

Hyperledger Fabric

Tutorials

教程

We offer tutorials to get you started with Hyperledger Fabric. The first is oriented to the Hyperledger Fabric application developer, Writing Your First Application. It takes you through the process of writing your first blockchain application for Hyperledger Fabric using the Hyperledger Fabric Node SDK.

我们提供教程让您开始使用 Hyperledger 结构。第一个面向 HyperledgerFabric 应用程序开发人员，编写第一个应用程序。它将带您完成使用 Hyperledger Fabric Node SDK 为 Hyperledger Fabric 编写第一个区块链应用程序的过程。

The second tutorial is oriented towards the Hyperledger Fabric network operators, Building Your First Network. This one walks you through the process of establishing a blockchain network using Hyperledger Fabric and provides a basic sample application to test it out.

第二个教程面向 HyperledgerFabric 网络运营商，构建您的第一个网络。这一个引导您完成使用超级账本结构建立区块链网络的过程，并提供一个基本的示例应用程序来测试它。

There are also tutorials for updating your channel, Adding an Org to a Channel, and upgrading your network to a later version of Hyperledger Fabric, Upgrading Your Network Components.

还有一些教程可以帮助您更新频道、向频道添加组织、将网络升级到更高版本的 Hyperledger Fabric、升级网络组件。

Finally, we offer two chaincode tutorials. One oriented to developers, Chaincode for Developers, and the other oriented to operators, Chaincode for Operators.

最后，我们提供两个链代码教程。一个面向开发人员，一个面向开发人员，另一个面向操作员，一个面向操作员。

一、Writing Your First Application

一、编写你的第一个应用

Note

注释

If you're not yet familiar with the fundamental architecture of a Fabric network, you may want to visit the Key Concepts section prior to continuing.

如果您还不熟悉结构网络的基本架构，那么在继续之前，您可能需要访问“关键概念”部分。

It is also worth noting that this tutorial serves as an introduction to Fabric applications and uses simple smart contracts and applications. For a more in-depth look at Fabric applications and smart contracts, check out our Developing Applications section or the Commercial paper tutorial.

值得注意的是，本教程作为结构应用程序的介绍，使用简单的智能合约和应用程序。要更深

入地了解结构应用程序和智能合约，请查看我们的开发应用程序部分或商业论文教程。

In this tutorial we'll be looking at a handful of sample programs to see how Fabric apps work. These applications and the smart contracts they use are collectively known as FabCar. They provide a great starting point to understand a Hyperledger Fabric blockchain. You'll learn how to write an application and smart contract to query and update a ledger, and how to use a Certificate Authority to generate the X.509 certificates used by applications which interact with a permissioned blockchain.

在本教程中，我们将查看一些示例程序，以了解结构应用程序如何工作。这些应用程序和它们使用的智能合约统称为 Fabcar。它们为理解超级账本结构区块链提供了一个很好的起点。您将学习如何编写应用程序和智能合约来查询和更新分类账，以及如何使用证书颁发机构生成与许可的区块链交互的应用程序使用的 X.509 证书。

We will use the application SDK — described in detail in the Application topic — to invoke a smart contract which queries and updates the ledger using the smart contract SDK — described in detail in section Smart Contract Processing.

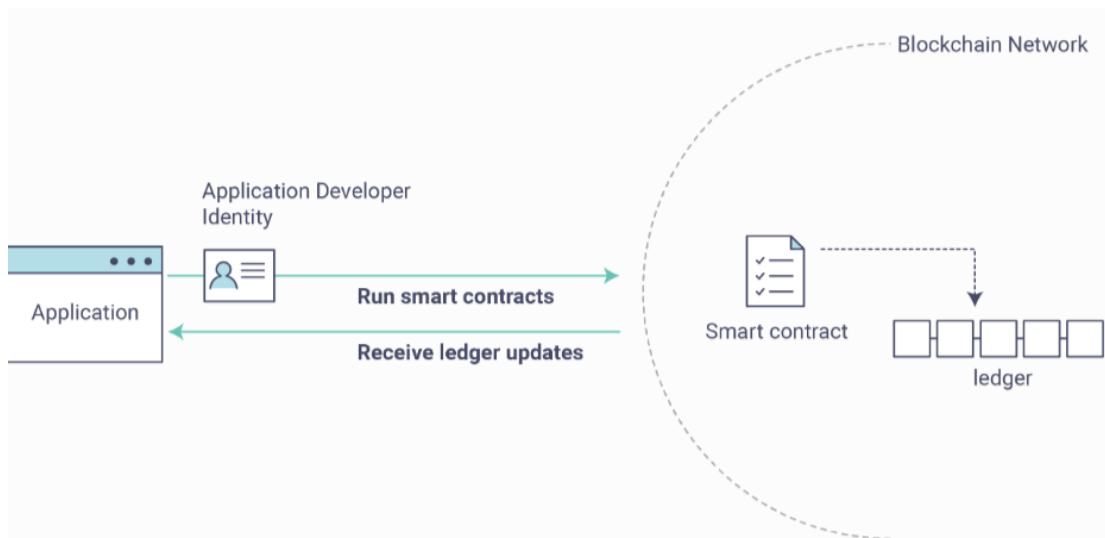
我们将使用应用程序 SDK（在应用程序主题中详细描述）来调用智能合约，该合约使用智能合约 SDK 查询和更新分类账（在智能合约处理一节中详细描述）。

We'll go through three principle steps:

我们将经历三个主要步骤

1. Setting up a development environment. Our application needs a network to interact with, so we'll get a basic network our smart contracts and application will use.

1、建立一个开发环境。我们的应用程序需要一个网络来进行交互，因此我们将得到一个我们的智能合约和应用程序将使用的基本网络。



2. Learning about a sample smart contract, FabCar. We use a smart contract written in JavaScript. We'll inspect the smart contract to learn about the transactions within them, and how they are used by applications to query and update the ledger.

2、了解智能合约范本 Fabcar。我们使用一个用 javascript 编写的智能合约。我们将检查智能合约，了解其中的交易，以及应用程序如何使用它们来查询和更新分类帐。

3. Develop a sample application which uses FabCar. Our application will use

the FabCar smart contract to query and update car assets on the ledger. We'll get into the code of the apps and the transactions they create, including querying a car, querying a range of cars, and creating a new car.

3、开发一个使用 Fabcar 的示例应用程序。我们的应用程序将使用 Fabcar 智能合约查询和更新分类账上的汽车资产。我们将了解应用程序的代码及其创建的事务,包括查询汽车、查询一系列汽车以及创建一辆新车。

After completing this tutorial you should have a basic understanding of how an application is programmed in conjunction with a smart contract to interact with the ledger hosted and replicated on the peers in a Fabric network.

完成本教程后,您应该基本了解如何将应用程序与智能合约一起编程,以与结构网络中对等端上托管和复制的分类帐交互

Note

注释

These applications are also compatible with Service Discovery and Private data, though we won't explicitly show how to use our apps to leverage those features.

这些应用程序还与服务发现和私有数据兼容,尽管我们不会明确显示如何使用我们的应用程序来利用这些功能。

1、Set up the blockchain network

1、建立区块链网络

Note

注释

This next section requires you to be in the first-network subdirectory within your local clone of the fabric-samples repo.

下一部分要求您克隆 fabric-samples repo 的到本地 first-network 子目录中。

If you've already run through Building Your First Network, you will have downloaded fabric-samples and have a network up and running. Before you run this tutorial, you must stop this network:

如果您已经完成了构建第一个网络的过程,那么您将下载了 fabric-samples,并启动并运行了一个网络。在运行本教程之前,必须停止此网络:

```
./byfn.sh down
```

If you have run through this tutorial before, use the following commands to kill any stale or active containers. Note, this will take down all of your containers whether they're Fabric related or not.

如果您以前运行过本教程,请使用以下命令杀死任何过时或活动的容器。请注意,这将删除所有容器,无论它们是否与 Fabric 相关。

```
docker rm -f $(docker ps -aq)
```

```
docker rmi -f $(docker images | grep fabcar | awk '{print $3}')
```

If you don't have a development environment and the accompanying artifacts for the network and applications, visit the Prerequisites page and ensure you have the necessary dependencies installed on your machine.

如果您没有网络 and 应用程序的开发环境和附带的工件,请访问“前提”页并确保您的计

算机上安装了必要的依赖项。

Next, if you haven't done so already, visit the Install Samples, Binaries and Docker Images page and follow the provided instructions. Return to this tutorial once you have cloned the fabric-samples repository, and downloaded the latest stable Fabric images and available utilities.

接下来, 如果您还没有这样做, 请访问 InstallSamples、Binaries 和 Docker Images 页面, 并按照提供的说明进行操作。一旦克隆了 fabric-samples 库, 并下载了最新的稳定结构映像和可用的实用程序, 请返回本教程。

If you are using Mac OS and running Mojave, you will need to install Xcode. 如果您使用的是 mac 操作系统并运行 mojave, 则需要安装 xcode。

Launch the network

启动网络

Note

注释

This next section requires you to be in the fabcar subdirectory within your local clone of the fabric-samples repo.

下一部分要求您进入 Fabcar 子目录中的 Fabric Samples repo 的本地克隆。

Launch your network using the startFabric.sh shell script. This command will spin up a blockchain network comprising peers, orderers, certificate authorities and more. It will also install and instantiate a javascript version of the FabCar smart contract which will be used by our application to access the ledger. We'll learn more about these components as we go through the tutorial.

使用 startfabric.sh shell 脚本启动网络。这个命令将启动一个由对等方、订购方、证书颁发机构等组成的区块链网络。它还将安装并实例化 fabcar 智能合约的 javascript 版本, 我们的应用程序将使用该版本访问分类账。在本教程中, 我们将进一步了解这些组件。

```
./startFabric.sh javascript
```

Alright, you've now got a sample network up and running, and the FabCar smart contract installed and instantiated. Let's install our application prerequisites so that we can try it out, and see how everything works together.

好吧, 现在你已经有了一个示例网络, 运行起来了, 安装并实例化了 Fabcar 智能合约。让我们安装我们的应用程序先决条件, 这样我们就可以试用它, 并看看所有东西是如何一起工作的。

Install the application

安装应用程序

Note

注释

The following instructions require you to be in the fabcar/javascript subdirectory within your local clone of the fabric-samples repo.

以下说明要求您在 Fabcar/javascript 子目录中, 该目录位于 Fabric 示例 repo 的本地克隆中。

Run the following command to install the Fabric dependencies for the applications. It will take about a minute to complete:

运行以下命令安装应用程序的结构依赖项。大约需要一分钟来完成:

```
npm install
```

This process is installing the key application dependencies defined in `package.json`. The most important of which is the `fabric-network` class; it enables an application to use identities, wallets, and gateways to connect to channels, submit transactions, and wait for notifications. This tutorial also uses the `fabric-ca-client` class to enroll users with their respective certificate authorities, generating a valid identity which is then used by `fabric-network` class methods.

此过程正在安装 `package.json` 中定义的关键应用程序依赖项。其中最重要的是 `Fabric Network` 类；它使应用程序能够使用标识、钱包和网关连接到通道、提交交易和等待通知。本教程还使用 `Fabric CA` 客户端类向用户注册各自的证书颁发机构，生成有效的标识，然后由 `Fabric` 网络类方法使用。

Once `npm install` completes, everything is in place to run the application. For this tutorial, you'll primarily be using the application JavaScript files in the `fabcar/javascript` directory. Let's take a look at what's inside:

一旦 `NPM` 安装完成，就可以运行应用程序了。对于本教程，您将主要使用 `fabcar/javascript` 目录中的应用程序 `javascript` 文件。让我们看看里面是什么：

```
ls
```

You should see the following:

您应该看到以下内容：

```
enrollAdmin.js  node_modules      package.json  registerUser.js
invoke.js       package-lock.json  query.js      wallet
```

There are files for other program languages, for example in the `fabcar/typescript` directory. You can read these once you've used the JavaScript example - the principles are the same.

在 `fabcar/typescript` 目录中有其他程序语言的文件。一旦使用了 JavaScript 示例，您就可以阅读这些内容——原理是相同的。

If you are using Mac OS and running Mojave, you will need to install Xcode. 如果您使用的是 `mac` 操作系统并运行 `mojave`，则需要安装 `xcode`。

2、Enrolling the admin user

2、注册管理用户

Note

注释

The following two sections involve communication with the Certificate Authority. You may find it useful to stream the CA logs when running the upcoming programs by opening a new terminal shell and running `docker logs -f ca.example.com`.

以下两个部分涉及与证书颁发机构的通信。当运行即将到来的程序时，通过打开新的终端 shell 并运行 `docker logs -f ca.example.com`，您可能会发现流式传输 CA 日志非常有用。

When we created the network, an admin user — literally called admin — was created as the registrar for the certificate authority (CA). Our first step is to generate the private key, public key, and X.509 certificate for admin using the `enroll.js` program. This process uses a Certificate Signing Request (CSR) — the private and public key are first generated locally and the public key is

then sent to the CA which returns an encoded certificate for use by the application. These three credentials are then stored in the wallet, allowing us to act as an administrator for the CA.

当我们创建网络时,一个名为 admin 的管理用户被创建为证书颁发机构(CA)的注册器。我们的第一步是使用 enroll.js 程序为管理员生成私钥、公钥和 X.509 证书。此过程使用证书签名请求(CSR)–私钥和公钥首先在本地生成,然后将公钥发送到 CA,CA 返回编码的证书供应用程序使用。然后,这三个凭证存储在钱包中,允许我们充当 CA 的管理员。

We will subsequently register and enroll a new application user which will be used by our application to interact with the blockchain.

我们随后将注册并注册一个新的应用程序用户,该用户将被我们的应用程序用于与区块链交互。

Let's enroll user admin:

让我们注册用户管理员:

```
node enrollAdmin.js
```

This command has stored the CA administrator's credentials in the wallet directory.

此命令已将 CA 管理员的凭据存储在 wallet 目录中。

3、Register and enroll user1

3、注册并注册 user1

Now that we have the administrator's credentials in a wallet, we can enroll a new user — user1 — which will be used to query and update the ledger:

现在,我们在钱包中有了管理员的凭证,我们可以注册一个新用户–用户 1–用于查询和更新分类账:

```
node registerUser.js
```

Similar to the admin enrollment, this program uses a CSR to enroll user1 and store its credentials alongside those of admin in the wallet. We now have identities for two separate users — admin and user1 — and these are used by our application.

与管理员注册类似,此程序使用 CSR 注册用户 1 并将其凭证与管理员凭证一起存储在钱包中。现在,我们有了两个独立用户的标识——admin 和 user1——这些都由我们的应用程序使用。

Time to interact with the ledger...

是时候与分类帐互动了...

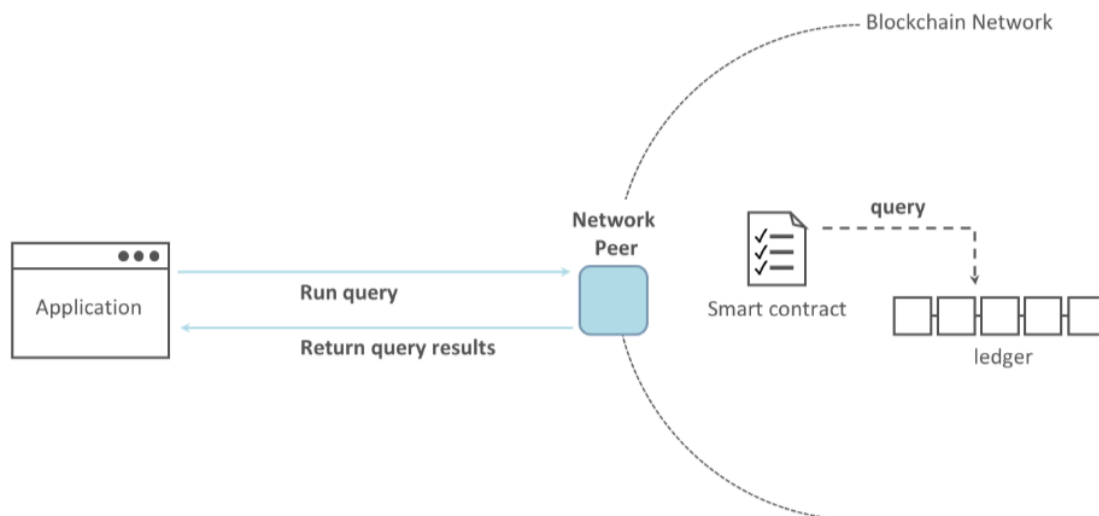
4、Querying the ledger

4、查询分类账

Each peer in a blockchain network hosts a copy of the ledger, and an application program can query the ledger by invoking a smart contract which queries the most recent value of the ledger and returns it to the application.

区块链网络中的每一个对等点都有一个分类账的副本,应用程序可以通过调用智能合约查询分类账,该智能合约查询分类账的最新值并将其返回给应用程序。

Here is a simplified representation of how a query works:
以下是查询工作方式的简化表示:



Applications read data from the ledger using a query. The most common queries involve the current values of data in the ledger - its world state. The world state is represented as a set of key-value pairs, and applications can query data for a single key or multiple keys. Moreover, the ledger world state can be configured to use a database like CouchDB which supports complex queries when key-values are modeled as JSON data. This can be very helpful when looking for all assets that match certain keywords with particular values; all cars with a particular owner, for example.

应用程序使用查询从分类帐中读取数据。最常见的查询涉及到分类帐中数据的当前值——它的世界状态。世界状态表示为一组键值对，应用程序可以查询单个键或多个键的数据。此外，可以将分类帐世界状态配置为使用类似 couchdb 的数据库，该数据库支持将键值建模为 JSON 数据时的复杂查询。这在查找与特定关键字匹配的所有资产时非常有用；例如，具有特定所有者的所有汽车。

First, let's run our query.js program to return a listing of all the cars on the ledger. This program uses our second identity - user1 - to access the ledger:

首先，让我们运行 query.js 程序返回分类帐上所有汽车的列表。这个程序使用我们的第二个身份——用户 1——访问分类帐：

`node query.js`

The output should look like this:

输出应该如下所示：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
[{"Key": "CAR0", "Record": {"colour": "blue", "make": "Toyota", "model": "Prius", "owner": "Tomoko"}},
{"Key": "CAR1", "Record": {"colour": "red", "make": "Ford", "model": "Mustang", "owner": "Brad"}},
{"Key": "CAR2", "Record": {"colour": "green", "make": "Hyundai", "model": "Tucson", "owner": "Jin Soo"}},
{"Key": "CAR3", "Record": {"colour": "yellow", "make": "Volkswagen", "model": "Passat", "owner": "Max"}},
{"Key": "CAR4", "Record": {"colour": "black", "make": "Tesla", "model": "S", "owner": "Adriana"}},
{"Key": "CAR5", "Record": {"colour": "purple", "make": "Peugeot", "model": "205", "owner": "Michel"}},
{"Key": "CAR6", "Record": {"colour": "white", "make": "Chery", "model": "S22L", "owner": "Aarav"}},
{"Key": "CAR7", "Record": {"colour": "violet", "make": "Fiat", "model": "Punto", "owner": "Pari"}},
{"Key": "CAR8", "Record": {"colour": "indigo", "make": "Tata", "model": "Nano", "owner": "Valeria"}},
{"Key": "CAR9", "Record": {"colour": "brown", "make": "Holden", "model": "Barina", "owner": "Shotaro"}}]
```



```

ke@ke-s-computer MINGW64 /d/blockchain/Golang/src/github.com/hyperledger/fabric/scripts/fabric-samples/fabcar/javas
$ node query.js
wallet path: D:\blockchain\Golang\src\github.com\hyperledger\fabric\scripts\fabric-samples\fabcar\javascript\wallet
Transaction has been evaluated, result is: [{"Key": "CAR0", "Record": {"colour": "blue", "make": "Toyota", "model": "Prius",
owner": "Brad"}}, {"Key": "CAR2", "Record": {"colour": "green", "make": "Hyundai", "model": "Tucson", "owner": "Jin Soo"}}, {"Key":
ey": "CAR4", "Record": {"colour": "black", "make": "Tesla", "model": "S", "owner": "Adriana"}}, {"Key": "CAR5", "Record": {"colou
": "white", "make": "Chery", "model": "S22", "owner": "Aarav"}}, {"Key": "CAR7", "Record": {"colour": "violet", "make": "Fiat",
el": "Nano", "owner": "Valeria"}}, {"Key": "CAR9", "Record": {"colour": "brown", "make": "Holden", "model": "Barina", "owner": "

```

Let's take a closer look at this program. Use an editor (e.g. atom or visual studio) and open query.js.

让我们仔细看看这个程序。使用编辑器(例如Atom或Visual Studio)并打开query.js。

The application starts by bringing in scope two key classes from the fabric-network module; FileSystemWallet and Gateway. These classes will be used to locate the user1 identity in the wallet, and use it to connect to the network:

应用程序首先从结构网络模块引入范围两个密钥类: filesystemwallet 和 gateway。这些类将用于定位钱包中的 user1 标识, 并使用它连接到网络:

```
const { FileSystemWallet, Gateway } = require('fabric-network');
```

The application connects to the network using a gateway:

应用程序使用网关连接到网络:

```
const gateway = new Gateway();
await gateway.connect(ccp, { wallet, identity: 'user1' });
```

This code creates a new gateway and then uses it to connect the application to the network. ccp describes the network that the gateway will access with the identity user1 from wallet. See how the ccp has been loaded from ../../basic-network/connection.json and parsed as a JSON file:

此代码创建一个新的网关, 然后使用它将应用程序连接到网络。CCP 描述了网关将通过钱包中的标识用户 1 访问的网络。请参阅如何从 ../../basic network/connection.json 加载 CCP 并将其解析为 JSON 文件:

```
const ccpPath = path.resolve(__dirname, '..', '..', 'basic-network',
'connection.json');
const ccpJSON = fs.readFileSync(ccpPath, 'utf8');
const ccp = JSON.parse(ccpJSON);
```

If you'd like to understand more about the structure of a connection profile, and how it defines the network, check out the connection profile topic.

如果您想进一步了解连接配置文件的结构以及它如何定义网络, 请查看连接配置文件主题。

A network can be divided into multiple channels, and the next important line of code connects the application to a particular channel within the network, mychannel:

网络可以分为多个通道, 下一行重要代码将应用程序连接到网络中的特定通道 mychannel:

Within this channel, we can access the smart contract fabcar to interact with the ledger:

在此渠道中, 我们可以访问智能合约 Fabcar 与分类账交互:

```
const contract = network.getContract('fabcar');
```

Within fabcar there are many different transactions, and our application initially uses the queryAllCars transaction to access the ledger world state data:

在 Fabcar 中有许多不同的事务，我们的应用程序最初使用 QueryAllCars 交易访问分类帐世界状态数据：

```
const result = await contract.evaluateTransaction('queryAllCars');
```

The evaluateTransaction method represents one of the simplest interaction with a smart contract in blockchain network. It simply picks a peer defined in the connection profile and sends the request to it, where it is evaluated. The smart contract queries all the cars on the peer's copy of the ledger and returns the result to the application. This interaction does not result in an update the ledger.

被评估交易方法是区块链网络中与智能合约最简单的交互方式之一。它只选择在连接配置文件中定义的一个对等机，并将请求发送给它，在那里对其进行评估。智能合约查询同级分类账副本上的所有汽车，并将结果返回给应用程序。此交互不会导致更新分类帐。

5、The FabCar smart contract

5、Fabcar 智能合约

Let's take a look at the transactions within the FabCar smart contract. Navigate to the chaincode/fabcar/javascript/lib subdirectory at the root of fabric-samples and open fabcar.js in your editor.

让我们看看 Fabcar 智能合约中的交易。导航到结构示例根目录下的 chaincode/fabcar/javascript/lib 子目录，并在编辑器中打开 fabcar.js。

See how our smart contract is defined using the Contract class:

查看如何使用合同类定义智能合同：

```
class FabCar extends Contract {...
```

Within this class structure, you'll see that we have the following transactions defined: initLedger, queryCar, queryAllCars, createCar, and changeCarOwner. For example:

在这个类结构中，您将看到我们定义了以下事务：initledger、querycar、queryallcars、createcar 和 changecarowner。例如：

```
async queryCar(ctx, carNumber) {...}
async queryAllCars(ctx) {...}
```

Let's take a closer look at the queryAllCars transaction to see how it interacts with the ledger.

让我们仔细看看 queryallcars 交易，看看它如何与分类账交互。

```
async queryAllCars(ctx) {
  const startKey = 'CAR0';
  const endKey = 'CAR999';
  const iterator = await ctx.stub.getStateByRange(startKey, endKey);
```

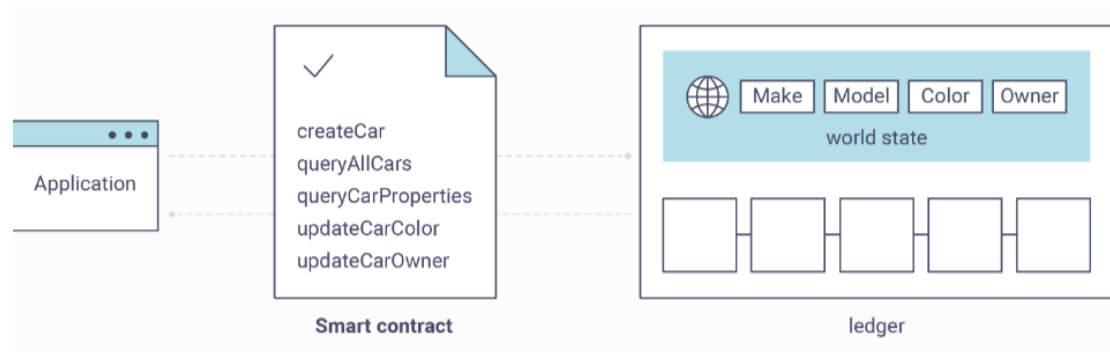
This code defines the range of cars that queryAllCars will retrieve from the ledger. Every car between CAR0 and CAR999 - 1,000 cars in all, assuming every key has been tagged properly - will be returned by the query. The remainder of the code iterates through the query results and packages them into JSON for the application.

此代码定义了 QueryAllCars 将从分类帐中检索的汽车范围。在 car0 和 car999 之间的

每一辆车——总共 1000 辆车，假设每一把钥匙都被正确标记了——将由查询返回。其余的代码迭代查询结果，并将其打包到应用程序的 JSON 中。

Below is a representation of how an application would call different transactions in a smart contract. Each transaction uses a broad set of APIs such as `getStateByRange` to interact with the ledger. You can read more about these APIs in detail.

下面是应用程序如何调用智能合约中的不同事务的表示。每个事务都使用一组广泛的 API（如 `GetStateByRange`）与分类帐进行交互。您可以详细阅读这些 API 的更多信息。



We can see our `queryAllCars` transaction, and another called `createCar`. We will use this later in the tutorial to update the ledger, and add a new block to the blockchain.

我们可以看到我们的 `queryallcars` 交易，以及另一个名为 `createcar` 的交易。我们稍后将在教程中使用这个更新分类账，并向区块链添加一个新块。

But first, go back to the query program and change the `evaluateTransaction` request to `query CAR4`. The query program should now look like this:

但首先，返回到查询程序，将被评估的事务请求更改为查询 `car4`。查询程序现在应该如下所示：

```
const result = await contract.evaluateTransaction('queryCar', 'CAR4');
```

Save the program and navigate back to your `fabcar/javascript` directory. Now run the query program again:

保存程序并导航回 `fabcar/javascript` 目录。现在再次运行查询程序：

```
node query.js
```

You should see the following:

您应该看到以下内容：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
```

```
Transaction has been evaluated, result is:
```

```
{"colour":"black","make":"Tesla","model":"S","owner":"Adriana"}
```

If you go back and look at the result from when the transaction was `queryAllCars`, you can see that `CAR4` was Adriana's black Tesla model S, which is the result that was returned here.

如果您回顾一下事务处理为 `queryallcars` 时的结果，您可以看到 `car4` 是 `adriana` 的黑色特斯拉 S 型，这是返回到这里的结果。

We can use the `queryCar` transaction to query against any car, using its key (e.g. `CAR0`) and get whatever make, model, color, and owner correspond to that car.

我们可以使用 query car 事务对任何汽车进行查询，使用其密钥（例如 car0），并获取与该汽车对应的任何品牌、型号、颜色和所有者。

Great. At this point you should be comfortable with the basic query transactions in the smart contract and the handful of parameters in the query program.

伟大的。此时，您应该熟悉智能合约中的基本查询事务和查询程序中的少量参数。

Time to update the ledger...

是时候更新分类帐了...

6、Updating the ledger

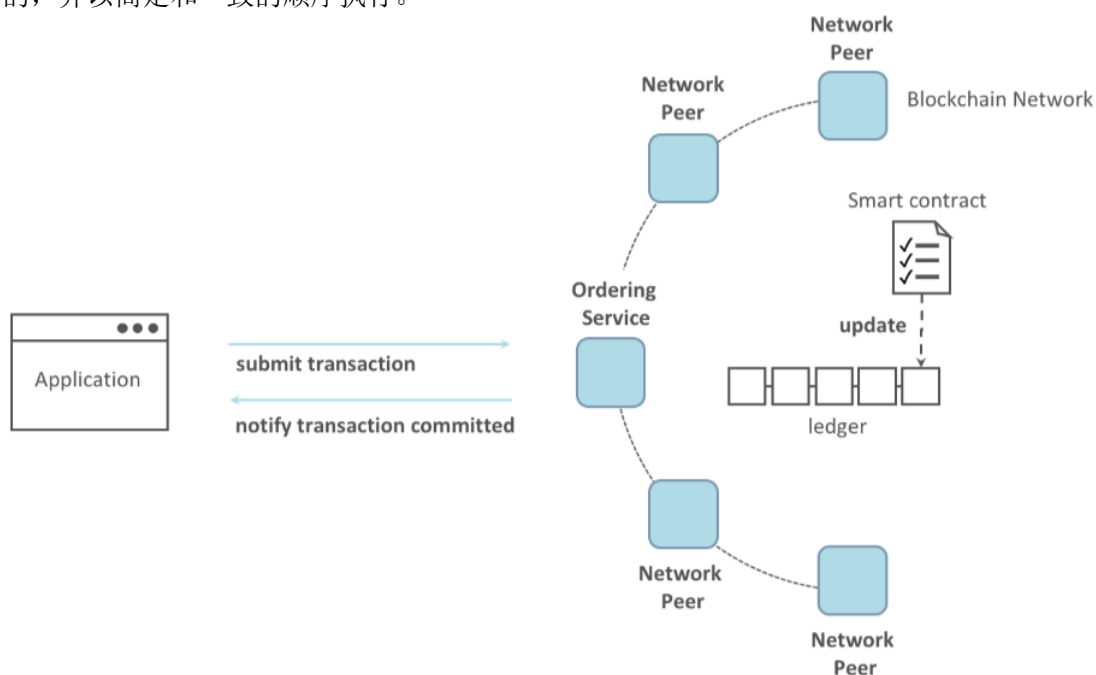
6、更新分类帐

Now that we've done a few ledger queries and added a bit of code, we're ready to update the ledger. There are a lot of potential updates we could make, but let's start by creating a new car.

现在我们已经完成了一些分类账查询并添加了一些代码，现在我们准备更新分类账了。我们可以进行很多潜在的更新，但我们先创建一辆新车。

From an application perspective, updating the ledger is simple. An application submits a transaction to the blockchain network, and when it has been validated and committed, the application receives a notification that the transaction has been successful. Under the covers this involves the process of consensus whereby the different components of the blockchain network work together to ensure that every proposed update to the ledger is valid and performed in an agreed and consistent order.

从应用程序的角度来看，更新分类帐很简单。一个应用程序向区块链网络提交一个交易，当它被验证和提交后，该应用程序收到一个交易成功的通知。在封面下，这涉及协商一致的过程，即区块链网络的不同组成部分共同工作，以确保对分类账的每一次拟议更新都是有效的，并以商定和一致的顺序执行。



Above, you can see the major components that make this process work. As well as the multiple peers which each host a copy of the ledger, and optionally a

copy of the smart contract, the network also contains an ordering service. The ordering service coordinates transactions for a network; it creates blocks containing transactions in a well-defined sequence originating from all the different applications connected to the network.

上面，您可以看到使这个过程工作的主要组件。除了多个对等点，每个对等点都承载一个分类账副本和一个智能合约副本（可选），网络还包含一个订购服务。订购服务协调网络的事务；它创建块，其中包含来自连接到网络的所有不同应用程序的定义良好的事务序列。

Our first update to the ledger will create a new car. We have a separate program called `invoke.js` that we will use to make updates to the ledger. Just as with queries, use an editor to open the program and navigate to the code block where we construct our transaction and submit it to the network:

我们对分类账的第一次更新将创建一辆新车。我们有一个名为 `invoke.js` 的单独程序，用于更新分类账。与查询一样，使用编辑器打开程序并导航到构建事务的代码块，然后将其提交到网络：

```
await contract.submitTransaction('createCar', 'CAR12', 'Honda', 'Accord', 'Black', 'Tom');
```

See how the applications calls the smart contract transaction `createCar` to create a black Honda Accord with an owner named Tom. We use `CAR12` as the identifying key here, just to show that we don't need to use sequential keys.

查看应用程序如何调用智能合约事务 `CreateCar` 来创建一个名为 Tom 的黑本田协议。我们在这里使用 `car12` 作为识别键，只是为了表明我们不需要使用顺序键。

Save it and run the program:

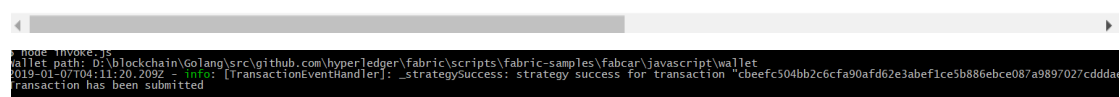
保存并运行程序：

```
node invoke.js
```

If the invoke is successful, you will see output like this:

如果调用成功，您将看到如下输出：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
2018-12-11T14:11:40.935Z - info: [TransactionEventHandler]: _strategySuccess: strategy success for
Transaction has been submitted
```



Notice how the invoke application interacted with the blockchain network using the `submitTransaction` API, rather than `evaluateTransaction`.

请注意，Invoke 应用程序如何使用 `SubmitTransaction` API 而不是 `EvaluateTransaction` 与区块链网络交互。

```
await contract.submitTransaction('createCar', 'CAR12', 'Honda', 'Accord', 'Black', 'Tom');
```

`submitTransaction` is much more sophisticated than `evaluateTransaction`. Rather than interacting with a single peer, the SDK will send the `submitTransaction` proposal to every required organization's peer in the blockchain network. Each of these peers will execute the requested smart contract using this proposal, to generate a transaction response which it signs and returns to the SDK. The SDK collects all the signed transaction responses into a single transaction, which it then sends to the orderer. The orderer collects

and sequences transactions from every application into a block of transactions. It then distributes these blocks to every peer in the network, where every transaction is validated and committed. Finally, the SDK is notified, allowing it to return control to the application.

SubmitTransaction 比 EvaluateTransaction 复杂得多。SDK 不会与单个对等端交互，而是将提交的传输建议发送给区块链网络中每个所需组织的对等端。这些对等方中的每一个都将使用此建议执行请求的智能合约，以生成一个事务响应，该响应由它签名并返回到 SDK。SDK 将所有签名的事务响应收集到一个事务中，然后发送给订购方。订购者从每个应用程序收集事务并将其排序为一个事务块。然后，它将这些块分发给网络中的每个对等端，在那里验证并提交每个事务。最后，通知 sdk，允许它将控制权返回到应用程序。

submitTransaction does all this for the application! The process by which the application, smart contract, peers and ordering service work together to keep the ledger consistent across the network is called consensus, and it is explained in detail in this section.

SubmitTransaction 为应用程序执行所有这些操作！应用程序、智能合约、同行和订购服务协同工作以保持整个网络中的分类账一致的过程称为共识，本节将对此进行详细说明。

To see that this transaction has been written to the ledger, go back to query.js and change the argument from CAR4 to CAR12.

要查看此交易记录是否已写入分类帐，请返回 query.js 并将参数从 car4 更改为 car12。

In other words, change this:

换言之，改变这一点：

```
const result = await contract.evaluateTransaction('queryCar', 'CAR4');
```

To this:

为：

```
const result = await contract.evaluateTransaction('queryCar', 'CAR12');
```

Save once again, then query:

再次保存，然后查询：

```
node query.js
```

Which should return this:

应该返回这个：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
```

```
Transaction has been evaluated, result is:
```

```
{"colour":"Black","make":"Honda","model":"Accord","owner":"Tom"}
```

Congratulations. You've created a car and verified that its recorded on the ledger!

祝贺你。您已经创建了一辆汽车，并验证它是否记录在分类帐上！

So now that we've done that, let's say that Tom is feeling generous and he wants to give his Honda Accord to someone named Dave.

既然我们已经这样做了，让我们假设汤姆很慷慨，他想把他的本田雅阁给一个叫戴夫的人。

To do this, go back to invoke.js and change the smart contract transaction from createCar to changeCarOwner with a corresponding change in input arguments:

为此，请返回 invoke.js 并将智能合约事务从 createcar 更改为 changecarowner，并在输入参数中进行相应更改：

```
await contract.submitTransaction('changeCarOwner', 'CAR12', 'Dave');
```

The first argument — CAR12 — identifies the car that will be changing owners. The second argument — Dave — defines the new owner of the car.

第一个论点-car12-确定了将要更换车主的汽车。第二个论点——戴夫——定义了汽车的新主人。

Save and execute the program again:

保存并再次执行程序:

```
node invoke.js
```

Now let's query the ledger again and ensure that Dave is now associated with the CAR12 key:

现在, 让我们再次查询分类账, 并确保 Dave 现在与 car12 键关联:

```
node query.js
```

It should return this result:

它应该返回这个结果:

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
```

```
Transaction has been evaluated, result is:
```

```
{"colour":"Black","make":"Honda","model":"Accord","owner":"Dave"}
```

The ownership of CAR12 has been changed from Tom to Dave.

Car12 的所有权已从 Tom 改为 Dave。

Note

注释

In a real world application the smart contract would likely have some access control logic. For example, only certain authorized users may create new cars, and only the car owner may transfer the car to somebody else.

在实际应用程序中, 智能合约可能具有一些访问控制逻辑。例如, 只有特定的授权用户可以创建新车, 只有车主可以将车转让给其他人。

7、Summary

7、总结

Now that we've done a few queries and a few updates, you should have a pretty good sense of how applications interact with a blockchain network using a smart contract to query or update the ledger. You've seen the basics of the roles smart contracts, APIs, and the SDK play in queries and updates and you should have a feel for how different kinds of applications could be used to perform other business tasks and operations.

既然我们已经完成了一些查询和更新, 那么您应该对应用程序如何使用智能合约与区块链网络交互以查询或更新分类账有了很好的理解。您已经了解了智能合约、API 和 SDK 在查询和更新中扮演的角色的基础知识, 您应该了解如何使用不同类型的应用程序执行其他业务任务和操作。

8、Additional resources

8、其他资源

As we said in the introduction, we have a whole section on Developing Applications that includes in-depth information on smart contracts, process and data design, a tutorial using a more in-depth Commercial Paper tutorial and a large amount of other material relating to the development of applications.

正如我们在引言中所说，我们有一个关于开发应用程序的完整部分，其中包括关于智能合约、流程和数据设计的深入信息、使用更深入的商业论文教程的教程以及大量与应用程序开发相关的其他材料。

二、Commercial paper tutorial

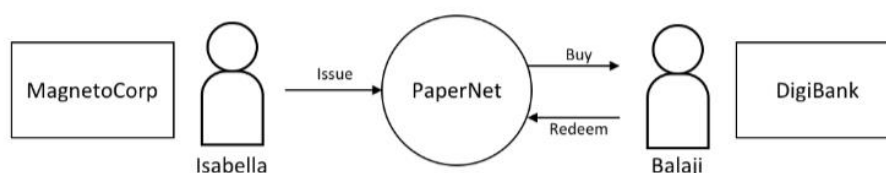
二、商业票据教程

Audience: Architects, application and smart contract developers, administrators

受众：架构师、应用程序和智能合约开发人员、管理员

This tutorial will show you how to install and use a commercial paper sample application and smart contract. It is a task-oriented topic, so it emphasizes procedures above concepts. When you'd like to understand the concepts in more detail, you can read the Developing Applications topic.

本教程将向您展示如何安装和使用商业票据示例应用程序和智能合约。它是一个以任务为导向的主题，所以它强调过程高于概念。当您想更详细地了解这些概念时，可以阅读“开发应用程序”主题。



In this tutorial two organizations, Magnetocorp and DigiBank, trade commercial paper with each other using PaperNet, a Hyperledger Fabric blockchain network.

在本教程中，Magnetorp 和 Digibank 这两个组织使用一个超级账本结构区块链网络 papernet 相互交换商业票据。

Once you've set up a basic network, you'll act as Isabella, an employee of Magnetocorp, who will issue a commercial paper on its behalf. You'll then switch hats to take the role of Balaji, an employee of DigiBank, who will buy this commercial paper, hold it for a period of time, and then redeem it with Magnetocorp for a small profit.

一旦你建立了一个基本的网络，你将扮演 Isabella，一个 Magnetocorp 公司的雇员，谁将代表它发行商业票据。然后，你将换个帽子来扮演 Balaji 的角色，Balaji 是 Digibank 的一名员工，他将购买这张商业票据，持有一段时间，然后和 Magnetorp 兑换，赚取一小笔利润。

You'll act as an developer, end user, and administrator, each in different organizations, performing the following steps designed to help you understand

what it's like to collaborate as two different organizations working independently, but according to mutually agreed rules in a Hyperledger Fabric network.

您将作为开发人员、最终用户和管理员，分别在不同的组织中执行以下步骤，旨在帮助您了解作为两个独立工作的不同组织（但根据 Hyperledger Fabric Network 中双方同意的规则）进行协作的感觉。

- Set up machine and download samples
- 设置机器并下载示例
- Create a network
- 创建网络
- Understand the structure of a smart contract
- 了解智能合约的结构
- Work as an organization, MagnetoCorp, to install and instantiate smart contract
- 作为一个组织, Magnetorp, 安装和实例化智能合约
- Understand the structure of a MagnetoCorp application, including its dependencies
- 了解 Magnetorp 应用程序的结构, 包括其依赖关系
- Configure and use a wallet and identities
- 配置和使用钱包和标识
- Run a MagnetoCorp application to issue a commercial paper
- 运行 Magnetorp 应用程序以发行商业票据
- Understand how a second organization, Digibank, uses the smart contract in their applications
- 了解第二个组织 Digibank 如何在其应用程序中使用智能合约
- As Digibank, run applications that buy and redeem commercial paper
- 作为 Digibank, 运行购买和兑换商业票据的应用程序

This tutorial has been tested on MacOS and Ubuntu, and should work on other Linux distributions. A Windows version is under development.

本教程已经在 MacOS 和 Ubuntu 上进行了测试, 并且应该可以在其他 Linux 发行版上使用。正在开发 Windows 版本。

1、Prerequisites

1、先决条件

Before you start, you must install some prerequisite technology required by the tutorial. We've kept these to a minimum so that you can get going quickly.

在开始之前, 必须安装本教程所需的一些必备技术。我们把这些放在最低限度, 这样你就能快点走了。

You must have the following technologies installed:

必须安装以下技术:

Node version 8.9.0, or higher. Node is a JavaScript runtime that you can use to run applications and smart contracts. You are recommended to use the LTS (Long Term Support) version of node. Install node [here](#).

Node 版本 8.9.0 或更高。Node 是一个 javascript 运行时，可以用来运行应用程序和智能合约。建议您使用节点的 LTS（长期支持）版本。在此处安装节点。

Docker version 18.06, or higher. Docker help developers and administrators create standard environments for building and running applications and smart contracts. Hyperledger Fabric is provided as a set of Docker images, and the PaperNet smart contract will run in a docker container. Install Docker here.

Docker 版本 18.06 或更高。Docker 帮助开发人员和管理员创建用于构建和运行应用程序和智能合约的标准环境。Hyperledger 结构作为一组 Docker 图像提供，PaperNet 智能合约将在 Docker 容器中运行。在这里安装 Docker。

You will find it helpful to install the following technologies:

您会发现安装以下技术很有帮助：

A source code editor, such as Visual Studio Code version 1.28, or higher. VS Code will help you develop and test your application and smart contract. Install VS Code here.

源代码编辑器，如 Visual Studio 代码版本 1.28 或更高版本。VS 代码将帮助您开发和测试应用程序和智能合约。在此处安装 vs 代码。

Many excellent code editors are available including Atom, Sublime Text and Brackets.

许多优秀的代码编辑器，包括 Atom、Sublime 文本和括号。

You may find it helpful to install the following technologies as you become more experienced with application and smart contract development. There's no requirement to install these when you first run the tutorial:

随着您在应用程序和智能合约开发方面的经验的增加，安装以下技术可能会有所帮助。第一次运行教程时，无需安装这些组件：

Node Version Manager. NVM helps you easily switch between different versions of node - it can be really helpful if you're working on multiple projects at the same time. Install NVM here.

Node Version Manager。NVM 可以帮助您轻松地在不同版本的节点之间进行切换，如果您同时处理多个项目，这将非常有用。在此处安装 NVM。

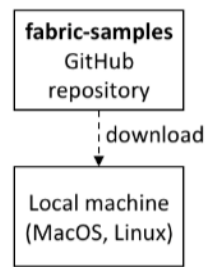
2、Download samples

2、下载示例

The commercial paper tutorial is one of the Hyperledger Fabric samples held in a public GitHub repository called fabric-samples. As you're going to run the tutorial on your machine, your first task is to download the fabric-samples repository.

商业票据教程是在一个名为 FabricSamples 的公共 Github 存储库中保存的 HyperledgeFabric 示例之一。当您要在您的机器上运行教程时，您的第一个任务是下载结构示例存储库。

<https://github.com/hyperledger/fabric-samples>



Download the fabric-samples GitHub repository to your local machine.

将结构示例 Github 存储库下载到本地计算机。

\$GOPATH is an important environment variable in Hyperledger Fabric; it identifies the root directory for installation. It is important to get right no matter which programming language you're using! Open a new terminal window and check your \$GOPATH is set using the env command:

\$gopath 是 hyperledger 结构中的一个重要环境变量；它标识要安装的根目录。无论您使用的是哪种编程语言，都必须正确处理！打开一个新的终端窗口，并检查使用 env 命令设置的 \$gopath：

```
$ env
...
GOPATH=/Users/username/go
NVM_BIN=/Users/username/.nvm/versions/node/v8.11.2/bin
NVM_IOJS_ORG_MIRROR=https://iojs.org/dist
...
```

Use the following instructions if your \$GOPATH is not set.

如果未设置 \$gopath，请使用以下说明。

You can now create a directory relative to \$GOPATH where fabric-samples will be installed:

现在，您可以创建一个与 \$gopath 相关的目录，其中将安装结构示例：

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

Use the git clone command to copy fabric-samples repository to this location:

使用 git clone 命令将结构示例存储库复制到此位置：

