Hyperledger Fabric 链码开发介绍

前面讲解了Hyperledger Fabric的环境搭建，搭建好环境后就可以开发Fabric了。

Fabric的智能合约称为链码(Chaincode)，分为系统链码和用户链码。系统链码用来实现系统层面的功能，用户链码实现用户的应用功能。链码被编译成一个独立的应用程序，运行于隔离的Docker容器中。

相较于以太坊，Fabric链码和底层账本是分开的，升级链码时并不需要迁移账本数据到新链码当中，真正实现了逻辑与数据的分离。

链码支持采用Go、Java、Node.js语言编写，对于大多数开发人员来说并不陌生，能快速上手。

以下的章节内容对应 `Hyperledger Fabric` 官方英文文档里面 `Chaincode for Developers` 一篇文章，介绍链码的开发。

因此，本篇算是中文翻译版吧，原文文章网址如下：

```
http://hyperledger-fabric.readthedocs.io/en/latest/chaincode4ade.html
```

1、什么是链码?

链码是一个用Go、Node.js编写的实现了规定接口的程序。事实上，它也支持其他语言，比如Java。链码在一个安全的Docker容器中运行，该容器与背书对等进程隔离。链码通过应用程序提交的事务来初始化和管理账本状态。

链代码通常处理由网络成员同意的业务逻辑，因此它类似于"智能合约"。可以调用链码来更新或者查询交易提案中的账本。考虑到适当的许可，一个链码可以调用另一个链码，无论是在同一通道还是在不同的通道中，来访问其状态。请注意，如果被调用的链码与调用的链码在不同的通道上，则只允许读取和查询。也就是说，不同通道上被调用的链码只是一个查询，它在随后的提交阶段不参与状态验证检查。

在接下来的章节中，我们将以应用程序开发人员的角度来探索链码。我们将介绍一个简单的链码应用程序示例，并介绍链码中Shim API的各种方法的用途。

2、链码API

每个链码程序都必须实现 `Chaincode接口` ，

- Go

- node.js

其方法被调用用于响应接收到的事务。特别是当链码接收实例化或升级事务时，将调用Init方法，以便链码可以执行任何必要的初始化，包括应用程序状态的初始化。调用`Invoke`方法是为了响应接收调用事务来处理事务提案。

在链码"shim"所有API中，另外一个接口是`ChaincodeStubInterface`：

- Go
- node.js

用于访问和修改账本，并在链码之间进行调用。
在本教程中，我们将通过实现一个管理简单"资产"的简单链码应用程序来演示如何使用这些API。

3、简单资产链码

我们的应用程序是用来在账本上创建资产（键值对）的一个基本示例链码。

3.1 选择代码的位置

如果您还没有用过Go语言来编程，您需要需要先确保已经在您的电脑中正确安装了Go并做好了正确的配置。

现在您需要在`$GOPATH/src/`目录下为链码应用程序创建一个子目录。

为了简单起见，我们执行如下命令：

```
mkdir -p $GOPATH/src/sacc && cd $GOPATH/src/sacc
```

现在，让我们创建并编写源代码：

```
touch sacc.go
```

3.2 准备工作

首先，我们做一些准备工作。和每个链码一样，它实现了Chaincode接口，特别是`Init`和`Invoke`函数。
因此，让我们使用go import语句来导入链码的必要依赖库，我们将导入链码的shim包和peer protobuf包。 接下来，让我们添加一个结构SimpleAsset作为链码函数的接收器。

```
package main
```

```
import (
    "fmt"

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

// SimpleAsset实现简单的链式代码来管理资产
type SimpleAsset struct {
}
```

3.3 初始化链码

下一步，我们实现 `Init` 函数。

```
// 在链码初始化过程中调用Init来初始化任何数据
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {

}
```

> 请注意，链码升级时也会调用此函数。 在编写要升级的现有链码的代码时，请确保适当地修改 `Init` 函数。 特别是如果没有"迁移"或没有任何内容作为升级的一部分进行初始化，请提供一个空的"Init"方法。

