Hyperledger Fabric 链码开发——shim包API详解

1、本篇背景

前面已经对Hyperledger Fabric的链码ChainCode开发作了详细的介绍，其中导入了链码的shim包。

shim包在链码开发中很重要，其提供了一些API，便于链码和底层区块链网络交互来访问状态变量、交易上下文、调用方证书和属性，并调用其他链码和执行其他操作。

本篇主要对链码中shim包的开发API，最后以官方提供的"fabcar"为案例讲解。

官方shim相关API文档如下：

```
https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim
```

先来回顾下链码开发介绍中简单的"资产"管理链码的源代码：

```
package main

import (
    "fmt"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAsset实现简单的链式代码来管理资产
type SimpleAsset struct {
}

// 在链码初始化过程中调用Init来初始化任何数据。
// 请注意，链码升级也会调用此函数来重置或迁移数据。
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // 从交易提案中获取参数
    args := stub.GetStringArgs()
    if len(args) != 2 {
            return shim.Error("Incorrect arguments. Expecting a key and a va
lue")
    }

    // 在这里，通过调用stub.PutState()来设置任何变量或者资产
```

```
    // 我们在账本中存储key和value
    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)
}

// 在链码每个事务中，Invoke会被调用。
// 每个事务都是由Init函数创建的资产，要么是'get'要么是'set'。
// Set方法可以通过指定新的键值对来创建新的资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response
 {
    // 从交易提案中提取函数和参数
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else { // fn为nil时，假设为get
            result, err = get(stub, args)
    }
    if err != nil {
            return shim.Error(err.Error())
    }

    // 将结果作为成功负载返回
    return shim.Success([]byte(result))
}

// 设置在账本上存储的资产（包括key和value）
// 如果key存在，它将覆盖新值
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key and a va
lue")
    }

    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
}

// 获取返回指定资产key的value值
```

```
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函数在实例化时启动容器中的链码
func main() {
    if err := shim.Start(new(SimpleAsset)); err != nil {
        fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
    }
}
```

如上所述，链码的Go代码实现需要定义一个struct（如其中的"SimpleAsset"），然后在该struct中实现Init和Invoke两个函数，最后定义一个main函数作为链码启动的入口。

之前介绍过每个链码程序都必须实现 `Chaincode接口` ，接口代码如下：

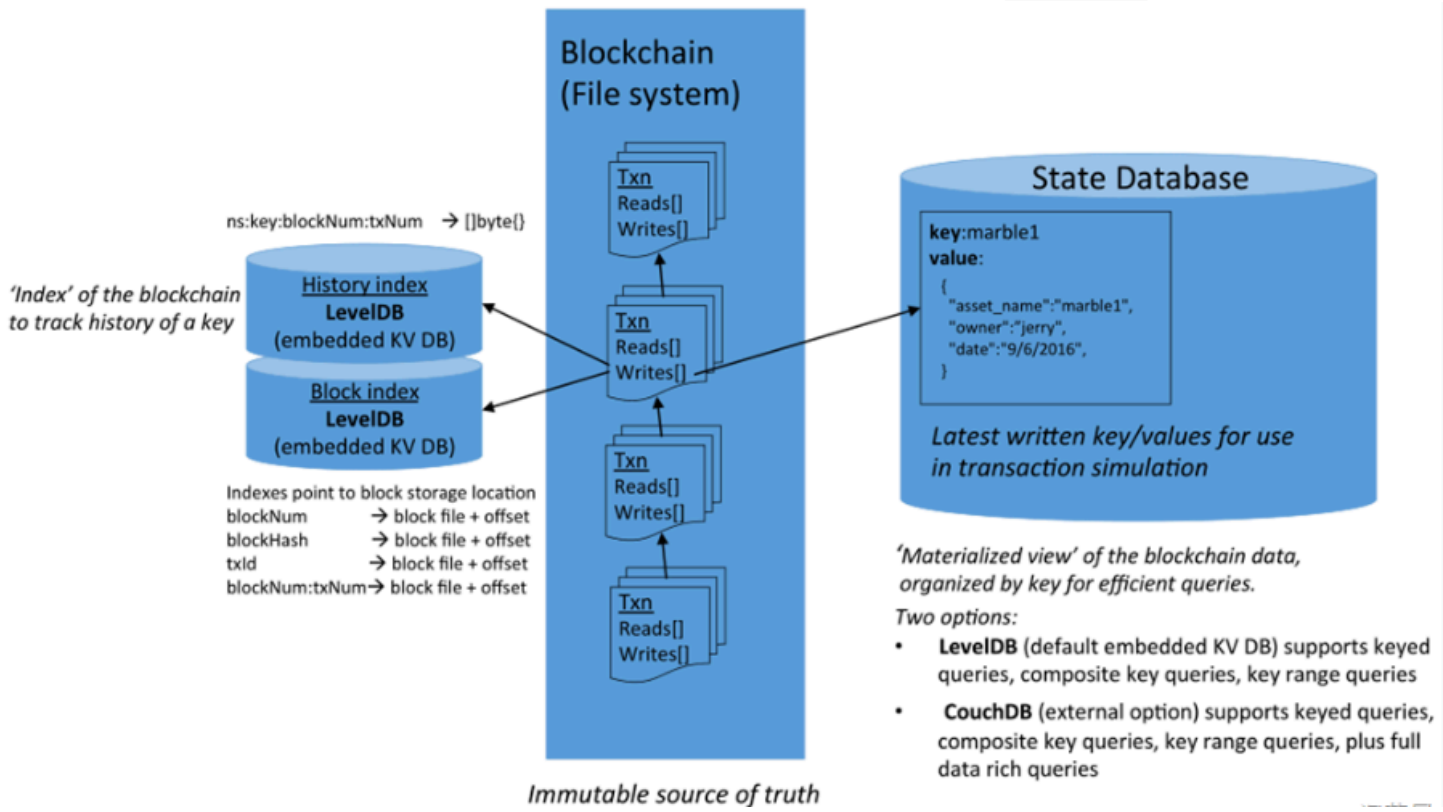
```
type Chaincode interface {
    // Init is called during Instantiate transaction after the chaincode con
tainer
    // has been established for the first time, allowing the chaincode to
    // initialize its internal data
    Init(stub ChaincodeStubInterface) pb.Response

    // Invoke is called to update or query the ledger in a proposal transact
ion.
    // Updated state variables are not committed to the ledger until the
    // transaction is committed.
    Invoke(stub ChaincodeStubInterface) pb.Response
}
```

