configtxlator转换配置/添加Org组织到channel

cryptogen

configtxgen

configtxlator转换

jq: 专用于处理JSON内容的工具

设置环境

进入到`fabric-samples/first-network`目录中, 执行`.byfn.sh -m down`关闭网络,清理之前的任何环境

```
$ cd $HOME/hyfa/fabric-samples/first-network/
$ sudo ./byfn.sh -m down
```

重新生成默认的BYFN构件

```
$ sudo ./byfn.sh -m generate
```

启用网络

```
$ sudo ./byfn.sh -m up
```

添加组织Org3

使用`eyfn.sh`脚本将Org3引入网络

```
$ sudo ./eyfn.sh up
```

从输出中可以看到添加的Org3加密资料，配置更新正在创建和签名，然后链接代码被安装以允许Org3执行分类账查询

如果执行成功, 会有如下输出

```
========= All GOOD, EYFN test execution completed ===========


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

进入CLI容器

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

导出 `ORDERER_CA` 与 `CHANNEL_NAME` 变量:

```
export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto
/ordererOrganizations/example.com/orderers/orderer.example.com/msp/t
lscacerts/tlsca.example.com-cert.pem  && export
CHANNEL_NAME=mychannel
```

检查环境变量是否正确设置:

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

> **如果重新启动了CLI容器，则必须重新导出两个环境变量**

查询

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":
["query","a"]}'
```

查询结果: `Query Result: 80`

调用,实现从a到b转账

```
peer chaincode invoke -o orderer.example.com:7050  --tls --cafile
$ORDERER_CA -C $CHANNEL_NAME -n mycc -c '{"Args":
["invoke","a","b","10"]}'
```

查询

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":
["query","a"]}'
```

查询结果: `Query Result: 70`

手动实现配置:

如果使用了`eyfn.sh`脚本，则需要将网络关闭.删除所有容器并撤销添加Org3所做的操作

在`fabric-samples/first-network/`目录中执行如下命令:

```
$ sudo ./eyfn.sh down
$ sudo ./byfn.sh -m down
```

```
$ sudo ./byfn.sh -m generate
$ sudo ./byfn.sh -m up
```

生成Org3加密材料

从 `frist-network` 目录转至 `org3-artifacts` 目录中

```
% cd org3-artifacts
```

生成加密材料

```
$ sudo ../../bin/cryptogen generate --config=./org3-crypto.yaml
```

为Org3以及与此新Org绑定的两个对等生成密钥和证书

新生成的文件被保存在当前目录下新生成的文件夹 `crypto-config` 中

使用 `configtxgen` 工具以JSON输出Org3特定的配置材料到指定的文件中

```
$ export FABRIC_CFG_PATH=$PWD
$ sudo ../../bin/configtxgen -printOrg Org3MSP > ../channel-
artifacts/org3.json
```

该文件包含Org3的策略定义，以及以Base 64格式提供的三个重要证书：admin用户证书（稍后将用作Org3的管理员），CA根证书和TLS根证书

后面会将这个JSON文件附加到通道配置中

```
$ cd ../
$ sudo cp -r crypto-config/ordererOrganizations org3-
artifacts/crypto-config/
```

将Orderer Org的MSP材料移植到Org3 `crypto-config`目录中

更新通道配置

进入CLI容器

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

安装jq工具

jq可以将所需要的数据格式转换成任意的数据格式

> jq工具允许脚本与`configtxlator`工具返回的JSON文件进行交互

```
apt update && apt install -y jq
```

参数说明:

-y: 忽略安装时的提示

导出 `ORDERER_CA` 与 `CHANNEL_NAME` 变量:

```
export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto
/ordererOrganizations/example.com/orderers/orderer.example.com/msp/t
lscacerts/tlsca.example.com-cert.pem  && export
CHANNEL_NAME=mychannel
```

检查环境变量是否正确设置:

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

> **如果重新启动了CLI容器，则必须重新导出两个环境变量**

获取配置

获取 `mychannel` 通道的最新配置块, 可以防止重复或替换配置更改, 有助于确保并发性, 防止删除两个组织

```
peer channel fetch config config_block.pb -o
orderer.example.com:7050 -c $CHANNEL_NAME --tls --cafile $ORDERER_CA
```

上述命令将二进制protobuf通道配置块保存到 `config_block.pb`

转换配置为JSON

利用 `configtxlator` 工具将此通道配置块解码为JSON格式, 删除所有与想要改变的内容无关的标题，元数据，创建者签名等等

通过 `jq` 工具来完成

```
configtxlator proto_decode --input config_block.pb --type
common.Block | jq .data.data[0].payload.data.config > config.json
```

> .data.data[0],payload.data.config域内数据代表了完整的通道配置信息

查看`config.json`文件中的内容

```
more config.json
```

使用jq将Org3配置定义追加`org3.json`到通道的应用程序组字段，并命名输出`modified_config.json`

```
jq -s '.[0] * {"channel_group":{"groups":{"Application":{"groups":
{"Org3MSP":.[1]}}}}}' config.json ./channel-artifacts/org3.json >
modified_config.json
```

> -s: 将所有的 JSON 输入放入一个数组中

将 `config.json` 中的内容输出为 `config.pb`

```
configtxlator proto_encode --input config.json --type common.Config
--output config.pb
```

将 `modified_config.json` 中的内容输出为 `modified_config.pb`

