环境搭建

环境说明

Ubuntu 18.04 64位

安装curl

首先利用如下命令安装依赖：

```
sudo apt install openssl libssl-dev
```

接着使用如下命令下载curl包并解压：

```
wget https://curl.se/download/curl-7.77.0.tar.gz
tar -zxf curl-7.77.0.tar.gz
```

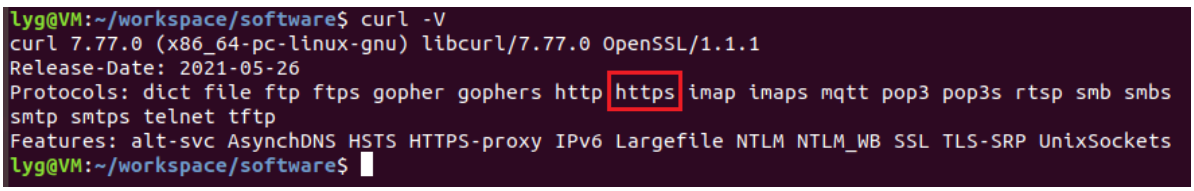
随后使用如下命令进行编译安装（需要make和gcc支持）：

```
cd curl-7.77.0
sudo ./configure --with-openssl
sudo make && sudo make install
```

> 可能遇到如下报错信息：
>
> > error while loading shared libraries: libcurl.so.4: cannot open shared object file: No such file or directory
>
> 解决方法是查看`libcurl`相关so文件所在位置，一般在`/usr/local/bin`目录下。然后在`/etc/ld.so.conf`文件中加入如下内容：
>
> ```
> /usr/local/lib  # 即库的位置
> ```
>
> 随后使用如下命令让配置文件生效：
>
> ```
> /sbin/ldconfig -v
> ```

安装结束后使用如下命令查看是否安装成功，同时注意要支持`https`协议：

```
curl -V
```



安装docker

如果之前有旧版本的docker，请先使用如下命令卸载：

```
sudo apt remove docker docker-engine docker.io
```

如果没有旧版本docker，直接使用如下命令安装。首先安装相关依赖：

```
sudo apt update
sudo apt install apt-transport-https ca-certificates software-properties-common
```

接着添加docker官方GPG密钥，命令如下：

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

随后使用如下命令设立仓库：

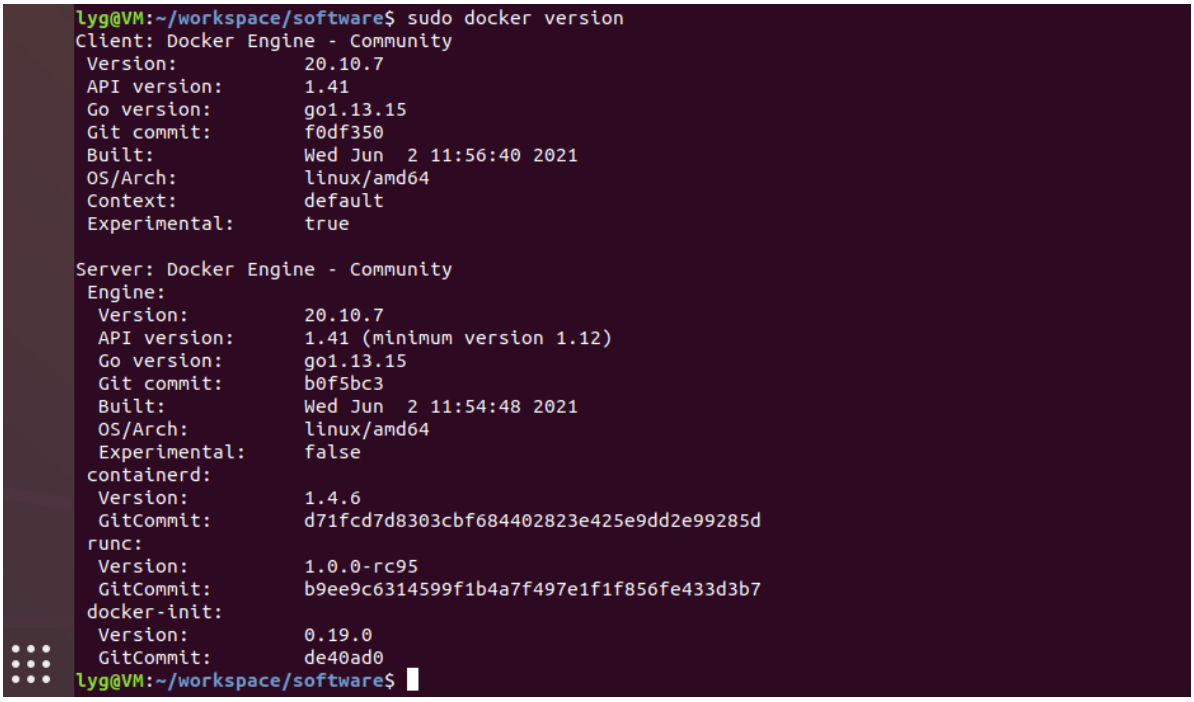
```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

接着使用如下命令安装docker：