```
$ git clone https://github.com/hyperledger/fabric-samples.git
```

Feel free to examine the directory structure of fabric-samples:

请随意检查结构示例的目录结构：

```
$ cd fabric-samples
$ ls
```

```
$ cd fabric-samples
$ ls
```

CODE_OF_CONDUCT.md	balance-transfer	fabric-ca
CONTRIBUTING.md	basic-network	first-network
Jenkinsfile	chaincode	high-throughput
LICENSE	chaincode-docker-devmode	scripts
MAINTAINERS.md	commercial-paper	README.md
fabcar		

Notice the commercial-paper directory - that's where our sample is located!

注意商业票据目录——这就是我们的样本所在的位置！

You've now completed the first stage of the tutorial! As you proceed, you'll open multiple command windows open for different users and components. For example:

您现在已经完成了教程的第一阶段！继续操作时，将为不同的用户和组件打开多个命令窗口。例如：

- to run applications on behalf of Isabella and Balaji who will trade commercial paper with each other
- 代表 Isabella 和 Balaji 运行应用程序，他们将彼此交换商业票据。
- to issue commands to on behalf of administrators from MagnetoCorp and DigiBank, including installing and instantiating smart contracts
- 代表 Magnetorp 和 Digibank 的管理员发出命令，包括安装和实例化智能合约
- to show peer, orderer and CA log output
- 显示对等、排序器和 CA 日志输出

We'll make it clear when you should run a command from particular command window; for example:

我们将明确您何时应该从特定的命令窗口运行命令；例如：

```
(isabella)$ ls
```

indicates that you should run the ls command from Isabella's window.

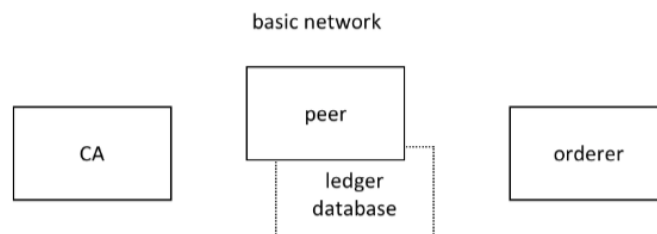
指示应该从 Isabella 的窗口运行 ls 命令。

3、Create network

3、创建网络

The tutorial currently uses the basic network; it will be updated soon to a configuration which better reflects the multi-organization structure of PaperNet. For now, this network is sufficient to show you how to develop an application and smart contract.

本教程目前使用的是基本网络；它将很快更新为更能反映 Papernet 多组织结构的配置。目前，这个网络足以向您展示如何开发应用程序和智能合约。



The Hyperledger Fabric basic network comprises a peer and its ledger database, an orderer and a certificate authority (CA). Each of these components runs as a docker container.

HyperledgeFabricBasic 网络由对等方及其分类数据库、订购方和证书颁发机构（CA）组成。每个组件都作为 Docker 容器运行。

The peer, its ledger, the orderer and the CA each run in their own docker container. In production environments, organizations typically use existing CAs that are shared with other systems; they're not dedicated to the Fabric network.

对等机、其分类帐、订购者和 CA 都在各自的 Docker 容器中运行。在生产环境中，组织通常使用与其他系统共享的现有 CA；它们不专用于结构网络。

You can manage the basic network using the commands and configuration included in the fabric-samples/basic-network directory. Let's start the network on your local machine with the start.sh shell script:

可以使用 fabric samples/basic network 目录中包含的命令和配置管理基本网络。让我们用 start.sh shell 脚本在本地计算机上启动网络：

```
$ cd fabric-samples/basic-network
$ ./start.sh

docker-compose -f docker-compose.yml up -d ca.example.com orderer.example.com peer0.org1.example.cc
Creating network "net_basic" with the default driver
Pulling ca.example.com (hyperledger/fabric-ca:)...
latest: Pulling from hyperledger/fabric-ca
3b37166ec614: Pull complete
504facff238f: Pull complete
(...)
Pulling orderer.example.com (hyperledger/fabric-orderer:)...
latest: Pulling from hyperledger/fabric-orderer
3b37166ec614: Already exists
504facff238f: Already exists
(...)
Pulling couchdb (hyperledger/fabric-couchdb:)...
latest: Pulling from hyperledger/fabric-couchdb
3b37166ec614: Already exists
504facff238f: Already exists
(...)
Pulling peer0.org1.example.com (hyperledger/fabric-peer:)...
latest: Pulling from hyperledger/fabric-peer
3b37166ec614: Already exists
504facff238f: Already exists
(...)
Creating orderer.example.com ... done
Creating couchdb ... done
Creating ca.example.com ... done
Creating peer0.org1.example.com ... done
(...)
2018-11-07 13:47:31.634 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connector
2018-11-07 13:47:31.730 UTC [channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to
```

Notice how the docker-compose -f docker-compose.yml up -d ca.example.com... command pulls the four Hyperledger Fabric container images from DockerHub, and then starts them. These containers have the most up-to-date version of the software for these Hyperledger Fabric components. Feel free to explore the basic-network directory - we'll use much of its contents during this tutorial.

注意 docker compose -f docker-compose.yml up -d ca.example.com 是如何组成的... 命令从 DockerHub 中提取四个 HyperledgeFabric 容器镜像，然后启动它们。这些容器具有最新版本的软件，用于这些 Hyperledger 结构组件。请随意浏览基本的网络目录——在本教程中，我们将使用其中的大部分内容。

You can list the docker containers that are running the basic-network components using the docker ps command:

您可以使用 docker ps 命令列出运行基本网络组件的 docker 容器：

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
ada3d078989b	hyperledger/fabric-peer	"peer node start"	About a minute ago	Up /
1fa1fd107bfb	hyperledger/fabric-orderer	"orderer"	About a minute ago	Up /
53fe614274f7	hyperledger/fabric-couchdb	"tini -- /docker-ent..."	About a minute ago	Up /
469201085a20	hyperledger/fabric-ca	"sh -c 'fabric-ca-se..."	About a minute ago	Up /

See if you can map these containers to the basic-network (you may need to horizontally scroll to locate the information):

- A peer peer0.org1.example.com is running in container ada3d078989b
- An orderer orderer.example.com is running in container 1fa1fd107bfb
- A CouchDB database couchdb is running in container 53fe614274f7
- A CA ca.example.com is running in container 469201085a20
- 对等对等机 0.org1.example.com 正在容器 ada3d078989b 中运行
- order order.example.com 正在容器 1fa1fd107bfb 中运行
- couchdb 数据库 couchdb 正在容器 53fe614274f7 中运行
- ca ca.example.com 正在容器 469201085A20 中运行

These containers all form a docker network called net_basic. You can view the network with the docker network command:

这些容器都形成了一个名为 net_basic 的 Docker 网络。您可以使用 docker network 命令查看网络:

```
$ docker network inspect net_basic

[
  {
    "Name": "net_basic",
    "Id": "62e9d37d00a0eda6c6301a76022c695f8e01258edaba6f65e876166164466ee5",
    "Created": "2018-11-07T13:46:30.4992927Z",
    "Containers": {
      "1fa1fd107bfb61522e4a26a57c2178d82b2918d5d423e7ee626c79b8a233624": {
        "Name": "orderer.example.com",
        "IPv4Address": "172.20.0.4/16",
      },
      "469201085a20b6a8f476d1ac993abce3103e59e3a23b9125032b77b02b715f2c": {
        "Name": "ca.example.com",
        "IPv4Address": "172.20.0.2/16",
      },
      "53fe614274f7a40392210f980b53b421e242484dd3deac52bbfe49cb636ce720": {
        "Name": "couchdb",
        "IPv4Address": "172.20.0.3/16",
      },
      "ada3d078989b568c6e060fa7bf62301b4bf55bed8ac1c938d514c81c42d8727a": {
        "Name": "peer0.org1.example.com",
        "IPv4Address": "172.20.0.5/16",
      },
    },
    "Labels": {}
  }
]
```

See how the four containers use different IP addresses, while being part of a single docker network. (We've abbreviated the output for clarity.)

了解这四个容器如何使用不同的 IP 地址, 同时作为单个 Docker 网络的一部分。(为了清晰起见, 我们对输出进行了缩写。)

To recap: you've downloaded the Hyperledger Fabric samples repository from GitHub and you've got the basic network running on your local machine. Let's now start to play the role of MagnetoCorp, who wish to trade commercial paper.

回顾一下: 您已经从 Github 下载了 HyperledgerFabricSamples 存储库, 并且您的本地计算机上运行了基本网络。现在让我们开始扮演希望交易商业票据的 MagnetoCorp 公司的角色。

4、Working as MagnetoCorp

4、作为 MagetoCorp 公司工作

To monitor the MagetoCorp components of PaperNet, an administrator can view the aggregated output from a set of docker containers using the logspout tool. It collects the different output streams into one place, making it easy to see what's happening from a single window. This can be really helpful for administrators when installing smart contracts or for developers when invoking smart contracts, for example.

要监视 Papernet 的 Magnetorp 组件, 管理员可以使用 logpute 工具查看一组 Docker 容器的聚合输出。它将不同的输出流收集到一个地方, 这样就可以很容易地从一个窗口看到正在发生的事情。例如, 在安装智能合约或在调用智能合约时, 这对管理员或开发人员非常有用。

Let's now monitor PaperNet as a MagetoCorp administrator. Open a new window in the fabric-samples directory, and locate and run the monitordocker.sh script to start the logspout tool for the PaperNet docker containers associated with the docker network net_basic:

现在让我们以 MagetoCorp 管理员的身份监视 Papernet。打开 fabric samples 目录中的一个新窗口, 找到并运行 monitordocker.sh 脚本以启动与 docker network net_basic 关联的 papernet docker 容器的 logspout 工具:

```
(magnetocorp admin)$ cd commercial-paper/organization/magnetocorp/configuration/cli/
(magnetocorp admin)$ ./monitordocker.sh net_basic
...
latest: Pulling from gliderlabs/logspout
4fe2ade4980c: Pull complete
decca452f519: Pull complete
(...)
Starting monitoring on all containers on the network net_basic
b7f3586e5d0233de5a454df369b8eadab0613886fc9877529587345fc01a3582
```

Note that you can pass a port number to the above command if the default port in monitordocker.sh is already in use.

请注意, 如果 monitordocker.sh 中的默认端口已在使用中, 则可以将端口号传递给上述命令。

```
(magnetocorp admin)$ ./monitordocker.sh net_basic <port_number>
```

This window will now show output from the docker containers, so let's start another terminal window which will allow the MagetoCorp administrator to interact with the network.

这个窗口现在将显示 Docker 容器的输出, 所以让我们启动另一个终端窗口, 它将允许 Magnetorp 管理员与网络交互。

A MagetoCorp administrator interacts with the network via a docker container. MagetoCorp 公司管理员通过 Docker 容器与网络交互。

To interact with PaperNet, a MagetoCorp administrator needs to use the Hyperledger Fabric peer commands. Conveniently, these are available pre-built in the hyperledger/fabric-tools docker image.

要与 Papernet 交互, Magnetorp 管理员需要使用 HyperledgeFabric 对等命令。这些工具可以很方便地在 Hyperledger/Fabric 工具 Docker 图像中预先构建。

Let's start a MagetoCorp-specific docker container for the administrator using the docker-compose command:

让我们使用 `docker compose` 命令为管理员启动一个 MagnetoCorp 公司特定的 docker 容器:

```
(magnetocorp admin)$ cd commercial-paper/organization/magnetocorp/configuration/cli/
(magnetocorp admin)$ docker-compose -f docker-compose.yml up -d cliMagnetoCorp

Pulling cliMagnetoCorp (hyperledger/fabric-tools:...)
latest: Pulling from hyperledger/fabric-tools
3b37166ec614: Already exists
(...)
Digest: sha256:058cff3b378c1f3ebe35d56deb7bf33171bf19b327d91b452991509b8e9c7870
Status: Downloaded newer image for hyperledger/fabric-tools:latest
Creating cliMagnetoCorp ... done
```

Again, see how the `hyperledger/fabric-tools` docker image was retrieved from Docker Hub and added to the network:

同样, 请参见如何从 Docker Hub 中检索 Hyperledger/Fabric 工具 Docker 映像并将其添加到网络:

```
(magnetocorp admin)$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
562a88b25149	hyperledger/fabric-tools	"/bin/bash"	About a minute ago	Up
b7f3586e5d02	gliderlabs/logspout	"/bin/logspout"	7 minutes ago	Up
ada3d078989b	hyperledger/fabric-peer	"peer node start"	29 minutes ago	Up
1fa1fd107bfb	hyperledger/fabric-orderer	"orderer"	29 minutes ago	Up
53fe614274f7	hyperledger/fabric-couchdb	"tiny -- /docker-ent..."	29 minutes ago	Up
469201085a20	hyperledger/fabric-ca	"sh -c 'fabric-ca-se..."	29 minutes ago	Up

The MagnetoCorp administrator will use the command line in container `562a88b25149` to interact with PaperNet. Notice also the `logspout` container `b7f3586e5d02`; this is capturing the output of all other docker containers for the `monitordocker.sh` command.

磁电机公司管理员将使用容器 `562A88B25149` 中的命令行与 PaperNet 交互。还要注意 `logspout` 容器 `b7f3586e5d02`; 这将捕获 `monitordocker.sh` 命令的所有其他 docker 容器的输出。

Let's now use this command line to interact with PaperNet as the MagnetoCorp administrator.

现在让我们使用此命令行作为 Magnetorp 管理员与 PaperNet 进行交互。

5、Smart contract

5、智能合约

`issue`, `buy` and `redeem` are the three functions at the heart of the PaperNet smart contract. It is used by applications to submit transactions which correspondingly `issue`, `buy` and `redeem` commercial paper on the ledger. Our next task is to examine this smart contract.

发行、购买和赎回是 PaperNet 智能合约的三个核心功能。它被应用程序用来提交相应地在分类账上发行、购买和赎回商业票据的交易。我们的下一个任务是检查这个智能合约。

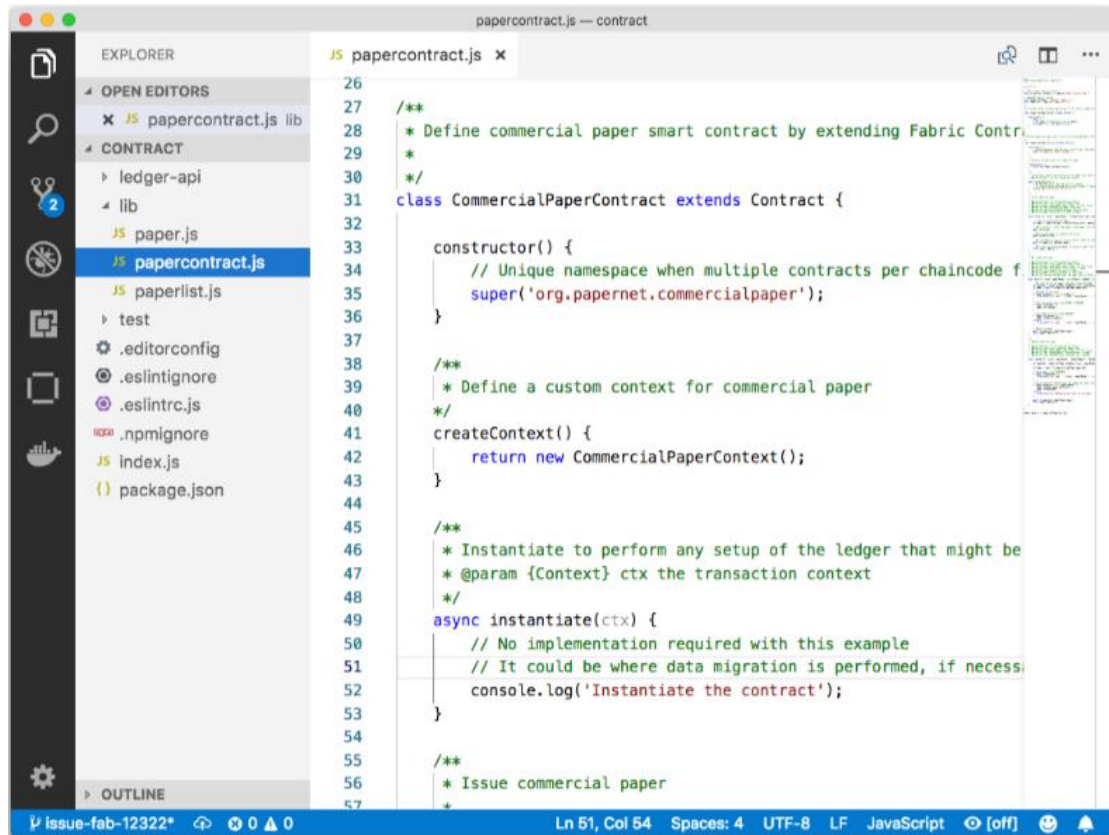
Open a new terminal window to represent a MagnetoCorp developer and change to the directory that contains MagnetoCorp's copy of the smart contract to view it with your chosen editor (VS Code in this tutorial):

打开一个新的终端窗口来表示 Magnetorp 开发人员, 并更改到包含 Magnetorp 智能合约副本的目录, 以便使用您选择的编辑器 (本教程中的 `vs` 代码) 查看该智能合约:

```
(magnetocorp developer)$ cd commercial-paper/organization/magnetocorp/contract
(magnetocorp developer)$ code .
```

In the lib directory of the folder, you'll see papercontract.js file - this contains the commercial paper smart contract!

在文件夹的 lib 目录中，您将看到 paper contract.js 文件——其中包含商业票据智能合约！



```
26
27 /**
28  * Define commercial paper smart contract by extending Fabric Contract
29  */
30
31 class CommercialPaperContract extends Contract {
32
33   constructor() {
34     // Unique namespace when multiple contracts per chaincode f
35     super('org.paper.net.commercialpaper');
36   }
37
38   /**
39    * Define a custom context for commercial paper
40    */
41   createContext() {
42     return new CommercialPaperContext();
43   }
44
45   /**
46    * Instantiate to perform any setup of the ledger that might be
47    * @param {Context} ctx the transaction context
48    */
49   async instantiate(ctx) {
50     // No implementation required with this example
51     // It could be where data migration is performed, if necess
52     console.log('Instantiate the contract');
53   }
54
55   /**
56    * Issue commercial paper
57   */
```

An example code editor displaying the commercial paper smart contract in papercontract.js

在 paper contract.js 中显示商业票据智能合约的示例代码编辑器

papercontract.js is a JavaScript program designed to run in the node.js environment. Note the following key program lines:

papercontract.js 是一个旨在在 node.js 环境中运行的 javascript 程序。注意以下关键程序行：

```
const { Contract, Context } = require('fabric-contract-api');
```

This statement brings into scope two key Hyperledger Fabric classes that will be used extensively by the smart contract - Contract and Context. You can learn more about these classes in the fabric-shim JSDOCS.

此声明将智能合约广泛使用的两个关键的超级账本结构类（契约和上下文）纳入范围。您可以在 fabric-shim JSdocs 中了解关于这些类的更多信息。

```
class CommercialPaperContract extends Contract {
```

This defines the smart contract class CommercialPaperContract based on the built-in Fabric Contract class. The methods which implement the key transactions to issue, buy and redeem commercial paper are defined within this class.

这定义了基于内置 Fabric 合同类的智能合同类商业纸质合同。在这个类中定义了实现发行、购买和赎回商业票据的关键交易的方法。

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime...) {
```

This method defines the commercial paper issue transaction for PaperNet. The parameters that are passed to this method will be used to create the new commercial paper.

此方法定义了 PaperNet 的商业票据发行交易。传递给此方法的参数将用于创建新的商业票据。

Locate and examine the buy and redeem transactions within the smart contract.
查找并检查智能合约中的买入和赎回交易。

```
let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime...);
```

Within the issue transaction, this statement creates a new commercial paper in memory using the CommercialPaper class with the supplied transaction inputs. Examine the buy and redeem transactions to see how they similarly use this class.

在 Issue 事务中，此语句使用商业票据类和提供的事务输入在内存中创建新的商业票据。检查买入和赎回交易，以了解它们如何类似地使用此类。

```
await ctx.paperList.addPaper(paper);
```

This statement adds the new commercial paper to the ledger using ctx.paperList, an instance of a PaperList class that was created when the smart contract context CommercialPaperContext was initialized. Again, examine the buy and redeem methods to see how they use this class.

此语句使用 ctx.paperlist 将新的商业票据添加到分类账中，该实例是在初始化智能合约上下文商业票据上下文时创建的 paperlist 类。再次检查购买和兑换方法，看看它们如何使用这个类。

```
return paper.toBuffer();
```

This statement returns a binary buffer as response from the issue transaction for processing by the caller of the smart contract.

此语句返回二进制缓冲区作为问题事务的响应，供智能合约调用方处理。

Feel free to examine other files in the contract directory to understand how the smart contract works, and read in detail how papercontract.js is designed in the smart contract topic.

请随意查看合同目录中的其他文件，了解智能合同的工作原理，并详细阅读智能合同主题中的 paperContract.js 是如何设计的。

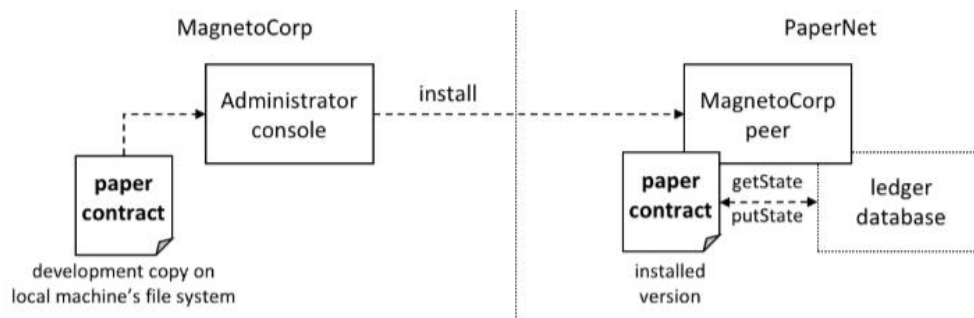
6、Install contract

6、安装合同

Before papercontract can be invoked by applications, it must be installed onto the appropriate peer nodes in PaperNet. MagnetoCorp and DigiBank administrators are able to install papercontract onto peers over which they respectively have authority.

在应用程序可以调用 PaperContract 之前，必须将其安装到 PaperNet 中相应的对等节

点上。Magnetorp 和 Digibank 管理员可以将纸质合同安装到他们各自拥有权限的对等机上。



A MagnetoCorp administrator installs a copy of the papercontract onto a MagnetoCorp peer.

MagnetoCorp 公司管理员将票据合同的副本安装到磁电机公司的对等机上。

Smart contracts are the focus of application development, and are contained within a Hyperledger Fabric artifact called chaincode. One or more smart contracts can be defined within a single chaincode, and installing a chaincode will allow them to be consumed by the different organizations in PaperNet. It means that only administrators need to worry about chaincode; everyone else can think in terms of smart contracts.

智能合约是应用程序开发的重点，包含在一个名为 chaincode 的超账本结构工件中。一个或多个智能合约可以在单个链码中定义，安装链码将允许不同组织在 PaperNet 中使用它们。这意味着只有管理员需要担心链码；其他人都可以考虑智能合约。

The MagnetoCorp administrator uses the peer chaincode install command to copy the papercontract smart contract from their local machine's file system to the file system within the target peer's docker container. Once the smart contract is installed on the peer and instantiated on a channel, papercontract can be invoked by applications, and interact with the ledger database via the putState() and getState() Fabric APIs. Examine how these APIs are used by StateList class within ledger-api\statelist.js.

MagnetoCorp 公司管理员使用 peer chaincode install 命令将 PaperContract 智能合约从本地计算机的文件系统复制到目标对等机 Docker 容器中的文件系统。一旦智能合约安装在对等机上并在通道上实例化，应用程序就可以调用 PaperContract，并通过 putstate() 和 getstate() 结构 API 与分类账数据库交互。检查这些 API 如何被 ledger api\statelist.js 中的 statelist 类使用。

Let's now install papercontract as the MagnetoCorp administrator. In the MagnetoCorp administrator's command window, use the docker exec command to run the peer chaincode install command in the cliMagnetoCorp container:

现在让我们以 Magnetorp 管理员的身份安装 PaperContract。在 Magnetorp 管理员的命令窗口中，使用 docker exec 命令在 cliMagnetoCorp 容器中运行 peer chaincode install 命令：

```
(magnetocorp admin)$ docker exec cliMagnetoCorp peer chaincode install -n
papercontract -v 0 -p /opt/gopath/src/github.com/contract -l node
2018-11-07 14:21:48.400 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001
Using default escc
```

```
2018-11-07 14:21:48.400 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
Using default vscc
2018-11-07 14:21:48.466 UTC [chaincodeCmd] install -> INFO 003 Installed
remotely response:<status:200 payload:"OK" >
```

The cliMagnetCorp container has set CORE_PEER_ADDRESS=peer0.org1.example.com:7051 to target its commands to peer0.org1.example.com, and the INFO 003 Installed remotely... indicates papercontract has been successfully installed on this peer. Currently, the MagnetCorp administrator only has to install a copy of papercontract on a single MagnetCorp peer.

climagnetcorp 容器已将 core_peer_address=peer0.org1.example.com:7051 设置为将其命令目标指向 peer0.org1.example.com, 并远程安装了 info 003...表示 PaperContract 已成功安装在此对等机上。目前, Magnetorp 管理员只需要在单个 Magentorp 对等机上安装纸质合同的副本。

Note how peer chaincode install command specified the smart contract path, -p, relative to the cliMagnetCorp container's file system: /opt/gopath/src/github.com/contract. This path has been mapped to the local file system path .../organization/magnetocorp/contract via the magnetocorp/configuration/cli/docker-compose.yml file:

请注意, peer chaincode 安装命令如何指定智能合约路径 -p (相对于 climagnetcorp 容器的文件系统): /opt/gopath/src/github.com/contract。此路径已通过 magecorp/configuration/cli/docker-compose.yml 文件映射到本地文件系统路径.../organization/magecorp/contract:

volumes:

```
- ...
- ../../../../../../organization/magnetocorp:/opt/gopath/src/github.com/
- ...
```

See how the volume directive maps organization/magnetocorp to /opt/gopath/src/github.com/ providing this container access to your local file system where MagnetCorp's copy of the papercontract smart contract is held.

请参阅 volume 指令如何将 organization/magnercorp 映射到 /opt/gopath/src/github.com/ 以提供此容器访问您的本地文件系统, 其中保存了 magnercorp 的纸质合同智能合约副本。

You can read more about docker compose here and peer chaincode install command here.

您可以阅读更多关于 docker compose here 和 peer chaincode install 命令的信息。

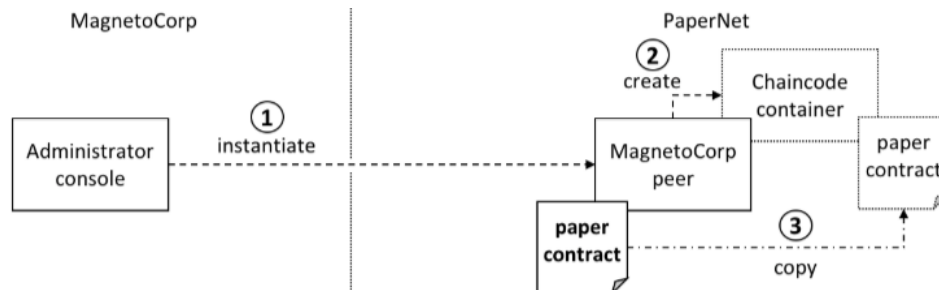
7、Instantiate contract

7、实例化合同

Now that papercontract chaincode containing the CommercialPaper smart contract is installed on the required PaperNet peers, an administrator can make it available to different network channels, so that it can be invoked by applications connected to those channels. Because we're using the basic network configuration for PaperNet, we're only going to make papercontract available in

a single network channel, mychannel.

现在,包含商业票据智能合约的PaperContract 链码安装在所需的PaperNet 对等端上,管理员可以将其提供给不同的网络通道,以便连接到这些通道的应用程序可以调用它。因为我们使用的是 PaperNet 的基本网络配置,所以我们只在一个网络通道 MyChannel 中提供 PaperContract。



A Magnetocorp administrator instantiates papercontract chaincode containing the smart contract. A new docker chaincode container will be created to run papercontract.

Magnetocorp 公司管理员实例化包含智能合约的纸质合同链码。将创建一个新的 Docker Chaincode 容器来运行 PaperContract。

The Magnetocorp administrator uses the peer chaincode instantiate command to instantiate papercontract on mychannel:

Magnetorp 管理员使用对等链代码实例化命令在 mychannel 上实例化 PaperContract:

```
(magnetocorp admin)$ docker exec cliMagnetocorp peer chaincode instantiate -n
papercontract -v 0 -l node -c
'{"Args":["org.paper.net.commercialpaper:instantiate"]}' -C mychannel -P "AND
('Org1MSP.member')"
```

```
2018-11-07 14:22:11.162 UTC [chaincodeCmd] InitCmdFactory -> INFO 001 Retrieved
channel (mychannel) orderer endpoint: orderer.example.com:7050
2018-11-07 14:22:11.163 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
Using default escc
2018-11-07 14:22:11.163 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default vscc
```

One of the most important parameters on instantiate is -P. It specifies the endorsement policy for papercontract, describing the set of organizations that must endorse (execute and sign) a transaction before it can be determined as valid. All transactions, whether valid or invalid, will be recorded on the ledger blockchain, but only valid transactions will update the world state.

实例化时最重要的参数之一是-p。它指定了书面合同的背书策略,描述了在确定交易有效之前必须对交易进行背书(执行和签署)的一组组织。所有交易,无论是有效的还是无效的,都将记录在分类区块链上,但只有有效的交易才能更新世界状态。

In passing, see how instantiate passes the orderer address orderer.example.com:7050. This is because it additionally submits an instantiate transaction to the orderer, which will include the transaction in the next block and distribute it to all peers that have joined mychannel, enabling any peer to

execute the chaincode in their own isolated chaincode container. Note that instantiate only needs to be issued once for papercontract even though typically it is installed on many peers.

在 passing 中,请参见 instantiate 如何传递 orderer 地址 order.example.com:7050。这是因为它还向订购方提交了一个实例化事务,该事务将包含在下一个块中,并将其分发给加入 mychannel 的所有对等方,从而使任何对等方都可以在自己的独立链码容器中执行链码。请注意,Instantation 只需要为 PaperContract 发布一次,即使它通常安装在许多对等机上。

See how a papercontract container has been started with the docker ps command:

请参阅如何使用 docker ps 命令启动 PaperContract 容器:

```
(magnetocorp admin)$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
CREATED	STATUS	PORTS NAMES
4fac1b91bfda	dev-peer0.org1.example.com-papercontract-0-d96...	"/bin/sh -c 'cd /usr..." 2 minutes ago Up 2 minutes dev-peer0.org1.example.com-papercontract-0

Notice that the container is named dev-peer0.org1.example.com-papercontract-0-d96... to indicate which peer started it, and the fact that it's running papercontract version 0.

请注意,容器名为 dev-peer0.org1.example.com-papercontract-0-d96...指出是哪个对等机启动的,以及它正在运行 PaperContract 版本 0 的事实。

Now that we've got a basic PaperNet up and running, and papercontract installed and instantiated, let's turn our attention to the MagnetoCorp application which issues a commercial paper.

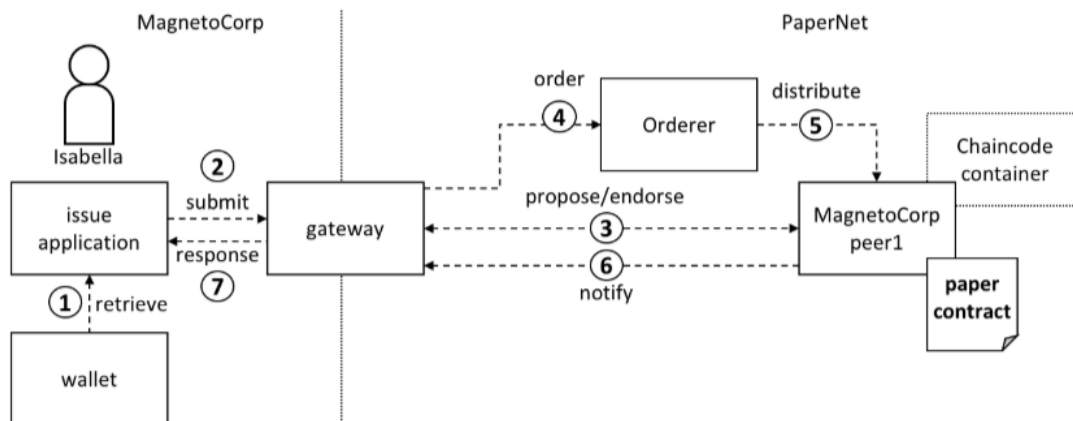
既然我们已经安装并运行了一个基本的 PaperNet,安装并实例化了 PaperContract,那么让我们将注意力转向 Magnetorp 应用程序,它发布了一份商业文件。

8、Application structure

8、应用程序结构

The smart contract contained in papercontract is called by MagnetoCorp's application issue.js. Isabella uses this application to submit a transaction to the ledger which issues commercial paper 00001. Let's quickly examine how the issue application works

papercontrac 中包含的智能合约由 Magnetorp 的 application issue.js 调用。Isabella 使用此应用程序向发出商业票据 00001 的分类帐提交交易。让我们快速检查问题应用程序的工作方式



A gateway allows an application to focus on transaction generation, submission and response. It coordinates transaction proposal, ordering and notification processing between the different network components.

网关允许应用程序专注于事务生成、提交和响应。它协调不同网络组件之间的事务建议、排序和通知处理。

Because the issue application submits transactions on behalf of Isabella, it starts by retrieving Isabella's X.509 certificate from her wallet, which might be stored on the local file system or a Hardware Security Module HSM. The issue application is then able to utilize the gateway to submit transactions on the channel. The Hyperledger Fabric SDK provides a gateway abstraction so that applications can focus on application logic while delegating network interaction to the gateway. Gateways and wallets make it straightforward to write Hyperledger Fabric applications.

因为 Issue 应用程序代表 Isabella 提交交易，所以首先从她的钱包中检索 Isabella 的 X.509 证书，该证书可能存储在本地文件系统或硬件安全模块 hsm 中。然后，问题应用程序能够利用网关在通道上提交事务。Hyperledger Fabric SDK 提供了一个网关抽象，这样应用程序就可以在将网络交互委托给网关的同时专注于应用程序逻辑。网关和钱包使得编写超级账本结构应用程序变得非常简单。

So let's examine the issue application that Isabella is going to use. open a separate terminal window for her, and in fabric-samples locate the MagnetoCorp/application folder:

因此，让我们检查一下 Isabella 将要使用的问题应用程序。为她打开一个单独的终端窗口，在 fabric 样本中找到 Magnetorp/Application 文件夹：

```
(magnetocorp user)$ cd commercial-paper/organization/magnetocorp/application/
(magnetocorp user)$ ls
addToWallet.js    issue.js          package.json
```

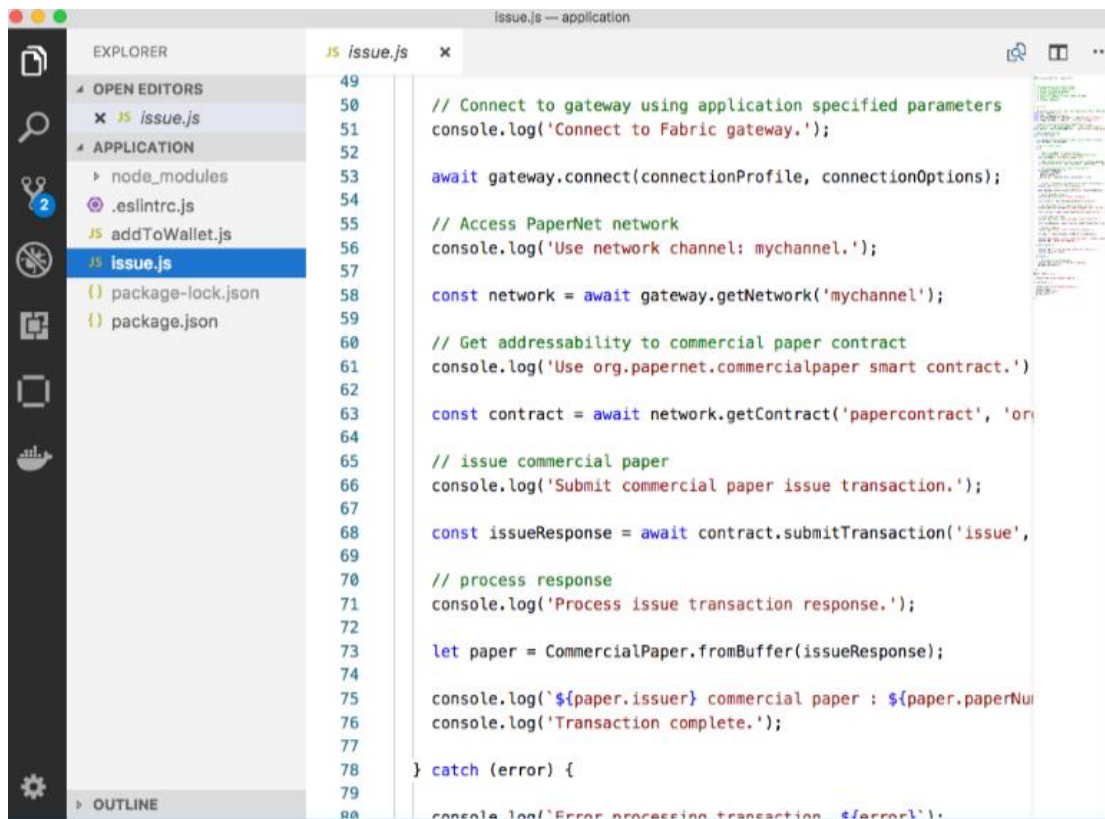
addToWallet.js is the program that Isabella is going to use to load her identity into her wallet, and issue.js will use this identity to create commercial paper 00001 on behalf of MagnetoCorp by invoking papercontract.

addtowallet.js 是 Isabella 将用于将其身份加载到钱包中的程序，issue.js 将使用此身份通过调用 PaperContract 代表 Magnetorp 创建商业票据 00001。

Change to the directory that contains MagnetoCorp's copy of the application issue.js, and use your code editor to examine it:

更改到包含 magedcorp 的 application issue.js 副本的目录，并使用代码编辑器检查它：

```
(magnetocorp user)$ cd commercial-paper/organization/magnetocorp/application
(magnetocorp user)$ code issue.js
```



A code editor displaying the contents of the commercial paper application directory.

显示商业票据应用目录内容的代码编辑器。

Note the following key program lines in issue.js:

请注意 issue.js 中的以下关键程序行：

```
const { FileSystemWallet, Gateway } = require('fabric-network');
```

This statement brings two key Hyperledger Fabric SDK classes into scope - Wallet and Gateway. Because Isabella's X.509 certificate is in the local file system, the application uses FileSystemWallet.

此声明将两个关键的 Hyperledger Fabric SDK 类引入范围-Wallet 和 Gateway。因为 isabella 的 X.509 证书在本地文件系统中，所以应用程序使用 filesystemwallet。

```
const wallet = new FileSystemWallet('../identity/user/isabella/wallet');
```

This statement identifies that the application will use isabella wallet when it connects to the blockchain network channel. The application will select a particular identity within isabella wallet. (The wallet must have been loaded with the Isabella's X.509 certificate - that's what addToWallet.js does.)

此声明确定应用程序在连接到区块链网络通道时将使用 Isabella 钱包。应用程序将在 Isabella 钱包中选择特定的身份。（钱包里一定装了 Isabella 的 X.509 证书——addtowallet.js 就是这么做的。）

```
await gateway.connect(connectionProfile, connectionOptions);
```

This line of code connects to the network using the gateway identified by `connectionProfile`, using the identity referred to in `ConnectionOptions`.

这一行代码使用 `ConnectionProfile` 标识的网关连接到网络，使用 `ConnectionOptions` 中引用的标识。

See how `../gateway/networkConnection.yaml` and `User1@org1.example.com` are used for these values respectively.

请参阅 `../gateway/networkconnection.yaml` 和 `user1@org1.example.com` 如何分别用于这些值。

```
const network = await gateway.getNetwork('mychannel');
```

This connects the application to the network channel `mychannel`, where the `papercontract` was previously instantiated.

这将应用程序连接到网络通道 `mychannel`，之前在那里实例化了 `PaperContract`。

```
const contract = await network.getContract('papercontract',  
'org.papernet.commm...');
```

This statement gives the application addressability to smart contract defined by the namespace `org.papernet.commercialpaper` within `papercontract`. Once an application has issued `getContract`, it can submit any transaction implemented within it.

此语句使应用程序能够访问 `PaperContract` 中由命名空间 `org.papernet.commercialpaper` 定义的智能合约。一旦应用程序发出 `getcontract`，它可以提交在其中实现的任何事务。

```
const issueResponse = await contract.submitTransaction('issue',  
'Magnetocorp', '00001'...);
```

This line of code submits the a transaction to the network using the `issue` transaction defined within the smart contract. `Magnetocorp`, `00001...` are the values to be used by the `issue` transaction to create a new commercial paper.

这一行代码使用智能合约中定义的问题事务将事务提交到网络。`Magnetorp`, `00001...`是发行交易用于创建新商业票据的值。

```
let paper = CommercialPaper.fromBuffer(issueResponse);
```

This statement processes the response from the `issue` transaction. The response needs to be deserialized from a buffer into `paper`, a `CommercialPaper` object which can be interpreted correctly by the application.

此语句处理来自问题事务的响应。响应需要从一个缓冲区反序列化为 `Paper`，这是一个可以被应用程序正确解释的商业纸对象。

Feel free to examine other files in the `/application` directory to understand how `issue.js` works, and read in detail how it is implemented in the application topic.

请随意查看 `/application` 目录中的其他文件，了解 `issue.js` 的工作原理，并详细阅读如何在应用程序主题中实现它。

9、Application dependencies

9、应用程序依赖项

The `issue.js` application is written in JavaScript and designed to run in the

node.js environment that acts as a client to the PaperNet network. As is common practice, MagnetoCorp's application is built on many external node packages - to improve quality and speed of development. Consider how `issue.js` includes the `js-yaml` package to process the YAML gateway connection profile, or the `fabric-network` package to access the Gateway and Wallet classes:

`issue.js` 应用程序是用 javascript 编写的，旨在在作为 Papernet 网络客户端的 node.js 环境中运行。与通常的做法一样，Magnetorp 的应用程序建立在许多外部节点包之上，以提高开发的质量和速度。考虑 `issue.js` 如何包含用于处理 yaml 网关连接配置文件的 `js-yaml` 包，或用于访问网关和钱包类的结构网络包：

```
const yaml = require('js-yaml');
const { FileSystemWallet, Gateway } = require('fabric-network');
```

These packages have to be downloaded from npm to the local file system using the `npm install` command. By convention, packages must be installed into an application-relative `/node_modules` directory for use at runtime.

这些包必须使用 `npm` 安装命令从 `npm` 下载到本地文件系统。按照惯例，包必须安装到应用程序相对/节点模块目录中，以便在运行时使用。

Examine the `package.json` file to see how `issue.js` identifies the packages to download and their exact versions:

检查 `package.json` 文件，查看 `issue.js` 如何识别要下载的包及其确切版本：

```
"dependencies": {
  "fabric-network": "^1.4.0-beta",
  "fabric-client": "^1.4.0-beta",
  "js-yaml": "^3.12.0"
},
```

npm versioning is very powerful; you can read more about it [here](#).

NPM 版本控制非常强大；您可以在[这里](#)阅读更多有关它的信息。

Let's install these packages with the `npm install` command - this may take up to a minute to complete:

让我们用 `npm install` 命令安装这些包——这可能需要一分钟的时间来完成：

```
(magnetocorp user)$ npm install
( ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
```

```
added 738 packages in 46.701s
```

See how this command has updated the directory:

查看此命令如何更新目录：

```
(magnetocorp user)$ ls
addToWallet.js node_modules package.json
issue.js package-lock.json
```

Examine the `node_modules` directory to see the packages that have been installed. There are lots, because `js-yaml` and `fabric-network` are themselves built on other npm packages! Helpfully, the `package-lock.json` file identifies the exact versions installed, which can prove invaluable if you want to exactly reproduce environments; to test, diagnose problems or deliver proven applications for example.

检查 `node_modules` 目录以查看已安装的包。有很多，因为 JS Yaml 和 Fabric Network 本身就是建立在其他 NPM 包之上的！有帮助的是，`package-lock.json` 文件可以识别安装的确切版本，如果您想精确地重现环境，那么可以证明这是非常宝贵的；例如，测试、诊断问题或交付经验证的应用程序。

10、Wallet

10、钱包

Isabella is almost ready to run `issue.js` to issue MagnetoCorp commercial paper 00001; there's just one remaining task to perform! As `issue.js` acts on behalf of Isabella, and therefore MagnetoCorp, it will use identity from her wallet that reflects these facts. We now need to perform this one-time activity of adding appropriate X.509 credentials to her wallet.

Isabella 已经准备好运行 `issue.js` 来发行 Magnetorp 商业票据 00001；还有一个任务要完成！由于 `issue.js` 代表 Isabella，因此 Magnetorp 将使用她钱包中反映这些事实的身份。我们现在需要执行这个一次性活动，将适当的 X.509 凭证添加到她的钱包中。

In Isabella's terminal window, run the `addToWallet.js` program to add identity information to her wallet:

在 Isabella 的终端窗口中，运行 `add to wallet.js` 程序将身份信息添加到她的钱包中：

```
(isabella)$ node addToWallet.js
done
完成
```

Isabella can store multiple identities in her wallet, though in our example, she only uses one - `User1@org.example.com`. This identity is currently associated with the basic network, rather than a more realistic PaperNet configuration - we'll update this tutorial soon.

Isabella 可以在钱包中存储多个身份信息，但在我们的示例中，她只使用一个 - `user1@org.example.com`。这个身份目前与基本网络相关联，而不是更现实的 Papernet 配置——我们将很快更新本教程。

`addToWallet.js` is a simple file-copying program which you can examine at your leisure. It moves an identity from the basic network sample to Isabella's wallet. Let's focus on the result of this program - the contents of the wallet which will be used to submit transactions to PaperNet:

`addtowallet.js` 是一个简单的文件复制程序，您可以在空闲时检查它。它将身份信息从基本网络样本移动到 Isabella 的钱包中。让我们集中讨论这个计划的结果——钱包的内容，它将用于向 Papernet 提交交易：

```
(isabella)$ ls ../identity/user/isabella/wallet/
User1@org1.example.com
```

See how the directory structure maps the `User1@org1.example.com` identity - other identities used by Isabella would have their own folder. Within this directory you'll find the identity information that `issue.js` will use on behalf of isabella:

查看目录结构如何映射 user1@org1.example.com 标识 - Isabella 使用的其他标识将拥有自己的文件夹。在这个目录中，您可以找到 issue.js 将代表 Isabella 使用的身份信息：

```
(isabella)$ ls ../identity/user/isabella/wallet/User1@org1.example.com
User1@org1.example.com c75bd6911a...-priv c75bd6911a...-pub
```

Notice:

注意事项：

a private key c75bd6911a...-priv used to sign transactions on Isabella's behalf, but not distributed outside of her immediate control.

