Hyperledger Fabric 开发——Node.js SDK与fabric链码交互

1、本篇背景

前面已经对链码开发作了比较详细的介绍，并且对官方提供的 `fabcar` 链码进行了解读，本篇将介绍如何使用 `Node.js SDK` 与区块链网络中的链码进行交互。

本篇内容基本来自官方 Hyperledger Fabric 文档中的 `Writing Your First Application` 章节，对文档进行翻译，原文网址如下：

```
http://hyperledger-fabric.readthedocs.io/en/latest/write_first_app.html
```

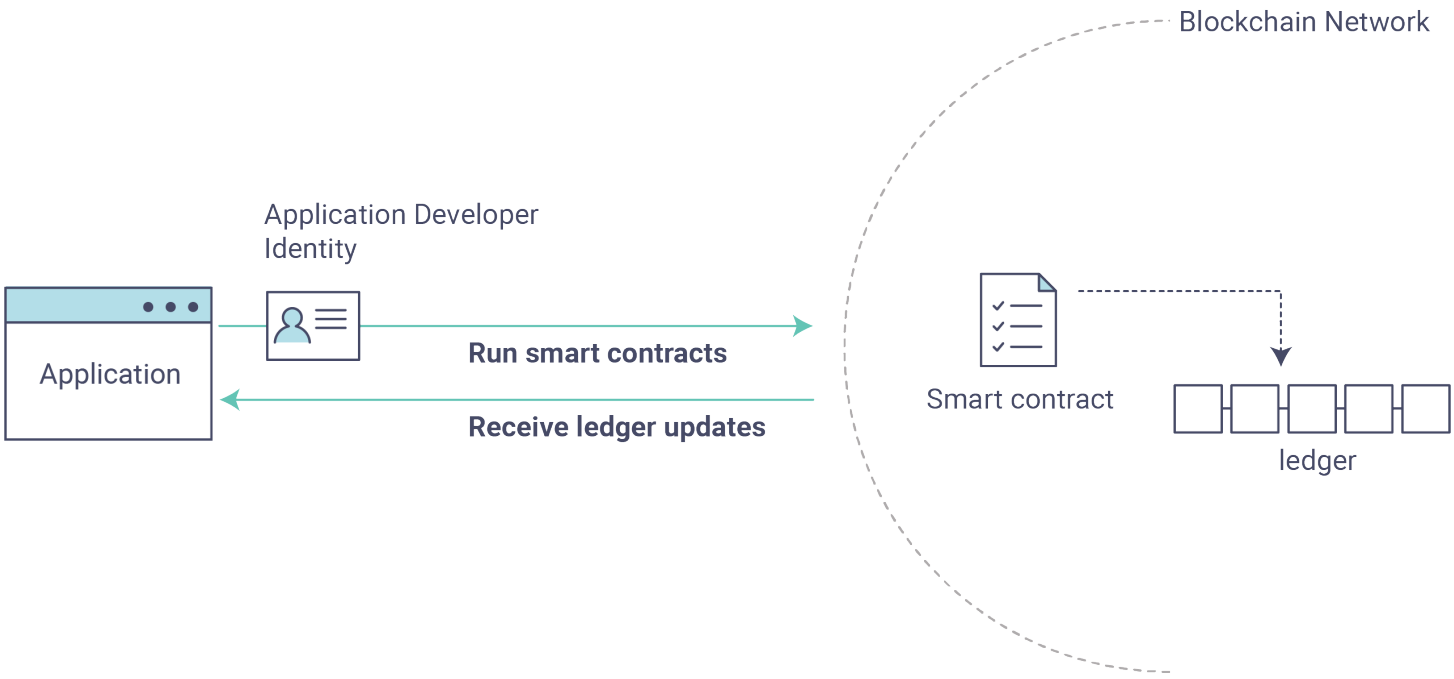
主要根据谷歌来翻译的，并稍微做一些修改。由于水平有限，最终的翻译质量不太好，欢迎大家拍砖。