Init和Invoke函数中都传入了一个参数，即"stub shim.ChaincodeStubInterface"，这个参数提供的接口为编写链码的业务逻辑提供了一系列实用的API，以下开始讲解这些API。

2、Hyperledger Fabric数据存储系统介绍

介绍shim包链码API之前，先了解下Hyperledger Fabric的数据存储系统。

在Hyperledger Fabric中，存在三种类型的数据存储。
1. 基于文件系统的区块链数据，跟比特币很像，比特币也是文件形式存储；
2. 存储了交易提案的读写集和，里面以键值对形式存储链码操作业务数据的`State Database`（也叫`World State`，中文名为世界状态），支持默认的`LevelDB`和用户可选的`CouchDB`）
3. 是对历史数据和区块链索引的数据库区块链式文件系统，用的是`LevelDB`。



（以上摘自"深蓝"的博客）

所以，Hyperledger Fabric常常会涉及到State DB，也就是常说的世界状态，后面也会涉及到相关内容（如账本状态State）。

3、shim包中的链码开发API

shim包中的链码开发API可分为四类：账本状态交互API、交易信息相关API、读取参数API和其他API。前面的简单"资产"管理链码也涉及到其中相关API，可以对照着来理解。

3.1 账本状态交互API

链码需要将一些数据记录在分布式账本中。需要记录的数据称为状态State，以键值对（key-value）的形式存储。其中key为字符串，value为二进制字节数组。账本状态API可以对账本状态进行操作，十分重要。方法的调用会更新交易提案的读写集和，在committer进行验证时会在此执行，跟账本状态进行比对，这类API的大致功能如下：

API	方法格式	说明
GetState	GetState(key string) ([]byte, error)	负责查询账本，返回指定键对应的值
PutState	PutState(key string, value []byte) error	尝试在账本中添加或更新一对键值。这一对键值会被添加到写集合中，等待 Committer 进一步的验证，验证通过后会真正写入到账本
DelState	DelState(key string) error	在账本中删除一对键值。同样，将对键值的删除记录到交易提案的写集合中，等待 Committer 进一步的验证，验证通过后会真正写入到账本
GetStateByRange	GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)	查询指定范围内的键值，startKey、endKey 分别指定起始（包括）和终止（不包括），当为空时默认是最大范围。返回结果是一个迭代器 StateQueryIteratorInterface 结构，可以按照字典序迭代每个键值对，最后需调用 Close() 方法关闭
GetStateByPartialCompositeKey	GetStateByPartialCompositeKey (objectType string, keys []string) (StateQueryIteratorInterface, error)	根据局部的复合键（前缀）返回所有匹配的键值。返回结果也是一个迭代器 StateQueryIteratorInterface 结构，可以按照字典序迭代每个键值对，最后需调用 Close() 方法关闭
GetHistoryForKey	GetHistoryForKey(key string) (HistoryQueryIteratorInterface, error)	返回某个键的所有历史值。需要在节点配置中打开历史数据库特性（ledger.history.enableHistoryDatabase = true）
GetQueryResult	GetQueryResult(query string) (StateQueryIteratorInterface, error)	对（支持富查询功能的）状态数据库进行富查询（rich query）。返回结果为迭代器结构 StateQueryIteratorInterface。注意该方法不会被 Committer 重新执行进行验证，因此，不应该用于更新账本状态的交易中。目前仅有 CouchDB 类型的状态数据库支持富查询

3.2 交易信息相关API

交易信息相关API可以获取到与交易信息自身相关的数据。用户对链码的调用（初始化和升级时调用Init()方法，运行时调用Invoke()方法）过程中会产生交易提案。这些API支持查询当前交易提案的一些属性，具体信息如下：

API	方法格式	说明
GetTxID	GetTxID() string。	该方法返回交易提案中指定的交易 ID。一般情况下，交易 ID 是在客户端生成提案时候产生的数字摘要，由 Nonce 随机串和签名者身份信息，一起进行 SHA256 哈希运行生成
GetTxTimestamp	GetTxTimestamp() (*timestamp.Timestamp, error)	返回交易被创建时的客户端打上的时间戳。这个时间戳是直接从交易 ChannelHeader 中提取的，所以在所有背书节点（endorsers）处看到的值都相同
GetBinding	GetBinding() ([]byte, error)	返回交易的 binding 信息。 注意：交易的 binding 信息是将交易提案的 nonse、Creator、epoch 等信息组合起来，再进行哈希得到的数字摘要
GetSignedProposal	GetSignedProposal() (*pb.SignedProposal, error)	返回该 stub 的 SignedProposal 结构，包括了跟交易提案相关的所有数据
GetCreator	GetCreator() ([]byte, error)	返回该交易的提交者的身份信息，从 signedProposal 中的 SignatureHeader.Creator 提取
GetTransient	GetTransient() (map[string][]byte, error)	返回交易中带有的一些临时信息，从 ChaincodeProposalPayload.transient 域提取，可以存放一些应用相关的保密信息，这些信息不会被写到账本中

