Hyperledger Fabric SDK Go:构建第一个应用-初始化环境

目录

一. 概述

首先，为新入门的开发小白普及一下何为SDK

软件开发工具包（外语首字母缩写：SDK、外语全称：Software Development Kit）一般都是一些软件工程师为特定的软件包、软件框架、硬件平台、操作系统等建立应用软件时的开发工具的集合。软件开发工具包括广义上指辅助开发某一类软件的相关文档、范例和工具的集合。软件开发工具包是一些被软件工程师用于为特定的软件包、软件框架、硬件平台、操作系统等创建应用软件的开发工具的集合，一般而言SDK即开发 Windows 平台下的应用程序所使用的 SDK。它可以简单的为某个程序设计语言提供应用程序接口 API 的一些文件，但也可能包括能与某种嵌入式系统通讯的复杂的硬件。一般的工具包括用于调试和其他用途的实用工具。SDK 还经常包括示例代码、支持性的技术注解或者其他的为基本参考资料澄清疑点的支持文档。为了鼓励开发者使用其系统或者语言，许多 SDK 是免费提供的。

Farbric的Peer节点和Orderer节点都提供了基于GRPC协议(Google开发的远程过程调用RPC)的接口，通过这些接口可以和Peer节点与Orderer节点进行命令/数据交互，为了简化开发，官方提供了多语言版本的SDK，官网原文

```
Hyperledger Fabric SDKs
Hyperledger Fabric intends to offer a number of SDKs for a wide variety of
programming languages. The first two delivered are the Node.js and Java
SDKs. We hope to provide Python, REST and Go SDKs in a subsequent release.

Hyperledger Fabric Node SDK documentation.
Hyperledger Fabric Java SDK documentation.
```

实际上目前主流支持的已经有Go版本了，列出主流的三个：

- Fabric Nodejs SDK
- Fabric Java SDK
- Fabric Go SDK

考虑到Golang是Fabric原生的开发语言，Fabric，Fabric-ca,Chaincode都是采用Golang开发的，所以本文还是围绕Golang版本的Fabric SDK进行阐述SDK的安装部署与测试。

本文内容翻译自:https://chainhero.io/2018/03/tutorial-build-blockchain-app-2/ ，文档中的命令操作均在实际环境进行验证，现将成果分享给大家。

二.安装环境

在Ubuntu 16.04上发布，但Hyperledger Fabric架构与Mac OS X，Windows和其他Linux发行版兼容。Hyperledger Fabric使用Docker轻松部署区块链网络。 另外，一些组件（同级）也部署docker容器来分离数据（通道）。 所以请确保所使用的平台支持这种虚拟化。

1. 查看Docker版本

需要Docker版本17.03.0-ce或更高版本。

```
$docker -v
```

返回结果

```
Docker version 17.12.1-ce, build 7390fc6
```

2. 查看Docker-Compose版本

```
$docker-compose  version
```

返回结果

```
docker-compose version 1.17.1, build unknown
docker-py version: 2.5.1
CPython version: 2.7.15rc1
OpenSSL version: OpenSSL 1.1.0g  2 Nov 2017
```

3. 查看Golang版本

需要版本1.9.x或更高版本

```
$go version
```

返回结果

```
go version go1.10.3 linux/amd64
```

查看GOPATH,GOROOT,GOBIN环境变量

```
$ go env |egrep 'GOROOT|GOPATH|GOBIN'
```

返回结果

```
GOBIN="/home/bruce/go/bin"
GOPATH="/home/bruce/go"
GOROOT="/usr/local/go"
```

三.安装Fabric SDK Go

1. 安装依赖包

```
$ sudo apt update
$ sudo apt install libltdl-dev
```

2. 安装SDK

(1) 下载软件包

```
$ go get -u github.com/hyperledger/fabric-sdk-go
```

(2) 安装依赖包

```
$ cd $GOPATH/src/github.com/hyperledger/fabric-sdk-go
$ chmod +x test/scripts/*.sh # make depend-install操作会调用dependencies.sh脚
本
$ make depend //注意1.1.0版本是make depend-install
```

以上命令会下载如下依赖包并安装至$GOBIN目录下

```
github.com/axw/gocov/...
github.com/AlekSi/gocov-xml
github.com/client9/misspell/cmd/misspell
github.com/golang/lint/golint
golang.org/x/tools/cmd/goimports
github.com/golang/mock/mockgen
```