2、编写你的第一个应用

> 提示：
> 如果您还不熟悉Fabric网络的基础架构，则可能需要在继续之前访问 简介 和 建立您的第一个网络 文档。

在本节中，我们将查看一些示例程序，以了解Fabric应用程序的工作方式。这些应用程序（以及他们使用的智能合约） - 统称为 `fabcar` --提供了Fabric功能的广泛演示。值得注意的是，我们将展示与证书颁发机构交互并生成注册证书的过程，之后我们将利用这些身份来查询和更新账本。

我们将通过三个主要步骤：
1.建立一个开发环境。我们的应用程序需要一个网络来进行交互，因此我们将下载一个简化为注册/登记，查询和更新所需的组件：



1.学习我们的应用将使用的示例智能合约的参数。我们的智能合约包含各种功能，使我们能够以不同的方式与账本进行交互。 我们将进入并检查该智能合约，以了解我们的应用程序将使用的功能。

3.开发应用程序以便能够查询和更新Fabric记录。 我们将自己进入应用程序代码（我们的应用程序已经使用JavaScript编写），并手动操作变量以运行不同类型的查询和更新。

完成本教程后，您应该基本了解如何将应用程序与智能合约一起编程，以便与Fabric网络上的账本（即节点peer）进行交互。

3、设置您的开发环境

如果您已经完成 建立您的第一个网络，您应该设置好您的开发环境，并下载好 `fabric-samples` 以及附带的工件。要运行本教程，您现在需要做的是移除您拥有的任何现有网络，您可以通过执行以下操作来完成此操作：

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

如果您没有开发环境以及网络和应用程序的附带工件，请访问 先决条件 页面，并确保您的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镜像 页面并按照提供的说明进行操作。

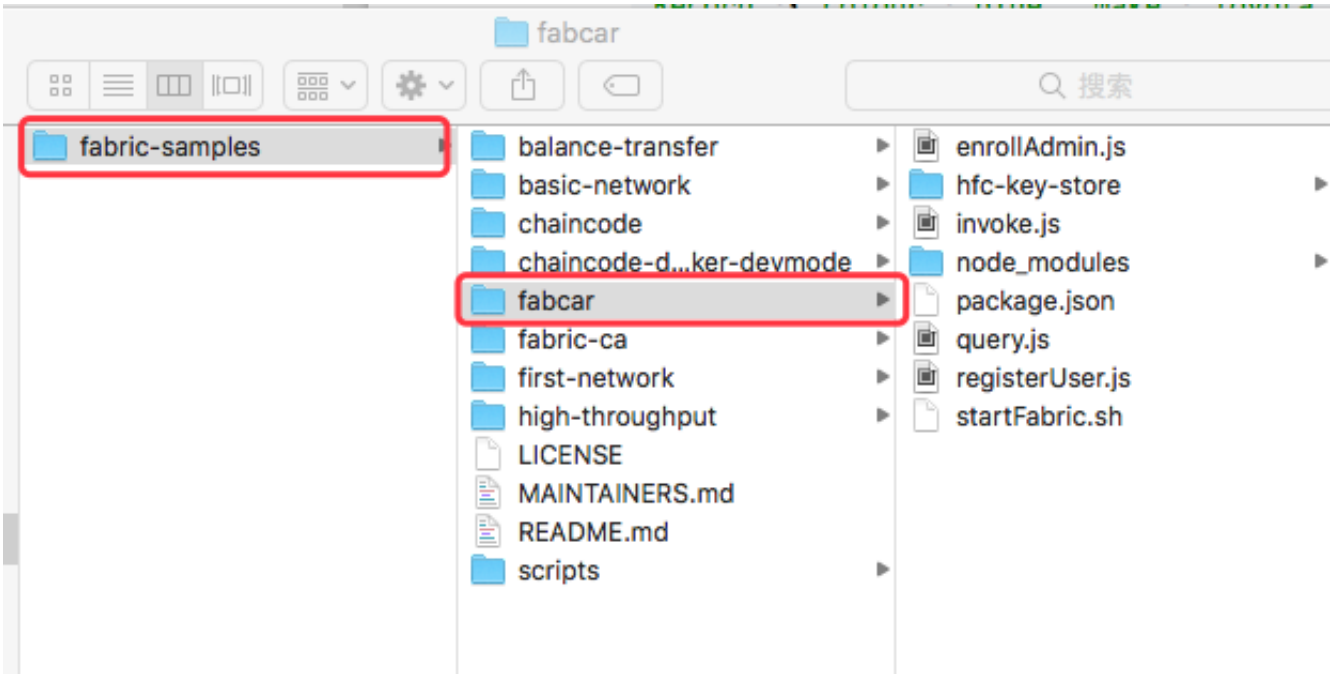
克隆 `fabric-samples` 库后，返回到本教程，并下载最新稳定版的Fabric镜像和可用的工具。