3.3 参数读取API

调用链码时支持传入若干参数，参数可通过API读取。具体信息如下：

API	方法格式	说明
GetArgs	GetArgs() [][]byte	提取调用链码时交易 Proposal 中指定的参数，以字节串（Byte Array）数组形式返回。可以在 Init 或 Invoke 方法中使用。这些参数从 ChaincodeSpec 结构中的 Input 域直接提取
GetArgsSlice	GetArgsSlice() ([]byte, error)	提取调用链码时交易 Proposal 中指定的参数，以字节串形式返回
GetFunctionAndParameters	GetFunctionAndParameters() (string, []string)	提取调用链码时交易 Proposal 中指定的参数，其中第一个参数作为被调用的函数名称，剩下的参数作为函数的执行参数。这是链码开发者和用户约定俗成的习惯，即在 Init/Invoke 方法中编写实现若干子函数，用户调用时以第一个参数作为函数名，链码中的代码根据函数名称可以仅执行对应的分支处理逻辑
GetStringArgs	GetStringArgs() []string	提取调用链码时交易 Proposal 中指定的参数，以字符串（String）数组形式返回

3.4 其他API

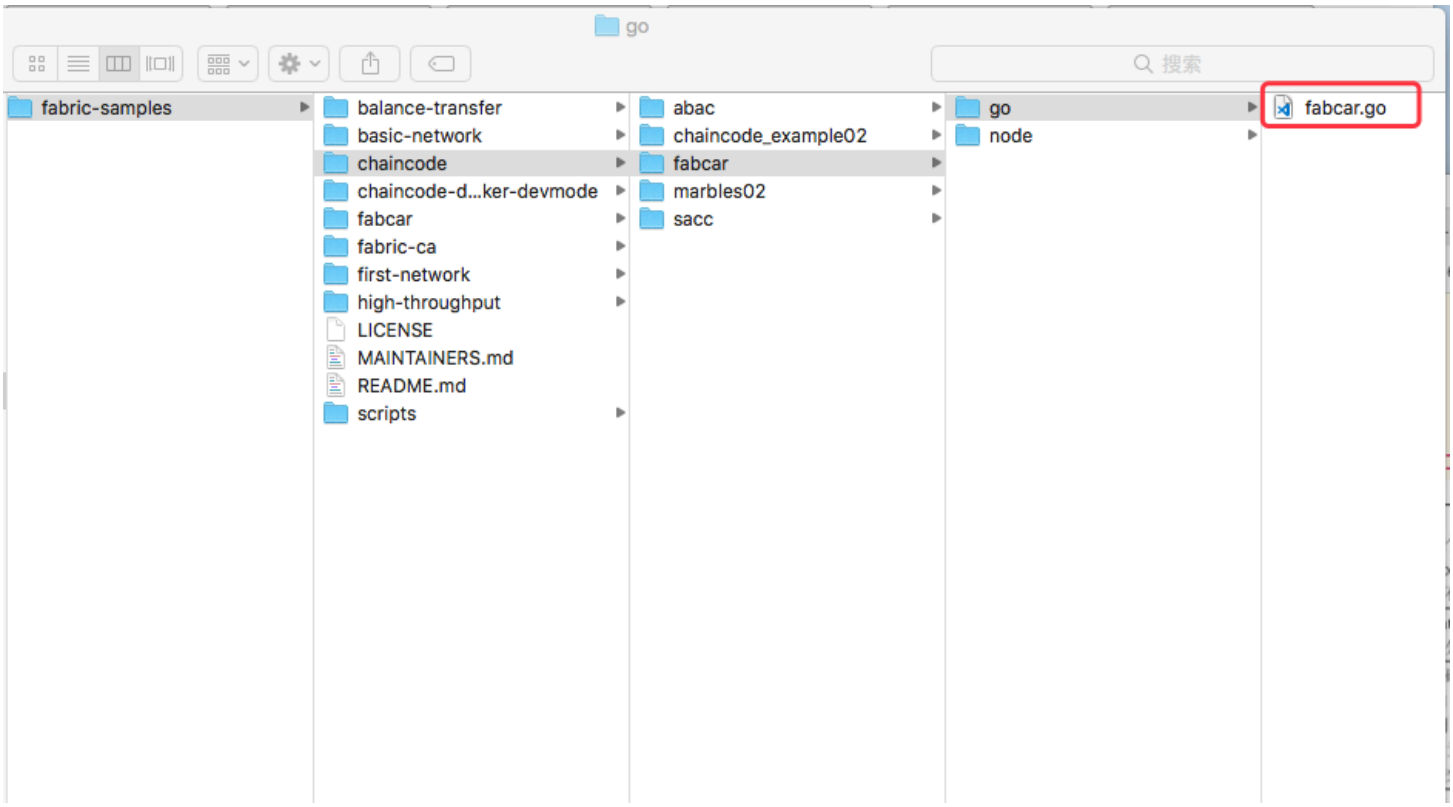
除了上面一些API以外还有一些辅助API，如下：

API	方法格式	说明
CreateCompositeKey	CreateCompositeKey(objectType string, attributes []string) (string, error)	给定一组属性（attributes），该 API 将这些属性组合起来构造返回一个复合键。返回的复合键可以被 PutState 等方法使用。objectType 和 attributes 只允许合法的 utf8 字符串，并且不能包含 U+0000 和 U+10FFFF
SplitCompositeKey	SplitCompositeKey(compositeKey string) (string, []string, error)	该方法与 CreateCompositeKey 方法相对，给定一个复合键，将其拆分为构造复合键时所用的属性
InvokeChaincode	InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response	调用另一个链码中的 Invoke 方法，如果被调用链码在同一个通道内，则添加其读写集合信息到调用交易；否则执行调用但不影响读写集合信息。如果 channel 为空，则默认为当前通道。目前仅限于读操作，同时不会生成新的交易
SetEvent	SetEvent(name string, payload []byte) error	设定当这个交易在 Committer 处被认证通过，写入到区块时发送的事件（event）

4、链码API案例详解

以下以官方提供的"fabcar"为案例讲解链码结构以及API。

4.1 链码结构解读



以下是其中部分源代码，对英文注释翻译成中文，并新增了一些注释说明：

```
// 表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。
package main

 /*
  * Go语言中导入包的语法为import("xxx")，import的作用是导入其他包
  * 4个实用程序库，用于格式化，处理字节，读取和写入JSON以及字符串操作
  * 2个特定的Hyperledger结构特定库，用于智能合同
  */
import (
    "bytes"
    "encoding/json"
    "fmt"
    "strconv"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    sc "github.com/hyperledger/fabric/protos/peer"
)

// 定义智能合约结构
type SmartContract struct {
}

// 定义汽车结构，具有4个变量属性，结构标签由编码/json库使用
type Car struct {
    Make   string `json:"make"`
    Model  string `json:"model"`
```

```
    Colour string `json:"colour"`
    Owner  string `json:"owner"`
}

 /*
  * 当智能合约“fabcar”由区块链网络实例化时，Init方法被调用
  * 最佳做法是在单独的函数中进行任何Ledger初始化 - 请参见initLedger()
  * (s *SmartContract)表示给SmartContract声明了一个方法
  * Init和Invoke函数中都传入了一个参数，即"stub shim.ChaincodeStubInterface"
  * 这个参数提供的接口为编写链码的业务逻辑提供了一系列实用的API
  * 假设一切顺利，将返回一个表示初始化已经成功的sc.Response对象。
  */
