**《课程综合实践Ⅱ》实验报告七**

**基于Hyperledger Fabric的智能合约和SDK初识**

**姓名\_\_\_ 沈云昊 专业 电子信息工程**

**学号 3220104126 联系方式 2766636730@qq.com**

## 一、实验目的

1. 在虚拟机中构建test-network区块链网络。

2. 在区块链网络中成功部署fabcar智能合约，掌握基于单机多节点的部署方法以及链码的初始化和调用方法。

3. 学会用Gin框架包装区块链SDK，并用SDK的方式调用智能合约。

## 二、实验内容及要求

**准备工作：为了方便后续的学习和实践，建议同学们自带电脑，在自己电脑上进行实验内容，并完成实验报告。**

**实验步骤（仔细阅读，按照步骤完成实验）**

**（本次实验全程在ubuntu上进行操作）**

### 运行test-network区块链网络

和上次实验的最后一步类似，我们首先建立一个基于三个节点、两个组织的区块链网络，参考以下命令进行操作：

## 进入到fabric为我们提供的test-network源码目录

cd ~/hyperledger/fabric/scripts/fabric-samples/test-network

## 确保之前可能构建的test-network关闭

./network.sh down

## 开始构建test-network区块链网络

./network.sh up

## 查看构建的节点docker容器，构建成功会输出4个容器的信息

docker ps -a

### Fabcar智能合约部署及调用（请截图保存在实验记录中）

智能合约又称为链码，是一段用go语言编写的程序，在Hyperledger Fabric中和运行着的区块链节点类似，也是以docker容器为运行环境的独立进程，旨在允许没有第三方的情况下进行可信交易（增删查改等），并保证交易可追踪不可逆转的特性，下面以fabric为我们提供的Fabcar智能合约源码为例，对智能合约的实现进行初步了解。

**2.1 Fabcar链码初识**

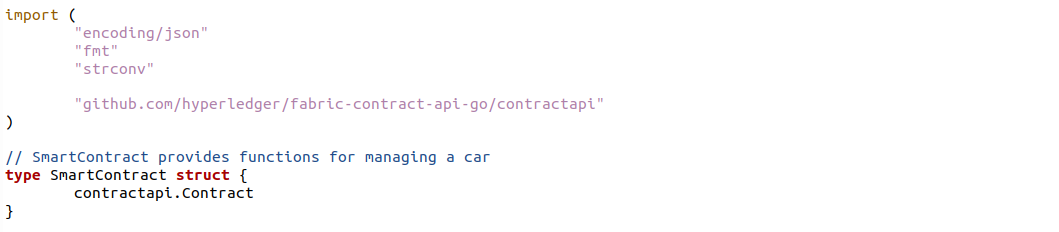
* 输入命令进入保存该合约源码的目录，并打开查看，输入以下命令：

## 进入到fabric为我们提供的fabcar源码目录

cd ~/hyperledger/fabric/scripts/fabric-samples/chaincode/fabcar/go

gedit fabcar.go

* 可以看到智能合约的源码并不长，下面我们对其进行简单分析。



首先看到文件的前几行，import是go语言提供的导入依赖包的方式，这里智能合约的编写需要导入fabric为我们提供的contractapi包，在该依赖包中提供了几乎全部编写链码时需要用到的封装好的函数，之后，源码定义了一个名为SmartContract的结构体，有了这个定义之后就可以在函数中利用智能合约对账本进行操作了

随后我们直接看源码的主函数即程序入口，



可以看到源码调用了contractapi提供的NewChaincode方法新建了一个智能合约SmartContract实例，随后用Start方法启动该链码。

注意到中间还定义了许多函数，这些函数名就是供给区块链中节点调用的方法名字，其代码也比较简单易懂，同学们可参考这篇博客进行学习(<https://blog.csdn.net/wyt813990159/article/details/120263900>)，至于go语法部分请同学们自行前往菜鸟教程等网站学习。

**2.2 Fabcar链码部署**

有了链码之后，我们还需要将其部署到我们构建的test-network中才能使其发挥真正的功能，部署的过程需要用到大量的peer命令，分为打包-安装-批准-提交四步，首先我们需要先创建一个智能合约的运行环境—channel.

* 保证当前位于test-network目录下，输入命令，创建一个channel。

## 进入到fabric为我们提供的test-network源码目录

cd ~/hyperledger/fabric/scripts/fabric-samples/test-network

## 创建一个channel通道，默认名为mychannel

./network.sh createChannel

* 为当前运行的终端A，添加临时环境变量，便于我们之后运行peer相关命令。

## 将peer命令所在的bin文件夹添加到环境变量中

export PATH=${PWD}/../bin:$PATH

## 设置FABRIC\_CFG\_PATH路径，指向core.yaml文件

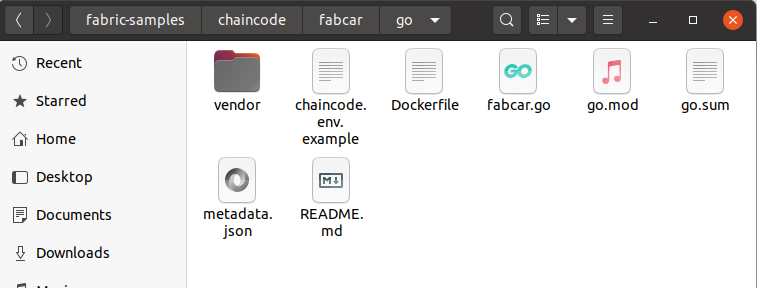
export FABRIC\_CFG\_PATH=$PWD/../config/

* 新建一个终端B，进入到存放fabcar源码的根目录下，为其安装依赖，输入命令如下

cd ~/hyperledger/fabric/scripts/fabric-samples/chaincode/fabcar/go

go mod vendor

你的fabcar源码目录应该如下所示，



至此，我们完成了所有准备工作，正式开始链码部署。

* **链码打包**（此后的几步务必在已经添加好临时环境变量的终端A中操作）

peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/go/ --lang golang --label fabcar\_1

命令说明：

**peer lifecycle chaincode package(链码打包命令)**

（1）--path：指向链码存放的目录，并将其打包成fabcar.tar.gz

（2）--lang：表明编写链码所用的语言

（3）--label：自定义标签，用于日后链码需要升级的情况

* **链码安装**

这一步是将打包完成的链码安装在节点上，由于test-network中存在org1、org2两个组织，因此我们要为分属于不同组织的两个节点分别安装链码，首先设置临时环境变量，表明是以org1的身份进行链码安装操作，命令如下：

export CORE\_PEER\_TLS\_ENABLED=true

export CORE\_PEER\_LOCALMSPID="Org1MSP"

export CORE\_PEER\_TLS\_ROOTCERT\_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt

export CORE\_PEER\_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp

export CORE\_PEER\_ADDRESS=localhost:7051

安装链码：

peer lifecycle chaincode install fabcar.tar.gz

切换身份至org2，同样进行安装链码操作：

export CORE\_PEER\_LOCALMSPID="Org2MSP"

export CORE\_PEER\_TLS\_ROOTCERT\_FILE=${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt

export CORE\_PEER\_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp

export CORE\_PEER\_ADDRESS=localhost:9051

peer lifecycle chaincode install fabcar.tar.gz

* **链码批准**

在两个组织分别完成链码安装后，下一步两个组织需要分别批准该链码的定义，以便于最后将该链码发布到channel上正式运行，首先我们需要先查询链码包的ID，输入命令如下：

peer lifecycle chaincode queryinstalled

查到的Package ID应该是类似于label1:ID的格式，如下图所示



将其复制下来，添加进环境变量中，命令如下：

export CC\_PACKAGE\_ID=刚刚复制的Package ID

现在终端登陆的身份是org2，先为org2批准链码定义，输入命令如下：

peer lifecycle chaincode approveformyorg -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name fabcar --version 1.0 --package-id $CC\_PACKAGE\_ID --sequence 1 --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --waitForEvent

