# 编写你的第一个应用

> **注解**
>
> 如果你对 Fabric 网络的基本架构还不熟悉，在继续本部分之前，你可能想先阅读 关键概念 部分。
>
> 本教程的价值仅限于介绍 Fabric 应用和使用简单的智能合约和应用。更深入的了解 Fabric 应用和智能合约请查看 开发应用 或 商业票据教程 部分。

本教程将介绍Fabric应用程序如何与已部署的区块链网络交互。 该教程使用Fabric SDKs构建的示例程序——详细阐述在 应用 专题中——来调用一个智能合约， 该合约使用智能合约API——详细阐述在 智能合约处理 中——来查询和更新账本。 我们还将使用我们的示例程序和已部署的证书颁发机构来生成X.509证书， 应用程序需要这些证书才能与许可性的区块链交互。

**关于 FabCar**

FabCar例子演示了如何查询保存在账本上的Car（我们业务对象例子），以及如何更新账本（向账本添加新的Car）。 它包含以下两个组件:

1. 示例应用程序：调用区块链网络，调用智能合约中实现的交易。
2. 智能合约：实现了涉及与账本交互的交易。

我们将按照以下三个步骤进行：

**1. 搭建开发环境。** 我们的应用程序需要和网络交互，所以我们需要一个智能合约和 应用程序使用的基础网络。

**2. 浏览一个示例智能合约。** 我们将查看示例智能合约 Fabcar 来学习他们的交易，还有应用程序是怎么使用他们来进行查询和更新账本的。

**3. 使用示例应用程序和智能合约交互。** 我们的应用程序将使用 FabCar 智能合约来查询和更新账本上的汽车资产。我们将进入到应用程序的代码和他们创建的交易，包括查询一辆汽车，查询一批汽车和创建一辆新车。

在完成这个教程之后，你将基本理解一个应用是如何通过编程关联智能合约来和 Fabric 网络上的多个节点的账本的进行交互的。

## 准备工作

除了Fabric的标准 准备阶段 之外，本教程还利用了Node.js对应的Hyperledger Fabric SDK。 有关最新的预备知识列表，请参阅Node.js SDK README 。

- 如果您正使用macOS系统, 请完成如下步骤:
  1. 安装 Homebrew 。

2. 检查Node SDK prerequisites 确认要安装的Node 版本
3. 运行命令 `brew install node` 以下载最新的node版本或者根据上述预备知识列表，选择具体被支持的版本，例如： `brew install node@10` 。
4. 运行命令 `npm install` 。

- 如果您是在Windows环境下，您可以通过npm安装 windows-build-tools ， 它会通过运行以下命令来安装所有需要的编译器和工具:

```
npm install --global windows-build-tools
```

- 如果您使用的是Linux，您需要安装 Python v2.7， make，和C/C++编译器工具链，如 GCC。可以执行如下命令安装其他工具:

```
sudo apt install build-essential
```

# 设置区块链网络

如果你已经学习了 使用Fabric的测试网络 而且已经运行起来了一个网络，本教程将在启动一个新网络之前关闭正在运行的网络。

## 启动网络

**注解**

这个教程演示了 Javascript 版本的 `FabCar` 智能合约和应用程序，但是 `fabric-samples` 仓库也包含 Go、Java 和 TypeScript 版本的样例。想尝试 Go、Java 或者 TypeScript 版本，改变下边的 `./startFabric.sh` 的 `javascript` 参数为 `go` 、 `java` 或者 `typescript` ，然后跟着介绍写到终端中。

进入你克隆到本地的 `fabric-samples` 仓库的 `fabcar` 子目录。

```
cd fabric-samples/fabcar
```

使用 `startFabric.sh` 脚本启动网络。

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

此命令将部署两个peer节点和一个排序节点以部署Fabric测试网络。 我们将使用证书颁发机构启动测试网络，而不是使用cryptogen工具。 我们将使用这些CA的其中一个来创建证书以及一些key， 这些加密资料将在之后的步骤中被我们的应用程序使用。 `startFabric.sh` 脚本还将部署和初始化在 `mychannel` 通道上的 FabCar智能合约的JavaScript版本，然后调用智能合约 来把初始数据存储在帐本上。

