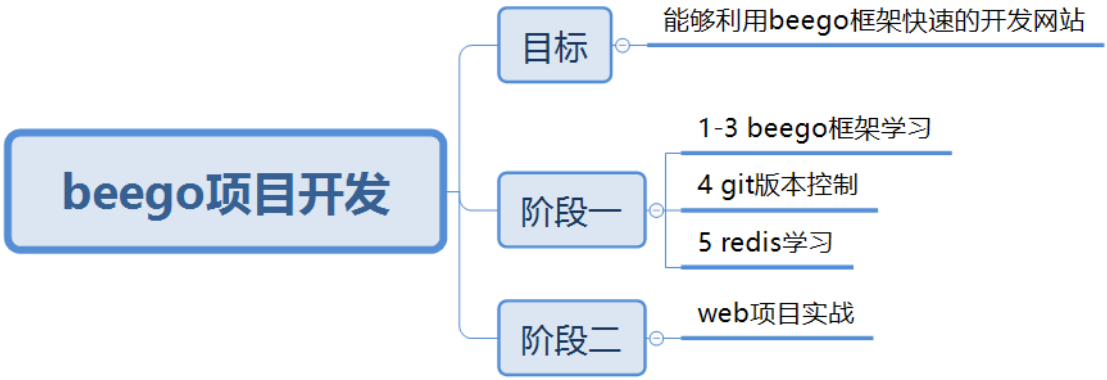
beego框架

1.课程规划



2.Beego框架快速入门

2.1beego框架了解

Beego作者：谢孟军

Beego框架是go语言开发的web框架。

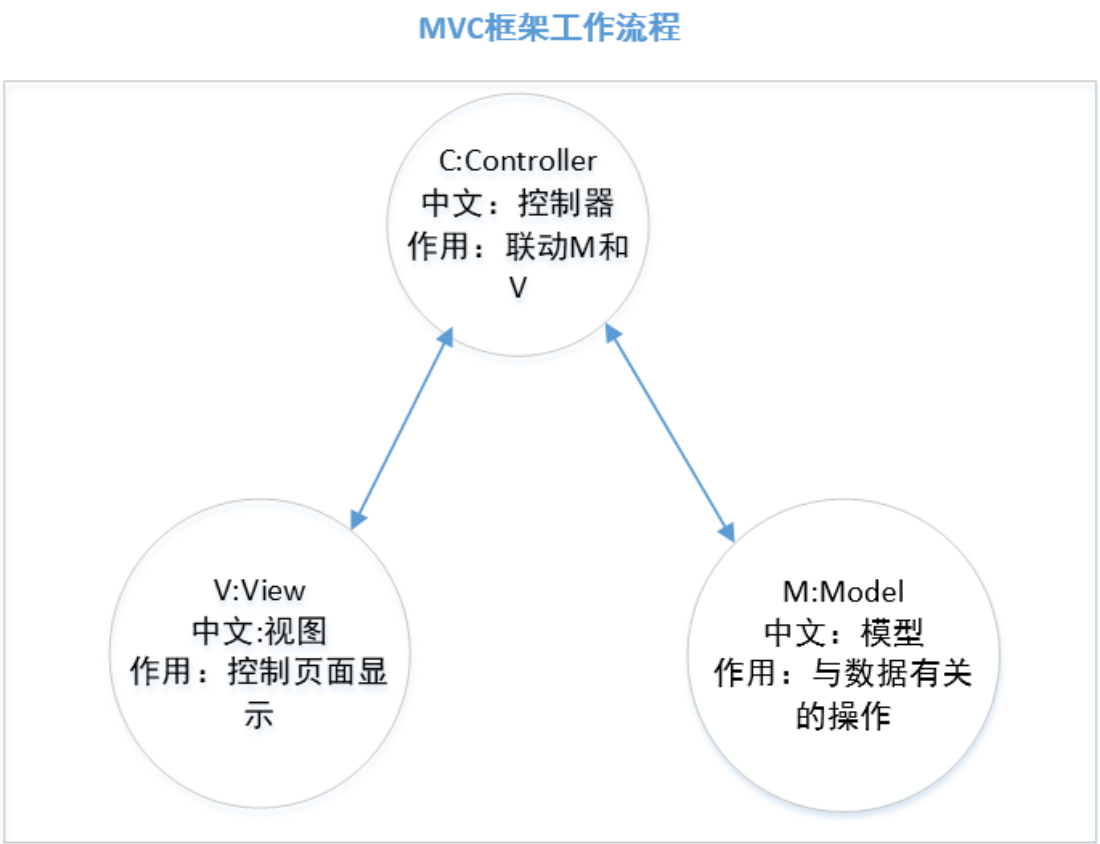
那什么是框架呢？就是别人写好的代码，我们可以直接使用！这个代码是专门针对某一个开发方向定制的，例如：我们要做一个网站，利用 beego 框架就能非常快的完成网站的开发，如果没有框架，每一个细节都需要我们处理，开发速度会大大降低。

go语言的web框架：beego,gin,echo等等，那为什么我们选择beego呢？

第一，beego是中国人开发的，开发文档比较详细，**beego官网网址:** https://beego.me/ 。第二，现在公司里面用beego的也比较多，比如今日头条，百度云盘，腾讯，阿里等。