命令说明：

**peer lifecycle chaincode approveformyorg (链码批准命令)**

（1）-o：排序节点地址

（2）--ordererTLSHostnameOverride：排序节点主机名

（3）--channelID：通道ID

（4）--name：已经在节点上安装的链码名称

（5）--version：自定义链码版本，用于日后链码需要升级的情况

（6）--package-id：链码安装包标识ID

（7）--sequence：通道链码定义序列号，默认为1

（8）--cafile：排序节点的CA证书路径

**随后请同学们自行切换到org1，同样为其批准链码定义。**

输入命令检查两个通道成员是否都已批准链码：

peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name fabcar --version 1.0 --sequence 1 --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --output json

看到均为true，说明链码批准成功。

* **链码提交**

最后，我们需要提交已经被组织批准的链码到通道中，输入命令如下：

peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name fabcar --version 1.0 --sequence 1 --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt

命令说明：

**peer lifecycle chaincode commit (链码提交命令)**

（1）--peerAddresses：peer节点地址

（2）--tlsRootCertFiles：对应peer节点的TLS根证书

提交完成后，验证是否提交成功，输入命令：

peer lifecycle chaincode querycommitted --channelID mychannel --name fabcar --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem

如果没有出现报错，说明Fabcar链码安装成功。

* **链码调用**

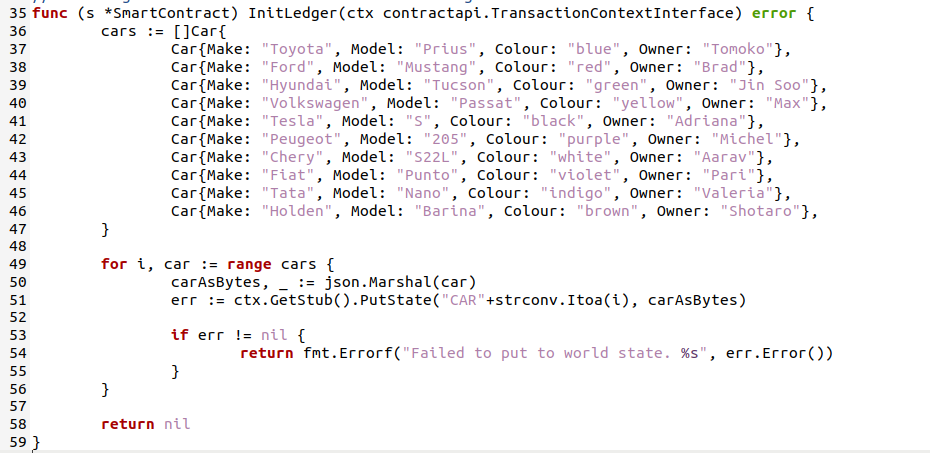
链码安装部署完成后，在配置好环境变量的终端内调用分析fabcar源码时看到的几个函数，首先调用initLedger函数对其进行初始化，输入命令如下：

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"function":"initLedger","Args":[]}'

命令说明：

**peer chaincode invoke (链码调用命令)**

（1）-c : 调用的方法名称和参数传递



根据源码，可以分析出调用了InitLedger方法后，在区块链的状态数据库中，添加了十组汽车信息，随后调用另一个函数查询所有汽车信息，输入命令如下：

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n fabcar --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"function":"queryAllCars","Args":[]}'

如果出现刚刚初始化的十组汽车信息，证明链码部署成功。

### fabric-go-SDK编写及gin包装（请截图保存在实验记录中）

fabric-go-SDK是fabric提供的基于go语言的应用程序开发包，用于和底层区块链网络进行通信，本次实验主要是实现SDK的调用链码功能，并在此基础上用gin网络框架包装为网络接口，方便使用http/https协议直接访问区块链底层链码，为之后的前后端交互做好准备。

gin框架是一个用Go语言编写的web框架，提供极其方便的路由构建及web服务功能，同学们可以前往官网文档自行学习查看(https://gin-gonic.com/zh-cn/docs/)

本次实验的SDK主要由两个文件组成，config.yaml(配置文件)和SDK.go(SDK主体)，新建一个终端并在主目录中新建一个文件夹作为工作目录，创建好相关文件，并用mod管理go依赖包工具，输入命令如下

cd ~

mkdir SDK

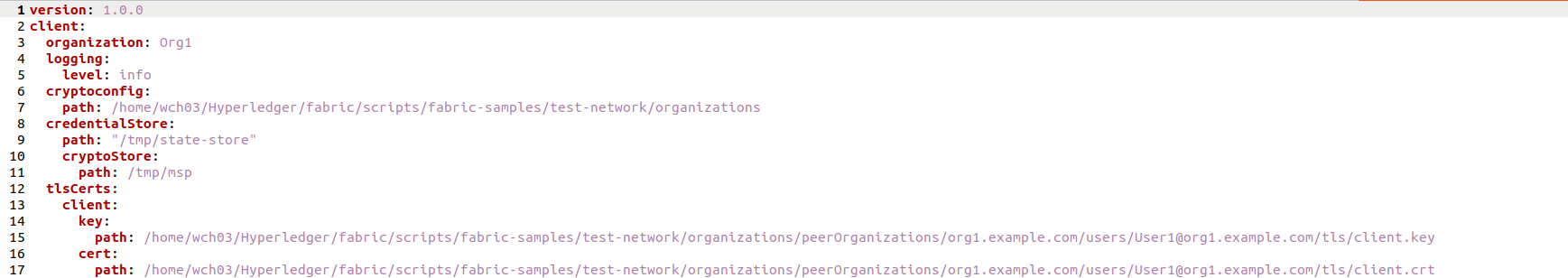
cd SDK

touch SDK.go

go mod init SDK

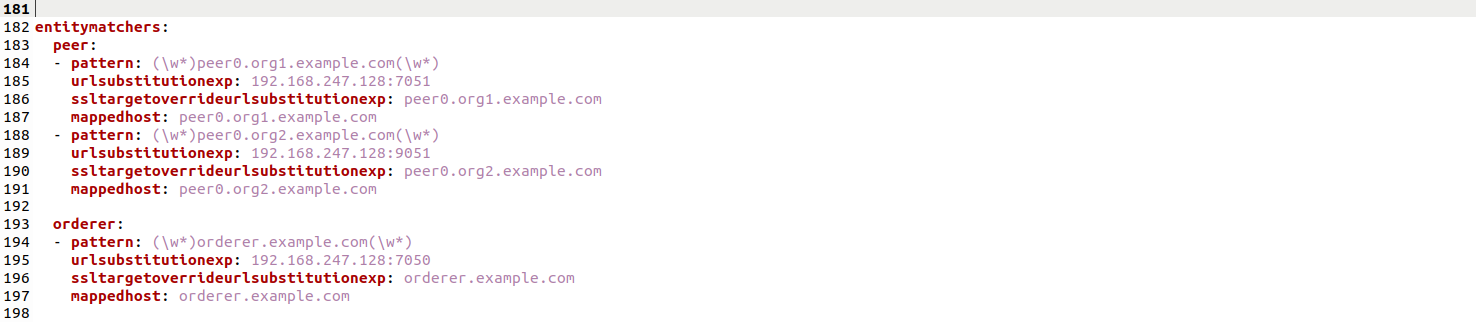
config.yaml配置文件模板已在学在浙大上给出，下载完后，同学们将模板文件拖至刚刚创建的SDK文件夹内，打开后根据本机情况主要修改两类内容：