此时应该安装好了一切。进入到 `fabric-samples` 库中的 `fabcar` 子目录，并查看内部内容：

```
cd fabric-samples/fabcar  && ls
```

你应该看到以下内容：

```
enrollAdmin.js  invoke.js   package.json    registerUser.js
hfc-key-store   node_modules    query.js    startFabric.sh
```



在开始之前，我们还需要做一点准备工作。运行以下命令来杀死当前运行或者活跃的容器：

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

清除所有缓存网络：

```
# Press 'y' when prompted by the command
# 命令提示时请输入'y'

docker network prune
```

最后，如果您已经完成了本教程，您还需要删除 `fabcar` 智能合约的底层链码镜像。如果您是第一次浏览此内容的用户，那么您的系统上不会有此链接代码镜像：

```
docker rmi dev-peer0.org1.example.com-fabcar-1.0-5c906e402ed29f20260ae42283216aa75549c571e2e380f3615826365d8269ba
```

3.1 装客户端并启动网络

> 提示：
> 以下说明需要您在克隆在本地的 `fabric-samples` 库中的 `fabcar` 子目录中。在本教程接下来的部分，请确保在此子目录的根目录下。

运行以下命令为应用程序安装Fabric依赖库。我们关注 `fabric-ca-client` ，它将允许我们的应用程序与CA服务器进行通信并检索身份资料，以及使用 `fabric-client` ，它允许我们加载身份资料并与节点交互并订阅服务。

```
npm install
```

通过使用 `startFabric.sh` shell脚本启动您的网络。该命令将启动我们的各种Fabric工具，并启动用Golang编写的链码的智能合约容器：

```
./startFabric.sh
```

您还可以选择运行本教程，以针对使用Node.js编写的链码。如果您想追求这种方式，请改为执行以下命令：

```
./startFabric.sh node
```

> 提示：
> 请注意，Node.js链代码方案大约需要90秒才能完成或者更久;该脚本并未挂起，反而增加的时间是在构建链码镜像时安装了fabric-shim的结果。

好吧，现在你已经有了一个示例网络和一些代码，让我们来看看不同部分如何组合在一起。

4、应用程序如何与网络进行交互

要更深入地了解我们 `fabcar` 网络中的组件（以及它们如何部署）以及应用程序如何与更细粒度级别的组件进行交互，请参阅了解Fabcar网络。

开发者更感兴趣的是应用程序之间作了什么 - 以及查看代码本身以了解应用程序是如何构建的。目前，更需要了解的事情是，应用程序使用软件开发工具包（SDK）访问允许查询和更新账本的API。

5、注册管理员用户

> 提示：

> 以下两节涉及与证书颁发机构的通信。在运行即将推出的程序时，您可能会发现流式传输CA日志很有用。

要流式处理您的CA日志，拆分您的终端或打开一个新的shell并发出以下命令：

```
docker logs -f ca.example.com
```

现在返回到您的终端 `fabcar` 内容。。。

当我们启动我们的网络时，管理员用户 - `admin` - 已在我们的认证中心注册。现在我们需要向CA服务器发送一个注册呼叫，并为该用户检索注册证书（eCert）。我们不会在这里详细介绍注册的详细信息，但可以说SDK和扩展我们的应用程序需要此证书才能形成管理员的用户对象。然后我们将使用这个管理对象来注册并注册一个新用户。将管理员注册呼叫发送到CA服务器：

```
node enrollAdmin.js
```

