启动网络手动实现

实现步骤

生成组织关系和身份证书

确定是在 `fabric-samples/first-network` 路径下

```
$ cd hyfa/fabric-samples/first-network/
```

为fabric网络生成指定拓扑结构的**组织关系和身份证书**

```
$ sudo ../bin/cryptogen generate --config=./crypto-config.yaml
```

> 此命令依赖 `crypto-config.yaml` 配置文件

会有如下输出:

```
org1.example.com
org2.example.com
```

证书和密钥（即MSP材料）将被输出到目录 `first-network/crypto-config` 的目录中

ordererOrganizations子目录下包括构成Orderer组织(1个Orderer节点)的身份信息

peerOrganizations子目录下为所有的Peer节点组织(2个组织, 4个节点)的相关身份信息. 其中最关键的是MSP目录, 代表了实体的身份信息

crypto-config文件目录结构如下

crypto-config ├── ordererOrganizations │ └── example.com │ ├── ca │ │ ├── ca.example.com-cert.pem │ │ └── df69b6d2aea8038270c5340d358bfe34eee039a8e16d4a849e67ec27a8ed53bd_sk │ ├── msp │ │ ├── admincerts │ │ │ └── Admin@example.com-cert.pem │ │ ├── cacerts │ │ │ └── ca.example.com-cert.pem │ │ └── tlscacerts │ │ └── tlsca.example.com-cert.pem │ ├── orderers │ │ └── orderer.example.com │ │ ├── msp │ │ │ ├── admincerts │ │ │ ├── cacerts │ │ │ ├── keystore │ │ │ ├── signcerts │ │ │ └── tlscacerts │ │ └── tls │ │ ├── ca.crt │ │ ├── server.crt │ │ └── server.key │ ├── tlsca │ │ ├── 138f1cfd2708bca1e9e525773af410d46cef12736c3673ed787d7bbc38f013a4_sk │ │ └── tlsca.example.com-cert.pem │ └── users │ └── Admin@example.com │ ├── msp │ │ ├── admincerts │ │ ├── cacerts │ │ ├── keystore │ │ ├── signcerts │ │ └── tlscacerts │ └── tls │ ├── ca.crt │ ├── client.crt │ └── client.key └── peerOrganizations ├── org1.example.com # 第一个组织的相关材料，每个组织会生成单独的根证书 │ ├── ca #存放组织的根证书和对应的私钥文件，默认采用EC 算法，证书为自签名。组织内的实体将基于该根证书作为证书根。 │ │ ├── 9b78dd1cc0570c9ef3f3fa31a1b343e7a6c0f157a2cc17f75412e12f2936898c_sk │ │ └── ca.org1.example.com-cert.pem │ ├── msp # 存放代表该组织的身份信息 │ │ ├── admincerts # 组织管理员的身份验证证书，被根证书签名 │ │ │ └── Admin@org1.example.com-cert.pem │ │ ├── cacerts # 组织的根证书，同ca 目录下文件 │ │ │ └── ca.org1.example.com-cert.pem │ │ ├── config.yaml │ │ └── tlscacerts # ：用于TLS 的CA 证书，自签名 │ │ └── tlsca.org1.example.com-cert.pem │ ├── peers # 存放属于该组织的所有Peer 节点 │ │ ├── peer0.org1.example.com # 第一个peer 的信息，包括其msp 证书和tls 证书两类 │ │ │ ├── msp │ │ │ │ ├── admincerts # 组织管理员的身份验证证书。Peer 将基于这些证书来认证交易签署者是否为管理员身份 │ │ │ │ ├── cacerts # 存放组织的根证书 │ │ │ │ ├── config.yaml

```
│   │   │   ├── keystore # 本节点的身份私钥，用来签名│   │   │   ├── signcerts # 验证本节点签名的证书，被组织根证书签名│   │   │   └── tlscacerts # TLS 连接用的身份证书，即组织TLS 证书│   │   └── tls # 存放tls 相关的证书和私钥│   │       ├── ca.crt # 组织的根证书│   │       ├── server.crt # 验证本节点签名的证书，被组织根证书签名│   │       └── server.key # 本节点的身份私钥，用来签名│   └── peer1.org1.example.com # 第二个peer 的信息，与peer0.org1.example.com结构类似│       ├── msp│       │   ├── admincerts│       │   ├── cacerts│       │   ├── config.yaml│       │   ├── keystore│       │   ├── signcerts│       │   └── tlscacerts│       └── tls│           ├── ca.crt│           ├── server.crt│           └── server.key│       ├── tlsca│       ├── cf4587814bc05f9f81ac3d990c365660dedf1479e60f737c7e9e707727a27168_sk│       └── tlsca.org1.example.com-cert.pem│   └── users # 存放属于该组织的用户的实体│       ├── Admin@org1.example.com # 管理员用户的信息，包括其msp 证书和tls 证书两类│       │   ├── msp│       │   │   ├── admincerts # 管理身份验证证书│       │   │   ├── cacerts # 存放组织的根证书│       │   │   ├── keystore # 本用户的身份私钥，用来签名│       │   │   ├── signcerts # 管理员用户的身份验证证书，被组织根证书签名。要被某个Peer认可，则必须放到该Peer 的msp/admincerts 下│       │   │   └── tlscacerts # TLS 连接用的身份证书，即组织TLS 证书│       │   └── tls # 存放tls 相关的证书和私钥│       │       ├── ca.crt # 组织的根证书│       │       ├── client.crt # 管理员的用户身份验证证书，被组织根证书签名│       │       └── client.key # 管理员用户的身份私钥，用来签名│       └── User1@org1.example.com # 第一个用户的信息，包括msp 证书和tls 证书两类│           ├── msp│           │   ├── admincerts # 管理身份验证证书│           │   ├── cacerts # 存放组织的根证书│           │   ├── keystore # 本用户的身份私钥，用来签名│           │   ├── signcerts # 验证本用户签名的身份证书，被组织根证书签名│           │   └── tlscacerts # TLS 连接用的身份证书，即组织TLS 证书│           └── tls # 存放tls 相关的证书和私钥│               ├── ca.crt # 组织的根证书│               ├── client.crt # 验证用户签名的身份证书，被组织根证书签名│               └── client.key # 用户的身份私钥，用来签名└── org2.example.com # 第二个组织的信息，与org1.example.com结构类似    ├── ca    │   ├── 91fd76daf883a57066303fb6842ff4fb07c6793dbc8fbbca6303efea455884b2_sk    │   └── ca.org2.example.com-cert.pem    ├── msp    │   ├── admincerts    │   │   └── Admin@org2.example.com-cert.pem    │   ├── cacerts    │   │   └──
```