func (s *SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Respons
e {
    return shim.Success(nil)
}

/*
 * Invoke方法由于运行智能合约“fabcar”的应用程序请求而被调用
 * 调用应用程序还指定了具有参数的特定智能合约函数
 * 通过使用GetFunctionAndParameters()方法来解析调用链码时传入的参数，从而判断需要调用
的方法以及传入调用方法的参数
 */
func (s *SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Respo
nse {

    // 检索请求的智能合约函数和参数
    // 方法完整描述为：func (stub *ChaincodeStub) GetFunctionAndParameters() (f
unction string, params []string)
    // 对调用链码时传入的字符串数组形式的参数作处理，也就是拆分为两部分
    // 数组第一个参数为function，第二个到最后一个参数为args
    function, args := APIstub.GetFunctionAndParameters()
    // 找到适当的处理函数以便与账本进行交互
    if function == "queryCar" {
        return s.queryCar(APIstub, args)
    } else if function == "initLedger" {
        return s.initLedger(APIstub)
    } else if function == "createCar" {
        return s.createCar(APIstub, args)
    } else if function == "queryAllCars" {
        return s.queryAllCars(APIstub)
    } else if function == "changeCarOwner" {
        return s.changeCarOwner(APIstub, args)
    }

    return shim.Error("Invalid Smart Contract function name.")
}
```

```
// 考虑到内容有点多，后续的讲解会补充对应的内容，暂时省略了以下函数的实现
func (s *SmartContract) queryCar(APIstub shim.ChaincodeStubInterface, args []string) sc.Response { ... }

func (s *SmartContract) initLedger(APIstub shim.ChaincodeStubInterface) sc.Response { ... }

func (s *SmartContract) createCar(APIstub shim.ChaincodeStubInterface, args []string) sc.Response { ... }

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

func (s *SmartContract) changeCarOwner(APIstub shim.ChaincodeStubInterface, args []string) sc.Response { ... }

// 主要功能仅适用于单元测试模式，这里只包括完整性。
// Go语言的入口是main函数
func main() {

    // 创建一个新的智能合约
    err := shim.Start(new(SmartContract))
    if err != nil {
        fmt.Printf("Error creating new Smart Contract: %s", err)
    }
}
```

5、对State DB的增删查改

`State DB`：`State Database`，状态数据库（也叫`World State`，中文名为世界状态）。

5.1 查询汽车

通过`GetState(key string) ([]byte, error)`来查询数据。因为我们是Key-Value数据库，所以根据Key来对数据库进行查询，是一件很常见，很高效的操作，返回的数据是byte数组。

```
func (s *SmartContract) queryCar(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {

    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting 1")
    }

    // 在Go语言中，下划线"_"是占位符，表示不需要返回这个值，直接忽略
    carAsBytes, _ := APIstub.GetState(args[0])
    return shim.Success(carAsBytes)
```

```
}
```

