

# 简单的一个fabric案例模仿实现和用处注明和环境搭建的证明

---

主要实现如下的功能：

- 初始化 A、B 两个账户，并为两个账户赋初始资产值；
- 在 A、B 两个账户之间进行资产交易；
- 分别查询 A、B 两个账户上的余额，确认交易成功；
- 删除账户。
- 新增账户

主要函数

- Init: 初始化 A、B 两个账户；
- Invoke: 调用其它函数的入口；
- transfer: 实现 A、B 账户间的转账；
- query: 查询 A、B 账户上的余额；
- delete: 删除账户。
- create: 新增账户

## 案例一

---

ps:主要感觉这个好像没什么典型的函数，所以后面还有一个

(1.创建链码目录，注：如果上述:mkdir 不能执行，可能是没有权限，加上sudo就可以了)

```
mkdir atcc && cd atcc
```

(2.创建模块和源文件)

```
go mod init atcc    (生成go.mod 文件)  
touch atcc.go
```

(3.housekeeping

也就是给链码输入进一些必要的依赖

我们输入fabric合约API的包并且导入自定义SmartContract

导入链码依赖包：)

```
package main    //必须在main包下
```

```
import (  
    "encoding/json"  
    "fmt"  
    "log"  
    "github.com/hyperledger/fabric-contract-api-go/contractapi"  
)
```

```
// SmartContract provides functions for managing an Asset
```

```
//定义一个结构体
```

```

type SmartContract struct {
    contractapi.Contract
}

```

(4.添加一个结构 Asset 来表示在账本上的简单的资产)

*// 获取用户传递给调用链码的所需参数*

*// Asset describes basic details of what makes up a simple asset*

```

type Asset struct {
    ID      string json:"ID"    //资产人的ID, 姓名, 余额
    Owner   string json:"owner"
    Value   int    json:"Value"
}

```

*//为结构体绑定init ()*

(5.初始化账本, 建立链码用 InitLedger 函数来让账本有些基础数据)

*// 初始化数据状态, 实例化/升级链码时被自动调用*

```

*func (s SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {
    assets := []Asset{
        {ID: "asset1", Owner: "ZhangSan", Value: 300},
        {ID: "asset2", Owner: "LiSi", Value: 400},
        {ID: "asset3", Owner: "Klay", Value: 500},
    }
}

```

*//每个人的账户信息和余额*

```

for _, asset := range assets {
    assetJSON, err := json.Marshal(asset)
    *if err != nil * //检查合法性

{
    return err
}

err = ctx.GetStub().PutState(asset.ID, assetJSON)
if err != nil {
    return fmt.Errorf("failed to put to world state. %v", err)
}
}

return nil
}

```

(6.编写新增资产函数, 在账本上面创建一个还不存在的资产)

*//创建资产向世界状态发布具有给定详细信息的新资产*

```

*func (s SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface, id string,
owner string, Value int) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if exists {

```

```

        return fmt.Errorf("the asset %s already exists", id)
    }

    asset := Asset{
        ID:      id,
        Owner:    owner,
        Value:    Value,
    } //添加具体的资产信息
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(id, assetJSON)
}

```

(7.编写读取资产函数)

//读取资产返回存储在具有给定 id 的世界状态中的资产

```

func (s SmartContract) ReadAsset(ctx contractapi.TransactionContextInterface, id string)
(Asset, error) {
    assetJSON, err := ctx.GetStub().GetState(id)
    if err != nil {
        return nil, fmt.Errorf("failed to read from world state: %v", err)
    }
    if assetJSON == nil {
        return nil, fmt.Errorf("the asset %s does not exist", id)
    }

    var asset Asset
    err = json.Unmarshal(assetJSON, &asset)
    if err != nil {
        return nil, err
    }

    return &asset, nil
}

```

(8.编写更新资产函数)

//更新资产使用提供的参数更新处于世界状态的现有资产

```

*func (s SmartContract) UpdateAsset(ctx contractapi.TransactionContextInterface, id string,
owner string, Value int) error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("the asset %s does not exist", id)
    }
}

