

# 滴雨科技区块链技术指南-讲解教程

## 1 讲解

我们提供的教程可帮助您开始使用 Hyperledger Fabric。第一个面向 Hyperledger Fabric **应用程序开发人员**, 即“编写您的第一个应用程序”。它带领您完成使用 Hyperledger Fabric [Node SDK](#) 为 Hyperledger Fabric 编写第一个区块链应用程序的过程。

第二本教程面向 Hyperledger Fabric 网络运营商，“构建您的第一个网络”。本教程将引导您完成使用 Hyperledger Fabric 建立区块链网络的过程，并提供一个基本的示例应用程序对其进行测试。

还有一些教程，用于更新您的频道，[向频道添加组织](#)，以及将网络升级到 Hyperledger Fabric 的更高版本，[升级网络组件](#)。

最后，我们提供了两个 chaincode 教程。一种面向开发人员，即[面向开发人员的 Chaincode](#)，另一种面向运营商，即[针对运营商的 Chaincode](#)。

### 注意

如果您有本文档未解决的问题，或在任何教程中遇到问题，请访问“[还有问题？](#)”。页面上有关在哪里可以找到其他帮助的一些提示。

### 讲解

- [编写您的第一个应用程序](#)
- [商业票据教程](#)
- [建立您的第一个网络](#)
- [将组织添加到频道](#)
- [升级网络组件](#)
- [在结构中使用私有数据](#)
- [Chaincode 教程](#)
- [开发人员链码](#)
- [运营商链码](#)
- [使用 CouchDB](#)
- [影片](#)

## 2 编写您的第一个应用程序

### 注意

如果您还不熟悉 Fabric 网络的基本架构，则可能需要先访问“[关键概念](#)”部分，然后再继续。

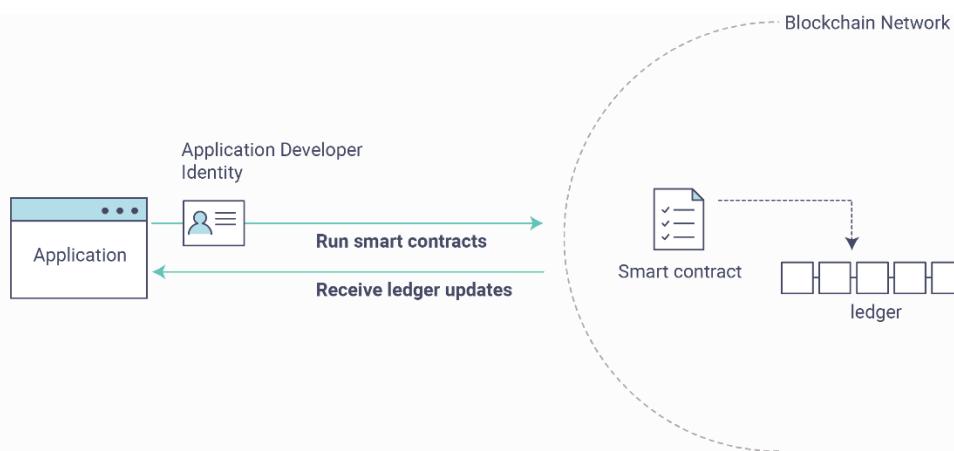
还值得注意的是，本教程是 Fabric 应用程序的入门，并使用简单的智能合约和应用程序。要更深入地了解 Fabric 应用程序和智能合约，请查看我们的“[开发应用程序](#)”部分或“[商业论文](#)”教程。

在本教程中，我们将查看一些示例程序，以了解 Fabric 应用程序如何工作。这些应用程序及其使用的智能合约统称为 [FabCar](#)。它们为了解 Hyperledger Fabric 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以查询和更新分类帐，以及如何使用证书颁发机构生成 X.509 证书，以供与许可区块链进行交互的应用程序使用。

我们将使用应用程序 SDK（在[应用程序](#)主题中进行了详细介绍）来调用智能合约，该智能合约使用智能合约 SDK 来查询和更新分类帐（在[智能合约处理](#)部分中进行了详细介绍）。

我们将经历三个主要步骤：

- 1. 设置开发环境。**我们的应用程序需要与之交互的网络，因此我们将获得一个智能合约和应用程序将使用的基本网络。



- 2. 了解样本智能合约 FabCar。**我们将检查智能合约以了解其中的交易以及应用程序如何使用它们查询和更新分类账。

**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 智能合约来查询和更新分类账上的汽车资产。我们将深入研究应用程序及其创建的交易的代码，包括查询汽车，查询一系列汽车以及创建新汽车。

完成本教程后，您应该对如何结合智能合约对应用程序进行编程进行基本了解，以与在 Fabric 网络中的对等节点上托管和复制的分类帐进行交互。

### 注意

这些应用程序还与 Service Discovery 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。

## 2.1 建立区块链网络

### 注意

下一部分将要求您 [first-network](#) 位于存储库本地克隆中的子目录中 [fabric-samples](#)。

如果您已经完成了构建第一个网络，则将下载 [fabric-samples](#) 并建立并运行一个网络。在运行本教程之前，必须停止该网络：

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

如果您之前已经阅读过本教程，请使用以下命令杀死任何陈旧或活动的容器。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}')
```

如果您没有开发环境以及网络和应用程序随附的工件，请访问“[先决条件](#)”页面，并确保在计算机上安装了必要的依赖项。

接下来，如果您尚未这样做，请访问“[安装示例，二进制文件和 Docker 映像](#)”页面，并按照提供的说明进行操作。克隆 [fabric-samples](#) 存储库并下载了最新的稳定 Fabric 映像和可用实用程序后，请返回本教程。

如果您使用 Mac OS 并运行 Mojave，则需要[安装 Xcode](#)。

## 2.1.1 启动网络

### 注意

下一部分将要求您 [fabcar](#) 位于存储库本地克隆中的子目录中 [fabric-samples](#)。

本教程演示了 [FabCar](#) 智能合约和应用程序的 JavaScript 版本，但该 [fabric-samples](#) 回购还包含此示例的 Java 和 TypeScript 版本。要尝试 Java 或 TypeScript 版本，请将下面的 [javascript](#) 参数更改为或，然后按照写入终端的说明进行操作。[./startFabric.shjavatypescript](#)

使用 [startFabric.sh](#) Shell 脚本启动网络。该命令将启动一个由对等方，排序者，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，供我们的应用程序用来访问分类帐。在学习本教程时，我们将了解有关这些组件的更多信息。

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

好了，您现在已经建立并运行了一个示例网络，并且 [FabCar](#) 已安装并实例化了智能合约。让我们先安装应用程序先决条件，以便我们可以尝试一下，并了解所有功能如何协同工作。

## 2.1.2 安装应用程序

### 注意

以下说明要求您 [fabcar/javascript](#) 位于存储库本地克隆中的子目录中 [fabric-samples](#)。

运行以下命令以安装应用程序的结构依赖关系。大约需要一分钟才能完成：

```
npm install
```

此过程将安装中定义的关键应用程序依赖项 [package.json](#)。其中最重要的是 [fabric-network](#) 课堂；它使应用程序能够使用身份，钱包和网关来连接到渠道，提交交易并等待通知。本教程还使用 [fabric-ca-client](#) 该类向具有各自证书颁发机构的用户注册，生成有效的身份，然后由 [fabric-network](#) 类方法使用。

一旦完成，一切就绪运行应用程序。对于本教程，您将主要使用目录中的应用程序 JavaScript 文件。让我们看看里面是什么：[npm installfabcar/javascript](#)

```
ls
```

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
```

在目录中有其他程序语言的 [fabcar/typescript](#) 文件。使用 JavaScript 示例后，您可以阅读这些内容-原理相同。

如果您使用 Mac OS 并运行 Mojave，则需要[安装 Xcode](#)。

## 2.2 注册管理员用户

## 注意

以下两节涉及与证书颁发机构的通信。您可能会发现，通过打开新的终端外壳并运行，在运行即将到来的程序时，流式传输 CA 日志很有用。`docker logs -f ca.example.com`

当我们创建网络时，创建了一个管理员用户（直称为 `admin`）作为证书颁发机构（CA）的注册商。我们的第一步是生成 `admin` 使用该 `enroll.js` 程序的私钥，公钥和 X.509 证书。此过程使用证书签名请求（CSR）-首先在本地生成私钥和公钥，然后将公钥发送到 CA，CA 返回编码的证书以供应用程序使用。然后将这三个凭证存储在钱包中，使我们能够充当 CA 的管理员。

随后，我们将注册并注册一个新的应用程序用户，我们的应用程序将使用该用户与区块链进行交互。

让我们注册用户 `admin`：

```
node enrollAdmin.js
```

此命令已将 CA 管理员的凭据存储在 `wallet` 目录中。

## 2.3 注册并报名 `user1`

现在，我们已经在钱包中拥有管理员的凭据，我们可以注册一个新用户—`user1`—将用于查询和更新分类帐：

```
node registerUser.js
```

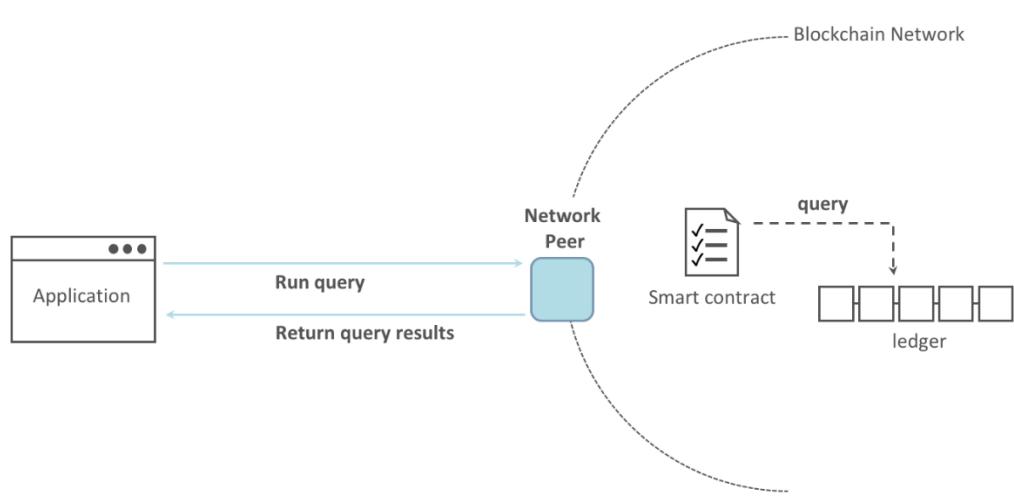
与管理员注册类似，此程序使用 CSR 来注册 `user1` 和存储其凭据以及 `admin` 钱包中的凭据。现在，我们有两个独立用户的身份，`admin` 并且 `user1`，并且我们的应用程序使用了这些身份。

是时候与分类帐进行交互了……

## 2.4 查询分类帐

区块链网络中的每个对等方都托管分类账的副本，应用程序可以通过调用智能合约来查询分类账，该合约查询分类账的最新值并将其返回给应用程序。

这是查询工作方式的简化表示：



应用程序使用查询从分类帐读取数据。最常见的查询涉及分类帐中数据的当前值-其世界状态。世界状态表示为一组键值对，应用程序可以查询数据以获取单个键或多个键。此外，分类帐世界状态可以配置为使用像 CouchDB 这样的数据库，当将键值建模为 JSON 数据时，该数据库支持复杂的查询。当寻找与某些关键字匹配特定值的所有资产时，这将非常有用。例如，所有具有特定所有者的汽车。

首先，让我们运行 `query.js` 程序以返回分类账中所有汽车的清单。该程序使用我们的第二个身份—`user1`—访问分类帐：

```
node query.js
```

输出应如下所示：

```
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"}}]
```

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`。

该应用程序首先从 `fabric-network` 模块中引入作用域两个关键类。`FileSystemWallet` 和 `Gateway`。这些类将用于 `user1` 在钱包中定位身份，并使用其连接到网络：

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

该应用程序使用网关连接到网络：

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

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` 介绍了网络网关将与身份访问 `user1` 的

`wallet`。查看如何 `ccp` 从 `./../basic-network/connection.json` JSON 文件加载和解析 JSON 文件：

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

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](#)。

一个网络可以分为多个通道，下一行重要的代码行将应用程序连接到网络中的特定通道 `mychannel`：

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

在此通道中，我们可以访问智能合约 `fabcar` 以与分类账进行交互：

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

在 `fabcar` 其中有许多不同的事务，我们的应用程序最初使用该 `queryAllCars` 事务来访问分类帐世界状态数据：

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

该 `evaluateTransaction` 方法代表了与区块链网络中智能合约最简单的交互之一。它只是选择一个在连接配置文件中定义的对等体，然后将请求发送到该对等体，并在此对其进行评估。智能合约查询对等方的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。

## 2.5 FabCar 智能合约

让我们看一下 `FabCar` 智能合约中的交易。导航到 `chaincode/fabcar/javascript/lib` 根目录下的子目录，`fabric-samples` 然后 `fabcar.js` 在编辑器中打开。

查看如何使用 `Contract` 类别定义智能合约：

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

在这一类结构，你会看到，我们有以下交易定义：`initLedger`，`queryCar`，`queryAllCars`，`createCar`，和 `changeCarOwner`。例如：

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

让我们仔细看一下 `queryAllCars` 交易，看看它如何与分类账交互。

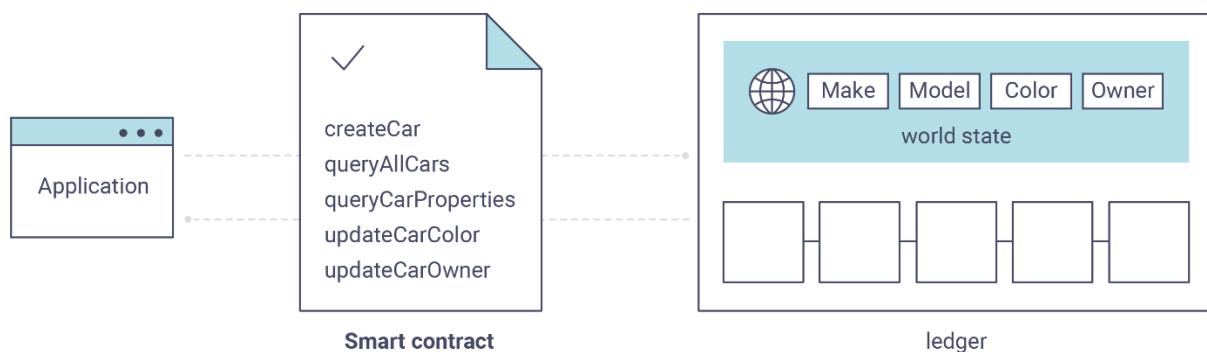
```
async queryAllCars(ctx) {
```

```
    const startKey = 'CAR0';
    const endKey = 'CAR999';

    const iterator = await ctx.stub.getStateByRange(startKey, endKey);
```

该代码定义了 `queryAllCars` 将从分类帐中检索的汽车范围。每节车厢之间 `CAR0` 和 `CAR999`- 1000 辆汽车所有，假设每个密钥已经被正确标记-将查询返回。其余代码遍历查询结果，并将其打包为应用程序的 JSON。

下面是应用程序如何调用智能合约中的不同交易的图示。每个交易使用广泛的 API，例如 `getStateByRange` 与分类帐进行交互。您可以详细了解有关这些 API 的更多[信息](#)。



我们可以看到我们的 `queryAllCars` 交易，另一个称为 `createCar`。我们将在本教程的后面部分中使用它来更新分类帐，并向区块链添加一个新块。

但首先，请返回该 `query` 程序并将 `evaluateTransaction` 请求更改为 query `CAR4`。该 `query` 程序现在应如下所示：

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

保存该程序并导航回到您的 `fabcar/javascript` 目录。现在 `query` 再次运行程序：

```
node query.js
```

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"}
```

如果您回头查看交易发生的时间 `queryAllCars`，您会发现这 `CAR4` 是 Adriana 的黑色 Tesla 模型 S，这是在此处返回的结果。

我们可以使用 `queryCar` 交易查询任何汽车，使用其键（例如 `CAR0`）来获取与该汽车相对应的任何品牌，型号，颜色和所有者。

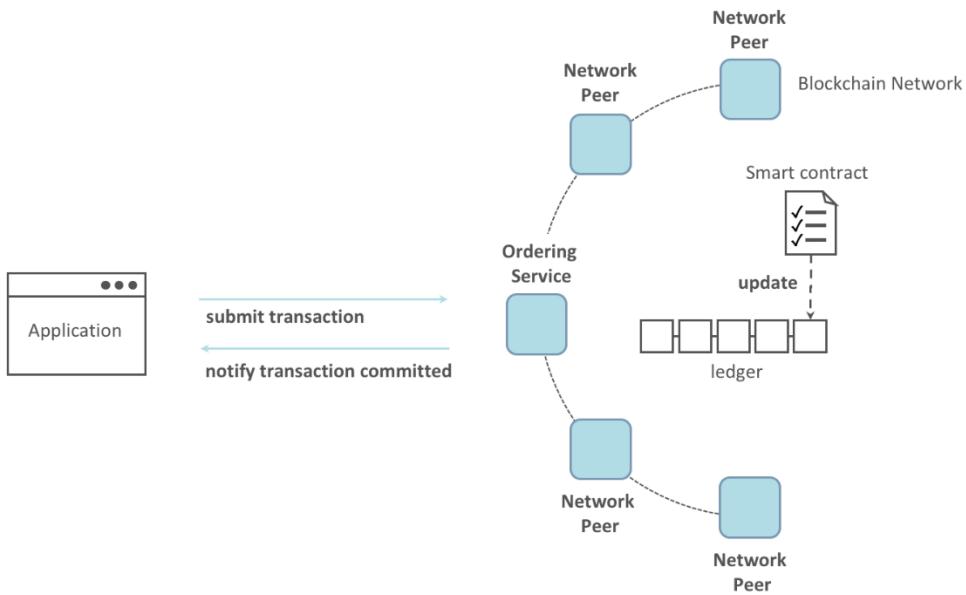
此时，您应该熟悉智能合约中的基本查询事务和查询程序中的少数参数。

是时候更新分类帐了.....

## 2.6 更新分类帐

现在，我们已经完成了一些分类帐查询并添加了一些代码，我们已经准备好更新分类帐。有很多我们可以让潜在的更新，但是让我们通过创建一个启动新汽车。

从应用程序的角度来看，更新分类帐很简单。应用程序将交易提交到区块链网络，并且在验证和提交交易后，应用程序会收到有关交易成功的通知。在幕后，这涉及共识过程，通过该过程，区块链网络的不同组件将共同努力，以确保对账本的每个提议更新均有效并以一致且一致的顺序执行。



在上方，您可以查看使此过程正常运行的主要组件。除了每个承载一个分类帐的副本以及可选的智能合约副本的多个对等点之外，网络还包含一个排序服务。排序服务协调网络的交易；它以明确定义的顺序创建包含交易的块，这些交易源自连接到网络的所有不同应用程序。

我们对分类帐的第一次更新将创建一辆新车。我们有一个单独的程序，称为 `invoke.js` 我们将用来更新分类帐的程序。与查询一样，使用编辑器打开程序并导航到代码块，我们在其中构建事务并将其提交给网络：

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

了解应用程序如何调用智能合约交易 `createCar` 来创建一个名为 Tom 的所有者的黑色 Honda Accord。我们 `CAR12` 在这里使用作为识别键，只是为了表明我们不需要使用顺序键。

保存并运行程序：

```
node invoke.js
```

如果调用成功，您将看到以下输出：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
2018-12-11T14:11:40.935Z - info: [TransactionEventHandler]: _strategySuccess: strategy success
for transaction "9076cd4279a71ecf99665aed0ed3590a25bba040fa6b4dd6d010f42bb26ff5d1"
Transaction has been submitted
```

请注意，`invoke` 应用程序如何使用 `submitTransaction` API 而非来与区块链网络进行交互 `evaluateTransaction`。

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

`submitTransaction` 比 `evaluateTransaction` 复杂得多。SDK 不会与单个对等方进行交互，而是将 `submitTransaction` 提案发送到区块链网络中每个所需组织的对等方。这些对等方中的每一个都将使用该提议执行所请求的智能合约，以生成交易响应，并签名并返回给 SDK。SDK 将所有已签名的交易响应收集到一个交易中，然后将其发送给排序者。排序者将来自每个应用程序的交易收集并排序成一个交易块。然后，将这些块分配给网络中的每个对等点，在此对每个事务进行验证和提交。最后，通知 SDK，使其可以将控制权返回给应用程序。

注意

`submitTransaction` 还包括一个侦听器，用于检查以确保交易已通过验证并已提交到分类账。应用程序应利用提交侦听器，或利用类似 API 的 API `submitTransaction`。否则，您的交易可能无法成功排序，验证并提交到分类账。

`submitTransaction` 为应用程序完成所有这些工作！应用程序，智能合约，对等方和排序服务一起工作以保持分类帐在整个网络中一致的过程称为共识，本节对此进行了详细说明。

要查看此事务已被写入分类帐，请返回 `query.js` 并将参数从更改 `CAR4` 为 `CAR12`。

换句话说，更改此：

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

对此：

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

再次保存，然后查询：

```
node query.js
```

该返回以下内容：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
{"colour":"Black", "make":"Honda", "model":"Accord", "owner": "Tom"}
```

恭喜你您已经创建了一辆汽车，并验证了其记录在分类帐中！

因此，既然我们已经做到了，那么就说汤姆很慷慨，他想将他的本田雅阁送给一个叫戴夫的人。

要做到这一点，回去 `invoke.js` 和更改智能合约交易 `createCar`，以 `changeCarOwner` 在输入参数的相应变化：

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

第一个参数- `CAR12` 标识将要更改所有者的汽车。第二个参数- `Dave` 定义了汽车的新主人。

保存并再次执行程序：

```
node invoke.js
```

现在，让我们再次查询分类帐，并确保 Dave 现在已与 `CAR12` 密钥关联：

```
node query.js
```

它应该返回以下结果：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
{"colour": "Black", "make": "Honda", "model": "Accord", "owner": "Dave"}
```

的所有权 `CAR12` 已从汤姆 (Tom) 更改为戴夫 (Dave)。

注意

在实际应用中，智能合约可能会具有一些访问控制逻辑。例如，仅某些授权用户可以创建新车，而只有车主可以将车转让给其他人。

## 2.7 摘要

现在我们已经完成了一些查询和一些更新，您应该对应用程序如何使用智能合约查询或更新分类帐与区块链网络进行交互有了很好的了解。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本知识，并且应该对如何使用不同类型的应用程序执行其他业务任务和操作有所了解。

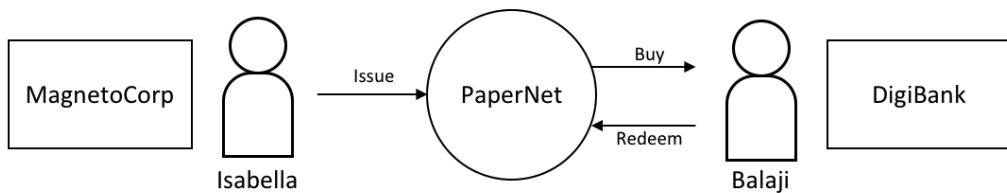
## 2.8 额外资源

正如我们在简介中所述，我们有一整节关于“[开发应用程序](#)”的内容，其中包括有关智能合约，流程和数据设计的深入信息，使用更深入的《商业论文》[教程](#)以及与之相关的大量其他材料。[应用程序的开发](#)。

# 3 商业票据教程

**受众：**架构师，应用程序和智能合约开发人员，管理员

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。这是一个面向任务的主题，因此强调了以上概念中的过程。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。



在本教程中，*MagnetoCorp* 和 *DigiBank* 这两个组织使用 *Hyperledger Fabric* 区块链网络 *PaperNet* 彼此进行商业票据交易。

建立基本网络后，您将扮演 *MagnetoCorp* 的员工 *Isabella*，他将代表该公司发行商业票据。然后，您将转而担任 *DigiBank* 员工 *Balaji* 的角色，他将购买此商业票据，保留一段时间，然后与 *MagnetoCorp* 赎回以获取少量利润。

您将分别在不同的组织中充当开发人员，最终用户和管理员的角色，执行以下步骤旨在帮助您了解作为两个不同的组织独立工作，但要根据 *Hyperledger Fabric* 中共同商定的规则进行协作的感觉。网络。

- [设置机器并下载样本](#)
- [创建一个网络](#)
- [了解智能合约的结构](#)
- 作为组织 *MagnetoCorp* 来 [安装和实例化智能合约](#)
- 了解 *MagnetoCorp* [应用程序](#)的结构，包括其 [依赖项](#)
- 配置和使用[钱包和身份](#)
- 运行 *MagnetoCorp* 应用程序以[发行商业票据](#)
- 了解第二家机构 *Digibank* 如何在其[应用程序](#)中使用智能合约
- 作为 *Digibank*，[运行购买和赎回商业票据的应用程序](#)

本教程已经在 Mac OS 和 Ubuntu 上进行了测试，并且可以在其他 Linux 发行版上使用。Windows 版本正在开发中。

## 3.1 先决条件

在开始之前，您必须安装本教程所需的一些必备技术。我们将这些限制降至最低，以便您可以快速上手。

**您必须安装以下技术：**

- [节点](#)版本 8.9.0 或更高版本。Node 是一个 JavaScript 运行时，可用于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 18.06 版或更高版本。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 Fabric 是作为一组 Docker 映像提供的, PaperNet 智能合约将在 Docker 容器中运行。[在此处](#)安装 Docker。

您会发现安装以下技术会有所帮助:

- 源代码编辑器, 例如 Visual Studio Code 1.28 版或更高版本。VS Code 将帮助您开发和测试您的应用程序和智能合约。[在此处](#)安装 VS Code。

许多出色的代码编辑器都可用, 包括 Atom, Sublime Text 和 Brackets。

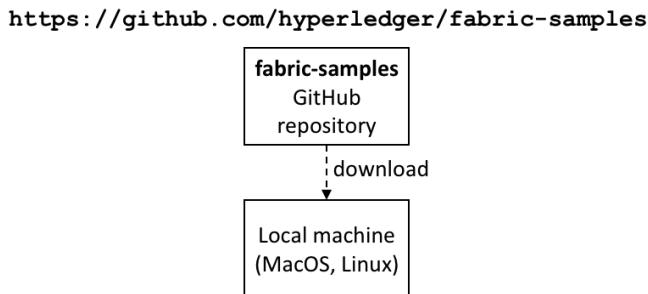
你可能会发现, 你变得更有经验, 与应用程序和智能合同发展是很有帮助的安装以下的技术。初次运行本教程时, 无需安装它们:

- 节点版本管理器。NVM 可帮助您轻松地在不同版本的节点之间切换—如果您同时处理多个项目, 这将非常有用。[在此处](#)安装 NVM。

## 3.2 下载样本

商业论文教程是在名为的公共 GitHub 存储库中保存的 Hyperledger Fabric [示例](#)之一。当您要在计算机上运行教程时, 首要任务是下载存储库。

[fabric-samples](https://github.com/hyperledger/fabric-samples)



将 [fabric-samples](#) GitHub 存储库下载到本地计算机。

`$GOPATH` 是 Hyperledger Fabric 中的重要环境变量; 它标识要安装的根目录。无论您使用哪种编程语言, 正确安装都非常重要! 打开一个新的终端窗口, 并 `$GOPATH` 使用以下 `env` 命令检查您的设置:

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

如果未设置, 请使用以下说明 `$GOPATH`。

现在, 您可以创建相对于目录 `$GOPATH`, 其中 [fabric-samples](#) 将被安装:

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

使用命令将存储库复制到此位置: `git clone fabric-samples`

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

随时检查以下内容的目录结构 [fabric-samples](#):

```
$ 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		

注意 `commercial-paper` 目录-这就是我们的样本所在的目录！

您现在已经完成了教程的第一阶段！在继续过程中，您将打开为不同用户和组件打开的多个命令窗口。例如：

- 代表 Isabella 和 Balaji 运行应用程序，他们将彼此进行商业票据交易
- 代表 MagnetoCorp 和 DigiBank 的管理员发出命令，包括安装和实例化智能合约
- 显示对等，排序者和 CA 日志输出

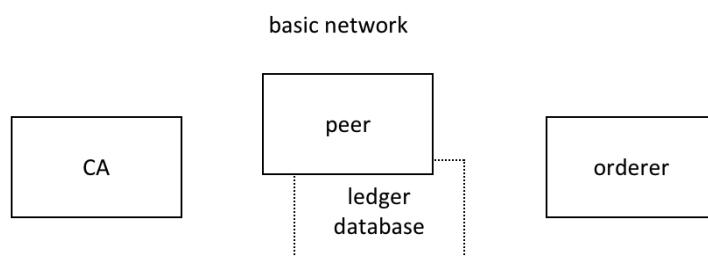
我们将在您应该从特定命令窗口运行命令时明确指出。例如：

(isabella)\$ ls

表示您应该 `ls` 从 Isabella 的窗口中运行命令。

### 3.3 建立网络

本教程当前使用基本网络。它将很快更新为更好地反映 PaperNet 的多组织结构的配置。目前，该网络足以向您展示如何开发应用程序和智能合约。



*Hyperledger Fabric* 基本网络包括一个对等方及其账本数据库，一个排序者和一个证书颁发机构 (CA)。这些组件中的每一个都作为 docker 容器运行。

对等方，其分类帐，排序者和 CA 均在各自的 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。他们不是专用于结构网络。

您可以使用 `fabric-samples\basic-network` 目录中包含的命令和配置来管理基本网络。让我们使用 `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.com couchdb  
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 connections initialized
2018-11-07 13:47:31.730 UTC [channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to join channel

```

注意命令如何从 [DockerHub](#) 中提取四个 Hyperledger Fabric 容器映像，然后启动它们。这些容器具有适用于这些 Hyperledger Fabric 组件的软件的最新版本。随意浏览目录-在本教程中，我们将使用其许多内容。

`docker-compose -f docker-compose.yml up -d ca.example.com...basic-network`

您可以使用以下命令列出运行基本网络组件的 Docker 容器：

`$ docker ps`

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
ada3d078989b	hyperledger/fabric-peer	"peer node start"	About a minute ago	Up
About a minute	0.0.0.0:7051->7051/tcp, 0.0.0.0:7053->7053/tcp	peer0.org1.example.com		
1fa1fd107bfb	hyperledger/fabric-orderer	"orderer"	About a minute ago	Up
About a minute	0.0.0.0:7050->7050/tcp	orderer.example.com		
53fe614274f7	hyperledger/fabric-couchdb	"tini -- /docker-ent..."	About a minute ago	Up
About a minute	4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp	couchdb		
469201085a20	hyperledger/fabric-ca	"sh -c 'fabric-ca-se..."	About a minute ago	Up
About a minute	0.0.0.0:7054->7054/tcp	ca.example.com		

查看是否可以将这些容器映射到基本网络（您可能需要水平滚动以查找信息）：

- 对等节点 `peer0.org1.example.com` 正在容器中运行 `ada3d078989b`
- 排序者 `orderer.example.com` 正在容器中运行 `1fa1fd107bfb`
- 一个 CouchDB 数据库 `couchdb` 正在容器中运行 `53fe614274f7`
- CA `ca.example.com` 正在容器中运行 `469201085a20`

这些容器都构成一个名为的 `docker` 网络 `net_basic`。您可以使用以下命令查看网络：

`$ docker network inspect net_basic`

```
{
  "Name": "net_basic",
  "Id": "62e9d37d00a0eda6c6301a76022c695f8e01258edaba6f65e876166164466ee5",
  "Created": "2018-11-07T13:46:30.4992927Z",
  "Containers": {
    "1fa1fd107bfbbe61522e4a26a57c2178d82b2918d5d423e7ee626c79b8a233624": {
      "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": {}
}
```

查看这四个容器如何成为一个 `docker` 网络的一部分，如何使用不同的 IP 地址。（为了清楚起见，我们将输出缩

写。)

回顾一下：您已经从 GitHub 下载了 Hyperledger Fabric 示例存储库，并且基本网络已在本地计算机上运行。现在让我们开始扮演希望交易商业票据的 MagnetoCorp 的角色。

## 3.4 担任 MagnetoCorp

要监视 PaperNet 的 MagnetoCorp 组件，管理员可以使用 `logspout` 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位置，从而可以轻松地从单个窗口查看正在发生的情况。例如，这对于安装智能合约的管理员或调用智能合约的开发人员确实很有帮助。

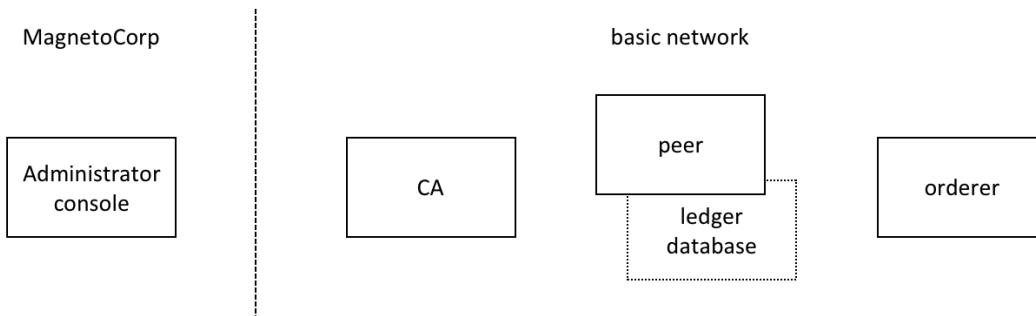
现在，让我们以 MagnetoCorp 管理员的身份监视 PaperNet。在 `fabric-samples` 目录中打开一个新窗口，找到并运行 `monitordocker.sh` 脚本以启动 `logspout` 用于与 docker 网络关联的 PaperNet docker 容器的工具 `net_basic`：

```
(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
```

请注意，如果默认端口 `monitordocker.sh` 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>
```

现在，该窗口将显示 docker 容器的输出，因此让我们启动另一个终端窗口，该窗口将允许 MagnetoCorp 管理员与网络进行交互。



MagnetoCorp 管理员通过 Docker 容器与网络进行交互。

为了与 PaperNet 进行交互，MagnetoCorp 管理员需要使用 Hyperledger Fabric `peer` 命令。方便地，这些都可以在 `hyperledger/fabric-tools` docker image 中预先构建。

让我们使用以下 `docker-compose` 命令为管理员启动一个特定于 MagnetoCorp 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
```