## 示例应用

FabCar的第一个组件，示例应用程序，适用于以下几种语言:

- Golang
- Java
- JavaScript
- Typescript

在本教程中，我们将阐释用 `javascript` 为nodejs编写的示例。

从 `fabric-samples/fabcar` 目录，进入到 `javascript` 文件夹。

```
cd javascript
```

该目录包含使用Node.js对应的Fabric SDK 开发的示例程序。运行以下命令安装应用程序依赖项。 这大约需要1分钟完成:

```
npm install
```

这个指令将安装应用程序的主要依赖，这些依赖定义在 `package.json` 中。其中最重要的是 `fabric-network` 类；它使得应用程序可以使用身份、钱包和连接到通道的网关，以及提交交易和等待通知。本教程也将使用 `fabric-ca-client` 类来注册用户以及他们的授权证书，生成一个 `fabric-network` 在后边会用到的合法身份。

完成 `npm install` ，运行应用程序所需要的一切就准备好了。让我们来看一眼教程中使用的示例 JavaScript 应用文件：

```
ls
```

你会看到下边的文件：

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

里边也有一些其他编程语言的文件，比如在 `fabcar/java` 目录中。当你使用过 JavaScript 示例之后，你可以看一下它们，主要的内容都是一样的。

# 登记管理员用户

**注解**

下边的部分执行和证书授权服务器通讯。你在运行下边的程序时，你会发现，打开一个新终端，并运行 `docker logs -f ca_org1` 来查看 CA 的日志流，会很有帮助。

当我们创建网络的时候，一个管理员用户（ `admin` ）被证书授权服务器（CA）创建成了 **注册员** 。我们第一步要使用 `enroll.js` 程序为 `admin` 生成私钥、公钥和 x.509 证书。这个程序使用一个 **证书签名请求** （CSR）——现在本地生成公钥和私钥，然后把公钥发送到 CA ，CA 会发布会一个让应用程序使用的证书。这三个证书会保存在钱包中，以便于我们以管理员的身份使用 CA 。

我们登记一个 `admin` 用户：

```
node enrollAdmin.js
```

这个命令将 CA 管理员的证书保存在 `wallet` 目录。 您可以在 `wallet/admin.id` 文件中找到管理员的证书和私钥。

## 注册和登记应用程序用户

既然我们的 `admin` 是用来与CA一起工作的。 我们也已经在钱包中有了管理员的凭据， 那么我们可以创建一个新的应用程序用户，它将被用于与区块链交互。 运行以下命令注册和记录一个名为 `appUser` 的新用户：

```
node registerUser.js
```

与admin注册类似，该程序使用CSR注册 `appUser` 并将其凭证与 `admin` 凭证一起存储在钱包中。 现在，我们有了两个独立用户的身份—— `admin` 和 `appUser` ——它们可以被我们的应用程序使用。

## 查询账本

区块链网络中的每个节点都拥有一个 账本 的副本，应用程序可以通过执行智能合约查询账本上最新的数据来实现来查询账本，并将查询结果返回给应用程序。

这里是一个查询工作如何进行的简单说明：

最常用的查询是查寻账本中询当前的值，也就是 世界状态 。世界状态是一个键值对的集合，应用程序可以根据一个键或者多个键来查询数据。而且，当键值对是以 JSON 值模式组织的时候，世界状态可以通过配置使用数据库（如 CouchDB ）来支持富查询。这对于查询所有资产来匹配特定的键的值是很有用的，比如查询一个人的所有汽车。

首先，我们来运行我们的 `query.js` 程序来返回账本上所有汽车的侦听。这个程序使用我们的第二个身份—— `user1` ——来操作账本。

```
node query.js
```