2.2MVC架构

Beego是MVC架构。MVC 是一种应用非常广泛的体系架构，**几乎所有的编程语言都会使用到**，而且**所有的程序员在工作中都会遇到**！用 MVC 的方式开发程序，可以让程序的结构更加合理和清晰。 我们画图说明



beego具体是如何内嵌MVC呢？我们搭起环境通过代码分析。

2.3环境搭建

这里默认大家已经搭建好了go语言的开发环境。

- 需要安装Beego源码和Bee开发工具

```
$ go get -u -v github.com/astaxie/beego
$ go get -u -v github.com/beego/bee
```

beego源码大家都了解，就是框架的源码。

Bee开发工具带有很多Bee命令。比如`bee new`创建项目，`bee run`运行项目等。

用bee运行项目，项目自带**热更新**（是现在后台程序常用的一种技术，即在服务器运行期间，可以不停服替换静态资源。替换go文件时会自动重新编译。）

安装完之后，bee可执行文件默认存放在`$GOPATH/bin`里面，所以需要把`$GOPATH/bin`添加到您的环境变量中才可以进行下一步

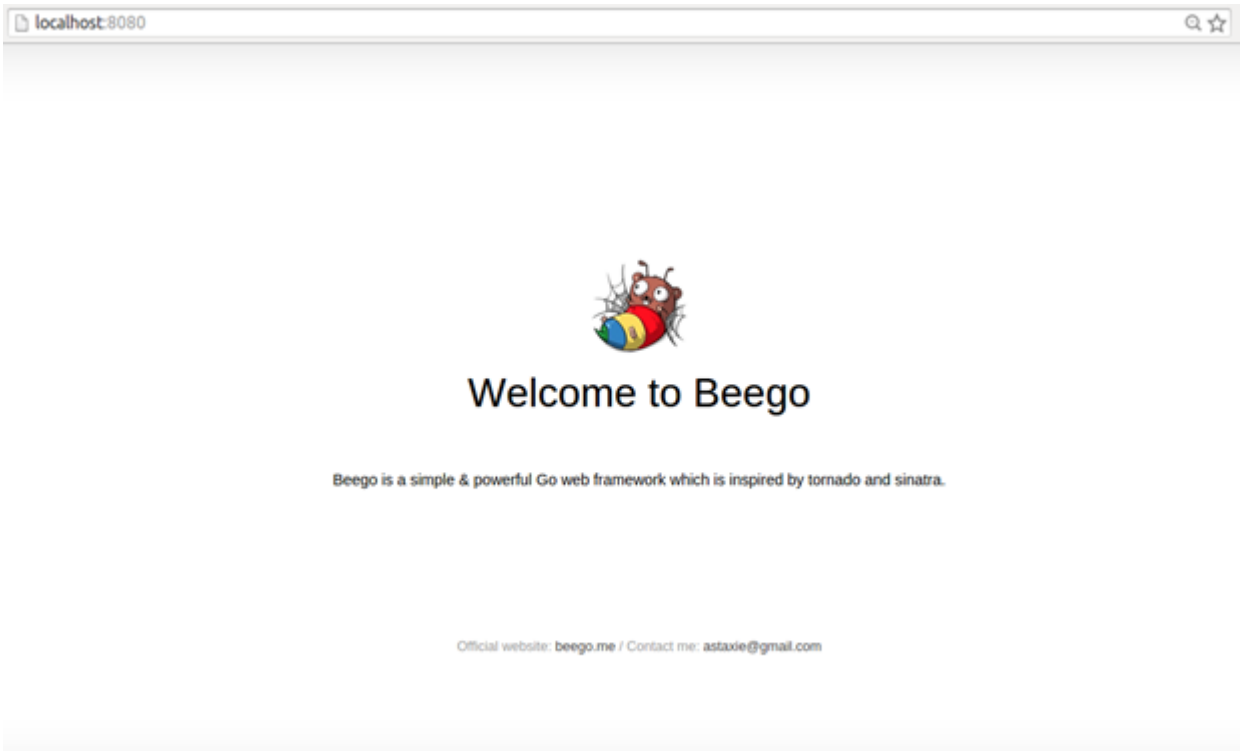
```
$ cd ~
$ vim .bashrc
//在最后一行插入
export PATH="$GOPATH/bin:$PATH"
//然后保存退出
$ source .bashrc
```

安装好之后，运行`bee new preojectName`来创建一个项目，注意：**通过bee创建的项目代码都是在`$GOPATH/src`目录下面的**

生成的项目目录结构如下:

```
quickstart
|-- conf
|   `-- app.conf
|-- controllers
|   `-- default.go
|-- main.go
|-- models
|-- routers
|   `-- router.go
|-- static
|   |-- css
|   |-- img
|   `-- js
|-- tests
|   `-- default_test.go
|-- views
    `-- index.tpl
```