再次，查看如何 `hyperledger/fabric-tools` 从 Docker Hub 检索 Docker 映像并将其添加到网络：

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

CONTAINER ID	IMAGE	COMMAND NAMES	CREATED	STATUS

562a88b25149	hyperledger/fabric-tools	"/bin/bash"	About a minute ago	Up
About a minute			cliMagnetoCorp	
b7f3586e5d02	gliderlabs/logspout	"/bin/logspout"	7 minutes ago	Up 7
minutes	127.0.0.1:8000->80/tcp	logspout		
ada3d078989b	hyperledger/fabric-peer	"peer node start"	29 minutes ago	Up 29
minutes	0.0.0.0:7051->7051/tcp, 0.0.0.0:7053->7053/tcp	peer0.org1.example.com		
1fa1fd107fb	hyperledger/fabric-orderer	"orderer"	29 minutes ago	Up 29
minutes	0.0.0.0:7050->7050/tcp	orderer.example.com		
53fe614274f7	hyperledger/fabric-couchdb	"tini -- /docker-ent..."	29 minutes ago	Up 29
minutes	4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp	couchdb		
469201085a20	hyperledger/fabric-ca	"sh -c 'fabric-ca-se..."	29 minutes ago	Up 29
minutes	0.0.0.0:7054->7054/tcp	ca.example.com		

MagnetoCorp 管理员将使用容器中的命令行 `562a88b25149` 与 PaperNet 进行交互。还要注意 `logspout` 容器 `b7f3586e5d02`；这是捕获 `monitordocker.sh` 命令的所有其他 docker 容器的输出。

现在，让我们使用此命令行以 MagnetoCorp 管理员的身份与 PaperNet 进行交互。

## 3.5 智能合约

`issue`, `buy` 并且 `redeem` 是在 PaperNet 智能合同心脏的三个功能。应用程序使用它来提交交易，这些交易相应地在分类账上发行，购买和赎回商业票据。我们的下一个任务是检查智能合约。

打开一个新的终端窗口，代表 MagnetoCorp 开发人员，并切换到包含 MagnetoCorp 智能合约副本的目录，以便使用您选择的编辑器进行查看（本教程中的 VS Code）：

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

在 `lib` 文件夹目录中，您将看到 `papercontract.js` 文件-其中包含商业票据智能合约！

```

papercontract.js — contract
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
35         super('org.paperNet.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 necessary
52         console.log('Instantiate the contract');
53     }
54
55     /**
56      * Issue commercial paper
57

```

一个示例

代码编辑器，在其中显示商业票据智能合约 `papercontract.js`

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

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

该语句将两个关键的 Hyperledger Fabric 类纳入了范围，它们将被智能合约广泛使用- `Contract` 和 `Context`。您可以在 [fabric-shim JSDOCS](#) 中了解有关这些类的更多信息。

- `class CommercialPaperContract extends Contract {`

这 `CommercialPaperContract` 基于内置的 Fabric `Contract` 类定义了智能合约类。其落实的关键交易的方法 `issue`，`buy` 和 `redeem` 商业票据是这个类中定义。

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

此方法 `issue` 为 PaperNet 定义了商业票据交易。传递给此方法的参数将用于创建新的商业票据。  
找到并检查智能合约中的 `buy` 和 `redeem` 交易。

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

在 `issue` 交易中，此语句使用 `CommercialPaper` 带有提供的交易输入的类在内存中创建新的商业票据。检查 `buy` 和 `redeem` 事务，以了解它们如何类似地使用此类。

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

该语句使用 `ctx.paperList`，将新的商业票据添加到分类帐中，`PaperList` 该类是在 `CommercialPaperContext` 初始化智能合约上下文时创建的类的实例。再次，检查 `buy` 和 `redeem` 方法，以了解它们如何使用此类。

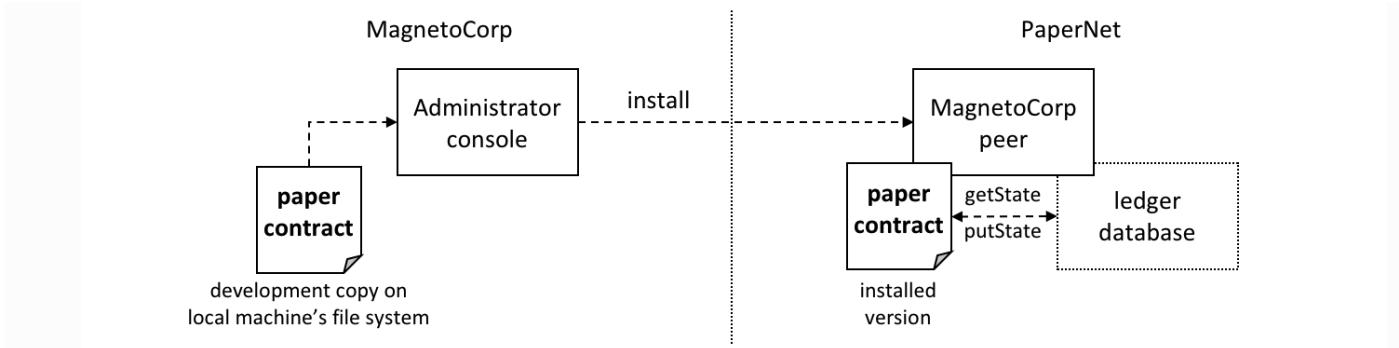
- `return paper.toBuffer();`

该语句返回一个二进制缓冲区作为 `issue` 事务的响应，由智能合约的调用方进行处理。

随时检查目录中的其他文件 `contract` 以了解智能合约的工作原理，并详细阅读 [papercontract.js](#) 智能合约主题中的设计方式。

## 3.6 安装合约

之前 `papercontract` 可通过应用程序所调用，它必须被安装到在 PaperNet 适当的对等节点。MagnetoCorp 和 DigiBank 管理员能够安装 `papercontract` 到他们各自具有权限的同级上。



*MagnetoCorp 管理员将的副本安装 `papercontract` 到 MagnetoCorp 对等方。*

智能合约是应用程序开发的重点，并且包含在称为 `chaincode` 的 Hyperledger Fabric 工件中。可以在单个链码中定义一个或多个智能合约，安装链码将允许 PaperNet 中的不同组织使用它们。这意味着只有管理员才需要担心链码。其他人都可以根据智能合约进行思考。

MagnetoCorp 管理员使用该命令将智能合约从其本地计算机的文件系统复制到目标对等方的 Docker 容器内的文件系统。一旦将智能合约安装在对等方上并在通道上实例化，便可以由应用程序调用，并通过 `putState()` 和 `getState()` Fabric API 与分类帐数据库进行交互。检查内的类 如何 使用这些 API 。

```
peer chaincode install papercontract papercontractStateList ledger-api\statelist.js
```

现在 `papercontract`，以 MagnetoCorp 管理员身份进行安装。在 MagnetoCorp 管理员的命令窗口中，使用以下命令在容器中运行命令：`docker exec peer chaincode install`

```
(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" >
```

所述 `cliMagnetoCorp` 容器具有设置 `CORE_PEER_ADDRESS=peer0.org1.example.com:7051` 到目标其命令 `peer0.org1.example.com`，并且指示 已经在此对等体已成功安装。目前，MagnetoCorp 管理员只需在单个 MagnetoCorp 对等方上安装的副本。`INFO 003 Installed remotely...papercontractpapercontract`

注意命令如何指定相对于容器文件系统的智能合约路径：。该路径已通过 文件映射到本地文件系统路径：

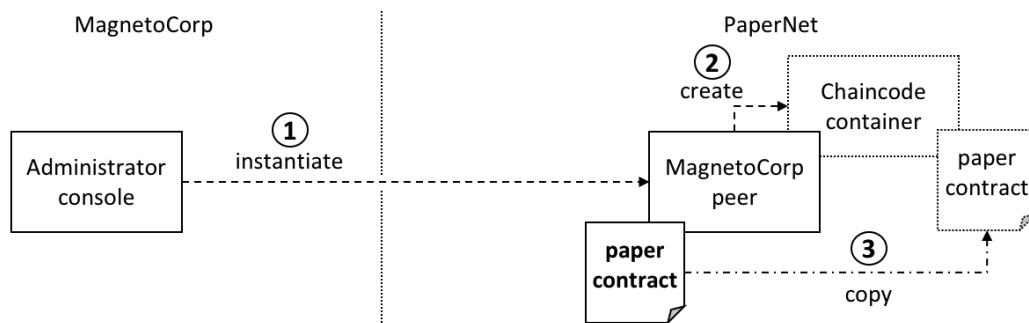
```
peer chaincode install-  
pcliMagnetoCorp/opt/gopath/src/github.com/contract.../organization/magnetocorp/contractmagnetocorp/  
configuration/cli/docker-compose.yml  
volumes:  
- ...  
- ./.../.../.../organization/magnetocorp:/opt/gopath/src/github.com/  
- ...
```

查看该 `volume` 指令如何映射 `organization/magnetocorp` 到 `/opt/gopath/src/github.com/` 提供此容器访问本地文件系统的位置，本地文件系统中保存了 MagnetoCorp 的 `papercontract` 智能合约副本。

您可以在此处了解更多信息并在此处 命令。`docker compose` `peer chaincode install`

## 3.7 实例化合同

现在，`papercontract` 包含 `CommercialPaper` 智能合约的链码已安装在所需的 PaperNet 对等方上，管理员可以将其提供给不同的网络通道使用，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，所以仅 `papercontract` 在单个网络通道中提供 `mychannel`。



`MagnetoCorp` 管理员实例化 `papercontract` 包含智能合约的链码。将创建一个新的 `docker chaincode` 容器以运行 `papercontract`。

该 `MagnetoCorp` 管理员使用命令来实例上：`peer chaincode instantiate papercontract mychannel`

```
(magnetocorp admin)$ docker exec cliMagnetoCorp peer chaincode instantiate -n papercontract -v 0 -l node -c '{"Args":["org.papernet.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

此命令可能需要几分钟才能完成。

上最重要的参数之一 `instantiate` 是 `-P`。它规定了 代言策略的 `papercontract`，描述了一套，才可以判断为有效必须认可（执行和符号）交易组织。所有交易（无论有效还是无效）都将记录在分类账区块链上，但是只有有效交易才能更新世界状态。

在传递过程中，查看 `instantiate` 排序者地址如何传递 `orderer.example.com:7050`。这是因为它另外将实例化事务提交给排序者，该事务将在下一个块中包括该事务，并将该事务分发给已加入的所有对等方 `mychannel`，从而使任何对等方都可以在其自己的隔离链代码容器中执行链代码。请注意，`instantiate` 对于 `papercontract` 通常运行的通道，只需要对其发出一次即可，即使通常已将其安装在许多对等端上也是如此。

查看如何 `papercontract` 使用以下命令启动容器： `docker ps`

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

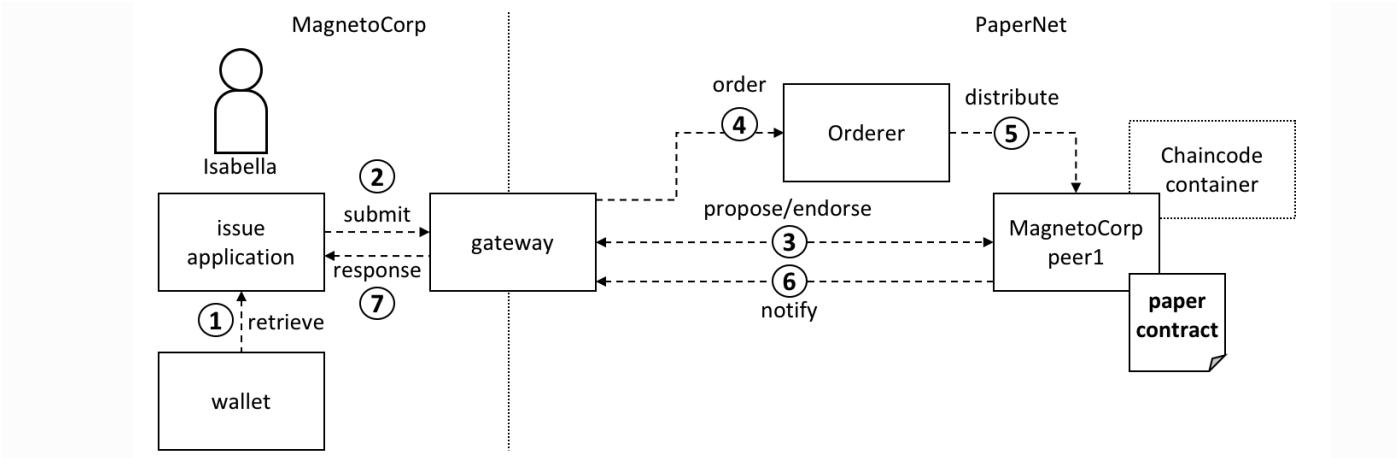
CONTAINER ID	IMAGE	COMMAND	CREATED
4fac1b91bfda	dev-peer0.org1.example.com-papercontract-0-d96...	/bin/sh -c 'cd /usr...'" 2	
minutes ago	Up 2 minutes	dev-peer0.org1.example.com-papercontract-0	

请注意，该容器已命名，`dev-peer0.org1.example.com-papercontract-0-d96...` 以指示哪个对等方启动了该容器，以及该容器正在运行 `papercontract` version 0。

现在我们已经建立并运行了一个基本的 PaperNet，并 `papercontract` 已安装并实例化了它，现在让我们将注意力转向发布商业论文的 MagnetoCorp 应用程序。

## 3.8 应用结构

`papercontract` MagnetoCorp 的应用程序调用其中包含的智能合约 `issue.js`。伊莎贝拉 (Isabella) 使用该应用程序向发行商业票据的分类账提交交易 `00001`。让我们快速检查一下 `issue` 应用程序是如何工作的。



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

因为该 `issue` 应用程序代表 Isabella 提交事务，所以它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，`issue` 应用程序可以利用网关在通道上提交事务。Hyperledger Fabric SDK 提供了 网关 抽象，因此应用程序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。网关和钱包使编写 Hyperledger Fabric 应用程序变得很简单。

因此，让我们检查一下 `issue` Isabella 将要使用的应用程序。为她打开一个单独的终端窗口，然后 `fabric-samples` 找到 MagnetoCorp `/application` 文件夹：

```
(magnetocorp user)$ cd commercial-paper/organization/magnetocorp/application/  
(magnetocorp user)$ ls
```

addToWallet.js issue.js package.json

`addToWallet.js` 是 Isabella 将用于将其身份加载到她的钱包中的程序, `issue.js` 并将使用该身份 `00001` 代表 MagnetoCorp 通过调用来创建商业票据 `papercontract`。

转到包含 MagnetoCorp 应用程序副本的目录 `issue.js`, 然后使用代码编辑器进行检查:

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

检查该目录; 它包含问题应用程序及其所有依赖项。

```
// Connect to gateway using application specified parameters
console.log('Connect to Fabric gateway.');

await gateway.connect(connectionProfile, connectionOptions);

// Access PaperNet network
console.log('Use network channel: mychannel.');

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

// Get addressability to commercial paper contract
console.log('Use org.paper.net.commercialpaper smart contract.')

const contract = await network.getContract('papercontract', 'org.paper.net.commercialpaper');

// issue commercial paper
console.log('Submit commercial paper issue transaction.');

const issueResponse = await contract.submitTransaction('issue', {})

// process response
console.log('Process issue transaction response.');

let paper = CommercialPaper.fromBuffer(issueResponse);

console.log(`${paper.issuer} commercial paper : ${paper.paperNumber}`);
console.log('Transaction complete.');

} catch (error) {

    console.log(`Error processing transaction: ${error}`);
}
```

一个代码编辑器, 显示商业票据应用程序目录的内容。

请注意以下关键程序行 `issue.js`:

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

该语句将两个关键的 Hyperledger Fabric SDK 类纳入了范围- `Wallet` 和 `Gateway`。由于 Isabella 的 X.509 证书位于本地文件系统中, 因此该应用程序使用 `FileSystemWallet`。

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

该语句标识该应用程序 `isabella` 在连接到区块链网络通道时将使用钱包。该应用程序将在 `isabella` 钱包中选择一个特定的身份。(钱包必须已经装有 Isabella 的 X.509 证书-这样做 `addToWallet.js` 是正确的。)

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

这行代码使用标识的网关 `connectionProfile`, 使用中标识的网关连接到网络 `ConnectionOptions`。

分别查看如何 `../gateway/networkConnection.yaml` 和 `User1@org1.example.com` 用于这些值。

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

此连接应用到网络信道 `mychannel`, 其中, 所述 `papercontract` 先前实例化。

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

这种结给人应用寻址通过命名空间中定义智能合同 `org.papernet.commercialpaper` 内 `papercontract`。一旦应用程序发布了 `getContract`, 它就可以提交在其中实现的任何交易。

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

此代码行使用 `issue` 智能合约中定义的交易将交易提交到网络。`MagnetoCorp`, `00001` ...是该 `issue` 交易用于创建新商业票据的值。

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

该语句处理来自 `issue` 事务的响应。需要将响应从缓冲区反序列化为对象 `paper`, 该 `CommercialPaper` 对象可以由应用程序正确解释。

随时检查目录中的其他文件 `/application` 以了解其 `issue.js` 工作原理, 并在应用程序主题中详细阅读其实现方式。

## 3.9 应用程序依赖

该 `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。按照惯例, MagnetoCorp 的应用程序建立在许多外部节点程序包上-以提高开发质量和速度。考虑一下 `issue.js` 包括 `js-yaml` 包处理 YAML 网关连接配置文件, 或 `fabric-network` 包访问的 `Gateway` 和 `Wallet` 类:

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

这些软件包必须使用命令从 `npm` 下载到本地文件系统。按照惯例, 必须将软件包安装到相对于应用程序的目录中, 以便在运行时使用。 `npm install/node_modules`

检查 `package.json` 文件以查看如何 `issue.js` 标识要下载的软件包及其确切版本:

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

`npm` 版本控制功能非常强大; 您可以[在此处](#)了解更多信息。

让我们使用命令安装这些软件包-这可能需要一分钟才能完成: `npm install`

```
(magnetocorp user)$ cd commercial-paper/organization/magnetocorp/application/
(magnetocorp user)$ npm install
```

```
(          ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

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
```

检查 `node_modules` 目录以查看已安装的软件包。有很多, 因为 `js-yaml` 和 `fabric-network` 本身是基于其他 npm 软件包构建的! 有用的是, 该 `package-lock.json` 文件标识了所安装的确切版本, 如果您想精确地复制环境, 这可以证明是非常宝贵的。以测试, 诊断问题或交付经过验证的应用程序。

## 3.10 钱包

伊莎贝拉几乎准备 `issue.js` 发行 MagnetoCorp 商业票据 `00001`; 剩下要做的一项任务! 由于 `issue.js` 代表伊莎贝拉 (Isabella) 并因此代表 MagnetoCorp, 它将使用她钱包里的身份来反映这些事实。现在, 我们需要执行这一—次性活动, 向她的钱包添加适当的 X.509 凭据。

在 Isabella 的终端窗口中，运行 `addToWallet.js` 程序以将身份信息添加到她的钱包：

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

```
done
```

Isabella 可以在她的钱包中存储多个身份，尽管在我们的示例中，她仅使用一个 - `User1@org.example.com`。该身份当前与基本网络相关联，而不是与更现实的 PaperNet 配置相关联。我们将尽快更新本教程。

`addToWallet.js` 是一个简单的文件复制程序，您可以随时检查。它将身份从基本网络样本转移到 Isabella 的钱包。让我们关注这个程序的结果 - 将用于向以下交易提交钱包的内容 `PaperNet`：

```
(isabella)$ ls ..\identity\user\isabella\wallet\
```

```
User1@org1.example.com
```

查看目录结构如何映射 `User1@org1.example.com` 身份 - Isabella 使用的其他身份将拥有自己的文件夹。在此目录中，您将找到 `issue.js` 将代表的身份信息 `isabella`：

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

```
User1@org1.example.com      c75bd6911a...-priv      c75bd6911a...-pub
```

注意：

- `c75bd6911a...-priv` 用于代表 Isabella 签署交易的私钥，但未在她的直接控制范围之外分发。
- 公共密钥 `c75bd6911a...-pub` 被加密连接到梁洛施的私人密钥。这完全包含在 Isabella 的 X.509 证书中。
- 证书 `User1@org.example.com`，其中包含 Isabella 的公钥以及证书创建时证书颁发机构添加的其他 X.509 属性。该证书分发到网络，以便不同的参与者在不同的时间可以通过密码验证由伊莎贝拉私钥创建的信息。

[在此处](#) 了解有关证书的更多信息。实际上，证书文件还包含一些特定于 Fabric 的元数据，例如 Isabella 的组织和角色。在 `wallet` 主题中了解更多。

## 3.11 发行申请

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

```
(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.
```

该 `node` 命令将初始化一个 `node.js` 环境，并运行 `issue.js`。从程序输出中我们可以看到，发行了面值为 500 万美元的 MagnetoCorp 商业票据 `00001`。

如您所见，为了实现这一点，应用程序调用 `issue` 中的 `CommercialPaper` 智能合约中定义的交易 `papercontract.js`。MagnetoCorp 管理员已在网络中安装并实例化了该文件。这是智能合约，它通过 Fabric API 与分类帐进行交互，最显着的是 `putState()` 和 `getState()`，将新的商业票据表示为世界状态内的向量状态。我们将看到如何通过智能合约中也定义的 `buy` 和 `redeem` 事务来操纵此向量状态。

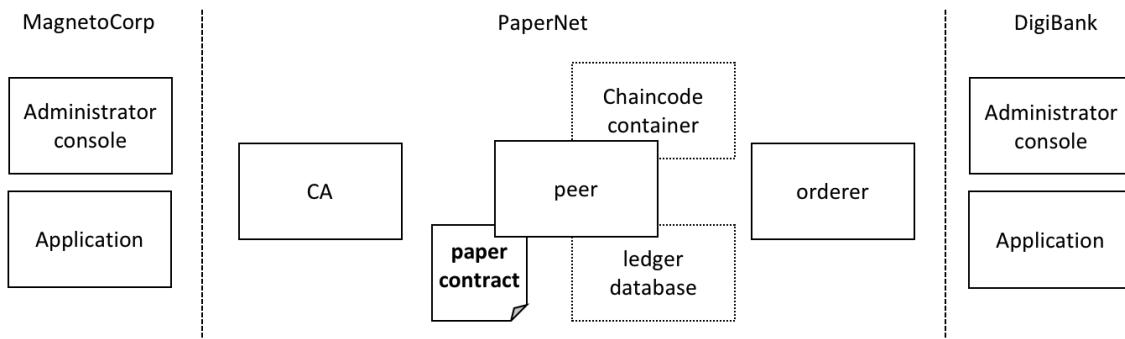
底层的 Fabric SDK 一直都在处理交易认可、排序和通知过程，从而使应用程序的逻辑简单明了；SDK 使用 [网关](#) 提取网络详细信息和 `connectionOptions` 来声明更高级的处理策略，例如事务重试。

现在，让 `00001` 我们关注 MagnetoCorp 的生命周期，将重点转移到将购买商业票据的 DigiBank。

## 3.12 作为 DigiBank

既然 `00001` MagnetoCorp 已经发布了商业票据，让我们切换上下文以与 DigiNet 作为 DigiBank 的雇员进行交互。首先，我们将以管理员身份创建一个配置为与 PaperNet 进行交互的控制台。然后，最终用户 Balaji 将使用 DigiBank

的 **buy** 应用程序购买商业票据 **00001**，将其转移到生命周期的下一个阶段。



DigiBank 管理员和应用程序与 PaperNet 网络交互。

由于本教程当前使用 PaperNet 的基本网络，因此网络配置非常简单。管理员使用类似于 MagnetoCorp 的控制台，但为 Digibank 的文件系统配置了控制台。同样，Digibank 最终用户将使用与 MagnetoCorp 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。无论是哪个应用程序调用它们，智能合约都可以捕获共享的业务流程，而分类账则可以存储共享的业务数据。

让我们打开一个单独的终端，使 DigiBank 管理员可以与 PaperNet 进行交互。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
```

现在，此 Docker 容器可供 Digibank 管理员与网络交互：

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

在本教程中，您将使用名为 **cliDigiBank** DigiBank 的命令行容器与网络交互。我们没有显示所有的 Docker 容器，在现实世界中，DigiBank 用户将只能看到他们可以访问的网络组件（对等，排序者，CA）。

Digibank 的管理员现在在本教程中不需要做很多事情，因为 PaperNet 网络配置非常简单。让我们把注意力转向 Balaji。

### 3.13 Digibank 应用

Balaji 使用 DigiBank 的 **buy** 应用程序向分类帐提交交易，分类帐将商业票据的所有权 **00001** 从 MagnetoCorp 转移到 DigiBank。该 **CommercialPaper** 智能合同是一样的，通过 MagnetoCorp 的应用程序使用，然而该交易不同，这一次-这是 **buy** 不是 **issue**。让我们研究一下 DigiBank 的应用程序如何工作。

打开 Balaji 的单独终端窗口。在中 **fabric-samples**，转到包含该应用程序的 DigiBank 应用程序目录，**buy.js**，然后使用编辑器将其打开：

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

如您所见，此目录同时包含 Balaji 将使用的 **buy** 和 **redeem** 应用程序。

```

buy.js — application
buy.js × JS redeem.js
10 // Bring key classes into scope, most importantly Fabric SDK network
11 const fs = require('fs');
12 const yaml = require('js-yaml');
13 const { FileSystemWallet, Gateway } = require('fabric-network');
14 const CommercialPaper = require('../contract/lib/paper.js');
15
16 // A wallet stores a collection of identities for use
17 const wallet = new FileSystemWallet('../identity/user/balaji/wallet');
18
19 // Main program function
20 async function main() {
21
22     // A gateway defines the peers used to access Fabric networks
23     const gateway = new Gateway();
24
25     // Main try/catch block
26     try {
27
28         // Specify userName for network access
29         // const userName = 'isabella.issuer@magnetcorp.com';
30         const userName = 'Admin@org1.example.com';
31
32         // Load connection profile; will be used to locate a gateway
33         let connectionProfile = yaml.safeLoad(fs.readFileSync('../gateways/gateway-profile.yaml'));
34
35         // Set connection options; identity and wallet
36         let connectionOptions = {
37             identity: userName,
38             wallet: wallet,
39             discovery: { enabled:false, aslocalhost: true }
40     
```

DigiBank 的商业票据目录包含 `buy.js` 和 `redeem.js` 应用程序。

DigiBank 的 `buy.js` 应用程序与 MagnetoCorp 的应用程序在结构上非常相似，但 `issue.js` 有两个重要区别：

- **身份**: 该用户是 DigiBank 用户，`Balaji` 而不是 MagnetoCorp 的 `Isabella`
- `const wallet = new FileSystemWallet('../identity/user/balaji/wallet');`  
查看应用程序 `balaji` 在连接到 PaperNet 网络通道时如何使用钱包。`buy.js` 选择 `balaji` 钱包中的特定身份。
- **交易**: 被调用的交易 `buy` 不是 `issue`
- `const buyResponse = await contract.submitTransaction('buy', 'MagnetoCorp', '00001'...);``  
— `buy` 交易与价值观提交 `MagnetoCorp`, `00001` ..., 被使用的 `CommercialPaper` 智能合同类商业票据的所有权转让 `00001` 给 DigiBank。

随时检查目录中的其他文件 `application` 以了解应用程序如何工作，并详细阅读 `buy.js` 应用程序主题中如何实现。

## 3.14 以 DigiBank 身份运行

购买和赎回商业票据的 DigiBank 应用程序与 MagnetoCorp 的发行应用程序具有非常相似的结构。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赎回商业票据。

与 MagnetoCorp 一样，Dibank 必须使用命令安装所需的应用程序包，这又需要很短的时间才能完成。

`npm install`

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

在 Balaji 的终端窗口中，运行 `addToWallet.js` 程序以将身份信息添加到他的钱包：

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

done

该 `addToWallet.js` 方案增加了身份信息 `balaji`，他的钱包，这将被用来 `buy.js` 和 `redeem.js` 提交交易 `PaperNet`。

与 Isabella 一样，Balaji 可以在他的钱包中存储多个身份，尽管在我们的示例中，他仅使用一个 - `Admin@org.example.com`。他对应的钱包结构 `digibank/identity/user/balaji/wallet/Admin@org1.example.com` 包含与伊莎贝拉非常相似的-随时检查。

## 3.15 购买申请

Balaji 现在可以 `buy.js` 用来提交一笔交易，该交易会将 MagnetoCorp 商业票据的所有权转让 `00001` 给 DigiBank。

`buy` 在 Balaji 的窗口中运行该应用程序：

```
(balaji)$ node buy.js
```

```
Connect to Fabric gateway.  
Use network channel: mychannel.  
Use org.papernet.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.
```

您可以看到程序输出，Balaji 代表 DigiBank 成功购买了 MagnetoCorp 商业用纸 00001。`buy.js` 调用了智能合约中 `buy` 定义的交易，该交易使用和 Fabric API 在世界范围内 `CommercialPaper` 更新了商业票据。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。`00001putState()getState()`

## 3.16 兑换申请

商业票据生命周期中的最后一笔交易 `00001` 是 DigiBank 用 MagnetoCorp 赎回它。Balaji 用于 `redeem.js` 提交交易以执行智能合约中的兑换逻辑。

`redeem` 在 Balaji 的窗口中运行事务：

```
(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.
```

再次，请参见在 `redeem.js` 调用中 `redeem` 定义的交易时如何成功赎回商业票据 00001 `CommercialPaper`。再次，它更新 `00001` 了世界范围内的商业票据，以反映所有权归还给票据发行人 MagnetoCorp。

## 3.17 进一步阅读

为了更详细地了解本教程中显示的应用程序和智能合约的工作原理，您会发现阅读[开发应用程序](#)很有帮助。本主题将为您提供有关商业票据方案，`PaperNet` 商业网络，其参与者以及他们使用的应用程序和智能合约如何工作的更详细说明。

也可以随时使用此示例开始创建自己的应用程序和智能合约！

# 4 建立您的第一个网络

## 注意

这些说明已通过验证，可与提供的 tar 文件中的最新稳定 Docker 映像和预编译的设置实用程序一起使用。如果使用当前主分支中的图像或工具运行这些命令，则可能会看到配置和紧急错误。

构建您的第一个网络 (BYFN) 方案将提供一个示例 Hyperledger Fabric 网络，该网络由两个组织组成，每个组织维护两个对等节点。尽管可以使用其他排序节点服务实现，但默认情况下还将部署“Solo”排序节点服务。

## 4.1 安装先决条件

在我们开始之前，如果您尚未这样做，则不妨检查一下是否已在要开发区块链应用程序和/或运行 Hyperledger Fabric 的平台上安装了所有**必备**软件。

您还需要安装 Samples, Binaries 和 Docker Images。您会注意到 `fabric-samples` 存储库中包含许多示例。我们将使用 `first-network` 示例。现在打开该子目录。

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

## 注意

本文档中提供的命令**必须**从存储库克隆 `first-network` 的子目录中运行 `fabric-samples`。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。

## 4.2 要立即运行吗？

我们提供了一个带有完整注释的脚本- `byfn.sh` 利用这些 Docker 映像快速引导 Hyperledger Fabric 网络，该网络默认情况下由代表两个不同组织的四个对等方和一个排序节点者节点组成。它还将启动一个容器来运行脚本执行，该脚本执行会将对等方加入到通道中，部署链码并根据已部署的链码推动事务的执行。

这是 `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>] [-o <consensus-type>] [-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 version 1.3.x to 1.4.0"  
-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"  
-o <consensus-type> - the consensus-type of the ordering service: solo (default), kafka, or  
etc_draft"  
-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 1.4.0"
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"
```

如果选择不提供标志，则脚本将使用默认值。

## 4.2.1 生成网络工件

准备好尝试了吗？好吧！执行以下命令：

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

您将看到有关发生的情况的简短说明，以及是/否命令行提示符。用 a 响应 **y** 或按回车键执行所描述的操作。

```
Generating certs and genesis block for channel 'mychannel' with CLI timeout of '10' seconds and CLI
delay of '3' seconds
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
InitFactories(). Falling back to bootBCCSP.
...
...
```

```
/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
[/Users/xxx/dev/byfn/crypto-config/ordererOrganizations/example.com/msp/intermediatecerts].
Skipping.: [stat /Users/xxx/dev/byfn/crypto-
config/ordererOrganizations/example.com/msp/intermediatecerts: no such file or directory]
...
2017-06-12 21:01:37.588 EDT [common/configtx/tool] doOutputBlock -> INFO 00b Generating genesis
block
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
new channel configtx
2017-06-12 21:01:37.645 EDT [common/configtx/tool] doOutputChannelCreateTx -> INFO 003 Writing new
channel tx
```

```
#####
##### 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
anchor peer update
2017-06-12 21:01:37.679 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 003 Writing
anchor peer update
```

```
#####
##### 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
anchor peer update
```

```
2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 003 Writing anchor peer update
```

第一步将为我们的各种网络实体生成所有证书和密钥，用于引导排序节点服务，以及配置 Channel 所需的配置事务的集合。**genesis block**

## 4.2.2 建立网络

接下来，您可以使用以下命令之一启动网络：

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

上面的命令将编译 Golang 链码图像并旋转相应的容器。Go 是默认的链码语言，但是还支持 [Node.js](#) 和 [Java](#) 链码。如果您想使用节点链码来完成本教程，请改用以下命令：

```
# we use the -l flag to specify the chaincode Language  
# forgoing the -l flag will default to GoLang
```

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

注意

有关 Node.js 填充程序的更多信息，请参阅其 [文档](#)。

注意

有关 Java 填充程序的更多信息，请参阅其 [文档](#)。

使样本使用 Java chaincode 运行，您必须指定如下内容：**-l java**

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

注意

不要同时运行这两个命令。除非您关闭并重新建立两者之间的网络，否则只能尝试一种语言。

除了支持多种链码语言外，您还可以发出一个标志，该标志将显示一个五节点的 Raft 排序节点服务或 Kafka 排序节点服务，而不是一个节点的 Solo 排序节点服务。有关当前支持的排序节点服务实现的更多信息，请查看 [The Ordering Service](#)。

要使用 Raft 排序节点服务启动网络，请发出：

```
./byfn.sh up -o etcdraft
```

要使用 Kafka 排序节点服务建立网络，请发出：

```
./byfn.sh up -o kafka
```

再次提示您是否要继续还是中止。回应一个 **y** 或点击返回键：

```
Starting for channel 'mychannel' with CLI timeout of '10' seconds and CLI delay of '3' seconds  
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
```



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

日志将从此处继续。这将启动所有容器，然后驱动一个完整的端到端应用程序方案。成功完成后，它将在您的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 =====
```