输入结果应该类似下边：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
[{"Key":"CAR0","Record":{"color":"blue","docType":"car","make":"Toyota","model":"Prius","owner":"
{"Key":"CAR1","Record":{"color":"red","docType":"car","make":"Ford","model":"Mustang","owner":"Br
{"Key":"CAR2","Record":{"color":"green","docType":"car","make":"Hyundai","model":"Tucson","owner"
{"Key":"CAR3","Record":{"color":"yellow","docType":"car","make":"Volkswagen","model":"Passat","ow
{"Key":"CAR4","Record":{"color":"black","docType":"car","make":"Tesla","model":"S","owner":"Adria
{"Key":"CAR5","Record":{"color":"purple","docType":"car","make":"Peugeot","model":"205","owner":"
{"Key":"CAR6","Record":{"color":"white","docType":"car","make":"Chery","model":"S22L","owner":"Aa
{"Key":"CAR7","Record":{"color":"violet","docType":"car","make":"Fiat","model":"Punto","owner":"P
{"Key":"CAR8","Record":{"color":"indigo","docType":"car","make":"Tata","model":"Nano","owner":"Va
{"Key":"CAR9","Record":{"color":"brown","docType":"car","make":"Holden","model":"Barina","owner":
```

让我们仔细看看 *query.js* 程序如何使用 Fabric Node SDK 提供的API与我们的Fabric网络交互。 使用一个编辑器（比如， atom 或 visual studio）打开 `query.js` 。

应用程序首先从 `fabric-network` 模块 引入两个key类： `Wallets` 和 `Gateway` 到scope中。 这些类将用于定位钱包中的 `appUser` 身份， 并使用它连接到网络:

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

首先，程序使用Wallet类从我们的文件系统获取应用程序用户。

```
const identity = await wallet.get('appUser');
```

一旦程序有了身份标识，它便会使用Gateway类连接到我们的网络。

```
const gateway = new Gateway();
await gateway.connect(ccpPath, { wallet, identity: 'appUser', discovery: { enabled: true, asLocal
```

`ccpPath` 描述了连接配置文件的路径， 我们的应用程序将使用该配置文件连接到我们的网络。 连接配置文件从 `fabric-samples/test-network` 目录中被加载进来， 并解析为JSON文件:

```
const ccpPath = path.resolve(__dirname, '..', '..', 'test-network','organizations','peerOrganizat
```

如果你想了解更多关于连接配置文件的结构，和它是怎么定义网络的，请查阅 链接配置主题 。

一个网络可以被差分成很多通道，代码中下一个很重的一行是将应用程序连接到网络中特定的通道 `mychannel` 上：

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

在这个通道中，我们可以通过 FabCar 智能合约来和账本进行交互：

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

在 `fabcar` 中有许多不同的 **交易** ，我们的应用程序先使用 `queryAllCars` 交易来查询账本世界状态的值：

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

`evaluateTransaction` 方法代表了一种区块链网络中和智能合约最简单的交互。它只是的根据配置文件中的定义连接一个节点，然后向节点发送请求，请求内容将在节点中执行。智能合约查询节点账本上的所有汽车，然后把结果返回给应用程序。这次交互没有导致账本的更新。

# FabCar 智能合约

FabCar智能合约示例有以下几种语言版本:

- Golang
- Java
- JavaScript
- Typescript

让我们来看看用JavaScript编写的FabCar智能合约中的交易。 打开一个新终端，并导航到 `fabric-samples` 仓库里 JavaScript版本的FabCar智能合约：