```
sudo apt update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

接着使用如下命令查看docker是否安装完成，请保证较高的docker版本（19.03.11以上），否则后续可能失败：

```
sudo docker version
```



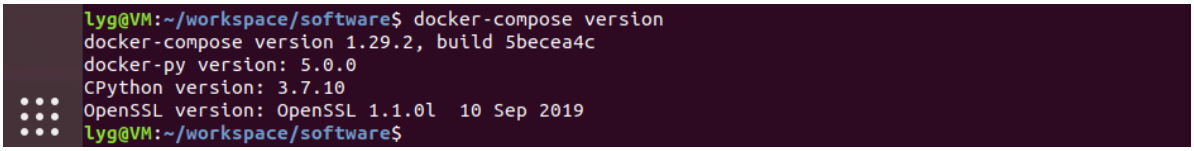
安装docker-compose

使用如下命令安装并赋予其执行权：

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

随后使用如下命令查看是否安装成功：

```
docker-compose version
```



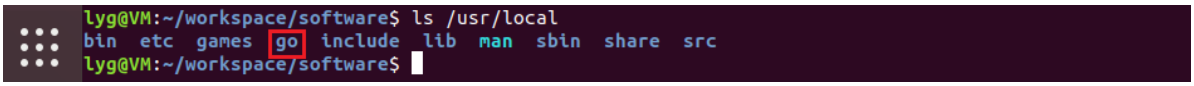
安装Go

在`https://golang.google.cn/dl/`网页下下载Go安装包，或直接使用如下命令下载：

```
wget https://golang.google.cn/dl/go1.16.5.linux-amd64.tar.gz
```

接着使用如下命令将文件解压进`/usr/local/go`文件夹中：

```
sudo cp go1.16.5.linux-amd64.tar.gz /usr/local
sudo tar -zxf go1.16.5.linux-amd64.tar.gz
```



接着使用如下命令建立Go工作文件夹：

```
cd ~
mkdir goDir
```

随后在`~/.bashrc`文件中声明Go相关路径，具体信息如下：

```
export GOPATH=/home/lyg/goDir
export GOROOT=/usr/local/go
export PATH=$PATH:$GOPATH/bin
export PATH=$PATH:$GOROOT/bin
```

接着使用`source ~/.bashrc`命令使配置生效。随后使用如下命令检测是否安装成功：

```
go version
```

安装fabric

安装依赖：

```
sudo apt install libtool libtdl-dev
```

拉取fabric源码

使用如下命令即可：

```
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger

git clone https://github.com/hyperledger/fabric.git
cd fabric
git branch -a
git checkout v1.4.3
```

拉取fabric-samples

使用如下命令即可：

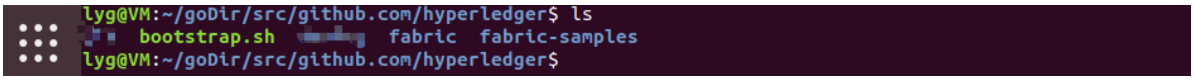
```
cd $GOPATH/src/github.com/hyperledger

git clone https://github.com/hyperledger/fabric-samples.git
cd ./fabric-samples
git branch -a
git checkout v1.4.3
```