5.2 初始化账本

通过 `PutState(key string, value []byte) error` 增改数据，对于State DB来说，增加和修改数据是统一的操作，因为State DB是一个Key-Value数据库，如果我们指定的Key在数据库中已经存在，那么就是修改操作，如果Key不存在，那么就是插入操作。对于实际的系统来说，我们的Key可能是单据编号，或者系统分配的自增ID+实体类型作为前缀，而Value则是一个对象经过JSON序列号后的字符串。

```
func (s *SmartContract) initLedger(APIstub shim.ChaincodeStubInterface) sc.R
esponse {
    cars := []Car{
        Car{Make: "Toyota", Model: "Prius", Colour: "blue", Owner: "Tomoko"}
,
        Car{Make: "Ford", Model: "Mustang", Colour: "red", Owner: "Brad"},
        Car{Make: "Hyundai", Model: "Tucson", Colour: "green", Owner: "Jin S
oo"},
        Car{Make: "Volkswagen", Model: "Passat", Colour: "yellow", Owner: "M
ax"},
        Car{Make: "Tesla", Model: "S", Colour: "black", Owner: "Adriana"},
        Car{Make: "Peugeot", Model: "205", Colour: "purple", Owner: "Michel"
},
        Car{Make: "Chery", Model: "S22L", Colour: "white", Owner: "Aarav"},
        Car{Make: "Fiat", Model: "Punto", Colour: "violet", Owner: "Pari"},
        Car{Make: "Tata", Model: "Nano", Colour: "indigo", Owner: "Valeria"}
,
        Car{Make: "Holden", Model: "Barina", Colour: "brown", Owner: "Shotar
o"},
    }

    i := 0
    for i < len(cars) {
        fmt.Println("i is ", i)
        carAsBytes, _ := json.Marshal(cars[i])
        APIstub.PutState("CAR"+strconv.Itoa(i), carAsBytes)
        fmt.Println("Added", cars[i])
        i = i + 1
    }

    return shim.Success(nil)
}
```

5.3 创建汽车

把对象转换为JSON的函数为 `json.Marshal()`，也就是说，这个函数接收任意类型的数据，并转换为字节数组类型，返回值就是我们想要的JSON数据和一个错误码。下划线"_"表示不需要返回这个值，直接忽略错误码。

```
func (s *SmartContract) createCar(APIstub shim.ChaincodeStubInterface, args
[]string) sc.Response {

    // 传入参数的个数必须为5
    if len(args) != 5 {
        return shim.Error("Incorrect number of arguments. Expecting 5")
    }

    // 第一个参数作为Key，其他参数作为构造函数的参数
    var car = Car{Make: args[1], Model: args[2], Colour: args[3], Owner: arg
s[4]}

    carAsBytes, _ := json.Marshal(car)
    APIstub.PutState(args[0], carAsBytes)

    return shim.Success(nil)
}
```

5.4 查询所有汽车

Key区间查询 `GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)` 提供了对某个区间的Key进行查询的接口，适用于任何State DB。由于返回的是一个StateQueryIteratorInterface（迭代器）接口，我们需要通过这个接口再做一个for循环，才能读取返回的信息，所有我们可以独立出一个方法，专门将该接口返回的数据以string的byte数组形式返回。

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

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

    resultsIterator, err := APIstub.GetStateByRange(startKey, endKey)
    if err != nil {
        return shim.Error(err.Error())
    }
    // defer关键字用来标记最后执行的Go语句，一般用在资源释放、关闭连接等操作，会在函数关
闭前调用。
    // 多个defer的定义与执行类似于栈的操作：先进后出，最先定义的最后执行。
    defer resultsIterator.Close()
```

```
    // buffer 是一个包含查询结果的JSON数组，bytes.buffer是一个缓冲byte类型的缓冲器存放着都是byte，这样直接定义一个 Buffer 变量，而不用初始化。
    var buffer bytes.Buffer
    buffer.WriteString("[")

    bArrayMemberAlreadyWritten := false
    // 迭代器的两个方法，hasNext：没有指针下移操作，只是判断是否存在下一个元素。next：指针下移，返回该指针所指向的元素。
    for resultsIterator.HasNext() {
        queryResponse, err := resultsIterator.Next()
        if err != nil {
            return shim.Error(err.Error())
        }
        // 在数组成员之前添加逗号，为第一个数组成员禁用它
        if bArrayMemberAlreadyWritten == true {
            buffer.WriteString(",")
        }
        buffer.WriteString("{\"Key\":")
        buffer.WriteString("\"")
        buffer.WriteString(queryResponse.Key)
        buffer.WriteString("\"")

        buffer.WriteString(", \"Record\":")
        // Record是一个JSON对象，所以我们按原样写入
        buffer.WriteString(string(queryResponse.Value))
        buffer.WriteString("}")
        bArrayMemberAlreadyWritten = true
    }
    buffer.WriteString("]")

    fmt.Printf("- queryAllCars:\n%s\n", buffer.String())

    return shim.Success(buffer.Bytes())
}
```

5.5 改变汽车拥有者

Unmarshal是用于反序列化json的函数根据data将数据反序列化到传入的对象中。

```
func (s *SmartContract) changeCarOwner(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {

    if len(args) != 2 {
        return shim.Error("Incorrect number of arguments. Expecting 2")
    }

    carAsBytes, _ := APIstub.GetState(args[0])
```

```
    car := Car{}

    json.Unmarshal(carAsBytes, &car)
    car.Owner = args[1]

    carAsBytes, _ = json.Marshal(car)
    APIstub.PutState(args[0], carAsBytes)

    return shim.Success(nil)
}
```

6、开发环境编译运行链码前提

请安装Hyperledger Fabric Samples和Docker镜像，如果已经安装好并配置好Docker环境，可以继续后面的操作。

进入到 `fabric-samples` 下的 `chaincode-docker-devmode` 目录下。