安装完成后检查$GOBIN目录下文件

```
-rwxrwxr-x 1 bruce bruce 13128127 Jul 19 17:38 dep
-rwxrwxr-x 1 bruce bruce  4332114 Jul 19 17:50 gocov
-rwxrwxr-x 1 bruce bruce  2752093 Jul 19 17:50 gocov-xml
-rwxrwxr-x 1 bruce bruce  5220554 Jul 19 17:50 goimports
-rwxrwxr-x 1 bruce bruce  5669065 Jul 19 17:50 golint
-rwxrwxr-x 1 bruce bruce  9470763 Jul 19 17:50 misspell
-rwxrwxr-x 1 bruce bruce  5070526 Jul 19 17:51 mockgen
```

(3) 安装vendor

```
$ make populate
```

返回结果

```
Populate script last ran 07-21-2018 on revision e230c04e with Gopkg.lock
revision d489eba9
Populating vendor ...
Populating dockerd vendor ...
Cloning into
'scripts/_go/src/chaincoded/vendor/github.com/hyperledger/fabric'...
remote: Counting objects: 4530, done.
remote: Compressing objects: 100% (3778/3778), done.
remote: Total 4530 (delta 543), reused 2596 (delta 376), pack-reused 0
Receiving objects: 100% (4530/4530), 16.51 MiB | 120.00 KiB/s, done.
Resolving deltas: 100% (543/543), done.
```

四.启动区块链网络

1. 准备环境

为了构建区块链网络，使用`docker`构建处理不同角色的虚拟计算机。 在这里我们将尽可能保持简单。 Hyperledger Fabric需要大量证书来确保在整个端到端流程（TSL，身份验证，签名块.....）期间进行加密。 创建这些文件需要一点时间，为了直接了解问题的核心，我们已经在此存储库的文件夹中为您准备了所有相关内容。

在`GOPATH`的`src`文件夹中新建一个目录如下：

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

新建`fixtures`文件夹

```
$ mkdir fixtures
```

将`channel-artifacts`及`crypto-config`两个文件夹复制到`fixture`目录中

```
$ cd fixtures
$ sudo cp -r ~/hyfa/fabric-samples/first-network/channel-artifacts .
$ sudo cp -r ~/hyfa/fabric-samples/first-network/crypto-config .
```

将`channel-artifacts`文件夹名称修改为`artifacts`

```
$ mv channel-artifacts/ artifacts
```

移除无用的文件

```
$ sudo rm -f artifacts/.gitkeep
```

2. 修改配置

将`fabric-samples/basic-network/docker-compose.yml`文件复制至当前的`fixtures`目录下, 进行编辑

```
$ sudo cp ~/hyfa/fabric-samples/basic-network/docker-compose.yml ./
$ sudo vim docker-compose.yml
```

(1) 修改网络模式

```
version: '2'

networks:
  default:
```

(2) orderer部分

```
services:
 orderer.example.com:
    container_name: orderer.example.com
    image: hyperledger/fabric-orderer
    environment:
      - ORDERER_GENERAL_LOGLEVEL=debug
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_GENESISMETHOD=file
      -
ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
      - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
      - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
      - ORDERER_GENERAL_LISTENPORT=7050
      # enabled TLS
      - ORDERER_GENERAL_TLS_ENABLED=true
      -
ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
      -
ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
      - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt,
/var/hyperledger/peerOrg1/tls/ca.crt, /var/hyperledger/peerOrg2/tls/ca.crt]
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
    command: orderer
    ports:
      - 7050:7050
    volumes:
      -
./artifacts/genesis.block:/var/hyperledger/orderer/orderer.genesis.block
      - ./crypto-
config/ordererOrganizations/example.com/orderers/orderer.example.com/msp:/v
ar/hyperledger/orderer/msp
      - ./crypto-
config/ordererOrganizations/example.com/orderers/orderer.example.com/tls:/v
ar/hyperledger/orderer/tls
      - ./crypto-
config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/:/va
r/hyperledger/peerOrg1
      - ./crypto-
config/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/:/va
r/hyperledger/peerOrg2
    networks:
```

```
      default:
        aliases:
          - orderer.example.com
```