拉取并配置依赖

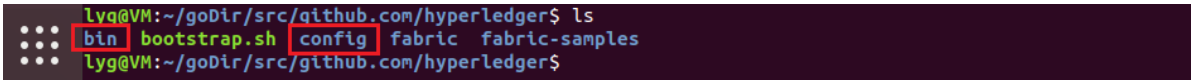
在**fabric/scripts**目录下找到**bootstrap.sh**脚本，复制到与**fabric**同级目录下，删除**bootstrap.sh**中的`samplesInstall()`和`binariesInstall()`两个方法。命令如下：

```
cd $GOPATH/src/github.com/hyperledger/fabric/scripts
cp bootstrap.sh ../../
```



hyperledger-fabric-linux-amd64-1.4.3.tar内有`bin`和`config`两个文件夹，***hyperledger-fabric-ca-linux-amd64-1.4.3.tar***内有`bin`文件夹，将两个`bin`文件夹内的二进制文件汇总在一个`bin`文件夹内。最后将`bin`和`config`文件夹复制到`fabric-samples`文件夹内。具体命令如下：

```
cd $GOPATH/src/github.com/hyperledger/fabric-samples
wget https://github.com/hyperledger/fabric/releases/download/v1.4.3/hyperledger-fabric-linux-amd64-1.4.3.tar.gz
tar -zxf hyperledger-fabric-linux-amd64-1.4.3.tar.gz
mv bin bin1
wget https://github.com/hyperledger/fabric-ca/releases/download/v1.4.3/hyperledger-fabric-ca-linux-amd64-1.4.3.tar.gz
tar -zxf hyperledger-fabric-ca-linux-amd64-1.4.3.tar.gz
mv bin1/* bin/
rm -rf bin1
```



```
lyg@VM:~/goDir/src/github.com/hyperledger/fabric-samples$ ls bin
configtxgen  configtxlator  cryptogen  discover  fabric-ca-client  idemixgen  orderer  peer
lyg@VM:~/goDir/src/github.com/hyperledger/fabric-samples$ ls config/
configtx.yaml  core.yaml  orderer.yaml
lyg@VM:~/goDir/src/github.com/hyperledger/fabric-samples$
```

fabric相关的docker镜像安装

由于docker官网镜像下载超级慢，因此需要使用docker镜像加速器。首先注册阿里云账号，然后登录`https://cr.console.aliyun.com`网站，按照下图方式配置即可：

具体为将配置文件中的加速器地址换成自己的地址即可，命令为：

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
"registry-mirrors": ["你的加速器网址"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

接着使用如下命令拉取镜像即可，该过程可能会较久：

```
cd $GOPATH/src/github.com/hyperledger
sudo ./bootstrap.sh 1.4.3 1.4.3 0.4.15
```

设置环境变量

在`~/.bashrc`文件中添加如下语句：

```
export PATH=$PATH:$GOPATH/src/github.com/hyperledger/fabric-samples/bin
```

随后使用`source ~/.bashrc`命令使配置生效即可。

测试

为保证正常网络正常启动，使用以下命令修改配置文件，并添加相应内容，如下：

```
cd $GOPATH/src/github.com/hyperledger/fabric-samples/first-network
# 在四个environment中添加："- GODEBUG=netdns=go"
vim $GOPATH/src/github.com/hyperledger/fabric-samples/first-network/base/docker-compose-base.yaml
vim $GOPATH/src/github.com/hyperledger/fabric-samples/chaincode-docker-devmode/docker-compose-simple.yaml
```

接着使用如下命令测试：

```
cd $GOPATH/src/github.com/hyperledger/fabric-samples/first-network
./byfn.sh up
```

```
df5b069997e5   hyperledger/fabric-peer:latest      "peer node start"   6 seconds ago   Up 1 secon
d             0.0.0.0:10051->10051/tcp, :::10051->10051/tcp   peer1.org2.example.com

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

Build your first network (BYFN) end-to-end test

Channel name : mychannel
Creating channel...
+ peer channel create -o orderer.example.com:7050 -c mychannel -f ./channel-artifacts/channel.tx
--tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganization
s/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
+ res=0
+ set +x
2021-06-24 15:14:57.397 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connecti
ons initialized
2021-06-24 15:14:57.437 UTC [cli.common] readBlock -> INFO 002 Received block: 0
===================== Channel 'mychannel' created =====================
```

