

内部培训资料

Higo 框架学习指南

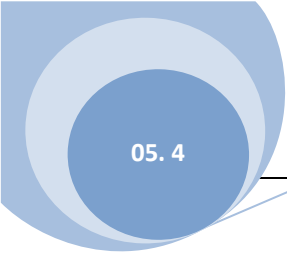
我们自己的 go 语言开发框架

华汇人寿保险股份有限公司 信息技术部

编辑:刘东城

2013/5/4

前 言.....	4
第一章 Go Web 开发之 Higo - 开发入门.....	5
一、 环境准备.....	5
二、 创建我们的第一个 App.....	8
三、 网页请求处理流程.....	9
四、 Hello World.....	15
五、 应用程序打包.....	21
第二章 Go Web 开发之 Higo - 开发简单介绍.....	25
1. 概念.....	25
2. 组织结构.....	28
3. Higo 运行原理	30
4. 路由 (Routing)	30
5. 参数绑定 (Binding Parameters)	33
6. 验证(Validation).....	37
7. 返回值(Result)	42
8. 拦截器 (Interceptors)	48
9. 插件(Plugins).....	50
10. 模块(Module)	51
11. Websockets.....	52
12. 测试(Testing)	53
13. 日志(Logging)	58
14. 部署(Deployment).....	60



15.	app.conf.....	61
16.	命令行工具(Command line)	63

前 言

Higo 是参考 revel 和 grails 框架开发的适用于企业开发的 go 语言 web 开发框架。

本框架重点考虑对 oracle 数据库的支持，实现了数据库连接池访问方式，框架总体设计由王海富先生完成，oracle 的支持部分由胡同召、姜洪波全力打造，软件开发处的同仁们在框架设计中参与了多次的讨论。

此本书只作为入门的指南，对其中的高级编程请在《Higo 框架开发手册》中学习了解。

刘东城

2013 年 5 月 4 日 星期六

第一章 Go Web 开发之 Higo - 开发入门

一、环境准备

我们先介绍一下 higo 的环境配置。

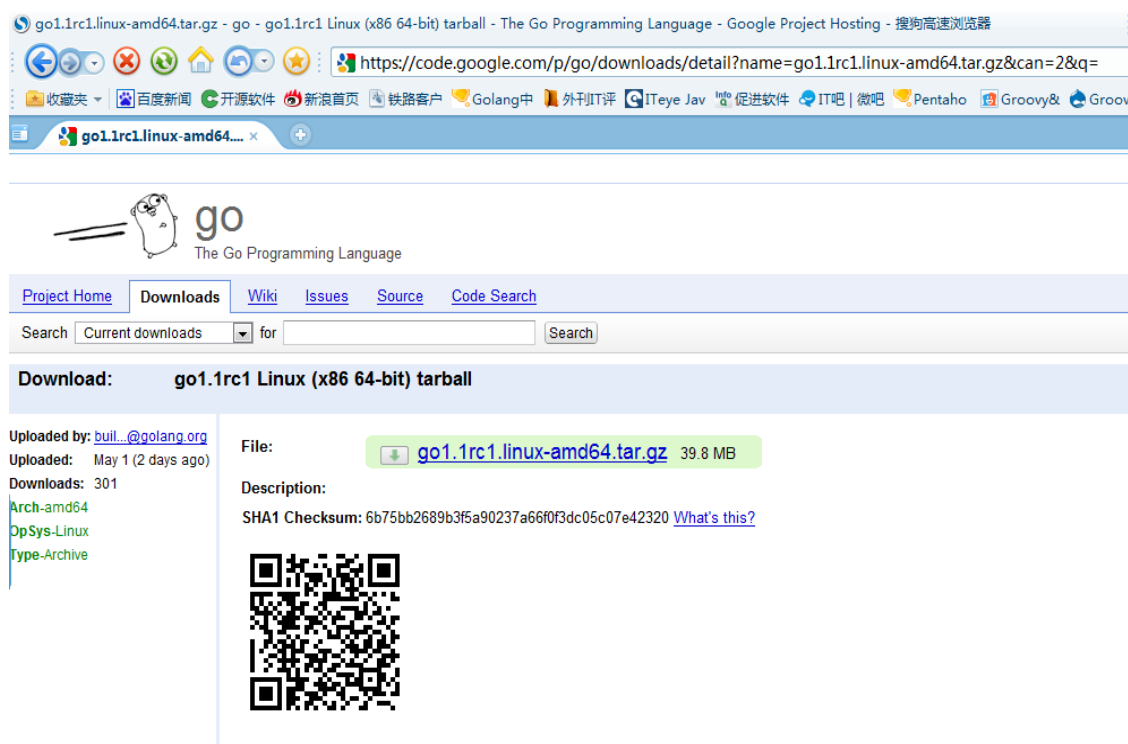
1. 安装 Go 开发环境

建立用户，如 gouser

下载 go 的 linux 64 位版本，我们将用此版本来举例

<http://www.oschina.net/news/40111/go-1-1-rc1>

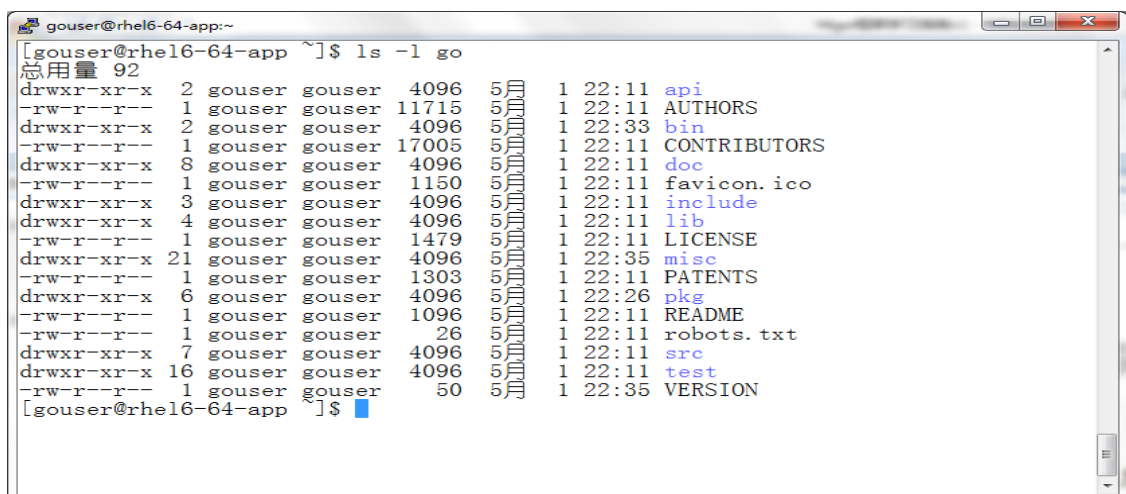
<https://code.google.com/p/go/downloads/detail?name=go1.1rc1.linux-amd64.tar.gz&can=2&q=>



下载后：

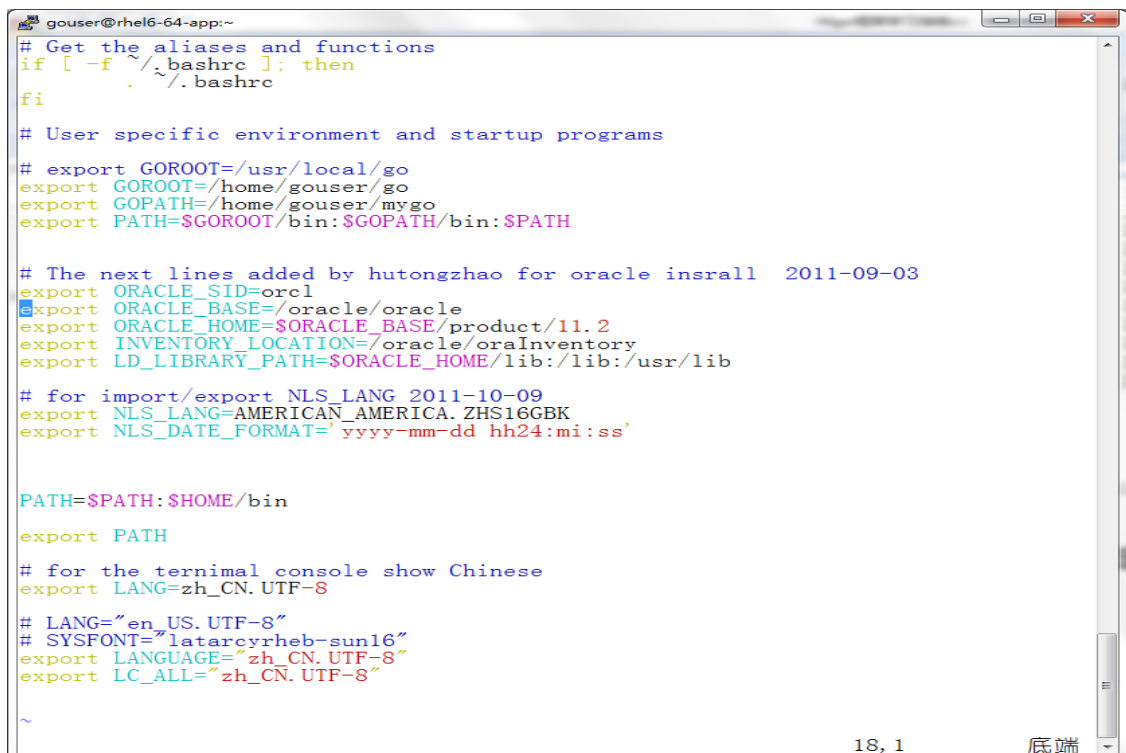
```
[gouser@rhel6-64-app MyTools]$ ls -l go1.1rc1.linux-amd64.tar.gz
-rw-r--r-- 1 gouser gouser 41704433 5月 3 17:44 go1.1rc1.linux-amd64.tar.gz
[gouser@rhel6-64-app MyTools]$
```