一把私人钥匙 c75bd6911a...-priv 曾代表 Isabella 签署交易，但未在她直接控制范围之外分发。

a public key c75bd6911a...-pub which is cryptographically linked to Isabella's private key. This is wholly contained within Isabella's X.509 certificate.

一个公钥 c75bd6911a...-pub, 通过密码与 Isabella 的私钥相连。这完全包含在 Isabella 的 X.509 证书中。

a certificate User1@org.example.com which contains Isabella's public key and other X.509 attributes added by the Certificate Authority at certificate creation. This certificate is distributed to the network so that different actors at different times can cryptographically verify information created by Isabella's private key.

证书 user1@org.example.com, 其中包含 Isabella 的公钥和证书颁发机构在创建证书时添加的其他 X.509 属性。这个证书被分发到网络，这样不同的参与者在不同的时间可以用密码验证 Isabella 的私钥创建的信息。

Learn more about certificates here. In practice, the certificate file also contains some Fabric-specific metadata such as Isabella's organization and role - read more in the wallet topic.

在此了解有关证书的更多信息。在实践中，证书文件还包含一些特定于结构的元数据，如 Isabella 的组织 and 角色-在钱包主题中了解更多信息。

11、Issue application

11、发行申请

Isabella can now use issue.js to submit a transaction that will issue MagnetoCorp commercial paper 00001:

Isabella 现在可以使用 issue.js 提交将发行 Magnetorp 商业票据 00001 的交易：

```
(isabella)$ node issue.js
Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper issue transaction.
Process issue transaction response.
MagnetoCorp commercial paper : 00001 successfully issued for value 5000000

Transaction complete.
```


Disconnect from Fabric gateway.

Issue program complete.

The node command initializes a node.js environment, and runs issue.js. We can see from the program output that MagnetoCorp commercial paper 00001 was issued with a face value of 5M USD.

node 命令初始化 node.js 环境，并运行 issue.js。从项目产出来看，磁电机公司商业票据 00001 的面值为 500 万美元。

As you've seen, to achieve this, the application invokes the issue transaction defined in the CommercialPaper smart contract within papercontract.js. This had been installed and instantiated in the network by the MagnetoCorp administrator. It's the smart contract which interacts with the ledger via the Fabric APIs, most notably putState() and getState(), to represent the new commercial paper as a vector state within the world state. We'll see how this vector state is subsequently manipulated by the buy and redeem transactions also defined within the smart contract.

如您所见，为了实现这一点，应用程序调用 papercontract.js 中商业票据智能合约中定义的问题事务。这是由磁电机公司管理员在网络中安装和实例化的。它是智能合约，通过结构 API 与分类账交互，最显著的是 putstate() 和 getstate()，将新商业票据表示为世界状态中的向量状态。我们将看到这个向量状态随后如何被智能合约中定义的买入和赎回交易所操纵。

All the time, the underlying Fabric SDK handles the transaction endorsement, ordering and notification process, making the application's logic straightforward; the SDK uses a gateway to abstract away network details and connectionOptions to declare more advanced processing strategies such as transaction retry.

一直以来，底层的 Fabric SDK 处理事务认可、排序和通知过程，使应用程序的逻辑更简单；SDK 使用网关抽象网络详细信息和连接选项，以声明更高级的处理策略，如事务重试。

Let's now follow the lifecycle of MagnetoCorp 00001 by switching our emphasis to DigiBank, who will buy the commercial paper.

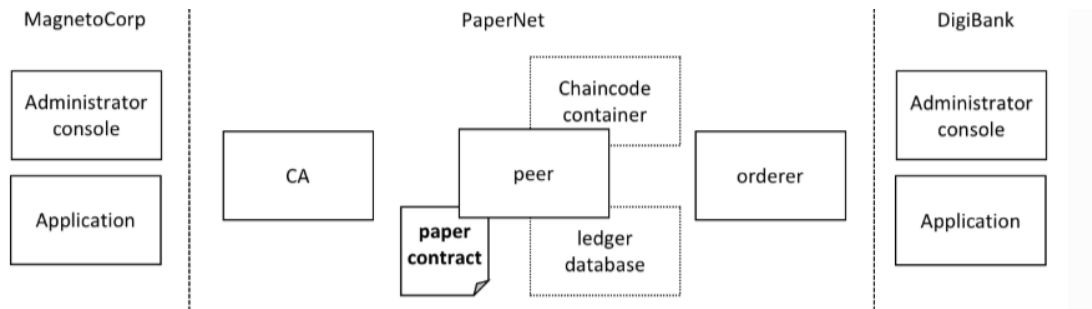
现在让我们跟随磁电机公司 00001 的生命周期，将我们的重点转移到 Digibank，他将购买商业票据。

12、Working as DigiBank

12、作为 Digibank 工作

Now that commercial paper 00001 has been issued by MagnetoCorp, let's switch context to interact with PaperNet as employees of DigiBank. First, we'll act as administrator who will create a console configured to interact with PaperNet. Then Balaji, an end user, will use Digibank's buy application to buy commercial paper 00001, moving it to the next stage in its lifecycle.

既然商业票据 00001 是由 MagnetoCorp 公司发行的，那么让我们作为 Digibank 的员工，切换上下文与 Papernet 进行交互。首先，我们将充当管理员，创建一个配置为与 Papernet 交互的控制台。然后，最终用户 Balaji 将使用 Digibank 的 Buy 应用程序购买商业票据 00001，将其移动到其生命周期的下一个阶段。



DigiBank administrators and applications interact with the PaperNet network. DigiBank 管理员和应用程序与 Papernet 网络交互。

As the tutorial currently uses the basic network for PaperNet, the network configuration is quite simple. Administrators use a console similar to MagnetoCorp, but configured for Digibank's file system. Likewise, Digibank end users will use applications which invoke the same smart contract as MagnetoCorp applications, though they contain Digibank-specific logic and configuration. It's the smart contract which captures the shared business process, and the ledger which holds the shared business data, no matter which applications call them.

由于本教程目前使用的是 Papernet 的基本网络，因此网络配置非常简单。管理员使用类似于 Magnetorp 的控制台，但配置为 Digibank 的文件系统。同样，Digibank 终端用户也将使用与 Magnetorp 应用程序调用相同智能合约的应用程序，尽管它们包含 Digibank 特定的逻辑和配置。智能合约捕获共享的业务流程，而分类账则保存共享的业务数据，不管哪个应用程序调用它们。

Let's open up a separate terminal to allow the DigiBank administrator to interact with PaperNet. In fabric-samples:

```
让我们打开一个单独的终端,让 Digibank 管理员与 Papernet 交互。在 fabric-samples:
(digibank admin)$ cd commercial-paper/organization/digibank/configuration/cli/
(digibank admin)$ docker-compose -f docker-compose.yml up -d cliDigiBank
```

```
(...)
Creating cliDigiBank ... done
```

This docker container is now available for Digibank administrators to interact with the network:

现在 Digibank 管理员可以使用此 Docker 容器与网络进行交互:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORT	NAMES
858c2d2961d4	hyperledger/fabric-tools	"/bin/bash"	seconds ago	Up 18 seconds		cliDigiBank

In this tutorial, you'll use the command line container named cliDigiBank to interact with the network on behalf of DigiBank. We've not shown all the docker containers, and in the real world DigiBank users would only see the network components (peers, orderers, CAs) to which they have access.

在本教程中，您将使用名为 clidigibank 的命令行容器代表 digibank 与网络交互。我们没有显示所有的 Docker 容器，在现实世界中，Digibank 用户只会看到他们可以访问的网

络组件（对等、订购者、CA）。

Digibank's administrator doesn't have much to do in this tutorial right now because the PaperNet network configuration is so simple. Let's turn our attention to Balaji.

Digibank 的管理员在本教程中没有太多工作要做，因为 PaperNet 网络配置非常简单。让我们把注意力转向 Balaji。

13、Digibank applications

13、Digibank 应用程序

Balaji uses DigiBank's buy application to submit a transaction to the ledger which transfers ownership of commercial paper 00001 from MagnetoCorp to DigiBank. The CommercialPaper smart contract is the same as that used by MagnetoCorp's application, however the transaction is different this time - it's buy rather than issue. Let's examine how DigiBank's application works.

Balaji 使用 Digibank 的购买申请向分类账提交交易，分类账将商业票据 00001 的所有权从磁电机公司转移到 Digibank。商业票据智能合约与磁电机公司的应用程序使用的是相同的，但是这次交易是不同的——它是购买而不是发行。让我们检查一下 Digibank 的应用程序是如何工作的。

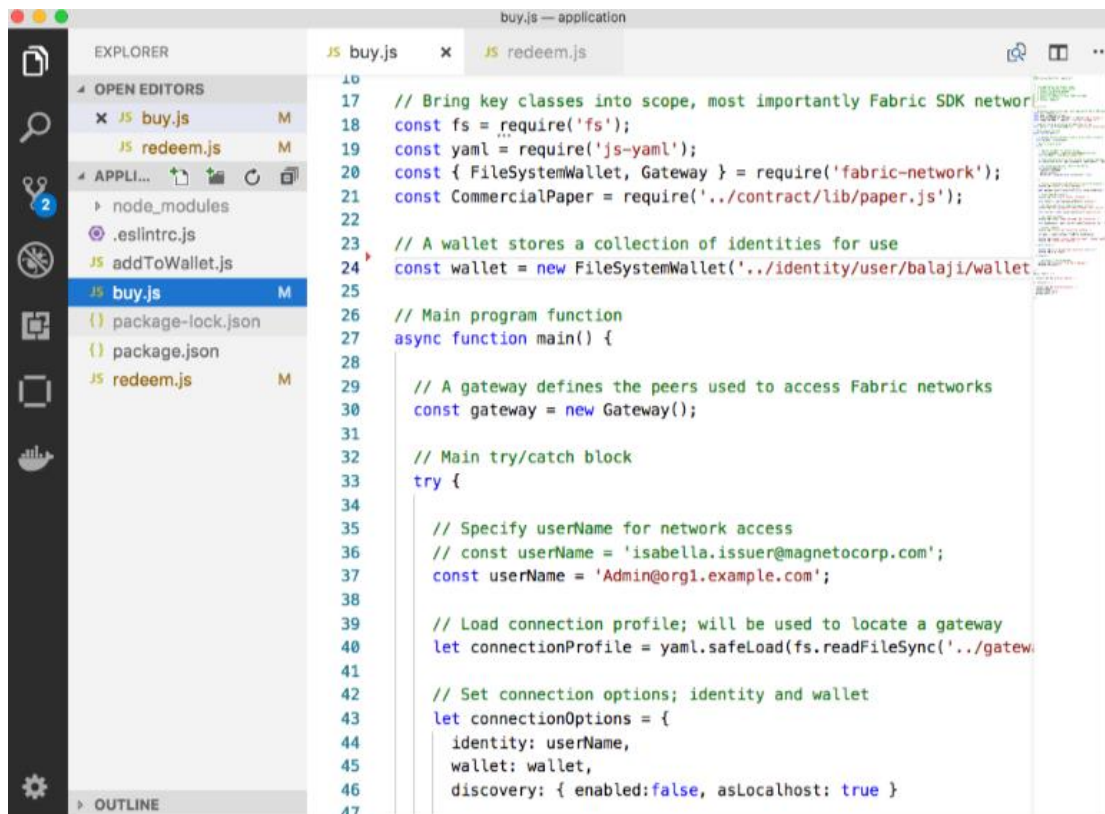
Open a separate terminal window for Balaji. In fabric-samples, change to the DigiBank application directory that contains the application, buy.js, and open it with your editor:

打开一个单独的巴拉吉终端窗口。在 Fabric 示例中，更改到包含该应用程序的 digibank 应用程序目录 buy.js，然后用编辑器打开它：

```
(balaji)$ cd commercial-paper/organization/digibank/application/  
(balaji)$ code buy.js
```

As you can see, this directory contains both the buy and redeem applications that will be used by Balaji.

如您所见，此目录包含 Balaji 将使用的购买和兑换应用程序。



DigiBank's commercial paper directory containing the buy.js and redeem.js applications.

Digibank 的商业票据目录，包含 buy.js 和 reduce.js 应用程序。

DigiBank's buy.js application is very similar in structure to MagnetoCorp's issue.js with two important differences:

Digibank 的 buy.js 应用程序在结构上与 Magnetorp 的 issue.js 非常相似，有两个重要区别：

Identity: the user is a DigiBank user Balaji rather than MagnetoCorp's Isabella

身份：用户是 Digibank 用户 Balaji，而不是 Magnetorp 的 Isabella。

```
const wallet = new FileSystemWallet('../identity/user/balaji/wallet');
```

See how the application uses the balaji wallet when it connects to the PaperNet network channel. buy.js selects a particular identity within balaji wallet.

查看应用程序在连接到 Papernet 网络通道时如何使用 Balaji 钱包。buy.js 在 Balaji 钱包中选择一个特定的身份。

Transaction: the invoked transaction is buy rather than issue

事务：调用的事务是购买而不是发行

```
`const buyResponse = await contract.submitTransaction('buy', 'MagnetoCorp', '00001'...);`
```

A buy transaction is submitted with the values MagnetoCorp, 00001..., that are used by the CommercialPaper smart contract class to transfer ownership of commercial paper 00001 to DigiBank.

商业票据智能合约类使用价值磁电机公司 (00001...) 向 Digibank 转让商业票据 00001 的所有权。

Feel free to examine other files in the application directory to understand how the application works, and read in detail how buy.js is implemented in the application topic.

请随意查看应用程序目录中的其他文件，了解应用程序的工作方式，并详细阅读 buy.js 在应用程序主题中的实现方式。

14、Run as DigiBank

14、作为数字银行运行

The DigiBank applications which buy and redeem commercial paper have a very similar structure to MagnetoCorp's issue application. Therefore, let's install their dependencies and set up Balaji's wallet so that he can use these applications to buy and redeem commercial paper.

购买和赎回商业票据的 Digibank 应用程序的结构与 Magnetorp 的发行应用程序非常相似。因此，让我们安装它们的依赖项并设置 Balaji 的钱包，以便他可以使用这些应用程序购买和兑换商业票据。

Like MagnetoCorp, Digibank must the install the required application packages using the npm install command, and again, this make take a short time to complete.

与 Magnetorp 一样，Digibank 必须使用 npm install 命令安装所需的应用程序包，而且这需要很短的时间才能完成。

In the DigiBank administrator window, install the application dependencies:
在 Digibank 管理员窗口中，安装应用程序依赖项：

```
(digibank admin)$ cd commercial-paper/organization/digibank/application/  
(digibank admin)$ npm install  
( ) extract:lodash: sill extract ansi-styles@3.2.1  
(...)  
added 738 packages in 46.701s
```

In Balaji's terminal window, run the addToWallet.js program to add identity information to his wallet:

在 Balaji 的终端窗口中，运行 add to wallet.js 程序将身份信息添加到他的钱包中：

```
(balaji)$ node addToWallet.js  
done
```

The addToWallet.js program has added identity information for balaji, to his wallet, which will be used by buy.js and redeem.js to submit transactions to PaperNet.

addtowallet.js 程序已将 Balaji 的身份信息添加到他的钱包中，buy.js 和 receive.js 将使用该钱包向 Papernet 提交交易。

Like Isabella, Balaji can store multiple identities in his wallet, though in our example, he only uses one - Admin@org.example.com. His corresponding wallet structure digibank/identity/user/balaji/wallet/Admin@org1.example.com contains is very similar Isabella's - feel free to examine it.

像 Isabella 一样，巴拉吉可以在他的钱包中存储多个身份信息，但在我们的示例中，

他只使用一个 - admin@org.example.com 。他相应的钱包结构 digibank/identity/user/balaji/wallet/admin@org1.example.com 包含的内容与 isabella 非常相似 - 请随意查看。

15、Buy application

15、购买应用

Balaji can now use buy.js to submit a transaction that will transfer ownership of MagnetoCorp commercial paper 00001 to DigiBank.

Balaji 现在可以使用 Buy.js 提交一份交易，将 Magnetorp 商业票据 00001 的所有权转让给 Digibank。

Run the buy application in Balaji's window:

在 Balaji 的窗口中运行 Buy 应用程序：

```
(balaji)$ node buy.js
Connect to Fabric gateway.
Use network channel: mychannel.
Use org.paper.net.commercialpaper smart contract.
Submit commercial paper buy transaction.
Process buy transaction response.
MagnetoCorp commercial paper : 00001 successfully purchased by DigiBank
Transaction complete.
Disconnect from Fabric gateway.
Buy program complete.
```

You can see the program output that MagnetoCorp commercial paper 00001 was successfully purchased by Balaji on behalf of DigiBank. buy.js invoked the buy transaction defined in the CommercialPaper smart contract which updated commercial paper 00001 within the world state using the putState() and getState() Fabric APIs. As you've seen, the application logic to buy and issue commercial paper is very similar, as is the smart contract logic.

您可以看到磁电机公司商业票据 00001 由巴拉吉代表 Digibank 成功购买的程序输出。buy.js 调用了商业票据智能合约中定义的 buy 事务，它使用 putstate() 和 getstate() 结构 API 更新了世界状态下的商业票据 00001。如您所见，购买和发行商业票据的应用程序逻辑与智能合约逻辑非常相似。

15、Redeem application

15、兑换申请

The final transaction in the lifecycle of commercial paper 00001 is for DigiBank to redeem it with MagnetoCorp. Balaji uses redeem.js to submit a transaction to perform the redeem logic within the smart contract.

商业票据 00001 生命周期中的最后一笔交易是 Digibank 用 Magnetorp 赎回它。Balaji 使用 reduce.js 提交事务以执行智能合约中的兑现逻辑。

Run the redeem transaction in Balaji's window:

在 Balaji 的窗口中运行兑换交易：

```
(balaji)$ node redeem.js
Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper redeem transaction.
Process redeem transaction response.
MagnetoCorp commercial paper : 00001 successfully redeemed with MagnetoCorp
Transaction complete.
Disconnect from Fabric gateway.
Redeem program complete.
```

Again, see how the commercial paper 00001 was successfully redeemed when `redeem.js` invoked the `redeem` transaction defined in `CommercialPaper`. Again, it updated commercial paper 00001 within the world state to reflect that the ownership returned to MagnetoCorp, the issuer of the paper.

同样，请看一下当 `ecrease.js` 调用商业票据中定义的兑换交易时，商业票据 00001 是如何成功兑换的。它再次更新了世界范围内的商业票据 00001，以反映该票据的发行人磁电机公司的所有权。

16、Further reading

16、进一步阅读

To understand how applications and smart contracts shown in this tutorial work in more detail, you'll find it helpful to read *Developing Applications*. This topic will give you a fuller explanation of the commercial paper scenario, the PaperNet business network, its actors, and how the applications and smart contracts they use work in detail.

要了解本教程中显示的应用程序和智能合约如何更详细地工作，您将发现阅读开发应用程序很有帮助。本主题将详细介绍商业票据场景、Papernet 业务网络及其参与者，以及它们使用的应用程序和智能合约如何工作。

Also feel free to use this sample to start creating your own applications and smart contracts!

也可以随意使用此示例开始创建自己的应用程序和智能合约！

三、Building Your First Network

三、构建你的第一个网络

Note

注释

These instructions have been verified to work against the latest stable Docker images and the pre-compiled setup utilities within the supplied tar file. If you run these commands with images or tools from the current master branch, it is possible that you will see configuration and panic errors.

这些说明已经过验证，可以与所提供的 tar 文件中最新的稳定 Docker 映像和预编译的安装实用程序进行比较。如果使用来自当前主分支的图像或工具运行这些命令，则可能会看到配

置和死机错误。

The build your first network (BYFN) scenario provisions a sample Hyperledger Fabric network consisting of two organizations, each maintaining two peer nodes, and a “solo” ordering service.

构建您的第一个网络(byfn)场景提供了一个由两个组织组成的样本超级账本结构网络，每个组织维护两个对等节点，以及一个“单独”订购服务。

1、Install prerequisites

1、安装必备组件

Before we begin, if you haven't already done so, you may wish to check that you have all the Prerequisites installed on the platform(s) on which you'll be developing blockchain applications and/or operating Hyperledger Fabric.

在我们开始之前，如果您还没有这样做，您可能希望检查您是否在开发区块链应用程序和/或操作超级账本结构的平台上安装了所有先决条件。

You will also need to Install Samples, Binaries and Docker Images. You will notice that there are a number of samples included in the fabric-samples repository. We will be using the first-network sample. Let's open that sub-directory now.

您还需要安装示例、二进制文件和 Docker 映像。您将注意到 Fabric 示例存储库中包含许多示例。我们将使用第一个网络示例。现在我们打开那个子目录。

```
cd fabric-samples/first-network
```

Note

注释

The supplied commands in this documentation MUST be run from your first-network sub-directory of the fabric-samples repository clone. If you elect to run the commands from a different location, the various provided scripts will be unable to find the binaries.

此文档中提供的命令必须从结构示例存储库克隆的第一个网络子目录运行。如果选择从其他位置运行命令，则提供的各种脚本将无法找到二进制文件。

2、Want to run it now?

2、想现在运行吗？

We provide a fully annotated script - byfn.sh - that leverages these Docker images to quickly bootstrap a Hyperledger Fabric network comprised of 4 peers representing two different organizations, and an orderer node. It will also launch a container to run a scripted execution that will join peers to a channel, deploy and instantiate chaincode and drive execution of transactions against the deployed chaincode.

我们提供了一个完全注释的脚本 (byfn.sh)，它利用这些 Docker 图像快速引导一个由代表两个不同组织的 4 个对等方和一个订购方节点组成的超级账本结构网络。它还将启动一个容器来运行一个脚本化的执行，该执行将连接到一个通道的对等方，部署和实例化链码，并根据部署的链码驱动事务的执行。

Here's the help text for the byfn.sh script:

下面是 byfn.sh 脚本的帮助文本:

Usage:

```
byfn.sh <mode> [-c <channel name>] [-t <timeout>] [-d <delay>] [-f <docker-  
compose-file>] [-s <dbtype>] [-l <language>] [-i <imagetag>] [-v]  
<mode> - one of 'up', 'down', 'restart', 'generate' or 'upgrade'  
- 'up' - bring up the network with docker-compose up  
- 'down' - clear the network with docker-compose down  
- 'restart' - restart the network  
- 'generate' - generate required certificates and genesis block  
- 'upgrade' - upgrade the network from v1.0.x to v1.1  
-c <channel name> - channel name to use (defaults to "mychannel")  
-t <timeout> - CLI timeout duration in seconds (defaults to 10)  
-d <delay> - delay duration in seconds (defaults to 3)  
-f <docker-compose-file> - specify which docker-compose file use (defaults  
to docker-compose-cli.yaml)
```

```
-s <dbtype> - the database backend to use: goleveldb (default) or couchdb
```

```
-l <language> - the chaincode language: golang (default), node or java
```

```
-i <imagetag> - the tag to be used to launch the network (defaults to "latest")
```

```
-v - verbose mode
```

```
byfn.sh -h (print this message)
```

Typically, one would first generate the required certificates and genesis block, then bring up the network. e.g.:

```
byfn.sh generate -c mychannel
```

```
byfn.sh up -c mychannel -s couchdb
```

```
byfn.sh up -c mychannel -s couchdb -i 1.1.0-alpha
```

```
byfn.sh up -l node
```

```
byfn.sh down -c mychannel
```

```
byfn.sh upgrade -c mychannel
```

Taking all defaults:

```
byfn.sh generate
```

```
byfn.sh up
```

```
byfn.sh down
```

If you choose not to supply a channel name, then the script will use a default name of mychannel. The CLI timeout parameter (specified with the -t flag) is an optional value; if you choose not to set it, then the CLI will give up on query requests made after the default setting of 10 seconds.

如果选择不提供频道名称, 则脚本将使用默认名称 mychannel。cli timeout 参数 (用 -t 标志指定) 是一个可选值; 如果选择不设置该参数, 则在默认设置为 10 秒后, cli 将放弃查询请求。

Generate Network Artifacts

生成网络项目

Ready to give it a go? Okay then! Execute the following command:

准备好了吗？那么好吧！执行以下命令：

```
./byfn.sh generate
```

You will see a brief description as to what will occur, along with a yes/no command line prompt. Respond with a y or hit the return key to execute the described action.

您将看到一个关于将发生什么的简短描述，以及一个是/否命令行提示。用 Y 键响应或按返回键执行所描述的操作。

```
Generating certs and genesis block for with channel 'mychannel' and CLI timeout of '10'
Continue? [Y/n] y
proceeding ...
/Users/xxx/dev/fabric-samples/bin/cryptogen

#####
#### Generate certificates using cryptogen tool #####
#####
org1.example.com
2017-06-12 21:01:37.334 EDT [bccsp] GetDefault -> WARN 001 Before using BCCSP, please call InitFact
...

/Users/xxx/dev/fabric-samples/bin/configtxgen
#####
##### Generating Orderer Genesis block #####
#####
2017-06-12 21:01:37.558 EDT [common/configtx/tool] main -> INFO 001 Loading configuration
2017-06-12 21:01:37.562 EDT [msp] getMspConfig -> INFO 002 intermediate certs folder not found at [
...
2017-06-12 21:01:37.588 EDT [common/configtx/tool] doOutputBlock -> INFO 00b Generating genesis blo
2017-06-12 21:01:37.590 EDT [common/configtx/tool] doOutputBlock -> INFO 00c Writing genesis block

#####
### Generating channel configuration transaction 'channel.tx' ###
#####
2017-06-12 21:01:37.634 EDT [common/configtx/tool] main -> INFO 001 Loading configuration
2017-06-12 21:01:37.644 EDT [common/configtx/tool] doOutputChannelCreateTx -> INFO 002 Generating n
2017-06-12 21:01:37.645 EDT [common/configtx/tool] doOutputChannelCreateTx -> INFO 003 Writing new

#####
##### Generating anchor peer update for Org1MSP #####
#####
2017-06-12 21:01:37.674 EDT [common/configtx/tool] main -> INFO 001 Loading configuration
2017-06-12 21:01:37.678 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 002 Generating
2017-06-12 21:01:37.679 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 003 Writing an

#####
##### Generating anchor peer update for Org2MSP #####
#####
2017-06-12 21:01:37.700 EDT [common/configtx/tool] main -> INFO 001 Loading configuration
2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 002 Generating
2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 003 Writing an
```

This first step generates all of the certificates and keys for our various network entities, the genesis block used to bootstrap the ordering service, and a collection of configuration transactions required to configure a Channel.

第一步为我们的各种网络实体生成所有证书和密钥，用于引导订购服务的 Genesis 块，以及配置通道所需的配置交易集合。

Bring Up the Network

打开网络

Next, you can bring the network up with one of the following commands:

接下来，您可以使用以下命令之一启动网络：

```
./byfn.sh up
```

The above command will compile Golang chaincode images and spin up the corresponding containers. Go is the default chaincode language, however there is also support for Node.js and Java chaincode. If you'd like to run through this tutorial with node chaincode, pass the following command instead:

上面的命令将编译 golang 链码图像并旋转相应的容器。GO 是默认的链接代码语言，但是也支持 Node.js 和 Java 链接代码。如果要使用节点链代码运行本教程，请改为传递以下命令：

```
# we use the -l flag to specify the chaincode language
```

我们使用-l 标志来指定链码语言

```
# forgoing the -l flag will default to Golang
```

放弃-l 标志将默认为 golang

```
./byfn.sh up -l node
```

Note

注释

For more information on the Node.js shim, please refer to its documentation.

有关 node.js 填充程序的更多信息，请参阅其文档。

Note

注释

For more information on the Java shim, please refer to its documentation.

有关 Java SIMM 的更多信息，请参阅其文档。

To make the sample run with Java chaincode, you have to specify -l java as follows:

使示例用 Java 链代码运行，您必须指定-l Java 如下：

```
./byfn.sh up -l java
```

Note

注释

Do not run both of these commands. Only one language can be tried unless you bring down and recreate the network between.

不要同时运行这两个命令。只有一种语言可以尝试，除非您关闭并重新创建之间的网络。

Once again, you will be prompted as to whether you wish to continue or abort. Respond with a y or hit the return key:

再次提示您是否要继续或中止。用 Y 键响应或按返回键：

```
Starting with channel 'mychannel' and CLI timeout of '10'
Continue? [Y/n]
proceeding ...
Creating network "net_byfn" with the default driver
Creating peer0.org1.example.com
Creating peer1.org1.example.com
Creating peer0.org2.example.com
Creating orderer.example.com
Creating peer1.org2.example.com
Creating cli
```

START

```
Channel name : mychannel
Creating channel...
```

The logs will continue from there. This will launch all of the containers, and then drive a complete end-to-end application scenario. Upon successful completion, it should report the following in your terminal window:

日志将从那里继续。这将启动所有容器，然后驱动一个完整的端到端应用程序场景。成功完成后，应在终端窗口中报告以下内容：

```
Query Result: 90
2017-05-16 17:08:15.158 UTC [main] main -> INFO 008 Exiting....
===== Query successful on peer1.org2 on channel 'mychannel' =====
===== All GOOD, BYFN execution completed =====
```

END

You can scroll through these logs to see the various transactions. If you don't get this result, then jump down to the Troubleshooting section and let's see whether we can help you discover what went wrong.

您可以滚动浏览这些日志以查看各种事务。如果您没有得到这个结果，那么跳到故障排除部分，让我们看看是否可以帮助您发现哪里出了问题。

Bring Down the Network

关闭网络

Finally, let's bring it all down so we can explore the network setup one step at a time. The following will kill your containers, remove the crypto material and four artifacts, and delete the chaincode images from your Docker Registry:

最后，让我们把它全部放下，这样我们可以一步一步地探索网络设置。以下操作将杀死您的容器，删除加密材料和四个工件，并从 Docker 注册表中删除链码图像：

```
./byfn.sh down
```

Once again, you will be prompted to continue, respond with a y or hit the return key:

再次，系统将提示您继续、用 y 响应或按回车键：

```
Stopping with channel 'mychannel' and CLI timeout of '10'
Continue? [Y/n] y
proceeding ...
WARNING: The CHANNEL_NAME variable is not set. Defaulting to a blank string.
WARNING: The TIMEOUT variable is not set. Defaulting to a blank string.
Removing network net_byfn
468aaa6201ed
...
Untagged: dev-peer1.org2.example.com-mycc-1.0:latest
Deleted: sha256:ed3230614e64e1c83e510c0c282e982d2b06d148b1c498bbdcc429e2b2531e91
...
```

If you'd like to learn more about the underlying tooling and bootstrap mechanics, continue reading. In these next sections we'll walk through the various steps and requirements to build a fully-functional Hyperledger Fabric network.

如果您想了解更多关于底层工具和引导程序机制的信息，请继续阅读。在下一节中，我们将介绍构建一个功能全面的 Hyperledger 结构网络的各种步骤和要求。

Note

注释

The manual steps outlined below assume that the FABRIC_LOGGING_SPEC in the cli container is set to DEBUG. You can set this by modifying the docker-compose-cli.yaml file in the first-network directory. e.g.

下面概述的手动步骤假定 cli 容器中的 fabric_logging_spec 设置为 debug。您可以通过修改 first-network 目录中的 docker-compose-cli.yaml 文件来设置它。例如

```
cli:
  container_name: cli
  image: hyperledger/fabric-tools:$IMAGE_TAG
  tty: true
  stdin_open: true
  environment:
    - GOPATH=/opt/gopath
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - FABRIC_LOGGING_SPEC=DEBUG
    #- FABRIC_LOGGING_SPEC=INFO
```

3、Crypto Generator

3、加密生成器

We will use the cryptogen tool to generate the cryptographic material (x509 certs and signing keys) for our various network entities. These certificates are representative of identities, and they allow for sign/verify authentication to take place as our entities communicate and transact.

我们将使用 Crypton 工具为各种网络实体生成加密材料（X509 证书和签名密钥）。这些证书代表身份，它们允许在我们的实体进行通信和交易时进行签名/验证身份验证。

How does it work?

它是如何工作的？

Cryptogen consumes a file - `crypto-config.yaml` - that contains the network topology and allows us to generate a set of certificates and keys for both the Organizations and the components that belong to those Organizations. Each Organization is provisioned a unique root certificate (`ca-cert`) that binds specific components (peers and orderers) to that Org. By assigning each Organization a unique CA certificate, we are mimicking a typical network where a participating Member would use its own Certificate Authority. Transactions and communications within Hyperledger Fabric are signed by an entity's private key (keystore), and then verified by means of a public key (signcerts).

Cryptogen 使用一个包含网络拓扑结构的文件 `crypto-config.yaml`, 它允许我们为组织和属于这些组织的组件生成一组证书和密钥。每个组织都有一个唯一的根证书 (CA 证书), 它将特定组件 (对等方和订购方) 绑定到该组织。通过为每个组织分配一个唯一的 CA 证书, 我们模拟了一个典型的网络, 其中参与成员将使用自己的证书颁发机构。Hyperledger 结构中的交易和通信由实体的私钥 (keystore) 签名, 然后通过公钥 (signcerts) 进行验证。

You will notice a count variable within this file. We use this to specify the number of peers per Organization; in our case there are two peers per Org. We won't delve into the minutiae of x.509 certificates and public key infrastructure right now. If you're interested, you can peruse these topics on your own time.

您将注意到这个文件中有一个 count 变量。我们使用它来指定每个组织的对等数; 在我们的例子中, 每个组织有两个对等数。我们现在不会深入研究 X.509 证书和公钥基础设施的细节。如果你感兴趣的话, 你可以在自己的时间阅读这些主题。

Before running the tool, let's take a quick look at a snippet from the `crypto-config.yaml`. Pay specific attention to the "Name", "Domain" and "Specs" parameters under the OrdererOrgs header:

在运行该工具之前, 让我们快速查看 `crypto-config.yaml` 中的一个片段。请特别注意 ordererorgs 头下的 "name"、"domain" 和 "specs" 参数:

```
OrdererOrgs:
#-----
# Orderer
#-----
- Name: Orderer
  Domain: example.com
  CA:
  Country: US
  Province: California
  Locality: San Francisco
# OrganizationalUnit: Hyperledger Fabric
# StreetAddress: address for org # default nil
# PostalCode: postalCode for org # default nil
#-----
# "Specs" - See PeerOrgs below for complete description
#-----
Specs:
```

```

- Hostname: orderer
# -----
# "PeerOrgs" - Definition of organizations managing peer nodes
# -----
PeerOrgs:
# -----
# Org1
# -----
- Name: Org1
Domain: org1.example.com
EnableNodeOUs: true

```

The naming convention for a network entity is as follows - “{.Hostname}}.{.Domain}”. So using our ordering node as a reference point, we are left with an ordering node named - orderer.example.com that is tied to an MSP ID of Orderer. This file contains extensive documentation on the definitions and syntax. You can also refer to the Membership Service Providers (MSP) documentation for a deeper dive on MSP.

网络实体的命名约定如下- “{.hostname}}.{.domain}”。因此，使用我们的订购节点作为参考点，我们留下一个名为-order.example.com的订购节点，它绑定到订购者的MSP ID。此文件包含有关定义和语法的大量文档。您还可以参考会员服务提供商（MSP）文档，进一步了解MSP。

After we run the cryptogen tool, the generated certificates and keys will be saved to a folder titled crypto-config.

运行加密工具后，生成的证书和密钥将保存到名为 crypto-config 的文件夹中。

4、Configuration Transaction Generator

4、配置交易生成器

The configtxgen tool is used to create four configuration artifacts:

configtxgen 工具用于创建四个配置工件：

- orderer genesis block,
- 订购者 Genesis 区块，
- channel configuration transaction,
- 通道配置事务，
- and two anchor peer transactions - one for each Peer Org.
- 以及两个锚定对等事务—每个对等组织一个。

Please see configtxgen for a complete description of this tool's functionality.

有关此工具功能的完整描述，请参阅 configtxgen。

The orderer block is the Genesis Block for the ordering service, and the channel configuration transaction file is broadcast to the orderer at Channel creation time. The anchor peer transactions, as the name might suggest, specify each Org's Anchor Peer on this channel.

订购节点是订购服务的 Genesis 块，通道配置交易文件在通道创建时广播给订购方。锚

定对等事务，顾名思义，在这个通道上指定每个组织的锚定对等。

How does it work?

它是如何工作的？

Configtxgen consumes a file - configtx.yaml - that contains the definitions for the sample network. There are three members - one Orderer Org (OrdererOrg) and two Peer Orgs (Org1 & Org2) each managing and maintaining two peer nodes. This file also specifies a consortium - SampleConsortium - consisting of our two Peer Orgs. Pay specific attention to the “Profiles” section at the top of this file. You will notice that we have two unique headers. One for the orderer genesis block - TwoOrgsOrdererGenesis - and one for our channel - TwoOrgsChannel.

configtxgen 使用一个包含示例网络定义的文件 configtx.yaml。有三个成员-一个订购者组织 (orderorg) 和两个对等组织 (org1 和 org2)，每个成员管理和维护两个对等节点。这个文件还指定了一个联合体-sampleconsortium-由我们的两个对等组织组成。请特别注意此文件顶部的“配置文件”部分。您会注意到我们有两个唯一的标题。一个是订购方 Genesis 区块-两个或多个区域-另一个是我们的频道-两个区域。

These headers are important, as we will pass them in as arguments when we create our artifacts.

这些头很重要，因为我们将创建工件时将它们作为参数传入。

Note

注释

Notice that our SampleConsortium is defined in the system-level profile and then referenced by our channel-level profile. Channels exist within the purview of a consortium, and all consortia must be defined in the scope of the network at large.

请注意，我们的 sampleconsortium 在系统级概要文件中定义，然后由通道级概要文件引用。渠道存在于联合体的权限内，所有联合体必须在整个网络范围内定义。

This file also contains two additional specifications that are worth noting. Firstly, we specify the anchor peers for each Peer Org (peer0.org1.example.com & peer0.org2.example.com). Secondly, we point to the location of the MSP directory for each member, in turn allowing us to store the root certificates for each Org in the orderer genesis block. This is a critical concept. Now any network entity communicating with the ordering service can have its digital signature verified.

该文件还包含两个值得注意的附加规范。首先，我们为每个对等组织 (peer0.org1.example.com 和 peer0.org2.example.com) 指定锚定对等。其次，我们指出每个成员的 msp 目录的位置，从而允许我们将每个组织的根证书存储在 order genesis 块中。这是一个关键的概念。现在，任何与订购服务通信的网络实体都可以验证其数字签名。

5、Run the tools

5、运行工具

You can manually generate the certificates/keys and the various configuration artifacts using the configtxgen and cryptogen commands. Alternately, you could try to adapt the byfn.sh script to accomplish your objectives.

您可以使用 `configtxgen` 和 `cryptogen` 命令手动生成证书/密钥和各种配置工件。或者，您可以尝试调整 `byfn.sh` 脚本以实现您的目标。

Manually generate the artifacts

手动生成工件

You can refer to the `generateCerts` function in the `byfn.sh` script for the commands necessary to generate the certificates that will be used for your network configuration as defined in the `crypto-config.yaml` file. However, for the sake of convenience, we will also provide a reference here.

您可以参考 `byfn.sh` 脚本中的 `generatecerts` 函数，获取生成证书所需的命令，这些证书将用于在 `crypto-config.yaml` 文件中定义的网络配置。然而，为了方便起见，我们也将在这里提供参考。

First let's run the `cryptogen` tool. Our binary is in the `bin` directory, so we need to provide the relative path to where the tool resides.

首先，让我们运行加密工具。我们的二进制文件在 `bin` 目录中，因此我们需要提供工具所在的相对路径。

```
../bin/cryptogen generate --config=./crypto-config.yaml
```

You should see the following in your terminal:

您应该在终端中看到以下内容：

```
org1.example.com
```

```
org2.example.com
```

The certs and keys (i.e. the MSP material) will be output into a directory - `crypto-config` - at the root of the `first-network` directory.

证书和密钥（即 MSP 材料）将输出到第一个网络目录根目录下的目录 `crypto-config` 中。

Next, we need to tell the `configtxgen` tool where to look for the `configtx.yaml` file that it needs to ingest. We will tell it look in our present working directory:

接下来，我们需要告诉 `configtxgen` 工具在哪里查找它需要接收的 `configtx.yaml` 文件。我们将告诉它在我们当前的工作目录中查找：

```
export FABRIC_CFG_PATH=$PWD
```

Then, we'll invoke the `configtxgen` tool to create the orderer genesis block:

然后，我们将调用 `configtxgen` 工具来创建 order genesis 块：

```
../bin/configtxgen -profile TwoOrgsOrdererGenesis -channelID byfn-sys-channel -outputBlock ./channel-artifacts/genesis.block
```

You should see an output similar to the following in your terminal:

您应该在终端中看到类似以下内容的输出：

```
2017-10-26 19:21:56.301 EDT [common/tools/configtxgen] main -> INFO 001
Loading configuration
2017-10-26 19:21:56.309 EDT [common/tools/configtxgen] doOutputBlock -> INFO
002 Generating genesis block
2017-10-26 19:21:56.309 EDT [common/tools/configtxgen] doOutputBlock -> INFO
003 Writing genesis block
```

Note

注意

The orderer genesis block and the subsequent artifacts we are about to create will be output into the channel-artifacts directory at the root of this project. The channelID in the above command is the name of the system channel.

订购者 Genesis 块和我们将要创建的后续工件将输出到这个项目根目录下的 channel artifacts 目录中。上面命令中的 channelId 是系统通道的名称。

Create a Channel Configuration Transaction

创建通道配置交易

Next, we need to create the channel transaction artifact. Be sure to replace \$CHANNEL_NAME or set CHANNEL_NAME as an environment variable that can be used throughout these instructions:

接下来，我们需要创建通道交易工件。请务必替换\$CHANNEL_NAME 或将 CHANNEL_NAME 设置为可在以下说明中使用的环境变量：

```
# The channel.tx artifact contains the definitions for our sample channel
export CHANNEL_NAME=mychannel && ../bin/configtxgen -profile TwoOrgsChannel
-outputCreateChannelTx ./channel-artifacts/channel.tx -channelID $CHANNEL_NAME
```

You should see an output similar to the following in your terminal:

您应该在终端中看到类似以下内容的输出：

```
2017-10-26 19:24:05.324 EDT [common/tools/configtxgen] main -> INFO 001
Loading configuration
2017-10-26 19:24:05.329 EDT [common/tools/configtxgen]
doOutputChannelCreateTx -> INFO 002 Generating new channel configtx
2017-10-26 19:24:05.329 EDT [common/tools/configtxgen]
doOutputChannelCreateTx -> INFO 003 Writing new channel tx
```

Next, we will define the anchor peer for Org1 on the channel that we are constructing. Again, be sure to replace \$CHANNEL_NAME or set the environment variable for the following commands. The terminal output will mimic that of the channel transaction artifact:

接下来，我们将在我们正在构建的通道上定义 org1 的定位点。同样，请确保替换 \$channel_name 或为以下命令设置环境变量。终端输出将模拟通道交易工件的输出：

```
../bin/configtxgen -profile TwoOrgsChannel -
outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx -channelID
$CHANNEL_NAME -asOrg Org1MSP
```

Now, we will define the anchor peer for Org2 on the same channel:

现在，我们将在同一个通道上定义 org2 的锚定对等体：

```
../bin/configtxgen -profile TwoOrgsChannel -
outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx -channelID
$CHANNEL_NAME -asOrg Org2MSP
```

6、Start the network

6、启动网络

Note

注释

If you ran the byfn.sh example above previously, be sure that you have brought

down the test network before you proceed (see Bring Down the Network).

如果您以前运行过上面的 byfn.sh 示例，请确保在继续之前已关闭测试网络（请参见关闭网络）。

We will leverage a script to spin up our network. The docker-compose file references the images that we have previously downloaded, and bootstraps the orderer with our previously generated genesis.block.

我们将利用一个脚本来加速我们的网络。docker-compose 文件引用了我们先前下载的图像，并用我们先前生成的 genesis.block 引导排序器。

We want to go through the commands manually in order to expose the syntax and functionality of each call.

为了公开每个调用的语法和功能，我们希望手动地遍历这些命令。

First let's start our network:

首先，让我们开始我们的网络：

```
docker-compose -f docker-compose-cli.yaml up -d
```

If you want to see the realtime logs for your network, then do not supply the -d flag. If you let the logs stream, then you will need to open a second terminal to execute the CLI calls.

如果您想查看网络的实时日志，那么不要提供 -d 标志。如果让日志流化，则需要打开第二个终端来执行 CLI 调用。

Environment variables

环境变量

For the following CLI commands against peer0.org1.example.com to work, we need to preface our commands with the four environment variables given below. These variables for peer0.org1.example.com are baked into the CLI container, therefore we can operate without passing them. HOWEVER, if you want to send calls to other peers or the orderer, then you can provide these values accordingly by editing the docker-compose-base.yaml before starting the container. Modify the following four environment variables to use a different peer and org.

为了使下面针对 peer0.org1.example.com 的 CLI 命令起作用，我们需要在命令前面加上下面给出的四个环境变量。peer0.org1.example.com 的这些变量被烘焙到 cli 容器中，因此我们可以在不传递它们的情况下进行操作。但是，如果您想向其他对等方或订购方发送调用，那么您可以在启动容器之前通过编辑 docker-compose-base.yaml 来相应地提供这些值。修改以下四个环境变量以使用不同的对等机和组织。

```
# Environment variables for PEERO
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/p
eer/crypto/peerOrganizations
```

Create & Join Channel

创建并加入频道

Recall that we created the channel configuration transaction using the configtxgen tool in the Create a Channel Configuration Transaction section, above.

You can repeat that process to create additional channel configuration transactions, using the same or different profiles in the configtx.yaml that you pass to the configtxgen tool. Then you can repeat the process defined in this section to establish those other channels in your network.

回想一下，我们在上面的创建通道配置交易部分中使用 configtxgen 工具创建了通道配置交易。您可以使用传递给 configtxgen 工具的 configtx.yaml 中相同或不同的概要文件，重复该过程以创建其他通道配置交易。然后，您可以重复本节中定义的过程，在您的网络中建立这些其他通道。

We will enter the CLI container using the docker exec command:

我们将使用 docker exec 命令输入 cli 容器：

```
docker exec -it cli bash
```

If successful you should see the following:

如果成功，您应该看到以下内容：

```
root@0d78bb69300d:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

If you do not want to run the CLI commands against the default peer peer0.org1.example.com, replace the values of peer0 or org1 in the four environment variables and run the commands:

如果不想对默认的 peer peer0.org1.example.com 运行 cli 命令，请替换四个环境变量中 peer0 或 org1 的值并运行以下命令：

```
# Environment variables for PEERO
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
export CORE_PEER_LOCALMSPID="Org1MSP"
export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

Next, we are going to pass in the generated channel configuration transaction artifact that we created in the Create a Channel Configuration Transaction section (we called it channel.tx) to the orderer as part of the create channel request.

