

## 1. beego简介

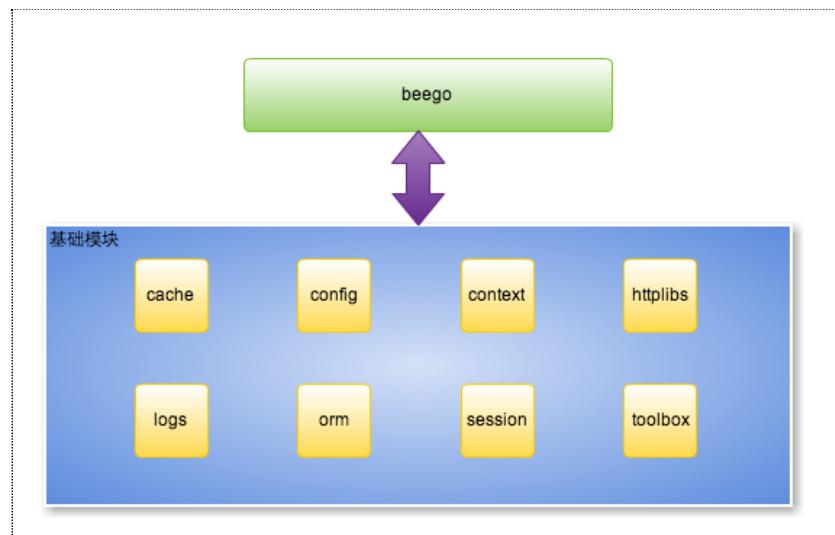
- 1. beego的架构
- 2. beego的执行逻辑
- 3. beego项目结构

## beego简介

beego是一个快速开发Go应用的http框架，他可以用来快速开发API、Web、后端服务等各种应用，是一个RESTful的框架，主要设计灵感来源于tornado、sinatra、flask这三个框架，但是结合了Go本身的一些特性(interface、struct继承等)而设计的一个框架。

## beego的架构

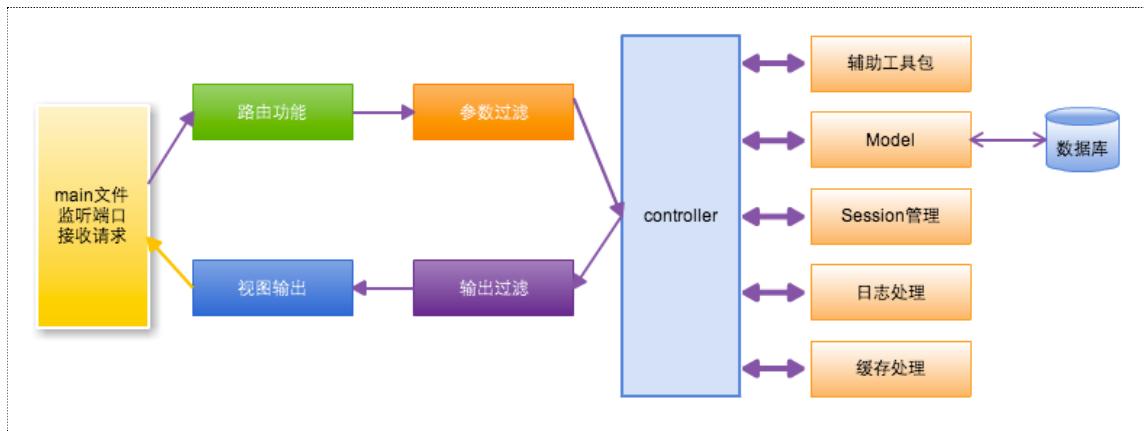
beego的整体设计架构如下所示：



beego是基于八大独立的模块之上构建的，是一个高度解耦的框架。当初设计beego的时候就是考虑功能模块化，用户即使不适用beego的http逻辑，也是可以在使用这些独立模块，例如你可以使用cache模块来做你的缓存逻辑，使用日志模块来记录你的操作信息，使用config模块来解析你各种格式的文件，所以不仅仅在beego开发中，你的socket游戏开发中也是很有用的模块，这也是beego为什么受欢迎的一个原因。大家如果玩过乐高的话，应该知道很多高级的东西都是一块一块的积木搭建出来的，而设计beego的时候，这些模块就是积木，高级机器人就是beego。至于这些模块的功能以及如何使用会在后面的文档会逐一介绍。

## beego的执行逻辑

既然beego是基于这些模块构建的，那么他的执行逻辑是怎么样的呢？beego是一个典型的MVC架构，他的执行逻辑如下图所示：



## beego项目结构

一般的beego项目的目录如下所示：

```

├── conf
│   └── app.conf
├── controllers
│   ├── admin
│   │   └── default.go
│   └── main.go
├── models
│   └── models.go
├── static
│   ├── css
│   ├── ico
│   ├── img
│   └── js
└── views
    ├── admin
    └── index.tpl

```

从上面的目录结构我们可以看出来M(models目录)、V(views目录)、C/controllers目录)的结构，`main.go`是入口文件。

你可以通过bee来新建项目

1. beego的安装
2. beego的升级
3. beego的git分支
4. 如何给beego做贡献

## beego的安装

beego的安装是典型的Go安装包的形式：

```
1. go get github.com/astaxie/beego
```

常见问题：

- git 没有安装，请自行安装不同平台的git，如何安装请自行搜索
- git https无法获取，请配置本地的git，关闭https验证：

```
1. git config --global http.sslVerify false
```

- 无法上网怎么安装beego，目前没有好的办法，接下来我会整理一个全包下载，每次发布正式版本都会提供这个全包下载，包含依赖包。

## beego的升级

beego升级分为go方式升级和源码下载升级：

- Go升级,通过该方式用户可以升级beego框架，强烈推荐该方式

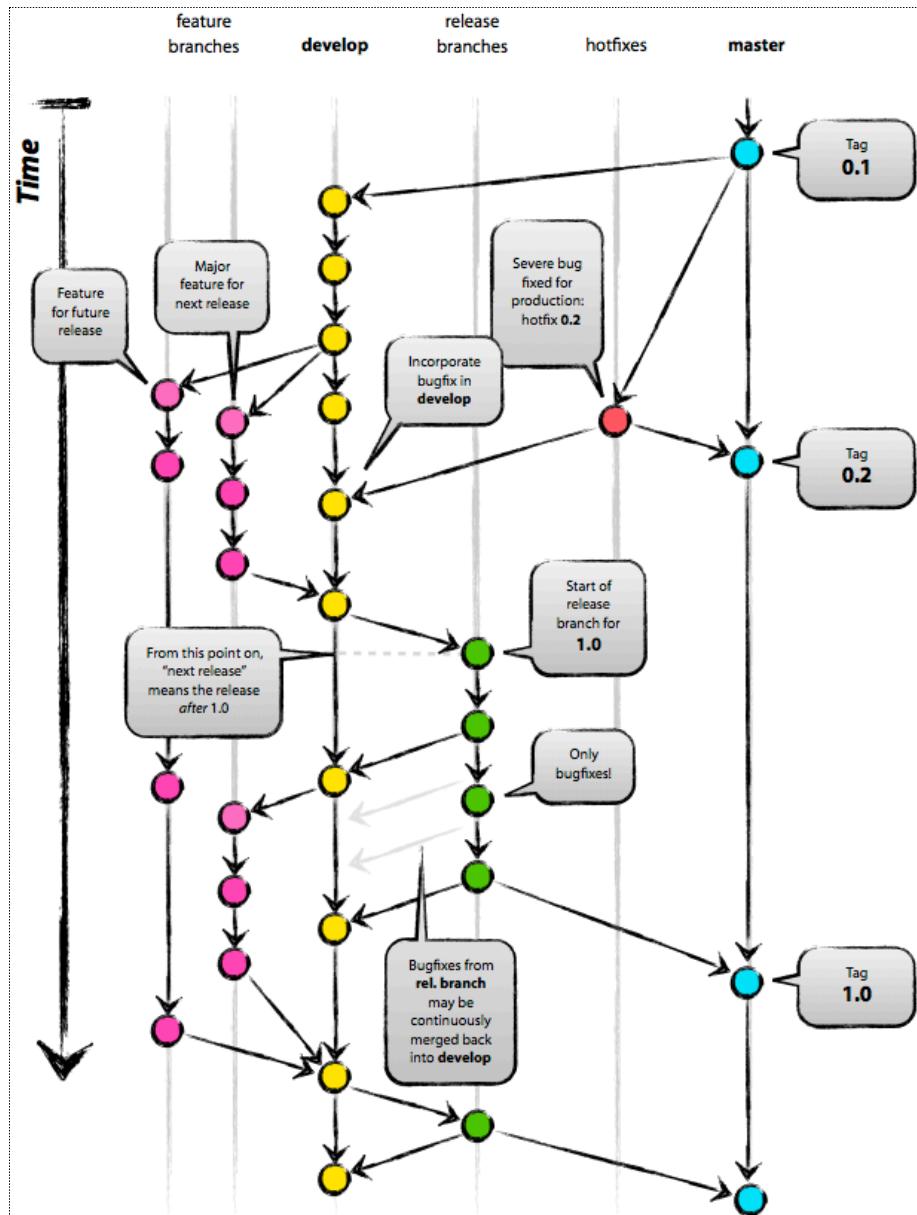
```
1. go get -u github.com/astaxie/beego
```

- 源码下载升级，用户访问 <https://github.com/astaxie/beego> ,下载源码，然后覆盖到gopath/github.com/astaxie/beego目录，然后通过本地执行安装就可以升级了

```
1. go install github.com/astaxie/beego
```

## beego的git分支

beego目前的开发方式是master是开发分支，但是在1.0版本之后我们将采用如下的方式进行分支管理：



## 如何给beego做贡献

beego的源码托管在github，还好github是一个非常方便就能协同工作的平台，你可以fork，然后在你的本地进行修改，然后发送pull request给我，我会在最快的时间进行代码的review，然后进行merger。

## 1. bee介绍

- 1. bee安装
- 2. bee命令详解
  - 1. new命令
  - 2. run命令
  - 3. api命令
  - 4. test命令
  - 5. pack命令
  - 6. router命令
  - 7. bale命令

## bee介绍

bee是一个为了快速开发beego项目而创建的项目，通过bee可以快速的创建项目，自动的热编译，开发测试以及开发完之后打包发布的一整套从创建、开发到部署的方案。

## bee安装

通过如下的方式可以正确的安装bee工具：

```
1. go get github.com/beego/bee
```

## bee命令详解

我们在命令行输入 bee，可以看到如下的信息：

```
Bee is a tool for managing beego framework.

Usage:

bee command [arguments]

The commands are:

new      create an application base on beego framework
run      run the app which can hot compile
pack     compress an beego project
api      create an api application base on beego framework
router   auto-generate routers for the app controllers
test     test the app
bale    packs non-Go files to Go source files
```

```
1.
```

## new命令

new 命令是新建一个web项目，我们在命令行下执行 bee new 项目名 就可以创建一个新的项目，但是注意该命令必须在gopath下执行，这样就会在gopath目录下生成如下目录结构的项目：

```
bee new myproject
[INFO] Creating application...
/gopath/src/myproject/
/gopath/src/myproject/conf/
/gopath/src/myproject/controllers/
/gopath/src/myproject/models/
/gopath/src/myproject/static/
/gopath/src/myproject/static/js/
/gopath/src/myproject/static/css/
/gopath/src/myproject/static/img/
/gopath/src/myproject/views/
/gopath/src/myproject/conf/app.conf
```

```
/gopath/src/myproject/controllers/default.go  
/gopath/src/myproject/views/index.tpl  
/gopath/src/myproject/main.go  
13-11-25 09:50:39 [SUCC] New application successfully created!
```

```
myproject  
├── conf  
│   └── app.conf  
├── controllers  
│   └── default.go  
├── main.go  
├── models  
├── static  
│   ├── css  
│   ├── img  
│   └── js  
└── views  
    └── index.tpl
```

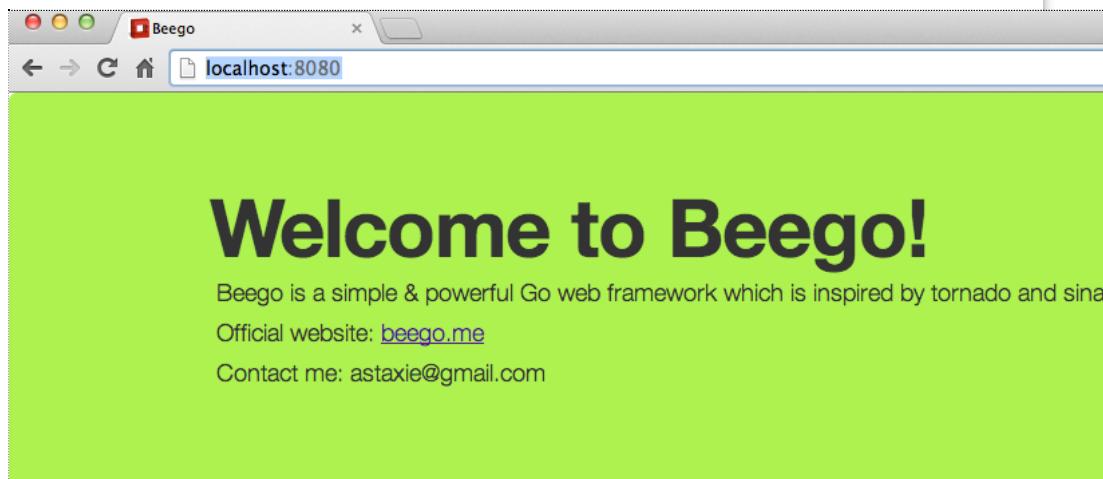
8 directories, 4 files

#### run命令

我们在开发Go项目的时候最大的问题是经常需要自己手动去编译，`bee run` 命令是监控beego的项目，通过inotify监控文件系统，这样只有我们在开发过程中，我们就可以实时的看到项目修改之后的效果：

```
bee run  
13-11-25 09:53:04 [INFO] Uses 'myproject' as 'appname'  
13-11-25 09:53:04 [INFO] Initializing watcher...  
13-11-25 09:53:04 [TRAC] Directory(/gopath/src/myproject/controllers)  
13-11-25 09:53:04 [TRAC] Directory(/gopath/src/myproject/models)  
13-11-25 09:53:04 [TRAC] Directory(/gopath/src/myproject)  
13-11-25 09:53:04 [INFO] Start building...  
13-11-25 09:53:16 [SUCC] Build was successful  
13-11-25 09:53:16 [INFO] Restarting myproject ...  
13-11-25 09:53:16 [INFO] ./myproject is running...
```

我们打开浏览器就可以看到效果 <http://localhost:8080/> :



如果我们修改了Controller下面的default.go文件，我们就可以看到命令行输出：

```
13-11-25 10:11:20 [EVEN] "/gopath/src/myproject/controllers/default.go": DELETE|MODIFY  
13-11-25 10:11:20 [INFO] Start building...  
13-11-25 10:11:20 [SKIP] "/gopath/src/myproject/controllers/default.go": CREATE  
13-11-25 10:11:23 [SKIP] "/gopath/src/myproject/controllers/default.go": MODIFY  
13-11-25 10:11:23 [SUCC] Build was successful  
13-11-25 10:11:23 [INFO] Restarting myproject ...
```

```
13-11-25 10:11:23 [INFO] ./myproject is running...
```

刷新浏览器我们看到新的修改内容已经输出。

### api命令

上面的 `new` 命令是用来新建Web项目，我自己大多数时候是采用beego来开发api应用，提供对外的API，那么这个 `api` 命令就是用来创建API应用的，执行命令之后如下所示：

```
bee api apiproject
create app folder: /gopath/src/apiproject
create conf: /gopath/src/apiproject/conf
create controllers: /gopath/src/apiproject/controllers
create models: /gopath/src/apiproject/models
create tests: /gopath/src/apiproject/tests
create conf app.conf: /gopath/src/apiproject/conf/app.conf
create controllers default.go: /gopath/src/apiproject/controllers/default.go
create tests default.go: /gopath/src/apiproject/tests/default_test.go
create models object.go: /gopath/src/apiproject/models/object.go
create main.go: /gopath/src/apiproject/main.go
```

这个项目的目录结构如下：

```
apiproject
├── conf
│   └── app.conf
├── controllers
│   └── default.go
├── main.go
└── models
    └── object.go
└── tests
    └── default_test.go
```

从上面的目录我们可以看到和Web项目相比，少了`static`和`views`目录，多了一个`test`模块，用来做单元测试的。

### test命令

这是基于 `go test` 进行封装的一个命令，执行beego项目`test`目录下的测试用例：

```
bee test apiproject
13-11-25 10:46:57 [INFO] Initializing watcher...
13-11-25 10:46:57 [TRAC] Directory(/gopath/src/apiproject/controllers)
13-11-25 10:46:57 [TRAC] Directory(/gopath/src/apiproject/models)
13-11-25 10:46:57 [TRAC] Directory(/gopath/src/apiproject)
13-11-25 10:46:57 [INFO] Start building...
13-11-25 10:46:58 [SUCC] Build was successful
13-11-25 10:46:58 [INFO] Restarting apiproject ...
13-11-25 10:46:58 [INFO] ./apiproject is running...
13-11-25 10:46:58 [INFO] Start testing...
13-11-25 10:46:59 [TRAC] ===== Test Begin =====
PASS
ok      apiproject/tests    0.100s
13-11-25 10:47:00 [TRAC] ===== Test End =====
13-11-25 10:47:00 [SUCC] Test finish
```

### pack命令

`pack` 目录用来发布应用的时候打包，会把项目打包成zip包，这样我们部署的时候直接打包之后的项目上传，解压就可以部署了：

```
bee pack
app path: /gopath/src/apiproject
GOOS darwin GOARCH amd64
build apiproject
build success
exclude prefix:
exclude suffix: .go:.DS_Store:.tmp
file write to `/gopath/src/apiproject/apiproject.tar.gz`
```

我们可以看到目录下有如下的压缩文件：

```
rwxr-xr-x  1 astaxie  staff  8995376 11 25 22:46 apiproject
-rw-r--r--  1 astaxie  staff  2240288 11 25 22:58 apiproject.tar.gz
drwxr-xr-x  3 astaxie  staff   102 11 25 22:31 conf
drwxr-xr-x  3 astaxie  staff   102 11 25 22:31 controllers
-rw-r--r--  1 astaxie  staff   509 11 25 22:31 main.go
drwxr-xr-x  3 astaxie  staff   102 11 25 22:31 models
drwxr-xr-x  3 astaxie  staff   102 11 25 22:31 tests
```

#### **router命令**

这个目录目前还没有作用，将来会用来分析controller的方法，自动生成路由

#### **bale命令**

这个命令现在还没有作用，将来用来压缩所有的静态文件变成一个变量申明文件，全部编译到二进制文件里面，用户发布的时候无需发布静态文件，包括js、css、img和views

1. 创建项目
2. 运行项目

## 创建项目

beego的项目基本都是通过 bee 命令来创建的，所以在创建项目之前确保你已经安装了bee和beego，如果你还没有安装，那么请访问[beego安装](#)和[bee安装](#)。

现在一切就绪我们就可以开始创建项目了，打开终端，进入gopath所在的目录：

```
1. → src bee new quickstart
2. [INFO] Creating application...
3. /gopath/src/quickstart/
4. /gopath/src/quickstart/conf/
5. /gopath/src/quickstart/controllers/
6. /gopath/src/quickstart/models/
7. /gopath/src/quickstart/static/
8. /gopath/src/quickstart/static/js/
9. /gopath/src/quickstart/static/css/
10. /gopath/src/quickstart/static/img/
11. /gopath/src/quickstart/views/
12. /gopath/src/quickstart/conf/app.conf
13. &l
t;
/SPAN>/gopath/src/quickstart/controllers/default.go
14. /gopath/src/quickstart/views/index.tpl
15. /gopath/src/quickstart/main.go
16. 13-11-26 10:34:10
[SUCC] New application successfully created!
```

通过一个简单的命令就创建了一个beego项目。他的目录结构如下所示

```
1. quickstart
2. └── conf
3.   └── app.conf
4. └── controllers
5.   └── default.go
6. └── main.go
7. └── models
8. └── static
9.   ├── css
10.  ├── img
11.  └── js
12. └── views
13.   └── index.tpl
```

从目录结构中我们也可以看出来这是一个典型的MVC架构的应用，`main.go` 是入口文件。

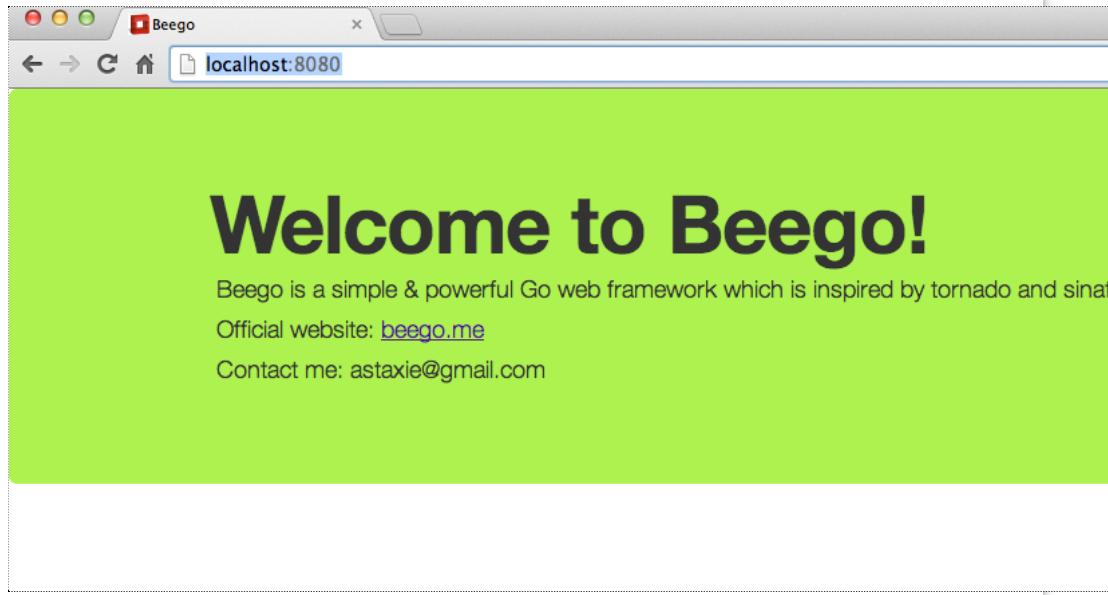
## 运行项目

beego项目创建之后，我们就开始运行项目，首先进入创建的项目，我们使用 `bee run` 来运行该项目，这样就可以做到热编译的效果：

```
1. → src cd quickstart
```

```
2. → quickstart bee run
3. 13-11-26 10:43:14 [INFO] Uses 'quickstart' as 'appname'
4. 13-11-26 10:43:14 [INFO] Initializing watcher...
5. 13-11-26 10:43:14 [TRAC] Directory(/gopath/src/quickstart/controllers)
6. 13-11-26 10:43:14 [TRAC] Directory(/gopath/src/quickstart/models)
7. 13-11-26 10:43:14 [TRAC] Directory(/gopath/src/quickstart)
8. 13-11-26 10:43:14 [INFO] Start building...
```

这样我们的应用已经在8080端口(beego的默认端口)跑起来了，你是不是觉得很神奇，为什么没有nginx和apache居然可以自己干这个事情，是的，Go其实已经做了网络层的东西，beego只是封装了一下，所以可以做到不需要nginx和apache。让我们打开浏览器看看效果吧：



你内心是否激动了？开发网页如此简单有没有。好了，接下来让我们一层一层的剥离来大概的了解beego是怎么运行起来的。

## 1. beego的路由规则

### beego的路由规则

前面我们已经创建了beego项目，而且我们也看到了他已经运行起来了，那么是如何运行起来的呢？让我们从入口文件先分析起来吧：

```
1. package main
2.
3. import (
4.     "quickstart/controllers"
5.     "github.com/astaxie/beego"
6. )
7.
8. func main() {
9.     beego.Router("/", &controllers.MainController{})
10.    beego.Run()
11. }
```

我们可以看到很简单的两句话，第一句是注册路由 `beego.Router`，第二句是 `beego.Run`，那么这两句话都做了一些什么呢？

1. `beego.Router`其实是注册了一个地址，第一个参数是uri(用户请求的地址)，这里我们注册的是 `/`，也就是我们访问的不带任何的uri，第二个参数是对应的Controller，也就是我们即将把请求分发到那个控制器来执行相应的逻辑，我们可以执行类似的方式注册如下路由：

```
1. beego.Router("/user", &controllers.UserController{})
```