进入项目目录 执行`bee run`命令，在浏览器输入网址：127.0.0.1：8080，显示如下：



2.4beego的项目结构分析

```
quickstart
|-- conf
|   `-- app.conf
|-- controllers
|   `-- default.go
|-- main.go
|-- models
|-- routers
|   `-- router.go
|-- static
|   |-- css
|   |-- img
|   `-- js
|-- tests
|   `-- default_test.go
|-- views
    `-- index.tpl
```

conf文件夹:放的是项目有关的配置文件

controllers:存放主要的业务代码

main.go:项目的入口文件

models:存放的是数据库有关内容

routers:存放路由文件，**路由作用是根据不同的请求指定不同的控制器**

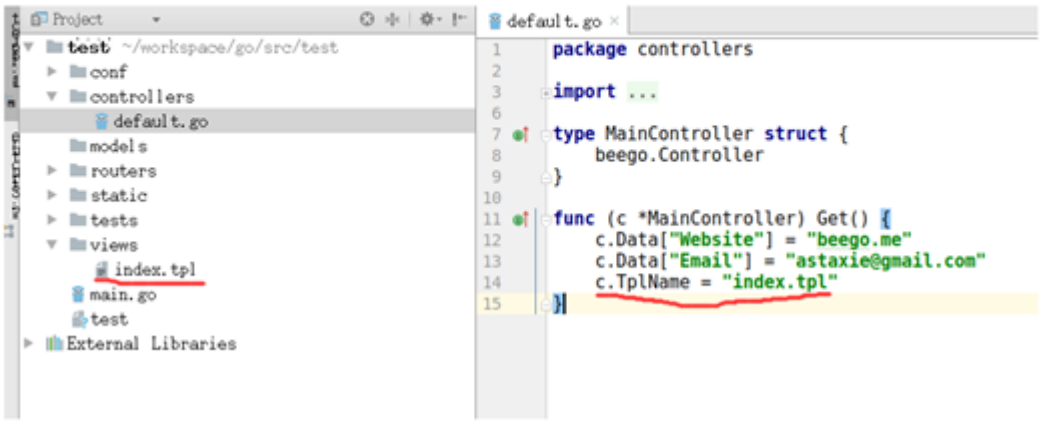
static：存放静态资源，包括图片，html页面，css样式，js文件等

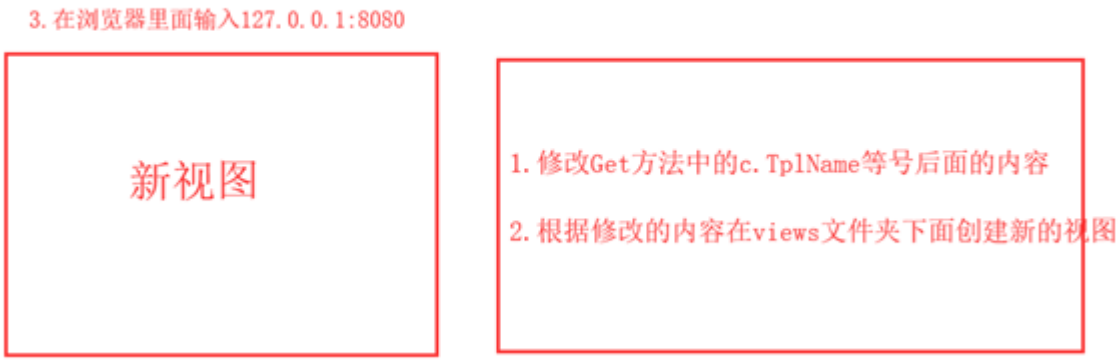
tests:测试文件

views：存放视图有关内容

后面我们重点需要操作的是MVC文件夹，routers文件夹。

2.5Beego快速体验





前面我们简单了解了 beego初始化的内容，那么就来个beego的快速体验吧！

根据上图所示的步骤，对自己创建的项目进行三步修改，然后在浏览器是否能看到修改之后的效果。

如果把你们前面做的静态网页放到views文件夹下呢？一个静态网站是不是就出现啦！有没有感受到beego开发网站的快捷！

代码分析

`c.Data["Email"] = "astaxie@gmail.com"`是给视图传递数据，在视图界面里面需要用`{{ }}`加上`.`才能获取到，比如这行代码的意思就是，给视图传递，**Key为Email，value为**astaxie@gmail.com 的数据。在视图中要通过`{{.Email}}`就能获取到value值。

`c.TplName = "index.tpl"`的作用是指定视图。这里面需要注意的是，默认指定的界面是tpl结尾，但是打开这个文件分析，发现还是一个html界面。所以我们也可以用html文件当视图文件。