```
cd fabric-samples/chaincode/fabcar/javascript/lib
```

在文本编辑器中打开 `fabcar.js` 文件。

看看我们的智能合约是如何使用 `Contract` 类定义的

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

在这个类结构中，你将看到定义了以下交易： `initLedger` , `queryCar` , `queryAllCars` , `createCar` 和 `changeCarOwner` 。例如：

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

让我们更进一步看一下 `queryAllCars` ，看一下它是怎么和账本交互的。

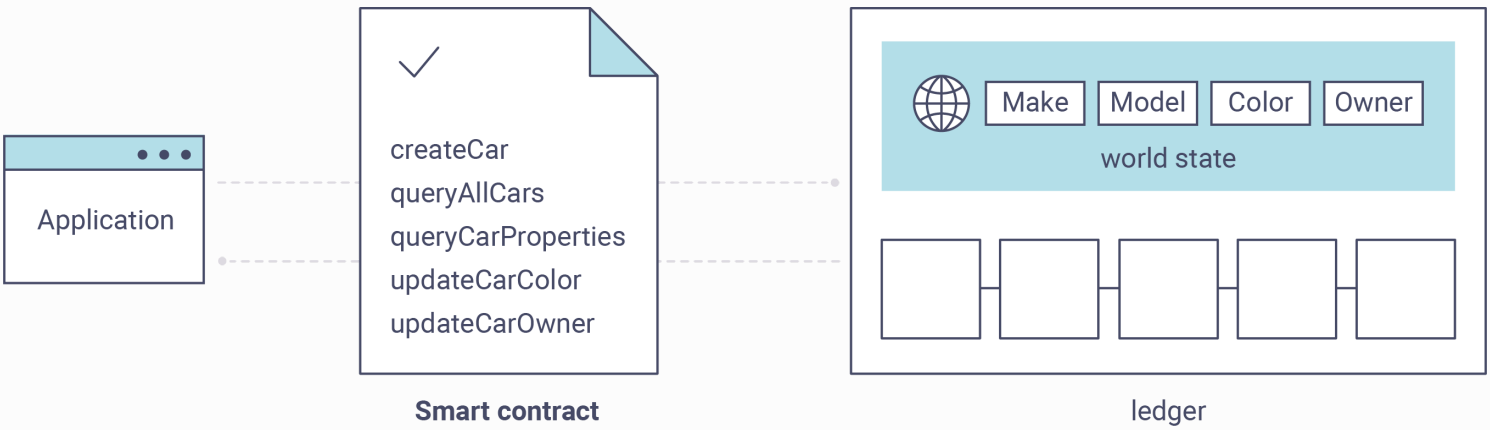
```
async queryAllCars(ctx) {

  const startKey = '';
  const endKey = '';

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

这段代码展示了如何使用 `getStateByRange` 在一个key范围内从账本中检索所有的汽车。 给出的空startKey和endKey将被解释为从起始到结束的所有key。 另一个例子是，如果您使用 `startKey = 'CAR0', endKey = 'CAR999'` ， 那么 `getStateByRange` 将以字典顺序检索在 ``CAR0``(包括自身) 和 ``CAR999``(不包括自身)之间key的汽车。 其余代码遍历查询结果，并将结果封装为JSON，以供示例应用程序使用。

下面展示了应用程序如何调用智能合约中的不同交易。每一个交易都使用一组 API 比如 `getStateByRange` 来和账本进行交互。了解更多 API 请阅读 detail.




你可以看到我们的 `queryAllCars` 交易，还有另一个叫做 `createCar` 。我们稍后将在教程中使用他们来更细账本，和添加新的区块。

但是在那之前，返回到 `query` 程序，更改 `evaluateTransaction` 的请求来查询 `CAR4` 。 `query` 程序现在看起来应该是这个样子：