这样用户就可以通过访问 `/user` 去执行 `UserController` 的逻辑。这就是我们所谓的路由，更多更复杂的路由规则请查询 [beego路由设置](#)

2. `beego.Run`执行之后，我们看到的效果好像只是监听服务端口这个过程，但是它内部做了很多事情：

- 解析配置文件

beego会自动在conf目录下面去解析相应的配置文件 `app.conf`，这样就可以通过配置文件配置一些例如开启的端口，是否开启session，应用名称等各种信息。

- 开启session

会根据上面配置文件的分析之后判断是否开启session，如果开启的话就初始化全局的session。

- 编译模板

beego会在启动的时候把views目录下的所有模板进行预编译，然后存在map里面，这样可以有效的提供模板运行的效率，无需进行多次编译。

- 启动管理模块

beego目前做了一个很帅的模块，应用内监控模块，会在8088端口做一个内部监听，我们可以通过这个端口查询到QPS、cpu、内存、GC、goroutine、thread等各种信息。

- 监听服务端口

这是最后一步也就是我们看到的访问8080看到的网页端口，内部其实调用了 `ListenAndServe`，充分利用了goroutine的优势

一旦run起来之后，我们的服务就监听在两个端口了，一个服务端口8080，对外服务，一个8088端口，对内监控。

通过这个代码的分析我们了解了beego运行起来的过程，以及内部的一些机制。接下来让我们去剥离Controller如何来处理逻辑的。

## 1. controller逻辑

### controller逻辑

前面我们了解了如何把用户的请求分发到控制器，这小节我们就介绍大家如何来写控制器，首先我们还是从源码分析入手：

```
1. package controllers
2.
3. import (
4.     "github.com/astaxie/beego"
5. )
6.
7. type MainController struct {
8.     beego.Controller
9. }
10.
11. func (this *MainController) Get() {
12.     this.Data["Website"] = "beego.me"
13.     this.Data["Email"] = "astaxie@gmail.com"
14.     this.TplNames = "index.tpl"
15. }
```

上面的代码显示首先我们什么了一个控制器 `MainController`，这个控制器里面内嵌了 `beego.Controller`，这就是Go的继承方式，也就是 `MainController` 继承了所有 `beego.Controller` 的方法。

而 `beego.Controller` 拥有很多方法，其中包括 `Init`、`Prepare`、`Post`、`Get`、`Delete`、`Head` 等方法。我们可以通过重写的方式来实现这些方法，而我们上面的代码就是重写了 `Get` 方法。

我们先前介绍过beego是一个RESTful的框架，所以我们的请求默认是执行对应 `req.Method` 的方法，例如浏览器的是 `GET` 请求，那么默认就会执行 `MainController` 下的 `Get` 方法。这样我们上面的Get方法就会被执行到，这样就进入了我们的逻辑处理。

里面的代码我们需要执行的逻辑，这里只是简单的输出数据，我们可以通过各种方式获取数据，然后赋值到 `this.Data` 中，这是一个用来存储输出数据的map，可以赋值任意类型的值，这里我们只是简单举例输出两个字符串。

最后一个就是需要去渲染的模板，`this.TplNames` 就是需要渲染的模板，这里指定了 `index.tpl`，如果用户不设置该参数，那么默认会去到模板目录的 `Controller/方法名.tpl` 查找，例如上面的方法会去 `MainController/Get.tpl`。

用户设置了模板之后系统会自动的调用 `Render` 函数（这个函数是在`beego.Controller`中实现的），所以无需用户自己来调用渲染。

至此我们的控制器分析基本完成了，接下来让我们看看如何来编写model。

## 1. model分析

### model分析

我们知道Web应用中我们用的最多的就是数据库操作，而model层一般用来做这些操作，我们的 `bee new` 例子不存在Model的演示，但是 `bee api` 应用中存在model的应用，说的简单一点，如果你的应用足够简单，那么Controller可以处理一切的逻辑，如果你的逻辑里面存在着可以复用的东西，那么就抽取出来变成一个模块，那么Model就是逐步抽象的过程，一般我们会在Model里面处理一些数据读取，如下是我对分析应用中的代码片段：

```
1. package models
2.
3. import (
4.     "loggo/utils"
5.     "filepath"
6.     "strconv"
7.     "strings"
8. )
9.
10. var (
11.     NotPV []string = []string{"css", "js", "class", "gif", "jpg", "jpeg", "png", "bmp", "ico", "rss", "xml"}
12. )
13.
14. const big = 0xFFFFFFFF
15.
16. func LogPV(urls string) bool {
17.     ext := filepath.Ext(urls)
18.     if ext == "" {
19.         return true
20.     }
21.     for _, v := range NotPV {
22.         if v == strings.ToLower(ext) {
23.             return false
24.         }
25.     }
26.     return true
27. }
```

所以如果你的应用足够简单，那么就不需要Model了，如果你的模块开始多了，需要复用，需要逻辑分离了，那么Model是必不可少的。接下来我们将分析如何编写View层的东西。

## View编写

在前面编写Controller的时候，我们在Get里面写过这样的语句 `this.TplNames = "index.tpl"`，设置显示的模板文件，默认支持 `tpl` 和 `html` 的后缀名，如果想设置其他后缀你可以调用 `beego.AddTemplateExt` 接口设置，那么模板如何来显示相应的数据呢？beego采用了Go语言默认的模板引擎，所以他的显示和Go的模板语法一样，Go模板的详细使用方法请参考[Go Web 编程 模板使用指南](#)

我们看看快速入门里面的代码（去掉了css样式）：

```
1. <!DOCTYPE html>
2.
3. <html>
4.   <head>
5.     <title>Beego</title>
6.     <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
7.   </head>
8.
9.   <body>
10.    <header class="hero-unit" style="background-color:#A9F16C">
11.      <div class="container">
12.        <div class="row">
13.          <div class="hero-text">
14.            <h1>Welcome to Beego!</h1>
15.            <p class="description">
16.              Beego is a simple & powerful Go web framework which is inspired by tornado and sinatra.
17.              <br />
18.              Official website: <a href="http://{{.Website}}">{{.Website}}</a>
19.              <br />
20.              Contact me: {{.Email}}
21.              </p>
22.            </div>
23.          </div>
24.        </div>
25.      </header>
26.    </body>
27.  </html>
```

我们在Controller里面把数据赋值给了data(map类型)，那么我们在模板中就直接通过key访问 `.Website` 和 `.Email`，这样就做到了数据的输出。接下来我们讲讲解如何让静态文件输出。

## 1. 静态文件处理

### 静态文件处理

前面我们介绍了如何输出静态页面，但是我们的网页往往包含了很多的静态文件，包括图片、JS、CSS等，刚才创建的应用里面就创建了如下目录：

```
|--- static
|   |--- css
|   |--- img
|   |--- js
```

默认beego注册了static目录为静态处理的目录，注册样式：url前缀和映射的目录

```
1. StaticDir["/static"] = "static"
```

用户可以设置多个静态文件处理目录，例如你有多个文件下载目录download1、download2，你可以这样映射：

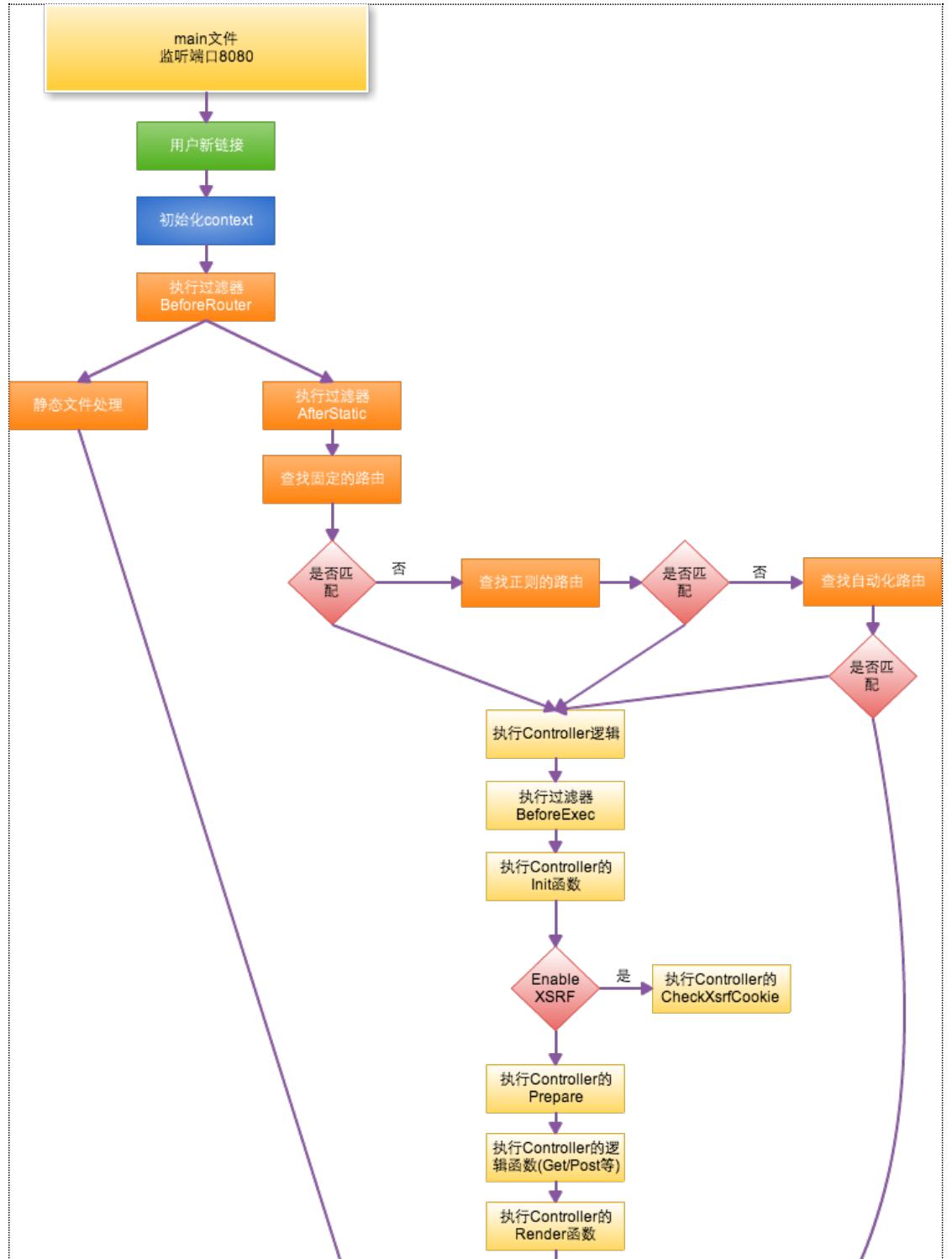
```
1. beego.SetStaticPath("/down1", "download1")
2. beego.SetStaticPath("/down2", "download2")
```

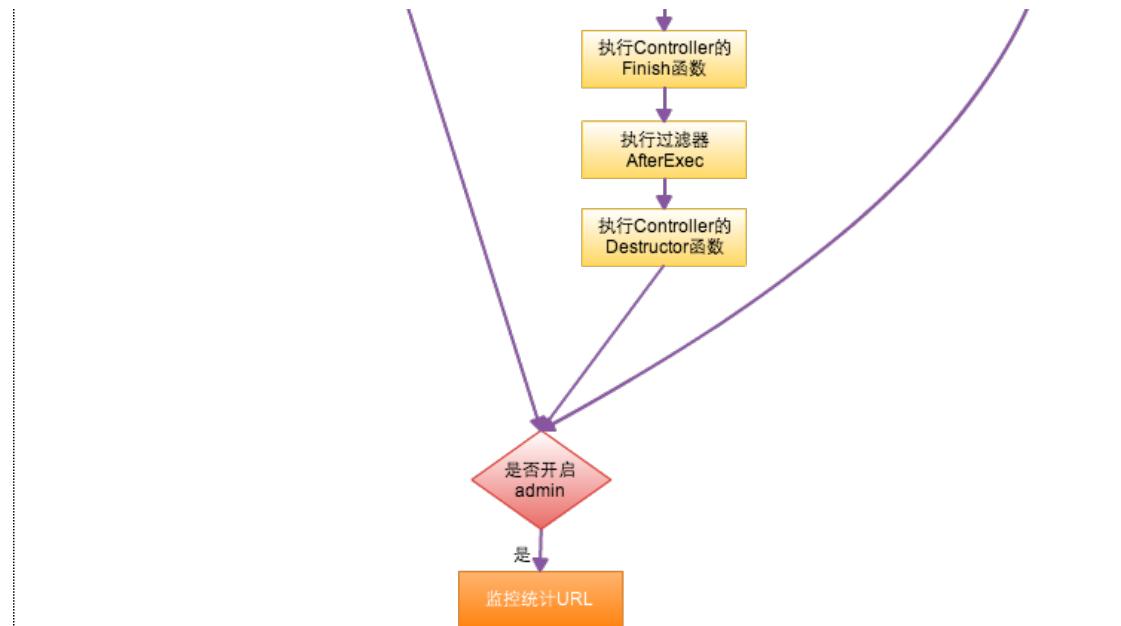
这样用户访问url `http://localhost/down1/123.txt`，默认请求download1目录下的123.txt文件。

## 1. beego的MVC结构介绍

### beego的MVC结构介绍

beego是一个典型的MVC框架，它的整个执行逻辑如下图所示：





通过文字来描述如下：

1. 在监听的端口接收数据，默认监听在8080端口
2. 用户请求到达8080端口之后进入beego的处理逻辑
3. 初始化Context对象，根据请求判断是否为WebSocket请求，如果是的话设置Input，同时判断请求的方法是否在标准请求方法中("get", "post", "put", "delete", "patch", "options", "head")，防止用户的恶意伪造请求攻击造成不必要的影响。
4. 执行BeforeRouter过滤器，当然在beego里面有开关设置，如果用户设置了过滤器，那么该开关打开，这样可以提高在没有开启过滤器的情况下提高执行效率。如果在执行过滤器过程中，responseWriter已经有数据输出了，那么就提前结束该请求，直接跳转到监控判断。
5. 开始执行静态文件的处理，查看用户的请求URL是否和注册在静态文件处理StaticDir中的prefix是否匹配，如果匹配的话，采用http包中默认的ServeFile来处理静态文件。
6. 如果不是静态文件开始初始化session模块(如果开启session的话)，所以这个里面大家需要主要，如果你的BeforeRouter过滤器用到了session就会报错，你应该把加入到AfterStatic过滤器中。
7. 开始执行AfterStatic过滤器，如果在执行过滤器过程中，responseWriter已经有数据输出了，那么就提前结束该请求，直接跳转到监控判断。
8. 执行过过滤器之后，开始从固定的路由规则中查找和请求URL相匹配的对象，这个匹配是全匹配规则，即如果用户请求的URL是 /hello/world，那么固定规则中 /hello 是不会匹配的，只有完全匹配才算匹配，如果匹配的话就进入逻辑执行，如果不匹配进入下一环节的正则匹配。
9. 正则匹配是进行正则的全匹配，这个正则是按照用户添加beego路由顺序来进行匹配的，也就是说，如果你在添加路由的时候你的顺序影响你的匹配。和固定匹配一样，如果匹配的话就进行逻辑执行，如果不匹配进入Auto匹配。
10. 如果用户注册了AutoRouter，那么会通过 controller/method 这样的方式去查找对应的Controller和他内置的方法，如果找到就开始执行逻辑，如果找不到就跳转到监控判断。
11. 如果找到Controller的话，那么就开始执行逻辑，首先执行BeforeExec过滤器，如果在执行过滤器过程中，responseWriter已经有数据输出了，那么就提前结束该请求，直接跳转到监控判断。
12. Controller开始执行Init函数，初始化基本的一些信息，这个函数一般都是beego.Controller的初始化，不建议用户继承的时候修改该函数。
13. 是否开启了XSRF，开启的话就调用Controller的XsrfToken，然后如果是POST请求就调用CheckXsrfCookie方法。
14. 继续执行Controller的Prepare函数，这个函数一般是预留给用户的，用来做Controller里面的一些参数初始化之类的工作。如果在初始化中responseWriter有输出，那么就直接进入Finish函数逻辑。
15. 如果没有输出的话，那么根据用户注册的方法执行相应的逻辑，如果用户没有注册，那么就调用http.Method对应的方法(Get/Post等)。执行相应的逻辑，例如数据读取，数据赋值，模板显示之类的，或者直接输出json或者xml。
16. 如果responseWriter没有输出，那么就调用Render函数进行模板输出。
17. 执行Controller的Finish函数，这个函数是预留给用户用来重写的，用于释放一些资源。释放在Init中初始化的信息数据。
18. 执行AfterExec过滤器，如果有输出的话就跳转到监控判断逻辑。
19. 执行Controller的Destructor，用于释放Init中初始化的一些数据。
20. 如果这一路执行下来都没有找到路由，那么会调用404显示找不到该页面。
21. 最后所有的逻辑都汇聚到了监控判断，如果用户开启了监控模块(默认是开启一个8088端口用于进程内监控)，这样就会把访问的请求链接扔给监控程序去记录当前访问的QPS，对应的链接访问的执行时间，请求链接等。

接下来就让我们开始进入beego的MVC核心第一步，路由设置：

- 路由设置

- 控制器函数
- xsrf过滤
- session控制
- flash数据
- 请求数据处理
- 多种格式数据输出
- 表单数据验证
- 模板输出
- 模板函数
- 错误处理
- 静态文件处理
- 参数配置
- 日志处理

1. 参数配置
2. 默认配置解析
3. 系统默认参数

## 参数配置

beego目前支持ini、xml、json、yaml格式的配置文件解析，但是默认采用了ini格式解析，用户可以通过简单的配置就可以获得很大的灵活性。

### 默认配置解析

beego 默认会解析当前应用下的 `conf/app.conf` 文件。

通过这个文件你可以初始化很多 beego 的默认参数：

```
1. appname = beepkg
2. httpaddr = "127.0.0.1"
3. httpport = 9090
4. runmode ="dev"
5. autorender = false
6. autorecover = false
7. viewspath = "myview"
```

上面这些参数会替换 beego 默认的一些参数。

你可以在配置文件中配置应用需要用的一些配置信息，例如下面所示的数据库信息：

```
1. mysqluser = "root"
2. mysqlpass = "rootpass"
3. mysqlurls = "127.0.0.1"
4. mysqldb   = "beego"
```

那么你就可以通过如下的方式获取设置的配置信息：

```
1. beego.AppConfig.String("mysqluser")
2. beego.AppConfig.String("mysqlpass")
3. beego.AppConfig.String("mysqlurls")
4. beego.AppConfig.String("mysqldb")
```

`AppConfig` 支持如下方法

- `Bool(key string) (bool, error)`
- `Int(key string) (int, error)`
- `Int64(key string) (int64, error)`
- `Float(key string) (float64, error)`
- `String(key string) string`

### 系统默认参数

Beego 中带有很多可配置的参数，我们来一一认识一下它们，这样有利于我们在接下来的 Beego 开发中可以充分的发挥他们的作用(你可以通过在 `conf/app.conf` 中设置对应的值，大小写都可以来重写这些默认值)：

- **AppName**

应用名称， 默认是 Beego。通过 `bee new` 创建的是创建的项目名。

- **AppPath**

当前应用的路径， 默认会通过设置 `os.Args[0]` 获得执行的命令的第一个参数， 所以你在使用supervisor管理进程的时候记得采用全路径启动。

- **AppConfigPath**

配置文件所在的路径， 默认是应用程序对应的目录下的 `conf/app.conf`， 用户可以修改该值配置自己的配置文件。

- **HttpAddr**

应用监听地址， 默认为空， 监听所有的网卡 IP。

- **HttpPort**

应用监听端口， 默认为 8080。

- **HttpTLS**

是否启用https， 默认是关闭

- **HttpCertFile**

开启https之后， certfile的路径

- **HttpKeyFile**

开启https之后， keyfile的路径

- **HttpServerTimeOut**

设置http的超时时间， 默认是0， 不超时

- **RunMode**

应用的模式， 默认是 `dev`， 为开发模式，在开发模式下出错会提示友好的出错页面， 如前面错误描述中所述。

- **AutoRender**

是否模板自动渲染， 默认值为 `true`， 对于 API 类型的应用， 应用需要把该选项设置为 `false`， 不需要渲染模板。

- **RecoverPanic**

是否异常恢复， 默认值为 `true`， 即当应用出现异常的情况， 通过 `recover` 恢复回来， 而不会导致应用异常退出。

- **ViewsPath**

模板路径， 默认值是 `views`。

- **SessionOn**

`session` 是否开启， 默认是 `false`。

- **SessionProvider**

`session` 的引擎， 默认是 `memory`。

- **SessionName**

存在客户端的 cookie 名称， 默认值是 `beegosessionID`。

- **SessionGCMaxLifetime**

`session` 过期时间， 默认值是 3600 秒。

- **SessionSavePath**

`session` 保存路径， 默认是空。

- SessionHashFunc

sessionID生成函数， 默认是sha1

- SessionHashKey

session hash的key

- SessionCookieLifeTime

session默认存在客户端的cookie的时间， 默认值是3600s

- UseFcgi

是否启用 fastcgi， 默认是 false。

- MaxMemory

文件上传默认内存缓存大小， 默认值是 `1 << 26` (64M)。

- EnableGzip

是否开启 gzip 支持， 默认为 false 不支持 gzip，一旦开启了 gzip，那么在模板输出的内容会进行 gzip 或者 zlib 压缩，根据用户的 Accept-Encoding 来判断。

- DirectoryIndex

是否开启静态目录的列表显示， 默认不显示目录，返回 403 错误。

- BeegoServerName

beego服务器默认在请求的时候输出server为beego

- EnableAdmin

是否开启进程内监控模块， 默认开启

- AdminHttpAddr

监控程序监听的地址， 默认值是 `localhost`

- AdminHttpPort

监控程序监听的端口， 默认值是8088

- TemplateLeft

模板左标签， 默认值是 `{{`

- TemplateRight

模板右标签， 默认值是 `}}`

- ErrorsShow

是否显示错误， 默认显示错误信息

- XSRFKEY

XSRF的key信息， 默认值是`beegoxsrf`

- XSRFExpire

XSRF过期时间， 默认值是0

1. 路由设置
  1. RESTful路由
  2. 固定路由
  3. 正则路由
  4. 自定义方法及 RESTful 规则
  5. 自动匹配

## 路由设置

什么时候路由设置呢？前面介绍的MVC结构执行时，介绍过beego存在三种方式的路由，固定路由、正则路由、自动路由，接下来详细的讲解如何使用这三种路由。

### RESTful路由

在介绍这三种beego的路由实现之前先介绍RESTful，我们知道RESTful是一种目前API开发中广泛采用的形式，beego默认就是支持这样的请求方法，也就是用户Get请求就执行Get方法，Post请求就执行Post方法。因此默认的路由是这样RESTful的请求方式。

### 固定路由

固定路由也就是全匹配的路由，如下所示：

```
1. beego.Router("/", &controllers.MainController{})  
2. beego.Router("/admin", &admin.UserController{})  
3. beego.Router("/admin/index", &admin.ArticleController{})  
4. beego.Router("/admin/addpkg", &admin.AddController{})
```

如上所示的路由就是我们最常用的路由方式，一个固定的路由，一个控制器，然后根据用户请求方法不同请求控制器中对应的方法，典型的RESTful方式。

### 正则路由

为了用户更加方便的路由设置，beego 参考了 sinatra 的路由实现，支持多种方式的路由：

- beego.Router("api/:id([0-9]+)", &controllers.RController{})  
    自定义正则匹配 //匹配 /api/123 :id= 123
- beego.Router("/news/:all", &controllers.RController{})  
    全匹配方式 //匹配 /news/path/to/123.html :all= path/to/123.html
- beego.Router("/user/:username([w]+)", &controllers.RController{})  
    正则字符串匹配 //匹配 /user astaxie :username = astaxie
- beego.Router("/download/", &controllers.RController{})  
    \*匹配方式 //匹配 /download/file/api.xml :path= file/api .ext=xml
- beego.Router("/download/ceshi/\*", &controllers.RController{})  
    \*全匹配方式 //匹配 /download/ceshi/file/api.json :splat=file/api.json
- beego.Router("/:id:int", &controllers.RController{})  
    int 类型设置方式 //匹配 :id为int类型，框架帮你实现了正则([0-9]+)

- `beego.Router("/:hi:string", &controllers.RController{})`

`string` 类型设置方式 // 匹配 `:hi` 为 `string` 类型。框架帮你实现了正则(`[w]+`)