```

//用新资产覆盖原来的资产

```

asset := Asset{
    ID:      id,
    Owner:    owner,

```

```

    Value:    Value,
}
assetJSON, err := json.Marshal(asset)
if err != nil {
    return err
}

return ctx.GetStub().PutState(id, assetJSON)
}

```

(9.编写删除函数)

//删除资产从世界状态中删除给定资产

```

*func (s SmartContract) DeleteAsset(ctx contractapi.TransactionContextInterface, id string)
error {
    exists, err := s.AssetExists(ctx, id)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("the asset %s does not exist", id)
    }

    return ctx.GetStub().DelState(id)
}

```

(10.编写判断资产存在函数)

// 当具有给定 ID 的资产存在于世界状态中时，AssetExists 返回 true，不然就会返回不存在

```

*func (s SmartContract) AssetExists(ctx contractapi.TransactionContextInterface, id string)
(bool, error) {
    assetJSON, err := ctx.GetStub().GetState(id)
    if err != nil {
        return false, fmt.Errorf("failed to read from world state: %v", err)
    }

    return assetJSON != nil, nil
}

```

(11.编写资产转移函数)

// 转移资产使用世界状态中的给定 ID 更新资产的所有者字段。

```

*func (s SmartContract) TransferAsset(ctx contractapi.TransactionContextInterface, id string,
newOwner string) error {
    asset, err := s.ReadAsset(ctx, id)
    if err != nil {
        return err
    }

    asset.Owner = newOwner
    assetJSON, err := json.Marshal(asset)//进行资产转移
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(id, assetJSON)
}

```

```
}
```

(12.编写查询函数)

//获取所有资产返回在世界状态中找到的所有资产

```
func (s SmartContract) GetAllAssets(ctx contractapi.TransactionContextInterface) ([]Asset, error) {
```

//范围查询，开始键和结束键为空字符串，对链码命名空间中的所有资产进行开放式查询。

```
    resultsIterator, err := ctx.GetStub().GetStateByRange("", "")
```

```
    if err != nil {
```

```
        return nil, err
```

```
    }
```

```
    defer resultsIterator.Close()
```

```
var assets []Asset*
```

```
for resultsIterator.HasNext() {
```

```
    queryResponse, err := resultsIterator.Next() //从账本中获取该账户的余额
```

```
    if err != nil {
```

```
        return nil, err
```

```
    }
```

```
    var asset Asset
```

```
    err = json.Unmarshal(queryResponse.Value, &asset)
```

```
    if err != nil {
```

```
        return nil, err
```

```
    }
```

```
    assets = append(assets, &asset)
```

```
}
```

```
return assets, nil
```

```
}
```

(13.编写主函数)

```
func main() {
```

```
    assetChaincode, err := contractapi.NewChaincode(&SmartContract{})
```

```
    if err != nil {
```

```
        log.Panicf("Error creating asset-transfer-basic chaincode: %v", err)
```

```
    }
```

```
    if err := assetChaincode.Start(); err != nil {
```

```
        log.Panicf("Error starting asset-transfer-basic chaincode: %v", err)
```

```
    }
```

```
}
```

整合可以得到一个简单的链码，可以实现对自定义资产结构增删改查的功能。

### 三、管理链码依赖

在将依赖部署到网络上前，需要将链码的相关依赖包含在软件包中，最简单的方法就是利用go mod 将相关依赖下载到本地。

```
go mod tidy      #增加缺失的包,移除没用的包
```

```
go mod vendor    #将依赖包复制到项目下的 vendor 目录
```

这样就把外部依赖放入一个本地的vendor目录

这些其实就是关于虚拟机上面的一些环境搭建了，

可以在home里面建一个文件夹把依赖打进去就可以了

```
peer lifecycle chaincode package atcc.tar.gz --path ../atcc/ --lang golang --label atcc_1.0
```