```
cd chaincode-docker-devmode
```

现在打开三个终端并进入到您的 `chaincode-docker-devmode` 目录下。

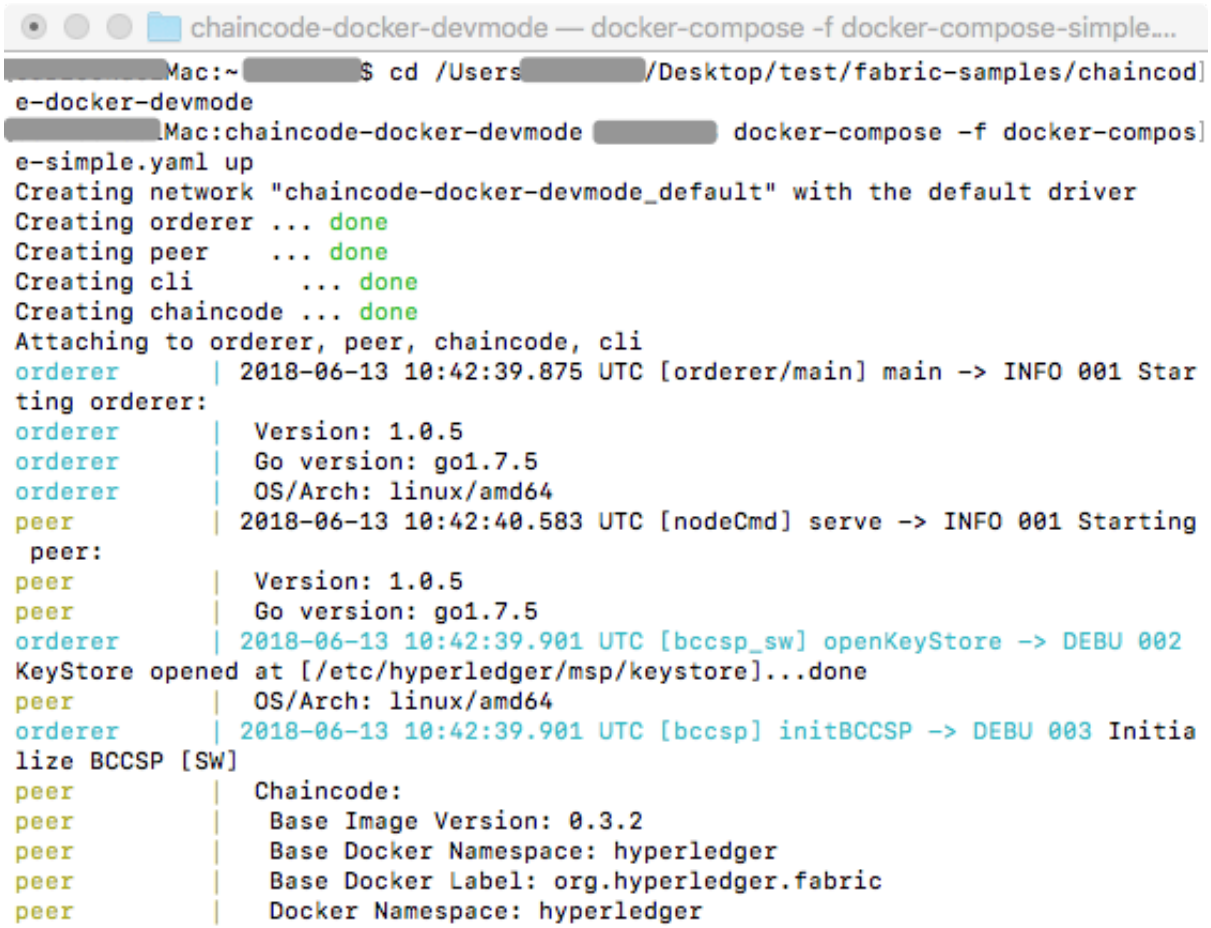
7、开发环境编译和测试链码

7.1 终端1 - 开启网络

```
docker-compose -f docker-compose-simple.yaml up
```

以上命令使用 `SingleSampleMSPSole` 配置启动 orderer并且以"开发模式"启动节点peer。它还启动了两个额外的容器，一个用于链码环境，一个用于与链码交互的CLI。创建和加入通道的命令被嵌入到CLI容器中， 因此我们可以立即跳转到链码的调用。

> 如果报错了，试着清除Docker容器，依次执行以下命令
> // 删除所有活跃的容器
> docker rm -f $(docker ps -aq)
> // 清理网络缓存
> docker network prune



7.2 终端2 - 编译和运行链码

```
docker exec -it chaincode bash
```

您应该看到类似以下内容：

```
root@d2629980e76b:/opt/gopath/src/chaincode#
```

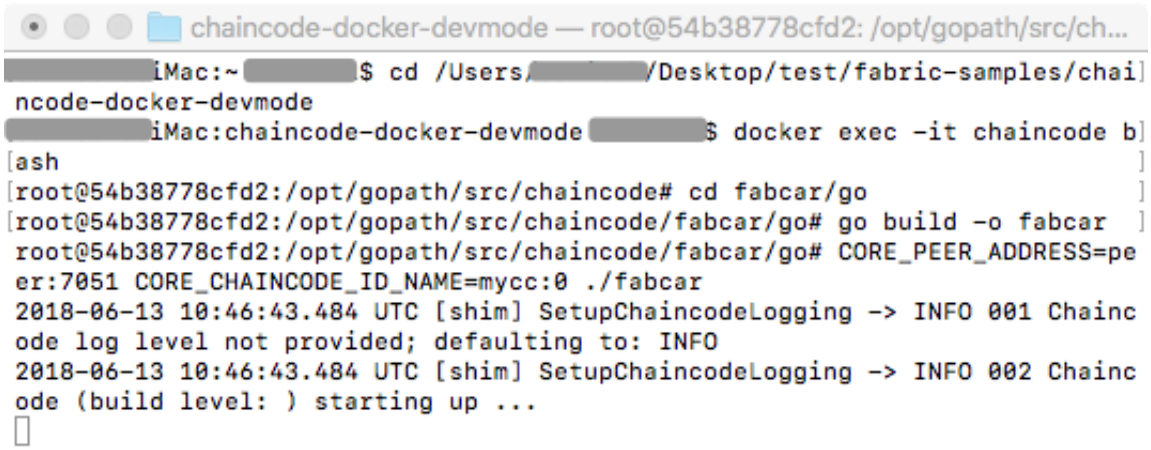
然后，编译您的链码：

