

# 目 录

致谢

框架介绍

开始使用

核心概念

中间件模块

核心服务

Gzip

路由模块

模板引擎

数据绑定与验证

本地化您的应用

缓存管理 (Cache)

会话管理 (Session)

跨域请求攻击 (CSRF)

验证码服务

嵌入二进制数据

多站点支持

高级用法

常见问题

## 致谢

当前文档《Macaron文档》由 进击的皇虫 使用 书栈 (BookStack.CN) 进行构建, 生成于 2018-02-07。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能, 以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理, 书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候, 发现文档内容有不恰当的地方, 请向我们反馈, 让我们共同携手, 将知识准确、高效且有效地传递给每一个人。

同时, 如果您在日常生活、工作和学习中遇到有价值有营养的知识文档, 欢迎分享到 书栈(BookStack.CN), 为知识的传承献上您的一份力量!

如果当前文档生成时间太久, 请到 书栈(BookStack.CN) 获取最新的文档, 以跟上知识更新换代的步伐。

文档地址: <http://www.bookstack.cn/books/Macaron>

书栈官网: <http://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享, 让知识传承更久远! 感谢知识的创造者, 感谢知识的分享者, 也感谢每一位阅读到此处的读者, 因为我们都将成为知识的传承者。

# 框架介绍

- [Macaron](#)
  - [主要特性](#)
  - [使用案例](#)
  - [快速导航](#)

## Macaron

---

Macaron 是一个具有高生产力和模块化设计的 Go Web 框架。框架秉承了 [Martini](#) 的基本思想，并在此基础上做出高级扩展。

### API 指南

Go 语言的最低版本要求为 **1.3**。

## 主要特性

---

- 支持子路由的强大路由设计
- 支持灵活多变的路由组合
- 支持无限路由组的无限嵌套
- 支持直接集成现有的服务
- 支持运行时动态设置需要渲染的模板集
- 支持使用内存文件作为静态资源和模板文件
- 支持对模块的轻松接入与解除
- 采用 [inject](#) 提供的便利的依赖注入
- 采用更好的路由层和更少的反射来提升执行速度

## 使用案例

---

- [Gogs](#): 极易搭建的自助 Git 服务
- [Peach](#): 现代 Web 文档服务器
- [Go Walker](#): Go 语言在线 API 文档
- [Switch](#): Go 语言包管理
- [YouGam](#): 在线论坛
- [Critical Stack Intel](#): A 100% free intel marketplace from Critical Stack, Inc.

## 快速导航

---

- 刚开始了解 Macaron 的话，不妨从 [开始使用](#) 看起。
- Macaron 已经拥有许多 [中间件](#) 来简化您的工作。
- 如果您有任何问题，建议先从 [常见问题](#) 中寻找答案。
- 如果您觉得文档有描述得不够清楚之处，请通过 [提交工单](#) 告知我们。

# 开始使用

- [开始使用 Macaron](#)
  - [最简示例](#)
  - [扩展示例](#)
  - [了解更多](#)

## 开始使用 Macaron

在我们开始之前，必须明确的一点就是，文档不会教授您任何有关 Go 语言的基础知识。所有对 Macaron 使用的讲解均是基于您已有的知识基础上展开的。

通过执行以下命令来安装 Macaron：

```
1. go get gopkg.in/macaron.v1
```

并且可以在今后使用以下命令来升级 Macaron：

```
1. go get -u gopkg.in/macaron.v1
```

## 最简示例

创建一个名为 `main.go` 的文件，然后输入以下代码：

```
1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.Classic()
7.     m.Get("/", func() string {
8.         return "Hello world!"
```

```

9.     })
10.    m.Run()
11. }

```

函数 `macaron.Classic` 创建并返回一个 **经典 Macaron** 实例。

方法 `m.Get` 是用于注册针对 HTTP GET 请求的路由。在本例中，我们注册了针对根路径 `/` 的路由，并提供了一个 **处理器** 函数来进行简单的处理操作，即返回内容为 `Hello world!` 的字符串作为响应。

您可能会问，为什么处理器函数可以返回一个字符串作为响应？这是由于 **返回值** 所带来的特性。换句话说，我们在本例中使用了 Macaron 中处理器的一个特殊语法来将返回值作为响应内容。

最后，我们调用 `m.Run` 方法来让服务器启动。在默认情况下，**Macaron 实例** 会监听 `0.0.0.0:4000`。

接下来，就可以执行命令 `go run main.go` 运行程序。您应该在程序启动后看到一条日志信息：

```
1. [Macaron] listening on 0.0.0.0:4000 (development)
```

现在，打开您的浏览器然后访问 `localhost:4000`。您会发现，一切是如此的美好！

## 扩展示例

现在，让我们对 `main.go` 做出一些修改，以便进行更多的练习。

```

1. package main
2.
3. import (
4.     "log"

```

```

5.     "net/http"
6.
7.     "gopkg.in/macaron.v1"
8. )
9.
10. func main() {
11.     m := macaron.Classic()
12.     m.Get("/", myHandler)
13.
14.     log.Println("Server is running...")
15.     log.Println(http.ListenAndServe("0.0.0.0:4000", m))
16. }
17.
18. func myHandler(ctx *macaron.Context) string {
19.     return "the request path is: " + ctx.Req.RequestURI
20. }

```

当您再次执行命令 `go run main.go` 运行程序的时候，您会看到屏幕上显示的内容为 `the request path is: /`。

那么，是什么改变了事物原本的样貌？（答：爱情）

首先，我们依旧使用了 **经典 Macaron** 来为根路径 `/` 注册针对 HTTP GET 请求的路由。但我们不再使用匿名函数，而是改用名为 `myHandler` 的函数作为处理器。需要注意的是，注册路由时，不需要在函数名称后面加上括号，因为我们不需要在此时调用这个函数。

函数 `myHandler` 接受一个类型为 `*macaron.Context` 的参数，并返回一个字符串。您可能已经发现我们并没有告诉 Macaron 需要传递什么参数给处理器，而且当您查看 `m.Get` 方法的声明时会发现，Macaron 实际上将所有的处理器（`macaron.Handler`）都当作类型 `interface{}` 来处理。那么，Macaron 又是怎么知道需要传递什么参数来调用处理器并执行逻辑的呢？

这就涉及到 **服务注入** 的概念了，`*macaron.Context` 就是默认注入

的服务之一，所以您可以直接使用它作为参数。如果您不明白怎么注入您自己的服务，没关系，反正还不是时候知道这些。

和之前的例子一样，我们需要让服务器监听在某个地址上。这一次，我们使用 Go 标准库的函数 `http.ListenAndServe` 来完成这项操作。如此一来，您便可以发现，任一 **Macaron 实例** 都是和标准库完全兼容的。

## 了解更多

---

您现在已经知道怎么基于 Macaron 来书写简单的代码，请尝试修改上文中的两个示例，并确保您已经完全理解上文中的所有内容。

当您觉得自己已经原地满血复活后，就可以继续学习之后的内容了。



## 核心概念

- Macaron 核心概念
  - 经典 Macaron
  - Macaron 实例
  - 处理器
    - 返回值
    - 服务注入
    - 中间件机制
  - Macaron 环境变量
  - 处理器工作流

## Macaron 核心概念

---

### 经典 Macaron

---

为了更快速的启用 Macaron, `macaron.Classic` 提供了一些默认的组件以方便 Web 开发:

```
1. m := macaron.Classic()  
2. // ... 可以在这里使用中间件和注册路由  
3. m.Run()
```

下面是 `macaron.Classic` 已经包含的功能:

- 请求/响应日志 - `macaron.Logger`
- 容错恢复 - `macaron.Recovery`
- 静态文件服务 - `macaron.Static`

## Macaron 实例

---

任何类型为 `macaron.Macaron` 的对象都可以被认为是 Macaron 的实例，您可以在单个程序中使用任意数量的 Macaron 实例。

## 处理器

处理器是 Macaron 的灵魂和核心所在。一个处理器基本上可以是任何的函数：

```
1. m.Get("/", func() string {
2.     return "hello world"
3. })
```

如果想要将同一个函数作用于多个路由，则可以使用一个命名函数：

```
1. m.Get("/", myHandler)
2. m.Get("/hello", myHandler)
3.
4. func myHandler() string {
5.     return "hello world"
6. }
```

除此之外，同一个路由还可以注册任意多个处理器：

```
1. m.Get("/", myHandler1, myHandler2)
2.
3. func myHandler1() {
4.     // ... 处理内容
5. }
6.
7. func myHandler2() string {
8.     return "hello world"
9. }
```

## 返回值

当一个处理器返回结果的时候，Macaron 将会把返回值作为字符串写入到当前的 `http.ResponseWriter` 里面：

```

1. m.Get("/", func() string {
2.     return "hello world" // HTTP 200 : "hello world"
3. })
4.
5. m.Get("/", func() *string {
6.     str := "hello world"
7.     return &str // HTTP 200 : "hello world"
8. })
9.
10. m.Get("/", func() []byte {
11.     return []byte("hello world") // HTTP 200 : "hello world"
12. })
13.
14. m.Get("/", func() error {
15.     // 返回 nil 则什么都不会发生
16.     return nil
17. }, func() error {
18.     // ... 得到了错误
19.     return err // HTTP 500 : <错误消息>
20. })

```

另外你也可以选择性的返回状态码（仅适用于 `string` 和 `[]byte` 类型）：

```

1. m.Get("/", func() (int, string) {
2.     return 418, "i'm a teapot" // HTTP 418 : "i'm a teapot"
3. })
4.
5. m.Get("/", func() (int, *string) {
6.     str := "i'm a teapot"
7.     return 418, &str // HTTP 418 : "i'm a teapot"
8. })
9.
10. m.Get("/", func() (int, []byte) {

```

```
11.     return 418, []byte("i'm a teapot") // HTTP 418 : "i'm a teapot"
12. })
```

## 服务注入

处理器是通过反射来调用的，Macaron 通过 **依赖注入** 来为处理器注入参数列表。 这样使得 **Macaron** 与 **Go** 语言的 `http.HandlerFunc` 接口完全兼容。

如果你加入一个参数到你的处理器，Macaron 将会搜索它参数列表中的服务，并且通过类型判断来解决依赖关系：

```
1. m.Get("/", func(resp http.ResponseWriter, req *http.Request) {
2.     // resp 和 req 是由 Macaron 默认注入的服务
3.     resp.WriteHeader(200) // HTTP 200
4. })
```

在您的代码中最常用的服务应该是 `*macaron.Context`：

```
1. m.Get("/", func(ctx *macaron.Context) {
2.     ctx.Resp.WriteHeader(200) // HTTP 200
3. })
```

下面的这些服务已经被包含在经典 Macaron 中

( `macaron.Classic` )：

- `*macaron.Context` - HTTP 请求上下文
- `*log.Logger` - Macaron 全局日志器
- `http.ResponseWriter` - HTTP 响应流
- `*http.Request` - HTTP 请求对象

## 中间件机制

中间件处理器是工作于请求和路由之间的。本质上来说和 Macaron 其他的处理器没有分别。您可以使用如下方法来添加一个中间件处理器到队列中：

```
1. m.Use(func() {
2.     // 处理中间件事务
3. })
```

您可以通过 `Handlers` 函数对中间件队列实现完全的控制。它将会替换掉之前的任何设置过的处理器：

```
1. m.Handlers(
2.     Middleware1,
3.     Middleware2,
4.     Middleware3,
5. )
```

中间件处理器可以非常好处理一些功能，包括日志记录、授权认证、会话 (sessions) 处理、错误反馈等其他任何需要在发生在 HTTP 请求之前或者之后的操作：

```
1. // 验证一个 API 密钥
2. m.Use(func(ctx *macaron.Context) {
3.     if ctx.Req.Header.Get("X-API-KEY") != "secret123" {
4.         ctx.Resp.WriteHeader(http.StatusUnauthorized)
5.     }
6. })
```