用 `tar zxvf go1.1rc1.linux-amd64.tar.gz` 解压到当前用户目录下 生成 `go` 目录，其中包含：



```
gouser@rhel6-64-app:~$ ls -l go
总用量 92
drwxr-xr-x  2 gouser gouser 4096 5月 1 22:11 api
-rw-r--r--  1 gouser gouser 11715 5月 1 22:11 AUTHORS
drwxr-xr-x  2 gouser gouser 4096 5月 1 22:33 bin
-rw-r--r--  1 gouser gouser 17005 5月 1 22:11 CONTRIBUTORS
drwxr-xr-x  8 gouser gouser 4096 5月 1 22:11 doc
-rw-r--r--  1 gouser gouser 1150 5月 1 22:11 favicon.ico
drwxr-xr-x  3 gouser gouser 4096 5月 1 22:11 include
drwxr-xr-x  4 gouser gouser 4096 5月 1 22:11 lib
-rw-r--r--  1 gouser gouser 1479 5月 1 22:11 LICENSE
drwxr-xr-x 21 gouser gouser 4096 5月 1 22:35 misc
-rw-r--r--  1 gouser gouser 1303 5月 1 22:11 PATENTS
drwxr-xr-x  6 gouser gouser 4096 5月 1 22:26 pkg
-rw-r--r--  1 gouser gouser 1096 5月 1 22:11 README
-rw-r--r--  1 gouser gouser 26 5月 1 22:11 robots.txt
drwxr-xr-x  7 gouser gouser 4096 5月 1 22:11 src
drwxr-xr-x 16 gouser gouser 4096 5月 1 22:11 test
-rw-r--r--  1 gouser gouser 50 5月 1 22:35 VERSION
[gouser@rhel6-64-app ~]$
```

在用户目录中的 `.bash_profile` 文件中定义：



```
gouser@rhel6-64-app:~$ cat .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

# export GOROOT=/usr/local/go
export GOROOT=/home/gouser/go
export GOPATH=/home/gouser/mygo
export PATH=$GOROOT/bin:$GOPATH/bin:$PATH

# The next lines added by hutongzhao for oracle insrall 2011-09-03
export ORACLE_SID=orcl
export ORACLE_BASE=/oracle/oracle
export ORACLE_HOME=$ORACLE_BASE/product/11.2
export INVENTORY_LOCATION=/oracle/oraInventory
export LD_LIBRARY_PATH=$ORACLE_HOME/lib:/lib:/usr/lib

# for import/export NLS_LANG 2011-10-09
export NLS_LANG=AMERICAN_AMERICA.ZHS16GBK
export NLS_DATE_FORMAT='yyyy-mm-dd hh24:mi:ss'

PATH=$PATH:$HOME/bin
export PATH

# for the terminal console show Chinese
export LANG=zh_CN.UTF-8

# LANG="en_US.UTF-8"
# SYSFONT="latarcyrheb-sun16"
export LANGUAGE="zh_CN.UTF-8"
export LC_ALL="zh_CN.UTF-8"

~
18, 1 底端
```

重点要素是：

```
export GOROOT=/home/gouser/go
```

```
export GOPATH=/home/gouser/mygo
```

```
export PATH=$GOROOT/bin:$GOPATH/bin:$PATH
```

```
export LANGUAGE="zh_CN.UTF-8"
```

```
export LC_ALL="zh_CN.UTF-8"
```

重新登录系统或运行 `./bash_profile` 使上面的设置生效。

此处设我们的开发项目代码放在 `mygo` 目录下。

2. higo 的安装

在 `mygo` 目录下建立所需目录：

```
$ mkdir mygo
```

```
$ cd mygo
```

```
$ mkdir src pkg bin
```

```
$ cd src
```

```
$ tar xvf higo.tar
```

```
$ go build -o ../bin/Higo higo/cmd
```

3. 现在验证 Higo 是否可以工作了

```
[gouser@rhel6-64-app src]$ pwd
/home/gouser/mygo/src
[gouser@rhel6-64-app src]$ ls -l
总用量 4
drwxr-xr-x 8 gouser gouser 4096 7月 2 18:42 higo
[gouser@rhel6-64-app src]$
```

```
[gouser@rhel6-64-app src]$ Higo
~ Higo! http://huser.github.com/higo
usage: Higo command [arguments]

The commands are:

    new          create a skeleton Higo application
    run          run a Higo application
    build        build a Higo application (e.g. for deployment)
    package      package a Higo application (e.g. for deployment)
    clean        clean a Higo application's temp files
    test         run all tests from the command-line

Use "Higo help [command]" for more information.
[gouser@rhel6-64-app src]$
```

二、 创建我们的第一个 App

Linux 环境下使用下面的 Higo 命令行工具在你的 GOPATH 中创建一个空的项目并运行。

```
[gouser@rhel6-64-app src]$ Higo new liuapp
```

```
~ Higo! http://huser.github.com/higo
```

```
Your application is ready:
/home/gouser/mygo/src/liuapp
```

```
You can run it with:
```

```
Higo run liuapp
```

```
[gouser@rhel6-64-app src]$
```

```
[gouser@rhel6-64-app src]$ Higo run liuapp
```

```
~ Higo! http://huser.github.com/higo
```

```
2013/07/03 15:32:54 revel.go:252: Loaded module static
2013/07/03 15:32:54 revel.go:252: Loaded module testrunner
2013/07/03 15:32:54 run.go:57: Running liuapp (liuapp) in dev mode
2013/07/03 15:32:54 harness.go:143: Listening on :9000
```


用浏览器登录出现界面为：



Your Application Is Ready

祝贺你已成功运行! Higo 开发组

Higo 为参考 `revel` 和 `Grails` 开发的用于企业级开发的 Go 语言框架。
Higo 使用时，请您注意下列的例子中的写法。

2013年5月3日

```
package controllers
```

```
import "higo/web"
```

```
type Application struct {  
    *web.Controller  
}
```

```
func (c Application) Index() web.Result {  
    return c.Render()  
}
```

三、 网页请求处理流程

我们已经创建了 `liuapp` 的应用，现在我们来看看 Higo 是如何处理一个浏览器访问

`http://10.1.101.11:9000` 的请求的。

Routes (路由)

首先 Higo 会检查 `conf/route/` 目录下的文件列表，这些文件包含多个路由，如下是一条：

GET	/	Application.Index
-----	---	-------------------

这个路由信息告诉 higo 当访问 `/` 路径是应该调用 `Application Controller` 的 `Index` 方法

Actions (行为)

下面我们来看一下 Controll 中的 Action，所在路径为 **app/controllers/app.go**

```
[gouser@rhel6-64-app controllers]$ pwd
/home/gouser/mygo/src/liuapp/app/controllers
[gouser@rhel6-64-app controllers]$ more app.go
package controllers

import "higo/web"

type Application struct {
    *web.Controller
}

func (c Application) Index() web.Result {
    return c.Render()
}
[gouser@rhel6-64-app controllers]$
```

所有的 controller 必须在 struct 类型里面嵌入 web.Controller 或 *web.Controller，在 Controller 中任何 Action 的返回值都是 web.Result。web Controller 提供了很多有用的方法来生成 Result，在上面的代码中它调用了 Render 方法来生成 Result，这个方法告诉 Higo 查找和渲染一个模板作为输出结果。

Templates (模板)

全部的模板都存放在 **app/views** 目录下，当一个模板的名字没有被显式声明的时候，web 会查找匹配 action 的名字，按照上面的代码 web 会找到 **app/views/Application/index.html** 这个文件同时把它作为一个 Go 模板 render 输出。

[illegible]

上面的函数是有 Go 模板提供的，Higo 也添加了一些自己辅助方法。

这个模板的意思如下

- 1.为 render 的上下文添加一个 title 变量
- 2.包含 header.html 模板文件
- 3.显示欢迎信息等

4.包含 footer.html

如果你看一下 header.html 文件，你会发现更多的模板标签

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{.title}}</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" type="text/css" media="screen" href="/public/stylesheets/main.css"
">
    <link rel="shortcut icon" type="image/png" href="/public/images/favicon.png">
    <script src="/public/javascripts/jquery-1.5.2.min.js" type="text/javascript" charset="utf-8">
</script>
    {{range .moreStyles}}
      <link rel="stylesheet" type="text/css" href="/public/{{.}}">
    {{end}}
    {{range .moreScripts}}
      <script src="/public/{{.}}" type="text/javascript" charset="utf-8"></script>
    {{end}}
  </head>
  <body>
```

你可以看到里面有用到之前定义 title 还可以通过调用模板方法包含更多的 JS 和 CSS 文件，主要是通过 **moreStyles** 和 **moreScripts** 这两个变量添加进去的。

Hot-reload (热重载)

Higo 支持热重载，修改一下 Index.html 文件的内容，马上就可在浏览器看到结果。

Higo 会监视如下内容（利用开源项目（[fsnotify](#)）实现的文件监控）。

- app 目录下面的所有代码
- app/views 下面的全部模板文件
- conf/route 下面的全部路由文件

改变任何上面所监控的文件 Higo 都会用最新代码更新你的应用,现在试试改变一下 **ap**

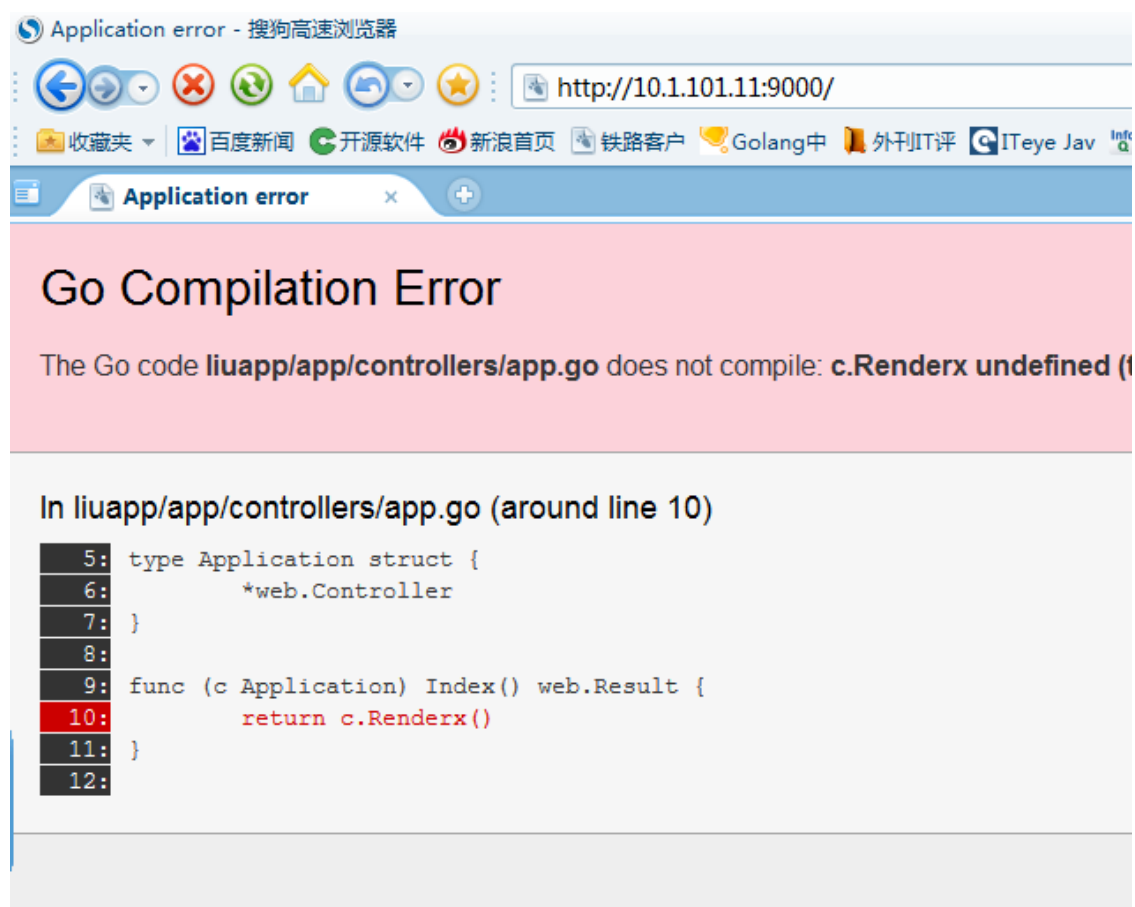
p/controllers/app.go 把