接下来，我们将把在创建通道配置交易部分（我们称之为 channel.tx）中创建的已生成通道配置交易工件作为创建通道请求的一部分传递给 order。

We specify our channel name with the -c flag and our channel configuration transaction with the -f flag. In this case it is channel.tx, however you can mount your own configuration transaction with a different name. Once again we will set the CHANNEL_NAME environment variable within our CLI container so that we don't have to explicitly pass this argument. Channel names must be all lower case, less than 250 characters long and match the regular expression [a-z][a-z0-9.-]*.

我们用 -c 标志指定通道名称，用 -f 标志指定通道配置交易。在这种情况下，它是 channel.tx，但是您可以使用不同的名称挂载自己的配置事务。我们将再次在 cli 容器中设

置 `channel_name` 环境变量，这样就不必显式传递此参数。通道名称必须全部小写，长度小于 250 个字符，并且与正则表达式 `[A-Z][A-Z0-9.-]` 匹配。

```
export CHANNEL_NAME=mychannel
# the channel.tx file is mounted in the channel-artifacts directory within
your CLI container
# as a result, we pass the full path for the file
# we also pass the path for the orderer ca-cert in order to verify the TLS
handshake
# be sure to export or replace the $CHANNEL_NAME variable appropriately
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/ordererOrganizations/
example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-
cert.pem
```

Note

注释

Notice the `--cafile` that we pass as part of this command. It is the local path to the orderer's root cert, allowing us to verify the TLS handshake.

注意我们作为这个命令的一部分传递的`--cafile`。它是订购方根证书的本地路径，允许我们验证 TLS 握手。

This command returns a genesis block - `<channel-ID.block>` - which we will use to join the channel. It contains the configuration information specified in `channel.tx`. If you have not made any modifications to the default channel name, then the command will return you a proto titled `mychannel.block`.

此命令返回一个 genesis 块-`<channel-id.block>`，我们将使用它来加入通道。它包含 `channel.tx` 中指定的配置信息。如果您没有对默认频道名称进行任何修改，那么该命令将返回一个名为 `mychannel.block` 的协议。

Note

注释

You will remain in the CLI container for the remainder of these manual commands. You must also remember to preface all commands with the corresponding environment variables when targeting a peer other than `peer0.org1.example.com`.

对于这些手动命令的其余部分，您将保留在 `cli` 容器中。当目标不是 `peer0.org1.example.com` 的对等机时，还必须记住在所有命令前面加上相应的环境变量。

Now let's join `peer0.org1.example.com` to the channel.

现在让我们把 `peer0.org1.example.com` 加入到频道中。

```
# By default, this joins ``peer0.org1.example.com`` only
# the <channel-ID.block> was returned by the previous command
# if you have not modified the channel name, you will join with
mychannel.block
# if you have created a different channel name, then pass in the appropriately
named block
peer channel join -b mychannel.block
```

You can make other peers join the channel as necessary by making appropriate

changes in the four environment variables we used in the Environment variables section, above.

您可以根据需要,通过对我们在上面的环境变量部分中使用的四个环境变量进行适当的更改,使其他对等方加入通道。

Rather than join every peer, we will simply join peer0.org2.example.com so that we can properly update the anchor peer definitions in our channel. Since we are overriding the default environment variables baked into the CLI container, this full command will be the following:

与加入每一个对等点不同,我们只需加入 peer0.org2.example.com,这样我们就可以正确地更新通道中的锚定对等点定义。由于我们将覆盖烘焙到 cli 容器中的默认环境变量,因此此完整命令如下:

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
peer channel join -b mychannel.block
```

Alternatively, you could choose to set these environment variables individually rather than passing in the entire string. Once they've been set, you simply need to issue the peer channel join command again and the CLI container will act on behalf of peer0.org2.example.com.

或者,您可以选择单独设置这些环境变量,而不是传递整个字符串。设置好后,只需再次发出 peer channel join 命令,cli 容器将代表 peer0.org2.example.com。

Update the anchor peers

更新定位点对等点

The following commands are channel updates and they will propagate to the definition of the channel. In essence, we adding additional configuration information on top of the channel's genesis block. Note that we are not modifying the genesis block, but simply adding deltas into the chain that will define the anchor peers.

以下命令是频道更新,它们将传播到频道的定义。实质上,我们在通道 Genesis 块的顶部添加了额外的配置信息。请注意,我们并没有修改 Genesis 块,只是简单地在链中添加三角洲,这将定义锚定对等体。

Update the channel definition to define the anchor peer for Org1 as peer0.org1.example.com:

更新通道定义,将 org1 的锚定对等定义为 peer0.org1.example.com:

```
peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f
./channel-artifacts/Org1MSPanchors.tx --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

Now update the channel definition to define the anchor peer for Org2 as

peer0.org2.example.com. Identically to the peer channel join command for the Org2 peer, we will need to preface this call with the appropriate environment variables.

现在更新通道定义，将 org2 的定位点定义为 peer0.org2.example.com。与 org2 对等机的 peer channel join 命令相同，我们需要在这个调用前面加上适当的环境变量。

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/Org2MSPanchors.tx --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

Install & Instantiate Chaincode

安装并实例化链代码

Note

注释

We will utilize a simple existing chaincode. To learn how to write your own chaincode, see the Chaincode for Developers tutorial.

我们将使用一个简单的现有链码。要了解如何编写自己的链代码，请参阅开发人员的链代码教程。

Applications interact with the blockchain ledger through chaincode. As such we need to install the chaincode on every peer that will execute and endorse our transactions, and then instantiate the chaincode on the channel.

应用程序通过链码与区块链分类账交互。因此，我们需要在每个将执行和认可我们的交易的对等机上安装链码，然后在通道上实例化链码。

First, install the sample Go, Node.js or Java chaincode onto the peer0 node in Org1. These commands place the specified source code flavor onto our peer's filesystem.

首先，将样本 Go、Node.js 或 Java 链式代码安装到 OR1 中的 PEE0 节点上。这些命令将指定的源代码风格放到对等文件系统上。

Note

注释

You can only install one version of the source code per chaincode name and version. The source code exists on the peer's file system in the context of chaincode name and version; it is language agnostic. Similarly the instantiated chaincode container will be reflective of whichever language has been installed on the peer.

每个链代码名称和版本只能安装一个版本的源代码。源代码存在于对等机的文件系统中，处于链代码名称和版本的上下文中；它与语言无关。类似地，实例化的链代码容器将反映安装

在对等机上的任何语言。

Golang

```
# this installs the Go chaincode. For go chaincode -p takes the relative
path from $GOPATH/src
```

```
peer chaincode install -n mycc -v 1.0 -p
github.com/chaincode/chaincode_example02/go/
```

Node.js

```
# this installs the Node.js chaincode
```

```
# make note of the -l flag to indicate "node" chaincode
```

```
# for node chaincode -p takes the absolute path to the node.js chaincode
```

```
peer chaincode install -n mycc -v 1.0 -l node -p
/opt/gopath/src/github.com/chaincode/chaincode_example02/node/
```

Java

```
# make note of the -l flag to indicate "java" chaincode
```

```
# for java chaincode -p takes the absolute path to the java chaincode
```

```
peer chaincode install -n mycc -v 1.0 -l java -p
/opt/gopath/src/github.com/chaincode/chaincode_example02/java/
```

When we instantiate the chaincode on the channel, the endorsement policy will be set to require endorsements from a peer in both Org1 and Org2. Therefore, we also need to install the chaincode on a peer in Org2.

当我们在通道上实例化链式代码时，背书策略将被设置为需要来自 ORG1 和 ORG2 中的对等方的背书。因此，我们还需要在 org2 中的对等机上安装链代码。

Modify the following four environment variables to issue the install command against peer0 in Org2:

修改以下四个环境变量以针对 ORG2 中的 Peer0 发出安装命令：

```
# Environment variables for PEER0 in Org2
```

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
```

```
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
```

```
CORE_PEER_LOCALMSPID="Org2MSP"
```

```
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/p
eer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/
ca.crt
```

Now install the sample Go, Node.js or Java chaincode onto a peer0 in Org2. These commands place the specified source code flavor onto our peer's filesystem.

现在将样本 GO、Node.js 或 Java 链式代码安装到 ORG2 中的 PEE0 上。这些命令将指定的源代码风格放到对等文件系统上。

Golang

```
# this installs the Go chaincode. For go chaincode -p takes the relative
path from $GOPATH/src
```

```
peer chaincode install -n mycc -v 1.0 -p
github.com/chaincode/chaincode_example02/go/
```

Node.js

```
# this installs the Node.js chaincode
```

```
# make note of the -l flag to indicate "node" chaincode
# for node chaincode -p takes the absolute path to the node.js chaincode
peer chaincode install -n mycc -v 1.0 -l node -p
/opt/gopath/src/github.com/chaincode/chaincode_example02/node/
```

Java

```
# make note of the -l flag to indicate "java" chaincode
# for java chaincode -p takes the absolute path to the java chaincode
peer chaincode install -n mycc -v 1.0 -l java -p
/opt/gopath/src/github.com/chaincode/chaincode_example02/java/
```

Next, instantiate the chaincode on the channel. This will initialize the chaincode on the channel, set the endorsement policy for the chaincode, and launch a chaincode container for the targeted peer. Take note of the `-P` argument. This is our policy where we specify the required level of endorsement for a transaction against this chaincode to be validated.

接下来，在通道上实例化链代码。这将初始化通道上的链码，为链码设置认可策略，并为目标对等端启动链码容器。注意`-p` 参数。这是我们的策略，我们在其中指定了针对要验证的链码的事务所需的认可级别。

In the command below you'll notice that we specify our policy as `-P "AND ('Org1MSP.peer','Org2MSP.peer')"`. This means that we need "endorsement" from a peer belonging to Org1 AND Org2 (i.e. two endorsement). If we changed the syntax to OR then we would need only one endorsement.

在下面的命令中，您会注意到我们将策略指定为`-p "和 ('org1MSP.peer', 'org2MSP.peer')"`。这意味着我们需要来自属于 org1 和 org2 的对等方的“认可”（即两个认可）。如果我们将语法改为或，那么我们只需要一个背书。

Golang

```
# be sure to replace the $CHANNEL_NAME environment variable if you have not
exported it
# if you did not install your chaincode with a name of mycc, then modify
that argument as well
peer chaincode instantiate -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n mycc -v 1.0 -c '{"Args":["init","a", "100",
"b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.peer')"
```

Node.js

Note

注释

The instantiation of the Node.js chaincode will take roughly a minute. The command is not hanging; rather it is installing the fabric-shim layer as the image is being compiled.

node.js 链代码的实例化大约需要一分钟。命令没有挂起；而是在编译图像时安装结构填充层。

```
# be sure to replace the $CHANNEL_NAME environment variable if you have not
exported it
```

if you did not install your chaincode with a name of mycc, then modify that argument as well

notice that we must pass the -l flag after the chaincode name to identify the language

```
peer chaincode instantiate -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n mycc -l node -v 1.0 -c '{"Args":["init","a","100",
"b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.peer')"
```

Java

Note

注释

Please note, Java chaincode instantiation might take time as it compiles chaincode and downloads docker container with java environment.

请注意, Java 链代码实例化可能需要时间,因为它编译链代码,并下载 Java 环境中的 DOCKER 容器。

```
peer chaincode instantiate -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n mycc -l java -v 1.0 -c '{"Args":["init","a","100",
"b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.peer')"
```

See the endorsement policies documentation for more details on policy implementation.

有关政策实施的更多详细信息,请参阅背书政策文档。

If you want additional peers to interact with ledger, then you will need to join them to the channel, and install the same name, version and language of the chaincode source onto the appropriate peer's filesystem. A chaincode container will be launched for each peer as soon as they try to interact with that specific chaincode. Again, be cognizant of the fact that the Node.js images will be slower to compile.

如果您希望其他对等端与 Ledger 交互,那么您需要将它们连接到通道,并将链码源的相同名称、版本和语言安装到相应端点的文件系统上。一旦每个对等端尝试与特定的链码进行交互,就会为它们启动一个链码容器。再次认识到 node.js 图像的编译速度会变慢。

Once the chaincode has been instantiated on the channel, we can forgo the l flag. We need only pass in the channel identifier and name of the chaincode.

一旦在通道上实例化了链码,我们就可以放弃 l 标志。我们只需要传递通道标识符和链码的名称。

Query

查询

Let's query for the value of a to make sure the chaincode was properly instantiated and the state DB was populated. The syntax for query is as follows:

让我们查询 a 的值,以确保链代码已正确实例化并且状态 db 已填充。查询的语法如下:

be sure to set the -C and -n flags appropriately

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

Invoke

调用

Now let's move 10 from a to b. This transaction will cut a new block and update the state DB. The syntax for invoke is as follows:

现在让我们将 10 从 A 移到 B。这个事务将剪切一个新块并更新状态 db。调用的语法如下：

```
# be sure to set the -C and -n flags appropriately
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 $CHANNEL_NAME -n mycc --peerAddresses peer0.org1.example.com:7051 -
-tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org
1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses
peer0.org2.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org
2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c
'{"Args":["invoke","a","b","10"]}'
```

Query

查询

Let's confirm that our previous invocation executed properly. We initialized the key a with a value of 100 and just removed 10 with our previous invocation. Therefore, a query against a should return 90. The syntax for query is as follows.

让我们确认上一次调用是否正确执行。我们初始化了值为 100 的键 A，并在前面的调用中删除了 10。因此，对 a 的查询应该返回 90。查询的语法如下。

```
# be sure to set the -C and -n flags appropriately
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

We should see the following:

我们应该看到以下内容：

Query Result: 90

Feel free to start over and manipulate the key value pairs and subsequent invocations.

请随意重新开始并操作键值对和后续调用。

Install

安装

Now we will install the chaincode on a third peer, peer1 in Org2. Modify the following four environment variables to issue the install command against peer1 in Org2:

现在我们将在第三个对等机上安装 chaincode，即 org2 中的 peer1。修改以下四个环境变量，以对 org2 中的 peer1 发出 install 命令：

```
# Environment variables for PEER1 in Org2
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer1.org2.example.com:7051
```

```
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
```

Now install the sample Go, Node.js or Java chaincode onto peer1 in Org2. These commands place the specified source code flavor onto our peer's filesystem.

现在将样本 GO、Node.js 或 Java 链码安装到 ORG2 上的 PEE1 上。这些命令将指定的源代码风格放到对等文件系统上。

Golang

```
# this installs the Go chaincode. For go chaincode -p takes the relative path from $GOPATH/src
```

```
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/chaincode_example02/go/
```

Node.js

```
# this installs the Node.js chaincode
```

```
# make note of the -l flag to indicate "node" chaincode
```

```
# for node chaincode -p takes the absolute path to the node.js chaincode
```

```
peer chaincode install -n mycc -v 1.0 -l node -p /opt/gopath/src/github.com/chaincode/chaincode_example02/node/
```

Java

```
# make note of the -l flag to indicate "java" chaincode
```

```
# for java chaincode -p takes the absolute path to the java chaincode
```

```
peer chaincode install -n mycc -v 1.0 -l java -p /opt/gopath/src/github.com/chaincode/chaincode_example02/java/
```

Query

查询

Let's confirm that we can issue the query to Peer1 in Org2. We initialized the key a with a value of 100 and just removed 10 with our previous invocation. Therefore, a query against a should still return 90.

让我们确认我们可以向 org2 中的 peer1 发出查询。我们初始化了值为 100 的键 A，并在前面的调用中删除了 10。因此，对 a 的查询仍应返回 90。

peer1 in Org2 must first join the channel before it can respond to queries. The channel can be joined by issuing the following command:

org2 中的 peer1 必须首先加入通道，然后才能响应查询。可以通过发出以下命令来加入通道：

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
```

```
CORE_PEER_ADDRESS=peer1.org2.example.com:7051
```

```
CORE_PEER_LOCALMSPID="Org2MSP"
```

```
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
```

```
peer channel join -b mychannel.block
```

After the join command returns, the query can be issued. The syntax for query

is as follows.

join 命令返回后，可以发出查询。查询的语法如下。

```
# be sure to set the -C and -n flags appropriately
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

We should see the following:

我们应该看到以下内容：

Query Result: 90

Feel free to start over and manipulate the key value pairs and subsequent invocations.

请随意重新开始并操作键值对和后续调用。

What's happening behind the scenes?

后台发生了什么？

Note

注释

These steps describe the scenario in which script.sh is run by './byfn.sh up'. Clean your network with ./byfn.sh down and ensure this command is active. Then use the same docker-compose prompt to launch your network again

这些步骤描述了由“./byfn.sh up”运行 script.sh 的场景。使用 ./byfn.sh down 清理网络，并确保此命令处于活动状态。然后使用相同的 docker compose 提示再次启动网络

A script - script.sh - is baked inside the CLI container. The script drives the createChannel command against the supplied channel name and uses the channel.tx file for channel configuration.

脚本-script.sh-在 cli 容器中烘焙。该脚本根据提供的通道名称驱动 createChannel 命令，并使用 channel.tx 文件进行通道配置。

The output of createChannel is a genesis block - <your_channel_name>.block - which gets stored on the peers' file systems and contains the channel configuration specified from channel.tx.

createChannel 的输出是一个 genesis 块-<your_channel_name>.block，它存储在对等文件系统中，并包含从 channel.tx 指定的通道配置。

The joinChannel command is exercised for all four peers, which takes as input the previously generated genesis block. This command instructs the peers to join <your_channel_name> and create a chain starting with <your_channel_name>.block.

joinchannel 命令用于所有四个对等机，它将先前生成的 genesis 块作为输入。此命令指示对等方加入<u_channel_name>并从<u_channel_name>块开始创建链。

Now we have a channel consisting of four peers, and two organizations. This is our TwoOrgsChannel profile.

现在我们有了一个由四个同行和两个组织组成的渠道。这是我们的 Twoorgschannel 简介。

peer0.org1.example.com and peer1.org1.example.com belong to Org1;
peer0.org2.example.com and peer1.org2.example.com belong to Org2

peer0.org1.example.com 和 peer1.org1.example.com 属于 org1 ;
peer0.org2.example.com 和 peer1.org2.example.com 属于 org2

These relationships are defined through the crypto-config.yaml and the MSP path is specified in our docker compose.

这些关系是通过 `crypto-config.yaml` 定义的，MSP 路径在 `docker compose` 中指定。

The anchor peers for Org1MSP (`peer0.org1.example.com`) and Org2MSP (`peer0.org2.example.com`) are then updated. We do this by passing the `Org1MSPanchors.tx` and `Org2MSPanchors.tx` artifacts to the ordering service along with the name of our channel.

然后更新 `org1MSP(peer0.org1.example.com)` 和 `org2MSP(peer0.org2.example.com)` 的定位点。我们通过将 `org1MSPanchors.tx` 和 `org2MSPanchors.tx` 工件连同频道名称一起传递给订购服务来实现这一点。

A chaincode - `chaincode_example02` - is installed on `peer0.org1.example.com` and `peer0.org2.example.com`

`chaincode-chaincode_example02` 安装在 `peer0.org1.example.com` 和 `peer0.org2.example.com` 上。

The chaincode is then “instantiated” on mychannel. Instantiation adds the chaincode to the channel, starts the container for the target peer, and initializes the key value pairs associated with the chaincode. The initial values for this example are [“a”, 100 “b”, 200]. This “instantiation” results in a container by the name of `dev-peer0.org2.example.com-mycc-1.0` starting.

然后在 `myChannel` 上“实例化”链码。实例化将链码添加到通道，启动目标对等机的容器，并初始化与链码关联的键值对。此示例的初始值为[“A”、“100”、“B”、“200”]。此“实例化”将以 `dev-peer0.org2.example.com-mycc-1.0` 的名称在容器中启动。

The instantiation also passes in an argument for the endorsement policy. The policy is defined as `-P “AND (‘Org1MSP.peer’, ‘Org2MSP.peer’)”`, meaning that any transaction must be endorsed by a peer tied to Org1 and Org2.

实例化还传递了背书策略的参数。该策略定义为 `-p “和 (‘org1MSP.peer’ , ‘org2MSP.peer’)”`，这意味着任何事务都必须由绑定到 `org1` 和 `org2` 的对等方背书。

A query against the value of “a” is issued to `peer0.org2.example.com`. A container for Org2 peer0 by the name of `dev-peer0.org2.example.com-mycc-1.0` was started when the chaincode was instantiated. The result of the query is returned. No write operations have occurred, so a query against “a” will still return a value of “100”.

向 `peer0.org2.example.com` 发出对“a”值的查询。当链码实例化时，启动了一个名为 `dev-peer0.org2.example.com-mycc-1.0` 的 `org2 peer0` 容器。返回查询结果。未发生写入操作，因此对“a”的查询仍将返回值“100”。

An invoke is sent to `peer0.org1.example.com` and `peer0.org2.example.com` to move “10” from “a” to “b”

调用被发送到 `peer0.org1.example.com` 和 `peer0.org2.example.com`，以将“10”从“a”移动到“b”。

A query is sent to `peer0.org2.example.com` for the value of “a”. A value of 90 is returned, correctly reflecting the previous transaction during which the value for key “a” was modified by 10.

将向 `peer0.org2.example.com` 发送一个值为“a”的查询。返回的值为 90，正确反映了上一个事务，在此事务期间，键“A”的值被修改了 10。

The chaincode - `chaincode_example02` - is installed on `peer1.org2.example.com`

chaincode-chaincode_example02-安装在 peer1.org2.example.com 上。

A query is sent to peer1.org2.example.com for the value of “a”. This starts a third chaincode container by the name of dev-peer1.org2.example.com-mycc-1.0. A value of 90 is returned, correctly reflecting the previous transaction during which the value for key “a” was modified by 10.

将向 peer1.org2.example.com 发送一个值为 “a” 的查询。这将以 dev-peer1.org2.example.com-mycc-1.0 的名称启动第三个 chaincode 容器。返回的值为 90，正确反映了上一个交易，在此事务期间，键 “A” 的值被修改了 10。

What does this demonstrate?

这说明了什么？

Chaincode MUST be installed on a peer in order for it to successfully perform read/write operations against the ledger. Furthermore, a chaincode container is not started for a peer until an init or traditional transaction - read/write - is performed against that chaincode (e.g. query for the value of “a”). The transaction causes the container to start. Also, all peers in a channel maintain an exact copy of the ledger which comprises the blockchain to store the immutable, sequenced record in blocks, as well as a state database to maintain a snapshot of the current state. This includes those peers that do not have chaincode installed on them (like peer1.org1.example.com in the above example) . Finally, the chaincode is accessible after it is installed (like peer1.org2.example.com in the above example) because it has already been instantiated.

必须在对等机上安装链码，才能对分类帐成功执行读/写操作。此外，在对对等端执行 init 或传统交易（读/写）之前，不会为对等端启动链码容器（例如查询 “a” 的值）。交易导致容器启动。此外，一个通道中的所有对等方都维护一个包含区块链的分类账的精确副本，以块形式存储不可变的序列记录，以及一个状态数据库，以维护当前状态的快照。这包括那些没有安装链码的对等机（如上面的例子中的 peer1.org1.example.com）。最后，在安装了 chaincode 之后可以访问它（如上面示例中的 peer1.org2.example.com），因为它已经被实例化了。

How do I see these transactions?

我如何看待这些交易？

Check the logs for the CLI Docker container.

检查 cli docker 容器的日志。

`docker logs -f cli`

You should see the following output:

您应该看到以下输出：

```

2017-05-16 17:08:01.366 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing local MSP
2017-05-16 17:08:01.366 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining default signing i
2017-05-16 17:08:01.366 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AB1070A6708031A0C08F1
2017-05-16 17:08:01.367 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: E61DB37F4E8B0D32C9FE10E39
Query Result: 90
2017-05-16 17:08:15.158 UTC [main] main -> INFO 008 Exiting.....
===== Query successful on peer1.org2 on channel 'mychannel' =====

===== All GOOD, BYFN execution completed =====

```

END

You can scroll through these logs to see the various transactions.

您可以滚动浏览这些日志以查看各种交易。

How can I see the chaincode logs?

如何查看链码日志？

Inspect the individual chaincode containers to see the separate transactions executed against each container. Here is the combined output from each container:

检查各个链码容器，查看针对每个容器执行的单独交易。以下是每个容器的组合输出：

```

$ docker logs dev-peer0.org2.example.com-myc-1.0
04:30:45.947 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Init
Aval = 100, Bval = 200

$ docker logs dev-peer0.org1.example.com-myc-1.0
04:31:10.569 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Invoke
Query Response:{"Name":"a","Amount":"100"}
ex02 Invoke
Aval = 90, Bval = 210

$ docker logs dev-peer1.org2.example.com-myc-1.0
04:31:30.420 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Invoke
Query Response:{"Name":"a","Amount":"90"}

```

7、Understanding the Docker Compose topology

7、了解 Docker 撰写拓扑

The BYFN sample offers us two flavors of Docker Compose files, both of which are extended from the docker-compose-base.yaml (located in the base folder). Our first flavor, docker-compose-cli.yaml, provides us with a CLI container, along with an orderer, four peers. We use this file for the entirety of the instructions on this page.

byfn 示例为我们提供了两种类型的 docker compose 文件，这两种文件都是从 docker-compose-base.yaml（位于 base 文件夹中）扩展而来的。我们的第一个特色是 docker-compose-cli.yaml，它为我们提供了一个 cli 容器，以及一个订购者，四个对等方。我们将此文件用于本页的全部说明。

Note

注释

the remainder of this section covers a docker-compose file designed for the SDK. Refer to the Node SDK repo for details on running these tests.

本节的其余部分介绍为 SDK 设计的 Docker 撰写文件。有关运行这些测试的详细信息，请参阅 node sdk repo。

The second flavor, docker-compose-e2e.yaml, is constructed to run end-to-end tests using the Node.js SDK. Aside from functioning with the SDK, its primary differentiation is that there are containers for the fabric-ca servers. As a result, we are able to send REST calls to the organizational CAs for user registration and enrollment.

第二种风格是 docker-compose-e2e.yaml, 它构造为使用 node.js sdk 运行端到端测试。除了与 SDK 一起工作之外, 它的主要区别在于存在用于结构 CA 服务器的容器。因此, 我们可以向组织 CA 发送 REST 调用, 以进行用户注册和注册。

If you want to use the docker-compose-e2e.yaml without first running the byfn.sh script, then we will need to make four slight modifications. We need to point to the private keys for our Organization's CA's. You can locate these values in your crypto-config folder. For example, to locate the private key for Org1 we would follow this path - crypto-config/peerOrganizations/org1.example.com/ca/. The private key is a long hash value followed by _sk. The path for Org2 would be - crypto-config/peerOrganizations/org2.example.com/ca/.

如果您想在不首先运行 byfn.sh 脚本的情况下使用 docker-compose-e2e.yaml, 那么我们需要做四个细微的修改。我们需要指向组织的 CA 的私钥。您可以在加密配置文件夹中找到这些值。例如, 要定位 org1 的私钥, 我们将遵循以下路径 -crypto config/peerorganizations/org1.example.com/ca/。私钥是一个长的哈希值, 后跟 _sk.org2 的路径应该是 -crypto config/peerorganizations/org2.example.com/ca/。

In the docker-compose-e2e.yaml update the FABRIC_CA_SERVER_TLS_KEYFILE variable for ca0 and ca1. You also need to edit the path that is provided in the command to start the ca server. You are providing the same private key twice for each CA container.

在 docker-compose-e2e.yaml 中, 更新 ca0 和 ca1 的 fabric_ca_server_tls_keyfile 变量。您还需要编辑命令中提供的路径来启动 CA 服务器。您为每个 CA 容器提供相同的私钥两次。

8、Using CouchDB

8、使用 CouchDB

The state database can be switched from the default (goleveldb) to CouchDB. The same chaincode functions are available with CouchDB, however, there is the added ability to perform rich and complex queries against the state database data content contingent upon the chaincode data being modeled as JSON.

状态数据库可以从默认值 (goleveldb) 切换到 couchdb。对于 couchdb, 也可以使用相同的 chaincode 函数, 但是, 根据建模为 JSON 的 chaincode 数据, 可以对状态数据库数据

内容执行丰富而复杂的查询。

To use CouchDB instead of the default database (goleveldb), follow the same procedures outlined earlier for generating the artifacts, except when starting the network pass docker-compose-couch.yaml as well:

要使用 couchdb 而不是默认数据库 (goleveldb)，请遵循前面概述的生成工件的相同过程，启动 network pass docker-compose-coach.yaml 时除外：

```
docker-compose -f docker-compose-cli.yaml -f docker-compose-couch.yaml up -d
```

chaincode_example02 should now work using CouchDB underneath.

链码示例 02 现在应该可以使用下面的 couchdb。

Note

注释

If you choose to implement mapping of the fabric-couchdb container port to a host port, please make sure you are aware of the security implications. Mapping of the port in a development environment makes the CouchDB REST API available, and allows the visualization of the database via the CouchDB web interface (Fauxton). Production environments would likely refrain from implementing port mapping in order to restrict outside access to the CouchDB containers.

如果您选择实现结构 couchdb 容器端口到主机端口的映射，请确保您了解安全隐患。开发环境中端口的映射使 CouchDB REST API 可用，并允许通过 CouchDB Web 界面（Fauxton）可视化数据库。生产环境可能会避免实现端口映射，以限制对 CouchDB 容器的外部访问。

You can use chaincode_example02 chaincode against the CouchDB state database using the steps outlined above, however in order to exercise the CouchDB query capabilities you will need to use a chaincode that has data modeled as JSON, (e.g. marbles02). You can locate the marbles02 chaincode in the fabric/examples/chaincode/go directory.

您可以使用 chaincode-example02 chaincode 对 couchdb 状态数据库使用上述步骤，但是为了使用 couchdb 查询功能，您需要使用具有 json 建模数据的 chaincode（例如 marbles02）。您可以在 fabric/examples/chaincode/go 目录中找到 marbles02 链码。

We will follow the same process to create and join the channel as outlined in the Create & Join Channel section above. Once you have joined your peer(s) to the channel, use the following steps to interact with the marbles02 chaincode:

我们将按照上面“创建和连接通道”一节中所述的相同过程来创建和连接通道。一旦您将您的对等端加入通道，请使用以下步骤与 Marbles02 链码进行交互：

Install and instantiate the chaincode on peer0.org1.example.com:

在 peer0.org1.example.com 上安装并实例化链码：

```
# be sure to modify the $CHANNEL_NAME variable accordingly for the instantiate command
```

```
peer chaincode install -n marbles -v 1.0 -p github.com/chaincode/marbles02/go
peer chaincode instantiate -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -v 1.0 -c '{"Args":["init"]}' -P "OR ('OrgOMSP.peer','Org1MSP.peer')"
```

Create some marbles and move them around:

创建一些 marbles 并移动它们:

```
# be sure to modify the $CHANNEL_NAME variable accordingly
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c ' {"Args":["initMarble","marble1","blue","35","tom"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c ' {"Args":["initMarble","marble2","red","50","tom"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c ' {"Args":["initMarble","marble3","blue","70","tom"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c ' {"Args":["transferMarble","marble2","jerry"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c ' {"Args":["transferMarblesBasedOnColor","blue","jerry"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c ' {"Args":["delete","marble1"]}'
```

If you chose to map the CouchDB ports in docker-compose, you can now view the state database through the CouchDB web interface (Fauxton) by opening a browser and navigating to the following URL:

如果选择在 Docker Compose 中映射 CouchDB 端口, 现在可以通过 CouchDB Web 界面 (Fauxton) 打开浏览器并导航到以下 URL 来查看状态数据库:

http://localhost:5984/_utils

You should see a database named mychannel (or your unique channel name) and the documents inside it.

您应该看到一个名为 mychannel (或您唯一的频道名称) 的数据库以及其中的文档。

Note

注释

For the below commands, be sure to update the \$CHANNEL_NAME variable

appropriately.

对于以下命令，请确保适当更新`$channel_name` 变量。

You can run regular queries from the CLI (e.g. reading marble2):

您可以从 CLI 运行常规查询（例如，读取 marble2）:

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c
'{"Args":["readMarble","marble2"]}'
```

The output should display the details of marble2:

输出应显示 marble2 的详细信息:

```
Query Result:
{"color":"red","docType":"marble","name":"marble2","owner":"jerry","size":50}
```

You can retrieve the history of a specific marble - e.g. marble1:

您可以检索特定大理石的历史记录-例如 marble1:

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c
'{"Args":["getHistoryForMarble","marble1"]}'
```

The output should display the transactions on marble1:

输出应显示 Marble1 上的交易:

```
Query Result:
[{"TxId":"1c3d3caf124c89f91a4c0f353723ac736c58155325f02890adebaa15e16e6464",
"Value":{"docType":"marble","name":"marble1","color":"blue","size":35,"owner":"tom"}}, {"TxId":"755d55c281889eaeefb405586f9e25d71d36eb3d35420af833a20a2f53a3eefd",
"Value":{"docType":"marble","name":"marble1","color":"blue","size":35,"owner":"jerry"}}, {"TxId":"819451032d813dde6247f85e56a89262555e04f14788ee33e28b232eef36d98f", "Value":{}}]
```

You can also perform rich queries on the data content, such as querying marble fields by owner jerry:

您还可以对数据内容执行丰富的查询，例如按所有者 Jerry 查询大理石字段:

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c
'{"Args":["queryMarblesByOwner","jerry"]}'
```

The output should display the two marbles owned by jerry:

输出应显示 Jerry 拥有的两个大理石:

```
Query Result: [{"Key":"marble2",
"Record":{"color":"red","docType":"marble","name":"marble2","owner":"
```

9、Why CouchDB

9、为什么 CouchDB

CouchDB is a kind of NoSQL solution. It is a document-oriented database where document fields are stored as key-value maps. Fields can be either a simple key-value pair, list, or map. In addition to keyed/composite-key/key-range queries which are supported by LevelDB, CouchDB also supports full data rich queries capability, such as non-key queries against the whole blockchain data, since its data content is stored in JSON format and fully queryable. Therefore, CouchDB can meet chaincode, auditing, reporting requirements for many use cases that not

supported by LevelDB.

CouchDB 是一种 NoSQL 解决方案。它是一个面向文档的数据库，其中文档字段存储为键值映射。字段可以是简单的键值对、列表或映射。除了 LevelDB 支持的键控/复合键/键范围查询外，CouchDB 还支持完整的数据丰富查询功能，例如针对整个区块链数据的非键查询，因为其数据内容以 JSON 格式存储，并且完全可查询。因此，CouchDB 可以满足许多不受级别数据库支持的用例的链代码、审计和报告需求。

CouchDB can also enhance the security for compliance and data protection in the blockchain. As it is able to implement field-level security through the filtering and masking of individual attributes within a transaction, and only authorizing the read-only permission if needed.

CouchDB 还可以增强区块链中的合规性和数据保护安全性。因为它能够通过过滤和屏蔽交易中的单个属性来实现字段级安全性，并且只在需要时授权只读权限。

In addition, CouchDB falls into the AP-type (Availability and Partition Tolerance) of the CAP theorem. It uses a master-master replication model with Eventual Consistency. More information can be found on the Eventual Consistency page of the CouchDB documentation. However, under each fabric peer, there is no database replicas, writes to database are guaranteed consistent and durable (not Eventual Consistency).

此外，couchdb 属于 cap 定理的 AP-type（可用性和分区公差）。它使用具有最终一致性的主复制模型。更多信息可以在 CouchDB 文档的最终一致性页面上找到。但是，在每个结构对等机下，没有数据库副本，对数据库的写操作保证一致性和持久性（而不是最终的一致性）。

CouchDB is the first external pluggable state database for Fabric, and there could and should be other external database options. For example, IBM enables the relational database for its blockchain. And the CP-type (Consistency and Partition Tolerance) databases may also in need, so as to enable data consistency without application level guarantee.

couchdb 是第一个用于结构的外部可插拔状态数据库，可以也应该有其他外部数据库选项。例如，IBM 为其区块链启用关系数据库。CP 类型（一致性和分区容限）数据库也可能需要，以便在没有应用程序级别保证的情况下实现数据一致性。

10、A Note on Data Persistence

10、关于数据持久性的说明

If data persistence is desired on the peer container or the CouchDB container, one option is to mount a directory in the docker-host into a relevant directory in the container. For example, you may add the following two lines in the peer container specification in the docker-compose-base.yaml file:

如果在对等容器或 CouchDB 容器上需要数据持久性，一个选项是将 Docker 主机中的目录装入容器中的相关目录。例如，您可以在 docker-compose-base.yaml 文件的对等容器规范中添加以下两行：

```
volumes:  
- /var/hyperledger/peer0:/var/hyperledger/production
```

For the CouchDB container, you may add the following two lines in the CouchDB container specification:

对于 couchdb 容器，可以在 couchdb 容器规范中添加以下两行：

```
volumes:  
- /var/hyperledger/couchdb0:/opt/couchdb/data
```

11、Troubleshooting

11、故障排除

Always start your network fresh. Use the following command to remove artifacts, crypto, containers and chaincode images:

始终重新启动网络。使用以下命令删除工件、加密、容器和链码图像：

```
./byfn.sh down
```

Note

注释

You will see errors if you do not remove old containers and images.

如果不删除旧的容器和图像，您将看到错误。

If you see Docker errors, first check your docker version (Prerequisites), and then try restarting your Docker process. Problems with Docker are oftentimes not immediately recognizable. For example, you may see errors resulting from an inability to access crypto material mounted within a container.

如果看到 Docker 错误，请首先检查 Docker 版本（先决条件），然后尝试重新启动 Docker 进程。Docker 的问题通常无法立即识别。例如，您可能会看到由于无法访问安装在容器中的加密材料而导致的错误。

If they persist remove your images and start from scratch:

如果它们仍然存在，请删除您的图像并从头开始：

```
docker rm -f $(docker ps -aq)  
docker rmi -f $(docker images -q)
```

If you see errors on your create, instantiate, invoke or query commands, make sure you have properly updated the channel name and chaincode name. There are placeholder values in the supplied sample commands.

如果在创建、实例化、调用或查询命令上看到错误，请确保已正确更新通道名和链码名。提供的示例命令中有占位符值。

If you see the below error:

如果您看到以下错误：

```
Error: Error endorsing chaincode: rpc error: code = 2 desc = Error installing  
chaincode                                code                                mycc:1.0(chaincode  
/var/hyperledger/production/chaincodes/mycc.1.0 exits)
```

You likely have chaincode images (e.g. dev-peer1.org2.example.com-mycc-1.0 or dev-peer0.org1.example.com-mycc-1.0) from prior runs. Remove them and try again.

您可能以前运行的链码映像（例如 dev-peer1.org2.example.com-mycc-1.0 或 dev-peer0.org1.example.com-mycc-1.0）。删除它们并重试。

```
docker rmi -f $(docker images | grep peer[0-9]-peer[0-9] | awk '{print $3}')
```

If you see something similar to the following:

如果您看到类似以下内容：

```
Error connecting: rpc error: code = 14 desc = grpc: RPC failed fast due to transport failure
```

```
Error: rpc error: code = 14 desc = grpc: RPC failed fast due to transport failure
```

Make sure you are running your network against the “1.0.0” images that have been retagged as “latest”.

确保您的网络是针对“1.0.0”图像运行的，这些图像已被重新标记为“最新”。

If you see the below error:

如果您看到以下错误：

```
[configtx/tool/localconfig] Load -> CRIT 002 Error reading configuration:
Unsupported Config Type ""
```

```
panic: Error reading configuration: Unsupported Config Type ""
```

Then you did not set the FABRIC_CFG_PATH environment variable properly. The configtxgen tool needs this variable in order to locate the configtx.yaml. Go back and execute an `export FABRIC_CFG_PATH=$PWD`, then recreate your channel artifacts.

然后，您没有正确设置 `fabric_cfg_path` 环境变量。configtxgen 工具需要这个变量来定位 configtx.yaml。返回并执行 `export fabric_cfg_path=$pwd`，然后重新创建通道工件。

To cleanup the network, use the down option:

要清理网络，请使用向下选项：

```
./byfn.sh down
```

If you see an error stating that you still have “active endpoints”, then prune your Docker networks. This will wipe your previous networks and start you with a fresh environment:

如果您看到一个错误声明您仍然有“活动端点”，那么修剪您的 Docker 网络。这将清除以前的网络并为您提供新的环境：

```
docker network prune
```

You will see the following message:

您将看到以下消息：

```
WARNING! This will remove all networks not used by at least one container.
```

```
Are you sure you want to continue? [y/N]
```

```
Select y.
```

If you see an error similar to the following:

如果您看到类似以下的错误：

```
/bin/bash: ./scripts/script.sh: /bin/bash^M: bad interpreter: No such file
or directory
```

Ensure that the file in question (script.sh in this example) is encoded in the Unix format. This was most likely caused by not setting `core.autocrlf` to false in your Git configuration (see Windows extras). There are several ways of fixing this. If you have access to the vim editor for instance, open the file:

确保相关文件（本例中的 script.sh）以 UNIX 格式编码。这很可能是由于在 Git 配置中没有将 `core.autocrlf` 设置为 false（请参见 Windows Extras）。有几种方法可以解决这个问题。例如，如果您可以访问 VIM 编辑器，请打开文件：

```
vim ./fabric-samples/first-network/scripts/script.sh
```

Then change its format by executing the following vim command:

然后通过执行以下 VIM 命令更改其格式:

```
:set ff=unix
```

Note

注释

If you continue to see errors, share your logs on the fabric-questions channel on Hyperledger Rocket Chat or on StackOverflow.

如果您继续看到错误,请在 Hyperledger Rocket Chat 或 StackOverflow 上的 Fabric Questions 频道上共享日志。

四、Adding an Org to a Channel

四、向渠道添加组织

Note

注释

Ensure that you have downloaded the appropriate images and binaries as outlined in Install Samples, Binaries and Docker Images and Prerequisites that conform to the version of this documentation (which can be found at the bottom of the table of contents to the left). In particular, your version of the fabric-samples folder must include the eyfn.sh (“Extending Your First Network”) script and its related scripts.

确保您已下载了安装示例、二进制文件和 Docker 映像中概述的适当映像和二进制文件,以及符合本文档版本的前提条件(可在左侧目录的底部找到)。尤其是, Fabric Samples 文件夹的版本必须包括 eyfn.sh (“扩展第一个网络”)脚本及其相关脚本。

This tutorial serves as an extension to the Building Your First Network (BYFN) tutorial, and will demonstrate the addition of a new organization - Org3 - to the application channel (mychannel) autogenerated by BYFN. It assumes a strong understanding of BYFN, including the usage and functionality of the aforementioned utilities.

本教程作为构建第一个网络 (byfn) 教程的扩展,将演示如何向 byfn 自动生成的应用程序通道 (mychannel) 添加新组织 org3。它假定对 byfn 有很强理解,包括上述实用程序的用法和功能。

le we will focus solely on the integration of a new organization here, the same approach can be adopted when performing other channel configuration updates (updating modification policies or altering batch size, for example). To learn more about the process and possibilities of channel config updates in general, check out Updating a Channel Configuration). It’s also worth noting that channel configuration updates like the one demonstrated here will usually be the responsibility of an organization admin (rather than a chaincode or application developer).

虽然我们将只关注新组织的集成,但在执行其他通道配置更新(例如更新修改策略或更改批大小)时,也可以采用相同的方法。要了解有关通道配置更新的过程和可能性的更多信息,请查看更新通道配置)。同样值得注意的是,像这里演示的那样的通道配置更新通常由组织管理员(而不是链码或应用程序开发人员)负责。

Note

注释

Make sure the automated byfn.sh script runs without error on your machine before continuing. If you have exported your binaries and the related tools (cryptogen, configtxgen, etc) into your PATH variable, you'll be able to modify the commands accordingly without passing the fully qualified path.

在继续之前，请确保 automated byfn.sh 脚本在您的计算机上没有错误地运行。如果已经将二进制文件和相关工具（cryptogen、configtxgen 等）导出到路径变量中，则可以相应地修改命令，而无需传递完全限定的路径。

1、Setup the Environment

1、设置环境

We will be operating from the root of the first-network subdirectory within your local clone of fabric-samples. Change into that directory now. You will also want to open a few extra terminals for ease of use.

我们将从结构示例的本地克隆中的第一个网络子目录的根目录进行操作。现在转到那个目录。为了方便使用，您还需要打开一些额外的终端。

First, use the byfn.sh script to tidy up. This command will kill any active or stale docker containers and remove previously generated artifacts. It is by no means necessary to bring down a Fabric network in order to perform channel configuration update tasks. However, for the sake of this tutorial, we want to operate from a known initial state. Therefore let's run the following command to clean up any previous environments:

首先，使用 byfn.sh 脚本进行整理。此命令将杀死任何活动或过时的 Docker 容器，并删除以前生成的工件。为了执行通道配置更新任务，不必关闭结构网络。但是，为了本教程的目的，我们希望从已知的初始状态进行操作。因此，让我们运行以下命令来清理以前的任何环境：

```
./byfn.sh down
```

Now generate the default BYFN artifacts:

现在生成默认的 byfn 工件：

```
./byfn.sh generate
```

And launch the network making use of the scripted execution within the CLI container:

并利用 cli 容器中的脚本执行启动网络：

```
./byfn.sh up
```

Now that you have a clean version of BYFN running on your machine, you have two different paths you can pursue. First, we offer a fully commented script that will carry out a config transaction update to bring Org3 into the network.

既然您的机器上运行了一个干净的 byfn 版本，那么您就有了两种不同的路径。首先，我们提供一个完全注释的脚本，该脚本将执行配置交易更新以将 ORG3 引入网络。

Also, we will show a “manual” version of the same process, showing each step and explaining what it accomplishes (since we show you how to bring down your network before this manual process, you could also run the script and then

look at each step).

此外，我们还将显示同一流程的“手动”版本，显示每个步骤，并解释它完成了什么（因为我们向您展示了如何在手动流程之前关闭网络，您还可以运行脚本，然后查看每个步骤）。

2、Bring Org3 into the Channel with the Script

2、用脚本将 org3 带入频道

You should be in first-network. To use the script, simply issue the following:
你应该在第一网络。要使用脚本，只需发出以下命令：

```
./eyfn.sh up
```

The output here is well worth reading. You'll see the Org3 crypto material being added, the config update being created and signed, and then chaincode being installed to allow Org3 to execute ledger queries.

这里的输出很值得一读。您将看到正在添加的 ORG3 加密材料、正在创建和签名的配置更新，然后安装链码以允许 ORG3 执行分类帐查询。

If everything goes well, you'll get this message:

如果一切顺利，您将收到以下信息：

