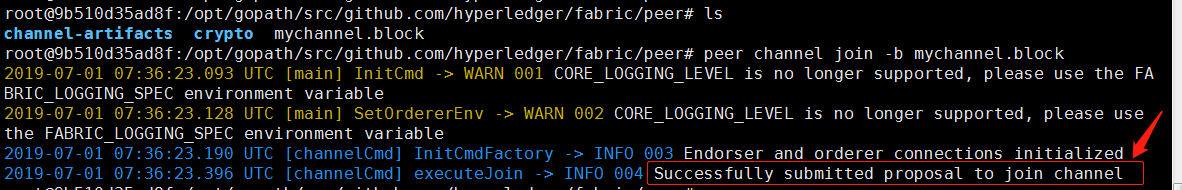
# exit

# docker cp /opt/gopath/src/github.com/hyperledger/fabric/multipeer/mychannel.block xxxxxxxx:/opt/gopath/src/github.com/hyperledger/fabric/peer/ #命令的xxxxxxxx为cli容器的ID

#### Peer加入Channel

# docker exec -it cli bash

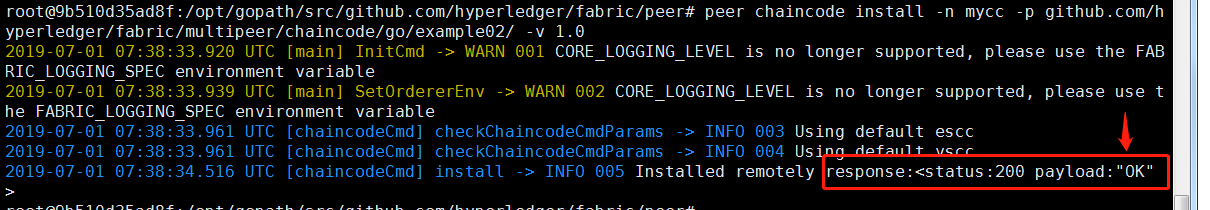
# peer channel join -b mychannel.block



### 安装与运行智能合约

智能合约在“部署智能合约”那一步骤已经将合约放在./multipeer/chaincode/go/example02/

# peer chaincode install -n mycc -p github.com/hyperledger/fabric/multipeer/chaincode/go/example02/ -v 1.0



### 运行还有问题待解决

## Peer0.org2.example.com

[root@peer0 multipeer]# vim docker-compose-peer.yaml

[root@peer0 multipeer]# sed -i '1,65s/peer0.org1.example.com/peer0.org2.example.com/' docker-compose-peer.yaml

[root@peer1 multipeer]# firewall-cmd --zone=public --add-port=7051/tcp --permanent

success

[root@peer1 multipeer]# firewall-cmd --zone=public --add-port=7052/tcp --permanent

success

[root@peer1 multipeer]# firewall-cmd --zone=public --add-port=7053/tcp –permanent

Success

[root@peer1 multipeer]# firewall-cmd --reload

[root@peer0 multipeer]# mkdir -p chaincode/go/

[root@peer0 multipeer]# cp -r ../examples/chaincode/go/example02 chaincode/go/

[root@peer0 multipeer]# docker-compose -f docker-compose-peer.yaml up -d

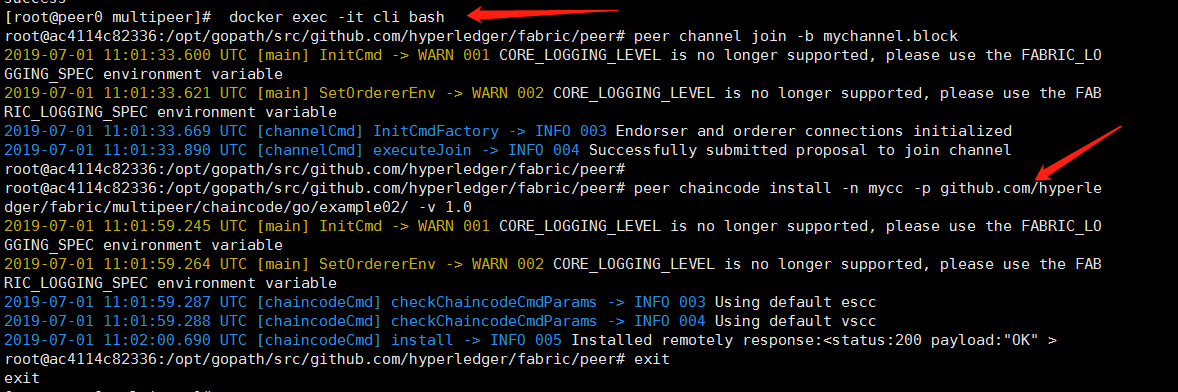
[root@peer0 multipeer]# docker exec -it cli bash