```
return c.Render()
```

改成

```
return c.Renderx()
```

刷新页面将会产生一个错误提示信息



最后我们来试试给模板传递一些参数

```
return c.Render()
```

改为

```
func (c Application) Index() web.Result {
    greeting:="欢迎您学习 Higo!"
```

```
    return c.Render(greeting)
}
```

修改一下模板文件 **app/views/Application/Index.html**

```
<h1>Your Application Is Ready</h1>
<h2>祝贺你已成功运行! Higo 开发组</h2>
```

改为

```
<h1>Your Application Is Ready</h1>
<h1>{{.greeting}}</h1>
<h2>祝贺你已成功运行! Higo 开发组</h2>
```

刷新浏览器将看到如下所示表示修改成功。



Your Application Is Ready

欢迎您学习 Higo!

祝贺你已成功运行! Higo 开发组

Higo 为参考 **revel** 和 **Grails** 开发的用于企业级开发的 Go 语言框架。
Higo 使用时, 请您注意下列的例子中的写法。

2013年5月3日

```
package controllers
```

```
import "higo/web"
```

```
type Application struct {  
    *web.Controller  
}
```

```
func (c Application) Index() web.Result {  
    return c.Render()  
}
```

四、 Hello World

下面结合之前创建的 liuapp 做一个提交表单的 demo

首先编辑 **app/views/Application/Index.html** 模板文件添加一下 form 表单

```
<h1>Your Application Is Ready</h1>
<h1>{{.greeting}}</h1>
<h2>祝贺你已成功运行! Higo 开发组</h2>

<form action="/Application/Hello" method="GET">
    <input type="text" name="myName" />
    <input type="submit" value="提交" />
</form>
```

```
<form action="/Application/Hello" method="GET">
    <input type="text" name="myName" />
    <input type="submit" value="提交" />
</form>
```

刷新表单



Your Application Is Ready

欢迎您学习 Higo!

祝贺你已成功运行! Higo 开发组

Higo 为参考 **revel** 和 **Grails** 开发的用于企业级开发的 Go 语言框架。
Higo 使用时, 请您注意下列的例子中的写法。

2013年5月3日

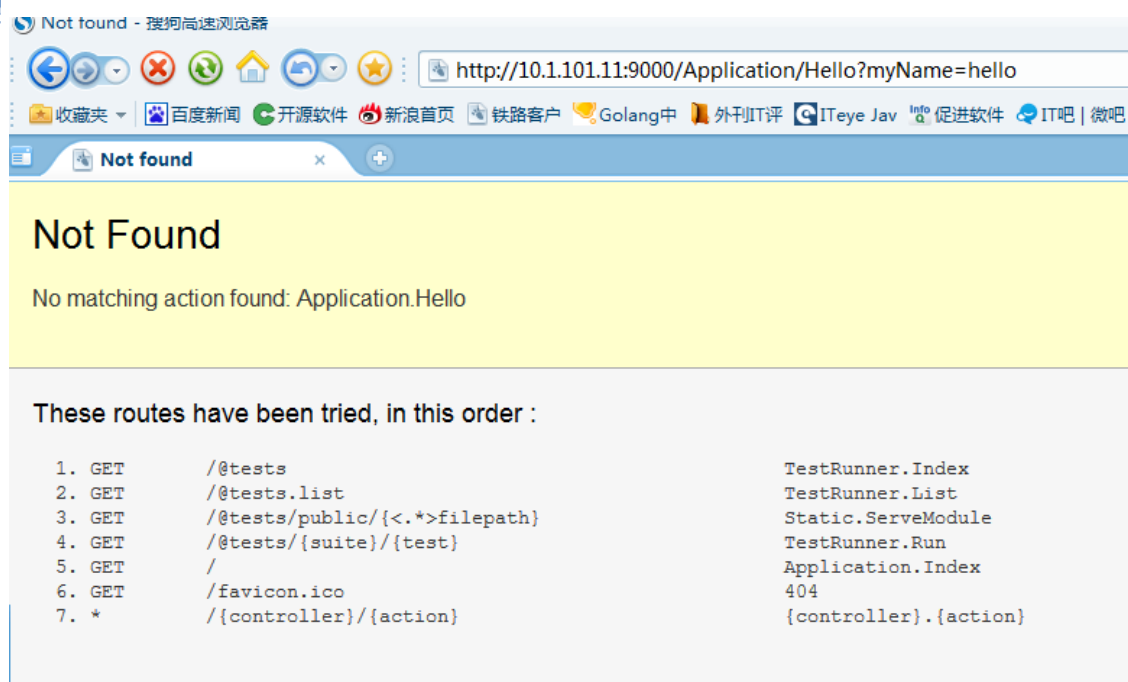
```
package controllers

import "higo/web"

type Application struct {
    *web.Controller
}

func (c Application) Index() web.Result {
    return c.Render()
}
```

我们提交一下表单



报错：没错！！!---没有找到匹配的 action，在 **app/controllers/app.go** 文件中处理。

下面我们来添加一个

```
func (c Application) Hello(myName string) web.Result {
    return c.Render(myName)
}
```

如下图：

```
package controllers

import "higo/web"

type Application struct {
    *web.Controller
}

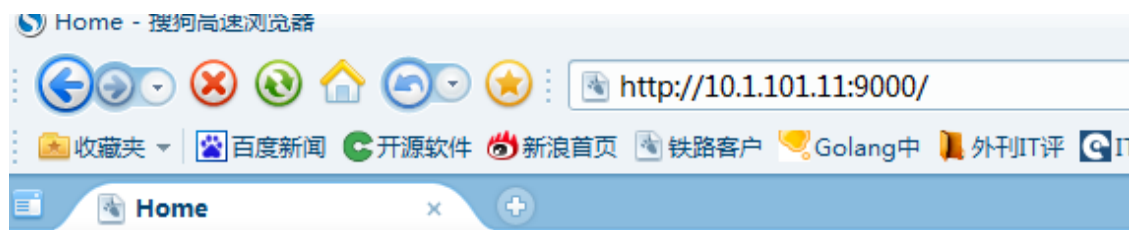
func (c Application) Index() web.Result {
    greeting := "欢迎您学习 Higo!"
    return c.Render(greeting)
}

func (c Application) Hello(myName string) web.Result {
    return c.Render(myName)
}
```

接着我们创建视图，路径为：**app/views/Application/Hello.html**，内容如下：

```
{{set . "title" "Home"}}
{{template "header.html" .}}
<h1>你好, {{.myName}} </h1>
<a href="/">返回前页</a>
{{template "footer.html" .}}
```

注意文件名大写要与 action 匹配。



Your Application Is Ready

欢迎您学习 Higo!

祝贺你已成功运行！Higo 开发组

Higo

Higo 为参考 **revel** 和 **Grails** 开发的用于企业级开发的 Go 语言框架。
Higo 使用时，请您注意下列的例子中的写法。

2013年5月3日

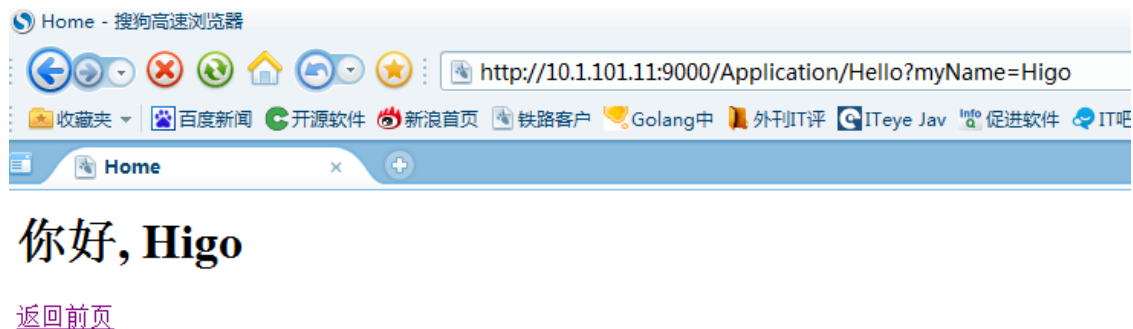
```
package controllers

import "higo/web"

type Application struct {
    *web.Controller
}

func (c Application) Index() web.Result {
    return c.Render()
}
```

在文本框中填入 Higo 的内容然后提交表单，结果如下：



最后我们添加一些验证，把文本框中的内容要求为必填并且最少三个字符，我们来编辑

你的 action **app/controllers/app.go** 文件

```
//添加 hello 的 action
func (c Application) Hello(myName string) web.Result {
    //添加输入校验
    c.Validation.Required(myName).Message("姓名是必填项，不要偷懒.")
    c.Validation.MinSize(myName,3).Message("姓名至少 3 个字符的")
    if c.Validation.HasErrors(){ //注意是 Errors
        c.Validation.Keep()
        c.FlashParams()
    }
    return c.Render(Application.Index)
}
```

修改 **app/views/Application/Index.html** 模板文件