您可以滚动浏览这些日志以查看各种事务。如果未得到此结果，请跳至“[故障排除](#)”部分，让我们看看我们是否可以帮助您发现问题所在。

### 4.2.3 中断网络

最后，让我们将其全部介绍下来，以便我们一次可以探索网络设置。以下内容将杀死您的容器，删除加密材料和四个工件，并从 Docker 注册表中删除链码映像：

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

再次，您将被提示继续，用 a 响应 `y` 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  
...
```

如果您想了解更多有关底层工具和引导机制的信息，请继续阅读。在接下来的几节中，我们将逐步介绍构建一个功能齐全的 Hyperledger Fabric 网络的各个步骤和要求。

注意

下面概述的手动步骤假定 `FABRIC_LOGGING_SPEC` 在 `cli` 容器被设置为 `DEBUG`。您可以通过修改目录中的 `docker-compose-cli.yaml` 文件来进行设置 `first-network`。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
```

## 4.3 加密发生器

我们将使用该 `cryptogen` 工具为我们的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。

### 4.3.1 它是如何工作的？

Cryptogen 使用一个文件 `crypto-config.yaml` 包含网络拓扑，并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-cert`)，它将特定组件 (对等和排序节点) 绑定到该组织。通过为每个组织分配唯一的 CA 证书，我们正在模仿一个典型的网络，参与的成员将使用其自己的证书颁发机构。Hyperledger Fabric 中的事务和通信由实体的私钥 (`keystore`) 签名，然后通过公钥 () 进行验证 `signcerts`。

您会 `count` 在此文件中注意到一个变量。我们使用它来指定每个组织的对等点数；在我们的案例中，每个单位有两个同级。我们现在不会深入研究 `x.509 证书和公钥基础结构` 的细节。如果您有兴趣，可以自行研究这些主

题。

运行该 `cryptogen` 工具后，生成的证书和密钥将保存到名为的文件夹中 `crypto-config`。请注意，该 `crypto-config.yaml` 文件列出了五个与排序节点者组织相关的排序节点者。尽管该 `cryptogen` 工具将为所有五个排序节点者创建证书，除非使用了 Raft 或 Kafka 排序节点服务，否则这些排序节点者中只有一个将用于 Solo 排序节点服务实现中并用于创建系统频道和 `mychannel`。

## 4.4 配置事务生成器

该 `configtxgen` 工具用于创建四个配置工件：

- 排序节点者，`genesis block`
- 通道，`configuration transaction`
- 和两个-每个对等组织一个。`anchor peer transactions`

有关此工具功能的完整说明，请参见 `configtxgen`。

排序节点者模块是排序节点服务的创世纪模块，并且在频道创建时将频道配置事务文件广播到排序节点者。顾名思义，锚点对等事务指定此通道上的每个组织的锚点对等体。

### 4.4.1 它是如何工作的？

Configtxgen 使用文件-- `configtx.yaml` 包含示例网络的定义。有三个成员-一个排序节点者组织(`OrdererOrg`) 和两个对等组织 (`Org1` & `Org2`)，每个成员都管理和维护两个对等节点。该文件还指定了一个财团 - `SampleConsortium` 由我们的两个对等组织组成。请特别注意此文件底部的“配置文件”部分。您会注意到我们有几个唯一的配置文件。一些值得注意的地方：

- `TwoOrgsOrdererGenesis`：生成 SOLO 排序节点服务的创始块。
- `SampleMultiNodeEtcdRaft`：生成 RAFT 排序节点服务的创始块。仅在发出 `-o` 标志并指定时使用 `etcdrift`。
- `SampleDevModeKafka`：为 Kafka 排序节点服务生成创世块。仅在发出 `-o` 标志并指定时使用 `kafka`。
- `TwoOrgsChannel`：为我们的频道生成创世块 `mychannel`。

这些标题很重要，因为在创建工作时，我们会将它们作为参数传递。

注意

注意，我们 `SampleConsortium` 是在系统级配置文件中定义的，然后由我们的通道级配置文件引用。渠道存在于联盟的权限范围内，所有联盟都必须在整个网络范围内定义。

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` & `peer0.org2.example.com`) 指定锚点对等体。其次，我们指向每个成员的 MSP 目录的位置，从而允许我们将每个组织的根证书存储在排序节点者创始块中。这是一个关键的概念。现在，与排序节点服务通信的任何网络实体都可以验证其数字签名。

## 4.5 运行工具

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，您可以尝试改编 `byfn.sh` 脚本以实现目标。

## 4.5.1 手动生成工件

您可以 `generateCerts` 在 `byfn.sh` 脚本中引用该函数，以获取生成用于 `crypto-config.yaml` 文件中定义的网络配置的证书所需的命令。但是，为方便起见，我们还将在此处提供参考。

首先，让我们运行该 `cryptogen` 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
```

您应该在终端中看到以下内容：

```
org1.example.com  
org2.example.com
```

证书和密钥（即 MSP 资料）将输出到目录 `crypto-config` - 位于目录根 `first-network` 目录下。

接下来，我们需要告诉该 `configtxgen` 工具在哪里寻找 `configtx.yaml` 需要提取的文件。我们将在当前的工作目录中告诉它：

```
export FABRIC_CFG_PATH=$PWD
```

然后，我们将调用该 `configtxgen` 工具来创建排序节点者创始块：

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

要输出 Raft 排序节点服务的创始块，此命令应为：

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

请注意 `SampleMultiNodeEtcdRaft` 此处使用的配置文件。

要输出 Kafka 排序节点服务的创世块，请发出：

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

如果您未使用 Raft 或 Kafka，则应看到类似于以下内容的输出：

```
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
```

注意

排序节点者的创世块和我们将要创建的后续工件将输出到 `channel-artifacts` 该项目根目录下的目录中。上面命令中的 `channelID` 是系统通道的名称。

## 4.5.2 创建渠道配置交易

接下来，我们需要创建通道事务构件。确保替换 `$CHANNEL_NAME` 或设置 `CHANNEL_NAME` 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
```

请注意，如果您使用的是 Raft 或 Kafka 排序节点服务，则不必对通道发出特殊命令。该 `TwoOrgsChannel` 配置文件将使用您在创建网络的创世纪模块时指定的排序节点服务配置。

如果您不使用 Raft 或 Kafka 排序节点服务，则应在终端中看到类似于以下内容的输出：

```
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
```

接下来，我们将在正在构建的通道上为 Org1 定义锚点。同样，请确保 `$CHANNEL_NAME` 为以下命令替换或设置环境变量。终端输出将模拟通道事务工件的输出：

```
./bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchor.tx -channelID $CHANNEL_NAME -asOrg Org1MSP
```

现在，我们将在同一通道上为 Org2 定义锚点：

```
./bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchor.tx -channelID $CHANNEL_NAME -asOrg Org2MSP
```

## 4.6 启动网络

注意

如果您 `byfn.sh` 以前运行了上面的示例，请确保在继续操作之前关闭了测试网络（请参阅 [关闭网络](#)）。

我们将利用脚本来启动我们的网络。`docker-compose` 文件引用了我们先前下载的图像，并使用我们先前生成的引导了排序节点程序 `genesis.block`。

我们希望手动检查命令，以显示每个调用的语法和功能。

首先，让我们开始我们的网络：

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

如果要查看网络的实时日志，请不要提供该 `-d` 标志。如果让日志流传输，则需要打开第二个终端以执行 CLI 调用。

### 4.6.1 创建并加入频道

回想一下，我们使用上面 `configtxgen` “创建渠道配置交易”部分中的工具创建了渠道配置交易。您可以使用 `configtx.yaml` 传递给 `configtxgen` 工具的相同或不同的配置文件，重复该过程以创建其他渠道配置事务。然后，您可以重复本节中定义的过程以在网络中建立其他通道。

我们将使用以下命令输入 CLI 容器：`docker exec`

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

如果成功，您应该看到以下内容：

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

为了使以下 CLI 命令 `peer0.org1.example.com` 无法正常工作，我们需要在命令前添加以下四个环境变量。这些 for `peer0.org1.example.com` 的变量被烘焙到 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**，如果要将呼叫发送给其他对等方或排序节点者，请保持 CLI 容器默认为 target `peer0.org1.example.com`，但是在进行任何 CLI 呼叫时覆盖环境变量，如下面的示例所示：

```
# Environment variables for PEER0
```

```
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/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

接下来，作为创建通道请求的一部分，我们将把在“创建通道配置事务”部分（称为 `channel.tx`）中创建的生成的通道配置事务工件传递给排序节点者。

我们用 `-c` 标志指定通道名称，并用标志指定通道配置事务 `-f`。在这种情况下为 `channel.tx`，但是您可以使用其他名称挂载自己的配置事务。再次，我们将 `CHANNEL_NAME` 在 CLI 容器中设置环境变量，以便我们不必显式传递此参数。频道名称必须全部为小写字母，少于 250 个字符，并且与正则表达式匹配 `[a-z][a-z0-9.-]*`。

```
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
```

#### 注意

请注意 `--cafile`，我们作为此命令的一部分进行了传递。这是排序节点者根证书的本地路径，使我们可以验证 TLS 握手。

此命令返回一个创世块- `<CHANNEL_NAME.block>` 我们将使用它来加入频道。它包含在中指定的配置信息。

`channel.tx` 如果您未对默认通道名称进行任何修改，则该命令将返回标题为的原型 `mychannel.block`。

#### 注意

这些手动命令的其余部分将保留在 CLI 容器中。当定位除以外的对等对象时，您还必须记住在所有命令前加上相应的环境变量 `peer0.org1.example.com`。

现在让我们加入 `peer0.org1.example.com` 频道。

```
# By default, this joins ``peer0.org1.example.com`` only
# the <CHANNEL_NAME.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
```

您可以通过在上面的“[创建和加入频道](#)”部分中使用的四个环境变量进行适当的更改，使其他对等节点加入频道。

与其加入每个对等点，不如简单地加入，`peer0.org2.example.com` 以便我们可以正确地更新频道中的锚点对等点定义。由于我们将覆盖烘焙到 CLI 容器中的默认环境变量，因此该完整命令如下：

```
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:9051
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
```

另外，您可以选择单独设置这些环境变量，而不是传入整个字符串。设置好之后，您只需再次发出命令，CLI 容器将代表。`peer channel join peer0.org2.example.com`

## 4.6.2 更新锚点对等点

以下命令是通道更新，它们将传播到通道的定义。本质上，我们在通道的创世块之上添加了其他配置信息。请注意，我们不是在修改创世块，而只是将增量添加到将定义锚点对等点的链中。

更新通道定义以将 Org1 的锚点对等点定义为 `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
```

现在更新通道定义，将 Org2 的锚点对等点定义为 `peer0.org2.example.com`。与 Org2 对等方的命令相同，我们需要在此调用之前加上适当的环境变量。`peer channel join`

```
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:9051
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
```

## 4.6.3 安装并定义一个链码

#### 注意

我们将利用一个简单的现有链码。要了解如何编写自己的链码，请参阅《[开发人员链码](#)》教程。

#### 注意

这些说明使用 v2.0 Alpha 版本中引入的 Fabric 链码生命周期。如果您想使用之前的生命周期来安装和实例化链码，请访问[构建您的第一个网络教程](#)的 v1.4 版本。

应用程序通过进行与区块链分类账的交互 `chaincode`。因此，我们需要在每个将执行并认可我们交易的对等方上安装一个链码。但是，在我们与链码进行交互之前，渠道成员需要就建立链码治理的链码定义达成共识。

我们需要打包链码，然后才能将其安装在我们的同级上。对于您创建的每个软件包，您需要提供一个链码软件包标签作为链码的描述。使用以下命令打包示例 Go, Node.js 或 Java chaincode。

## Golang

```
# this packages a Golang chaincode.  
# make note of the --lang flag to indicate "goLang" chaincode  
# for go chaincode --path takes the relative path from $GOPATH/src  
# The --label flag is used to create the package label  
peer lifecycle chaincode package mycc.tar.gz --path github.com/hyperledger/fabric-samples/chaincode/abstore/go/ --lang golang --label mycc_1
```

## Node.js

```
# this packages a Node.js chaincode  
# make note of the --lang flag to indicate "node" chaincode  
# for node chaincode --path takes the absolute path to the node.js chaincode  
# The --label flag is used to create the package label  
peer lifecycle chaincode package mycc.tar.gz --path /opt/gopath/src/github.com/hyperledger/fabric-samples/chaincode/abstore/node/ --lang node --label mycc_1
```

## Java

```
# this packages a java chaincode  
# make note of the --lang flag to indicate "java" chaincode  
# for java chaincode --path takes the absolute path to the java chaincode  
# The --label flag is used to create the package label  
peer lifecycle chaincode package mycc.tar.gz --path /opt/gopath/src/github.com/hyperledger/fabric-samples/chaincode/abstore/java/ --lang java --label mycc_1
```

上面的每个命令都会创建一个名为的链码包 `mycc.tar.gz`，我们可以使用它在对等节点上安装链码。发出以下命令在 Org1 的 peer0 上安装软件包。

```
# this command installs a chaincode package on your peer  
peer lifecycle chaincode install mycc.tar.gz
```

成功的安装命令将返回一个链码包标识符。您应该看到类似于以下内容的输出：

```
2019-03-13 13:48:53.691 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 001 Installed  
remotely: response:<status:200  
payload:"\nE\nmycc_1:3a8c52d70c36313cfbbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173" >  
2019-03-13 13:48:53.691 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 002 Chaincode  
code package identifier: mycc_1:3a8c52d70c36313cfbbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173
```

您还可以通过查询对等方以获取有关已安装软件包的信息来找到链码软件包标识符。

```
# this returns the details of the chaincode packages installed on your peers  
peer lifecycle chaincode queryinstalled
```

上面的命令将返回与安装命令相同的软件包标识符。您应该看到类似于以下内容的输出：

```
Get installed chaincodes on peer:  
Package ID: mycc_1:3a8c52d70c36313cfbbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173, Label: mycc_1
```

我们将来的命令将需要软件包 ID，所以让我们继续将其保存为环境变量。将对等生命周期链码 `queryinstalled` 命令返回的软件包 ID 粘贴到以下命令中。软件包 ID 对于所有用户而言可能都不相同，因此您需要使用从控制台返回的软件包 ID 来完成此步骤。

```
# Save the package ID as an environment variable.
```

```
CC_PACKAGE_ID=mycc_1:3a8c52d70c36313cfbbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173
```

背书政策 `mycc` 将设置为要求来自 Org1 和 Org2 的对等方的背书。因此，我们还需要在 Org2 的对等节点上安装 chaincode。

修改以下四个环境变量以将安装命令作为 Org2 发出：

```
# 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:9051
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
```

现在，将 chaincode 软件包安装到 Org2 的 peer0 上。以下命令将安装链码，并返回与我们作为 Org1 发出的安装命令相同的标识符。

```
# this installs a chaincode package on your peer
peer lifecycle chaincode install mycc.tar.gz
```

安装软件包后，需要批准组织的链码定义。链码定义包括链码治理的重要参数，包括链码名称和版本。该定义还包括包标识符，该包标识符用于将对等方安装的链码包与组织批准的链码定义相关联。

因为我们将环境变量设置为可以作为 Org2 进行操作，所以可以使用以下命令来批准 mycc Org2 的链码的定义。批准分配给每个组织内的对等方，因此该命令不需要针对组织内的每个对等方。

```
# this approves a chaincode definition for your org
# make note of the --package-id flag that provides the package ID
# use the --init-required flag to request the ``Init`` function be invoked to initialize the
chaincode
peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name mycc --version 1.0 --init-
required --package-id $CC_PACKAGE_ID --sequence 1 --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
```

我们可以在上面的命令中提供 `--signature-policy` 或 `--channel-config-policy` 参数来设置 chaincode 认可策略。背书策略指定需要多少个属于不同通道成员的对等点才能根据给定的链码验证交易。因为我们没有设置策略，所以定义 mycc 将使用默认的认可策略，该策略要求在提交交易时，必须由存在的大多数渠道成员认可交易。这意味着，如果将新组织添加到渠道中或从渠道中删除，则背书策略会自动更新以要求更多或更少的背书。在本教程中，默认策略将要求来自属于 Org1 和 Org2 的对等方的背书（即两个背书）。见[背书政策](#) 文档以获取有关政策实施的更多详细信息。

所有组织都必须先同意定义，然后才能使用链码。修改以下四个环境变量以用作 Org1：

```
# Environment variables for PEER0
```

```
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/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

现在，您可以将 mycc 链码的定义批准为 Org1。Chaincode 在组织级别得到批准。即使您有多个对等端，也可以发出一次命令。

```
# this defines a chaincode for your org
# make note of the --package-id flag that provides the package ID
# use the --init-required flag to request the Init function be invoked to initialize the chaincode
peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name mycc --version 1.0 --init-
required --package-id $CC_PACKAGE_ID --sequence 1 --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
```

一旦足够数量的渠道成员批准了链码定义，一个成员就可以将该定义提交给渠道。默认情况下，大多数通道成员需要批准定义才能提交。通过发出以下查询，可以检查链码定义是否准备就绪，可以按组织查看当前的批准：

```
# the flags used for this command are identical to those used for approveformyorg
# except for --package-id which is not required since it is not stored as part of
# the definition
peer lifecycle chaincode checkcommitreadiness --channelID $CHANNEL_NAME --name mycc --version 1.0 --
init-required --sequence 1 --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 --output json
```

该命令将生成一个 JSON 映射作为输出，以显示通道中的组织是否已批准 checkcommitreadiness 命令中提供的链码定义。在这种情况下，假设两个组织都已批准，我们将获得：

```
{
```

```

        "Approvals": {
            "Org1MSP": true,
            "Org2MSP": true
        }
    }
}

```

由于两个渠道成员都批准了该定义，因此我们现在可以使用以下命令将其提交给渠道。您可以将此命令作为 Org1 或 Org2 发出。请注意，该交易针对 Org1 和 Org2 中的同级收集签注。

```

# this commits the chaincode definition to the channel
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID $CHANNEL_NAME --name mycc --
version 1.0 --sequence 1 --init-required --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 --peerAddresses
peer0.org1.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p
eer0.org2.example.com/tls/ca.crt

```

## 4.6.4 调用链码

将链码定义提交到通道后，我们准备调用链码并开始与分类帐进行交互。我们要求 `Init` 使用 `--init-required` 标志在链码定义中执行该功能。结果，我们需要将 `--isInit` 标志传递给它的第一次调用，并将参数提供给 `Init` 函数。发出以下命令来初始化链码并将初始数据放在分类帐中。

```

# be sure to set the -C and -n flags appropriately
# use the --isInit flag if you are invoking an Init function
peer chaincode invoke -o orderer.example.com:7050 --isInit --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/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p
eer0.org2.example.com/tls/ca.crt -c '{"Args":["Init","a","100","b","100"]}' --waitForEvent

```

第一次调用将启动 chaincode 容器。我们可能需要等待容器启动。Node.js 图像将花费更长的时间。

## 4.6.5 询问

让我们查询链码，以确保正确启动了容器并填充了状态数据库。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"]}'

```

## 4.6.6 调用

现在让我们 `10` 从 `a` 移至 `b`。此事务将剪切一个新块并更新状态数据库。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/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p
eer0.org2.example.com/tls/ca.crt -c '{"Args":["invoke","a","b","10"]}' --waitForEvent

```

## 4.6.7 询问

让我们确认先前的调用已正确执行。我们 `a` 使用值初始化键，`100` 并在 `10` 之前的调用中将其删除。因此，针对的查询 `a` 应返回 `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"]}'

```

我们应该看到以下内容：

Query Result: 90

## 4.6.8 在其他对等节点上安装链码

如果希望其他对等方与分类帐进行交互，则需要将它们加入通道并在对等方上安装相同的链码包。您只需要从您的组织批准一次链码定义。一旦每个对等方尝试与该特定链式代码进行交互，就会为每个对等者启动一个链式代码容器。同样，请注意 Node.js 映像的构建速度较慢，并在首次调用时启动。

我们将链码安装在 Org2 中的第三个对等方 peer1 上。修改以下四个环境变量以对 Org2 中的 peer1 发出安装命令：

```
# 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:10051
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
```

现在，将 `mycc` 软件包安装在 Org2 的 peer1 上：

```
# this command installs a chaincode package on your peer
peer lifecycle chaincode install mycc.tar.gz
```

## 4.6.9 询问

让我们确认我们可以向 Org2 中的 Peer1 发出查询。我们 `a` 使用值初始化键，`100` 并在 `10` 之前的调用中将其删除。因此，针对的查询 `a` 仍应返回 `90`。

Org2 中的 Peer1 必须首先加入通道，然后它才能响应查询。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:10051
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
```

连接命令返回后，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"]}'
```

我们应该看到以下内容：

Query Result: 90

如果收到错误，则可能是因为对等方加入并赶上当前的区块链高度需要花费几秒钟的时间。您可以根据需要重新查询。也可以随意执行其他调用。

## 4.6.10 幕后发生了什么？

### 注意

这些步骤描述了 `script.sh` 由'./byfn.sh up'运行的情况。使用清洁网络，并确保此命令处于活动状态。然后使用相同的 docker-compose 提示符再次启动网络 `./byfn.sh down`

- 脚本-- `script.sh` 在 CLI 容器中烘焙。该脚本 `createChannel` 根据提供的通道名称驱动命令，并使用 `channel.tx` 文件进行通道配置。
- 它输出 `createChannel` 的一个创世块-- `<your_channel_name>.block` 存储在对等方的文件系统上，并包含从 `channel.tx` 指定的通道配置。
- `joinChannel` 对所有四个对等设备执行该命令，该命令将先前生成的创世块作为输入。此命令指示对等方加入 `<your_channel_name>` 并创建以开头的链 `<your_channel_name>.block`。

- 现在，我们有一个由四个对等方和两个组织组成的渠道。这是我们的 `TwoOrgsChannel` 个人资料。
- `peer0.org1.example.com` 和 `peer1.org1.example.com` 属于 ORG1; `peer0.org2.example.com` 和 `peer1.org2.example.com` 属于 ORG2
- 这些关系是通过定义的 `crypto-config.yaml`，MSP 路径是在我们的 docker compose 中指定的。
- 然后，更新 Org1MSP (`peer0.org1.example.com`) 和 Org2MSP (`peer0.org2.example.com`) 的锚点。为此，我们将 `Org1MSPanchor.tx` 和 `Org2MSPanchor.tx` 工件以及渠道名称一起传递给排序节点服务。
- 一个 `chaincode - abstore` 被打包并安装在 `peer0.org1.example.com` 与 `peer0.org2.example.com`
- 然后，链码分别由 Org1 和 Org2 批准，然后在通道上提交。由于未指定背书策略，因此将利用大多数组织的渠道默认背书策略，这意味着任何交易都必须由与 Org1 和 Org2 绑定的对等方背书。
- 然后调用链码初始化，它启动目标对等方的容器，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"]`。第一次调用将以 `dev-peer0.org2.example.com-mycc-1.0` 开始的名称生成一个容器。
- 向“a”的值发出查询 `peer0.org2.example.com`。`dev-peer0.org2.example.com-mycc-1.0` 初始化链码时，已启动名为 Org2 peer0 的容器。返回查询结果。没有发生写操作，因此对“a”的查询仍将返回值“100”。
- 发送调用并将“10”从“a”移动到“b”`peer0.org1.example.com` 并将 `peer0.org2.example.com` 其移动
- 查询发送到 `peer0.org2.example.com` “a”的值。返回值 90，正确反映了先前的交易，在该交易中，键“a”的值被修改了 10。
- 链码-`abstore`-安装在 `peer1.org2.example.com`
- 查询发送到 `peer1.org2.example.com` “a”的值。这将启动名为的第三个 chaincode 容器 `dev-peer1.org2.example.com-mycc-1.0`。返回值 90，正确反映了先前的交易，在该交易中，键“a”的值被修改了 10。

## 4.6.11这说明了什么？

必须在对等方上安装 Chaincode，以使其能够对分类帐成功执行读/写操作。此外，直到 `init` 针对该链码执行了传统的交易（读/写）（例如，查询“a”的值）后，方才启动链码容器。事务导致容器启动。而且，通道中的所有对等方都维护分类帐的精确副本，该副本包括将区块，不可变的顺序记录存储在块中的区块链，以及维护当前状态快照的状态数据库。这包括未在其上安装链码的那些对等方（`peer1.org1.example.com` 如上例所示）。最后，安装链码后即可访问它（例如 `peer1.org2.example.com` 在上面的示例中），因为其定义已在通道上提交。

## 4.6.12我如何看待这些交易？

检查 CLI Docker 容器的日志。

```
docker logs -f cli
```

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
identity
2017-05-16 17:08:01.366 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext:
0AB1070A6708031A0C08F1E3ECC80510...6D7963631A0A0571756572790A0161
2017-05-16 17:08:01.367 UTC [msp/identity] Sign -> DEBU 007 Sign: digest:
E61DB37F4E8B0D32C9FE10E3936BA9B8CD278FAA1F3320B08712164248285C54
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 =====
```



您可以滚动浏览这些日志以查看各种事务。

#### 4.6.13 如何查看链码日志？

您可以检查各个 chaincode 容器，以查看针对每个容器执行的单独事务。使用以下命令查找正在运行的容器的列表以查找您的链码容器：

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
7aa7d9e199f5        dev-peer1.org2.example.com-mycc_1-
27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8-
2eba360c66609a3ba78327c2c86bc3abf041c78f5a35553191a1acf1efdd5a0d    "chaincode -peer.add..."  About a
minute ago   Up About a minute
mycc_1-27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8
82ce129c0fe6        dev-peer0.org2.example.com-mycc_1-
27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8-
1297906045aa77086daba21aba47e8eef359f9498b7cb2b010dff3e2a354565a    "chaincode -peer.add..."  About a
minute ago   Up About a minute
mycc_1-27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8
eaef1a8f7acf      dev-peer0.org1.example.com-mycc_1-
27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8-
00d8dbe7d5a4aeb9428b7df95df9744be1325b2a60900ac7a81796e67e4280a    "chaincode -peer.add..."  2
minutes ago   Up 2 minutes
peer0.org1.example.com-mycc_1-27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8
da403175b785      hyperledger/fabric-tools:latest
"/bin/bash"         4 minutes ago     Up 4 minutes
cli
c62a8d03818f      hyperledger/fabric-peer:latest
"peer node start"  4 minutes ago     Up 4 minutes
7051/tcp, 0.0.0.0:9051->9051/tcp
peer0.org2.example.com
06593c4f3e53      hyperledger/fabric-peer:latest
"peer node start"  4 minutes ago     Up 4 minutes
0.0.0.0:7051->7051/tcp
peer0.org1.example.com
4ddc928ebffe      hyperledger/fabric-orderer:latest
"orderer"          4 minutes ago     Up 4 minutes
0.0.0.0:7050->7050/tcp
orderer.example.com
6d79e95ec059      hyperledger/fabric-peer:latest
"peer node start"  4 minutes ago     Up 4 minutes
7051/tcp, 0.0.0.0:10051->10051/tcp
peer1.org2.example.com
6aad6b40fd30      hyperledger/fabric-peer:latest
"peer node start"  4 minutes ago     Up 4 minutes
7051/tcp, 0.0.0.0:8051->8051/tcp
peer1.org1.example.com
```

chaincode 容器是以 *dev-peer* 开头的图像。然后，您可以使用容器 ID 从每个 chaincode 容器中查找日志。

```
$ docker logs 7aa7d9e199f5
ABstore Init
Aval = 100, Bval = 100
ABstore Invoke
Aval = 90, Bval = 110

$ docker logs eaef1a8f7acf
ABstore Init
Aval = 100, Bval = 100
ABstore Invoke
Query Response:{ "Name": "a", "Amount": "100" }
ABstore Invoke
Aval = 90, Bval = 110
ABstore Invoke
Query Response:{ "Name": "a", "Amount": "90" }
```

您还可以查看对等日志，以查看链码调用消息和阻止提交消息：

```
$ docker logs peer0.org1.example.com
```

#### 4.7 了解 Docker Compose 拓扑

BYFN 示例向我们提供了两种 Docker Compose 文件，这两种文件都是从扩展的 `docker-compose-base.yaml`（位于 `base` 文件夹中）。我们的第一个风格，`docker-compose-cli.yaml` 为我们提供了一个 CLI 容器以及一个排序节点者和四个对等方。我们将此文件用于此页面上的全部说明。

#### 注意

本节的其余部分介绍了为 SDK 设计的 `docker-compose` 文件。有关运行这些测试的详细信息，请参考 [Node SDK 仓库](#)。

第二种风味 `docker-compose-e2e.yaml` 构造为使用 Node.js SDK 运行端到端测试。除了可以使用 SDK 之外，它的主要区别还在于，Fabric-ca 服务器具有容器。因此，我们能够将 REST 调用发送到组织 CA，以进行用户注册和注册。

如果要在 `docker-compose-e2e.yaml` 没有先运行 `byfn.sh` 脚本的情况下使用，那么我们将需要进行四个小修改。我们需要指向本组织 CA 的私钥。您可以在 `crypto-config` 文件夹中找到这些值。例如，要找到 Org1 的私钥，我们将遵循以下路径- `crypto-config/peerOrganizations/org1.example.com/ca/`。私钥是一个长哈希值，后跟 `_sk`。Org2 的路径为- `crypto-config/peerOrganizations/org2.example.com/ca/`。

在 `docker-compose-e2e.yaml` 更新中，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以启动 ca 服务器。您为每个 CA 容器两次提供相同的私钥。

## 4.8 使用 CouchDB

可以将状态数据库从默认（`goleveldb`）切换到 CouchDB。CouchDB 可以使用相同的链码功能，但是，还具有对链码数据建模为 JSON 的状态数据库数据内容执行丰富和复杂查询的功能。

要使用 CouchDB 代替默认数据库（`goleveldb`），请遵循前面概述的用于生成工件的步骤，但同时也要启动网络传递 `docker-compose-couch.yaml`：

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

**abstore** 现在应该在下面使用 CouchDB 进行工作。

#### 注意

如果您选择实现 `fabric-couchdb` 容器端口到主机端口的映射，请确保您了解安全隐患。在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施端口映射，以限制外部对 CouchDB 容器的访问。

您可以使用上述步骤针对 CouchDB 状态数据库使用 **abstore** 链码，但是，为了行使 CouchDB 查询功能，您将需要使用数据建模为 JSON 的链码。如果使用的是 CouchDB 数据库，则已编写示例样本链代码 `marbles02` 来演示可从链代码发出的查询。您可以在目录中找到 `marbles02` 链码 `fabric/examples/chaincode/go`。

我们将按照上述创建和加入频道部分中概述的相同步骤来创建和 加入频道。将同伴加入频道后，请执行以下步骤与 `marbles02` 链码进行交互：

- 在以下位置打包并安装 chaincode `peer0.org1.example.com`：

```
peer lifecycle chaincode package marbles.tar.gz --path github.com/hyperledger/fabric-samples/chaincode/marbles02/go/ --lang golang --label marbles_1  
peer lifecycle chaincode install marbles.tar.gz
```

The install command will **return** a chaincode packageID that you will use to approve a chaincode definition.  
2019-04-08 20:10:32.568 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 001 Installed remotely: response:<status:200 payload:"\nJmarbles\_1:cfb623954827aef3f35868764991cc7571b445a45cf3325f7002f14156d61ae\022\tmarbles\_1" >

2019-04-08 20:10:32.568 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 002 Chaincode code package identifier: marbles\_1:cfb623954827aef3f35868764991cc7571b445a45cf3325f7002f14156d61ae

- 将 packageID 保存为环境变量，以便将其传递给以后的命令：
- `CC_PACKAGE_ID=marbles_1:3a8c52d70c36313cfecbbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173`
- 批准链码定义为 Org1：

```

# be sure to modify the $CHANNEL_NAME variable accordingly for the command

peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name marbles --version 1.0 --
package-id $CC_PACKAGE_ID --sequence 1 --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

```

- 在以下位置安装 chaincode `peer0.org2.example.com`：

```

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:9051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizati
ons/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
peer lifecycle chaincode install marbles.tar.gz

```

- 批准链码定义为 Org2，然后将该定义提交到通道：

```
# be sure to modify the $CHANNEL_NAME variable accordingly for the command
```

```

peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name marbles --version 1.0 --
package-id $CC_PACKAGE_ID --sequence 1 --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
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID $CHANNEL_NAME --name marbles
--version 1.0 --sequence 1 --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 --peerAddresses
peer0.org1.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p
eer0.org2.example.com/tls/ca.crt

```

- 现在，我们可以创建一些弹珠。链码的第一次调用将启动链码容器。您可能需要等待容器启动。

```
# 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 --
peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p
eer0.org2.example.com/tls/ca.crt -c '{"Args": ["initMarble", "marble1", "blue", "35", "tom"]}'

```

容器启动后，您可以发出其他命令来创建一些弹珠并移动它们：

```
# 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 --
peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p
eer0.org2.example.com/tls/ca.crt -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 --
peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p
eer0.org2.example.com/tls/ca.crt -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 --
peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p
eer0.org2.example.com/tls/ca.crt -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 --  
peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles  
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p  
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles  
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p  
eer0.org2.example.com/tls/ca.crt -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 --  
peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles  
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p  
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles  
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/p  
eer0.org2.example.com/tls/ca.crt -c '{"Args": ["delete", "marble1"]}'
```

- 如果选择在 docker-compose 中映射 CouchDB 端口，则现在可以通过打开浏览器并导航至以下 URL，通过 CouchDB Web 界面 (Fauxton) 查看状态数据库：

[http://localhost:5984/\\_utils](http://localhost:5984/_utils)

你应该看到一个名为 `mychannel` (或您唯一的频道名称) 的数据库以及其中的文档。

注意

对于以下命令，请确保适当地更新 `$CHANNEL_NAME` 变量。

您可以从 CLI 运行常规查询 (例如阅读 `marble2`)：

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args": ["readMarble", "marble2"]}'
```

输出应显示以下内容的详细信息 `marble2`：

```
Query Result: {"color": "red", "docType": "marble", "name": "marble2", "owner": "jerry", "size": 50}
```

您可以检索特定 Marbles 的历史记录-例如 `marble1`：

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args": ["getHistoryForMarble", "marble1"]}'
```

输出应显示以下交易 `marble1`：

```
Query Result: [{"TxId": "1c3d3caf124c89f91a4c0f353723ac736c58155325f02890adebaa15e16e6464",  
"Value": {"docType": "marble", "name": "marble1", "color": "blue", "size": 35, "owner": "tom"}}, {"TxId": "755d5  
5c281889eaebf405586f9e25d71d36eb3d35420af833a20a2f53a3eefdf",  
"Value": {"docType": "marble", "name": "marble1", "color": "blue", "size": 35, "owner": "jerry"}}, {"TxId": "819  
451032d813dde6247f85e56a89262555e04f14788ee33e28b232eef36d98f", "Value": {}}]
```