该程序将调用证书签名请求（CSR），并最终将eCert和密钥材料输出到此项目的根目录中新创建的文件夹- `hfc-key-store` 中。然后，我们的应用程序将在他们需要为我们的各种用户创建或加载身份对象时查找此位置。

6、注册普通用户user1

使用我们新生成的管理员eCert，我们现在将再次与CA服务器进行通信，以注册和注册新用户。这个user - user1 - 将是我们在查询和更新账本时使用的身份。这里需要注意的一点是，为我们的新用户发布注册和注册呼叫的管理员身份（即，此用户正在扮演注册员的角色）。发送注册并为 `user1` 注册呼叫：

```
node registerUser.js
```

注：终端显示如下

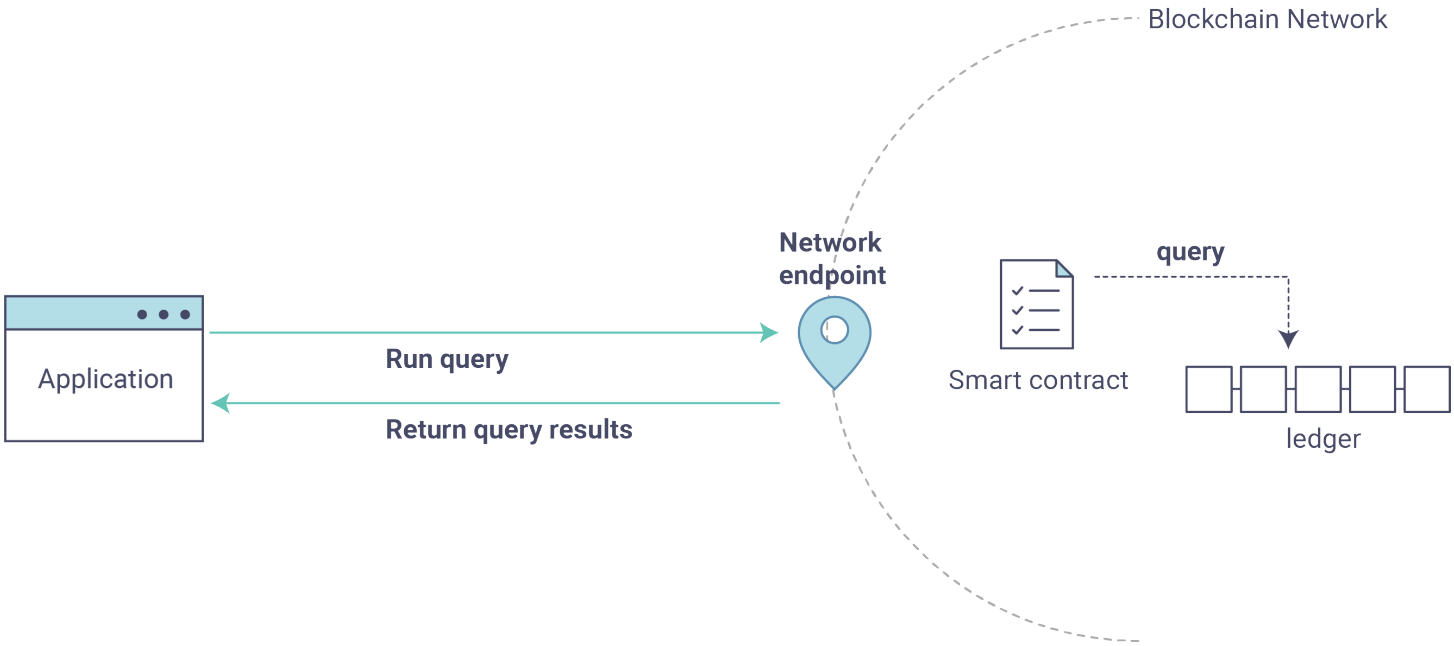
```
wenzildeiMac:fabcar wenzil$ node registerUser.js
Store path:/Users/wenzil/Desktop/study/fabric-samples/fabcar/hfc-key-store
Successfully loaded admin from persistence
Successfully registered user1 - secret:PBpPiXbnokEz
Successfully enrolled member user "user1"
User1 was successfully registered and enrolled and is ready to intreact with
 the fabric network
```

