目　录

- 关于
- 安装
- 开始使用
- Host
  - 带有TLS的自动公共域名
- 配置
- 路由
  - 路径参数类型
  - 反向查询
  - 中间件
  - 处理HTTP错误
  - 子域名
  - 包装路由器
  - 重写Context
  - Context方法
- API版本控制
- 内容协商
- 响应记录器
- HTTP Referer
- 请求认证
- URL查询参数
- 表单
- 模型验证
- 缓存
- 文件服务
- 视图
- Cookies
- Sessions
- Websockets
- MVC
- 测试

关于

`Iris` 是一个通过 Go 编写的快速的，简单的，但是功能齐全和非常有效率的 web 框架。

`Iris` 为你下一个网站或者 API 提供了一个精美的、使用简单的基础。

`Iris` 为它的使用者提供了一个完整且体面的支持。

我们的哲学

`Iris` 的哲学是为 HTTP 提供强大的工具，使其成为单页应用、网站或者公共 HTTP API 的好的解决方案。记住，目前为止，就实际性能而言，`Iris` 是至今为止最快的 web 框架。

`Iris` 不会强制你使用任何特定的 `ORM` 或者 `模板引擎` 。支持最强大和快速的模板引擎，你可以快速开发出完美的应用程序。

为什么还要使用其他 Web 框架呢?

Go 是一个伟大的技术栈，可用来为 Web 应用构建可扩展的、基于网络的后端系统。

当你考虑用 Go 构建 web 应用程序和 web API，或者简单构建 HTTP 服务器，你是否考虑过标准库 `net/http` ？然后你不得不解决一些常见的情况，例如静态路由、安全和用户认证，实时通信和许多其他问题，而这些问题 `net/http` 无法解决。

`net/http` 还不够完整，无法用来快速构建设计良好的 web 后端系统。当你意识到这个的时候，你可以思考下下面的话：

- `net/http` 不适合我，但是有许多框架，我选择哪个呢？
- 每个框架都告诉我它是最好的。我不知道该怎么选择。

真相

为了选择哪个框架适合我和我的新项目，我做了一些深入的研究，通过 wrk 和 ab 做了些基准测试。而不幸的是，结果相当让我失望。

我开始想知道 Golang 是否像我从网上阅读的资料中描述的那样，对于网络或者http请求是非常快速的。 在我永久离开 Golang，再次回到 nodejs 和 .net core 之前，我轻声对自己说：

> Makis，不要丢失希望，至少给 Golang 一个机会。在不基于你以前看到的那些”慢”的代码的条件下，创建一个新的东西；学习这个语言的秘密，让 *其他人* 跟随你的脚步。

这些话是在 **2016年3月13号** 我对我自己说的。

在这同一天的晚上，我读到一本关于希腊神话的书。我看到一个古老的女神的名字，我立即得到启发，使用这个女神的名字来给我早已开始设计和编码的新的web框架命名 - `Iris` 。

几周后， `Iris` 在 Github 上已经排到了所有语言的第一了。这对于一个个人项目是非常罕见的现象，而当时，是一个年轻的开发者在背后。从那以后，那些不能承受失败的竞争者和一些奇怪的人开始表现对他人成功的嫉妒，开始在周围随意散播诽谤和谣言。当然，它们甚至从未思考过 `Iris` 就性能和特性而言是最快和最伟大的框架的事实，因为数据不会撒谎，而且他们的所有论证都将会消失。相反地，他们试图通过我的性格来欺负我，他们试图摧毁我来阻止 `Iris` 积极的发展。

然而，我坚信我们不应该用同样的招式去反击，通过宽恕他们及其他们的所作所为，期待一个没有恐惧和种族主义的地方表达自己，向他们证明同情和爱在我们每个人的心中。

如今， `Iris` 比曾经更受欢迎了。事实上，他们的帖子无意间使 `Iris` 更加受欢迎，人们开始非常支持我的梦想，这也是我们始终在这里的原因。

安装

`Iris` 是一个跨平台的软件。

唯一的需求就是 Go 语言，版本为1.13。

```
$ cd $YOUR_PROJECT_PATH
$ export GO111MODULE=on
```

安装

```
go get github.com/kataras/iris/v12@latest
```

或者编辑你项目的 `go.mod` 文件。

```
module your_project_name

go 1.13

require (
    github.com/kataras/iris/v12 v12.0.0
)
```

然后：

```
go build
```

怎么更新

这个 `go-get` 命令来获取最新和最好的 `Iris` 版本。 `Master` 分支通常足够稳定了。

```
$ go get -u github.com/kataras/iris/v12@latest
```

故障排除

如果你在安装过程中出现了网络错误，你确保你设置了一个可用的 `GOPROXY` 环境变量，例如 `GOPROXY=https://goproxy.io` 或者 `GOPROXY=https://goproxy.cn` 。

开始使用

`Iris` 具有用于路由的表达语法，感觉就像在家一样。路由算法是强大的，通过 muxie 来处理请求和比其替代品(httprouter、gin、echo) 更快地匹配路由。

让我们没有任何麻烦地开始吧！

新建一个空文件，命名为 `example.go` ，然后打开它复制粘贴下面的代码。

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.Default()
    app.Use(myMiddleware)

    app.Handle("GET", "/ping", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"message": "pong"})
    })

    // Listens and serves incoming http requests
    // on http://localhost:8080.
    app.Run(iris.Addr(":8080"))
}

func myMiddleware(ctx iris.Context) {
    ctx.Application().Logger().Infof("Runs before %s", ctx.Path())
    ctx.Next()
}
```

打开终端会话，执行下面的指令，然后再浏览器上访问 `http://localhost:8000/ping` ：

```
$ go run example.go
```

展示更多(Show me more)

我们简单地启动和运行程序来获取一个页面。

```
package main

import "github.com/kataras/iris/v12"
```

```
func main() {
    app := iris.New()
    // Load all templates from the "./views" folder
    // where extension is ".html" and parse them
    // using the standard `html/template` package.
    app.RegisterView(iris.HTML("./views", ".html"))

    // Method:    GET
    // Resource:  http://localhost:8080
    app.Get("/", func(ctx iris.Context) {
        // Bind: {{.message}} with "Hello world!"
        ctx.ViewData("message", "Hello world!")
        // Render template file: ./views/hello.html
        ctx.View("hello.html")
    })

    // Method:    GET
    // Resource:  http://localhost:8080/user/42
    //
    // Need to use a custom regexp instead?
    // Easy;
    // Just mark the parameter's type to 'string'
    // which accepts anything and make use of
    // its `regexp` macro function, i.e:
    // app.Get("/user/{id:string regexp(^[0-9]+$)}")
    app.Get("/user/{id:uint64}", func(ctx iris.Context) {
        userID, _ := ctx.Params().GetUint64("id")
        ctx.Writef("User ID: %d", userID)
    })

    // Start the server using a network address.
    app.Run(iris.Addr(":8080"))
}
```

html 文件： `./views/hello.html`