产生的压缩包复制到test-network后搭建环境就可了

## 案例二

---

```
package main
```

```
import (
```

```
    "fmt"
```

```
    "strconv"
```

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

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

```
)
```

```
type SimpleChaincode struct {
```

```
}
```

```
// 初始化数据状态，实例化/升级链码时被自动调用
```

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

```
    // println 函数的输出信息会出现在链码容器的日志中
```

```
// 获取用户传递给调用链码的所需参数
```

```
_, args := stub.GetFunctionAndParameters()
```

```
var A, B string // 两个账户
```

```
var Aval, Bval int // 两个账户的余额
```

```
var err error
```

```
// 检查合法性，检查参数数量是否为 4 个，如果不是，则返回错误信息
```

```
if len(args) != 4 {
```

```
    return shim.Error("Incorrect number of arguments. Expecting 4")
```

```
}
```

```
A = args[0] // 账户 A 用户名
```

```
Aval, err = strconv.Atoi(args[1]) // 账户 A 余额
```

```
if err != nil {
```

```
    return shim.Error("Expecting integer value for asset holding")
```

```
}
```

```
*B = args[2] // 账户 B 用户名*
```

```
*Bval, err = strconv.Atoi(args[3]) // 账户 B 余额*
```

```
if err != nil {
```

```
    return shim.Error("Expecting integer value for asset holding")
```

```
}
```

```
fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)
```

```

// 将账户 A 的状态写入账本中
err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
if err != nil {
    return shim.Error(err.Error())
}

// 将账户 B 的状态写入账本中
err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
if err != nil {
    return shim.Error(err.Error())
}

// 一切成功, 返回 nil (shim.Success)
return shim.Success(nil)
}

// 对账本数据进行操作时(query, invoke)被自动调用
**func (t SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {*
    fmt.Println("ex02 Invoke")*
    // 获取用户传递给调用链码的函数名称及参数
    *function, args := stub.GetFunctionAndParameters()

    // 对获取到的函数名称进行判断
    if function == "invoke" {
        // 调用 invoke 函数实现转账操作
        return t.invoke(stub, args)
    } else if function == "delete" {
        // 调用 delete 函数实现账户注销
        return t.delete(stub, args)
    } else if function == "query" {
        // 调用 query 实现账户查询操作
        return t.query(stub, args)
    }
    // 传递的函数名出错, 返回 shim.Error()
    return shim.Error("Invalid invoke function name. Expecting \"invoke\" \"delete\" \"query\"")
}

// 账户间转账
func (t SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response
{
    var A, B string // 账户 A 和 B
    var Aval, Bval int // 账户余额
    var X int // 转账金额
    var err error

    if len(args) != 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }

    A = args[0] // 账户 A 用户名
    B = args[1] // 账户 B 用户名

```

```

// 从账本中获取 A 的余额
Avalbytes, err := stub.GetState(A)
if err != nil {
    return shim.Error("Failed to get state")
}
if Avalbytes == nil {
    return shim.Error("Entity not found")
}
Aval, _ = strconv.Atoi(string(Avalbytes))

// 从账本中获取 B 的余额
Bvalbytes, err := stub.GetState(B)
if err != nil {
    return shim.Error("Failed to get state")
}
if Bvalbytes == nil {
    return shim.Error("Entity not found")
}
Bval, _ = strconv.Atoi(string(Bvalbytes))

// X 为 转账金额
X, err = strconv.Atoi(args[2])
if err != nil {
    return shim.Error("Invalid transaction amount, expecting a integer value")
}

// 转账
Aval = Aval - X
Bval = Bval + X
fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

// 更新转账后账本中 A 余额
err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
if err != nil {
    return shim.Error(err.Error())
}

// 更新转账后账本中 B 余额
err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
if err != nil {
    return shim.Error(err.Error())
}

return shim.Success(nil)
}

// 账户注销
*func (t SimpleChaincode) delete(stub shim.ChaincodeStubInterface, args []string) pb.Response
{
    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting 1")
    }

    A := args[0] // 账户用户名