接下来，我们将使用ChaincodeStubInterface.GetStringArgs函数检索 `Init` 调用的参数并检查合法性。在我们的例子中，应当接收的是一个键值对。

```
// 在链码初始化过程中调用Init来初始化任何数据。
// 请注意，链码升级也会调用此函数来重置或迁移数据。
// 因此，要小心避免无意中破坏账本数据的情况！
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  // 从交易提案中获取参数
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }
}
```

接下啦，我们已经确保该调用时有效的，我们将把初始化状态存储到账本中。要做到这一点，我们将调用ChaincodeStubInterface.PutState并以键值对为参数传入。假设一切顺利，返回一个初始化成功的peer.Response对象。

```
// 在链码初始化过程中调用Init来初始化任何数据。
// 请注意，链码升级也会调用此函数来重置或迁移数据。
// 因此，要小心避免无意中破坏账本数据的情况！
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  // 从交易提案中获取参数
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }

  // 在这里，通过调用stub.PutState()来设置任何变量或者资产

  // 我们在账本中存储key和value
  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)
}
```

3.4 调用链码

首先，我们添加 `Invoke` 函数的签名。

```
// 在链码每个事务中，Invoke会被调用。
// 每个事务都是由Init函数创建的资产，要么是'get'要么是'set'。
// Set方法可以通过指定新的键值对来创建新的资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response
 {

}
```

和上面的 `Init` 函数一样，我们需要从ChaincodeStubInterface中提取参数。 `Invoke` 函数的参数将会是要调用的链码应用程序中函数的名称。在我们的例子中，我们的应用程序只有两个参数： `set` 和 `get` ，它们允许设置资产的值或检索当前状态。我们首先调用ChaincodeStubInterface.GetFunctionAndParameters来提取链码应用函数名称和参数。

```
// 在链码每个事务中，Invoke会被调用。
// 每个事务都是由Init函数创建的资产，要么是'get'要么是'set'。
// Set方法可以通过指定新的键值对来创建新的资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response
 {
    // 从交易提案中提取函数和参数
    fn, args := stub.GetFunctionAndParameters()
```

```
}
```

接下来，我们将验证函数名称为`set`或`get`，并调用这些链码的应用函数。通过`shim.Success`或者`shim.Error`函数返回适当的响应，将响应序列化到gRPC protobuf消息中。

```
// 在链码每个事务中，Invoke会被调用。
// 每个事务都是由Init函数创建的资产，要么是'get'要么是'set'。
// Set方法可以通过指定新的键值对来创建新的资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response
 {
    // 从交易提案中提取函数和参数
    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())
    }

    // 将结果作为成功负载返回
    return shim.Success([]byte(result))
}
```

3.5 实现链码

如上所述，我们的链码应用程序实现了两个可以通过`Invoke`函数调用的函数。现在我们来实现这些功能。请注意，正如我们上面提到的那样，为了访问账本的状态，我们将利用链码的shim API的ChaincodeStubInterface.PutState和ChaincodeStubInterface.GetState函数。

```
// 设置在账本上存储的资产（包括key和value）
// 如果key存在，它将覆盖新值
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
}

// 获取返回指定资产key的value值
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
}
```