```
===== All GOOD, EYFN test execution completed =====
```

eyfn.sh can be used with the same Node.js chaincode and database options as byfn.sh by issuing the following (instead of ./byfn.sh up):

eyfn.sh 可以与 byfn.sh 相同的 node.js chaincode 和数据库选项一起使用，方法是发出以下命令（而不是 ./byfn.sh up）：

```
./byfn.sh up -c testchannel -s couchdb -l node
```

And then:

然后：

```
./eyfn.sh up -c testchannel -s couchdb -l node
```

For those who want to take a closer look at this process, the rest of the doc will show you each command for making a channel update and what it does.

对于那些想更深入地了解这个过程的人，文档的其余部分将向您展示进行通道更新的每个命令以及它的作用。

3、Bring Org3 into the Channel Manually

3、手动将 ORG3 引入频道

Note

注释

The manual steps outlined below assume that the FABRIC_LOGGING_SPEC in the cli and Org3cli containers is set to DEBUG.

下面概述的手动步骤假定 cli 和 org3cli 容器中的 FABRIC_LOGGING_SPEC 为调试。

For the cli container, you can set this by modifying the docker-compose-cli.yaml file in the first-network directory. e.g.

对于 cli 容器，可以通过修改第一个网络目录中的 docker-compose-cli.yaml 文件来设置它。例如

```
cli:
```

```

container_name: cli
image: hyperledger/fabric-tools:$IMAGE_TAG
tty: true
stdin_open: true
environment:
- GOPATH=/opt/gopath
- CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
#- FABRIC_LOGGING_SPEC=INFO
- FABRIC_LOGGING_SPEC=DEBUG

```

For the Org3cli container, you can set this by modifying the docker-compose-org3.yaml file in the first-network directory. e.g.

对于 org3cli 容器，可以通过修改第一个网络目录中的 docker-compose-org3.yaml 文件来设置。例如

```

Org3cli:
container_name: Org3cli
image: hyperledger/fabric-tools:$IMAGE_TAG
tty: true
stdin_open: true
environment:
- GOPATH=/opt/gopath
- CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
#- FABRIC_LOGGING_SPEC=INFO
- FABRIC_LOGGING_SPEC=DEBUG

```

If you've used the eyfn.sh script, you'll need to bring your network down.

This can be done by issuing:

如果使用了 eyfn.sh 脚本，则需要关闭网络。这可以通过发布：

```
./eyfn.sh down
```

This will bring down the network, delete all the containers and undo what we've done to add Org3.

这将关闭网络，删除所有容器，并撤销我们为添加 org3 所做的操作。

When the network is down, bring it back up again.

当网络关闭时，再次将其恢复。

```
./byfn.sh generate
```

Then:

然后：

```
./byfn.sh up
```

This will bring your network back to the same state it was in before you executed the eyfn.sh script.

这将使您的网络恢复到执行 eyfn.sh 脚本之前的状态。

Now we're ready to add Org3 manually. As a first step, we'll need to generate Org3's crypto material.

现在我们准备手动添加 org3。作为第一步，我们需要生成 org3 的加密材料。

4、Generate the Org3 Crypto Material

4、生成 ORG3 加密材料

In another terminal, change into the org3-artifacts subdirectory from first-network.

在另一个终端中，从第一个网络切换到 org3 工件子目录。

```
cd org3-artifacts
```

There are two yaml files of interest here: org3-crypto.yaml and configtx.yaml. First, generate the crypto material for Org3:

这里有两个感兴趣的 yaml 文件: org3-crypto.yaml 和 configtx.yaml。首先，生成 org3 的加密材料:

```
../../bin/cryptogen generate --config=./org3-crypto.yaml
```

This command reads in our new crypto yaml file - org3-crypto.yaml - and leverages cryptogen to generate the keys and certificates for an Org3 CA as well as two peers bound to this new Org. As with the BYFN implementation, this crypto material is put into a newly generated crypto-config folder within the present working directory (in our case, org3-artifacts).

这个命令读取我们新的加密 yaml 文件 org3-crypto.yaml，并利用 crypten 为 org3 CA 以及绑定到这个新组织的两个对等机生成密钥和证书。与 byfn 实现一样，这个加密材料被放入当前工作目录（在我们的例子中是 org3 工件）中新生成的 crypto config 文件夹中。

Now use the configtxgen utility to print out the Org3-specific configuration material in JSON. We will preface the command by telling the tool to look in the current directory for the configtx.yaml file that it needs to ingest.

现在，使用 configtxgen 实用程序在 JSON 中打印出 ORG3 特定的配置材料。我们将通过告诉工具在当前目录中查找需要接收的 configtx.yaml 文件来作为命令的开头。

```
export FABRIC_CFG_PATH=$PWD && ../../bin/configtxgen -printOrg  
Org3MSP > ../channel-artifacts/org3.json
```

The above command creates a JSON file - org3.json - and outputs it into the channel-artifacts subdirectory at the root of first-network. This file contains the policy definitions for Org3, as well as three important certificates presented in base 64 format: the admin user certificate (which will be needed to act as the admin of Org3 later on), a CA root cert, and a TLS root cert. In an upcoming step we will append this JSON file to the channel configuration.

上面的命令创建一个 json 文件——org3.json——并将其输出到第一个网络根目录下的 channelartures 子目录中。此文件包含 org3 的策略定义，以及三个基本 64 格式的重要证书：管理用户证书（稍后将需要它作为 org3 的管理员）、CA 根证书和 TLS 根证书。在接下来的步骤中，我们将把此 JSON 文件附加到通道配置中。

Our final piece of housekeeping is to port the Orderer Org's MSP material into the Org3 crypto-config directory. In particular, we are concerned with the Orderer's TLS root cert, which will allow for secure communication between Org3 entities and the network's ordering node.

我们的最后一项维护工作是将订购者组织的 MSP 材料移植到 ORG3 加密配置目录中。特别是，我们关注订购方的 TLS 根证书，这将允许 ORG3 实体和网络订购节点之间的安全通信。

```
cd ../ && cp -r crypto-config/ordererOrganizations org3-artifacts/crypto-  
config/
```

Now we're ready to update the channel configuration...
现在我们准备更新通道配置...

5、Prepare the CLI Environment

5、准备 CLI 环境

The update process makes use of the configuration translator tool - configtxlator. This tool provides a stateless REST API independent of the SDK. Additionally it provides a CLI, to simplify configuration tasks in Fabric networks. The tool allows for the easy conversion between different equivalent data representations/formats (in this case, between protobufs and JSON). Additionally, the tool can compute a configuration update transaction based on the differences between two channel configurations.

更新过程使用了配置转换器工具——configtxlator。此工具提供独立于 SDK 的无状态 REST API。此外，它还提供了一个 CLI，以简化结构网络中的配置任务。该工具允许在不同的等效数据表示/格式（在本例中，Protobufs 和 JSON 之间）之间轻松转换。此外，该工具还可以根据两个通道配置之间的差异计算配置更新交易。

First, exec into the CLI container. Recall that this container has been mounted with the BYFN crypto-config library, giving us access to the MSP material for the two original peer organizations and the Orderer Org. The bootstrapped identity is the Org1 admin user, meaning that any steps where we want to act as Org2 will require the export of MSP-specific environment variables.

首先，在 cli 容器中执行。回想一下，这个容器已经安装了 byfn 加密配置库，使我们能够访问两个原始对等组织和订购方组织的 MSP 材料。引导标识是 org1 管理用户，这意味着我们要充当 org2 的任何步骤都需要导出 MSP 特定的环境变量。

```
docker exec -it cli bash
```

Export the ORDERER_CA and CHANNEL_NAME variables:

设置 order_ca 和 channel_name 变量：

```
export
```

```
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem && export CHANNEL_NAME=mychannel
```

Check to make sure the variables have been properly set:

检查以确保变量设置正确：

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

Note

注释

If for any reason you need to restart the CLI container, you will also need to re-export the two environment variables - ORDERER_CA and CHANNEL_NAME.

如果出于任何原因需要重新启动 CLI 容器，还需要重新导出两个环境变量 - order_ca 和 channel_name。

6、Fetch the Configuration

6、获取配置

Now we have a CLI container with our two key environment variables - ORDERER_CA and CHANNEL_NAME exported. Let's go fetch the most recent config block for the channel - mychannel.

现在我们有了一个cli容器,其中有两个关键的环境变量——order_ca和channel_name。让我们去获取通道的最新配置块——mychannel。

The reason why we have to pull the latest version of the config is because channel config elements are versioned. Versioning is important for several reasons. It prevents config changes from being repeated or replayed (for instance, reverting to a channel config with old CRLs would represent a security risk). Also it helps ensure concurrency (if you want to remove an Org from your channel, for example, after a new Org has been added, versioning will help prevent you from removing both Orgs, instead of just the Org you want to remove).

我们之所以需要获取配置的最新版本,是因为通道配置元素的版本是经过版本控制的。版本控制很重要,有几个原因。它可以防止配置更改被重复或重播(例如,使用旧的crl恢复到通道配置会带来安全风险)。此外,它还有助于确保并发性(如果您希望从通道中删除某个组织,例如,在添加了新的组织之后,版本控制将有助于防止您同时删除这两个组织,而不仅仅是要删除的组织)。

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CHANNEL_NAME --tls --cafile $ORDERER_CA
```

This command saves the binary protobuf channel configuration block to config_block.pb. Note that the choice of name and file extension is arbitrary. However, following a convention which identifies both the type of object being represented and its encoding (protobuf or JSON) is recommended.

此命令将二进制protobuf通道配置块保存到config_block.pb。请注意,名称和文件扩展名的选择是任意的。但是,建议遵循标识所表示对象的类型又标识其编码(protobuf或json)的约定。

When you issued the peer channel fetch command, there was a decent amount of output in the terminal. The last line in the logs is of interest:

当您发出对等通道获取命令时,终端中有相当数量的输出。日志中的最后一行是感兴趣的:

```
2017-11-07 17:17:57.383 UTC [channelCmd] readBlock -> DEBU 011 Received block: 2
```

This is telling us that the most recent configuration block for mychannel is actually block 2, NOT the genesis block. By default, the peer channel fetch config command returns the most recent configuration block for the targeted channel, which in this case is the third block. This is because the BYFN script defined anchor peers for our two organizations - Org1 and Org2 - in two separate channel update transactions.

这告诉我们MyChannel的最新配置块实际上是块2,而不是Genesis块。默认情况下,peer channel fetch config命令返回目标通道的最新配置块,在本例中是第三个块。这是因为byfn脚本在两个单独的通道更新事务中为我们的两个组织(org1和org2)定义了锚定对等体。

As a result, we have the following configuration sequence:

因此,我们有以下配置顺序:

```
block 0: genesis block
block 1: Org1 anchor peer update
block 2: Org2 anchor peer update
```

7、Convert the Configuration to JSON and Trim It Down

7、将配置转换为 JSON 并将其装饰

Now we will make use of the `configtxlator` tool to decode this channel configuration block into JSON format (which can be read and modified by humans). We also must strip away all of the headers, metadata, creator signatures, and so on that are irrelevant to the change we want to make. We accomplish this by means of the `jq` tool:

现在，我们将使用 `configtxlator` 工具将这个通道配置块解码为 JSON 格式（可以由人读取和修改）。我们还必须除去所有与我们想要做的更改无关的头、元数据、创建者签名等等。我们通过 `JQ` 工具来实现这一点：

```
configtxlator proto_decode --input config_block.pb --type common.Block |
jq .data.data[0].payload.data.config > config.json
```

This leaves us with a trimmed down JSON object - `config.json`, located in the `fabric-samples` folder inside `first-network` - which will serve as the baseline for our config update.

这给我们留下了一个修剪过的 json 对象 `config.json`，位于第一个网络中的 `fabric samples` 文件夹中，它将作为配置更新的基线。

Take a moment to open this file inside your text editor of choice (or in your browser). Even after you're done with this tutorial, it will be worth studying it as it reveals the underlying configuration structure and the other kind of channel updates that can be made. We discuss them in more detail in [Updating a Channel Configuration](#).

花点时间在您选择的文本编辑器中（或在浏览器中）打开此文件。即使完成了本教程的学习，也值得学习它，因为它揭示了底层的配置结构和可以进行的其他类型的通道更新。我们在更新通道配置时更详细地讨论它们。

8、Add the Org3 Crypto Material

8、添加 ORG3 加密材料

Note

注释

The steps you've taken up to this point will be nearly identical no matter what kind of config update you're trying to make. We've chosen to add an org with this tutorial because it's one of the most complex channel configuration updates you can attempt.

无论您尝试进行哪种配置更新，到目前为止所采取的步骤都将几乎相同。我们选择在本教程中添加一个组织，因为它是您可以尝试的最复杂的通道配置更新之一。

We'll use the `jq` tool once more to append the Org3 configuration definition - `org3.json` - to the channel's application groups field, and name the output

- modified_config.json.

我们将再次使用 jq 工具将 org3 配置定义(org3.json)附加到通道的应用程序组字段, 并将输出命名为 modified_config.json。

```
jq -s '.[0] * {"channel_group":{"groups":{"Application":{"groups":{"Org3MSP":.[1]}}}}}' config.json ./channel-artifacts/org3.json > modified_config.json
```

Now, within the CLI container we have two JSON files of interest - config.json and modified_config.json. The initial file contains only Org1 and Org2 material, whereas “modified” file contains all three Orgs. At this point it’s simply a matter of re-encoding these two JSON files and calculating the delta.

现在, 在 cli 容器中, 我们有两个感兴趣的 JSON 文件 - config.json 和 modified_config.json。初始文件只包含 org1 和 org2 材料, 而 “修改” 文件包含所有三个 org。此时, 只需重新编码这两个 JSON 文件并计算 delta。

First, translate config.json back into a protobuf called config.pb:

首先, 将 config.json 转换回名为 config.pb 的 protobuf:

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

Next, encode modified_config.json to modified_config.pb:

接下来, 将 modified_config.json 编码为 modified_config.pb:

```
configtxlator proto_encode --input modified_config.json --type common.Config --output modified_config.pb
```

Now use configtxlator to calculate the delta between these two config protobufs. This command will output a new protobuf binary named org3_update.pb:

现在使用 configtxlator 计算这两个配置协议之间的增量。此命令将输出名为 org3_update.pb 的新 protobuf 二进制文件:

```
configtxlator compute_update --channel_id $CHANNEL_NAME --original config.pb --updated modified_config.pb --output org3_update.pb
```

This new proto - org3_update.pb - contains the Org3 definitions and high level pointers to the Org1 and Org2 material. We are able to forgo the extensive MSP material and modification policy information for Org1 and Org2 because this data is already present within the channel’s genesis block. As such, we only need the delta between the two configurations.

这个新的协议——org3_update.pb——包含 org3 定义和指向 org1 和 org2 材料的高级指针。我们可以放弃 org1 和 org2 的大量 MSP 材料和修改策略信息, 因为这些数据已经存在于通道的 Genesis 块中。因此, 我们只需要两种配置之间的增量。

Before submitting the channel update, we need to perform a few final steps. First, let’s decode this object into editable JSON format and call it org3_update.json:

在提交频道更新之前, 我们需要执行一些最后的步骤。首先, 让我们将此对象解码为可编辑的 JSON 格式, 并将其命名为 org3_update.json:

```
configtxlator proto_decode --input org3_update.pb --type common.ConfigUpdate | jq . > org3_update.json
```

Now, we have a decoded update file - org3_update.json - that we need to

wrap in an envelope message. This step will give us back the header field that we stripped away earlier. We'll name this file `org3_update_in_envelope.json`:

现在,我们有了一个解码后的更新文件——`org3_update.json`——我们需要将其封装在一个信封消息中。这一步将返回前面剥离的标题字段。我们将把这个文件命名为 `org3_update_in_envelope.json`:

```
echo      '{"payload":{"header":{"channel_header":{"channel_id":"mychannel",
"type":2}}, "data":{"config_update":"'$(cat  org3_update.json)'"}}}' | jq . >
org3_update_in_envelope.json
```

Using our properly formed JSON - `org3_update_in_envelope.json` - we will leverage the `configtxlator` tool one last time and convert it into the fully fledged protobuf format that Fabric requires. We'll name our final update object `org3_update_in_envelope.pb`:

使用我们正确形成的 `json-org3-update-in-envelope.json`, 我们将最后一次利用 `configtxlator` 工具, 并将其转换为结构所需的完全成熟的 `protobuf` 格式。我们将在 `envelope.pb` 中命名最终更新对象 `org3_update`:

```
configtxlator proto_encode --input  org3_update_in_envelope.json --type
common.Envelope --output org3_update_
```

9、Sign and Submit the Config Update

9、签署并提交配置更新

Almost done!

差不多完成了!

We now have a `protobuf` binary - `org3_update_in_envelope.pb` - within our CLI container. However, we need signatures from the requisite Admin users before the config can be written to the ledger. The modification policy (`mod_policy`) for our channel Application group is set to the default of "MAJORITY", which means that we need a majority of existing org admins to sign it. Because we have only two orgs - `Org1` and `Org2` - and the majority of two is two, we need both of them to sign. Without both signatures, the ordering service will reject the transaction for failing to fulfill the policy.

现在,我们的cli容器中有一个`protobuf`二进制文件——`org3_update_in_envelope.pb`。但是,在将配置写入分类帐之前,我们需要来自必需的管理员用户的签名。我们的频道应用程序组的修改策略(`mod_policy`)设置为默认的“多数”,这意味着我们需要大多数现有的组织管理员来签署它。因为我们只有两个组织——`org1` 和 `org2`——而其中大多数是两个组织,所以我们需要两个组织都签名。如果没有这两个签名,订购服务将拒绝未满足策略的事务。

First, let's sign this update proto as the `Org1 Admin`. Remember that the CLI container is bootstrapped with the `Org1 MSP` material, so we simply need to issue the `peer channel signconfigtx` command:

首先,让我们以 `org1` 管理员的身份签署这个更新协议。记住,cli容器是用 `org1 msp` 材料引导的,因此我们只需要发出 `peer channel signconfigtx` 命令:

```
peer channel signconfigtx -f org3_update_in_envelope.pb
```

The final step is to switch the CLI container's identity to reflect the

Org2 Admin user. We do this by exporting four environment variables specific to the Org2 MSP.

最后一步是切换 cli 容器的标识以反映 org2 管理用户。我们通过导出特定于 ORG2MSP 的四个环境变量来实现这一点。

Note

注释

Switching between organizations to sign a config transaction (or to do anything else) is not reflective of a real-world Fabric operation. A single container would never be mounted with an entire network's crypto material. Rather, the config update would need to be securely passed out-of-band to an Org2 Admin for inspection and approval.

在组织之间切换以签署配置事务（或执行其他任何操作）并不能反映真实的结构操作。一个容器永远不会与整个网络的加密材料一起安装。相反，配置更新需要安全地传递给 org2 管理员进行检查和批准。

Export the Org2 environment variables:

导出 org2 环境变量：

```
# you can issue all of these commands at once
```

```
export CORE_PEER_LOCALMSPID="Org2MSP"
```

```
export
```

```
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

```
export
```

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
```

```
export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
```

Lastly, we will issue the peer channel update command. The Org2 Admin signature will be attached to this call so there is no need to manually sign the protobuf a second time:

最后，我们将发出对等通道更新命令。ORG2 管理员签名将附加到此呼叫，因此无需再次手动签名协议：

Note

注释

The upcoming update call to the ordering service will undergo a series of systematic signature and policy checks. As such you may find it useful to stream and inspect the ordering node's logs. From another shell, issue a `docker logs -f orderer.example.com` command to display them.

即将对订购服务进行的更新调用将接受一系列系统签名和策略检查。因此，您可能会发现流式处理和检查排序节点的日志很有用。从另一个 shell 中，发出 `docker logs -f order.example.com` 命令来显示它们。

Send the update call:

发送更新呼叫：

```
peer channel update -f org3_update_in_envelope.pb -c $CHANNEL_NAME -o orderer.example.com:7050 --tls --cafile $ORDERER_CA
```


You should see a message digest indication similar to the following if your update has been submitted successfully:

如果已成功提交更新，则应看到类似以下内容的消息摘要指示：

```
2018-02-24 18:56:33.499 UTC [msp/identity] Sign -> DEBU 00f Sign: digest:
3207B24E40DE2FAB87A2E42BC004FEAA1E6FDCA42977CB78C64F05A88E556ABA
```

You will also see the submission of our configuration transaction:

您还将看到我们的配置事务的提交：

```
2018-02-24 18:56:33.499 UTC [channelCmd] update -> INFO 010 Successfully
submitted channel update
```

The successful channel update call returns a new block - block 5 - to all of the peers on the channel. If you remember, blocks 0-2 are the initial channel configurations while blocks 3 and 4 are the instantiation and invocation of the mycc chaincode. As such, block 5 serves as the most recent channel configuration with Org3 now defined on the channel.

成功的通道更新调用将向通道上的所有对等方返回一个新的块（块 5）。如果您还记得，块 0-2 是初始通道配置，而块 3 和 4 是 mycc 链码的实例化和调用。因此，块 5 是最新的通道配置，ORG3 现在定义在通道上。

[Inspect the logs for peer0.org1.example.com:](#)

```
docker logs -f peer0.org1.example.com
```

Follow the demonstrated process to fetch and decode the new config block if you wish to inspect its contents.

如果您希望检查新配置块的内容，请按照演示的过程获取并解码它。

10、Configuring Leader Election

10、配置领导人选举

Note

注释

This section is included as a general reference for understanding the leader election settings when adding organizations to a network after the initial channel configuration has completed. This sample defaults to dynamic leader election, which is set for all peers in the network in peer-base.yaml.

在完成初始通道配置后，将组织添加到网络时，此部分作为了解领导人选择设置的一般参考。此示例默认为“动态领导者选举”，在 peer-base.yaml 中为网络中的所有同级设置。

Newly joining peers are bootstrapped with the genesis block, which does not contain information about the organization that is being added in the channel configuration update. Therefore new peers are not able to utilize gossip as they cannot verify blocks forwarded by other peers from their own organization until they get the configuration transaction which added the organization to the channel. Newly added peers must therefore have one of the following configurations so that they receive blocks from the ordering service:

新加入的对等机是通过 Genesis 块引导的，Genesis 块不包含有关要在通道配置更新中添加的组织的信息。因此，新的对等方无法利用流言，因为他们无法验证其他对等方从自己的组织转发的块，直到他们获得将组织添加到通道的配置事务。因此，新添加的对等机必须

具有以下配置之一，以便它们接收来自订购服务的块：

1. To utilize static leader mode, configure the peer to be an organization leader:

1. 要使用静态领导模式，请将对等机配置为组织领导：

```
CORE_PEER_GOSSIP_USELEADERELECTION=false
```

```
CORE_PEER_GOSSIP_ORGLEADER=true
```

Note

注释

This configuration must be the same for all new peers added to the channel.

对于添加到通道的所有新对等端，此配置必须相同。

2. To utilize dynamic leader election, configure the peer to use leader election:

2. 要使用动态领导选择，请将对等机配置为使用领导选择：

```
CORE_PEER_GOSSIP_USELEADERELECTION=true
```

```
CORE_PEER_GOSSIP_ORGLEADER=false
```

Note

注释

Because peers of the newly added organization won't be able to form membership view, this option will be similar to the static configuration, as each peer will start proclaiming itself to be a leader. However, once they get updated with the configuration transaction that adds the organization to the channel, there will be only one active leader for the organization. Therefore, it is recommended to leverage this option if you eventually want the organization's peers to utilize leader election.

由于新添加的组织对等方将无法形成成员视图，因此此选项将类似于静态配置，因为每个对等方将开始声明自己是领导者。但是，一旦他们使用将组织添加到通道的配置事务进行了更新，组织将只有一个活动的领导。因此，如果您最终希望组织的同事利用领导人选举，建议利用此选项。

11、Join Org3 to the Channel

11、将 org3 加入频道

At this point, the channel configuration has been updated to include our new organization - Org3 - meaning that peers attached to it can now join mychannel.

此时，通道配置已经更新为包含我们的新组织 ORG3，这意味着与之相连的对等方现在可以加入 MyChannel。

First, let's launch the containers for the Org3 peers and an Org3-specific CLI.

首先，让我们为 ORG3 对等机和特定于 ORG3 的 CLI 启动容器。

Open a new terminal and from first-network kick off the Org3 docker compose:

打开一个新的终端，从第一个网络启动 org3 docker compose:

```
docker-compose -f docker-compose-org3.yaml up -d
```

This new compose file has been configured to bridge across our initial network, so the two peers and the CLI container will be able to resolve with the

existing peers and ordering node. With the three new containers now running, exec into the Org3-specific CLI container:

这个新的 compose 文件已经配置为跨我们的初始网络桥接, 因此这两个对等端和 cli 容器将能够与现有的对等端和排序节点进行解析。当三个新容器正在运行时, 执行到 ORG3 特定的 CLI 容器中:

```
docker exec -it Org3cli bash
```

Just as we did with the initial CLI container, export the two key environment variables: ORDERER_CA and CHANNEL_NAME:

正如我们对初始 cli 容器所做的那样, 导出两个关键环境变量: order_ca 和 channel_name:

```
export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem && export CHANNEL_NAME=mychannel
```

Check to make sure the variables have been properly set:

检查以确保变量设置正确:

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

Now let's send a call to the ordering service asking for the genesis block of mychannel. The ordering service is able to verify the Org3 signature attached to this call as a result of our successful channel update. If Org3 has not been successfully appended to the channel config, the ordering service should reject this request.

现在, 让我们给订购服务部门打个电话, 询问 MyChannel 的 Genesis 块。订购服务能够验证我们成功更新频道后附加到此呼叫的 ORG3 签名。如果 org3 未成功附加到通道配置, 订购服务应拒绝此请求。

Note

注释

Again, you may find it useful to stream the ordering node's logs to reveal the sign/verify logic and policy checks.

同样, 您可能会发现流化排序节点的日志以显示签名/验证逻辑和策略检查很有用。

Use the peer channel fetch command to retrieve this block:

使用 peer channel fetch 命令检索此块:

```
peer channel fetch 0 mychannel.block -o orderer.example.com:7050 -c $CHANNEL_NAME --tls --cafile $ORDERER_CA
```

Notice, that we are passing a 0 to indicate that we want the first block on the channel's ledger (i.e. the genesis block). If we simply passed the peer channel fetch config command, then we would have received block 5 - the updated config with Org3 defined. However, we can't begin our ledger with a downstream block - we must start with block 0.

注意, 我们要传递 0 来表示我们想要频道分类账上的第一个区块 (即 Genesis 区块)。如果我们简单地传递了 peer channel fetch config 命令, 那么我们将收到块 5——定义了 org3 的更新配置。但是, 我们不能从下游区块开始分类账-我们必须从 0 区块开始。

Issue the peer channel join command and pass in the genesis block - mychannel.block:

发出 `peer channel join` 命令并在 `genesis` 块 - `mychannel.block` 中传递:

```
peer channel join -b mychannel.block
```

If you want to join the second peer for Org3, export the TLS and ADDRESS variables and reissue the peer channel join command:

如果要加入 ORG3 的第二个对等端,请导出 `tls` 和地址变量,然后重新发出 `peer channel join` 命令:

```
export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org3.example.com/peers/peer1.org3.example.com/tls/ca.c
rt && export CORE_PEER_ADDRESS=peer1.org3.example.com:7051
peer channel join -b mychannel.block
```

12、Upgrade and Invoke Chaincode

12、升级并调用链码

The final piece of the puzzle is to increment the chaincode version and update the endorsement policy to include Org3. Since we know that an upgrade is coming, we can forgo the futile exercise of installing version 1 of the chaincode. We are solely concerned with the new version where Org3 will be part of the endorsement policy, therefore we'll jump directly to version 2 of the chaincode.

最后一个难题是增加链码版本并更新背书策略以包含 `org3`。既然我们知道升级即将到来,我们就可以放弃安装 chaincode 版本 1 的徒劳练习。我们只关注新版本,其中 `org3` 将成为背书策略的一部分,因此我们将直接跳转到 chaincode 的版本 2。

From the Org3 CLI:

从 `org3 cli`:

```
peer chaincode install -n mycc -v 2.0 -p
github.com/chaincode/chaincode_example02/go/
```

Modify the environment variables accordingly and reissue the command if you want to install the chaincode on the second peer of Org3. Note that a second installation is not mandated, as you only need to install chaincode on peers that are going to serve as endorsers or otherwise interface with the ledger (i.e. query only). Peers will still run the validation logic and serve as committers without a running chaincode container.

相应地修改环境变量,如果要在 ORG3 的第二个对等机上安装链码,请重新发出该命令。请注意,第二次安装不是强制的,因为您只需要在将用作背书人的对等机上安装链码,或者在与分类帐的其他接口上安装链码(即仅查询)。对等方仍将运行验证逻辑,并且在没有运行链码容器的情况下充当提交者。

Now jump back to the original CLI container and install the new version on the Org1 and Org2 peers. We submitted the channel update call with the Org2 admin identity, so the container is still acting on behalf of `peer0.org2`:

现在跳回到原来的 `cli` 容器,并在 `org1` 和 `org2` 对等机上安装新版本。我们使用 `org2` 管理标识提交了通道更新调用,因此容器仍代表 `peer0.org2`:

```
peer chaincode install -n mycc -v 2.0 -p
github.com/chaincode/chaincode_example02/go/
```

Flip to the peer0.org1 identity:

翻到 peer0.org1 标识:

```
export CORE_PEER_LOCALMSPID="Org1MSP"
export
```

```
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.c
rt
```

```
export
```

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/cryp
to/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
```

```
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
```

And install again:

然后重新安装:

```
peer chaincode install -n mycc -v 2.0 -p
github.com/chaincode/chaincode_example02/go/
```

Now we're ready to upgrade the chaincode. There have been no modifications to the underlying source code, we are simply adding Org3 to the endorsement policy for a chaincode - mycc - on mychannel.

现在我们准备升级 chaincode。没有对底层源代码进行任何修改，我们只是将 org3 添加到 mychannel 上的链式代码 mycc 的认可策略中。

Note

注释

Any identity satisfying the chaincode's instantiation policy can issue the upgrade call. By default, these identities are the channel Admins.

满足链码实例化策略的任何标识都可以发出升级调用。默认情况下，这些标识是频道管理员。

Send the call:

发送呼叫:

```
peer chaincode upgrade -o orderer.example.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -C $CHANNEL_NAME -n mycc -v 2.0 -c
'{"Args":["init","a","90","b","210"]}' -P "OR ('Org1MSP.peer', 'Org2MSP.peer', 'Org3MSP.peer')"
```

You can see in the above command that we are specifying our new version by means of the v flag. You can also see that the endorsement policy has been modified to -P "OR ('Org1MSP.peer', 'Org2MSP.peer', 'Org3MSP.peer')", reflecting the addition of Org3 to the policy. The final area of interest is our constructor request (specified with the c flag).

您可以在上面的命令中看到，我们正通过 v 标志来指定新版本。您还可以看到背书策略已修改为 -p “或 ('org1MSP.peer', 'org2MSP.peer', 'org3MSP.peer')”，反映了将 org3 添加到策略中。最后一个感兴趣的领域是我们的构造函数请求（用 C 标志指定）。

As with an instantiate call, a chaincode upgrade requires usage of the init method. If your chaincode requires arguments be passed to the init method, then you will need to do so here.

与实例化调用一样，链代码升级需要使用 init 方法。如果您的 chaincode 需要将参数传递给 init 方法，那么您需要在这里这样做。

The upgrade call adds a new block - block 6 - to the channel's ledger and allows for the Org3 peers to execute transactions during the endorsement phase. Hop back to the Org3 CLI container and issue a query for the value of a. This will take a bit of time because a chaincode image needs to be built for the targeted peer, and the container needs to start:

升级调用将一个新的块（块 6）添加到通道的分类账中，并允许 ORG3 对等方在认可阶段执行事务。返回到 org3 cli 容器并发出值 a 的查询。这将花费一点时间，因为需要为目标对等机生成链码映像，并且容器需要启动：

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

We should see a response of Query Result: 90.

我们应该看到查询结果的响应：90。

Now issue an invocation to move 10 from a to b:

现在发出一个调用，将 10 从 A 移动到 B:

```
peer chaincode invoke -o orderer.example.com:7050 --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -C $CHANNEL_NAME -n mycc -c
'{"Args":["invoke","a","b","10"]}'
```

Query one final time:

最后一次查询：

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

We should see a response of Query Result: 80, accurately reflecting the update of this chaincode's world state.

我们应该看到查询结果的响应：80，准确地反映了这个链代码的世界状态的更新。

13、Conclusion

13、结论

The channel configuration update process is indeed quite involved, but there is a logical method to the various steps. The endgame is to form a delta transaction object represented in protobuf binary format and then acquire the requisite number of admin signatures such that the channel configuration update transaction fulfills the channel's modification policy.

通道配置更新过程确实非常复杂，但是各个步骤都有一个逻辑方法。最终游戏是形成一个以 protobuf 二进制格式表示的 delta 事务对象，然后获取所需数量的管理签名，以便通道配置更新事务满足通道的修改策略。

The configtxlator and jq tools, along with the ever-growing peer channel commands, provide us with the functionality to accomplish this task.

configtxlator 和 jq 工具以及不断增长的对等通道命令为我们提供了完成此任务的功能。

五、Upgrading Your Network Components

五、升级网络组件

Note
注释

When we use the term “upgrade” in this documentation, we’re primarily referring to changing the version of a component (for example, going from a v1.3 binary to a v1.4 binary). The term “update,” on the other hand, refers not to versions but to configuration changes, such as updating a channel configuration or a deployment script. As there is no data migration, technically speaking, in Fabric, we will not use the term “migration” or “migrate” here.

当我们在本文档中使用术语“升级”时，我们主要指的是更改组件的版本（例如，从 v1.3 二进制文件转换为 v1.4 二进制文件）。另一方面，术语“更新”不是指版本，而是指配置更改，例如更新通道配置或部署脚本。因为在结构中没有数据迁移，从技术上讲，我们在这里不会使用术语“migration”或“migrate”。

Note

注释

Also, if your network is not yet at Fabric v1.3, follow the instructions for Upgrading Your Network to v1.3. The instructions in this documentation only cover moving from v1.3 to v1.4, not from any other version to v1.4.

另外，如果您的网络还没有在 Fabric v1.3 上，请按照将网络升级到 v1.3 的说明操作。本文档中的说明仅涵盖从 v1.3 版移动到 v1.4 版，而不是从任何其他版本移动到 v1.4 版。

1、Overview

1、概述

Because the Building Your First Network (BYFN) tutorial defaults to the “latest” binaries, if you have run it since the release of v1.4, your machine will have v1.4 binaries and tools installed on it and you will not be able to upgrade them.

因为构建您的第一个网络（byfn）教程默认为“最新”的二进制文件，如果您自 v1.4 发布以来就运行它，那么您的计算机上将安装 v1.4 二进制文件和工具，并且您将无法升级它们。

As a result, this tutorial will provide a network based on Hyperledger Fabric v1.3 binaries as well as the v1.4 binaries you will be upgrading to.

因此，本教程将提供一个基于 HyperledgerFabricv1.3 二进制文件以及要升级到的 v1.4 二进制文件的网络。

At a high level, our upgrade tutorial will perform the following steps:

在较高的层次上，我们的升级教程将执行以下步骤：

Backup the ledger and MSPs.

备份分类帐和 MSP。

Upgrade the orderer binaries to Fabric v1.4.

将 Orderer 二进制文件升级到 Fabric v1.4。

Upgrade the peer binaries to Fabric v1.4.

将对等二进制文件升级到 Fabric v1.4。

Note

注释

There are no new Capability Requirements in v1.4. As a result, we do not have to update any channel configurations as part of an upgrade to v1.4.

v1.4 中没有新的功能需求。因此，作为 v1.4 升级的一部分，我们不需要更新任何通道配置。

This tutorial will demonstrate how to perform each of these steps individually with CLI commands. We will also describe how the CLI tools image can be updated.

本教程将演示如何使用 CLI 命令分别执行这些步骤。我们还将描述如何更新 CLI 工具映像。

Note

注释

Because BYFN uses a “SOLO” ordering service (one orderer), our script brings down the entire network. However, in production environments, the orderers and peers can be upgraded simultaneously and on a rolling basis. In other words, you can upgrade the binaries in any order without bringing down the network.

因为 byfn 使用“单独”订购服务（一个订购者），所以我们的脚本会关闭整个网络。但是，在生产环境中，订购方和对等方可以同时进行滚动升级。换句话说，您可以在不关闭网络的情况下以任何顺序升级二进制文件。

Because BYFN is not compatible with the following components, our script for upgrading BYFN will not cover them:

由于 byfn 与以下组件不兼容，因此用于升级 byfn 的脚本将不覆盖这些组件：

Fabric CA

Kafka

CouchDB

SDK

The process for upgrading these components — if necessary — will be covered in a section following the tutorial. We will also show how to upgrade the Node chaincode shim.

如有必要，升级这些组件的过程将在本教程后面的一节中介绍。我们还将演示如何升级节点链代码填充程序。

From an operational perspective, it’s worth noting that the process for gathering logs has changed in v1.4, from CORE_LOGGING_LEVEL (for the peer) and ORDERER_GENERAL_LOGLEVEL (for the orderer) to FABRIC_LOGGING_SPEC (the new operations service). For more information, check out the Fabric release notes.

从操作的角度来看，值得注意的是，收集日志的过程在 v1.4 中已经发生了变化，从核心日志级别（对等）和订购方常规日志级别（订购方）到结构日志级别（新的操作服务）。有关更多信息，请参阅 Fabric 发行说明。

Prerequisites

先决条件

If you haven’t already done so, ensure you have all of the dependencies on your machine as described in Prerequisites.

如果您还没有这样做，请确保您的机器上有所有的依赖项，如先决条件中所述。

2、Launch a v1.3 network

2、启动 v1.3 网络

Before we can upgrade to v1.4, we must first provision a network running

Fabric v1.3 images.

在升级到 v1.4 之前，我们必须首先提供一个运行 Fabricv1.3 映像的网络。

Just as in the BYFN tutorial, we will be operating from the first-network subdirectory within your local clone of fabric-samples. Change into that directory now. You will also want to open a few extra terminals for ease of use.

正如有 byfn 教程中一样，我们将从结构示例的本地克隆中的第一个子目录开始操作。现在转到那个目录。为了方便使用，您还需要打开一些额外的终端。

Clean up

清理

We want to operate from a known state, so we will use the byfn.sh script to kill any active or stale docker containers and remove any previously generated artifacts. Run:

我们希望能从一个已知的状态进行操作，因此我们将使用 byfn.sh 脚本杀死任何活动或过时的 docker 容器，并删除任何以前生成的工件。运行：

```
./byfn.sh down
```

Generate the crypto and bring up the network

生成加密并启动网络

With a clean environment, launch our v1.3 BYFN network using these four commands:

在干净的环境中，使用以下四个命令启动我们的 v1.3 byfn 网络：

```
git fetch origin
```

```
git checkout v1.3.0
```

```
./byfn.sh generate
```

```
./byfn.sh up -t 3000 -i 1.3.0
```

Note

注释

If you have locally built v1.3 images, they will be used by the example. If you get errors, please consider cleaning up your locally built v1.3 images and running the example again. This will download v1.3 images from docker hub.

如果您已经在本地构建了 v1.3 映像，那么示例将使用它们。如果出现错误，请考虑清理本地构建的 v1.3 映像并再次运行该示例。这将从 Docker Hub 下载 v1.3 图像。

If BYFN has launched properly, you will see:

如果 BYFN 已正确启动，您将看到：

```
===== All GOOD, BYFN execution completed =====
```

We are now ready to upgrade our network to Hyperledger Fabric v1.4.

我们现在准备将我们的网络升级到 Hyperledger Fabric v1.4。

Get the newest samples

获取最新样本

Note

注释

The instructions below pertain to whatever is the most recently published version of v1.4.x. Please substitute 1.4.x with the version identifier of the published release that you are testing. In other words, replace ‘1.4.x’ with ‘1.4.0’

if you are testing the first release.

下面的说明适用于最新发布的 v1.4.x 版本。请用正在测试的已发布版本的版本标识符替换 1.4.x。换句话说，如果您正在测试第一个版本，请将“1.4.x”替换为“1.4.0”。

Before completing the rest of the tutorial, it's important to get the v1.4.x version of the samples, you can do this by issuing:

在完成本教程的其余部分之前，获取样本的 v1.4.x 版本是很重要的，您可以通过发布：

```
git fetch origin
git checkout v1.4.x
```

Want to upgrade now?

想现在升级吗？

We have a script that will upgrade all of the components in BYFN as well as enable any capabilities (note, no new capabilities are required for v1.4). If you are running a production network, or are an administrator of some part of a network, this script can serve as a template for performing your own upgrades.

我们有一个脚本，它将升级 byfn 中的所有组件，并启用任何功能（注意，v1.4 不需要新功能）。如果您运行的是生产网络，或者是网络某些部分的管理员，则此脚本可以用作执行自己升级的模板。

Afterwards, we will walk you through the steps in the script and describe what each piece of code is doing in the upgrade process.

然后，我们将引导您完成脚本中的步骤，并描述每段代码在升级过程中所做的工作。

To run the script, issue these commands:

要运行脚本，请发出以下命令：

```
# Note, replace '1.4.x' with a specific version, for example '1.4.0'.
# Don't pass the image flag '-i 1.4.x' if you prefer to default to 'latest'
images.
./byfn.sh upgrade -i 1.4.x
```

If the upgrade is successful, you should see the following:

如果升级成功，您将看到以下内容：

```
===== All GOOD, End-2-End UPGRADE Scenario execution
completed =====
```

If you want to upgrade the network manually, simply run ./byfn.sh down again and perform the steps up to — but not including — ./byfn.sh upgrade -i 1.4.x. Then proceed to the next section.

如果您想手动升级网络，只需再次运行 ./byfn.sh，然后执行到 ./byfn.sh upgrade -i 1.4.x 的步骤，但不包括 ./byfn.sh upgrade -i 1.4.x。然后继续下一节。

Note

注释

Many of the commands you'll run in this section will not result in any output.

In general, assume no output is good output.

在本节中运行的许多命令不会产生任何输出。一般来说，假设没有输出是好的输出。

3、Upgrade the orderer containers

3、升级订购方容器

Orderer containers should be upgraded in a rolling fashion (one at a time).
At a high level, the orderer upgrade process goes as follows:

订购方容器应以滚动方式升级（一次一个）。从高层来看，订购方升级过程如下：

Stop the orderer.

停止订购者。

Back up the orderer's ledger and MSP.

备份订购者的分类帐和 MSP。

Restart the orderer with the latest images.

使用最新图像重新启动医嘱程序。

Verify upgrade completion.

验证升级完成。

As a consequence of leveraging BYFN, we have a solo orderer setup, therefore, we will only perform this process once. In a Kafka setup, however, this process will have to be repeated on each orderer.

由于利用了 byfn, 我们有一个单独的订购者设置, 因此, 我们将只执行此过程一次。但是, 在卡夫卡设置中, 每个订购者都必须重复此过程。

Note

注释

This tutorial uses a docker deployment. For native deployments, replace the file orderer with the one from the release artifacts. Backup the orderer.yaml and replace it with the orderer.yaml file from the release artifacts. Then port any modified variables from the backed up orderer.yaml to the new one. Utilizing a utility like diff may be helpful.

本教程使用 Docker 部署。对于本机部署, 将文件排序器替换为发布工件中的文件排序器。备份 order.yaml, 并将其替换为版本工件中的 order.yaml 文件。然后将任何修改过的变量从备份的 order.yaml 移植到新的 order.yaml。使用类似 diff 的实用程序可能会有所帮助。

Let's begin the upgrade process by bringing down the orderer:

让我们通过关闭订购方来开始升级过程:

```
docker stop orderer.example.com
export LEDGERS_BACKUP=./ledgers-backup
# Note, replace '1.4.x' with a specific version, for example '1.4.0'.
# Set IMAGE_TAG to 'latest' if you prefer to default to the images tagged
'latest' on your system.
export IMAGE_TAG=$(go env GOARCH)-1.4.x
```

We have created a variable for a directory to put file backups into, and exported the IMAGE_TAG we'd like to move to.

我们已经为一个目录创建了一个变量, 用于将文件备份放入其中, 并导出了要移动到的映像标记。

Once the orderer is down, you'll want to backup its ledger and MSP:

一旦订购者关闭, 您将需要备份其分类帐和 MSP:

```
mkdir -p $LEDGERS_BACKUP
docker cp orderer.example.com:/var/hyperledger/production/orderer/ ./ $LEDGERS_BACKUP/orderer.example.com
```

In a production network this process would be repeated for each of the Kafka-based orderers in a rolling fashion.

在一个生产网络中，这个过程将以滚动的方式为每个基于卡夫卡的订购者重复。

Now download and restart the orderer with our new fabric image:

现在下载并用我们的新结构映像重新启动订购程序：

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps
orderer.example.com
```

Because our sample uses a “solo” ordering service, there are no other orderers in the network that the restarted orderer must sync up to. However, in a production network leveraging Kafka, it will be a best practice to issue peer channel fetch <blocknumber> after restarting the orderer to verify that it has caught up to the other orderers.

因为我们的示例使用“单独”订购服务，所以网络中没有重新启动的订购者必须同步的其他订购者。但是，在利用 Kafka 的生产网络中，在重新启动医嘱者以验证它是否赶上了其他医嘱者之后，发出对等通道 fetch<blocknumber>将是最佳实践。

4、Upgrade the peer containers

4、升级对等容器

Next, let's look at how to upgrade peer containers to Fabric v1.4. Peer containers should, like the orderers, be upgraded in a rolling fashion (one at a time). As mentioned during the orderer upgrade, orderers and peers may be upgraded in parallel, but for the purposes of this tutorial we've separated the processes out. At a high level, we will perform the following steps:

接下来，让我们看看如何将对等容器升级到 Fabric v1.4。对等容器应该像订购者一样，以滚动方式升级（一次一个）。正如在医嘱程序升级过程中提到的，医嘱程序和对等程序可以并行升级，但是为了本教程的目的，我们已经将流程分离出来了。从高层次上讲，我们将执行以下步骤：

Stop the peer.

停止节点。

Back up the peer's ledger and MSP.

备份同伴的分类账和 MSP。

Remove chaincode containers and images.

删除链码容器和图像。

Restart the peer with latest image.

用最新的映像重新启动对等机。

Verify upgrade completion.

验证升级完成。

We have four peers running in our network. We will perform this process once for each peer, totaling four upgrades.

我们的网络中有四个对等机。我们将为每个对等端执行此过程一次，共四次升级。

Note

注释

Again, this tutorial utilizes a docker deployment. For native deployments,

replace the file peer with the one from the release artifacts. Backup your core.yaml and replace it with the one from the release artifacts. Port any modified variables from the backed up core.yaml to the new one. Utilizing a utility like diff may be helpful.