可以在 `Controller` 中通过如下方式获取上面的变量：

```
1. this.Ctx.Input.Param(":id")
2. this.Ctx.Input.Param(":username")
3. this.Ctx.Input.Param(":splat")
4. this.Ctx.Input.Param(":path")
5. this.Ctx.Input.Param(":ext")
```

## 自定义方法及 RESTful 规则

上面列举的是默认的请求方法名（请求的 `method` 和函数名一致，例如 `GET` 请求执行 `Get` 函数，`POST` 请求执行 `Post` 函数），如果用户期望自定义函数名，那么可以使用如下方式：

```
1. beego.Router("/",&IndexController{},":Index")
```

使用第三个参数，第三个参数就是用来设置对应 `method` 到函数名，定义如下

- \* 表示任意的 `method` 都执行该函数
- 使用 `httpmethod:funcname` 格式来展示
- 多个不同的格式使用 ; 分割
- 多个 `method` 对应同一个 `funcname`，`method` 之间通过 , 来分割

以下是一个 RESTful 的设计示例：

```
1. beego.Router("/api/list",&RestController{},":ListFood")
2. beego.Router("/api/create",&RestController{},"post:CreateFood")
3. beego.Router("/api/update",&RestController{},"put:UpdateFood")
4. beego.Router("/api/delete",&RestController{},"delete:DeleteFood")
```

以下是多个 HTTP Method 指向同一个函数的示例：

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

一下是不同的 `method` 对应不同的函数，通过 ; 进行分割的示例：

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

可用的 HTTP Method：

- \*：包含一下所有的函数
- get：GET 请求
- post：POST 请求
- put：PUT 请求
- delete：DELETE 请求
- patch：PATCH 请求
- options：OPTIONS 请求
- head：HEAD 请求

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

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

那么执行 `POST` 请求的时候，执行 `PostFunc` 而不执行 `AllFunc`。

## 自动匹配

用户首先需要把需要路由的控制器注册到自动路由中：

```
1. beego.AutoRouter(&controllers.ObjectController{})
```

那么 `beego` 就会通过反射获取该结构体中所有的实现方法，你就可以通过如下的方式访问到对应的方法中：

```
1. /object/login 调用 ObjectController 中的 Login 方法  
2. /object/logout 调用 ObjectController 中的 Logout 方法
```

除了前缀两个 `/:controller/:method` 的匹配之外，剩下的 url `beego` 会帮你自动化解析为参数，保存在 `this.Ctx.Input.Param` 当中：

```
1. /object/blog/2013/09/12 调用 ObjectController 中的 Blog 方法，参数如下: map[0:2013 1:09 2:12]
```

方法名在内部是保存了用户设置的，例如 `Login`，url 匹配的时候都会转化为小写，所以，`/object/LOGIN` 这样的 url 也一样可以路由到用户定义的 `Login` 方法中。

现在已经可以通过自动识别出来下面类似的所有url，都会把请求分发到 `controller` 的 `simple` 方法：

```
1. /controller/simple  
2. /controller/simple.html  
3. /controller/simple.json  
4. /controller/simple.rss
```

可以通过 `this.Ctx.Input.Param(":ext")` 获取后缀名

## 1. 控制器介绍

# 控制器介绍

基于 beego 的 Controller 设计，只需要匿名组合 `beego.Controller` 就可以了，如下所示：

```
1. type xxxController struct {
2.     beego.Controller
3. }
```

`beego.Controller` 实现了接口 `beego.ControllerInterface`，`beego.ControllerInterface` 定义了如下函数：

- `Init(ct *context.Context, childName string, app interface{})`

这个函数主要初始化了 Context、相应的 Controller 名称，模板名，初始化模板参数的容器 Data，app即为当前执行的 Controller的reflecttype，这个app可以用来执行子类的方法。

- `Prepare()`

这个函数主要是为了用户扩展用的，这个函数会在下面定义的这些 Method 方法之前执行，用户可以重写这个函数实现类似用户验证之类。

- `Get()`

如果用户请求的 HTTP Method 是 GET，那么就执行该函数，默认是 403，用户继承的子 struct 中可以实现了该方法以处理 Get 请求。

- `Post()`

如果用户请求的 HTTP Method 是 POST，那么就执行该函数，默认是 403，用户继承的子 struct 中可以实现了该方法以处理 Post 请求。

- `Delete()`

如果用户请求的 HTTP Method 是 DELETE，那么就执行该函数，默认是 403，用户继承的子 struct 中可以实现了该方法以处理 Delete 请求。

- `Put()`

如果用户请求的 HTTP Method 是 PUT，那么就执行该函数，默认是 403，用户继承的子 struct 中可以实现了该方法以处理 Put 请求。

- `Head()`

如果用户请求的 HTTP Method 是 HEAD，那么就执行该函数，默认是 403，用户继承的子 struct 中可以实现了该方法以处理 Head 请求。

- `Patch()`

如果用户请求的 HTTP Method 是 PATCH，那么就执行该函数，默认是 403，用户继承的子 struct 中可以实现了该方法以处理 Patch 请求。

- `Options()`

如果用户请求的 HTTP Method 是 OPTIONS，那么就执行该函数，默认是 403，用户继承的子 struct 中可以实现了该方法以处理 Options 请求。

- `Finish()`

这个函数实在执行完相应的 HTTP Method 方法之后执行的，默认是空，用户可以在子 struct 中重写这个函数，执行例如数据库关闭，清理数据之类的工作。

- `Render() error`

这个函数主要用来实现渲染模板，如果 `beego.AutoRender` 为 `true` 的情况下才会执行。

所以通过子 `struct` 的方法重写，用户就可以实现自己的逻辑，接下来我们看一个实际的例子：

```
1. type AddController struct {
2.     beego.Controller
3. }
4. func (this *AddController) Prepare() {
5. }
6. func (this *AddController) Get() {
7.     this.Data["content"] = "value"
8.     this.Layout = "admin/layout.html"
9.     this.TplNames = "admin/add.tpl"
10. }
11. func (this *AddController) Post() {
12.     pkgname := this.GetString("pkgname")
13.     content := this.GetString("content")
14.     pk := models.GetCruPkg(pkgname)
15.     if pk.Id == 0 {
16.         var pp models.PkgEntity
17.         pp.Pid = 0
18.         pp.Pathname = pkgname
19.         pp.Intro = pkgname
20.         models.InsertPkg(pp)
21.         pk = models.GetCruPkg(pkgname)
22.     }
23.     var at models.Article
24.     at.Pkgid = pk.Id
25.     at.Content = content
26.     models.InsertArticle(at)
27.     this.Ctx.Redirect(302, "/admin/index")
28. }
```

从上面的例子可以看出来，通过重写方法可以实现对应method的逻辑，实现RESTful结构的逻辑处理。

下面我们再来看一种比较流行的架构，首先实现一个自己的基类`baseController`，实现一些初始化的方法，然后其他所有的逻辑继承自该基类：

```
1. type NestPreparer interface {
2.     NestPrepare()
3. }
4.
5. // baseRouter implemented global settings for all other routers.
6. type baseRouter struct {
7.     beego.Controller
8.     i18n.Locale
9.     user    models.User
10.    isLoggedIn bool
11. }
12. // Prepare implemented Prepare method for baseRouter.
13. func (this *baseRouter) Prepare() {
```

```

14.
15.        // page start time
16.        this.Data["PageStartTime"] = time.Now()
17.
18.        // Setting properties.
19.        this.Data["AppDescription"] = utils.AppDescription
20.        this.Data["AppKeywords"] = utils.AppKeywords
21.        this.Data["AppName"] = utils.AppName
22.        this.Data["AppVer"] = utils.AppVer
23.        this.Data["AppUrl"] = utils.AppUrl
24.        this.Data["AppLogo"] = utils.AppLogo
25.        this.Data["AvatarURL"] = utils.AvatarURL
26.        this.Data["IsProMode"] = utils.IsProMode
27.
28.        if app, ok := this.AppController.(NestPreparer); ok {
29.            app.NestPrepare()
30.        }
31.    }

```

上面定义了基类，大概是初始化了一些变量，最后有一个Init函数中那个app的应用，判断当前运行的Controller是否是NestPreparer实现，如果是的话调用子类的方法，下面我们来看一下NestPreparer的实现：

```

1. type BaseAdminRouter struct {
2.     baseRouter
3. }
4.
5. func (this *BaseAdminRouter) NestPrepare() {
6.     if this.CheckActiveRedirect() {
7.         return
8.     }
9.
10.    // if user isn't admin, then logout user
11.    if !this.user.IsAdmin {
12.        models.LogoutUser(&this.Controller)
13.
14.        // write flash message
15.        this.FlashWrite("NotPermit", "true")
16.
17.        this.Redirect("/login", 302)
18.        return
19.    }
20.
21.    // current in admin page
22.    this.Data["IsAdmin"] = true
23.
24.    if app, ok := this.AppController.(ModelPreparer); ok {
25.        app.ModelPrepare()
26.        return
27.    }

```

```
28. }
29.
30. func (this *BaseAdminRouter) Get(){
31.     this.TplNames = "Get.tpl"
32. }
33.
34. func (this *BaseAdminRouter) Post(){
35.     this.TplNames = "Post.tpl"
36. }
```

这样我们的执行器执行的逻辑是这样的，首先执行Prepare，这个就是Go语言中struct中寻找方法的顺序，依次往父类寻找。执行 `BaseAdminRouter` 时，查找他是否有 `Prepare` 方法，没有就寻找 `baseRouter`，找到了，那么就执行逻辑，然后在 `baseRouter` 里面的 `this.AppController` 即为当前执行的控制器 `BaseAdminRouter`，因为会执行 `BaseAdminRouter.NestPrepare` 方法。然后开始执行相应的Get方法或者Post方法。

## 1. 跨站请求伪造

1. 在表单中使用
2. 在 JavaScript 中使用
3. 扩展 jQuery

## 跨站请求伪造

跨站请求伪造(Cross-site request forgery)，简称为 **XSRF**，是 Web 应用中常见的一个安全问题。前面的链接也详细讲述了 XSRF 攻击的实现方式。

当前防范 XSRF 的一种通用的方法，是对每一个用户都记录一个无法预知的 cookie 数据，然后要求所有提交的请求(POST/PUT/DELETE)中都必须带有这个 cookie 数据。如果此数据不匹配，那么这个请求就可能是被伪造的。

beego 有内建的 XSRF 的防范机制，要使用此机制，你需要在应用配置文件中加上 `enablexsrf` 设定：

```
1. enablexsrf = true
2. xsrfkey = 61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o
3. xsrfexpire = 3600
```

或者直接在 main 入口处这样设置：

```
1. beego.EnableXSRF = true
2. beego.XSRFKEY = "61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o"
3. beego.XSRFExpire = 3600 //过期时间，默认60秒
```

如果开启了 XSRF，那么 beego 的 Web 应用将对所有用户设置一个 `_xsrf` 的 cookie 值（默认过期 60 秒），如果 POST PUT DELETE 请求中没有这个 cookie 值，那么这个请求会被直接拒绝。如果你开启了这个机制，那么在所有被提交的表单中，你都需要加上一个域来提供这个值。你可以通过在模板中使用专门的函数 `XsrfFormHtml()` 来做到这一点：

过期时间上面我们设置了全局的过去时间 `beego.XSRFExpire`，但是有些时候我们也可以在控制器中修改这个过期时间，专门针对某一类处理逻辑：

```
1. func (this *HomeController) Get(){
2.     this.XSRFExpire = 7200
3.     this.Data["xsrfdata"] = template.HTML(this.XsrfFormHtml())
4. }
```

### 在表单中使用

在 Controller 中这样设置数据：

```
1. func (this *HomeController) Get(){
2.     this.Data["xsrfdata"] = template.HTML(this.XsrfFormHtml())
3. }
```

然后在模板中这样设置：

```
1. <form action="/new_message" method="post">
2.   {{ .xsrfdata }}
3.   <input type="text" name="message"/>
4.   <input type="submit" value="Post"/>
5. </form>
```

### 在 JavaScript 中使用

如果你提交的是 AJAX 的 POST 请求，你还是需要在每一个请求中通过脚本添加上 `_xsrf` 这个值。下面是在 AJAX 的 POST 请求，使用了 jQuery 函数来为所有请求组添加 `_xsrf` 值：

jQuery cookie插件: <https://github.com/carhartl/jquery-cookie> base64 插件:  
[http://phpjs.org/functions/base64\\_decode/](http://phpjs.org/functions/base64_decode/)

```
1.  jQuery.postJSON = function(url, args, callback) {
2.      var xsrf, xsrflist;
3.      xsrf = $.cookie("_xsrf");
4.      xsrflist = xsrf.split("|");
5.      args._xsrf = base64_decode(xsrflist[0]);
6.      $.ajax({url: url, data: $.param(args), dataType: "text", type: "POST",
7.          success: function(response) {
8.              callback(eval("(" + response + ")"));
9.          }});
10.     };

```

#### 扩展 jQuery

通过扩展 ajax 给每个请求加入 xsrf 的 header

需要你在 html 里保存一个 \_xsrf 值

```
1.  func (this *HomeController) Get(){
2.      this.Data["xsrf_token"] = this.XsrfToken()
3.  }
```

放在你的 head 中

```
1.  <head>
2.      <meta name="_xsrf" content="{{.xsrf_token}}>
3.  </head>
```

扩展 ajax 方法, 将 \_xsrf 值加入 header, 扩展后支持 jquery post/get 等内部使用了 ajax 的方法

```
1.  var ajax = $.ajax;
2.  $.extend({
3.      ajax: function(url, options) {
4.          if (typeof url === 'object') {
5.              options = url;
6.              url = undefined;
7.          }
8.          options = options || {};
9.          url = options.url;
10.         var xsrftoken = $('meta[name=_xsrf]').attr('content');
11.         var headers = options.headers || {};
12.         var domain = document.domain.replace(/^.+/, '');
13.         if (!/^((http|https):).*/.test(url) || eval('/^(http|https):\\//(.+\\.)*' + domain + '.*').test(
14.             headers = $.extend(headers, {'X-XSRFTOKEN':xsrftoken}));
15.         }
16.         options.headers = headers;
17.         return ajax(url, options);
18.     }
19. });

```

对于 PUT 和 DELETE 请求 (以及不使用将 form 内容作为参数的 POST 请求) 来说, 你也可以在 HTTP 头中以 X-XSRFToken 这个参数传递 CSRF token。

如果你需要针对每一个请求处理器定制 CSRF 行为, 你可以重写 Controller 的 CheckXsrfCookie 方法。例如你需要使用一个不支持 cookie 的 API, 你可以通过将 CheckXsrfCookie() 函数设空来禁用 CSRF 保护机制。然而如果 你需要同时支持 cookie 和非 cookie 认证方式, 那么只要当前请求是通过 cookie 进行认证的, 你就应该对其使用 CSRF 保护机制, 这一点至关重要。



## 1. session控制

### session控制

beego 内置了 session 模块，目前 session 模块支持的后端引擎包括 memory、file、mysql、redis 四种，用户也可以根据相应的 interface 实现自己的引擎。

beego 中使用 session 相当方便，只要在 main 入口函数中设置如下：

```
1. beego.SessionOn = true
```

或者通过配置文件配置如下：

```
1. sessionon = true
```

通过这种方式就可以开启 session，如何使用 session，请看下面的例子：

```
1. func (this *MainController) Get() {
2.     v := this.GetSession("asta")
3.     if v == nil {
4.         this.SetSession("asta", int(1))
5.         this.Data["num"] = 0
6.     } else {
7.         this.SetSession("asta", v.(int)+1)
8.         this.Data["num"] = v.(int)
9.     }
10.    this.TplNames = "index.tpl"
11. }
```

session 有几个方便的方法：

- SetSession(name string, value interface{})
- GetSession(name string) interface{}
- DelSession(name string)
- SessionRegenerateID()
- DestroySession()

session 操作主要有设置 session、获取 session、删除 session。

当然你要可以通过下面的方式自己控制相应的逻辑这些逻辑：

```
1. sess:=this.StartSession()
2. defer sess.SessionRelease()
```

sess对象具有如下方法：

- sess.Set()
- sess.Get()
- sess.Delete()
- sess.SessionID()
- sess.Flush()

但是我还是建议大家采用 `SetSession`、`GetSession`、`DelSession` 三个方法来操作，避免自己在操作的过程中资源没释放的问题。

关于 `Session` 模块使用中的一些参数设置：

- `SessionOn`

设置是否开启 `Session`，默认是 `false`，配置文件对应的参数名：`sessionon`。

- `SessionProvider`

设置 `Session` 的引擎，默认是 `memory`，目前支持还有 `file`、`mysql`、`redis` 等，配置文件对应的参数名：`sessionprovider`。

- `SessionName`

设置 `cookies` 的名字，`Session` 默认是保存在用户的浏览器 `cookies` 里面的，默认名是 `beegosessionID`，配置文件对应的参数名是：`sessionname`。

- `SessionGCMaxLifetime`

设置 `Session` 过期的时间，默认值是 3600 秒，配置文件对应的参数：`sessiongcmaxlifetime`。

- `SessionSavePath`

设置对应 `file`、`mysql`、`redis` 引擎的保存路径或者链接地址，默认值是空，配置文件对应的参数：`sessionsavepath`。

- `SessionHashFunc`

默认值为 `sha1`，采用 `sha1` 加密算法生产 `sessionid`

- `SessionHashKey`

默认的 `key` 是 `beegoserversessionkey`，建议用户使用的时候修改该参数

- `SessionCookieLifeTime`

设置 `cookie` 的过期时间，`cookie` 是用来存储保存在客户端的数据。

当 `SessionProvider` 为 `file` 时，`SessionSavePath` 是只保存文件的目录，如下所示：

```
1. beego.SessionProvider = "file"
2. beego.SessionSavePath = "./tmp"
```

当 `SessionProvider` 为 `mysql` 时，`SessionSavePath` 是链接地址，采用 `go-sql-driver`，如下所示：

```
1. beego.SessionProvider = "mysql"
2. beego.SessionSavePath = "username:password@protocol(address)/dbname?param=value"
```

当 `SessionProvider` 为 `redis` 时，`SessionSavePath` 是 `redis` 的链接地址，采用了 `redigo`，如下所示：

```
1. beego.SessionProvider = "redis"
2. beego.SessionSavePath = "127.0.0.1:6379"
```

## 1. flash数据

### flash数据

这个 flash 与 Adobe/Macromedia Flash 没有任何关系。它主要用于在两个逻辑间传递临时数据，flash 中存放的所有数据会在紧接着的下一个逻辑中调用后清除。一般用于传递提示和错误消息。它适合 Post/Redirect/Get 模式。下面看使用的例子：

```
1. // 显示设置信息
2. func (c *MainController) Get() {
3.     flash:=beego.ReadFromRequest(&c.Controller)
4.     if n,ok:=flash.Data["notice"];ok{
5.         //显示设置成功
6.         c.TplNames = "set_success.html"
7.     }else if n,ok=flash.Data["error"];ok{
8.         //显示错误
9.         c.TplNames = "set_error.html"
10.    }else{
11.        // 不然默认显示设置页面
12.        this.Data["list"]=GetInfo()
13.        c.TplNames = "setting_list.html"
14.    }
15. }
16.
17. // 处理设置信息
18. func (c *MainController) Post() {
19.     flash:=beego.NewFlash()
20.     setting:=Settings{}
21.     valid := Validation{}
22.     c.ParseForm(&setting)
23.     if b, err := valid.Valid(setting);err!=nil {
24.         flash.Error("Settings invalid!")
25.         flash.Store(&c.Controller)
26.         c.Redirect("/setting",302)
27.         return
28.     }else if b!=nil{
29.         flash.Error("validation err!")
30.         flash.Store(&c.Controller)
31.         c.Redirect("/setting",302)
32.         return
33.     }
34.     saveSetting(setting)
35.     flash.Notice("Settings saved!")
36.     flash.Store(&c.Controller)
37.     c.Redirect("/setting",302)
38. }
```

上面的代码执行的大概逻辑是这样的：

1. Get 方法执行，因为没有 flash 数据，所以显示设置页面。
2. 用户设置信息之后点击递交，执行 Post，然后初始化一个 flash，通过验证，验证出错或者验证不通过设置 flash 的错误，如果通过了就保存设置，然后设置 flash 成功设置的信息。
3. 设置完成后跳转到 Get 请求。
4. Get 请求获取到了 Flash 信息，然后执行相应的逻辑，如果出错显示出错的页面，如果成功显示成功的页面。

默认情况下 `ReadFromRequest` 函数已经实现了读取的数据赋值给 flash，所以在你的模板里面你可以这样读取数据：

```
1. {{.flash.error}}
2. {{.flash.warning}}
3. {{.flash.notice}}
```

flash 对象有三个级别的设置：

- Notice 提示信息
- Warning 警告信息
- Error 错误信息

1. 获取参数
  1. 直接解析到 struct
  2. 获取 Request Body 里的内容
  3. 文件上传

## 获取参数

我们经常需要获取用户传递的数据，包括 Get、POST 等方式的请求，beego 里面会自动解析这些数据，你可以通过如下方式获取数据：

- GetString(key string) string
- GetStrings(key string) []string
- GetInt(key string) (int64, error)
- GetBool(key string) (bool, error)
- GetFloat(key string) (float64, error)

使用例子如下：

```
1. func (this *MainController) Post() {
2.     jsoninfo := this.GetString("jsoninfo")
3.     if jsoninfo == "" {
4.         this.Ctx.WriteString("jsoninfo is empty")
5.         return
6.     }
7. }
```

如果你需要的数据可能是其他类型的，例如是 int 类型而不是 int64，那么你需要这样处理：

```
1. func (this *MainController) Post() {
2.     id := this.Input().Get("id")
3.     intid, err := strconv.Atoi(id)
4. }
```

更多其他的 request 的信息，用户可以通过 `this.Ctx.Request` 获取信息，关于该对象的属性和方法参考手册[Request](#)。

## 直接解析到 struct

如果要把表单里的内容赋值到一个 struct 里，除了用上面的方法一个一个获取再赋值外，Beego提供了通过另外一个更便捷的方式，就是通过 struct 的字段名或 tag 与表单字段对应直接解析到 struct。

定义struct：

```
1. type user struct {
2.     Id      int      `form:"-"`
3.     Name   interface{} `form:"username"`
4.     Age    int      `form:"age"`
5.     Email  string
6. }
```

表单：

```
1. <form id="user">
2.   名字: <input name="username" type="text" />
3.   年龄: <input name="age" type="text" />
4.   邮箱: <input name="Email" type="text" />
5.   <input type="submit" value="提交" />
6. </form>
```

Controller 里解析:

```
1. func (this *MainController) Post() {
2.   u := user{}
3.   if err := this.ParseForm(&u); err != nil {
4.     //handle error
5.   }
6. }
```

注意:

- StructTag form 的定义和 renderform方法 共用一个标签
- 定义 struct 时，字段名后如果有 form 这个 tag，则会以把 form 表单里的 name 和 tag 的名称一样的字段赋值给这个字段，否则就会把 form 表单里与字段名一样的表单内容赋值给这个字段。如上面例子中，会把表单中的 username 和 age 分别赋值给 user 里的 Name 和 Age 字段，而 Email 里的内容则会赋给 Email 这个字段。
- 调用 Controller ParseForm 这个方法的时候，传入的参数必须为一个 struct 的指针，否则对 struct 的赋值不会成功并返回 **xx must be a struct pointer** 的错误。
- 如果要忽略一个字段，有两种办法，一是：字段名小写开头，二是： form 标签的值设置为 -

## 获取 Request Body 里的内容

在 API 的开发中，我们经常会用到 **JSON** 或 **XML** 来作为数据交互的格式，如何在 beego 中获取 Request Body 里的 JSON 或 XML 的数据呢？

1. 在配置文件里设置 `copyrequestbody = true`

2. 在 Controller 中

```
func (this *ObjectController) Post() { var ob models.Object json.Unmarshal(this.Ctx.Input.RequestBody, &ob) objectid := models.AddOne(ob) this.Data["json"] = "{\"objectId\":\"" + objectid + "\"}" this.ServeJson() }
```

## 文件上传

在 beego 中你可以很容易的处理文件上传，就是别忘记在你的 form 表单中增加这个属性 `enctype="multipart/form-data"`，否者你的浏览器不会传输你的上传文件。

文件上传之后一般是放在系统的内存里面，如果文件的 size 大于设置的缓存内存大小，那么就放在临时文件中，默认的缓存内存是 64M，你可以通过如下来调整这个缓存内存大小：

```
1. beego.MaxMemory = 1<<22
```