root@ac4114c82336:/opt/gopath/src/github.com/hyperledger/fabric/peer# exit

[root@peer0 multipeer]# docker cp mychannel.block ac4114c82336:/opt/gopath/src/github.com/hyperledger/fabric/peer/

root@ac4114c82336:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer channel join -b mychannel.block

root@ac4114c82336:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode install -n mycc -p github.com/hyperledger/fabric/multipeer/chaincode/go/example02/ -v 1.0



## Peer1.org2.example.com

[root@peer1 multipeer]# cp /root/docker-compose-peer.yaml .

[root@peer1 multipeer]# sed -i '1,65s/peer0.org2.example.com/peer1.org2.example.com/' docker-compose-peer.yaml

[root@peer1 multipeer]# vim docker-compose-peer.yaml

[root@peer1 multipeer]# mkdir -p chaincode/go/

[root@peer1 multipeer]# firewall-cmd --zone=public --add-port=7051/tcp --permanent

success

[root@peer1 multipeer]# firewall-cmd --zone=public --add-port=7052/tcp --permanent

success

[root@peer1 multipeer]# firewall-cmd --zone=public --add-port=7053/tcp –permanent

Success

[root@peer1 multipeer]# firewall-cmd --reload

[root@peer1 multipeer]# docker-compose -f docker-compose-peer.yaml up –d

[root@peer1 multipeer]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

920d0ca31067 hyperledger/fabric-tools "/bin/bash" 8 seconds ago Up 6 seconds cli

44e7323804d5 hyperledger/fabric-peer "peer node start" 9 seconds ago Up 7 seconds 0.0.0.0:7051-7053->7051-7053/tcp peer1.org2.example.com

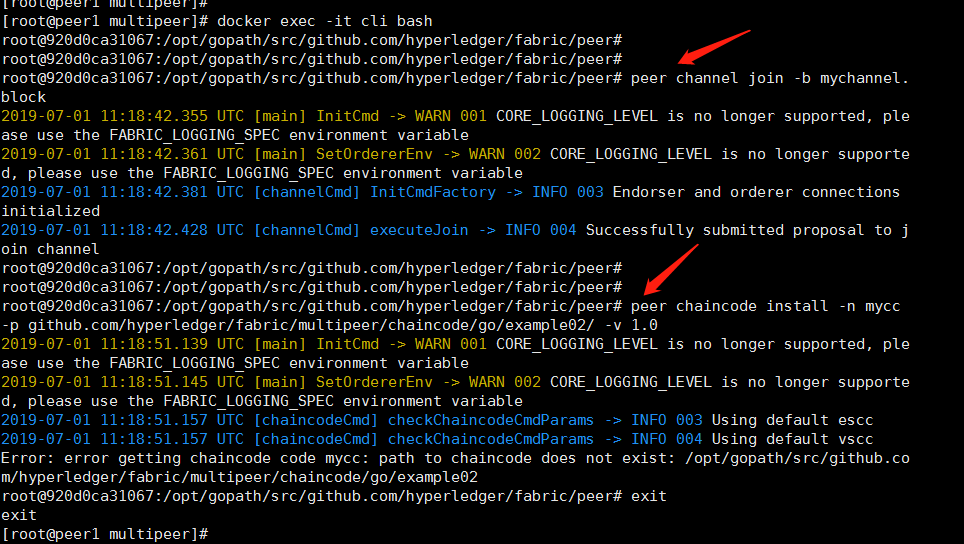
[root@peer1 multipeer]# docker cp mychannel.block 920d0ca31067:/opt/gopath/src/github.com/hyperledger/fabric/peer/