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

保存程序，然后返回到 `fabcar/javascript` 目录。现在，再次运行 `query` 程序：

```
node query.js
```

你应该会看到如下：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been evaluated, result is:
{"color":"black","docType":"car","make":"Tesla","model":"S","owner":"Adriana"}
```

如果你回头去看一下 `queryAllCars` 的交易结果，你会看到 `CAR4` 是 Adriana 的黑色 Tesla model S，也就是这里返回的结果。

我们可以使用 `queryCar` 交易来查询任意汽车，使用它的键 （比如 `CAR0` ）得到车辆的制造商、型号、颜色和车主等相关信息。

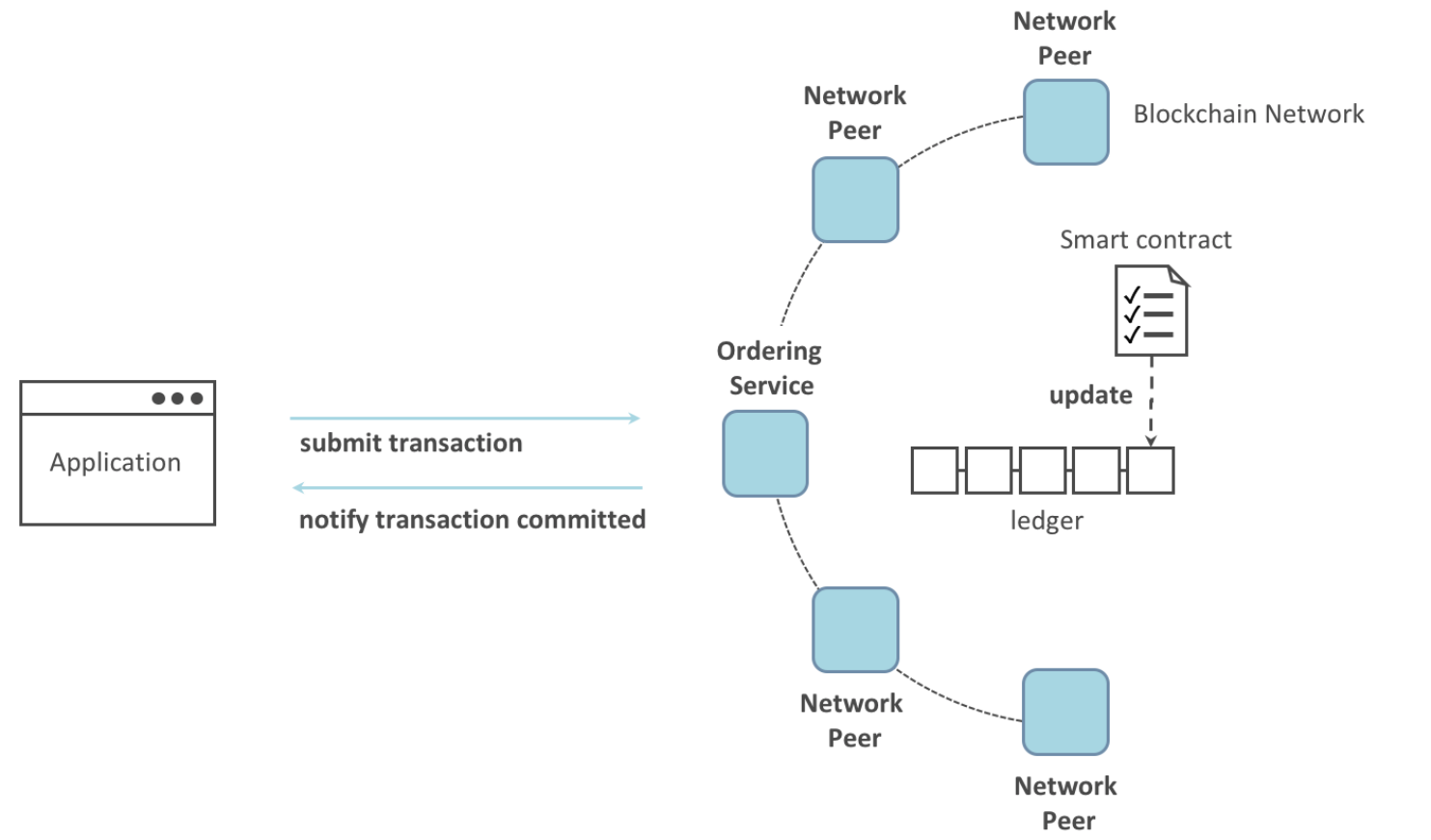
很棒。现在你应该已经了解了智能合约中基础的查询交易，也手动修改了查询程序中的参数。

账本更新时间。。。

## 更新账本

现在我们已经完成一些账本的查询和添加了一些代码，我们已经准备好更新账本了。有很多的更新操作我们可以做，但是我们从创建一个 **新** 车开始。

从一个应用程序的角度来说，更新一个账本很简单。应用程序向区块链网络提交一个交易，当交易被验证和提交后，应用程序会收到一个交易成功的提醒。但是在底层，区块链网络中各组件中不同的 **共识** 程序协同工作，来保证账本的每一个更新提案都是合法的，而且有一个大家一致认可的顺序。




上图中，我们可以看到完成这项工作的主要组件。同时，多个节点中每一个节点都拥有一份账本的副本，并可选的拥有一份智能合约的副本，网络中也有一个排序服务。排序服务保证网络中交易的一致性；它也将连接到网络中不同的应用程序的交易以定义好的顺序生成区块。

我们对账本的的第一个更新是创建一辆新车。我们有一个单独的程序叫做 `invoke.js` ，用来更新账本。和查询一样，使用一个编辑器打开程序定位到我们构建和提交交易到网络的代码段：

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

看一下应用程序如何调用智能合约的交易 `createCar` 来创建一量车主为 Tom 的黑色 Honda Accord 汽车。我们使用 `CAR12` 作为这里的键，这也说明了我们不必使用连续的键。

保存并运行程序：

