使用Fabric SDK Go

a. 配置

应用程序需要很多参数，特别是Fabric组件的通信地址。 现在把所有内容放入新的配置文件中(Fabric SDK Go配置和自定义参数)。

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

config.yaml 文件内容如下:

```
name: "bill-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.co
m/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/kongyixueyuan.com/bill/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/kongyixueyuan.com/bill/fixtures/crypto-
config/peerOrganizations/org1.example.com/tlsca/tlsca.org1.example.c
om-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/kongyixueyuan.com/bill/fixtures/crypto-
config/peerOrganizations/org1.example.com/tlsca/tlsca.org1.example.c
om-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
```

b. 初始化

添加一个名为`blockchain`的新文件夹，其中将包含与网络通信的整个界面。 我们将只在该文件夹中看到Fabric SDK Go。

```
$ mkdir blockchain
```

现在，添加一个名为`startInit.go`的go文件：

```
$ vim blockchain/startInit.go
```

```go
package blockchain

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

// FabricSetup implementation
type FabricSetup struct {
    ConfigFile      string
    ChannelID       string
    initialized     bool
    ChannelConfig   string
    OrgAdmin        string
    OrgName         string
    admin           resmgmtclient.ResourceMgmtClient
    sdk             *fabsdk.FabricSDK
}

// Initialize reads the configuration file and setsup the client,
chain and event hub
func (setup *FabricSetup) Initialize() error {

    fmt.Println("开始初始化......")

    if setup.initialized {
        return fmt.Errorf("sdk已初始化完毕\n")
    }

    // 使用指定的配置文件创建SDK
    sdk, err := fabsdk.New(config.FromFile(setup.ConfigFile))
    if err != nil {
        return fmt.Errorf("SDK创建失败: %v\n", err)
    }
```

```
    setup.sdk = sdk

    // 根据指定的具有特权的用户创建用于管理通道的客户端API
    chMgmtClient, err :=
setup.sdk.NewClient(fabsdk.WithUser(setup.OrgAdmin),
fabsdk.WithOrg(setup.OrgName)).ChannelMgmt()
    if err != nil {
        return fmt.Errorf("SDK添加管理用户失败: %v\n", err)
    }

    // 获取客户端的会话用户(目前只有session方法能够获取)
    session, err :=
setup.sdk.NewClient(fabsdk.WithUser(setup.OrgAdmin),
fabsdk.WithOrg(setup.OrgName)).Session()
    if err != nil {
        return fmt.Errorf("获取会话用户失败 %s, %s: %s\n",
setup.OrgName, setup.OrgAdmin, err)
    }
    orgAdminUser := session

    // 指定用于创建或更新通道的参数
    req := chmgmtclient.SaveChannelRequest{ChannelID:
setup.ChannelID, ChannelConfig: setup.ChannelConfig,
SigningIdentity: orgAdminUser}
    // 使用指定的参数创建或更新通道
    err = chMgmtClient.SaveChannel(req)
    if err != nil {
        return fmt.Errorf("创建通道失败: %v\n", err)
    }

    time.Sleep(time.Second * 5)

    // 创建一个用于管理系统资源的客户端API。
    setup.admin, err =
setup.sdk.NewClient(fabsdk.WithUser(setup.OrgAdmin)).ResourceMgmt()
    if err != nil {
        return fmt.Errorf("创建资源管理客户端失败: %v\n", err)
    }
```

```
    // 将peer加入通道
    if err = setup.admin.JoinChannel(setup.ChannelID); err != nil {
        return fmt.Errorf("peer加入通道失败: %v\n", err)
    }

    fmt.Println("初始化成功")
    setup.initialized = true
    return nil
}
```

在这个阶段，我们只初始化一个客户端，它将与 peer，CA 和 orderer进行通信。还创建了一个新通道, 并将Peer节点加入到此通道中

C. 测试

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go代码。 由于没有任何主文件，所以必须添加一个：

```
$ vim main.go
```

```
package main

import (
    "fmt"
    "github.com/kongyixueyuan.com/bill/blockchain"
    "os"
)

func main() {
    // 定义SDK属性
    fSetup := blockchain.FabricSetup{
        OrgAdmin:        "Admin",
        OrgName:         "Org1",
        ConfigFile:      "config.yaml",
```

```
        // 通道相关
        ChannelID:       "mychannel",
        ChannelConfig:   os.Getenv("GOPATH") +
"/src/github.com/kongyixueyuan.com/bill/fixtures/artifacts/channel.t
x",
    }

    // 初始化SDK
    err := fSetup.Initialize()
    if err != nil {
        fmt.Printf("无法初始化Fabric SDK: %v\n", err)
    }
}
```

指定了环境的GOPATH，用来编译链码

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。

当您安装SDK依赖关系时，DEP会自动安装。 如果不是这种情况，您可以阅读以下说明安装它： dep安装

创建一个名为`Gopkg.toml`的文件并将其复制到里面：

```
$ vim Gopkg.toml
```

```
[[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（这可能需要一段时间）：

```
$ dep ensure
```

提醒：`dep ensure`由于时间久，执行一次后即可，在后面的Makefile中可注释`@dep ensure`命令。

编译应用程序

```
$ go build
```

一段时间后，一个名为 bill的新二进制文件将出现在项目的根目录下。 尝试像这样启动二进制文件：

```
$ ./bill
```

./bill命令执行失败

> 此时，它将无法工作，因为没有可以与SDK进行通信的网络。 须先启动网络，然后再次启动应用程序：

```
$ cd fixtures
$ docker-compose up -d
$ cd ..
$ ./bill
```

```
kevin@kevin-hf:~/go/src/github.com/kongyixueyuan.com/bill$ go build
kevin@kevin-hf:~/go/src/github.com/kongyixueyuan.com/bill$ ./bill
开始初始化SDK......
 [fabric_sdk_go] 2018/06/07 10:21:38 UTC - config.initConfig -> INFO config fabric_sdk_go log
ging level is set to: INFO
SDK初始化成功
kevin@kevin-hf:~/go/src/github.com/kongyixueyuan.com/bill$
```

> **注意** ：需要看到“初始化成功”。 如果没有看到则说明出现问题。

现在只用本地网络初始化SDK。 在下一步中，将与链码进行交互。

d. 清理和Makefile

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 ..."
    @./bill

##### CLEAN
clean: env-down
    @echo "Clean up ..."
    @rm -rf /tmp/bill-* bill
    @docker rm -f -v `docker ps -a --no-trunc | grep "bill" | cut -d
' ' -f 1` 2>/dev/null || true
    @docker rmi `docker images --no-trunc | grep "bill" | 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. 之后将部署网络
4. 最后该应用程序将启动并运行。

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` ）
- ...


区块链部落
专注于区块链技术
识别图中二维码关注我们