```
cd fabcar/go
go build -o fabcar
```

运行您的链码：

```
CORE_PEER_ADDRESS=peer:7051 CORE_CHAINCODE_ID_NAME=mycc:0 ./fabcar
```

链码随着peer节点启动并且在peer节点成功注册后链码日志会开始显示。请注意，在此阶段，链码与任何通道都没有关联。这个会在后续步骤中使用实例化命令完成。



7.3 终端3 - 使用链码

首先，启动利用Docker CLI容器：

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

1、安装链码：

```
peer chaincode install -p chaincodedev/chaincode/fabcar/go -n mycc -v 0
#####最终输出如下(省略了很多内容)#####
2018-06-13 10:49:10.910 UTC [chaincodeCmd] install -> DEBU 00f Installed remotely response:<status:200 payload:"OK" >
```

2、实例化链码：

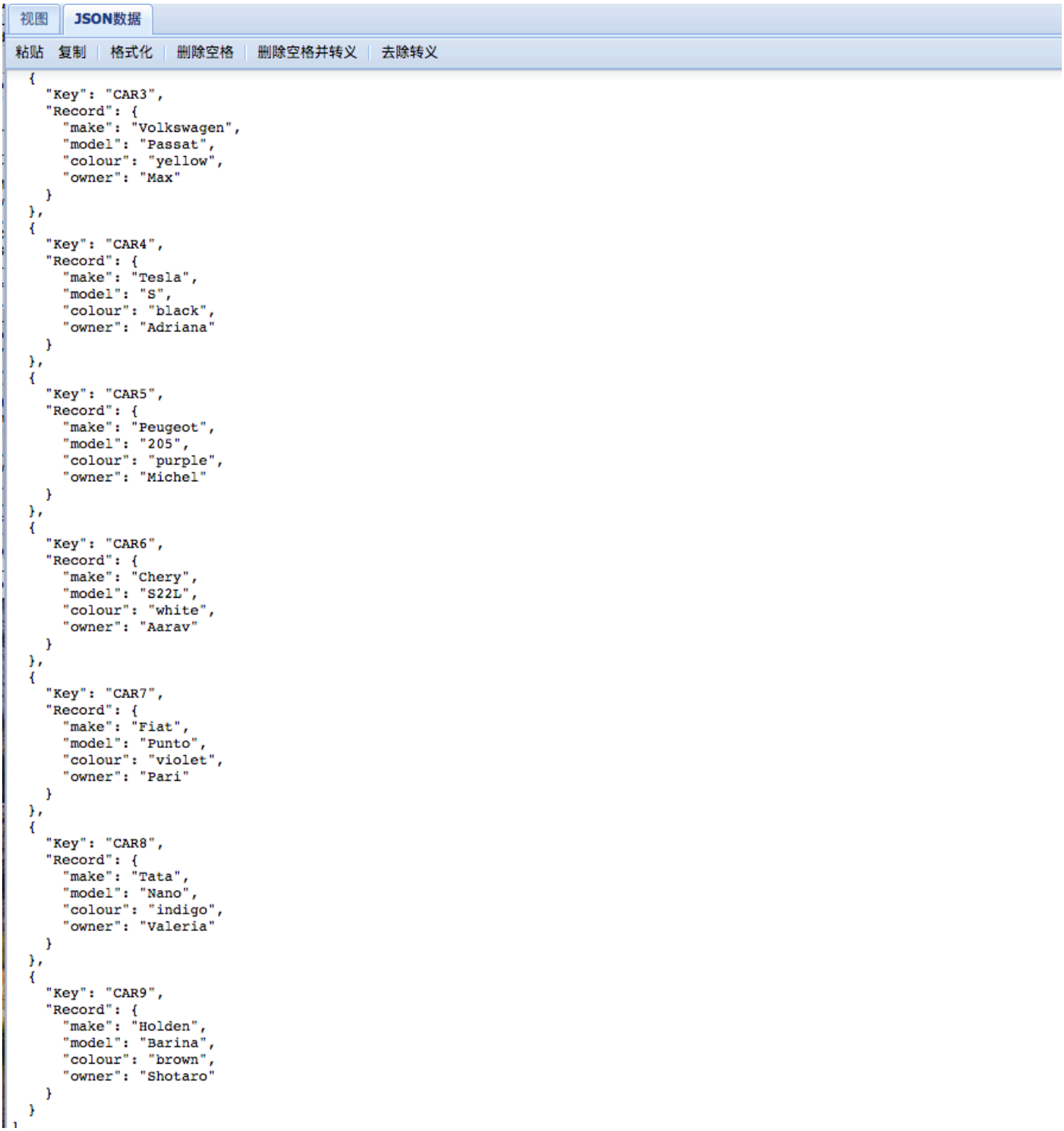
```
peer chaincode instantiate -n mycc -v 0 -c '{"Args":[]}' -C myc
#####最终输出如下(省略了很多内容)#####
2018-06-13 10:51:18.760 UTC [msp/identity] Sign -> DEBU 0a4 Sign: digest: 189AC2DE682A98749B8C76FE672DC5A325A46B6A3FB1AE2569E424A7D3566A35
2018-06-13 10:51:18.765 UTC [main] main -> INFO 0a5 Exiting.....
```

3、初始化账本：