```
{{set . "title" "Home"}}
{{template "header.html" .}}
```

```

<h1>{{.greeting}}</h1>
<form action="/Application/Hello" method="GET">
    <input type="text" name="myName" value="{{.flash.myName}}" />
    <input type="submit" value="提交" />
</form>
{{template "footer.html" .}}
{{range .errors}}
<p style="color:#c00">
    {{.Message}}
</p>
{{end}}

```

出错提示信息书写的位置决定输出的位置，要注意。

现在回到 Index 页面，如果填写内容不符合要求将提示错误信息如下所示：

Your Application Is Ready

欢迎您学习 Higo!

祝贺你已成功运行！Higo 开发组

Higo 为参考 [revel](#) 和 [Grails](#) 开发的用于企业级开发的 Go 语言框架。
Higo 使用时，请您注意下列的例子中的写法。
2013年5月3日

```

package controllers

import "higo/web"

type Application struct {
    *web.Controller
}

func (c Application) Index() web.Result {
    return c.Render()
}

```

姓名是必填项，不要偷懒。

Your Application Is Ready

欢迎您学习 Higo!

祝贺你已成功运行！Higo 开发组

Higo 为参考 [revel](#) 和 [Grails](#) 开发的用于企业级开发的 Go 语言框架。
Higo 使用时，请您注意下列的例子中的写法。

2013年5月3日

```
package controllers

import "higo/web"

type Application struct {
    *web.Controller
}

func (c Application) Index() web.Result {
    return c.Render()
}
```

姓名至少3个字符的

五、应用程序打包

程序开发好之后，就可以打包并部署到其他机器上了。

键命令 Higo 就可以看到使用说明了。

```
[gouser@rhel6-64-app src]$ Higo
~ Higo! http://huser.github.com/higo
```

```
usage: Higo command [arguments]
```

The commands are:

new	create a skeleton Higo application
run	run a Higo application
build	build a Higo application (e.g. for deployment)
package	package a Higo application (e.g. for deployment)
clean	clean a Higo application's temp files
test	run all tests from the command-line

Use "Higo help [command]" for more information.

```
[gouser@rhel6-64-app src]$ Higo package liuapp
```

```
~ Higo! http://huser.github.com/higo
```

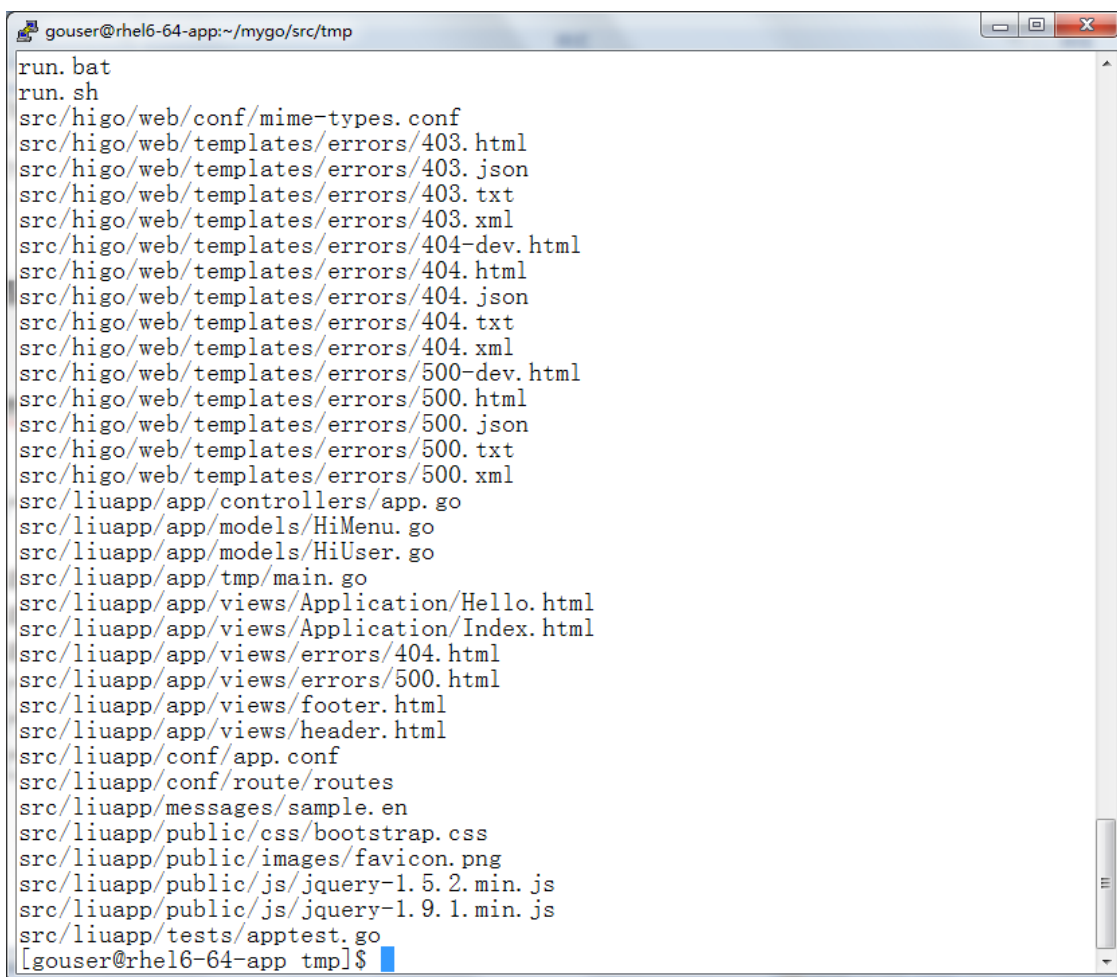
```
2013/07/03 16:24:38 revel.go:252: Loaded module static
```

```
2013/07/03 16:24:38 build.go:90: Exec: [/home/gouser/go/bin/go build -tags -o /
home/gouser/mygo/bin/liuapp liuapp/app/tmp]
Your archive is ready: liuapp.tar.gz
[gouser@rhel6-64-app src]$
```

生成了 liuapp.tar.gz 包。

我们测试一下，建立 tmp 目录并将此包打开。

```
[gouser@rhel6-64-app src]$ ls -l
总用量 2188
drwxr-xr-x 8 gouser gouser 4096 7月 2 18:42 higo
drwxrwxr-x 7 gouser gouser 4096 7月 3 15:32 liuapp
-rw-rw-r-- 1 gouser gouser 2231986 7月 3 16:24 liuapp.tar.gz
[gouser@rhel6-64-app src]$ mkdir tmp
[gouser@rhel6-64-app src]$ cd tmp
[gouser@rhel6-64-app tmp]$ tar zxvf ../liuapp.tar.gz
```

A terminal window titled 'gouser@rhel6-64-app:~/mygo/src/tmp' displays a list of files and directories. The files include configuration files, error templates, application controllers, models, views, and static assets. The prompt at the bottom is '[gouser@rhel6-64-app tmp]\$' with a blue cursor.

```
run.bat
run.sh
src/higo/web/conf/mime-types.conf
src/higo/web/templates/errors/403.html
src/higo/web/templates/errors/403.json
src/higo/web/templates/errors/403.txt
src/higo/web/templates/errors/403.xml
src/higo/web/templates/errors/404-dev.html
src/higo/web/templates/errors/404.html
src/higo/web/templates/errors/404.json
src/higo/web/templates/errors/404.txt
src/higo/web/templates/errors/404.xml
src/higo/web/templates/errors/500-dev.html
src/higo/web/templates/errors/500.html
src/higo/web/templates/errors/500.json
src/higo/web/templates/errors/500.txt
src/higo/web/templates/errors/500.xml
src/liuapp/app/controllers/app.go
src/liuapp/app/models/HiMenu.go
src/liuapp/app/models/HiUser.go
src/liuapp/app/tmp/main.go
src/liuapp/app/views/Application/Hello.html
src/liuapp/app/views/Application/Index.html
src/liuapp/app/views/errors/404.html
src/liuapp/app/views/errors/500.html
src/liuapp/app/views/footer.html
src/liuapp/app/views/header.html
src/liuapp/conf/app.conf
src/liuapp/conf/route/routes
src/liuapp/messages/sample.en
src/liuapp/public/css/bootstrap.css
src/liuapp/public/images/favicon.png
src/liuapp/public/js/jquery-1.5.2.min.js
src/liuapp/public/js/jquery-1.9.1.min.js
src/liuapp/tests/apptest.go
[gouser@rhel6-64-app tmp]$
```

```
[gouser@rhel6-64-app tmp]$ sh run.sh
Listening on port 9000...
```

看看浏览器上的反应：



Your Application Is Ready

欢迎您学习 Higo!

祝贺你已成功运行！Higo 开发组

Higo 为参考 **revel** 和 **Grails** 开发的用于企业级开发的 Go 语言框架。
Higo 使用时，请您注意下列的例子中的写法。

2013年5月3日

```
package controllers

import "higo/web"

type Application struct {
    *web.Controller
}

func (c Application) Index() web.Result {
    return c.Render()
}
```


第二章 Go Web 开发之 Higo - 开发简单介绍

1. 概念

App 下的目录结构：

```
[gouser@rhel6-64-app app]$ pwd
/home/gouser/mygo/src/liuapp/app
[gouser@rhel6-64-app app]$ ls -l
总用量 20
drwxrwxr-x 2 gouser gouser 4096 7月 3 16:16 controllers
drwxrwxr-x 2 gouser gouser 4096 7月 3 15:32 models
drwxrwxr-x 2 gouser gouser 4096 7月 3 15:32 services
drwxrwxr-x 2 gouser gouser 4096 7月 3 16:24 tmp
drwxrwxr-x 4 gouser gouser 4096 7月 3 15:32 views
[gouser@rhel6-64-app app]$
```

MVC 结构

Controller：处理请求的执行，他们执行用户期待的 Action，他们决定哪个视图将被用于显示，他们还视图准备和提供必要的用于渲染视图。

Model：用于描述你的应用程序域的基本数据对象，Model 也包含特定领域的逻辑为了查询和更新数据

Services：用于处理应用中的服务。

View：描述怎样展示和操作数据

每个请求产生一个 Goroutine

Higo 构建于 Go HTTP server 之上，它为每一个进来的请求创建一个 go-routine（轻量级线程），这意味着你的代码可以自由的阻塞，但必须处理并发请求处理。

Controllers and Actions

每一个 HTTP 请求调用一个 action , 它自己处理请求或调用服务处理请求并输出响应内容 , 相关联的 action 被分组到 controller 中。

一个 controller 是任意嵌入 web.Controller 的类型 (直接或间接)

典型的 Controller :

```
type AppController struct {
    *web.Controller
}
```

(web.Controller 必须作为这个 struct 的第一个类型被嵌入)

web.Controller 是请求的上下文 , 它包含 request 和 response 的数据。



```
type Controller struct {
    Name      string
    Type      *ControllerType
    MethodType *MethodType

    Request *Request
    Response *Response

    Flash    Flash           // User cookie, cleared after each request.
    Session  Session           // Session, stored in cookie, signed.
    Params   Params            // Parameters from URL and form (including multipart).
    Args     map[string]interface{} // Per-request scratch space.
    RenderArgs map[string]interface{} // Args passed to the template.
    Validation *Validation        // Data validation helpers
    Txn       *sql.Tx             // Nil by default, but may be used by the app / plugins
}
```

// Flash represents a cookie that gets overwritten on each request.

// It allows data to be stored across one page at a time.

// This is commonly used to implement success or error messages.

// e.g. the Post/Redirect/Get pattern: <http://en.wikipedia.org/wiki/Post/Redirect/Get>

```
type Flash struct {
    Data, Out map[string]string
}
```

// These provide a unified view of the request params.

// Includes:

// - URL query string

```
// - Form values
// - File uploads
type Params struct {
    url.Values
    Files map[string][]*multipart.FileHeader
}

// A signed cookie (and thus limited to 4kb in size).
// Restriction: Keys may not have a colon in them.
type Session map[string]string

type Request struct {
    *http.Request
    ContentType string
}

type Response struct {
    Status      int
    ContentType string
    Headers     http.Header
    Cookies     []*http.Cookie

    Out http.ResponseWriter
}
```



作为处理 HTTP 请求的一部分，Higo 实例化一个你 Controller 的实例，它设置全部的属性

在 web.Controller 上面，因此 Higo 不在请求之间共享 Controller 实例。

Action 是 Controller 里面任意一个符合下面要求的方法：

被导出的

返回一个 web.Result

实例如下：

```
func (c AppController) ShowLogin(username string) web.Result {
    ..
    return c.Render(username)
}
```

这个例子调用 web.Controller.Render 来执行一个模板，将 username 作为参数传递，

Controller 中有许多方法产生 web.Result，但是应用程序也是自由的创建他们自己的

Controller

Results

一个结果是任意符合接口的东西

```
type Result interface {
    Apply(req *Request, resp *Response)
}
```

没用任何东西被写入 response,直到 action 返回一个 Result,此时 Higo 输出 header 和 cookie,然后调用 Result.Apply 写入真正的输出内容.

(action 可以选择直接输出内容,但是这只用于特殊情况下,在那些情况下它将必须自己处理保存 Session 和 Flash 数据).

2. 组织结构

Higo 需要它自身和用户应用程序被安装到 GOPATH 下面.

实例目录结构

mygo	GOPATH 根目录
src	GOPATH src 目录
higo	higo 源代码
...	
sample	用户应用程序根目录
app	App 源
controllers	App controllers
models	App 域模型
services	App 服务
views	模板
tests	测试工具
conf	配置文件
app.conf	主配置文件
route	路由定义目录
public	公共资源文件
css	CSS 文件
js	Javascript 文件



images

Image 文件

app/ 目录

app 目录包含源代码和模板文件

app/controllers

app/models

app/services

app/views

Higo 需要:

全部的模板文件在 app/views 下

全部的 controller 在 app/controllers 下

除了上面的要求应用程序可以任意的组织,Higo 将监控 app 下全部的目录,当发现文件改变时重新编译

应用程序,任何超出 app/目录的更改都将不被监控--开发人员只能自己手动编译.

public 目录

资源文件和静态文件都存放在 public 目录下通过 Web server 提供服务,它们被分在了 3 个目录

images、css 和 javascript. 3 个目录名是任意的,开发人员只需修改路由即可。

conf 目录

conf 目录包括应用程序的配置文件.这里有两个主要的配置文件:

app.conf 这个主要的配置文件包括了标准的配置参数

route 此目录下存放多个路由定义文件

特别说明,每次调用 Higo new 生成新项目时,调用如下目录中的骨架程序,你自己可以修改。

```
[gouser@rhel6-64-app skeleton]$ pwd
/home/gouser/mygo/src/higo/web/skeleton
[gouser@rhel6-64-app skeleton]$ ls -l
总用量 20
drwxr-xr-x 6 gouser gouser 4096 7月 3 11:11 app
drwxr-xr-x 3 gouser gouser 4096 7月 3 10:39 conf
drwxr-xr-x 2 gouser gouser 4096 6月 1 17:48 messages
drwxr-xr-x 5 gouser gouser 4096 6月 1 17:48 public
drwxr-xr-x 2 gouser gouser 4096 7月 3 10:15 tests
[gouser@rhel6-64-app skeleton]$
```

3. Higo 运行原理

- 命令行工具运行 harness,harness 作为一个反向代理运行
- Higo 监听 9000 端口和 app 的文件更改
- Higo 转发请求到运行中的 Server,如果 server 没有运行或者源代码被改变了,在最新的请求中 Higo 将重建应用程序.
- 如果 Higo 需要重建应用程序,harness 将分析源代码并生成 app/tmp/main.go 文件,这个文件包括全部的必要的元信息以支持能够运行在真正的 app server 上
- Higo 使用 go build 编译应用程序,如果有一个编译错误,它将显示错误页帮助用户发现错误
- 如果应用程序编译成功,当 Higo 检查到 app server 已经完成启动,它将运行应用程序并转发请求

4. 路由 (Routing)

路由使用以下语法定义,定义被保存在 route 目录下的文件中.

基础语法:

```
(METHOD) (URL Pattern) (Controller.Action)
```

下面这个例子展示了所有的使用方法

```
# conf/route/routes

# This file defines all application routes (Higher priority routes first)

GET    /login                Application.Login      <b># 一个简单的路径 </b>
GET    /hotels/?             Hotels.Index           <b># 匹配 /hotels 和 /hotels/ (可选的尾斜线) </b>
GET    /hotels/{id}          Hotels.Show            <b># 提取一个 URI 参数 (匹配 /^[^/]+)/ </b>
POST   /hotels/{<[0-9]+>id}  Hotels.Save            <b># 自定义正则 URI </b>
WS     /hotels/{id}/feed     Hotels.Feed            <b># WebSockets. </b>
POST   /hotels/{id}/{action} Hotels.{action}         <b># 自动路由一些 actions. </b>
GET    /public/              staticDir:public       <b># 映射 /app/public 资源在 /public/ 目录下 </b>
*      /{controller}/{action} {controller}.{action} <b># 捕获全部;自动 URL 生成 </b>
```

一个简单的路径

```
GET    /login                Application.Login
```

最简单路由使用一个完全匹配方法和路径.它将调用 Application Controller 的 Login 方法.

可选尾斜线

```
GET    /hotels/?             Hotels.Index
```

问号被视为一个正则表达式:他们允许路径匹配使用或不使用前面的字符,当路径是/hotels 或 /hotels 时,这个路由调用 Hotels.Index 方法

URL 参数

```
GET    /hotels/{id}          Hotels.Show
```

路径的片段可以被匹配和提取.默认情况下,{id}将匹配除了/外的任意字符.在这个案例中,/hotels/123 和/hotels/abc 都将被路由匹配.

可以在 `Controller.Params`(map 类型)中提取参数,也可以通过 `action` 方法参数来提取,如下所示:

```
func (c Hotels) Show(id int) web.Result {  
    ...  
}
```

或

```
func (c Hotels) Show() web.Result {  
    var id string = c.Params.Get("id")  
    ...  
}
```

或

```
func (c Hotels) Show() web.Result {  
    var id int = c.Params.Bind("id", reflect.TypeOf(0))  
    ...  
}
```

自定义正则表达式的 URL 参数

```
POST    /hotels/{<[0-9]+>id}    Hotels.Save
```

路由也可以定义为正则表达式并用他们的参数来限定他们将匹配什么,在这个例子中我们限定了 Hotel ID 为数字。

WebSockets

```
WS      /hotels/{id}/feed      Hotels.Feed
```

Websockets 也是已相同的方式路由,只是使用一个 `WS` 的标示符,对应下面的匹配方法

```
func (c Hotels) Feed(ws *websocket.Conn, id int) web.Result {  
    ...  
}
```


Static 服务

```
GET    /public/          staticDir:public
```

为了服务静态资源文件 ,Higo 提供了 StaticDir 指令.这个路由告诉 Higo 使用 [http.ServeFile](#) 来处理前缀是/public/的路径的请求 , 相应的静态文件对应在 public 目录。

自动路由

```
POST   /hotels/{id}/{action} Hotels.{action}
*      /{controller}/{action} {controller}.{action}
```

Url 参数提取也能被应用于决定哪个 action 将被调用.匹配 controller 和 action 是大小写不敏感的.

第一条的路由将影响下面的路由

```
/hotels/1/show    => Hotels.Show
/hotels/2/details => Hotels.Details
```

相似的 , 第二条路由可以用于匹配任意的 action

```
/application/login => Application.Login
/users/list         => Users.List
```

因为匹配 controller 和 action 是大小写不敏感的 , 下面的路由也将正常工作

```
/APPLICATION/LOGIN => Application.Login
/Users/List         => Users.List
```

使用自动路由作为捕获所有的请求 , 对于没有定义的路由很有帮助.

5. 参数绑定 (Binding Parameters)

Higo 尝试尽可能简单的转换参数到 Go 的类型.这个转换从 string 到另一种类型被称为数据绑定.

参数

全部的请求参数被收集到一个 Params 对象中.它包括如下:

URL 路径参数

URL 查询参数

表单值 (Multipart or not)

上传文件

定义如下

```
type Params struct {  
    url.Values  
    Files map[string][]*multipart.FileHeader  
}
```

这个嵌入的 url.Values 提供了访问简单值的方式,但是开发人员将使用更容易的 Higo 数据绑定机制来查找非字符串值.

Action 参数

参数可以直接作为方法的参数被接收,如下:

```
func (c ApplicationController) Action(name string, ids []int, user User, img []byte) web.Result {  
    ...  
}
```

在调用 action 前,Higo 要求它的绑定器来对这些名字的参数进行类型转换,如果由于任何原因造成绑定失败,参数将有一个默认值.

绑定器

绑定参数到数据类型,使用 Higo 的绑定器,它整合参数对象如下:

```
func (c SomeController) Action() web.Result {  
    var ids []int = c.Params.Bind("ids[]", reflect.TypeOf([]int{}))  
    ...  
}
```

下面的数据类型是被支持的:

Ints of all widths

Bools

Pointers to any supported type

Slices of any supported type

Structs

time.Time for dates and times

*os.File, []byte, io.Reader, io.ReadSeeker for file uploads

下面的部分将描述这些类型的语法,你也可以参考[源代码](#)

Booleans

字符串的值"true","on"和"1"都被视为 **true**, 其他的绑定值为 **false**.

Slices

绑定 Slices 有两个被支持的语法:排序和未排序.

排序:

```
?ids[0]=1
&ids[1]=2
&ids[3]=4
```

slice 中的结果为: []int{1, 2, 0, 4}

未排序:

```
?ids[]=1
&ids[]=2
&ids[]=3
```

slice 中的结果为: []int{1, 2, 3}

注意:绑定一个 slice 结构时,只有排序的 slice 应该被使用.

