链码案例2.0

下面是这个链码示例的完整的Go语言源代码，并未将其改写成摘要，而是尽量保持原汁原味，只是为节约空间作了少许语法上等价的压缩，并加上了一些注释和讲解。

```
package main //Go语言程序是以package为单位编译和连接的
import( //导入的模块/函数库
"errors"
"fmt"
"strconv"
"hyperledger/cci/appinit" //cci是“Chain Code Information”的缩写
"hyperledger/cci/org/hyperledger/chaincode/example02"
"hyperledger/ccs" //ccs是“Chain Code Support”的缩写
" github.com/golang/protobuf/proto " //Protobuf是底层的消息传递和远程过程调用机制
" github.com/hyperledger/fabric/core/chaincode/shim "
//Shim是Fabric提供给链码程序的接口, stub就是由Shim提供。
)
type ChaincodeExample struct {
} //定义名为ChaincodeExample的类。里面没有数据，但是有下面所定义的这些函数。
//这个类就代表着所部署的合约，部署这个合约时，以及每次调用这个合约的时候，
//都会在其main()函数(见下)中创建这个类，并调用交易请求中指定的目标函数。
//在部署一个合约的时候则会调用其Init()函数。
//Called to initialize the chaincode,部署一个链码的时候会调用其Init()函数。
func(t*ChaincodeExample) Init(stub shim.ChaincodeStubInterface, param *appinit. Init) error
//这是ChaincodeExample类里面的一个函数，在函数代码中以t代表该类对象。
//调用参数stub是一个shim. ChaincodeStubInterface类对象,见上面的导入。
//参数param是个appinit. Init类结构指针。
//这个函数有一个返回值，就是error，是个出错代码。
var err error
fmt. Printf("Aval= %d, Bval=%d\n", param.PartyA. Value, param.PartyB. Value)
//这些格式化的信息会被写入本进程的stdout通道，原是要在终端屏幕上显示的，//但被重定向到Pipe中，会被Fabric一侧的链码支持线程从与之对接的通道读出。
//Write the state to the ledger

err= t.PutState(stub, param.PartyA)
//调用本类中的函数PutState(),将调用参数param.PartyA写入数据库。
//注意虽然param是结构指针,但Go语言中用param.PartyA表示其所指的成分。
if err !=nil {return err}
err=t.PutState(stub, param.PartyB)
//再调用本类中的函数PutState(),将调用参数param.PartyB也写入数据库。
if err != nil {return err}
return nil
}
//Transaction makes payment of X units from A to B
func(t*ChaincodeExample) MakePayment(stub shim.ChaincodeStubInterface,
param*example02.PaymentParams) error
// MakePayment()也是ChaincodeExample类中的一个函数,也是一个返回值error。
//本函数用于支付转账
//这里的参数param是个example02.PaymentParams类的结构指针
var err error //定义一个类型为error的局部量err
```

```
//Get the state from the ledger,从账本(数据库)读取付款方的余额:
src, err :=t.GetState(stub, param.PartySrc)
if err !=nil {return err}
dst, err:=t.GetState(stub, param.PartyDst) //从账本(数据库)读取收款方的余额
if err!=nil {return err}
//Perform the execution
X:= int(param. Amount) //转账金额在参数param. Amount中
src=src-X //从付款方余额中减去转账金额
dst=dst+X //在收款方余额中加上转账金额
fmt. Printf("Aval = %d, Bval= %d\n", src, dst) //会被重定向到Pipe中
//Write the state back to the ledger
err=stub.PutState(param.PartySrc,[]byte(strconv. Itoa(src)))
//将付方余额转换成文字形式并写回账本。
if err != nil { return err }
err=stub.PutState(param.PartyDst,[]byte(strconv. Itoa(dst)))
//将收方余额也转换成文字形式并写回账本。
if err != nil { return err}
return nil

}
//Deletes an entity from state
func(tChaincodeExample) DeleteAccount(stub shim.ChaincodeStablntertice.
paramexample02. Entity) emver/
//Delete the key from the state in ledger,从账本中删去一个账户。
err:=stub.DelState(param. Id) //调用由stub提供的DeIState()函数
if err!=nil { return errors. New("Failed to delete state")}
return nil
//Query callback representing the query of a chaincode
func(tChaincodeExample) CheckBalance(stub shim.ChaincodeStubInterface,
paramexample02. Entity)(example02.BalanceResult, error)
//查询账户的余额
//这个函数有两个返回值,一个是example02.BalanceResult结构指针,另一个是error.
var err error
//Get the state from the ledger
val, err :=t.GetState(stub, param. Id)
if err !=nil {return nil, err}
fmt. Printf("Query Response: %d\n", val) //被重定向到Pipe中由链码支持线程读出。
return&example02.BalanceResult{Balance:proto.Int32(int32(val))}, nil
}
func main() {//链码进程的主函数。每当本链码程序被调用时，就从这里开始执行。
self:= &ChaincodeExample{}
interfaces:=ccs. Interfaces{"org. hyperledger. chaincode.example02":self,"appinit":self,]
err:=ccs. Start(interfaces) //启动与Fabric进程中的链码支持线程对接
//要到程序从被交易请求调用的函数返回以后，才会从这个函数返回，
// Our one instance implements both Transactions and Queries interfaces
if err != nil {fmt. Printf("Error starting example chaincode: %s", err) }
}
//Helpers

func(tChaincodeExample) PutState(stub shim. ChaincodeStubInterface,
partyappinit. Party) error {
return stub.PutState(party. Entity,[]byte(strconv. Itoa(int(party. Value))))
//前面对t.PutState()的调用,实际上是通过stub.PutState()完成的。
```

```
//传输前须通过Itoa()把整型数值转换成字符串形式。
}
func(t*ChaincodeExample) GetState(stub shim. ChaincodeStubInterface,
entity string)(int, error){
bytes, err:=stub. GetState(entity)//对t.GetState()的调用是通过stub. GetState()完成的。
if err!=nil {return 0, errors. New("Failed to get state")}
if bytes== nil {return 0, errors. New("Entity not found")}
val,_:= strconv. Atoi( string( bytes)) //把从账本读出的字符串转换成整型数
return val, nil
}
```