当出现如下结果后，说明环境搭建完成：

```
===================== Chaincode is installed on peer1.org2 =====================

Querying chaincode on peer1.org2...
===================== Querying on peer1.org2 on channel 'mychannel'... =====================
Attempting to Query peer1.org2 ...3 secs
+ peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
+ res=0
+ set +x

90
===================== Query successful on peer1.org2 on channel 'mychannel' =====================


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


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

lyg@VM:~/goDir/src/github.com/hyperledger/fabric-samples/first-network$
```

随后使用如下命令关闭网络即可：

```
./byfn.sh down
```

至此，环境搭建完成。

任务1：Building Your First Network

> 要求：根据参考链接的资料，在不使用官方 bash 脚本，使用现 有工具的前提的情况下，使用终端逐步完成部署、链码安装、初始化、 调用、查询等相关操作。

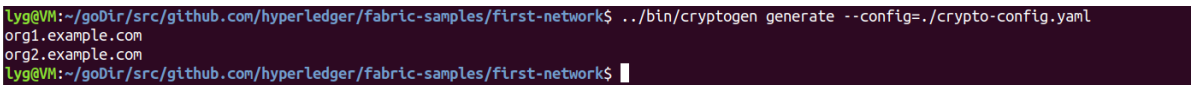
```
../bin/cryptogen generate --config=./crypto-config.yaml
export FABRIC_CFG_PATH=$PWD
../bin/configtxgen -profile TwoOrgsOrdererGenesis -channelID byfn-sys-channel -
outputBlock ./channel-artifacts/genesis.block
export CHANNEL_NAME=mychannel && ../bin/configtxgen -profile TwoOrgsChannel -
outputCreateChannelTx ./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
```

运行工具

首先运行cryptogen工具。由于二进制文件在bin目录下，所以需要提供工具所在的相对路径，如下：

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

此时会得到如下图所示：



证书和钥匙（即MSP）将被输出到某一目录根部的目录即crypto-config中。接下来，需要告诉configtxgen工具在哪里寻找它需要摄入的configtx.yaml文件。我们将告诉它在当前工作目录中寻找。

```
export FABRIC_CFG_PATH=$PWD
```

然后，我们将调用configtxgen工具来创建订购者创世块：

```
../bin/configtxgen -profile TwoOrgsOrdererGenesis -channelID byfn-sys-channel -
outputBlock ./channel-artifacts/genesis.block
```

接下来，需要创建通道交易工件。请确保替换$CHANNEL_NAME或将CHANNEL_NAME设置为环境变量：

```
export CHANNEL_NAME=mychannel && ../bin/configtxgen -profile TwoOrgsChannel -
outputCreateChannelTx ./channel-artifacts/channel.tx -channelID $CHANNEL_NAME
```

接下来，在构建的通道上为Org1定义锚点对等体，终端输出将模仿通道交易工件的输出：

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

现在，我们将在同一通道上为Org2定义锚点对等体：

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

启动网络

首先，打开网络：

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

接着使用docker exec命令进入CLI容器，注意需要root权限：

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

我们用-c标志指定我们的通道名称，用-f标志指定我们的通道配置事务。在上述情况下为channel.tx，但是可以用不同的名字来装载自己的配置事务。随后，再次在CLI容器中设置CHANNEL_NAME环境变量，这样就不必明确地传递这个参数。频道名称必须全部为小写，小于250个字符长，并匹配常规表达。

```
export CHANNEL_NAME=mychannel
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/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

现在，使用如下命令加入到通道：

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

尽管没有加入每一个对等体，而只是加入peer0.org2.example.com，这样就可以正确地更新通道中的锚定对等体定义。这是由于以下命令将覆盖CLI容器中的默认环境变量，此完整命令如下：

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:9051 CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
peer channel join -b mychannel.block
```