```
<html>
<head>
    <title>Hello Page</title>
</head>
<body>
    <h1>{{ .message }}</h1>
</body>
</html>
```

想要当你的代码发生改变时自动重启你的程序？安装 rizla 工具，执行 `rizla main.go` 来替代 `go run main.go` 。

Host

你可以开启服务监听任何 `net.Listener` 或者 `http.Server` 类型的实例。初始化服务器的方法应该在最后传递给 `Run` 函数。

Go 开发者最常用的方法是通过传递一个形如 `hostname:ip` 形式的网络地址来开启一个服务。 `Iris` 中我们使用 `iris.Addr` ，它是一个 `iris.Runner` 类型。

```
// Listening on tcp with network address 0.0.0.0:8080
app.Run(iris.Addr(":8080"))
```

有时候你在你的应用程序的其他地方创建一个标准库 `het/http` 服务器，并且你想使用它作为你的 Iris web 程序提供服务。

```
// Same as before but using a custom http.Server which may being used somewhere
 else too
app.Run(iris.Server(&http.Server{Addr:":8080"}))
```

最高级的用法是创建一个自定义的或者标准的 `net/Listener` ，然后传递给 `app.Run` 。

```
// Using a custom net.Listener
l, err := net.Listen("tcp4", ":8080")
if err != nil {
    panic(err)
}
app.Run(iris.Listener(l))
```

一个更加完整的示例，使用的是仅Unix套接字文件特性

```
package main

import (
    "os"
    "net"

    "github.com/kataras/iris/v12"
)

func main() {
    app := iris.New()
```

```
    // UNIX socket
    if errOs := os.Remove(socketFile); errOs != nil && !os.IsNotExist(errOs) {
        app.Logger().Fatal(errOs)
    }

    l, err := net.Listen("unix", socketFile)

    if err != nil {
        app.Logger().Fatal(err)
    }

    if err = os.Chmod(socketFile, mode); err != nil {
        app.Logger().Fatal(err)
    }

    app.Run(iris.Listener(l))
}
```

UNIX 和 BSD 主机可以使用重用端口的功能。

```
package main

import (
    // Package tcplisten provides customizable TCP net.Listener with various
    // performance-related options:
    //
    //   - SO_REUSEPORT. This option allows linear scaling server performance
    //     on multi-CPU servers.
    //     See https://www.nginx.com/blog/socket-sharding-nginx-release-1-9-1/ for details.
    //
    //   - TCP_DEFER_ACCEPT. This option expects the server reads from the accepted
    //     connection before writing to them.
    //
    //   - TCP_FASTOPEN. See https://lwn.net/Articles/508865/ for details.
    "github.com/valyala/tcplisten"

    "github.com/kataras/iris/v12"
)

// go get github.com/valyala/tcplisten
// go run main.go

func main() {
```

```
    app := iris.New()

    app.Get("/", func(ctx iris.Context) {
        ctx.HTML("<h1>Hello World!</h1>")
    })

    listenerCfg := tcplisten.Config{
        ReusePort:   true,
        DeferAccept: true,
        FastOpen:    true,
    }

    l, err := listenerCfg.NewListener("tcp", ":8080")
    if err != nil {
        app.Logger().Fatal(err)
    }

    app.Run(iris.Listener(l))
}
```

HTTP/2和安全

如果你有签名文件密钥，你可以使用 `iris.TLS` 基于这些验证密钥开启 `https` 服务。

```
// TLS using files
app.Run(iris.TLS("127.0.0.1:443", "mycert.cert", "mykey.key"))
```

当你的应用准备部署**生产**时，你可以使用 `iris.AutoTLS` 方法，它通过 https://letsencrypt.org 免费提供的证书来开启一个安全的服务。

```
// Automatic TLS
app.Run(iris.AutoTLS(":443", "example.com", "admin@example.com"))
```

任意iris.Runner

有时你想要监听一些特定的东西，并且这些东西不是 `net.Listener` 类型的。你能够通过 `iris.Raw` 方法做到，但是你得对此方法负责。

```
// Using any func() error,
// the responsibility of starting up a listener is up to you with this way,
```

```
// for the sake of simplicity we will use the
// ListenAndServe function of the `net/http` package.
app.Run(iris.Raw(&http.Server{Addr:":8080"}).ListenAndServe)
```

host配置

形如上面所示的监听方式都可以在最后接受一个 `func(*iris.Supervisor)` 的可变参数。通过函数的传递用来为特定 host 添加配置器。

例如，我们想要当服务器关闭的时候触发的回调函数：

```
app.Run(iris.Addr(":8080", func(h *iris.Supervisor) {
    h.RegisterOnShutdown(func() {
        println("server terminated")
    })
}))
```

你甚至可以在再 `app.Run` 之前配置，但是不同的是，这个 host 配置器将会在所有的主机上执行(我们将在稍后看到 `app.NewHost`)

```
app := iris.New()
app.ConfigureHost(func(h *iris.Supervisor) {
    h.RegisterOnShutdown(func() {
        println("server terminated")
    })
})
app.Run(iris.Addr(":8080"))
```

当 `Run` 方法运行之后，通过 `Application#Hosts` 字段提供的所有 hosts 你的应用服务都可以访问。

但是最常用的场景是你可能需要在运行 `app.Run` 之前访问 hosts，这里有2中方法来获得访问 hosts 的监管，阅读下面。

我们已经看到通过 `app.Run` 的第二个参数或者 `app.ConfigureHost` 方法来配置所有的应用程序 hosts 。还有一种更加适合简单场景的方法，那就是使用 `app.NewHost` 来创建一个新的 host，然后使用它的 `Serve` 或者 `Listen` 函数， 通过 `iris#Raw` 来启动服务。

记住这个方法需要额外导入 `net/http` 包。

示例代码：

```
h := app.NewHost(&http.Server{Addr:":8080"})
h.RegisterOnShutdown(func() {
    println("server terminated")
})

app.Run(iris.Raw(h.ListenAndServe))
```

多个主机

你可以使用多个 hosts 来启动你的 iris 程序， `iris.Router` 兼容 `net/http/Handler` 函数，因此我们可以理解为，它可以被适用于任何 `net/http` 服务器，然而，通过使用 `app.NewHost` 是一个更加简单的方法，它也会复制所有的 host 配置器，并在 `app.Shutdown` 时关闭所有依附在特定web 服务的主机 host 。
app := iris.New()
app.Get("/", indexHandler)

```
// run in different goroutine in order to not block the main "goroutine".
go app.Run(iris.Addr(":8080"))
// start a second server which is listening on tcp 0.0.0.0:9090,
// without "go" keyword because we want to block at the last server-run.
app.NewHost(&http.Server{Addr:":9090"}).ListenAndServe()
```

优雅地关闭

让我们继续学习怎么接受 `CONTROL + C` / `COMMAND + C` 或者 unix `kill` 命令，优雅地关闭服务器。(默认是启用的)

为了手动地管理app被中断时需要做的事情，我们需要通过使用 `WithoutInterruptHandler` 选项禁用默认的行为，然后注册一个新的中断处理器(在所有可能的hosts上)。

示例代码：

```
package main

import (
    "context"
    "time"