## Macaron 环境变量

一些 Macaron 处理器依赖 `macaron.Env` 全局变量为开发模式和部署模式表现出不同的行为，不过更建议使用环境变量

`MACARON_ENV=production` 来指示当前的模式为部署模式。

# 处理器工作流

---



# 中间件模块

- [中间件及辅助模块](#)
  - [注册中间件的最佳顺序](#)

## 中间件及辅助模块

---

中间件及辅助模块允许您轻易地对模块的进行接入与解除到您的 Macaron 应用中。

现在已经有许多 [中间件和模块](#) 来简化您的工作：

- [gzip](#) - Gzip 压缩所有响应
- [binding](#) - 请求数据绑定和校验
- [i18n](#) - 应用的国际化与本地化
- [cache](#) - Cache 管理器
- [session](#) - Session 管理器
- [csrf](#) - 生成和管理 CSRF 令牌
- [captcha](#) - 验证码服务
- [pongo2](#) - Pongo2 模板引擎支持
- [sockets](#) - WebSockets 管道绑定
- [bindata](#) - 嵌入二进制数据作为静态资源和模板文件
- [toolbox](#) - 健康检查、性能调试和路由统计等服务
- [oauth2](#) - OAuth 2.0 服务器后端客户端
- [authz](#) - 支持 ACL、RBAC 和 ABAC 的权限管理，基于 [Casbin](#)
- [switcher](#) - 多站点支持
- [method](#) - HTTP 方法覆盖
- [permissions2](#) - Cookies、多用户和权限管理
- [renders](#) - 类 Beego 模板引擎 (Macaron 已有内置模板引

擎，此为可选)

## 注册中间件的最佳顺序

有些中间件会依赖其它中间件，以下为最佳的注册顺序列表：

1. `macaron.Logger()`
2. `macaron.Recovery()`
3. `gzip.Gziper()`
4. `macaron.Static()`
5. `macaron.Renderer()` / `pongo2.Pongoer()`
6. `i18n.I18n()`
7. `cache.Cacher()`
8. `captcha.Captchaer()`
9. `session.Sessioner()`
0. `csrf.Csrfer()`
1. `toolbox.Toolboxer()`



## 核心服务

- 核心服务
  - 请求上下文 ( Context )
    - Next ( )
    - Cookie
    - 其它辅助方法
  - 路由日志
  - 容错恢复
  - 静态文件
    - 使用示例
    - 自定义选项
    - 注册多个静态处理器
  - 其它服务
    - 全局日志
    - 响应流
    - 请求对象

## 核心服务

Macaron 会注入一些默认服务来驱动您的应用，这些服务被称之为核心服务。也就是说，您可以直接使用它们作为处理器参数而不需要任何附加工作。

## 请求上下文 ( Context )

该服务通过类型 `*macaron.Context` 来体现。这是 Macaron 最为核心的服务，您的任何操作都是基于它之上。该服务包含了您所需要的请求对象、响应流、模板引擎接口、数据存储和注入与获取其它服务。

## 使用方法：

```

1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func Home(ctx *macaron.Context) {
6.     // ...
7. }

```

## Next()

方法 `Context.Next` 是一个可选的功能，它可以用于中间件处理器暂时放弃执行，等待其他的处理器都执行完毕后继续执行。这样就可以很好的处理在 HTTP 请求完成后需要做的操作：

```

1. // log before and after a request
2. m.Use(func(ctx *macaron.Context, log *log.Logger){
3.     log.Println("before a request")
4.
5.     ctx.Next()
6.
7.     log.Println("after a request")
8. })

```

## Cookie

### 最基本的 Cookie 用法：

- `*macaron.Context.SetCookie`
- `*macaron.Context.GetCookie`、`*macaron.Context.GetCookieInt`、`*macaron.Context.GetCookieInt64`、`*macaron.Context.GetCookieFloat64`

## 使用方法：

```

1. // ...
2. m.Get("/set", func(ctx *macaron.Context) {
3.     ctx.SetCookie("user", "Unknwon", 1)
4. })
5.
6. m.Get("/get", func(ctx *macaron.Context) string {
7.     return ctx.GetCookie("user")
8. })
9. // ...

```

使用以下顺序的参数来设置更多的属性：`SetCookie(<name>, <value>, <max age>, <path>, <domain>, <secure>, <http only>)`。

因此，设置 Cookie 最完整的用法为：`SetCookie("user", "unknwon", 999, "/", "localhost", true, true)`。

需要注意的是，参数的顺序是固定的。

如果需要更加安全的 Cookie 机制，可以先使用

`macaron.SetDefaultCookieSecret` 设定密钥，然后使用：

- `*macaron.Context.SetSecureCookie`
- `*macaron.Context.GetSecureCookie`

这两个方法将会自动使用您设置的默认密钥进行加密/解密 Cookie 值。

使用方法：

```

1. // ...
2. m.SetDefaultCookieSecret("macaron")
3. m.Get("/set", func(ctx *macaron.Context) {
4.     ctx.SetSecureCookie("user", "Unknwon", 1)
5. })
6.
7. m.Get("/get", func(ctx *macaron.Context) string {

```

```

8.     name, _ := ctx.GetSecureCookie("user")
9.     return name
10. })
11. // ...

```

对于那些对安全性要求特别高的应用，可以为每次设置 Cookie 使用不同的密钥加密/解密：

- `*macaron.Context.SetSuperSecureCookie`
- `*macaron.Context.GetSuperSecureCookie`

使用方法：

```

1. // ...
2. m.Get("/set", func(ctx *macaron.Context) {
3.     ctx.SetSuperSecureCookie("macaron", "user", "Unknwon", 1)
4. })
5.
6. m.Get("/get", func(ctx *macaron.Context) string {
7.     name, _ := ctx.GetSuperSecureCookie("macaron", "user")
8.     return name
9. })
10. // ...

```

## 其它辅助方法

- 设置/获取 URL 参数：`ctx.SetParams` /  
`ctx.Params`、`ctx.ParamsEscape`、`ctx.ParamsInt`、`ctx.ParamsInt64`、`ctx.ParamsFloat64`
- 获取查询参数：`ctx.Query`、`ctx.QueryEscape`、`ctx.QueryInt`、`ctx.QueryInt64`、`ctx.QueryFloat64`、`ctx.QueryStrings`、`ctx.QueryTrim`
- 服务内容或文件：`ctx.ServeContent`、`ctx.ServeFile`、`ctx.ServeFile`，

`ctx.ServeFileContent`

- 获取远程 IP 地址: `ctx.RemoteAddr`

## 路由日志

该服务可以通过函数 `macaron.Logger` 来注入。该服务主要负责应用的路由日志。

使用方法：

```
1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.New()
7.     m.Use(macaron.Logger())
8.     // ...
9. }
```

备注 当您使用 `macaron.Classic` 时，该服务会被自动注入。

从 [Peach](#) 项目中提取的样例输出：

```
1. [Macaron] Started GET /docs/middlewares/core.html for [::1]
2. [Macaron] Completed /docs/middlewares/core.html 200 OK in
   2.114956ms
```

## 容错恢复

该服务可以通过函数 `macaron.Recovery` 来注入。该服务主要负责在应用发生恐慌（panic）时进行恢复。

使用方法：

```
1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.New()
7.     m.Use(macaron.Recovery())
8.     // ...
9. }
```

备注 当您使用 `macaron.Classic` 时，该服务会被自动注入。

## 静态文件

该服务可以通过函数 `macaron.Static` 来注入。该服务主要负责应用静态资源的服务，当您的应用拥有多个静态目录时，可以对其进行多次注入。

使用方法：

```
1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.New()
7.     m.Use(macaron.Static("public"))
8.     m.Use(macaron.Static("assets"))
9.     // ...
10. }
```

备注 当您使用 `macaron.Classic` 时，该服务会以 `public` 为静态目录被自动注入。

默认情况下，当您请求一个目录时，该服务不会列出目录下的文件，而

是去寻找 `index.html` 文件。

从 [Peach](#) 项目中提取的样例输出：

```
1. [Macaron] Started GET /css/prettify.css for [::1]
2. [Macaron] [Static] Serving /css/prettify.css
3. [Macaron] Completed /css/prettify.css 304 Not Modified in 97.584us
4. [Macaron] Started GET /imgs/macaron.png for [::1]
5. [Macaron] [Static] Serving /imgs/macaron.png
6. [Macaron] Completed /imgs/macaron.png 304 Not Modified in 123.211us
7. [Macaron] Started GET /js/gogsweb.min.js for [::1]
8. [Macaron] [Static] Serving /js/gogsweb.min.js
9. [Macaron] Completed /js/gogsweb.min.js 304 Not Modified in 47.653us
10. [Macaron] Started GET /css/main.css for [::1]
11. [Macaron] [Static] Serving /css/main.css
12. [Macaron] Completed /css/main.css 304 Not Modified in 42.58us
```

## 使用示例

假设您的应用拥有以下目录结构：

```
1. public/
2.     |__ html
3.         |__ index.html
4.     |__ css/
5.         |__ main.css
```

响应结果：

请求 URL	匹配文件
<code>/html/main.html</code>	匹配失败
<code>/html/</code>	<code>index.html</code>
<code>/css/main.css</code>	<code>main.css</code>

## 自定义选项

该服务允许接受第二个参数来进行自定义选项操作

( `macaron.StaticOptions` ) :

```

1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.New()
7.     m.Use(macaron.Static("public",
8.         macaron.StaticOptions{
9.             // 请求静态资源时的 URL 前缀, 默认没有前缀
10.            Prefix: "public",
11.            // 禁止记录静态资源路由日志, 默认为不禁止记录
12.            SkipLogging: true,
13.            // 当请求目录时的默认索引文件, 默认为 "index.html"
14.            IndexFile: "index.html",
15.            // 用于返回自定义过期响应头, 默认为不设置
16.            //
17.            Expires: func() string {
18.                return time.Now().Add(24 * 60 *
19.                    time.Minute).UTC().Format("Mon, 02 Jan 2006 15:04:05 GMT")
20.            },
21.        })
22. }

```

## 注册多个静态处理器

如果您需要一次注册多个静态处理器, 可以使用方法 `macaron.Statics` 来简化您的工作。

使用方法:

```
1. // ...
```



```

2. m.Use(macaron.Statics(macaron.StaticOptions{}, "public", "views"))
3. // ...