或者在配置文件中通过如下设置：

```
1. maxmemory = 1<<22
```

Beego 提供了两个很方便的方法来处理文件上传：

- `GetFile(key string) (multipart.File, *multipart.FileHeader, error)`

该方法主要用于用户读取表单中的文件名 `the_file`，然后返回相应的信息，用户根据这些变量来处理文件上传：过滤、保存文件等。

- `SaveToFile(fromfile, tofile string) error`

该方法是在 `GetFile` 的基础上实现了快速保存的功能

保存的代码例子如下：

```
1. func (this *MainController) Post() {  
2.     this.SaveToFile("the_file", "/var/www/uploads/uploaded_file.txt")  
3. }
```

## JSON、XML、JSONP

beego 当初设计的时候就考虑了 API 功能的设计，而我们在设计 API 的时候经常是输出 JSON 或者 XML 数据，那么 beego 提供了这样的方式直接输出：

- JSON 数据直接输出：

```
go func (this *AddController) Get() { mystruct := { ... } this.Data["json"] = &mystruct  
this.ServeJson() }
```

调用 ServeJson 之后，会设置 `content-type` 为 `application/json`，然后同时把数据进行 JSON 序列化输出。

- XML 数据直接输出：

```
go func (this *AddController) Get() { mystruct := { ... } this.Data["xml"] = &mystruct  
this.ServeXml() }
```

调用 ServeXML 之后，会设置 `content-type` 为 `application/xml`，同时数据会进行 XML 序列化输出。

- jsonp 调用

```
go func (this *AddController) Get() { mystruct := { ... } this.Data["json"] = &mystruct  
this.ServeJsonp() }
```

调用 ServeJsonp 之后，会设置 `content-type` 为 `application/json`，然后同时把数据进行 JSON 序列化，然后根据请求的 `callback` 参数设置 jsonp 输出。

- 表单验证
  - 安装及测试
  - 示例
  - API 文档

## 表单验证

表单验证是用于数据验证和错误收集的模块。

### 安装及测试

安装:

```
1. go get github.com/astaxie/beego/validation
```

测试:

```
1. go test github.com/astaxie/beego/validation
```

### 示例

直接使用示例:

```
1. import (
2.     "github.com/astaxie/beego/validation"
3.     "log"
4. )
5.
6. type User struct {
7.     Name string
8.     Age int
9. }
10.
11. func main() {
12.     u := User{"man", 40}
13.     valid := validation.Validation{}
14.     valid.Required(u.Name, "name")
15.     valid.MaxValue(u.Name, 15, "nameMax")
16.     valid.Range(u.Age, 0, 18, "age")
17.
18.     if valid.HasErrors() {
19.         // 如果有错误信息, 证明验证没通过
20.         // 打印错误信息
21.         for _, err := range valid.Errors {
22.             log.Println(err.Key, err.Message)
23.         }
24.     }
25. }
```

```

24.     }
25.     // or use like this
26.     if v := valid.Max(u.Age, 140, "age"); !v.Ok {
27.         log.Println(v.Error.Key, v.Error.Message)
28.     }
29.     // 定制错误信息
30.     minAge := 18
31.     valid.Min(u.Age, minAge, "age").Message("少儿不宜! ")
32.     // 错误信息格式化
33.     valid.Min(u.Age, minAge, "age").Message("%d不禁", minAge)
34. }
```

通过 StructTag 使用示例:

```

1. import (
2.     "log"
3.     "strings"
4.
5.     "github.com/astaxie/beego/validation"
6. )
7.
8. // 验证函数写在 "valid" tag 的标签里
9. // 各个函数之间用分号 ";" 分隔, 分号后面可以有空格
10. // 参数用括号 "()" 括起来, 多个参数之间用逗号 "," 分开, 逗号后面可以有空格
11. // 正则函数(Match)的匹配模式用两斜杠 "/" 括起来
12. // 各个函数的结果的key值为字段名.验证函数名
13. type user struct {
14.     Id      int
15.     Name   string `valid:"Required;Match(/^Bee.*/)` // Name 不能为空并且以Bee开头
16.     Age    int     `valid:"Range(1, 140)` // 1 <= Age <= 140, 超出此范围即为不合法
17.     Email  string `valid:"Email; MaxSize(100)` // Email字段需要符合邮箱格式, 并且最大长度不能大于100个字符
18.     Mobile string `valid:"Mobile"` // Mobile必须为正确的手机号
19.     IP     string `valid:"IP"` // IP必须为一个正确的IPv4地址
20. }
21.
22. // 如果你的 struct 实现了接口 validation.ValidFormer
23. // 当 StructTag 中的测试都成功时, 将会执行 Valid 函数进行自定义验证
24. func (u *user) Valid(v *validation.Validation) {
25.     if strings.Index(u.Name, "admin") != -1 {
26.         // 通过 SetError 设置 Name 的错误信息, HasErrors 将会返回 true
27.         v.SetError("Name", "名称里不能含有 admin")
28.     }
29. }
30.
31. func main() {
32.     valid := validation.Validation{}
33.     u := user{Name: "Beego", Age: 2, Email: "dev@beego.me"}
34.     b, err := valid.Valid(u)
35.     if err != nil {
```

```
36.         // handle error
37.     }
38.     if !b {
39.         // validation does not pass
40.         // blabla...
41.         for _, err := range valid.Errors {
42.             log.Println(err.Key, err.Message)
43.         }
44.     }
45. }
```

StructTag 可用的验证函数:

- **Required** 不为空, 即各个类型要求不为其零值
- **Min(min int)** 最小值, 有效类型: `int`, 其他类型都将不能通过验证
- **Max(max int)** 最大值, 有效类型: `int`, 其他类型都将不能通过验证
- **Range(min, max int)** 数值的范围, 有效类型: `int`, 其他类型都将不能通过验证
- **MinSize(min int)** 最小长度, 有效类型: `string slice`, 其他类型都将不能通过验证
- **MaxSize(max int)** 最大长度, 有效类型: `string slice`, 其他类型都将不能通过验证
- **Length(length int)** 指定长度, 有效类型: `string slice`, 其他类型都将不能通过验证
- **Alpha** alpha字符, 有效类型: `string`, 其他类型都将不能通过验证
- **Numeric** 数字, 有效类型: `string`, 其他类型都将不能通过验证
- **AlphaNumeric** alpha字符或数字, 有效类型: `string`, 其他类型都将不能通过验证
- **Match(pattern string)** 正则匹配, 有效类型: `string`, 其他类型都将被转成字符串再匹配(`fmt.Sprintf("%v", obj).Match`)
- **AlphaDash** alpha字符或数字或横杠 `_`, 有效类型: `string`, 其他类型都将不能通过验证
- **Email** 邮箱格式, 有效类型: `string`, 其他类型都将不能通过验证
- **IP** IP格式, 目前只支持IPv4格式验证, 有效类型: `string`, 其他类型都将不能通过验证
- **Base64** base64编码, 有效类型: `string`, 其他类型都将不能通过验证
- **Mobile** 手机号, 有效类型: `string`, 其他类型都将不能通过验证
- **Tel** 固定电话号, 有效类型: `string`, 其他类型都将不能通过验证
- **Phone** 手机号或固定电话号, 有效类型: `string`, 其他类型都将不能通过验证
- **ZipCode** 邮政编码, 有效类型: `string`, 其他类型都将不能通过验证

## API 文档

请移步 [Go Walker](#)。

## 1. 错误处理

### 错误处理

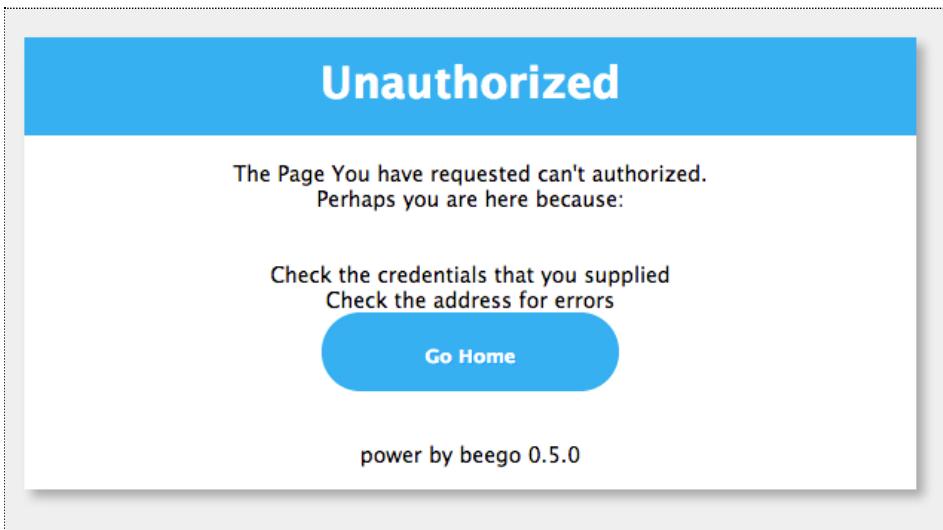
我们在做 Web 开发的时候，经常会遇到页面调整和错误处理，Beego 这这方面也进行了考虑，通过 `Redirect` 方法来进行跳转：

```
1. func (this *AddController) Get() {
2.     this.Redirect("/", 302)
3. }
```

如何中止此次请求并抛出异常，Beego 可以在控制器中这操作：

```
1. func (this *MainController) Get() {
2.     this.Abort("401")
3.     v := this.GetSession("asta")
4.     if v == nil<
/SPAN> {
5.         this.SetSession("asta", int(1))
6.         this.Data["Email"] = 0
7.     } else {
8.         this.SetSession("asta", v.(int)+1)
9.         this.Data["Email"] = v.(int)
10.    }
11.    this.TplNames = "index.tpl"
12. }
```

这样 `this.Abort("401")` 之后的代码不会再执行，而且会默认显示给用户如下页面：



Beego 框架默认支持 404、401、403、500、503 这几种错误的处理。用户可以自定义相应的错误处理，例如下面重新定义 404 页面：

```
1. func page_not_found(rw http.ResponseWriter, r *http.Request){
2.     t,_:= template.New("beegoerrortemp").ParseFiles(beego.ViewsPath+"/404.html")
```

```
3.     data :=make(map[string]interface{})
4.     data["content"] = "page not found"
5.     t.Execute(rw, data)
6. }
7.
8. func main() {
9.     beego.Errorhandler("404",page_not_found)
10.    beego.Router("/", &controllers.MainController{})
11.    beego.Run()
12. }
```

我们可以通过自定义错误页面 `404.html` 来处理404错误。

Beego 更加人性化的还有一个设计就是支持用户自定义字符串错误类型处理函数，例如下面的代码，用户注册了一个数据库出错的处理页面：

```
1. func dbError(rw http.ResponseWriter, r *http.Request){
2.     t,_:= template.New("beegoerrortemp").ParseFiles(beego.ViewsPath+"/dberror.html")
3.     data :=make(map[string]interface{})
4.     data["content"] = "database is now down"
5.     t.Execute(rw, data)
6. }
7.
8. func main() {
9.     beego.Errorhandler("dbError",dbError)
10.    beego.Router("/", &controllers.MainController{})
11.    beego.Run()
12. }
```

一旦在入口注册该错误处理代码，那么你可以在任何你的逻辑中遇到数据库错误调用 `this.Abort("dbError")` 来进行异常页面处理。

1. 日志处理
  1. 使用入门
  2. 设置输出
  3. 设置级别

## 日志处理

beego之前介绍的时候说过是基于几个模块搭建的，beego的日志处理是基于logs模块搭建的，内置了一个变量 `BeeLogger`，默认已经是 `logs.BeeLogger` 类型，初始化了 `console`，也就是默认输出到 `console`。

### 使用入门

一般在程序中我们使用如下的方式进行输出：

```
1. beego.Trace("this is trace")
2. beego.Debug("this is debug")
3. beego.Info("this is info")
4. beego.Warn("this is warn")
5. beego.Error(
    "this is error")
6. beego.Critical("this is critical")
```

### 设置输出

我们的程序往往期望把信息输出到log中，现在设置输出到文件很方便，如下所示：

```
1. beego.SetLogger("file", `{"filename":"logs/test.log"}`)
```

更多详细的日志配置请查看[日志配置](#)

这个默认情况就会同时输出到两个地方，一个 `console`，一个 `file`，如果只想输出到文件，就需要调用删除操作：

```
1. beego.BeeLogger.DelLogger("console")
```

### 设置级别

日志的级别如上所示的代码这样分为六个级别：

```
1. LevelTrace
2. LevelDebug
3. LevelInfo
4. LevelWarn
5. LevelError
6. LevelCritical
```

级别依次上升，默认全部打印，但是一般我们在部署环境，可以通过设置级别设置日志级别：

1. beego.SetLevel(beego.LevelInfo)

## 1. ORM 使用方法

- 1. models.go:
  - 1. RegisterDriver
  - 2. RegisterDataBase
  - 3. SetMaxIdleConns
  - 4. SetMaxOpenConns
  - 5. 时区设置
- 2. 注册模型:
  - 1. RegisterModel
  - 2. RegisterModelWithPrefix
- 3. ORM 接口使用:
  - 1. QueryTable
  - 2. Using
  - 3. Raw
  - 4. Driver
- 4. 调试模式打印查询语句

## ORM 使用方法

beego/orm 的使用例子

后文例子如无特殊说明都以这个为基础。

### models.go:

```
1. package main
2.
3. import (
4.     "github.com/astaxie/beego/orm"
5. )
6.
7. type User struct {
8.     Id        int
9.     Name      string
10.    Profile   *Profile `orm:"rel(one)"` // OneToOne relation
11. }
12.
13. type Profile struct {
14.     Id        int
15.     Age       int16
16.     User     *User   `orm:"reverse(one)"` // 设置反向关系(可选)
17. }
18.
19. func init() {
20.     // 需要在init中注册定义的model
21.     orm.RegisterModel(new(User), new(Profile))
22. }
```

### main.go

```
1. package main
2.
3. import (
4.     "fmt"
5.     "github.com/astaxie/beego/orm"
6.     _ "github.com/go-sql-driver/mysql"
7. )
```

```
8.
9.     func init() {
10.         orm.RegisterDriver("mysql", orm.DR_MySQL)
11.
12.         orm.RegisterDataBase("default", "mysql", "root:root@/orm_test?charset=utf8")
13.     }
14.
15.    func main() {
16.        o := orm.NewOrm()
17.        o.Using("default") // 默认使用 default, 你可以指定为其他数据库
18.
19.        profile := new(Profile)
20.        profile.Age = 30
21.
22.        user := new(User)
23.        user.Profile = profile
24.        user.Name = "slene"
25.
26.        fmt.Println(o.Insert(profile))
27.        fmt.Println(o.Insert(user))
28.    }
```

## 数据库的设置

目前 ORM 支持三种数据库，以下为测试过的 driver

将你需要使用的 driver 加入 import 中

```
1. import (
2.     _ "github.com/go-sql-driver/mysql"
3.     _ "github.com/lib/pq"
4.     _ "github.com/mattn/go-sqlite3"
5. )
```

### RegisterDriver

三种默认数据库类型

```
1. orm.DR_MySQL
2. orm.DR_Sqlite
3. orm.DR_Postgres
4.
5.
6.
7. // 参数1  driverName
8. // 参数2  数据库类型
9. // 这个用来设置 driverName 对应的数据库类型
10. // mysql / sqlite3 / postgres 这三种是默认已经注册过的，所以可以无需设置
11. orm.RegisterDriver("mymysql", orm.DR_MySQL)
```

### RegisterDataBase

ORM 必须注册一个别名为 `default` 的数据库，作为默认使用。

ORM 使用 golang 自己的连接池

```
1. // 参数1      数据库的别名，用来在ORM中切换数据库使用
```

```
2. // 参数2      driverName
3. // 参数3      对应的链接字符串
4. orm.RegisterDataBase("default", "mysql", "root:root@/orm_test?charset=utf8")
5.
6. // 参数4(可选) 设置最大空闲连接
7. // 参数5(可选) 设置最大数据库连接 (go >= 1.2)
8. maxIdle := 30
9. maxConn := 30
10. orm.RegisterDataBase("default", "mysql", "root:root@/orm_test?charset=utf8", maxIdle, maxConn)
```

### SetMaxIdleConns

根据数据库的别名，设置数据库的最大空闲连接

```
1. orm.SetMaxIdleConns("default", 30)
```

### SetMaxOpenConns

根据数据库的别名，设置数据库的最大数据库连接 (go >= 1.2)

```
1. orm.SetMaxOpenConns("default", 30)
```

### 时区设置

ORM 默认使用 time.Local 本地时区

- 作用于 ORM 自动创建的时间
- 从数据库中取回的时间转换成 ORM 本地时间

如果需要的话，你也可以进行更改

```
1. // 设置为 UTC 时间
2. orm.DefaultTimeLoc = time.UTC
```

ORM 在进行 RegisterDataBase 的同时，会获取数据库使用的时区，然后在 time.Time 类型存取的时做相应转换，以匹配时间系统，从而保证时间不会出错。

**注意:** 鉴于 Sqlite3 的设计，存取默认都为 UTC 时间

## 注册模型

如果使用 orm.QuerySeter 进行高级查询的话，这个是必须的。

反之，如果只使用 Raw 查询和 map struct，是无需这一步的。您可以去查看 [Raw SQL 查询](#)

### RegisterModel

将你定义的 Model 进行注册，最佳设计是有单独的 models.go 文件，在他的 init 函数中进行注册。

迷你版 models.go

```
1. package main
2.
3. import "github.com/astaxie/beego/orm"
4.
5. type User struct {
6.     Id    int
7.     name string
8. }
9.
10. func init(){
11.     orm.RegisterModel(new(User))
```

```
12. }
```

RegisterModel 也可以同时注册多个 model

```
1. orm.RegisterModel(new(User), new(Profile), new(Post))
```

详细的 struct 定义请查看文档 [模型定义](#)

#### RegisterModelWithPrefix

使用表名前缀

```
1. orm.RegisterModelWithPrefix("prefix_", new(User))
```

创建后的表名为 prefix\_user

## ORM 接口使用

使用 ORM 必然接触的 Ormer 接口，我们来熟悉一下

```
1. var o Ormer
2. o = orm.NewOrm() // 创建一个 Ormer
3. // NewOrm 的同时会执行 orm.BootStrap (整个 app 只执行一次)，用以验证模型之间的定义并缓存。
```

切换数据库，或者，进行事务处理，都会作用于这个 Ormer 对象，以及对其进行的任何查询。

所以：需要 切换数据库 和 事务处理 的话，不要使用全局保存的 Ormer 对象。

- type Ormer interface {
  - Read(interface{}, ...string) error
  - Insert(interface{}) (int64, error)
  - Update(interface{}, ...string) (int64, error)
  - Delete(interface{}) (int64, error)
  - LoadRelated(interface{}, string, ...interface{}) (int64, error)
  - QueryM2M(interface{}, string) QueryM2Mer
  - QueryTable(interface{}) QuerySeter
  - Using(string) error
  - Begin() error
  - Commit() error
  - Rollback() error
  - Raw(string, ...interface{}) RawSeter
  - Driver() Driver
- }

#### QueryTable

传入表名，或者 Model 对象，返回一个 QuerySeter

```
1. o := orm.NewOrm()
2. var qs QuerySeter
3. qs = o.QueryTable("user")
4. // 如果表没有定义过，会立刻 panic
```

#### Using

切换为其他数据库

```
1. orm.RegisterDataBase("db1", "mysql", "root:root@/orm_db2?charset=utf8")
2. orm.RegisterDataBase("db2", "sqlite3", "data.db")
3.
4. o1 := orm.NewOrm()
```

```
5.     o1.Using("db1")
6.
7.     o2 := orm.NewOrm()
8.     o2.Using("db2")
9.
10.    // 切换为其他数据库以后
11.    // 这个 Ormer 对象的其下的 api 调用都将使用这个数据库
```

默认使用 `default` 数据库，无需调用 `Using`

#### Raw

使用 `sql` 语句直接进行操作

`Raw` 函数，返回一个 `RawSeter` 用以对设置的 `sql` 语句和参数进行操作

```
1.     o := NewOrm()
2.     var r RawSeter
3.     r = o.Raw("UPDATE user SET name = ? WHERE name = ?", "testing", "slene")
```

#### Driver

返回当前 ORM 使用的 db 信息

```
1.     type Driver interface {
2.         Name() string
3.         Type() DriverType
4.     }
5.
6.
7.
8.     orm.RegisterDataBase("db1", "mysql", "root:root@/orm_db2?charset=utf8")
9.     orm.RegisterDataBase("db2", "sqlite3", "data.db")
10.
11.    o1 := orm.NewOrm()
12.    o1.Using("db1")
13.    dr := o1.Driver()
14.    fmt.Println(dr.Name() == "db1") // true
15.    fmt.Println(dr.Type() == orm.DR_MySQL) // true
16.
17.    o2 := orm.NewOrm()
18.    o2.Using("db2")
19.    dr = o2.Driver()
20.    fmt.Println(dr.Name() == "db2") // true
21.    fmt.Println(dr.Type() == orm.DR_Sqlite) // true
```

## 调试模式打印查询语句

简单的设置 `Debug` 为 `true` 打印查询的语句

可能存在性能问题，不建议使用在产品模式

```
1.     func main() {
2.         orm.Debug = true
3.         ...
```

默认使用 `os.Stderr` 输出日志信息

改变输出到你自己的 io.Writer

```
1. var w io.Writer
2. ...
3. // 设置为你的 io.Writer
4. ...
5. orm.DebugLog = orm.NewLog(w)
```

日志格式

```
1. [ORM] - 时间 - [Queries/数据库名] - [执行操作/执行时间] - [SQL语句] - 使用标点 `，` 分隔的参数列表 - 打印遇到的错误
2. [ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.Exec / 0.4ms] - [INSERT INTO `user` (`name`)
3. [ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.Exec / 0.5ms] - [UPDATE `user` SET `name` = ?
4. [ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.QueryRow / 0.4ms] - [SELECT `id`, `name` FROM `us
5. [ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.Exec / 0.4ms] - [INSERT INTO `post` (`user_id
6. [ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.Query / 0.4ms] - [SELECT T1.`name` `User__Name
7. [ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.Exec / 0.4ms] - [DELETE FROM `user` WHERE `ic
8. [ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.Query / 0.3ms] - [SELECT T0.`id` FROM `post` T0
9. [ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.Exec / 0.4ms] - [DELETE FROM `post` WHERE `ic
```

日志内容包括 所有的数据库操作, 事务, Prepare, 等

## 1. 对象的CRUD操作

- 1. Read
- 2. Insert
- 3. Update
- 4. Delete

# 对象的CRUD操作

如果已知主键的值，那么可以使用这些方法进行CRUD操作

对 object 操作的四个方法 Read / Insert / Update / Delete

```
1. o := orm.NewOrm()
2. user := new(User)
3. user.Name = "slene"
4.
5. fmt.Println(o.Insert(user))
6.
7. user.Name = "Your"
8. fmt.Println(o.Update(user))
9. fmt.Println(o.Read(user))
10. fmt.Println(o.Delete(user))
```

如果需要通过条件查询获取对象，请参见[高级查询](#)

## Read

```
1. o := orm.NewOrm()
2. user := User{Id: 1}
3.
4. err = o.Read(&user)
5.
6. if err == orm.ErrNoRows {
7.     fmt.Println("查询不到")
8. } else if err == orm.ErrMissPK {
9.     fmt.Println("找不到主键")
10. } else {
11.     fmt.Println(user.Id, user.Name)
12. }
```

[Read](#) 默认通过查询主键赋值，可以使用指定的字段进行查询：

```
1. user := User{Name: "slene"}
2. err = o.Read(&user, "Name")
3. ...
```

对象的其他字段值将会是对应类型的默认值

复杂的单个对象查询参见 [One](#)

## Insert

```
1. o := orm.NewOrm()
2. var user User
3. user.Name = "slene"
4. user.IsActive = true
5.
6. fmt.<
    /SPAN>Println(o.Insert(&user))
7. fmt.Println(user.Id)
```

创建后会自动对 `auto` 的 `field` 赋值

## Update

```
1. o := orm.NewOrm()
2. user := User{Id: 1}
3. if o.Read(&user) == nil {
4.     user.Name = "MyName"
5.     o.Update(&user)
6. }
```

`Update` 默认更新所有的字段，可以更新指定的字段：

```
1. // 只更新 Name
2. o.Update(&user, "Name")
3. // 指定多个字段
4. // o.Update(&user, "Field1", "Field2", ...)
5. ...
```

根据复杂条件更新字段值参见 [Update](#)

## Delete

```
1. o := orm.NewOrm()
2. o.Delete(&User{Id: 1})
```

`Delete` 操作会对反向关系进行操作，此例中 `Post` 拥有一个到 `User` 的外键。删除 `User` 的时候。如果 `on_delete` 设置为默认的级联操作，将删除对应的 `Post`