与管理员注册类似，此程序调用CSR并将密钥和eCert输出到 `hfc-key-store` 子目录中。所以现在我们拥有两个独立用户的身份资料 - `admin` 和 `user1` 。 与账本交互的时间。。。

7、查询账本

查询是指如何从账本中读取数据。这些数据存储为一系列键值对，您可以查询单个键，多个键的值，或者 - 如果账本是使用JSON等丰富的数据存储格式编写的，则可以对其执行复杂的搜索（例如，查找包含特定关键字的所有资产）。

这是查询如何工作的表示形式：



首先，让我们运行我们的 `query.js` 程序返回账本上所有汽车的列表。我们将使用我们的第二个身份 - user1 - 作为此应用程序的签名实体。我们程序中的以下行将user1指定为签名者：

```
fabric_client.getUserContext('user1', true);
```

回想一下， `user1` 注册资料已经被放入我们的 `hfc-key-store` 子目录，所以我们只需要告诉我们的应用程序来获取这个身份。通过定义用户对象，我们现在可以继续从账本中读取数据。查询所有汽车的函数 `queryAllCars` 在应用程序中预加载，所以我们可以简单地按照原样运行程序：

```
node query.js
```

它应该返回像这样的东西：

```
Successfully loaded user1 from persistence
Query has completed, checking results
Response is  [{"Key":"CAR0", "Record":{"colour":"blue","make":"Toyota","mode
```

```
l":"Prius","owner":"Tomoko"}},
{"Key":"CAR1",   "Record":{"colour":"red","make":"Ford","model":"Mustang","o
wner":"Brad"}},
{"Key":"CAR2", "Record":{"colour":"green","make":"Hyundai","model":"Tucson",
"owner":"Jin Soo"}},
{"Key":"CAR3", "Record":{"colour":"yellow","make":"Volkswagen","model":"Pass
at","owner":"Max"}},
{"Key":"CAR4", "Record":{"colour":"black","make":"Tesla","model":"S","owner"
:"Adriana"}},
{"Key":"CAR5", "Record":{"colour":"purple","make":"Peugeot","model":"205","o
wner":"Michel"}},
{"Key":"CAR6", "Record":{"colour":"white","make":"Chery","model":"S22L","own
er":"Aarav"}},
{"Key":"CAR7", "Record":{"colour":"violet","make":"Fiat","model":"Punto","ow
ner":"Pari"}},
{"Key":"CAR8", "Record":{"colour":"indigo","make":"Tata","model":"Nano","own
er":"Valeria"}},
{"Key":"CAR9", "Record":{"colour":"brown","make":"Holden","model":"Barina","
owner":"Shotaro"}}]
```


这是10辆车。 由Adriana拥有的黑色特斯拉Model S，由Brad拥有的红色Ford Mustang，由Pari拥有的紫色Fiat Punto等等。账本是基于关键值的，在我们的实施中，关键是 `CAR0` 到 `CAR9` 。这一点将变得特别重要。

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` 。

应用程序的初始部分定义了某些变量，例如通道名称，证书存储位置和网络端点。在我们的示例应用程序中，这些变量已被内置，但在真实应用程序中，这些变量必须由应用程序开发者指定。

```
var channel = fabric_client.newChannel('mychannel');
var peer = fabric_client.newPeer('grpc://localhost:7051');
channel.addPeer(peer);

var member_user = null;
var store_path = path.join(__dirname, 'hfc-key-store');
console.log('Store path:'+store_path);
var tx_id = null;
```