[root@peer1 multipeer]# docker cp mychannel.block 920d0ca31067:/opt/gopath/src/github.com/hyperledger/fabric/peer/

[root@peer1 multipeer]# docker exec -it cli bash

root@920d0ca31067:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer channel join -b mychannel.block

root@920d0ca31067:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode install -n mycc -p github.com/hyperledger/fabric/multipeer/chaincode/go/example02/ -v 1.0



# WBY的fabric环境

192.168.10.107 orderer

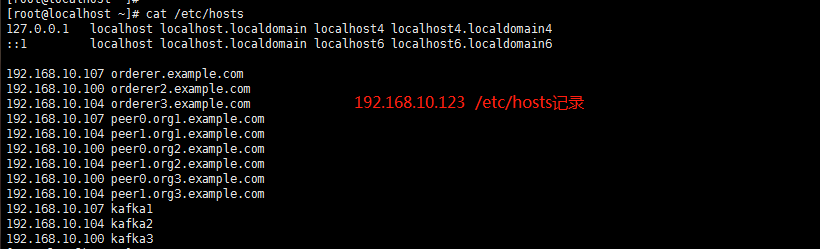
192.168.10.100 orderer2

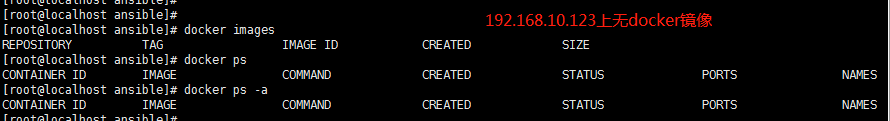
192.168.10.123 localhost

192.168.10.104 orderer3

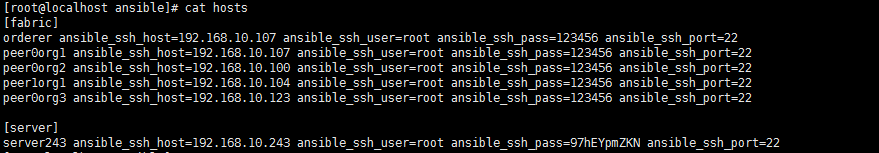
启动步骤（在192.168.10.123上从上倒下按顺序执行）：

192.168.10.123 相关环境





Ansible环境



启动:进入123的/etc/ansible目录顺序由上向下执行：

ansible-playbook docker-compose-couchdb.yml --tags docker-compose

ansible-playbook docker-compose-kafka.yml

ansible-playbook fabric-orderer.yml

ansible-playbook fabric-peer.yml

## 第一步：启动couchdb数据库

启动:进入123的/etc/ansible目录执行如下命令

ansible-playbook docker-compose-couchdb.yml --tags docker-compose #在/etc/ansible目录中执行

[root@localhost ansible]# pwd

/etc/ansible

[root@localhost ansible]# vim docker-compose-couchdb.yml

- hosts: peer0org2 peer1org1 peer0org1

remote\_user: root

tasks:

- name: copy couchdb.tar.gz

copy: src=fabric-couchdb.tar.gz dest=/opt/

- name: docker load couchdb

shell: docker load -i fabric-couchdb.tar.gz

args:

chdir: /opt/

- name: copy docker-compose-couchdb

copy: src=docker-compose-couchdb.yaml dest=/opt/

tags: docker-compose #执行1：拷贝文件

- name: docker-compose up

shell: docker-compose -f docker-compose-couchdb.yaml up -d

args:

chdir: /opt/ #进入远程主机/opt目录

tags: docker-compose #执行2：启动镜像

#remote\_user可以定义指定用户通过sudo的方法在被管理主机上运行指令，甚至可以在使用become指定sudo切换的用户。

#执行1：将123主机的/etc/ansible/files/docker-compose-couchdb.yaml 拷贝到peer0org2 peer1org1 peer0org1 三台主机的/opt目录(注意文件所在路径)