```

这样，就可以同时注册 `public` 和 `views` 为静态目录了。

## 其它服务

### 全局日志

该服务通过类型 `*log.Logger` 来体现。该服务为可选，只是为没有日志器的应用提供一定的便利。

使用方法：

```

1. package main
2.
3. import (
4.     "log"
5.
6.     "gopkg.in/macaron.v1"
7. )
8.
9. func main() {
10.     m := macaron.Classic()
11.     m.Get("/", myHandler)
12.     m.Run()
13. }
14.
15. func myHandler(ctx *macaron.Context, logger *log.Logger) string {
16.     logger.Println("the request path is: " + ctx.Req.RequestURI)
17.     return "the request path is: " + ctx.Req.RequestURI
18. }

```

备注 所有 `Macaron` 实例 都会自动注册该服务。

## 响应流

该服务通过类型 `http.ResponseWriter` 来体现。该服务为可选，一般情况下可直接使用 `*macaron.Context.Resp`。

使用方法：

```

1. package main
2.
3. import (
4.     "gopkg.in/macaron.v1"
5. )
6.
7. func main() {
8.     m := macaron.Classic()
9.     m.Get("/", myHandler)
10.    m.Run()
11. }
12.
13. func myHandler(ctx *macaron.Context) {
14.     ctx.Resp.Write([]byte("the request path is: " +
15.         ctx.Req.RequestURI))
16. }
```

备注 所有 [Macaron 实例](#) 都会自动注册该服务。

## 请求对象

该服务通过类型 `*http.Request` 来体现。该服务为可选，一般情况下可直接使用 `*macaron.Context.Req`。

除此之外，该服务还提供了 3 个便利的方法来获取请求体：

- `*macaron.Context.Req.Body().String()`：获取 `string` 类型的请求体

- `*macaron.Context.Req.Body().Bytes()` : 获取 `[]byte` 类型的请求体
- `*macaron.Context.Req.Body().ReadCloser()` : 获取 `io.ReadCloser` 类型的请求体

使用方法：

```

1. package main
2.
3. import (
4.     "gopkg.in/macaron.v1"
5. )
6.
7. func main() {
8.     m := macaron.Classic()
9.     m.Get("/body1", func(ctx *macaron.Context) {
10.         reader, err := ctx.Req.Body().ReadCloser()
11.         // ...
12.     })
13.     m.Get("/body2", func(ctx *macaron.Context) {
14.         data, err := ctx.Req.Body().Bytes()
15.         // ...
16.     })
17.     m.Get("/body3", func(ctx *macaron.Context) {
18.         data, err := ctx.Req.Body().String()
19.         // ...
20.     })
21.     m.Run()
22. }
```

需要注意的是，请求体在每个请求中只能被读取一次。

有时您需要传递类型为 `*http.Request` 的参数，则应该使用

`*macaron.Context.Req.Request` 。

备注 所有 **Macaron 实例** 都会自动注册该服务。





# Gzip

- [Gzip](#)
  - [下载安装](#)
  - [使用示例](#)
  - [自定义选项](#)

## Gzip

中间件 `gzip` 为 [Macaron 实例](#) 的响应内容提供 Gzip 压缩。请确保在其它会向响应流写入内容的中间件之前注册该服务。

- [GitHub](#)
- [API 文档](#)

## 下载安装

```
1. go get github.com/go-macaron/gzip
```

## 使用示例

```
1. package main
2.
3. import (
4.     "github.com/go-macaron/gzip"
5.     "gopkg.in/macaron.v1"
6. )
7.
8. func main() {
9.     m := macaron.Classic()
10.    m.Use(gzip.Gzipper())
11.    // 注册路由
```

```

12.     m.Run()
13. }

```

在这个例子中，静态资源不会被 Gzip 压缩，如果想压缩它们，则可以使用以下方法：

```

1. package main
2.
3. import (
4.     "github.com/go-macaron/gzip"
5.     "gopkg.in/macaron.v1"
6. )
7.
8. func main() {
9.     m := macaron.New()
10.    m.Use(macaron.Logger())
11.    m.Use(macaron.Recovery())
12.    m.Use(gzip.Gzipper())
13.    m.Use(macaron.Static("public"))
14.    // 注册路由
15.    m.Run()
16. }

```

或者选择只压缩某一组路由的响应内容：

```

1. // ...
2.
3. func main() {
4.     m := macaron.Classic()
5.     m.Group("/gzip", func() {
6.         // ...
7.     }, gzip.Gzipper())
8.     // ...
9.     m.Run()
10. }

```

## 自定义选项

该服务允许接受一个参数来进行自定义选项 ( `gzip.Options` ) :

```
1. // ...
2. m.Use(gzip.Gzipper(gzip.Options{
3.     // 压缩级别, 可以是 DefaultCompression (-1)、
   ConstantCompression (-2)
4.     // 或介于包括 BestSpeed (1) 和 BestCompression (9) 在内, 这两者之间的任意整数。
5.     // 默认为 4
6.     CompressionLevel: 4,
7. })))
```

## 路由模块

- 路由模块
  - 命名参数
    - 占位符
    - 全局匹配
    - 正则表达式
  - 匹配优先级
    - 构建 URL 路径
      - 配合 Go 模板引擎使用
      - 配合 Pongo2 模板引擎使用
  - 高级路由定义
    - 组路由

## 路由模块

在 Macaron 中，路由是一个 HTTP 方法配对一个 URL 匹配模型。每一个路由可以对应一个或多个处理器方法：

```
1. m.Get("/", func() {
2.     // show something
3. })
4.
5. m.Patch("/", func() {
6.     // update something
7. })
8.
9. m.Post("/", func() {
10.    // create something
11. })
12.
13. m.Put("/", func() {
```



```
14.      // replace something
15.  })
16.
17.  m.Delete("/", func() {
18.      // destroy something
19.  })
20.
21.  m.Options("/", func() {
22.      // http options
23.  })
24.
25.  m.Any("/", func() {
26.      // do anything
27.  })
28.
29.  m.Route("/", "GET,POST", func() {
30.      // combine something
31.  })
32.
33.  m.Combo("/").
34.      Get(func() string { return "GET" }).
35.      Patch(func() string { return "PATCH" }).
36.      Post(func() string { return "POST" }).
37.      Put(func() string { return "PUT" }).
38.      Delete(func() string { return "DELETE" }).
39.      Options(func() string { return "OPTIONS" }).
40.      Head(func() string { return "HEAD" })
41.
42.  m.NotFound(func() {
43.      // 自定义 404 处理逻辑
44.  })
```

### 几点说明：

- 路由匹配的顺序是按照他们被定义的顺序执行的，
- ...但是，匹配范围较小的路由优先级比匹配范围大的优先级高（详见 匹配优先级）。

- 最先被定义的路由将会首先被用户请求匹配并调用。

在一些时候，每当 GET 方法被注册的时候，都会需要注册一个一模一样的 HEAD 方法。为了达到减少代码的目的，您可以使用一个名为

`SetAutoHead` 的方法来协助您自动注册：

```
1. m := New()
2. m.SetAutoHead(true)
3. m.Get("/", func() string {
4.     return "GET"
5. }) // 路径 "/" 的 HEAD 也已经被自动注册
```

如果您想要使用子路径但让路由代码保持简洁，可以调用

`m.SetURLPrefix(suburl)` 。

## 命名参数

路由模型可能包含参数列表，可以通过 `*Context.Params` 来获取：

## 占位符

使用一个特定的名称来代表路由的某个部分：

```
1. m.Get("/hello/:name", func(ctx *macaron.Context) string {
2.     return "Hello " + ctx.Params(":name")
3. })
4.
5. m.Get("/date/:year/:month/:day", func(ctx *macaron.Context) string
6.     {
7.         return fmt.Sprintf("Date: %s/%s/%s", ctx.Params(":year"),
8.             ctx.Params(":month"), ctx.Params(":day"))
9.     })
```

当然，想要偷懒的时候可以将 `:` 前缀去掉：

```

1. m.Get("/hello/:name", func(ctx *macaron.Context) string {
2.     return "Hello " + ctx.Params("name")
3. })
4.
5. m.Get("/date/:year/:month/:day", func(ctx *macaron.Context) string
6.     {
7.     return fmt.Sprintf("Date: %s/%s/%s", ctx.Params("year"),
8.         ctx.Params("month"), ctx.Params("day"))
9. })

```

## 全局匹配

路由匹配可以通过全局匹配的形式：

```

1. m.Get("/hello/*", func(ctx *macaron.Context) string {
2.     return "Hello " + ctx.Params("*")
3. })

```

那么，如果将 `*` 放在路由中间会发生什么呢？

```

1. m.Get("/date/*//*/events", func(ctx *macaron.Context) string {
2.     return fmt.Sprintf("Date: %s/%s/%s", ctx.Params("*0"),
3.         ctx.Params("*1"), ctx.Params("*2"))
4. })

```

## 正则表达式

您还可以使用正则表达式来书写路由规则：

- 常规匹配：

```

1. m.Get("/user/:username([\\w]+)", func(ctx *macaron.Context)
2.     string {
3.     return fmt.Sprintf("Hello %s", ctx.Params(":username"))
4. })

```

```

5.     m.Get("/user/:id([0-9]+)", func(ctx *macaron.Context) string
      {
6.         return fmt.Sprintf("User ID: %s", ctx.Params(":id"))
7.     })
8.
9.     m.Get("/user/*.*", func(ctx *macaron.Context) string {
10.         return fmt.Sprintf("Last part is: %s, Ext: %s",
            ctx.Params(":path"), ctx.Params(":ext"))
11.     })

```

- 混合匹配：

```

1.     m.Get("/cms_:id([0-9]+).html", func(ctx *macaron.Context)
      string {
2.         return fmt.Sprintf("The ID is %s", ctx.Params(":id"))
3.     })

```

- 可选匹配：

- `/user/?id` 可同时匹配 `/user/` 和 `/user/123`。

- 简写：

- `/user/:id:int`： `:int` 是 `([0-9]+)` 正则的简写。
- `/user/:name:string`： `:string` 是 `([\w]+)` 正则的简写。

## 匹配优先级

以下为从高到低的不同模式的匹配优先级：

- 静态路由：

- `/`
- `/home`

- 正则表达式路由：

- `/(.+).html`
- `/([0-9]+).css`

- 路径-后缀路由：

- `/*.*`

- 占位符路由：

- `/:id`

- `/:name`

- 全局匹配路由：

- `/*`

其它说明：

- 相同模式的匹配优先级是根据添加的先后顺序决定的。
- 层级相对明确的模式匹配优先级要高于相对模糊的模式：

- `/*/*/events` > `/*`

## 构建 URL 路径

您可以通过 `*Route.Name` 方法配合命名参数来构建 URL 路径，不过首先需要为路由命名：

```
1. // ...
2. m.Get("/users/:id([0-9]+)/:name:string.profile",
    handler).Name("user_profile")
3. m.Combo("/api/:user/:repo").Get(handler).Post(handler).Name("user_repo")
4. // ...
```

然后通过 `*Router.URLFor` 方法来为指定名称的路由构建 URL 路径：

```
1. // ...
2. func handler(ctx *macaron.Context) {
3.     // /users/12/unknown.profile
4.     userProfile := ctx.URLFor("user_profile", ":id", "12", ":name",
    "unknown")
}
```

```

5.      // /api/unknown/macaron
6.      userRepo := ctx.URLFor("user_repo", ":user", "unknown",
    ":repo", "macaron")
7.  }
8.  // ...

```

## 配合 Go 模板引擎使用

```

1.  // ...
2.  m.Use(macaron.Renderer(macaron.RenderOptions{
3.      Funcs:      []template.FuncMap{map[string]interface{}{
4.          "URLFor": m.URLFor,
5.      }},
6.  }))
7.  // ...