ca.org2.example.com-cert.pem │ ├── config.yaml │ └── tlscacerts │ └── tlsca.org2.example.com-cert.pem ├── peers │ ├── peer0.org2.example.com │ │ ├── msp │ │ │ ├── admincerts │ │ │ ├── cacerts │ │ │ ├── config.yaml │ │ │ ├── keystore │ │ │ ├── signcerts │ │ │ └── tlscacerts │ │ └── tls │ │ ├── ca.crt │ │ ├── server.crt │ │ └── server.key │ └── peer1.org2.example.com │ ├── msp │ │ ├── admincerts │ │ ├── cacerts │ │ ├── config.yaml │ │ ├── keystore │ │ ├── signcerts │ │ └── tlscacerts │ └── tls │ ├── ca.crt │ ├── server.crt │ └── server.key ├── tlsca │ ├── d3c3e6e37d306992bc9fb826415ed77971031418db48c195d5a24521916f32f3_sk │ └── tlsca.org2.example.com-cert.pem └── users ├── Admin@org2.example.com │ ├── msp │ │ ├── admincerts │ │ ├── cacerts │ │ ├── keystore │ │ ├── signcerts │ │ └── tlscacerts │ └── tls │ ├── ca.crt │ ├── client.crt │ └── client.key └── User1@org2.example.com ├── msp │ ├── admincerts │ ├── cacerts │ ├── keystore │ ├── signcerts │ └── tlscacerts └── tls ├── ca.crt ├── client.crt └── client.key