您还可以对数据内容执行丰富的查询，例如按所有者查询 Marbles 字段 `jerry`：

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args": ["queryMarblesByOwner", "jerry"]}'
```

输出应显示以下所有的两个弹珠 `jerry`：

```
Query Result: [{"Key": "marble2", "Record": {"color": "red", "docType": "marble", "name": "marble2", "owner": "jerry", "size": 50}}, {"Key": "marb  
le3", "Record": {"color": "blue", "docType": "marble", "name": "marble3", "owner": "jerry", "size": 70}}]
```

## 4.9 为什么选择 CouchDB

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/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，并且完全可查询的。因此，CouchDB 可以满足 LevelDB 不支持的许多用例的链码，审计，报告要求。

CouchDB 还可以增强区块链中合规性和数据保护的安全性。因为它能够通过过滤和屏蔽事务中的各个属性来实现字段级安全性，并且仅在需要时才授权只读权限。

另外，CouchDB 属于 CAP 定理的 AP 类型 (可用性和分区容差)。它使用带有的主-主复制模型。在 [CouchDB 文档](#)的“[最终一致性](#)”页面上可以找到更多信息。但是，在每个结构同位体下，没有数据库副本，可以保证对数据库的写入是一致且持久的（不是）。 [Eventual Consistency](#)

CouchDB 是第一个用于 Fabric 的外部可插入状态数据库，并且可能并且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。并且可能还需要 CP 类型 (一致性和分区容差) 数据库，以便在不保证应用程序级别的情况下实现数据一致性。

## 4.10 关于数据持久性的说明

如果需要在对等容器或 CouchDB 容器上保持数据持久性，一种选择是将 docker-host 中的目录挂载到容器中的相关目录中。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
```

对于 CouchDB 容器，您可以在 CouchDB 容器规范中添加以下两行：

```
volumes:  
- /var/hyperledger/couchdb0:/opt/couchdb/data
```

## 4.11 故障排除

- 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
```

注意

你会看到错误，如果你不删除旧容器和图像。

- 如果您看到 Docker 错误，请首先检查您的 Docker 版本 (Prerequisites)，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。

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)
```

- 如果在创建，批准，提交，调用或查询命令时看到错误，请确保已正确更新了通道名称和链码名称。提供的示例命令中包含占位符值。
- 如果看到以下错误：

`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)`

您可能具有先前运行的链码图像（例如 `dev-peer1.org2.example.com-mycc-1.0` 或 `dev-peer0.org1.example.com-mycc-1.0`）。删除它们，然后重试。

```
docker rmi -f $(docker images | grep peer[0-9]-peer[0-9] | awk '{print $3}')
```

- 如果您看到类似以下内容的内容：
  - `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`
- 确保针对重新标记为“最新”的“1.0.0”图像运行网络。
- 如果看到以下错误：

`[configtx/tool/localconfig] Load -> CRIT 002 Error reading configuration: Unsupported Config Type ""`

`panic: Error reading configuration: Unsupported Config Type ""`  
然后，您没有 `FABRIC_CFG_PATH` 正确设置环境变量。configtxgen 工具需要此变量才能找到 configtx.yaml。返回并执行一个，然后重新创建您的通道工件。`export FABRIC_CFG_PATH=$PWD`

- 要清理网络，请使用以下 `down` 选项：
- `./byfn.sh down`
- 如果看到错误消息指出您仍然具有“活动端点”，请修剪 Docker 网络。这将清除以前的网络，并为您提供一个全新的环境：

`docker network prune`

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]
```

选择 `y`。

- 如果看到类似于以下内容的错误：

- `/bin/bash: ./scripts/script.sh: /bin/bash^M: bad interpreter: No such file or directory`

确保有问题的文件（在本示例中为 `script.sh`）以 Unix 格式编码。这很可能是由于未在 Git 配置中将设置 `core.autocrlf` 为造成的 `false`（请参见 Windows extras）。有几种解决方法。例如，如果您有权访问 vim 编辑器，请打开文件：

```
vim ./fabric-samples/first-network/scripts/script.sh
```

然后通过执行以下 vim 命令来更改其格式：

```
:set ff=unix
```

注意

如果仍然看到错误，请在 [Hyperledger Rocket Chat](#) 或 [StackOverflow](#) 的结构问题通道上共享日志。

## 5 将组织添加到频道

注意

确保您已下载符合本文档版本的安装样本，二进制文件和 Docker 映像和先决条件中概述的适当的映像和二进制文件（可在左侧目录的底部找到）。特别是，您的 `fabric-samples` 文件夹版本必须包含 `eyfn.sh`（“扩展您的第一个网络”）脚本及其相关脚本。

本教程是对构建您的第一个网络（BYFN）教程的扩展，并将演示向 BYFN 自动生成 `Org3` 的应用程序通道（`mychannel`）添加新的组织。它假定您对 BYFN 有深入的了解，包括上述实用程序的用法和功能。

尽管我们在此仅专注于新组织的集成，但是在执行其他渠道配置更新（例如，更新修改策略或更改批次大小）时，可以采用相同的方法。要总体了解有关频道配置更新的过程和可能性的更多信息，请查看“[更新频道配置](#)”。还值得注意的是，通道配置更新（如此处演示的更新）通常是组织管理员（而不是链码或应用程序开发人员）的责任。

注意

`byfn.sh` 在继续之前，请确保自动脚本在计算机上运行没有错误。如果您已将二进制文件和相关工具（`cryptogen`，`configtxgen` 等）导出到 PATH 变量中，则可以在不传递完全限定路径的情况下相应地修改命令。

### 5.1 设置环境

我们将从 `first-network` 您本地克隆的子目录的根目录中进行操作 `fabric-samples`。现在切换到该目录。您还需要打开一些额外的终端以方便使用。

首先，使用 `byfn.sh` 脚本进行整理。该命令将杀死所有活动或陈旧的 Docker 容器并删除以前生成的工件。它绝不是必要，以便执行信道配置更新任务打倒一个织物网。但是，出于本教程的考虑，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
```

现在生成默认的 BYFN 构件：

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

并使用 CLI 容器中的脚本执行来启动网络：

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

现在，您的计算机上已运行了干净的 BYFN 版本，您可以采用两种不同的方法。首先，我们提供一个带有完整注释的脚本，该脚本将执行配置事务更新以将 Org3 带入网络。

另外，我们将显示同一过程的“手动”版本，显示每个步骤并说明完成的步骤（由于我们向您展示了如何在此手动过程之前关闭网络，因此您还可以运行脚本，然后查看每个步骤步）。

## 5.2 使用脚本将 Org3 带入频道

您应该在 `first-network`。要使用该脚本，只需发出以下命令：

```
./eyfn.sh up
```

这里的输出非常值得一读。您将看到添加了 Org3 加密材料，正在创建和签名配置更新，然后安装了链码以允许 Org3 执行分类账查询。

如果一切顺利，您将收到以下消息：

```
===== All GOOD, EYFN test execution completed =====
```

`eyfn.sh` 可以与 `byfn.sh` 发出以下命令（而不是）一起使用相同的 Node.js 链码和数据库选项：`./byfn.sh up`

```
./byfn.sh up -c testchannel -s couchdb -l node
```

然后：

```
./eyfn.sh up -c testchannel -s couchdb -l node
```

对于那些想更仔细地了解此过程的人，文档的其余部分将向您展示用于进行频道更新的每个命令及其作用。

## 5.3 手动将 Org3 放入频道

注意

下面概述的手动步骤假定 `FABRIC_LOGGING_SPEC` 在 `cli` 和 `Org3cli` 容器设置为 `DEBUG`。

对于 `cli` 容器，可以通过修改目录中的 `docker-compose-cli.yaml` 文件来进行设置 `first-network`。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
```

对于 `Org3cli` 容器，可以通过修改目录中的 `docker-compose-org3.yaml` 文件来进行设置 `first-network`。

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
```

如果使用了 `eyfn.sh` 脚本，则需要关闭网络。可以通过发出以下命令来完成：

```
./eyfn.sh down
```

这将关闭网络，删除所有容器，并撤消添加 Org3 的操作。

当网络断开时，再次将其恢复。

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

然后：

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

这将使您的网络恢复到执行 `eyfn.sh` 脚本之前的状态。

现在，我们准备手动添加 Org3。第一步，我们需要生成 Org3 的加密材料。

## 5.4 生成 Org3 加密材料

在另一个终端中，`org3-artifacts` 从进入子目录 `first-network`。

```
cd org3-artifacts
```

这里有两个 `yaml` 有趣的文件: `org3-crypto.yaml` 和 `configtx.yaml`。首先, 为 Org3 生成加密材料:  
`../../bin/cryptogen generate --config=./org3-crypto.yaml`  
此命令读取我们的新加密 `yaml` 文件 `org3-crypto.yaml` 并利用它 `cryptogen` 来生成 Org3 CA 以及绑定到此新 Org 的两个对等方的密钥和证书。与 BYFN 实施一样, 此加密材料将放入 `crypto-config` 当前工作目录 (在我们的情况下为 `org3-artifacts`) 中新生成的文件夹中。

现在, 使用该 `configtxgen` 实用程序以 JSON 打印出 Org3 特定的配置材料。我们将通过告诉该工具在当前目录中查找 `configtx.yaml` 需要提取的文件的方式来开始命令。

```
export FABRIC_CFG_PATH=$PWD && ../../bin/configtxgen -printOrg Org3MSP > ../channel-artifacts/org3.json
```

上面的命令创建一个 JSON 文件 `org3.json` 并将其输出到 `channel-artifacts` 根目录下的子目录中 `first-network`。该文件包含 Org3 的策略定义以及以 base 64 格式显示的三个重要证书: admin 用户证书 (以后将需要用作 Org3 的管理员), CA 根证书和 TLS 根证书 在接下来的步骤中, 我们会将这个 JSON 文件附加到通道配置中。

我们最后的整理工作是将 Orderer Org 的 MSP 材料移植到 Org3 `crypto-config` 目录中。特别是, 我们关注排序节点的 TLS 根证书, 该证书将允许 Org3 实体与网络排序节点之间的安全通信。

```
cd .. && cp -r crypto-config/ordererOrganizations org3-artifacts/crypto-config/
```

现在我们准备更新频道配置...

## 5.5 准备 CLI 环境

更新过程使用配置转换工具- `configtxlator`。该工具提供了独立于 SDK 的无状态 REST API。此外, 它提供了 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形式/格式之间轻松转换 (在这种情况下, 在 protobuf 和 JSON 之间)。此外, 该工具可以基于两个通道配置之间的差异来计算配置更新事务。

首先, 执行到 CLI 容器中。回想一下, 该容器已经安装了 BYFN `crypto-config` 库, 使我们可以访问两个原始对等组织和 Orderer Org 的 MSP 材料。自举身份是 Org1 管理员用户, 这意味着我们要用作 Org2 的任何步骤都需要导出 MSP 特定的环境变量。

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

导出 `ORDERER_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
```

检查并确保已正确设置变量:

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

注意

如果出于任何原因需要重新启动 CLI 容器, 则还需要重新导出两个环境变量- `ORDERER_CA` 和 `CHANNEL_NAME`。

## 5.6 获取配置

现在, 我们有了一个带有两个关键环境变量的 CLI 容器- `ORDERER_CA` 并已 `CHANNEL_NAME` 导出。让我们获取通道-的最新配置块 `mychannel`。

之所以必须提取最新版本的配置, 是因为通道配置元素已版本化。版本控制很重要, 原因有几个。它可以防止重复或重放配置更改 (例如, 使用旧的 CRL 恢复到通道配置会带来安全风险)。它还有助于确保并发性 (例如, 如果要从通道中删除组织, 例如, 在添加了一个新组织之后, 版本控制将有助于防止您同时删除这两个组织, 而不是仅删除要删除的组织)。

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CHANNEL_NAME --tls --cafile $ORDERER_CA
```

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, 建议遵循一个约定, 该约定既标识要表示的对象的类型, 又标识其编码 (protobuf 或 JSON)。

发出命令时，终端中有相当数量的输出。日志中的最后一行很有趣： `peer channel fetch`

```
2017-11-07 17:17:57.383 UTC [channelCmd] readBlock -> DEBU 011 Received block: 2
```

这告诉我们，最新的配置块 `mychannel` 实际上是块 2，而不是创世块。默认情况下，该命令返回目标通道的最新配置块，在这种情况下为第三个块。这是因为 BYFN 脚本定义锚同行对我们的两个组织- 和-在两个独立的频道更新的交易。 `peer channel fetch configOrg10rg2`

结果，我们具有以下配置顺序：

- 块 0：创世块
- 块 1：Org1 锚点对等更新
- 块 2：Org2 锚点对等更新

## 5.7 将配置转换为 JSON 并进行修整

现在，我们将使用该 `configtxlator` 工具将该通道配置块解码为 JSON 格式（人类可以读取和修改）。我们还必须删除所有与我们要进行的更改无关的标题，元数据，创建者签名等。我们通过 `jq` 工具来完成此任务：

```
configtxlator proto_decode --input config_block.pb --type common.Block | jq .data.data[0].payload.data.config > config.json
```

这为我们提供了一个经过精简的 JSON 对象- `config.json` 位于 `fabric-samples` 内部文件夹中 `first-network`-将作为配置更新的基准。

花一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。即使您已完成本教程的学习，也值得对其进行研究，因为它揭示了潜在的配置结构以及可以进行的其他类型的通道更新。我们将在“更新通道配置”中更详细地讨论它们。

## 5.8 添加 Org3 加密材料

注意

到目前为止，无论您要进行哪种配置更新，您都将执行几乎相同的步骤。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频道配置更新之一。

我们将 `jq` 再次使用该工具将 Org3 配置定义- `org3.json`- 附加到通道的应用程序组字段中，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
```

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`。初始文件仅包含 Org1 和 Org2 资料，而“修改”文件包含所有三个 Org。此时，只需重新编码这两个 JSON 文件并计算增量即可。

首先，将其翻译 `config.json` 回 protobuf `config.pb`：

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

接下来，编码 `modified_config.json` 为 `modified_config.pb`：

```
configtxlator proto_encode --input modified_config.json --type common.Config --output modified_config.pb
```

现在用于 `configtxlator` 计算这两个配置协议之间的差异。此命令将输出一个新的 protobuf 二进制文件，名为 `org3_update.pb`：

```
configtxlator compute_update --channel_id $CHANNEL_NAME --original config.pb --updated modified_config.pb --output org3_update.pb
```

这个新的原型- `org3_update.pb` 包含 Org3 定义以及 Org1 和 Org2 材料的高级指针。我们可以放弃针对 Org1 和 Org2 的大量 MSP 材料和修改策略信息，因为该数据已存在于频道的创始区块中。这样，我们只需要两个配置之间的增量。

在提交频道更新之前，我们需要执行一些最后的步骤。首先，让我们将该对象解码为可编辑的 JSON 格式，并

将其称为 `org3_update.json`:

```
configtxlator proto_decode --input org3_update.pb --type common.ConfigUpdate | jq . > org3_update.json
```

现在，我们有一个解码后的更新文件 `org3_update.json`，我们需要将其包装在信封消息中。此步骤将使我们返回之前删除的标头字段。我们将这个文件命名为 `org3_update_in_envelope.json`:

```
echo '{"payload": {"header": {"channel_header": {"channel_id": "'$CHANNEL_NAME'", "type": 2}}, "data": {"config_update": "$(cat org3_update.json)"} }}' | jq . > org3_update_in_envelope.json
```

使用我们正确格式的 JSON – `org3_update_in_envelope.json` 我们将 `configtxlator` 最后一次使用该工具，并将其转换为 Fabric 所需的完整 protobuf 格式。我们将命名我们的最终更新对象 `org3_update_in_envelope.pb`:

```
configtxlator proto_encode --input org3_update_in_envelope.json --type common.Envelope --output org3_update_in_envelope.pb
```

## 5.9 签名并提交配置更新

快完成了！

现在 `org3_update_in_envelope.pb`，我们的 CLI 容器中有一个 protobuf 二进制文件。但是，在将配置写入账本之前，我们需要来自必需的 Admin 用户的签名。我们的频道应用程序组的修改策略 (mod\_policy) 设置为默认值 “MAJORITY”，这意味着我们需要大多数现有的组织管理员来对其进行签名。因为我们只有两个组织–Org1 和 Org2–且大多数是两个，所以我们都需要它们进行签名。没有这两个签名，排序服务将因未能履行政策而拒绝交易。

首先，让我们将此更新协议签名名为 Org1 Admin。请记住，CLI 容器是由 Org1 MSP 材料引导的，因此我们只需要发出以下命令:

```
peer channel signconfigtx
```

```
peer channel signconfigtx -f org3_update_in_envelope.pb
```

最后一步是切换 CLI 容器的身份以反映 Org2 Admin 用户。为此，我们导出了四个特定于 Org2 MSP 的环境变量。

注意

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 Fabric 操作。单个容器永远不会安装整个网络的加密材料。而是需要将配置更新安全地带外传递给 Org2 管理员进行检查和批准。

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:9051
```

最后，我们将发出命令。Org2 管理员签名将附加到此调用中，因此无需再次手动签署 Protobuf:

```
peer channel update
```

注意

即将到来的排序服务更新电话将进行一系列系统的签名和策略检查。这样，您可能会发现流式传输和检查排序节点的日志很有用。在另一个外壳上，发出命令以显示它们。`docker logs -f orderer.example.com`

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
```

如果您的更新已成功提交，则应该看到类似于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
```

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
```

成功的通道更新调用将新的块（块 5）返回给通道上的所有对等方。记住，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。这样，模块 5 用作 Org3 现在通道上已定义的最新通道配置。

检查日志 `peer0.org1.example.com`：

```
docker logs -f peer0.org1.example.com
```

如果要检查其内容，请按照演示的过程来获取和解码新的配置块。

## 5.10 配置领导者选举

### 注意

包含本部分作为一般参考，以了解在初始渠道配置完成后向网络添加组织时的领导者选举设置。此示例默认为动态领导者选举，该选举在 `peer-base.yaml` 中为网络中的所有对等体设置。

新加入的对等方使用创世块引导，该创世纪块不包含有关在通道配置更新中添加的组织的信息。因此，新对等方无法利用八卦，因为它们无法验证其他对等方从其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，新添加的对等方必须具有以下配置之一，以便它们从排序服务接收块：

1. 要使用静态领导者模式，请将对等方配置为组织负责人：

```
CORE_PEER_GOSSIP_USELEADERELECTION=false  
CORE_PEER_GOSSIP_ORGLEADER=true
```

### 注意

对于添加到通道的所有新对等方，此配置必须相同。

2. 要使用动态领导者选举，请将对等体配置为使用领导者选举：

```
CORE_PEER_GOSSIP_USELEADERELECTION=true  
CORE_PEER_GOSSIP_ORGLEADER=false
```

### 注意

由于新加入的组织的对等方将无法形成成员资格视图，因此此选项将类似于静态配置，因为每个对等方都将开始宣称自己为领导者。但是，一旦他们使用将组织添加到渠道的配置事务进行更新，组织将只有一位活跃的领导者。因此，如果您最终希望组织的同行使用领导者选举，则建议利用此选项。

## 5.11 加入 Org3 到频道

至此，渠道配置已更新为包括我们的新组织 `Org3` -意味着与之相连的对等方现在可以加入 `mychannel`。

首先，让我们启动 Org3 对等方和特定于 Org3 的 CLI 的容器。

打开一个新终端，然后从 `first-network` Org3Docker 开始撰写：

```
docker-compose -f docker-compose-org3.yaml up -d
```

这个新的撰写文件已配置为跨我们的初始网络桥接，因此两个对等方和 CLI 容器将能够与现有对等方和排序节点进行解析。随着这三个新容器的运行，请执行到 Org3 特定的 CLI 容器中：

```
docker exec -it Org3cli bash
```

就像我们使用初始 CLI 容器一样，导出两个关键环境变量：`ORDERER_CA` 和 `CHANNEL_NAME`：

```
export  
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.co  
m/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem && export  
CHANNEL_NAME=mychannel
```

检查并确保已正确设置变量：

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

现在，让我们向排序服务发送呼叫，询问的生成块 `mychannel`。由于我们成功地更新了频道，因此排序服务能够验证此呼叫所附加的 Org3 签名。如果 Org3 尚未成功附加到通道配置，则排序服务应拒绝此请求。

### 注意

同样，您可能会发现，流送排序节点的日志以显示签名/验证逻辑和策略检查很有用。

使用命令检索此块：`peer channel fetch`

```
peer channel fetch 0 mychannel.block -o orderer.example.com:7050 -c $CHANNEL_NAME --tls --cafile  
$ORDERER_CA
```

注意，我们传递了 a **0** 来表示我们想要通道分类账上的第一个块（即创世块）。如果我们简单地传递命令，那么我们将收到块 5 – 已定义 Org3 的更新配置。但是，我们不能从下游区块开始分类帐–我们必须从区块 0 开始。

```
peer channel fetch config
```

发出命令，并通过在成因块-：  
**peer channel join mychannel.block**

如果要加入 Org3 的第二个对等方，请导出 **TLS** 和 **ADDRESS** 变量，然后重新发出：  
**peer channel join command**

```
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.crt && export  
CORE_PEER_ADDRESS=peer1.org3.example.com:12051
```

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

## 5.12 安装、定义和调用链码

加入频道后，您可以打包链码并将其安装在 Org3 的对等节点上。然后，您需要批准链码定义为 org3。由于链码定义已经提交给您加入的频道，因此您可以在批准定义后开始使用链码。

注意

这些说明使用 v2.0 Alpha 版本中引入的 Fabric 链码生命周期。如果您想使用以前的生命周期来安装和实例化链码，请访问 v1.4 版本的“[将组织添加到频道教程](#)”。

第一步是从 Org3 CLI 打包链码：

```
peer lifecycle chaincode package mycc.tar.gz --path github.com/hyperledger/fabric-samples/chaincode/abstore/go/ --lang golang --label mycc_1
```

此命令将创建一个名为的链码包 **mycc.tar.gz**，我们可以使用它在对等方上安装链码。在此命令中，您需要提供一个链码包标签作为链码的描述。如果通道正在运行用 Java 或 Node.js 编写的链码，请相应地修改命令。发出以下命令以将软件包安装在 Org3 的 peer0 上：

```
# this command installs a chaincode package on your peer  
peer lifecycle chaincode install mycc.tar.gz
```

如果要在 Org3 的第二个对等节点上安装链码，还可以修改环境变量并重新发出命令。请注意，第二次安装不是强制性的，因为您只需要在要用作背书人或与账本进行交互的对等节点上安装链码（即仅用于查询）。在没有运行 chaincode 容器的情况下，对等方仍将运行验证逻辑并充当提交方。

下一步是批准 **mycc** as Org3 的链码定义。Org3 需要批准与 Org1 和 Org2 批准并提交给渠道的定义相同的定义。链码定义还需要包括链码包标识符。您可以通过查询对等方来找到包标识符：

```
# this returns the details of the packages installed on your peers  
peer lifecycle chaincode queryinstalled
```

您应该看到类似于以下内容的输出：

```
Get installed chaincodes on peer:  
Package ID: mycc_1:3a8c52d70c36313cfecbbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173, Label: mycc_1
```

我们将在以后的命令中需要软件包 ID，因此让我们继续将其保存为环境变量。将由安装的对等生命周期链代码查询返回的程序包 ID 粘贴到以下命令中。软件包 ID 对于所有用户而言可能都不相同，因此您需要使用从控制台返回的软件包 ID 来完成此步骤。

```
# Save the package ID as an environment variable.
```

```
CC_PACKAGE_ID=mycc_1:3a8c52d70c36313cfecbbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173
```

使用以下命令批准 **mycc** Org3 的链码定义：

```
# this approves a chaincode definition for your org  
# use the --package-id flag to provide the package identifier  
# use the --init-required flag to request the ``Init`` function be invoked to initialize the  
chaincode  
peer lifecycle chaincode approveformyorg --channelID $CHANNEL_NAME --name mycc --version 1.0 --init-  
required --package-id $CC_PACKAGE_ID --sequence 1 --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 --waitForEvent
```

您可以使用该命令来检查您批准的链码定义是否已经提交给通道。

```
peer lifecycle chaincode querycommitted
```

```
# use the --name flag to select the chaincode whose definition you want to query
peer lifecycle chaincode querycommitted --channelID $CHANNEL_NAME --name mycc --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

成功的命令将返回有关已提交定义的信息：

```
Committed chaincode definition for chaincode 'mycc' on channel 'mychannel':
Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc
```

由于已经提交了链码定义，因此您可以 `mycc` 在批准定义后使用链码。链码定义使用默认的认可策略，该策略要求渠道上的大多数组织认可交易。这意味着，如果将组织添加到渠道中或从渠道中删除，则背书策略将自动更新。我们以前需要 Org1 和 Org2 的认可（2 之 2）。现在，我们需要 Org1、Org2 和 Org3 两个组织（3 个中的 2 个）的认可。

查询链码以确保其已启动。请注意，您可能需要等待 chaincode 容器启动。

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

我们应该看到的回应。`Query Result: 90`

现在发出 `10` 从 `a` 移至的调用 `b`。在下面的命令中，我们以 Org1 和 Org3 中的对等对象为目标，以收集足够数量的认可。

```
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"]}' --peerAddresses
peer0.org1.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/p
eer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org3.example.com:11051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org3.example.com/peers/p
eer0.org3.example.com/tls/ca.crt
```

最后查询一次：

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

我们应该看到的响应，准确地反映了该链码世界状态的更新。`Query Result: 80`

## 5.13 结论

通道配置更新过程确实涉及很多，但是各个步骤都有一个逻辑方法。最终的结果是形成一个以 protobuf 二进制格式表示的增量交易对象，然后获取必要数量的管理员签名，以使频道配置更新交易满足频道的修改策略。

在 `configtxlator` 和 `jq` 工具，与日益增长沿着命令，为我们提供了完成这项任务所需的功能。`peer channel`

## 5.14 更新通道配置以包括 Org3 锚点对等体（可选）

由于 Org1 和 Org2 在通道配置中定义了锚定对等方，因此 Org3 对等方能够建立与 Org1 和 Org2 对等方的八卦连接。同样，新添加的组织（如 Org3）也应在通道配置中定义其锚点对等点，以便来自其他组织的任何新对等点都可以直接发现 Org3 对等点。

从 Org3 CLI 继续，我们将进行通道配置更新以定义 Org3 锚点对等体。该过程将类似于先前的配置更新，因此这次我们会更快。

和以前一样，我们将获取最新的通道配置以开始使用。在 Org3 的 CLI 容器中，使用以下命令获取通道的最新配置块。`peer channel fetch`

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CHANNEL_NAME --tls --
cafile $ORDERER_CA
```

提取配置块后，我们将其转换为 JSON 格式。为此，我们将使用 `configtxlator` 工具，就像之前将 Org3 添加到通道中一样。在转换它时，我们需要使用 `jq` 工具删除更新 Org3 以包括锚点对等点所需的所有标头，元数据和签名。在我们继续更新通道配置之前，稍后将重新合并此信息。

```
configtxlator proto_decode --input config_block.pb --type common.Block |
jq .data.data[0].payload.data.config > config.json
```

现在 `config.json` 是经过裁剪的 JSON，代表我们将更新的最新通道配置。

再次使用 jq 工具，我们将使用我们要添加的 Org3 锚点对等更新配置 JSON。

```
jq '.channel_group.groups.Application.groups.Org3MSP.values += {"AnchorPeers":{"mod_policy": "Admins","value":[{"anchor_peers": [{"host": "peer0.org3.example.com","port": 11051}]}],"version": "0"}}' config.json > modified_anchor_config.json
```

现在，我们有两个 JSON 文件，一个用于当前通道配置， config.json 一个用于所需通道配置 modified\_anchor\_config.json。接下来，我们将这些转换回 protobuf 格式，并计算两者之间的差异。

转换 config.json 为 protobuf 格式为 config.pb

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

转换 modified\_anchor\_config.json 成 protobuf 格式为 modified\_anchor\_config.pb

```
configtxlator proto_encode --input modified_anchor_config.json --type common.Config --output modified_anchor_config.pb
```

计算两个 protobuf 格式的配置之间的差异。

```
configtxlator compute_update --channel_id $CHANNEL_NAME --original config.pb --updated modified_anchor_config.pb --output anchor_update.pb
```

现在，我们已经有了所需的频道更新，我们必须将其包装在信封消息中，以便可以正确读取它。为此，我们必须首先将 protobuf 转换回可以包装的 JSON。

我们将再次使用 configtxlator 命令将其转换 anchor\_update.pb 为 anchor\_update.json

```
configtxlator proto_decode --input anchor_update.pb --type common.ConfigUpdate | jq . > anchor_update.json
```

接下来，我们将更新内容封装在信封消息中，还原先前剥离的标头，然后将其输出到 anchor\_update\_in\_envelope.json

```
echo '{"payload": {"header": {"channel_header": {"channel_id": "'$CHANNEL_NAME'", "type": 2}}, "data": {"config_update": "'$(cat anchor_update.json)'}}}' | jq . > anchor_update_in_envelope.json
```

现在，我们已经重新合并了信封，我们需要将其转换为 protobuf，以便可以对其进行正确签名并提交给排序节点以进行更新。

```
configtxlator proto_encode --input anchor_update_in_envelope.json --type common.Envelope --output anchor_update_in_envelope.pb
```

现在，更新已正确格式化，是时候注销并提交更新了。由于这只是对 Org3 的更新，因此我们只需要在更新上注销 Org3。由于我们位于 Org3 CLI 容器中，因此无需切换 CLI 容器标识，因为它已经在使用 Org3 标识。因此，我们只能使用该命令，因为在将更新提交给排序节点之前，它还将以 Org3 管理员的身份注销。 peer channel update

```
peer channel update -f anchor_update_in_envelope.pb -c $CHANNEL_NAME -o orderer.example.com:7050 --tls --cafile $ORDERER_CA
```

排序节点收到配置更新请求，并使用更新后的配置切割一个块。当对等方收到该阻止时，他们将处理配置更新。

检查对等方之一的日志。在处理来自新块的配置事务时，您会看到八卦使用 Org3 的新锚点重新建立连接。这证明配置更新已成功应用！

```
docker logs -f peer0.org1.example.com
2019-06-12 17:08:57.924 UTC [gossip.gossip] learnAnchorPeers -> INFO 89a Learning about the configured anchor peers of Org1MSP for channel mychannel : [{peer0.org1.example.com 7051}]
2019-06-12 17:08:57.926 UTC [gossip.gossip] learnAnchorPeers -> INFO 89b Learning about the configured anchor peers of Org2MSP for channel mychannel : [{peer0.org2.example.com 9051}]
2019-06-12 17:08:57.926 UTC [gossip.gossip] learnAnchorPeers -> INFO 89c Learning about the configured anchor peers of Org3MSP for channel mychannel : [{peer0.org3.example.com 11051}]
```

恭喜，您现在进行了两次配置更新-一次将 Org3 添加到通道，第二次为 Org3 定义锚点。

# 6 升级网络组件

## 注意

在本文档中使用“升级”一词时，主要是指更改组件的版本（例如，从 v1.3 二进制版本更改为 v1.4.x 二进制版本）。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。从技术上讲，由于没有数据迁移，因此在 Fabric 中，在此我们将不使用术语“迁移”或“迁移”。

## 注意

不支持从 Fabric v1.4 升级到 v2.0 Alpha 版本。在 2.0 Alpha 版本之后，本教程将进行更新。

## 6.1 总览

由于“构建第一个网络（BYFN）”教程默认为“最新”二进制文件，因此，如果您是从 v1.4.x 版本开始运行的，则您的计算机将安装 v1.4.x 二进制文件和工具，并且您将无法升级它们。

因此，本教程将提供一个基于 Hyperledger Fabric v1.3 二进制文件以及将要升级到的 v1.4.x 二进制文件的网络。在较高级别，我们的升级教程将执行以下步骤：

1. 备份分类帐和 MSP。
2. 将排序者二进制文件升级到 Fabric v1.4.x，**因为不支持从 Solo 到 Raft 的迁移，并且 Fabric 1.4.1 版本是第一个支持 Raft 的版本，因此本教程将不介绍升级到 Raft 排序服务的过程。**
3. 将对等二进制文件升级到 Fabricv1.4.x。

## 注意

v1.4.x 中没有新的“**定义**”功能要求，因此，作为升级到 v1.4.x 的一部分，我们不必更新任何通道配置。

本教程将演示如何使用 CLI 命令分别执行每个步骤。我们还将描述如何 **tools** 更新 CLI 映像。

## 注意

因为 BYFN 使用“Solo”排序服务（一个排序者），所以我们的脚本使整个网络中断。但是，在生产环境中，排序者和对等方可以同时滚动升级。换句话说，您可以按任何顺序升级二进制文件而无需关闭网络。

由于 BYFN 与以下组件不兼容，因此我们用于升级 BYFN 的脚本不会涵盖它们：

- **Fabric CA**
- **KAFKA**
- **CouchDB**
- **开发包**

本教程后面的部分将介绍升级这些组件的过程（如有必要）。我们还将展示如何升级 Node 链码填充程序。

从操作的角度来看，值得注意的是，v1.4 中日志的收集过程已从 **CORE\_LOGGING\_LEVEL**（对等方）和 **ORDERER\_GENERAL\_LOGLEVEL**（对于排序方）更改为 **FABRIC\_LOGGING\_SPEC**（新操作服务）。有关更多信息，请查看 [Fabric 发行说明](#)。

## 6.1.1 先决条件

如果尚未执行此操作，请确保您具有计算机上的所有依赖项，如前提条件中所述。

## 6.2 启动 v1.3 网络

在升级到 v1.4 之前，我们必须首先配置一个运行 Fabric v1.3 映像的网络。

就像在 BYFN 教程中一样，我们将 `first-network` 在的本地克隆中的子目录中进行操作 `fabric-samples`。现在切换到该目录。您还需要打开一些额外的终端以方便使用。

### 6.2.1 清理

我们希望从已知状态进行操作，因此我们将使用 `byfn.sh` 脚本杀死所有活动或陈旧的 Docker 容器并删除任何先前生成的工件。跑：

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

### 6.2.2 生成加密并启动网络

在干净的环境中，使用以下四个命令启动 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
```

注意

如果您具有本地构建的 v1.3 映像，则示例将使用它们。如果遇到错误，请考虑清理本地生成的 v1.3 映像，然后再次运行该示例。这将从 docker hub 下载 v1.3 映像。

如果 BYFN 已正确启动，您将看到：

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

现在，我们准备将网络升级到 Hyperledger Fabricv1.4.x。

### 6.2.3 获取最新样本

注意

以下说明与 v1.4.x 的最新发布版本有关。请用要测试的已发布发行版的版本标识符替换 1.4.x。换句话说，如果要测试第一个发行版，请用“1.4.0”替换“1.4.x”。