（1）证书路径修改



类似这里以/home/wch03/Hyperledger开头的路径应全部改为对应你的虚拟机的本地路径

（2）IP地址修改



在配置文件的最后，要修改ip和主机名的映射，查看你的虚拟机的ip并修改，端口部分不用变。

修改完成后保存，正式开始SDK主体的编写。

打开刚才创建的SDK.go空白文件，并按以下步骤编写SDK

* 确定包名并引入必要的依赖

package main

import (

"log"

"net/http"

"fmt"

"github.com/hyperledger/fabric-sdk-go/pkg/client/channel"

"github.com/hyperledger/fabric-sdk-go/pkg/core/config"

"github.com/hyperledger/fabric-sdk-go/pkg/fabsdk"

"github.com/gin-gonic/gin"

)

* 定义连接区块链底层时需要的变量

var (

SDK \*fabsdk.FabricSDK

channelClient \*channel.Client

channelName = "mychannel"

chaincodeName = "fabcar"

orgName = "Org1"

orgAdmin = "Admin"

org1Peer0 = "peer0.org1.example.com"

org2Peer0 = "peer0.org2.example.com"

)

* 实现操作通道调用链码函数

func ChannelExecute(funcName string, args [][]byte)(channel.Response,error){

var err error

configPath := "./config.yaml"

configProvider := config.FromFile(configPath)

SDK,err = fabsdk.New(configProvider)

if err != nil{

log.Fatalf("Failed to create new SDK: %s", err)

}

ctx := SDK.ChannelContext(channelName,fabsdk.WithOrg(orgName),fabsdk.WithUser(orgAdmin))

channelClient,err = channel.New(ctx)

response,err := channelClient.Execute(channel.Request{

ChaincodeID : chaincodeName,

Fcn : funcName,

Args: args,

})

if err != nil{

return response,err

}

SDK.Close()

return response,nil

}

简单的函数说明：

注意到该函数有两个输入，分别为string类型的funcName和二维byte数组类型的args，前者对应想要调用的智能合约名称，后者对应调用智能合约所需的参数。

首先读入我们刚才编写的config.yaml文件，调用fabric提供的fabsdk工具创建一个SDK实例，随后使用channel工具新建一个用户通道实例，便于我们访问通道内的智能合约，最后调用channelClient工具提供的Execute方法执行智能合约获得结果并返回。

* 构建主函数，基于gin框架实现程序入口

func main() {

r := gin.Default()

r.GET("/queryAllCars",func(c \*gin.Context){

var result channel.Response

result,err := ChannelExecute("queryAllCars",[][]byte{})

fmt.Println(result)

if err != nil{

log.Fatalf("Failed to evaluate transaction: %s\n", err)

}

c.JSON(http.StatusOK,gin.H{

"code" : "200",

"message" : "Query All Success",

"result" : string(result.Payload),

})

})

r.Run(":9099")

}

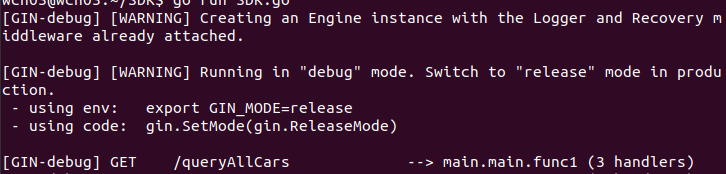
可以看到我们在9099端口上开启了gin服务，并监听/queryAllCars路由，当用户访问该路由时，程序会执行ChannelExecute函数，并传入想要执行的智能合约名称—queryAllCars，获得函数返回结果result后，gin将result以http的方式返回。

至此我们就完成了对fabcar智能合约中queryAllCars的SDK调用，保存文件后，在SDK目录下运行该项目，输入命令：

go mod tidy

go run SDK.go

出现以下界面说明运行成功。



随后用curl命令进行测试，新建一个终端（务必保持运行SDK.go的终端开启），输入命令

curl localhost:9099/queryAllCars

如果出现十组车辆的信息说明SDK和区块链底层的连接成功。

* SDK调用带参数的智能合约

下面我们对fabcar中的CreateCar合约进行SDK调用，ctrl+C关闭原先运行的SDK.go，打开SDK.go文件继续编辑。

按fabcar源码中的要求，定义一个Car结构体，方便我们之后参数传入。

在import和var之间添加以下代码：

type Car struct {

CarNumber string `json:"carnumber"`

Make string `json:"make"`

Model string `json:"model"`

Colour string `json:"colour"`

Owner string `json:"owner"`

}

在主函数中的r.GET和r.RUN之间新建一个路由，并填入以下代码：

r.POST("/createCar",func(c \*gin.Context){

var car Car

c.BindJSON(&car)

var result channel.Response

result,err := ChannelExecute("CreateCar",[][]byte{[]byte(car.CarNumber),[]byte(car.Make),[]byte(car.Model),[]byte(car.Colour),[]byte(car.Owner)})

fmt.Println(result)

if err != nil{

log.Fatalf("Failed to evaluate transaction: %s\n", err)

}

c.JSON(http.StatusOK,gin.H{

"code" : "200",

"message" : "Create Success",

"result" : string(result.Payload),

})

})

其中，c.BindJSON要求用户在向该路由发起POST请求时必须带有JSON格式的数据，具体数据类型由我们之前创建的类型Car给出，除此之外和queryAllCars唯一不同的地方在于调用ChannelExecute函数时，多传入了参数项。

保存文件后，再次运行SDK.go，随后用curl进行测试，输入命令：

curl -v http://localhost:9099/createCar -d '{"carnumber":"test","make":"test","model":"test","colour":"test","owner":"test"}' -X POST -H "Content-Type: application/json"

再次查看所有汽车，验证汽车是否添加成功：

curl localhost:9099/queryAllCars

如果出现新增的test项，说明SDK成功调用了智能合约CreateCar。

下面请同学们参照之前的例子，以这种方式对fabcar中的另一个智能合约QueryCar进行SDK调用，并用curl命令进行测试。

### 另一种添加Gin支持的方式（请截图保存在实验记录中）

* 下载学在浙大上的go.zip，解压后替换掉虚拟机上路径为~/hyperledger/fabric/scripts/fabric-samples/fabcar中的go目录
* 进入fabcar目录，先安装必要的jq命令，后运行startFabric.sh脚本,重新创建一个区块链网络

cd ~/hyperledger/fabric/scripts/fabric-samples/fabcar

sudo apt-get install jq

./startFabric.sh

* 运行完成后，进入替换后的go目录，将fabcar.go文件重新复制一份，输入命令

cd go

cp fabcar.go fabcarAPI.go

* 打开fabcarAPI.go，并修改源码如下：

1. 在import部分增加gin和http依赖包，添加以下代码

"github.com/gin-gonic/gin"

"net/http"

1. 开启gin支持

在contract := network.GetContract("fabcar")之后，加上

**router := gin.Default()**

1. 开启gin支持

为queryAllCars增加API接口queryallcars

找到程序段

result, err := contract.EvaluateTransaction("queryAllCars")

if err != nil {

fmt.Printf("Failed to evaluate transaction: %s\n", err)

os.Exit(1)

}

fmt.Println(string(result))

更改为：

var result []byte

router.GET("/queryallcars", func(c \*gin.Context) {

result, err := contract.EvaluateTransaction("queryAllCars")

if err != nil {

fmt.Printf("Failed to evaluate transaction: %s\n", err)

os.Exit(1)

}

fmt.Println(string(result))

c.String(http.StatusOK, string(result))