Cryptogen 按照配置文件中指定的结构生成了对应的组织和密钥、证书文件

其中最关键的是各个资源下的msp 目录内容，存储了生成的代表MSP 身份的各种证书文件，一般包括：

- admincerts ：管理员的身份证书文件
- cacerts ：信任的根证书文件
- key store ：节点的签名私钥文件
- signcerts ：节点的签名身份证书文件
- tlscacerts: TLS 连接用的证书
- intermediatecerts （可选）：信任的中间证书
- crls （可选）：证书撤销列表
- config.yaml （可选）：记录OrganizationalUnitIdentifiers 信息，包括根证书位置和ID信息

这些身份文件随后可以分发到对应的Orderer 节点和Peer 节点上，并放到对应的MSP路径下，用于签名使用

配置环境变量

告诉configtxgen工具在哪里寻找configtx.yaml 文件

```
$ export FABRIC_CFG_PATH=$PWD
```

创建Ordering服务启动初始区块

指定使用 `configtx.yaml` 文件中定义的 `TwoOrgsOrdererGenesis` 模板, 生成Ordering服务系统通道的初始区块文件

```
$ sudo ../bin/configtxgen -profile TwoOrgsOrdererGenesis -
outputBlock ./channel-artifacts/genesis.block
```

命令执行后输出如下:

```
10:49:21.181 CST [common/tools/configtxgen] main -> INFO 001 Loading
configuration
10:49:21.207 CST [msp] getMspConfig -> INFO 002 Loading NodeOUs
10:49:21.208 CST [msp] getMspConfig -> INFO 003 Loading NodeOUs
10:49:21.210 CST [common/tools/configtxgen] doOutputBlock -> INFO
004 Generating genesis block
10:49:21.211 CST [common/tools/configtxgen] doOutputBlock -> INFO
005 Writing genesis block
```

创建一个应用通道的配置交易

务必替换$CHANNEL_NAME或设置CHANNEL_NAME为可在整个说明中使用的环境变量

```
$ export CHANNEL_NAME=mychannel
```

指定使用 `configtx.yaml` 配置文件中的 `TwoOrgsChannel` 模板, 来生成新建通道的配置交易文件, `TwoOrgsChannel` 模板指定了Org1和Org2都属于后面新建的应用通道

```
$ sudo ../bin/configtxgen -profile TwoOrgsChannel -
outputCreateChannelTx ./channel-artifacts/channel.tx -channelID
$CHANNEL_NAME
```

输出如下

```
11:13:24.984 CST [common/tools/configtxgen] main -> INFO 001 Loading
configuration
11:13:24.992 CST [common/tools/configtxgen] doOutputChannelCreateTx
-> INFO 002 Generating new channel configtx
11:13:24.993 CST [msp] getMspConfig -> INFO 003 Loading NodeOUs
11:13:24.994 CST [msp] getMspConfig -> INFO 004 Loading NodeOUs
11:13:25.016 CST [common/tools/configtxgen] doOutputChannelCreateTx
-> INFO 005 Writing new channel tx
```

生成锚节点配置更新文件

锚节点配置更新文件用来对组织的锚节点进行配置

同样基于 `configtx.yaml` 配置文件生成新建通道文件, 每个组织都需要分别生成且注意指定对应的组织名称

```
$ sudo ../bin/configtxgen -profile TwoOrgsChannel -
outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx -
channelID $CHANNEL_NAME -asOrg Org1MSP

$ sudo ../bin/configtxgen -profile TwoOrgsChannel -
outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx -
channelID $CHANNEL_NAME -asOrg Org2MSP
```

启动网络

`$ sudo docker-compose -f docker-compose-cli.yaml up -d`

-f: 指定docker-compose文件

注:

> 如果想查看网络的实时日志，则不需要提供 -d 参数

> CLI容器将闲置1000秒。如果在需要时它消失了，可以用一个简单的命令重新启动它：

> $ sudo docker start cli

网络启动顺序: 首先启动Orderer节点, 然后启动Peer节点, 日志输出如下:

```
......
orderer.example.com       | 02:48:25.080 UTC [orderer/common/server]
initializeServerConfig -> INFO 002 Starting orderer with TLS enabled
orderer.example.com       | 02:48:25.101 UTC [fsblkstorage]
newBlockfileMgr -> INFO 003 Getting block information from block
storage
orderer.example.com       | 02:48:25.138 UTC
[orderer/commmon/multichannel] NewRegistrar -> INFO 004 Starting
system channel 'testchainid' with genesis block hash
67662e918ab76b4a8863cc625d67fcc31e9cb3a7c3c4f9f707af1c05ba5be686 and
orderer type solo
orderer.example.com       | 02:48:25.138 UTC [orderer/common/server]
Start -> INFO 005 Starting orderer:
......
```

Peer节点启动后, 默认情况下没有加入网络中的任何应用通道, 也不会与Orderer服务建立连接.需要通过客户端对其进行操作, 让它加入网络和指定的应用通道中

进入Docker容器

执行如下命令进入到CLI容器中(后继操作都在容器中执行)

```
$ sudo docker exec -it cli bash
```

如果成功, 命令提示符会变为如下内容:

`root@b240e1643244:/opt/gopath/src/github.com/hyperledger/fabric/peer#`

创建通道

检查环境变量是否正确设置

```
echo $CHANNEL_NAME
```

设置环境变量

```
export CHANNEL_NAME=mychannel
```

创建通道

```
peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f
./channel-artifacts/channel.tx --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrg
anizations/example.com/orderers/orderer.example.com/msp/tlscacerts/t
lsca.example.com-cert.pem
```

> 该命令自动在本地生成与该应用通道同名的初始区块 **mychannel.block**, 只有拥有该文件才可以加入创建的应用通道中

参数说明:

-o: 指定orderer节点的地址

-c: 指定要创建的应用通道的名称(必须与在创建应用通道交易配置文件时的通道名称一致)

-f: 指定创建应用通道时所使用的应用通道交易配置文件

--tls: 开启TLS验证

--cafile: 指定TLS_CA证书路径

查看:

```
root@086adb802655:/opt/gopath/src/github.com/hyperledger/fabric/peer
# ll
total 36
drwxr-xr-x 5 root root  4096 Apr 29 03:34 ./
drwxr-xr-x 3 root root  4096 Apr 29 02:48 ../
drwxr-xr-x 2 root root  4096 Apr 29 02:47 channel-artifacts/
drwxr-xr-x 4 root root  4096 Apr 29 02:35 crypto/
-rw-r--r-- 1 root root 15660 Apr 29 03:34 mychannel.block
drwxr-xr-x 2 root root  4096 Apr 29 02:13 scripts/
```

加入通道

应用通道所包含组织的成员节点可以加入通道中

```
peer channel join -b mychannel.block
```

join命令: 将本Peer节点加入到应用通道中

参数说明:

-b: 指定当前节点要加入/联接至应用通道

更新锚点

使用Org1的管理员身份更新锚节点配置

```
peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f
./channel-artifacts/Org1MSPanchors.tx --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrg
anizations/example.com/orderers/orderer.example.com/msp/tlscacerts/t
lsca.example.com-cert.pem
```

使用Org2的管理员身份更新锚节点配置

```
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.ex
ample.com/msp
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/f
abric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org
2.example.com/tls/ca.crt

peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f
./channel-artifacts/Org2MSPanchors.tx --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrg
anizations/example.com/orderers/orderer.example.com/msp/tlscacerts/t
lsca.example.com-cert.pem
```

手动配置网络完成, 可以测试Chaincode

切换为pee1.org1.example.com

CORE_PEER_LOCALMSPID="Org1MSP"

CORE_PEER_ADDRESS=peer1.org1.example.com:7051

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp

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

切换为Org2的peer0.org2.example.com

CORE_PEER_LOCALMSPID="Org2MSP"

CORE_PEER_ADDRESS=peer0.org2.exmple.com:7051

CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/pee0.org2.example.com/tls/ca.crt

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp