# 使用Golang设计数据和交易模型

在Hyperledger Fabric中，链码是由开发人员编写的智能合约的一种形式。链码实现了由区块链网络的利益相关者商定的业务逻辑。该功能暴露给客户端应用程序供其调用，只要它们具有正确的权限。

Chaincode在其自己的容器中作为独立进程运行，与Fabric网络的其他组件隔离。一个背书节点(endorsing peer)管理链码和事务调用的生命周期。通过响应客户调用，链码查询和更新账本并生成交易提议。

在本章中，我们将学习如何使用Go语言开发链码，并实施该场景下的智能合约业务逻辑。最后，我们将探讨开发全功能链码所需的关键概念和库。

在接下来的部分中，我们将探讨与概念相关的代码片段，您可以在以下地址完整实现链式代码：<https://github.com/HyperledgerHandsOn/trade-finance-logistics/tree/master/chaincode/src/github.com/trade_workflow_v1>

提示: 请注意，这也可以在我们上一章创建的本地git克隆中获得。我们有两个版本的链码，一个在trade\_workflow文件夹中，另一个在trade\_workflow\_v1文件夹中。 我们需要两个版本来演示第9章“区块链网络中的生活”中的升级。 在本章中，我们使用v1版本来演示如何在Go中编写链接代码。

在本章中，我们将涵盖以下内容：

* 创建chaincode
* 访问控制
* 实施chaincode功能
* 测试chaincode
* Chaincode设计主题
* 输出记录

## 开始链码开发

在我们开始编写链码之前，我们需要首先启动我们的开发环境。

在第3章“用业务场景设置平台”中描述了建立开发环境的步骤。但是，我们现在继续以开发模式启动Fabric网络。这种模式允许我们控制如何构建和运行链码。我们将使用这个网络在开发环境中运行我们的

下面是我们如何用开发模式开启Fabric网络：

$ cd $GOPATH/src/trade-finance-logistics/network

$ ./trade.sh up -d true

提示：如果在网络启动时遇到任何错误，可能是由一些遗留下来的Docker容器引起的。

您可以通过使用./trade.sh down -d true停止网络并运行以下命令来解决此问题：./trade.sh clean -d true。

该-d true选项告诉我们的脚本采取行动在dev网络上。

我们的开发网络现在是四个Docker容器运行。该网络由单个order，在devmode中运行的单个peer，链码容器和CLI容器组成。CLI容器在启动时创建一个名为tradechannel的区块链通道。我们将使用CLI与链码进行交互。

我们可以随意在日志目录中检查日志消息。它列出了网络启动期间执行的组件和功能。我们将保持终端打开，因为一旦chaincode被安装并调用，我们将在这里收到更多的日志消息。

## 编译和运行链码

克隆的源代码已经包括使用Go vendoring的所有依赖。考虑到这一点，我们现在可以开始构建代码并通过以下步骤运行链式代码：

1. 编译链码：在一个新的终端中，连接到链码容器并使用以下命令构建链码：

$ docker exec –it chaincode bash

$ cd trade\_workflow\_v1

$ go build

2.运行chaincode时执行以下命令：

$ CORE\_PEER\_ADDRESS=peer:7052 CORE\_CHAINCODE\_ID\_NAME=tw:0 ./trade\_workflow\_v1

我们现在有一个连接到peer的正在运行的链码。这里的日志消息表明链代码已启动并正在运行。您还可以检查网络终端中的日志消息，该消息列出与peer上到该链码的所有连接。

## 安装和实例链码

我们现在需要在启动链码之前在通道上安装它，它将调用方法Init：

1. 安装链码：在一个新的终端中，连接到CLI容器并按照以下名称tw安装链码：

$ docker exec -it cli bash

$ peer chaincode install -p chaincodedev/chaincode/trade\_workflow\_v1 -n tw –v 0

2.现在，实例以下链码：

$ peer chaincode instantiate -n tw -v 0 -c '{"Args": ["init","LumberInc","LumberBank","100000","WoodenToys","ToyBank","200000","UniversalFreight"," -C tradechannel