这是我们构建查询的代码块：

```
// queryCar chaincode function - requires 1 argument, ex: args: ['CAR4'],
// queryAllCars chaincode function - requires no arguments , ex: args: [''],
const request = {
  //targets : --- letting this default to the peers assigned to the channel
  chaincodeId: 'fabcar',
```

```
  fcn: 'queryAllCars',
  args: ['']
};
```

当应用程序运行时，它调用节点peer上的 `fabcar` 链码，在其中运行 `queryAllCars` 函数，并且不传递任何参数。

要查看我们智能合约中的可用功能，进入到 `fabric-samples` 根目录下的 `chaincode/fabcar/go` 子目录，然后在您的编辑器打开 `fabcar.go` 。

> 提示：
> 这些相同的功能在 `fabcar` 链码的Node.js版本中定义。

你会看到我们有以下功能可供调用： `initLedger` ， `queryCar` ， `queryAllCars` ， `createCar` 和 `changeCarOwner` 。

让我们仔细看看 `queryAllCars` 函数，看看它如何与账本交互。

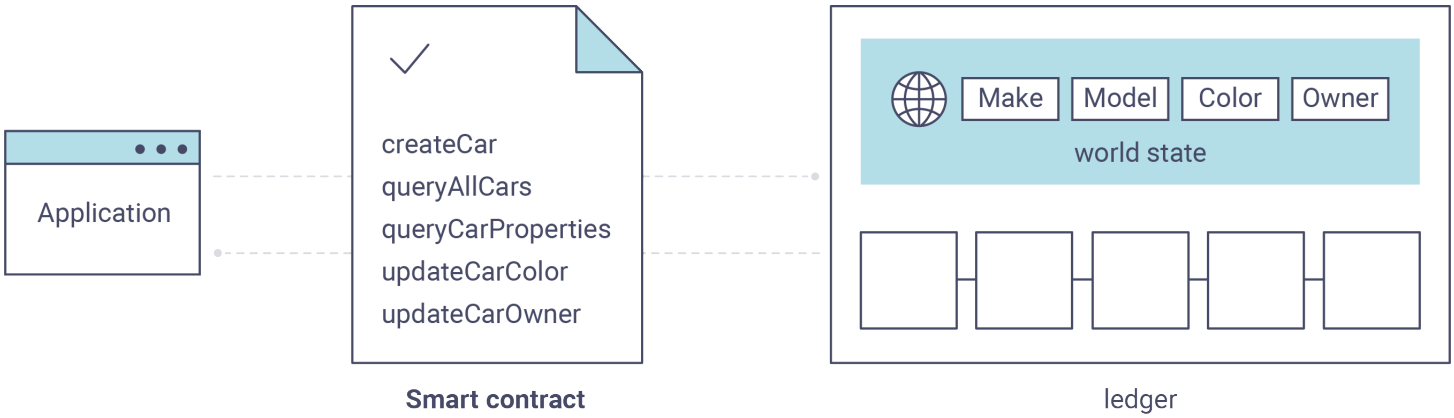
```
func (s *SmartContract) queryAllCars(APIstub shim.ChaincodeStubInterface) sc
.Response {

      startKey := "CAR0"
      endKey := "CAR999"

      resultsIterator, err := APIstub.GetStateByRange(startKey, endKey)
```

这定义了queryAllCars的范围。CAR0和CAR999之间的每辆车 - 总共1,000辆汽车，假设每个钥匙都被正确标记 - 将由查询返回。

以下是应用程序如何在链码中调用不同功能的表示形式。每个功能必须根据链码shim接口中的可用API进行编码，这反过来又允许智能合同容器与对等账本正确对接。



我们可以看到我们的 `queryAllCars` 函数以及一个叫做 `createCar` 的函数，它允许我们更新账本并最终在链中添加一个新块。

但首先，返回 `query.js` 程序并编辑构造函数请求以查询CAR4。我们通过将 `query.js` 中的函数从 `queryAllCars` 更改为 `queryCar` 并将CAR4作为特定键传递来实现此目的。

`query.js` 程序现在应该如下所示：

```
const request = {
  //targets : --- letting this default to the peers assigned to the channel
  chaincodeId: 'fabcar',
  fcn: 'queryCar',
  args: ['CAR4']
};
```

