实验九 End2End案例详解

【实验介绍】

在上一个实验中，我们完成了Fabric基本环境的安装与配置，并成功运行了官方提供的End2End网络案例，对Fabric的架构特点有了初步的认识。

本次实验还是使用End2End案例，因为End2End案例几乎涉及了 Fabric 网络中超过80%的情况，有种案例即生产的即视感，其中所涉及的节点证书文件生成、节点用户管理和节点频道管理，后续还有频道及智能合约相关操作，该脚本可以说是入门学习的必备。

【实验要求】

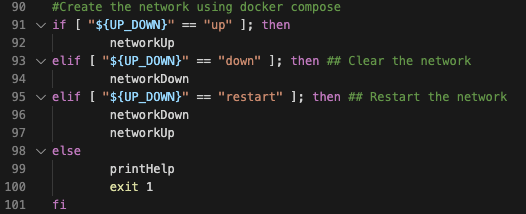
1. 理解End2End案例的启动逻辑
2. 手动调用案例中的合约
3. 理解Fabric中智能合约的逻辑

【实验过程】

1. **理解End2End案例的启动逻辑**

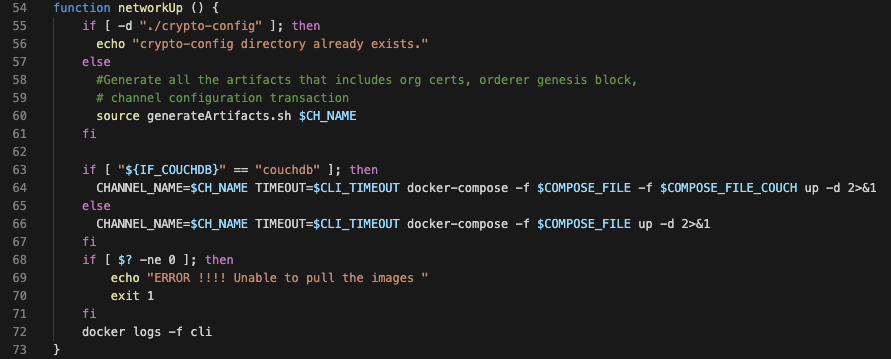
这一小节的内容主要是介绍End2End案例的启动逻辑，作为一个了解，不影响后续的操作，允许直接进入下一小节的内容。

首先查看案例的入口文件，network\_setup.sh，可以看到：



程序先检查参数，然后根据参数调用不同函数。

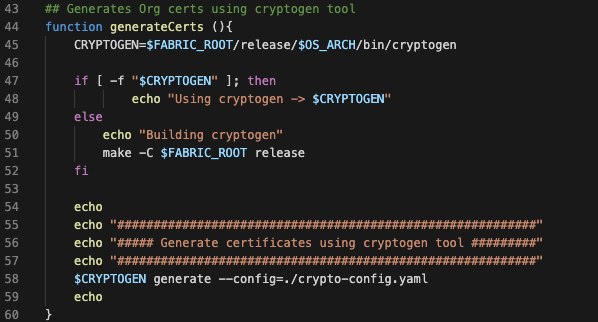
networkUp命令，执行 generateArtifacts.sh文件:



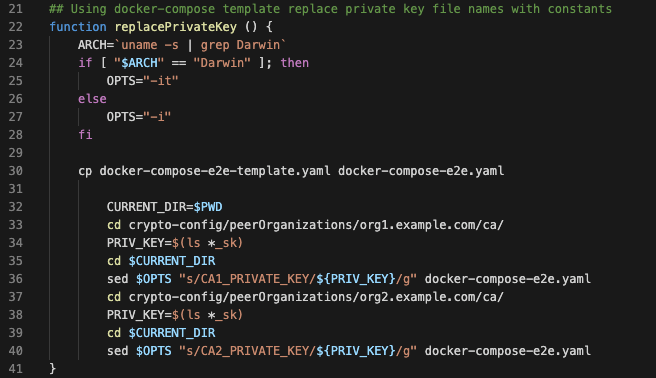
程序运行generateArtifacts.sh文件，运行完毕后，再调用compose-docker 命令。