(3) ca配置

```
  ca.org1.example.com:
    image: hyperledger/fabric-ca
    environment:
      - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
      - FABRIC_CA_SERVER_CA_NAME=ca.org1.example.com
      - FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-
config/ca.org1.example.com-cert.pem
      - FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-
config/b4a9a9585aebe52646e1987d4eca4a0ecf3f0ab688ca7924c07249c0303553ba_sk
      - FABRIC_CA_SERVER_TLS_ENABLED=true
      - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-
config/ca.org1.example.com-cert.pem
      - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-
config/b4a9a9585aebe52646e1987d4eca4a0ecf3f0ab688ca7924c07249c0303553ba_sk
    ports:
      - "7054:7054"
    command: sh -c 'fabric-ca-server start -b admin:adminpw -d'
    volumes:
      - ./crypto-
config/peerOrganizations/org1.example.com/ca/:/etc/hyperledger/fabric-ca-
server-config
    container_name: ca.org1.example.com
    networks:
      default:
        aliases:
          - ca.org1.example.com
```

> 注意:
>
> FABRIC_CA_SERVER_CA_KEYFILE与FABRIC_CA_SERVER_TLS_KEYFILE的参数值需要设置成
>
> ...fixtures/crypto-config/peerOrganizations/org1.example.com/ca目录下面的私钥文件,否则会启动失败，报错证书与秘钥不匹配。

(4) Peer配置

- peer0.org1.example.com 内容如下

```
  peer0.org1.example.com:
    image: hyperledger/fabric-peer
    container_name: peer0.org1.example.com
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_VM_DOCKER_ATTACHSTDOUT=true
      - CORE_LOGGING_LEVEL=DEBUG
      - CORE_PEER_NETWORKID=bill
      - CORE_PEER_PROFILE_ENABLED=true
      - CORE_PEER_TLS_ENABLED=true
      - CORE_PEER_TLS_CERT_FILE=/var/hyperledger/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/var/hyperledger/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/var/hyperledger/tls/ca.crt
      - CORE_PEER_ID=peer0.org1.example.com
      - CORE_PEER_ADDRESSAUTODETECT=true
      - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
      - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
      - CORE_PEER_GOSSIP_USELEADERELECTION=true
      - CORE_PEER_GOSSIP_ORGLEADER=false
      - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_MSPCONFIGPATH=/var/hyperledger/msp
      - CORE_PEER_TLS_SERVERHOSTOVERRIDE=peer0.org1.example.com
    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/peer0.org1.example.com/msp:
/var/hyperledger/msp
      - ./crypto-
config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls:
/var/hyperledger/tls
    ports:
      - 7051:7051
      - 7053:7053
    depends_on:
      - orderer.example.com
    links:
      - orderer.example.com
    networks:
      default:
        aliases:
```

```
      - peer0.org1.example.com
```

- peer1.org1.example.com内容如下

```
  peer1.org1.example.com:
    image: hyperledger/fabric-peer
    container_name: peer1.org1.example.com
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_VM_DOCKER_ATTACHSTDOUT=true
      - CORE_LOGGING_LEVEL=DEBUG
      - CORE_PEER_NETWORKID=bill
      - CORE_PEER_PROFILE_ENABLED=true
      - CORE_PEER_TLS_ENABLED=true
      - CORE_PEER_TLS_CERT_FILE=/var/hyperledger/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/var/hyperledger/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/var/hyperledger/tls/ca.crt
      - CORE_PEER_ID=peer1.org1.example.com
      - CORE_PEER_ADDRESSAUTODETECT=true
      - CORE_PEER_ADDRESS=peer1.org1.example.com:7051
      - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer1.org1.example.com:7051
      - CORE_PEER_GOSSIP_USELEADERELECTION=true
      - CORE_PEER_GOSSIP_ORGLEADER=false
      - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_MSPCONFIGPATH=/var/hyperledger/msp
      - CORE_PEER_TLS_SERVERHOSTOVERRIDE=peer1.org1.example.com
    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:
/var/hyperledger/msp
      - ./crypto-
config/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls:
/var/hyperledger/tls
    ports:
      - 8051:7051
      - 8053:7053
    depends_on:
      - orderer.example.com
```

```
    links:
      - orderer.example.com
    networks:
      default:
        aliases:
          - peer1.org1.example.com
```