```

## 配合 Pongo2 模板引擎使用

```

1.  // ...
2.  ctx.Data["URLFor"] = ctx.URLFor
3.  ctx.HTML(200, "home")
4.  // ...

```

## 高级路由定义

路由处理器可以被相互叠加使用，例如很有用的地方可以是在验证和授权的时候：

```

1.  m.Get("/secret", authorize, func() {
2.      // this will execute as long as authorize doesn't write a
    response
3.  })

```

让我们来看一个比较极端的例子：

```

1. package main
2.
3. import (
4.     "fmt"
5.
6.     "gopkg.in/macaron.v1"
7. )
8.
9. func main() {
10.     m := macaron.Classic()
11.     m.Get("/",
12.         func(ctx *macaron.Context) {
13.             ctx.Data["Count"] = 1
14.         },
15.         func(ctx *macaron.Context) {
16.             ctx.Data["Count"] = ctx.Data["Count"].(int) + 1
17.         },
18.         func(ctx *macaron.Context) {
19.             ctx.Data["Count"] = ctx.Data["Count"].(int) + 1
20.         },
21.         func(ctx *macaron.Context) {
22.             ctx.Data["Count"] = ctx.Data["Count"].(int) + 1
23.         },
24.         func(ctx *macaron.Context) {
25.             ctx.Data["Count"] = ctx.Data["Count"].(int) + 1
26.         },
27.         func(ctx *macaron.Context) string {
28.             return fmt.Sprintf("There are %d handlers before this",
29.                 ctx.Data["Count"])
30.         },
31.     )
32.     m.Run()
33. }

```

先意淫下结果？没错，输出结果会是 `There are 5 handlers before this`。Macaron 并没有对您可以使用多少个处理器有一个硬性的限制。不过，Macaron 又是怎么知道什么时候停止调用下一个处理器的

呢？

想要回答这个问题，我们先来看下下一个例子：

```
1. package main
2.
3. import (
4.     "fmt"
5.
6.     "gopkg.in/macaron.v1"
7. )
8.
9. func main() {
10.     m := macaron.Classic()
11.     m.Get("/",
12.         func(ctx *macaron.Context) {
13.             ctx.Data["Count"] = 1
14.         },
15.         func(ctx *macaron.Context) {
16.             ctx.Data["Count"] = ctx.Data["Count"].(int) + 1
17.         },
18.         func(ctx *macaron.Context) {
19.             ctx.Data["Count"] = ctx.Data["Count"].(int) + 1
20.         },
21.         func(ctx *macaron.Context) {
22.             ctx.Data["Count"] = ctx.Data["Count"].(int) + 1
23.         },
24.         func(ctx *macaron.Context) string {
25.             return fmt.Sprintf("There are %d handlers before this",
26.                 ctx.Data["Count"])
27.         },
28.         func(ctx *macaron.Context) string {
29.             return fmt.Sprintf("There are %d handlers before this",
30.                 ctx.Data["Count"])
31.         },
32.     )
33.     m.Run()
34. }
```



在这个例子中，输出结果将会变成 `There are 4 handlers before this`，而最后一个处理器永远也不会被调用。这是为什么呢？因为我们已经在第 5 个处理器中向响应流写入了内容。所以说，一旦任一处理器向响应流写入任何内容，Macaron 将不会再调用下一个处理器。

## 组路由

路由还可以通过 `macaron.Group` 来注册组路由：

```
1. m.Group("/books", func() {
2.     m.Get("/:id", GetBooks)
3.     m.Post("/new", NewBook)
4.     m.Put("/update/:id", UpdateBook)
5.     m.Delete("/delete/:id", DeleteBook)
6.
7.     m.Group("/chapters", func() {
8.         m.Get("/:id", GetBooks)
9.         m.Post("/new", NewBook)
10.        m.Put("/update/:id", UpdateBook)
11.        m.Delete("/delete/:id", DeleteBook)
12.    })
13. })
```

同样的，您可以为某一组路由设置集体的中间件：

```
1. m.Group("/books", func() {
2.     m.Get("/:id", GetBooks)
3.     m.Post("/new", NewBook)
4.     m.Put("/update/:id", UpdateBook)
5.     m.Delete("/delete/:id", DeleteBook)
6.
7.     m.Group("/chapters", func() {
8.         m.Get("/:id", GetBooks)
9.         m.Post("/new", NewBook)
10.        m.Put("/update/:id", UpdateBook)
```

```
11.         m.Delete("/delete/:id", DeleteBook)
12.     }, MyMiddleware3, MyMiddleware4)
13. }, MyMiddleware1, MyMiddleware2)
```

同样的，Macaron 不在乎您使用多少层嵌套的组路由，或者多少个组级别处理器（中间件）。

## 模板引擎

- [name：模板引擎](#)
- [模板引擎](#)
  - [渲染 HTML](#)
    - [Go 模板引擎](#)
      - [使用示例](#)
      - [自定义选项](#)
    - [Pongo2 模板引擎](#)
      - [使用示例](#)
      - [自定义选项](#)
    - [模板集](#)
      - [模板集辅助方法](#)
    - [小结](#)
  - [渲染 XML、JSON 和原始数据](#)
  - [响应状态码、错误和重定向](#)
  - [运行时修改模板路径](#)
    - [使用示例](#)

---

## name：模板引擎

---

## 模板引擎

---

目前 Macaron 应用有两款官方模板引擎中间件可供选择，即

`macaron.Renderer` 和 `pongo2.Pongoer`。

您可以自由选择使用哪一款模板引擎，并且您只能为一个 [Macaron 实例](#) 注册一款模板引擎。

## 共有特性：

- 均支持 XML、JSON 和原始数据格式的响应，它们之间的不同只体现在 HTML 渲染上。
- 均使用 `templates` 作为默认模板文件目录。
- 均使用 `.tmpl` 和 `.html` 作为默认模板文件后缀。
- 均支持通过 `Macaron 环境变量` 来判断是否缓存模板文件（当 `macaron.Env == macaron.PROD` 时）。

## 渲染 HTML

### Go 模板引擎

该服务可以通过函数 `macaron.Renderer` 来注入，并通过类型 `macaron.Render` 来体现。该服务为可选，一般情况下可直接使用 `*macaron.Context.Render`。该服务使用 Go 语言内置的模板引擎来渲染 HTML。如果想要了解更多有关使用方面的信息，请参见 [官方文档](#)。

### 使用示例

假设您的应用拥有以下目录结构：

```
1. main/
2.   |__ main.go
3.   |__ templates/
4.       |__ hello.tmpl
```

hello.tmpl:

```
1. <h1>Hello {{.Name}}</h1>
```

main.go:

```

1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.Classic()
7.     m.Use(macaron.Renderer())
8.
9.     m.Get("/", func(ctx *macaron.Context) {
10.         ctx.Data["Name"] = "jeremy"
11.         ctx.HTML(200, "hello") // 200 为响应码
12.     })
13.
14.     m.Run()
15. }
```

## 自定义选项

该服务允许接受一个参数来进行自定义选项

( `macaron.RenderOptions` ) :

```

1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.Classic()
7.     m.Use(macaron.Renderer(macaron.RenderOptions{
8.         // 模板文件目录, 默认为 "templates"
9.         Directory: "templates",
10.        // 模板文件后缀, 默认为 [".tmpl", ".html"]
11.        Extensions: []string{".tmpl", ".html"},
12.        // 模板函数, 默认为 []
13.        Funcs: []template.FuncMap{map[string]interface{}{
14.            "AppName": func() string {
```

```

15.         return "Macaron"
16.     },
17.     "AppVer": func() string {
18.         return "1.0.0"
19.     },
20. },
21. // 模板语法分隔符, 默认为 ["{{", "}}"]
22. Delims: macaron.Delims["{{", "}}"],
23. // 追加的 Content-Type 头信息, 默认为 "UTF-8"
24. Charset: "UTF-8",
25. // 渲染具有缩进格式的 JSON, 默认为不缩进
26. IndentJSON: true,
27. // 渲染具有缩进格式的 XML, 默认为不缩进
28. IndentXML: true,
29. // 渲染具有前缀的 JSON, 默认为无前缀
30. PrefixJSON: []byte("macaron"),
31. // 渲染具有前缀的 XML, 默认为无前缀
32. PrefixXML: []byte("macaron"),
33. // 允许输出格式为 XHTML 而不是 HTML, 默认为 "text/html"
34. HTMLContentType: "text/html",
35. )))
36. // ...
37. }

```

## Pongo2 模板引擎

该服务可以通过函数 `pongo2.Pongoer` 来注入, 并通过类型 `macaron.Render` 来体现。该服务为可选, 一般情况下可直接使用 `*macaron.Context.Render`。该服务使用 Pongo2 v3 模板引擎来渲染 HTML。如果想要了解更多有关使用方面的信息, 请参见 [官方文档](#)。

## 使用示例

假设您的应用拥有以下目录结构:

```
1. main/
```

```

2.     |__ main.go
3.     |__ templates/
4.         |__ hello.tpl

```

hello.tpl:

```
1. <h1>Hello {{Name}}</h1>
```

main.go:

```

1. package main
2.
3. import (
4.     "github.com/go-macaron/pongo2"
5.     "gopkg.in/macaron.v1"
6. )
7.
8. func main() {
9.     m := macaron.Classic()
10.    m.Use(pongo2.Pongoer())
11.
12.    m.Get("/", func(ctx *macaron.Context) {
13.        ctx.Data["Name"] = "jeremy"
14.        ctx.HTML(200, "hello") // 200 is the response code.
15.    })
16.
17.    m.Run()
18. }

```

## 自定义选项

该服务允许接受一个参数来进行自定义选项 ( `pongo2.Options` ) :

```

1. package main
2.
3. import (
4.     "github.com/go-macaron/pongo2"

```

```

5.     "gopkg.in/macaron.v1"
6. )
7.
8. func main() {
9.     m := macaron.Classic()
10.    m.Use(pongo2.Pongoer(pongo2.Options{
11.        // 模板文件目录, 默认为 "templates"
12.        Directory: "templates",
13.        // 模板文件后缀, 默认为 [".tmpl", ".html"]
14.        Extensions: []string{".tmpl", ".html"},
15.        // 追加的 Content-Type 头信息, 默认为 "UTF-8"
16.        Charset: "UTF-8",
17.        // 渲染具有缩进格式的 JSON, 默认为不缩进
18.        IndentJSON: true,
19.        // 渲染具有缩进格式的 XML, 默认为不缩进
20.        IndentXML: true,
21.        // 允许输出格式为 XHTML 而不是 HTML, 默认为 "text/html"
22.        HTMLContentType: "text/html",
23.    })
24.    // ...
25. }

```

## 模板集

当您的应用存在多套模板时, 就需要使用模板集来实现运行时动态设置需要渲染的模板。

Go 模板引擎的使用方法:

```

1. // ...
2. m.Use(macaron.Renderers(macaron.RenderOptions{
3.     Directory: "templates/default",
4. }, "theme1:templates/theme1", "theme2:templates/theme2"))
5.
6. m.Get("/foobar", func(ctx *macaron.Context) {
7.     ctx.HTML(200, "hello")
8. })

```



```

9.
10. m.Get("/foobar1", func(ctx *macaron.Context) {
11.     ctx.HTMLSet(200, "theme1", "hello")
12. })
13.
14. m.Get("/foobar2", func(ctx *macaron.Context) {
15.     ctx.HTMLSet(200, "theme2", "hello")
16. })
17. // ...

```

## Pongo2 模板引擎的使用方法：

```

1. // ...
2. m.Use(pongo2.Pongoers(pongo2.Options{
3.     Directory: "templates/default",
4. }, "theme1:templates/theme1", "theme2:templates/theme2"))
5.
6. m.Get("/foobar", func(ctx *macaron.Context) {
7.     ctx.HTML(200, "hello")
8. })
9.
10. m.Get("/foobar1", func(ctx *macaron.Context) {
11.     ctx.HTMLSet(200, "theme1", "hello")
12. })
13.
14. m.Get("/foobar2", func(ctx *macaron.Context) {
15.     ctx.HTMLSet(200, "theme2", "hello")
16. })
17. // ...

```

正如您所看到的那样，其实就是 2 个方法的不同：

`macaron.Renderers`

和 `pongo2.Pongoers`。

第一个配置参数用于指定默认模板集和配置选项，之后则是一个模板集名称和目录（通过 `:` 分隔）的列表。

如果您的模板集名称和模板集路径的最后一部分相同，则可以省略名

称：

```

1. // ...
2. m.Use(macaron.Renderers(RenderOptions{
3.     Directory: "templates/default",
4. }, "templates/theme1", "templates/theme2"))
5.
6. m.Get("/foobar", func(ctx *macaron.Context) {
7.     ctx.HTML(200, "hello")
8. })
9.
10. m.Get("/foobar1", func(ctx *macaron.Context) {
11.     ctx.HTMLSet(200, "theme1", "hello")
12. })
13.
14. m.Get("/foobar2", func(ctx *macaron.Context) {
15.     ctx.HTMLSet(200, "theme2", "hello")
16. })
17. // ...

