转账

安装Go及Docker, Docker-compose, 并配置Go相应的环境变量

转账链码

能够实现对账户的查询, 转账, 删除账户的功能

整合存及取的功能

创建目录

为chaincode应用程序创建一个目录作为其子目录

```
$ mkdir -p $GOPATH/src/payment
$ cd $GOPATH/src/payment
```

新建文件

新建一个文件,用于编写Go代码

```
$ touch payment.go
$ vim payment.go
```

编写代码

必须实现 Chaincode接口的 `Init` 和 `Invoke` 函数。因此，须在文件中添加go import语句以获取链代码的依赖。

导入chaincode shim包和 peer protobuf包。然后添加一个结构 `SimpleChaincode` 作为Chaincode函数的接收器

```
package main

import (
    "fmt"

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

type SimpleChaincode struct {
}
```

初始化Chaincode

实现Init功能

```
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface)
peer.Response {

}
```

注意，chaincode升级也会调用此函数。

在编写升级现有升级链代码时，请确保正确修改该 `Init` 功能。特别是，如果没有任何可实现功能，须提供一个空的“Init”方法

将 `Init` 使用GetStringArgs 函数检索调用 的参数并检查合法性

获取传入的函数名称及参数, 函数名称使用下划线忽略

定义账户名称及对应值的变量

```
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface)
peer.Response {
    _, args := stub.GetFunctionAndParameters()
    var A, B string
    var Aval, Bval int
    var err error

    if len(args) != 4 {
        return shim.Error("Incorrect number of arguments. Expecting
4")
    }

}
```

调用 PutState ，并将键和值作为参数传入。如果顺利，返回一个指示初始化成功的peer.Response对象

- 判断参数个数是否为4
- 获取args[0]的值赋给A
- strconv.Atoi(args[1]) 转换为整数, 返回aval, err
- 判断err
- 获取args[2]的值赋给B
- strconv.Atoi(args[3]) 转换为整数, 返回bval, err
- 判断err
- 将A的状态值记录到分布式账本中
- 判断err
- 将B的状态值记录到分布式账本中
- 判断err
- return shim.Success(nil)

```
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface)
peer.Response {
    fmt.Println("Init......")
    // 通过stub的GetFunctionAndParameters方法提取本次调用的交易中所指定
的参数
    _, args := stub.GetFunctionAndParameters()
    var a, b string
    var aval, bval int
    var err error

}
```

调用Chaincode

添加`Invoke`函数, 参数是将要调用的chaincode应用程序函数的名称

应用程序将具有三个不同的分支功能：`invoke` 、 `delete` 、 `query`分别实现转账、删除、查询的功能, 根据交易参数定位到不同的分支处理逻辑

- 获取函数名称与参数
- 判断函数名称并调用相应的方法
- return shim.Error("Invalid invoke function name. Expecting \"invoke\" \"delete\" \"query\"")

```
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface)
pb.Response {
    fmt.Println("Invoke......")

}
```

实现Chaincode应用

query方法

根据给定的账户名称查询对应的状态信息

- 判断参数是否为1个
- 根据传入的参数调用GetState查询状态, aval, err为接收返回值

- 如果返回err不为空, 则返回错误
- 如果返回的状态为空, 则返回错误
- 如果无错误, 返回查询到的值

```
func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface,
args []string) pb.Response {
    var A string // Entities
    var err error


    return shim.Success(val)
}
```

invoke方法

根据指定的两个账户名称及数量, 实现转账

- 判断参数是否为3
- 获取两个账户名称(args[0]与args[1])值, 赋给两个变量
- 调用GetState获取a账户状态, avalsByte, err为返回值
  - 判断有无错误(err不为空, avalsByte为空)
- 类型转换: aval, _ = strconv.Atoi(string(avalsByte))
- 调用GetState获取b账户状态, bvalsByte, err为返回值
  - 判断有无错误(err不为空, bvalsByte为空)
- 类型转换: bval, _ = strconv.Atoi(string(bvalsByte))
- 将要转账的数额进行类型转换: x, err = strconv.Atoi(args[2])
- 判断 err 是否为空
- aval, bval 执行转账操作
- 记录状态, err = PutState(a, []byte(strconv.Itoa(aval)))
  - Itoa: 将整数转换为十进制字符串形式
  - 判断有无错误.