删除以后会清除 `auto field` 的值

## 高级查询

ORM 以 **QuerySeter** 来组织查询，每个返回 **QuerySeter** 的方法都会获得一个新的 **QuerySeter** 对象。

基本使用方法：

```
1. o := orm.NewOrm()
2.
3. // 获取 QuerySeter 对象, user 为表名
4. qs := o.QueryTable("user")
5.
6. // 也可以直接使用对象作为表名
7. user := new(User)
8. qs = o.QueryTable(user) // 返回 QuerySeter
```

## expr

QuerySeter 中用于描述字段和 sql 操作符，使用简单的 expr 查询方法

字段组合的前后顺序依照表的关系，比如 User 表拥有 Profile 的外键，那么对 User 表查询对应的 Profile.Age 为条件，则使用 **Profile\_\_Age** 注意，字段的分隔符号使用双下划线 **\_**，除了描述字段，expr 的尾部可以增加操作符以执行对应的 sql 操作。比如 **Profile\_\_Age\_\_gt** 代表 Profile.Age > 18 的条件查询。

注释后面将描述对应的 sql 语句，仅仅是描述 expr 的类似结果，并不代表实际生成的语句。

```
1. qs.Filter("id", 1) // WHERE id = 1
2. qs.Filter("profile__age", 18) // WHERE profile.age = 18
3. qs.Filter("Profile__Age", 18) // 使用字段名和Field名都是允许的
4. qs.Filter("profile__age", 18) // WHERE profile.age = 18
5. qs.Filter("profile__age__gt", 18) // WHERE profile.age > 18
6. qs.Filter("profile__age__gte", 18) // WHERE profile.age >= 18
7. qs.Filter("profile__age__in", 18, 20) // WHERE profile.age IN (18, 20)
8.
9. qs.Filter("profile__age__in", 18, 20).Exclude("profile__lt", 1000)
10. // WHERE profile.age IN (18, 20) AND NOT profile_id < 1000
```

## Operators

当前支持的操作符号：

- **exact / iexact** 等于
- **contains /icontains** 包含
- **gt / gte** 大于 / 大于等于
- **lt / lte** 小于 / 小于等于
- **startswith / istartswith** 以...起始
- **endswith / iendswith** 以...结束
- **in**
- **isnull**

后面以 `i` 开头的表示：大小写不敏感

## exact

Filter / Exclude / Condition expr 的默认值

```
qs.Filter("name", "slene") // WHERE name = 'slene'  
qs.Filter("name_exact", "slene") // WHERE name = 'slene'  
// 使用 = 匹配，大小写是否敏感取决于数据表使用的 collation  
qs.Filter("profile", nil) // WHERE profile_id IS NULL
```

## iexact

```
1. qs.Filter("name_iexact", "slene")  
2. // WHERE name LIKE 'slene'  
3. // 大小写不敏感，匹配任意 'Slene' 'sLENE'
```

## contains

```
1. qs.Filter("name_contains", "slene")  
2. // WHERE name LIKE BINARY '%slene%'  
3. // 大小写敏感，匹配包含 slene 的字符
```

## icontains

```
1. qs.Filter("name_icontains", "slene")  
2. // WHERE name LIKE '%slene%'  
3. // 大小写不敏感，匹配任意 'im Slene', 'im sLENE'
```

## in

```
1. qs.Filter("profile_age_in", 17, 18, 19, 20)  
2. // WHERE profile.age IN (17, 18, 19, 20)
```

## gt / gte

```
1. qs.Filter("profile_age_gt", 17)  
2. // WHERE profile.age > 17  
3.  
4. qs.Filter("profile_age_gte", 18)  
5. // WHERE profile.age >= 18
```

## lt / lte

```
1. qs.Filter("profile_age_lt", 17)  
2. // WHERE profile.age < 17  
3.  
4. qs.Filter("profile_age_lte", 18)  
5. // WHERE profile.age <= 18
```

## startswith

```
1. qs.Filter("name__startswith", "slene")
2. // WHERE name LIKE BINARY 'slene%'
3. // 大小写敏感，匹配以 'slene' 起始的字符串
```

### startswith

```
1. qs.Filter("name__startswith", "slene")
2. // WHERE name LIKE 'slene%'
3. // 大小写不敏感，匹配任意以 'slene', 'Slene' 起始的字符串
```

### endswith

```
1. qs.Filter("name__endswith", "slene")
2. // WHERE name LIKE BINARY '%slene'
3. // 大小写敏感，匹配以 'slene' 结束的字符串
```

### iendswith

```
1. qs.Filter("name__startswith", "slene")
2. // WHERE name LIKE '%slene'
3. // 大小写不敏感，匹配任意以 'slene', 'Slene' 结束的字符串
```

### isnull

```
1. qs.Filter("profile__isnull", true)
2. qs.Filter("profile_id__isnull", true)
3. // WHERE profile_id IS NULL
4.
5. qs.Filter("profile__isnull", false)
6. // WHERE profile_id IS NOT NULL
```

## 高级查询接口使用

QuerySeter 是高级查询使用的接口，我们来熟悉下他的接口方法

- type QuerySeter interface {
  - Filter(string, ...interface{}) QuerySeter
  - Exclude(string, ...interface{}) QuerySeter
  - SetCond(Condition) QuerySeter
  - Limit(int, ...int64) QuerySeter
  - Offset(int64) QuerySeter
  - OrderBy(...string) QuerySeter
  - RelatedSel(...interface{}) QuerySeter
  - Count() (int64, error)
  - Exist() bool
  - Update(Params) (int64, error)
  - Delete() (int64, error)
  - PrepareInsert() (Inserter, error)
  - All(interface{}, ...string) (int64, error)
  - One(interface{}, ...string) error

- `Values([]Params, ...string) (int64, error)`
  - `ValuesList([]ParamsList, ...string) (int64, error)`
  - `ValuesFlat(ParamsList, string) (int64, error)`
- }
  - 每个返回 `QuerySet` 的 api 调用时都会新建一个 `QuerySet`, 不影响之前创建的。
  - 高级查询使用 `Filter` 和 `Exclude` 来做常用的条件查询。囊括两种清晰的过滤规则: 包含, 排除

## Filter

用来过滤查询结果, 起到 包含条件 的作用

多个 `Filter` 之间使用 `AND` 连接

```
1. qs.Filter("profile__isnull", true).Filter("name", "slene")
2. // WHERE profile_id IS NULL AND name = 'slene'
```

## Exclude

用来过滤查询结果, 起到 排除条件 的作用

使用 `NOT` 排除条件

多个 `Exclude` 之间使用 `AND` 连接

```
1. qs.Exclude("profile__isnull", true).Filter("name", "slene")
2. // WHERE NOT profile_id IS NULL AND name = 'slene'
```

## SetCond

自定义条件表达式

```
1. cond := NewCondition()
2. cond1 := cond.And("profile__isnull", false).AndNot("status__in", 1).Or("profile__age__gt", 2000)
3.
4. qs := orm.QueryTable("user")
5. qs = qs.SetCond(cond1)
6. // WHERE ... AND ... AND NOT ... OR ...
7.
8. cond2 := cond.AndCond(cond1).OrCond(cond.And("name", "slene"))
9. qs = qs.SetCond(cond2).Count()
10. // WHERE (... AND ... AND NOT ... OR ...) OR (...)
```

## Limit

限制最大返回数据行数, 第二个参数可以设置 `Offset`

```
1. var DefaultRowsLimit = 1000 // ORM 默认的 limit 值为 1000
2.
3. // 默认情况下 select 查询的最大行数为 1000
4. // LIMIT 1000
5.
6. qs.Limit(10)
7. // LIMIT 10
8.
```

```
9.   qs.Limit(10, 20)
10.  // LIMIT 10 OFFSET 20
11.
12.  qs.Limit(-1)
13.  // no limit
14.
15.  qs.Limit(-1, 100)
16.  // LIMIT 18446744073709551615 OFFSET 100
17.  // 18446744073709551615 是 1<<64 - 1 用来指定无 limit 限制 但有 offset 偏移的情况
```

## Offset

设置 偏移行数

```
1.  qs.Offset(20)
2.  // LIMIT 1000 OFFSET 20
```

## OrderBy

参数使用 **expr**

在 **expr** 前使用减号 **-** 表示 **DESC** 的排列

```
1.  qs.OrderBy("id", "-profile__age")
2.  // ORDER BY id ASC, profile.age DESC
3.
4.  qs.OrderBy("-profile__age", "profile")
5.  // ORDER BY profile.age DESC, profile_id ASC
```

## RelatedSel

关系查询，参数使用 **expr**

```
1.  var DefaultRelsDepth = 5 // 默认情况下直接调用 RelatedSel 将进行最大 5 层的关系查询
2.
3.  qs := o.QueryTable("post")
4.
5.  qs.RelateSel()
6.  // INNER JOIN user ... LEFT OUTER JOIN profile ...
7.
8.  qs.RelateSel("user")
9.  // INNER JOIN user ...
10. // 设置 expr 只对设置的字段进行关系查询
11.
12. // 对设置 null 属性的 Field 将使用 LEFT OUTER JOIN
```

## Count

依据当前的查询条件，返回结果行数

```
1.  cnt, err := o.QueryTable("user").Count() // SELECT COUNT(*) FROM USER
2.  fmt.Printf("Count Num: %s, %s", cnt, err)
```

## Exist

```
1. exist := o.QueryTable("user").Filter("UserName", "Name").Exist()
2. fmt.Printf("Is Exist: %s", exist)
```

## Update

依据当前查询条件，进行批量更新操作

```
1. num, err := o.QueryTable("user").Filter("name", "slene").Update(orm.Params{
2.     "name": "astaxie",
3. })
4. fmt.Printf("Affected Num: %s, %s", num, err)
5. // SET name = "astaxie" WHERE name = "slene"
```

原子操作增加字段值

```
1. // 假设 user struct 里有一个 nums int 字段
2. num, err := o.QueryTable("user").Update(orm.Params{
3.     "nums": orm.ColValue(orm.Op_Add, 100),
4. })
5. // SET nums = nums + 1
```

orm.ColValue 支持以下操作

```
1. Col_Add      // 加
2. Col_Minus    // 减
3. Col_Multiply // 乘
4. Col_Except    // 除
```

## Delete

依据当前查询条件，进行批量删除操作

```
1. num, err := o.QueryTable("user").Filter("name", "slene").Delete()
2. fmt.Printf("Affected Num: %s, %s", num, err)
3. // DELETE FROM user WHERE name = "slene"
```

## PrepareInsert

用于一次 prepare 多次 insert 插入，以提高批量插入的速度。

```
1. var users []*User
2. ...
3. qs := o.QueryTable("user")
4. i, _ := qs.PrepareInsert()
5. for _, user := range users {
6.     id, err := i.Insert(user)
7.     if err != nil {
8.         ...
9.     }
}
```

```
10. }
11. // PREPARE INSERT INTO user (`name`, ...) VALUES (?, ...)
12. // EXECUTE INSERT INTO user (`name`, ...) VALUES ("slene", ...)
13. // EXECUTE ...
14. // ...
15. i.Close() // 别忘记关闭 statement
```

## All

返回对应的结果集对象

All 的参数支持 `[]Type` 和 `*[]Type` 两种形式的 slice

```
1. var users []*User
2. num, err := o.QueryTable("user").Filter("name", "slene").All(&users)
3. fmt.Printf("Returned Rows Num: %s, %s", num, err)
```

All / Values / ValuesList / ValuesFlat 受到 Limit 的限制，默认最大行数为 1000

可以指定返回的字段：

```
1. type Post struct {
2.     Id      int
3.     Title   string
4.     Content string
5.     Status  int
6. }
7.
8. // 只返回 Id 和 Title
9. var posts []Post
10. o.QueryTable("post").Filter("Status", 1).All(&posts, "Id", "Title")
```

对象的其他字段值将会是对应类型的默认值

## One

尝试返回单条记录

```
1. var user User
2. err := o.QueryTable("user").Filter("name", "slene").One(&user)
3. if err == orm.ErrMultiRows {
4.     // 多条的时候报错
5.     fmt.Printf("Returned Multi Rows Not One")
6. }
7. if err == orm.ErrNoRows {
8.     // 没有找到记录
9.     fmt.Printf("Not row found")
10. }
```

可以指定返回的字段：

```
1. // 只返回 Id 和 Title
```

```
2. var post Post
3. o.QueryTable("post").Filter("Content__istartswith", "prefix string").One(&post, "Id", "Title")
```

对象的其他字段值将会是对应类型的默认值

## Values

返回结果集的 key => value 值

key 为 Model 里的 Field name, value 的值以 string 保存

```
1. var maps []orm.Params
2. num, err := o.QueryTable("user").Values(&maps)
3. if err == nil {
4.     fmt.Printf("Result Num: %d\n", num)
5.     for _, m := range maps {
6.         fmt.Println(m["Id"], m["Name"])
7.     }
8. }
```

返回指定的 Field 数据

**TODO:** 暂不支持级联查询 **RelatedSel** 直接返回 Values

但可以直接指定 expr 级联返回需要的数据

```
1. var maps []orm.Params
2. num, err := o.QueryTable("user").Values(&maps, "id", "name", "profile", "profile__age")
3. if err == nil {
4.     fmt.Printf("Result Num: %d\n", num)
5.     for _, m := range maps {
6.         fmt.Println(m["Id"], m["Name"], m["Profile"], m["Profile__Age"])
7.         // map 中的数据都是展开的, 没有复杂的嵌套
8.     }
9. }
```

## ValuesList

顾名思义, 返回的结果集以slice存储

结果的排列与 Model 中定义的 Field 顺序一致

返回的每个元素值以 string 保存

```
1. var lists []orm.ParamsList
2. num, err := o.QueryTable("user").ValuesList(&lists)
3. if err == nil {
4.     fmt.Printf("Result Num: %d\n",
5.     num)
6.     for _, row := range lists {
7.         fmt.Println(row)
8.     }
9. }
```

当然也可以指定 `expr` 返回指定的 Field

```
1. var lists []orm.ParamsList
2. num, err := o.QueryTable("user").ValuesList(&lists, "name", "profile__age")
3. if err == nil {
4.     fmt.Printf("Result Nums: %d\n", num)
5.     for _, row := range lists {
6.         fmt.Printf("Name: %s, Age: %s\n", row[0], row[1])
7.     }
8. }
```

## ValuesFlat

只返回特定的 Field 值，讲结果集展开到单个 slice 里

```
1. var list orm.ParamsList
2. num, err := o.QueryTable("user").ValuesFlat(&list, "name")
3. if err == nil {
4.     fmt.Printf("Result Nums: %d\n", num)
5.     fmt.Printf("All User Names: %s", strings.Join(list, ", "))
6. }
```

## 关系查询

以例子里的模型定义来看下怎么进行关系查询

User 和 Profile 是 OneToOne 的关系

已经取得了 User 对象，查询 Profile：

```
1. user := &User{Id: 1}
2. o.Read(user)
3. if user.Profile != nil {
4.     o.Read(user.Profile)
5. }
```

直接关联查询：

```
1. user := &User{}
2. o.QueryTable("user").Filter("Id", 1).RelatedSel().One(user)
3. // 自动查询到 Profile
4. fmt.Println(user.Profile)
5. // 因为在 Profile 里定义了反向关系的 User，所以 Profile 里的 User 也是自动赋值过的，可以直接取用。
6. fmt.Println(user.Profile.User)
```

通过 User 反向查询 Profile：

```
1. var profile Profile
2. err := o.QueryTable("profile").Filter("User__Id", 1).One(&profile)
3. if err == nil {
```

```
4.     fmt.Println(profile)
5. }
```

#### Post 和 User 是 ManyToOne 关系，也就是 ForeignKey 为 User

```
1. type Post struct {
2.     Id      int
3.     Title   string
4.     User    *User  `orm:"rel(fk)"` // User 为外键
5.     Tags    []*Tag `orm:"rel(m2m)"`
6. }
7.
8.
9.
10. var posts []*Post
11. num, err := o.QueryTable("post").Filter("User", 1).RelatedSel().All(&posts)
12. if err == nil {
13.     fmt.Printf("%d posts read\n", num)
14.     for _, post := range posts {
15.         fmt.Printf("Id: %d, UserName: %d, Title: %s\n", post.Id, post.User.UserName, post.Title)
16.     }
17. }
```

根据 Post.Title 查询对应的 User:

RegisterModel 时，ORM 也会自动建立 User 中 Post 的反向关系，所以可以直接进行查询

```
1. var user User
2. err := o.QueryTable("user").Filter("Post__Title", "The Title").Limit(1).One(&user)
3. if err == nil {
4.     fmt.Printf(user)
5. }
```

#### Post 和 Tag 是 ManyToMany 关系

设置 rel(m2m) 以后，ORM 会自动创建中间表

```
1. type Post struct {
2.     Id      int
3.     Title   string
4.     User    *User  `orm:"rel(fk)"` // User 为外键
5.     Tags    []*Tag `orm:"rel(m2m)"`
6. }
7.
8.
9.
10. type Tag struct {
11.     Id      int
12.     Name   string
13.     Posts  []*Post `orm:"reverse(many)"` // 反向多对多
14. }
```

通过 tag name 查询哪些 post 使用了这个 tag

```
1. var posts []*Post
2. num, err := dORM.QueryTable("post").Filter("Tags__Tag__Name", "golang").All(&posts)
```

通过 post title 查询这个 post 有哪些 tag

```
1. var tags []*Tag
2. num, err := dORM.QueryTable("post").Filter("Tags__Post__Title", "Introduce Beego ORM").All(&tags)
```

## 载入关系字段

LoadRelated 用于载入模型的关系字段，包括所有的 rel/reverse - one/many 关系

ManyToMany 关系字段载入

```
1. // 载入相应的 Tags
2. post := Post{Id: 1}
3. err := o.Read(&post)
4. num, err := o.LoadRelated(&post, "Tags")
5.
6.
7.
8. // 载入相应的 Posts
9. tag := Tag{Id: 1}
10. err := o.Read(&tag)
11. num, err := o.LoadRelated(&tag, "Posts")
```

User 是 Post 的 ForeignKey，对应的 ReverseMany 关系字段载入

```
1. type User struct {
2.     Id     int
3.     Name   string
4.     Posts  []*Post `orm:"reverse(many)"` 
5. }
6.
7. user := User{Id: 1}
8. err := dORM.Read(&user)
9. num, err := dORM.LoadRelated(&user, "Posts")
10. for _, post := range user.Posts {
11.     //...
12. }
```

## 多对多关系操作

- type QueryM2Mer interface {
  - Add(...interface{}) (int64, error)
  - Remove(...interface{}) (int64, error)

- Exist(interface{}) bool
- Clear() (int64, error)
- Count() (int64, error)
- }

创建一个 QueryM2Mer 对象

```

1. o := orm.NewOrm()
2. post := Post{Id: 1}
3. m2m := o.QueryM2M(&post, "Tags")
4. // 第一个参数的对象，主键必须有值
5. // 第二个参数为对象需要操作的M2M字段
6. // QueryM2Mer 的 api 将作用于 Id 为 1 的 Post

```

### QueryM2Mer Add

```

1. tag := &Tag{Name: "golang"}
2. o.Insert(tag)
3.
4. num, err := m2m.Add(tag)
5. if err == nil {
6.     fmt.Println("Added nums: ", num)
7. }

```

Add 支持多种类型 Tag Tag []Tag []interface{}

```

1. var tags []*Tag
2. ...
3. // 读取 tags 以后
4. ...
5. num, err := m2m.Add(tags)
6. if err == nil {
7.     fmt.Println("Added nums: ", num)
8. }
9. // 也可以多个作为参数传入
10. // m2m.Add(tag1, tag2, tag3)

```

### QueryM2Mer Remove

从M2M关系中删除 tag

Remove 支持多种类型 Tag Tag []Tag []interface{}

```

1. var tags []*Tag
2. ...
3. // 读取 tags 以后
4. ...
5. num, err := m2m.Remove(tags)
6. if err == nil {
7.     fmt.Println("Removed nums: ", num)
8. }
9. // 也可以多个作为参数传入

```

```
10. // m2m.Remove(tag1, tag2, tag3)
```

### QueryM2Mer Exist

判断 Tag 是否存在于 M2M 关系中

```
1. if m2m.Exist(&Tag{Id: 2}) {  
2.     fmt.Println("Tag Exist")  
3. }
```

### QueryM2Mer Clear

清楚所有 M2M 关系

```
1. nums, err := m2m.Clear()  
2. if err == nil {  
3.     fmt.Println("Removed Tag Nums: ", nums)  
4. }
```

### QueryM2Mer Count

计算 Tag 的数量

```
1. nums, err := m2m.Count()  
2. if err == nil {  
3.     fmt.Println("Total Nums: ", nums)  
4. }
```

## 1. 使用SQL语句进行查询

1. Exec
2. QueryRow
3. QueryRows
4. SetArgs
5. Values / ValuesList / ValuesFlat
6. Values
7. ValuesList
8. ValuesFlat
9. Prepare

## 使用SQL语句进行查询

- 使用 Raw SQL 查询，无需使用 ORM 表定义
- 多数据库，都可直接使用占位符号 ?，自动转换
- 查询时的参数，支持使用 Model Struct 和 Slice, Array

```
ids := []int{1, 2, 3} p.Raw("SELECT name FROM user WHERE id IN (?, ?, ?)", ids)
```

创建一个 **RawSeter**

```
1. o := NewOrm()
2. var r RawSeter
3. r = o.Raw("UPDATE user SET name = ? WHERE name = ?", "testing", "slene")
```

- type RawSeter interface {
  - Exec() (sql.Result, error)
  - QueryRow(...interface{}) error
  - QueryRows(...interface{}) (int64, error)
  - SetArgs(...interface{}) RawSeter
  - Values([]Params) (int64, error)
  - ValuesList([]ParamsList) (int64, error)
  - ValuesFlat(\*ParamsList) (int64, error)
  - Prepare() (RawPreparer, error)
- }

### Exec

执行sql语句，返回 `sql.Result` 对象

```
1. res, err := o.Raw("UPDATE user SET name = ?", "your").Exec()
2. if err == nil {
3.     num, _ := res.RowsAffected()
4.     fmt.Println("mysql row affected nums: ", num)
5. }
```

### QueryRow

QueryRow 和 QueryRows 提供高级 sql mapper 功能

支持 struct

```
1. type User struct {
2.     Id    int
3.     Name string
4. }
```

```
5.  
6.     var user User  
7.     err := o.Raw("SELECT id, name FROM user WHERE id = ?", 1).QueryRow(&user)
```

支持同时 map 多个对象

```
1. type User struct {  
2.     Id      int  
3.     Skip    string `orm:"-"`  
4.     SkipYet int  
5.     Name    string  
6. }  
7.  
8. var user User  
9. var age int  
10. var created time.Time  
11. o.Raw("SELECT id, NULL, name, age, created FROM user WHERE id = ?", 1).QueryRow(&user, &age, &created)
```

struct 进行 mapper 的时候，可以进行忽略字段，如果有需要临时忽略的 Field 可以在 SELECT 中使用 NULL

支持判断 NULL 对象

```
1. type User struct {  
2.     Id      int  
3.     Name   string  
4. }  
5.  
6. type Profile struct {  
7.     Id    int  
8.     Age   int  
9. }  
10.  
11. var user *User  
12. var profile *Profile  
13. err := o.Raw(`SELECT id, name, p.id, p.age FROM user  
14.     LEFT OUTER JOIN profile AS p ON p.id = profile_id WHERE id = ?`, 1).QueryRow(&user, &profile)  
15. if err == nil {  
16.     if profile == nil {  
17.         fmt.Println("user's profile is empty")  
18.     }  
19. }
```

## QueryRows

QueryRows 支持的对象还有 map 规则是和 QueryRow 一样的，但都是 slice

```
1. type User struct {  
2.     Id      int  
3.     Name   string  
4. }  
5.  
6. var users []User
```

```

7.     num, err := o.Raw("SELECT id, name FROM user WHERE id = ?", 1).QueryRows(&users)
8.     if err == nil {
9.         fmt.Println("user nums: ", num)
10.    }

```

查询多个对象

```

1. type Profile struct {
2.     Id   int
3.     Age  int
4. }
5.
6. var users []*User
7. var profiles []*Profile
8. err := o.Raw(`SELECT id, name, p.id, p.age FROM user
9.      LEFT OUTER JOIN profile AS p ON p.id = profile_id WHERE id = ?`, 1).QueryRows(&users, &profiles)
10. if err == nil {
11.     for i, user := range users {
12.         profile := users[i]
13.         if profile == nil {
14.             fmt.Println("user's profile is empty")
15.         }
16.     }
17. }

```

### SetArgs

改变 Raw(sql, args...) 中的 args 参数，返回一个新的 RawSeter

用于单条 sql 语句，重复利用，替换参数然后执行。

```

1. res, err := r.SetArgs("arg1", "arg2").Exec()
2. res, err := r.SetArgs("arg1", "arg2").Exec()
3. ...