在完成本教程的其余部分之前，获取示例的 v1.4.x（例如 1.4.1）版本很重要，您可以通过发出以下命令来执行此操作：

```
git fetch origin  
git checkout v1.4.x
```

### 6.2.4 要立即升级吗？

我们有一个脚本，该脚本将升级 BYFN 中的所有组件并启用任何功能（请注意，v1.4 不需要任何新功能）。如果您正在运行生产网络，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的升级的模板。

之后，我们将引导您完成脚本中的步骤，并描述升级过程中每段代码的作用。

要运行脚本，请发出以下命令：

```
# Note, replace '1.4.x' with a specific version, for example '1.4.1'.
```

```
# 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
```

如果升级成功，您应该看到以下内容：

```
===== All GOOD, End-2-End UPGRADE Scenario execution completed =====
```

如果要手动升级网络，只需再次运行并执行直至（但不包括）的步骤即可。然后继续下一节。`./byfn.sh down ./byfn.sh upgrade -i 1.4.x`

注意

您将在本节中运行的许多命令不会产生任何输出。通常，假设没有输出是好的输出。

## 6.3 升级排序容器

排序容器应以滚动方式升级（一次升级）。总体而言，排序者升级过程如下：

1. 停止排序者。
2. 备份排序者的分类帐和 MSP。
3. 重新启动排序者以最新的图像。
4. 验证升级完成。

利用 BYFN 的结果是，我们有一个 Solo 排序程序设置，因此，我们将只执行一次此过程。但是，在 Kafka 设置中，必须对每个排序者重复此过程。

注意

本教程使用 docker 部署。对于本机部署，请 `orderer` 使用发布工件中的文件替换该文件。备份 `orderer.yaml` 并将其替换 `orderer.yaml` 为发布工件中的文件。然后将所有修改后的变量从备份移植 `orderer.yaml` 到新变量。使用类似的实用程序 `diff` 可能会有所帮助。

让我们通过降低排序者来开始升级过程：

```
docker stop orderer.example.com
```

```
export LEDGERS_BACKUP=./ledgers-backup
```

```
# Note, replace '1.4.x' with a specific version, for example '1.4.1'.  
# 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
```

我们为目录创建了一个变量，以将文件备份放入其中，并导出了 `IMAGE_TAG` 我们要移至的目录。

下订单后，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
```

在生产网络中，将对每个基于 Kafka 的排序者以滚动方式重复此过程。

现在，使用我们的新结构映像下载并重新启动排序者：

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps orderer.example.com
```

由于我们的示例使用“Solo”排序服务，因此网络中没有其他排序者必须与重新启动的排序者同步。但是，在利用 Kafka 的生产网络中，最好的方法是在重新启动排序程序后发出以验证其是否已赶上其他排序程序。

```
peer channel fetch <blocknumber>
```

## 6.4 升级对等容器

接下来，让我们看一下如何将对等容器升级到 Fabricv1.4.x。对等容器应像排序者一样，以滚动方式升级（一次升级）。如排序者升级期间所述，排序者和对等者可以并行升级，但是出于本教程的目的，我们将流程分开。在较高级别，我们将执行以下步骤：

1. 停止同伴。
2. 备份对等方的分类帐和 MSP。
3. 删除 chaincode 容器和图像。
4. 用最新映像重新启动对等体。
5. 验证升级完成。

我们的网络中有四个对等节点。我们将对每个对等节点执行一次此过程，共进行四次升级。

#### 注意

同样，本教程利用 docker 部署。对于**本机**部署，请 `peer` 使用发布工件中的文件替换该文件。备份您的文件，`core.yaml` 并用发布工件中的文件替换。将所有修改后的变量从备份移植 `core.yaml` 到新变量。使用类似的实用程序 `diff` 可能会有所帮助。

让我们使用以下命令**关闭第一个对等方**：

```
export PEER=peer0.org1.example.com
```

```
docker stop $PEER
```

然后，我们可以**备份对等方的分类帐和 MSP**：

```
mkdir -p $LEDGERS_BACKUP
```

```
docker cp $PEER:/var/hyperledger/production ./$LEDGERS_BACKUP/$PEER
```

在对等方停止并且分类帐已备份的情况下，**删除对等 chaincode 容器**：

```
CC_CONTAINERS=$(docker ps | grep dev-$PEER | awk '{print $1}')  
if [ -n "$CC_CONTAINERS" ] ; then docker rm -f $CC_CONTAINERS ; fi
```

和对等链码图像：

```
CC_IMAGES=$(docker images | grep dev-$PEER | awk '{print $1}')  
if [ -n "$CC_IMAGES" ] ; then docker rmi -f $CC_IMAGES ; fi
```

现在，我们将使用 v1.4.x image 标签重新启动对等方：

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps $PEER
```

#### 注意

尽管 BYFN 支持使用 CouchDB，但在本教程中我们选择了更简单的实现。但是，如果使用的是 CouchDB，请发出以下命令，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
```

#### 注意

您不需要重新启动 chaincode 容器。当对等方收到对链码的请求（调用或查询）时，它首先检查它是否正在运行该链码的副本。如果是这样，它将使用它。否则，在这种情况下，对等方将启动链码（如果需要，则重建映像）。

## 6.4.1 验证对等升级完成

我们已经为第一个对等方完成了升级，但是在继续之前，我们先检查一下以确保通过链码调用正确完成了升级。

#### 注意

在尝试执行此操作之前，您可能需要升级足够组织中的对等方，以满足您的认可策略。虽然，这仅在升级过程中要更新链码时才是必需的。如果您不是在升级过程中更新链码，则可能会获得运行在不同 Fabric 版本上的对等节点的认可。

在进入 CLI 容器并发出调用之前，请通过发出以下命令确保 CLI 更新到最新版本：

```
docker-compose -f docker-compose-cli.yaml stop cli
```

```
docker-compose -f docker-compose-cli.yaml up -d --no-deps cli
```

如果您特别想要 CLI 的 v1.3 版本，请发出：

```
IMAGE_TAG=$(go env GOARCH)-1.3.x docker-compose -f docker-compose-cli.yaml up -d --no-deps cli
```

获得所需的 CLI 版本后，进入 CLI 容器：

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

现在，您需要设置两个环境变量-通道名称和的名称 `ORDERER_CA`：

`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
```

现在您可以发出调用：

```
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:9051 --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"]}'
```

先前的查询显示 `a` 其值为，`90` 而我们刚刚 `10` 通过调用将其删除。因此，针对的查询 `a` 应显示 `80`。让我们来看看：

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

我们应该看到以下内容：

Query Result: `80`

验证对等节点已正确升级之后，请确保 `exit` 在继续升级对等节点之前发出一个离开容器的提示。您可以通过导出不同的对等名称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
```

## 6.5 升级组件 BYFN 不支持

尽管这是我们更新教程的结尾，但是生产网络中存在其他与 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。

### 6.5.1 Fabric CA 容器

要了解如何升级 Fabric CA 服务器，请单击以查看 [CA 文档](#)。

### 6.5.2 升级 Node SDK 客户端

注意

在升级 Node SDK 客户端之前，先升级 Fabric 和 FabricCA。测试 Fabric 和 Fabric CA 与旧版 SDK 客户端的向后兼容性。尽管较新的 SDK 客户端通常可以与较早的 Fabric 和 Fabric CA 版本一起使用，但是它们可能会公开较早的 Fabric 和 Fabric CA 版本中尚不可用的功能，并且未经过全面兼容性测试。

`Node.js` 通过在应用程序的根目录中执行以下命令，使用 NPM 升级任何客户端：

```
npm install fabric-client@latest
```

```
npm install fabric-ca-client@latest
```

这些命令将安装 Fabric 客户端和 Fabric-CA 客户端的新版本，并编写新版本 `package.json`。

### 6.5.3 升级 Kafka 集群

不需要，但是建议升级 Kafka 群集，并使其与 Fabric 的其余部分保持同步。较新版本的 Kafka 支持较旧的协议版本，因此您可以在其余 Fabric 之前或之后升级 Kafka。

如果您遵循了[将网络升级到 v1.3 教程](#)，则您的 Kafka 群集应位于 v1.0.0。如果不是，请参考 Apache Kafka 官方文档，以[从以前的版本](#)升级 Kafka 以升级 Kafka 集群代理。

## 6.5.4 升级 Zookeeper

Apache Kafka 集群需要 Apache Zookeeper 集群。Zookeeper API 长期稳定，因此，Kafka 可以容忍几乎任何版本的 Zookeeper。如果有特定要求升级到特定版本的 Zookeeper，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。

## 6.5.5 升级 CouchDB

如果将 CouchDB 用作状态数据库，则应在升级对等方的同时升级对等方的 CouchDB。CouchDB v2.2.0 已通过 Fabric v1.4.x 进行了测试。

要升级 CouchDB：

1. 停止 CouchDB。
2. 备份 CouchDB 数据目录。
3. 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（Fabric v1.4 随附提供了 CouchDB v2.2.0 预先配置的 Docker 映像）。
4. 重新启动 CouchDB。

## 6.5.6 升级节点链码 SHIM

要移至新版本的 Node 链码填充程序，开发人员需要：

1. 将 `fabric-shim` 其链码的级别 `package.json` 从 1.3 更改为 1.4.x。
2. 重新包装此新的 chaincode 程序包，并将其安装在通道中的所有认可对等方上。
3. 执行对此新链码的升级。要查看如何执行此操作，请查看对等 chaincode。

注意

此流程并非特定于从 1.3 升级到 1.4.x，这也是从任何增量版本的节点结构垫片升级的方式。

## 6.5.7 使用供应商的 SHIM 升级链码

注意

v1.3.0 填充程序与 v1.4.x 对等体兼容，但是，升级链码填充程序以匹配对等体的当前级别仍然是最佳实践。

存在许多第三方工具，这些工具可让您供应链码垫片。如果您使用了其中一种工具，请使用同一工具来更新供应商的链码填充程序并重新打包链码。

如果您的链码供应商提供了垫片，则在更新垫片版本后，您必须将其安装到所有已经拥有链码的对等方。使用相同的名称安装它，但是使用较新的版本。然后，应在已部署此链码的每个通道上执行链码升级，以移至新版本。

# 7 在结构中使用私有数据

本教程将演示如何使用收集到的组织授权的同行提供 blockchain 网络上存储和私有数据的检索。

本教程中的信息假定您掌握私有数据存储及其使用案例。有关更多信息，请查看[私有数据](#)。

这些说明使用 Fabric v2.0 Alpha 版本中引入的新 Fabric 链码生命周期。如果您想使用以前的生命周期模型来通过链码使用私有数据，请访问[在 Fabric 教程中使用私有数据](#)的 v1.4 版本。

本教程将带您完成以下步骤，以练习定义，配置和使用 Fabric 私有数据：

- 构建集合定义 JSON 文件
- 使用链码 API 读取和写入私有数据
- 安装并定义带有集合的链码
- 存储私人数据
- 作为授权对等方查询私有数据
- 查询未经授权的对等数据
- 清除私人数据
- 使用私人数据的索引
- 额外资源

本教程将使用在“构建您的第一个网络（BYFN）”教程网络上运行的 [Marbles 私有数据示例](#) 来演示如何创建，部署和使用私有数据集合。Marbles 私人数据样本将部署到“建立您的第一个网络（BYFN）”教程网络。您应该已经完成安装样本，二进制文件和 Docker 映像任务；但是，运行 BYFN 教程不是本教程的前提条件。而是在整个教程中提供了使用网络所需的命令。我们将描述每个步骤发生的情况，从而可以在不实际运行示例的情况下理解本教程。

## 7.1 构建集合定义 JSON 文件

私有化通道上数据的第一步是建立一个集合定义，该定义定义对私有数据的访问。

集合定义描述了谁可以保留数据，将数据分发给多少对等方，需要多少对等方来传播私有数据以及私有数据在私有数据库中保留的时间长短。稍后，我们将演示如何使用链码 API [PutPrivateData](#) 并将链映射 API [GetPrivateData](#) 用于将集合映射到受保护的私有数据。

集合定义由以下属性组成：

- `name`：集合名称。
- `policy`：定义允许持久保存收集数据的组织对等方。
- `requiredPeerCount`：传播私有数据所需的对等点数目，以作为对链码的认可
- `maxPeerCount`：出于数据冗余的目的，当前认可对等方将尝试向其分发数据的其他对等方的数量。

如果认可对等方发生故障，则在有请求拉私有数据的请求时，这些其他对等方在提交时可用。

• `blockToLive`：对于非常敏感的信息（例如价格或个人信息），此值表示数据应以块为单位在专用数据库上保留的时间。数据将在专用数据库上保留此指定数量的块，然后将被清除，从而使该数据从网络中删除。要无限期保留私有数据，即从不清除私有数据，请将 `blockToLive` 属性设置为 `0`。

- `memberOnlyRead`: 值为, `true` 表示对等方自动强制仅允许属于集合成员组织之一的客户端读取私有数据。

为了说明私有数据的用法, marbles 私有数据示例包含两个私有数据集合定义: `collectionMarbles` 和 `collectionMarblePrivateDetails`。定义中的 `policy` 属性 `collectionMarbles` 允许通道的所有成员 (Org1 和 Org2) 在私有数据库中拥有私有数据。该 `collectionMarblesPrivateDetails` 集合仅允许 Org1 成员在其私有数据库中拥有私有数据。

有关构建策略定义的更多信息, 请参考[认可策略](#)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
  }
]
```

这些策略保护的数据以链码映射, 并将在本教程的后面显示。

当使用[对等生命周期 chaincode commit 命令](#)将链码定义提交到通道时, 将部署此集合定义文件。下文第 3 节提供了有关此过程的更多详细信息。

## 7.2 使用链码 API 读取和写入私有数据

了解如何对通道上的数据进行私有化的下一步是在链码中建立数据定义。Marbles 私有数据样本根据如何访问数据将私有数据分为两个单独的数据定义。

```
// 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"`
}
```

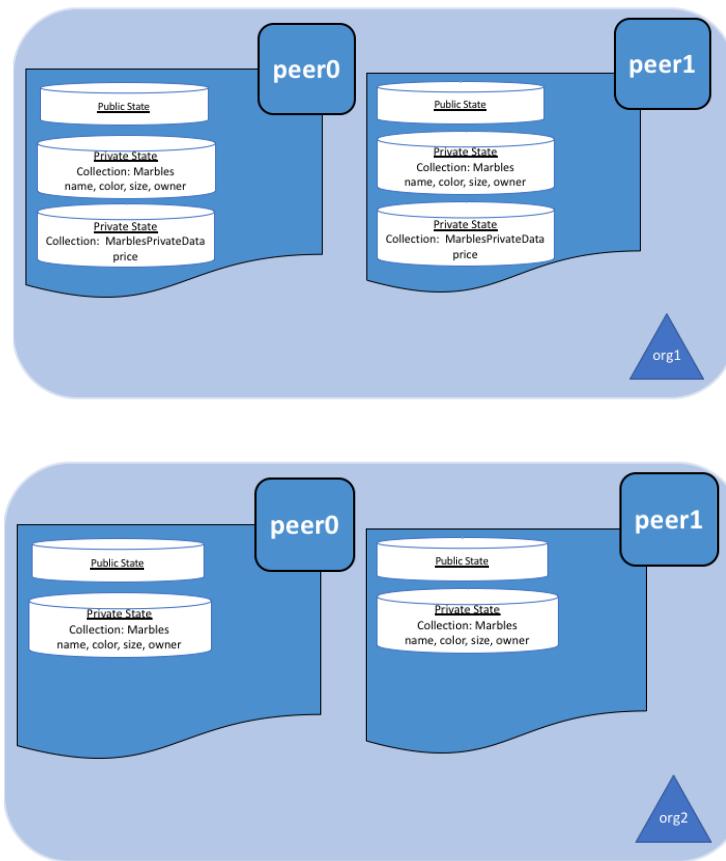
具体地, 对私有数据的访问将受到以下限制:

- `name, color, size, and owner` 将对频道的所有成员 (Org1 和 Org2) 可见
- `price` 仅对 Org1 成员可见

因此, 在 Marbles 私有数据样本中定义了两组不同的私有数据。此数据到限制访问的收集策略的映射由链码 API 控制。具体来说, 通过调用 `GetPrivateData()` 和执行使用集合定义读取和写入私有数据 `PutPrivateData()`, 可

以在此处找到。

下图说明了 Marbles 私有数据样本使用的私有数据模型。



### 7.2.1 读取收集数据

使用 chaincode API `GetPrivateData()` 查询数据库中的私有数据。`GetPrivateData()` 接受两个参数，**集合名称** 和数据键。回想一下，集合 `collectionMarbles` 允许 Org1 和 Org2 的成员将私有数据保存在边数据库中，而集合 `collectionMarblePrivateDetails` 仅允许 Org1 的成员将私有数据保存在边数据库中。有关实现的详细信息，请参考以下两个 Marbles 私有数据函数：

- **readMarble** 用于查询属性的值 `name, color, size and owner`
- **readMarblePrivateDetails** 用于查询 `price` 属性的值

在本教程后面的内容中使用对等命令发出数据库查询时，我们将调用这两个函数。

### 7.2.2 写入私人数据

使用 chaincode API `PutPrivateData()` 将私有数据存储到私有数据库中。该 API 还需要集合的名称。由于 Marbles 私有数据样本包含两个不同的集合，因此在链码中被称为两次：

1. 使用名为的集合写入私有数据。`name, color, size and owner` `collectionMarbles`
2. `price` 使用名为的集合 写入私有数据 `collectionMarblePrivateDetails`。

例如，在以下 `initMarble` 函数片段中，`PutPrivateData()` 两次调用，对于每组私有数据一次。

```
// === 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())
}
```

总而言之，上面针对我们的策略定义 `collection.json` 允许 Org1 和 Org2 中的所有对等方在其私有数据库中存储和处理 Marbles 私有数据。但是，只有 Org1 中的对等方可以在其私有数据库中存储和处理私有数据。

`name, color, size, ownerprice`

作为附加的数据隐私优势，由于使用了集合，因此只有私有数据散列会通过排序者，而不是私有数据本身，从而使私有数据免受排序者的机密。

## 7.3 启动网络

现在，我们准备逐步执行一些命令，这些命令演示了如何使用私有数据。

自己尝试

在安装，定义和使用下面的 Marbles 专用数据链代码之前，我们需要启动 BYFN 网络。出于本教程的考虑，我们希望在已知的初始状态下进行操作。以下命令将杀死所有活动或陈旧的 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
```

如果您已经完成了本教程，则还需要删除 Marbles 专用数据链代码的基础 docker 容器。让我们运行以下命令来清理以前的环境：

```
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}')
```

通过运行以下命令，使用 CouchDB 启动 BYFN 网络：

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

这将创建一个简单的 Fabric 网络，该网络由 `mychannel` 使用两个组织（每个组织维护两个对等节点）命名的单个通道和一个排序服务（使用 CouchDB 作为状态数据库）组成。LevelDB 或 CouchDB 都可以与集合一起使用。选择

CouchDB 来演示如何对私有数据使用索引。

### 注意

为了使集合正常工作，正确配置跨组织 Gossip 很重要。请参阅我们的有关 Gossip 数据分发协议的文档，尤其要注意“锚点对等”部分。考虑到八卦已在 BYFN 示例中进行了配置，因此我们的教程不关注八卦，但是在配置通道时，八卦锚点对等方对于配置集合以使其正常工作至关重要。

## 7.4 安装并定义带有集合的链码

客户端应用程序通过链码与区块链分类账进行交互。因此，我们需要在每个将执行并认可我们交易的对等方上安装一个链码。但是，在我们与链码进行交互之前，渠道成员需要就链码定义达成共识，以建立链码治理，包括私有数据收集配置。我们将打包，安装，然后使用对等生命周期 chaincode 在通道上定义链码。

### 7.4.1 在所有对等节点上安装 chaincode

链码需要先打包，然后才能安装在我们的同级上。我们可以使用 `peer lifecycle chaincode package` 命令来打包 Marbles 链代码。

BYFN 网络包括两个组织 Org1 和 Org2，每个组织都有两个对等方。因此，chaincode 软件包必须安装在四个对等点上：

- peer0.org1.example.com
- peer1.org1.example.com
- peer0.org2.example.com
- peer1.org2.example.com

打包链码后，我们可以使用 `peer lifecycle chaincode install` 命令在每个对等点上安装 Marbles 链码。

#### 自己尝试

假设您已启动 BYFN 网络，请输入 CLI 容器：

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

您的命令提示符将类似于以下内容：

```
bash-4.4#
```

1. 使用以下命令将 git 存储库中的 Marbles 私有数据链代码打包到本地容器中。

```
peer lifecycle chaincode package marblesp.tar.gz --path github.com/hyperledger/fabric-samples/chaincode/marbles02_private/go/ --lang golang --label marblespv1
```

该命令将创建一个名为 marblesp.tar.gz 的链码包。

2. 使用以下命令将 chaincode 软件包安装到 `peer0.org1.example.com` 您的 BYFN 网络中的对等方。默认情况下，启动 BYFN 网络后，活动对等方设置为 `CORE_PEER_ADDRESS=peer0.org1.example.com:7051`：

```
peer lifecycle chaincode install marblesp.tar.gz
```

成功的安装命令将返回链码标识符，类似于以下响应：

```
2019-04-22 19:09:04.336 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 001 Installed remotely: response:<status:200 payload:"\nKmarblespv1:57f5353b2568b79cb5384b5a8458519a47186efc4fcadb98280f5eae6d59c1cd\022\nmarbles pv1" >
```

```
2019-04-22 19:09:04.336 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 002 Chaincode code package identifier: marblespv1:57f5353b2568b79cb5384b5a8458519a47186efc4fcadb98280f5eae6d59c1cd
```

3. 使用 CLI 将活动对等方切换到 Org1 中的第二个对等方并安装链码。将以下整个命令块复制并粘贴到 CLI 容器中并运行它们：

```
export CORE_PEER_ADDRESS=peer1.org1.example.com:8051
peer lifecycle chaincode install marblesp.tar.gz
```

4. 使用 CLI 切换到 Org2。将以下命令块作为一个组复制并粘贴到对等容器中，然后一次运行所有命令：

```
export CORE_PEER_LOCALMSPID=Org2MSP
```

```
export  
PEER0_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=$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
```

5. 将活动对等方切换到 Org2 中的第一个对等方并安装链码：

```
export CORE_PEER_ADDRESS=peer0.org2.example.com:9051  
peer lifecycle chaincode install marblesp.tar.gz
```

6. 将活动对等方切换到 org2 中的第二个对等方并安装链码：

```
export CORE_PEER_ADDRESS=peer1.org2.example.com:10051  
peer lifecycle chaincode install marblesp.tar.gz
```

## 7.4.2 批准链码定义

每个想要使用链码的渠道成员都需要为其组织批准链码定义。由于两个组织都将在本教程中使用链码，因此我们需要批准 Org1 和 Org2 的链码定义。

链码定义包括安装命令返回的软件包标识符。该软件包 ID 用于将对等方安装的链码软件包与组织批准的链码定义相关联。我们还可以使用对等生命周期链码 `queryinstalled` 命令查找的程序包 ID `marblesp.tar.gz`。

有了包 ID 后，就可以使用对等生命周期链代码 `approveformyorg` 命令来批准 Org1 和 Org2 的 Marbles 链代码的定义。要批准 `marbles02_private` 样本随附的私有数据集合定义，请使用 `--collections-config` 标记提供集合 JSON 文件的路径。

自己尝试

在 CLI 容器中运行以下命令以批准 Org1 和 Org2 的定义。

1. 使用以下命令向您的对等方查询已安装的链码的软件包 ID。

```
peer lifecycle chaincode queryinstalled
```

该命令将返回与安装命令相同的软件包标识符。您应该看到类似于以下内容的输出：

```
Installed chaincodes on peer:  
Package ID: marblespv1:57f5353b2568b79cb5384b5a8458519a47186efc4fcadb98280f5eae6d59c1cd, Label:  
marblespv1  
Package ID: mycc_1:27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8, Label:  
mycc_1
```

2. 将程序包 ID 声明为环境变量。将由返回的 marblespv1 的程序包 ID 粘贴到以下命令中。软件包 ID 对于所有用户而言可能都不相同，因此您需要使用从控制台返回的软件包 ID 来完成此步骤。

```
peer lifecycle chaincode queryinstalled
```

```
export CC_PACKAGE_ID=marblespv1:57f5353b2568b79cb5384b5a8458519a47186efc4fcadb98280f5eae6d59c1cd
```

3. 确保我们以 Org1 的身份运行 CLI。将以下命令块作为一个组复制并粘贴到对等容器中，然后一次运行所有命令：

```
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051  
export CORE_PEER_LOCALMSPID=Org1MSP  
export  
PEER0_ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt  
export CORE_PEER_TLS_ROOTCERT_FILE=$PEER0_ORG1_CA  
export  
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
```

4. 使用以下命令批准 Org1 的 Marbles 专用数据链代码的定义。此命令包括集合定义文件的路径。批准使用八卦在每个组织内分配，因此该命令不需要针对组织内的每个对等方。

```

export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.co
m/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
peer lifecycle chaincode approveformyorg --channelID mychannel --name marblesp --version 1.0 --
collections-config $GOPATH/src/github.com/hyperledger/fabric-
samples/chaincode/marbles02_private/collections_config.json --signature-policy
"OR('Org1MSP.member','Org2MSP.member')" --init-required --package-id $CC_PACKAGE_ID --sequence 1 --
tls true --cafile $ORDERER_CA

```

命令成功完成后，您应该会看到类似以下内容的内容：

```

2019-03-18 16:04:09.046 UTC [cli.lifecycle.chaincode] InitCmdFactory -> INFO 001 Retrieved
channel (mychannel) orderer endpoint: orderer.example.com:7050
2019-03-18 16:04:11.253 UTC [chaincodeCmd] ClientWait -> INFO 002 txid
[efba188ca77889cc1c328fc98e0bb12d3ad0abcd3f84da3714471c7c1e6c13c] committed with status (VALID) at

```

5. 使用 CLI 切换到 Org2。将以下命令块作为一个组复制并粘贴到对等容器中，然后一次运行它们。

```

export CORE_PEER_ADDRESS=peer0.org2.example.com:9051
export CORE_PEER_LOCALMSPID=Org2MSP
export
PEER0_ORG2_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.examp
le.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

```

6. 现在，您可以批准 Org2 的链码定义：

```

peer lifecycle chaincode approveformyorg --channelID mychannel --name marblesp --version 1.0 --
collections-config $GOPATH/src/github.com/hyperledger/fabric-
samples/chaincode/marbles02_private/collections_config.json --signature-policy
"OR('Org1MSP.member','Org2MSP.member')" --init-required --package-id $CC_PACKAGE_ID --sequence 1 --
tls true --cafile $ORDERER_CA

```

### 7.4.3 提交链码定义

一旦足够多的组织（在本例中为大多数）批准了链码定义，则一个组织将定义提交给渠道。

使用 `peer lifecycle chaincode commit` 命令来提交链码定义。此命令需要以 Org1 和 Org2 中的对等对象为目标，以收集提交事务的认可。同行仅在其组织批准了链码定义的情况下才认可交易。此命令还将集合定义部署到通道。

在链代码定义已提交给通道之后，我们准备使用链代码。由于 marbles 私有数据链码包含一个初始化函数，因此在使用链码中的 `Init()` 其他功能之前，需要使用对等链码 `invoke` 命令来调用。

**自己尝试**

1. 运行以下命令，将 Marbles 专用数据链代码的定义提交到 BYFN 通道 `mychannel`。

```

export
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.co
m/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
export
ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com
/peers/peer0.org1.example.com/tls/ca.crt
export
ORG2_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com
/peers/peer0.org2.example.com/tls/ca.crt
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID mychannel --name
marblesp --version 1.0 --sequence 1 --collections-config $GOPATH/src/github.com/hyperledger/fabric-
samples/chaincode/marbles02_private/collections_config.json --signature-policy
"OR('Org1MSP.member','Org2MSP.member')" --init-required --tls true --cafile $ORDERER_CA --
peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles $ORG1_CA --peerAddresses
peer0.org2.example.com:9051 --tlsRootCertFiles $ORG2_CA
.. note:: 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:
.. code:: bash

```

```
--collections-config $GOPATH/src/github.com/hyperledger/fabric-samples/chaincode/marbles02_private/collections_config.json
```

When the commit transaction completes successfully you should see something similar to:

```
.. code:: bash
```

```
[chaincodeCmd] checkChaincodeCmdParams -> INFO 001 Using default escc
[chaincodeCmd] checkChaincodeCmdParams -> INFO 002 Using default vscc
```

## 2. 使用以下命令来调用 `Init` 函数以初始化链码:

```
peer chaincode invoke -o orderer.example.com:7050 --channelID mychannel --name marblesp --isInit
--tls true --cafile $ORDERER_CA --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles
$ORG1_CA -c '{"Args":["Init"]}'
```

## 7.5 存储私人数据

充当 Org1 的成员，该 Org1 被授权与 Marbles 私有数据样本中的所有私有数据进行交易，切换回 Org1 对等方并提交添加 Marble 的请求：

自己尝试

将以下命令集复制并粘贴到 CLI 命令行。

```
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
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
export
PEER0_ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

调用 Marbles `initMarble` 功能可创建具有私人数据的 Marbles-名称 `marble1` 归其所有 `tom`，其颜色 `blue`，大小 `35` 和价格为 `99`。回想一下，私有数据 **价格** 将与私有数据 **名称**，**所有者**，**颜色**，**大小** 分开存储。因此，该 `initMarble` 函数调用 `PutPrivateData()` API 两次以保留私有数据，每个集合一次。另请注意，私有数据是使用 `--transient` 标志传递的。作为瞬态数据传递的输入将不会保留在事务中，以保持数据私有。瞬态数据作为二进制数据传递，因此在使用 CLI 时，必须以 base64 编码。我们使用环境变量来捕获 base64 编码值，并使用 `tr` 命令去除 linux base64 命令添加的有问题的换行符。

```
export MARBLE=$(echo -n
"{"name":"marble1","color":"blue","size":35,"owner":"tom","price":99}" | base64 | tr -d '\n')
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n marblesp -c
'{"Args":["initMarble"]}' --transient "{\"marble\":\"$MARBLE\"}"
```

您应该看到类似于以下内容的结果：

```
[chaincodeCmd] chaincodeInvokeOrQuery->INFO 001 Chaincode invoke successful. result: status:200
```

## 7.6 作为授权对等方查询私有数据

我们的集合定义允许 Org1 和 Org2 的所有成员在其辅助数据库中拥有私有数据，但是只有 Org1 中的对等方可以在其辅助数据库中拥有私有数据。作为 Org1 中的授权对等方，我们将查询两组私有数据。

`name, color, size, ownerprice`

第一个 `query` 命令调用作为参数 `readMarble` 传递 `collectionMarbles` 的函数。

```
// =====
// 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)
}

```

第二个 `query` 命令调用作为参数 `readMarblePrivateDetails` 传递 `collectionMarblePrivateDetails` 的函数。

```

// =====
// 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)
}

```

现在 **自己尝试**

查询作为 Org1 成员的私有数据。请注意，由于查询不会记录在分类帐中，因此无需将 Marbles 名称传递为瞬时输入。

```
name, color, size and owner marble1
```

```
peer chaincode query -C mychannel -n marble -c '{"Args":["readMarble","marble1"]}'
```

您应该看到以下结果：

```
{"color":"blue","docType":"marble","name":"marble1","owner":"tom","size":35}
```

查询作为 Org1 成员的 `price` 私有数据 `marble1`。

```
peer chaincode query -C mychannel -n marble -c
```

```
'{"Args":["readMarblePrivateDetails","marble1"]}'
```

您应该看到以下结果：

```
{"docType":"marblePrivateDetails","name":"marble1","price":99}
```

## 7.7 查询未经授权的对等数据

现在，我们将切换到 Org2 的成员，该成员的边数据库中有 Marbles 专用数据，但边数据库中没有 Marbles 专用数据。我们将查询两组私人数据。`name, color, size, ownerprice`

## 7.7.1 切换到 Org2 中的对等方

从 Docker 容器中，运行以下命令以切换到未经授权访问 Marbles `price` 私有数据的对等方。

自己尝试

```
export CORE_PEER_ADDRESS=peer0.org2.example.com:9051
export CORE_PEER_LOCALMSPID=Org2MSP
export
PEER0_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=$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
```

## 7.7.2 查询私有数据 Org2 被授权执行

Org2 中的对等方应在其边数据库中拥有第一组 Marbles 私有数据 ()，并且可以使用通过参数调用的函数进行访问。`name, color, size and ownerreadMarble()collectionMarbles`

自己尝试

```
peer chaincode query -C mychannel -n marble -c '{"Args":["readMarble","marble1"]}'
```

您应该看到类似于以下结果：

```
{"docType": "marble", "name": "marble1", "color": "blue", "size": 35, "owner": "tom"}
```

## 7.7.3 查询私有数据 Org2 未经授权

Org2 `price` 中的对等方在其边数据库中没有 Marbles 的私有数据。当他们尝试查询此数据时，他们将获取与公共状态匹配但不具有私有状态的键的哈希。

自己尝试

```
peer chaincode query -C mychannel -n marble -c
'{"Args":["readMarblePrivateDetails","marble1"]}'
```

您应该看到类似于以下结果：

```
{"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)"}
```

Org2 的成员将只能看到私有数据的公共哈希。

## 7.8 清除私人数据

对于仅在将私有数据复制到链外数据库之前才需要将其保存在分类账上的用例，可以在一定数量的块之后“清除”数据，而仅保留那些散列的数据。作为交易的不变证据。

可能存在包括个人或机密信息的私有数据，例如我们示例中的定价数据，交易双方不希望在渠道上向其他组织披露这些数据。因此，它的寿命有限，并且可以使用 `blockToLive` 集合定义中的属性在指定数量的块上在区块链上保持不变后清除。

我们的 `collectionMarblePrivateDetails` 定义的 `blockToLive` 属性值为 3，这意味着该数据将在辅助数据库中保留三个块，然后将其清除。将所有部分捆绑在一起，回想一下此集合定义 在调用 API 并将参数作为参数传递时与 函数 中 `collectionMarblePrivateDetails` 的 `price` 私有数据 相关联。

### initMarble()PutPrivateData()collectionMarblePrivateDetails

我们将逐步在链中添加块，然后通过发布四个新交易（创建一个新的 Marbles，然后进行三个 Marbles 转移）来观察价格信息是否被清除，这会在链中添加四个新块。在第四次交易（第三次 Marbles 交易）之后，我们将验证价格私人数据是否已清除。

#### 自己尝试

使用以下命令在 Org1 中切换回 peer0。复制并粘贴以下代码块，然后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.crt
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
export
PEER0_ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

打开一个新的终端窗口，并通过运行以下命令查看此对等方的私有数据日志：

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

您应该看到与以下类似的结果。注意列表中的最高块号。在以下示例中，最高块高度为 4。