    "github.com/kataras/iris/v12"
)
```

```
func main() {
    app := iris.New()

    iris.RegisterOnInterrupt(func() {
        timeout := 5 * time.Second
        ctx, cancel := context.WithTimeout(context.Background(), timeout)
        defer cancel()
        // close all hosts
        app.Shutdown(ctx)
    })

    app.Get("/", func(ctx iris.Context) {
        ctx.HTML(" <h1>hi, I just exist in order to see if the server is closed
</h1>")
    })

    app.Run(iris.Addr(":8080"), iris.WithoutInterruptHandler)
```

带有TLS的自动公共域名

在一个公共的、远程地址而不是localhost的“真实的环境”下测试会不会很棒？

有很多提供此类功能的第三方库，但是以我的观点，ngrok 是它们中最棒的一个。像 Iris 一样，它受欢迎并且经过了多年的测试。

Iris 提供了 ngrok 的集成。这个功能简单但是强大。当你想要向你的同事或者项目领导在一个远程会议上快速展示你的开发进度，它将会很有帮助。

请按照以下步骤临时将您的本地Iris Web服务器转换为公共服务器。

1. 下载ngrok，然后把它加入到你的 `$PATH` 环境变量中。
2. 简单传递 `WithTunneling` 选项到 `app.Run` 中
3. 启动

```go
package main

import "github.com/kataras/iris"

func main() {
    app := iris.New()

    app.Get("/", func(ctx iris.Context) {
        ctx.HTML("<h1>Hello World!</h1>")
    })

    app.Run(iris.Addr(":8080"), iris.WithTunneling)
}
```

```
PS C:\mygopath\src\github.com\kataras\iris\_examples\http-listening\listen-addr-public> go run main.go

Now listening on: http://localhost:8080
Application started. Press CTRL+C to shut d
• Public Address: https://376146d4.ngrok.io
```

Hello World!

- `ctx.Application().ConfigurationReadOnly().GetVHost()` 返回公共域名。很少有用，但是可以为您服务。更多的时候你使用相对 URL 路径来替代绝对路径。
- ngrok是否已在运行都无关紧要，Iris框架足够聪明，可以使用ngrok的Web API创建隧道。

完整的 `Tunneling` 配置：

```
app.Run(iris.Addr(":8080"), iris.WithConfiguration(
    iris.Configuration{
        Tunneling: iris.TunnelingConfiguration{
            AuthToken:    "my-ngrok-auth-client-token",
            Bin:          "/bin/path/for/ngrok",
            Region:       "eu",
            WebInterface: "127.0.0.1:4040",
            Tunnels: []iris.Tunnel{
                {
                    Name: "MyApp",
                    Addr: ":8080",
                },
            },
        },
}))
```

配置

在前面的章节中我们学习了 `app.Run` 方法传入的第一个参数，接下来我们看一下第二个参数。

我们从基础开始。 `iris.New` 函数返回一个 `iris.Application` 实例。这个实例可以通过它的 `Configure(...iris.Configurator)` 和 `Run` 方法进行配置。

`app.Run` 方法的第二个参数是可选的、可变长的，接受一个或者多个 `iris.Configurator` 。一个 `iris.Configurator` 是 `func(app *iris.Application)` 类型的函数。自定义的 `iris.Configurator` 能够修改你的 `*iris.Application` 。

每个核心的配置字段都有一个内建的 `iris.Configurator` 。例如， `iris.WithoutStartupLog` ， `iris.WithCharset("UTF-8")` ， `iris.WithOptimizations` ， `iris.WithConfiguration(iris.Congiguration{...})` 函数。

每个模块，例如视图引擎，websockets，sessions和每个中间件都有它们各自配置器和选项，它们大多数都与核心的配置分离。

使用配置

唯一的配置结构体是 `iris.Configuration` 。让我们从通过 `iris.WithConfiguration` 函数来创建 `iris.Configurator` 开始。

所有的 `iris.Configuration` 字段的默认值都是最常用的。 **Iris** 在 `app.Run` 运行之前不需要任何的配置。但是你想要在运行服务之前使用自定义的 `iris.Configurator` ，你可以把你的配置器传递给 `app.Configure` 方法。

```
config := iris.WithConfiguration(iris.Configuration {
  DisableStartupLog: true,
  Optimizations: true,
  Charset: "UTF-8",
})

app.Run(iris.Addr(":8080"), config)
```

从YAML加载

使用 `iris.YAML("path")`

File：iris.yml

```
FireMethodNotAllowed: true
DisableBodyConsumptionOnUnmarshal: true
TimeFormat: Mon, 01 Jan 2006 15:04:05 GMT
Charset: UTF-8
```

File：main.go

```
config := iris.WithConfiguration(iris.YAML("./iris.yml"))
app.Run(iris.Addr(":8080"), config)
```

从TOML加载

使用 `iris.TOML("path")`

File：iris.tml

```
FireMethodNotAllowed = true
DisableBodyConsumptionOnUnmarshal = false
TimeFormat = "Mon, 01 Jan 2006 15:04:05 GMT"
Charset = "UTF-8"

[Other]
    ServerName = "my fancy iris server"
    ServerOwner = "admin@example.com"
```

File：main.go

```
config := iris.WithConfiguration(iris.TOML("./iris.tml"))
app.Run(iris.Addr(":8080"), config)
```

使用函数式方式

我们已经提到，你可以传递任何数量的 `iris.Configurator` 到 `app.Run` 的第二个参数。I `Iris` 为每个 `iris.Configuration` 的字段提供了一个选项。

```
app.Run(iris.Addr(":8080"), iris.WithoutInterruptHandler,
    iris.WithoutServerError(iris.ErrServerClosed),
    iris.WithoutBodyConsumptionOnUnmarshal,
    iris.WithoutAutoFireStatusCode,
```

```
    iris.WithOptimizations,
    iris.WithTimeFormat("Mon, 01 Jan 2006 15:04:05 GMT"),
)
```

当你想要改变一些 `iris.Configuration` 的字段的时候，这是一个很好的做法。通过前缀 : `With` 或者 `Without` ，代码编辑器能够帮助你浏览所有的配置选项，甚至你都不需要翻阅文档。

自定义值

`iris.Configuration` 包含一个名为 `Other map[string]interface` 的字段，它可以接受任何自定义的 `key:value` 选项，因此你可以依据需求使用这个字段来传递程序需要的指定的值。

```
app.Run(iris.Addr(":8080"),
    iris.WithOtherValue("ServerName", "my amazing iris server"),
    iris.WithOtherValue("ServerOwner", "admin@example.com"),
)
```

你可以通过 `app.ConfigurationReadOnly` 来访问这些字段。

```
serverName := app.ConfigurationReadOnly().Other["MyServerName"]
serverOwner := app.ConfigurationReadOnly().Other["ServerOwner"]
```

从 Context 中访问配置

在一个处理器中，通过下面的方式访问这些字段。

```
ctx.Application().ConfigurationReadOnly()
```

路由

处理器类

处理器，物如其名，就是处理请求的东西。

一个处理器响应一个 HTTP 请求。它写入响应头和数据到 `Context.ResponseWriter()` ，然后再返回。返回信号表明请求已经完成；在处理完成调用当时或者之后使用 `Context` 是无效的。

由于 HTTP 客户端软件，HTTP 协议版本，和任何客户端和 iris 服务器中间媒介等因素，在向 `context.ResponseWriter()` 写入后可能无法从 `context.Request().Body` 中读取数据。注意应该先从 `context.Request().Body` 中读取数据后，然后再响应它。

除了读取请求体，处理器不应该改变提供的 `context`

如果处理器出现 `panic` ，服务器(处理器的调用者)会假定这个 panic 的影响与存活的请求无关。它会 recover 这个 panic，记录栈追踪日志到服务器错误日志中，并中断连接。

```
type Handler func(iris.Context)
```

一旦处理器被注册，我们可以给返回的 `路由` 实例指定一个名字，以便更加容易地调试和在视图中匹配相对路径。更多信息，查看 `反向查询(Reverse Lookups)` 章节。

行为

`Iris` 默认接受和注册形如 `/api/user` 这样的路径的路由，且尾部不带斜杠。如果客户端尝试访问 `$your_host/api/user/` ， `Iris` 路由会自动永久重定向(301)到 `$your_host/api/user` ，以便由注册的路由进行处理。这是设计 APIs的现代化的方式。

然而如果你想禁用请求的 `路径更正` 的功能的话，你可以在 `app.Run` 传递 `iris.WithoutPathCorrection` 配置选项。例如：

```
// [app := iris.New...]
// [...]

app.Run(iris.Addr(":8080"), iris.WithoutPathCorrection)
```

如果你想 `/api/user` 和 `/api/user/` 在不重定向的情况下(常用场景)拥有相同的处理器，只需要 `iris.WithoutPathCorrectionRedirection` 选项即可：

```
app.Run(iris.Addr(":8080"), iris.WithoutPathCorrectionRedirection)
```

API

支持所有的 HTTP 方法，开发者也可以在相同路劲的不同的方法注册处理器(比如 `/user` 的 GET 和 POST)。

第一个参数是 HTTP 方法，第二个参数是请求的路径，第三个可变参数应该包含一个或者多个 `iris.Handler` ，当客户端请求到特定的资源路径时，这些处理器将会按照注册的顺序依次执行。

示例代码：

```
app := iris.New()

app.Handle("GET", "/contact", func(ctx iris.Context) {
    ctx.HTML("<h1> Hello from /contact </h1>")
})
```

为了让后端开发者做事更加容易， `Iris` 为所有的 HTTP 方法提供了"帮手"。第一个参数是路由的请求路径，第二个可变参数是一个或者多个 `iris.Handler` ，也会按照注册的顺序依次执行。

示例代码

```
app := iris.New()

// Method: "GET"
app.Get("/", handler)

// Method: "POST"
app.Post("/", handler)

// Method: "PUT"
app.Put("/", handler)

// Method: "DELETE"
app.Delete("/", handler)

// Method: "OPTIONS"
app.Options("/", handler)

// Method: "TRACE"
```

```
app.Trace("/", handler)

// Method: "CONNECT"
app.Connect("/", handler)

// Method: "HEAD"
app.Head("/", handler)

// Method: "PATCH"
app.Patch("/", handler)

// register the route for all HTTP Methods
app.Any("/", handler)

func handler(ctx iris.Context){
    ctx.Writef("Hello from method: %s and path: %s\n", ctx.Method(), ctx.Path())
}
```

离线路由

在 `Iris` 中有一个特殊的方法你可以使用。它被成为 `None` ，你可以使用它向外部隐藏一条路由，但仍然可以从其他路由处理中通过 `Context.Exec` 方法调用它。每个 API 处理方法返回 Route 值。一个 Route 的 `IsOnline` 方法报告那个路由的当前状态。你可以通过它的 `Route.Method` 字段的值来改变路由 `离线` 状态为 `在线` 状态，反之亦然。当然每次状态的改变需要调用 `app.RefreshRouter()` 方法，这个使用是安全的。看看下面一个完整的例子：

```
// file: main.go
package main

import (
    "github.com/kataras/iris/v12"
)

func main() {
    app := iris.New()

    none := app.None("/invisible/{username}", func(ctx iris.Context) {
        ctx.Writef("Hello %s with method: %s", ctx.Params().Get("username"), ct
x.Method())

        if from := ctx.Values().GetString("from"); from != "" {
            ctx.Writef("\nI see that you're coming from %s", from)
        }
```

```
    })

    app.Get("/change", func(ctx iris.Context) {

        if none.IsOnline() {
            none.Method = iris.MethodNone
        } else {
            none.Method = iris.MethodGet
        }

        // refresh re-builds the router at serve-time in order to
        // be notified for its new routes.
        app.RefreshRouter()
    })

    app.Get("/execute", func(ctx iris.Context) {
        if !none.IsOnline() {
            ctx.Values().Set("from", "/execute with offline access")
            ctx.Exec("NONE", "/invisible/iris")
            return
        }

        // same as navigating to "http://localhost:8080/invisible/iris"
        // when /change has being invoked and route state changed
        // from "offline" to "online"
        ctx.Values().Set("from", "/execute")
        // values and session can be
        // shared when calling Exec from a "foreign" context.
        //     ctx.Exec("NONE", "/invisible/iris")
        // or after "/change":
        ctx.Exec("GET", "/invisible/iris")
    })

    app.Run(iris.Addr(":8080"))
}
```

怎么运行

1. 运行 `go run main.go`
2. 打开浏览器访问 `http://localhost:8080/invisible/iris` ，你将会看到 `404 not found` 的错误。
3. 然而 `http://localhost:8080/execute` 将会执行这个路由。
4. 现在，如果你导航至 `http://localhost:8080/change` ，然后刷新 `/invisible/iris` 选项卡，你将会看到它。

路由组

一些列路由可以通过路径的前缀(可选的)进行分组，共享相同的中间件处理器和模板布局。一个组也可以有一个内嵌的组。

`.Party` 用来路由分组，开发者可以声明不限数量的组。

示例代码：

```
app := iris.New()

users := app.Party("/users", myAuthMiddlewareHandler)

// http://localhost:8080/users/42/profile
users.Get("/{id:uint64}/profile", userProfileHandler)
// http://localhost:8080/users/messages/1
users.Get("/messages/{id:uint64}", userMessageHandler)
```

你可以使用 `PartyFunc` 方法编写相同的内容，它接受子路由器或者Party。

```
app := iris.New()

app.PartyFunc("/users", func(users iris.Party) {
    users.Use(myAuthMiddlewareHandler)

    // http://localhost:8080/users/42/profile
    users.Get("/{id:uint64}/profile", userProfileHandler)
    // http://localhost:8080/users/messages/1
    users.Get("/messages/{id:uint64}", userMessageHandler)
})
```

路径参数

与你见到的其他路由器不同， Iris 的路由器可以处理各种路由路径而不会发生冲突。

只匹配 `GET "/"`

```
app.Get("/", indexHandler)
```

下面所示能匹配所有以 `/assets/**/*` 前缀的 GET 请求，它是一个通配符， `ctx.Params().Get("asset")` 获取 `/assets/` 后面的所有路径。

```
app.Get("/assets/{asset:path}", assetsWildcardHandler)
```

下面所示的能匹配所有以 `/profile/` 前缀的 GET 请求，但是获取的只是单个路径的部分。

```
app.Get("/profile/{username:string}", userHandler)
```

下面所示的只能匹配 `/profile/me` 的 GET 请求，它不与 `/profile/{username:string}` 或者 `/{root:path}` 冲突。

```
app.Get("/profile/me", userHandler)
```

下面所示的能匹配所有以 `/users/` 前缀的 GET 请求，并且后面的是一个数字，且数字要大于等于1。

```
app.Get("/user/{userid:int min(1)}", getUserHandler)
```

下面所示的能匹配所有以 `/users/` 前缀的 DELETE 请求，并且后面的是一个数字，且数字要大于等于1。

```
app.DELETE("/user/{userid:int min(1)}", getUserHandler)
```

下面所示的能匹配所有除了被其他路由器处理的 GET 请求。例如在这种情况下，上面的路由： `/` ， `/assets/{asset:path}` ， `/profile/{username}` ， `/profile/me` ， `/user/{userid:int ...}` ，它将不会与余下的路由冲突。

```
app.Get("{root:path}", rootWildcardHandler)
```

匹配所有的请求：

1. `/u/adcd` 映射为 `:alphabetical` (如果 `:alphabetical` 注册， 否则 `:string`)
2. `/u/42` 映射为 `:uint` (如果 `:uint` 注册， 否则 `:int`)
3. `/u/-1` 映射为 `:int` (如果 `:int` 注册， 否则 `:string`)
4. `/u/adcd123` 映射为 `:string`

```
app.Get("/u/{username:string}", func(ctx iris.Context) {
    ctx.Writef("username (string): %s", ctx.Params().Get("username"))
})
```

```
app.Get("/u/{id:int}", func(ctx iris.Context) {
    ctx.Writef("id (int): %d", ctx.Params().GetIntDefault("id", 0))
})
app.Get("/u/{uid:uint}", func(ctx iris.Context) {
    ctx.Writef("uid (uint): %d", ctx.Params().GetUintDefault("uid", 0))
})
app.Get("/u/{firstname:alphabetical}", func(ctx iris.Context) {
    ctx.Writef("firstname (alphabetical): %s", ctx.Params().Get("firstname"
))
})
```

分别匹配 `/abctenchars.xml` 和 `/abcdtenchars` 的所有 GET 请求

```
app.Get("/{alias:string regexp(^[a-z0-9]{1,10}\\.xml$)}", PanoXML)
app.Get("/{alias:string regexp(^[a-z0-9]{1,10}$)}", Tour)
```

你可能知道 `{id:uint64}` 、 `:path` 、 `min(1)` 是什么。它们是在注册时可以键入的动态参数和函数。了解更多请阅读 `路径参数类型(Path Parameter Types)`

路径参数类型

`Iris` 拥有你见过的的最简单和强大路由处理。

`Iris` 自己拥有用于路由路径语法解析和判定的解释器(就像一门编程语言)。

它是快速的。它计算它的需求，如果没有特殊的正则需要，它仅仅会使用低级的路径语法来注册路由，除此之外，它预编译正则，然后加入到必需的中间件中。这就意味着与其他的路由器或者 web 框架相比，你的性能成本为零。

参数

一个路径参数的名字应该仅仅包含 `字母` 。数字和类似 `"_"` 这样的符号是 `不允许的` 。

不要迷惑于 `ctx.Params()` 和 `ctx.Values()`

- 路径的参数值可以通过 `ctx.Params()` 取出。
- `ctx` 中用于处理器与中间件之间通信的本地存储可以存储在 `ctx.Values()` 中。

下表是内建可用的参数类型：

参数类型	golang类型	取值范围	取值方式
`:string`	`string`	任何值(单个字段路径)	`Params().Get`
`:int`	`int`	-9223372036854775808 - 9223372036854775807 (x64) -2147483648 - 2147483647 (x32)	`Params().GetIn`
`:int8`	`int8`	-128 - 127	`Params().GetIn`
`:int16`	`int16`	-32768 - 32767	`Params().GetIn`
`:int32`	`int32`	-2147483648 - 2147483647	`Params().GetIn`
`:int64`	`int64`	-9223372036854775808 - 9223372036854775807	`Params().GetIn`
`:uint8`	`uint8`	0 - 255	`Params().GetUi`

参数类型	golang类型	取值范围	取值方式
:uint16	uint16	0 - 65535	Params().GetUi
:uint32	uint32	0 - 4294967295	Params().GetUi
:uint64	uint64	0 - 18446744073709551615	Params().GetUi
:bool	bool	“1”，”t”，”T”，” TRUE”，”true”，” True”，”0”，”f”， “F”，“FALSE”,，”false”，” False”	Params().GetBo
:alphabetical	string	小写或大写字母	Params().Get
:file	string	大小写字母，数字，下划线(_)，横线(-)，点(.)，以及没有空格或者其他对文件名无效的特殊字符	Params().Get
:path	string	任何可以被斜线(/)分隔的路径段，但是应该为路由的最后一部分	Params().Get

示例：

```
app.Get("/users/{id:uint64}", func(ctx iris.Context){
    id := ctx.Params().GetUint64Default("id", 0)
    // [...]
})
```

内建函数

内建函数	参数类型
regexp(expr string)	:string
prefix(prefix string)	:string

内建函数	参数类型
`suffix(suffix string)`	`:string`
`contains(s string)`	`:string`
`min(最小值)，接收：int，int8，int16，int32，int64，uint8`，`uint16，uint32，uint64，float32，float64)`	`:string(字符长度)，:int，:int16，:int32，:int64`，`:uint，:uint16，:uint32，:uint64`
`max(最大值)，接收：int，int8，int16，int32，int64，uint8`，`uint16，uint32，uint64，float32，float64)`	`:string(字符长度)，:int，:int16，:int32，:int64`，`:uint，:uint16，:uint32，:uint64`
`range(最小值，最大值)，接收：int，int8，int16，int32，int64，uint8`，`uint16，uint32，uint64，float32，float64)`	`:int，:int16，:int32，:int64`，`:uint，:uint16，:uint32，:uint64`

示例：

```
app.Get("/profile/{name:alphabetical max(255)}", func(ctx iris.Context){
    name := ctx.Params().Get("name")
    // len(name) <=255 otherwise this route will fire 404 Not Found
    // and this handler will not be executed at all.
})
```

自己做

`RegisterFunc` 可以接受任何返回 `func(paramValue string) bool` 的函数。如果验证失败将会触发 `404` 或者任意 `else` 关键字拥有的状态码。

```
latLonExpr := "^-?[0-9]{1,3}(?:\\.[0-9]{1,10})?$"
latLonRegex, _ := regexp.Compile(latLonExpr)

// Register your custom argument-less macro function to the :string param type.
// MatchString is a type of func(string) bool, so we use it as it is.
app.Macros().Get("string").RegisterFunc("coordinate", latLonRegex.MatchString)

app.Get("/coordinates/{lat:string coordinate()}/{lon:string coordinate()}",
func(ctx iris.Context) {
    ctx.Writef("Lat: %s | Lon: %s", ctx.Params().Get("lat"), ctx.Params().Get("l
on"))
})
```

注册接受两个 `int` 参数的自定义的宏函数。

