链码开发者教程

链码是什么?

链码是一段程序，由 Go 、 node.js 、或者`Java <https://java.com/en/>`_ 编写，来实现一些预定义的接口。链码运行在一个和背书节点分开的独立进程中，通过应用程序提交的交易来初始化和管理账本状态。

链码一般处理网络中的成员一致同意的商业逻辑，所以它类似于“智能合约”。链码可以在提案交易中被调用用来升级或者查询账本。赋予适当的权限，链码就可以调用其他链码访问它的状态，不管是在同一个通道还是不同的通道。注意，如果被调用链码和调用链码在不同的通道，就只能执行只读查询。就是说，调用不同通道的链码只能进行“查询”，在提交的子语句中不能参与状态的合法性检查。

在下边的章节中，我们站在应用开发者的角度来介绍链码。我们将演示一个简单的链码示例应用，并浏览 Chaincode Shim API 中每一个方法的作用。如果你是网络管理员，负责将链码部署在运行中的网络上，请查看 Deploying a smart contract to a channel 教程和 Fabric 链码生命周期 概念主题。

本教程以底层视角提供 Fabric Chaincode Shim API 的概览。你也可以使用 Fabric Contract API 提供的高级 API。关于使用 Fabric Contract API 开发智能合约的更多信息，请访问 智能合约处理 主题。

Chaincode API

每一个链码程序都必须实现 `Chaincode` 接口，该接口的方法在接收 到交易时会被调用。你可以在下边找到不同语言 Chaincode Shim API 的参考文档：

- Go
- Node.js
- Java

在每种语言中，客户端提交交易提案都会调用 `Invoke` 方法。该方法可以让你使用链码来读写通道账本上的数据。

你还要包含 `Init` 方法，该方法是实例化方法。该方法是链码接口需要的，你的应用程序没有必要调用。你可以使用 Fabric 链码生命周期过程来指定在 Invoke 之前是否必须调用 `Init` 方法。更多信息，请参考 Fabric 链码生命周期文档中 批准链码定义 步骤的实例化参数。

链码 “shim” API 中的其他接口是 `ChaincodeStubInterface`：

- Go
- Node.js
- Java

用来访问和修改账本，以及在链码间发起调用。

在本教程中使用 Go 链码，我们将通过实现一个管理简单的“资产”示例链码应用来演示如何使用这些 API 。

简单资产链码

我们的应用程序是一个基本的示例链码，用来在账本上创建资产（键-值对）。

选择一个位置存放代码

如果你没有写过 Go 的程序，你可能需要确认一下你是否安装了 Go 以及你的系统是否配置正确。我们假设你用的是支持模块的版本。

现在你需要为你的链码应用程序创建一个目录。

简单起见，我们使用如下命令：

```
mkdir sacc && cd sacc
```

现在，我们创建一个用于编写代码的源文件：

```
go mod init sacc
touch sacc.go
```

内务

首先，我们从内务开始。每一个链码都要实现 Chaincode 接口 中的 `Init` 和 `Invoke` 方法。所以，我们先使用 Go import 语句来导入链码必要的依赖。我们将导入链码 shim 包和 peer protobuf 包 。然后，我们加入一个 `SimpleAsset` 结构体来作为 Chaincode shim 方法的接收者。

```
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 {
}
```

初始化链码

然后，我们将实现 `Init` 方法。

```
// Init is called during chaincode instantiation to initialize any data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {

}
```

注解

注意，链码升级的时候也要调用这个方法。当写一个用来升级已存在的链码的时候，请确保合理更改 `Init` 方法。特别地，当升级时没有“迁移”或者没东西需要初始化时，可以提供一个空的 `Init` 方法。

接下来，我们将使用 ChaincodeStubInterface.GetStringArgs 方法获取 `Init` 调用的参数，并且检查其合法性。在我们的用例中，我们希望得到一个键-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，我们已经确定了调用是合法的，我们将把初始状态存入账本中。我们将调用 ChaincodeStubInterface.PutState 并将键和值作为参数传递给它。假设一切正常，将返回一个 peer.Response 对象，表明初始化成功。

```
// 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)
}
```

调用链码

首先，我们增加一个 `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 {

}
```

就像上边的 `Init` 函数一样，我们需要从 `ChaincodeStubInterface` 中解析参数。`Invoke` 函数的参数是将要调用的链码应用程序的函数名。在我们的用例中，我们的应用程序将有两个方法：`set` 和 `get`，用来设置或者获取资产当前的状态。我们先调用 ChaincodeStubInterface.GetFunctionAndParameters 来为链码应用程序的方法解析方法名和参数。

```
// 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()

}
```

然后，我们将验证函数名是否为 `set` 或者 `get`，并执行链码应用程序的方法，通过 `shim.Success` 或 `shim.Error` 返回一个适当的响应，这个响应将被序列化为 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))
}
```

实现链码应用程序

就像我们说的，我们的链码应用程序实现了两个功能，它们可以通过 `Invoke` 方法调用。我们现在来实现这些方法。注意我们之前提到的，要访问账本状态，我们需要使用链码 shim API 中的 ChaincodeStubInterface.PutState 和 ChaincodeStubInterface.GetState 方法。

```
// 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
}
```

把它们组合在一起

最后，我们增加一个 `main` 方法，它将调用 shim.Start 方法。下边是我们链码程序的完整源码。

```
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)
    }
}
```

链码访问控制

链码可以通过调用 GetCreator() 方法来使用客户端（提交者）证书进行访问控制决策。另外，Go shim 提供了扩展 API，用于从提交者的证书中提取客户端标识用于访问控制决策，该证书可以是客户端身份本身，或者组织身份，或客户端身份属性。

例如，一个以键-值对表示的资产可以将客户端的身份作为值的一部分保存其中（比如以 JSON 属性标识资产主人），以后就只有被授权的客户端才可以更新键-值对。

详细信息请查阅 client identity (CID) library documentation

To add the client identity shim extension to your chaincode as a dependency, see 管理 Go 链码的扩展依赖.

将客户端身份 shim 扩展作为依赖添加到你的链码，请查阅 管理 Go 链码的扩展依赖 。

管理 Go 链码的扩展依赖

你的 Go 链码需要 Go 标准库之外的一些依赖包（比如链码 shim）。当链码安装到 peer 的时候，这些报的源码必须被包含在你的链码包中。如果你将你的链码构造为一个模块，最简单的方法就是在打包你的链码之前使用 `go mod vendor` 来 “vendor” 依赖。

```
go mod tidy
go mod vendor
```

这就把你链码的扩展依赖放进了本地的 `vendor` 目录。

当依赖都引入到你的链码目录后，`peer chaincode package` 和 `peer chaincode install` 操作将把这些依赖一起放入链码包中。