#执行2：根据拷贝到peer0org2 peer1org1 peer0org1 三台主机的/opt/docker-compose-couchdb.yaml 文件启动couchdb容器

三台主机启动couchdb的/opt/docker-compose-couchdb.yaml 文件内容如下

[root@orderer opt]# vim docker-compose-couchdb.yaml #peer0org2 peer1org1 peer0org1 三台主机配置相同

version: '2'

restart:

no是默认的重启策略，在任何情况下都不会重启容器。 指定为always时，容器总是重新启动。 如果退出代码指示出现故障错误，则on-failure将重新启动容器。

services:

couchdb:

container\_name: couchdb

restart: always

network\_mode: "host"

network\_mode:

Docker自身的4种网络工作方式，和一些自定义网络模式

安装Docker时，它会自动创建三个网络，bridge（创建容器默认连接到此网络）、 none 、host

host：容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。

Container：创建的容器不会创建自己的网卡，配置自己的IP，而是和一个指定的容器共享IP、端口范围。

None：该模式关闭了容器的网络功能。

Bridge：此模式会为每一个容器分配、设置IP等，并将容器连接到一个docker0虚拟网桥，通过docker0网桥以及Iptables nat表配置与宿主机通信。

image: hyperledger/fabric-couchdb:amd64-0.4.14

environment:

- COUCHDB\_USER=

- COUCHDB\_PASSWORD=

ports:

- "5984:5984"

volumes:

- /opt/couchdb/data:/opt/couchdb/data

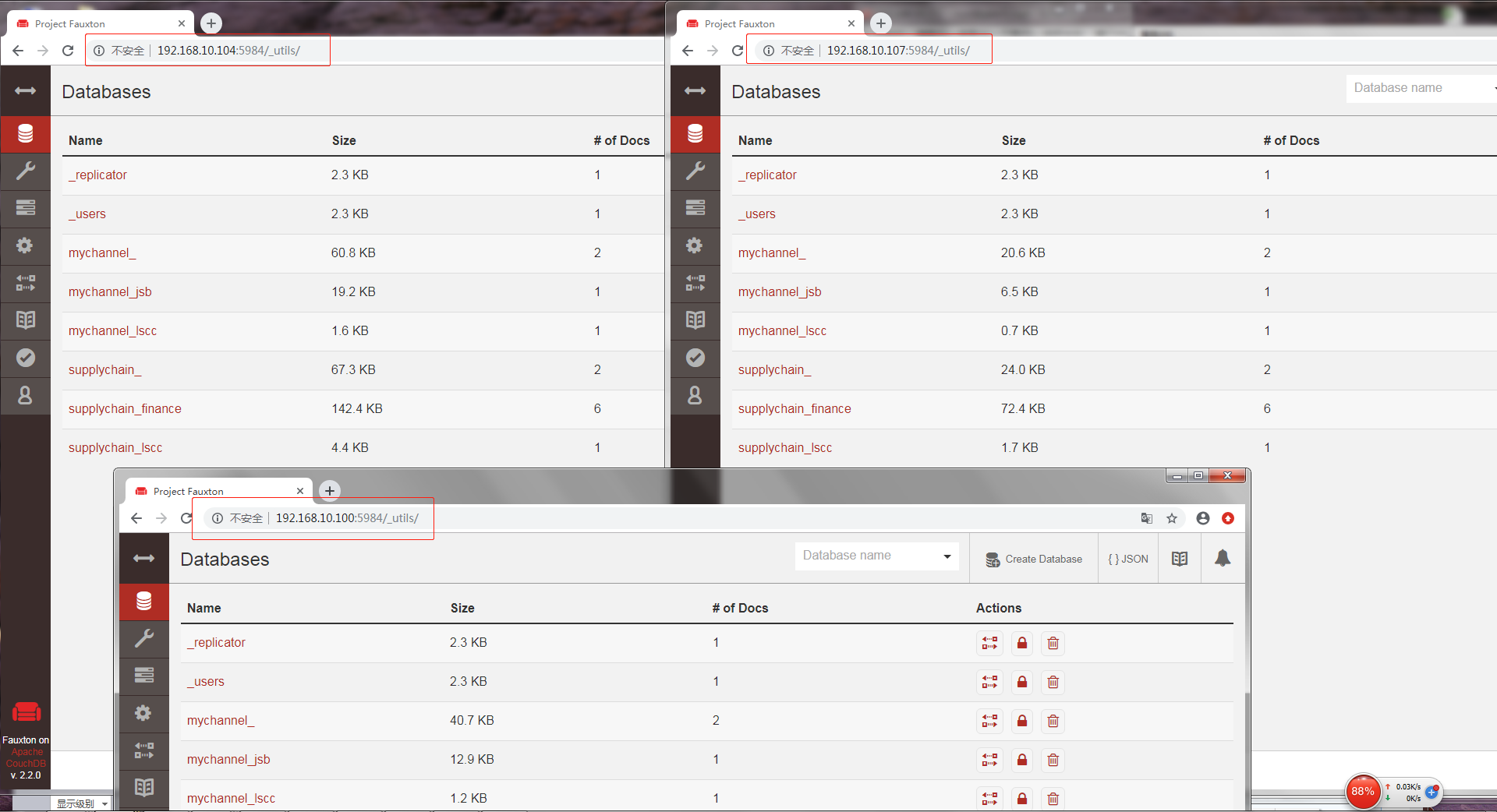
#docker stop $(docker ps -q) && docker rm $(docker ps -qa) 停止所有的容器，并删除