})

找到程序段

result, err = contract.EvaluateTransaction("queryCar", "CAR10")

if err != nil {

fmt.Printf("Failed to evaluate transaction: %s\n", err)

os.Exit(1)

}

fmt.Println(string(result))

更改为：

router.GET("/querycar", func(c \*gin.Context) {

var car string

car = c.Query("car")

result, err = contract.EvaluateTransaction("queryCar", string(car))

if err != nil {

fmt.Printf("Failed to evaluate transaction: %s\n", err)

//os.Exit(1)

}

fmt.Println(string(result))

c.String(http.StatusOK, string(result))

})

1. 下面请同学们自行修改查找createCar、queryCar、changeCarOwner

等程序段的地方，增加API接口函数createcar、querycar以及changecarowner。

1. 在func main() {…}的末尾增加一句话router.Run(":8000")

* 运行fabcarAPI.go，并用curl命令进行测试

go mod tidy

go run fabcarAPI.go

curl localhost:8000/queryallcars

* 在站点目录(/var/www/html)下新建三个php文件，命名为queryallcars.php(查询所有car)、querycar.php(查询具体car)、carindex.php(主页)，文件内容如下所示，其中querycar.php需要同学们自行完善。

**queryallcars.php**

<?php

$loginUrl = 'http://127.0.0.1:8000/queryallcars';

$ch = curl\_init();

curl\_setopt($ch, CURLOPT\_SSL\_VERIFYPEER, false);

curl\_setopt($ch, CURLOPT\_RETURNTRANSFER, true);

curl\_setopt($ch, CURLOPT\_URL,$loginUrl);

$result=curl\_exec($ch);//call the api and return the result

curl\_close($ch);

echo "After running the API url: ".$loginUrl. "<br>";

echo "The result of JSON data:<hr>";

var\_dump($result);

echo "<br><br>";

echo "<hr>";

echo "To JSON object data:<hr>";

$cars=json\_decode($result,true);

var\_dump($cars);

echo "<br><br>";

//Get the total count of CARS:

$N=count($cars);

echo 'There are total &nbsp;'.$N.'&nbsp; cars';

//=====以下是解析json结果，动态生成界面

//first car

echo "<hr>";

$car=$cars[0];

echo $car["Key"].':';

$rec=$car["Record"];

echo $rec["make"].'|'.$rec["model"].'|'.$rec["make"].'|'.$rec["color"].'|'.$rec["owner"];

echo "<hr>";

//second car

$car=$cars[1];

echo $car["Key"].':';

$rec=$car["Record"];

echo $rec["make"].'|'.$rec["model"].'|'.$rec["make"].'|'.$rec["color"].'|'.$rec["owner"];

//for loop

for($i=2;$i<$N;$i++){

$car=$cars[$i];

echo $car["Key"].':';

$rec=$car["Record"];

echo $rec["make"].'|'.$rec["model"].'|'.$rec["make"].'|'.$rec["color"].'|'.$rec["owner"];

echo "<hr>";

}

?>

**querycar.php**

<?php

$car\_s=$\_POST['txtCar'];

$loginUrl = 'http://127.0.0.1:8000/querycar?car='.$car\_s;

$ch = curl\_init();

curl\_setopt($ch, CURLOPT\_SSL\_VERIFYPEER, false);

curl\_setopt($ch, CURLOPT\_RETURNTRANSFER, true);

curl\_setopt($ch, CURLOPT\_URL,$loginUrl);

$result=curl\_exec($ch);

curl\_close($ch);

echo "After running the API url: ".$loginUrl. "<br>";

echo "The result of JSON data:<hr>";

var\_dump($result);

echo "<br><br>";

echo "<hr>";

echo "To JSON object data:<hr>";

$cars=json\_decode($result,true);

var\_dump($cars);

/\*以下是解析json数据的结果\*/

/\*请大家自己补充HTML代码\*/

?>

**carindex.php**

<html>

<head></head>

<body>

这是一个测试访问Fabric后端的Fabcar信息的主页。

<hr>

<form action="queryallcars.php" method="POST">

要查看Faric区块链上的所有的CAR：

<input type="submit" name="btnQueryAllCars" id="btnQueryAllCars" value="QueryAllCars">

</form>

<br>

<hr>

<form action="querycar.php" method="POST">请输入一个CAR名称：

<input type=text id="txtCar" name="txtCar" value="CAR10">

<input type="submit" name="btnQueryCar" id="btnQueryCar" value="QueryCar">

</form>

</body>

</html>

* 三个php文件新建完成后，保持fabricAPI的运行状态，在虚拟机浏览器内打开地址进行测试。

localhost/carindex.php

**作业上交内容与事项：**

1．实验报告文档：按照要求完成实验并将关键步骤实验结果进行截图记录。注意文档工整；

2．程序代码：如果有程序代码或其他相关材料，可汇总压缩所有文件并上传压缩包。注意文件命名；

**本期实验作业上交期限：**

请在实验设置的截止日期内提交实验报告，若逾期提交，成绩会被适当打折。

**本次作业上交内容：**

* 实验报告文档(.docx)
* SDK文件夹(包含config.yaml、SDK.go以及querycar.php)

## 三、实验感受及记录

* 1. **实验感受（本次实验遇到的问题、主要收获等内容）**

本次实验令我收获良多，使得我的认识不再只是停留在对区块链的理论上，而实实在在地接触到了其技术实现。

智能合约是本次实验的核心部分。在实验过程中，我编写并部署了智能合约。虽然过程略显复杂，但通过具体操作，我更加明确了智能合约的作用和优势，它不仅仅是一种自动化的合同执行方式，更是确保数据不可篡改和交易透明的关键技术。从中体现出智能合约在去中心化应用中的潜力，以及它在实现可信计算方面的巨大价值。

另一个让我感触颇深的是对Hyperledger Fabric SDK的使用。在实验中，我学会了如何通过SDK与区块链网络进行交互。通过调用SDK，能够在不同的节点之间执行交易、查询账本数据，这让我认识到SDK在区块链应用开发中的重要性。它不仅是开发人员的工具，更是实现区块链技术与实际业务对接的桥梁。

