**Fabric-kafka共识orderer集群**

在fabric中支持的三种共识机制：solo（单台orderer相当于没有共识）、kafka、pbft(还在开发中)。Orderer节点起着至关重要的作用，主要作用一个是排序，一个是维护区块（账本）。所以在生产环境中，为了防止因只有一个orderer节点，并且orderer节点意外停止服务导致fabric网络异常等情况。采用kafka共识，kafka对比solo能极大的提高吞吐量以达到提高交易的效率。

为更贴近生产环境，这里不再使用fabric官网上例子的文件目录，而是自定义目录。所依赖的文件工具则从之前的目录中复制。

# 环境准备

## 服务器准备

本文实现3个orderer节点的集群，kafka节点和zookeeper节点都为3个，or1和org2对应的节点暂时只为一个。每个服务器对应的节点信息如下表：

|  |  |
| --- | --- |
| 192.168.15.143 | orderer1、zookeeper1、kafka1 |
| 192.168.15.153 | orderer2、zookeeper2、kafka2、org1peer0 |
| 192.168.15.154 | orderer3、zookeeper3、kafka3、org2peer0 |

## Fabric和fabric-samples的版本为1.1.0

git checkout -b release-1.1 origin/release-1.1

# 目录和依赖文件准备

## 新建运行目录

mkdir -p /root/myFabric/fabric/kafka/

新建交易配置文件目录

mkdir -p /root/myFabric/fabric/kafka/channel-artifacts

## 设置docker images的版本为1.1.0

进入/root/chsoft/fabric/kafka/目录，新增.env文件添加值：

IMAGE\_TAG=1.1.0

## 复制依赖文件

复制cryptogen和configtxgen工具

在《搭建第一个fabric网络》中已下载工具，可以复制或下载。

cp -r /root/fabric-samples/bin/ /root/myFabric/fabric/

复制fabric项目中e2e的base文件夹，此文件夹中的为之后docker-compose模板提供官网默认配置

cp -r /root/go/src/github.com/hyperledger/fabric/examples/e2e\_cli/base/ /root/myFabric/fabric/kafka/

这里使用cryptogen和configtxgen工具生成证书和配置文件，生产可以选择使用fabricCA生成证书。复制工具消费的配置文件，后面会根据需要进行修改。

cp /root/go/src/github.com/hyperledger/fabric/examples/e2e\_cli/configtx.yaml /root/myFabric/fabric/kafka/

cp /root/go/src/github.com/hyperledger/fabric/examples/e2e\_cli/crypto-config.yaml /root/myFabric/fabric/kafka/

# 生成证书和配置文件

## 修改crypto-config.yaml

修改Specs，设置三个orderer：orderer1、orderer2、orderer3。红色部分为修改的地方。

# ---------------------------------------------------------------------------

# "OrdererOrgs" - Definition of organizations managing orderer nodes

# ---------------------------------------------------------------------------

OrdererOrgs:

# ---------------------------------------------------------------------------

# Orderer

# ---------------------------------------------------------------------------

- Name: Orderer

Domain: example.com

# --------------------------------------------------------------------------

# "Specs" - See PeerOrgs below for complete description

# --------------------------------------------------------------------------

Specs:

- Hostname: orderer1

- Hostname: orderer2

- Hostname: orderer3

# --------------------------------------

## 修改configtx.yaml

* 将OrdererType修改成kafka
* 修改Addresses，这里我是设置了三个orderer：orderer1、orderer2、orderer3。
* 修改Kafka的Brokers，这里我使用三台虚拟机运行kafka

红色部分为修改的地方。

###############################################################################

Orderer: &OrdererDefaults

OrdererType: kafka

Addresses:

- orderer1.example.com:7050

- orderer2.example.com:7050

- orderer3.example.com:7050

# Batch Timeout: The amount of time to wait before creating a batch

BatchTimeout: 2s

BatchSize:

MaxMessageCount: 10

AbsoluteMaxBytes: 99 MB

PreferredMaxBytes: 512 KB

Kafka:

Brokers:

- 192.168.15.143:9092

- 192.168.15.153:9092

- 192.168.15.154:9092

## 生成证书和配置文件

参照《搭建第一个fabric网络》，进入/root/myFabric/fabric/kafka/目录执行以下命令，根据上面两个文件生成证书和配置文件。命令如下：

如果允许允许命令显示权限不足，给bin文件夹赋权限。

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

export FABRIC\_CFG\_PATH=$PWD

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

export CHANNEL\_NAME=mychannel

../bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/channel.tx -channelID $CHANNEL\_NAME

../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx -channelID $CHANNEL\_NAME -asOrg Org1MSP

../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx -channelID $CHANNEL\_NAME -asOrg Org2MSP

# Docker-compose文件配置

## 修改docker-compose-base.yaml和peer-base.yaml

在docker-compose-base.yaml中的orderer服务中，镜像添加版本变量。此变量的值在上面步骤.env中已赋值。

orderer.example.com:

container\_name: orderer.example.com

image: hyperledger/fabric-orderer:$IMAGE\_TAG

因为是多机部署，所以将下面peer节点中容器与本机的映射都修改为相同值。

ports:

- 7051:7051

- 7052:7052

- 7053:7053

在peer-base.yaml文件中也添加peer镜像版本变量，然后修改容器的网络连接模式为kafka\_default。

version: '2'

services:

peer-base:

image: hyperledger/fabric-peer:$IMAGE\_TAG

environment:

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

# 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=kafka\_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

## 新增zookeeper运行文件

因为在三台机器上都得运行一个zookeeper节点，所以我们新增三个zookeeper的运行docker-compose文件。zookeeper1.yaml、zookeeper2.yaml、zookeeper3.yaml文件，下面以zookeeper1.yaml为例，描述zookeeper节点文件的配置。

因为zookeeper集群是中心化管理，zookeeper节点的命名和docker容器的命名是有关联的，并且在zookeeper节点互相连接时，名称是有唯一性并且有作用的，所以zookeeper节点的服务名称，docker名称必须一致。（作者已尝试，若不一致或者全部相同，在启动节点后，节点之间将无法正常连接）

ZOO\_MY\_ID为zookeeper节点的标识，每个节点的ZOO\_MY\_ID值在集群当中必须唯一。

下面为zookeeper1.yaml文件内容

version: '2'

volumes:

zookeeper1: # 三个配置文件中此处分别为zookeeper1、zookeeper2、zookeeper3

services:

zookeeper1: # 三个配置文件中此处分别为zookeeper1、zookeeper2、zookeeper3

extends:

file: base/docker-compose-base.yaml

service: zookeeper

container\_name: zookeeper1 #三个配置文件中此处分别为zookeeper1、zookeeper2、zookeeper3

hostname: zookeeper1 #三个配置文件中此处分别为zookeeper1、zookeeper2、zookeeper3

environment:

- ZOO\_MY\_ID=1 # ID在集合中必须是唯一的并且应该有一个值，在1和255之间。

# 组成ZK集合的服务器列表。客户端使用的列表必须与ZooKeeper服务器列表所拥有的每一个ZK服务器相匹配。

# 有两个端口号 `nnnnn`。第一个是追随者用来连接领导者的东西，第二个是领导人选举。

- ZOO\_SERVERS=server.1=zookeeper1:2888:3888 server.2=zookeeper2:2888:3888 server.3=zookeeper3:2888:3888

ports:

- 2181:2181

- 2888:2888

- 3888:3888

extra\_hosts:

- zookeeper1:192.168.15.143

- zookeeper2:192.168.15.153

- zookeeper3:192.168.15.154

## 新增kafka运行文件

类似zookeeper，因为需要在3台机器上运行，所以需要三个运行文件。kafka1.yaml、kafka2.yaml、kafka3.yaml。Kafka节点和docker容器名称也是和zookeeper一样的约束，名称不能重复，KAFKA\_BROKER\_ID必须唯一。

下面为kafka1.yaml文件内容：

version: '2'

volumes:

kafka1: # 三个配置文件中此处分别为kafka1、kafka2、kafka3

services:

kafka1: # 三个配置文件中此处分别为kafka1、kafka2、kafka3

extends:

file: base/docker-compose-base.yaml

service: kafka

container\_name: kafka1 # 三个配置文件中此处分别为kafka1、kafka2、kafka3

hostname: kafka1 # 三个配置文件中此处分别为kafka1、kafka2、kafka3

environment:

- KAFKA\_BROKER\_ID=1 #此kafka的唯一标识

#min.insync.replicas = M---设置一个M值（例如1<M<N，查看下面的default.replication.factor）

#数据提交时会写入至少M个副本（这些数据然后会被同步并且归属到in-sync 副本集合或ISR）。

#其它情况，写入操作会返回一个错误。接下来：

#如果channel写入的数据多达N-M个副本变的不可用，操作可以正常执行。

#如果有更多的副本不可用，Kafka不可以维护一个有M数量的ISR集合，因此Kafka停止接收写操作。Channel只有当同步M个副本后才可以重新可以写。

- KAFKA\_MIN\_INSYNC\_REPLICAS=2

# 设置一个值N，N<K。

# 设置replication factor参数为N代表着每个channel都保存N个副本的数据到Kafka的代理上。

# 这些都是一个channel的ISR集合的候选。

# 如同在上边min.insync.replicas section设置部分所描述的，不是所有的代理（orderer）在任何时候都是可用的。

# N的值必须小于K，如果少于N个代理的话，channel的创建是不能成功的。

# 因此，如果设置N的值为K，一个代理失效后，那么区块链网络将不能再创建新的channel---orderering service的crash容错也就不存在了。

- KAFKA\_DEFAULT\_REPLICATION\_FACTOR=3

# 指向Zookeeper节点的集合，其中包含ZK的集合。

- KAFKA\_ZOOKEEPER\_CONNECT=192.168.15.143:2181,192.168.15.153:2181,192.168.15.154:2181

ports:

- 9092:9092

extra\_hosts:

- zookeeper1:192.168.15.143

- zookeeper2:192.168.15.153

- zookeeper3:192.168.15.154

- kafka1:192.168.15.143

- kafka2:192.168.15.153

- kafka3:192.168.15.154

## 新增orderer运行文件

类似zookeeper，因为需要在3台机器上运行，所以需要三个运行文件，orderer1.yaml、orderer2.yaml、orderer3.yam。下面以orderer1.yaml文件为例，描述orderer节点文件的配置。

与正常orderer节点不同的主要是需要指明kafka集群对应服务器的IP和端口号。然后因为证书密钥文件的相对位置发生了改变，在启动文件中，覆盖默认的证书密钥文件的位置。

下面为orderer1.yaml文件内容：

version: '2'

volumes:

orderer.example.com:

services:

orderer.example.com:

extends:

file: base/docker-compose-base.yaml

service: orderer.example.com

container\_name: orderer.example.com

environment:

- ORDERER\_KAFKA\_VERBOSE=true

- ORDERER\_KAFKA\_BROKERS=[192.168.15.143:9092,192.168.15.153:9092,192.168.15.154:9092]

volumes:

- ./crypto-config/ordererOrganizations/example.com/orderers/orderer1.example.com/msp:/var/hyperledger/orderer/msp

- ./crypto-config/ordererOrganizations/example.com/orderers/orderer1.example.com/tls:/var/hyperledger/orderer/tls

ports:

- 7050:7050

extra\_hosts:

- kafka1:192.168.15.143

- kafka2:192.168.15.153

- kafka3:192.168.15.154

## 新增peer和cli运行文件

要新建peer节点和该节点下cli容器的运行文件，用于测试orderer节点是否搭建成功。与正常peer节点节点不同的地方是添加连接的orderer节点集群的IP列表。还有一个就是添加chaincode的位置。

下面为文件的内容：

version: '2'

services:

peer0.org1.example.com:

container\_name: peer0.org1.example.com

extends:

file: base/docker-compose-base.yaml

service: peer0.org1.example.com

extra\_hosts:

- orderer1.example.com:192.168.15.143

- orderer2.example.com:192.168.15.153

- orderer3.example.com:192.168.15.154

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=peer0.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/peer0.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/peer0.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/peer0.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

# command: /bin/bash -c './scripts/script.sh ${CHANNEL\_NAME}; sleep $TIMEOUT'

volumes:

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

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

- /root/fabric-samples/chaincode/:/opt/gopath/src/github.com/chaincode

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

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

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

depends\_on:

- peer0.org1.example.com

extra\_hosts:

- orderer1.example.com:192.168.15.143

- orderer2.example.com:192.168.15.153

- orderer3.example.com:192.168.15.154

- peer0.org1.example.com:192.168.15.153

# 启动网络

## 启动顺序

docker-compose -f zookeeper1.yaml up -d

docker-compose -f zookeeper2.yaml up -d

docker-compose -f zookeeper3.yaml up -d

docker-compose -f kafka1.yaml up -d

docker-compose -f kafka1.yaml up -d

docker-compose -f kafka3.yaml up -d

docker-compose -f orderer1.yaml up -d

docker-compose -f orderer2.yaml up -d

docker-compose -f orderer3.yaml up -d

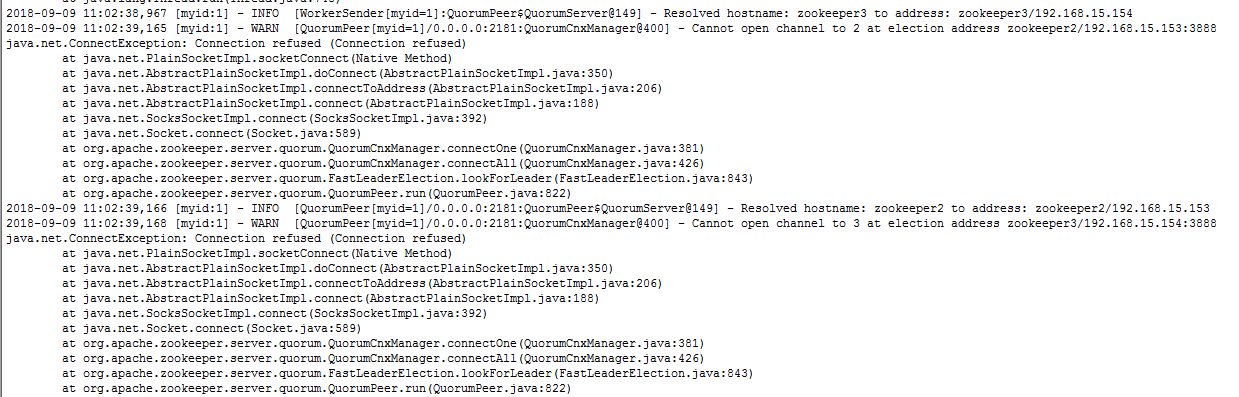
docker-compose -f docker-compose-peer0org1.yaml up -d

docker-compose -f docker-compose-peer0org2.yaml up -d

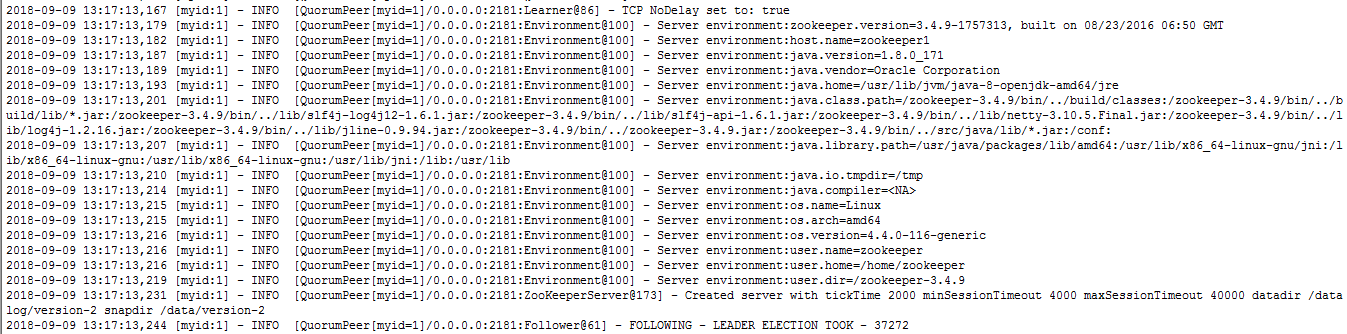
## zookeeper启动情况

按照上述启动zookeeper集群，启动zookeeper1时，docker日志会显示无法连接zookeeper2和zookeeper3。启动zookeeper2后，如果配置成功，因为已经启动zookeeper1，还未启动zookeeper3，所以zookeeper3的docker日志会显示连接上zookeeper1但无法连接zookeeper3。这时再查看zookeeper1的日志会显示zookeeper2已连接。之后再启动zookeeper3，zookeeper3的日志则不会再报错，并显示已连上其他两个节点。其他两个节点的日志也会显示连接上zookeeper3。下面以zookeeper1为例。