```
?user[0].Id=1
&user[0].Name=rob
&user[1].Id=2
&user[1].Name=jenny
```

Structs

结构使用简单的点符号绑定:

```
?user.Id=1
&user.Name=rob
&user.Friends[]=2
&user.Friends[]=3
&user.Father.Id=5
&user.Father.Name=Hermes
```

将绑定下面的结构定义:

```
type User struct {
    Id int
    Name string
    Friends []int
    Father User
}
```

注意:为了绑定,属性必须被导出.

Date / Time

内建的 SQL 标准时间格式是 "2006-01-02" , "2006-01-02 15:04"

应用程序可以添加更多的,使用[官方的模式](#).简单的添加模式来识别 TimeFormats 变量,如下:

```
func init() {
    web.TimeFormats = append(web.TimeFormats, "01/02/2006")
}
```

文件上传

文件上传可以被绑定到下面的类型:

os.File

[]byte

io.Reader

io.ReadSeeker

这是一个有 [Go multipart 包](#)提供的包装器,bytes 存放在内存中除非他们超过了一个阈值(默认 10MB), 在这种情况下它们被写入临时文件.

注意:绑定一个文件上传到 `os.File` 需要 Higo 把它写入到一个临时文件, 它的效率要低于其他的类型.

自定义绑定器

应用程序可以定义它自己的绑定器来利用这个框架.

它只需要实现绑定器接口和注册这个类型它将在哪里被调用:

```
func myBinder(params Params, name string, typ reflect.Type) reflect.Value {  
    ...  
}  
  
func init() {  
    web.TypeBinders[reflect.TypeOf(MyType{})] = myBinder  
}
```

6. 验证(Validation)

Higo 提供内建的函数来验证参数.这里有一对部件:

一个验证上下文收集器和消息验证错误(keys 和消息)

帮助函数检查数据并把错误信息加入上下文

一个模板函数从验证上下的 key 获得错误信息

内联错误信息

这个示例演示用内联错误信息验证字段.

```
func (c MyApp) SaveUser(username string) web.Result {  
    // Username (required) must be between 4 and 15 letters (inclusive).  
    c.Validation.Required(username)  
    c.Validation.MaxSize(username, 15)
```

```

c.Validation.MinSize(username, 4)
c.Validation.Match(username, regexp.MustCompile("^\\w*$"))

if c.Validation.HasErrors() {
    // Store the validation errors in the flash context and redirect.
    c.Validation.Keep()
    c.FlashParams()
    return c.Redirect(Hotels.Settings)
}

// All the data checked out!
...
}

```

分开说明如下:

在 username 上评估 4 个不同的条件(需要,最大值,最小值,匹配).

每一个评估返回一个 ValidationResult 失败的 ValidationResults 存放在 Validation 上下文.

作为构建应用程序的一部分.Higo 记录被验证的变量的名字并用它作为 Validation 上下文的

缺省 key (为了之后查询)

Validation.HasErrors() 如果上下文是非空的返回 true

Validation.Keep() 告诉 Higo 序列化 ValidationErrors 到 Flash cookie.

Higo 返回一个转向到 Hotels.Settings action

Hotels.Settings aciton 渲染一个模板:

```

{{/* app/views/Hotels/Settings.html */}}
...
{{if .errors}}Please fix errors marked below!{{end}}
...
<p class="{{if .errors.username}}error{{end}}">
    Username:
    <input name="username" value="{{.flash.username}}"/>
    <span class="error">{{.errors.username.Message}}</span>
</p>

```

它做了 3 件事:

检查 errors 字典的 username 看看是否这个字段有一个错误

用 flash 的参数值预填充 input

在字段旁边显示错误提示信息(我们没有规定任何错误信息,但是每一个验证函数提供了一个默认的)

注意:字段模板帮助函数让使用验证错误框架写模板更方便一些.

顶部错误信息

如果错误信息被收集到一个地方,模板可以被简化(例如, 一个大的红色红框在页面的最上面)

这里和之前的示例只有两个不同

我们在 ValidationErrors 上规定一个消息来代替 key

我们在表单的最上方打印全部的错误消息

下面是代码:

```
func (c MyApp) SaveUser(username string) web.Result {
    // Username (required) must be between 4 and 15 letters (inclusive).
    c.Validation.Required(username).Message("Please enter a username")
    c.Validation.MaxSize(username, 15).Message("Username must be at most 15 characters long")
    c.Validation.MinSize(username, 4).Message("Username must be at least 4 characters long")
    c.Validation.Match(username, regexp.MustCompile("^\\w*$")).Message("Username must be all letters")

    if c.Validation.HasErrors() {
        // Store the validation errors in the flash context and redirect.
        c.Validation.Keep()
        c.FlashParams()
        return c.Redirect(Hotels.Settings)
    }

    // All the data checked out!
    ...
}
```

模板:

```

{{/* app/views/Hotels/Settings.html */}}
...
{{if .errors}}
<div class="error">
  <ul>
    {{range .errors}}
      <li> {{.Message}}
    {{end}}
  </ul>
</div>
{{end}}
...

```

Session/Flash

Higo 提供两个基于 cookie 的存储机制.



```

// A signed cookie (and thus limited to 4kb in size).
// Restriction: Keys may not have a colon in them.
type Session map[string]string

// Flash represents a cookie that gets overwritten on each request.
// It allows data to be stored across one page at a time.
// This is commonly used to implement success or error messages.
// e.g. the Post/Redirect/Get pattern: http://en.wikipedia.org/wiki/Post/Redirect/Get
type Flash struct {
    Data, Out map[string]string
}

```



Session

Higo session 是一个字符串字典, 存储为加密签名的 cookie.

它有下面的暗示:

大小不超过 4kb

全部的数据必须被序列化为一个字符串存储

全部的数据可以被用户查看(它没有被编码), 但他是安全的。

Flash

Flash 提供一个单次使用的字符串存储. 它对于实现 the Post/Redirect/Get 模式很有帮助 , 或者用于转换 "操作成功!" 或 "操作失败!" 消息.

下面是这个模式的例子:

```
// Show the Settings form
func (c App) ShowSettings() web.Result {
    return c.Render()
}

// Process a post
func (c App) SaveSettings(setting string) web.Result {
    c.Validation.Required(setting)
    if c.Validation.HasErrors() {
        c.Flash.Error("Settings invalid!")
        c.Validation.Keep()
        c.Params.Flash()
        return c.Redirect(App.ShowSettings)
    }

    saveSetting(setting)
    c.Flash.Success("Settings saved!")
    return c.Redirect(App.ShowSettings)
}
```

我们来看一下这个例子:

用户加载 settings 页面

用户 post 一个 setting

应用程序处理这个 request, 保存一个错误或成功信息到 flash 并重定向用户到 setting 页面

用户加载 settings 页面, 模板将显示 flash 带来的信息

它使用两个方便的函数:

Flash.Success(message string) 是 Flash.Out["success"] = message 的缩写

Flash.Error(message string) 是 Flash.Out["error"] = message 的缩写

7. 返回值(Result)

返回值必须返回一个 `web.Result`, 它处理 response 的生成并依附于一个简单的接口:

```
type Result interface {
    Apply(req *Request, resp *Response)
}
```

`web.Controller` 提供几个方法来生成结果:

`Render`, `RenderTemplate` - 渲染一个模板, 传递参数.

`RenderJson`, `RenderXml` - 序列化一个结构的 json 或 xml.

`RenderText` - 返回一个纯文本 response.

`Redirect` - 重定向到另一个 action 或 URL

`RenderFile` - 返回一个文件, 通常作为一个附件下载.

`RenderError` - 返回一个 500 response 它渲染 `errors/500.html` 模板.

`NotFound` - 返回一个 404 response 它渲染 `errors/404.html` 模板.

`Todo` - 返回一个 stub response (500)

此外,开发人员可以定义他们自己的 `web.Result` 并返回它.

设置状态码 / Content Type

每一个内建的结果都有一个默认的状态码和 Content Type. 要重写它们的值只需在

response 时简单的设置那些要改变的属性:

```
func (c Application) Action() web.Result {
    c.Response.Status = http.StatusTeapot
    c.Response.ContentType = "application/dishware"
    return c.Render()
}
```

渲染

render 在 action 中被调用, `mvc.Controller.Render` 做了两件事情:

添加全部的参数到 controller 的 `RenderArgs`. 使用他们的本地标示符作为 key.

执行模板 “`views/Controller/Action.html`”, 传入 controller 的 `RenderArgs` 作为数据字典.

如果不成功(例如它不能找到模板文件), 它将返回一个 `ActionResult` 替换.

允许开发人员这样写:

```
func (c MyApp) Action() web.Result {  
    myValue := calculateValue()  
    return c.Render(myValue)  
}
```

在他们的模板中使用 “`myValue`”. 这通常比显式的构造一个字典更方便, 因为在很多的案例中数据将需要作为本地变量来处理.

注意:Higo 通过调用的方法名称来决定模板路径和查找的参数名称.因此, `c.Render()`可能只能从 action 中调用.

RenderJson / RenderXml

应用程序可以调用 `RenderJson` 或 `RenderXml` 来传入任意的 Go 类型(通常是一个 struct). Higo 将用 `json.Marshal` 或 `xml.Marshal` 来序列化它.

如果 `results.pretty=true` 在 `app.conf` 中, 序列化将使用 `MarshalIndent` 来完成, 以便输出更优雅的带缩进的比较利于人们查看的版本.

Redirect

一个帮助函数提供了重定向. 它可以被用于两种方式.

1. 重定向到一个不带参数的 Action

```
return c.Redirect(Hotels.Settings)
```

这种形式对提供一个深度的类型安全和独立的路由很有帮助.(它生成自动的 URL)

2. 重定向到一个格式化字符串

```
return c.Redirect("/hotels/%d/settings", hotelId)
```

这种形式需要必要的参数传递.

它返回一个 302(临时跳转)状态码.

添加你自己的 Result

下面是一个添加简单 Result 的示例.

创建类型:

```
type Html string
```

```
func (r Html) Apply(req *Request, resp *Response) {  
    resp.WriteHeader(http.StatusOK, "text/html")  
    resp.Out.Write([]byte(r))  
}
```

然后在 action 中使用它

```
func (c *Application) Action() web.Result {  
    return Html("<html><body>Hello World</body></html>")  
}
```

状态码

每一个 Result 将默认设置一个状态码.你能通过设置一个你自己的来重写这个默认的状态码.