#启动后确保三台主机5984端口都开启，如果未开启则为启动失败，尝试重启docker然后再启动

http://192.168.10.107:5984/\_utils

http://192.168.10.100:5984/\_utils

http://192.168.10.104:5984/\_utils



Size大小不一样，但是name与# of Docs数量一致

## 第二步：启动zookeeper/kafka

启动:进入123的/etc/ansible目录执行如下命令

ansible-playbook docker-compose-kafka.yml

[root@localhost ansible]# pwd

/etc/ansible

[root@localhost ansible]# vim docker-compose-kafka.yml

- hosts: peer0org1

remote\_user: root

tasks:

- name: copy docker-compose-kafka.yaml

copy: src=docker-compose-kafka.yaml dest=/opt/

- name: docker up kafka1

shell: docker-compose -f docker-compose-kafka.yaml up -d kafka1

args:

chdir: /opt/

- hosts: peer1org1

remote\_user: root

tasks:

- name: copy docker-compose-kafka.yaml

copy: src=docker-compose-kafka.yaml dest=/opt/

- name: docker up kafka2

shell: docker-compose -f docker-compose-kafka.yaml up -d kafka2

args:

chdir: /opt/

- hosts: peer0org2

remote\_user: root

tasks:

- name: copy docker-compose-kafka.yaml

copy: src=docker-compose-kafka.yaml dest=/opt/

- name: docker up kafka3

shell: docker-compose -f docker-compose-kafka.yaml up -d kafka3

args:

chdir: /opt/

#1、三个hosts分别为三个节点，都是先将123上的/etc/ansible/files/docker-compose-kafka.yaml 文件拷贝到三个主机的/opt/ 目录中，

#2、然后启动依据拷贝而来的docker-compose-kafka.yaml文件，peer0org1启动zoo1、kafka1，peer1org1启动zoo2、kafka2，peer0org2启动zoo3、kafka3。

/etc/ansible/files/docker-compose-kafka.yaml 内容如下

[root@localhost ansible]# vim files/docker-compose-kafka.yaml

version: '2'

services:

zoo1:

image: hyperledger/fabric-zookeeper:amd64-0.4.14

restart: always

hostname: zoo1

container\_name: zoo1

network\_mode: "host"

ports:

- 2181:2181

volumes:

- /opt/zoo/:/data

- /opt/zoo/datalog:/datalog

environment:

ZOO\_MY\_ID: 1

ZOOKEEPER\_TICK\_TIME: 2000

ZOO\_SERVERS: server.1=0.0.0.0:2888:3888 server.2=kafka2:2888:3888 server.3=kafka3:2888:3888

zoo2:

image: hyperledger/fabric-zookeeper:amd64-0.4.14

restart: always

hostname: zoo2

container\_name: zoo2

network\_mode: "host"

ports:

- 2181:2181

volumes:

- /opt/zoo/:/data

- /opt/zoo/datalog:/datalog

environment:

ZOO\_MY\_ID: 2

ZOOKEEPER\_TICK\_TIME: 2000

ZOO\_SERVERS: server.1=kafka1:2888:3888 server.2=0.0.0.0:2888:3888 server.3=kafka3:2888:3888

zoo3:

image: hyperledger/fabric-zookeeper:amd64-0.4.14

restart: always

hostname: zoo3

container\_name: zoo3

network\_mode: "host"

ports:

- 2181:2181

volumes:

- /opt/zoo/:/data

- /opt/zoo/datalog:/datalog

environment:

ZOO\_MY\_ID: 3

ZOOKEEPER\_TICK\_TIME: 2000

ZOO\_SERVERS: server.1=kafka1:2888:3888 server.2=kafka2:2888:3888 server.3=0.0.0.0:2888:3888

kafka1:

image: hyperledger/fabric-kafka:amd64-0.4.14

restart: always

network\_mode: "host"

hostname: kafka1

container\_name: kafka1

depends\_on:

- zoo1

ports:

- 9092:9092

environment:

KAFKA\_ADVERTISED\_HOST\_NAME: kafka1

KAFKA\_MIN\_INSYNC\_REPLICAS: 1

KAFKA\_BROKER\_ID: 1

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 3

KAFKA\_MESSAGE\_MAX\_BYTES: 1048576

KAFKA\_REPLICA\_FETCH\_MAX\_BYTES: 1048576

KAFKA\_UNCLEAN\_LEADER\_ELECTION\_ENABLE: "false"

KAFKA\_LOG\_RETENTION\_MS: -1

KAFKA\_ADVERTISED\_PORT: 9092

KAFKA\_ZOOKEEPER\_CONNECT: kafka1:2181,kafka2:2181,kafka3:2181

volumes:

- /opt/kafka/:/kafka

kafka2:

image: hyperledger/fabric-kafka:amd64-0.4.14

restart: always

network\_mode: "host"

hostname: kafka2

container\_name: kafka2

depends\_on:

- zoo2

ports:

- 9092:9092

environment:

KAFKA\_ADVERTISED\_HOST\_NAME: kafka2

KAFKA\_MIN\_INSYNC\_REPLICAS: 1

KAFKA\_BROKER\_ID: 2

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 3

KAFKA\_MESSAGE\_MAX\_BYTES: 1048576

KAFKA\_REPLICA\_FETCH\_MAX\_BYTES: 1048576

KAFKA\_UNCLEAN\_LEADER\_ELECTION\_ENABLE: "false"

KAFKA\_LOG\_RETENTION\_MS: -1

KAFKA\_ADVERTISED\_PORT: 9092

KAFKA\_ZOOKEEPER\_CONNECT: kafka1:2181,kafka2:2181,kafka3:2181

volumes:

- /opt/kafka/:/kafka

kafka3:

image: hyperledger/fabric-kafka:amd64-0.4.14

restart: always

network\_mode: "host"

hostname: kafka3

container\_name: kafka3

depends\_on:

- zoo3

ports:

- 9092:9092

environment:

KAFKA\_ADVERTISED\_HOST\_NAME: kafka3

KAFKA\_MIN\_INSYNC\_REPLICAS: 1

KAFKA\_BROKER\_ID: 3

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 3

KAFKA\_MESSAGE\_MAX\_BYTES: 1048576

KAFKA\_REPLICA\_FETCH\_MAX\_BYTES: 1048576

KAFKA\_UNCLEAN\_LEADER\_ELECTION\_ENABLE: "false"