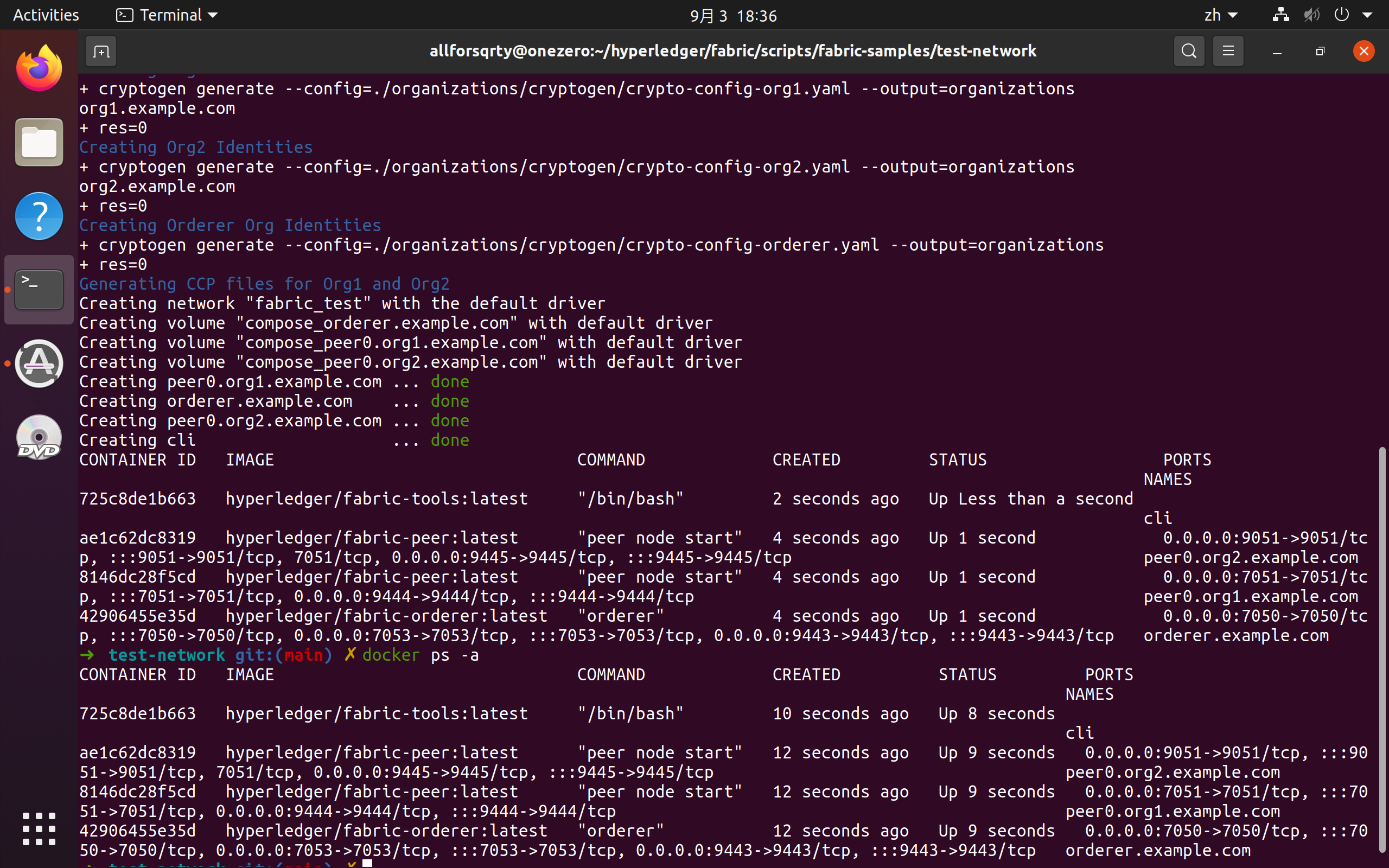
实验还让我对Hyperledger Fabric的架构有了更深的理解。Fabric的模块化设计让我认识到，不同的模块如共识机制、身份管理、通道等都可以根据业务需求进行定制，这使得Fabric在处理复杂业务场景时具备了强大的适应能力。