```

## 模板集辅助方法

检查某个模板集是否存在：

```

1. // ...
2. m.Get("/foobar", func(ctx *macaron.Context) {
3.     ok := ctx.HasTemplateSet("theme2")
4.     // ...
5. })
6. // ...

```

修改模板集的目录：

```

1. // ...
2. m.Get("/foobar", func(ctx *macaron.Context) {
3.     ctx.SetTemplatePath("theme2", "templates/new/theme2")
4.     // ...

```

```
5. })
6. // ...
```

## 小结

也许您已经发现，除了在 HTML 语法上的不同之外，两款引擎在代码层面的用法是完全一样的。

如果您只是想要得到 HTML 渲染后的结果，则可以调用方法

```
*macaron.Context.Render.HTMLString :
```

```
1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.Classic()
7.     m.Use(macaron.Renderer())
8.
9.     m.Get("/", func(ctx *macaron.Context) {
10.         ctx.Data["Name"] = "jeremy"
11.         output, err := ctx.HTMLString("hello")
12.         // 进行其它操作
13.     })
14.
15.     m.Run()
16. }
```

## 渲染 XML、JSON 和原始数据

相对于渲染 HTML 而言，渲染 XML、JSON 和原始数据的工作要简单的多。

```
1. package main
2.
```

```

3. import "gopkg.in/macaron.v1"
4.
5. type Person struct {
6.     Name string
7.     Age  int
8.     Sex  string
9. }
10.
11. func main() {
12.     m := macaron.Classic()
13.     m.Use(macaron.Renderer())
14.
15.     m.Get("/xml", func(ctx *macaron.Context) {
16.         p := Person{"Unknwon", 21, "male"}
17.         ctx.XML(200, &p)
18.     })
19.     m.Get("/json", func(ctx *macaron.Context) {
20.         p := Person{"Unknwon", 21, "male"}
21.         ctx.JSON(200, &p)
22.     })
23.     m.Get("/raw", func(ctx *macaron.Context) {
24.         ctx.RawData(200, []byte("raw data goes here"))
25.     })
26.     m.Get("/text", func(ctx *macaron.Context) {
27.         ctx.PlainText(200, []byte("plain text goes here"))
28.     })
29.
30.     m.Run()
31. }

```

## 响应状态码、错误和重定向

如果您希望响应指定状态码、错误和重定向操作，则可以参照以下代码：

```

1. package main
2.

```

```

3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.Classic()
7.     m.Use(macaron.Renderer())
8.
9.     m.Get("/status", func(ctx *macaron.Context) {
10.        ctx.Status(403)
11.    })
12.    m.Get("/error", func(ctx *macaron.Context) {
13.        ctx.Error(500, "Internal Server Error")
14.    })
15.    m.Get("/redirect", func(ctx *macaron.Context) {
16.        ctx.Redirect("/") // 第二个参数为响应码, 默认为 302
17.    })
18.
19.    m.Run()
20. }

```

## 运行时修改模板路径

如果您希望在运行时修改应用的模板路径，则可以调用方法

`*macaron.Context.SetTemplatePath`。需要注意的是，修改操作是全局生效的，而不只是针对当前请求。

## 使用示例

假设您的应用拥有以下目录结构：

```

1. main/
2.     |__ main.go
3.     |__ templates/
4.         |__ hello.tpl
5.     |__ templates2/
6.         |__ hello.tpl

```

templates/hello.tmpl:

```
1. <h1>Hello {{.Name}}</h1>
```

templates2/hello.tmpl:

```
1. <h1>What's up, {{.Name}}</h1>
```

main.go:

```
1. package main
2.
3. import "gopkg.in/macaron.v1"
4.
5. func main() {
6.     m := macaron.Classic()
7.     m.Use(macaron.Renderer())
8.
9.     m.Get("/old", func(ctx *macaron.Context) {
10.        ctx.Data["Name"] = "Unknwon"
11.        ctx.HTML(200, "hello")
12.        // 空字符串表示操作默认模板集
13.        ctx.SetTemplatePath("", "templates2")
14.    })
15.    m.Get("/new", func(ctx *macaron.Context) {
16.        ctx.Data["Name"] = "Unknwon"
17.        ctx.HTML(200, "hello")
18.    })
19.
20.    m.Run()
21. }
```

当您首次请求 `/old` 页面时，响应结果为 `<h1>Hello Unknwon</h1>`，然后便执行了修改模板路径为 `template2`。此时，当您请求 `/new` 页面时，响应结果会变成 `<h1>What's up,`

Unknwon</h1> 。

## 数据绑定与验证

- [数据绑定与验证](#)
  - [下载安装](#)
  - [使用示例](#)
    - [获取表单数据](#)
      - [命名约定](#)
    - [获取 JSON 数据](#)
    - [绑定到接口](#)
  - [处理器说明](#)
    - [Bind](#)
    - [Form](#)
    - [MultipartForm 和文件上传](#)
      - [使用示例](#)
    - [Json](#)
    - [Validate](#)
      - [验证规则](#)
  - [自定义操作](#)
    - [自定义验证](#)
    - [自定义验证规则](#)
    - [自定义错误处理](#)

## 数据绑定与验证

中间件 `binding` 为 [Macaron 实例](#) 提供了请求数据绑定与验证的功能。

- [GitHub](#)
- [API 文档](#)



## 下载安装

```
1. go get github.com/go-macaron/binding
```

## 使用示例

### 获取表单数据

假设您有一个联系人信息的表单，其中姓名和信息为必填字段，则我们可以使用如下结构来进行表示：

```
1. type ContactForm struct {
2.     Name          string `form:"name" binding:"Required"`
3.     Email          string `form:"email"`
4.     Message        string `form:"message" binding:"Required"`
5.     MailingAddress string `form:"mailing_address"`
6. }
```

然后通过 Macaron 增加如下路由：

```
1. m.Post("/contact/submit", binding.Bind(ContactForm{}), func(contact
   ContactForm) string {
2.     return fmt.Sprintf("Name: %s\nEmail: %s\nMessage: %s\nMailing
   Address: %v",
3.         contact.Name, contact.Email, contact.Message,
   contact.MailingAddress)
4. })
```

搞定！函数 `binding.Bind` 会帮助您完成对必选字段的数据验证。

默认情况下，如果在验证过程中发生任何错误（例如：必填字段的值为空），binding 中间件就会直接向客户端返回错误信息，提前终止请求的处理。如果您不希望 binding 中间件自动终止请求的处理，则

可以使用 `binding.BindIgnErr` 函数来忽略对错误的自动处理。

**警告** 请不要使用类型为指针的嵌入结构，这会导致错误。请查看 [martini-contrib/binding issue 30](#) 上的相关讨论获取完整信息。

## 命名约定

默认情况下，`form` 标签的名称使用以下命名约定：

- `Name` -> `name`
- `UnitPrice` -> `unit_price`

也就是说，上面例子中的结构定义可以简化为如下代码：

```
1. type ContactForm struct {
2.     Name      string `binding:"Required"`
3.     Email      string
4.     Message    string `binding:"Required"`
5.     MailingAddress string
6. }
```

超赞！有木有？

如果您想要自定义命名约定，可以通过 `binding.SetNameMapper` 函数来设置。该函数接受一个类型为 `binding.NameMapper` 的值作为参数。

## 获取 JSON 数据

将指定 `form` 标签的地方替换为 `json`，就可以完成对 JSON 数据的绑定。

友情提示 使用 [JSON-to-Go](#) 网站工具可以帮助您更好更快地得根据 JSON 数据生成 Go 语言中对应的结构。

## 绑定到接口

如果您希望传递接口而不是一个具体的结构，则可以使用如下方法：

```
1. m.Post("/contact/submit", binding.Bind(ContactForm{},
    (*MyInterface)(nil)), func(contact MyInterface) {
2.     // ... 您接收到的值为一个接口
3. })
```

## 处理器说明

原则上，每个处理器之间是相互独立的，但在特定情况下，它们之间会相互调用。

### Bind

函数 `binding.Bind` 是一个便利性的高层封装，它能够自动识别表单类型并完成数据绑定与验证。

请求处理流程：

1. 反序列化请求数据到结构
2. 通过 `binding.Validate` 函数完成数据验证
3. 如果您的结构实现了 `binding.ErrorHandler` 接口，则会调用相应的错误处理方法 `ErrorHandler.Error`；否则会使用默认的错误处理机制。

备注：

- 当使用默认的错误处理机制时，您的应用（队列后方的处理器）将根本不会意识到当前请求的存在。
- 头信息 `Content-Type` 是用于决定如何对请求数据进行反序列化的根本条件。

**重要安全提示** 请不要尝试绑定指向某个结构的指针，`binding` 中间件会直接 `panic` 并退出程序 [以防止可能发生的数据竞争](#)。

## Form

函数 `binding.Form` 用于反序列化表单数据，可以是查询或 `form-urlencoded` 类型的请求。

请求处理流程：

1. 反序列化请求数据到结构
2. 通过 `binding.Validate` 函数完成数据验证

需要注意的是，该函数不具有默认错误处理机制。您可以通过获取类型为 `binding.Errors` 的参数来完成自定义错误处理。

## MultipartForm 和文件上传

类似 `binding.Form`，函数 `binding.MultipartForm` 同样是反序列化表单数据到结构。除此之外，它还能处理 `enctype="multipart/form-data"` 类型的 POST 请求。如果结构中包含类型为 `*multipart.FileHeader`（或 `[]*multipart.FileHeader`）的字段，您可以直接从该字段读取客户端上传的文件。

请求处理流程：

1. 反序列化请求数据到结构
2. 通过 `binding.Validate` 函数完成数据验证

同样的，和函数 `binding.Form` 一样，该函数不具有默认错误处理机制，但您可以通过获取类型为 `binding.Errors` 的参数来完成自定义错误处理。

## 使用示例

```

1. type UploadForm struct {
2.     Title      string           `form:"title"`
3.     TextUpload *multipart.FileHeader `form:"txtUpload"`
4. }
5.
6. func main() {
7.     m := macaron.Classic()
8.     m.Post("/", binding.MultipartForm(UploadForm{}),
9.         uploadHandler(uf UploadForm) string {
10.             file, err := uf.TextUpload.Open()
11.             // ... 您可以在这里读取上传的文件内容
12.         })
13.     m.Run()
14. }
```

## Json

函数 `binding.Json` 反序列化 JSON 数据。

请求处理流程：

1. 反序列化请求数据到结构
2. 通过 `binding.Validate` 函数完成数据验证

与函数 `binding.Form`，该函数不具有默认错误处理机制，但您可以通过获取类型为 `binding.Errors` 的参数来完成自定义错误处理。

## Validate

函数 `binding.Validate` 接受一个结构并对它进行基本数据验证。如果该结构实现了 `binding.Validator` 接口，则会调用 `Validator.Validate()` 方法完成后续的数据验证。