```

### Values / ValuesList / ValuesFlat

Raw SQL 查询获得的结果集 Value 为 `string` 类型，NULL 字段的值为空 ``

#### Values

返回结果集的 key => value 值

```

1. var maps []orm.Params
2. num, err = o.Raw("SELECT user_name FROM user WHERE status = ?", 1).Values(&maps)
3. if err == nil && num > 0 {
4.     fmt.Println(maps[0]["user_name"]) // slene
5. }

```

#### ValuesList

返回结果集 slice

```

1. var lists []orm.ParamsList

```

```
2. num, err = o.Raw("SELECT user_name FROM user WHERE status = ?", 1).ValuesList(&lists)
3. if err == nil && num > 0 {
4.     fmt.Println(lists[0][0]) // slene
5. }
```

### ValuesFlat

返回单一字段的平铺 slice 数据

```
1. var list orm.ParamsList
2. num, err = o.Raw("SELECT id FROM user WHERE id < ?", 10).ValuesList(&list)
3. if err == nil && num > 0 {
4.     fmt.Println(list) // []{"1","2","3",...}
5. }
```

### Prepare

用于一次 prepare 多次 exec，以提高批量执行的速度。

```
1. p, err := o.Raw("UPDATE user SET name = ? WHERE name = ?").Prepare()
2. res, err := p.Exec("testing", "slene")
3. res, err = p.Exec("testing", "astaxie")
4. ...
5. ...
6. p.Close() // 别忘记关闭 statement
```

## 1. 事务处理

### 事务处理

ORM 可以简单的进行事务操作

```
1. o := NewOrm()
2. err := o.Begin()
3. // 事务处理过程
4. ...
5. ...
6. // 此过程中的所有使用 o Ormer 对象的查询都在事务处理范围内
7. if SomeError {
8.     err = o.Rollback()
9. } else {
10.    err = o.Commit()
11. }
```

1. 模型定义
  1. 自定义表名
  2. 自定义索引
  3. 自定义引擎
  4. 设置参数
    1. 忽略字段
    2. auto
    3. pk
    4. null
    5. index
    6. unique
    7. column
    8. size
    9. digits / decimals
  10. autonow / autonow\_add
  11. type
  12. default
5. 表关系设置
  1. rel / reverse
  2. reltable / relthrough
  3. on\_delete
  4. 关于 on\_delete 的相关例子
6. 模型字段与数据库类型的对应
  1. MySQL
  2. Sqlite3
  3. PostgreSQL
7. 关系型字段

## 模型定义

复杂的模型定义不是必须的，此功能用作数据库数据转换和[自动建表](#)

默认的表名规则，使用驼峰转蛇形：

```
1. AuthUser -> auth_user
2. Auth_User -> auth__user
3. DB_AuthUser -> d_b__auth_user
```

除了开头的大写字母以外，遇到大写会增加 `_`，原名称中的下划线保留。

## 自定义表名

```
1. type User struct {
2.     Id int
3.     Name string
4. }
5.
6. func (u *User) TableName() string {
7.     return "auth_user"
8. }
```

如果前缀设置为 `prefix_` 那么表名为：prefixauthuser

## 自定义索引

为单个或多个字段增加索引

```
1. type User struct {
2.     Id    int
3.     Name  string
4.     Email string
5. }
6.
7. // 多字段索引
8. func (u *User) TableIndex() [][]string {
9.     return [][]string{
10.         []string{"Id", "Name"},
11.     }
12. }
13.
14. // 多字段唯一键
15. func (u *User) TableUnique() [][]string {
16.     return [][]string{
17.         []string{"Name", "Email"},
18.     }
19. }
```

## 自定义引擎

仅支持 MySQL

默认使用的引擎，为当前数据库的默认引擎，这个是由你的 mysql 配置参数决定的。

你可以在模型里设置 TableEngine 函数，指定使用的引擎

```
1. type User struct {
2.     Id    int
3.     Name  string
4.     Email string
5. }
6.
7. // 设置引擎为 INNODB
8. func (u *User) TableEngine() string {
9.     return "INNODB"
10. }
```

## 设置参数

```
1. orm:"null;rel(fk)"
```

多个设置间使用 ; 分隔，设置的值如果是多个，使用 , 分隔。

#### 忽略字段

设置 - 即可忽略 struct 中的字段

```
1. type User struct {
2.     ...
3.     AnyField string `orm:"-"`  
4.     ...
5. }
```

#### auto

当 Field 类型为 int, int32, int64, uint, uint32, uint64 时，可以设置字段为自增键

- 当模型定义里没有主键时，符合上述类型且名称为 **Id** 的 Field 将被视为自增键。

鉴于 go 目前的设计，即使使用了 uint64，但你也不能存储到他的最大值。依然会作为 int64 处理。

参见 [issue 6113](#)

#### pk

设置为主键，适用于自定义其他类型为主键

#### null

数据库表默认为 NOT NULL，设置 null 代表 ALLOW NULL

```
1. Name string `orm:"null"`
```

#### index

为单个字段增加索引

#### unique

为单个字段增加 unique 键

```
1. Name string `orm:"unique"`
```

#### column

为字段设置 db 字段的名称

```
1. Name string `orm:"column(user_name)"`
```

#### size

string 类型字段默认为 varchar(255)

设置 size 以后，db type 将使用 varchar(size)

```
1. Title string `orm:"size(60)"`
```

#### digits / decimals

设置 float32, float64 类型的浮点精度

```
1. Money float64 `orm:"digits(12);decimals(4)"`
```

总长度 12 小数点后 4 位 eg: 99999999.9999

#### autonow / autonow\_add

```
1. Created time.Time `orm:"auto_now_add;type(datetime)"`  
2. Updated time.Time `orm:"auto_now;type(datetime)"`
```

- auto\_now 每次 model 保存时都会对时间自动更新
- autonowadd 第一次保存时才设置时间

对于批量的 update 此设置是不生效的

#### type

设置为 date 时，time.Time 字段的对应 db 类型使用 date

```
1. Created time.Time `orm:"auto_now_add;type(date)"`
```

设置为 datetime 时，time.Time 字段的对应 db 类型使用 datetime

```
1. Created time.Time `orm:"auto_now_add;type(datetime)"`
```

#### default

为字段设置默认值，类型必须符合（目前仅用于级联删除时的默认值）

```
1. type User struct {  
2.     ...  
3.     Status int `orm:"default(1)"`  
4.     ...  
5. }
```

## 表关系设置

#### rel / reverse

##### RelOneToOne:

```
1. type User struct {  
2.     ...  
3.     Profile *Profile `orm:"null;rel(one);on_delete(set_null)"`  
4.     ...  
5. }
```

对应的反向关系 RelReverseOne:

```
1. type Profile struct {  
2.     ...  
3.     User *User `orm:"reverse(one)"`  
4.     ...  
5. }
```

##### RelForeignKey:

```
1. type Post struct {
2.     ...
3.     User *User `orm:"rel(fk)"` // RelForeignKey relation
4.     ...
5. }
```

对应的反向关系 **RelReverseMany**:

```
1. type User struct {
2.     ...
3.     Posts []*Post `orm:"reverse(many)"` // fk 的反向关系
4.     ...
5. }
```

**RelManyToMany**:

```
1. type Post struct {
2.     ...
3.     Tags []*Tag `orm:"rel(m2m)"` // ManyToMany relation
4.     ...
5. }
```

对应的反向关系 **RelReverseMany**:

```
1. type Tag struct {
2.     ...
3.     Posts []*Post `orm:"reverse(many)"` 
4.     ...
5. }
```

### reltable / relthrough

此设置针对 `orm:"rel(m2m)"` 的关系字段

1.	<code>rel_table</code>	设置自动生成的 m2m 关系表的名称
2.	<code>rel_through</code>	如果要在 m2m 关系中使用自定义的 m2m 关系表
3.		通过这个设置其名称，格式为 <code>pkg.path.ModelName</code>
4.		eg: <code>app.models.PostTagRel</code>
5.		<code>PostTagRel</code> 表需要有到 <code>Post</code> 和 <code>Tag</code> 的关系

当设置 `reltable` 时会忽略 `relthrough`

设置方法:

```
orm:"rel(m2m);rel_table(the_table_name)"  
orm:"rel(m2m);rel_through(pkg.path.ModelName)"
```

### on\_delete

设置对应的 rel 关系删除时，如何处理关系字段。

1.	<code>cascade</code>	级联删除(默认值)
----	----------------------	-----------

```

2.     set_null      设置为 NULL, 需要设置 null = true
3.     set_default   设置为默认值, 需要设置 default 值
4.     do_nothing    什么也不做, 忽略
5.
6.
7.     type User struct {
8.         ...
9.         Profile *Profile `orm:"null;rel(one);on_delete(set_null)"` // 删除 Profile 时将设置 User.Profile 的数据库字段为 NULL
10.        ...
11.    }
12.    type Profile struct {
13.        ...
14.        User *User `orm:"reverse(one)"` // 反向引用
15.        ...
16.    }
17.
18. // 删

```

#### 关于 on\_delete 的相关例子

```

1.     type User struct {
2.         Id int
3.         Name string
4.     }
5.
6.     type Post struct {
7.         Id int
8.         Title string
9.         User *User `orm:"rel(fk)"` // 外键
10.    }

```

假设 Post->User 是 ManyToOne 的关系，也就是外键。

```
1.     o.Filter("Id", 1).Delete()
```

这个时候即会删除 Id 为 1 的 User 也会删除其发布的 Post

不想删除的话，需要设置 set\_null

```

1.     type Post struct {
2.         Id int
3.         Title string
4.         User *User `orm:"rel(fk);null;on_delete(set_null)"` // 将设置 User 的 Post 数组为 nil
5.     }

```

那这个时候，删除 User 只会把对应的 Post.user\_id 设置为 NULL

当然有时候为了高性能的需要，多存点数据无所谓啊，造成批量删除才是问题。

```
1.     type Post struct {
2.         Id int
```

```
3.     Title string
4.     User *User `orm:"rel(fk);null;on_delete(do_nothing)"`  
5. }
```

那么只要删除的时候，不操作 Post 就可以了。

## 模型字段与数据库类型的对应

在此列出 ORM 推荐的对应数据库类型，自动建表功能也会以此为标准。

默认所有的字段都是 **NOT NULL**

### MySQL

I go I mysql I :--- I int, int32 - 设置 auto 或者名称为 **Id** 时 I integer AUTOINCREMENT I int64 - 设置 auto 或者名称为 **Id** 时 I bigint AUTOINCREMENT I uint, uint32 - 设置 auto 或者名称为 **Id** 时 I integer unsigned AUTOINCREMENT I uint64 - 设置 auto 或者名称为 **Id** 时 I bigint unsigned AUTOINCREMENT I bool I bool I string - 默认为 size 255 I varchar(size) I string - 设置 type(text) 时 I longtext I time.Time - 设置 type 为 date 时 I date I time.Time I datetime I byte I tinyint unsigned I rune I integer I int I integer I int8 I tinyint I int16 I smallint I int32 I integer I int64 I bigint I uint I integer unsigned I uint8 I tinyint unsigned I uint16 I smallint unsigned I uint32 I integer unsigned I uint64 I bigint unsigned I float32 I double precision I float64 I double precision I float64 - 设置 digits, decimals 时 I numeric(digits, decimals)

### Sqlite3

I go I sqlite3 I :--- I :--- I int, int32, int64, uint, uint32, uint64 - 设置 auto 或者名称为 **Id** 时 I integer AUTOINCREMENT I bool I bool I string - 默认为 size 255 I varchar(size) I string - 设置 type(text) 时 I text I time.Time - 设置 type 为 date 时 I date I time.Time I datetime I byte I tinyint unsigned I rune I integer I int I integer I int8 I tinyint I int16 I smallint I int32 I integer I int64 I bigint I uint I integer unsigned I uint8 I tinyint unsigned I uint16 I smallint unsigned I uint32 I integer unsigned I uint64 I bigint unsigned I float32 I real I float64 - 设置 digits, decimals 时 I decimal

### PostgreSQL

I go I postgres I :--- I :--- I int, int32, int64, uint, uint32, uint64 - 设置 auto 或者名称为 **Id** 时 I serial I bool I bool I string - 默认为 size 255 I varchar(size) I string - 设置 type(text) 时 I text I time.Time - 设置 type 为 date 时 I date I time.Time I timestamp with time zone I byte I smallint CHECK("column" >= 0 AND "column" <= 255) I rune I integer I int I integer I int8 I smallint CHECK("column" >= -127 AND "column" <= 128) I int16 I smallint I int32 I integer I int64 I bigint I uint I bigint CHECK("column" >= 0) I uint8 I smallint CHECK("column" >= 0 AND "column" <= 255) I uint16 I integer CHECK("column" >= 0) I uint32 I bigint CHECK("column" >= 0) I uint64 I bigint CHECK("column" >= 0) I float32 I double precision I float64 I double precision I float64 - 设置 digits, decimals 时 I numeric(digits, decimals)

## 关系型字段

其字段类型取决于对应的主键。

- RelForeignKey
- RelOneToOne
- RelManyToMany
- RelReverseOne
- RelReverseMany

1. 命令模式
  1. 自动建表
  2. 打印建表SQL

## 命令模式

注册模型与数据库以后，调用 RunCommand 执行 orm 命令。

```
1. func main() {  
2.     // orm.RegisterModel...  
3.     // orm.RegisterDataBase...  
4.     ...  
5.     orm.RunCommand()  
6. }  
7.  
8.  
9.  
10. go build main.go  
11. ./main orm  
12. # 直接执行可以显示帮助  
13. # 如果你的程序可以支持的话，直接运行 go run main.go orm 也是同样的效果
```

## 自动建表

```
1. ./main orm syncdb -h  
2. Usage of orm command: syncdb:  
3.   -db="default": DataBase alias name  
4.   -force=false: drop tables before create  
5.   -v=false: verbose info
```

使用 **-force=1** 可以 drop table 后再建表

使用 **-v** 可以查看执行的 sql 语句

在程序中直接调用自动建表：

```
1. // 数据库别名  
2. name := "default"  
3.  
4. // drop table 后再建表  
5. force := true  
6.  
7. // 打印执行过程  
8. verbose := true  
9.  
10. // 遇到错误立即返回
```

```
11.     err := orm.RunSyncdb(name, force, verbose)
12.     if err != nil {
13.         fmt.Println(err)
14.     }
```

自动建表功能在非 `force` 模式下，是会自动创建新增加的字段的。也会创建新增加的索引。

对于改动过的旧字段，旧索引，需要用户自行进行处理。

## 打印建表SQL

```
1. ./main orm sqlall -h
2. Usage of orm command: syncdb:
3. -db="default": DataBase alias name
```

默认使用别名为 `default` 的数据库。

## 1. ORM Test

- 1. MySQL
- 2. Sqlite3
- 3. PostgreSQL

# ORM Test

测试代码参见

- 表定义 `models_test.go`
- 测试用例 `orm_test.go`

### MySQL

```
mysql -u root -e 'create database orm_test;'  
export ORM_DRIVER=mysql  
export ORM_SOURCE="root:@/orm_test?charset=utf8"  
go test -v github.com/astaxie/beego/orm
```

### Sqlite3

```
touch /path/to/orm_test.db  
export ORM_DRIVER=sqlite3  
export ORM_SOURCE=/path/to/orm_test.db  
go test -v github.com/astaxie/beego/orm
```

### PostgreSQL

```
psql -c 'create database orm_test;' -U postgres  
export ORM_DRIVER=postgres  
export ORM_SOURCE="user=postgres dbname=orm_test sslmode=disable"  
go test -v github.com/astaxie/beego/orm
```

1. 模板处理
  1. 模板目录
  2. 自动渲染
  3. 模板标签
  4. 模板数据
  5. 模板名称
  6. Layout 设计
  7. renderform 使用

## 模板处理

beego的模板处理引擎采用的是Go内置的 `html/template` 包进行处理，而且beego的模板处理逻辑是采用了缓存编译方式，也就是所有的模板会在beego应用启动的时候全部编译然后缓存在map里面。

### 模板目录

beego 中默认的模板目录是 `views`，用户可以把模板文件放到该目录下，beego 会自动在该目录下的所有模板文件进行解析并缓存，开发模式下每次都会重新解析，不做缓存。当然，用户也可以通过如下的方式改变模板的目录（只能指定一个目录为模板目录）：

```
1. beego.ViewsPath = "myviewpath"
```

### 自动渲染

用户无需手动的调用渲染输出模板，beego 会自动的在调用完相应的 `method` 方法之后调用 `Render` 函数，当然如果您的应用是不需要模板输出的，那么可以在配置文件或者在 `main.go` 中设置关闭自动渲染。

配置文件配置如下：

```
1. autorender = false
```

`main.go` 文件中设置如下：

```
1. beego.AutoRender = false
```

### 模板标签

Go语言的默认模板采用了 `{{ 和 }}` 作为左右标签，但是我们有时候在开发中可能界面是采用了angularJS开发，他的模板也是这个标签，故而引起了冲突，在beego中你可以通过配置文件或者直接设置配置变量修改：

```
1. beego.TemplateLeft = "<<<"  
2. beego.TemplateRight = ">>>"
```

### 模板数据

模板中的数据是通过在 Controller 中 `this.Data` 获取的，所以如果你想在模板中获取内容 `{{.Content}}`，那么你需要在 Controller 中如下设置：

```
1. this.Data["Content"] = "value"
```

如何使用各种类型的数据渲染:

- 结构体

结构体结构

```
1. type A struct{  
2.     Name string  
3.     Age  int  
4. }
```

控制器数据赋值

```
1. this.Data["a"]=&A{Name:"astaxie",Age:25}
```

模板渲染数据如下:

```
1. the username is {{.a.Name}}  
2. the age is {{.a.Age}}
```

- map

控制器数据赋值

```
1. mp["name"]="astaxie"  
2. mp["nickname"] = "haha"  
3. this.Data["m"]=mp
```

模板渲染数据如下:

```
1. the username is {{.m.name}}  
2. the username is {{.m.nickname}}
```

- slice

控制器数据赋值

```
1. ss :=[]string{"a","b","c"}  
2. this.Data["s"]=ss
```

模板渲染数据如下:

```
1. {{range $key, $val := .s}}  
2. {{$key}}  
3. {{$val}}  
4. {{end}}
```

模板名称

beego 采用了 Go 语言内置的模板引擎，所有模板的语法和 Go 的一模一样，至于如何写模板文件，详细的请参考 [模板教程](#)。

用户通过在 Controller 的对应方法中设置相应的模板名称，beego 会自动的在 viewpath 目录下查询该文件并渲染，例如下面的设置，beego 会在 admin 下面找 add.tpl 文件进行渲染：

```
1. thisTplNames = "admin/add.tpl"
```

我们看到上面的模板后缀名是 `tpl`，Beego 默认情况下支持 `tpl` 和 `html` 后缀名的模板文件，如果你的后缀名不是这两种，请进行如下设置：

```
1. beego.AddTemplateExt("你文件的后缀名")
```

当你设置了自动渲染，然后在你的 Controller 中没有设置任何的 `TplNames`，那么 Beego 会自动设置你的模板文件如下：

```
1. cTplNames = c.ChildName + "/" + c.Ctx.Request.Method + "." + cTplExt
```

也就是你对应的 Controller 名字+请求方法名.模板后缀，也就是如果你的 Controller 名是 `AddController`，请求方法是 `POST`，默认的文件后缀是 `tpl`，那么就会默认请求 `/viewpath/AddController/post.tpl` 文件。

## Layout 设计

Beego 支持 layout 设计，例如你在管理系统中，整个管理界面是固定的，只会变化中间的部分，那么你可以通过如下的设置：

```
1. thisLayout = "admin/layout.html"
2. thisTplNames = "admin/add.tpl"
```

在 `layout.html` 中你必须设置如下的变量：

```
1. {{.LayoutContent}}
```

Beego 就会首先解析 `TplNames` 指定的文件，获取内容赋值给 `LayoutContent`，然后最后渲染 `layout.html` 文件。

目前采用首先把目录下所有的文件进行缓存，所以用户还可以通过类似这样的方式实现 layout：

```
1. {{template "header.html"}}
2. 处理逻辑
3. {{template "footer.html"}}
```

## renderform 使用

定义 struct:

```
1. type User struct {
2.     Id      int      `form:"-"`
3.     Name   interface{} `form:"username"`
4.     Age    int      `form:"age,text,年龄:"`
5.     Sex    string
6.     Intro  string `form:",textarea"'
```

```
7. }
```

- StructTag 的定义用的标签用为 `form`，和 ParseForm方法 共用一个标签，标签后面有三个可选参数，用 `,` 分割。第一个参数为表单中类型的 `name` 的值，如果为空，则以 `struct field name` 为值。第二个参数为表单组件的类型，如果为空，则为 `text`。表单组件的标签默认为 `struct field name` 的值，否则为第三个值。
- 如果 `form` 标签只有一个值，则为表单中类型 `name` 的值，除了最后一个值可以忽略外，其他位置的必须要有 `,` 号分割，如：  
`form:",,姓名: "`
- 如果要忽略一个字段，有两种办法，一是：字段名小写开头，二是： `form` 标签的值设置为 `-`

controller:

```
1. func (this *AddController) Get() {
2.     this.Data["Form"] = &User{}
3.     this.TplNames = "index.tpl"
4. }
```

Form 的参数必须是一个 struct 的指针。

template:

```
1. <form action="" method="post">
2. {{.Form | renderform}}
3. </form>
```

上面的代码生成的表单为：

```
Name: <input name="username" type="text" value="test"><br>
年龄: <input name="age" type="text" value="0"><br>
Sex: <input name="Sex" type="text" value=""><br>
Intro: <input name="Intro" type="textarea" value="">
```

```
1.
```

## 模板函数

beego 支持用户定义模板函数，但是必须在 `beego.Run()` 调用之前，设置如下：

```
1. func hello(in string)(out string){
2.     out = in + "world"
3.     return
4. }
5.
6. beego.AddFuncMap("hi",hello)
```

定义之后你就可以在模板中这样使用了：

```
1. {{.Content | hi}}
```

目前 Beego 内置的模板函数如下所示：

- `dateformat`

实现了时间的格式化，返回字符串，使用方法 `{{dateformat .Time "2006-01-02T15:04:05Z07:00"}}`。

- `date`

实现了类似 PHP 的 `date` 函数，可以很方便的根据字符串返回时间，使用方法 `{{date .T "Y-m-d H:i:s"}}`。

- `compare`

实现了比较两个对象的比较，如果相同返回 `true`，否者 `false`，使用方法 `{{compare .A .B}}`。

- `substr`

实现了字符串的截取，支持中文截取的完美截取，使用方法 `{{substr .Str 0 30}}`。

- `html2str`

实现了把 `html` 转化为字符串，剔除一些 `script`、`css` 之类的元素，返回纯文本信息，使用方法 `{{html2str .Htmlinfo}}`。

- `str2html`

实现了把相应的字符串当作 `HTML` 来输出，不转义，使用方法 `{{str2html .Strhtml}}`。

- `htmlquote`

实现了基本的 `html` 字符转义，使用方法 `{{htmlquote .quote}}`。

- `htmlunquote`

实现了基本的反转移字符，使用方法 `{{htmlunquote .unquote}}`。

- `renderform`

根据 `StructTag` 直接生成对应的表单，使用方法 `{{&struct | renderform}}`。

## 1. 静态文件

### 静态文件

Go 语言内部其实已经提供了 `http.ServeFile`，通过这个函数可以实现静态文件的服务。beego 针对这个功能进行了一层封装，通过下面的方式进行静态文件注册：

```
1. beego.SetStaticPath("/static","public")
```

- 第一个参数是路径，url 路径信息
- 第二个参数是静态文件目录（相对应用所在的目录）

beego 支持多个目录的静态文件注册，用户可以注册如下的静态文件目录：

```
1. beego.SetStaticPath("/images","images")
2. beego.SetStaticPath("/css","css")
3. beego.SetStaticPath("/js","js")
```

设置了如上的静态目录之后，用户访问 `/images/login/login.png`，那么就会访问应用对应的目录下面的 `images/login/login.png` 文件。如果是访问 `/static/img/logo.png`，那么就访问 `public/img/logo.png` 文件。