更新通道定义，将Org1的锚点对等体定义为：

```
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/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

现在更新通道定义，将Org2的锚定对等体定义为peer0.org2.example.com。与Org2对等体的对等体通道加入命令相同，使用如下命令设置适当的环境变量来预示这个调用：

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:9051 CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.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/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

应用程序通过chaincode与区块链账本进行交互。因此，需要在每个将执行和认可我们交易的对等体上安装链码，然后将链码实例化到通道上。首先，在Org1的peer0节点上安装Go的chaincode样本。这些命令将指定的源代码味道放到我们的peer的文件系统上。命令如下：

```
peer chaincode install -n mycc -v 1.0 -p
github.com/chaincode/chaincode_example02/go/
```

当在通道上实例化链码时，背书策略将被设置为需要来自Org1和Org2中的一个对等体的背书。因此，还需要在Org2的一个对等体上安装链码。修改以下四个环境变量，对Org2的peer0发出安装命令：

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:9051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

现在将Go的chaincode样本安装到Org2的peer0上，这些命令将指定的源代码味道放到我们对等体的文件系统中，命令如下：

```
peer chaincode install -n mycc -v 1.0 -p
github.com/chaincode/chaincode_example02/go/
```

接下来，在通道上实例化chaincode。这将初始化通道上的chaincode，为chaincode设置背书策略，并为目标peer启动一个chaincode容器。请注意-P参数，该策略指定了针对该链码的交易所需的背书级别，以便进行验证。

在下面的命令中，指定的策略是-P "AND ('Org1MSP.peer', 'Org2MSP.peer')"，这意味着需要属于Org1和Org2的同伴的 "认可"（即两个认可）；而如果把语法改为OR，那么就只需要一个背书。命令如下：

```
peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.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
"AND ('Org1MSP.peer','Org2MSP.peer')"
```

上述配置过程如下图所示：

测试：查询调用与安装

首先，查询a的值以确保链码被正确地实例化，状态数据库被填充，查询的语法如下：

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

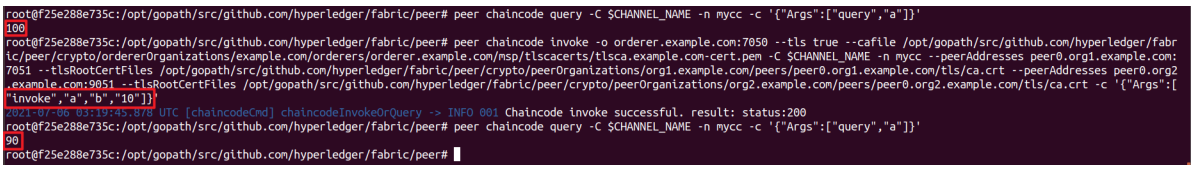
现在把10从a移到b。这个事务将切割一个新的块并更新状态DB，调用的语法如下：

```
peer chaincode invoke -o orderer.example.com:7050 --tls true --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/e
xample.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
-C $CHANNEL_NAME -n mycc --peerAddresses peer0.org1.example.com:7051 --
tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1
.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses
peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2
.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"Args":
["invoke","a","b","10"]}'
```

接着确认之前的调用是正确执行的。用100的值初始化了键a，并在之前的调用中删除了10，因此，对a的查询应该返回90。查询的语法如下：

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

观察到上述过程成功执行并得到如下结果：



现在将在第三个对等体，即Org2中的peer1上安装chincode。修改以下四个环境变量，对Org2的peer1发出安装命令：

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypt
o/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer1.org2.example.com:10051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/c
rypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
```

现在将Go的chaincode样本安装到Org2的peer1上，这些命令将指定的源代码味道放到我们对等体的文件系统中：

```
peer chaincode install -n mycc -v 1.0 -p
github.com/chaincode/chaincode_example02/go/
```

随后确认可以向Org2中的Peer1发出查询。用100的值初始化了键a，并在之前的调用中删除了10，因此，对a的查询仍应返回90。Org2中的peer1必须首先加入通道，然后才能响应查询，可以通过发出以下命令来加入通道：