通过本次实验，我不仅掌握了基本的区块链开发技能，并且我开始真正理解区块链技术在现实世界中的应用价值，收获良多。

* 1. **实验记录（实验过程中关键步骤截图记录及文字描述）**

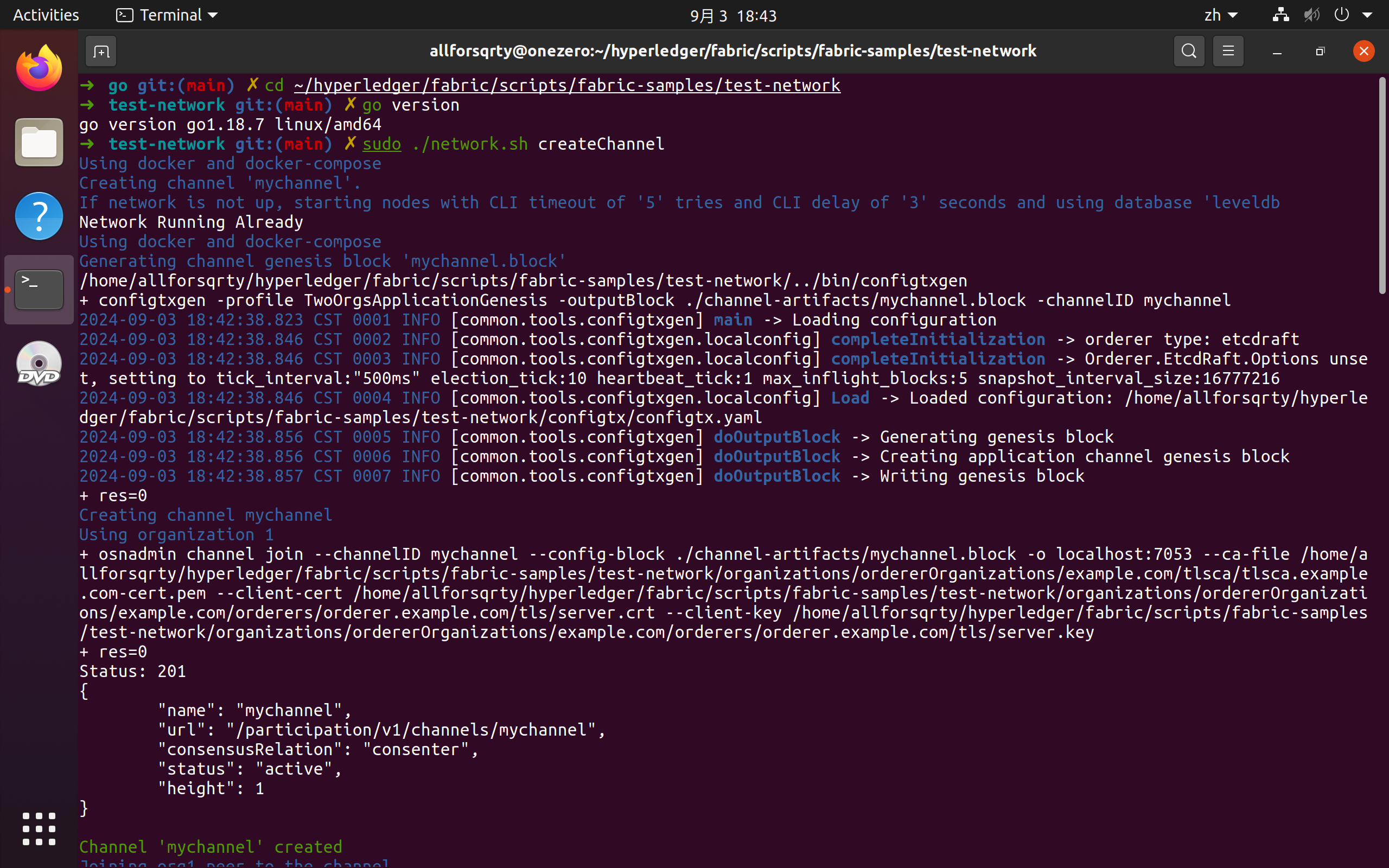
**1 运行test-network区块链网络**

如下图所示，首先建立一个基于三个节点、两个组织的区块链网络。构建成功后输出四个容器的信息。

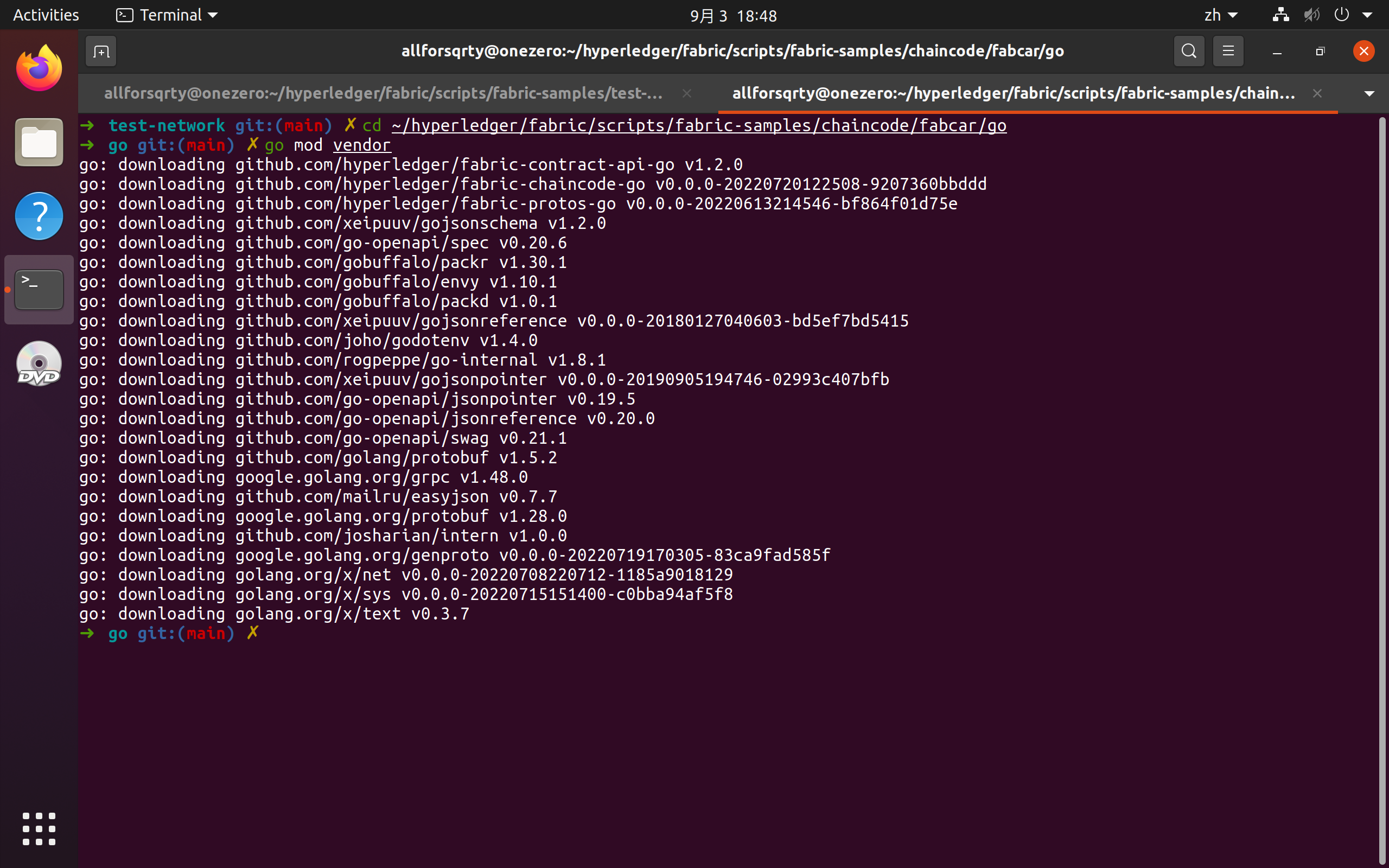