```
configtxlator proto_encode --input modified_config.json --type
common.Config --output modified_config.pb
```

利用这两个配置文件, 使用`configtxlator`计算出更新配置时的更新量信息。该命令将输出一个新的二进制文件，命名为`org3_update.pb`：

```
configtxlator compute_update --channel_id $CHANNEL_NAME --original
config.pb --updated modified_config.pb --output org3_update.pb
```

将 `org3_update.pb` 中的内容解码为可编辑的JSON格式并将其称为 `org3_update.json`

```
configtxlator proto_decode --input org3_update.pb --type
common.ConfigUpdate | jq . > org3_update.json
```

对通道配置进行更新时, 还需要封装为 `org3_update_in_envelope` 结构的配置更新交易. 因此, 需要将 `org3_update` 结构数据进行补全

```
echo '{"payload":{"header":{"channel_header":
{"channel_id":"mychannel", "type":2}},"data":{"config_update":'$(cat
org3_update.json)'}}}' | jq . > org3_update_in_envelope.json
```

将其转换为Fabric所需的二进制交易配置文件。命名为最终更新对象 `org3_update_in_envelope.pb`

```
configtxlator proto_encode --input org3_update_in_envelope.json --
type common.Envelope --output org3_update_in_envelope.pb
```

签署并提交配置更新

```
peer channel signconfigtx -f org3_update_in_envelope.pb
```

导出Org2环境变量:

```
export CORE_PEER_LOCALMSPID="Org2MSP"

export
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

export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.ex
ample.com/msp

export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
```

更新通道

```
peer channel update -f org3_update_in_envelope.pb -c $CHANNEL_NAME -
o orderer.example.com:7050 --tls --cafile $ORDERER_CA
```

命令执行后会有如下输出: `Successfully submitted channel update`

新终端中(终端2)中输出日志

打开一个新的终端(**终端2**)执行如下命令:

```
$ sudo docker logs -f peer0.org1.example.com
```

将Org3加入到通道

打开一个新的终端(终端3), 从 `first-network` 中启动Org3 docker compose

```
$ cd hyfa/fabric-samples/first-network/
$ sudo docker-compose -f docker-compose-org3.yaml up -d
```

进入Org3特定的CLI容器中:

```
$ sudo docker exec -it Org3cli bash
```

导出环境变量:

```
export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto
/ordererOrganizations/example.com/orderers/orderer.example.com/msp/t
lscacerts/tlsca.example.com-cert.pem && export
CHANNEL_NAME=mychannel
```

检查变量 :

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

检索该块:

```
peer channel fetch 0 mychannel.block -o orderer.example.com:7050 -c
$CHANNEL_NAME --tls --cafile $ORDERER_CA
```

发出命令并通过创世区块:

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

导出TLS与ADDRESS变量并重新发布

```
export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/f
abric/peer/crypto/peerOrganizations/org3.example.com/peers/peer1.org
3.example.com/tls/ca.crt && export
CORE_PEER_ADDRESS=peer1.org3.example.com:7051

peer channel join -b mychannel.block
```

升级并调用Chaincode

在Org3的CLI中执行(**终端3**):

```
peer chaincode install -n mycc -v 2.0 -p
github.com/chaincode/chaincode_example02/go/
```

终端1中执行

使用Org2管理员身份提交了频道更新通话(**在终端1中执行**):

```
export CORE_PEER_LOCALMSPID="Org2MSP"

export
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

export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.ex
ample.com/msp

export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
```

安装

```
peer chaincode install -n mycc -v 2.0 -p
github.com/chaincode/chaincode_example02/go/
```

切换为Org1身份:

```
export CORE_PEER_LOCALMSPID="Org1MSP"

export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/f
abric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org
1.example.com/tls/ca.crt

export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabri
c/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.ex
ample.com/msp

export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
```

再次安装：

```
peer chaincode install -n mycc -v 2.0 -p
github.com/chaincode/chaincode_example02/go/
```

发送电话：

```
peer chaincode upgrade -o orderer.example.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -C $CHANNEL_NAME -n mycc
-v 2.0 -c '{"Args":["init","a","90","b","210"]}' -P "OR
('Org1MSP.peer','Org2MSP.peer','Org3MSP.peer')"
```

如果报如下错误:

```
Error: Error getting broadcast client: failed to load config for
OrdererClient: unable to load orderer.tls.rootcert.file: open
/etc/hyperledger/fabric/-C: no such file or directory
则需要检查  echo $ORDERER_CA && echo $CHANNEL_NAME  变量是否正确设置

export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto
/ordererOrganizations/example.com/orderers/orderer.example.com/msp/t
lscacerts/tlsca.example.com-cert.pem && export
CHANNEL_NAME=mychannel
```

通过`v`标志指定新版本。可以看到Org3添加到背书政策中

与实例化调用一样，链式代码升级需要使用该`init`方法

终端3中执行

升级调用将新的块 - 块6 - 添加到频道的分类账中，并允许Org3同行在认可阶段执行交易。跳回Org3 CLI容器(**终端3**)并发出一个查询值`a`：

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":
["query","a"]}'
```

在终端1与终端3中通过,正确显示查询结果:`Query Result: 90`

调用,实现从a到b转账(**终端3**):

```
peer chaincode invoke -o orderer.example.com:7050  --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -C $CHANNEL_NAME -n mycc
-c '{"Args":["invoke","a","b","10"]}'
```

查询:

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":
["query","a"]}'
```

在终端1与终端3中通过,正确显示查询结果: `Query Result: 80`