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypt
o/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer1.org2.example.com:10051 CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/c
rypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
peer channel join -b mychannel.block
```

随后利用如下命令进行查询：

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

观察到成功返回结果如下（包含配置过程）：

任务2：chaincode编写

> 在完成前一个任务的前提下，使用 Go 语言编写一个用于投票的 chaincode，并完成其功能性测试（如投票，统计，查询等）。可以选13 择多种测试手段，如编写 chaincode 的测试代码，或者是在 dev 模式 下，使用命令行测试。

代码

在“~/goDir/src/github.com/hyperledger/fabric-samples/chaincode”下创建“vote”文件夹，并编写投票链码“vote.go”，如下：

```
package main

import(
    "fmt"
    "encoding/json"
    "bytes"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

type VoteChaincode struct {
}
```

```
type Vote struct {
    Username string `json:"username"`
    Votenum int `json:"votenum"`
}

func (t *VoteChaincode) Init(stub shim.ChaincodeStubInterface) peer.Response {
    return shim.Success(nil)
}

func (t *VoteChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {

    fn , args := stub.GetFunctionAndParameters()

    if fn == "voteUser" {
        return t.voteUser(stub,args)
    } else if fn == "getUserVote" {
        return t.getUserVote(stub,args)
    }

    return shim.Error("Invoke 调用方法有误！")
}

func (t *VoteChaincode) voteUser(stub shim.ChaincodeStubInterface , args
[]string) peer.Response{
    // 查询当前用户的票数，如果用户不存在则新添一条数据，如果存在则给票数加1
    fmt.Println("start voteUser")
    vote := Vote{}
    username := args[0]
    voteAsBytes, err := stub.GetState(username)

    if err != nil {
        shim.Error("voteUser 获取用户信息失败！")
    }

    if voteAsBytes != nil {
        err = json.Unmarshal(voteAsBytes, &vote)
        if err != nil {
            shim.Error(err.Error())
        }
        vote.Votenum += 1
    }else {
        vote = Vote{ Username: args[0], Votenum: 1}
    }

    //将 Vote 对象 转为 JSON 对象
    voteJsonAsBytes, err := json.Marshal(vote)
    if err != nil {
        shim.Error(err.Error())
    }

    err = stub.PutState(username,voteJsonAsBytes)
    if err != nil {
        shim.Error("voteUser 写入账本失败！")
    }

    fmt.Println("end voteUser")
    return shim.Success(nil)
```

```
}

func (t *VoteChaincode) getUserVote(stub shim.ChaincodeStubInterface, args
[]string) peer.Response{

    fmt.Println("start getUserVote")
    // 获取所有用户的票数
    resultIterator, err := stub.GetStateByRange("","")
    if err != nil {
        return shim.Error("获取用户票数失败！")
    }
    defer resultIterator.Close()

    var buffer bytes.Buffer
    buffer.WriteString("[")

    isWritten := false

    for resultIterator.HasNext() {
        queryResult , err := resultIterator.Next()
        if err != nil {
            return shim.Error(err.Error())
        }

        if isWritten == true {
            buffer.WriteString(",")
        }

        buffer.WriteString(string(queryResult.Value))
        isWritten = true
    }

    buffer.WriteString("]")

    fmt.Printf("查询结果：\n%s\n",buffer.String())
    fmt.Println("end getUserVote")
    return shim.Success(buffer.Bytes())
}

func main(){
    err := shim.Start(new(VoteChaincode))
    if err != nil {
        fmt.Println("vote chaincode start err")
    }
}
```

测试

首先执行如下命令保证环境正确：

```
./byfn.sh down
docker stop $(docker ps -aq)
docker rm $(docker ps -aq)
docker volume prune
sudo service docker restart
sudo service network-manager restart
./byfn.sh up  # 要保证可以正常启动
./byfn.sh down  # 可以正常启动后关闭
```

使用如下命令启动网络：

```
cd $GOPATH/src/github.com/hyperledger/fabric-samples/chaincode-docker-devmode
docker-compose -f docker-compose-simple.yaml up
```

在新的终端中编译链码并启动，命令如下：

```
docker exec -it chaincode bash
cd vote
go build
CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=mycc:0 ./vote
```

打开新终端，进入docker，命令如下：