**2 Fabcar智能合约部署及调用**

1. 创建一个channel通道，默认名为mychannel，并设置好环境变量。

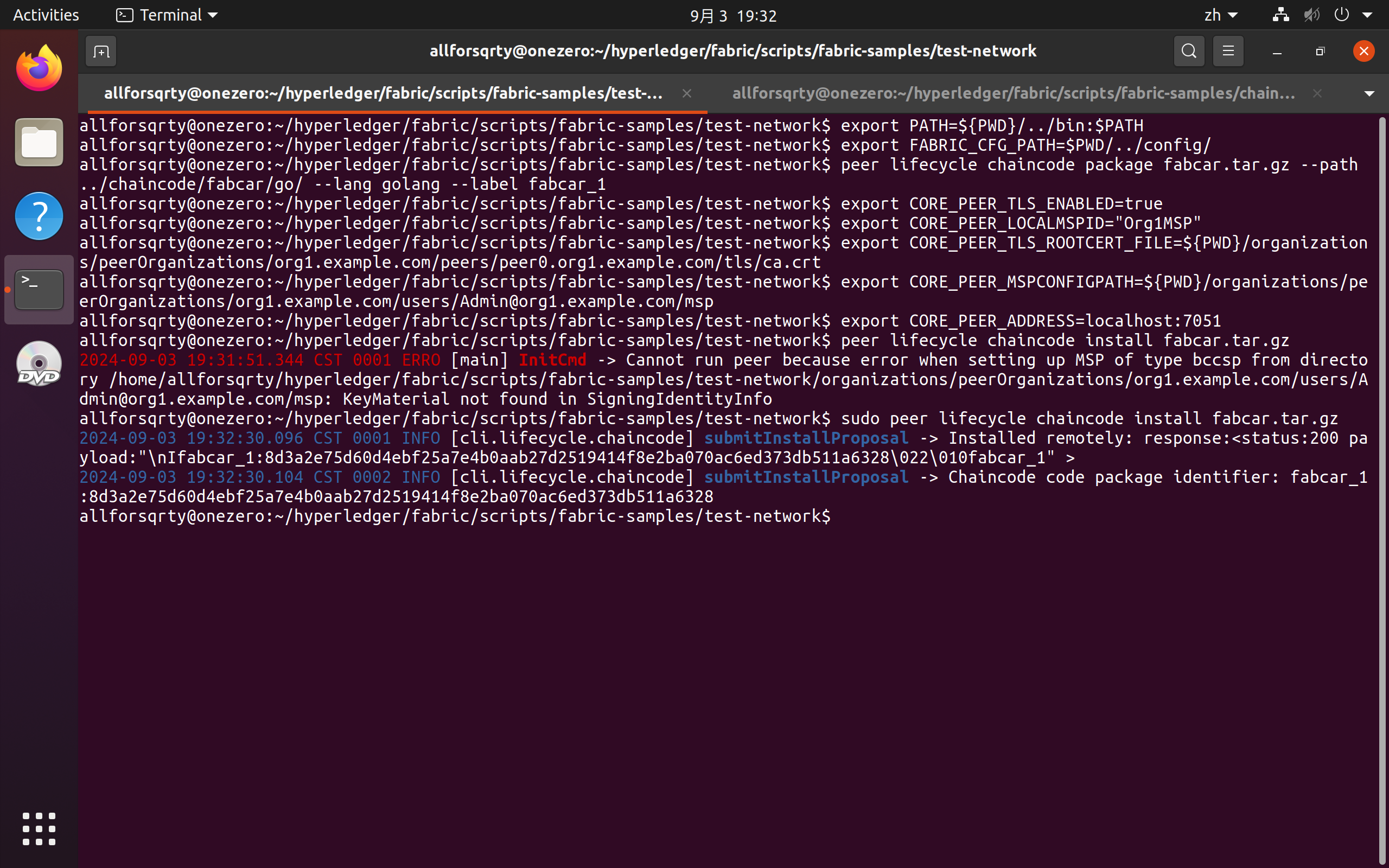


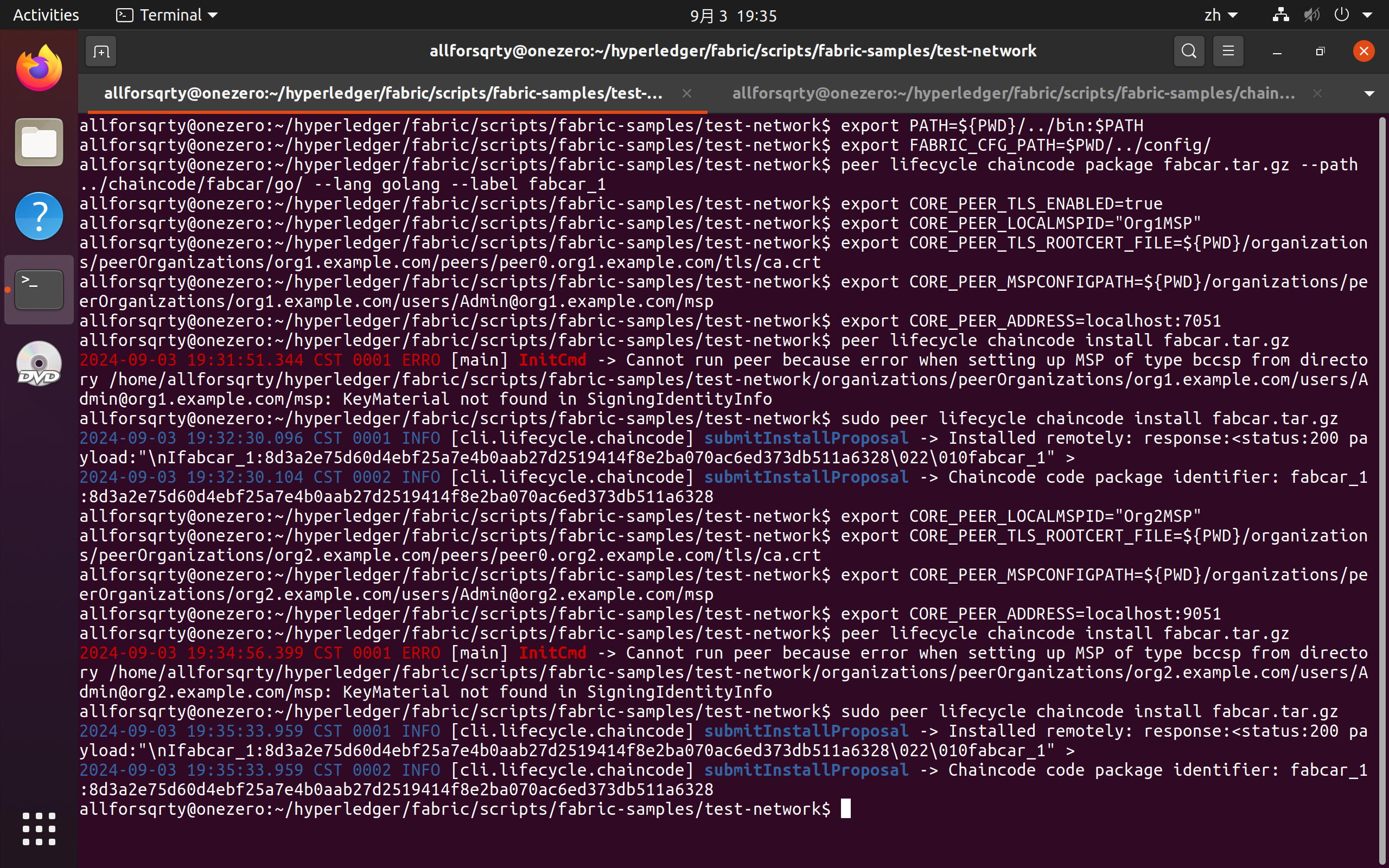
1. 新建终端B，并安装依赖项。



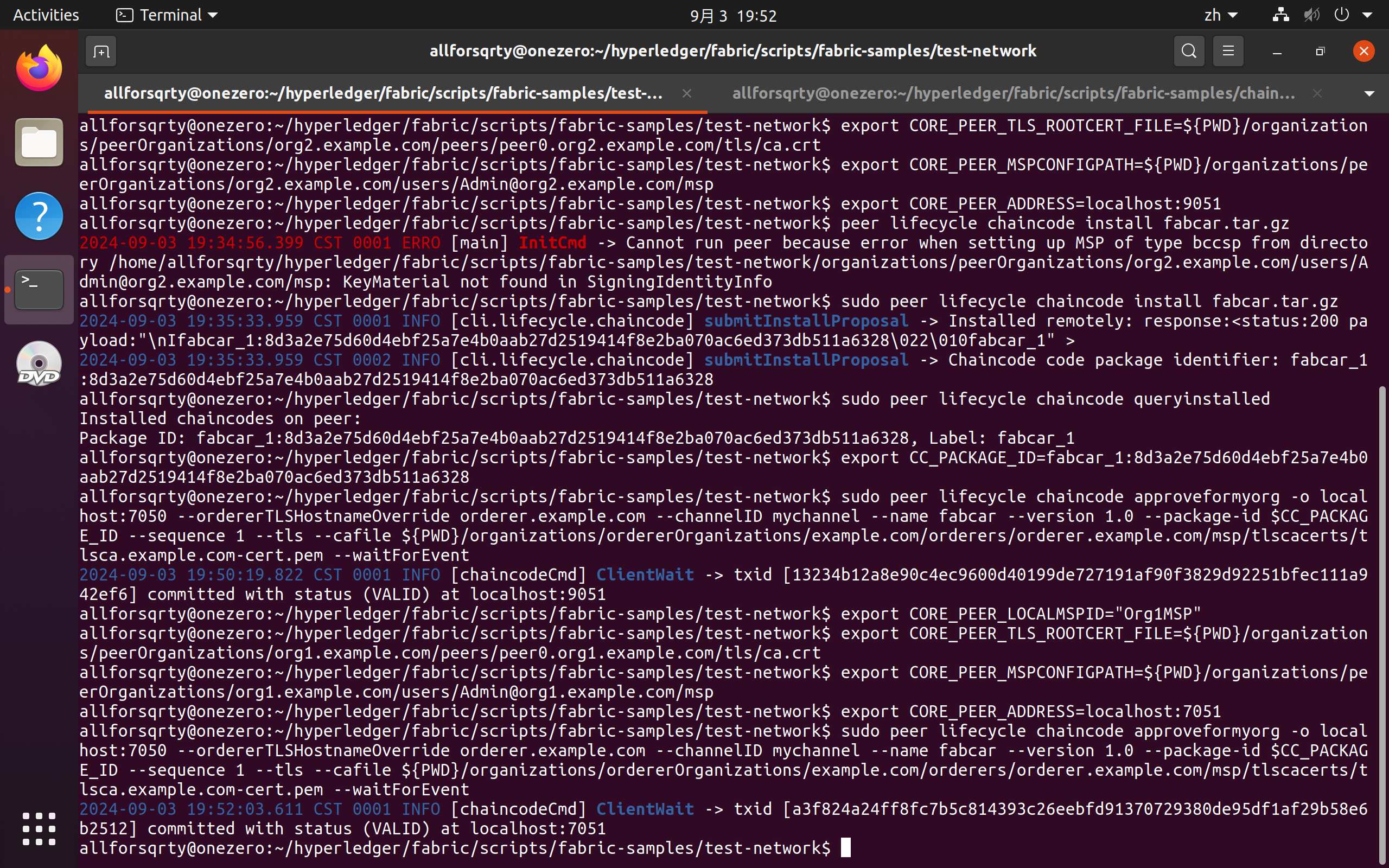
1. 在终端A中进行链码打包操作，之后进行链码安装

