Web应用实现

基础环境

此应用示例是只是为了帮助读者快速掌握使用 fabric-sdk-go 开发简单的web应用。

本应用实现是在基于 Ubuntu 16.04（推荐） 系统的基础上完成的，但 Hyperledger Fabric 与Mac OS X、Windows和其他Linux发行版相兼容。

所需环境及工具

```
Ubuntu 16.04
vim、git
docker 17.03.0-ce+
docker-compose 1.8+
Golang 1.10.x+(建议使用1.13及以上)
GoLang(建议使用，JetBrain全家桶)
```

MVC是什么：合理的设计我们的应用

MVC

应用程序开发人员在工作中都需要考虑一些问题，如：

- 如何简化开发
- 如何降低应用程序的耦合性
- 如何提高代码的重用性
- 如何提高应用程序的可扩展性及维护性
- ..... 1979年，由 Trygve Reenskaug 在 Smalltalk-80 系统上首次提出了 MVC 的概念，主要核心就是由专业的人做专业的事。MVC 模式代表 Model（模型）－View（视图）－Controller（控制器）模式。这种模式主要应用于应用程序的分层开发。将应用程序中显示什么数据、数据由谁处理、怎么处理进行分离，可以由不同的开发人员专注于自己的领域，而无需关注其它。

MVC架构概念

MVC将应用程序划分为三种组件，并明确定义了它们之间的相互作用：

- 模型（Model）： 用于封装与应用程序的业务逻辑相关的数据以及对数据的处理法。"Model"有对数据直接访问的权力，例如对数据库的访问。"Model"不依赖"View"和"Controller"，也就是说， Model 不关心它会被如何显示或是如何被操作。但是 Model 中数据的变化一般会通过一种刷新机制被公布。为了实现这种机制，那些用于监视此 Model 的 View 必须事先在此 Model 上注册，从而，View 可以了解在数据 Model 上发生的改变
- 视图（View）： 能够实现数据有目的的显示。在 View 中一般没有程序上的逻辑。为了实现 View 上的刷新功能，View 需要访问它监视的数据模型（Model），因此应该事先在被它监视的数据那里注册
- 控制器（Controller）： 可以将控制器理解为一个桥梁，处理事件并作出响应。负责从视图读取数据，控制用户输入，并向模型发送数据

MVC架构优缺点

不论是什么技术，使用什么理念，尤其是能够经过长时间的应用实践，都有其显著的优点，有优点，也必然存在一些缺点，下面我们来看一下 MVC 模式的优缺点。

优点

- 高内聚性 – MVC可以在控制器上对相关操作进行逻辑分组。特定模型的视图也可以组合在一起。
- 耦合性低 – MVC框架的本质是模型，视图或控制器之间的耦合较低
- 重用性高 – 允许使用各种不同样式的视图来访问同一个服务器端的代码；将数据和业务规则从表示层分开，可以最大化的重用代码。模型也有状态管理和数据持久性处理的功能
- 提高可扩展性与可维护性 – 由于分离视图层和业务逻辑层也使得WEB应用更易于维护和修改
- 利于项目的管理 – 由于不同的层各司其职，每一层不同的应用具有某些相同的特征，有利于通过工程化、工具化管理程序代码。控制器也提供了一个好处，就是可以使用控制器来联接不同的模型和视图去完成用户的需求
- 简化了分组开发 – 不同的开发人员可同步开发应用项目中的视图、控制器逻辑和业务逻辑 **缺点**
- 增加系统结构及代码量 – 对于简单的界面，严格遵循MVC，使模型、视图与控制器分离，会增加结构的复杂性及相应的代码量
- 不适用于中小型应用项目 – 将MVC应用到规模并不是很大的应用程序通常会降低开发效率，增加开发成本
- 显著的学习曲线 – 关于掌握多种技术的知识成为常态。使用MVC的开发人员需要熟练掌握多种技术

MVC架构的实际应用

MVC 不是一种技术，而是一种理念。下面是一个通过 JavaScript 所实现的基于 MVC 模型，在这个简短的代码中就写成了一个具有完整 MVC 架构模式概念的示例。

```
/** 模拟 Model, View, Controller */
var M = {}, V = {}, C = {};

/** Model 负责存储数据 */
M.data = "hello world";

/** View 负责将数据显示在屏幕上 */
V.render = (M) => { alert(M.data); }

/** Controller 作为一个 M 和 V 的桥梁 */
C.handleOnload = () => { V.render(M); }

/** 在网页加载时调用 Controller */
window.onload = C.handleOnload;
```

调用链码－设计业务层

fabric-sdk 不仅提供了相应的强大功能，而且还给开发人员设计提供了相应的API 接口，以方便开发人员随时调用。做为开发设计人员，我们不仅要考虑用户操作的方便性及可交互性，还需要考虑应用程序后期的可扩展性及维护性，为此我们将为应用增加一个业务层，所有的客户请求都由业务层发送给链码，通过对链码的调用，进而实现对分类账本状态的操作。

事件处理

在项目根目录下创建一个 service 目录作为业务层，在业务层中，我们使用 Fabric-SDK-Go 提供的接口对象调用相应的 API 以实现对链码的访问，最终实现对分类账本中的状态进行操作。

```
$ cd $GOPATH/src/github.com/kongyixueyuan.com/kongyixueyuan
$ mkdir service
```

在 service 目录下创建 domain.go 文件并进行编辑， 声明一个结构体及对事件相关而封装的源代码

```
$ vim service/domain.go
```

domain.go 文件完整内容如下：

```
package service

import (
    "github.com/hyperledger/fabric-sdk-go/pkg/client/channel"
    "fmt"
    "time"
    "github.com/hyperledger/fabric-sdk-go/pkg/common/providers/fab"
)

type ServiceSetup struct {
    ChaincodeID    string
    Client    *channel.Client
}

func regitserEvent(client *channel.Client, chaincodeID, eventID string)
(fab.Registration, <-chan *fab.CCEvent) {

    reg, notifier, err := client.RegisterChaincodeEvent(chaincodeID,
eventID)
    if err != nil {
        fmt.Println("注册链码事件失败: %s", err)
    }
    return reg, notifier
}

func eventResult(notifier <-chan *fab.CCEvent, eventID string) error {
    select {
    case ccEvent := <-notifier:
        fmt.Printf("接收到链码事件: %v\n", ccEvent)
    case <-time.After(time.Second * 20):
        return fmt.Errorf("不能根据指定的事件ID接收到相应的链码事件(%s)",
eventID)
    }
```

```
    return nil
}
```

调用链码添加状态

在 service 目录下创建 SimpleService.go 文件

```
$ vim service/SimpleService.go
```

在 SimpleService.go 文件中编写内容如下，通过一个 SetInfo 函数实现链码的调用，向分类账本中添加状态的功能：

```
package service

import (
    "github.com/hyperledger/fabric-sdk-go/pkg/client/channel"
)

func (t *ServiceSetup) SetInfo(name, num string) (string, error) {

    eventID := "eventSetInfo"
    reg, notifier := regitserEvent(t.Client, t.ChaincodeID, eventID)
    defer t.Client.UnregisterChaincodeEvent(reg)

    req := channel.Request{ChaincodeID: t.ChaincodeID, Fcn: "set", Args:
[][]byte{[]byte(name), []byte(num), []byte(eventID)}}
    respone, err := t.Client.Execute(req)
    if err != nil {
        return "", err
    }

    err = eventResult(notifier, eventID)
    if err != nil {
        return "", err
    }

    return string(respone.TransactionID), nil
}
```

测试添加状态 编辑 main.go 文件

```
$ vim main.go
```

main.go 中创建一个对象，并调用 SetInfo 函数，内容如下：

```
package main

import (
    [......]
    "github.com/kongyixueyuan.com/kongyixueyuan/service"
)

[......]
    //=========================================//

    serviceSetup := service.ServiceSetup{
        ChaincodeID:SimpleCC,
        Client:channelClient,
    }

    msg, err := serviceSetup.SetInfo("hanxiaodong", "kongyixueyuan")
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println(msg)
    }

    //=========================================//

}
```

执行 make 命令运行应用程序

```
make
```

调用链码查询状态

通过上面的 setInfo(name, num string) 函数，实现了向分类账本中添加状态，那么我们还需要实现从该分类账本中根据指定的 key 查询出相应的状态，编辑 service/SimpleService.go 文件，向该文件中添加实现查询状态的相应代码。

```
$ vim service/SimpleService.go
```