3.6 集合代码

最后，我们需要添加 `main` 函数，它将调用shim.Start函数。这是整个链码程序的完整源代码。

```
package main

import (
    "fmt"

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

// SimpleAsset实现简单的链式代码来管理资产
type SimpleAsset struct {
}

// 在链码初始化过程中调用Init来初始化任何数据。
// 请注意，链码升级也会调用此函数来重置或迁移数据。
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // 从交易提案中获取参数
    args := stub.GetStringArgs()
    if len(args) != 2 {
            return shim.Error("Incorrect arguments. Expecting a key and a va
```

```
lue")
    }

    // 在这里，通过调用stub.PutState()来设置任何变量或者资产

    // 我们在账本中存储key和value
    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)
}

// 在链码每个事务中，Invoke会被调用。
// 每个事务都是由Init函数创建的资产，要么是'get'要么是'set'。
// Set方法可以通过指定新的键值对来创建新的资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response
 {
    // 从交易提案中提取函数和参数
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else { // fn为nil时，假设为get
            result, err = get(stub, args)
    }
    if err != nil {
            return shim.Error(err.Error())
    }

    // 将结果作为成功负载返回
    return shim.Success([]byte(result))
}

// 设置在账本上存储的资产（包括key和value）
// 如果key存在，它将覆盖新值
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
}

// 获取返回指定资产key的value值
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函数在实例化时启动容器中的链码
func main() {
    if err := shim.Start(new(SimpleAsset)); err != nil {
            fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
    }
}
```

3.7 编译链码

现在，让我们执行如下命令编译您的链码。

> 译者注：
> 这一步可能会执行不了或者出错，可以跳过这一步操作。
> 国内使用"go get"命令经常会碰到无法网络的问题，使用VPN就能很好的解决，但是VPN基本被封了。

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

如果没有错误，现在我们可以继续下一步，测试链码。

3.8 使用开发模式测试

通常，链码由peer启动和维护。然而，在"开发模式"下，链码由用户构建并启动。在此模式下，

以快速编写、编译、运行、调试等周期周转的链码开发阶段是非常有用的。

我们通过利用示例开发网络中预先生成的排序和通道工件来启动“开发模式”。因此，用户可以立即进入编译链码和驱动调用的过程。

4、安装Hyperledger Fabric Samples

如果您还没有这样做，请安装Samples示例和Docker镜像。

进入到 `fabric-samples` 下的 `chaincode-docker-devmode` 目录下。

```
cd chaincode-docker-devmode
```

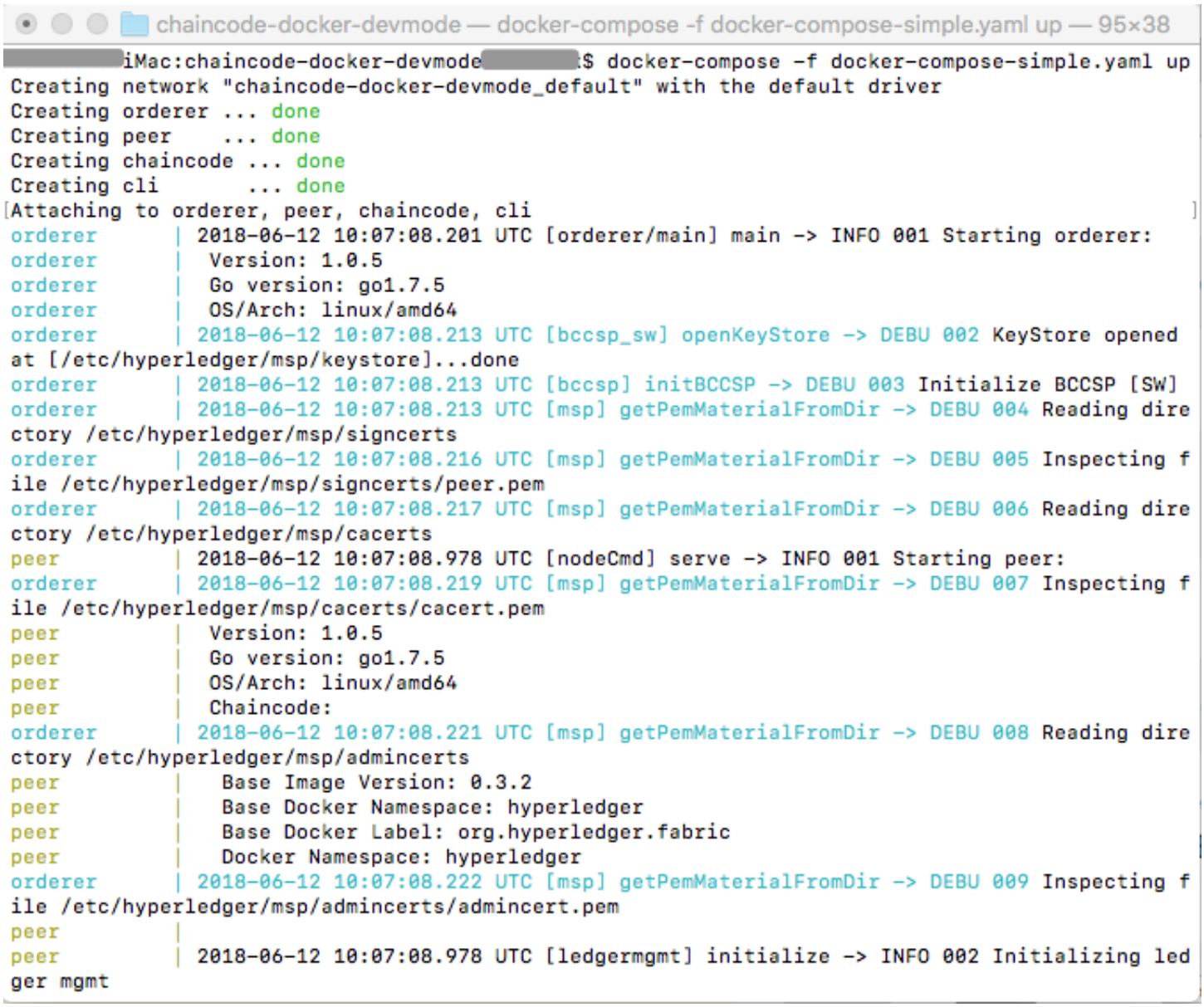
现在打开三个终端并进入到您的 `chaincode-docker-devmode` 目录下。

5、终端1 - 开启网络

```
docker-compose -f docker-compose-simple.yaml up
```

以上命令使用 `SingleSampleMSPSole` 配置启动 orderer并且以"开发模式"启动节点peer。它还启动了两个额外的容器，一个用于链码环境，一个用于与链码交互的CLI。创建和加入通道的命令被嵌入到CLI容器中， 因此我们可以立即跳转到链码的调用。

> 译者注：
> 如果报错了，试着清除Docker容器，依次执行以下命令
> // 删除所有活跃的容器（失败重试会用到）
> docker rm -f $(docker ps -aq)
> // 清理网络缓存（失败重试会用到）
> docker network prune



6、终端2 - 编译和运行链码

```
docker exec -it chaincode bash
```

你应该看到以下内容：

```
root@d2629980e76b:/opt/gopath/src/chaincode#
```

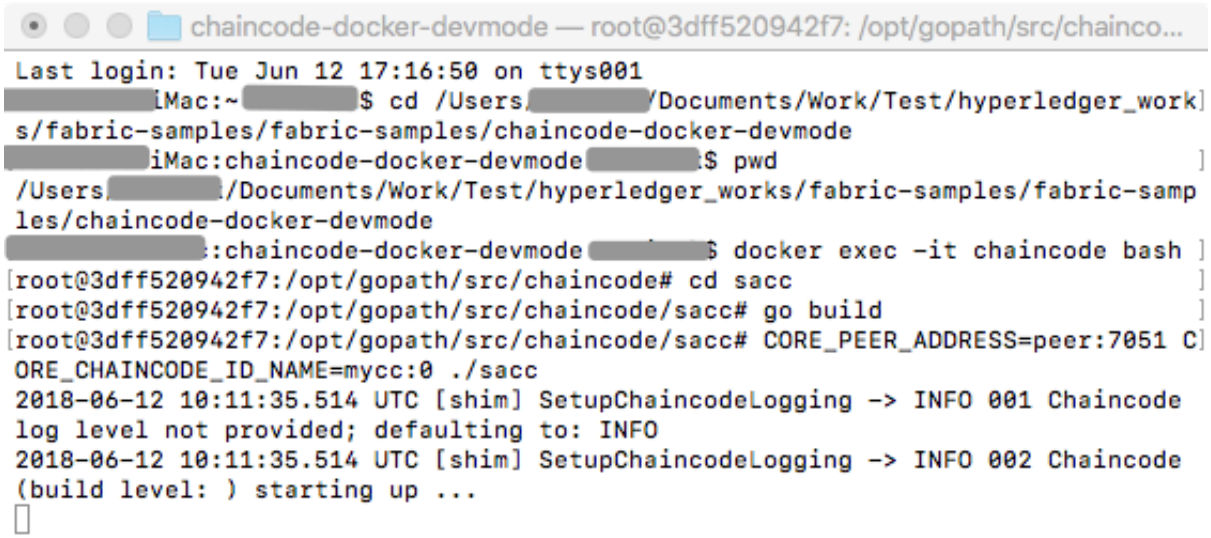
现在，编译您的链码：

```
cd sacc
go build
```

运行您的链码：

```
CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=mycc:0 ./sacc
```