保存程序并返回到您的 `fabcar` 目录。 现在再次运行该程序：

```
node query.js
```

您将看到如下内容：

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

如果你回头看看我们之前查询过每辆车的结果，可以看到 `CAR4` 是Adriana的黑色特斯拉模型S，这是在这里返回的结果。

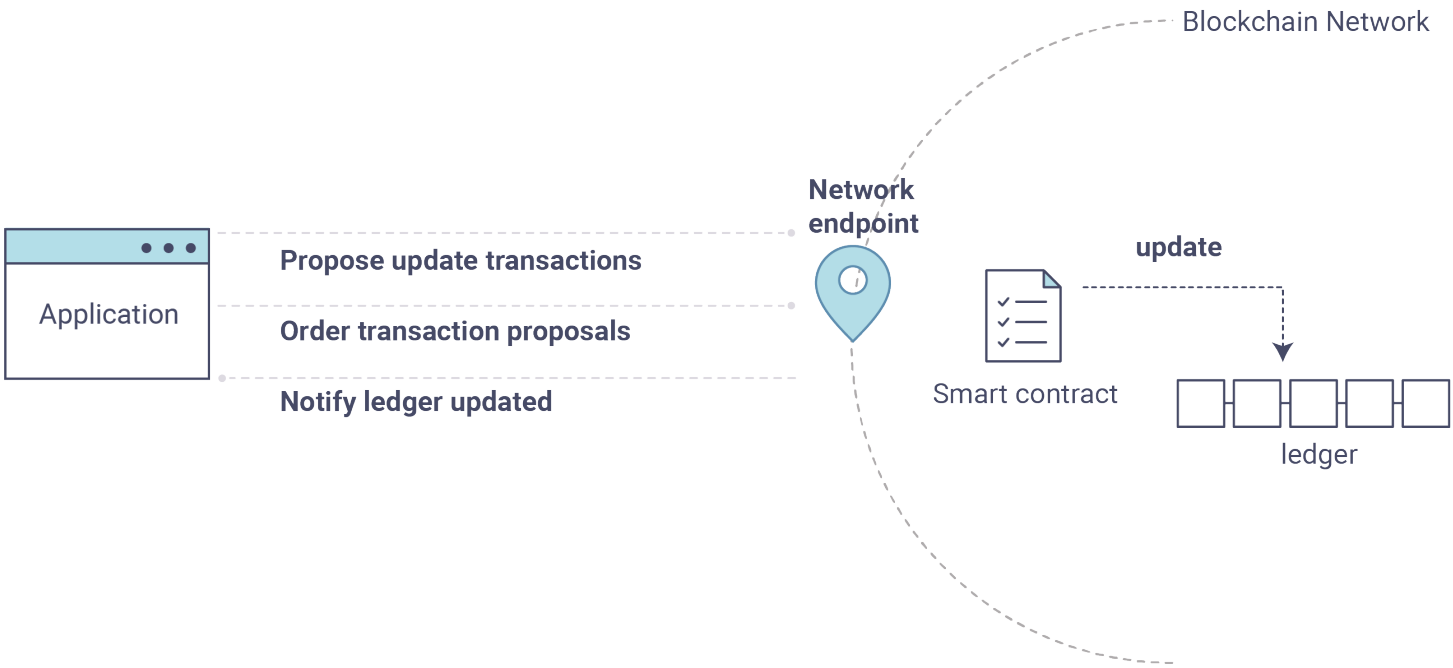
使用 `queryCar` 函数，我们可以查询任何关键字（例如 `CAR0` ）并获取与该车相对应的任何品牌，型号，颜色和所有者。