同样，本教程使用 Docker 部署。对于本机部署，将文件对等端替换为发布工件中的文件对等端。备份 core.yaml 并将其替换为发布工件中的版本。将任何修改过的变量从 backed up core.yaml 移植到新的 core.yaml。使用类似 diff 的实用程序可能会有所帮助。

Let's bring down the first peer with the following command:

让我们用以下命令关闭第一个对等机：

```
export PEER=peer0.org1.example.com
docker stop $PEER
```

We can then backup the peer's ledger and MSP:

然后，我们可以备份对等方的分类账和 MSP：

```
mkdir -p $LEDGERS_BACKUP
docker cp $PEER:/var/hyperledger/production ./$LEDGERS_BACKUP/$PEER
```

With the peer stopped and the ledger backed up, remove the peer chaincode containers:

停止对等机并备份分类帐后，删除对等机链码容器：

```
CC_CONTAINERS=$(docker ps | grep dev-$PEER | awk '{print $1}')
if [ -n "$CC_CONTAINERS" ] ; then docker rm -f $CC_CONTAINERS ; fi
```

And the peer chaincode images:

以及对等链码图像：

```
CC_IMAGES=$(docker images | grep dev-$PEER | awk '{print $1}')
if [ -n "$CC_IMAGES" ] ; then docker rmi -f $CC_IMAGES ; fi
```

Now we'll re-launch the peer using the v1.4 image tag:

现在，我们将使用 v1.4 图像标记重新启动对等机：

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps $PEER
docker compose-f docker-compose-cli.yaml up-d--无 deps$peer
```

Note

注释

Although, BYFN supports using CouchDB, we opted for a simpler implementation in this tutorial. If you are using CouchDB, however, issue this command instead of the one above:

尽管 byfn 支持使用 couchdb，但我们在本教程中选择了更简单的实现。但是，如果您使用的是 couchdb，请发出此命令，而不是上面的命令：

```
docker-compose -f docker-compose-cli.yaml -f docker-compose-couch.yaml up -d --no-deps $PEER
```

Note

注释

You do not need to relaunch the chaincode container. When the peer gets a request for a chaincode, (invoke or query), it first checks if it has a copy of that chaincode running. If so, it uses it. Otherwise, as in this case, the peer launches the chaincode (rebuilding the image if required).

您不需要重新启动 chaincode 容器。当对等端收到一个链码请求（调用或查询）时，它首先

检查它是否运行了链码的副本。如果是这样，它就使用它。否则，在本例中，对等机启动链码（如果需要，则重建图像）。

Verify peer upgrade completion

验证对等升级完成

We've completed the upgrade for our first peer, but before we move on let's check to ensure the upgrade has been completed properly with a chaincode invoke.

我们已经完成了第一个对等机的升级，但是在继续之前，让我们检查一下，以确保使用 chaincode 调用正确完成了升级。

Note

注释

Before you attempt this, you may want to upgrade peers from enough organizations to satisfy your endorsement policy. Although, this is only mandatory if you are updating your chaincode as part of the upgrade process. If you are not updating your chaincode as part of the upgrade process, it is possible to get endorsements from peers running at different Fabric versions.

在尝试此操作之前，您可能希望从足够多的组织升级对等方，以满足您的认可策略。但是，只有在升级过程中更新链码时，这才是必需的。如果在升级过程中没有更新您的链码，则可以从运行在不同结构版本的对等机获得认可。

Before we get into the CLI container and issue the invoke, make sure the CLI is updated to the most current version by issuing:

在进入 cli 容器并发出 invoke 之前，请确保通过发出以下命令将 cli 更新为最新版本：

```
docker-compose -f docker-compose-cli.yaml stop cli
```

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps cli
```

If you specifically want the v1.3 version of the CLI, issue:

如果您特别需要 1.3 版的 CLI，请发布：

```
IMAGE_TAG=$(go env GOARCH)-1.3.x docker-compose -f docker-compose-cli.yaml up -d --no-deps cli
```

Once you have the version of the CLI you want, get into the CLI container:

一旦您拥有了所需的 CLI 版本，请进入 CLI 容器：

```
docker exec -it cli bash
```

Now you'll need to set two environment variables — the name of the channel and the name of the ORDERER_CA:

现在，您需要设置两个环境变量-通道名称和订购方名称：

```
CH_NAME=mychannel
```

```
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

Now you can issue the invoke:

现在可以发出调用：

```
peer chaincode invoke -o orderer.example.com:7050 --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:7051 --tlsRootCertFiles
```



```
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt      --tls      --cafile $ORDERER_CA -C $CH_NAME -n mycc -c '{"Args":["invoke","a","b","10"]}'
```

Our query earlier revealed a to have a value of 90 and we have just removed 10 with our invoke. Therefore, a query against a should reveal 80. Let's see:

我们之前的查询显示 a 的值为 90，我们刚刚用 invoke 删除了 10。因此，对 a 的查询应该显示 80。让我们看看：

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

We should see the following:

我们应该看到以下内容：

Query Result: 80

After verifying the peer was upgraded correctly, make sure to issue an exit to leave the container before continuing to upgrade your peers. You can do this by repeating the process above with a different peer name exported.

验证对等机是否正确升级后，确保在继续升级对等机之前发出退出以离开容器。您可以通过使用导出的其他对等名重复上述过程来完成此操作。

```
export PEER=peer1.org1.example.com
export PEER=peer0.org2.example.com
export PEER=peer1.org2.example.com
```

5、Upgrading components BYFN does not support

5、升级组件 byfn 不支持

Although this is the end of our update tutorial, there are other components that exist in production networks that are not compatible with the BYFN sample. In this section, we'll talk through the process of updating them.

虽然这是我们的更新教程的结尾，但是生产网络中存在与 byfn 示例不兼容的其他组件。在本节中，我们将讨论更新它们的过程。

Fabric CA container

Fabric CA 容器

To learn how to upgrade your Fabric CA server, click over to the CA documentation.

要了解如何升级 Fabric CA 服务器，请单击 CA 文档。

Upgrade Node SDK clients

升级节点 sdk 客户端

Note

注释

Upgrade Fabric and Fabric CA before upgrading Node SDK clients. Fabric and Fabric CA are tested for backwards compatibility with older SDK clients. While newer SDK clients often work with older Fabric and Fabric CA releases, they may expose features that are not yet available in the older Fabric and Fabric CA releases, and are not tested for full compatibility.

升级 node sdk 客户机之前升级 fabric 和 fabric ca。Fabric 和 Fabric CA 测试了与旧的 SDK 客户端的向后兼容性。虽然较新的 SDK 客户机通常使用较旧的结构和结构 CA 版本，但

它们可能会公开在较旧的结构和结构 CA 版本中尚不可用的功能，并且没有对其进行完全兼容性测试。

Use NPM to upgrade any Node.js client by executing these commands in the root directory of your application:

通过在应用程序的根目录中执行以下命令，使用 npm 升级任何 node.js 客户机：

```
npm install fabric-client@latest
```

```
npm install fabric-ca-client@latest
```

These commands install the new version of both the Fabric client and Fabric-CA client and write the new versions package.json.

这些命令安装了 Fabric 客户端和 Fabric CA 客户端的新版本，并编写了新版本 package.json。

Upgrading the Kafka cluster

升级 kafka 集群

It is not required, but it is recommended that the Kafka cluster be upgraded and kept up to date along with the rest of Fabric. Newer versions of Kafka support older protocol versions, so you may upgrade Kafka before or after the rest of Fabric.

不需要，但建议升级 Kafka 集群并与其他结构保持最新。Kafka 的较新版本支持较旧的协议版本，因此您可以在结构的其余部分之前或之后升级 Kafka。

If you followed the Upgrading Your Network to v1.3 tutorial, your Kafka cluster should be at v1.0.0. If it isn't, refer to the official Apache Kafka documentation on upgrading Kafka from previous versions to upgrade the Kafka cluster brokers.

如果您按照将网络升级到 v1.3 教程，那么您的 kafka 集群应该是 v1.0.0。如果不是，请参阅有关从以前版本升级 kafka 以升级 kafka 集群代理的官方 apache kafka 文档。

Upgrading Zookeeper

升级 Zookeeper

An Apache Kafka cluster requires an Apache Zookeeper cluster. The Zookeeper API has been stable for a long time and, as such, almost any version of Zookeeper is tolerated by Kafka. Refer to the Apache Kafka upgrade documentation in case there is a specific requirement to upgrade to a specific version of Zookeeper. If you would like to upgrade your Zookeeper cluster, some information on upgrading Zookeeper cluster can be found in the Zookeeper FAQ.

Apache Kafka 集群需要 Apache ZooKeeper 集群。ZooKeeper API 已经稳定了很长时间，因此，卡夫卡几乎可以容忍任何版本的 ZooKeeper。如果需要升级到 ZooKeeper 的特定版本，请参阅 Apache Kafka 升级文档。如果您想升级 ZooKeeper 集群，可以在 ZooKeeper 常见问题解答中找到有关升级 ZooKeeper 集群的一些信息。

Upgrading CouchDB

升级 couchdb

If you are using CouchDB as state database, you should upgrade the peer's CouchDB at the same time the peer is being upgraded. CouchDB v2.2.0 has been tested with Fabric v1.4.

如果使用 couchdb 作为状态数据库，则应在升级对等机的同时升级对等机的 couchdb。CouchDB 2.2.0 版已通过 1.4 版织物测试。

To upgrade CouchDB:

要升级 couchdb:

Stop CouchDB.

停止 CouchDB。

Backup CouchDB data directory.

备份 couchdb 数据目录。

Install CouchDB v2.2.0 binaries or update deployment scripts to use a new Docker image (CouchDB v2.2.0 pre-configured Docker image is provided alongside Fabric v1.4).

安装 couchdb v2.2.0 二进制文件或更新部署脚本以使用新的 docker 映像 (couchdb v2.2.0 预配置的 docker 映像与 fabric v1.4 一起提供)。

Restart CouchDB.

重新启动 couchdb。

Upgrade Node chaincode shim

升级节点链码填充程序

To move to the new version of the Node chaincode shim a developer would need to:

要移动到新版本的节点链代码填充程序，开发人员需要：

Change the level of fabric-shim in their chaincode package.json from 1.3 to 1.4.

将 chaincode package.json 中的结构垫片级别从 1.3 更改为 1.4。

Repackage this new chaincode package and install it on all the endorsing peers in the channel.

重新打包这个新的 chaincode 包，并将其安装在通道中所有认可的对等端上。

Perform an upgrade to this new chaincode. To see how to do this, check out peer chaincode.

执行此新链码的升级。要了解如何做到这一点，请查看对等链代码。

Note

注释

This flow isn't specific to moving from 1.3 to 1.4. It is also how one would upgrade from any incremental version of the node fabric shim.

此流程并不特定于从 1.3 移动到 1.4。这也是如何从节点结构填充程序的任何增量版本升级的。

Upgrade Chaincodes with vendored shim

使用 vendored shim 升级链码

Note

注释

The v1.3.0 shim is compatible with the v1.4 peer, but, it is still best practice to upgrade the chaincode shim to match the current level of the peer.

v1.3.0 填充程序与 v1.4 对等机兼容，但升级链码填充程序以匹配对等机的当前级别仍然是最佳实践。

A number of third party tools exist that will allow you to vendor a chaincode shim. If you used one of these tools, use the same one to update your vendoring and re-package your chaincode.

存在许多第三方工具，允许您提供链码填充程序。如果您使用了这些工具中的一个，请使用相同的工具更新您的销售并重新打包您的链码。

If your chaincode vendors the shim, after updating the shim version, you must install it to all peers which already have the chaincode. Install it with the same name, but a newer version. Then you should execute a chaincode upgrade on each channel where this chaincode has been deployed to move to the new version.

如果您的 chaincode 提供了填充程序，则在更新填充程序版本之后，必须将其安装到已经具有该链代码的所有对等方。安装时使用相同的名称，但版本较新。然后，您应该在部署了该链代码的每个通道上执行链代码升级，以移动到新版本。

If you did not vendor your chaincode, you can skip this step entirely.

如果没有提供链码，则可以完全跳过此步骤。

六、Using Private Data in Fabric

六、在 Fabric 中使用私有数据

This tutorial will demonstrate the use of collections to provide storage and retrieval of private data on the blockchain network for authorized peers of organizations.

本教程将使用到的 demonstrate 提供存储和检索的研究 collections 私人数据在网络节点的可靠方法 blockchain 部授权的组织。

The information in this tutorial assumes knowledge of private data stores and their use cases. For more information, check out Private data.

在本教程 assumes 知识信息的私人数据存储部和它们的使用的用例。方法的更多信息，检查出的私人数据。

The tutorial will take you through the following steps to practice defining, configuring and using private data with Fabric:

在下面的教程将带你一步的做法，对 defining、配置和使用私人数据与 fabric:

- 1、Build a collection definition JSON file
- 1、建立一个清晰的 JSON 文件集
- 2、Read and Write private data using chaincode APIs
- 2、读写数据的 API 和私人使用 chaincode
- 3、Install and instantiate chaincode with a collection
- 3、安装和 chaincode 集与一个实例化
- 4、Store private data
- 4、商店的私人数据
- 5、Query the private data as an authorized peer
- 5、查询的私人授权数据为一点
- 6、Query the private data as an unauthorized peer
- 6、查询的私人数据，作为一 unauthorized 屁
- 7、Purge Private Data
- 7、电磁的私人数据
- 8、Using indexes with private data
- 8、利用指标体系与私人数据
- 9、Additional resources

9、额外的资源

This tutorial will use the marbles private data sample — running on the Building Your First Network (BYFN) tutorial network — to demonstrate how to create, deploy, and use a collection of private data. The marbles private data sample will be deployed to the Building Your First Network (BYFN) tutorial network. You should have completed the task Install Samples, Binaries and Docker Images; however, running the BYFN tutorial is not a prerequisite for this tutorial. Instead the necessary commands are provided throughout this tutorial to use the network. We will describe what is happening at each step, making it possible to understand the tutorial without actually running the sample.

本教程将使用的数据样本 marbles 私人办学的建筑你的第一网络（网络教程 byfn）如何对 demonstrate to create a, deploy, 和使用收集的私人数据。在 marbles 私人数据的样本会对你的 deployed 建材第一网（byfn）网络教程。你应该 completed 安装二进制文件和样本的任务, docker Images); 然而, 运行的是不 byfn 教程本教程中的脱附等温线——instead commands 是必要的 throughout 本教程提供的对使用网络。我们将在什么 happening describe 在每一步, 尽可能的去理解它的制作教程不争的运行的样本。

1、Build a collection definition JSON file

1、建立一个清晰的 JSON 文件集

The first step in privatizing data on a channel is to build a collection definition which defines access to the private data.

在私有化的第一步是建立在一个通道的数据收集到的清晰度, defines 接入到私人数据。

The collection definition describes who can persist data, how many peers the data is distributed to, how many peers are required to disseminate the private data, and how long the private data is persisted in the private database. Later, we will demonstrate how chaincode APIs PutPrivateData and GetPrivateData are used to map the collection to the private data being secured.

世界卫生组织的集合, 可以清晰 describes 坚持数据, 如何对分布在多节点的可靠的数据, 如何对多节点的可靠的数据是需要 disseminate 私募和长的私人数据, 如何在 persisted 在私人数据库。之后, 我们将如何 chaincode putprivatedata demonstrate API 和 getprivatedata 图是用来收集到的数据被 secured 私募。

A collection definition is composed of the following properties:

在一个清晰的组合集的以下属性:

name: Name of the collection.

名称: 名称的集合。

policy: Defines the organization peers allowed to persist the collection data.

该组织的政策: defines 节点的可靠的数据收集到 allowed 坚持。

requiredPeerCount: Number of peers required to disseminate the private data as a condition of the endorsement of the chaincode

requiredpeercount: 数需要对节点的可靠的私人数据 disseminate 作为一部空调部 endorsement chaincode

maxPeerCount: For data redundancy purposes, the number of other peers that

the current endorsing peer will attempt to distribute the data to. If an endorsing peer goes down, these other peers are available at commit time if there are requests to pull the private data.

用途:用于数据 maxpeercount redundancy 的号码,那其他节点的可靠的点对 endorsing 经常会 attempt distribute 今天的数据。如果一个 endorsing 对等节点的可靠的 GOES 下,这些是其他可用的时间,如果有是在提交的要求对公牛的私人数据。

blockToLive: For very sensitive information such as pricing or personal information, this value represents how long the data should live on the private database in terms of blocks. The data will live for this specified number of blocks on the private database and after that it will get purged, making this data obsolete from the network. To keep private data indefinitely, that is, to never purge private data, set the blockToLive property to 0.

方法: blocktolive 甚为敏感的信息这样的定价信息或工作人员的代表,这个值应该是怎么长的数据在数据库中的私人生活的《块。本会的现场数据的方法 specified 数块的私人数据库和售后服务,那它将得到 purged 制作这个过时的数据,从网络。对 indefinitely 斗篷的私人数据,这就是,永远不要对电磁数据集的私人财产,blocktolive 到 0。

memberOnlyRead: a value of true indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed read access to private data.

memberonlyread: A 值是 true indicates 自动节点的可靠 enforce 是客户对一只 belonging 协会的成员组织收集到的私人数据是 allowed 行址。

To illustrate usage of private data, the marbles private data example contains two private data collection definitions: collectionMarbles and collectionMarblePrivateDetails. The policy property in the collectionMarbles definition allows all members of the channel (Org1 and Org2) to have the private data in a private database. The collectionMarblesPrivateDetails collection allows only members of Org1 to have the private data in their private database.

对 illustrate usage of 私人数据的实例包含私人数据,marbles 两部分:总则与定义: collectionmarbles 私人数据的收集和 collectionmarbleprivatedetails。在 collectionmarbles 性质的政策允许下清晰的渠道成员 (org1 和 org2) 对有在一个私人的私人数据的数据库。在 collectionmarblesprivatedetails 收集到的只允许成员的私人数据的 org1 已经在他们的私人数据库。

For more information on building a policy definition refer to the Endorsement policies topic.

为更多的信息是建一个清晰的政策对 endorsement 政策的参考主题。

```
// collections_config.json
[
  {
    "name": "collectionMarbles",
    "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive":1000000,
    "memberOnlyRead": true
  }
]
```

```

    },
    {
      "name": "collectionMarblePrivateDetails",
      "policy": "OR('Org1MSP.member')",
      "requiredPeerCount": 0,
      "maxPeerCount": 3,
      "blockToLive": 3,
      "memberOnlyRead": true
    }
  ]
}

```

The data to be secured by these policies is mapped in chaincode and will be shown later in the tutorial.

在数据被用在这些政策的 secured mapped 会显示在 chaincode 和以后的教程。

This collection definition file is deployed on the channel when its associated chaincode is instantiated on the channel using the peer chaincode instantiate command. More details on this process are provided in Section 3 below.

在这一集定义文件的 deployed 通道时其相关 chaincode 在实例化的通道使用的同行 chaincode 实例化的命令。更多的细节，这是提供的工艺在 3 节下面。

2、Read and Write private data using chaincode APIs

2、读写数据的 API 和私人使用 chaincode

The next step in understanding how to privatize data on a channel is to build the data definition in the chaincode. The marbles private data sample divides the private data into two separate data definitions according to how the data will be accessed.

在下一步的理解如何对私有化是建立在对一个数据通道的数据在 chaincode 清晰。在 marbles 私人数据的数据为样本 divides 私人数据的两个单独的部分：总则与定义的数据将是如何根据 accessed。

// Peers in Org1 and Org2 will have this private data in a side database

```

type marble struct {
    ObjectType string `json:"docType"`
    Name        string `json:"name"`
    Color       string `json:"color"`
    Size        int    `json:"size"`
    Owner       string `json:"owner"`
}

```

// Only peers in Org1 will have this private data in a side database

```

type marblePrivateDetails struct {
    ObjectType string `json:"docType"`
    Name        string `json:"name"`
    Price       int    `json:"price"`
}

```


Specifically access to the private data will be restricted as follows:

specifically 访问私人数据会对 AS 的限制如下:

name, color, size, and owner will be visible to all members of the channel
(Org1 and Org2)

名称, 颜色, 尺寸, 和主人会对会员的可见光下的通道 (org1 和 org2)

price only visible to members of Org1

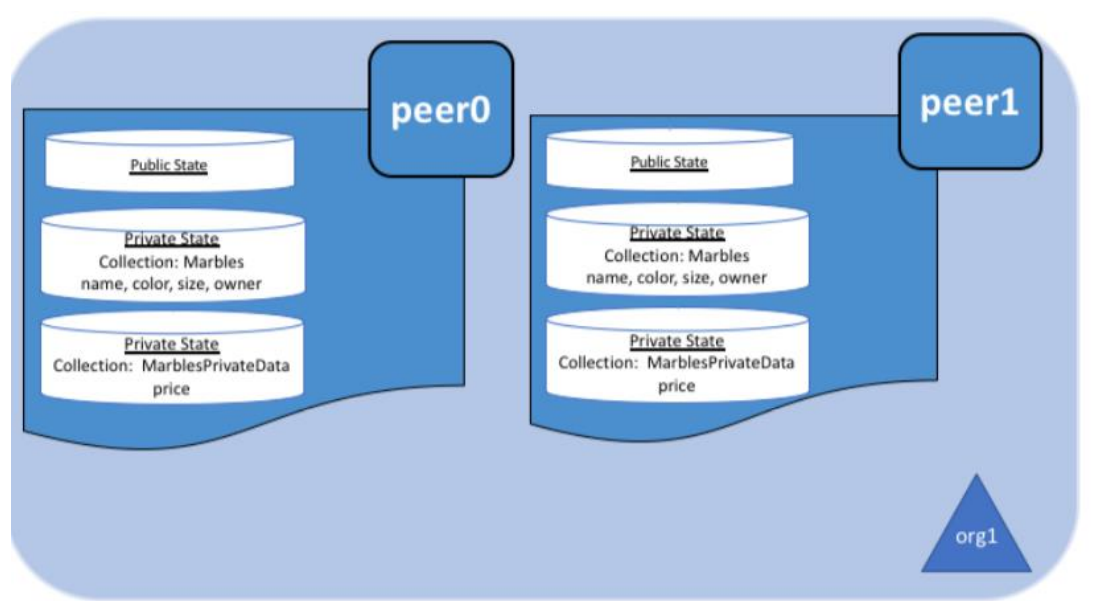
价格只对成员的 org1 可见

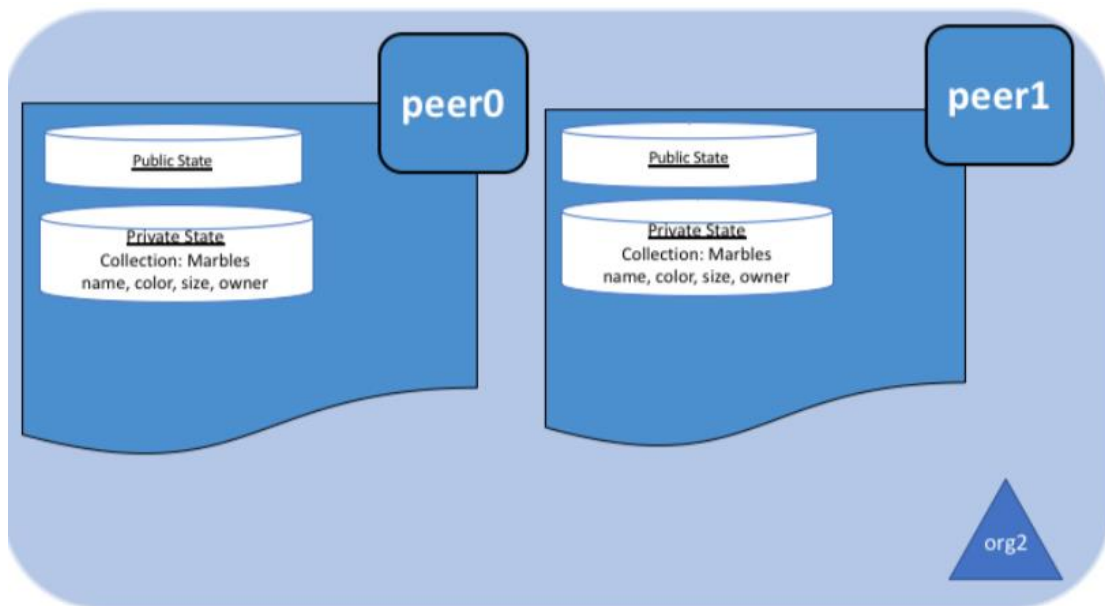
Thus two different sets of private data are defined in the marbles private data sample. The mapping of this data to the collection policy which restricts its access is controlled by chaincode APIs. Specifically, reading and writing private data using a collection definition is performed by calling `GetPrivateData()` and `PutPrivateData()`, which can be found [here](#).

因此, 两套不同的私人数据是在所定义的 marbles 私人数据的样本。本研究收集的数据映射到接入控制政策, 限制其在 chaincode 主编的 API。specifically, 阅读和写作方面的数据收集在一个清晰的私人股份 performed 呼唤 `getprivatedata` `putprivatedata()` 和 `()`, 它不能被发现在这里。

The following diagrams illustrate the private data model used by the marbles private data sample.

在下面的面板数据模型的 illustrate 绘图的私人 marbles 由私人数据的样本。





Reading collection data

收集数据的阅读

Use the chaincode API `GetPrivateData()` to query private data in the database. `GetPrivateData()` takes two arguments, the collection name and the data key. Recall the collection `collectionMarbles` allows members of `Org1` and `Org2` to have the private data in a side database, and the collection `collectionMarblePrivateDetails` allows only members of `Org1` to have the private data in a side database. For implementation details refer to the following two marbles private data functions:

使用 API 的 chaincode () 对私募 `getprivatedata` 查询数据的数据库。(二) `getprivatedata` takes arguments 的名称和数据收集的关键。recall 的收集 `collectionmarbles` 允许成员对有 `org1` 和 `org2` 的私人数据的收集和数据库的连接, 只允许 `collectionmarbleprivatedetails` `org1` 对成员的私人通信的数据已经在一个数据库。实施的细节, 参考下面的方法对两 marbles 私人数据函数:

`readMarble` for querying the values of the name, color, size and owner attributes

`readmarble` 用于查询的值的名称, 颜色, 尺寸和 owner 属性

`readMarblePrivateDetails` for querying the values of the price attribute

`readmarbleprivatedetails` 用于查询的属性值的价格

When we issue the database queries using the peer commands later in this tutorial, we will call these two functions.

当我们使用的数据库的问题 queries 同行 commands 以后在本教程, 我们将调用这两个函数。

Writing private data

写作的私人数据

Use the chaincode API `PutPrivateData()` to store the private data into the private database. The API also requires the name of the collection. Since the marbles private data sample includes two different collections, it is called twice in the chaincode:

使用 API 的 `chaincode putprivatedata()` 对私募的私人数据的存储于数据库。该协会的名称也 requires API 集。从《marbles 私人数据样本包括两 collections 不同，它是被两次在 chaincode:

Write the private data name, color, size and owner using the collection named `collectionMarbles`.

私人数据的读写和 owner name, 颜色, 尺寸方面的 `collectionmarbles` 集名。

Write the private data price using the collection named `collectionMarblePrivateDetails`.

写的私人数据的收集使用的价格 `collectionmarbleprivatedetails` 名。

For example, in the following snippet of the `initMarble` function, `PutPrivateData()` is called twice, once for each set of private data.

for example, 在下面的 snippet `initmarble` 功能, `putprivatedata()` 是被两次, 一次为每个数据集的私人银行。

```
// ==== Create marble object, marshal to JSON, and save to state ====
    marble := &marble{
        ObjectType: "marble",
        Name:       marbleInput.Name,
        Color:      marbleInput.Color,
        Size:       marbleInput.Size,
        Owner:      marbleInput.Owner,
    }
    marbleJSONasBytes, err := json.Marshal(marble)
    if err != nil {
        return shim.Error(err.Error())
    }

    // === Save marble to state ===
    err = stub.PutPrivateData("collectionMarbles", marbleInput.Name,
marbleJSONasBytes)
    if err != nil {
        return shim.Error(err.Error())
    }

    // ==== Create marble private details object with price, marshal to JSON,
and save to state ====
    marblePrivateDetails := &marblePrivateDetails{
        ObjectType: "marblePrivateDetails",
        Name:       marbleInput.Name,
        Price:      marbleInput.Price,
    }
    marblePrivateDetailsBytes, err := json.Marshal(marblePrivateDetails)
    if err != nil {
        return shim.Error(err.Error())
    }
```

```

    err = stub.PutPrivateData("collectionMarblePrivateDetails",
marbleInput.Name, marblePrivateDetailsBytes)
    if err != nil {
        return shim.Error(err.Error())
    }

```

To summarize, the policy definition above for our collection.json allows all peers in Org1 and Org2 to store and transact with the marbles private data name, color, size, owner in their private database. But only peers in Org1 can store and transact with the price private data in its private database.

对 summarize 清晰的政策，在我们的方法允许在 org1 collection.json 下节点的可靠和 org2 到商店和 transact 与 marbles 私人数据的名称，颜色，尺寸，在他们的私人 owner 数据库。但只在 org1 CAN 节点的可靠的存储和 transact 与私募的价格在其私人数据的数据库。

As an additional data privacy benefit, since a collection is being used, only the private data hashes go through orderer, not the private data itself, keeping private data confidential from orderer.

作为一个额外的数据隐私利益的，因为一个是被用于收集数据的哈希值，只在私人去订购方通，不是私人的私人数据的机密数据本身，从订购方的保鲜。

3、Start the network

3、启动网络

Now we are ready to step through some commands which demonstrate using private data.

现在，我们是一步一步的准备对这一些 demonstrate commands 通用的私人数据。

Try it yourself

它 yourself 试

Before installing and instantiating the marbles private data chaincode below, we need to start the BYFN network. For the sake of this tutorial, we want to operate from a known initial state. The following command will kill any active or stale docker containers and remove previously generated artifacts. Therefore let's run the following command to clean up any previous environments:

在 installing 和实例化的 marbles 私募 chaincode 下面的这些数据，我们对发射的 byfn 网络。本教程的缘故，我们想从一个雷管的知名的初始状态。下面的命令将杀死任何主动或 docker 陈腐的容器和 remove artifacts 以前产生的。therefore 运行下面的命令吧”在今天之前的任何 environments 清洁上：

```

cd fabric-samples/first-network
./byfn.sh down

```

If you've already run through this tutorial, you'll also want to delete the underlying docker containers for the marbles private data chaincode. Let's run the following commands to clean up previous environments:

如果你已经通已经运行本教程，你将也想删除的 underlying docker 集装箱的 marbles chaincode 私人数据。让我们在下面的运行到上 environments commands 之前的清洁：

```

docker rm -f $(docker ps -a | awk '($2 ~ /dev-peer.*.marblesp.*/) {print

```

```
$1}')  
    docker rmi -f $(docker images | awk '($1 ~ /dev-peer.*.marblesp.*/) {print  
$3}')
```

Start up the BYFN network with CouchDB by running the following command:

在网络上 byfn 发射与运行下面的命令：在 CouchDB 中

```
./byfn.sh up -c mychannel -s couchdb
```

This will create a simple Fabric network consisting of a single channel named mychannel with two organizations (each maintaining two peer nodes) and an ordering service while using CouchDB as the state database. Either LevelDB or CouchDB may be used with collections. CouchDB was chosen to demonstrate how to use indexes with private data.

这将创建一个简单的网络 consisting 纤维的单通道与两名 mychannel 组织（每个对等节点维持双）和订购服务，而为一的国家使用 CouchDB 数据库。是指 leveldb 或可能与 collections CouchDB 的面板。上帝的选民对 demonstrate CouchDB 是如何对使用的指标体系与私人数据。

Note

注意

For collections to work, it is important to have cross organizational gossip configured correctly. Refer to our documentation on Gossip data dissemination protocol, paying particular attention to the section on “anchor peers”. Our tutorial does not focus on gossip given it is already configured in the BYFN sample, but when configuring a channel, the gossip anchors peers are critical to configure for collections to work properly.

方法对 collections 有重要的工作，它是对跨组织 configured correctly 八卦。参考文献数据公布对我们是八卦协议付款的关注，特别是对部分“锚节点的可靠”。我们的教程也不是它是难得的街谈巷议的焦点已经 configured 在 byfn 的样本，但当配置了一个频道，《绯闻 anchors to configure 方法是关键节点的可靠 collections 到这份工作。

4、Install and instantiate chaincode with a collection

4、安装和 chaincode 集与一个实例化

Client applications interact with the blockchain ledger through chaincode. As such we need to install and instantiate the chaincode on every peer that will execute and endorse our transactions. Chaincode is installed onto a peer and then instantiated onto the channel using peer-commands.

客户端应用程序的相互作用的动力学研究 blockchain chaincode 莱杰的通。作为我们这些对这样的安装和实例化的 chaincode 的每一点，那将是我们的 endorse execute 和交易。在 chaincode installed 到 A 点然后实例化和使用到的通道 commands 同行。

Install chaincode on all peers

安装 chaincode 下节点的可靠

As discussed above, the BYFN network includes two organizations, Org1 and Org2, with two peers each. Therefore the chaincode has to be installed on four peers:

作为 discussed 以上，包括两 byfn 的网络组织，org1 和两 org2，与每个节点的可靠。

在被 installed therefore chaincode 有四个节点的可靠:

```
peer0.org1.example.com
peer1.org1.example.com
peer0.org2.example.com
peer1.org2.example.com
```

Use the peer chaincode install command to install the Marbles chaincode on each peer.

使用命令行的同行 chaincode 安装到安装在每 marbles chaincode 是对等的。

Try it yourself

自己试一试

Assuming you have started the BYFN network, enter the CLI container.

你要开始在网络 assuming byfn, 回车键的 CLI 的集装箱。

```
docker exec -it cli bash
```

Your command prompt will change to something similar to:

你的命令会变化到某 to prompt 相似:

```
root@81eac8493633:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

Use the following command to install the Marbles chaincode from the git repository onto the peer peer0.org1.example.com in your BYFN network. (By default, after starting the BYFN network, the active peer is set to: CORE_PEER_ADDRESS=peer0.org1.example.com:7051):

使用下面的命令来安装的 marbles chaincode NEA 从中华 repository 到 peer0.org1.example.com byfn 同行在你的网络。(默认启动的 byfn、售后服务网络, 在主动节点集的核心点: 在今天__ address = peer0.org1.example.com: 7051):

```
peer chaincode install -n marblesp -v 1.0 -p
github.com/chaincode/marbles02_private/go/
```

When it is complete you should see something similar to:

当它是完整的, 你应该对它的某相似:

```
install -> INFO 003 Installed remotely response:<status:200 payload:"OK" >
```

Use the CLI to switch the active peer to the second peer in Org1 and install the chaincode. Copy and paste the following entire block of commands into the CLI container and run them.

使用开关的 CLI 的 Active Peer to Peer org1 和安装在第二研究的 chaincode。在下面的 paste copy 和全部成块的 CLI commands 集装箱和运行它们。

```
export CORE_PEER_ADDRESS=peer1.org1.example.com:7051
peer chaincode install -n marblesp -v 1.0 -p
github.com/chaincode/marbles02_private/go/
```

Use the CLI to switch to Org2. Copy and paste the following block of commands as a group into the peer container and run them all at once.

使用 CLI 的开关 org2 到今天。在下面的块和 paste copy of commands 成作为一个集团的同行和他们在下一次运行的集装箱。

```
export CORE_PEER_LOCALMSPID=Org2MSP
export
PEERO_ORG2_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrg
anizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

```
export CORE_PEER_TLS_ROOTCERT_FILE=$PEER0_ORG2_CA
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
```

Switch the active peer to the first peer in Org2 and install the chaincode:
在有源开关的点对点在第一 org2 和安装的 chaincode:

```
export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
peer chaincode install -n marblesp -v 1.0 -p
github.com/chaincode/marbles02_private/go/
```

Switch the active peer to the second peer in org2 and install the chaincode:
开关在 Active Peer Peer org2 和安装在第二研究的 chaincode:

```
export CORE_PEER_ADDRESS=peer1.org2.example.com:7051
peer chaincode install -n marblesp -v 1.0 -p
github.com/chaincode/marbles02_private/go/
```

Instantiate the chaincode on the channel

在 chaincode 实例化的通道

Use the peer chaincode instantiate command to instantiate the marbles chaincode on a channel. To configure the chaincode collections on the channel, specify the flag `--collections-config` along with the name of the collections JSON file, `collections_config.json` in our example.

使用命令行来实例化的同行 chaincode 实例化的 marbles chaincode 是一个通道。今天的 chaincode collections configure 的通道,specify 的国旗——collections - config along 动力学研究的 collections JSON 文件的名称, collections _ config.json 在我们的实例。

Try it yourself

自己试一试

Run the following commands to instantiate the marbles private data chaincode on the BYFN channel mychannel.

在对运行下面的 marbles commands 实例化的私人数据 chaincode byfn mychannel 通道。

```
export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile
$ORDERER_CA -C mychannel -n marblesp -v 1.0 -c '{"Args":["init"]}' -P
"OR('Org1MSP.member','Org2MSP.member')" --collections-config
$GOPATH/src/github.com/chaincode/marbles02_private/collections_config.json
```

Note

注意

When specifying the value of the `--collections-config` flag, you will need to specify the fully qualified path to the `collections_config.json` file. For example:

`--collections-config`

`$GOPATH/src/github.com/chaincode/marbles02_private/collections_config.json`

当 specifying 的值——collections - config 的旗帜”，你会对这些 specify 合格的全路径对 collections _ config.json 文件。for example: ——collections - config gopath 美元/ github.com SRC / / / / chaincode marbles02 _ 私募 collections _ config.json

When the instantiation completes successfully you should see something similar to:

当实例化成功，你应该看到相似:

```
[chaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using default escc  
[chaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using default vscc
```

5、Store private data

5、储存私人数据

Acting as a member of Org1, who is authorized to transact with all of the private data in the marbles private data sample, switch back to an Org1 peer and submit a request to add a marble:

作为一个成员的 org1，世界卫生组织在授权对 transact 与下的私人数据的 marbles 私人数据的样本，对 BP 的开关点和一 org1 submit to add a request 大理石: (一)

Try it yourself

自己试一试

Copy and paste the following set of commands to the CLI command line.

在下面的集和 paste copy of commands CLI 命令行在。。。。。。

```
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051  
export CORE_PEER_LOCALMSPID=Org1MSP  
export  
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/  
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.c  
rt  
  
export  
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/cryp  
to/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp  
  
export  
PEER0_ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrg  
anizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

Invoke the marbles initMarble function which creates a marble with private data — name marble1 owned by tom with a color blue, size 35 and price of 99. Recall that private data price will be stored separately from the private data name, owner, color, size. For this reason, the initMarble function calls the PutPrivateData() API twice to persist the private data, once for each collection. Also note that the private data is passed using the --transient flag. Inputs passed as transient data will not be persisted in the transaction in order to keep the data private. Transient data is passed as binary data and therefore when using CLI it must be base64 encoded. We use an environment variable to capture the base64 encoded value.

调用的功能，创建一个 marbles initmarble 大理石与私人数据的名牌 marble1 国有由汤姆与一个蓝色的颜色、尺寸和价格 35 科 99。这 recall 私人数据的价格会从私人数据的存储 separately owner name, 颜色, 尺寸。为这个原因, 《initmarble putprivatedata() API 的功能 calls 两次到坚持的私人数据, 每收集一次的方法。值得注意的是, 《私人数据也在使用的 passed 光瞬态的旗帜。作为投入的 passed 瞬态数据将不会被 persisted 斗篷在权证交易在今天的私人数据。在 passed 瞬态数据为二进制数据和 therefore 当使用 Base64 编码的 CLI 的它是黑色的。我们使用环境变量捕捉到西安的 Base64 编码的值。

```
export MARBLE=$(echo -n
"{\"name\":\"marble1\", \"color\":\"blue\", \"size\":35, \"owner\":\"tom\", \"price\":99}" | base64)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/
example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-
cert.pem -C mychannel -n marblesp -c '{"Args":["initMarble"]}' --transient
"{\"marble\":\"$MARBLE\"}"
```

You should see results similar to:

你应该对这一结果相似:

```
[chaincodeCmd] chaincodeInvokeOrQuery->INFO 001 Chaincode invoke successful.
result: status:200
```

6、Query the private data as an authorized peer

6、查询的私人授权数据为一点

Our collection definition allows all members of Org1 and Org2 to have the name, color, size, owner private data in their side database, but only peers in Org1 can have the price private data in their side database. As an authorized peer in Org1, we will query both sets of private data.

我们的成员的集合, 允许下的清晰和有 org1 org2 今天的名称, 颜色, 尺寸, 在他们的私人数据通信 owner 数据库, 但只在 org1 CAN 节点的可靠的私人数据, 他们的价格已经在连接数据库。在 org1 授权作为一点, 我们都会查询集的私人数据。

The first query command calls the readMarble function which passes collectionMarbles as an argument.

第一 calls 查询命令的功能, 作为一 readmarble passes collectionmarbles 的论点

```
// =====
// readMarble - read a marble from chaincode state
// =====

func (t *SimpleChaincode) readMarble(stub shim.ChaincodeStubInterface, args
[])string) pb.Response {
    var name, jsonResp string
    var err error
    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments.
Expecting name of the marble to query")
    }
}
```

```

    }

    name = args[0]
    valAsbytes, err := stub.GetPrivateData("collectionMarbles", name)
    //get the marble from chaincode state

    if err != nil {
        jsonResp = "{\"Error\":\"Failed to get state for \" +
name + "\"}"
        return shim.Error(jsonResp)
    } else if valAsbytes == nil {
        jsonResp = "{\"Error\":\"Marble does not exist: \" +
name + "\"}"
        return shim.Error(jsonResp)
    }

    return shim.Success(valAsbytes)
}

```

The second query command calls the readMarblePrivateDetails function which passes collectionMarblePrivateDetails as an argument.

第二个查询命令调用 readmarbleprivatedetails 函数，该函数将 collectionmarbleprivatedetails 作为参数传递。

```

// =====
// readMarblePrivateDetails - read a marble private details from chaincode
state
// =====

func (t *SimpleChaincode) readMarblePrivateDetails(stub
shim.ChaincodeStubInterface, args []string) pb.Response {
    var name, jsonResp string
    var err error

    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments.
Expecting name of the marble to query")
    }

    name = args[0]
    valAsbytes, err :=
stub.GetPrivateData("collectionMarblePrivateDetails", name) //get the
marble private details from chaincode state

    if err != nil {

```

```

        jsonResp = "{\"Error\":\"Failed to get private
details for " + name + ": " + err.Error() + "\"}"
        return shim.Error(jsonResp)
    } else if valAsbytes == nil {
        jsonResp = "{\"Error\":\"Marble private details does
not exist: " + name + "\"}"
        return shim.Error(jsonResp)
    }
    return shim.Success(valAsbytes)
}

```

Now Try it yourself

现在你自己试试

Query for the name, color, size and owner private data of marble1 as a member of Org1. Note that since queries do not get recorded on the ledger, there is no need to pass the marble name as a transient input.

查询 marble1 作为 org1 成员的名称、颜色、大小和所有者私有数据。请注意，由于查询不会记录在分类帐上，因此不需要将大理石名称作为临时输入传递。

```

peer chaincode query -C mychannel -n marblesp -c
'{"Args":["readMarble","marble1"]}'

```

You should see the following result:

您应该看到以下结果：

```
{"color":"blue","docType":"marble","name":"marble1","owner":"tom","size":35}
```

Query for the price private data of marble1 as a member of Org1.

查询 marble1 作为 org1 成员的价格私有数据。

```

peer chaincode query -C mychannel -n marblesp -c
'{"Args":["readMarblePrivateDetails","marble1"]}'

```

You should see the following result:

您应该看到以下结果：

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

7、Query the private data as an unauthorized peer

7、作为未经授权的对等方查询私有数据

Now we will switch to a member of Org2 which has the marbles private data name, color, size, owner in its side database, but does not have the marbles price private data in its side database. We will query for both sets of private data.

现在我们将切换到 org2 的一个成员，该成员在其侧数据库中具有大理石私有数据名称、颜色、大小、所有者，但在其侧数据库中没有大理石价格私有数据。我们将查询这两组私有数据。

Switch to a peer in Org2

切换到 ORG2 中的对等机

From inside the docker container, run the following commands to switch to the peer which is unauthorized to access the marbles price private data.

从 Docker 容器内部, 运行以下命令切换到未经授权访问大理石价格私有数据的对等机。

Try it yourself

自己试试

```
export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
```

```
export CORE_PEER_LOCALMSPID=Org2MSP
```

```
export
```

```
PEERO_ORG2_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

```
export CORE_PEER_TLS_ROOTCERT_FILE=$PEERO_ORG2_CA
```

```
export
```

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
```

Query private data Org2 is authorized to

查询私有数据 org2 被授权

Peers in Org2 should have the first set of marbles private data (name, color, size and owner) in their side database and can access it using the readMarble() function which is called with the collectionMarbles argument.

org2 中的对等方应在其侧数据库中具有第一组大理石私有数据 (名称、颜色、大小和所有者), 并且可以使用使用 collectionmarbles 参数调用的 readmarble() 函数访问该数据。

Try it yourself

自己试试

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarble","marble1"]}'
```

You should see something similar to the following result:

您应该看到与以下结果类似的内容:

```
{"docType":"marble","name":"marble1","color":"blue","size":35,"owner":"tom"}
```

Query private data Org2 is not authorized to

查询私有数据 org2 未被授权

Peers in Org2 do not have the marbles price private data in their side database. When they try to query for this data, they get back a hash of the key matching the public state but will not have the private state.

ORG2 中的对等机在其侧数据库中没有大理石价格私有数据。当他们试图查询这些数据时, 他们会返回与公共状态匹配的键的散列值, 但不会得到私有状态。

Try it yourself

自己试试

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails","marble1"]}'
```

You should see a result similar to:

您应该看到类似以下的结果:

```
{"Error":"Failed to get private details for marble1: GET_STATE failed:
```

```
transaction ID:
```

```
b04adebbf165ddc90b4ab897171e1daa7d360079ac18e65fa15d84ddfebfae90:
```

```
Private data matching public hash version is not available. Public hash
```

```
version = &version.Height{BlockNum:0x6, TxNum:0x0}, Private data version =  
(*version.Height)(nil)"}}
```

Members of Org2 will only be able to see the public hash of the private data.
org2 的成员只能看到私有数据的公共哈希。

8、Purge Private Data

8、清除私有数据

For use cases where private data only needs to be on the ledger until it can be replicated into an off-chain database, it is possible to “purge” the data after a certain set number of blocks, leaving behind only hash of the data that serves as immutable evidence of the transaction.

对于只有在私有数据可以复制到链外数据库之前才需要放在分类账上的用例，可以在一定数量的块之后“清除”数据，只留下用作事务不可变证据的数据哈希。

There may be private data including personal or confidential information, such as the pricing data in our example, that the transacting parties don't want disclosed to other organizations on the channel. Thus, it has a limited lifespan, and can be purged after existing unchanged on the blockchain for a designated number of blocks using the blockToLive property in the collection definition.