链码随着peer节点启动并且在peer节点成功注册后链码日志会开始显示。请注意，在此阶段，链码与任何通道都没有关联。这个会在后续步骤中使用实例化命令完成。



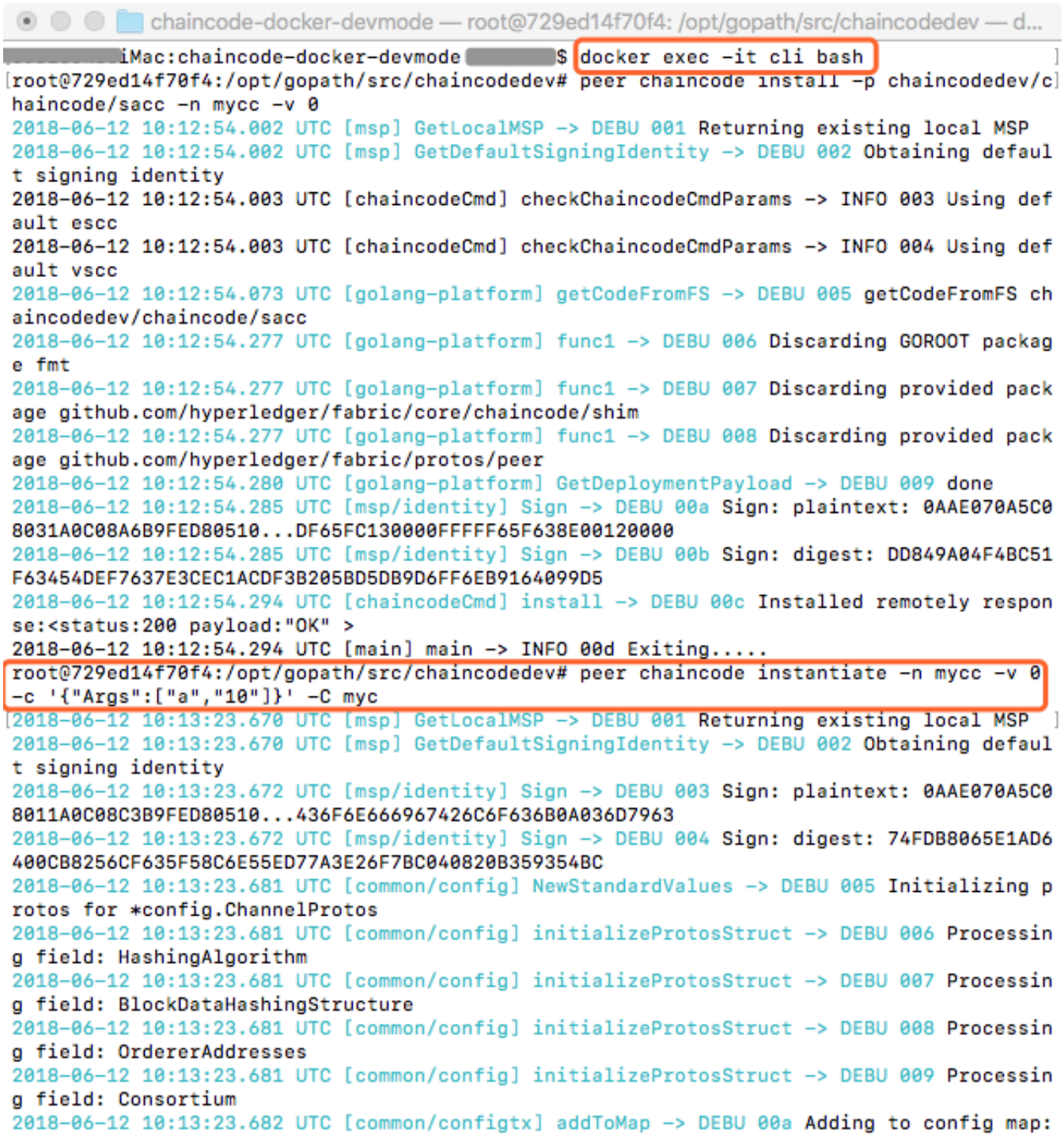
7、终端3 - 使用链码

即使您处于 `--peer-chaincodedev` 模式，您仍然必须安装链码，以便生命周期链码可以正常检查。在 `--peer-chaincodedev` 模式下，这个要求未来可能会被移除。

我们将利用CLI容器来驱动这些调用：

```
docker exec -it cli bash
```

```
peer chaincode install -p chaincodedev/chaincode/sacc -n mycc -v 0
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc
```



现在，执行一个调用将"a"的值变为“20”。

```
peer chaincode invoke -n mycc -c '{"Args":["set", "a", "20"]}' -C myc
#####输出结果如下#####
2018-06-12 10:13:31.785 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 0a6 Chaincode invoke successful. result: status:200 payload:"20"
```

最后，查询 `a` 。我们将会看到 `20` 的值。

```
peer chaincode query -n mycc -c '{"Args":["query","a"]}' -C myc
#####输出结果如下#####
```

```
Query Result: 20
```

8、测试新链码