KAFKA\_LOG\_RETENTION\_MS: -1

KAFKA\_ADVERTISED\_PORT: 9092

KAFKA\_ZOOKEEPER\_CONNECT: kafka1:2181,kafka2:2181,kafka3:2181

volumes:

- /opt/kafka/:/kafka

## 第三步：启动orderer

启动:进入123的/etc/ansible目录执行如下命令

ansible-playbook fabric-orderer.yml

123上的/etc/ansible/fabric-orderer.yml文件内容如下：

Peer0org1:

1、创建目录：/opt/app/fabric/orderer

2、创建目录：/opt/app/org1/peer

3、将123主机/opt/fabric-deploy/orderer.example.com/ 目录内所有文件拷贝到/opt/app/fabric/orderer/目录并设置权限0755

4、执行 ./orderer 并且将正常和非正常日志全部输入到orderer.log文件里面

[root@localhost ansible]# vim fabric-orderer.yml

- hosts: peer0org1

remote\_user: root

tasks:

- name: mkdir ordererfile

file: state=directory path=/opt/app/fabric/orderer

- name: mkdir peerfile

file: state=directory path=/opt/app/org1/peer

- name: copy orderer File

copy: src=/opt/fabric-deploy/orderer.example.com/ dest=/opt/app/fabric/orderer/ mode=0755

- name: bash orderer

shell: nohup ./orderer >> ./orderer.log 2>&1 &

args:

chdir: /opt/app/fabric/orderer/

Peer0org2:

1、创建目录：/opt/app/fabric/orderer

2、创建目录：/opt/app/org2/peer /opt/app/org3/peer

3、将123主机/opt/fabric-deploy/orderer2.example.com/ 目录内所有文件拷贝到/opt/app/fabric/orderer/目录并设置权限0755

4、执行 ./orderer 并且将正常和非正常日志全部输入到orderer.log文件里面

- hosts: peer0org2

remote\_user: root

tasks:

- name: mkdir ordererfile

file: state=directory path=/opt/app/fabric/orderer

- name: mkdir peerfile

file: state=directory path=/opt/app/{{item}}/peer

with\_items:

- org2

- org3

- name: copy orderer File

copy: src=/opt/fabric-deploy/orderer2.example.com/ dest=/opt/app/fabric/orderer/ mode=0755

- name: bash orderer

shell: nohup ./orderer >> ./orderer.log 2>&1 &

args:

chdir: /opt/app/fabric/orderer/

- hosts: peer1org1

Peer1org1:

1、创建目录：/opt/app/fabric/orderer

2、创建目录：/opt/app/org1/peer /opt/app/org2/peer

/opt/app/org3/peer

3、将123主机/opt/fabric-deploy/orderer3.example.com/ 目录内所有文件拷贝到/opt/app/fabric/orderer/目录并设置权限0755

4、执行 ./orderer 并且将正常和非正常日志全部输入到orderer.log文件里面

remote\_user: root

tasks:

- name: mkdir ordererfile

file: state=directory path=/opt/app/fabric/orderer

- name: mkdir peerfile

file: state=directory path=/opt/app/{{item}}/peer

with\_items:

- org1

- org2

- org3

- name: copy orderer File

copy: src=/opt/fabric-deploy/orderer3.example.com/ dest=/opt/app/fabric/orderer/ mode=0755

- name: bash orderer

shell: nohup ./orderer >> ./orderer.log 2>&1 &

args:

chdir: /opt/app/fabric/orderer/

>nohup ./start-dishi.sh >output 2>&1 &

解释：

1. 带&的命令行，即使terminal（终端）关闭，或者电脑死机程序依然运行（前提是你把程序递交到服务器上)；

2. 2>&1的意思：是把标准错误（2）重定向到标准输出中（1），而标准输出又导入文件output里面，所以结果是标准错误和标准输出都导入文件output里面了。 至于为什么需要将标准错误重定向到标准输出的原因，那就归结为标准错误没有缓冲区，而stdout有。这就会导致 >output 2>output 文件output被两次打开，而stdout和stderr将会竞争覆盖，这肯定不是我门想要的.