我们分析一下generateArtifacts文件，该文件主要执行了三个步骤：

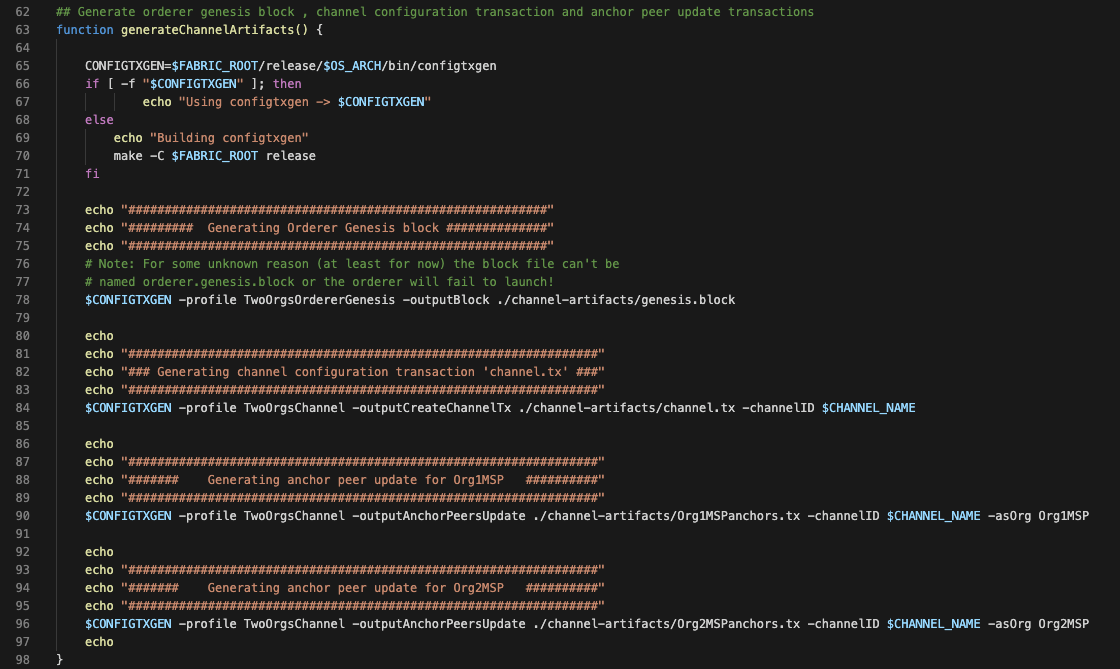
generateCerts



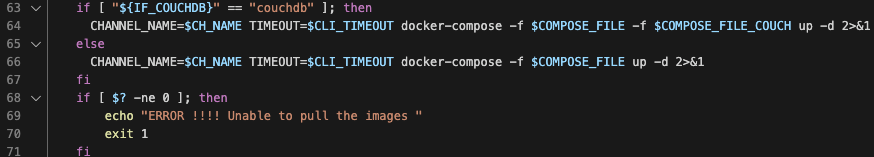
replacePrivateKey



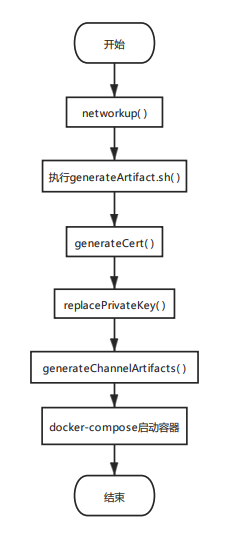
generateChannelArtifacts



最后，generateArtifacts.sh 文件执行完毕。执行返回到network\_setup.sh文件执行：

****

整个过程执行完毕，其运行过程大致如下：

****

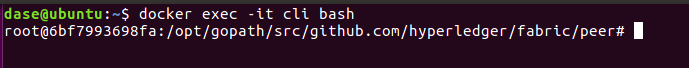
1. **手动调用案例中的合约**

这里，我们尝试手动调用该案例中部署的智能合约。

成功运行End2End案例后，重新打开一个窗口，输入如下命令：

|  |
| --- |
| docker exec -it cli bash |

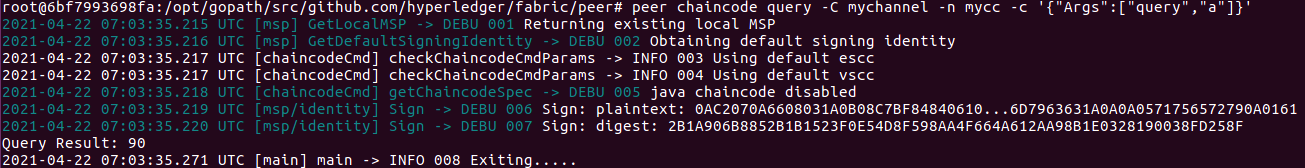
此时会进入cli客户端，如下所示：