暂时只需要如上功能模块即可

将 `fixtures` 文件的所属修改为当前用户及组

```
$ sudo chown -R bruce:bruce ../fixtures
```

3. 启动网络

为了检查网络是否正常工作，使用`docker-compose`同时启动或停止所有容器。 进入`fixtures`文件夹，运行：

```
$ cd $GOPATH/src/github.com/kongyixueyuan.com/bill/fixtures
$ docker-compose up
```

启动完毕，将看到：两个peer，orderer和一个CA容器。 代表已成功创建了一个新的网络，可以随SDK一起使用。 要停止网络，请返回到上一个终端，按`Ctrl+C`并等待所有容器都停止。

```
CONTAINER ID        IMAGE                          COMMAND
CREATED             STATUS              PORTS
             NAMES
932b5364664f        hyperledger/fabric-peer      "peer node start"
26 hours ago        Up 26 hours         0.0.0.0:7051->7051/tcp,
0.0.0.0:7053->7053/tcp   peer0.org1.example.com
cf9385a5e1ae        hyperledger/fabric-peer      "peer node start"
26 hours ago        Up 26 hours         0.0.0.0:8051->7051/tcp,
0.0.0.0:8053->7053/tcp   peer1.org1.example.com
a1cd2a83af57        hyperledger/fabric-orderer   "orderer"
26 hours ago        Up 26 hours         0.0.0.0:7050->7050/tcp
              orderer.example.com
6a9a54d9d82b        hyperledger/fabric-ca        "/bin/sh -c 'fabric-…"
26 hours ago        Up 26 hours         0.0.0.0:7054->7054/tcp
              ca.org1.example.com
```

> **提示**：当网络停止时，所有使用的容器都可以访问。 例如，这对检查日志非常有用。 可以用`docker ps -a`来看它们。 为了清理这些容器，需要使用`docker rm $(docker ps -aq)`将其删除，或者如果使用了`docker-compose`文件，请转至此文件的位置并运行`docker-compose down` 。

> **提示**：可以在后台运行`docker-compose`命令以保持提示。 为此，请使用参数`-d` ，如下所示：`docker-compose up -d` 。 要停止容器，请在`docker-compose.yaml`所在的文件夹中运行命令：`docker-compose stop` （或`docker-compose down`进行清理停止所有容器）。