```
docker exec -it cli bash
```

随后使用如下命令安装链码：

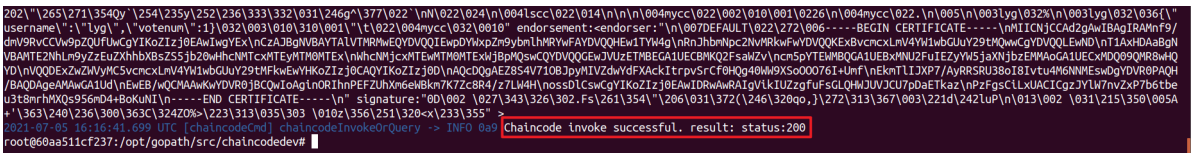
```
peer chaincode install -p chaincodedev/chaincode/vote -n mycc -v 0
```

接着实例化链码，命令如下：

```
peer chaincode instantiate -n mycc -v 0 -c '{"Args":[]}' -C myc
```

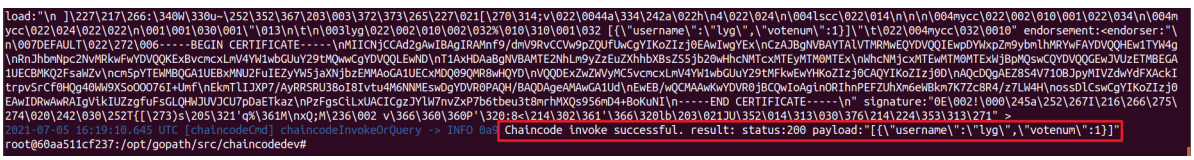
创建投票人信息，初始创建票数加一：

```
peer chaincode invoke -n mycc -c '{"Args":["voteUser", "lyg"]}' -C myc
```

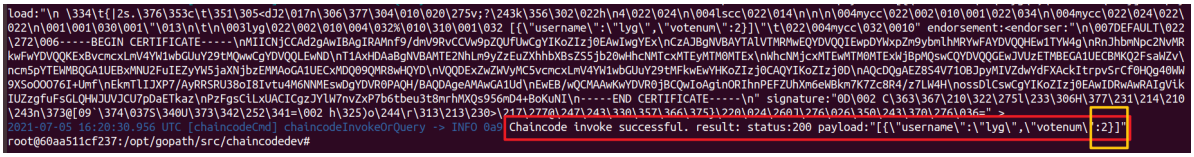


查询投票人信息：

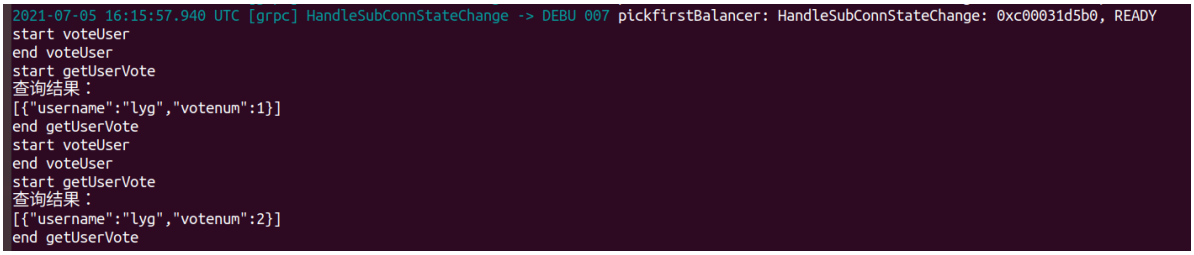
```
peer chaincode invoke -n mycc -c '{"Args":["getUserVote"]}' -C myc
```



再次执行投票后查看，观察到票数变为2，如下图：


Chaincode invoke successful. result: status:200 payload:"[{\"username\":\"lyg\",\"votenum\":2}]"

查看输出结果，符合预期：


2021-07-05 16:15:57.940 UTC [grpc] HandleSubConnStateChange -> DEBU 007 pickfirstBalancer: HandleSubConnStateChange: 0xc00031d5b0, READY
start voteUser
end voteUser
start getUserVote
查询结果：
[{"username":"lyg","votenum":1}]
end getUserVote
start voteUser
end voteUser
start getUserVote
查询结果：
[{"username":"lyg","votenum":2}]
end getUserVote