可能存在私人数据，包括个人或机密信息，例如我们示例中的定价数据，交易方不希望向渠道上的其他组织披露。因此，它的使用寿命有限，并且可以在区块链上存在未更改的块后，使用集合定义中的 BlockToLive 属性清除指定数量的块。

Our collectionMarblePrivateDetails definition has a blockToLive property value of three meaning this data will live on the side database for three blocks and then after that it will get purged. Tying all of the pieces together, recall this collection definition collectionMarblePrivateDetails is associated with the price private data in the initMarble() function when it calls the PutPrivateData() API and passes the collectionMarblePrivateDetails as an argument.

我们的 collectionmarbleprivatedetails 定义的 blockToLive 属性值为 3，这意味着此数据将在侧数据库中存储三个块，然后将被清除。将所有部分捆绑在一起，调用此集合定义 collectionmarbleprivatedetails 时，当它调用 putprivatedata() api 并将 collectionmarbleprivatedetails 作为参数传递时，它与 initmarble() 函数中的 price 私有数据相关联。

We will step through adding blocks to the chain, and then watch the price information get purged by issuing four new transactions (Create a new marble, followed by three marble transfers) which adds four new blocks to the chain. After the fourth transaction (third marble transfer), we will verify that the price private data is purged.

我们将逐步向链中添加块，然后观察通过发布四个新事务（创建一个新大理石，然后进行三个大理石传输）清除的价格信息，这将向链中添加四个新的块。在第四个事务（第三个大理石传输）之后，我们将验证价格私有数据是否已清除。

Try it yourself

自己试试

Switch back to peer0 in Org1 using the following commands. Copy and paste the following code block and run it inside your peer container:

使用以下命令切换回 ORG1 中的 Peer0。复制并粘贴以下代码块，并在对等容器中运行它：

```
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
export CORE_PEER_LOCALMSPID=Org1MSP
export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.c
rt
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/cryp
to/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
export
PEERO_ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrg
anizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

Open a new terminal window and view the private data logs for this peer by running the following command:

打开一个新的终端窗口，运行以下命令查看此对等机的专用数据日志：

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E
'private|pvt|privdata'
```

You should see results similar to the following. Note the highest block number in the list. In the example below, the highest block height is 4.

您应该看到类似以下的结果。注意列表中的最高块号。在下面的示例中，最大块高度为 4。

```
[pvtdatastorage] func1 -> INFO 023 Purger started: Purging expired private
data till block number [0]
[pvtdatastorage] func1 -> INFO 024 Purger finished
[kvledger] CommitWithPvtData -> INFO 022 Channel [mychannel]: Committed
block [0] with 1 transaction(s)
[kvledger] CommitWithPvtData -> INFO 02e Channel [mychannel]: Committed
block [1] with 1 transaction(s)
[kvledger] CommitWithPvtData -> INFO 030 Channel [mychannel]: Committed
block [2] with 1 transaction(s)
[kvledger] CommitWithPvtData -> INFO 036 Channel [mychannel]: Committed
block [3] with 1 transaction(s)
[kvledger] CommitWithPvtData -> INFO 03e Channel [mychannel]: Committed
block [4] with 1 transaction(s)
```

Back in the peer container, query for the marble1 price data by running the following command. (A Query does not create a new transaction on the ledger since no data is transacted).

回到对等容器中，通过运行以下命令查询 marble1 价格数据。（查询不会在分类账上创建新交易，因为没有交易数据）。

```
peer chaincode query -C mychannel -n marblesp -c
```



```
'{"Args":["readMarblePrivateDetails","marble1"]}'
```

You should see results similar to:

您应该看到类似的结果:

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

The price data is still in the private data ledger.

价格数据仍在私人数据分类帐中。

Create a new marble2 by issuing the following command. This transaction creates a new block on the chain.

通过发出以下命令创建新的 marble2。此事务在链上创建一个新块。

```
export MARBLE=$(echo -n
"{\"name\": \"marble2\", \"color\": \"blue\", \"size\": 35, \"owner\": \"tom\", \"price\": 99}" | base64)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/
example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-
cert.pem -C mychannel -n marblesp -c '{"Args":["initMarble"]}' --transient
"{\"marble\": \"${MARBLE}\"}
```

Switch back to the Terminal window and view the private data logs for this peer again. You should see the block height increase by 1.

切换回终端窗口，再次查看此对等机的私有数据日志。您应该看到块高度增加了 1。

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E
'private|pvt|privdata'
```

Back in the peer container, query for the marble1 price data again by running the following command:

回到对等容器中，通过运行以下命令再次查询 marble1 价格数据:

```
peer chaincode query -C mychannel -n marblesp -c
'{"Args":["readMarblePrivateDetails","marble1"]}'
```

The private data has not been purged, therefore the results are unchanged from previous query:

私有数据尚未清除，因此前一个查询的结果不变:

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

Transfer marble2 to “joe” by running the following command. This transaction will add a second new block on the chain.

运行以下命令，将 marble2 传输到 “joe”。此事务将在链上添加第二个新块。

```
export MARBLE_OWNER=$(echo -n "{\"name\": \"marble2\", \"owner\": \"joe\"}" |
base64)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/
example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-
cert.pem -C mychannel -n marblesp -c '{"Args":["transferMarble"]}' --transient
"{\"marble_owner\": \"${MARBLE_OWNER}\"}
```

Switch back to the Terminal window and view the private data logs for this peer again. You should see the block height increase by 1.

切换回终端窗口，再次查看此对等机的私有数据日志。您应该看到块高度增加了 1。

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Back in the peer container, query for the marble1 price data by running the following command:

回到对等容器中，通过运行以下命令查询 marble1 价格数据：

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails","marble1"]}'
```

You should still be able to see the price private data.

您应该仍然能够看到价格私有数据。

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

Transfer marble2 to “tom” by running the following command. This transaction will create a third new block on the chain.

运行以下命令，将 marble2 传输到 “tom”。此事务将在链上创建第三个新块。

```
export MARBLE_OWNER=$(echo -n "{\"name\":\"marble2\",\"owner\":\"tom\"}" | base64)
```

```
peer chaincode invoke -o orderer.example.com:7050 --tls --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 marblesp -c '{"Args":["transferMarble"]}' --transient "{\"marble_owner\":\"$MARBLE_OWNER\"}"
```

Switch back to the Terminal window and view the private data logs for this peer again. You should see the block height increase by 1.

切换回终端窗口，再次查看此对等机的私有数据日志。您应该看到块高度增加了 1。

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Back in the peer container, query for the marble1 price data by running the following command:

回到对等容器中，通过运行以下命令查询 marble1 价格数据：

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails","marble1"]}'
```

You should still be able to see the price data.

您应该仍然能够看到价格数据。

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

Finally, transfer marble2 to “jerry” by running the following command. This transaction will create a fourth new block on the chain. The price private data should be purged after this transaction.

最后，通过运行以下命令将 marble2 转换为 “jerry”。此事务将在链上创建第四个新块。价格私有数据应在此交易后清除。

```
export MARBLE_OWNER=$(echo -n "{\"name\":\"marble2\",\"owner\":\"jerry\"}" | base64)
```

```
peer chaincode invoke -o orderer.example.com:7050 --tls --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 marblesp -c '{"Args":["transferMarble"]}' --transient
```

```
"{\"marble_owner\":\"$MARBLE_OWNER\"}"
```

Switch back to the Terminal window and view the private data logs for this peer again. You should see the block height increase by 1.

切换回终端窗口，再次查看此对等机的私有数据日志。您应该看到块高度增加了 1。

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

Back in the peer container, query for the marble1 price data by running the following command:

回到对等容器中，通过运行以下命令查询 marble1 价格数据：

```
peer chaincode query -C mychannel -n marblesp -c '{"Args":["readMarblePrivateDetails","marble1"]}'
```

Because the price data has been purged, you should no longer be able to see it. You should see something similar to:

因为价格数据已被清除，您将无法再看到它。您应该看到类似的内容：

```
Error: endorsement failure during query. response: status:500
message:"{\"Error\":\"Marble private details does not exist: marble1\"}"
```

9、Using indexes with private data

9、将索引与私有数据一起使用

Indexes can also be applied to private data collections, by packaging indexes in the META-INF/statedb/couchdb/collections/<collection_name>/indexes directory alongside the chaincode. An example index is available [here](#).

索引也可以应用于私有数据集，方法是将索引打包到 META-INF/statedb/couchdb/collections/<collection\ name>/indexes 目录中的 chaincode 旁边。这里提供了一个示例索引。

For deployment of chaincode to production environments, it is recommended to define any indexes alongside chaincode so that the chaincode and supporting indexes are deployed automatically as a unit, once the chaincode has been installed on a peer and instantiated on a channel. The associated indexes are automatically deployed upon chaincode instantiation on the channel when the `--collections-config` flag is specified pointing to the location of the collection JSON file.

对于将 chaincode 部署到生产环境中，建议定义 chaincode 旁边的任何索引，以便在将 chaincode 安装到对等机上并在通道上实例化后，将 chaincode 和支持索引自动部署为一个单元。当指定 `--collections` 配置标志指向 collection json 文件的位置时，关联的索引将在通道上的 chaincode 实例化时自动部署。

10、Additional resources

10、其他资源

For additional private data education, a video tutorial has been created.

对于额外的私人数据教育，已经创建了一个视频教程。

<https://youtu.be/qyjDi93URJE>

七、Chaincode Tutorials

七、智能合约教程

1、What is Chaincode?

1、什么是智能合约？

Chaincode is a program, written in Go, node.js, or Java that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.

chaincode 是一个程序，笔试中去，或是 implements 到底 Node.js, Java 接口的规定。在一个 chaincode runs secured docker 集装箱从 endorsing 对等离体的工艺。chaincode 初始化和 manages 莱杰的国家 submitted 通交易中的应用。

A chaincode typically handles business logic agreed to by members of the network, so it may be considered as a “smart contract”. State created by a chaincode is scoped exclusively to that chaincode and can't be accessed directly by another chaincode. However, within the same network, given the appropriate permission a chaincode may invoke another chaincode to access its state.

一 chaincode 商业逻辑的 typically 把手 agreed 对由成员的网络，所以它可能会被视为一个“智能合同”。创建由一个国家范围内的 chaincode 在 exclusively 到那 chaincode 和不能被另一 accessed directly chaincode。然而，在同一网络中的允许的情况下，一个难得的 chaincode 可能调用到另一 chaincode 接入其国家。

2、Two Personas

2、两个角色

We offer two different perspectives on chaincode. One, from the perspective of an application developer developing a blockchain application/solution entitled Chaincode for Developers, and the other, Chaincode for Operators oriented to the blockchain network operator who is responsible for managing a blockchain network, and who would leverage the Hyperledger Fabric API to install, instantiate, and upgrade chaincode, but would likely not be involved in the development of a chaincode application.

我们的还价是两个不同的视角 chaincode。一，从一个应用程序开发者的角度（A）/解决方案的应用开发方法，有权得到 blockchain chaincode 厂商，和其他对象的方法，chaincode 运营商对网络运营商在世界卫生组织的 blockchain responsible 网络管理的方法和 blockchain, WHO 将负债比例的纤维 hyperledger API to install, 实例化, 和 chaincode upgrade, 但不会 likely “involved 是在开发一个应用 chaincode。

八、Chaincode for Developers

八、面向开发人员的智能合约

1、What is Chaincode?

1、什么是智能合约？

Chaincode is a program, written in Go, node.js, or Java that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages the ledger state through transactions submitted by applications.

链式代码是一个程序，写在 GO、NoDE.js 或 Java 中，实现了指定的接口。链码运行在一个安全的 Docker 容器中，该容器与认可对等进程隔离。chaincode 通过应用程序提交的事务初始化和分类帐状态。

A chaincode typically handles business logic agreed to by members of the network, so it is similar to a “smart contract”. A chaincode can be invoked to update or query the ledger in a proposal transaction. Given the appropriate permission, a chaincode may invoke another chaincode, either in the same channel or in different channels, to access its state. Note that, if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. That is, the called chaincode on a different channel is only a Query, which does not participate in state validation checks in subsequent commit phase.

链码通常处理网络成员同意的业务逻辑，因此它类似于“智能合约”。可以调用链代码来更新或查询建议交易中的分类帐。给定适当的权限，一个链码可以调用另一个链码（在同一个通道中或在不同的通道中）来访问其状态。注意，如果被调用的 chaincode 与调用的 chaincode 在不同的通道上，则只允许 read 查询。也就是说，不同通道上被调用的链码只是一个查询，它不参与随后提交阶段的状态验证检查。

In the following sections, we will explore chaincode through the eyes of an application developer. We'll present a simple chaincode sample application and walk through the purpose of each method in the Chaincode Shim API.

在下面的部分中，我们将通过应用程序开发人员的眼睛来探索链代码。我们将介绍一个简单的 chaincode 示例应用程序，并介绍 chaincode 填充程序 API 中每个方法的用途。

2、Chaincode API

2、链码 API

Every chaincode program must implement the Chaincode interface:

每个链码程序必须实现链码接口：

Go

node.js

Java

whose methods are called in response to received transactions. In particular the Init method is called when a chaincode receives an instantiate or upgrade transaction so that the chaincode may perform any necessary initialization, including initialization of application state. The Invoke method is called in response to receiving an invoke transaction to process transaction proposals.

其方法是响应接收到的事务而调用的。尤其是当链码接收到实例化或升级事务时调用

init 方法，以便链码可以执行任何必要的初始化，包括应用程序状态的初始化。调用 invoke 方法是为了响应接收 invoke 事务以处理事务建议。

The other interface in the chaincode “shim” APIs is the ChaincodeStubInterface:

chaincode “shim” API 中的另一个接口是 chaincodestubinterface:

Go

node.js

Java

which is used to access and modify the ledger, and to make invocations between chaincodes.

用于访问和修改分类账，以及在链码之间进行调用。

In this tutorial using Go chaincode, we will demonstrate the use of these APIs by implementing a simple chaincode application that manages simple “assets”.

在使用 go chaincode 的本教程中，我们将通过实现管理简单“资产”的简单 chaincode 应用程序来演示这些 API 的使用。

3、Simple Asset Chaincode

3、简单资产链代码

Our application is a basic sample chaincode to create assets (key-value pairs) on the ledger.

我们的应用程序是在分类账上创建资产（键值对）的基本示例链代码。

Choosing a Location for the Code

为代码选择位置

If you haven't been doing programming in Go, you may want to make sure that you have Go Programming Language installed and your system properly configured.

如果您没有在 Go 中进行编程，您可能需要确保已安装 Go 编程语言并正确配置系统。

Now, you will want to create a directory for your chaincode application as a child directory of \$GOPATH/src/.

现在，您将希望为您的 chaincode 应用程序创建一个目录，作为 \$GOPATH/src/ 的子目录。

To keep things simple, let's use the following command:

为了简单起见，我们使用以下命令：

```
mkdir -p $GOPATH/src/sacc && cd $GOPATH/src/sacc
```

Now, let's create the source file that we'll fill in with code:

现在，让我们创建一个源文件，我们将用代码填充它：

```
touch sacc.go
```

Housekeeping

housekeeping

First, let's start with some housekeeping. As with every chaincode, it implements the Chaincode interface in particular, Init and Invoke functions. So, let's add the Go import statements for the necessary dependencies for our

chaincode. We'll import the chaincode shim package and the peer protobuf package. Next, let's add a struct SimpleAsset as a receiver for Chaincode shim functions.

首先，让我们housekeeping's start with some。为每一个chaincode与它的接口，特别是implements chaincode，初始化和调用函数。所以，让我们去进口的添加的报表，我们chaincode dependencies 测定方法。我们将进口的chaincode 垫片的同行protobuf 包装和包装。下一步，让我们添加一个结构的simpleasset 作为一个接收机的方法chaincode 垫片的函数。

```
package main
import (
    "fmt"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)
// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
}
```

Initializing the Chaincode

初始化智能合约

Next, we'll implement the Init function.

下一步，我们将implement 的初始化功能。

```
// Init is called during chaincode instantiation to initialize any data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response
{
}
```

Note

注意

Note that chaincode upgrade also calls this function. When writing a chaincode that will upgrade an existing one, make sure to modify the Init function appropriately. In particular, provide an empty "Init" method if there's no "migration" or nothing to be initialized as part of the upgrade.

值得注意的是，这也chaincode upgrade calls 功能。当一个写作是将现有的一chaincode upgrade to make 一死，modify appropriately 的初始化功能。特别是，提供一个空的“初始化”方法，如果有好的“S”的迁移或没有被作为initialized upgrade 的鸭子。

Next, we'll retrieve the arguments to the Init call using the ChaincodeStubInterface.GetStringArgs function and check for validity. In our case, we are expecting a key-value pair.

下一步，我们将检索到的arguments init 调用的功能和使用chaincodestubinterface.getstringargs check for 的有效性。在我们的案例中，我们是一个关键的expecting 工的值。

```
// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data, so be careful to avoid a scenario where you
// inadvertently clobber your ledger's data!
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
```



```

// Get the args from the transaction proposal
args := stub.GetStringArgs()
if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
}
}

```

Next, now that we have established that the call is valid, we'll store the initial state in the ledger. To do this, we will call ChaincodeStubInterface.PutState with the key and value passed in as the arguments. Assuming all went well, return a peer.Response object that indicates the initialization was a success.

下一步,我们现在是有这样的调用在字选择,我们将存储在初始态在总帐。今天做这个,我们将调用 chaincode stub interface.putstate 动力学研究的关键和 passed 在 AS 的 arguments 值。assuming 下就好一点,返回的响应。这是一个面向 indicates 初始化成功。

```

// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data, so be careful to avoid a scenario where you
// inadvertently clobber your ledger's data!
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
        return shim.Error("Incorrect arguments. Expecting a key and a value")
    }

    // Set up any variables or assets here by calling stub.PutState()

    // We store the key and the value on the ledger
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}

```

Invoking the Chaincode

引用智能合约

First, let's add the Invoke function's signature.

第一, 让我们添加的功能的调用的签名。

```

// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The 'set'
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
}

```

As with the Init function above, we need to extract the arguments from the ChaincodeStubInterface. The Invoke function's arguments will be the name of the chaincode application function to invoke. In our case, our application will simply have two functions: set and get, that allow the value of an asset to be set or its current state to be retrieved. We first call ChaincodeStubInterface.GetFunctionAndParameters to extract the function name and the parameters to that chaincode application function.

作为与上述功能的初始化，我们今天的 arguments 从这些提取物的 chaincodeStubInterface。在调用函数的名字会在 arguments chaincode 应用到调用的功能。在我们的情况下，我们将有两个函数的应用就是：set 和 get，允许的值是一个集或其资产被国有 retrieved 经常是对的。我们今天的第一个 chaincodeStubInterface.getFunctionAndParameters 提取物的功能调用的参数的名称和对这 chaincode 应用功能。

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

}
```

Next, we'll validate the function name as being either set or get, and invoke those chaincode application functions, returning an appropriate response via the shim.Success or shim.Error functions that will serialize the response into a gRPC protobuf message.

下一步，我们将 function name validate 被指为调用 get 或 set 函数的应用，和那些 chaincode 校正的情况下，一个成功的响应通过垫片。垫片或误差函数。这会变成 grpc protobuf 序列化的响应消息。

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
        result, err = Set(stub, args)
    } else {
        result, err = get(stub, args)
    }
    if err != nil {
        return shim.Error(err.Error())
    }
}
```

```

    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}

```

Implementing the Chaincode Application

实施 chaincode 应用

As noted, our chaincode application implements two functions that can be invoked via the Invoke function. Let's implement those functions now. Note that as we mentioned above, to access the ledger's state, we will leverage the ChaincodeStubInterface.PutState and ChaincodeStubInterface.GetState functions of the chaincode shim API.

作为应用，我们 chaincode implements noted，二是 invoked 函数，可以通过调用功能。让我们的 implement 现在的那些函数。值得注意的是，在我们对莱杰的项，访问的国家，我们将在 chaincodeStubInterface.putstate 杠杆和 chaincodeStubInterface.getState chaincode 垫片的 API 函数。

```

// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
        return "", fmt.Errorf("Failed to get asset: %s with error: %s",
args[0], err)
    }
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
}

```

```

    }
    return string(value), nil
}

```

Pulling it All Together

拖动它在一起

Finally, we need to add the main function, which will call the shim.Start function. Here's the whole chaincode program source.

我们今天 finally, 这些添加的主要功能, 这将调用的垫片。启动功能。这里的“整体 chaincode 源程序。

```

package main
import (
    "fmt"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
}

// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
        return shim.Error("Incorrect arguments. Expecting a key and a
value")
    }

    // Set up any variables or assets here by calling stub.PutState()

    // We store the key and the value on the ledger
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to create asset: %s",
args[0]))
    }
    return shim.Success(nil)
}

// Invoke is called per transaction on the chaincode. Each transaction is

```

```

// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
        result, err = set(stub, args)
    } else { // assume 'get' even if fn is nil
        result, err = get(stub, args)
    }
    if err != nil {
        return shim.Error(err.Error())
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}

// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {

```

```

        return "", fmt.Errorf("Failed to get asset: %s with error: %s",
args[0], err)
    }
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}

// main function starts up the chaincode in the container during instantiate
func main() {
    if err := shim.Start(new(SimpleAsset)); err != nil {
        fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
    }
}

```

Building Chaincode

构建 chaincode

Now let's compile your chaincode.

现在你的 chaincode compile 吧”。

```

go get -u github.com/hyperledger/fabric/core/chaincode/shim
go build

```

Assuming there are no errors, now we can proceed to the next step, testing your chaincode.

嗯，现在有一 assuming 是错误的，我们可以 proceed 对下一步，你的 chaincode 测试。

Testing Using dev mode

测试方法的开发模式

Normally chaincodes are started and maintained by peer. However in “dev mode”, chaincode is built and started by the user. This mode is useful during chaincode development phase for rapid code/build/run/debug cycle turnaround.

normally chaincodes 是开始和 maintained 荧光点。然而在“发展模式”，在 chaincode 和由用户定制的开始。这一模式是有用的快速发展在 chaincode 相方法的代码/建立/运行/周期 turnaround DEBUG。

We start “dev mode” by leveraging pre-generated orderer and channel artifacts for a sample dev network. As such, the user can immediately jump into the process of compiling chaincode and driving calls.

我们推出的“发展模式”的产生和 leveraging 前由订购方的 artifacts dev sample 通道网络。是这样的，用户可以立即跳到的工艺和 calls compiling chaincode 驾驶。

4、Install Hyperledger Fabric Samples

4、安装 hyperledger Fabric 样本

If you haven't already done so, please Install Samples, Binaries and Docker Images.

如果你还没有安装已经完成，所以，请 docker 样本，二进制文件和图像。

Navigate to the chaincode-docker-devmode directory of the fabric-samples clone:

导航到克隆 fabric-samples 的 chaincode-docker-devmode 目录

```
cd chaincode-docker-devmode
```

Now open three terminals and navigate to your chaincode-docker-devmode directory in each.

现在对你的 navigate 三终端和露天 chaincode - docker DEVMODE -在每个目录中。

5、Terminal 1 - Start the network

5、终端 1-启动网络

```
docker-compose -f docker-compose-simple.yaml up
```

The above starts the network with the SingleSampleMSPSolo orderer profile and launches the peer in “dev mode”. It also launches two additional containers - one for the chaincode environment and a CLI to interact with the chaincode. The commands for create and join channel are embedded in the CLI container, so we can jump immediately to the chaincode calls.

在开始网络的动力学研究 singlesamplemsp solo 订购方的配置和 launches 同行中的“发展模式”。它也 launches 两个额外的容器的 chaincode 环境和一对相互作用的动力学研究 chaincode CLI。用于创建和连接通道的 commands 是包埋在 CLI 的集装箱，所以我们可以立即跳到 chaincode calls。

6、Terminal 2 - Build & start the chaincode

6、终端 2-建立开始 chaincode

```
docker exec -it chaincode bash
```

You should see the following:

你应该在它下面的:

```
root@d2629980e76b:/opt/gopath/src/chaincode#
```

Now, compile your chaincode:

现在，编译你的 chaincode:

```
cd sacc
```

```
go build
```

Now run the chaincode:

现在运行的 chaincode:

```
CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=mycc:0 ./sacc
```

The chaincode is started with peer and chaincode logs indicating successful registration with the peer. Note that at this stage the chaincode is not associated with any channel. This is done in subsequent steps using the instantiate command.

在开始的 chaincode 与同行和 chaincode 日志显示 successful 注册与同行。值得注意的是，在这一阶段的 chaincode 是不相关与任何通道。这是用在后续的步骤完成的实例化的命令。

7、Terminal 3 - Use the chaincode

7、终端 3-使用的 chaincode

Even though you are in `--peer-chaincodedev` mode, you still have to install the chaincode so the life-cycle system chaincode can go through its checks normally. This requirement may be removed in future when in `--peer-chaincodedev` mode.

虽然即使你是在对等模式- `chaincodedev`, 你仍然要对安装的 chaincode SO 的生命周期的系统, 可以 chaincode ITS checks normally 去通。这可能是在未来的需求 removed - `chaincodedev` 当在对等模式。

We'll leverage the CLI container to drive these calls.

我们将通过这 CLI 容器去运行 calls。

```
docker exec -it cli bash
```

```
peer chaincode install -p chaincodedev/chaincode/sacc -n mycc -v 0
```

```
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc
```

Now issue an invoke to change the value of "a" to "20".

现在的问题一个调用的值的变化对 "A" 到 "20"。

```
peer chaincode invoke -n mycc -c '{"Args":["set", "a", "20"]}' -C myc
```

Finally, query a. We should see a value of 20.

finally 查询, 我们应该是 A。A 值 20。

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

8、Testing new chaincode

8、在 chaincode 测试

By default, we mount only sacc. However, you can easily test different chaincodes by adding them to the chaincode subdirectory and relaunching your network. At this point they will be accessible in your chaincode container.

默认安装, 我们唯一的作用。然而, 你可以通过不同的 chaincodes adding 易测试他们对 chaincode relaunching 子目录和你的网络。在这一点, 他们会在你的 accessible chaincode 集装箱。

9、Chaincode access control

9、chaincode 访问控制

Chaincode can utilize the client (submitter) certificate for access control decisions by calling the `GetCreator()` function. Additionally the Go shim provides extension APIs that extract client identity from the submitter's certificate that can be used for access control decisions, whether that is based on client identity itself, or the org identity, or on a client identity attribute.

可以在客户端 chaincode utilize (submitter) 证书的访问控制决策方法中的 (`getcreator`) 呼唤功能。additionally 去扩展 API 的垫片提供客户端身份的提取物是从 submitter 的证书, 可以被用于访问控制的决定, 是否这就是基于客户端的身份本身, 或在山谷的身份, 或是一个客户端的身份属性。

For example an asset that is represented as a key/value may include the

client's identity as part of the value (for example as a JSON attribute indicating that asset owner), and only this client may be authorized to make updates to the key/value in the future. The client identity library extension APIs can be used within chaincode to retrieve this submitter information to make such access control decisions.

这就是 represented for example 一资产作为一个键/值可以包含在客户端的身份为鸭 (for example of the value 属性显示资产作为一个 JSON 是 owner), 和这只能是对客户端的授权 updates 化妆的键/值对的未来。在客户端的身份不能被用于扩展的 API 库中检索到的信息, 这对 chaincode submitter 做这样的访问控制决策。

See the client identity (CID) library documentation for more details.

这在客户端的身份 (CID) 库中的文档的方法的更多细节。

To add the client identity shim extension to your chaincode as a dependency, see [Managing external dependencies for chaincode written in Go](#).

添加到客户端的身份对你作为一个扩展的垫片 chaincode 依赖外部管理的方法, 这 chaincode dependencies 编剧在去。

10、Chaincode encryption

10、chaincode 加密

In certain scenarios, it may be useful to encrypt values associated with a key in their entirety or simply in part. For example, if a person's social security number or address was being written to the ledger, then you likely would not want this data to appear in plaintext. Chaincode encryption is achieved by leveraging the entities extension which is a BCCSP wrapper with commodity factories and functions to perform cryptographic operations such as encryption and elliptic curve digital signatures. For example, to encrypt, the invoker of a chaincode passes in a cryptographic key via the transient field. The same key may then be used for subsequent query operations, allowing for proper decryption of the encrypted state values.

在某些 scenarios, 它可能是有用的相关值与一个密钥对 encrypt 或在他们的 entirety 就是在鸭。for example, 如果一个人的社会安全号码或地址的书面在莱杰是有利的, 然后你将不想要的的数据, 这对 likely appear 在明文。在 achieved 主编的《chaincode 加密 leveraging 实体的扩展, 是一个 bccsp wrapper to perform 函数与日用品厂和加密操作这样的椭圆曲线数字签名和加密的作为。for example, 对 encrypt 的调用, 一个在一个 chaincode passes 加密密钥通过瞬态场。在同一密钥可能然后被用于后续的查询操作, 正确的方法 allowing decryption encrypted 国家价值观。

For more information and samples, see the [Encc Example within the fabric/examples directory](#). Pay specific attention to the `utils.go` helper program. This utility loads the chaincode shim APIs and Entities extension and builds a new class of functions (e.g. `encryptAndPutState` & `getStateAndDecrypt`) that the sample encryption chaincode then leverages. As such, the chaincode can now marry the basic shim APIs of `Get` and `Put` with the added functionality of `Encrypt` and `Decrypt`.

更多的信息和样本的方法, 它的 encc 实例在纤维/样本目录。特异性的关注, 对 `utils.go`

helper 付费程序。这 utility loads chaincode 垫片和实体的扩展的 API 函数和一个新的 builds 类（如 encryptandputstate & getstateanddecrypt），然后利用加密 chaincode 的样本。是这样的，在现在的基础 chaincode CAN marry API 的 get 和 put 垫片系动力学研究所和 decrypt encrypt functionality 补充说。

To add the encryption entities extension to your chaincode as a dependency, see Managing external dependencies for chaincode written in Go.

对添加到你的 chaincode 加密扩展的实体作为一个依赖外部管理的方法，这 chaincode dependencies 编剧在去。

11、Managing external dependencies for chaincode written in Go

11、外部管理为用 GO 写的智能合约

If your chaincode requires packages not provided by the Go standard library, you will need to include those packages with your chaincode. It is also a good practice to add the shim and any extension libraries to your chaincode as a dependency.

如果你不 chaincode requires packages 由标准库提供的游戏，你将与你的 chaincode packages to include 的那些。它是一个好的做法，也添加到任何的垫片和扩展到你的 chaincode 图书馆作为一个依赖。

There are many tools available for managing (or “vendoring”) these dependencies. The following demonstrates how to use govendor:

有许多方法都是可用的管理工具（或 “vendoring” 这些 dependencies）。在下面的 demonstrates 如何对使用的 govendor:

```
govendor init
govendor add +external // Add all external package, or
govendor add github.com/external/pkg // Add specific external package
```

This imports the external dependencies into a local vendor directory. If you are vendoring the Fabric shim or shim extensions, clone the Fabric repository to your \$GOPATH/src/github.com/hyperledger directory, before executing the govendor commands.

这 imports 成本地厂商的外部 dependencies 目录。如果你是在 vendoring 纤维垫片或垫片的扩展，克隆人战争的 repository 纤维对你的 gopath 美元/ / / hyperledger github.com Src 目录，在 govendor commands 通常执行的。

Once dependencies are vendored in your chaincode directory, peer chaincode package and peer chaincode install operations will then include code associated with the dependencies into the chaincode package.

一次是在你的 chaincode vendored dependencies 目录，chaincode 屁屁会 chaincode 包装和安装业务相关的代码，然后用 include 的 chaincode dependencies 成包。

九、Chaincode for Operators

九、面向操作员智能合约

1、What is Chaincode?

1、什么是智能合约？

Chaincode is a program, written in Go, node.js, or Java that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.

链式代码是一个程序，写在 GO、NoDE.js 或 Java 中，实现了指定的接口。链码运行在一个安全的 Docker 容器中，该容器与认可对等进程隔离。chaincode 通过应用程序提交的事务初始化和分类帐状态。

A chaincode typically handles business logic agreed to by members of the network, so it may be considered as a “smart contract”. State created by a chaincode is scoped exclusively to that chaincode and can't be accessed directly by another chaincode. However, within the same network, given the appropriate permission a chaincode may invoke another chaincode to access its state.

链码通常处理网络成员同意的业务逻辑，因此它可以被视为“智能合约”。由链码创建的状态仅限于该链码，不能由其他链码直接访问。但是，在同一个网络中，如果获得适当的许可，链码可以调用另一个链码来访问其状态。

In the following sections, we will explore chaincode through the eyes of a blockchain network operator, Noah. For Noah's interests, we will focus on chaincode lifecycle operations; the process of packaging, installing, instantiating and upgrading the chaincode as a function of the chaincode's operational lifecycle within a blockchain network.

在下面的章节中，我们将通过区块链网络运营商 Noah 的眼睛来探索链码。为了诺亚的利益，我们将专注于链码生命周期操作：打包、安装、实例化和升级链码的过程，作为链码在区块链网络内运行生命周期的函数。

2、Chaincode lifecycle

2、链码生命周期

The Hyperledger Fabric API enables interaction with the various nodes in a blockchain network – the peers, orderers and MSPs – and it also allows one to package, install, instantiate and upgrade chaincode on the endorsing peer nodes. The Hyperledger Fabric language-specific SDKs abstract the specifics of the Hyperledger Fabric API to facilitate application development, though it can be used to manage a chaincode's lifecycle. Additionally, the Hyperledger Fabric API can be accessed directly via the CLI, which we will use in this document.

Hyperledger Fabric API 可以与区块链网络中的各个节点（对等点、订购方和 MSP）进行交互，还可以在认可的对等点上打包、安装、实例化和升级链码。hyperledgerfabric 语言特定的 sdk 抽象了 hyperledgerfabric API 的细节，以促进应用程序开发，尽管它可以用于管理链代码的生命周期。此外，可以通过 CLI 直接访问 Hyperledger Fabric API，我们将在本文档中使用该 API。

We provide four commands to manage a chaincode's lifecycle: package, install,

instantiate, and upgrade. In a future release, we are considering adding stop and start transactions to disable and re-enable a chaincode without having to actually uninstall it. After a chaincode has been successfully installed and instantiated, the chaincode is active (running) and can process transactions via the invoke transaction. A chaincode may be upgraded any time after it has been installed.

我们提供了四个命令来管理链代码的生命周期：打包、安装、实例化和升级。在未来的版本中，我们正在考虑添加停止和启动事务来禁用和重新启用链码，而不必实际卸载它。成功安装和实例化链代码后，链代码处于活动状态（正在运行），可以通过调用事务处理事务。链码可以在安装后随时升级。

3、Packaging

3、包装

The chaincode package consists of 3 parts:

链码包由 3 部分组成：

the chaincode, as defined by ChaincodeDeploymentSpec or CDS. The CDS defines the chaincode package in terms of the code and other properties such as name and version,

由 chaincodeDeploymentSpec 或 cds 定义的链码。CDS 根据代码和其他属性（如名称和版本）定义了 chaincode 包。

an optional instantiation policy which can be syntactically described by the same policy used for endorsement and described in Endorsement policies, and
可选的实例化策略，可由用于背书的同一策略语法描述，并在背书策略中描述，以及
a set of signatures by the entities that “own” the chaincode.

“拥有”链码的实体的一组签名。

The signatures serve the following purposes:

签名的目的如下：

- to establish an ownership of the chaincode,

- 要建立链码的所有权，

- to allow verification of the contents of the package, and

- 允许验证包装内容物，以及

- to allow detection of package tampering.

- 允许检测包装篡改。

The creator of the instantiation transaction of the chaincode on a channel is validated against the instantiation policy of the chaincode.

根据链码的实例化策略，对通道上链码的实例化事务的创建者进行验证。

Creating the package

正在创建包

There are two approaches to packaging chaincode. One for when you want to have multiple owners of a chaincode, and hence need to have the chaincode package signed by multiple identities. This workflow requires that we initially create a signed chaincode package (a SignedCDS) which is subsequently passed serially to each of the other owners for signing.

有两种方法可以打包链码。当您希望拥有一个链码的多个所有者，因此需要用多个标识对链码包进行签名时，可以使用一个。此工作流程要求我们首先创建一个已签名的链码包 (signedCds)，然后将其串行传递给每个其他所有者进行签名。

The simpler workflow is for when you are deploying a SignedCDS that has only the signature of the identity of the node that is issuing the install transaction. 更简单的工作流用于部署仅具有发出安装事务的节点标识签名的已签名 DCD。

We will address the more complex case first. However, you may skip ahead to the Installing chaincode section below if you do not need to worry about multiple owners just yet.

我们将首先处理更复杂的案件。但是，如果您还不需要担心多个所有者的话，可以跳到下面的 InstallingChaincode 部分。

To create a signed chaincode package, use the following command:

要创建签名的链代码包，请使用以下命令：

```
peer chaincode package -n mycc -p
github.com/hyperledger/fabric/examples/chaincode/go/example02/cmd -v 0 -s -S -i
"AND('OrgA.admin')" ccpack.out
```

The -s option creates a package that can be signed by multiple owners as opposed to simply creating a raw CDS. When -s is specified, the -S option must also be specified if other owners are going to need to sign. Otherwise, the process will create a SignedCDS that includes only the instantiation policy in addition to the CDS.

-s 选项创建一个可由多个所有者签名的包，而不是简单地创建原始 CD。当指定了 -s 时，如果其他所有者需要签名，则还必须指定 -s 选项。否则，进程将创建一个已签名的 DCD，除了 CDS 之外，它只包含实例化策略。

The -S option directs the process to sign the package using the MSP identified by the value of the localMspid property in core.yaml.

-s 选项指示进程使用由 core.yaml 中 localmspids 属性的值标识的 msp 对包进行签名。

The -S option is optional. However if a package is created without a signature, it cannot be signed by any other owner using the signpackage command.

-s 选项是可选的。但是，如果创建的包没有签名，则任何其他所有者都不能使用 signpackage 命令对其进行签名。

The optional -i option allows one to specify an instantiation policy for the chaincode. The instantiation policy has the same format as an endorsement policy and specifies which identities can instantiate the chaincode. In the example above, only the admin of OrgA is allowed to instantiate the chaincode. If no policy is provided, the default policy is used, which only allows the admin identity of the peer's MSP to instantiate chaincode.

可选的 -i 选项允许为链代码指定实例化策略。实例化策略的格式与认可策略的格式相同，并指定哪些标识可以实例化链码。在上面的示例中，只允许 Orga 的管理员实例化链代码。如果没有提供策略，则使用默认策略，该策略只允许对等的 MSP 的管理员标识实例化链代码。

Package signing

包签

A chaincode package that was signed at creation can be handed over to other

owners for inspection and signing. The workflow supports out-of-band signing of chaincode package.

创建时签名的链码包可以移交给其他所有者进行检查和签名。工作流支持链码包的带外签名。

The ChaincodeDeploymentSpec may be optionally be signed by the collective owners to create a SignedChaincodeDeploymentSpec (or SignedCDS). The SignedCDS contains 3 elements:

chaincodedeploymentspec 可以由集体所有者选择性地签名，以创建签名的 chaincodedeploymentspec (或签名的 dcd)。签名的 DCD 包含 3 个元素：

The CDS contains the source code, the name, and version of the chaincode.

CD 包含源代码、链代码的名称和版本。

An instantiation policy of the chaincode, expressed as endorsement policies.

链式代码的实例化策略，表示为认可策略。

The list of chaincode owners, defined by means of Endorsement.

通过背书定义的链码所有者列表。

Note

注释

Note that this endorsement policy is determined out-of-band to provide proper MSP principals when the chaincode is instantiated on some channels. If the instantiation policy is not specified, the default policy is any MSP administrator of the channel.

请注意，当链码在某些通道上实例化时，此认可策略是在带外确定的，以提供适当的 MSP 主体。如果未指定实例化策略，则默认策略为该通道的任何 MSP 管理员。

Each owner endorses the ChaincodeDeploymentSpec by combining it with that owner's identity (e.g. certificate) and signing the combined result.

每个所有者通过将 chaincodedeploymentspec 与该所有者的身份（例如证书）组合并签署组合结果来认可 chaincodedeploymentspec。

A chaincode owner can sign a previously created signed package using the following command:

chaincode 所有者可以使用以下命令对以前创建的已签名包进行签名：

```
peer chaincode signpackage ccpack.out signedccpack.out
```

Where ccpack.out and signedccpack.out are the input and output packages, respectively. signedccpack.out contains an additional signature over the package signed using the Local MSP.

其中 ccpack.out 和 signedccpack.out 分别是输入和输出包。signedccpack.out 包含使用本地 MSP 签名的包上的附加签名。

Installing chaincode

安装链码

The install transaction packages a chaincode's source code into a prescribed format called a ChaincodeDeploymentSpec (or CDS) and installs it on a peer node that will run that chaincode.

安装事务将链码的源代码打包成一种指定的格式，称为 chaincodedeploymentspec (或 cds)，并将其安装在将运行该链码的对等节点上。

Note

注释

You must install the chaincode on each endorsing peer node of a channel that will run your chaincode.

必须在运行链码的渠道的每个认可对等节点上安装链码。

When the install API is given simply a ChaincodeDeploymentSpec, it will default the instantiation policy and include an empty owner list.

当只给 install api 一个 chaincodeDeploymentSpec 时，它将默认实例化策略并包含一个空的所有者列表。

Note

注释

Chaincode should only be installed on endorsing peer nodes of the owning members of the chaincode to protect the confidentiality of the chaincode logic from other members on the network. Those members without the chaincode, can't be the endorsers of the chaincode's transactions; that is, they can't execute the chaincode. However, they can still validate and commit the transactions to the ledger.

chaincode 只能安装在拥有 chaincode 成员的认可对等节点上，以保护 chaincode 逻辑的机密性不受网络上其他成员的影响。没有 chaincode 的成员不能成为 chaincode 事务的代言人；也就是说，他们不能执行 chaincode。但是，他们仍然可以验证交易并将其提交到分类帐。

To install a chaincode, send a SignedProposal to the lifecycle system chaincode (LSCC) described in the System Chaincode section. For example, to install the sacc sample chaincode described in section Simple Asset Chaincode using the CLI, the command would look like the following:

要安装 chaincode，请将已签名的路径发送到系统 chaincode 部分中描述的生命周期系统 chaincode (lsc)。例如，要使用 CLI 安装简单资产链代码一节中描述的 SACC 示例链代码，该命令如下所示：

```
peer chaincode install -n asset_mgmt -v 1.0 -p sacc
```

The CLI internally creates the SignedChaincodeDeploymentSpec for sacc and sends it to the local peer, which calls the Install method on the LSCC. The argument to the -p option specifies the path to the chaincode, which must be located within the source tree of the user's GOPATH, e.g. \$GOPATH/src/sacc. Note if using -l node or -l java for node chaincode or java chaincode, use -p with the absolute path of the chaincode location. See the Commands Reference for a complete description of the command options.

cli 在内部为 sac 创建 signedchaincodeDeploymentSpec 并将其发送给本地对等方，后者在 lsc 上调用 install 方法。-p 选项的参数指定链码的路径，链码必须位于用户 GOPATH 的源树中，例如 \$GOPATH/src/sac。注意，如果使用 -L 节点或 -L Java 用于节点链码或 Java 链码，则使用 -p 与链码位置的绝对路径。有关命令选项的完整描述，请参见“命令参考”。

Note that in order to install on a peer, the signature of the SignedProposal must be from 1 of the peer's local MSP administrators.

请注意，要在对等机上安装，签名的 dproposal 的签名必须来自对等机的本地 MSP 管理员中的一个。

Instantiate

实例化

The instantiate transaction invokes the lifecycle System Chaincode (LSCC) to create and initialize a chaincode on a channel. This is a chaincode-channel binding process: a chaincode may be bound to any number of channels and operate on each channel individually and independently. In other words, regardless of how many other channels on which a chaincode might be installed and instantiated, state is kept isolated to the channel to which a transaction is submitted.

实例化事务调用 LifecycleSystemChaincode (LSCC) 来创建和初始化通道上的 Chaincode。这是一个链码通道绑定过程：一个链码可以绑定到任意数量的通道上，并分别独立地操作每个通道。换句话说，无论安装和实例化链码的其他通道有多少个，状态都与提交事务的通道保持隔离。

The creator of an instantiate transaction must satisfy the instantiation policy of the chaincode included in SignedCDS and must also be a writer on the channel, which is configured as part of the channel creation. This is important for the security of the channel to prevent rogue entities from deploying chaincodes or tricking members to execute chaincodes on an unbound channel.

实例化事务的创建者必须满足 signedCDS 中包含的链码的实例化策略，并且必须是通道上的编写器，该通道被配置为通道创建的一部分。这对于通道的安全性很重要，以防止流氓实体部署链码或诱使成员在未绑定通道上执行链码。

For example, recall that the default instantiation policy is any channel MSP administrator, so the creator of a chaincode instantiate transaction must be a member of the channel administrators. When the transaction proposal arrives at the endorser, it verifies the creator's signature against the instantiation policy. This is done again during the transaction validation before committing it to the ledger.

例如，回想一下，默认的实例化策略是任何通道 MSP 管理员，因此 chaincode 实例化事务的创建者必须是通道管理员的成员。当事务建议到达背书人时，它根据实例化策略验证创建者的签名。在将交易验证提交到分类帐之前，再次执行此操作。

The instantiate transaction also sets up the endorsement policy for that chaincode on the channel. The endorsement policy describes the attestation requirements for the transaction result to be accepted by members of the channel.

实例化事务还为通道上的链代码设置认可策略。背书政策描述了渠道成员接受交易结果的认证要求。

For example, using the CLI to instantiate the sacc chaincode and initialize the state with john and 0, the command would look like the following:

例如，使用 cli 实例化 sac 链码并使用 john 和 0 初始化状态，该命令如下所示：