超级账本网络中只有三种交易，即链码部署、链码调用和查询。其实查询是不经过背书节点评审，也不进入区块链的，所以实际上就只有两种，就是链码部署和链码调用。

如前所述，链码程序是在Docker所提供的“沙箱”中作为独立进程运行，而在超级账本的网络节点软件Fabric进程里面则会创建与其对接的链码支持线程。链码支持线程与链码进程之间有两种连接，一个是通过Socket互连的消息通道，另一个是通过管道Pipe把链码进程的stdin、stdout、srderr三个标准通道重定向到链码支持线程。

链码程序需要“导入(import)”由超级账本SDK即软件开发包提供的一些模块，其中特别重要的是shim。所谓shim，是夹在链码程序与链码支持线程之间的接口层,其中最关键的是对象ChaincodeStubInterface,实际上就是Fabric为链码程序提供的接口。具体链码提供给用户在交易中调用的函数，其第一个调用参数总是个名为stub的shim.ChaincodeStubInterface结构,这个参数无需调用者给出,是由shim模块自动插入的。这样，通过调用由stub提供的一些(标准)函数，链码程序就能与Fabric一侧的链码支持线程交互。

比方说,调用stub. GetState(), shim模块中的这个函数就会:

•向链码支持线程发送一个类型为ChaincodeMessage_GET_STATE的链码消息,要求对方替它从账本，即账户数据库中读出某个状态变量的值，实际上就是某个账户的余额。

·而链码支持线程，则从数据库中读出目标数据，生成并发回一个类型为ChaincodeMessage_RESPONSE的消息。

•收到对方发回的消息， shim模块从中取出搭载的数据，将其返回给stub.GerStane()的调用者。

函数stub.GetState()是这样, stub.PutState()也是一样,只是把向链码支持线程发送的信息类型改成ChaincodeMessage_PUT_STATE,要求把数据写入数据库。这样, 链码进程和Fabric进程中的链码支持线程就能互相配合，直至本次链码执行完成。即从被(交易请求)调用的函数返回。

在链码部署交易中，交易发起者上传所欲部署的链码程序给各个背书节点，这些节点如常加以检验和执行，并加以签名背书和发回对提案的回复。不过此时的执 行一方面是把链码程序部署在文件系统中，另一方面也要执行一下链码，调用其函数Init()。所有的链码程序，不管是否有实际的需要，都要有个函数Init()，供部署 链码的时候调用。注意所谓“执行一下”这个链码就是调用其main()函数，这就启动了进程的运行，就像启动了一个服务进程；而对其Init()函数的调用则相当于由链码支持线程对其进行RPC远程调用。

这个示例中的Init()，其实是没有什么要做的，但是这里创建了两个账户，这是在为以后在转账交易中对MakePayment()的调用做好准备。所创建的具体账户PartyA 和PartyB,是在链码部署交易中作为参数传入的。注意PartyA和PartyB是两个结构，结构中有Id和Value两个成分，那就是账户Id及其余额。注意这只是示例，在实际使用的链码程序中应该是不会这样做的，不会在部署链码的时候附带就为某些用户开了账户，而是会提供专门用于开户和注资的函数。

部署了链码之后，就可以在交易请求中调用这个链码了。所谓调用这个链码，其实是调用这个链码向外提供的某个函数,例如MakePayment(),例如CheckBalance(),上面代码中凡是第一个参数为stub的都是。每个链码调用交易都只调用其中的一个函数。

每当受到调用时， Docker都会启动这个链码程序作为一个进程运行，都会从调用它的main()函数开始执行程序。这里的main()函数中调用ccs. Start(),这个函数建立起本进程与对应链码支持线程的连接，根据对方的要求调用本地向外提供的某个函数，例如MakePayment()，到程序从所调用的函数返回后，就向链码支持线程发送一个类型为ChaincodeMessage_COMPLETED的链码消息,并把所产生的读写数据集搭载在这个信息中，然后就返回到main()中，结束链码程序的本次运行。所以，对ccs. Start()的调用要到被交易请求调用的那个链码函数结束运行返回以后才会返回。

其余就留给读者自己阅读了，代码中已经加了注释。

注意这个示例并不意味着在超级账本中只能采用账户模式，在超级账本中也可以通过不同的链码实现UTXO模式，事实上超级账本的代码中也确实有一个链码示例是采用UTXO模式的。而且，UTXO模式与账户模式也可以并存。这是因为，超级账本的一切交易都得通过链码，即智能合约的执行，而智能合约是用户可以提交部署的，用户可以在其提交的链码中自行实现账户模式的机制。而调用这个(或者一些)链码的用户，就会在超级账本区块链网络上形成一个使用账户模式的特定的小圈子。