```



```

// 从账本中删除该账户状态
err := stub.DelState(A)
if err != nil {
    return shim.Error("Failed to delete state")
}

return shim.Success(nil)

}

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

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

    A = args[0] // 账户用户名

    // 从账本中获取该账户余额
    Avalbytes, err := stub.GetState(A)
    if err != nil {
        jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
        return shim.Error(jsonResp)
    }

    if Avalbytes == nil {
        jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
        return shim.Error(jsonResp)
    }

    jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
    fmt.Printf("Query Response:%s\n", jsonResp)
    // 返回转账金额
    return shim.Success(Avalbytes)
}

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

```

## 案例二相关函数补充说明

### Init 方法

在链代码首次部署到区块链网络时调用，将由部署自己的链代码实例的每个对等节点执行。此方法可用于任何与初始化、引导或设置相关的任务，一般情况下仅被调用一次。

注意：值得注意的是chaincode升级同样会调用该函数。当我们编写的chaincode会升级现有chaincode时，需要确保适当修正Init函数

### Query 方法

只要在区块链状态上执行任何读取/获取/查询操作，就会调用 Query 方法。根据链代码的复杂性，此方法含有你的读取/获取/查询逻辑，或者它可以外包给可从 Query 方法内调用的不同方法。

Query 方法不会更改底层链代码的状态，因此它不会在交易上下文内运行。如果尝试在 Query 方法内修改区块链的状态，将出现一个错误显示缺少交易上下文。另外，因为此方法仅用于读取区块链的状态，所以对它的调用不会记录在区块链上。

### Invoke 方法

只要修改区块链的状态，就会调用 Invoke 方法。简言之，所有创建、更新和删除操作都应封装在 Invoke 方法内。因为此方法将修改区块链的状态，所以区块链 Fabric 代码会自动创建一个交易上下文，以便此方法在其中执行。对此方法的所有调用都会在区块链上记录为交易，这些交易最终被写入区块中。