```
peer chaincode instantiate -n sacc -v 1.0 -c '{"Args":["john","0"]}' -P "AND ('Org1.member','Org2.member')"
```

Note

注释

Note the endorsement policy (CLI uses polish notation), which requires an endorsement from both a member of Org1 and Org2 for all transactions to sacc. That is, both Org1 and Org2 must sign the result of executing the Invoke on sacc for the transactions to be valid.

注意背书策略（cli 使用波兰符号），它要求 org1 和 org2 的成员对 SAC 的所有事务都进行背书。也就是说，org1 和 org2 都必须在 sac 上执行调用的结果上签名，以使事务有效。

After being successfully instantiated, the chaincode enters the active state on the channel and is ready to process any transaction proposals of type ENDORSER_TRANSACTION. The transactions are processed concurrently as they arrive at the endorsing peer.

成功实例化后，chaincode 在通道上进入活动状态，并准备好处理任何类型的背书人事务的事务建议。当交易到达认可的对等方时，它们被同时处理。

Upgrade

升级

A chaincode may be upgraded any time by changing its version, which is part of the SignedCDS. Other parts, such as owners and instantiation policy are optional. However, the chaincode name must be the same; otherwise it would be considered as a totally different chaincode.

链码可以随时通过更改其版本进行升级，该版本是已签名的 DCD 的一部分。其他部分，如所有者和实例化策略是可选的。但是，chaincode 名称必须相同；否则它将被视为完全不同的 chaincode。

Prior to upgrade, the new version of the chaincode must be installed on the required endorsers. Upgrade is a transaction similar to the instantiate transaction, which binds the new version of the chaincode to the channel. Other channels bound to the old version of the chaincode still run with the old version. In other words, the upgrade transaction only affects one channel at a time, the channel to which the transaction is submitted.

在升级之前，必须在所需的背书人上安装新版本的 chaincode。升级是一个类似于实例化事务的事务，它将新版本的链码绑定到通道。绑定到旧版本的链码的其他通道仍然使用旧版本运行。换句话说，升级事务一次只影响一个通道，即提交事务的通道。

Note

注释

Note that since multiple versions of a chaincode may be active simultaneously, the upgrade process doesn't automatically remove the old versions, so user must manage this for the time being.

请注意，由于一个链码的多个版本可能同时处于活动状态，因此升级过程不会自动删除旧版本，因此用户必须暂时管理此版本。

There's one subtle difference with the instantiate transaction: the upgrade transaction is checked against the current chaincode instantiation policy, not the new policy (if specified). This is to ensure that only existing members specified in the current instantiation policy may upgrade the chaincode.

与实例化事务有一个细微的区别：升级事务是根据当前的 chaincode 实例化策略检查的，而不是新策略（如果指定的话）。这是为了确保只有当前实例化策略中指定的现有成员才能升级链码。

Note

注释

Note that during upgrade, the chaincode Init function is called to perform any data related updates or re-initialize it, so care must be taken to avoid resetting

states when upgrading chaincode.

请注意，在升级过程中，调用 `chaincode init` 函数来执行任何与数据相关的更新或重新初始化，因此在升级 chaincode 时必须小心避免重置状态。

Stop and Start

停止启动

Note that stop and start lifecycle transactions have not yet been implemented. However, you may stop a chaincode manually by removing the chaincode container and the SignedCDS package from each of the endorsers. This is done by deleting the chaincode's container on each of the hosts or virtual machines on which the endorsing peer nodes are running, and then deleting the SignedCDS from each of the endorsing peer nodes:

请注意，停止和启动生命周期事务尚未实现。但是，您可以通过从每个背书人中删除 chaincode 容器和已签名的 dcds 包来手动停止 chaincode。这可以通过删除运行认可对等节点的每个主机或虚拟机上的链码容器来完成，然后从每个认可对等节点中删除签署的 DCD:

Note

注释

TODO - in order to delete the CDS from the peer node, you would need to enter the peer node's container, first. We really need to provide a utility script that can do this.

TODO-要删除对等节点的 CD，您需要首先输入对等节点的容器。我们真的需要提供一个可以做到这一点的实用程序脚本。

```
docker rm -f <container id>
```

```
rm /var/hyperledger/production/chaincodes/<ccname>:<ccversion>
```

Stop would be useful in the workflow for doing upgrade in controlled manner, where a chaincode can be stopped on a channel on all peers before issuing an upgrade.

Stop 在以受控方式进行升级的工作流中非常有用，在发布升级之前，可以在所有对等端的通道上停止链码。

十、System Chaincode Plugins

十、系统智能合约插件

System chaincodes are specialized chaincodes that run as part of the peer process as opposed to user chaincodes that run in separate docker containers. As such they have more access to resources in the peer and can be used for implementing features that are difficult or impossible to be implemented through user chaincodes. Examples of System Chaincodes include QSCC (Query System Chaincode) for ledger and other Fabric-related queries, CSCC (Configuration System Chaincode) which helps regulate access control, and LSCC (Lifecycle System Chaincode).

系统链码是作为对等进程的一部分运行的专用链码，而不是在单独的 Docker 容器中运行的用户链码。因此，它们对对等端中的资源具有更多的访问权限，可以用于实现难以或不可能通过用户链码实现的功能。系统链码的例子包括用于分类帐和其他与结构相关的查询的 qsc（查询系统链码）、帮助管理访问控制的 csc（配置系统链码）和 lsc（生命周期系

统链码)。

Unlike a user chaincode, a system chaincode is not installed and instantiated using proposals from SDKs or CLI. It is registered and deployed by the peer at start-up.

与用户链代码不同，系统链代码没有使用来自 `sdk` 或 `cli` 的建议进行安装和实例化。它由对等机在启动时注册和部署。

System chaincodes can be linked to a peer in two ways: statically, and dynamically using Go plugins. This tutorial will outline how to develop and load system chaincodes as plugins.

系统链代码可以通过两种方式链接到对等端：静态和动态使用 `go` 插件。本教程将概述如何开发和加载系统链代码作为插件。

Developing Plugins

开发插件

A system chaincode is a program written in Go and loaded using the Go plugin package.

系统智能合约是用 `go` 插件包编写和加载的程序。

A plugin includes a main package with exported symbols and is built with the command `go build -buildmode=plugin`.

插件包含一个带有导出符号的主包，并使用命令 `go build -buildmode=plugin` 生成。

Every system chaincode must implement the Chaincode Interface and export a constructor method that matches the signature `func New() shim.Chaincode` in the main package. An example can be found in the repository at `examples/plugin/scc`.

每个系统链码都必须实现链码接口，并导出一个与主包中的签名 `func new() shim.Chaincode` 匹配的构造函数方法。示例可以在存储库的 `examples/plugin/scc` 中找到。

Existing chaincodes such as the QSCC can also serve as templates for certain features, such as access control, that are typically implemented through system chaincodes. The existing system chaincodes also serve as a reference for best-practices on things like logging and testing.

现有的链码（如 `qsc`）也可以用作某些特性（如访问控制）的模板，这些特性通常通过系统链码实现。现有的系统链代码也可作为日志记录和测试等方面的最佳实践的参考。

Note

注释

On imported packages: the Go standard library requires that a plugin must include the same version of imported packages as the host application (Fabric, in this case).

对于导入的包：Go 标准库要求插件必须包含与宿主应用程序（在本例中是 Fabric）相同版本的导入包。

Configuring Plugins

配置插件

Plugins are configured in the `chaincode.systemPlugin` section in `core.yaml`:

插件配置在 `core.yaml` 中的 `chaincode.systemplugin` 部分：

```
chaincode:
  systemPlugins:
    - enabled: true
```

```
name: mysyscc
path: /opt/lib/syscc.so
invokableExternal: true
invokableCC2CC: true
```

A system chaincode must also be whitelisted in the chaincode.system section in core.yaml:

系统链码也必须在 core.yaml 中的 chaincode.system 部分中白色列出:

```
chaincode:
  system:
    mysyscc: enable
```

十一、Using CouchDB

十一、使用 CouchDB

This tutorial will describe the steps required to use the CouchDB as the state database with Hyperledger Fabric. By now, you should be familiar with Fabric concepts and have explored some of the samples and tutorials.

本教程将描述使用 couchdb 作为带有 hyperledger 结构的状态数据库所需的步骤。到目前为止，您应该已经熟悉了结构概念，并且已经探索了一些示例和教程。

The tutorial will take you through the following steps:

本教程将指导您完成以下步骤:

Enable CouchDB in Hyperledger Fabric

在 Hyperledger 结构中启用 CouchDB

Create an index

创建索引

Add the index to your chaincode folder

将索引添加到 chaincode 文件夹中

Install and instantiate the Chaincode

安装并实例化链代码

Query the CouchDB State Database

查询 couchdb 状态数据库

Query the CouchDB State Database With Pagination

使用分页查询 couchdb 状态数据库

Update an Index

更新索引

Delete an Index

删除索引

For a deeper dive into CouchDB refer to CouchDB as the State Database and for more information on the Fabric ledger refer to the Ledger topic. Follow the tutorial below for details on how to leverage CouchDB in your blockchain network.

要深入了解 couchdb，请参阅 couchdb 作为状态数据库，有关结构分类账的更多信息，请参阅分类账主题。有关如何在区块链网络中利用 CouchDB 的详细信息，请遵循以下教程。

Throughout this tutorial we will use the Marbles sample as our use case to demonstrate how to use CouchDB with Fabric and will deploy Marbles to the Building

Your First Network (BYFN) tutorial network. You should have completed the task Install Samples, Binaries and Docker Images. However, running the BYFN tutorial is not a prerequisite for this tutorial, instead the necessary commands are provided throughout this tutorial to use the network.

在本教程中，我们将使用 Marbles 示例作为用例，演示如何将 couchdb 与结构一起使用，并将大理石部署到构建第一个网络（byfn）教程网络。您应该已经完成了安装示例、二进制文件和 Docker 映像的任务。但是，运行 byfn 教程并不是本教程的先决条件，而是在本教程中提供使用网络所需的命令。

1、Why CouchDB?

1、为什么 CouchDB?

Fabric supports two types of peer databases. LevelDB is the default state database embedded in the peer node and stores chaincode data as simple key-value pairs and supports key, key range, and composite key queries only. CouchDB is an optional alternate state database that supports rich queries when chaincode data values are modeled as JSON. Rich queries are more flexible and efficient against large indexed data stores, when you want to query the actual data value content rather than the keys. CouchDB is a JSON document datastore rather than a pure key-value store therefore enabling indexing of the contents of the documents in the database.

Fabric 支持两种类型的对等数据库。LEVELDB 是嵌入在对等节点中的默认状态数据库，它将链码数据存储为简单的键值对，并且仅支持键、键范围和复合键查询。couchdb 是一个可选的备用状态数据库，当 chaincode 数据值被建模为 json 时，它支持丰富的查询。当您希望查询实际数据值内容而不是键时，富查询对于大型索引数据存储更灵活和高效。couchdb 是一个 JSON 文档数据存储，而不是一个纯粹的键值存储，因此可以对数据库中的文档内容进行索引。

In order to leverage the benefits of CouchDB, namely content-based JSON queries, your data must be modeled in JSON format. You must decide whether to use LevelDB or CouchDB before setting up your network. Switching a peer from using LevelDB to CouchDB is not supported due to data compatibility issues. All peers on the network must use the same database type. If you have a mix of JSON and binary data values, you can still use CouchDB, however the binary values can only be queried based on key, key range, and composite key queries.

为了充分利用 couchdb 的好处，即基于内容的 JSON 查询，您的数据必须以 JSON 格式建模。在设置网络之前，必须决定是使用 leveldb 还是 couchdb。由于数据兼容性问题，不支持将对等端从使用 leveldb 切换到 couchdb。网络上的所有对等方必须使用相同的数据库类型。如果您混合了 JSON 和二进制数据值，那么仍然可以使用 couchdb，但是只能基于键、键范围和复合键查询来查询二进制值。

2、Enable CouchDB in Hyperledger Fabric

2、在 Hyperledger 结构中启用 CouchDB

CouchDB runs as a separate database process alongside the peer, therefore

there are additional considerations in terms of setup, management, and operations. A docker image of CouchDB is available and we recommend that it be run on the same server as the peer. You will need to setup one CouchDB container per peer and update each peer container by changing the configuration found in `core.yaml` to point to the CouchDB container. The `core.yaml` file must be located in the directory specified by the environment variable `FABRIC_CFG_PATH`:

CouchDB 作为一个独立的数据库进程与对等进程一起运行，因此在设置、管理和操作方面还有其他的考虑。CouchDB 的 Docker 映像可用，我们建议它与对等服务器在同一服务器上运行。您需要为每个对等机设置一个 couchdb 容器，并通过更改 `core.yaml` 中的配置来更新每个对等机容器，以指向 couchdb 容器。`core.yaml` 文件必须位于环境变量 `fabric_cfg_path` 指定的目录中：

For docker deployments, `core.yaml` is pre-configured and located in the peer container `FABRIC_CFG_PATH` folder. However when using docker environments, you typically pass environment variables by editing the `docker-compose-couch.yaml` to override the `core.yaml`

对于 Docker 部署，`core.yaml` 是预先配置的，位于 peer container `fabric_cfg_path` 文件夹中。但是，在使用 Docker 环境时，通常通过编辑 `docker-compose-coach.yaml` 来覆盖 `core.yaml` 来传递环境变量。

For native binary deployments, `core.yaml` is included with the release artifact distribution.

对于本机二进制部署，`core.yaml` 包含在发布工件分发中。

Edit the `stateDatabase` section of `core.yaml`. Specify CouchDB as the `stateDatabase` and fill in the associated `couchDBConfig` properties. For more details on configuring CouchDB to work with fabric, refer here. To view an example of a `core.yaml` file configured for CouchDB, examine the `BYFN docker-compose-couch.yaml` in the `HyperLedger/fabric-samples/first-network` directory.

编辑 `core.yaml` 的 `statedatabase` 部分。将 `couchdb` 指定为 `statedatabase` 并填写关联的 `couchdbconfig` 属性。有关配置 CouchDB 以使用结构的更多详细信息，请参阅此处。要查看为 `couchdb` 配置的 `core.yaml` 文件的示例，请检查 `hyperledger/fabric samples/first network` 目录中的 `byfn docker-compose-coach.yaml`。

3、Create an index

3、创建索引

Why are indexes important?

为什么索引很重要？

Indexes allow a database to be queried without having to examine every row with every query, making them run faster and more efficiently. Normally, indexes are built for frequently occurring query criteria allowing the data to be queried more efficiently. To leverage the major benefit of CouchDB - the ability to perform rich queries against JSON data - indexes are not required, but they are strongly recommended for performance. Also, if sorting is required in a query, CouchDB requires an index of the sorted fields.

索引允许查询数据库，而不必使用每个查询检查每一行，从而使数据库运行更快、更高

效。通常，索引是为频繁出现的查询条件构建的，这样可以更有效地查询数据。为了充分利用 couchdb 的主要优势（能够对 JSON 数据执行丰富的查询），不需要索引，但强烈建议使用索引来提高性能。此外，如果查询中需要排序，则 couchdb 需要排序字段的索引。

Note

注释

Rich queries that do not have an index will work but may throw a warning in the CouchDB log that the index was not found. However, if a rich query includes a sort specification, then an index on that field is required; otherwise, the query will fail and an error will be thrown.

没有索引的富查询将工作，但可能会在 couchdb 日志中引发一条警告，指出找不到索引。但是，如果富查询包含排序规范，则需要该字段的索引；否则，查询将失败并引发错误。

To demonstrate building an index, we will use the data from the Marbles sample. In this example, the Marbles data structure is defined as:

为了演示建立索引，我们将使用 Marbles 样本中的数据。在本例中，Marbles 数据结构定义为：

```
type marble struct {
    ObjectType string `json:"docType"` //docType is used to distinguish the
various types of objects in state database
    Name string `json:"name"` //the field tags are needed to keep case from
bouncing around
    Color string `json:"color"`
    Size int `json:"size"`
    Owner string `json:"owner"`
}
```

In this structure, the attributes (docType, name, color, size, owner) define the ledger data associated with the asset. The attribute docType is a pattern used in the chaincode to differentiate different data types that may need to be queried separately. When using CouchDB, it recommended to include this docType attribute to distinguish each type of document in the chaincode namespace. (Each chaincode is represented as its own CouchDB database, that is, each chaincode has its own namespace for keys.)

在此结构中，属性 (doctype、name、color、size、owner) 定义与资产关联的分类帐数据。属性 doctype 是链式代码中使用的一种模式，用于区分可能需要单独查询的不同数据类型。使用 couchdb 时，建议包含此 doctype 属性以区分 chaincode 命名空间中的每种文档类型。（每个链码都表示为自己的 couchdb 数据库，也就是说，每个链码都有自己的键命名空间。）

With respect to the Marbles data structure, docType is used to identify that this document/asset is a marble asset. Potentially there could be other documents/assets in the chaincode database. The documents in the database are searchable against all of these attribute values.

对于 Marbles 数据结构，doctype 用于标识此文档/资产是大理石资产。链码数据库中可能存在其他文档/资产。数据库中的文档可以根据所有这些属性值进行搜索。

When defining an index for use in chaincode queries, each one must be defined in its own text file with the extension *.json and the index definition must be

formatted in the CouchDB index JSON format.

定义用于链码查询的索引时，每个索引都必须在自己的文本文件中定义，扩展名为 *.json，并且索引定义必须以 couchdb index json 格式格式化。

To define an index, three pieces of information are required:

要定义索引，需要三条信息：

fields: these are the frequently queried fields

name: name of the index

type: always json in this context

For example, a simple index named foo-index for a field named foo.

例如，名为 foo 的字段的一个名为 foo index 的简单索引。

```
{
  "index": {
    "fields": ["foo"]
  },
  "name" : "foo-index",
  "type" : "json"
}
```

Optionally the design document attribute ddoc can be specified on the index definition. A design document is CouchDB construct designed to contain indexes. Indexes can be grouped into design documents for efficiency but CouchDB recommends one index per design document.

也可以在索引定义上指定设计文档属性 ddoc。设计文档是为包含索引而设计的 couchdb 结构。为了提高效率，索引可以分为设计文档，但 CouchDB 建议每个设计文档使用一个索引。

Tip

小技巧

When defining an index it is a good practice to include the ddoc attribute and value along with the index name. It is important to include this attribute to ensure that you can update the index later if needed. Also it gives you the ability to explicitly specify which index to use on a query.

在定义索引时，最好将 ddoc 属性和值与索引名一起包含。包含此属性是很重要的，以确保您以后可以根据需要更新索引。它还使您能够显式地指定要在查询上使用的索引。

Here is another example of an index definition from the Marbles sample with the index name indexOwner using multiple fields docType and owner and includes the ddoc attribute:

下面是 Marbles 示例中索引定义的另一个示例，索引名为 index owner，使用多个字段 doctype 和 owner，包括 ddoc 属性：

```
{
  "index":{
    "fields":["docType","owner"] // Names of the fields to be queried
  },
  "ddoc":"indexOwnerDoc", // (optional) Name of the design document in which
the index will be created.
  "name":"indexOwner",
}
```

```
"type": "json"
}
```

In the example above, if the design document `indexOwnerDoc` does not already exist, it is automatically created when the index is deployed. An index can be constructed with one or more attributes specified in the list of fields and any combination of attributes can be specified. An attribute can exist in multiple indexes for the same docType. In the following example, `index1` only includes the attribute `owner`, `index2` includes the attributes `owner` and `color` and `index3` includes the attributes `owner`, `color` and `size`. Also, notice each index definition has its own `ddoc` value, following the CouchDB recommended practice.

在上面的示例中，如果设计文档 `indexownerDoc` 不存在，则在部署索引时自动创建它。可以使用字段列表中指定的一个或多个属性构造索引，并且可以指定任何属性组合。同一 `doctype` 的多个索引中可以存在一个属性。在下面的示例中，`index1` 只包括属性所有者，`index2` 包括属性所有者和颜色，`index3` 包括属性所有者、颜色和大小。另外，注意每个索引定义都有自己的 `ddoc` 值，遵循 `couchdb` 推荐的实践。

```
{
  "index": {
    "fields": ["owner"] // Names of the fields to be queried
  },
  "ddoc": "index1Doc", // (optional) Name of the design document in which the
index will be created.
  "name": "index1",
  "type": "json"
}
{
  "index": {
    "fields": ["owner", "color"] // Names of the fields to be queried
  },
  "ddoc": "index2Doc", // (optional) Name of the design document in which the
index will be created.
  "name": "index2",
  "type": "json"
}
{
  "index": {
    "fields": ["owner", "color", "size"] // Names of the fields to be queried
  },
  "ddoc": "index3Doc", // (optional) Name of the design document in which the
index will be created.
  "name": "index3",
  "type": "json"
}
```

In general, you should model index fields to match the fields that will be used in query filters and sorts. For more details on building an index in JSON

format refer to the CouchDB documentation.

通常，您应该对索引字段建模，以匹配将在查询筛选和排序中使用的字段。有关以 JSON 格式构建索引的更多详细信息，请参阅 CouchDB 文档。

A final word on indexing, Fabric takes care of indexing the documents in the database using a pattern called index warming. CouchDB does not typically index new or updated documents until the next query. Fabric ensures that indexes stay ‘warm’ by requesting an index update after every block of data is committed. This ensures queries are fast because they do not have to index documents before running the query. This process keeps the index current and refreshed every time new records are added to the state database.

Fabric 是索引的最后一个词，它使用一种称为索引预热的模式来为数据库中的文档编制索引。在下一个查询之前，CouchDB 通常不会为新的或更新的文档编制索引。Fabric 通过在提交每个数据块后请求索引更新来确保索引保持“温暖”。这样可以确保查询速度很快，因为在运行查询之前，它们不必为文档编制索引。这个过程使索引保持当前状态，并在每次向状态数据库添加新记录时刷新索引。

4、Add the index to your chaincode folder

4、将索引添加到 chaincode 文件夹中

Once you finalize an index, it is ready to be packaged with your chaincode for deployment by being placed alongside it in the appropriate metadata folder.

完成索引后，就可以将其与链代码一起打包部署了，方法是将其放在相应的元数据文件夹中。

If your chaincode installation and instantiation uses the Hyperledger Fabric Node SDK, the JSON index files can be located in any folder as long as it conforms to this directory structure. During the chaincode installation using the `client.installChaincode()` API, include the attribute `(metadataPath)` in the installation request. The value of the `metadataPath` is a string representing the absolute path to the directory structure containing the JSON index file(s).

如果您的 chaincode 安装和实例化使用了 hyperledger fabric node sdk，那么只要 JSON 索引文件符合这个目录结构，它就可以位于任何文件夹中。在使用 `client.installChaincode()` API 安装 chaincode 期间，请在安装请求中包含属性 `(metadataPath)`。`metadataPath` 的值是一个字符串，表示包含 JSON 索引文件的目录结构的绝对路径。

Alternatively, if you are using the peer-commands to install and instantiate the chaincode, then the JSON index files must be located under the path `META-INF/statedb/couchdb/indexes` which is located inside the directory where the chaincode resides.

或者，如果使用 peer 命令安装和实例化 chaincode，则 json 索引文件必须位于路径 `meta-inf/statedb/couchdb/indexes` 下，该路径位于 chaincode 所在的目录中。

The Marbles sample below illustrates how the index is packaged with the chaincode which will be installed using the peer commands.

下面的 Marbles 示例说明了如何使用使用 peer 命令安装的 chaincode 打包索引。



Start the network

启动网络

Try it yourself

自己试试

Before installing and instantiating the marbles chaincode, we need to start up the BYFN network. For the sake of this tutorial, we want to operate from a known initial state. The following command will kill any active or stale docker containers and remove previously generated artifacts. Therefore let's run the following command to clean up any previous environments:

在安装和实例化 Marbles 链码之前，我们需要启动 byfn 网络。为了本教程的目的，我们希望从已知的初始状态进行操作。以下命令将杀死任何活动或过时的 Docker 容器，并删除以前生成的工件。因此，让我们运行以下命令来清理以前的任何环境：

```
cd fabric-samples/first-network
```

```
./byfn.sh down
```

Now start up the BYFN network with CouchDB by running the following command:

现在，通过运行以下命令，使用 couchdb 启动 byfn 网络：

```
./byfn.sh up -c mychannel -s couchdb
```

This will create a simple Fabric network consisting of a single channel named mychannel with two organizations (each maintaining two peer nodes) and an ordering service while using CouchDB as the state database.

这将创建一个简单的结构网络，由一个名为 mychannel 的单通道和两个组织（每个组织维护两个对等节点）以及一个订购服务组成，同时使用 couchdb 作为状态数据库。

5、Install and instantiate the Chaincode

5、安装并实例化链代码

Client applications interact with the blockchain ledger through chaincode. As such we need to install the chaincode on every peer that will execute and endorse our transactions and instantiate the chaincode on the channel. In the

previous section, we demonstrated how to package the chaincode so they should be ready for deployment.

客户端应用程序通过链码与区块链分类账交互。因此，我们需要在每一个执行和认可我们的事务的对等端上安装链码，并在通道上实例化链码。在前一节中，我们演示了如何打包链代码，以便它们可以进行部署。

Chaincode is installed onto a peer and then instantiated onto the channel using peer-commands.

chaincode 安装在对等机上，然后使用对等命令实例化到通道上。

1、Use the peer chaincode install command to install the Marbles chaincode on a peer.

1、使用 peer chaincode install 命令在对等机上安装 Marbles chaincode。

Try it yourself

自己试试

Assuming you have started the BYFN network, navigate into the CLI container using the command:

假设您已经启动了 byfn 网络，请使用以下命令导航到 cli 容器：

```
docker exec -it cli bash
```

Use the following command to install the Marbles chaincode from the git repository onto a peer in your BYFN network. The CLI container defaults to using peer0 of org1:

使用以下命令将大理石链代码从 Git 存储库安装到 byfn 网络中的对等机上。cli 容器默认为使用 org1 的 peer0:

```
peer chaincode install -n marbles -v 1.0 -p github.com/chaincode/marbles02/go
```

2、Issue the peer chaincode instantiate command to instantiate the chaincode on a channel.

2、发出 peer chaincode instantiate 命令在通道上实例化 chaincode。

Try it yourself

自己试试

To instantiate the Marbles sample on the BYFN channel mychannel run the following command:

要在 byfn channel mychannel 上实例化大理石示例，请运行以下命令：

```
export CHANNEL_NAME=mychannel
```

```
peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile /opt/gopath/src/github.com/hy
```

Verify index was deployed

验证索引是否已部署

Indexes will be deployed to each peer's CouchDB state database once the chaincode is both installed on the peer and instantiated on the channel. You can verify that the CouchDB index was created successfully by examining the peer log in the Docker container.

一旦在对等机上安装了链码并在通道上实例化了链码，索引就会部署到每个对等机的 couchdb 状态数据库中。您可以通过检查 Docker 容器中的对等日志来验证是否成功创建了 couchdb 索引。

Try it yourself

自己试试

To view the logs in the peer docker container, open a new Terminal window and run the following command to grep for message confirmation that the index was created.

要查看对等 Docker 容器中的日志，请打开一个新的终端窗口，并向 grep 运行以下命令，以确认已创建索引。

```
docker logs peer0.org1.example.com 2>&1 | grep "CouchDB index"
```

You should see a result that looks like the following:

您应该看到如下结果：

```
[couchdb] CreateIndex -> INFO Obe Created CouchDB index [indexOwner] in
state database [mychannel_marbles] using design document [_design/indexOwnerDoc]
```

Note

注释

If Marbles was not installed on the BYFN peer peer0.org1.example.com, you may need to replace it with the name of a different peer where Marbles was installed.

如果在 byfn peer peer0.org1.example.com 上没有安装大理石，则可能需要将其替换为安装大理石的其他对等机的名称。

6、Query the CouchDB State Database

6、查询 couchdb 状态数据库

Now that the index has been defined in the JSON file and deployed alongside the chaincode, chaincode functions can execute JSON queries against the CouchDB state database, and thereby peer commands can invoke the chaincode functions.

现在索引已经在 JSON 文件中定义并与 chaincode 一起部署，chaincode 函数可以对 couchdb 状态数据库执行 JSON 查询，从而对等命令可以调用 chaincode 函数。

Specifying an index name on a query is optional. If not specified, and an index already exists for the fields being queried, the existing index will be automatically used.

在查询上指定索引名称是可选的。如果未指定，并且正在查询的字段已经存在索引，则将自动使用现有索引。

Tip

小技巧

It is a good practice to explicitly include an index name on a query using the use_index keyword. Without it, CouchDB may pick a less optimal index. Also CouchDB may not use an index at all and you may not realize it, at the low volumes during testing. Only upon higher volumes you may realize slow performance because CouchDB is not using an index and you assumed it was.

在使用 use_index 关键字的查询中显式包含索引名是一个好的实践。没有它，CouchDB 可能会选择一个不太理想的索引。同样，couchdb 可能根本不使用索引，并且在测试过程中，在低容量的情况下，您可能没有意识到它。只有在更高的卷上，您才可能实现缓慢的性能，因为 CouchDB 没有使用索引，而您认为它是。

Build the query in chaincode

在 chaincode 中构建查询

You can perform complex rich queries against the chaincode data values using the CouchDB JSON query language within chaincode. As we explored above, the marbles02 sample chaincode includes an index and rich queries are defined in the functions – queryMarbles and queryMarblesByOwner:

您可以使用 chaincode 中的 couchdb json 查询语言对 chaincode 数据值执行复杂的富查询。如上所述，Marbles02 示例链码包含一个索引，丰富的查询在函数 queryMarbles 和 queryMarblesByOwner 中定义：

queryMarbles –

查询 Marbles–

Example of an ad hoc rich query. This is a query where a (selector) string can be passed into the function. This query would be useful to client applications that need to dynamically build their own selectors at runtime. For more information on selectors refer to CouchDB selector syntax.

即席富查询的示例。这是一个可以向函数传递（选择器）字符串的查询。对于需要在运行时动态构建自己的选择器的客户机应用程序，此查询将非常有用。有关选择器的更多信息，请参阅 CouchDB 选择器语法。

queryMarblesByOwner –

查询 MarblesByOwner –

Example of a parameterized query where the query logic is baked into the chaincode. In this case the function accepts a single argument, the marble owner. It then queries the state database for JSON documents matching the docType of “marble” and the owner id using the JSON query syntax.

参数化查询的示例，其中查询逻辑烘焙到链代码中。在这种情况下，函数接受一个参数，即大理石所有者。然后，它使用 JSON 查询语法查询状态数据库中与 doctype “大理石” 和所有者 ID 匹配的 JSON 文档。

Run the query using the peer command

使用 peer 命令运行查询

In absence of a client application to test rich queries defined in chaincode, peer commands can be used. Peer commands run from the command line inside the docker container. We will customize the peer chaincode query command to use the Marbles index indexOwner and query for all marbles owned by “tom” using the queryMarbles function.

如果没有客户端应用程序来测试在 chaincode 中定义的富查询，则可以使用 peer 命令。对等命令从 Docker 容器内的命令行运行。我们将自定义 peer-chaincode 查询命令，以使用大理石索引 indexowner，并使用 query marbles 函数查询 “tom” 拥有的所有大理石。

Try it yourself

自己试试

Before querying the database, we should add some data. Run the following command in the peer container to create a marble owned by “tom” :

在查询数据库之前，我们应该添加一些数据。在对等容器中运行以下命令以创建 “tom” 拥有的大理石：

```
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c
'{"Args":["initMarble","marble1","blue","35","tom"]}'
```

After an index has been deployed during chaincode instantiation, it will automatically be utilized by chaincode queries. CouchDB can determine which index to use based on the fields being queried. If an index exists for the query criteria it will be used. However the recommended approach is to specify the use_index keyword on the query. The peer command below is an example of how to specify the index explicitly in the selector syntax by including the use_index keyword:

在 chaincode 实例化期间部署索引之后，chaincode 查询将自动使用索引。CouchDB 可以根据查询的字段确定要使用的索引。如果查询条件存在索引，则将使用该索引。但是，建议的方法是在查询中指定 use_index 关键字。下面的 peer 命令是一个示例，说明如何通过包含 use_index 关键字在选择器语法中显式指定索引：

```
// Rich Query with index name explicitly specified:
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles",
{"\"selector\":{\"docType\":\"marble\",\"owner\":\"tom\"},
\"use_index\":{\"_design/indexOwnerDoc\", \"indexOwner\"}]"}']'
```

Delving into the query command above, there are three arguments of interest:
深入研究上面的查询命令，有三个感兴趣的参数：

queryMarbles

查询 Marbles

Name of the function in the Marbles chaincode. Notice a shim shim.ChaincodeStubInterface is used to access and modify the ledger. The getQueryResultForQueryString() passes the queryString to the shim API getQueryResult().

大理石链码中函数的名称。请注意，Shim.ChaincodeStubInterface 用于访问和修改分类帐。getQueryResultForQueryString() 将 queryString 传递给填充程序 api getQueryResult()。

```
func (t *SimpleChaincode) queryMarbles(stub shim.ChaincodeStubInterface,
args []string) pb.Response {
    // 0
    // "queryString"
    if len(args) < 1 {
        return shim.Error("Incorrect number of arguments. Expecting 1")
    }
    queryString := args[0]
    queryResults, err := getQueryResultForQueryString(stub, queryString)
    if err != nil {
        return shim.Error(err.Error())
    }
    return shim.Success(queryResults)
}
{"selector":{"docType":"marble","owner":"tom"}}
```

This is an example of an ad hoc selector string which finds all documents of

type marble where the owner attribute has a value of tom.

这是一个特设选择器字符串的示例，它查找所有类型为大理石的文档，其中 owner 属性的值为 tom。

```
"use_index":["_design/indexOwnerDoc", "indexOwner"]
```

Specifies both the design doc name indexOwnerDoc and index name indexOwner. In this example the selector query explicitly includes the index name, specified by using the use_index keyword. Recalling the index definition above Create an index, it contains a design doc, "ddoc":"indexOwnerDoc". With CouchDB, if you plan to explicitly include the index name on the query, then the index definition must include the ddoc value, so it can be referenced with the use_index keyword.

指定设计文档名称 indexownerDoc 和索引名称 indexowner。在本例中，选择器查询显式包含使用 use_index 关键字指定的索引名称。调用上面的索引定义创建一个索引，它包含一个设计文档“ddoc”：“indexownerDoc”。对于 couchdb，如果您计划在查询中显式包含索引名，那么索引定义必须包含 ddoc 值，以便可以使用 use_index 关键字引用它。

The query runs successfully and the index is leveraged with the following results:

查询运行成功，索引与以下结果一起使用：

```
Query                                     Result:                                     [{"Key":"marble1",
"Record":{"color":"blue","docType":"marble","name":"marble1","owner":"to
```

7、Query the CouchDB State Database With Pagination

7、使用分页查询 couchdb 状态数据库

When large result sets are returned by CouchDB queries, a set of APIs is available which can be called by chaincode to paginate the list of results. Pagination provides a mechanism to partition the result set by specifying a pagesize and a start point - a bookmark which indicates where to begin the result set. The client application iteratively invokes the chaincode that executes the query until no more results are returned. For more information refer to this topic on pagination with CouchDB.

当 couchdb 查询返回大型结果集时，可以使用一组 API，这些 API 可以通过 chaincode 调用以分页结果列表。分页提供了一种机制，通过指定页面大小和起始点来划分结果集——一个指示结果集起始位置的书签。客户端应用程序反复调用执行查询的链代码，直到不再返回结果。有关更多信息，请参阅有关 couchdb 分页的主题。

We will use the Marbles sample function queryMarblesWithPagination to demonstrate how pagination can be implemented in chaincode and the client application.

我们将使用 Marbles 示例函数 queryMarblesWithPagination 演示如何在链代码和客户端应用程序中实现分页。

queryMarblesWithPagination -

查询 MarblesWithPagination -

Example of an ad hoc rich query with pagination. This is a query where a (selector) string can be passed into the function similar to the above example. In this case, a pageSize is also included with the query as well as a bookmark.

带分页的即席富查询示例。这是一个可以将（选择器）字符串传递到类似于上面示例的函数中的查询。在这种情况下，页面大小也包括在查询和书签中。

In order to demonstrate pagination, more data is required. This example assumes that you have already added marble1 from above. Run the following commands in the peer container to create four more marbles owned by “tom”, to create a total of five marbles owned by “tom”:

为了演示分页，需要更多的数据。这个例子假设您已经从上面添加了 marble1。在对等容器中运行以下命令，以创建 “tom” 拥有的四个弹珠，以创建 “tom” 拥有的五个弹珠：

Try it yourself

自己试试

```
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c
'{"Args":["initMarble","marble2","yellow","35","tom"]}'
```

```
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c
'{"Args":["initMarble","marble3","green","20","tom"]}'
```

```
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c
'{"Args":["initMarble","marble4","purple","20","tom"]}'
```

```
peer chaincode invoke -o orderer.example.com:7050 --tls --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 $CHANNEL_NAME -n marbles -c
'{"Args":["initMarble","marble5","blue","40","tom"]}'
```

In addition to the arguments for the query in the previous example, queryMarblesWithPagination adds pagesize and bookmark. PageSize specifies the number of records to return per query. The bookmark is an “anchor” telling couchDB where to begin the page. (Each page of results returns a unique bookmark.)

除了上一个示例中查询的参数外，querymarblewithpagination 还添加了 pagesize 和 bookmark。pageSize 指定每个查询返回的记录数。书签是一个“锚”，告诉 CouchDB 从哪里开始页面。（结果的每一页都返回一个唯一的书签。）

queryMarblesWithPagination

查询 MarblesWithPagination

Name of the function in the Marbles chaincode. Notice a shim shim.ChaincodeStubInterface is used to access and modify the ledger. The getQueryResultForQueryStringWithPagination() passes the queryString along

大理石链码中函数的名称。请注意，Shim.ChaincodeStubInterface 用于访问和修改分类帐。getQueryResultForQueryStringWithPagination () 传递查询字符串

with the pageSize and bookmark to the shim API GetQueryResultWithPagination().
将页面大小和书签添加到填充程序 API GetQueryResultWithPagination ()。

```
func (t *SimpleChaincode) queryMarblesWithPagination(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    // 0
    // "queryString"
    if len(args) < 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }
    queryString := args[0]
    //return type of ParseInt is int64
    pageSize, err := strconv.ParseInt(args[1], 10, 32)
    if err != nil {
        return shim.Error(err.Error())
    }
    bookmark := args[2]
    queryResults, err := getQueryResultForQueryStringWithPagination(stub,
        queryString, int32(pageSize), bookmark)
    if err != nil {
        return shim.Error(err.Error())
    }
    return shim.Success(queryResults)
}
```

The following example is a peer command which calls queryMarblesWithPagination with a pageSize of 3 and no bookmark specified.

下面的示例是一个对等命令，它调用页面大小为 3 且未指定书签的 querymarblewithpagination。

Tip

小技巧

When no bookmark is specified, the query starts with the “first” page of records.

如果未指定书签，则查询从记录的“第一”页开始。

Try it yourself

自己试试

```
// Rich Query with index name explicitly specified and a page size of 3:
peer chaincode query -C $CHANNEL_NAME -n marbles -c
'{"Args":["queryMarblesWithPagination",
{"\"selector\":{\"docType\": \"marble\", \"owner\": \"tom\"},
\"use_index\": [\"_design/indexOwnerDoc\", \"indexOwner\"]}, \"3\", \"\"]}'
```

The following response is received (carriage returns added for clarity), three of the five marbles are returned because the pageSize was set to 3:

接收到以下响应（为了清晰起见添加了回车符），五个大理石中的三个返回，因为 pageSize 设置为 3:

```
[{"Key": "marble1",
```

```

"Record":{"color":"blue","docType":"marble","name":"marble1","owner":"tom","size":35}},
  {"Key":"marble2",
"Record":{"color":"yellow","docType":"marble","name":"marble2","owner":"tom","size":35}},
  {"Key":"marble3",
"Record":{"color":"green","docType":"marble","name":"marble3","owner":"tom","size":20}}}
[{"ResponseMetadata":{"RecordsCount":"3",
  "Bookmark":"g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT81PzkzJBYqz5yYWJeWkGoOkOWD
SOSANIFk2iCyIyVySn5uVBQAGEhRz"}]}]

```

Note

注释

Bookmarks are uniquely generated by CouchDB for each query and represent a placeholder in the result set. Pass the returned bookmark on the subsequent iteration of the query to retrieve the next set of results.

书签由 couchdb 为每个查询唯一生成，并表示结果集中的一个占位符。在查询的后续迭代中传递返回的书签，以检索下一组结果。

The following is a peer command to call queryMarblesWithPagination with a pageSize of 3. Notice this time, the query includes the bookmark returned from the previous query.

以下是一个对等命令，用于调用页面大小为 3 的 querymarblewithpagination。请注意，这次查询包括从上一个查询返回的书签。

Try it yourself

自己试试

```

peer chaincode query -C $CHANNEL_NAME -n marbles -c
'{"Args":["queryMarblesWithPagination",
{"\\"selector\":"{\\"docType\":"\\"marble\\"","\\"owner\":"\\"tom\\""},
\\"use_index\":[\\"_design/indexOwnerDoc\\",
\\"indexOwner\\"],"3","g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT81PzkzJBYqz5yYWJeWkGoOkOWD
SOSANIFk2iCyIyVySn5uVBQAGEhRz"}]}'

```

The following response is received (carriage returns added for clarity). The last two records are retrieved:

接收到以下响应（为了清晰起见，添加了回车）。检索最后两个记录：

```

[{"Key":"marble4",
"Record":{"color":"purple","docType":"marble","name":"marble4","owner":"tom","size":20}},
  {"Key":"marble5",
"Record":{"color":"blue","docType":"marble","name":"marble5","owner":"tom","size":40}}}
[{"ResponseMetadata":{"RecordsCount":"2",
  "Bookmark":"g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT81PzkzJBYqz5yYWJekmoKkOWD
SOSANIFk2iCyIyVySn5uVBQAGYhR1"}]}]

```

The final command is a peer command to call queryMarblesWithPagination with

a pageSize of 3 and with the bookmark from the previous query.

最后一个命令是一个对等命令，用于调用页面大小为 3 的 QueryMarbleWithPagination 和上一个查询中的书签。

Try it yourself

自己试试

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c
'{"Args":["queryMarblesWithPagination",
{"\"selector\":{\"docType\": \"marble\", \"owner\": \"tom\"},
\"use_index\": [\"_design/indexOwnerDoc\",
\"indexOwner\"]}", "3", "g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT81PzkzJBYqz5yYWJeWkmoKkOWDSOSANIFk2iCyIyVySn5uVBQAGYhR1"]}]'
```

The following response is received (carriage returns added for clarity). No records are returned, indicating that all pages have been retrieved:

接收到以下响应（为了清晰起见，添加了回车）。未返回任何记录，表明已检索到所有页：

```
[]
[{"ResponseMetadata":{"RecordsCount":"0",
"Bookmark":"g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT81PzkzJBYqz5yYWJeWkmoKkOWDSOSANIFk2iCyIyVySn5uVBQAGYhR1"}}]
```

For an example of how a client application can iterate over the result sets using pagination, search for the `getQueryResultForQueryStringWithPagination` function in the Marbles sample.

有关客户端应用程序如何使用分页在结果集上迭代的示例，请在 Marbles 示例中搜索 `getQueryResultForQueryStringWithPagination` 函数。

8、Update an Index

8、更新索引

It may be necessary to update an index over time. The same index may exist in subsequent versions of the chaincode that gets installed. In order for an index to be updated, the original index definition must have included the design document ddoc attribute and an index name. To update an index definition, use the same index name but alter the index definition. Simply edit the index JSON file and add or remove fields from the index. Fabric only supports the index type JSON, changing the index type is not supported. The updated index definition gets redeployed to the peer's state database when the chaincode is installed and instantiated. Changes to the index name or ddoc attributes will result in a new index being created and the original index remains unchanged in CouchDB until it is removed.

随着时间的推移，可能需要更新索引。在随后安装的 chaincode 版本中可能存在相同的索引。为了更新索引，原始索引定义必须包含设计文档 ddoc 属性和索引名称。要更新索引定义，请使用相同的索引名称，但要更改索引定义。只需编辑索引 JSON 文件并从索引中添加或删除字段。Fabric 只支持索引类型 json，不支持更改索引类型。当安装并实例化链码时，更新的索引定义会重新部署到对等端的状态数据库中。对索引名称或 ddoc 属性的更改

将导致创建新索引，并且在移除之前，couchdb 中的原始索引保持不变。

Note

注释

If the state database has a significant volume of data, it will take some time for the index to be re-built, during which time chaincode invokes that issue queries may fail or timeout.

如果状态数据库有大量的数据，则重新构建索引需要一些时间，在此期间，chaincode 调用该问题查询可能会失败或超时。

Iterating on your index definition

迭代索引定义

If you have access to your peer's CouchDB state database in a development environment, you can iteratively test various indexes in support of your chaincode queries. Any changes to chaincode though would require redeployment. Use the CouchDB Fauxton interface or a command line curl utility to create and update indexes.

如果您可以在开发环境中访问对等的 couchdb 状态数据库，那么可以迭代地测试各种索引，以支持链式代码查询。但是，对 chaincode 的任何更改都需要重新部署。使用 couchdb fauxton 接口或命令行 curl 实用程序创建和更新索引。

Note

注释

The Fauxton interface is a web UI for the creation, update, and deployment of indexes to CouchDB. If you want to try out this interface, there is an example of the format of the Fauxton version of the index in Marbles sample. If you have deployed the BYFN network with CouchDB, the Fauxton interface can be loaded by opening a browser and navigating to http://localhost:5984/_utils.

Fauxton 接口是用于创建、更新和部署到 CouchDB 的索引的 Web UI。如果您想尝试这个界面，这里有一个示例，说明了 Marbles 示例中索引的 Fauxton 版本的格式。如果您已经使用 couchdb 部署了 byfn 网络，那么可以通过打开浏览器并导航到 <http://localhost:5984/utils> 来加载 fauxton 接口。

Alternatively, if you prefer not use the Fauxton UI, the following is an example of a curl command which can be used to create the index on the database mychannel_marbles:

或者，如果您不喜欢使用 Fauxton UI，下面是一个 curl 命令的示例，该命令可用于在数据库 mychannel 大理石上创建索引：

// Index for docType, owner. // Example curl command line to define index in the CouchDB channel_chaincode database

```
curl -i -X POST -H "Content-Type: application/json" -d
"{\"index\":{\"fields\":[\"docType\",\"owner\"],
\"name\":\"indexOwner\",
\"ddoc\":\"indexOwnerDoc\",
\"type\":\"json\"}" http://hostname:port/mychannel_marbles/_index
```

Note

注释

If you are using BYFN configured with CouchDB, replace hostname:port with

localhost:5984.

如果使用配置了 couchdb 的 byfn, 请将 hostname:port 替换为 localhost:5984。

9、Delete an Index

9、删除索引

Index deletion is not managed by Fabric tooling. If you need to delete an index, manually issue a curl command against the database or delete it using the Fauxton interface.

索引删除不由结构工具管理。如果需要删除索引, 请对数据库手动发出 curl 命令, 或者使用 fauxton 接口将其删除。

The format of the curl command to delete an index would be:

删除索引的 curl 命令格式为:

```
curl -X DELETE
http://localhost:5984/{database_name}/_index/{design_doc}/json/{index_name} -H
"accept: */*" -H "Host: localhost:5984"
```

To delete the index used in this tutorial, the curl command would be:

要删除本教程中使用的索引, curl 命令将是:

```
curl -X DELETE
http://localhost:5984/mychannel_marbles/_index/indexOwnerDoc/json/indexOwner -H
"accept: */*" -H "Host: localhost:5984"
```

十二、Videos

十二、视频

Refer to the Hyperledger Fabric channel on YouTube

请参阅 YouTube 上的 Hyperledger Fabric 频道

https://youtu.be/ZgKAahU3FcM?list=PLfuKAwZlKV0_--JYykteXjKyq0GA9j_il&t=27

This collection contains developers demonstrating various v1 features and components such as: ledger, channels, gossip, SDK, chaincode, MSP, and more...

此集合包含演示各种 v1 功能和组件的开发人员, 如: 分类帐、频道、八卦、SDK、链码、MSP 等等...