```
[pvtdastorage] func1 -> INFO 023 Purger started: Purging expired private data till block number [0]
[pvtdastorage] func1 -> INFO 024 Purger finished
[kvledger] CommitLegacy -> INFO 022 Channel [mychannel]: Committed block [0] with 1 transaction(s)
[kvledger] CommitLegacy -> INFO 02e Channel [mychannel]: Committed block [1] with 1 transaction(s)
[kvledger] CommitLegacy -> INFO 030 Channel [mychannel]: Committed block [2] with 1 transaction(s)
[kvledger] CommitLegacy -> INFO 036 Channel [mychannel]: Committed block [3] with 1 transaction(s)
[kvledger] CommitLegacy -> INFO 03e Channel [mychannel]: Committed block [4] with 1 transaction(s)
```

返回对等容器，通过运行以下命令来查询 marble1 价格数据。（由于没有事务处理，因此查询不会在分类帐上创建新交易）。

```
peer chaincode query -C mychannel -n marblesp -c
'{"Args":["readMarblePrivateDetails","marble1"]}'
```

您应该看到类似于以下内容的结果：

```
{"docType": "marblePrivateDetails", "name": "marble1", "price": 99}
```

该 price 数据仍然在私有数据台账。

通过发出以下命令来创建新的 marble2。此交易在链上创建了一个新块。

```
export MARBLE=$(echo -n
"{"name": "marble2", "color": "blue", "size": 35, "owner": "tom", "price": 99}" | base64 | tr -d '\n')
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n marblesp -c
'{"Args":["initMarble"]}' --transient "{\"marble\": \"$MARBLE\"}"
```

切换回“终端”窗口，并再次查看该对等方的私有数据日志。您应该看到块高增加了 1。

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

返回对等容器，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"]}'
```

私有数据尚未清除，因此结果与上一个查询相同：

```
{"docType": "marblePrivateDetails", "name": "marble1", "price": 99}
```

通过运行以下命令，将 marble2 转移到“joe”。该交易将在链上添加第二个新区块。

```
export MARBLE_OWNER=$(echo -n "{\"name": \"marble2\", \"owner\": \"joe\"}" | base64 | tr -d '\n')
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\"}"
```

切换回“终端”窗口，并再次查看该对等方的私有数据日志。您应该看到块高增加了 1。

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

返回对等容器，通过运行以下命令来查询 marble1 价格数据：

```
peer chaincode query -C mychannel -n marblesp -c
'{"Args": ["readMarblePrivateDetails", "marble1"]}'
```

您仍然应该能够看到价格私有数据。

```
{"docType": "marblePrivateDetails", "name": "marble1", "price": 99}
```

通过运行以下命令，将 marble2 传输到“tom”。该交易将在链上创建第三个新区块。

```
export MARBLE_OWNER=$(echo -n "{\"name\": \"marble2\", \"owner\": \"tom\"}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n marblesp -c
'{"Args": ["transferMarble"]}' --transient "{\"marble_owner\": \"$MARBLE_OWNER\"}"
```

切换回“终端”窗口，并再次查看该对等方的私有数据日志。您应该看到块高增加了 1。

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

返回对等容器，通过运行以下命令来查询 marble1 价格数据：

```
peer chaincode query -C mychannel -n marblesp -c
'{"Args": ["readMarblePrivateDetails", "marble1"]}'
```

您仍然应该能够看到价格数据。

```
{"docType": "marblePrivateDetails", "name": "marble1", "price": 99}
```

最后，通过运行以下命令将 marble2 转移到“jerry”。该交易将在链上创建第四个新区块。该 **price** 私人数据应本次交易后，被清除。

```
export MARBLE_OWNER=$(echo -n "{\"name\": \"marble2\", \"owner\": \"jerry\"}" | base64 | tr -d \\n)
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/
orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n marblesp -c
'{"Args": ["transferMarble"]}' --transient "{\"marble_owner\": \"$MARBLE_OWNER\"}"
```

切换回“终端”窗口，并再次查看该对等方的私有数据日志。您应该看到块高增加了 1。

```
docker logs peer0.org1.example.com 2>&1 | grep -i -a -E 'private|pvt|privdata'
```

返回对等容器，通过运行以下命令来查询 marble1 价格数据：

```
peer chaincode query -C mychannel -n marblesp -c
'{"Args": ["readMarblePrivateDetails", "marble1"]}'
```

由于价格数据已清除，因此您将不再能够看到它。您应该看到类似以下内容：

```
Error: endorsement failure during query. response: status:500
message: "{\"Error\": \"Marble private details does not exist: marble1\"}"
```

## 7.9 使用私人数据的索引

通过将索引包装在 **META-INF/statedb/couchdb/collections/<collection\_name>/indexes** 链码旁边的目录中，索引也可以应用于私有数据集合。[此处](#) 提供示例索引。

为了将链码部署到生产环境，建议在链码旁边定义任何索引，以便一旦链码已安装在对等端并在通道上实例化后，链码和支持索引将自动作为单元部署。当 **--collections-config** 指定标志指向集合 JSON 文件的位置时，关联索引将在通道上链代码实例化时自动部署。

# 8 Chaincode 教程

## 8.1 什么是 Chaincode?

Chaincode 是用 [Go](#), [node.js](#) 或 [Java](#) 编写的程序，可实现规定的接口。Chaincode 在与认可对等进程隔离的安全 Docker 容器中运行。Chaincode 通过应用程序提交的事务初始化和管理分类帐状态。

链码通常处理网络成员同意的业务逻辑，因此可以将其视为“智能合约”。由一个链码创建的状态仅适用于该链码，并且不能被另一个链码直接访问。但是，在同一个网络中，如果获得适当的许可，则链码可以调用另一个链码以访问其状态。

## 8.2 两个角色

我们对链码提供两种不同的观点。从应用程序开发人员的角度出发，开发名为 [Chaincode for Developers](#) 的区块链应用程序/解决方案，另一种是面向面向负责管理区块链网络并利用 Hyperledger Fabric API 的区块链网络运营商的运营商 Chaincode。安装和管理 Chaincode，但可能不会参与 Chaincode 应用程序的开发。

## 8.3 织物链码生命周期

Fabric 链码生命周期负责在信道上使用链码之前管理链码的安装及其参数的定义。从 Fabric 2.0 Alpha 开始，链码的管理已完全分散：在链码用于与分类帐交互之前，多个组织可以使用 Fabric 链码生命周期来就链码的参数（例如链码认可策略）达成协议。。

新模型在以前的生命周期中提供了一些改进：

- **多个组织必须同意链码的参数：**在 Fabric 的 1.x 版本中，一个组织可以为所有其他渠道成员设置链码的参数（例如，认可策略）。新的 Fabric 链码生命周期更加灵活，因为它既支持集中式信任模型（例如先前生命周期模型的模型），也支持分散模型，这些模型需要足够数量的组织才能在背书策略上生效。
- **更安全的链码升级过程：**在先前的链码生命周期中，升级事务可能由单个组织发出，这给尚未安装新链码的渠道成员带来了风险。新模型仅在足够数量的组织批准升级后才允许升级链码。
- **更容易的背书策略更新：**Fabric 生命周期允许您更改背书策略，而无需重新打包或重新安装链码。用户还可以利用新的默认策略，该策略需要频道中大多数成员的认可。在从渠道中添加或删除组织时，该策略会自动更新。
- **可检查的链代码包：**Fabric 生命周期将链代码打包在易于阅读的 tar 文件中。这使得检查链码包和协调跨多个组织的安装变得更加容易。
- **使用一个程序包在通道上启动多个链码：**上一个生命周期使用安装链码包时指定的名称和版本定义了通道上的每个链码。现在，您可以使用单个 chaincode 程序包，并在相同或不同的通道上以不同的名称多次部署它。

要了解有关新 Fabric 生命周期的更多信息，请访问 [Chaincode for Operators](#)。

注意

v2.0 Alpha 版本中新的 Fabric 链码生命周期尚未完成。具体来说，请注意 Alpha 版本中的以下限制：

- 尚不支持 CouchDB 索引
- 通过服务发现尚无法发现使用新生命周期定义的链码

这些限制将在 Alpha 版本发布后解决。要使用旧的生命周期模型来安装和实例化 Chaincode，请访问 [Chaincode for Operators v1.4 版本](#)

您可以通过创建新通道并将通道功能设置为 v2\_0 来使用 Fabric 链码生命周期。您将无法使用旧的生命周期在启用了 v2\_0 功能的频道上安装，实例化或更新链码。但是，启用 v2\_0 功能后，仍可以调用使用以前的生命周期模型安装的链码。Fabric v2.0 Alpha 不支持从以前的生命周期迁移到新的生命周期。

# 9 开发人员链码

## 9.1 什么是 Chaincode?

Chaincode 是用 [Go](#), [node.js](#) 或 [Java](#) 编写的程序，可实现规定的接口。Chaincode 在与认可对等进程隔离的安全 Docker 容器中运行。Chaincode 通过应用程序提交的事务初始化和管理分类帐状态。

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”。可以调用链码来更新或查询投标交易中的分类帐。在获得适当许可的情况下，一个链码可以在同一通道或不同通道中调用另一个链码以访问其状态。请注意，如果被调用链码与调用链码位于不同的频道，则仅允许读取查询。也就是说，在另一个通道上的被调用链码仅为 a [Query](#)，它不参与后续提交阶段中的状态验证检查。

在以下各节中，我们将通过应用程序开发人员的眼光探索链代码。我们将提供一个简单的 chaincode 示例应用程序，并逐步介绍 Chaincode Shim API 中每种方法的目的。

## 9.2 Chaincode API

每个 chaincode 程序都必须实现 [Chaincode](#) 接口，该接口的方法被调用以响应收到的事务。您可以在下面找到针对不同语言的 Chaincode Shim API 的参考文档：

- 走 GO
- [node.js](#)
- JAVA

每种语言 [Invoke](#) 都由客户调用该方法来提交交易建议。此方法允许您使用链码在通道分类帐上读写数据。

您还需要包括一个 [Init](#) 将用作链码初始化函数的方法。在启动或升级链码时，将调用此方法以对其进行初始化。默认情况下，此功能从不执行。但是，您可以使用链码定义来请求 [Init](#) 执行该功能。如果 [Init](#) 请求执行，fabric 将确保 [Init](#) 先调用该函数，然后再调用一次。通过此选项，您可以进一步控制哪些用户可以初始化链码以及向账本添加初始数据的功能。如果您使用对等 CLI 批准链码定义，请使用该 [--init-required](#) 标志请求执行该 [Init](#) 功能。然后 [Init](#) 使用对等 [chaincode](#) 调用命令并传递 [--isInit](#) 标志。如果您使用 Fabric SDK for Node.js，请访问 [如何安装和启动链码](#)。有关更多信息，请参见 [运营商链码](#)。

链码“shim”API 中的另一个接口是 [ChaincodeStubInterface](#)：

- GO
- [node.js](#)
- JAVA

用于访问和修改分类帐，以及在链码之间进行调用。

在使用 Go 链码的本教程中，我们将通过实现一个管理简单“资产”的简单链码应用程序来演示这些 API 的用法。

## 9.3 简单资产链码

我们的应用程序是一个基本示例链代码，用于在分类账上创建资产（键值对）。

### 9.3.1 选择代码的位置

如果您尚未使用 Go 进行编程，则可能需要确保已安装 Go 编程语言并正确配置了系统。

现在，您将要为 Chaincode 应用程序创建一个目录，作为的子目录 [\\$GOPATH/src/](#)。

为简单起见，让我们使用以下命令：

```
mkdir -p $GOPATH/src/sacc && cd $GOPATH/src/sacc
```

现在，让我们创建将用代码填充的源文件：

```
touch sacc.go
```

## 9.3.2 家政

首先，让我们开始做一些整理工作。与每个链码一样，它 特别实现了 [链码接口](#) [Init](#) 和 [Invoke](#) 功能。因此，让我们添加 Go import 语句以获取链码所需的依赖关系。我们将导入 `chaincode shim` 软件包和对 [等 protobuf 软件包](#)。接下来，让我们添加一个结构 [SimpleAsset](#) 作为 Chaincode 填充函数的接收器。

```
package main

import (
    "fmt"

    "github.com/hyperledger/fabric-chaincode-go/shim"
    "github.com/hyperledger/fabric-protos-go/peer"
)

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
```

## 9.3.3 初始化链码

接下来，我们将实现该 [Init](#) 功能。

```
// Init is called during chaincode instantiation to initialize any data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
}
```

注意

请注意，链码升级也调用此函数。编写链码以升级现有的链码时，请确保 [Init](#) 适当地修改功能。特别是，如果没有“迁移”或在升级过程中没有任何初始化要提供，请提供一个空的“[Init](#)”方法。

接下来，我们将 [Init](#) 使用 [ChaincodeStubInterface.GetStringArgs](#) 函数检索调用 的参数，并检查其有效性。在我们的例子中，我们期望一个键值对。

```
// 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")
    }
}
```

接下来，既然我们已经确定调用是有效的，我们将把初始状态存储在分类帐中。为此，我们将使用[传入](#) 的键和值调用 [ChaincodeStubInterface.PutState](#)。假设一切顺利，请返回一个 `peer.Response` 对象，该对象指示初始化成功。

```
// 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)
}

```

### 9.3.4 调用链码

首先，让我们添加 `Invoke` 函数的签名。

```

// 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 {
}

```

与 `Init` 上面的函数一样，我们需要从中提取参数 `ChaincodeStubInterface`。该 `Invoke` 函数的参数将是要调用的 chaincode 应用程序函数的名称。在我们的案例中，我们的应用程序将仅具有两个功能：`set` 和 `get`，它们允许设置资产的值或检索其当前状态。我们首先调用 `ChaincodeStubInterface.GetFunctionAndParameters` 以提取函数名称和该 Chaincode 应用程序函数的参数。

```

// 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()
}

```

接下来，我们将验证函数名称为 `set` 或 `get`，并调用那些 chaincode 应用程序函数，通过 `shim.Success` 或 `shim.Error` 函数返回适当的响应，该响应会将响应序列化为 gRPC protobuf 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))
}

```

### 9.3.5 实施 Chaincode 应用程序

如前所述，我们的 chaincode 应用程序实现了两个可以通过该 `Invoke` 函数调用的函数。让我们现在实现这些功能。请注意，如上所述，要访问分类帐的状态，我们将利用 chaincode 填充程序 API 的 `ChaincodeStubInterface.PutState` 和 `ChaincodeStubInterface.GetState` 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
}

```

### 9.3.6 合并在一起

最后，我们需要添加 `main` 函数，该函数将调用 `shim.Start` 函数。这是整个 chaincode 程序的源代码。

```

package main

import (
    "fmt"

    "github.com/hyperledger/fabric-chaincode-go/shim"
    "github.com/hyperledger/fabric-protos-go/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)
    }
}

```

## 9.3.7 构造链码

现在，让我们编译您的链码。

```
go get -u github.com/hyperledger/fabric-chaincode-go
go build
```

假设没有错误，现在我们可以继续下一步，测试您的链码。

## 9.3.8 使用开发人员模式进行测试

通常，链码由对等方启动和维护。但是，在“开发模式”下，链码由用户构建和启动。在链代码开发阶段，此模式对于快速执行代码/构建/运行/调试周期周转非常有用。

我们通过利用示例开发网络的预生成订单程序和通道工件来启动“开发模式”。这样，用户可以立即进入编译链码和拨打电话的过程。

## 9.4 安装 Hyperledger Fabric 示例

如果您尚未这样做，请安装 Samples, Binaries 和 Docker Images。

导航到 克隆 `chaincode-docker-devmode` 目录 `fabric-samples` :

```
cd chaincode-docker-devmode
```

现在打开三个终端，并 `chaincode-docker-devmode` 在每个终端中导航到您的目录。

## 9.5 终端 1-启动网络

```
docker-compose -f docker-compose-simple.yaml up
```

上面的命令使用 `SingleSampleMSPSolo` 排序者配置文件启动网络，并以“开发模式”启动对等方。它还启动了另外两个容器一个用于链代码环境的容器和一个与链代码进行交互的 CLI。用于创建和加入通道的命令嵌入在 CLI 容器中，因此我们可以立即跳转到链码调用。

- 注意：对等方未使用 TLS，因为开发模式不适用于 TLS。

## 9.6 2 号航站楼-构建并启动链码

```
docker exec -it chaincode sh
```

您应该看到以下内容：

```
/opt/gopath/src/chaincode $
```

现在，编译您的链码：

```
cd sacc  
go build
```

现在运行 chaincode：

```
CORE_CHAINCODE_ID_NAME=mycc:0 CORE_PEER_TLS_ENABLED=false ./sacc -peer.address peer:7052
```

链码以对等方和链码日志开始，指示向对等方成功注册。请注意，在此阶段，链码未与任何通道关联。这是在后续步骤中使用 `instantiate` 命令完成的。

## 9.7 第 3 航站楼-使用链码

即使您处于 `--peer-chaincodedev` 模式下，您仍然必须安装 chaincode，以便生命周期系统 chaincode 可以正常进行检查。将来在 `--peer-chaincodedev` 模式下可以删除此要求。

我们将利用 CLI 容器来驱动这些调用。

```
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
```

现在发出一个调用，将“a”的值更改为“20”。

```
peer chaincode invoke -n mycc -c '{"Args":["set", "a", "20"]}' -C myc
```

最后，查询 `a`。我们应该看到的值 `20`。

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

## 9.8 测试新的链码

默认情况下，我们仅挂载 `sacc`。但是，您可以通过将不同的链码添加到 `chaincode` 子目录并重新启动网络来轻松测试它们。此时，您可以在您的 `chaincode` 容器中访问它们。

## 9.9 链码访问控制

Chaincode 可以通过调用 `GetCreator()` 函数来利用客户端（提交者）证书进行访问控制决策。此外，Go shim 提供了扩展 API，这些 API 从提交者的证书中提取客户端身份，可用于访问控制决策，无论是基于客户端身份本身，组织身份还是基于客户端身份属性。

例如，表示为键/值的资产可以将客户端的身份包括为值的一部分（例如，作为表示该资产所有者的 JSON 属性），并且只有该客户端可以被授权对键/值进行更新在将来。客户端身份库扩展 API 可以在链码中使用，以检索此提交者信息以做出此类访问控制决策。

有关更多详细信息，请参见[客户端身份（CID）库文档](#)。

要将客户端身份填充程序扩展名作为依赖项添加到您的链码中，请参阅[管理用 Go 编写的链码的外部依赖项](#)。

## 9.10 管理用 Go 编写的链码的外部依赖关系

您的 Go 链代码需要不属于 Go 标准库的软件包（例如链代码 shim）。这些软件包必须包含在您的 chaincode 软件包中。

有许多工具可用于管理（或“供应”）这些依赖项。以下演示了如何使用 `govendor`：

```
govendor init  
govendor add +external // Add all external package, or  
govendor add github.com/external/pkg // Add specific external package
```

这会将外部依赖项导入本地 `vendor` 目录。如果要供应 Fabric 垫片或 shim 扩展，请在执行 `govendor` 命令之前将 Fabric 存储库克隆到`$ GOPATH/src/github.com/hyperledger` 目录中。

一旦将依赖项供应到 chaincode 目录中，然后操作便会将与该依赖项关联的代码包含在 chaincode 程序包中。

```
peer chaincode package peer chaincode install
```

# 10 运营商链码

## 10.1 什么是 Chaincode?

Chaincode 是用 [Go](#), [Node.js](#) 或 [Java](#) 编写的程序，可实现规定的接口。Chaincode 在与认可对等进程隔离的安全 Docker 容器中运行。Chaincode 通过应用程序提交的事务初始化和管理分类帐状态。

链码通常处理网络成员同意的业务逻辑，因此可以将其视为“智能合约”。由一个链码创建的分类帐更新仅限于该链码的范围，并且不能被另一个链码直接访问。但是，在同一个网络中，如果获得适当的许可，则链码可以调用另一个链码以访问其状态。

在以下各节中，我们将通过区块链网络运营商而不是应用程序开发人员的视角来探索链代码。链码运营商可以使用本教程来学习如何使用 Fabric 链码生命周期在其网络上部署和管理链码。

## 10.2 链码生命周期

Fabric 链码生命周期是一个过程，它允许多个组织在链码可以在通道上使用之前就如何操作链码达成一致。本教程将讨论链码操作员如何使用 Fabric 生命周期执行以下任务：

- [安装并定义一个链码](#)
- [升级链码](#)
- [部署方案](#)
- [迁移到新的 Fabric 生命周期](#)

注意：v2.0 Alpha 版本中新的 Fabric 链码生命周期尚未完成。具体来说，请注意 Alpha 版本中的以下限制：

- 尚不支持服务发现
- 通过服务发现尚无法发现使用新生命周期定义的链码

这些限制将在 Alpha 版本发布后解决。要使用旧的生命周期模型来安装和实例化 Chaincode，请访问 v1.4 版本的 [Chaincode for Operators 教程](#)。

## 10.3 安装并定义一个链码

Fabric 链码生命周期要求组织同意定义链码的参数，例如名称，版本和链码认可策略。渠道成员通过以下四个步骤达成协议。并非渠道上的每个组织都需要完成每个步骤。

1. **打包链码：**可以由一个组织或每个组织完成此步骤。
2. **在您的对等方上安装链码：**每个将使用链码来认可交易或查询分类帐的组织都需要完成此步骤。
3. **批准组织的链码定义：**每个将使用链码的组织都需要完成此步骤。链码定义需要得到足够多的组织的批准，才能满足频道的 LifecycleEndorsement 策略（默认情况下，大多数是默认），然后才能在频道上启动链码。

4. 将链码定义提交到渠道：一旦批准了渠道上所需数目的组织，提交事务就需要由一个组织提交。提交者首先从已经批准的组织的足够的同龄人那里收集背书，然后提交交易以提交链码定义。

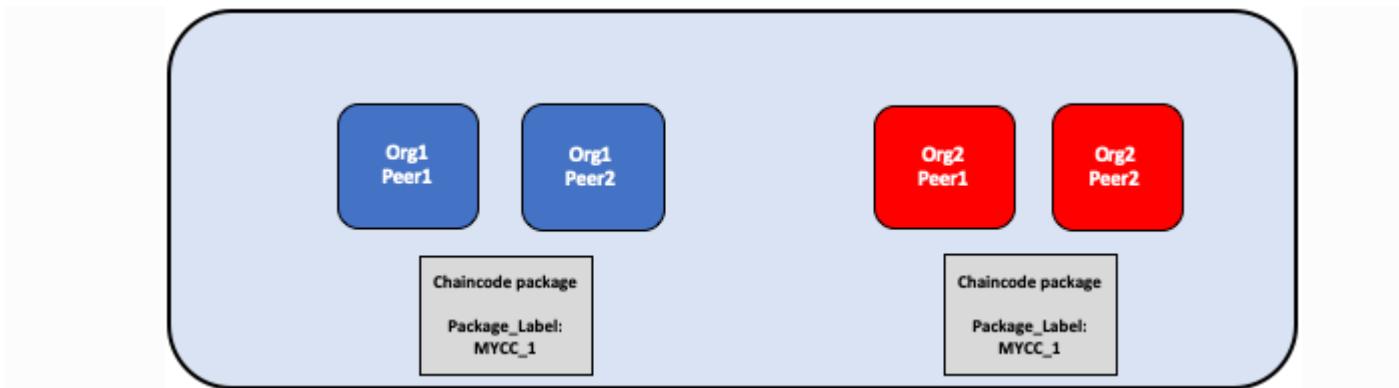
本教程详细介绍了 Fabric 链码生命周期的操作，而不是特定的命令。要了解有关如何使用对等 CLI 来使用 Fabric 生命周期的更多信息，请参阅《构建您的第一个网络教程》或《[peer lifecycle 命令参考](#)》中的“安装并定义链码”。要了解有关如何使用用于 Node.js 的 Fabric SDK 来使用 Fabric 生命周期的更多信息，请访问[如何安装和启动链码](#)。

### 10.3.1 第一步：打包智能合约

Chaincode 必须先包装在 tar 文件中，然后才能安装在对等方上。您可以使用 Fabric 对等二进制文件，Node Fabric SDK 或第三方工具（例如 GNU tar）打包链码。创建 chaincode 程序包时，需要提供一个 chaincode 程序包标签，以创建该程序包的简洁易读的描述。

如果您使用第三方工具打包链码，则生成的文件需要采用以下格式。Fabric 对等二进制文件和 Fabric SDK 将自动创建此格式的文件。

- 链码需要打包在 tar 文件中，并以 `.tar.gz` 文件扩展名结尾。
- tar 文件需要包含两个文件（无目录）：元数据文件“Chaincode-Package-Metadata.json”和另一个包含链码文件的 tar。
- “Chaincode-Package-Metadata.json”包含用于指定链码语言，代码路径和包标签的 JSON。您可以在下面看到元数据文件的示例：
  - `{ "Path": "github.com/chaincode/fabcar/go", "Type": "golang", "Label": "fabcarv1" }`

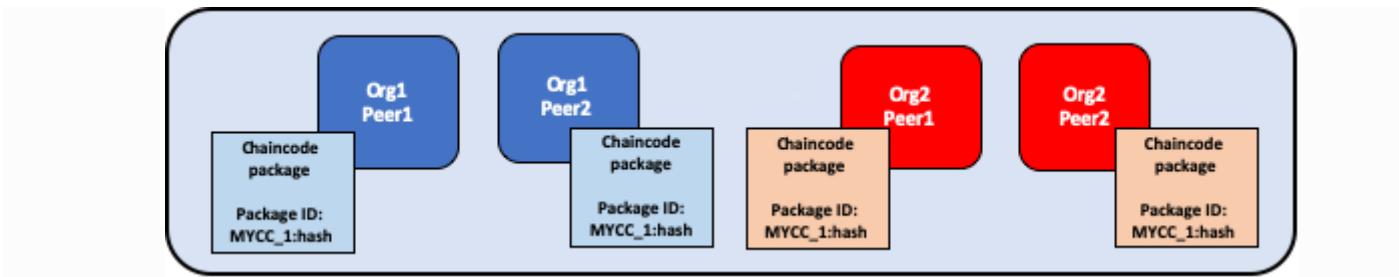


链码由 Org1 和 Org2 分别打包。两个组织都使用 MYCC\_1 作为其包装标签，以便使用名称和版本来标识包装。组织没有必要使用相同的包装标签。

### 10.3.2 第二步：在您的 Peer 节点上安装 chaincode

您需要在将执行和认可交易的每个对等方上安装 chaincode 软件包。无论使用 CLI 还是 SDK，都需要使用 **Peer Administrator** 来完成此步骤，**Peer Administrator** 的签名证书位于对等 MSP 的 `admincerts` 文件夹中。建议组织仅将链码打包一次，然后在属于其组织的每个对等方上安装相同的软件包。如果某个渠道想要确保每个组织都运行相同的链码，则一个组织可以打包一个链码并将其发送到带外其他渠道成员。

成功的安装命令将返回一个链码软件包标识符，该标识符是软件包标签和软件包的哈希值。此软件包标识符用于将对等方安装的链码软件包与组织批准的链码定义相关联。**保存标识符** 以进行下一步。您还可以通过使用对等 CLI 查询对等方上安装的软件包来找到软件包标识符。



来自 Org1 和 Org2 的对等管理员将链码包 MYCC\_1 安装在加入该通道的对等体上。安装 chaincode 程序包将创建一个程序包标识符 MYCC\_1: hash。

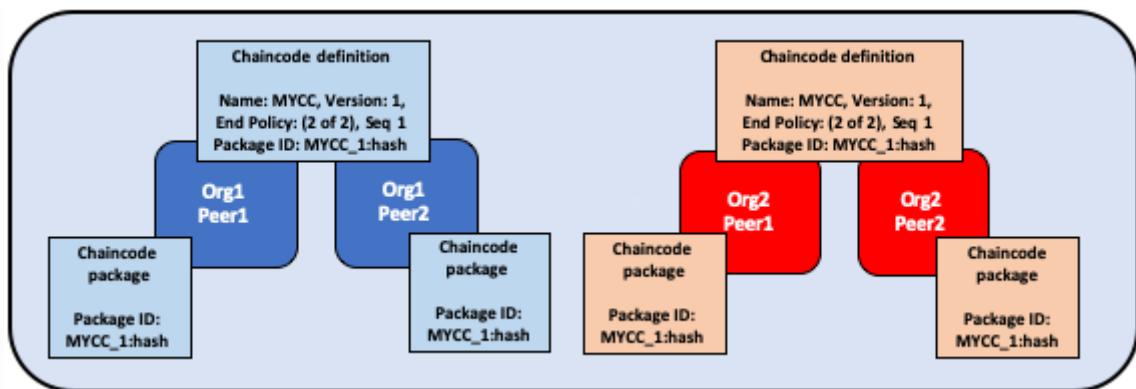
### 10.3.3 第三步：批准组织的链码定义

**链码受链码定义约束。**渠道成员批准链码定义时，批准将作为组织对其接受的链码参数的投票。这些批准的组织定义允许渠道成员在可在渠道上使用链码之前达成共识。链码定义包含以下参数，这些参数在组织之间必须保持一致：

- **名称：**应用程序在调用链码时将使用的名称。
- **版本：**与给定的链码包关联的版本号或值。如果升级链码二进制文件，则还需要更改链码版本。
- **顺序：**定义链码的次数。该值是一个整数，用于跟踪链码升级。例如，当您第一次安装并批准链码定义时，序列号将为 1。下次升级链码时，序列号将增加为 2。
- **认可政策：**哪些组织需要执行和验证交易输出。认可策略可以表示为传递给 CLI 或 SDK 的字符串，也可以在通道配置中引用策略。默认情况下，背书策略设置为 `Channel/Application/Endorsement`，默认情况下要求渠道中的大多数组织背书交易。
- **集合配置：**与链码相关联的私有数据集合定义文件的路径。有关私有数据收集的更多信息，请参阅[私有数据体系结构参考](#)。
- **初始化：**所有链码都需要包含一个 `Init` 用于初始化链码的函数。默认情况下，此功能从不执行。但是，您可以使用链码定义来请求该 `Init` 函数可调用。如果 `Init` 请求执行，fabric 将确保 `Init` 先调用该函数，然后再调用一次。
- **ESCC / VSCC 插件：**此链码将使用的自定义认可或验证插件的名称。

链码定义还包括 **Package Identifier**。这是每个要使用链码的组织的必需参数。包 ID 不必对于所有组织都相同。组织可以批准链码定义，而无需安装链码包或在定义中包括标识符。

每个想要使用链码的渠道成员都需要为其组织批准链码定义。该批准需要提交给排序服务，然后再分发给所有对等方。该批准需要由您的**组织管理员**提交，该组织的签名证书在您组织定义的 MSP 中列为管理员证书。成功提交批准交易后，批准的定义将存储在一个集合中，该集合可供组织的所有对等方使用。因此，即使您有多个对等方，您也只需要为组织批准一次链码。



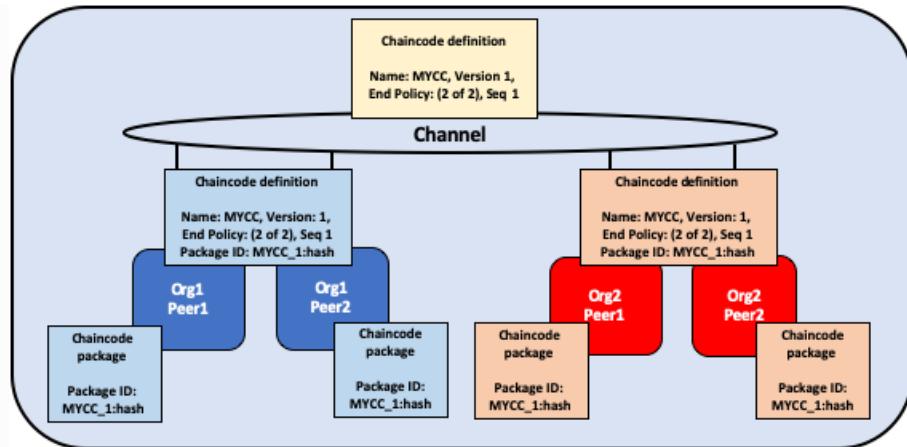
Org1 和 Org2 的组织管理员批准其组织的 MYCC 链代码定义。链码定义包括链码名称，版本和认可策略以及其他字段。由于两个组织都将使用链码来认可交易，因此两个组织的批准定义都需要包括 packageID。

#### 10.3.4第四步：将链码定义提交给通道

一旦足够数量的渠道成员批准了链码定义，则一个组织可以将定义提交给渠道。您可以使用 `checkcommitreadiness` 命令检查基于哪个通道成员已批准定义的提交链码定义是否成功，然后再使用对等 CLI 将其提交给通道。首先将提交交易建议发送给渠道成员的对等方，渠道成员的对等方查询为其组织批准的链码定义，并在组织批准的前提下认可该定义。然后，将交易提交给排序服务，然后排序服务将链码定义提交给渠道。提交定义事务需要以**组织管理员的身份**提交，其签名证书在您的组织定义的 MSP 中列为管理证书。

在将定义成功提交到渠道之前，需要批准其定义的组织数量受 `Channel/Application/LifecycleEndorsement` 策略控制。默认情况下，此策略要求渠道中的大多数组织都认可该交易。LifecycleEndorsement 策略与链码认可策略是分开的。例如，即使链码认可策略仅需要一个或两个组织的签名，大多数渠道成员仍需要根据默认策略批准链码定义。提交渠道定义时，您需要在渠道中定位足够的对等组织，以满足您的 LifecycleEndorsement 策略。