```
app.Macros().Get("string").RegisterFunc("range",
func(minLength, maxLength int) func(string) bool {
    return func(paramValue string) bool {
        return len(paramValue) >= minLength && len(paramValue) <= maxLength
    }
})

app.Get("/limitchar/{name:string range(1,200) else 400}", func(ctx iris.Context)
{
    name := ctx.Params().Get("name")
    ctx.Writef(`Hello %s | the name should be between 1 and 200 characters lengt
h
    otherwise this handler will not be executed`, name)
})
```

注册接受一个 `[]string` 参数的自定义的宏函数。

```
app.Macros().Get("string").RegisterFunc("has",
func(validNames []string) func(string) bool {
    return func(paramValue string) bool {
        for _, validName := range validNames {
            if validName == paramValue {
                return true
            }
        }

        return false
    }
})
```

```
app.Get("/static_validation/{name:string has([kataras,maropoulos])}",
func(ctx iris.Context) {
    name := ctx.Params().Get("name")
    ctx.Writef(`Hello %s | the name should be "kataras" or "maropoulos"
    otherwise this handler will not be executed`, name)
})
```

示例代码：

```
func main() {
    app := iris.Default()

    // This handler will match /user/john but will not match neither /user/ or /
user.
    app.Get("/user/{name}", func(ctx iris.Context) {
        name := ctx.Params().Get("name")
        ctx.Writef("Hello %s", name)
    })

    // This handler will match /users/42
    // but will not match /users/-1 because uint should be bigger than zero
    // neither /users or /users/.
    app.Get("/users/{id:uint64}", func(ctx iris.Context) {
        id := ctx.Params().GetUint64Default("id", 0)
        ctx.Writef("User with ID: %d", id)
    })

    // However, this one will match /user/john/send and also /user/john/everythi
ng/else/here
    // but will not match /user/john neither /user/john/.
    app.Post("/user/{name:string}/{action:path}", func(ctx iris.Context) {
        name := ctx.Params().Get("name")
        action := ctx.Params().Get("action")
        message := name + " is " + action
        ctx.WriteString(message)
    })

    app.Run(iris.Addr(":8080"))
}
```

当没有指定参数类型时，默认为 `string` ，因此 `{name:string}` 和 `{name}` 是完全相同的。

反向查询

如“路由”一章所述， `Iris` 提供了一些处理器注册方法，每个方法都返回一个 `Route` 实例。

路由命名

路由命名非常简单，我们只需要调用返回的 `*Route` 的 `Name` 字段来定义名字。

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()
    // define a function
    h := func(ctx iris.Context) {
        ctx.HTML("<b>Hi</b1>")
    }

    // handler registration and naming
    home := app.Get("/", h)
    home.Name = "home"
    // or
    app.Get("/about", h).Name = "about"
    app.Get("/page/{id}", h).Name = "page"g

    app.Run(iris.Addr(":8080"))
}
```

从路由名字生成URL

当我们为特定的路径注册处理器时，我们可以根据传递给 Iris 的结构化数据创建 URLs。如上面的例子所示，我们命名了三个路由，其中之一甚至带有参数。如果我们使用默认的 `html/template` 视图引擎，我们可以使用一个简单的操作来反转路由(生成示例的 URLs)：

```
Home: {{ urlpath "home" }}
About: {{ urlpath "about" }}
Page 17: {{ urlpath "page" "17" }}
```

上面的代理可以生成下面的输出：

```
Home: http://localhost:8080/
About: http://localhost:8080/about
Page 17: http://localhost:8080/page/17
```

在代码中使用路由名字

我们可以使用以下方法/函数来处理命名路由（及其参数）：

- `GetRoutes` 函数获取所有注册的路由
- `GetRoute(routeName string)` 方法通过名字获得路由
- `URL(routeName string, paramValues ...interface{})` 方法通过提供的值来生成URL 字符串
- `Path(routeName string, paramValues ...interface{})` 方法通过提供的值生成URL的路径部分(没有主机地址和协议)。

中间件

当我们谈论 Iris 中的中间件时，我们谈论的是一个 HTTP 请求的生命周期中主处理器代码运行前/后运行的代码。例如，日志中间件可能记录一个传入请求的详情到日志中，然后调用处理器代码，然后再编写有关响应的详细信息到日志中。关于中间件的一件很酷的事情是，这些单元非常灵活且可重复使用。

中间件仅是一个 `Handler` 格式的函数 `func(ctx iris.Context)` ，当前一个中间件调用 `ctx.Next()` 方法时，此中间件被执行，这可以用作身份验证，即如果请求验证通过，就调用 `ctx.Next()` 来执行该请求剩下链上的处理器，否则触发一个错误响应。

编写一个中间件

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()
    // or app.Use(before) and app.Done(after).
    app.Get("/", before, mainHandler, after)
    app.Run(iris.Addr(":8080"))
}

func before(ctx iris.Context) {
    shareInformation := "this is a sharable information between handlers"

    requestPath := ctx.Path()
    println("Before the mainHandler: " + requestPath)

    ctx.Values().Set("info", shareInformation)
    ctx.Next() // execute the next handler, in this case the main one.
}

func after(ctx iris.Context) {
    println("After the mainHandler")
}

func mainHandler(ctx iris.Context) {
    println("Inside mainHandler")

    // take the info from the "before" handler.
    info := ctx.Values().GetString("info")
```

```
    // write something to the client as a response.
    ctx.HTML("<h1>Response</h1>")
    ctx.HTML("<br/> Info: " + info)

    ctx.Next() // execute the "after".
}
```

```
$ go run main.go # and navigate to the http://localhost:8080
Now listening on: http://localhost:8080
Application started. Press CTRL+C to shut down.
Before the mainHandler: /
Inside mainHandler
After the mainHandler
```

全局范围

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()

    // register our routes.
    app.Get("/", indexHandler)
    app.Get("/contact", contactHandler)

    // Order of those calls does not matter,
    // `UseGlobal` and `DoneGlobal` are applied to existing routes
    // and future routes also.
    //
    // Remember: the `Use` and `Done` are applied to the current party's and its
children,
    // so if we used the `app.Use/Done before the routes registration
    // it would work like UseGlobal/DoneGlobal in this case,
    // because the `app` is the root "Party".
    app.UseGlobal(before)
    app.DoneGlobal(after)

    app.Run(iris.Addr(":8080"))
}
```

```
func before(ctx iris.Context) {
    // [...]
}

func after(ctx iris.Context) {
    // [...]
}

func indexHandler(ctx iris.Context) {
    // write something to the client as a response.
    ctx.HTML("<h1>Index</h1>")

    ctx.Next() // execute the "after" handler registered via `Done`.
}

func contactHandler(ctx iris.Context) {
    // write something to the client as a response.
    ctx.HTML("<h1>Contact</h1>")

    ctx.Next() // execute the "after" handler registered via `Done`.
}
```

你也可以使用 `ExecutionRules` 强制处理器在没有 `ctx.Next()` 的情况下完成执行。你可以这样做：

```
app.SetExecutionRules(iris.ExecutionRules{
    // Begin: ...
    // Main:  ...
    Done: iris.ExecutionOptions{Force: true},
})
```