然后输入如下命令，查询一下账户a的资产：

|  |
| --- |
| peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}' |

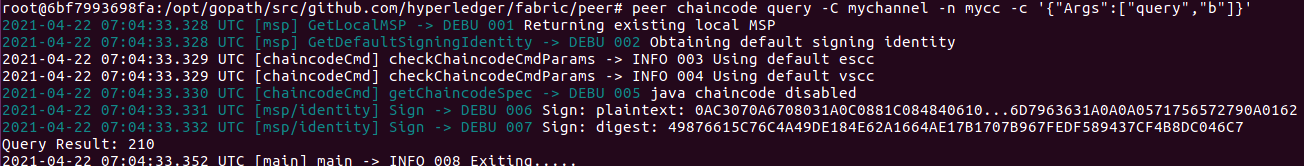
输出如下，可以看到账户a的资产为90:

****

同样的，也查询一下账户b的资产：

|  |
| --- |
| peer chaincode query -C mychannel -n mycc -c '{"Args":["query","b"]}' |

输出如下，可以看到账户b的资产为210:

****

然后我们来做一下转账操作，账户a给账户b转账50:

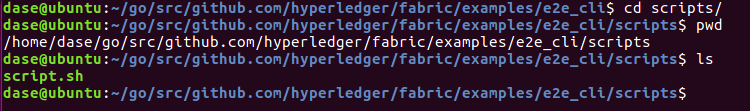
|  |
| --- |
| peer chaincode invoke -o orderer.example.com:7050 --tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n mycc -c '{"Args":["invoke","a","b","50"]}' |