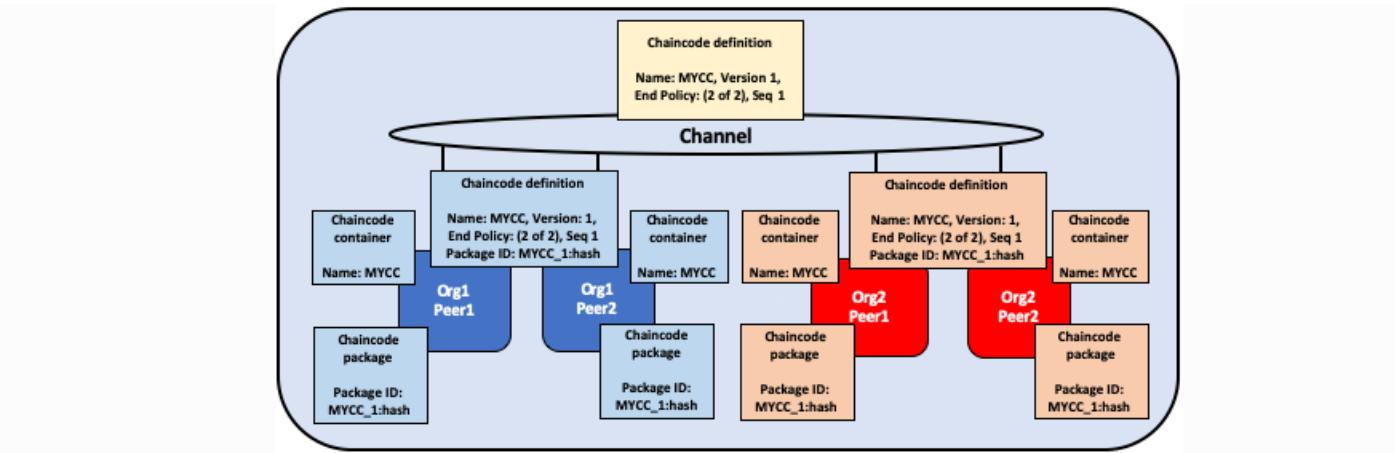
您还可以将 `Channel/Application/LifecycleEndorsement` 策略设置为签名策略，并在通道上显式指定可以批准链码定义的组织集。这使您可以创建一个通道，由一定数量的组织充当链码管理员，并管理该通道使用的业务逻辑。如果您的频道有大量的 Idemix 组织，它们不能批准链码定义或认可链码，并因此可能阻止频道占多数，您也可以使用签名策略。



来自 Org1 或 Org2 的一位组织管理员将链码定义提交给通道。通道上的定义不包括 packageID。

组织可以批准链码定义而不安装链码包。如果组织不需要使用链码，则可以批准不带包标识符的链码定义，以确保满足“生命周期认可”策略。

将链码定义提交给通道后，通道成员可以开始使用链码。链码的首次调用将在交易提议所针对的所有对等点上启动链码容器，只要这些对等点已经安装了链码包即可。您可以使用链码定义来要求调用 `Init` 函数来启动链码。否则，通道成员可以通过调用链码中的任何事务来启动链码容器。首次调用，无论是 `Init` 功能调用还是其他事务调用，都应遵循链码认可策略。启动 chaincode 容器可能需要几分钟。

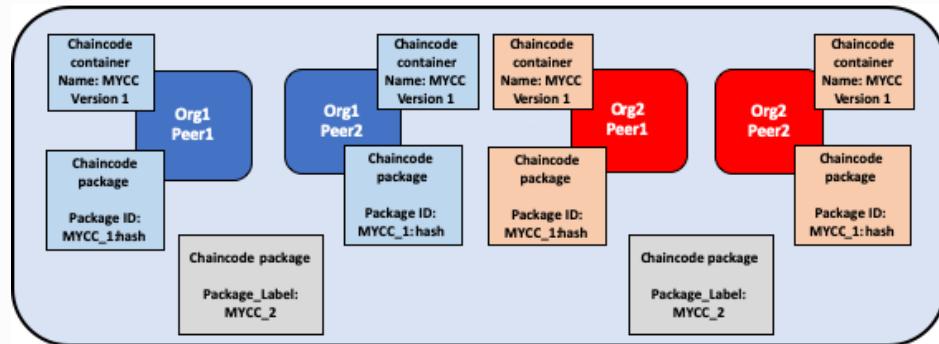


一旦在通道上定义了 MYCC, Org1 和 Org2 就可以开始使用链码了。在每个对等方上首次调用 `chaincode`, 将在该对等方上启动 `chaincode` 容器。

## 10.4 升级链码

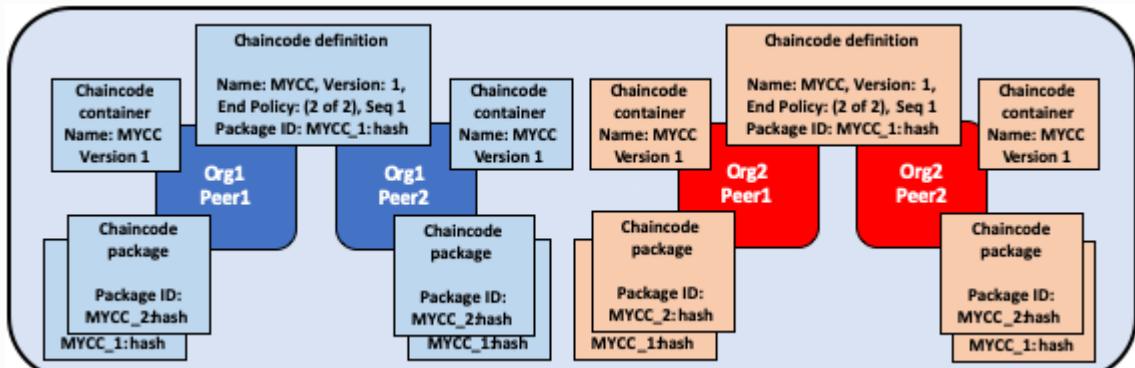
您可以使用与安装和启动链码相同的 Fabric 生命周期过程来升级链码。您可以升级链码二进制文件，或仅更新链码策略。请按照以下步骤升级链码：

- 重新打包链码：**仅在升级链码二进制文件时才需要完成此步骤。



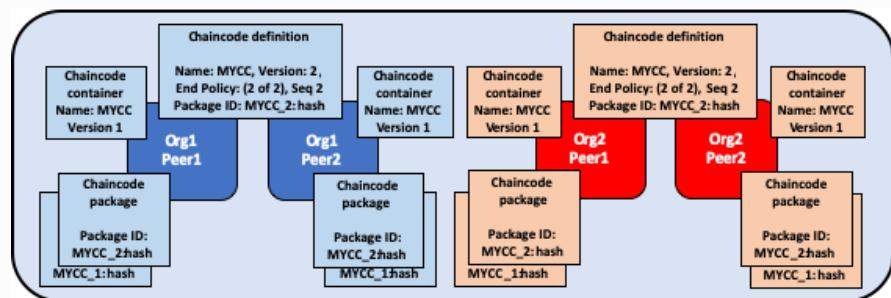
Org1 和 Org2 升级链码二进制文件并重新打包链码。两家公司都使用不同的包装标签。

- 在对等方上安装新的 chaincode 软件包：**再次，如果要升级 chaincode 二进制文件，则仅需要完成此步骤。安装新的 chaincode 软件包将生成一个软件包 ID，您需要将其传递给新的 chaincode 定义。您还需要更改链码版本。



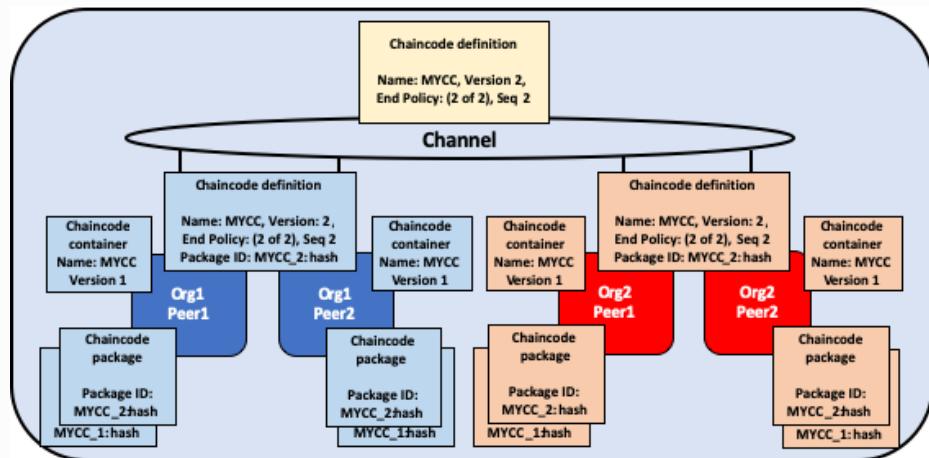
Org1 和 Org2 在同级上安装新软件包。安装将创建一个新的 packageID。

**3. 批准新的链码定义：**如果要升级链码二进制文件，则需要更新链码定义中的链码版本和程序包 ID。您也可以更新链码认可策略，而不必重新打包链码二进制文件。渠道成员只需要批准新政策的定义。新定义需要将定义中的序列变量加 1。



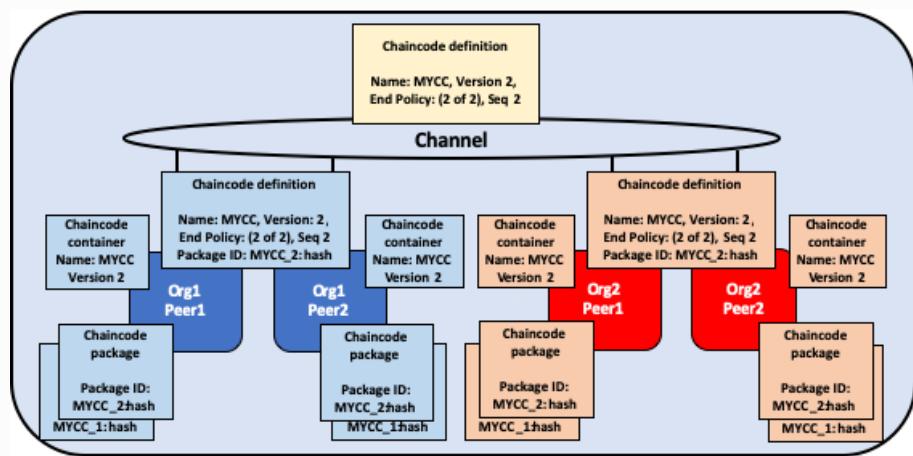
Org1 和 Org2 的组织管理员为各自的组织批准新的链码定义。新定义引用了新的 packageID 并更改了链码版本。由于这是链码的第一次更新，因此序列从一递增到二。

**4. 将定义提交给通道：**当足够数量的通道成员批准了新的链码定义时，一个组织可以提交新定义以将链码定义升级到通道。作为生命周期过程的一部分，没有单独的升级命令。



来自 Org1 或 Org2 的组织管理员将新的链码定义提交到通道。链码容器仍在运行旧的链码。

**5. 升级链码容器：**如果在不升级链码包的情况下更新了链码定义，则无需升级链码容器。如果确实升级了链码二进制文件，则新的调用将升级链码容器。如果 Init 在链码定义中请求执行该功能，则需要在 Init 成功提交新定义后再次调用该功能，以升级链码容器。



将新定义提交到通道后，每个对等方的下一次调用将自动启动新的链码容器。

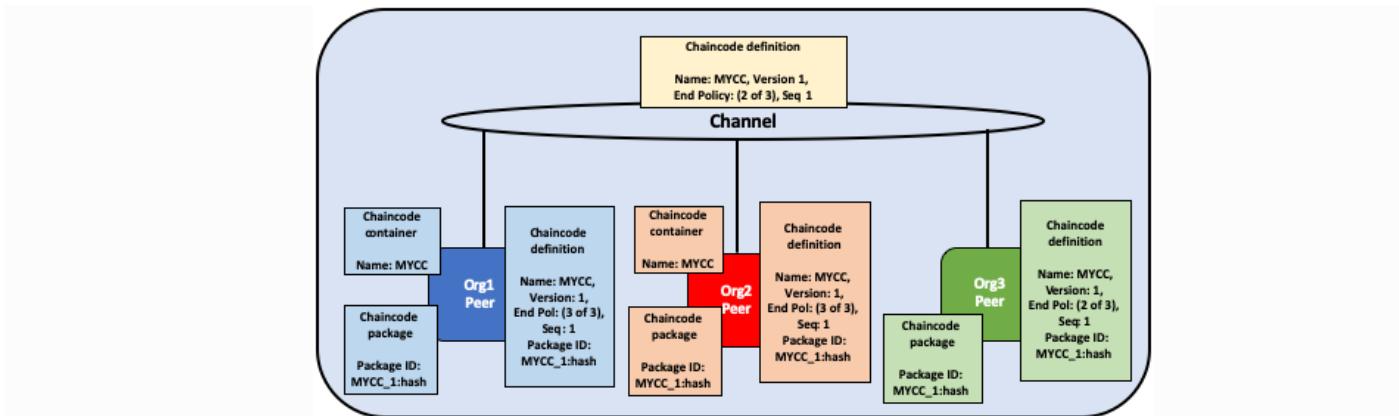
Fabric 链码生命周期使用链码定义中的序列来跟踪升级。所有通道成员都需要将序列号增加一个，并批准新的定义以升级链码。版本参数用于跟踪链码二进制文件，仅在升级链码二进制文件时才需要更改。

## 10.5 部署方案

以下示例说明了如何使用 Fabric 链码生命周期来管理通道和链码。

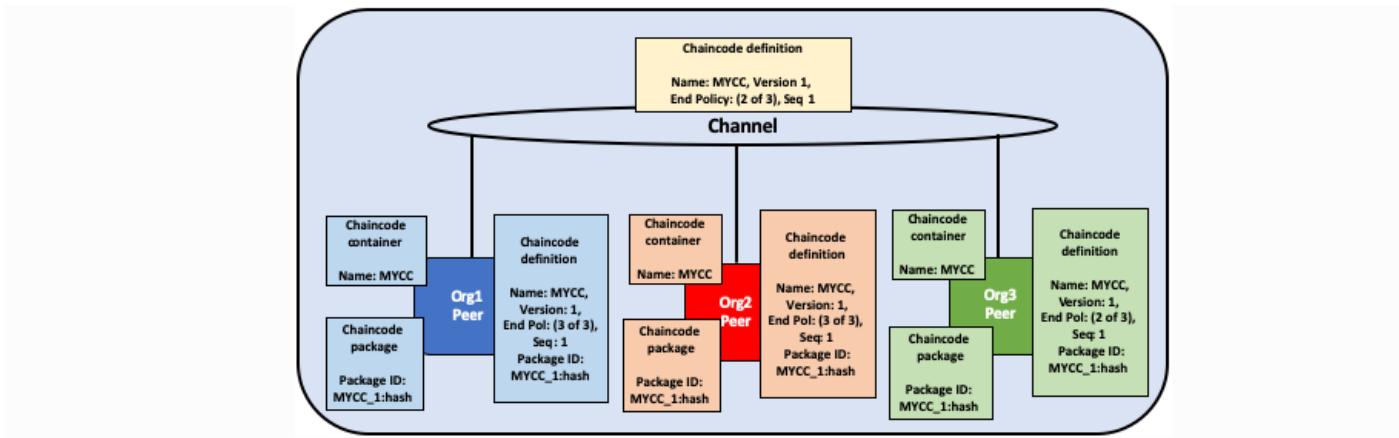
### 10.5.1 加入频道

新的组织可以使用已定义的链码加入频道，并在安装链码包并批准已经提交给该频道的链码定义后开始使用链码。



Org3 加入频道并批准先前由 Org1 和 Org2 提交给频道的相同链码定义。

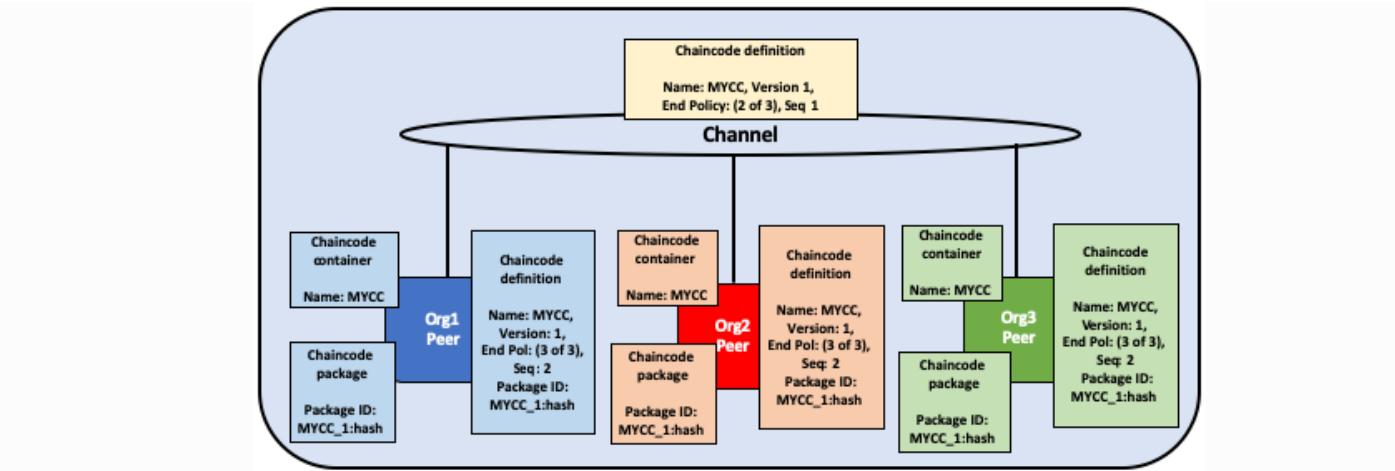
批准链码定义后，新组织可以在将软件包安装到对等方后开始使用链码。该定义不需要再次提交。如果将背书策略设置为默认策略，要求大多数渠道成员进行背书，则背书策略将自动更新以包括新组织。



链码容器将在 Org3 对等体上首次调用链码后启动。

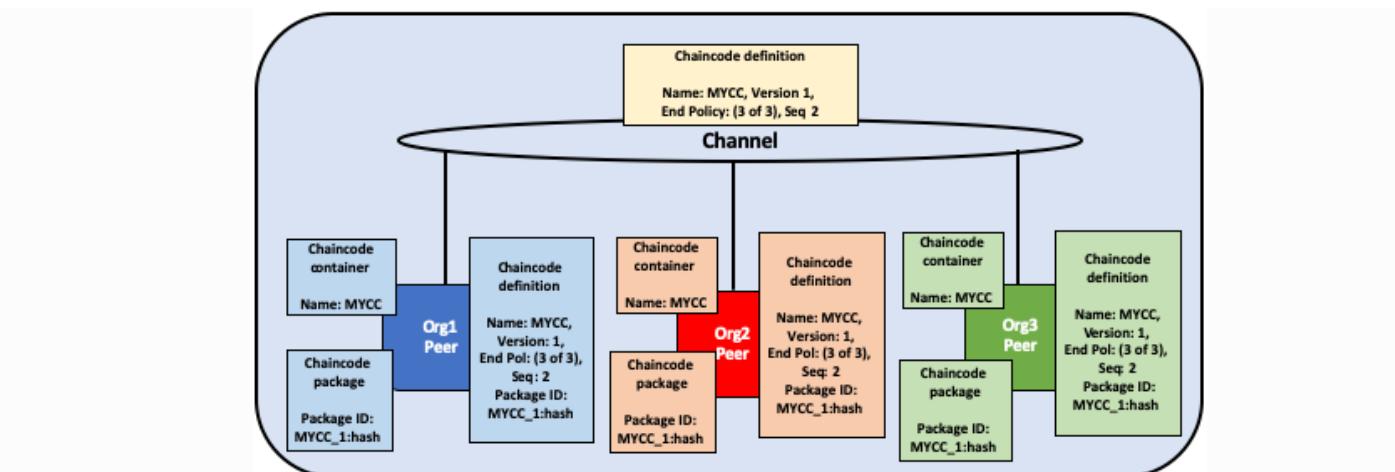
### 10.5.2 更新背书政策

您可以使用链码定义来更新背书策略，而不必重新打包或重新安装链码。渠道成员可以批准带有新认可策略的链码定义，并将其提交给渠道。



Org1, Org2 和 Org3 批准了一项新的认可政策，要求所有三个组织都认可一项交易。它们将定义序列从一递增到两，但不需要更新链码版本。

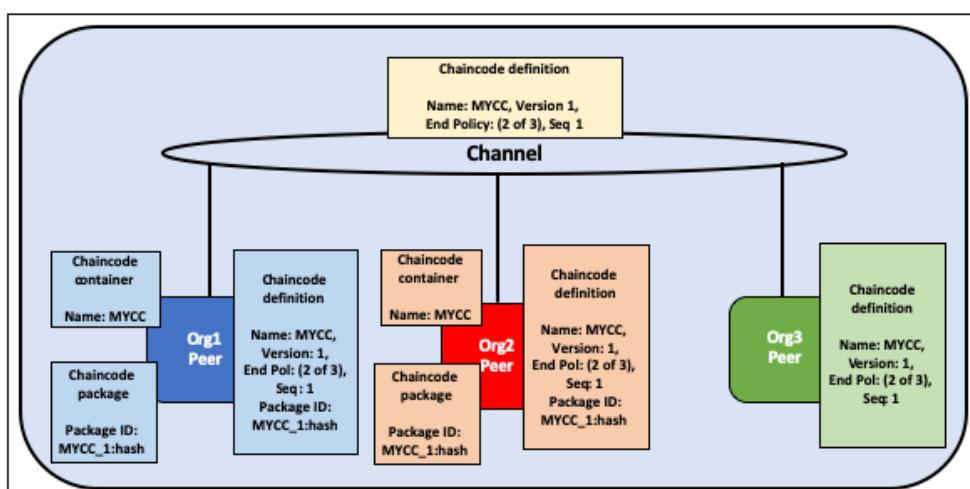
新的认可政策将在新定义提交到渠道后生效。通道成员不必通过调用链码或执行 `Init` 功能来重启链码容器即可更新背书策略。



一个组织将新的链码定义提交给渠道以更新认可策略。

### 10.5.3 批准定义而不安装链码

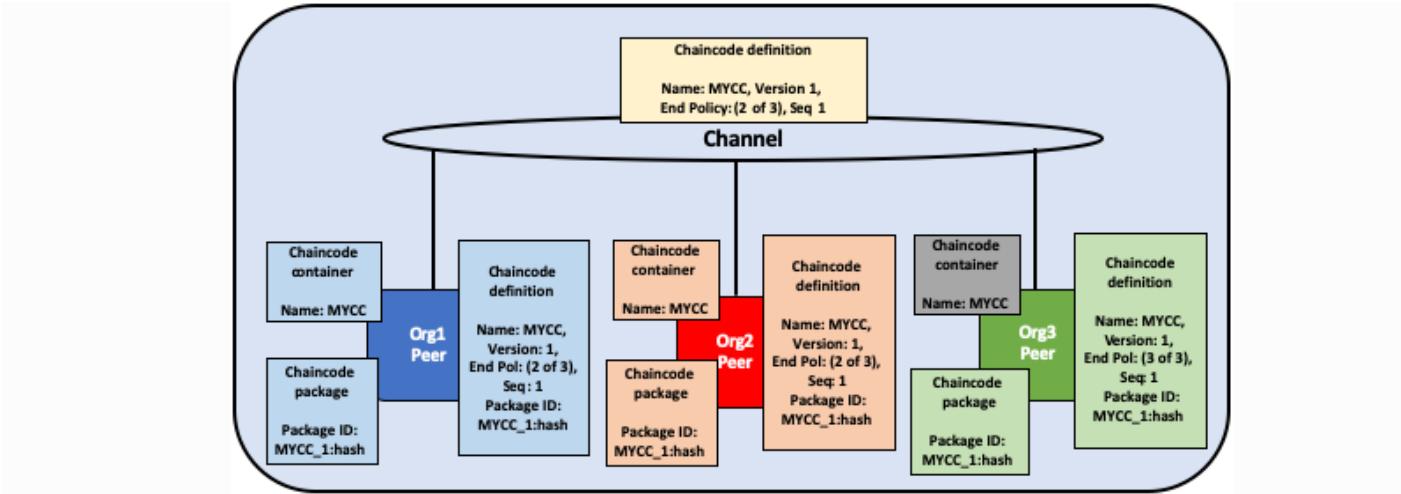
您可以批准链码定义而不安装链码包。这使您可以在将链码定义提交到通道之前对其进行背书，即使您不想使用该链码对交易进行背书或查询分类帐。您需要批准与通道的其他成员相同的参数，但不需要将 packageID 包含在链码定义中。



Org3 不会安装 chaincode 软件包。结果，他们不需要提供 packageID 作为链码定义的一部分。但是，Org3 仍然可以认可已提交给该频道的 MYCC 的定义。

#### 10.5.4一个组织不同意链码定义

不批准已提交给渠道的链码定义的组织不能使用链码。未批准链码定义或批准其他链码定义的组织将无法在其对等方上执行链码。

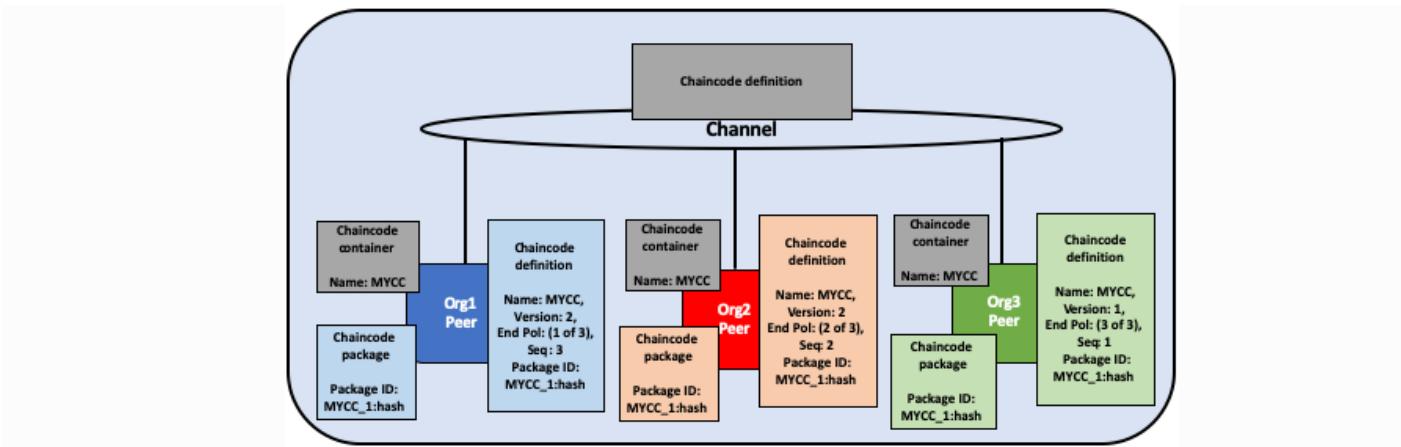


Org3 批准的链码定义具有与 Org1 和 Org2 不同的认可策略。结果，Org3 无法在通道上使用 MYCC 链码。但是，Org1 或 Org2 仍然可以获得足够的认可，以将定义提交到通道并使用链码。链码中的交易仍将添加到分类帐中并存储在 Org3 对等项中。但是，Org3 无法支持交易。

组织可以批准具有任何序列号或版本的新链码定义。这使您可以批准已提交给通道的定义并开始使用链码。您也可以批准新的链码定义，以更正在批准或打包链码过程中犯的任何错误。

#### 10.5.5频道在链码定义上不一致

如果渠道上的组织不同意链码定义，则无法将该定义提交给渠道。任何频道成员都将无法使用链码。

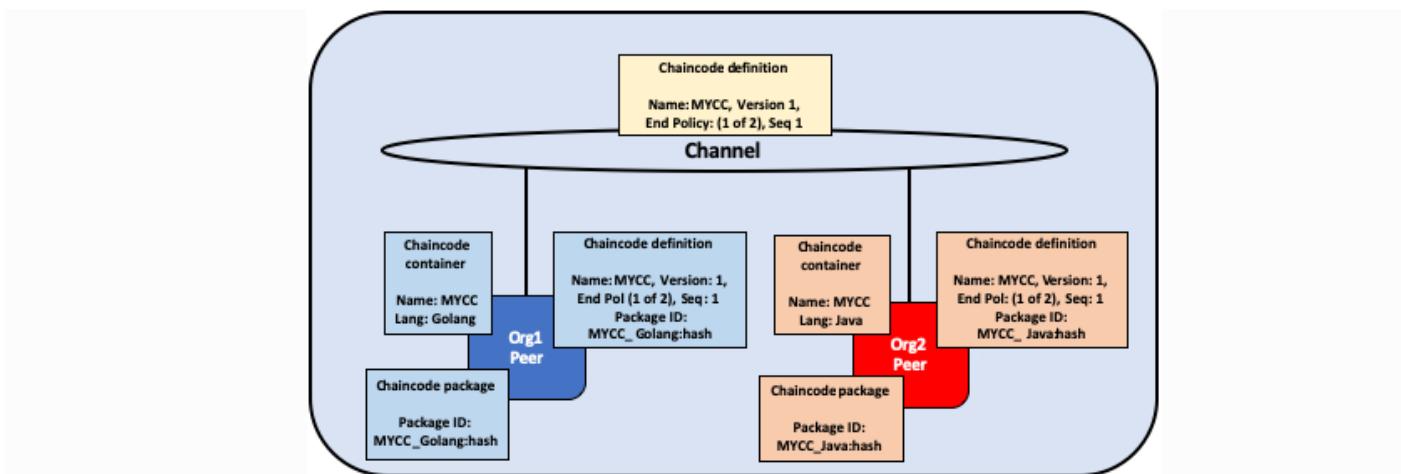


Org1, Org2 和 Org3 都认可不同的链码定义。结果，该频道的任何成员都无法获得足够的认可以将链码定义提交给该频道。没有频道成员将能够使用链码。

#### 10.5.6组织安装不同的 Chaincode 包

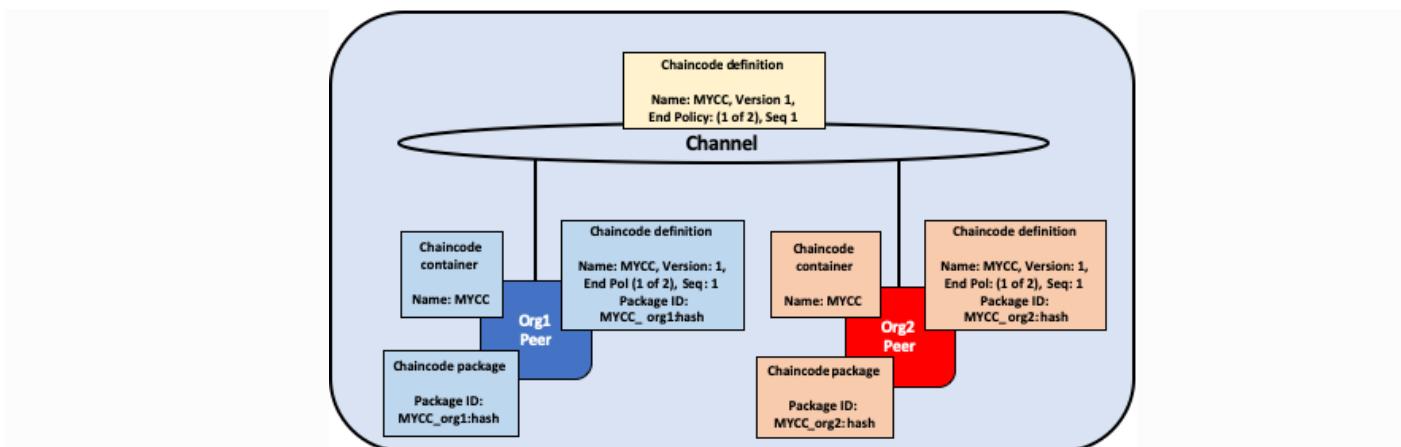
每个组织在批准链码定义时都可以使用不同的 packageID。这允许通道成员安装使用相同签注策略的不同链代码二进制文件，并在同一链代码名称空间中读取和写入数据。

渠道成员可以使用此功能来安装用不同语言编写的链码，并使用他们最喜欢的语言来工作。只要链码生成相同的读写集，使用不同语言的链码的通道成员将能够认可交易并将它们提交到分类账。但是，组织应在生产中的渠道上定义链码之前，先对其链码进行验证，并能够生成有效的认可。



Org1 安装以 Golang 编写的 MYCC 链代码的软件包，而 Org2 安装以 Java 编写的 MYCC 的软件包。

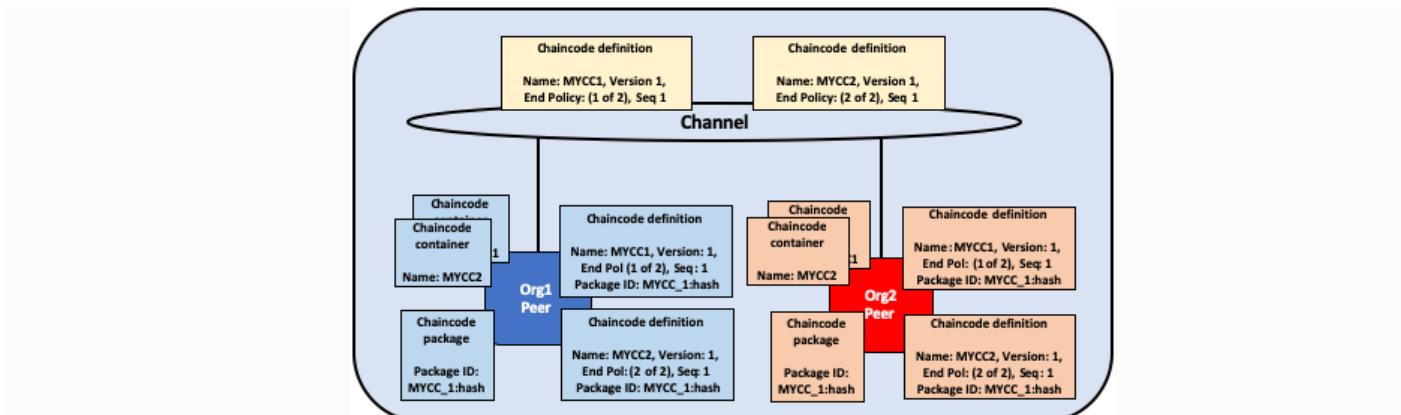
组织还可以使用此功能来安装包含特定于其组织的业务逻辑的智能合约。每个组织的智能合约都可以包含组织在其同级认可交易之前需要的其他验证。每个组织还可以编写代码，以帮助将智能合约与他们现有系统中的数据集成在一起。



Org1 和 Org2 每个都安装 MYCC 链码的版本，其中包含特定于其组织的业务逻辑。

## 10.5.7 使用一个包创建多个链码

您可以通过批准并提交多个链码定义，使用一个链码包在一个通道上创建多个链码实例。每个定义都需要指定一个不同的链码名称。这使您可以在一个通道上运行智能合约的多个实例，但是要让合约遵循不同的背书策略。



Org1 和 Org2 使用 MYCC\_1 链码包来批准和提交两个不同的链码定义。结果，两个对等方都有两个在其对等方上运行的链码容器。MYCC1 的背书政策为 2 分之一，而 MYCC2 的背书政策为 2 分之 2。