转化 `http.Handler/HandlerFunc`

使用它们是没有限制的 - 你很自由地使用任何与 `net/http` 包兼容的第三方的中间件。

Iris与其他框架不同，它是 100% 与标准库兼容，这就是为什么多数大型公司信任 Iris，使用 Go 来完成它们的工作流程，如著名的 US Television Network；它是最新的，并且始终与标准库 `net/http` 包保持一致，标准库 `net/http` 在每次 Go 版本更新的时候都会进行优化。

任意用 `net/http` 编写的第三方中间件通过 `iris.FromStd(AThirdPartyMiddleware)` 与 Iris 兼容。记住，`ctx.ResponseWriter()` 和 `ctx.Request()` 返回与 `net/http` 的 `http.Handler` 相同的输入参数。

这里是一系列为 Iris 特定功能创建的 handlers：

内建的

- basic authentication
- Google reCAPTCHA
- localization and internationalization
- request logger
- profiling (pprof)
- recovery

社区创建的

请查看文档。

处理HTTP错误

你可以定义自己的处理器来处理特定 http 错误。

> 错误码是 大于等于400的Http 状态码，例如 `404 not found` ， `500 internal server` 。

示例代码：

```
package main

import "github.com/kataras/iris/v12"

func main(){
    app := iris.New()
    app.OnErrorCode(iris.StatusNotFound, notFound)
    app.OnErrorCode(iris.StatusInternalServerError, internalServerError)
    // to register a handler for all "error"
    // status codes(kataras/iris/context.StatusCodeNotSuccessful)
    // defaults to < 200 || >= 400:
    // app.OnAnyErrorCode(handler)
    app.Get("/", index)
    app.Run(iris.Addr(":8080"))
}

func notFound(ctx iris.Context) {
    // when 404 then render the template
    // $views_dir/errors/404.html
    ctx.View("errors/404.html")
}

func internalServerError(ctx iris.Context) {
    ctx.WriteString("Oups something went wrong, try again")
}

func index(ctx iris.Context) {
    ctx.View("index.html")
}
```

更多内容查看 视图(View) 章节。

问题类型

Iris 内建支持 HTTP APIs 的错误详情。

`Context.Problem` 编写一个 JSON 或者 XML 问题响应，行为完全类似 `Context.JSON` ，但是默认 `ProblemOptions.JSON` 的缩进是 `" "` ，响应的 `Content-type` 为 `application/problem+json` 。

使用 `options.RenderXML` 和 `XML` 字段来改变他的行为，用 `application/problem+xml` 的文本类型替代。

```
func newProductProblem(productName, detail string) iris.Problem {
    return iris.NewProblem().
        // The type URI, if relative it automatically convert to absolute.
        Type("/product-error").
        // The title, if empty then it gets it from the status code.
        Title("Product validation problem").
        // Any optional details.
        Detail(detail).
        // The status error code, required.
        Status(iris.StatusBadRequest).
        // Any custom key-value pair.
        Key("productName", productName)
        // Optional cause of the problem, chain of Problems.
        // .Cause(other iris.Problem)
}

func fireProblem(ctx iris.Context) {
    // Response like JSON but with indent of "  " and
    // content type of "application/problem+json"
    ctx.Problem(newProductProblem("product name", "problem details"),
        iris.ProblemOptions{
            // Optional JSON renderer settings.
            JSON: iris.JSON{
                Indent: "  ",
            },
            // OR
            // Render as XML:
            // RenderXML: true,
            // XML:       iris.XML{Indent: "  "},
            // Sets the "Retry-After" response header.
            //
            // Can accept:
            // time.Time for HTTP-Date,
            // time.Duration, int64, float64, int for seconds
            // or string for date or duration.
            // Examples:
            // time.Now().Add(5 * time.Minute),
```

```
            // 300 * time.Second,
            // "5m",
            //
            RetryAfter: 300,
            // A function that, if specified, can dynamically set
            // retry-after based on the request.
            // Useful for ProblemOptions reusability.
            // Overrides the RetryAfter field.
            //
            // RetryAfterFunc: func(iris.Context) interface{} { [...] }
    })
}
```

输出 `application/problem+json`

```json
{
  "type": "https://host.domain/product-error",
  "status": 400,
  "title": "Product validation problem",
  "detail": "problem error details",
  "productName": "product name"
}
```

当 `RenderXML` 设置为 `true` 的时候，响应将被渲染为 `xml` 。

输出 `application/problem+xml`

```xml
<Problem>
    <Type>https://host.domain/product-error</Type>
    <Status>400</Status>
    <Title>Product validation problem</Title>
    <Detail>problem error details</Detail>
    <ProductName>product name</ProductName>
</Problem>
```

子域名

`Iris` 具有已知最简单的注册子域名到单个应用程序的方式。当然你也可以在生成环境中使用 `nginx` 或者 `caddy` 。

子域名被分为两类：**静态** 和 **动态/通配**。

- `静态` ：你所知的子域名，例如： `analytics.mydomain.com` 。
- `通配` ：翻译不通(when you don't know the subdomain but you know that it's before a particular subdomain or root domain, i.e :)，即： `user_created.mydomain.com` ， `otheruser.mydomain.com` ，就像 `username.github.io` 这样。

我们使用 `iris.Party` 或者 `iris.Application` 的 `Subdomain` 和 `WildcardSubdomain` 方法注册子域名。

`Subdomain` 方法返回的是一个新的 `Party` 对象，它负责为特定的子域名注册路由。

与常规 `Party` 不同的是，如果子 party 调用它，域名将会被添加到路径的前面，而不是追加到路径后面。因此如果 `app.Sundomain("admin").Subdomain("panel")` ，结果是： `panel.admin` 。

```
Subdomain(subdomain string, middleware ...Handler) Party
```

`WildcardSubdomain` 方法返回一个新的 `Party` ，它负责注册路由到一个动态的，通配的子域名中。一个动态的子域名能处理多个子域名请求。服务器将会接受多个子域名(如果没有找到对应的静态子域名)，它也会搜索和和执行这个 `Party` 的处理器。

```
WildcardSubdomain(middleware ...Handler) Party
```

示例代码：

```
// [app := iris.New...]
admin := app.Subdomain("admin")

// admin.mydomain.com
admin.Get("/", func(ctx iris.Context) {
    ctx.Writef("INDEX FROM admin.mydomain.com")
})

// admin.mydomain.com/hey
```

```
admin.Get("/hey", func(ctx iris.Context) {
    ctx.Writef("HEY FROM admin.mydomain.com/hey")
})

// [other routes here...]

app.Run(iris.Addr("mydomain.com:80"))
```

对于本地开发系统，你要修改你的 `hosts` 文件，例如在windows操作系统中，打开 `C:\Windows\System32\Drivers\etc\hosts` 文件，然后追加：

```
127.0.0.1 mydomain.com
127.0.0.1 admin.mydomain.com
```

为了证明子域名像其它正则 `Party` 一样工作，你也可以用下面这种另类的方法注册一个子域名：

```
adminSubdomain:= app.Party("admin.")
// or
adminAnalayticsSubdomain := app.Party("admin.analytics.")
// or for a dynamic one:
anySubdomain := app.Party("*.")
```

还有一个 `iris.Application` 方法，允许为子域名创建全局重定向规则。

`SubdomainRedirect` 设置(当使用超过1次时添加)一个路由包装器，它可以使一个(子)域名在执行路由处理器之前尽可能快地重定向(永久重定向)到另一个子域名或根域名。