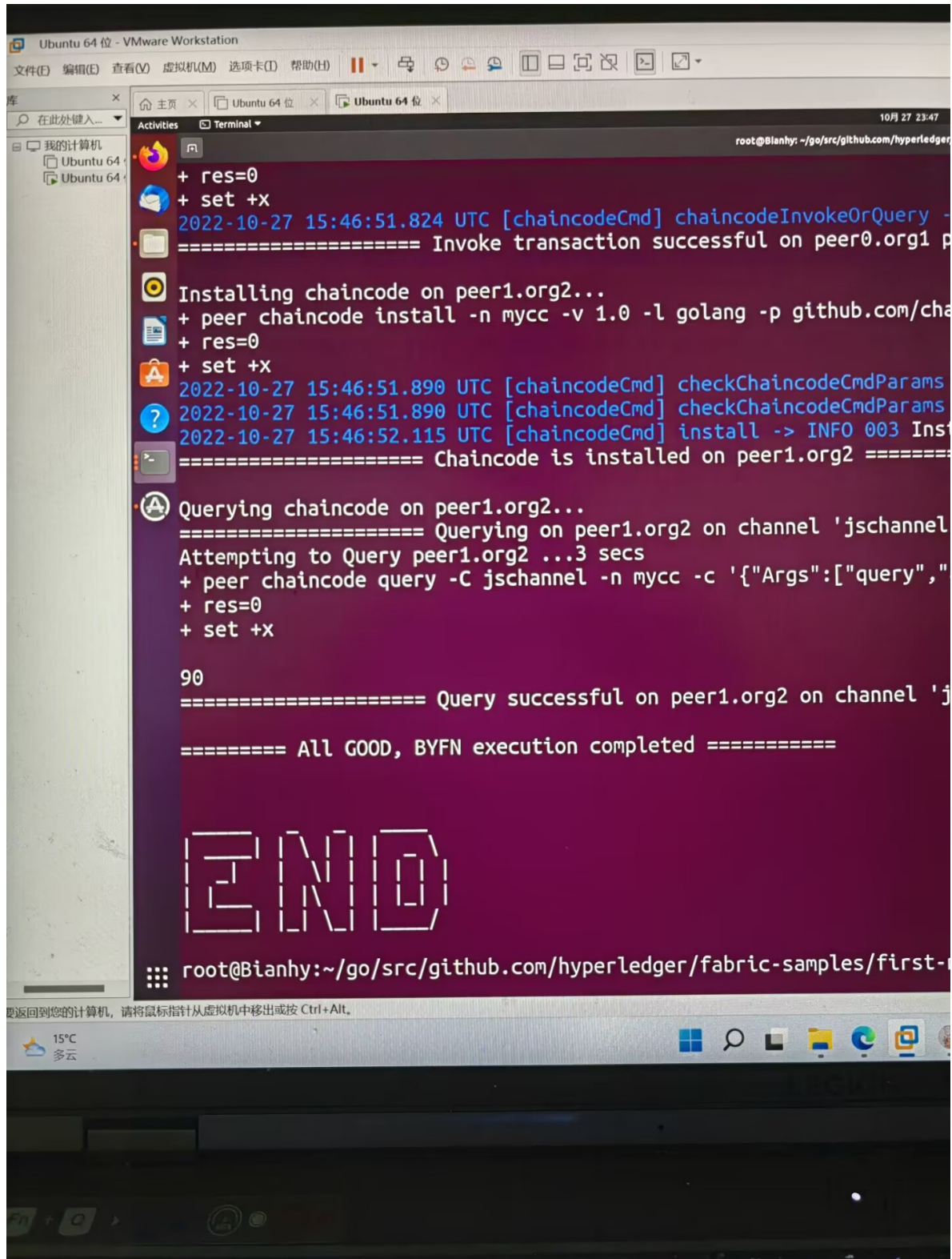
### 使用shim.Start 启动链码

在main函数中shim.Start(new(SampleChaincode)) 启动了链码并向对等节点注册它

## 4.搭建环境

---

附图



The image shows a VMware Workstation window titled 'Ubuntu 64 位 - VMware Workstation'. Inside, there is a terminal window with a dark background and light-colored text. The terminal output shows a series of commands and their results for installing and querying a chaincode on a peer. The commands include setting environment variables, invoking a transaction, installing the chaincode, and querying it. The output indicates that the chaincode was installed successfully on peer1.org2 and the query was also successful. The terminal ends with a large 'END' in a stylized font and a prompt line showing the user is root at Bianhy in the directory ~/go/src/github.com/hyperledger/fabric-samples/first-.

```
root@Bianhy: ~/go/src/github.com/hyperledger/fabric-samples/first-
+ res=0
+ set +x
2022-10-27 15:46:51.824 UTC [chaincodeCmd] chaincodeInvokeOrQuery -
===== Invoke transaction successful on peer0.org1 p
Installing chaincode on peer1.org2...
+ peer chaincode install -n mycc -v 1.0 -l go -p github.com/cha
+ res=0
+ set +x
2022-10-27 15:46:51.890 UTC [chaincodeCmd] checkChaincodeCmdParams
2022-10-27 15:46:51.890 UTC [chaincodeCmd] checkChaincodeCmdParams
2022-10-27 15:46:52.115 UTC [chaincodeCmd] install -> INFO 003 Inst
===== Chaincode is installed on peer1.org2 =====
Querying chaincode on peer1.org2...
===== Querying on peer1.org2 on channel 'jschannel'
Attempting to Query peer1.org2 ...3 secs
+ peer chaincode query -C jschannel -n mycc -c '{"Args":["query","
+ res=0
+ set +x

90
===== Query successful on peer1.org2 on channel 'j
===== All GOOD, BYFN execution completed =====

END

root@Bianhy:~/go/src/github.com/hyperledger/fabric-samples/first-
```