默认情况下beego会判断目录下文件是否存在，不存在直接返回404页面，如果请求的是 `index.html`，那么由于 `http.Servefile` 默认是会跳转的，不提供该页面的显示，因此beego可以设置 `beego.DirectoryIndex=true` 这样来使得显示 `index.html` 页面。而且开启该功能之后，用户访问目录就会显示该目录下所有的文件列表。

## 分页处理

---

模板处理过程中经常需要分页，那么如何进行有效的开发和操作呢？我们开发组针对这个需求开发了如下的例子，希望对大家有用

- 工具类 <https://github.com/beego/wetalk/blob/master/utils/paginator.go>
- 模板 <https://github.com/beego/wetalk/blob/master/views/base/paginator.html>
- 使用方法 <https://github.com/beego/wetalk/blob/master/routers/post.go#L131>

## 1. 模块介绍

### 模块介绍

beego正在逐步的走向乐高模式，也就是把系统逐步的模块化，让一个一个的模块成为乐高的积木，用户可以把这些积木搭建成自己想要的东西，这个就是目前beego的发展方向，beego1.0发布的时候目前包含下面这些模块，这些模块都是我们平常开发过程中非常有用的：

- session模块
- cache模块
- logs模块
- httplib模块
- context模块
- toolbox模块
- config模块

1. session介绍
  1. session使用
  2. 引擎设置
  3. 如何创建自己的引擎

## session介绍

session模块是用来存储客户端用户，session模块目前只支持cookie方式的请求，如果客户端不支持cookie，那么就无法使用该模块。

session模块参考了 `database/sql` 的引擎写法，采用了一个接口，多个实现的方式，目前实现了memory, file, Redis 和 MySQL四种存储引擎。

通过下面的方式安装session：

```
1. go get github.com/astaxie/beego/session
```

## session使用

首先你必须import包

```
1. import (
2.     "github.com/astaxie/beego/session"
3. )
```

然后你初始化一个全局的变量用来存储session控制器：

```
1. var globalSessions *session.Manager
```

接着在你的入口函数中初始化数据：

```
1. func init() {
2.     globalSessions, _ = session.NewManager("memory", "gosessionid", 3600,"",false,"sha1","sessionidkey",3
3.     go globalSessions.GC()
4. }
```

NewManager函数的参数的函数如下所示

1. 引擎名字，可以是memory、file、mysql、redis
2. 客户端存储cookie的名字
3. 服务器端存储的数据的过期时间，同时也是触发GC的时间
4. 配置信息，根据不同的引擎设置不同的配置信息，详细的配置请看下面的引擎设置
5. 是否开启https，在cookie设置的时候有cookie.Secure设置
6. sessionID生产的函数，默认是sha1算法
7. hash算法中的key
8. 客户端存储的cookie的时间，默认值是0，浏览器生命周期

最后我们的业务逻辑处理函数中可以这样调用：

```
1. func login(w http.ResponseWriter, r *http.Request) {
2.     sess := globalSessions.SessionStart(w, r)
3.     defer sess.SessionRelease()
4.     username := sess.Get("username")
5.     if r.Method == "GET" {
6.         t, _ := template.ParseFiles("login.gtpl")
```

```
7.         t.Execute(w, nil)
8.     } else {
9.         sess.Set("username", r.Form["username"])
10.    }
11. }
```

globalSessions有多个函数如下所示：

- SessionStart 根据当前请求返回session对象
- SessionDestroy 销毁当前session对象
- SessionRegenerateId 重新生成sessionID
- GetActiveSession 获取当前活跃的session用户
- SetHashFunc 设置sessionID生成的函数
- SetSecure 设置是否开启cookie的Secure设置

返回的session对象是一个Interface，包含下面的方法

- Set(key, value interface{}) error
- Get(key interface{}) interface{}
- Delete(key interface{}) error
- SessionID() string
- SessionRelease()
- Flush() error

## 引擎设置

上面已经展示了memory的设置，接下来我们看一下其他三种引擎的设置方式：

- mysql

其他参数一样，只是第四个参数配置设置如下所示，详细的配置请参考mysql：

```
1.   username:password@protocol(address)/dbname?param=value
```

- redis

配置文件信息如下所示，表示链接的地址，连接池，访问密码，没有保持为空：

```
1.   127.0.0.1:6379,100,astaxie
```

- file

配置文件如下所示，表示需要保存的目录，默认是两级目录新建文件，例如sessionID是 xsnkjklkjkh27hjh78908，那么目录文件应该是 ./tmp/x/s/xsnkjklkjkh27hjh78908：

```
1.   ./tmp
```

## 如何创建自己的引擎

在开发应用中，你可能需要实现自己的session引擎，beego的这个session模块设计的时候就是采用了interface，所以你可以根据接口实现任意的引擎，例如memcache的引擎。

```
1. type SessionStore interface {
2.     Set(key, value interface{}) error //set session value
3.     Get(key interface{}) interface{} //get session value
4.     Delete(key interface{}) error //delete session value
5.     SessionID() string           //back current sessionID
6.     SessionRelease()             // release the resource & save data to provider
7.     Flush() error                //delete all data
8. }
```

```
9.  
10. type Provider interface {  
11.     SessionInit(maxlifetime int64, savePath string) error  
12.     SessionRead(sid string) (SessionStore, error)  
13.     SessionExist(sid string) bool  
14.     SessionRegenerate(oldsid, sid string) (SessionStore, error)  
15.     SessionDestroy(sid string) error  
16.     SessionAll() int //get all active session  
17.     SessionGC()  
18. }
```

最后需要注册自己写的引擎:

```
1. func init() {  
2.     Register("own", ownadaper)  
3. }
```

- 缓存模块
  - 使用入门
  - 引擎设置
  - 开发自己的引擎

name: cache模块

## sort: 2

## 缓存模块

beego的cache模块是用来做数据缓存的，设计思路来自于 [database/sql](#)，目前支持file、memcache、memory和redis四种引擎，安装方式如下：

```
1. go get github.com/astaxie/beego/cache
```

## 使用入门

首先引入包：

```
1. import (
2.     "github.com/astaxie/beego/cache"
3. )
```

然后初始化一个全局变量对象：

```
1. bm, err := NewCache("memory", `{"interval":60}`)
```

然后我们就可以使用bm增删改缓存：

```
1. bm.Put("astaxie", 1, 10)
2. bm.Get("astaxie")
3. bm.Exists("astaxie")
4. bm.Delete("astaxie")
```

## 引擎设置

目前支持四种不同的引擎，接下来分别介绍这四种引擎如何设置：

- memory

配置信息如下所示，配置的信息表示GC的时间，表示每个60s会进行一次过期清理：

```
1. {"interval":60}
```

- file

配置信息如下所示，配置 CachePath 表示缓存的文件目录， FileSuffix 表示文件后缀， DirectoryLevel 表示目录层级， EmbedExpiry 表示过期设置

```
1. {"CachePath": "./cache", "FileSuffix": ".cache", "DirectoryLevel": 2, "EmbedExpiry": 120}
```

- redis

配置信息如下所示，redis采用了库redigo，表示redis的连接地址：

```
1. {"conn": ":6039"}
```

- memcache

配置信息如下所示，memcache采用了vitess的库，表示memcache的连接地址：

```
1. {"conn": "127.0.0.1:11211"}
```

## 开发自己的引擎

cache模块采用了接口的方式实现，因此用户可以很方便的实现接口，然后注册就可以实现自己的Cache引擎：

```
1. type Cache interface {
2.     Get(key string) interface{}
3.     Put(key string, val interface{}, timeout int64) error
4.     Delete(key string) error
5.     Incr(key string) error
6.     Decr(key string) error
7.     IsExist(key string) bool
8.     ClearAll() error
9.     StartAndGC(config string) error
10. }
```

用户开发完毕在最后写类似这样的：

```
1. func init() {
2.     Register("myowncache", NewOwnCache())
3. }
```

1. 日志处理
  1. 如何使用
  2. 引擎配置设置

## 日志处理

这是一个用来处理日志的库，它的设计思路来自于 [database/sql](#)，目前支持的引擎有file、console、net、smtp，可以通过如下方式进行安装：

```
1. go get github.com/astaxie/beego/logs
```

### 如何使用

首先引入包：

```
1. import (
2.     "github.com/astaxie/beego/logs"
3. )
```

然后初始化log变量(10000表示缓存的大小)：

```
1. log := NewLogger(10000)
```

然后添加输出引擎(log支持同时输出到多个引擎)，这里我们以console为例，第一个参数是引擎名(包括：console、file、conn、smtp)，第二个参数表示配置信息，详细的配置请看下面介绍：

```
1. log.SetLogger("console", "")
```

然后我们就可以在我们的逻辑中开始任意的使用了：

```
1. log.Trace("trace %s %s", "param1", "param2")
2. log.Debug("debug")
3. log.Info("info")
4. log.Warn("warning")
5. log.Error("error")
6. log.Critical("critical")
```

## 引擎配置设置

### • console

可以设置输出的级别，或者不设置保持默认，默认输出到 `os.Stdout`

```
1. log := NewLogger(10000)
2. log.SetLogger("console", `{"level":1}`)
```

### • file

设置的例子如下所示：

```
1. log := NewLogger(10000)
2. log.SetLogger("file", `{"filename":"test.log"}`)
```

主要的参数如下说明：

- filename 保存的文件名
- maxlines 每个文件保存的最大行数， 默认值1000000
- maxsize 每个文件保存的最大尺寸， 默认值是 $1 << 28$ , //256 MB
- daily 是否按照每天logrotate，默认是true
- maxdays 文件最多保存多少天， 默认保存7天
- rotate 是否开启logrotate，默认是true
- level 日志保存的时候的级别， 默认是Trace级别

- conn

网络输出，设置的例子如下所示：

```
1. log := NewLogger(1000)
2. log.SetLogger("conn", `{"net":"tcp","addr":"7020"})
```

主要的参数说明如下：

- reconnectOnMsg 是否每次链接都重新打开链接， 默认是false
- reconnect 是否自动重新链接地址， 默认是false
- net 发开网络链接的方式， 可以使用tcp、 unix、 udp等
- addr 网络链接的地址
- level 日志保存的时候的级别， 默认是Trace级别

- smtp

邮件发送，设置的例子如下所示：

```
1. log := NewLogger(10000)
2. log.SetLogger("smtp", `{"username":"beegotest@gmail.com","password":"xxxxxxxxx","host":"smtp.gmail.com:5`)
```

主要的参数说明如下：

- username smtp验证的用户名
- password smtp验证密码
- host 发送的邮箱地址
- sendTos 邮件需要发送的人， 支持多个
- subject 发送邮件的标题， 默认是 Diagnostic message from server
- level 日志发送的级别， 默认是Trace级别

1. 客户端请求
  1. 如何使用
  2. 支持的方法对象
  3. 支持debug输出
  4. 支持HTTPS请求
  5. 支持超时设置
  6. 设置请求参数
  7. 发送大片的数据
  8. 设置header信息
  9. 获取返回结果

## 客户端请求

httpplib库主要用来模拟客户端发送http请求，类似于Curl工具，支持jquery的链式操作。使用起来相当的方便；通过如下方式进行安装

```
1. go get github.com/astaxie/beego/httpplib
```

### 如何使用

首先导入包

```
1. import (
2.     "github.com/astaxie/beego/httpplib"
3. )
```

然后初始化请求方法，返回对象

```
1. req:=httpplib.Get("http://beego.me/")
```

然后我们就可以获取数据了

```
1. str, err := req.String()
2. if err != nil {
3.     t.Fatal(err)
4. }
5. fmt.Println(str)
```

### 支持的方法对象

httpplib包里面支持如下的方法返回request对象：

- Get(url string)
- Post(url string)
- Put(url string)
- Delete(url string)
- Head(url string)

### 支持debug输出

可以根据上面五个方法返回的对象进行调试信息的输出：

```
1. req.Debug(true)
```

这样就可以看到请求数据的详细输出

```
1. Get("http://beego.me/").Debug(true).Response()
2.
3. //输出数据如下
4. GET / HTTP/0.0
5. Host: beego.me
6. User-Agent: beegoServer
```

## 支持HTTPS请求

如果请求的网站是https的，那么我们就需要设置client的TLS信息，如下所示：

```
1. httplib.SetTLSClientConfig(&tls.Config{InsecureSkipVerify: true})
```

关于如何设置这些信息请访问：<http://golang.org/pkg/crypto/tls/#Config>

## 支持超时设置

通过如下接口可以设置请求的超时时间和数据读取时间：

```
1. SetTimeout(connectTimeout, readWriteTimeout)
```

以上方法都是针对request对象的，所以你第一步必须是返回request对象，然后链式操作，类似这样的代码：

```
1. Get("http://beego.me/").SetTimeout(100 * time.Second, 30 * time.Second).Response()
```

## 设置请求参数

对于Put或者Post请求，需要发送参数，那么可以通过Param发送k/v数据，如下所示：

```
1. req:=httplib.Post("http://beego.me/")
2. req.Param("username", "astaxie")
3. req.Param("password", "123456")
```

## 发送大片的数据

有时候需要上传文件之类的模拟，那么如何发送这个文件数据呢？可以通过Body函数来操作，举例如下：

```
1. req:=httplib.Post("http://beego.me/")
2. bt,err:=ioutil.ReadFile("hello.txt")
3. if err!=nil{
4.     log.Fatal("read file err:",err)
5. }
6. req.Body(bt)
```

## 设置header信息

除了请求参数之外，我们有些时候需要模拟一些头信息，例如

```
1. Accept-Encoding:gzip,deflate,sdch
2. Host:beego.me
3. User-Agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/31.
```

可以通过Header函数来设置，如下所示：

```
1. req:=httplib.Post("http://beego.me/")
2. req.Header("Accept-Encoding","gzip,deflate,sdch")
3. req.Header("Host","beego.me")
4. req.Header("User-Agent","Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_0) AppleWebKit/537.36 (KHTML, like Ge
```

## 获取返回结果

上面这些都是在发送请求之前的设置，接下来我们开始发送请求，然后如何来获取数据呢？主要有如下几种方式： - 返回Response对象，`req.Response()`方法

```
1. 这个是http.Response对象，用户可以自己读取body的数据等。
```

- 返回bytes，`req.Bytes()`方法  
直接返回请求url返回的内容
- 返回string，`req.String()`方法  
直接返回请求url返回的内容
- 保存为文件，`req.ToFile(filename)`方法  
返回结果保存到文件名为filename的文件中
- 解析为json结构，`req.ToJson(result)`方法  
返回结构直接解析为json格式，解析到result对象中
- 解析为xml结构，`req.ToXML(result)`方法  
返回结构直接解析为xml格式，解析到result对象中

1. 上下文模块
  1. context对象
  2. Input对象
  3. Output对象

## 上下文模块

上下文模块主要是针对http请求中，request和response的进一步封装，他包括用户的输入和输出，用户的输入即为request，context模块中提供了Input对象进行解析，用户的输出即为response，context模块中提供了Output对象进行输出。

### context对象

context对象是对Input和Output的封装，里面封装了几个方法： - Redirect - Abort - WriteString - GetCookie - SetCookie

context对象是Filter函数的参数对象，这样你就可以通过filter来修改相应的数据，或者提前结束整个的执行过程。

### Input对象

Input对象是针对request的封装，里面通过request实现很多方便的方法，具体如下：

- Protocol  
获取用户请求的协议，例如 `HTTP/1.0`
- Uri  
用户请求的RequestURI，例如 `/hi`
- Url  
请求的URL地址，例如 `http://beego.me/about?username=astaxie`
- Site  
请求的站点地址,scheme+doamin的组合，例如 `http://beego.me`
- Scheme  
请求的scheme，例如"http"或者"https"
- Domain  
请求的域名，例如 `beego.me`
- Host  
请求的域名，和domain一样
- Method  
请求的方法，标准的http请求方法，例如 `GET`、`POST` 等
- Is  
判断是否是某一个方法，例如 `Is("GET")` 返回true
- IsAjax  
判断是否是ajax请求，如果是返回true，不是返回false
- IsSecure

判断当前请求是否https请求，是返回true，否返回false

- **IsWebsocket**

判断当前请求是否Websocket请求，如果是返回true，否返回false

- **IsUpload**

判断当前请求是否有文件上传，有返回true，否返回false

- **IP**

返回请求用户的IP，如果用户通过代理，一层一层剥离获取真实的IP

- **Proxy**

返回用户代理请求的所有IP

- **Refer**

返回请求的refer信息

- **SubDomains**

返回请求域名的子域名，例如请求是 `blog.beego.me`，那么调用该函数返回 `beego.me`

- **Port**

返回请求的端口，例如返回8080

- **UserAgent**

返回请求的 `UserAgent`，例如 `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/31.0.1650.57 Safari/537.36`

- **Param**

在路由设置的时候可以设置参数，这个是用来获取那些参数的，例如 `Param(":id")`，返回12

- **Query**

该函数返回Get请求和Post请求中的所有数据，和PHP中 `$_REQUEST` 类似

- **Header**

返回相应的header信息，例如 `Header("Accept-Language")`，就返回请求头中对应的信息 `zh-CN,zh;q=0.8,en;q=0.6`

- **Cookie**

返回请求中的cookie数据，例如 `Cookie("username")`，就可以获取请求头中携带的cookie信息中username对应的值

- **Session**

session是用户可以初始化的信息，默认采用了beego的session模块中的Session对象，用来获取存储在服务器端中的数据。

- **Body**

返回请求Body中数据，例如API应用中，很多用户直接发送json数据包，那么通过Query这种函数无法获取数据，就必须通过该函数获取数据。

- **GetData**

用来获取Input中Data中的数据

- **SetData**

用来设置Input中Data的值，上面GetData和这个函数都是用来方便用户在Filter中传递数据到Controller中来执行

## Output对象

`Output`是针对`Response`的封装，里面提供了很多方便的方法：

- `Header`

设置输出的header信息，例如 `Header("Server", "beego")`

- `Body`

设置输出的内容信息，例如 `Body([]byte("astaxie"))`

- `Cookie`

设置输出的cookie信息，例如 `Cookie("sessionID", "beegoSessionID")`

- `Json`

把Data格式化为Json，然后调用Body输出数据

- `Jsonp`

把Data格式化为Jsonp，然后调用Body输出数据

- `Xml`

把Data格式化为Xml，然后调用Body输出数据

- `Download`

把file路径传递进来，然后输出文件给用户

- `ContentType`

设置输出的ContentType

- `SetStatus`

设置输出的status

- `Session`

设置在服务器端保存的值，例如 `Session("username", "astaxie")`，这样用户就可以在下次使用的时候读取

- `IsCachable`

根据status判断，是否为缓存类的状态

- `IsEmpty`

根据status判断，是否为输出内容为空的状态

- `isOk`

根据status判断，是否为200的状态

- `IsSuccessful`

根据status判断，是否为正常的状态

- `IsRedirect`

根据status判断，是否为跳转类的状态

- `IsForbidden`

根据status判断，是否为禁用类的状态

- `IsNotFound`

根据status判断，是否为找不到资源类的状态

- `IsClientError`

根据status判断，是否为请求客户端错误的状态

- IsServerError

根据status判断，是否为服务器端错误的状态

- 核心工具模块
  - 如何安装
  - healthcheck
  - profile
  - statistics
  - task
  - spec详解

## 核心工具模块

这个模块主要是参考了Dropwizard框架，是一位用户提醒我说有这么一个框架，然后里面实现一些很酷的东西，那个issue详细描述了改功能的雏形，然后就在参考该功能的情况下增加了一些额外的很酷的功能，接下来我讲一一，介绍这个模块中的几个功能：健康检查、性能调试、访问统计、计划任务。

### 如何安装

```
1. go get github.com/astaxie/beego/toolbox
```

### healthcheck

监控检查是用于当你应用于产品环境中进程，检查当前的状态是否正常，例如你要检查当前数据库是否可用，如下例子所示：

```
1. type DatabaseCheck struct {
2. }
3.
4. func (dc *DatabaseCheck) Check() error {
5.     if dc.isConnected() {
6.         return nil
7.     } else {
8.         return errors.New("can't connect database")
9.     }
10. }
```

然后就可以通过如下方式增加检测项：

```
toolbox.AddHealthCheck("database",&DatabaseCheck{})
```

加入之后，你可以往你的管理端口 `/healthcheck` 发送GET请求：

```
1. $ curl http://beego.me:8088/healthcheck
2. * deadlocks: OK
3. * database: OK
```

如果检测显示是正确的，那么输出OK，如果检测出错，显示出错的信息。

### profile

对于运行中的进程的性能监控是我们进行程序调优和查找问题的最佳方法，例如GC、goroutine等基础信息。profile提供了方便的入口方便用户来调试程序，他主要是通过入口函数 `ProcessInput` 来进行处理各类请求，主要包括以下几种调试：

- `lookup goroutine`

打印出来当前全部的goroutine执行的事情，非常方便查找各个goroutine在做的事情

```
1. goroutine 3 [running]:
2.   runtime/pprof.writeGoroutineStacks(0x634238, 0xc210000008, 0x62b000, 0xd200000000000000)
3.     /Users/astaxie/go/src/pkg/runtime/pprof/pprof.go:511 +0x7c
4.   runtime/pprof.writeGoroutine(0x634238, 0xc210000008, 0x2, 0xd2676410957b30fd, 0xae98)
5.     /Users/astaxie/go/src/pkg/runtime/pprof/pprof.go:500 +0x3c
6.   runtime/pprof.(*Profile).WriteTo(0x52ebe0, 0x634238, 0xc210000008, 0x2, 0x1, ...)
7.     /Users/astaxie/go/src/pkg/runtime/pprof/pprof.go:229 +0xb4
8. _/Users/astaxie/github/beego/toolbox.ProcessInput(0x2c89f0, 0x10, 0x634238, 0xc210000008)
9.     /Users/astaxie/github/beego/toolbox/profile.go:26 +0x256
10.    _/Users/astaxie/github/beego/toolbox.TestProcessInput(0xc21004e090)
11.      /Users/astaxie/github/beego/toolbox/profile_test.go:9 +0x5a
12. testing.tRunner(0xc21004e090, 0x532320)
13.   /Users/astaxie/go/src/pkg/testing/testing.go:391 +0x8b
14. created by testing.RunTests
15.   /Users/astaxie/go/src/pkg/testing/testing.go:471 +0x8b2
16.
17.
18. goroutine 1 [chan receive]:
19. testing.RunTests(0x315668, 0x532320, 0x4, 0x4, 0x1)
20.   /Users/astaxie/go/src/pkg/testing/testing.go:472 +0x8d5
21. testing.Main(0x315668, 0x532320, 0x4, 0x4, 0x537700, ...)
22.   /Users/astaxie/go/src/pkg/testing/testing.go:403 +0x84
23. main.main()
24.   _/Users/astaxie/github/beego/toolbox/_test/_testmain.go:53 +0x9c
```

- lookup heap

用来打印当前heap的信息：

- **lookup threadcreate**

查看创建线程的信息

```
1.   threadcreate profile: total 4
2.   1 @ 0x17f68 0x183c7 0x186a8 0x188cc 0x19ca9 0xcf41 0x139a3 0x196c0
3.   # 0x183c7 newm+0x27      /Users/astaxie/go/src/pkg/runtime/proc.c:896
4.   # 0x186a8 startm+0xb8     /Users/astaxie/go/src/pkg/runtime/proc.c:974
5.   # 0x188cc handoffp+0x1ac  /Users/astaxie/go/src/pkg/runtime/proc.c:992
6.   # 0x19ca9 runtime.entersyscallblock+0x129 /Users/astaxie/go/src/pkg/runtime/proc.c:1514
7.   # 0xcf41 runtime.notetsleepg+0x71    /Users/astaxie/go/src/pkg/runtime/lock_sema.c:253
8.   # 0x139a3 runtime.MHeap_Scavenger+0xa3   /Users/astaxie/go/src/pkg/runtime/mheap.c:463
9.
10.
11. 1 @ 0x17f68 0x183c7 0x186a8 0x188cc 0x189c3 0x1969b 0x2618b
12. # 0x183c7 newm+0x27      /Users/astaxie/go/src/pkg/runtime/proc.c:896
13. # 0x186a8 startm+0xb8     /Users/astaxie/go/src/pkg/runtime/proc.c:974
14. # 0x188cc handoffp+0x1ac  /Users/astaxie/go/src/pkg/runtime/proc.c:992
15. # 0x189c3 stoplockeddm+0x83   /Users/astaxie/go/src/pkg/runtime/proc.c:1049
16. # 0x1969b runtime.gosched0+0x8b   /Users/astaxie/go/src/pkg/runtime/proc.c:1382
17. # 0x2618b runtime.mcall+0x4b   /Users/astaxie/go/src/pkg/runtime/asm_amd64.s:178
18.
19.
20. 1 @ 0x17f68 0x183c7 0x170bc 0x196c0
21. # 0x183c7 newm+0x27      /Users/astaxie/go/src/pkg/runtime/proc.c:896
22. # 0x170bc runtime.main+0x3c   /Users/astaxie/go/src/pkg/runtime/proc.c:191
23.
24.
25. 1 @
```