- 记录状态, err = PutState(b, []byte(strconv.Itoa(bval)))
    - 判断有无错误.
- return shim.Success(nil)

```
// 从A到B支付X单位
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface,
args []string) pb.Response {
    var a, b string    // Entities
    var aval, bval int // Asset holdings
    var x int          // Transaction value
    var err error

}
```

delete方法

根据指定的名称删除对应的实体信息

- 判断参数个数是否为1
- 调用DelState方法, err接收返回值
- 如果err不为空, 返回错误
- 返回成功shim.Success(nil)

```
func (t *SimpleChaincode) delete(stub shim.ChaincodeStubInterface,
args []string) pb.Response {

}
```

main方法

```
func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}
```

构建Chaincode

编译chaincode

```
$ go get -u --tags nopkcs11
github.com/hyperledger/fabric/core/chaincode/shim
$ go build --tags nopkcs11
```

使用开发模式测试

正常情况下chaincode由对等体启动和维护。然而，在“开发模式”下，链码由用户构建并启动

如果没有安装Hyperledger Fabric Samples请先安装

如果没有下载Docker images请先下载

跳转至`fabric-samples`的`chaincode-docker-devmode`目录

```
$ cd ~/hyfa/fabric-samples/chaincode-docker-devmode/
```

使用`docker images`查看Docker镜像信息(显示本地Docker Registry)

```
$ sudo docker images
```

会看到如下输出

```
REPOSITORY                     TAG                 IMAGE ID
   CREATED             SIZE
hyperledger/fabric-tools       latest              b7bfddf508bc
About an hour ago   1.46GB
hyperledger/fabric-tools       x86_64-1.1.0        b7bfddf508bc
About an hour ago   1.46GB
hyperledger/fabric-orderer     latest              ce0c810df36a
About an hour ago   180MB
hyperledger/fabric-orderer     x86_64-1.1.0        ce0c810df36a
About an hour ago   180MB
hyperledger/fabric-peer        latest              b023f9be0771
About an hour ago   187MB
hyperledger/fabric-peer        x86_64-1.1.0        b023f9be0771
About an hour ago   187MB
hyperledger/fabric-javaenv     latest              82098abb1a17
About an hour ago   1.52GB
hyperledger/fabric-javaenv     x86_64-1.1.0        82098abb1a17
About an hour ago   1.52GB
hyperledger/fabric-ccenv       latest              c8b4909d8d46
About an hour ago   1.39GB
hyperledger/fabric-ccenv       x86_64-1.1.0        c8b4909d8d46
About an hour ago   1.39GB
......
```

使用三个终端

终端1 启动网络

启动网络

```
$ sudo docker-compose -f docker-compose-simple.yaml up -d
```

上面的命令以`SingleSampleMSPSolo`orderer配置文件启动网络，并以“dev模式”启动对等体。它还启动了两个额外的容器：一个用于chaincode环境，一个用于与chaincode交互的CLI。创建和加入通道的命令被嵌入到CLI容器中，因此可以立即跳转到链式代码调用

如果执行命令时出现如下错误;

终端2 建立并启动链码

打开第二个终端, 进入到`chaincode-docker-devmode` 目录

```
$ cd ~/hyfa/fabric-samples/chaincode-docker-devmode/
```

进入

```
$ sudo docker exec -it chaincode bash
```

命令提示符变为:

`root@858726aed16e:/opt/gopath/src/chaincode#`

编译

进入payment目录编译chaincode

```
root@858726aed16e:/opt/gopath/src/chaincode#  cd payment
root@858726aed16e:/opt/gopath/src/chaincode/payment#  go build
```

运行chaincode

```
CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=paycc:0 ./payment
```

终端3 使用链码

```
$ sudo docker exec -it cli bash
```

安装及实例化

进入CLI容器后执行如下命令安装及实例化chaincode

```
peer chaincode install -p chaincodedev/chaincode/payment -n paycc -v
0
peer chaincode instantiate -n paycc -v 0 -c '{"Args":["init","a",
"100", "b","200"]}' -C myc
```

调用

进行调用, `a` 向 `b` 转账 `20`

```
peer chaincode invoke -n paycc -c '{"Args":["invoke",
"a","b","20"]}' -C myc
```

执行成功, 输出如下内容

```
......
...... Chaincode invoke successful. result: status:200 payload:"20"
......
```

查询

查询 `a` 的值

```
peer chaincode query -n paycc -c '{"Args":["query","a"]}' -C myc
```

执行成功, 输出: `Query Result: 80`