看到执行成功的提示后，你可以再分别查询一下两个账户的余额，这里不再赘述。

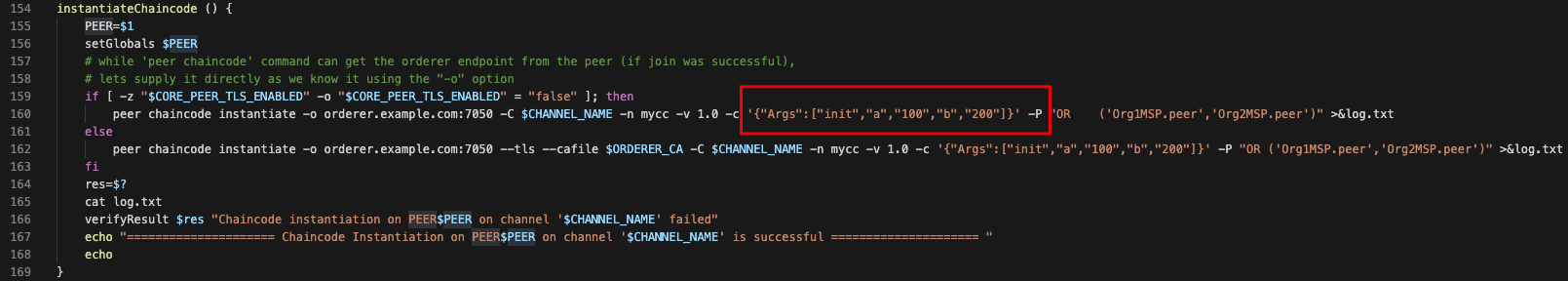
1. **理解Fabric中智能合约的逻辑**

通过上述手动调用智能合约的过程，我们可以得知，End2End案例不仅仅搭建了一个fabric网络，还部署了一个智能合约，并进行了合约实例化操作以及调用操作（其实从一开始的余额查询就可以看出来，账户a余额90，账户b余额210，显然一开始在运行End2End案例过程中，存在转账操作）。

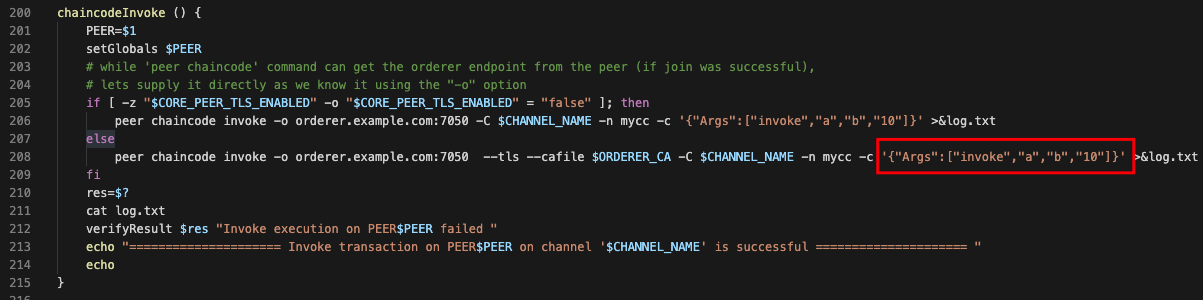
进入e2e案例的scripts文件夹中：

****

可以看到其中的脚本文件，该文件中的内容验证了上述猜测。



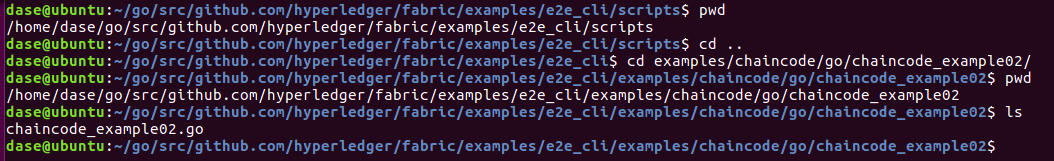
可以看到，instantiateChaincode方法，该方法对智能合约进行了初始化操作，从其中的初始化参数可以看到，账户a资产100，账户b资产200。