- **lookup block**

查看block信息

```
1.   --- contention:
2.   cycles/second=2294781025
```

- **start cpuprof**

开始记录cpuprof信息，生产一个文件cpu-pid.pprof，开始记录当前进程的CPU处理信息

- **stop cpuprof**

关闭记录信息

- **get memprof**

开启记录memprof，生产一个文件mem-pid.memprof

- **gc summary**

查看GC信息

```
1.   NumGC:2 Pause:54.54us Pause(Avg):170.82us Overhead:177.49% Alloc:248.97K Sys:3.88M Alloc(Rate):1.23G/s
```

## statistics

请先看下面这张效果图，你有什么想法，很酷？是的，很酷，现在toolbox就是支持这样的功能了：

api	times	used (s)	avg used (μs)
health_check	1569	0.389025	247.94450
user	84	0.020222	240.74013
user	11013	2.488854	225.99236
user	6769	1.529432	225.94646
user	1510	0.304061	201.36476
user	448	0.078789	175.86885
user	8	0.001320	165.05237
user	30	0.004907	163.57646
user	2	0.000306	153.02400
user	68	0.009889	145.42408
user	4265	0.615608	144.33961
user	13987	1.899781	135.82476
user	64824	8.611979	132.85170
user	466	0.060057	128.87832
user	572	0.069561	121.61032
user	3	0.000328	109.46133
user	2449	0.260781	106.48469
user	15370	1.626340	105.81263
user	1618	0.169566	104.79974
user	2094	0.211268	100.89219
user	8894	0.861414	96.85342
user	1543	0.148274	96.09460
user	1497	0.141289	94.38137
user	3053	0.276917	90.70320
user	1030	0.091681	89.01052
user	779	0.067895	87.15641
user	3563	0.299025	83.92517
user	2479	0.364974	82.88200

如何使用这个统计呢？如下所示添加统计：

```

1. toolbox.StatisticsMap.AddStatistics("POST", "/api/user", "&admin.user", time.Duration(2000))
2. toolbox.StatisticsMap.AddStatistics("POST", "/api/user", "&admin.user", time.Duration(120000))
3. toolbox.StatisticsMap.AddStatistics("GET", "/api/user", "&admin.user", time.Duration(13000))
4. toolbox.StatisticsMap.AddStatistics("POST", "/api/admin", "&admin.user", time.Duration(14000))
5. toolbox.StatisticsMap.AddStatistics("POST", "/api/user/astaxie", "&admin.user", time.Duration(12000))
6. toolbox.StatisticsMap.AddStatistics("POST", "/api/user/xiemengjun", "&admin.user", time.Duration(13000))
7. toolbox.StatisticsMap.AddStatistics("DELETE", "/api/user", "&admin.user", time.Duration(1400))

```

获取统计信息

```
1. toolbox.StatisticsMap.GetMap(os.Stdout)
```

输出如下格式的信息：

requestUrl	method	times	used	ms
/api/user	POST	2	122.00us	12
/api/user	GET	1	13.00us	13
/api/user	DELETE	1	1.40us	1.
/api/admin	POST	1	14.00us	14
/api/user/astaxie	POST	1	12.00us	12
/api/user/xiemengjun	POST	1	13.00us	13

## task

玩过linux的用户都知道有一个计划任务的工具crontab，我们经常利用该工具来定时的做一些任务，但是有些时候我们的进程中也希望定时的来处理一些事情，例如定时的汇报当前进程的内存信息，goroutine信息等。或者定时的进行手工触发GC，或者定时的清理一些日志数据等，所以实现了秒级别的定时任务，首先让我们看看如何使用：

1. 初始化一个任务

```
1. tk1 := toolbox.NewTask("tk1", "0 12 * * *", func() error { fmt.Println("tk1"); return nil })
```

函数原型：

```
NewTask(tname string, spec string, f TaskFunc) *Task
```

- tname 任务名称
- spec 定时任务格式，请参考下面的详细介绍
- f 执行的函数 func() error

## 2. 可以测试开启运行

可以通过如下的代码运行TaskFunc，和spec无关，用于检测写的函数是否如预期所希望的这样

```
1. err := tk.Run()  
2. if err != nil {  
3.     t.Fatal(err)  
4. }
```

## 3. 加入全局的计划任务列表

```
1. toolbox.AddTask("tk1", tk1)
```

## 4. 开始执行全局的任务

```
1. toolbox.StartTask()  
2. defer toolbox.StopTask()
```

## spec详解

spec格式是参照crontab做的，详细的解释如下所示：

```
//前6个字段分别表示：  
//      秒钟：0-59  
//      分钟：0-59  
//      小时：1-23  
//      日期：1-31  
//      月份：1-12  
//      星期：0-6（0表示周日）  
  
//还可以用一些特殊符号：  
//      *： 表示任何时刻  
//      ,： 表示分割，如第三段里：2,4，表示2点和4点执行  
//      -： 表示一个段，如第三端里： 1-5，就表示1到5点  
//      /n： 表示每个n的单位执行一次，如第三段里，*/1，就表示每隔1个小时执行一次命令。也可以写成1-23/1.  
//      //： 表示两个字段之间是空格  
//      0/30 * * * *          每30秒 执行  
//      0 43 21 * * *          21:43 执行  
//      0 15 05 * * *          05:15 执行  
//      0 0 17 * * *          17:00 执行  
//      0 0 17 * * 1          每周一的 17:00 执行  
//      0 0,10 17 * * 0,2,3    每周日,周二,周三的 17:00 和 17:10 执行  
//      0 0-10 17 1 * *        每月1日从 17:00 到 17:10 每隔1分钟 执行  
//      0 0 0 1,15 * *        每月1日和 15日和 一日的 0:00 执行  
//      0 42 4 1 * *          每月1日的 4:42分 执行  
//      0 0 21 * * 1-6        周一到周六 21:00 执行  
//      0 0,10,20,30,40,50 * * * *  每隔10分 执行  
//      0 */10 * * * *        每隔10分 执行  
//      0 * 1 * * *          从1:0到1:59 每隔1分钟 执行  
//      0 0 1 * * *          1:00 执行  
//      0 0 */1 * * *        每时0分 每隔1小时 执行  
//      0 0 * * * *          每时0分 每隔1小时 执行  
//      0 2 8-20/3 * * *      8:02,11:02,14:02,17:02,20:02 执行  
//      0 30 5 1,15 * *       1日 和 15日的 5:30 执行
```

## 1. 配置文件解析

### 1. 如何使用

## 配置文件解析

这是一个用来解析文件的库，它的设计思路来自于 [database/sql](#)，目前支持解析的文件格式有ini、json、xml、yaml，可以通过如下方式进行安装：

```
1. go get github.com/astaxie/beego/config
```

## 如何使用

首先初始化一个解析器对象

```
1. iniconf, err := NewConfig("ini", "testini.conf")
2. if err != nil {
3.     t.Fatal(err)
4. }
```

然后通过对象获取数据

```
1. iniconf.String("appname")
```

解析器对象支持的函数有如下：

- Set(key, val string) error
- String(key string) string
- Int(key string) (int, error)
- Int64(key string) (int64, error)
- Bool(key string) (bool, error)
- Float(key string) (float64, error)
- DIY(key string) (interface{}, error)

ini配置文件支持section操作，key通过 `section::key` 的方式获取

例如下面这样的配置文件

```
1. [demo]
2.     key1="asta"
3.     key2 = "xie"
4.
```

那么可以通过 `iniconf.String("demo::key2")` 获取值

## 1. beego 高级编程

root: true name: beego高级编程

**sort: 6**

## beego 高级编程

前面介绍了beego的一些基础信息，如果你想通过beego使用更多高级的功能，那么这里就是你需要的资料。

- [进程内监控](#)

beego默认会开启两个端口，一个是8080应用端口，对外服务，一个是8088端口，用于监控进程内的信息，执行定时任务等。

- [过滤器](#)

过滤器极大的方便了用户对业务逻辑的扩充，用户可以通过过滤器实现用户认证，访问日志记录、兼容性跳转等。

- [热升级](#)

热升级是业务开发中经常提到的，需要在不中断当前用户请求的情况下部署新应用。

 这个功能目前我觉得还不是很成熟，而且只在mac和linux下面测试通过，没有经过线上大量案例的测试，目前属于尝鲜阶段，所以使用请小心。目前推荐使用nginx的upstream实现。

1. 进程内监控
  1. 请求统计信息
  2. 性能调试
  3. 健康检查
  4. 定时任务

## 进程内监控

前面介绍了toolbox模块，beego默认已经在进程开启的时候监控端口，但是默认是监听在 `127.0.0.1:8088`，这样无法通过外网访问，当然你可以通过各种方法访问，例如nginx代理。

为了安全，建议用户在防火墙把8088端口给屏蔽了。

默认监控是开放的，你可以通过设置参数配置关闭监控：

```
1. beego.EnableAdmin = false
```

而且你还可以修改监听的地址和端口：

```
1. beego.AdminHttpAddr = "localhost"  
2. beego.AdminHttpPort = 8888
```

打开浏览器，输入URL：`http://localhost:8088/`，你会看到一句欢迎词：`Welcome to Admin Dashboard`。

目前由于刚做出来第一版本，因此还需要后续继续界面的开发。

## 请求统计信息

访问统计的url地址 `http://localhost:8088/qps`，展现如下所示：

1.	requestUrl	method	times	used	ms
2.	/	GET	2	2.35ms	1.
3.	/favicon.ico	GET	1	79.30us	7%
4.	/src/xx	GET	1	923.09us	9%
5.	/src	GET	1	792.93us	7%
6.	/123	GET	1	906.04us	9%

## 性能调试

性能监控包含多个命令，请求地址 `http://localhost:8088/prof`

当你输入的时候会提示你如何进行详细的调试，显示如下界面：

```
1. request url like '/prof?command=lookup goroutine'  
2. the command have below types:  
3. 1. lookup goroutine  
4. 2. lookup heap  
5. 3. lookup threadcreate  
6. 4. lookup block  
7. 5. start cpuprof  
8. 6. stop cpuprof  
9. 7. get memprof  
10. 8. gc summary
```

用户可以通过传递不同的command值获取不同的性能调试信息，比较常用的有 `lookup goroutine`、`lookup heap` 和 `gc`

## summary

### 健康检查

需要手工注册相应的健康检查逻辑，才能通过URL <http://localhost:8088/healthcheck> 获取当前执行的健康检查的状态。

### 定时任务

用户需要在应用中添加了task，才能执行相应的任务检查和手工触发任务。

- 检查任务状态URL: <http://localhost:8088/task>
- 手工执行任务URL: <http://localhost:8088/runtask?taskname=任务名>

## 1. 过滤器

# 过滤器

beego 支持自定义过滤中间件，例如安全验证，强制跳转等。

过滤器函数如下所示：

```
1. beego.AddFilter(pattern, action string, filter FilterFunc)
```

AddFilter函数的三个参数

- pattern 路由规则，可以根据一定的规则进行路由，如果你全匹配可以用 \*
- action 执行 Filter 的地方，四个固定参数如下，分别表示不同的执行过程
  - BeforeRouter 寻找路由之前
  - AfterStatic 静态渲染之后
  - BeforeExec 找到路由之后，开始执行相应的 Controller 之前
  - AfterExec 执行完 Controller 逻辑之后执行的过滤器
- filter filter 函数 type FilterFunc func(\*context.Context)

如下例子所示，验证用户是否已经登录，应用于全部的请求：

```
1. var FilterUser = func(ctx *context.Context) {
2.     _, ok := ctx.Input.Session("uid").(int)
3.     if !ok {
4.         ctx.Redirect(302, "/login")
5.     }
6. }
7.
8. beego.AddFilter("*", "AfterStatic", FilterUser)
```

这里需要特别注意使用session的Filter必须在AfterStatic之后才能获取，因为session没有在这之前初始化

还可以通过正则路由进行过滤，如果匹配参数就执行：

```
1. var FilterUser = func(ctx *context.Context) {
2.     _, ok := ctx.Input.Session("uid").(int)
3.     if !ok {
4.         ctx.Redirect(302, "/login")
5.     }
6. }
7. beego.AddFilter("/user/:id([0-9]+)", "AfterStatic", FilterUser)
```

## 1. 热升级

- 1. 热升级是什么？
- 2. 热升级有必要吗？
- 3. Beego 如何支持热升级
- 4. 如何演示热升级

## 热升级

### 热升级是什么？

热升级是什么呢？了解 nginx 的同学都知道，nginx 是支持热升级的，可以用老进程服务先前链接的链接，使用新进程服务新的链接，即在不停止服务的情况下完成系统的升级与运行参数修改。那么热升级和热编译是不同的概念，热编译是通过监控文件的变化重新编译，然后重启进程，例如 `bee run` 就是这样的工具。

### 热升级有必要吗？

很多人认为 HTTP 的应用有必要支持热升级吗？那么我可以很负责的说非常有必要，不中断服务始终是我们所追求的目标，虽然很多人说可能服务器会坏掉等等，这个是属于高可用的设计范畴，不要搞混了，这个是可预知的问题，所以我们需要避免这样的升级带来的用户不可用。你还在为以前升级搞到凌晨升级而烦恼嘛？那么现在就赶紧拥抱热升级吧。

### Beego 如何支持热升级

热升级的原理基本上就是：主进程 `fork` 一个进程，然后子进程 `exec` 相应的程序。那么这个过程中发生了什么呢？我们知道进程 `fork` 之后会把主进程的所有句柄、数据和堆栈继承过来、但是里面所有的句柄存在一个叫做 `CloseOnExec`，也就是执行 `exec` 的时候，`copy` 的所有的句柄都被关闭了，除非特别申明，而我们期望的是子进程能够复用主进程的 `net.Listener` 的句柄。一个进程一旦调用 `exec` 类函数，它本身就“死亡”了，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段与堆栈段，唯一留下的，就是进程号，也就是说，对系统而言，还是同一个进程，不过已经是另一个程序了。

那么我们要做的第一步就是让子进程继承主进程的这个句柄，我们可以通过 `os.StartProcess` 的参数来附加 `Files`，把需要继承的句柄写在里面。

第二步就是我们希望子进程能够从这个句柄启动监听，还好 Go 里面支持 `net.FileListener`，直接从句柄来监听，但是我们需要子进程知道这个 FD，所以在启动子进程的时候我们设置了一个环境变量设置这个 FD。

第三步就是我们期望老的链接继续服务完，而新的链接采用新的进程，这里面有两个细节，第一就是老的链接继续服务，那么我们怎么知道有老链接存在？所以我们必须每次接收一个链接记录一下，这样我们就知道还存在没有服务完的链接，第二就是怎么让老进程停止接收数据，让新进程接收数据呢？大家都监听在同一个端口，理论上是随机来接收的，所以这里我们只要关闭老的链接的接收就行，这样就会使得在 `l.Accept` 的时候报错。

上面是我们需要解决的三个方面的问题，具体的实现大家可以看我实现的代码逻辑。

### 如何演示热升级

1. 编写代码，在 Beego 应用的控制器中 `Get` 方法实现大概如下：

```
1. func (this *MainController) Get() {  
2.     a, _ := this.GetInt("sleep")  
3.     time.Sleep(time.Duration(a) * time.Second)  
4.     this.Ctx.WriteString("ospid:" + strconv.Itoa(os.Getpid()))  
5. }
```

2. 打开两个终端：

一个终端输入：`ps -ef|grep 应用名`

一个终端输入请求：`curl "http://127.0.0.1:8080/?sleep=20"`

3. 热升级：

`kill -HUP 进程ID`

4. 打开一个终端输入请求：`curl "http://127.0.0.1:8080/?sleep=0"`

我们可以看到这样的结果，第一个请求等待 20s，但是处理他的是老的进程，热升级之后，第一个请求还在执行，最后会输出老的进程 ID，而第二次请求，输出的是新的进程 ID。

1. 发行部署
  1. 开发模式
  2. 发行部署

## 发行部署

### 开发模式

通过bee创建的项目，beego默认情况下是开发模式。

我们可以通过如下方式改变我们的模式：

```
1. beego.RunMode = "prod"
```

或者我们在conf/app.conf下面设置如下：

```
1. runmode = prod
```

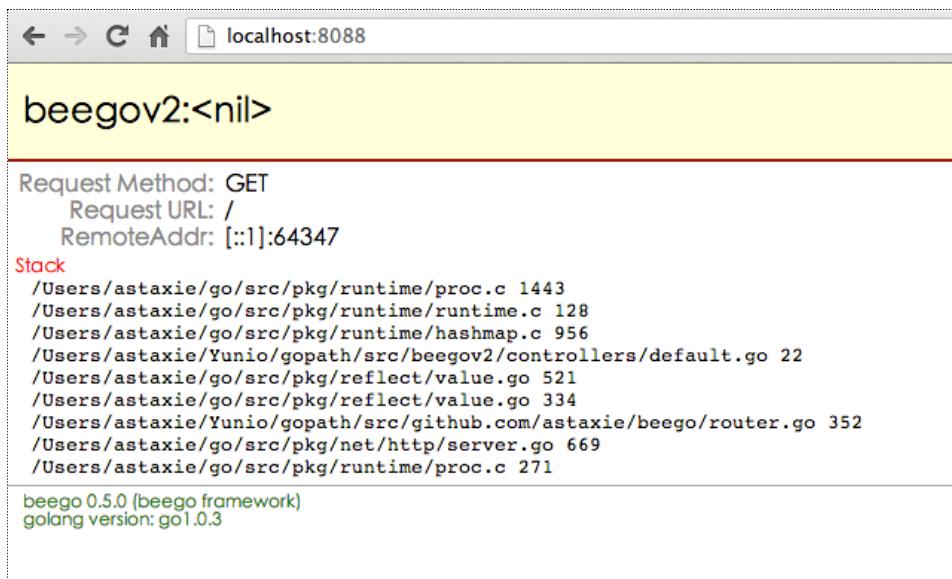
以上两种效果一样。

### 开发模式中

- 开发模式下，如果你的目录不存在views目录，那么会出现类似下面的错误提示：

```
1. 2013/04/13 19:36:17 [W] [stat views: no such file or directory]
```

- 模板每次使用都会重新加载，不进行缓存。
- 如果服务端出错，那么就会在浏览器端显示如下类似的截图：



### 发行部署

Go语言的应用最后编译之后是一个二进制文件，你只需要 copy 这个应用到服务器上，运行起来就行。beego 由于带有几个静态文件、配置文件、模板文件三个目录，所以用户部署的时候需要同时 copy 这三个目录到相应的部署应用之下，下面以我实际的应用部署为例：

```
1. $ mkdir /opt/app/beepkg
2. $ cp beepkg /opt/app/beepkg
3. $ cp -fr views /opt/app/beepkg
4. $ cp -fr static /opt/app/beepkg
5. $ cp -fr conf /opt/app/beepkg
```

这样在 `/opt/app/beepkg` 目录下面就会显示如下的目录结构：

```
1. .
2. └── conf
3.   |   ├── app.conf
4.   └── static
5.     |   ├── css
6.     |   ├── img
7.     |   └── js
8.   └── views
9.     └── index.tpl
10.  └── beepkg
```

这样我们就已经把我们需要的应用搬到服务器了，那么接下来就可以开始部署了。

这里部署首先你需要把应用跑起来，这分为两种方式：

- 独立部署
- Supervisord部署

上面只是把应用程序完全暴露在外部，我们大多数的应用会在前端部署一个nginx或者apache利用这些成熟的HTTP服务器做负载均衡或者其他认证之类的。

- Nginx部署
- Apache部署

## 独立部署

---

独立部署即为在后端运行程序，让程序跑在后台。

### linux

---

在linux下面部署，我们可以利用 `nohup` 命令，把应用部署在后端，如下所示：

```
1. nohup ./beepkg &
```

这样你的应用就跑在了Linux系统的守护进程

### windows

---

在window系统中，设置开机自动，后台运行，有如下几种方式：

1. 制作bat文件，放在“启动”里面
2. 制作成服务

1. Supervisord
  1. supervisord 安装
  2. supervisord 管理

name: supervisor部署

**sort: 2**

## Supervisord

Supervisord是用Python实现的一款非常实用的进程管理工具，supervisord还要求管理的程序是非daemon程序，supervisord会帮你把它转成daemon程序，因此如果用supervisord来管理nginx的话，必须在nginx的配置文件里添加一行设置daemon off让nginx以非daemon方式启动。

### supervisord 安装

#### 1. 安装 setuptools

```
1. wget http://pypi.python.org/packages/2.7/s/setuptools/setuptools-0.6c11-py2.7.egg
2.
3.
4. sh setuptools-0.6c11-py2.7.egg
5.
6.
7. easy_install supervisor
8.
9.
10. echo_supervisord_conf >/etc/supervisord.conf
11.
12.
13. mkdir /etc/supervisord.conf.d
```

#### 2. 修改配置 /etc/supervisord.conf

```
1. [include]
2. files = /etc/supervisord.conf.d/*.conf
```

#### 3. 新建管理的应用

```
1. cd /etc/supervisord.conf.d
2. vim beepkg.conf
```

配置文件：

```
1. [program:beepkg]
2. directory = /opt/app/beepkg
3. command = /opt/app/beepkg/beepkg
```

```
4.  autostart = true
5.  startsecs = 5
6.  user = root
7.  redirect_stderr = true
8.  stdout_logfile = /var/log/supervisord/beepkg.log
```

## supervisord 管理

Supervisord安装完成后有两个可用的命令行supervisord和supervisorctl，命令使用解释如下：

- supervisord, 初始启动Supervisord, 启动、管理配置中设置的进程。
- supervisorctl stop programxxx, 停止某一个进程(programxxx), programxxx为[program:beepkg]里配置的值, 这个示例就是beepkg。
- supervisorctl start programxxx, 启动某个进程
- supervisorctl restart programxxx, 重启某个进程
- supervisorctl stop groupworker:, 重启所有属于名为groupworker这个分组的进程(start,restart同理)
- supervisorctl stop all, 停止全部进程, 注: start、restart、stop都不会载入最新的配置文件。
- supervisorctl reload, 载入最新的配置文件, 停止原有进程并按新的配置启动、管理所有进程。
- supervisorctl update, 根据最新的配置文件, 启动新配置或有改动的进程, 配置没有改动的进程不会受影响而重启。

注意：显示用stop停止掉的进程，用reload或者update都不会自动重启。

## nginx部署

Go是一个独立的http服务器，但是我们有些时候为了nginx可以帮我做很多工作，例如访问日志，cc攻击，静态服务等，nginx已经做的很成熟了，Go只要专注于业务逻辑和功能就好，所以通过nginx配置代理就可以实现多应用同时部署，如下就是典型的两个应用共享80端口，通过不同的域名访问，反向代理到不同的应用。

```
server {
    listen      80;
    server_name www.a.com;
    charset utf-8;
    access_log /home/a.com.access.log main;
    location / {
        proxy_pass http://127.0.0.1:8080;
    }
}

server {
    listen      80;
    server_name www.b.com;
    charset utf-8;
    access_log /home/b.com.access.log main;
    location / {
        proxy_pass http://127.0.0.1:8081;
    }
}
```

## 1. Apache配置

### Apache配置

apache和nginx的实现原理一样，都是做一个反向代理，把请求向后端传递，配置如下所示：

```
NameVirtualHost *:80
<VirtualHost *:80>
    ServerAdmin webmaster@dummy-host.example.com
    ServerName www.a.com
    ProxyRequests Off
    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>
    ProxyPass / http://127.0.0.1:8080/
    ProxyPassReverse / http://127.0.0.1:8080/
</VirtualHost>

<VirtualHost *:80>
    ServerAdmin webmaster@dummy-host.example.com
    ServerName www.b.com
    ProxyRequests Off
    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>
    ProxyPass / http://127.0.0.1:8081/
    ProxyPassReverse / http://127.0.0.1:8081/
</VirtualHost>
```

1. [示例程序](#)
  1. 在线聊天
  2. 短域名 API 服务
  3. Todo 列表

## 示例程序

本手册详细介绍了使用 beego 框架开发的示例应用程序，旨在帮助您更好的学习和理解 beego。

### 在线聊天

该示例通过使用 beego 来构建一个在线聊天应用向您展示如何结合 WebSocket 来开发应用程序。

### 短域名 API 服务

短域名 API 服务可以帮助你理解如何通过 beego 来开发高性能的 API 服务。

### Todo 列表

该示例通过编写一个 Todo 列表来展示如何使用 beego 来创建典型的 Web 应用程序。