```
func (c *Application) CreateEntity() web.Result {  
    c.Response.Status = 201  
    return c.Render()  
}
```

模板(Templates)

Higo 使用 Go Templates. 它搜索两个目录来查找模板:

应用程序的 views 目录和全部子目录

Higo 自己的 Templates 目录

Higo 为错误页面提供模板(在开发模式中显示友好的编译错误), 但是应用程序可以通过创建

一个相同名字的模板来重写它, 例如 app/views/errors/500.html

渲染上下文

Higo 使用 `RenderArgs` 的数据字典执行模板. 除了应用程序提供的数据外, Higo 提供了下面的入口:

"errors" - `Validation.ErrorMap` 返回的字典

"flash" - 前一个请求的 flash 数据

模板函数

Go 提供了一些模板函数用于你的模板中. Higo 添加了如下这些, 你可以查看文档或源代码.

eq

一个简单的 "a == b" 测试, 如下所示:

```
<div class="message" {{if eq .User "you"}}you{{end}}>
```

set

设置一个变量到给定的上下文, 如下所示:

```
{{set . "title" "Basic Chat room"}}
```

```
<h1>{{.title}}</h1>
```

append

添加一个变量到一个数组或开始一个新数组到给定的上下文, 如下所示:

```
{{append . "moreScripts" "js/jquery-ui-1.7.2.custom.min.js"}}
```

```
{{range .moreStyles}}
```

```
  <link rel="stylesheet" type="text/css" href="/public/{{.}}">
```

```
{{end}}
```

field

一个 input 元素的帮助函数.

给定一个元素名称, 它返回一个包含了下面成员的 struct

Id: 元素名称, 转换成一个适当的 html 元素的 ID.

Name: 元素名称

Value: 元素在当前 RenderArgs 的值

Flash: 元素的 flash 值

Error: 错误信息,任何与这个元素相关的错误信息

ErrorClass: 如果这里有一个错误为字面量字符串"error", 否则为""

```
{{with $field := field "booking.CheckInDate" .}}
  <p class="{{ $field.ErrorClass }}">
    <strong>Check In Date:</strong>
    <input type="text" size="10" name="{{ $field.Name }}" class="datepicker"
value="{{ $field.Flash }}">
    * <span class="error">{{ $field.Error }}</span>
  </p>
{{end}}
```

option

协助构造 HTML 的 option 元素, 结合 field 帮助函数, 如下所示:

```
{{with $field := field "booking.Beds" .}}
<select name="{{ $field.Name }}">
  {{option $field "1" "One king-size bed"}}
  {{option $field "2" "Two double beds"}}
  {{option $field "3" "Three beds"}}
</select>
{{end}}
```

radio

协助构建 HTML 的 radio 元素, 结合 field 帮助函数, 如下所示:

```
{{with $field := field "booking.Smoking" .}}
  {{radio $field "true"}} Smoking
```

```
{{radio $field "false"}} Non smoking  
{{end}}
```

包含

Go 模板允许你通过包含来组建模板, 如下所示:

```
{{include "header.html"}}
```

这里有两件需要注意的事:

路径是相对的 app/views

任何被包含的模板必须在根目录下(app/views). 这是一个(希望是临时的)限制

技巧

这个示例演示了 Higo 尝试有效的使用 Go 模板. 如下所示:

revel/samples/booking/app/views/header.html

revel/samples/booking/app/views/Hotels/Book.html

它利用帮助函数来在模板中设置 title 和额外的样式, header 示例如下:

```
<html>  
  <head>  
    <title>{{.title}}</title>  
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">  
    <link rel="stylesheet" type="text/css" media="screen" href="/public/css/main.css">  
    <link rel="shortcut icon" type="image/png" href="/public/img/favicon.png">  
    {{range .moreStyles}}  
      <link rel="stylesheet" type="text/css" href="/public/{{.}}">  
    {{end}}  
    <script src="/public/js/jquery-1.3.2.min.js" type="text/javascript" charset="utf-8"></script>  
    <script src="/public/js/sessvars.js" type="text/javascript" charset="utf-8"></script>  
    {{range .moreScripts}}  
      <script src="/public/{{.}}" type="text/javascript" charset="utf-8"></script>  
    {{end}}  
  </head>
```

模板包含它看起来像这样

```

{{set . title "Hotels"}}
{{append . "moreStyles" "ui-lightness/jquery-ui-1.7.2.custom.css"}}
{{append . "moreScripts" "js/jquery-ui-1.7.2.custom.min.js"}}
{{template "header.html" .}}

```

自定义函数

应用程序可以在模板中注册使用自定义函数, 示例如下:

```

func init() {
    web.TemplateFuncs["eq"] = func(a, b interface{}) bool { return a == b }
}

```

8. 拦截器 (Interceptors)

一个拦截器是一个框架在调用 action 方法前或后调用的函数. 它允许一种 AOP 的形式,

它经常被用于做下面几种事情:

Request logging

Error handling

Stats keeping

在 Higo 里, 一个拦截器能接受两种形式:

1. 函数拦截器: 一个函数满足 InterceptFunc 接口

没有访问特定的应用程序 Controller 被调用

在应用程序中可以应用于任意或全部 Controller

2. 方法拦截器: 一个 controller 的方法接受没有参数和 web.Result 的返回值

可以只拦截受约束的 Controller

可以修改被调用的 controller 作为想得到的

拦截器按照被添加的顺序被调用.

拦截器次数

一个拦截器能在请求生命周期被注册并在 4 个点运行:

BEFORE: 在请求被路由, session, flash 和参数反编码后, 但在 action 被调用前

AFTTER: 在请求已经返回一个 Result 后, 但在 Result 被应用之前. 如果 aciton 产生 panic

这些拦截器将不会被调用

PANIC: 一个 panic 退出一个 action 或提出应用返回 Result 后

FINALLY: 在一个 action 完成和 Result 被应用后

结果

拦截器典型的返回 nil, 在这种情况下请求不用拦截器并继续处理.

返回非 nil 的效果 web.Result 依赖拦截器的调用.

BEFORE: 没有更深远的拦截器被调用, action 也没有

AFTTER: 全部的拦截器都在运行

PANIC: 全部的拦截器都在运行

FINALLY: 全部的拦截器都在运行

在所有情况下, 任何返回的 Result 将代替任何目前的 Result.

在 BEFORE 情况下, 无论如何, 返回的结果保证是最终结果, 当在 AFTER 情况下, 它可能是

一个更进一步的拦截器可以发出它自己的 Result.

例子

函数对拦截器

这里有一个简单的示例定义和注册一个函数拦截器.

```
func checkUser(c *web.Controller) web.Result {  
    if user := connected(c); user == nil {  
        c.Flash.Error("Please log in first")  
    }  
}
```

```

    return c.Redirect(Application.Index)
}
return nil
}

func init() {
    web.InterceptFunc(checkUser, web.BEFORE, &Hotels{})
}

```

方法拦截器

一个方法拦截器签名可以是两种形式中的一个:

```

func (c ApplicationController) example() web.Result
func (c *AppController) example() web.Result

```

这是个相同的示例它只操作应用程序的 controller

```

func (c Hotels) checkUser() web.Result {
    if user := connected(c); user == nil {
        c.Flash.Error("Please log in first")
        return c.Redirect(Application.Index)
    }
    return nil
}

func init() {
    web.InterceptMethod(checkUser, web.BEFORE)
}

```

9. 插件(Plugins)

插件被注册到应用程序的 hook 上面和请求生命周期事件相联系.

一个插件就像下面的接口 (每一个事件都将被通知):

```

type Plugin interface {
    // Server 启动时被 call (每一次代码重新加载).
    OnAppStart()
    // 路由器完成配置后被 call.
    OnRoutesLoaded(router *Router)
}

```

```

// 每一次 request 之前被 call.
BeforeRequest(c *Controller)
// 每一次 request 之后被 call. ( 除了 panics )
AfterRequest(c *Controller)
// 当一个 panic 退出一个 action 时被 call , 并有一个 recovered 值
OnException(c *Controller, err interface{})
// 每次 request 后 ( 无论是不是 panic ), Result 已经被应用后被 call
Finally(c *Controller)
}

```

定义一个你自己的插件, 声明一个嵌入 `web.EmptyPlugin` 的类型并重写你想要的方法然后注册 `web.RegisterPlugin`.

```

type DbPlugin struct {
    web.EmptyPlugin
}

func (p DbPlugin) OnAppStart() {
    ...
}

func init() {
    web.RegisterPlugin(DbPlugin{})
}

```



Higo 将调用提供给 `RegisterPlugin` 的单例的全部方法, 所以请确保方法是线程安全的.

开发区域

添加更多的插件可以处理的东西: 完整的请求, 渲染模板, 一个最终的调用不管是否有 panic.

提供一个正式的地方调用 `RegisterPlugin`.

10. 模块(Module)

模块是可以被插入到应用程序的包. 他们允许在多个 Higo 应用或第三方程序中共享

controller、view、资源文件和其他代码

模块应该有相同的布局来作为一个 Higo 应用程序. 主应用程序将把模块按照下面的方式合并：

任何在 module/app/views 中的模板将被添加到模板加载器的搜索路径

任何在 module/app/controllers 中的 controller 将被视为他们在你的应用程序中

通过一个路由的形式 staticDir:module:public，资源文件就可以使用了

开启一个 module

为了添加一个模块到你的应用程序，添加如下一行代码到 app.conf:

```
module.mymodulename = go/import/path/to/module
```

一个空的导入路径将禁用模块:

```
module.mymodulename =
```

例如, 开启 test runner 模块:

```
module.testrunner = higo/web/modules/testrunner
```

开发区域

模块的文件 conf/routes 应该是可以被主应用程序挂载的.

11. Websockets

Higo 支持 Websockets.

处理一个 Websocket 连接:

1. 使用 WS 方法添加一个路由.
2. 添加一个 action 接受一个 *websocket.Conn 参数.

例如添加如下代码到你的 routes 文件:

```
WS /app/feed Application.Feed
```

然后写一个 action 如下面:

```
import "higo/websocket"

func (c Application) Feed(user string, ws *websocket.Conn) web.Result {
    ...
}
```

12. 测试(Testing)

Higo 提供了一个测试框架,这使得在应用程序中写和运行测试函数变得很容易.

skeleton 应用程序骨架中带有简单的测试来帮助我们测试.

概要

测试保存在 tests 目录

```
corp/myapp
  app/
  conf/
  public/
  tests/  <----
```

一个简单的测试看起来像下面这样:



```
type ApplicationTest struct {
    web.TestSuite
}

func (t ApplicationTest) Before() {
    println("Set up")
}

func (t ApplicationTest) TestThatIndexPageWorks() {
    t.Get("/")
```

```
t.AssertOk()
t.AssertContentType("text/html")
}
```

```
func (t ApplicationTest) After() {
    println("Tear down")
}
```



上面的示例代码展示了几件事:

一个测试工具是任意嵌入 `web.TestSuite` 的 struct

如果存在 `Before()` 和 `After()` 方法, 它们将在每一个测试方法的前后被调用

`web.TestSuite` 为发布请求到应用程序和断言响应信息提供帮助

一个断言失败产生一个 panic, 它将被 harness 捕获

你可以已两种方式运行测试:

交互式的, 从你的浏览器运行在测试部署时很有帮助

非交互式的, 从命令行运行对结合一个持续集成很有帮助

开发一个测试工具

创建一个你自己的测试工具, 定义一个嵌入 `web.Testsuite` 的 struct, 它提供一个 HTTP 客

户端和许多帮助方法来发出请求到你的应用程序.