## 10.6 迁移到新的 Fabric 生命周期

您可以通过创建新频道并将频道功能设置为来使用 Fabric 链码生命周期 [v2\\_0](#)。您将无法使用以前的生命周期在 [v2\\_0](#) 启用了功能的频道上安装，实例化或更新链码。没有对 v2.0 Alpha 发行版的升级支持，也没有从 Alpha 发行版到 v2.x 未来版本的预期升级支持。

## 11 使用 CouchDB

本教程将描述将 CouchDB 用作 Hyperledger Fabric 的状态数据库所需的步骤。到目前为止，您应该已经熟悉 Fabric 的概念，并已经探索了一些示例和教程。

### 注意

这些说明使用 Fabric v2.0 Alpha 版本中引入的新 Fabric 链码生命周期。如果您想使用以前的生命周期模型来使用带有链码的索引，请访问[使用 CouchDB](#)的 v1.4 版本。

本教程将指导您完成以下步骤：

1. 在 Hyperledger Fabric 中启用 CouchDB
2. 创建一个索引
3. 将索引添加到您的 chaincode 文件夹
4. 安装并定义 Chaincode
5. 查询 CouchDB 状态数据库
6. 使用最佳做法进行查询和索引
7. 通过分页查询 CouchDB 状态数据库
8. 更新索引
9. 删除索引

要更深入地了解 CouchDB，请参考 [CouchDB 作为状态数据库](#)，有关 Fabric 分类帐的更多信息，请参考 [Ledger](#) 主题。请遵循以下教程，详细了解如何在区块链网络中利用 CouchDB。

在整个教程中，我们将使用 [Marbles 示例](#) 作为用例，以演示如何将 CouchDB 与 Fabric 一起使用，并将 Marbles 部署到“[构建您的第一个网络 \(BYFN\)](#)”教程网络中。您应该已经完成了安装样本，二进制文件和 Docker 映像任务。但是，运行 BYFN 教程不是本教程的前提条件，而是在本教程中提供了必要的命令以使用网络。

## 11.1 为什么选择 CouchDB？

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，并且将链码数据存储为简单的键值对，并且仅支持键，键范围和组合键查询。CouchDB 是可选的备用状态数据库，当链码数据值建模为 JSON 时，它支持丰富查询。当您要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加灵活高效。CouchDB 是 JSON 文档数据存储区，而不是纯键值存储区，因此可以对数据库中文档的内容建立索引。

为了利用 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。设置网络之前，您必须决定使用 LevelDB 还是 CouchDB。由于数据兼容性问题，不支持将对等方从使用 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。如果混合使用 JSON 和二进制数据值，则仍然可以使用 CouchDB，但是只能根据键，键范围和组合键查询来查询二进制值。

## 11.2 在 Hyperledger Fabric 中启用 CouchDB

CouchDB 作为对等体与单独的数据库进程一起运行，因此在设置、管理和操作方面还有其他注意事项。[CouchDB](#) 的 docker 映像可用，我们建议将其与对等方在同一服务器上运行。您将需要为每个对等方设置一个 CouchDB 容器，并通过更改找到的配置 `core.yaml` 以指向 CouchDB 容器来更新每个对等方容器。该 `core.yaml` 文件必须位于环境变量 `FABRIC_CFG_PATH` 指定的目录中：

- 对于 docker 部署，`core.yaml` 已预先配置并位于对等容器 `FABRIC_CFG_PATH` 文件夹中。但是，在使用 docker 环境时，通常通过编辑 `docker-compose-couch.yaml` 来覆盖 `core.yaml` 来传递环境变量。
- 对于本机二进制部署，`core.yaml` 包含在发布工件分发中。

编辑的 `stateDatabase` 部分 `core.yaml`。指定 `CouchDB` 为 `stateDatabase` 并填写关联的 `couchDBConfig` 属性。有关配置 CouchDB 与 Fabric 配合使用的更多详细信息，请参见[此处](#)。以查看配置用于 CouchDB 的一个 `core.yaml` 文件的一个例子，检查 BYFN `docker-compose-couch.yaml` 在 `HyperLedger/fabric-samples/first-network` 目录中。

## 11.3 创建一个索引

为什么索引很重要？

索引允许查询数据库，而不必每次查询都检查每一行，从而使它们运行得更快，更有效。通常，为频繁出现的查询条件构建索引，从而可以更有效地查询数据。为了利用 CouchDB 的主要优点（对 JSON 数据执行丰富查询的能力），不需要索引，但是强烈建议使用索引以提高性能。另外，如果查询中需要排序，则 CouchDB 需要排序字段的索引。

注意

没有索引的富查询可以使用，但可能会在 CouchDB 日志中引发警告，提示找不到索引。但是，如果富查询包含排序规范，则需要该字段的索引；否则，查询将失败并且将引发错误。

为了演示构建索引，我们将使用 [Marbles 样本中](#)的数据。在此示例中，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"`
}
```

在此结构中，属性 (`docType`, `name`, `color`, `size`, `owner`) 定义与资产相关联的分类账数据。该属性 `docType` 是链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，建议包括此 `docType` 属性以区分 chaincode 名称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密钥命名空间。）

关于 Marbles 数据结构，`docType` 用于标识此文档/资产是 Marbles 资产。链码数据库中可能还会有其他文档/资产。可针对所有这些属性值搜索数据库中的文档。

定义在链码查询中使用的索引时，每个索引都必须在其自己的文本文件中定义，扩展名为 `*.json`，并且索引定

义的格式必须为 CouchDB 索引 JSON 格式。

要定义索引，需要三项信息：

- **字段**: 这些是经常查询的字段
- **名称**: 索引名称
- **类型**: 在此上下文中始终为 json

例如，为名为 `foo-index` 的字段命名的简单索引 `foo`。

```
{  
  "index": {  
    "fields": ["foo"]  
  },  
  "name": "foo-index",  
  "type": "json"  
}
```

可选地，`ddoc` 可以在索引定义上指定设计文档属性。一个设计文档是 CouchDB 的结构设计，包含索引。可以将索引分组到设计文档中以提高效率，但是 CouchDB 建议每个设计文档使用一个索引。

提示：

定义索引时，最好将 `ddoc` 属性和值以及索引名称包括在内。包含此属性很重要，以确保您以后可以根据需要更新索引。它还使您能够显式指定要在查询中使用的索引。

这是来自 Marbles 样本的索引定义的另一个示例，其中索引名称 `indexOwner` 使用多个字段 `docType`，`owner` 并且包括 `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"  
}
```

在上面的示例中，如果设计文档 `indexOwnerDoc` 尚不存在，则在部署索引时会自动创建它。可以使用在字段列表中指定的一个或多个属性来构造索引，并且可以指定属性的任意组合。一个属性可以存在于同一 `docType` 的多个索引中。在以下示例中，`index1` 仅包括属性 `owner`，`index2` 包括属性，并且包括属性。另外，请注意，遵循 CouchDB 建议的做法，每个索引定义都有其自己的值。`owner and color` `index3` `owner, color and size`

```
{  
  "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"  
}
```

```
"ddoc": "index3Doc", // (optional) Name of the design document in which the index will be created.  
  "name": "index3",  
  "type": "json"  
}
```

通常，您应该对索引字段建模以匹配查询过滤器和排序中将使用的字段。有关以 JSON 格式构建索引的更多详细信息，请参考 [CouchDB 文档](#)。

关于索引的最后一句话，Fabric 使用称为模式来照顾数据库中文档的索引。CouchDB 通常在下一次查询之前不会索引新的或更新的文档。通过在提交每个数据块后请求索引更新，Fabric 确保索引保持“热”状态。这样可以确保查询快速，因为它们不必在运行查询之前就对文档进行索引。每次将新记录添加到状态数据库时，此过程都会使索引保持最新并刷新。[index warming](#)

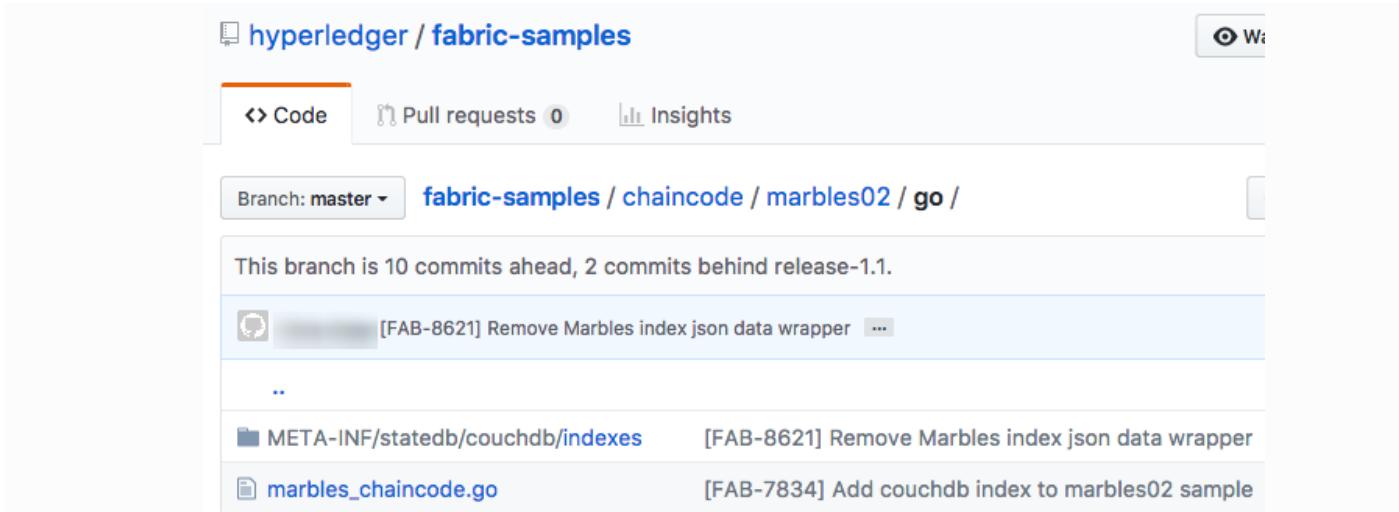
## 11.4 将索引添加到您的 chaincode 文件夹

最终确定索引后，可以将其与链码打包在一起，以将其放置在相应的元数据文件夹中，以进行部署。

如果您的链码安装和实例化使用 Hyperledger Fabric Node SDK，则 JSON 索引文件可以位于任何文件夹中，只要它符合此目录结构即可。在使用 `client.installChaincode()` API 安装链码期间，请 `metadataPath` 在安装请求中包含属性 ()。`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。

另外，如果您使用 `peer` 命令安装和实例化链码，则 JSON 索引文件必须位于 `META-INF/statedb/couchdb/indexes` 链码所在目录内的路径下。

下面的 [Marbles 示例](#) 说明了如何将索引与将使用对等命令安装的链码打包在一起。



此示例包括一个名为 `indexOwnerDoc` 的索引：

```
{"index": {"fields": ["docType", "owner"]}, "ddoc": "indexOwnerDoc",  
"name": "indexOwner", "type": "json"}
```

### 11.4.1 启动网络

#### 自己尝试

在安装和实例化 Marbles 链代码之前，我们需要启动 BYFN 网络。出于本教程的考虑，我们希望在已知的初始状态下进行操作。以下命令将杀死所有活动或陈旧的 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
```

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
```

这将创建一个简单的 Fabric 网络，该网络由一个名为 `mychannel` 的通道组成，该通道具有两个组织（每个组织维护两个对等节点）和一个排序服务，同时使用 CouchDB 作为状态数据库。

## 11.5 安装并定义 Chaincode

客户端应用程序通过链码与区块链分类账进行交互。因此，我们需要在每个将执行并认可我们交易的对等方上安装一个链码。但是，在我们与链码进行交互之前，渠道成员需要就建立链码治理的链码定义达成共识。在上一节中，我们演示了如何将索引添加到 chaincode 文件夹，以便可以进行部署。

链码需要先打包，然后才能安装在我们的同级上。我们可以使用 `peer lifecycle chaincode package` 命令来打包 Marbles 链代码。

自己尝试

假设您已启动 BYFN 网络，请输入 CLI 容器：

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

您的命令提示符将类似于以下内容：

```
bash-4.4#
```

1. 使用以下命令将 git 存储库中的 Marbles 链代码打包在本地容器中。

```
peer lifecycle chaincode package marbles.tar.gz --path github.com/hyperledger/fabric-samples/chaincode/marbles02/go --lang golang --label marbles_1
```

此命令将创建一个名为 `marbles.tar.gz` 的链码包。

2. 使用以下命令将 chaincode 软件包安装到 `peer0.org1.example.com` 您的 BYFN 网络中的对等方。默认情况下，启动 BYFN 网络后，活动对等方设置为 `CORE_PEER_ADDRESS=peer0.org1.example.com:7051`：

```
peer lifecycle chaincode install marbles.tar.gz
```

成功的安装命令将返回链码标识符，类似于以下响应：

```
2019-04-22 18:47:38.312 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 001 Installed remotely: response:<status:200 payload:"\nJmarbles_1:0907c1f3d3574afca69946e1b6132691d58c2f5c5703df7fc3b692861e92ecd3\022\tmarbles_1">
```

```
2019-04-22 18:47:38.312 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 002 Chaincode code package identifier: marbles_1:0907c1f3d3574afca69946e1b6132691d58c2f5c5703df7fc3b692861e92ecd3
```

在对等方上安装链码之后，我们需要批准组织的链码定义，并将链码定义提交给渠道。链码定义包括安装命令返回的软件包标识符。我们还可以使用对等生命周期链码 `queryinstalled` 命令查找的程序包 ID `marbles.tar.gz`。

3. 使用以下命令向对等方查询已安装链码的软件包 ID。

```
peer lifecycle chaincode queryinstalled
```

该命令将返回与安装命令相同的软件包标识符。您应该看到类似于以下内容的输出：

```
Installed chaincodes on peer:  
Package ID: marbles_1:0907c1f3d3574afca69946e1b6132691d58c2f5c5703df7fc3b692861e92ecd3, Label:  
marbles_1  
Package ID: mycc_1:27ef99cb3cbd1b545063f018f3670eddc0d54f40b2660b8f853ad2854c49a0d8, Label:  
mycc_1
```

4. 将程序包 ID 声明为环境变量。将命令返回的 `marbles_1` 的程序包 ID 粘贴到下面的命令中。软件包 ID 对于所有用户而言可能都不相同，因此您需要使用从控制台返回的软件包 ID 来完成此步骤。

```
peer lifecycle chaincode queryinstalled
```

```
export CC_PACKAGE_ID=marbles_1:0907c1f3d3574afca69946e1b6132691d58c2f5c5703df7fc3b692861e92ecd3
```

5. 使用以下命令批准 Org1 的 Marbles 链代码的定义。批准分布在每个组织内，因此该命令不需要针对组织的每个对等方。

```
export  
ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.co  
m/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem  
peer lifecycle chaincode approveformyorg --channelID mychannel --name marbles --version 1.0 --  
signature-policy "OR('Org1MSP.member','Org2MSP.member')" --init-required --package-id $CC_PACKAGE_ID  
--sequence 1 --tls true --cafle $ORDERER_CA
```

命令成功完成后，您应该会看到类似于以下内容：

```
2019-03-18 16:04:09.046 UTC [cli.lifecycle.chaincode] InitCmdFactory -> INFO 001 Retrieved  
channel (mychannel) orderer endpoint: orderer.example.com:7050
```

```
2019-03-18 16:04:11.253 UTC [chaincodeCmd] ClientWait -> INFO 002 txid  
[efba188ca77889cc1c328fc98e0bb12d3ad0abcd3f84da3714471c7c1e6c13c] committed with status (VALID) at
```

6.我们需要大多数组织批准链码定义，然后才能将其提交给渠道。这意味着我们还需要 Org2 批准链码定义。因为我们不需要 Org2 认可 chaincode，也不需要在任何 Org2 对等方上安装软件包，所以我们不需要提供 packageID 作为 chaincode 定义的一部分。

使用 CLI 切换到 Org2。将以下命令块作为一个组复制并粘贴到对等容器中，然后一次运行它们。

```
export CORE_PEER_ADDRESS=peer0.org2.example.com:9051  
export CORE_PEER_LOCALMSPID=Org2MSP  
export  
PEER0_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=$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
```

7.现在，您可以批准 Org2 的链码定义：

```
peer lifecycle chaincode approveformyorg --channelID mychannel --name marbles --version 1.0 --signature-policy "OR('Org1MSP.member','Org2MSP.member')" --init-required --sequence 1 --tls true --cafile $ORDERER_CA
```

一旦足够数量的组织(在本例中为大多数)批准了链码定义，则一个组织可以使用对等生命周期 chaincode commit 命令将该定义提交到通道。

在链代码定义已提交给通道之后，我们准备使用链代码。由于 Marbles 链代码包含初始化函数，因此在使用链 Init() 代码中的其他功能之前，需要使用对等生命周期 invoke 命令进行调用。

8.运行以下命令，将 Marbles 专用数据链代码的定义提交到 BYFN 通道 mychannel。此命令需要以 Org1 和 Org2 中的对等对象为目标，以收集提交事务的认可。

```
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  
ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt  
export  
ORG2_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt  
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID mychannel --name marbles --version 1.0 --sequence 1 --signature-policy "OR('Org1MSP.member','Org2MSP.member')" --init-required --tls true --cafile $ORDERER_CA --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles $ORG1_CA --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles $ORG2_CA
```

当提交事务成功完成时，您应该看到类似以下内容：

```
2019-04-22 18:57:34.274 UTC [chaincodeCmd] ClientWait -> INFO 001 txid  
[3da8b0bb8e128b5e1b6e4884359b5583dff823fce2624f975c69df6bce614614] committed with status (VALID) at  
peer0.org2.example.com:9051
```

```
2019-04-22 18:57:34.709 UTC [chaincodeCmd] ClientWait -> INFO 002 txid  
[3da8b0bb8e128b5e1b6e4884359b5583dff823fce2624f975c69df6bce614614] committed with status (VALID) at  
peer0.org1.example.com:7051
```

9.使用以下命令来调用 Init 函数以初始化链码。这将在上启动链码 peer0.org1.example.com:7051。

```
peer chaincode invoke -o orderer.example.com:7050 --channelID mychannel --name marbles --isInit -  
-tls true --cafile $ORDERER_CA --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles  
$ORG1_CA -c '{"Args":["Init"]}'
```

## 11.5.1 验证索引已部署

一旦链码同时安装在对等方并在通道上启动，索引将部署到每个对等方的 CouchDB 状态数据库。您可以通过检查 Docker 容器中的对等日志来验证 CouchDB 索引是否已成功创建。

自己尝试

要查看对等 Docker 容器中的日志，请打开一个新的 Terminal 窗口，然后运行以下命令 grep 以确认创建索引的

消息。

```
docker logs peer0.org1.example.com 2>&1 | grep "CouchDB index"
```

您应该看到如下所示的结果：

```
[couchdb] CreateIndex -> INFO 0be Created CouchDB index [indexOwner] in state database  
[mychannel_marbles] using design document [_design/indexOwnerDoc]
```

注意

如果未在 BYFN 对等方上安装 Marbles，则 `peer0.org1.example.com` 可能需要将其替换为安装了 Marbles 的其他对等方的名称。

## 11.6 查询 CouchDB 状态数据库

现在已经在 JSON 文件中定义了索引并与链码一起部署了索引，链码功能可以对 CouchDB 状态数据库执行 JSON 查询，因此对等命令可以调用链码功能。

在查询中指定索引名称是可选的。如果未指定，并且要查询的字段已经存在索引，则将自动使用现有索引。

小提示：

优良作法是使用 `use_index` 关键字在查询中显式包括索引名称。没有它，CouchDB 可能会选择不太理想的索引。另外，在测试过程中，CouchDB 可能根本不使用索引，并且您可能无法意识到索引的数量。仅在更高的卷上，您可能会意识到性能降低，因为 CouchDB 没有使用索引，而您假设已经使用了索引。

### 11.6.1 用链码建立查询

您可以使用链码中的 CouchDB JSON 查询语言对链码数据值执行复杂的丰富查询。如前所述，[marbles02 代码示例](#) 包含一个索引，并且在函数- `queryMarbles` 和中定义了丰富的查询 `queryMarblesByOwner`：

- **queryMarbles** –

**临时富查询的示例。**这是一个查询，其中（选择器）字符串可以传递到函数中。该查询对于需要在运行时动态构建自己的选择器的客户端应用程序很有用。有关选择器的更多信息，请参考 [CouchDB 选择器语法](#)。

- **queryMarblesByOwner** –

参数化查询的示例，其中查询逻辑包含在链码中。在这种情况下，该函数接受单个参数 Marbles 所有者。然后，它使用 JSON 查询语法在状态数据库中查询与 docType 为“ marble”和所有者标识匹配的 JSON 文档。

### 11.6.2 使用 peer 命令运行查询

在没有客户端应用程序测试链码中定义的丰富查询的情况下，可以使用对等命令。对等命令从 Docker 容器内部的命令行运行。我们将自定义对等的 `chaincode` 查询 命令以使用 Marbles 索引，`indexOwner` 并使用 `queryMarbles` 函数查询“ tom”拥有的所有 Marbles。

[自己尝试](#)

在查询数据库之前，我们应该添加一些数据。在对等容器中以 Org1 的身份运行以下命令，以创建“ tom”拥有的 Marbles：

```
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
export CORE_PEER_LOCALMSPID=Org1MSP
export
PEER0_ORG1_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
export CORE_PEER_TLS_ROOTCERT_FILE=$PEER0_ORG1_CA
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
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 marbles -c
'{"Args":["initMarble","marble1","blue","35","tom"]}'
```

After an index has been deployed when the chaincode is initialized, 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:

```
.. code:: bash

// Rich Query with index name explicitly specified:
peer chaincode query -C mychannel -n marbles -c '{"Args":["queryMarbles",
"\\"selector\":{\\\"docType\\\":\\\"marble\\\",\\\"owner\\\":\\\"tom\\\"}, \\\"use_index\\\":[_design/indexOwnerDoc\\",
\\\"indexOwner\\\"]]}'
```

深入研究上面的查询命令，需要关注三个参数：

- `queryMarbles`

Marbles 链码中的函数名称。注意，垫片 `shim.ChaincodeStubInterface` 用于访问和修改分类帐。在 `getQueryResultForQueryString()` 通过查询字符串的垫片 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"}}`

这是一个示例特设选择其中发现类型的所有文件的字符串 `marble`，其中 `owner` 属性具有值 `tom`。

- `"use_index": ["_design/indexOwnerDoc", "indexOwner"]`

同时指定设计文档名称 `indexOwnerDoc` 和索引名称 `indexOwner`。在此示例中，选择器查询显式包含通过使用 `use_index` 关键字指定的索引名称。回顾上面创建索引的索引定义，它包含一个设计文档 `"ddoc": "indexOwnerDoc"`。使用 CouchDB，如果您计划在查询中显式包括索引名称，则索引定义必须包含该 `ddoc` 值，以便可以用 `use_index` 关键字引用它。

查询成功运行，并且索引具有以下结果：

```
Query Result: [{"Key": "marble1",
"Record": {"color": "blue", "docType": "marble", "name": "marble1", "owner": "tom", "size": 35}}]
```

## 11.7 使用最佳做法进行查询和索引

使用索引的查询将更快地完成，而不必扫描 CouchDB 中的整个数据库。了解索引将使您能够编写查询以提高性能，并帮助您的应用程序处理网络上的大量数据或数据块。

计划使用链码安装的索引也很重要。每个链码只应安装几个支持大多数查询的索引。添加太多索引，或在索引中使用过多字段，都会降低网络性能。这是因为在提交每个块后都会更新索引。通过“索引预热”需要更新的索引越多，完成事务所需的时间就越长。

本节中的示例将帮助演示查询如何使用索引以及哪种类型的查询将具有最佳性能。编写查询时，请记住以下几点：

- 索引中的所有字段也必须在查询的选择器或排序部分中，才能使用索引。
- 更复杂的查询将具有较低的性能，并且不太可能使用索引。
- 你应该尽量避免运营商，这将导致全表扫描或全索引扫描，如 `$or`，`$in` 和 `$regex`。

在本教程的上一部分中，您对 Marbles 链代码发出了以下查询：

```
// Example one: query fully supported by the index
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles",
"\\"selector\\":{\\\"docType\\\":\\\"marble\\\",\\\"owner\\\":\\\"tom\\\"}, \\\"use_index\\\":[\\\"indexOwnerDoc\\\",
\\\"indexOwner\\\"]}"]}'
```

Marbles 链代码安装了 `indexOwnerDoc` 索引：

```
{"index": {"fields": ["docType", "owner"]}, "ddoc": "indexOwnerDoc",
"name": "indexOwner", "type": "json"}
```

请注意，查询中的 `docType` 和字段 `owner` 均包含在索引中，从而使其成为完全受支持的查询。结果，该查询将能够使用索引中的数据，而不必搜索整个数据库。像这样的完全受支持的查询将比链码中的其他查询返回得更快。

如果您在上面的查询中添加了额外的字段，它仍将使用索引。但是，该查询将另外必须扫描索引数据中的额外字段，从而导致更长的响应时间。例如，下面的查询仍将使用索引，但返回的时间将比前一个示例更长。

```
// Example two: query fully supported by the index with additional data
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles",
"\\"selector\\":{\\\"docType\\\":\\\"marble\\\",\\\"owner\\\":\\\"tom\\\",\\\"color\\\":\\\"red\\\"},
\\\"use_index\\\":[\\\"indexOwnerDoc\\\", \\\"indexOwner\\\"]}"]}'
```

不包含索引中所有字段的查询将不得不扫描整个数据库。例如，下面的查询搜索所有者，而不指定所拥有项目的类型。由于 `ownerIndexDoc` 包含 `owner` 和 `docType` 字段，因此该查询将无法使用索引。

```
// Example three: query not supported by the index
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles",
"\\"selector\\":{\\\"owner\\\":\\\"tom\\\"}, \\\"use_index\\\":[\\\"indexOwnerDoc\\\", \\\"indexOwner\\\"]}"]}'
```

通常，更复杂的查询将具有更长的响应时间，并且被索引支持的可能性较低。运营商如 `$or`，`$in` 和 `$regex` 往往会导致查询扫描完整的索引或不会使用索引的。

例如，下面的查询包含一个 `$or` 词，该词将搜索 `tom` 拥有的每块 Marbles 和物品。

```
// Example four: query with $or supported by the index
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles",
"\\"selector\\":{\\\"$or\\\":[{\\\"docType\\\":\\\"marble\\\"}, {\\\"owner\\\":\\\"tom\\\"}]},
\\\"use_index\\\":[\\\"indexOwnerDoc\\\", \\\"indexOwner\\\"]}"]}'
```

此查询仍将使用索引，因为它会搜索中包含的字段 `indexOwnerDoc`。但是，`$or` 查询中的条件要求扫描索引中的所有项目，从而导致更长的响应时间。

以下是索引不支持的复杂查询的示例。

```
// Example five: Query with $or not supported by the index
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarbles",
"\\"selector\\":{\\\"$or\\\":[{\\\"docType\\\":\\\"marble\\\",\\\"owner\\\":\\\"tom\\\"}, {\\\"color\\\":\\\"yellow\\\"}]},
\\\"use_index\\\":[\\\"indexOwnerDoc\\\", \\\"indexOwner\\\"]}"]}'
```

该查询将搜索 `tom` 拥有的所有 Marbles 或其他任何黄色项目。该查询将不使用索引，因为它将需要搜索整个表以满足 `$or` 条件。根据您分类帐中的数据量，此查询将需要很长时间才能响应或可能超时。

尽管遵循最佳做法进行查询很重要，但是使用索引并不是收集大量数据的解决方案。区块链数据结构经过优化以验证和确认交易，不适合数据分析或报告。如果要构建仪表板作为应用程序的一部分或分析网络中的数据，最佳做法是查询可复制对等方数据的链下数据库。这将使您能够理解区块链上的数据，而不会降低网络性能或中断交易。

您可以使用应用程序中的阻止或链码事件将事务数据写入链下数据库或分析引擎。对于接收到的每个块，块侦

听器应用程序将循环访问块事务，并使用每个有效事务的键/值写入来构建数据存储 `rwset`。基于对等通道的事件服务提供可重播事件，以确保下游数据存储的完整性。有关如何使用事件侦听器将数据写入外部数据库的示例，请访问结构样本中的脱链数据样本。

## 11.8 通过分页查询 CouchDB 状态数据库

当 CouchDB 查询返回大型结果集时，可以使用一组 API，这些 API 可以由链码调用以对结果列表进行分页。分页提供了一种机制，可通过指定一个 `pagesize` 和起始点—一个 `bookmark`（表示从何处开始结果集）来对结果集进行分区。客户端应用程序迭代地调用执行查询的链码，直到没有更多结果返回为止。有关更多信息，请参阅[有关使用 CouchDB 进行分页的主题](#)。

我们将使用 [Marbles 示例](#) 函数 `queryMarblesWithPagination` 来演示如何在链码和客户端应用程序中实现分页。

- `queryMarblesWithPagination` –

**带有分页的临时丰富查询的示例。**这是一个查询，其中可以将（选择器）字符串传递到类似于上述示例的函数中。在这种情况下，`pageSize` 查询中还包括一个 `bookmark`。

为了演示分页，需要更多数据。本示例假定您已经从上方添加了 marble1。在对等容器中运行以下命令以创建“tom”拥有的另外四个 Marbles，以创建“tom”拥有的总共五个 Marbles：

自己尝试

```
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"]}'
```

除了上一个示例中用于查询的参数之外，`queryMarblesWithPagination` 还添加了 `pagesize` 和 `bookmark`。  
`PageSize` 指定每个查询要返回的记录数。这 `bookmark` 是一个“anchor”，告诉 bedDB 从哪里开始页面。（每页结果返回一个唯一的书签。）

- `queryMarblesWithPagination`

Marbles 链码中的函数名称。注意，[垫片](#) `shim.ChaincodeStubInterface` 用于访问和修改分类帐。在 `getQueryResultForQueryStringWithPagination()` 与页面大小和书签垫片 API 一起传递查询字符串 `GetQueryResultWithPagination()`。

```
func (t *SimpleChaincode) queryMarblesWithPagination(stub shim.ChaincodeStubInterface, args []string) pb.Response {  
    //  
    // "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)
}

```

以下示例是一个对等命令，该命令使用 pageSize 为 3 且未指定书签来调用 queryMarblesWithPagination。

**小提示：**

如果未指定书签，则查询从记录的“第一”页开始。

### 自己尝试

```
// 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", ""]}'
```

下面的响应被接收（加入回车为了清楚起见），则返回 3 五个弹珠因为 pagsize 被设定为 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": "g1AAAABLeJzLYWBgYmpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkGoOkOWDSOSANIFk2iCyIyVySn5uVB
QAGEhRz"}}]
```

**注意**

书签由 CouchDB 为每个查询唯一生成，并在结果集中代表一个占位符。在查询的后续迭代中传递返回的书签，以检索下一组结果。

以下是一个对等命令，以 pageSize 为调用 queryMarblesWithPagination 3。请注意，这次查询包含了上一个查询返回的书签。

### 自己尝试

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarblesWithPagination",
"\\"selector\\":{\"docType\":\"marble\", \"owner\":\"tom\"}, \\"use_index\\\":[_design/indexOwnerDoc\",
\"indexOwner\"]}], "3", "g1AAAABLeJzLYWBgYmpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkGoOkOWDSOSANIFk2iCyIy
VySn5uVBQAGEhRz"]}'
```

收到以下响应（为清楚起见添加了回车）。检索到最后两个记录：

```
[{"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": "g1AAAABLeJzLYWBgYmpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkmoKkOWDSOSANIFk2iCyIy
VySn5uVBQAGYhR1"}}]
```

最后一个命令是一个对等命令，用于调用 queryMarblesWithPagination，它的 pageSize 为，3 并使用来自先前查询的书签。

### 自己尝试

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarblesWithPagination",
"\\"selector\\":{\"docType\":\"marble\", \"owner\":\"tom\"}, \\"use_index\\\":[_design/indexOwnerDoc\",
\"indexOwner\"]}], "3", "g1AAAABLeJzLYWBgYmpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkmoKkOWDSOSANIFk2iCyIy
VySn5uVBQAGYhR1"]}'
```

收到以下响应（为清楚起见添加了回车）。没有记录返回，表明已检索到所有页面：

```
[{"ResponseMetadata": {"RecordsCount": "0", "Bookmark": "g1AAAABLeJzLYWBgYMpgSmHgKy5JLCrJTq2MT8lPzkzJBYqz5yYWJeWkmoKkOWDSOSANIFk2iCyIyVySn5uVBQAGYhR1"}}]
```

有关客户端应用程序如何使用分页迭代结果集 [getQueryResultForQueryStringWithPagination](#) 的示例，请在 [Marbles](#) 示例中搜索该函数。

## 11.9 更新索引

随着时间的流逝，可能有必要更新索引。在安装的链码的后续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，然后从索引中添加或删除字段即可。Fabric 仅支持索引类型 JSON，不支持更改索引类型。安装并实例化链码后，更新的索引定义将重新部署到对等方的状态数据库。更改索引名称或 [ddoc](#) 属性将导致创建新索引，并且在删除之前，原始索引在 CouchDB 中保持不变。

注意

如果状态数据库中有大量数据，则重建索引将花费一些时间，在此期间，链码调用会导致查询失败或超时。

### 11.9.1 迭代索引定义

如果您可以在开发环境中访问对等方的 CouchDB 状态数据库，则可以迭代地测试各种索引以支持链码查询。但是，对链码的任何更改都需要重新部署。使用 [CouchDB Fauxton 界面](#) 或命令行 curl 实用程序来创建和更新索引。

注意

Fauxton 界面是一个 Web UI，用于创建索引，更新索引以及将索引部署到 CouchDB。如果要尝试使用此界面，请在 Marbles 示例中找到索引的 Fauxton 版本的格式示例。如果您已使用 CouchDB 部署了 BYFN 网络，则可以通过打开浏览器并导航到加载 Fauxton 接口 [http://localhost:5984/\\_utils](http://localhost:5984/_utils)。

另外，如果您不喜欢使用 Fauxton UI，下面是 curl 命令的示例，可用于在数据库上创建索引 [mychannel\\_marbles](#)：

// docType 的索引，所有者。//示例 curl 命令行，用于在 CouchDB channel\_chaincode 数据库中定义索引

```
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
```

注意

如果您正在使用配置了 CouchDB 的 BYFN，则将 hostname: port 替换为 [localhost:5984](#)。

## 11.10 删除索引

索引删除不是由 Fabric 工具管理的。如果需要删除索引，请对数据库手动执行 curl 命令或使用 Fauxton 界面删除它。

curl 命令用于删除索引的格式为：

```
curl -X DELETE http://localhost:5984/{database_name}/_index/{design_doc}/json/{index_name} -H
"accept: */*" -H "Host: localhost:5984"
```

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"
```