这就是为什么有人会写成： nohup ./command.sh >output 2>output出错的原因了

#1：创建目录：/opt/app/fabric/orderer 创建目录：/opt/app/org1/peer

将123主机上/opt/fabric-deploy/orderer.example.com/ 目录内所有文件拷贝到/opt/app/fabric/orderer/目录并设置权限0755

执行 ./orderer 并且将正常和非正常日志全部输入到orderer.log文件里面

由以上可知orderer情况为：

Peer0org1主机(107)上有：orderer.example.com org1目录

Peer0org2主机(100)上有：orderer2.example.com org2、org3目录

Peer1org1主机(104)上有：orderer3.example.com org1、org2、org3目录

## 第四步：启动peer

启动:进入123的/etc/ansible目录执行如下命令

ansible-playbook fabric-peer.yml

123上的/etc/ansible/fabric-peer.yml 文件内容如下：

Peer1org1:

1、将123主机/opt/fabric-deploy/peer0.org1.example.com/ 目录内所有文件拷贝到/opt/app/org1/peer/目录并设置权限0755

2、执行 ./peer node start 并且将正常和非正常日志全部输入到peer.log文件里面

[root@localhost ansible]# vim fabric-peer.yml

- hosts: peer0org1

remote\_user: root

tasks:

- name: copy peer File

copy: src=/opt/fabric-deploy/peer0.org1.example.com/ dest=/opt/app/org1/peer/ mode=0755

- name: bash peer

shell: nohup ./peer node start >> ./peer.log 2>&1 &

args:

Peer0org2:

1、将123主机/opt/fabric-deploy/peer0.org2.example.com/ 目录内所有文件拷贝到/opt/app/org2/peer/目录并设置权限0755

2、执行 ./peer node start 并且将正常和非正常日志全部输入到peer.log文件里面

3、peer0.org3 与上操作相同

chdir: /opt/app/org1/peer/

- hosts: peer0org2

remote\_user: root

tasks:

- name: copy org2 File

copy: src=/opt/fabric-deploy/peer0.org2.example.com/ dest=/opt/app/org2/peer/ mode=0755

- name: bash peer

shell: nohup ./peer node start >> ./peer.log 2>&1 &

args:

chdir: /opt/app/org2/peer/

- name: copy org3 File

copy: src=/opt/fabric-deploy/peer0.org3.example.com/ dest=/opt/app/org3/peer/ mode=0755

- name: bash peer

shell: nohup ./peer node start >> ./peer.log 2>&1 &

args:

chdir: /opt/app/org3/peer/

- hosts: peer1org1

remote\_user: root

tasks:

- name: copy org1 File

copy: src=/opt/fabric-deploy/peer1.org1.example.com/ dest=/opt/app/org1/peer/ mode=0755

- name: bash peer

shell: nohup ./peer node start >> ./peer.log 2>&1 &

args:

chdir: /opt/app/org1/peer/

- name: copy org2 File

copy: src=/opt/fabric-deploy/peer1.org2.example.com/ dest=/opt/app/org2/peer/ mode=0755

- name: bash peer

shell: nohup ./peer node start >> ./peer.log 2>&1 &

args:

chdir: /opt/app/org2/peer/

- name: copy org3 File

copy: src=/opt/fabric-deploy/peer1.org3.example.com/ dest=/opt/app/org3/peer/ mode=0755

- name: bash peer

shell: nohup ./peer node start >> ./peer.log 2>&1 &

args:

chdir: /opt/app/org3/peer/

Peer1org1:

1、将123主机/opt/fabric-deploy/peer1.org1.example.com/ 目录内所有文件拷贝到/opt/app/org1/peer/目录并设置权限0755

2、执行 ./peer node start 并且将正常和非正常日志全部输入到peer.log文件里面

3、peer1.org2 与上操作相同

4、peer1.org3 与上操作相同

由以上可知peer情况为：

Peer0org1主机(107)上有：peer0.org1.example.com

Peer0org2主机(100)上有：peer0.org2.example.com、peer0.org3.example.com

Peer1org1主机(104)上有：peer1.org1.example.com、peer1.org2.example.com、peer1.org3.example.co