CLI连接的终端现在包含与链代码交互的日志消息列表。链码终端显示来自链码方法调用的消息，网络终端显示来自peer和order之间通信的消息。

## 调用链码

现在我们有一个运行的链码，我们就可以开始调用一些功能。我们的链码有几种创建和检索资产的方法。现在，我们只会调用其中的两个; 第一个创建一个新的贸易协议，第二个协议从账本中检索到它。要做到这一点，请完成以下步骤：

1. 使用以下命令将新的贸易协定放到账本中：

$ peer chaincode invoke -n tw -c '{"Args":["requestTrade", "50000", "Wood for Toys"]}' -C tradechannel

2. 使用以下命令检索账本中的该贸易协定：

$ peer chaincode invoke -n tw -c '{"Args":["getTradeStatus", "50000"]}' -C tradechannel

我们现在在devmode上有一个运行网络，我们已经成功测试了我们的链码。在下一节中，我们将学习如何从头开始创建和测试链码。

提示：dev mode

在生产环境中，chaincode的寿命是由peer管理。当我们需要在开发环境中反复修改和测试链码时，我们可以使用devmode，它允许开发人员控制链码的生命周期。此外，devmode将stdout和stderr标准文件引导到终端; 这些在生产环境中是被禁用的。

要使用devmode，peer必须连接到其他网络组件（如生产环境中），并以参数peer-chaincodedev = true开始。链码然后单独启动并配置为连接到peer节点。在开发过程中，链码可以根据需要从终端反复编译，启动，调用和停止。

我们将在下面的章节中使用DEVMODE启用网络。

## 创建链码

我们现在准备开始实施我们的链码，我们将使用Go语言进行编程。有几个IDE可用于为Go提供支持。一些更好的IDE包括Atom，Visual Studio Code等等。无论你选择任何环境都可以用我们的例子。

## 链码接口

每个链代码必须实现链码接口，它的方法被调用以响应收到的交易提议。在SHIM包中定义的链码接口如下所示：

type Chaincode interface {

Init(stub ChaincodeStubInterface) pb.Response

Invoke(stub ChaincodeStubInterface) pb.Response

}

正如你所看到的，链码类型定义了两个函数：init和invoke。

这两个函数都有一个类型为Chaincode Stub Interface的参数stub。

stub参数是我们在实现链码功能时使用的主要对象，因为它提供了访问和修改账本，获取调用参数等功能。

另外，SHIM包提供了其他类型和功能以构建链码; 你可以在https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim上查看整个软件包。

## 建立链码文件

现在，让我们建立链码文件。

我们将使用从GitHub克隆的文件夹。 链码文件位于以下文件夹中：

$GOPATH/src/trade-finance-logistics/chaincode/src/github.com/trade\_workflow\_v1

您可以按照以下步骤检查文件夹中的代码文件，也可以创建一个新文件夹并按照说明创建代码文件。

1.首先，我们需要创建链码文件

在您最喜欢的编辑器中，创建一个文件tradeWorkflow.go，并包含以下包和导入语句：

package main

import (

"fmt"

"errors"

"strconv"

"strings"

"encoding/json"

"github.com/hyperledger/fabric/core/chaincode/shim"

"github.com/hyperledger/fabric/core/chaincode/lib/cid"

pb "github.com/hyperledger/fabric/protos/peer"

)

在上述代码片段中，我们可以看到第4到第8行导入了Go语言系统包，第9到11行导入了shim，cid和pb Fabric包。pb包提供了对peer节点 protobuf类型的定义，cid提供访问控制功能。我们将在访问控制部分详细了解CID。