定义一个 GetInfo 函数，接收一个字符串类型的参数，该函数实现通过调用链码而查询状态的功能，该函数完整代码如下:

```
[......]

func (t *ServiceSetup) GetInfo(name string) (string, error){

```

```
    req := channel.Request{ChaincodeID: t.ChaincodeID, Fcn: "get", Args:
[][]byte{[]byte(name)}}
    respone, err := t.Client.Query(req)
    if err != nil {
        return "", err
    }

    return string(respone.Payload), nil
}
```

测试查询状态 编辑 main.go 文件

```
$ vim main.go
```

在 main.go 文件中添加调用代码如下内容：

```
[......]

    msg, err = serviceSetup.GetInfo("hanxiaodong")
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println(msg)
    }

    //==========================================//

}
```

执行 make 命令运行应用程序

```
make
```

MVC架构应用实现

目录结构

通过业务层已经实现了利用 fabric-sdk-go 调用链码查询或操作分类账本状态，但是开发人员的工作不可能就此而止，需要考虑用户该如何使用此应用程序，一般情况下，交给用户使用的应用程序有以下两种方式：

- 桌面应用：传统实现方式，将应用程序打包成为一个可执行的安装程序之后，由用户安装在本地然后运行（可能需要特定的环境），进而进行相关操作。
- Web浏览器应用：此方式相对于用户而言，非常方便，用户只需要在本地的浏览器中就可以使用应用程序的相关功能。 为了方便用户的操作使用，我们使用第二种方式来实现。以便于让用户通过浏览器就可

> 以实现对分类账的操作。同样我们需要考虑应用程序后期的可扩展性及维护性，为此我们将应用程序进行了分层管理，设计增加了控制层及视图层。

视图层提供用户的可视界面与交互，控制层接收用户的请求，由控制层访问业务层，进而调用链码对分类账进行操作，之后将操作结果响应给客户端浏览器。

Go 语言本身提供了一个 Web 服务器来处理 HTTP 请求，并为 HTML 页面提供模板。下面我们来实现 Web 应用程序。

新建web目录，包含三个其他目录的目录。将使用 MVC（Model（模型）－View（视图） - Controller（控制器））模式使其更具可读性及扩展性、维护性。模型将是区块链部分，视图是模板，控制器由controllers目录中的功能提供。

- web/tpl：包含所有的HTML页面
- web/static：包含所有静态CSS，Javascript，图片等文件
- web/controllers： 包含将呈现模板的所有函数

```
$ cd $GOPATH/src/github.com/kongyixueyuan.com/kongyixueyuan
```

clone现成的代码：

```
git clone https://github.com/kevin-hf/hfsdkgoweb.git
```

web/controller 目录

controller/controllerHandler.go： 用于接收并处理各种客户端请求的源代码文件

controller/controllerResponse：用于编写响应客户端请求的源代码文件

web/static目录下包括三个子目录，分别为：

web/static/css： 用于存放页面布局及显示样式所需的 CSS 文件

web/static/js： 用于存放编写的与用户交互的 JavaScript 源码文件

web/static/img：用户存放页面显示所需的所有图片文件

web/tpl 目录下包括三个静态 HTML 页面文件，分别为：

web/tpl/index.html： 用户访问的首页面

web/tpl/queryReq.html： 用于显示显示查询结果的页面

web/tpl/setInfo.html： 用户设置/修改状态的页面

web/webServer.go：用于指定启动Web服务及相应的路由信息

启动Web服务

编辑 main.go ， 以便启动Web界面实现Web应用程序

```
$ vim main.go
```

添加如下内容:

```
import(
    [......]
    "github.com/kongyixueyuan.com/kongyixueyuan/web"
    "github.com/kongyixueyuan.com/kongyixueyuan/web/controller"
)

func main(){}
    [......]

    app := controller.Application{
        Fabric: &serviceSetup,
    }
    web.WebStart(&app)
}
```

执行 make 命令启动Web应用：

```
make
```

页面访问

打开浏览器访问: http://localhost:9000/