通过我们对Beego的快速体验能够得出如下结论：

控制器(Controller)的作用

1.能够给视图传递数据

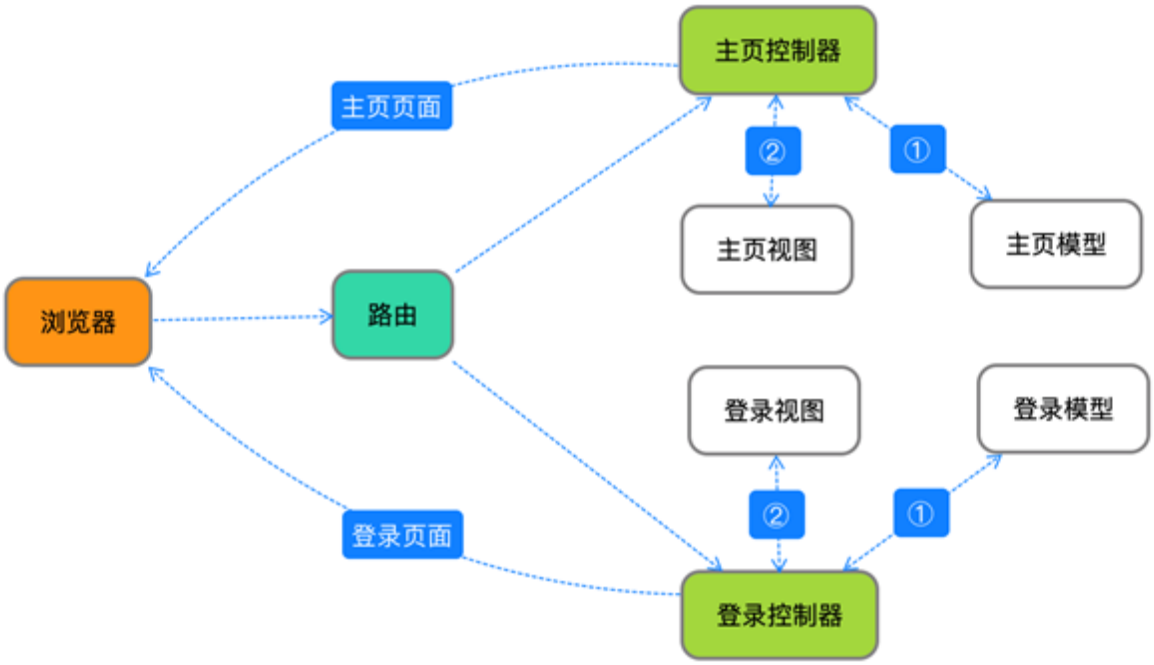
2.能够指定视图

视图(View)的作用

1.view本质就是个html。所以能在浏览器显示

2.能够接收控制器传递过来的数据

2.6Beego运行流程分析



- 浏览器发出请求
- 路由拿到请求，并给相应的请求指定相应的控制器
- 找到指定的控制器之后，控制器看是否需要查询数据库
- 如果需要查询数据库就找model取数据
- 如果不需要数据库，直接找view要视图
- 控制器拿到视图页面之后，把页面返回给浏览器

  根据文字流程分析代码流程
- 从项目的入口main.go开始
- 找到router.go文件的Init函数
- 找到路由指定的控制器文件default.go的Get方法
- 然后找到指定视图的语法，整个项目就串起来啦。

2.7Post案例实现

刚才我们分析了beego项目的整个运行流程，最终是如何调到Get方法的呢？**beego通过内部语法给不同的http请求指定了不同的方法**，因为我们是从浏览器地址栏发送的请求，属于get请求，所以调用的是Get方法。为了检验老师说的对不对，我们可以实现一个post请求，看看效果。

2.7.1前端修改

前端代码如下:

修改我们刚才创建的新的视图，为了能够发送post请求，我们在视图中添加一个能发送post请求的控件`form`

```
<form method="post" action="/index">
    <input type="submit">
</form>
```

然后设置一个能接收后台传递过来的数据的标签

```
<h1>hello {{.data}}</h1>
```

全部代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<form method="post" action="/index">
    <input type="submit">
</form>
<h1>hello {{.data}}</h1>

</body>
</html>
```

2.7.2后台代码修改

后台代码

先设置我们Get请求要传递的数据和要显示的视图页面

```
func (c *MainController) Get() {
    c.Data["data"] = "world"
    c.TplName = "test.html"  //渲染
}
```

再设置我们post请求要传递的数据和要显示的视图页面

```
func (c *MainController) Post() {
    c.Data["data"] = "区块链一期最棒"
    c.TplName = "test.html"  //渲染
}
```

操作

先在浏览器输入网址，然后点击页面上的按钮，看一下页面的变化，有没有出现`区块链一期最棒`几个字

2.8Beego中路由的快速体验

2.8.1路由的简单设置

路由的作用：**根据不同的请求指定不同的控制器**

路由函数：`beego.Router("/path",&controller.Main.Controller{})`

函数参数：

先分析一下Url地址由哪几部分组成?

http://192.168.110.71:8080/index

http://地址:端口/资源路径

第一个参数：资源路径，也就是`/`后面的内容

第二个参数：需要指定的控制器指针

了解上面的内容之后我们来看几个简单的例子：

```
beego.Router("/", &controllers.MainController{})
beego.Router("/index", &controllers.IndexController{})
beego.Router("/login", &controllers.LoginController{})
```

2.8.2高级路由设置

一般在开发过程中，我们基本不使用beego提供的默认请求访问方法，都是自定义相应的方法。那我们来看一下如何来自定义请求方法。

自定义请求方法需要用到Router的第三个参数。这个参数是用来给不同的请求指定不同的方法。具体有如下几种情况。

- 一个请求访问一个方法(**也是最常用的**)，请求和方法之间用`:`隔开，不同的请求用`;`隔开:

  ```
  beego.Router("/simple",&SimpleController{},"get:GetFunc;post:PostFunc")
  ```

- 可以多个请求，访问一个方法，请求之间用`,`隔开，请求与方法之间用`:`隔开：

  ```
  beego.Router("/api",&RestController{},"get,post:ApiFunc")
  ```

- 所有的请求访问同一个方法，用`*`号代表所有的请求，和方法之间用`:`隔开：

```
beego.Router("/api/list",&RestController{},"*:ListFood")
```

- 如果同时存在 * 和对应的 HTTP请求，那么优先执行 HTTP请求所对应的方法，例如同时注册了如下所示的路由：

```
beego.Router("/simple",&SimpleController{},"*:AllFunc;post:PostFunc")
```

那么当遇到Post请求的时候，执行PostFunc而不是AllFunc。

> 如果用了自定义方法之后，默认请求将不能访问。

2.9Go操作MySQL数据库（复习）

- 安装go操作MySQL的驱动

```
go get -u github.com/go-sql-driver/mysql
```

- go简单操作MySQL数据库
  - 导包

    ```
    import "github.com/go-sql-driver/mysql"
    ```

  - 连接数据库，用sql.Open()方法,open()方法的第一个参数是驱动名称,第二个参数是**用户名:密码@tcp(ip:port)/数据库名称?编码方式**,返回值是连接对象和错误信息，例如：

    ```
    conn,err := sql.Open("mysql","root:123456@tcp(127.0.0.1:3306)/test?charset=utf8")
    defer conn.Close()//随手关闭数据库是个好习惯
    ```

  - 执行数据库操作，这一步分为两种情况，一种是增删改，一种是查询，因为增删改不返回数据，只返回执行结果，查询要返回数据，所以这两块的操作函数不一样。

    查询操作

    用的函数是Query(),参数是SQL语句，返回值是查询结果集和错误信息，然后循环结果集取出其中的数据。代码如下：

    ```
    data ,err :=conn.Query("SELECT name from user")
        var userName string
        if err == nil{
            for data.Next(){
                data.Scan(&userName)
                beego.Error(userName)
            }
        }
    ```

    增删改操作

执行增删改操作语句的是Exec()，参数是SQL语句，返回值是结果集和错误信息，通过对结果集的判断，得到执行结果的信息。以插入数据为例代码如下：

```
res,_:=stmt.Exec("insert user(name,pwd) values (?,?)","tony","tony")
count,_:=res.RowsAffected()
this.Ctx.WriteString(strconv.Itoa(int(count)))
```

创建表

创建表的方法也是Exec(),参数是SQL语句，返回值是结果集和错误信息：

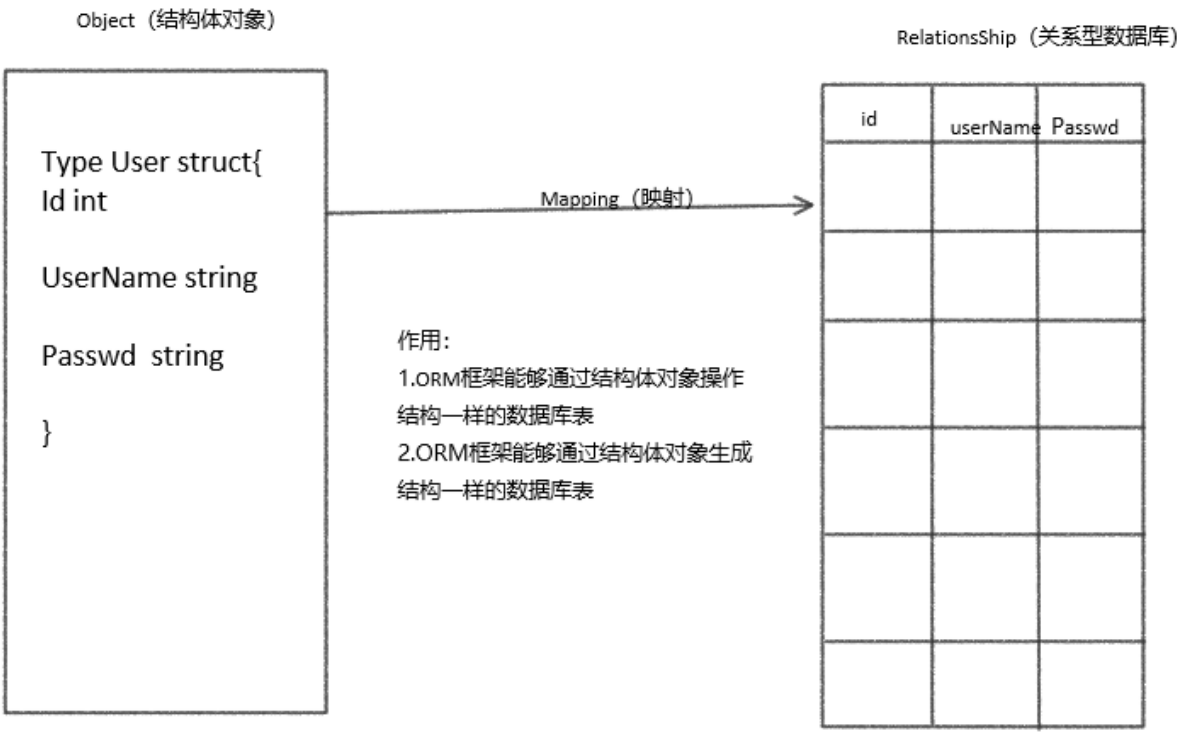
```
res ,err:= conn.Exec("create table user(name VARCHAR(40),pwd VARCHAR(40))")
beego.Info("create table result=",res,err)
```

全部代码

```
//连接数据库
conn,err := sql.Open("mysql","root:123456@tcp(127.0.0.1:3306)/testtest?charset=utf8")
    if err != nil{
        beego.Info("链接失败")
    }
    defer conn.Close()
//建表
    res ,err:= conn.Exec("create table user(userName VARCHAR(40),passwd VARCHAR(40))")
    beego.Info("create table result=",res,err)
//插入数据
    res,err =conn.Exec("insert user(userName,passwd) values(?,?)","itcast","heima")
    beego.Info(res,err)
//查询数据
    data ,err :=conn.Query("SELECT userName from user")
    var userName string
    if err == nil{
        for data.Next(){
            data.Scan(&userName)
            beego.Error(userName)
        }
    }
```

2.10 ORM框架

Beego中内嵌了ORM框架，用来操作数据库。那么ORM框架是什么呢？ORM框架是Object-RelationShip-Mapping的缩写，中文叫关系对象映射，他们之间的关系，我们用图来表示：



2.10.1 ORM初始化

- 首先要导包

```
import "github.com/astaxie/beego/orm"
```

- 然后要定义一个结构体

```
type User struct{
    Id int
    Name string
    Passwd string
}
```

> 思考:如果字段名为小写会发生什么结果?

- 然后像数据库中注册表，这一步又分为三步：
  - 连接数据库

    用RegisterDataBase()函数，第一个参数为数据库别名，也可以理解为数据库的key值，**项目中必须有且只能有一个别名为`default`的连接**，第二个参数是数据库驱动，这里我们用的是MySQL数据库，所以以MySQL驱动为例，第三个参数是连接字符串，和传统操作数据库连接字符串一样，格式为：**用户名:密码@tcp(ip:port)/数据库名称?编码方式**，代码如下：

```
orm.RegisterDataBase("default","mysql","root:123456@tcp(127.0.0.1:3306)/class1?charset=utf8")
```

> 注意：ORM只能操作表，不能操作数据库，所以我们连接的数据库要提前在MySQL终端创建好。

- 注册数据库表

  用orm.RegisterModel()函数，参数是结构体对象，如果有多个表，可以用 `,` 隔开，多new几个对象：

  ```
  orm.RegisterModel(new(User))
  ```

- 生成表

  用orm.RunSyncdb()函数，这个函数有三个参数，

  第一个参数是数据库的别名和连接数据库的第一个参数相对应。

  第二个参数是是否强制更新，一般我们写的都是false，如果写true的话，每次项目编译一次数据库就会被清空一次，fasle的话会在数据库发生重大改变（比如添加字段）的时候更新数据库。

  第三个参数是用来说，生成表过程是否可见，如果我们写成课件，那么生成表的时候执行的SQL语句就会在终端看到。反之看不见。代码如下:

  ```
  orm.RunSyncdb("default",false,true)
  ```

完整代码如下:

```
import "github.com/astaxie/beego/orm"

type User struct {
    Id int
    Name string
    Passwd string
}

func init(){
    //1.连接数据库
    orm.RegisterDataBase("default","mysql","root:123456@tcp(127.0.0.1:3306)/test?charset=utf8")
    //2.注册表
    orm.RegisterModel(new(User))
    //3.生成表
    //1.数据库别名
    //2.是否强制更新
    //3.创建表过程是否可见
    orm.RunSyncdb("default",false,true)
}
```

因为这里我们把ORM初始化的代码放到了 models包的init()函数里面，所以如果我们想让他执行的话就需要在main.go里面加入这么一句代码:

```
import _ "classOne/models"
```

2.10.2 简单的ORM增删改查操作

在执行ORM的操作之前需要先把ORM包导入，但是GoLand会自动帮我们导包，也可以手动导包

```
import "github.com/astaxie/beego/orm"
```

插入

- 先获取一个ORM对象,用orm.NewOrm()即可获得

  ```
  o := orm.NewOrm()
  ```

- 定义一个要插入数据库的结构体对象

  ```
  var user User
  ```

- 给定义的对象赋值

  ```
  user.Name = "itcast"
  user.Passwd = "heima"
  ```

  > 这里不用给Id赋值，因为建表的时候我们没有指定主键，ORM默认会以变量名为Id，类型为int的字段当主键，至于如何指定主键，我们明天详细介绍。

- 执行插入操作，o.Insert()插入，参数是结构体对象，返回值是插入的id和错误信息。

  ```
  id, err := o.Insert(&user)
  if err == nil {
      fmt.Println(id)
  }
  ```

查询

- 也是要先获得一个ORM对象

  ```
  o := orm.NewOrm()
  ```

- 定义一个要获取数据的结构体对象

  ```
  var user User
  ```

- 给结构体对象赋值，相当于给查询条件赋值

```
user.Name = "itcast"
```

- 查询,用o.Read()，第一个参数是对象地址，第二个参数是指定查询字段，返回值只有错误信息。

```
err := o.Read(&user,"Name")
if err != nil{
        beego.Info("查询数据错误",err)
        return
    }
```

> 如果查询字段是查询对象的主键的话，可以不用指定查询字段

更新

- 一样的套路，先获得ORM对象

```
o := orm.NewOrm()
```

- 定义一个要更新的结构体对象

```
var user User
```

- 给结构体对象赋值，相当于给查询条件赋值

```
user.Name = "itcast"
```

- 查询要更新的对象是否存在

```
err := o.Read(&user)
if err != nil{
    beego.Info("查询数据错误",err)
    return
}
```

- 如果查找到了要更新的对象,就给这个对象赋新值

```
user.Passwd = "itheima"
```

- 执行更新操作,用o.Update()函数，参数是结构体对象指针，返回值是更新的条目数和错误信息

```
count,err=o.Update(&user)
if err != nil{
    beego.Info("更新数据错误",err)
    return
}
```

删除

- 同样的，获取ORM对象，获取要删除的对象

```
o := orm.NewOrm()
var user User
```

- 给删除对象赋值，删除的对象主键必须有值，如果主键没值，就查询一下。我们这里直接给主键赋值。

```
user.Id = 1
```

- 执行删除操作，用的方法是o.Delete()，参数是删除的结构体对象,返回值是删除的条目数和错误信息

```
num, err := o.Delete(&User{Id: 1})
if err == nil {
    fmt.Println(num)
}
```

3.当天案例

我们今天以注册和登陆作为我们今天的大练习，把今天讲到的内容都串起来。先注册，然后登陆。

3.1注册

确定注册请求路径,修改路由文件

我们这里以`/register`作为注册的请求路径。所以这里我们需要修改router.go文件的内容。

在router.go文件的init()函数中加下面这行代码:

```
beego.Router("/register", &controllers.UserController{},"get:ShowRegister")
```

根据上面路由的指定，我们需要添加注册控制器

在controllers文件夹下创建一个user.go，然后在这个文件里面定义一个结构体`UserController`当控制器。

```
type UserController struct{
    beego.Controller
}
```

> 注意这里面添加的`beego.Controller`,是为了继承自beego自带的控制器。

添加显示注册页面函数

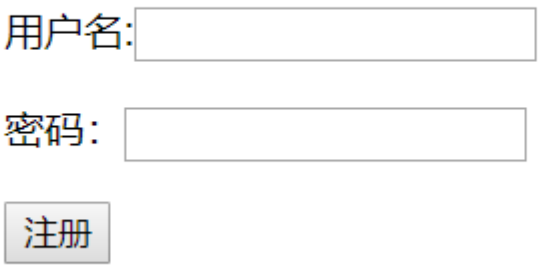
添加函数的时候需要注意，这个函数必须是`UserController`的函数才可以，不然在路由里面调用不到。那如何把函数设置成`UserController`的成员函数呢？是在函数名前面加上括号，然后放上`UserController`的指针。这里我们先指定注册的视图。代码如下：

```
func (this*UserController)ShowRegister(){
    this.TplName = "register.html"
}
```

> 注意：这里如果函数名首字母小写，路由同意找不到函数，所以函数名首字母必须大写

添加视图页面

在views文件夹下面创建一个名字为`register.html`的文件。然后实现成类似界面：



我们做后台的不关注样式，明天直接拿现成的样式来用即可，我们重在实现功能。

form标签里面需要添加两个属性，一个是action,一个是method，action其实就是请求路径，这里处理的还是注册业务，所以我们还用register请求，`action = "/register"`,因为是上传数据，所以我们把method设置为post,即`method="post"`,代码如下:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>注册页面</title>
</head>
<body>

<div style="position:absolute;left:50%; top:50%;">
    <form action="/register" method="post">
        用户名:<input type="text" name="userName">
        <p>   </p>
        密码: <input type="password" name="passwd">
        <p>   </p>
        <input type="submit" value="注册">
    </form>
</div>
</body>
</html>
```

让项目运行起来，然后我们在浏览器里面输入相应的地址就能看见我们的注册页面了。

显示完注册页面之后，接着我们来处理注册的post请求。因为`action="/register", method="post"`,所以我们可以去router.go界面给post请求指定相应的方法。修改如下：

```
beego.Router("/register", &controllers.UserController{},"get:ShowRegister;post:HandleRegister")
```

指定方法名之后我们就需要去控制器中实现他。

注册业务处理

- 首先在user.go中添加这个函数：

  ```
  func (this*UserController)HandleRegister(){
  }
  ```

- 接着开始处理注册业务
  - 首先要获取数据。这里给大家介绍一类方法，这类方法将会在我们项目一中高频率的出现，因为他的作用太强大了。

    this.GetString()：获取字符串类型值

    this.GetInt()：获取整型值

    this.GetFloat：获取浮点型值

    ...

    this.GetFile()：获取上传的文件

    作用：接收前端传递过来的数据，不管是get请求还是post请求，都能接收。

    参数: 是传递数据的key值，一般情况下是form表单中 标签的name属性值

    返回值：根据返回类型不同，返回值也不一样，最常用的GetString()只有一个返回值，如果没有取到值就返回空字符串，其他几个函数会返回一个错误类型。获取的值一般是 标签里面的value属性值。至于比较特殊的，我们用到的时候给大家做介绍。

    知道了获取数据函数，我们就可以获取前端传递过来的数据啦。
  - 获取注册的用户名和密码

    ```
    userName := this.GetString("userName")
    passwd := this.GetString("passwd")
    ```

  - 对数据进行校验

    一般情况下，我们做服务器开发，从前端拿过来数据都要进行一系列的校验，然后才会用数据对数据库进行操作。不做校验的服务器很容易被黑掉。这里我们只做简单的判空校验。

    ```
    if userName == "" || passwd == ""{
        beego.Info("数据数据不完整，请重新输入！")
        this.TplName = "register.html"
        return
    }
    ```

> 思考：如何把那句错误提示传递给视图?