## 验证规则

目前有一些内置的验证规则，通过格式为 `binding:"<Name>"` 的标签使用。

名称	说明
<code>OmitEmpty</code>	值为空时忽略后续验证
<code>Required</code>	必须为相同类型的非零值
<code>AlphaDash</code>	必须为半角英文字母、阿拉伯数字或 <code>-_</code>
<code>AlphaDashDot</code>	必须为半角英文字母、阿拉伯数字、 <code>-_</code> 或 <code>.</code>
<code>Size(int)</code>	固定长度
<code>MinSize(int)</code>	最小长度
<code>MaxSize(int)</code>	最大长度
<code>Range(int,int)</code>	取值范围（包含边界值）
<code>Email</code>	必须为邮箱地址
<code>Url</code>	必须为 HTTP/HTTPS URL 地址
<code>In(a,b,c,...)</code>	必须为数组的一个元素
<code>NotIn(a,b,c,...)</code>	必须不是数组的元素
<code>Include(string)</code>	必须包含
<code>Exclude(string)</code>	必须不包含
<code>Default(string)</code>	当字段为零值时设置默认值（当使用接口绑定时不能设置该规则）

当需要使用多条规则时：`binding:"Required;MinSize(10)"`。

## 自定义操作

### 自定义验证

如果您想要进行自定义的附加验证操作，您的结构可以通过实现接口

`binding.Validator` 来完成：

```
1. func (cf ContactForm) Validate(ctx *macaron.Context, errs
   binding.Errors) binding.Errors {
2.     if strings.Contains(cf.Message, "Go needs generics") {
```

```

3.         errs = append(errors, binding.Error{
4.             FieldNames:    []string{"message"},
5.             Classification: "ComplaintError",
6.             Message:        "Go has generics. They're called
           interfaces.",
7.         })
8.     }
9.     return errs
10. }

```

现在，任何包含信息 “Go needs generics” 的联系人表单都会报错。

## 自定义验证规则

当您觉得内置的验证规则不够时，可以通过函数

`binding.AddParamRule`

来增加自定义验证规则。该函数接受一个类型

为

`binding.ParamRule`

的参数。

假设您需要验证字段的最小值：

```

1. binding.AddParamRule(&binding.ParamRule{
2.     IsMatch: func(rule string) bool {
3.         return strings.HasPrefix(rule, "Min(")
4.     },
5.     IsValid: func(errs binding.Errors, rule, name string, v
           interface{}) (bool, binding.Errors) {
6.         num, ok := v.(int)
7.         if !ok {
8.             return false, errs
9.         }
10.        min, _ := strconv.Atoi(rule[4 : len(rule)-1])
11.        if num < min {
12.            errs.Add([]string{name}, "MinimumValue", "Value is too
           small")
13.            return false, errs

```

```

14.         }
15.         return true, errs
16.     },
17. })

```

如果您的规则非常简单，也可以使用 `binding.AddRule`，它接受类型为 `binding.Rule` 的参数：

```

1. binding.AddRule(&binding.Rule{
2.     IsMatch: func(rule string) bool {
3.         return rule == "String"
4.     },
5.     IsValid: func(errs binding.Errors, name string, v interface{})
6.         (bool, binding.Errors) {
7.         _, ok := v.(string)
8.         return ok, errs
9.     },
10. })

```

自定义规则的应用发生在内置规则之后。

## 自定义错误处理

如果您即不想使用默认的错误处理机制，又希望 `binding` 中间件自动化地调用您的自定义错误处理，则可以通过实现接口

`binding.ErrorHandler` 来完成：

```

1. func (cf ContactForm) Error(ctx *macaron.Context, errs
2.     binding.Errors) {
3.     // 自定义错误处理过程
4. }

```

该操作发生在自定义验证规则被应用之后。





## 本地化您的应用

- [本地化您的应用](#)
  - [下载安装](#)
  - [使用示例](#)
    - [Pongo2 模板引擎](#)
  - [自定义选项](#)
  - [加载本地化文件](#)
  - [其它说明](#)

## 本地化您的应用

中间件 `i18n` 为 [Macaron 实例](#) 提供了国际化和本地化应用的功能。

- [GitHub](#)
- [API 文档](#)

## 下载安装

```
1. go get github.com/go-macaron/i18n
```

## 使用示例

```
1. // main.go
2. import (
3.     "github.com/go-macaron/i18n"
4.     "gopkg.in/macaron.v1"
5. )
6.
7. func main() {
```

```

8.     m := macaron.Classic()
9.     m.Use(i18n.I18n(i18n.Options{
10.         Langs:    []string{"en-US", "zh-CN"},
11.         Names:    []string{"English", "简体中文"},
12.     }))
13.
14.     m.Get("/", func(locale i18n.Locale) string {
15.         return "current language is" + locale.Lang
16.     })
17.
18.     // 在处理器中使用
19.     m.Get("/trans", func(ctx *macaron.Context) string {
20.         return ctx.Tr("hello %s", "world")
21.     })
22.
23.     m.Run()
24. }

```

```

1. <!-- templates/hello.tpl -->
2. <h2>{{i18n.Tr "hello %s" "world"}}!</h2>

```

## Pongo2 模板引擎

在 [pongo2](#) 模板引擎中使用 `i18n` 中间件：

```

1. <!-- templates/hello.tpl -->
2. <h2>{{Tr(Lang, "hello %s", "world")}}!</h2>

```

## 自定义选项

该服务允许接受一个参数来进行自定义选项（`i18n.Options`）：

```

1. // ...
2. m.Use(i18n.I18n(i18n.Options{
3.     // 存放本地化文件的目录，默认为 "conf/locale"

```

```

4.     Directory:    "conf/locale",
5.     // 支持的语言，顺序是有意义的
6.     Langs:        []string{"en-US", "zh-CN"},
7.     // 语言的本地化名称
8.     Names:        []string{"English", "简体中文"},
9.     // 本地化文件命名风格，默认为 "locale_%s.ini"
10.    Format:        "locale_%s.ini",
11.    // 指示当前语言的 URL 参数名，默认为 "lang"
12.    Parameter:     "lang",
13.    // 当通过 URL 参数指定语言时是否重定向，默认为 false
14.    Redirect:      false,
15.    // 存放在模板中的本地化对象变量名称，默认为 "i18n"
16.    TmplName:      "i18n",
17.  )))
18.  // ...

```

## 加载本地化文件

默认情况下，本地化文件应当存放在相对当前目录的 `conf/locale` 文件夹下：

```

1.  conf/
2.    |
3.    |__ locale/
4.        |
5.        |__ locale_en-US.ini
6.        |
7.        |__ locale_zh-CN.ini

```

## 其它说明

- 请查看 [Unknwon/i18n](#) 包来了解本地化使用规范。
- 您可以将 [Peach](#) 作为学习案例。



## 缓存管理 (Cache)

- [缓存管理 \(Cache\)](#)
  - [下载安装](#)
  - [使用示例](#)
  - [自定义选项](#)
  - [适配器](#)
    - [内存](#)
    - [文件](#)
    - [Redis](#)
    - [Memcache](#)
    - [PostgreSQL](#)
    - [MySQL](#)
    - [Ledis](#)
    - [Nodb](#)

## 缓存管理 (Cache)

---

中间件 `cache` 为 [Macaron 实例](#) 提供了缓存管理的功能。

- [GitHub](#)
- [API Reference](#)

## 下载安装

---

```
1. go get github.com/go-macaron/cache
```

## 使用示例

---

```

1. import (
2.     "github.com/go-macaron/cache"
3.     "gopkg.in/macaron.v1"
4. )
5.
6. func main() {
7.     m := macaron.Classic()
8.     m.Use(cache.Cacher())
9.
10.    m.Get("/", func(c cache.Cache) string {
11.        c.Put("cache", "cache middleware", 120)
12.        return c.Get("cache")
13.    })
14.
15.    m.Run()
16. }

```

## 自定义选项

该服务允许接受一个参数来进行自定义选项（ `cache.Options` ）：

```

1. //...
2. m.Use(cache.Cacher(cache.Options{
3.     // 适配器的名称，默认为 "memory".
4.     Adapter:      "memory",
5.     // 适配器的配置，根据适配器而不同
6.     AdapterConfig: "",
7.     // GC 执行时间间隔，默认为 60 秒
8.     Interval:     60,
9.     // 配置分区名称，默认为 "cache"
10.    Section:      "cache",
11.    }))
12. //...

```

## 适配器

目前有 8 款内置的适配器，除了 内存 和 文件 提供器外，您都必须显式导入其它适配器的驱动。

以下为适配器的基本用法：

## 内存

```
1. //...
2. m.Use(cache.Cacher())
3. //...
```

## 文件

```
1. //...
2. m.Use(cache.Cacher(cache.Options{
3.     Adapter:      "file",
4.     AdapterConfig: "data/caches",
5. })))
6. //...
```

## Redis

特别注意 只能存取 string 和 int 相关类型。

```
1. import _ "github.com/go-macaron/cache/redis"
2.
3. //...
4. m.Use(cache.Cacher(cache.Options{
5.     Adapter:      "redis",
6.     // e.g.:
7.     network=tcp, addr=127.0.0.1:6379, password=macaron, db=0, pool_size=100, i
8.     AdapterConfig: "addr=127.0.0.1:6379,password=macaron",
9.     OccupyMode:    false,
10. })))
10. //...
```





当您使用 Redis 作为缓存器时，可以通过将 `OccupyMode` 的值设置为 `true` 来启用独占模式。在该模式下，缓存器将直接占用所选用的整个数据库，而不是通过维护一个索引集合来判断哪些数据是属于您的应用的。当您的缓存数据非常巨大时，该模式可以有效降低应用的 CPU 和内存使用率。

## Memcache

```
1. import _ "github.com/go-macaron/cache/memcache"
2.
3. //...
4. m.Use(cache.Cacher(cache.Options{
5.     Adapter:      "memcache",
6.     // e.g.: 127.0.0.1:9090;127.0.0.1:9091
7.     AdapterConfig: "127.0.0.1:11211",
8. }))
9. //...
```

## PostgreSQL

可以使用以下 SQL 语句创建数据库：

```
1. CREATE TABLE cache (
2.     key      CHAR(32) NOT NULL,
3.     data     BYTEA,
4.     created  INTEGER NOT NULL,
5.     expire   INTEGER NOT NULL,
6.     PRIMARY KEY (key)
7. );
```

```
1. import _ "github.com/go-macaron/cache/postgres"
2.
3. //...
```

```

4. m.Use(cache.Cacher(cache.Options{
5.     Adapter:      "postgres",
6.     AdapterConfig: "user=a password=b host=localhost port=5432
       dbname=c sslmode=disable",
7. }))
8. //...

```

## MySQL

可以使用以下 SQL 语句创建数据库：

```

1. CREATE TABLE `cache` (
2.     `key`      CHAR(32) NOT NULL,
3.     `data`     BLOB,
4.     `created`  INT(11) UNSIGNED NOT NULL,
5.     `expire`   INT(11) UNSIGNED NOT NULL,
6.     PRIMARY KEY (`key`)
7. ) ENGINE=MyISAM DEFAULT CHARSET=utf8;

```

```

1. import _ "github.com/go-macaron/cache/mysql"
2.
3. //...
4. m.Use(cache.Cacher(cache.Options{
5.     Adapter:      "mysql",
6.     AdapterConfig: "username:password@protocol(address)/dbname?
       param=value",
7. }))
8. //...