默认情况下，我们只安装 `sacc` 。然而，您可以通过将链码添加到chaincode子目录中并重新启动网络来轻松地测试不同的链码。在这个点上，它们将可以在您的链码容器中被访问。

9、链码加密

在某些情况下，加密与密钥相关的值可能是完整或部分加密的。例如，如果一个人的社会安全号码或地址正在写入账本中欧，那么您可能不希望这些数据以明文形式出现。链码加密通过利用实体扩展实现，该实体扩展是具有商品工厂和功能的BCCSP包装以执行诸如加密和椭圆曲线数字签名的加密操作。例如，为了加密，链码的调用者通过序列化字段传递加密密钥。然后可以将相同的密钥用于随后的查询操作，从而允许对加密的状态值进行适当的解密。

有关更多信息和示例，请参阅 `fabric/examples` 目录中的Encc Example示例。特别注意 `utils.go` 帮助程序。该实用程序加载链码API和实体扩展，并构建样本加密链码随后利用的新类函数（例如， `encryptAndPutState` & `getStateAndDecrypt` ）。因此，链码现在可以结合 `Get` 和 `Put` 的基本填充API以及 `Encrypt` 和 `Decrypt` 的附加功能。

10、管理用Go编写的链码的外部依赖关系

如果您的链码需要非Go标准库提供的软件包，则需要将这些软件包包含在您的链码中。有许多工具可用于管理（或"声明"）这些依赖关系。以下演示如何使用 `govendor` ：

```
govendor init
govendor add +external  // Add all external package, or
govendor add github.com/external/pkg // Add specific external package
```

这将外部依赖关系导入本地 `vendor` 目录。 `peer chaincode package` 和 `peer chaincode install` 操作将导入与链码包相关的代码。