1. 现在，我们需要定义链码类型。让我们添加TradeWorkflowChaincode类型来实现chaincode函数，如下面的片段所示：

type TradeWorkflowChaincode struct {

testMode bool

}

记下第2行中的testMode字段bool。我们将使用此字段来规避测试期间的访问控制检查。

1. TradeWorkflowChaincode类型是实现shim.Chaincode接口所必需的。必须实现接口方法才能使TradeWorkflowChaincode成为Shim包的有效Chaincode类型。
2. 链码被安装到区块链网络后，将调用Init方法。每个背书节点只执行一次，部署自己的链式代码实例。该方法可用于初始化，引导和设置链码。下面的代码片段显示了Init方法的默认实现。请注意，第3行中的方法将一行写入标准输出以报告其调用。在第4行中，该方法返回调用函数shim的结果。运行成功是使用nil的参数值表示成功执行且结果为空，如下所示：

// TradeWorkflowChaincode implementation

func (t \*TradeWorkflowChaincode) Init(stub SHIM.ChaincodeStubInterface)

pb.Response {

fmt.Println("Initializing Trade Workflow")

return shim.Success(nil)

}

链码方法的调用必须返回pb.Response对象的一个实例。下面的代码片段列出了SHIM包中的两个帮助函数来创建响应对象。接下来的函数将响应对象序列化为gRPC protobuf消息：

// Creates a Response object with the Success status and with argument of a 'payload' to return

// if there is no value to return, the argument 'payload' should be set to 'nil'

func shim.Success(payload []byte)

// creates a Response object with the Error status and with an argument of a message of the error

func shim.Error(msg string)

5. 现在是时候继续来看调用的参数。在这里，该方法将使用stub.GetFunctionAndParameters函数检索调用的参数，并验证是否提供了所需的参数个数。Init方法期望不接收参数，因此将账本保持原样。当Init函数被调用时会发生这种情况，因为Chaincode在账本上升级到更新的版本。当安装了第一次chaincode，预计接收八个参数，其中包括参与者的详细信息，这些参数将被记录为初始状态。如果提供的参数个数不正确，该方法将返回一个错误。 验证参数的代码块如下所示：

\_, args := stub.GetFunctionAndParameters()

var err error

// Upgrade Mode 1: leave ledger state as it was

if len(args) == 0 {

return shim.Success(nil)

}

// Upgrade mode 2: change all the names and account balances

if len(args) != 8 {

err = errors.New(fmt.Sprintf("Incorrect number of arguments.Expecting 8: {" + "Exporter, " +

"Exporter's Bank, " +

"Exporter's Account Balance, " +

"Importer, " +

"Importer's Bank, " +

"Importer's Account Balance, " +

"Carrier, " +

"Regulatory Authority" + "}. Found %d", len(args)))

return shim.Error(err.Error())

}

正如我们在前面的代码片段中看到的那样，当提供了包含参与者的名称和角色的期望数量的参数时，该方法验证并将参数转换为正确的数据类型，并将它们作为初始状态记录在账本上。

在下面的代码片段中，第2行和第7行中，该方法将参数转换为整数。如果转换失败，则返回一个错误。在第14行中，字符串数组由字符串常量构造而成。在这里，我们引用文件constants.go中定义的词法常量，它位于chaincode文件夹中。常量表示初始值将被记录到账本中的键。最后，在第16行中为每个常量写一个记录（资产）到账本上。函数stub.PutState将一个键和值对记录在账本上。

注意，在该账本中数据存储为byte数组; 我们想要在帐本上存储的任何数据都必须先转换为byte数组，如下面的代码片段所示：

// Type checks

\_, err = strconv.Atoi(string(args[2]))

if err != nil {

fmt.Printf("Exporter's account balance must be an integer. Found%s\n", args[2])

return shim.Error(err.Error())

}

\_, err = strconv.Atoi(string(args[5]))

if err != nil {

fmt.Printf("Importer's account balance must be an integer. Found %s\n", args[5])

return shim.Error(err.Error())

}

// Map participant identities to their roles on the ledger

roleKeys := []string{ expKey, ebKey, expBalKey, impKey, ibKey, impBalKey, carKey, raKey }

for i, roleKey := range roleKeys {

err = stub.PutState(roleKey, []byte(args[i]))

if err != nil {

fmt.Errorf("Error recording key %s: %s\n", roleKey, err.Error())

return shim.Error(err.Error())

}

}

(到该章第13页)