```

## Ledis

```

1. import _ "github.com/go-macaron/cache/ledis"
2.
3. //...
4. m.Use(cache.Cacher(cache.Options{
5.     Adapter:      "ledis",

```

```
6.     AdapterConfig: "data_dir=./app.db,db=0",
7.   })))
8.   //...
```

## Nodb

```
1. import _ "github.com/go-macaron/cache/nodb"
2.
3. //...
4. m.Use(cache.Cacher(cache.Options{
5.     Adapter:      "nodb",
6.     AdapterConfig: "data/cache.db",
7.   })))
8.   //...
```

## 会话管理 (Session)

- [会话管理 \(Session\)](#)
  - [下载安装](#)
  - [使用示例](#)
    - [Pongo2](#)
    - [将 Flash 输出到当前响应](#)
  - [自定义选项](#)
  - [提供器](#)
    - [内存](#)
    - [文件](#)
    - [Redis](#)
    - [Memcache](#)
    - [PostgreSQL](#)
    - [MySQL](#)
    - [Couchbase](#)
    - [Ledis](#)
    - [Nodb](#)
  - [实现提供器接口](#)

## 会话管理 (Session)

---

中间件 `session` 为 [Macaron 实例](#) 提供了会话管理的功能。

- [GitHub](#)
- [API 文档](#)

## 下载安装

---

```
1. go get github.com/go-macaron/session
```

## 使用示例

```
1. import (
2.     "github.com/go-macaron/session"
3.     "gopkg.in/macaron.v1"
4. )
5.
6. func main() {
7.     m := macaron.Classic()
8.     m.Use(macaron.Renderer())
9.     m.Use(session.Sessioner())
10.
11.    m.Get("/", func(sess session.Store) string {
12.        sess.Set("session", "session middleware")
13.        return sess.Get("session").(string)
14.    })
15.
16.    m.Get("/signup", func(ctx *macaron.Context, f *session.Flash) {
17.        f.Success("yes!!!")
18.        f.Error("opps...")
19.        f.Info("aha?!")
20.        f.Warning("Just be careful.")
21.        ctx.HTML(200, "signup")
22.    })
23.
24.    m.Run()
25. }
```

```
1. <!-- templates/signup.tmpl -->
2. <h2>{{.Flash.SuccessMsg}}</h2>
3. <h2>{{.Flash.ErrorMsg}}</h2>
4. <h2>{{.Flash.InfoMsg}}</h2>
5. <h2>{{.Flash.WarningMsg}}</h2>
```

## Pongo2

如果您正在使用 `pongo2` 作为应用的模板引擎，则需要对 HTML 进行如下修改：

```
1. <!-- templates/signup.tpl -->
2. <h2>{{Flash.SuccessMsg}}</h2>
3. <h2>{{Flash.ErrorMsg}}</h2>
4. <h2>{{Flash.InfoMsg}}</h2>
5. <h2>{{Flash.WarningMsg}}</h2>
```

## 将 Flash 输出到当前响应

默认情况下，Flash 的数据只会在相对应会话的下一个响应中使用，但函数 `Success`、`Error`、`Info` 和 `Warning` 均接受第二个参数来指示是否在当前响应输出数据：

```
1. // ...
2. f.Success("yes!!!", true)
3. f.Error("oops...", true)
4. f.Info("aha?!", true)
5. f.Warning("Just be careful.", true)
6. // ...
```

但是请注意，不管您选择什么时候输出 Flash 的数据，它都只能够被使用一次。

## 自定义选项

该服务允许接受一个参数来进行自定义选项（`session.Options`）：

```
1. //...
2. m.Use(session.Sessioner(session.Options{
3.     // 提供器的名称，默认为 "memory"
```

```

4.     Provider:      "memory",
5.     // 提供器的配置, 根据提供器而不同
6.     ProviderConfig: "",
7.     // 用于存放会话 ID 的 Cookie 名称, 默认为 "MacaronSession"
8.     CookieName:     "MacaronSession",
9.     // Cookie 储存路径, 默认为 "/"
10.    CookiePath:      "/",
11.    // GC 执行时间间隔, 默认为 3600 秒
12.    Gclifetime:       3600,
13.    // 最大生存时间, 默认和 GC 执行时间间隔相同
14.    Maxlifetime:      3600,
15.    // 仅限使用 HTTPS, 默认为 false
16.    Secure:           false,
17.    // Cookie 生存时间, 默认为 0 秒
18.    CookieLifeTime:  0,
19.    // Cookie 储存域名, 默认为空
20.    Domain:           "",
21.    // 会话 ID 长度, 默认为 16 位
22.    IDLength:         16,
23.    // 配置分区名称, 默认为 "session"
24.    Section:          "session",
25.  )))
26.  //...

```

## 提供器

目前有 9 款内置的提供器, 除了 内存 和 文件 提供器外, 您都必须显式导入其它提供器的驱动。

以下为提供器的基本用法:

## 内存

```

1.  //...
2.  m.Use(session.Sessioner())
3.  //...

```

## 文件

```

1. //...
2. m.Use(session.Sessioner(session.Options{
3.     Provider:      "file",
4.     ProviderConfig: "data/sessions",
5. }))
6. //...

```

## Redis

```

1. import _ "github.com/go-macaron/session/redis"
2.
3. //...
4. m.Use(session.Sessioner(session.Options{
5.     Provider:      "redis",
6.     // e.g.:
       network=tcp,addr=127.0.0.1:6379,password=macaron,db=0,pool_size=100,i
7.     ProviderConfig: "addr=127.0.0.1:6379,password=macaron",
8. }))
9. //...

```

## Memcache

```

1. import _ "github.com/go-macaron/session/memcache"
2.
3. //...
4. m.Use(session.Sessioner(session.Options{
5.     Provider:      "memcache",
6.     // e.g.: 127.0.0.1:9090;127.0.0.1:9091
7.     ProviderConfig: "127.0.0.1:9090",
8. }))
9. //...

```



## PostgreSQL

可以使用以下 SQL 语句创建数据库（请确保 `key` 的长度和您设置的 `Options.IDLength` 一致）：

```
1. CREATE TABLE session (
2.     key      CHAR(16) NOT NULL,
3.     data     BYTEA,
4.     expiry   INTEGER NOT NULL,
5.     PRIMARY KEY (key)
6. );
```

```
1. import _ "github.com/go-macaron/session/postgres"
2.
3. //...
4. m.Use(session.Sessioner(session.Options{
5.     Provider:      "postgres",
6.     ProviderConfig: "user=a password=b dbname=c sslmode=disable",
7. }))
8. //...
```

## MySQL

可以使用以下 SQL 语句创建数据库：

```
1. CREATE TABLE `session` (
2.     `key`      CHAR(16) NOT NULL,
3.     `data`     BLOB,
4.     `expiry`   INT(11) UNSIGNED NOT NULL,
5.     PRIMARY KEY (`key`)
6. ) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

```
1. import _ "github.com/go-macaron/session/mysql"
2.
3. //...
```

```

4. m.Use(session.Sessioner(session.Options{
5.     Provider:      "mysql",
6.     ProviderConfig: "username:password@protocol(address)/dbname?
       param=value",
7. }))
8. //...

```

## Couchbase

```

1. import _ "github.com/go-macaron/session/couchbase"
2.
3. //...
4. m.Use(session.Sessioner(session.Options{
5.     Provider:      "couchbase",
6.     ProviderConfig: "username:password@protocol(address)/dbname?
       param=value",
7. }))
8. //...

```

## Ledis

```

1. import _ "github.com/go-macaron/session/ledis"
2.
3. //...
4. m.Use(session.Sessioner(session.Options{
5.     Provider:      "ledis",
6.     ProviderConfig: "data_dir=./app.db,db=0",
7. }))
8. //...

```

## Nodb

```

1. import _ "github.com/go-macaron/session/nodb"
2.
3. //...

```

```

4. m.Use(session.Sessioner(session.Options{
5.     Provider:      "nodb",
6.     ProviderConfig: "data/cache.db",
7. })))
8. //...

```

## 实现提供器接口

如果您需要实现自己的会话存储和提供器，可以通过实现下面两个接口实现，同时还可以将 内存 提供器作为学习案例。

```

1. // RawStore is the interface that operates the session data.
2. type RawStore interface {
3.     // Set sets value to given key in session.
4.     Set(key, value interface{}) error
5.     // Get gets value by given key in session.
6.     Get(key interface{}) interface{}
7.     // Delete deletes a key from session.
8.     Delete(key interface{}) error
9.     // ID returns current session ID.
10.    ID() string
11.    // Release releases session resource and save data to provider.
12.    Release() error
13.    // Flush deletes all session data.
14.    Flush() error
15. }
16.
17. // Provider is the interface that provides session manipulations.
18. type Provider interface {
19.     // Init initializes session provider.
20.     Init(gclifetime int64, config string) error
21.     // Read returns raw session store by session ID.
22.     Read(sid string) (RawStore, error)
23.     // Exist returns true if session with given ID exists.
24.     Exist(sid string) bool
25.     // Destory deletes a session by session ID.
26.     Destory(sid string) error

```

```
27.      // Regenerate regenerates a session store from old session ID
        to new one.
28.      Regenerate(oldsid, sid string) (RawStore, error)
29.      // Count counts and returns number of sessions.
30.      Count() int
31.      // GC calls GC to clean expired sessions.
32.      GC()
33. }
```

## 跨域请求攻击 (CSRF)

- [跨域请求攻击 \(CSRF\)](#)
  - [下载安装](#)
  - [使用示例](#)
  - [自定义选项](#)

## 跨域请求攻击 (CSRF)

中间件 `csrf` 用于为 [Macaron 实例](#) 生成和验证 CSRF 令牌。

- [GitHub](#)
- [API 文档](#)

## 下载安装

```
1. go get github.com/go-macaron/csrf
```

## 使用示例

想要使用该中间件，您必须同时使用 [session](#) 中间件。

```
1. package main
2.
3. import (
4.     "github.com/go-macaron/csrf"
5.     "github.com/go-macaron/session"
6.     "gopkg.in/macaron.v1"
7. )
8.
9. func main() {
10.     m := macaron.Classic()
11.     m.Use(macaron.Renderer())
```

```

12.     m.Use(session.Sessioner())
13.     m.Use(csrf.Csrf())
14.
15.     // 模拟验证过程, 判断 session 中是否存在 uid 数据。
16.     // 若不存在, 则跳转到一个生成 CSRF 的页面。
17.     m.Get("/", func(ctx *macaron.Context, sess session.Store) {
18.         if sess.Get("uid") == nil {
19.             ctx.Redirect("/login")
20.             return
21.         }
22.         ctx.Redirect("/protected")
23.     })
24.
25.     // 设置 session 中的 uid 数据。
26.     m.Get("/login", func(ctx *macaron.Context, sess session.Store)
27.     {
28.         sess.Set("uid", 123456)
29.         ctx.Redirect("/")
30.     })
31.
32.     // 渲染一个需要验证的表单, 并传递 CSRF 令牌到表单中。
33.     m.Get("/protected", func(ctx *macaron.Context, sess
34.     session.Store, x csrf.CSRF) {
35.         if sess.Get("uid") == nil {
36.             ctx.Redirect("/login", 401)
37.             return
38.         }
39.
40.         ctx.Data["csrf_token"] = x.GetToken()
41.         ctx.HTML(200, "protected")
42.     })
43.
44.     // 验证 CSRF 令牌。
45.     m.Post("/protected", csrf.Validate, func(ctx *macaron.Context,
46.     sess session.Store) {
47.         if sess.Get("uid") != nil {
48.             ctx.RenderData(200, []byte("You submitted a valid
49.             token"))

```

```

46.         return
47.     }
48.     ctx.Redirect("/login", 401)
49. })
50.
51. m.Run()
52. }

```

```

1. <!-- templates/protected.tpl -->
2. <form action="/protected" method="post">
3.     <input type="hidden" name="_csrf" value="{{.csrf_token}}">
4.     <button>提交</button>
5. </form>

```

## 自定义选项

该服务允许接受一个参数来进行自定义选项 ( `csrf.Options` ) :