```
type TestSuite struct {
    Client      *http.Client
    Response    *http.Response
    ResponseBody []byte
}

// Some request methods
func (t *TestSuite) Get(path string)
func (t *TestSuite) Post(path string, contentType string, reader io.Reader)
func (t *TestSuite) PostForm(path string, data url.Values)
func (t *TestSuite) MakeRequest(req *http.Request)

// Some assertion methods
```

```
func (t *TestSuite) AssertOk()
func (t *TestSuite) AssertContentType(contentType string)
func (t *TestSuite) Assert(exp bool)
func (t *TestSuite) Assertf(exp bool, formatStr string, args ...interface{})
```



全部的请求方法表现相似:

它们接收一个路径(例如: /users/)

它们发出请求到应用程序服务器

它们把响应存储了 Response 属性中

它们读取全部的响应 body 到 ResponseBody 属性

如果开发人员希望使用自定义的 HTTP Client 代替默认的 [http.DefaultClient](#), 它们应该在 Before()方法里面替换它.

如果它们没有满足条件全部断言都将产生一个 panic. 全部的 panic 被测试 harness 捕获并展示为错误.

运行一个测试工具

为了运行任何测试, testrunner 模块必须被激活. 添加下面一行代码到 app.conf 以保证激活它

```
module.testrunner = higo/web/modules/testrunner
```

完成上面之后测试就被运行了(交互式或非交互式)

运行交互式的测试

利用 Higo 的热编译功能, 一个交互式的测试运行器用来提供给快速编辑刷新的循环工作.

例如, 开发人员在他们的浏览器加载 /@tests

Test Runner

Run all of your application's tests from here.

Run All Tests

ApplicationTest

TestThatIndexPageWorks

Run

然后他们添加一个测试方法

```
func (t ApplicationTest) TestSomethingImportant() {  
    t.Get("/")  
    t.AssertOk()  
    t.AssertContentType("text/xml")  
}
```

刷新页面将看到新的测试方法

Test Runner

Run all of your application's tests from here.

Run All Tests

ApplicationTest

TestSomethingImportant

Run

TestThatIndexPageWorks

Run

运行这个测试

Test Runner

Run all of your application's tests from here.

Run All Tests

ApplicationTest

TestSomethingImportant

Header Content-Type: (expected) text/xml != text/html (actual)
In /tests/apptest.go (around line 22) : t.AssertContentType("text/xml")
[Show Stack](#)

Run

TestThatIndexPageWorks

Run

它没有正常工作. 我们来修复这个问题替换 “text/xml” 为 “text/html”, 刷新浏览器:

Test Runner

Run all of your application's tests from here.

Run All Tests

ApplicationTest

TestSomethingImportant

Run

TestThatIndexPageWorks

Run

成功.

运行非交互式的测试

Higo 命令行工具 提供了一个 `test` 命令, 它运行全部的应用程序在命令行工具中运行测试.

示例如下:



```
$ Higo test liuapp dev
```

...

Go to `/@tests` to run the tests.

1 test suite to run.

```
ApplicationTest      PASSED      0s
```

All Tests Passed.

在控制台只有一个简单的 `PASSED/FAILED` 概要通过测试工具来显示. 这个工具写入更多的结果到文件系统:

```
$ cd src/liuapp
$ find test-results
test-results
test-results/app.log
test-results/ApplicationTest.passed.html
test-results/result.passed
```



它写入了 3 个不同的东西:

应用程序的 `stdout` 和 `stderr` 被重定向到 `app.log`

一个 HTML 文件每个测试工具都写入描述测试的通过和失败的信息

要么 `result.passed` 要么 `result.failed` 被写入, 依赖于总体是否成功

这里有两个集成这个到持续构建的建议机制

检查返回代码, 0 表示成功非 0 另外

运行后需要 `result.success` 或者不允许 `result.failed`.

实现说明

Higo 做了什么:

为嵌套 `TestSuite` 类型扫描测试源代码

在生成 `main.go` 时设置 `web.TestSuites` 变量到那些类型的列表

使用反射在 `TestSuite` 类型上查找全部的以 `Test` 开头的方法并调用它们来运行测试

从 bugs 或失败的断言中捕获 panics 并显示有帮助的错误信息

开发区域

可以使用以下方式改进测试框架

Fixtures 来填充测试数据

记录器写入一个文件(替换 `stderr` / `stdout`)也应该被重定向到 `test-results/app.log`

13. 日志(Logging)

Higo 提供 4 个日志记录器

- TRACE - 只有 debug 信息.
- INFO - 信息报告.
- WARN - 一些意外的但无害的.

- ERROR - 错误信息，不需要看一看了。

日志可以在 `app.conf` 中配置。下面是一个例子：

```
app.name = liuapp

[dev]
log.trace.output = stdout
log.info.output  = stdout
log.warn.output  = stderr
log.error.output = stderr

log.trace.prefix = "TRACE "
log.info.prefix  = "INFO  "

log.trace.flags  = 10
log.info.flags   = 10

[prod]
log.trace.output = off
log.info.output  = off
log.warn.output  = log/%(app.name)s.log
log.error.output = log/%(app.name)s.log
```

在 **dev** 模式下：

- 大多数详细的日志将被显示。
- **info** 或 **trace** 的日志信息将会加上前缀。

在 **prod** 模式下：

- **info** 和 **trace** 的日志将被忽略。
- warnings 和 errors 日志将被添加到 **log/liuapp.log** 文件。

设置日志的标志位, 你必须从 the flag constants 计算标志位的值. 例如, 要格式化 01:23:23 /a/b/c/d.go:23 信息 需要标志位 $Ltime \mid Llongfile = 2 \mid 8 = 10$.

开发区域:

- 如果日志目录不存在 Higo 应该创建日志目录.

14. 部署(Deployment)

SCP

Higo 应用程序可以被部署到没有安装 Go 功能的机器上. 命令行工具 提供了 package 命令, 它可以编译和打包应用程序并附带一个运行它的脚本.

一个典型的部署看起来像下面这样:

```
# Run and test my app.
$ Higo run import/path/to/app
.. test app ..

# Package it up.
$ Higo package import/path/to/app
Your archive is ready: app.zip

# Copy to the target machine.
$ scp app.zip target:/srv/

# Run it on the target machine.
$ ssh target
$ cd /srv/
$ ./run.sh
```



开发区域:

交叉编译 (e.g. 在 OSX 上开发, 在 Linux 上部署).

15.app.conf

应用程序配置文件被命名为 `app.conf` ,它使用 goconfig 的语法 ,它看起来有点像微软的 INI 文件。

下面是一个例子文件 :



```
app.name=chat
app.secret=pJLzyoiDe17L36mytqC912j81PfTiolHm1veQK6Grn1En3YFdB5lvEHVTwFEaWvj
http.addr=
http.port=9000

[dev]
results.pretty=true
watch=true

log.trace.output = off
log.info.output  = stderr
log.warn.output  = stderr
log.error.output = stderr

[prod]
results.pretty=false
watch=false

log.trace.output = off
log.info.output  = off
log.warn.output  = %(app.name)s.log
log.error.output = %(app.name)s.log
```



每一个 section 是一个运行模式.在文件最上面的 Key (没有在任何 section 里面) 被应用到全部的运行模式中 , 类似于全局变量。在[prod]下面的 key 只应用于 prod 模式。这允许默认值可以跨多种运行模式 , 当需要时也可以被重写。

新的应用程序以 dev 和 prod 运行模式运行 , 但用户也可以创建任意的 section.运行模式的选择是在使用 Higo run 时通过参数决定的。(命令行工具)

自定义属性

开发人员可以定义自己的 key 并通过 web.Config variable 来访问 , 它暴露了一个简单的 api

内建属性

Higo 使用下面的内部属性 :

- app.name
- app.secret - 密钥用于 session cookie 的签名 (and anywhere the application uses web.Sign)
- http.port - 监听端口
- http.addr - 绑定的 ip 地址 (空字符串为通配符)
- results.pretty - RenderXml 和 RenderJson 格式化 XML/JSON.
- watch - 允许源码监控. 如果设置为 false 将禁用监控 (default true)
- watch.templates - Higo 应该监控和重新加载视图和模板文件的修改 (default True)
- watch.routes - Higo 应该监控和重新加载 routes (default True)

- watch.code - Higo 应该监控和重新加载 code (default True)
- cookie.prefix - 重命名 Higo 的 cookie 名称 (default "HIGO")
- log.* - 日志配置

16. 命令行工具(Command line)

构建和运行

为了使用 Higo 你必须构建命令行工具.从你的 GOPATH 根目录开始.

```
$ go build -o bin/Higo higo/cmd
```

现在运行：

```
$ Higo
~
~ Higo! http://huser.github.com/higo
~
usage: Higo command [arguments]

The commands are:

new          create a skeleton Higo application
run          run a Higo application
build        build a Higo application (e.g. for deployment)
package      package a Higo application (e.g. for deployment)
clean        clean a Higo application's temp files
test         run all tests from the command-line
```

Use "Higo help [command]" for more information.

请参考工具的内建帮助函数已得到更多的帮助信息.

感谢您的阅读，请继续学习和关注《**Higo 框架开发手册**》。 * 完 *