Chaincode services 启动调用过程：

我们知道在fabric 2.0以前，chaincode都是作为client端进行维护，执行步骤是chaincode 启动完成后到peer端进行注册，以此建立chaincode 与peer 节点的连接。自fabric2.0以后开始支持chaincode 作为服务器端，我们可以在外部完成chaicode的启动，然后当我们invoke chaincode时,peer 端会去主动连接chaincode.下面我们就介绍chaincode 外部启动步骤以及我们需要规避的坑。

1：前置条件

我们本次测试使用的fabric网络的启动环境主要使用fabric-sample 提供的fabric-testnetwork.提供的fabric 测试网络的启动脚本(安装与启动步骤请参考fabric2.0 安装步骤文档)

2：chaincode 服务器代码开发。Chaincode 代码例子

**package** main  
  
**import** (  
 **"fmt"  
 "github.com/hyperledger/fabric-chaincode-go/shim"** pb **"github.com/hyperledger/fabric-protos-go/peer"  
 "flag"**)  
*// SimpleChaincode example simple Chaincode implementation***type** SimpleChaincode **struct** {  
}  
**func** (s \*SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {  
 *// init code* fmt.Println(**"some one invoke init operation!"**)  
 **return** shim.Success([]byte(**"your init success"**))  
}  
**func** (s \*SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {  
 *// invoke code* fmt.Println(**"some one invoke invoke operation!"**)  
 **return** shim.Success([]byte(**"your invoke success"**))  
}  
*//NOTE - parameters such as ccid and endpoint information are hard coded here for illustration. This can be passed in in a variety of standard ways***func** main() {  
 flag.Parse()  
 *//The ccid is assigned to the chaincode on install (using the “peer lifecycle chaincode install <package>” command) for instance* Ccid := **"mycc:fcbf8724572d42e859a7dd9a7cd8e2efb84058292017df6e3d89178b64e6c831"** server := &shim.ChaincodeServer{  
 CCID: Ccid,  
 Address: Address,  
 CC: new(SimpleChaincode),  
 TLSProps: shim.TLSProperties{  
 Disabled: Disabled,  
 },  
 }  
 err := server.Start()  
 **if** err != nil {  
 fmt.Printf(**"Error starting Simple chaincode: %s"**, err)  
 }  
}  
**var** (  
 Disabled bool  
 Ccid string  
 Address string  
)  
**func** init() {  
 ResetFlags()  
}  
**func** ResetFlags(){ *//启动命令的时候-d 必须放在最后* flag.BoolVar(&Disabled, **"d"**, ***true***, **"cc tls disabled"**)  
 flag.StringVar(&Ccid, **"c"**, **""**, **"ccid"**)  
 flag.StringVar(&Address, **"a"**, **":9088"**, **"chaincode servers address "**)  
}

大家从例子中可以发现chaincode 作为服务端的写法与以往唯一的变化便是

server := &shim.ChaincodeServer{  
 CCID: Ccid, //ccid 便是我们install chaincode package 时的PACKAGE\_ID  
 Address: Address, //当前chaincode server 监听地址（这里需要特别说明一下如果是在阿里云的话建议不要添加localhost 前缀,因为我们在阿里云申请的服务器是一个docker，如果添加ip 的话它监听的是本地回环地址，docker 外部的容器是没有办法访问的。当我们使用”:9088”侦听端点时，这将使您的服务器侦听其容器的所有IP地址。 ）  
 CC: new(SimpleChaincode),  
 TLSProps: shim.TLSProperties{  
 Disabled: Disabled, //是否不启用tls 通信，如果启用我们还需要设置tls cert path  
 },  
 }  
 err := server.Start() //开始启动chaincode server

3：package chaincode

因为我们使用chaincode 作为服务端，所以我们在打包时，只需要打包connection.json(存储了chaincode server的相关信息)和metadata.json (标识chaincode 的启动类型)以及一个空的metadata 目录

connection.json // 格式

{

"address": "your.chaincode.host.com:9999",

"dial\_timeout": "10s",

"tls\_required": true,

"client\_auth\_required": true,

"client\_key": "-----BEGIN EC PRIVATE KEY----- ... -----END EC PRIVATE KEY--",

"client\_cert": "-----BEGIN CERTIFICATE----- ... -----END CERTIFICATE-----",

"root\_cert": "-----BEGIN CERTIFICATE---- ... -----END CERTIFICATE-----"

}

1：address **：**链码服务器地址

2：dial\_timeout ：与链码服务器链接的超时时间

3：tls\_required ：是否需要“client\_auth\_required”if true 则启用

4：client\_auth\_required ：如果为true，则需要“ client\_key”和“ client\_cert”。默认为false。如果tls\_required为false，则将其忽略。

5：client\_key ：客户端私钥的PEM编码的字符串。

6：client\_cert：客户端证书的PEM编码的字符串。

7：root\_cert：服务器（对等）根证书的PEM编码的字符串。

Metadata.json //格式

{"path":"",//chaincode 打包相对路径，我们当前可以置为空

"type":"external",//chaincode 启动类型，

"label":"mycc" //chaincode 描述，不要太长

}

4：修改peer配置文件core.yaml.

externalBuilders:

- path: /etc/hyperledger/fabric/golang

name: external-golang

propagateEnvironment:

- ENVVAR\_NAME\_TO\_PROPAGATE\_FROM\_PEER

- GOPROXY

5: 配置peer 外部启动脚本

外部启动脚本  
└── bin

├── build

├── detect

└── release

外部构建器共需要三个脚本文件

第一个脚本bin/detect

#!/bin/sh

# 注意因为我们是使用peer docker 执行当前脚本命令这里必须是/bin/sh 而不是/bin/bash

set -euo pipefail

METADIR**=**$2

*#check if the "type" field is set to "external"*

**if** **[** "**$(**jq -r .type "$METADIR/metadata.json"**)**" **==** "external" **]**; **then**

exit 0

**fi**

exit 1

从脚本内容可以发现，bin/detect脚本主要检查 metadata.json 的type 类型是否是external。

第二个脚本bin/build

#!/bin/sh

set -euo pipefail

SOURCE**=**$1

OUTPUT**=**$3

*#external chaincodes expect connection.json file in the chaincode package*

**if** **[** ! -f "$SOURCE/connection.json" **]**; **then**

>&2 echo "$SOURCE/connection.json not found"

exit 1

**fi**

*#simply copy the endpoint information to specified output location*

cp $SOURCE/connection.json $OUTPUT/connection.json

**if** **[** -d "$SOURCE/metadata" **]**; **then**

cp -a $SOURCE/metadata $OUTPUT/metadata

**fi**

exit 0

从脚本内容可以发现，bin/build脚本主要检查connection.json 文件是否存在，并将文件copy 到对应位置。

第三个脚本bin/release

#!/bin/sh

set -euo pipefail

BLD**=**"$1"

RELEASE**=**"$2"

if [ -d "$BLD/metadata" ]; then

if [ "`ls -A $BLD`"= ""];then

echo "$BLD/metadata not exist file"

else

cp -a "$BLD/metadata/"\* "$RELEASE/"

fi

else

echo "$BLD/metadata not directory"

exit 1

fi

#external chaincodes expect artifacts to be placed under "$RELEASE"/chaincode/server

if [ -f $BLD/connection.json ]; then

mkdir -p "$RELEASE"/chaincode/server

cp $BLD/connection.json "$RELEASE"/chaincode/server

#if tls\_required is true, copy TLS files (using above example, the fully qualified path for these fils would be "$RELEASE"/chaincode/server/tls)

exit 0

fi

exit 1

从脚本内容可以发现，bin/release首先检查metadata 目录下是否存在文件，如果存在全部拷贝到“$RELEASE”目录下，最后将connection.json 文件拷贝到新建的"$RELEASE"/chaincode/server目录下。Peer 端连接chaincode 服务器时我们会从"$RELEASE"/chaincode/server目录下读取connection.json 文件。

以上的三个脚本内容，用户可以根据自己的需求进行定制。

6：docker 配置

前面我们说过我们是使用fabric-sample 的测试网络脚本。所以这里我们需要修改fabric-samples/test-network/docker/docker-compose-test-net.yaml 将peer0.org1.example.com 和 peer0.org2.example.com 的volumes 参数替换。

peer0.org1.example.com

volumes:

- ../golang/core.yaml:/etc/hyperledger/fabric/core.yaml //修改后的core.yaml 地址

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

- ../golang:/etc/hyperledger/fabric/golang //脚本文件存放目录

- ../organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp:/etc/hyperledger/fabric/msp

- ../organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls:/etc/hyperledger/fabric/tls

- peer0.org1.example.com:/var/hyperledger/production

peer0.org2.example.com:

volumes:

- ../golang/core.yaml:/etc/hyperledger/fabric/core.yaml //修改后的core.yaml 地址

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

- ../golang:/etc/hyperledger/fabric/golang //脚本文件存放目录

- ../organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/msp:/etc/hyperledger/fabric/msp

- ../organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls:/etc/hyperledger/fabric/tls

- peer0.org2.example.com:/var/hyperledger/production

7：加入jq

因为peer 调用外部启动脚本时需要使用jq,所以在执行install chaincode 操作前我们需要将jq 安装到peer docker 中。当我们完成这一切操作以后就可以使用fabric-sample提供的启动脚本进行启动。

8：tls 启动

在测试环境中我们可以不使用tls,进行服务端与客户端进行通信，但如果在正式环境中我们就必须使用tls加密通信了.

Chaincode 与peer 使用tls 加密通信。我们可以直接使用cryptogen 来生成我们要使用的server tls 和 client tls.server tls 存放在我们生成的peer 路径下的tls 目录下.client tls 存放在我们生成的users 路径下。

如我的client tls是在

$gopath/github.com/hyperledger/fabric/scripts/fabric-samples/test-network/organizations/peerOrganizations/org1.example.com/users/

server tls是在

$gopath/github.com/hyperledger/fabric/scripts/fabric-samples/test-network/organizations/peerOrganizations/org1.example.com/peers/

Note:这里我们需要特别注意当我们生成证书时，SANS 的值（记录我们服务端域名），否则client 端将无法连接server 端。如下便是我生成证书信息的配置文件。

**Template:  
 Count:** 1  
 **SANS:** - localhost  
 *# Start: 5  
 # Hostname: {{.Prefix}}{{.Index}} # default  
# ---------------------------------------------------------------------------  
# "Users"  
# ---------------------------------------------------------------------------  
# Count: The number of user accounts \_in addition\_ to Admin  
# ---------------------------------------------------------------------------***Users:  
 Count:** 1