后续的chaincodeInvoke方法，账户a给账户b转账10。

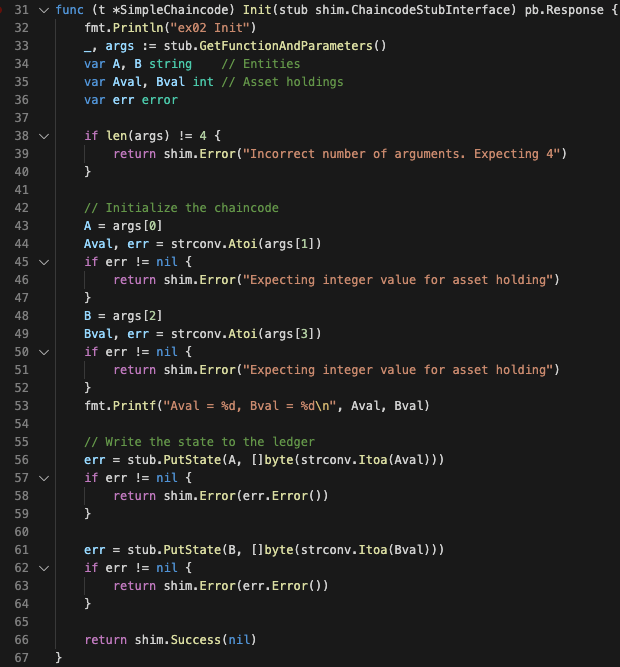
那么，这个脚本所操作的智能合约在哪？调用的时候到底发生了啥？

刚才我们在scripts文件夹下，通过如下操作，找到相应智能合约所在的位置：



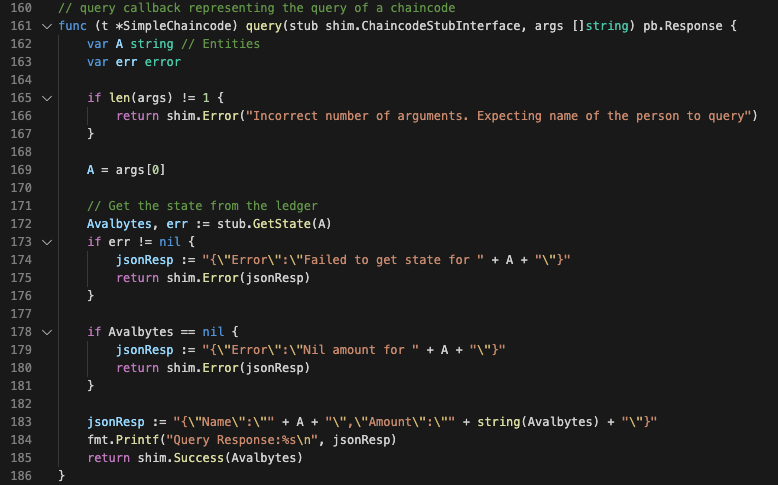
可以看到，e2e案例使用的智能合约是用Go写的（另外也支持Java），这里，我们大致理解一下这个合约即可，后续的合约会使用Java进行编写。

在script.sh脚本中，通过具体的实例化智能合约命令，可以观察到其中的-c 参数指定了智能合约初始化时传入的参数内容，需要对比智能合约中的初始化方法来判断在该操作中发生了什么。具体的合约方法如下：



阅读该方法，可以看出该方法会接收四个参数，而在-c '{"Args":["init","a","100","b","200"]}'命令中传入了五个参数，第一个参数为方法名，后四个参数为该方法将要接收的参数内容。所以，在 init 方法中接收的四个参数分别是“a”、“100”、“b”和“200”。通过代码不难观察出初始化方法的目的是创建一个 key 为 a 的账户并给该账户一个值为 100 的资产，同创建一个 key 为 b 的账户并也给该账户一个值为 200 的资产。这里就和上述实验内容串起来了。

接下来需要验证该智能合约中的 invoke 和 query 方法。先看 query 方法，如下所示：



该方法需要接收且仅能接受一个参数，所接收的参数为智能合约中所创建账户的 key 值返回内容为该账户下的资产，因此我们通过执行如下命令对a账户的资产进行查询：

|  |
| --- |
| peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}' |

接下来进入 invoke 方法，具体代码示例如下：



仔细阅读该方法，可以看到该方法需要传入三个参数，第一个和第二个参数分别是不同的账户名，第三个参数是资产值。方法的核心是第一个账户将自己资产中第三个参数值的资产转移到第二个账户名下。如刚才我们执行的转账命令：

|  |
| --- |
| peer chaincode invoke -o orderer.example.com:7050 --tls true --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 mychannel -n mycc -c '{"Args":["invoke","a","b","50"]}' |

至此，本次实验的内容就结束了。主要是基于e2e案例，带大家理解他的运行逻辑，然后手动调用该案例中的智能合约（ChainCode），最后学习了一下该案例中的示例合约。后面的实验，我们将会使用Java来实现智能合约，然后进行部署和调用。