首先通过更改环境变量，切换到组织一安装。再切换到组织二安装。

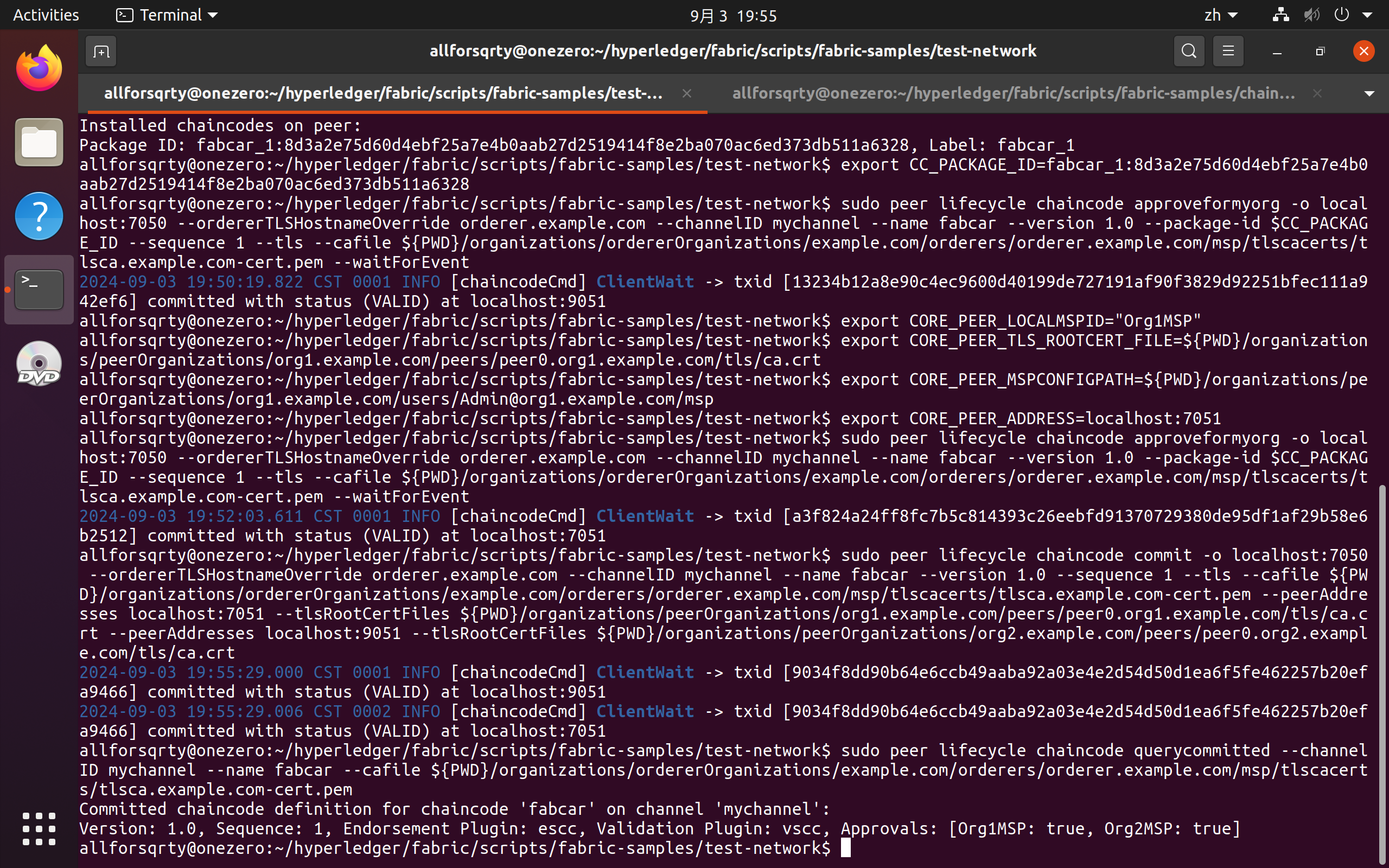




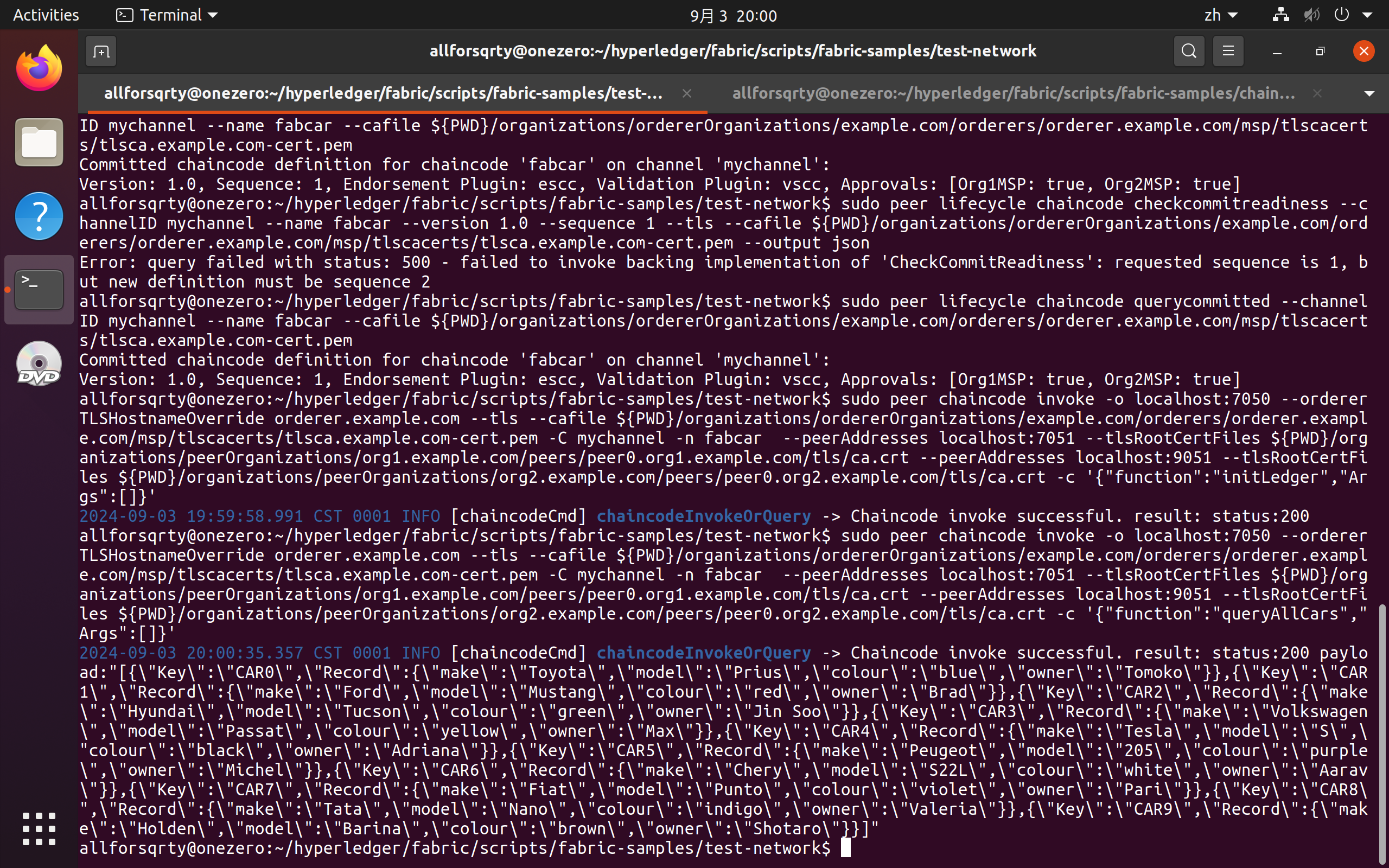
1. 批准链码定义，先批准org2，再切换到org1批准。



1. 将链码提交，显示成功提交。



1. 链码调用



如上图所示，先初始化，调用InitLedger方法，添加十组汽车信息。再调用query

AllCars方法，成功查看十组汽车信息。