五.使用Fabric SDK Go

1.创建配置文件

```
$ cd $GOPATH/src/github.com/ticket
$ vim config.yaml
```

配置文件内容如下：

```
name: "ticket-network"

# Describe what the target network is/does.
description: "The network which will host my first blockchain"

# Schema version of the content. Used by the SDK to apply the corresponding
parsing rules.
version: 2

# The client section used by GO SDK.
client:
  # Which organization does this application instance belong to? The value
must be the name of an org
  organization: Org1
  logging:
    level: info

  # Global configuration for peer, event service and orderer timeouts
  peer:
    timeout:
      connection: 3s
      queryResponse: 45s
      executeTxResponse: 30s
```

```
  eventService:
    timeout:
      connection: 3s
      registrationResponse: 3s
  orderer:
    timeout:
      connection: 3s
      response: 5s

  # Root of the MSP directories with keys and certs. The Membership Service
Providers is component that aims to offer an abstraction of a membership
operation architecture.
  cryptoconfig:
    path: "${GOPATH}/src/github.com/kongyixueyuan.com/bill/fixtures/crypto-
config"

  # Some SDKs support pluggable KV stores, the properties under
"credentialStore" are implementation specific
  credentialStore:
    path: "/tmp/bill-kvs"

     # [Optional]. Specific to the CryptoSuite implementation used by GO
SDK. Software-based implementations requiring a key store. PKCS#11 based
implementations does not.
    cryptoStore:
      path: "/tmp/bill-msp"

  # BCCSP config for the client. Used by GO SDK. It's the Blockchain
Cryptographic Service Provider.
  # It offers the implementation of cryptographic standards and algorithms.
  BCCSP:
    security:
     enabled: true
     default:
      provider: "SW"
     hashAlgorithm: "SHA2"
     softVerify: true
     ephemeral: false
     level: 256

  tlsCerts:
    systemCertPool: false

# [Optional]. But most apps would have this section so that channel objects
can be constructed based on the content below.
# If one of your application is creating channels, you might not use this
channels:
  mychannel:
    orderers:
```

```
      - orderer.example.com

    # Network entity which maintains a ledger and runs chaincode containers
in order to perform operations to the ledger. Peers are owned and maintained
by members.
    peers:
      peer0.org1.example.com:
        # [Optional]. will this peer be sent transaction proposals for
endorsement? The peer must
        # have the chaincode installed. The app can also use this property
to decide which peers
        # to send the chaincode install request. Default: true
        endorsingPeer: true

        # [Optional]. will this peer be sent query proposals? The peer must
have the chaincode
        # installed. The app can also use this property to decide which
peers to send the
        # chaincode install request. Default: true
        chaincodeQuery: true

        # [Optional]. will this peer be sent query proposals that do not
require chaincodes, like
        # queryBlock(), queryTransaction(), etc. Default: true
        ledgerQuery: true

        # [Optional]. will this peer be the target of the SDK's listener
registration? All peers can
        # produce events but the app typically only needs to connect to one
to listen to events.
        # Default: true
        eventSource: true

      peer1.org1.example.com:

# List of participating organizations in this network
organizations:
  Org1:
    mspid: Org1MSP
    cryptoPath:
"peerOrganizations/org1.example.com/users/{userName}@org1.example.com/msp"
    peers:
      - peer0.org1.example.com
      - peer1.org1.example.com
    certificateAuthorities:
      - ca.org1.example.com

# List of orderers to send transaction and channel create/update requests
to.
```

```
# The orderers consent on the order of transactions in a block to be
committed to the ledger. For the time being only one orderer is needed.
orderers:
  orderer.example.com:
    url: grpcs://localhost:7050
    grpcOptions:
      ssl-target-name-override: orderer.example.com
      grpc-max-send-message-length: 15
    tlsCACerts:
      path: "${GOPATH}/src/github.com/ticket/fixtures/crypto-
config/ordererOrganizations/example.com/tlsca/tlsca.example.com-cert.pem"

# List of peers to send various requests to, including endorsement, query
and event listener registration.
peers:
  peer0.org1.example.com:
    # this URL is used to send endorsement and query requests
    url: grpcs://localhost:7051
    # this URL is used to connect the EventHub and registering event
listeners
    eventUrl: grpcs://localhost:7053
    # These parameters should be set in coordination with the keepalive
policy on the server
    grpcOptions:
      ssl-target-name-override: peer0.org1.example.com
      grpc.http2.keepalive_time: 15

    tlsCACerts:
      path: "${GOPATH}/src/github.com/ticket/fixtures/crypto-
config/peerOrganizations/org1.example.com/tlsca/tlsca.org1.example.com-
cert.pem"

  peer1.org1.example.com:
    url: grpcs://localhost:8051
    eventUrl: grpcs://localhost:8053
    grpcOptions:
      ssl-target-name-override: peer1.org1.example.com
      grpc.http2.keepalive_time: 15
    tlsCACerts:
      # Certificate location absolute path
      path: "${GOPATH}/src/github.com/ticket/fixtures/crypto-
config/peerOrganizations/org1.example.com/tlsca/tlsca.org1.example.com-
cert.pem"

# Fabric-CA is a special kind of Certificate Authority provided by
Hyperledger Fabric which allows certificate management to be done via REST
APIs.
certificateAuthorities:
  ca.org1.example.com:
```

```
    url: https://localhost:7054
    # the properties specified under this object are passed to the 'http'
client verbatim when making the request to the Fabric-CA server
    httpOptions:
      verify: false
    registrar:
      enrollId: admin
      enrollSecret: adminpw
    caName: ca.org1.example.com
```

以上配置文件模板可以通过config.yaml获取

2. 编写初始化代码

创建一个`blockchain`的新文件夹，其中将包含与区块链网络通讯的所有接口。

```
$ mkdir blockchain
$ vim blockchain/setup.go
```