```
node invoke.js
```

如果执行成功，你将看到类似输出：

```
Wallet path: ...fabric-samples/fabcar/javascript/wallet
Transaction has been submitted
```

注意 `inovke` 程序是怎样使用 `submitTransaction` API 和区块链网络交互的，而不是 `evaluateTransaction` 。

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

`submitTransaction` 比 `evaluateTransaction` 更加复杂。除了跟一个单独的 peer 进行互动外，SDK 会将 `submitTransaction` 提案发送给在区块链网络中的每个需要的组织的 peer。其中的每个 peer 将会使用这个提案来执行被请求的智能合约，以此来产生一个建议的回复，它会为这个回复签名并将其返回给 SDK。SDK 搜集所有签过名的交易反馈到一个单独的交易中，这个交易会被发送给排序节点。排序节点从每个应用程序那里搜集并将交易排序，然后打包进一个交易的区块中。接下来它会将这些区块分发给网络中的每个 peer，在那里每笔交易会被验证并提交。最后，SDK 会被通知，这允许它能够将控制返回给应用程序。

**注解**

`submitTransaction` 也包含一个监听者，它会检查来确保交易被验证并提交到账本中。应用程序应该使用一个提交监听者，或者使用像 `submitTransaction` 这样的 API 来给你做这件事情。如果不做这个，你的交易就可能没有被成功地排序、验证以及提交到账本。

应用程序中的这些工作由 `submitTransaction` 完成！应用程序、智能合约、节点和排序服务一起工作来保证网络中账本一致性的程序被称为共识，它的详细解释在这里 section 。

为了查看这个被写入账本的交易，返回到 `query.js` 并将参数 `CAR4` 更改为 `CAR12` 。

就是说，将：

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

改为：

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

再次保存，然后查询：

```
node query.js
```

应该返回这些：

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

恭喜。你创建了一辆汽车并验证了它记录在账本上！

现在我们已经完成了，我们假设 Tom 很大方，想把他的 Honda Accord 送给一个叫 Dave 的人。

为了完成这个，返回到 `invoke.js` 然后利用输入的参数，将智能合约的交易从 `createCar` 改为 `changeCarOwner` ：

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

第一个参数 `CAR12` 表示将要易主的车。第二个参数 `Dave` 表示车的新主人。

再次保存并执行程序：

```
node invoke.js
```

现在我们来再次查询账本，以确定 Dave 和 `CAR12` 键已经关联起来了：

```
node query.js
```

将返回如下结果：

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

`CAR12` 的主人已经从 Tom 变成了 Dave。

> Wallet path: ...fabric-samples/fabcar/javascript/wallet Transaction has been evaluated, result is: {“color”:”Black”,”docType”:”car”,”make”:”Honda”,”model”:”Accord”,”owner”:”Tom”}

**注解**

在真实世界中的一个应用程序里，智能合约应该有一些访问控制逻辑。比如，只有某些有权限的用户能够创建新车，并且只有车辆的拥有者才能够将车辆交换给其他人。

## 清除数据

当你完成FabCar示例的尝试后，您就可以使用 `networkDown.sh` 脚本关闭测试网络。

```
./networkDown.sh
```

该命令将关闭我们创建的网络的CA、peer节点和排序节点。 它还将删除保存在 `wallet` 目录中的 `admin` 和 `appUser` 加密资料。 请注意，帐本上的所有数据都将丢失。 如果您想再次学习本教程，您将会以初始状态的形式启动网络。

## 总结

现在我们完成了一些查询和跟新，你应该已经比较了解如何通过智能合约和区块链网络进行交互来查询和更新账本。我们已经看过了查询和更新的基本角智能合约、API 和 SDK ，你也应该对如何在其他的商业场景和操作中使用不同应用有了一些认识。

## 其他资源

就像我们在介绍中说的，我们有一整套文章在 开发应用 包含了关于智能合约、程序和数据设计的更多信息，一个更深入的使用商业票据的 教程 和大量应用开发的相关资料。