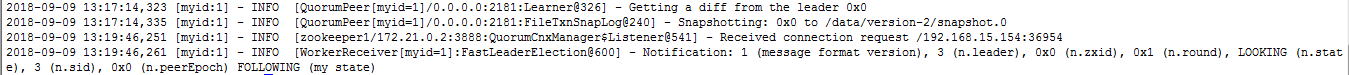
下图为zookeeper1启动时，zookeeper1的docker日志显示无法连接zookeeper2和zookeeper3



下图为zookeeper1启动后之后再启动zookeeper2，zookeeper1的docker日志显示已连接上zookeeper2。



下图为zookeeper1和zookeeper2启动后之后再启动zookeeper3，zookeeper1的docker日志显示已连接上zookeeper3。

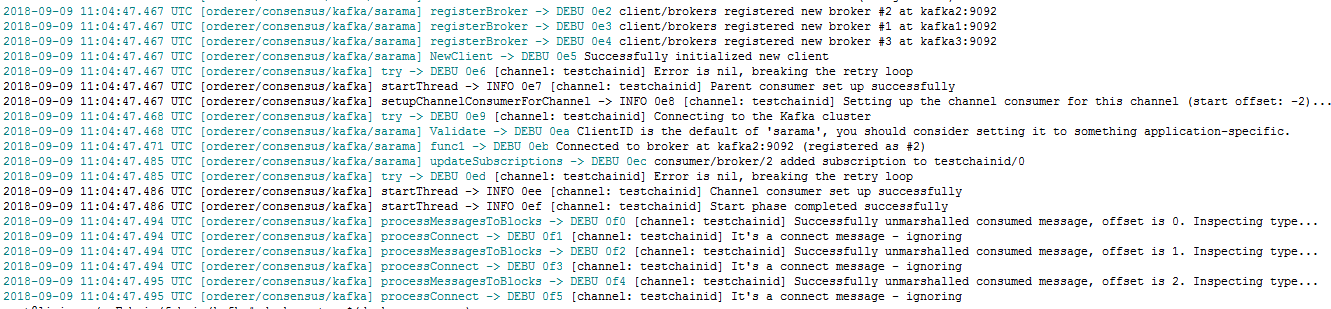


## kafka启动情况

如果zookeeper集群启动成功，3个kafka的启动日志都不会报错。下图为kafka1.yaml启动的docker日志。

## orderer启动情况

Orderer集群安装顺序启动，docker日志会显示orderer由3个broker，然后显示连接创世区块channel名称为testchainid所连接的orderer。每启动一个orderer，下面日志则会多出一条记录显示，多一个连接的orderer点。



# 多机联动测试

## peer上创建channel

测试时可以在org1peer0或者org1peer1上进行测试。因为集群所要达成的主要就是达成分布式效果，这里我们可以在创建channel时，任意选择一个orderer节点发起创建channel的命令。如下：

peer channel create -o orderer3.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/channel.tx --tls $CORE\_PEER\_TLS\_ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer3.example.com/msp/tlscacerts/tlsca.example.com-cert.pem

## peer获取已创建的channel文件

如果org1peer0已创建的channel，org2peer0可以向orderer集群获取channel的区块文件，类似上述，向orderer集群的任何一个节点发起命令都可以成功获取到文件。命令如下：

export ORDERER\_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer2.example.com/msp/tlscacerts/tlsca.example.com-cert.pem

export CHANNEL\_NAME=mychannel

peer channel fetch 0 mychannel.block -o orderer2.example.com:7050 -c $CHANNEL\_NAME --tls --cafile $ORDERER\_CA

## peer加入channel并安装chaincode

peer channel join -b mychannel.block

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

peer chaincode instantiate -o orderer3.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer3.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n mycc -v 1.0 -c '{"Args":["init","a", "100", "b","200"]}' -P "OR ('Org1MSP.peer','Org2MSP.peer')"

## peer执行chaincode交易

peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}'

peer chaincode instantiate -o orderer3.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer3.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL\_NAME -n mycc -v 1.0 -c '{"Args":["init","a", "100", "b","200"]}' -P "OR ('Org1MSP.peer','Org2MSP.peer')"

## 关闭一orderer节点之后操作测试

如果正确启动orderer集群，并且是能达到kafka的节点容错条件（详细请自行了解kafka和zookeeper）。则在容错条件内，故障一个orderer节点，不会影响整个网络的排序和同步区块。这里使用关闭一个orderer节点的方式进行测试，测试成功。综合上述多机联动测试，Fabric使用kafka共识搭建orderer集群成功。