- 把数据插入数据库

  如果数据校验没有问题，那我们就需要把数据插入到数据库中。数据库插入操作前面刚讲过，这里就不一步一步的分开介绍了，代码如下：

```
//获取orm对象
    o := orm.NewOrm()
    //获取要插入的数据对象
    var user models.User
    //给对象赋值
    user.Name = userName
    user.Passwd = passwd
    //把数据插入到数据库
    if _,err := o.Insert(&user);err != nil{
        beego.Info("注册失败，请更换用户名再次注册!")
        this.TplName = "register.html"
        return
    }
```

  因为我们现在还没有其他界面，如果跳转成功就返回一句话`注册成功`,等我们实现了登陆界面之后再实现注册之后跳转登陆界面的操作。

  给浏览器返回一句化的代码如下：

```
this.Ctx.WriteString("注册成功!")
```

- 完整后台代码如下

```
//显示注册页面
func(this*UserController)ShowRegister(){
    this.TplName = "register.html"
}

//处理注册业务
func(this*UserController)HandleRegister(){
    //获取前端传递的数据
    userName := this.GetString("userName")
    passwd := this.GetString("passwd")
    //对数据进行校验
    if userName == "" || passwd == ""{
        beego.Info("数据数据不完整，请重新输入！")
        this.TplName = "register.html"
        return
    }
    //把数据插入到数据库
    //获取orm对象
    o := orm.NewOrm()
    //获取要插入的数据对象
    var user models.User
    //给对象赋值
```

```
    user.Name = userName
    user.Passwd = passwd
    //把数据插入到数据库
    if _,err := o.Insert(&user);err != nil{
        beego.Info("注册失败，请更换用户名再次注册!")
        this.TplName = "register.html"
        return
    }

    //返回提示信息
    this.Ctx.WriteString("注册成功!")
}
```

3.2登陆

登陆和注册业务流程差不多，差别也就体现在一个是对数据的查询一个是数据的插入，讲义里面就不做详细分析，直接贴代码。

路由文件修改

添加下面一行代码：

```
    beego.Router("/login", &controllers.UserController{},"get:ShowLogin;post:HandleLogin")
```

后台代码修改

在控制器中添加展示登录页的函数`ShowLogin`和处理登陆数据的函数`HandleLogin`。完整代码如下：

```
//显示登陆界面
func(this*UserController)ShowLogin(){
    this.TplName = "login.html"
}
//处理登陆业务
func(this*UserController)HandleLogin(){
    //获取前端传递的数据
    userName := this.GetString("userName")
    passwd := this.GetString("passwd")
    //对数据进行校验
    if userName == "" || passwd == ""{
        beego.Info("数据数据不完整，请重新输入! ")
        this.TplName = "login.html"
        return
    }
    //查询数据库，判断用户名和密码是否正确
    //获取orm对象
    o := orm.NewOrm()
    //获取要插入的数据对象
    var user models.User
    //给对象赋值

    user.Name = userName
```

```
    //根据用户名查询
    if err := o.Read(&user,"Name");err != nil{
        beego.Info("用户名错误，请重新输入！ ")
        this.TplName = "login.html"
        return
    }
    if user.Passwd != passwd{
        beego.Info("密码错误，请重新输入！ ")
        this.TplName = "login.html"
        return
    }

    //返回提示信息
    this.Ctx.WriteString("登陆成功!")
}
```

添加视图文件

登陆界面和注册界面很相似，拷贝过来简单修改一下即可，代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>登陆页面</title>
</head>
<body>

<div style="position:absolute;left:50%; top:50%;">
    <form action="/login" method="post">
        用户名:<input type="text" name="userName">
        <p>   </p>
        密码: <input type="password" name="passwd">
        <p>   </p>
        <input type="submit" value="登陆">
    </form>
</div>
</body>
</html>
```

这样我们的登陆和注册就算完成了，但是有一个问题，我们的登陆注册还是各干各的，没有关联起来，我们前面等登陆页面实现完之后，注册成功就跳转到登陆页面。现在我们来实现一下跳转。

3.3页面之间的跳转

beego里面页面跳转的方式有两种，一种是重定向，一种是渲染。

3.3.1重定向

重定向用到的方法是`this.Redirect()`函数，有两个参数，第一个参数是请求路径，第二个参数是http状态码。

请求路径就不说了，就是和超链接一样的路径。

我们重点介绍一下状态码：

状态码一共分为五类：

1xx : 服务端已经接收到了客户端请求，客户端应当继续发送请求 。常见的请求：100

2xx :请求已成功 （已实现）常见的请求：200

3xx :请求的资源转换路径了，请求被跳转。常见的请求：300，302

4xx :客户端请求失败。常见的请求：404

5xx :服务器端错误。常见的请求：500

状态码详解:http://tool.oschina.net/commons?type=5

重定向的工作流程是：

```
1：当服务端向客户端响应 redirect后，并没有提供任何view数据进行渲染，仅仅是告诉浏览器响应为 redirect，以及重定向的目标地址

2：浏览器收到服务端 redirect 过来的响应，会再次发起一个 http 请求

3：由于是浏览器再次发起了一个新的 http 请求，所以浏览器地址栏中的 url 会发生变化

4：浏览中最终得到的页面是最后这个 重定向的url 请求后的页面

5：所以redirect("/register",302) 相当于你在浏览器中手动输入 localhost/register
```

3.3.2渲染

渲染就是控制期把一些数据传递给视图，然后视图用这些输出组织成html界面。所以不会再给浏览器发请求，是服务器自己的行为，所以浏览器的地址栏不会改变，但是显示的页面可能会发生变化。用的函数是:`this.TplName = "login.html"`

4.小结

1.了解MVC结构

2.了解beego初始化之后各个模块的作用

3.了解beego项目的运行流程

4.能够对路由进行简单的设置

5.能够通过ORM对数据库进行简单的增删改查。

6.能够独立实现登陆和注册业务

7.了解重定向和渲染之间的区别。