很好。此时，您应该熟悉智能合约中的基本查询功能以及查询程序中的少量参数。 时间更新账本。。。

8、更新账本

现在我们已经完成了几个账本查询并添加了一些代码，我们已经准备好更新账本。有很多潜在的更新我们可以做，但我们先创建一辆新车。

下面我们可以看到这个过程如何工作。提案更新，通过认可，然后返回到应用程序，然后将其发送到订单并写入每个节点peer账户：



我们对账本的第一次更新将是创造一辆新车。我们有一个单独的Javascript程序 - invoke.js - 我们将用它来进行更新。与查询一样，使用编辑器打开程序并进入到构建我们的调用的代码块：

```
// createCar chaincode function - requires 5 args, ex: args: ['CAR12', 'Hond
a', 'Accord', 'Black', 'Tom'],
// changeCarOwner chaincode function - requires 2 args , ex: args: ['CAR10',
 'Barry'],
// must send the proposal to endorsing peers
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
  fcn: '',
  args: [''],
  chainId: 'mychannel',
  txId: tx_id
};
```

您会看到我们可以调用两个函数之一 - `createCar` 或 `changeCarOwner` 。首先，让我们创建一个红色雪佛兰Volt并将其交给名为Nick的所有者。我们在账本上使用 `CAR9` ，因此我们将在此使用 `CAR10` 作为识别密钥。编辑代码如下：

```
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
  fcn: 'createCar',
  args: ['CAR10', 'Chevy', 'Volt', 'Red', 'Nick'],
  chainId: 'mychannel',
  txId: tx_id
};
```

保存并运行程序：

```
node invoke.js
```

终端中会有一些关于 `ProposalResponse` 和Promise的输出。 然而，我们所关心的只是这个信息：

```
The transaction has been committed on peer localhost:7053
```

要查看此事务已写入，请返回 `query.js` 并将参数从 `CAR4` 更改为 `CAR10` 。

换句话说，把这里：

```
const request = {
  //targets : --- letting this default to the peers assigned to the channel
  chaincodeId: 'fabcar',
  fcn: 'queryCar',
  args: ['CAR4']
};
```

改为：

```
const request = {
  //targets : --- letting this default to the peers assigned to the channel
  chaincodeId: 'fabcar',
  fcn: 'queryCar',
  args: ['CAR10']
};
```

再次保存，然后查询：

```
node query.js
```

这应该返回这个：

```
Response is  {"colour":"Red","make":"Chevy","model":"Volt","owner":"Nick"}
```

恭喜！您已经创建了一辆车！

所以，现在我们已经做到了，让我们说Nick很慷慨，他想把他的雪佛兰Volt交给一个名叫Dave的人。

要做到这一点，请返回到 `invoke.js` 并将函数从 `createCar` 更改为 `changeCarOwner` 并输入如下所示的参数：

```
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
  fcn: 'changeCarOwner',
  args: ['CAR10', 'Dave'],
  chainId: 'mychannel',
  txId: tx_id
};
```

第一个参数 - `CAR10` - 表明将改变车主的汽车。第二个参数 - `Dave` - 表明为汽车的新主人。

再次保存并执行该程序：

```
node invoke.js
```

现在，让我们再次查询账本并确保Dave现在与 `CAR10` 键相关联：

```
node query.js
```

它应该返回这个结果：

```
Response is  {"colour":"Red","make":"Chevy","model":"Volt","owner":"Dave"}
```

`CAR10` 的所有权从Dave变成了Dave。

> 提示:
> 在现实世界的应用程序中，链码可能会有一些访问控制逻辑。例如，只有特定的授权用户可以创建新车，并且只有车主才可以将车辆转移给其他人。

9、总结

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与网络进行交互有一个很好的理解。您已经了解了智能合约，API和SDK在查询和更新中扮演的角色的基本知识，您应该了解如何使用不同类型的应用程序来执行其他业务任务和操作。

在随后的文档中，我们将学习如何实际编写智能合约，以及如何利用这些更低级别的应用程序功能中的一些功能（特别是与身份和会员服务有关的功能）。

10、额外的资源

Hyperledger Fabric Node SDK repo是很好的资源，里面有更深入的文档和示例代码。 您还可以在Hyperledger Rocket Chat上咨询Fabric社区和组件专家。

由于水平有限，翻译质量不太好，欢迎大家拍砖。

对应官网连接地址如下：
Hyperledger Fabric - Writing Your First Application