```
peer chaincode invoke -n mycc -c '{"Args":["initLedger"]}' -C myc
#####最终输出如下(省略了很多内容)#####
2018-06-13 10:52:20.701 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0a
6 Chaincode invoke successful. result: status:200
```

4、查询所有汽车：

```
peer chaincode invoke -n mycc -c '{"Args":["queryAllCars"]}' -C myc
#####最终输出如下(省略了很多内容)#####
2018-06-13 10:55:47.505 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0a
6 Chaincode invoke successful. result: status:200 payload:"[{\"Key\":\"CAR0\
", \"Record\":{\"make\":\"Toyota\",\"model\":\"Prius\",\"colour\":\"blue\",\
"owner\":\"Tomoko\"}},{\"Key\":\"CAR1\", \"Record\":{\"make\":\"Ford\",\"mod
el\":\"Mustang\",\"colour\":\"red\",\"owner\":\"Brad\"}},{\"Key\":\"CAR2\",
\"Record\":{\"make\":\"Hyundai\",\"model\":\"Tucson\",\"colour\":\"green\",\
"owner\":\"Jin Soo\"}},{\"Key\":\"CAR3\", \"Record\":{\"make\":\"Volkswagen\
",\"model\":\"Passat\",\"colour\":\"yellow\",\"owner\":\"Max\"}},{\"Key\":\"
CAR4\", \"Record\":{\"make\":\"Tesla\",\"model\":\"S\",\"colour\":\"black\",
\"owner\":\"Adriana\"}},{\"Key\":\"CAR5\", \"Record\":{\"make\":\"Peugeot\",
\"model\":\"205\",\"colour\":\"purple\",\"owner\":\"Michel\"}},{\"Key\":\"CA
R6\", \"Record\":{\"make\":\"Chery\",\"model\":\"S22L\",\"colour\":\"white\"
,\"owner\":\"Aarav\"}},{\"Key\":\"CAR7\", \"Record\":{\"make\":\"Fiat\",\"mo
del\":\"Punto\",\"colour\":\"violet\",\"owner\":\"Pari\"}},{\"Key\":\"CAR8\"
, \"Record\":{\"make\":\"Tata\",\"model\":\"Nano\",\"colour\":\"indigo\",\"o
wner\":\"Valeria\"}},{\"Key\":\"CAR9\", \"Record\":{\"make\":\"Holden\",\"mo
del\":\"Barina\",\"colour\":\"brown\",\"owner\":\"Shotaro\"}}]"
```



5、查询某辆汽车：

```
peer chaincode invoke -n mycc -c '{"Args":["queryCar","CAR9"]}' -C myc
#####最终输出如下(省略了很多内容)#####
2018-06-13 10:59:20.621 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0a
6 Chaincode invoke successful. result: status:200 payload:"{\"make\":\"Holde
n\",\"model\":\"Barina\",\"colour\":\"brown\",\"owner\":\"Shotaro\"}"
```

6、创建一辆汽车：

```
peer chaincode invoke -n mycc -c '{"Args":["createCar","CAR10","Tesla","Mode
l S","red","Wenzil"]}' -C myc
#####最终输出如下(省略了很多内容)#####
2018-06-13 11:00:06.422 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0a
6 Chaincode invoke successful. result: status:200
```

7、查询创建的汽车：

```
peer chaincode invoke -n mycc -c '{"Args":["queryCar","CAR10"]}' -C myc
#####最终输出如下(省略了很多内容)#####
2018-06-13 11:00:56.791 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0a
6 Chaincode invoke successful. result: status:200 payload:"{\"make\":\"Tesla
\",\"model\":\"Model S\",\"colour\":\"red\",\"owner\":\"Wenzil\"}"
```

8、改变汽车拥有者：

```
peer chaincode invoke -n mycc -c '{"Args":["changeCarOwner","CAR10","Elon Mu
sk"]}' -C myc
#####最终输出如下(省略了很多内容)#####
2018-06-13 11:02:15.317 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 0a
5 ESCC invoke result: version:1 response:<status:200 message:"OK" > payload:
"......"
2018-06-13 11:02:15.318 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0a
6 Chaincode invoke successful. result: status:200
```

9、查询改变拥有者后的汽车：

```
peer chaincode invoke -n mycc -c '{"Args":["queryCar","CAR10"]}' -C myc
#####最终输出如下(省略了很多内容)#####
2018-06-13 11:03:35.509 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0a
6 Chaincode invoke successful. result: status:200 payload:"{\"make\":\"Tesla
\",\"model\":\"Model S\",\"colour\":\"red\",\"owner\":\"Elon Musk\"}"
```

从链码API的介绍到链码的结构讲解，再到在开发环境编译、安装以及调用链码等一系列流程算是走完了。

文章大部分内容来自如下博客，稍微作了下整理，添加了 `编译和运行链码` 相关内容，如有侵权联系我删除文章：

搭建基于hyperledger fabric的联盟社区（四） --chaincode开发