代码如下

```
package blockchain

import (
    "github.com/hyperledger/fabric-sdk-go/api/apitxn/resmgmtclient"
    "github.com/hyperledger/fabric-sdk-go/pkg/fabsdk"
    "github.com/hyperledger/fabric-sdk-go/pkg/config"
    "fmt"
    "github.com/hyperledger/fabric-sdk-go/api/apitxn/chmgmtclient"
    "time"
)

//定义结构体
type FabricSetup struct {
    ConfigFile    string                          //sdk配置文件所在路径
    ChannelID     string                          //应用通道名称
    ChannelConfig string                          //应用通道交易配置文件所在路
径
    OrgAdmin      string                          // 组织管理员名称
    OrgName       string                          //组织名称
    Initialized   bool                            //是否初始化
    Admin         resmgmtclient.ResourceMgmtClient //fabric环境中资源管理者
    SDK           *fabsdk.FabricSDK               //SDK实例
}

//1. 创建SDK实例并使用SDK实例创建应用通道，将Peer节点加入到创建的应用通道中
func (f *FabricSetup) Initialize() error {
    //判断是否已经初始化
```

```
    if f.Initialized {
        return fmt.Errorf("SDK已被实例化")
    }

    //创建SDK对象
    sdk, err := fabsdk.New(config.FromFile(f.ConfigFile))

    if err != nil {
        return fmt.Errorf("SDK实例化失败:%v", err)
    }
    f.SDK = sdk
    //创建一个具有管理权限的应用通道客户端管理对象
    chmClient, err := f.SDK.NewClient(fabsdk.WithUser(f.OrgAdmin),
fabsdk.WithOrg(f.OrgName)).ChannelMgmt()
    if err != nil {
        return fmt.Errorf("创建应用通道管理客户端管理对象失败,%v", err)
    }
    //获取当前的会话用户对象
    session, err := f.SDK.NewClient(fabsdk.WithUser(f.OrgAdmin),
fabsdk.WithOrg(f.OrgName)).Session()
    if err != nil {
        return fmt.Errorf("获取当前会话用户对象失败%v", err)
    }

    orgAdminUser := session
    //指定创建应用通道所需要的所有参数
    /*
    $ 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/ordererOrganizati
ons/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.c
om-cert.pem
     */
    chReq := chmgmtclient.SaveChannelRequest{ChannelID: f.ChannelID,
ChannelConfig: f.ChannelConfig, SigningIdentity: orgAdminUser}

    //创建应用通道

    err = chmClient.SaveChannel(chReq)
    if err != nil {
        return fmt.Errorf("创建应用通道失败:%v", err)
    }

    time.Sleep(time.Second * 5)

    //创建一个管理资源的客户端对象
    f.Admin, err =
f.SDK.NewClient(fabsdk.WithUser(f.OrgAdmin)).ResourceMgmt()
```

```
    if err != nil {
        return fmt.Errorf("创建资源管理对象失败:%v", err)
    }
    //将peer 节点加入到应用通道中
    err = f.Admin.JoinChannel(f.ChannelID)
    if err != nil {
        return fmt.Errorf("peer加入节点失败:%v", err)
    }

    f.Initialized = true
    fmt.Println("SDK实例化成功")

    return nil


}
```

该测试代码可以从[blockchain/setup.go](blockchain/setup.go)获取

在这个阶段

- 初始化一个客户端，它将与 peer，CA 和 orderer进行通信。
- 创建了一个新通道, 并将Peer节点加入到此通道中

3. 编写测试代码

为了确保客户端能够初始化所有组件，将在启动网络的情况下进行简单的测试。 为了做到这一点，我们需要构建主程序代码进行功能调用

```
$ vim main.go
```

```
package main


import (
    "FabricDev/ticket/blockchain"
    "os"
    "fmt"
)

func main() {

    fsetup := blockchain.FabricSetup{
        ConfigFile:    "config.yaml",
        ChannelID:     "mychannel",
        ChannelConfig: os.Getenv("GOPATH") +
"src/github.com/ticket/fixtures/artifacts/channel.tx",
        OrgAdmin:      "Admin",
```

```
        OrgName:       "Org1",
    }

    err := fsetup.Initialize()

    if err != nil {
        fmt.Errorf("Fabric SDK初始化失败:%v", err)
        fmt.Println(err.Error())
    }

}
```

代码模板:main.go

4. 打包依赖关系

在开始编译之前，最后一件事是使用一个vendor目录来包含我们所有的依赖关系。 在我们的GOPATH中，我们有Fabric SDK Go和其他项目。 当尝试编译应用程序时，Golang会在GOPATH中搜索依赖项，但首先会检查项目中是否存在vendor文件夹。 如果依赖性得到满足，那么Golang就不会去看GOPATH或GOROOT。 这在使用几个不同版本的依赖关系时非常有用（可能会发生一些冲突，比如在例子中有多个BCCSP定义，通过使用像`dep`这样的工具来处理这些依赖关系在`vendor`目录中。

```
$ vim Gopkg.toml
```