```

1. // ...
2. m.Use(csrf.Csrfer(csrf.Options{
3.     // 用于生成令牌的全局密钥, 默认为随机字符串
4.     Secret:      "mysecret",
5.     // 用于传递令牌的 HTTP 请求头信息字段, 默认为 "X-CSRFToken"
6.     Header:      "X-CSRFToken",
7.     // 用于传递令牌的表单字段名, 默认为 "_csrf"
8.     Form:        "_csrf",
9.     // 用于传递令牌的 Cookie 名称, 默认为 "_csrf"
10.    Cookie:       "_csrf",
11.    // Cookie 设置路径, 默认为 "/"
12.    CookiePath:   "/",
13.    // 用于保存用户 ID 的 session 名称, 默认为 "uid"
14.    SessionKey:   "uid",
15.    // 用于指定是否将令牌设置到响应的头信息中, 默认为 false
16.    SetHeader:    false,
17.    // 用于指定是否将令牌设置到响应的 Cookie 中, 默认为 false
18.    SetCookie:    false,

```

```
19.      // 用于指定是否要求只有使用 HTTPS 时才设置 Cookie, 默认为 false
20.      Secure:      false,
21.      // 用于禁止请求头信息中包括 Origin 字段, 默认为 false
22.      Origin:      false,
23.      // 错误处理函数, 默认为简单的错误输出
24.      ErrorFunc:   func(w http.ResponseWriter) {
25.          http.Error(w, "Invalid csrf token.", http.StatusBadRequest)
26.      },
27.      )))
28.  // ...
```



## 验证码服务

- [验证码服务](#)
  - [下载安装](#)
- [使用示例](#)
- [自定义选项](#)

## 验证码服务

中间件 `captcha` 用于为 [Macaron 实例](#) 提供验证码服务。

- [GitHub](#)
- [API 文档](#)

## 下载安装

```
1. go get github.com/go-macaron/captcha
```

## 使用示例

想要使用该中间件，您必须同时使用 [cache](#) 中间件。

```
1. // main.go
2. import (
3.     "github.com/go-macaron/cache"
4.     "github.com/go-macaron/captcha"
5.     "gopkg.in/macaron.v1"
6. )
7.
8. func main() {
9.     m := macaron.Classic()
10.    m.Use(cache.Cacher())
11.    m.Use(captcha.Captchaer())
```

```

12.
13.     m.Get("/", func(ctx *macaron.Context, cpt *captcha.Captcha)
        string {
14.         if cpt.VerifyReq(ctx.Req) {
15.             return "valid captcha"
16.         }
17.         return "invalid captcha"
18.     })
19.
20.     m.Run()
21. }

```

```

1. <!-- templates/hello.tpl -->
2. {{.Captcha.CreateHtml}}

```

## 自定义选项

该服务允许接受一个参数来进行自定义选项（ `captcha.Options` ）：

```

1. // ...
2. m.Use(captcha.Captchaer(captcha.Options{
3.     // 获取验证码图片的 URL 前缀, 默认为 "/captcha/"
4.     URLPrefix:         "/captcha/",
5.     // 表单隐藏元素的 ID 名称, 默认为 "captcha_id"
6.     FieldIdName:       "captcha_id",
7.     // 用户输入验证码值的元素 ID, 默认为 "captcha"
8.     FieldCaptchaName:  "captcha",
9.     // 验证字符的个数, 默认为 6
10.    ChallengeNums:      6,
11.    // 验证码图片的宽度, 默认为 240 像素
12.    Width:              240,
13.    // 验证码图片的高度, 默认为 80 像素
14.    Height:             80,
15.    // 验证码过期时间, 默认为 600 秒
16.    Expiration:         600,
17.    // 用于存储验证码正确值的 Cache 键名, 默认为 "captcha_"

```

```
18.     CachePrefix:      "captcha_",  
19.  )))  
20.  // ...
```

# 嵌入二进制数据

- [name：二进制数据](#)
- [嵌入二进制数据](#)
  - [下载安装](#)
- [使用示例](#)

## name：二进制数据

# 嵌入二进制数据

模块 `bindata` 用于为 [Macaron 实例](#) 提供支持内存的静态文件服务和模板文件系统。

- [GitHub](#)
- [API 文档](#)

## 下载安装

```
1. go get github.com/go-macaron/bindata
```

## 使用示例

使用 `go-bindata` 将相应的静态文件和模板文件转换成单独的包。

导入相应的包并通过如下方法实现支持：

```
1. import (  
2.     "path/to/bindata/public"  
3.     "path/to/bindata/templates"  
4.     "github.com/go-macaron/bindata"
```

```
5. )
6.
7. m.Use(macaron.Static("public",
8.     macaron.StaticOptions{
9.         FileSystem: bindata.Static(bindata.Options{
10.             Asset:      public.Asset,
11.             AssetDir:   public.AssetDir,
12.             AssetNames: public.AssetNames,
13.             Prefix:     "",
14.         }),
15.     },
16. ))
17.
18. m.Use(macaron.Renderer(macaron.RenderOptions{
19.     TemplateFileSystem: bindata.Templates(bindata.Options{
20.         Asset:      templates.Asset,
21.         AssetDir:   templates.AssetDir,
22.         AssetNames: templates.AssetNames,
23.         Prefix:     "",
24.     }),
25. })))
```

## 多站点支持

- [name：多站点支持](#)
- [在一个应用内服务多个站点](#)
  - [下载安装](#)
  - [使用示例](#)
    - [动态匹配](#)

---

### name：多站点支持

---

## 在一个应用内服务多个站点

辅助模块 `switcher` 为您的应用提供多个 [Macaron 实例](#) 的支持。

- [GitHub](#)
- [API 文档](#)

## 下载安装

```
1. go get github.com/go-macaron/switcher
```

## 使用示例

如果您想要运行 2 个或 2 个以上的 [Macaron 实例](#) 在一个程序中，该辅助模块便可为此类需求提供便利：

```
1. func main() {  
2.     m1 := macaron.Classic()  
3.     // 注册 m1 实例的中间件和路由  
4.
```

```

5.     m2 := macaron.Classic()
6.     // 注册 m2 实例的中间件和路由
7.
8.     hs := switcher.NewHostSwitcher()
9.     // 设置实例所对应的主机地址
10.    hs.Set("gowalker.org", m1)
11.    hs.Set("gogs.io", m2)
12.    hs.Run()
13. }

```

默认情况下，即 `macaron.DEV` 模式，出于对调试的便利性，该程序会监听多个端口，包括 `4000`（用于实例 `m1`）和 `4001`（用于实例 `m2`）。而当模式为 `macaron.PROD` 时，则只会监听一个端口，即 `4000`。

## 动态匹配

如果您有多个子域名需要使用一个 Macaron 实例来处理，则可以通过以下方式来动态匹配：

```

1. // ...
2. m := macaron.Classic()
3. // 注册 m 实例的中间件和路由
4.
5. hs := macaron.NewHostSwitcher()
6. // 设置实例所对应的主机地址
7. hs.Set("*.example.com", m)
8. hs.Run()
9. // ...

```

## 高级用法

- 自定义服务
  - 全局映射
  - 请求级别的映射
  - 映射值到接口

## 自定义服务

服务即是被注入到处理器中的参数。你可以映射一个服务到 全局 或者 请求 的级别。

### 全局映射

因为 Macaron 实现了 `inject.Injector` 的接口，那么映射一个服务就变得非常简单：

```
1. db := &MyDatabase{}
2. m := macaron.Classic()
3. m.Map(db) // Service will be available to all handlers as
   *MyDatabase
4. m.Get("/", func(db *MyDatabase) {
5.     // Operations with db.
6. })
7. m.Run()
```

### 请求级别的映射

映射在请求级别的服务可以通过 `*macaron.Context` 来完成：

```
1. func MyCustomLoggerHandler(ctx *macaron.Context) {
2.     logger := &MyCustomLogger{ctx.Req}
3.     ctx.Map(logger) // mapped as *MyCustomLogger
4. }
```



```

5.
6. func main() {
7.     //...
8.     m.Get("/", MyCustomLoggerHandler, func(logger *MyCustomLogger)
9.     {
10.         // Operations with logger.
11.     })
12.     m.Get("/panic", func(logger *MyCustomLogger) {
13.         // This will panic because no logger service maps to this
14.         request.
15.     })
16.     //...
17. }

```

## 映射值到接口

关于服务最强悍的地方之一就是它能够映射服务到接口。例如说，假设你想要覆盖 `http.ResponseWriter` 成为一个对象，那么你可以封装它并包含你自己的额外操作，你可以如下这样来编写你的处理器：

```

1. func WrapResponseWriter(ctx *macaron.Context) {
2.     rw := NewSpecialResponseWriter(ctx.Resp)
3.     // override ResponseWriter with our wrapper ResponseWriter
4.     ctx.MapTo(rw, (*http.ResponseWriter)(nil))
5. }

```

如此一来，您不仅可以修改自定义的实现而不对客户代码做任何修改，还可以允许对于相同类型的服务使用多种实现。

## 常见问题

- 常见问题
  - 如何集成到我已有的服务中？
  - 如何修改监听地址和端口？
  - 如何优雅地终止程序（Graceful Shutdown）？
  - 除了注入服务以外，如何在同一个请求内传递数据？
  - 为什么不直接使用 Martini 而要另外创建一个框架？
  - 为什么 Logo 是一条龙？
  - 有代码实时编译运行工具吗？

## 常见问题

---

### 如何集成到我已有的服务中？

每个 **Macaron 实例** 都实现了 `http.Handler` 接口，因此可以很容易地将它们以子集的形式集成到已有服务中。例如，您可以将 Macaron 应用集成到 GAE 中：

```
1. package hello
2.
3. import (
4.     "net/http"
5.
6.     "gopkg.in/macaron.v1"
7. )
8.
9. func init() {
10.     m := macaron.Classic()
11.     m.Get("/", func() string {
12.         return "Hello world!"
13.     })
14.     http.Handle("/", m)
```

```
15. }
```

## 如何修改监听地址和端口？

Macaron 的 `Run` 函数会首先根据环境变量 `PORT` 和 `HOST` 来确定监听地址和端口。如果未找到相应设置，则会默认使用 `localhost:4000`。如果您想要更加灵活便利的方案，可以使用 `http.ListenAndServe` 函数来实现。

```
1. m := macaron.Classic()
2. // ...
3. log.Fatal(http.ListenAndServe(":8080", m))
```

或者以下方式：

- `m.Run("0.0.0.0")`，监听在 `0.0.0.0:4000`
- `m.Run(8080)`，监听在 `0.0.0.0:8080`
- `m.Run("0.0.0.0", 8080)`，监听在 `0.0.0.0:8080`

## 如何优雅地终止程序（Graceful Shutdown）？

```
1. package main
2.
3. import (
4.     ...
5.     "net/http"
6.
7.     "gopkg.in/macaron.v1"
8.     "gopkg.in/tylerb/graceful.v1"
9. )
10.
11. func main() {
12.     m := macaron.Classic()
13.
```

```

14.     ...
15.
16.     mux := http.NewServeMux()
17.     mux.Handle("/", m)
18.     graceful.Run(":4000", 60*time.Second, mux)
19. }

```

除了注入服务以外，如何在同一个请求内传递数据？

对象 `*macaron.Context` 中包含一个类型为 `map[string]interface{}` 的字段 `Data` 可供您在同个请求的不同处理器之间传递数据。

可以到 [这里](#) 查看使用方法。

为什么不直接使用 Martini 而要另外创建一个框架？

- 集成常用组件和方法来减少反射次数。
- 使用速度更快的多叉树路由替换原本的路由层。
- 更好地驱动 Gogs 项目。
- 对 Martini 源码进行一次深度学习。

为什么 Logo 是一条龙？

不应该是一种甜品吗？

正所谓 `马卡龙`，此龙乃是名为 `马卡` 的龙，哈哈！

有代码实时编译运行工具吗？

[Bra](#) 可以作为 Macaron 及其它应用的实时编译运行工具。