配置文件内容

```
[[constraint]]
  name = "github.com/hyperledger/fabric"
  revision = "014d6befcf67f3787bb3d67ff34e1a98dc6aec5f"

[[constraint]]
  name = "github.com/hyperledger/fabric-sdk-go"
  revision = "614551a752802488988921a730b172dada7def1d"
```

这是`dep`一个限制，以便在 vendor 中指定希望SDK转到特定版本。

保存该文件，然后执行此命令将vendor目录与项目的依赖关系同步：

```
$ dep ensure
```

```
$ls vendor -l //查看vendor目录内容
```

```
drwxrwxr-x 13 bruce bruce 4096 Jul 20 14:06 github.com
drwxrwxr-x  3 bruce bruce 4096 Jul 20 14:06 golang.org
drwxrwxr-x  4 bruce bruce 4096 Jul 20 14:06 google.golang.org
drwxrwxr-x  3 bruce bruce 4096 Jul 20 14:06 gopkg.in
```

5.构建代码

```
$go build //构建代码生成ticket可执行文件
$ls ticket -l
-rwxrwxr-x 1 bruce bruce 19784971 Jul 20 14:06 ticket
```

6.执行命令

```
$ ./ticket
```

```
[fabric_sdk_go] 2018/07/21 09:19:23 UTC - config.initConfig -> INFO config
fabric_sdk_go logging level is set to: INFO
SDK实例化成功
```

7.清理环境

Fabric SDK生成一些文件，如证书，二进制文件和临时文件。 关闭网络不会完全清理环境，当需要重新启动时，这些文件将被重复使用以避免构建过程。 对于开发，可以快速测试，但对于真正的测试，需要清理所有内容并从头开始。

如何清理环境

- 关闭你的网络： `cd $GOPATH/src/github.com/kongyixueyuan.com/bill/fixtures && docker-compose down`
- 删除证书存储（在配置文件中， `client.credentialStore` 中定义）： `rm -rf /tmp/bill-*`
- 删除一些不是由 `docker-compose` 命令生成的docker容器和docker镜像：

  ```
  docker rm -f -v `docker ps -a --no-trunc | grep "bill" | cut -d ' ' -f
  1` 2>/dev/null
  和
  docker rmi `docker images --no-trunc | grep "bill" | cut -d ' ' -f 1`
  2>/dev/null
  ```

如何更有效率?

可以在一个步骤中自动完成所有这些任务。 构建和启动过程也可以自动化。 为此，将创建一个Makefile。 首先，确保 `make` 工具：

`make --version`

> 如果没有安装 `make` （Ubuntu）：
>
> ```
> sudo apt install make
> ```

然后使用以下内容在项目的根目录下创建一个名为`Makefile`的文件：

```
$ cd $GOPATH/src/github.com/kongyixueyuan.com/bill
$ vim Makefile
```

```
.PHONY: all dev clean build env-up env-down run

all: clean build env-up run

dev: build run

##### BUILD
build:
    @echo "Build ..."
    @dep ensure
    @go build
    @echo "Build done"

##### ENV
env-up:
    @echo "Start environment ..."
    @cd fixtures && docker-compose up --force-recreate -d
    @echo "Sleep 15 seconds in order to let the environment setup
correctly"
    @sleep 15
    @echo "Environment up"

env-down:
    @echo "Stop environment ..."
    @cd fixtures && docker-compose down
    @echo "Environment down"

##### RUN
run:
    @echo "Start app ..."
    @./ticket

##### CLEAN
clean: env-down
    @echo "Clean up ..."
    @rm -rf /tmp/bill-* bill
    @docker rm -f -v `docker ps -a --no-trunc | grep "ticket" | cut -d ' '
-f 1` 2>/dev/null || true
    @docker rmi `docker images --no-trunc | grep "ticket" | cut -d ' ' -f
1` 2>/dev/null || true
    @echo "Clean up done"
```

现在完成任务：

1. 整个环境将被清理干净
2. go程序将被编译
3. 启动区块链网络
4. 启动程序

要使用它，请进入项目的根目录并使用`make`命令：

- 任务`all`： `make`或`make all`
- 任务`clean`：清理一切并释放网络（`make clean`）
- 任务`build`：只需构建应用程序（`make build`）
- 任务`env-up`：只需建立网络（`make env-up`）