它接收2个参数，它们是 `from` 和 `to/target` 的位置， `from` 也可以是一个通配的子域名(`app.WildcardSubdomain()`)， `to` 不允许是通配的。当 `to` 不是根域名时 `from` 可以是跟域名，反正亦然。

```
SubdomainRedirect(from, to Party) Party
```

使用：

```
www := app.Subdomain("www")
app.SubdomainRedirect(app, www)
```

上面的所有 `htt(s)://mydomain.com/%anypath%` 将会重定向到 `https(s)://www.mydomain.com/%anypath%` 。

当你使用子域名时， `Context` 提供了四个对你很有帮助的主要方法。

```go
// Host returns the host part of the current url.
Host() string
// Subdomain returns the subdomain of this request, if any.
// Note that this is a fast method which does not cover all cases.
Subdomain() (subdomain string)
// IsWWW returns true if the current subdomain (if any) is www.
IsWWW() bool
// FullRqeuestURI returns the full URI,
// including the scheme, the host and the relative requested path/resource.
FullRequestURI() string
```

使用：

```go
func info(ctx iris.Context) {
    method := ctx.Method()
    subdomain := ctx.Subdomain()
    path := ctx.Path()

    ctx.Writef("\nInfo\n\n")
    ctx.Writef("Method: %s\nSubdomain: %s\nPath: %s", method, subdomain, path)
}
```

包装路由器

你可能永远不需要这个，但是以防万一。

有时候你可能需要覆写或者决定一个请求进入时这个路由器是否执行。如果你在以前有许多 `net/http` 和其他 web 框架的经验，这个函数将会使你感到熟悉(它有 `net/http` 中间件的格式，但是不接收下一个处理器，它接收的是一个处理器，来作为是否执行的函数)。

```
// WrapperFunc is used as an expected input parameter signature
// for the WrapRouter. It's a "low-level" signature which is compatible
// with the net/http.
// It's being used to run or no run the router based on a custom logic.
type WrapperFunc func(w http.ResponseWriter, r *http.Request, router http.HandlerFunc)

// WrapRouter adds a wrapper on the top of the main router.
// Usually it's useful for third-party middleware
// when need to wrap the entire application with a middleware like CORS.
//
// Developers can add more than one wrappers,
// those wrappers' execution comes from last to first.
// That means that the second wrapper will wrap the first, and so on.
//
// Before build.
func WrapRouter(wrapperFunc WrapperFunc)
```

路由器基于 `Subdomain` ， `HTTP 方法` 和他的动态路径来寻找它的路由。路由包装器可以重写这种行为，执行自定义的代码。

在这个示例中，你将会看到 `.WrapRouter` 的一个用例。你可以使用 `.WrapRouter` 添加自定义逻辑，来决定路由器什么时候执行或者不执行，来达到控制是否执行注册的路由处理器的目的。这仅仅是为了证明概念，你可以跳过本篇教程。

示例代码：

```
package main

import (
	"net/http"
	"strings"

	"github.com/kataras/iris/v12"
)
```

```
func newApp() *iris.Application {
    app := iris.New()

    app.OnErrorCode(iris.StatusNotFound, func(ctx iris.Context) {
        ctx.HTML("<b>Resource Not found</b>")
    })

    app.Get("/profile/{username}", func(ctx iris.Context) {
        ctx.Writef("Hello %s", ctx.Params().Get("username"))
    })

    app.HandleDir("/", "./public")

    myOtherHandler := func(ctx iris.Context) {
        ctx.Writef("inside a handler which is fired manually by our custom route
r wrapper")
    }

    // wrap the router with a native net/http handler.
    // if url does not contain any "." (i.e: .css, .js...)
    // (depends on the app , you may need to add more file-server exceptions),
    // then the handler will execute the router that is responsible for the
    // registered routes (look "/" and "/profile/{username}")
    // if not then it will serve the files based on the root "/" path.
    app.WrapRouter(func(w http.ResponseWriter, r *http.Request, router http.Hand
lerFunc) {
        path := r.URL.Path

        if strings.HasPrefix(path, "/other") {
                // acquire and release a context in order to use it to execute
                // our custom handler
                // remember: we use net/http.Handler because here
                // we are in the "low-level", before the router itself.
                ctx := app.ContextPool.Acquire(w, r)
                myOtherHandler(ctx)
                app.ContextPool.Release(ctx)
                return
            }

            // else continue serving routes as usual.
            router.ServeHTTP(w, r)
    })

    return app
}
```

```go
func main() {
	app := newApp()

	// http://localhost:8080
	// http://localhost:8080/index.html
	// http://localhost:8080/app.js
	// http://localhost:8080/css/main.css
	// http://localhost:8080/profile/anyusername
	// http://localhost:8080/other/random
	app.Run(iris.Addr(":8080"))

	// Note: In this example we just saw one use case,
	// you may want to .WrapRouter or .Downgrade in order to
	// bypass the Iris' default router, i.e:
	// you can use that method to setup custom proxies too.
}
```

这里不需要多说，它仅是一个接受原生 `ResponseWriter` 、 `Request` 以及路由器下一个处理器的函数包装器，**它是全部路由器的一个中间件**。

重写Context

在这个小节你将会学习如果重写已存在的 `Context` 的方法。

`Context` 是一个接口。然而你可能了解，当使用其它框架的时，即使它是一个接口，你也无法重写它。Iris 使用 `app.ContextPool.Attach` 方法连接你的实现到 `context pool` 中。

1. 首先我们导入 `github.com/kataras/iris/v12/context` ，这里需要它。
2. 其次编写你自己的实现
3. 然后添加 `Do` 和 `Next` 两个包级别函数
4. 使用 `Application` 的 `ContextPool` 将其设置为应用于路由处理程序的Context实现。

示例代码：

请读注释：

```
package main

import (
    "reflect"

    "github.com/kataras/iris/v12"
    // 1.
    "github.com/kataras/iris/v12/context"
)

// 2.
// Create your own custom Context, put any fields you'll need.
type MyContext struct {
    // Embed the `iris.Context` -  嵌入 iris.Context
    // It's totally optional but you will need this if you
    // don't want to override all the context's methods!
        // 这是可选的，但是你不想重写所有的 context的方法，需要这么做。
    iris.Context
}

// Optionally: validate MyContext implements iris.Context on compile-time.
// 可选的，在编译的时候验证是否实现 iris.Context
var _ iris.Context = &MyContext{}

// 3.
func (ctx *MyContext) Do(handlers context.Handlers) {
```

```
    context.Do(ctx, handlers)
}
```

```
// 3.
func (ctx *MyContext) Next() {
    context.Next(ctx)
}

// [Override any context's method you want here...]
// Like the HTML below:

func (ctx *MyContext) HTML(htmlContents string) (int, error) {
    ctx.Application().Logger().Infof("Executing .HTML function from MyContext")

    ctx.ContentType("text/html")
    return ctx.WriteString(htmlContents)
}

func main() {
    app := iris.New()

    // 4.
    app.ContextPool.Attach(func() iris.Context {
        return &MyContext{
            // If you use the embedded Context,
            // call the `context.NewContext` to create one:
            Context: context.NewContext(app),
        }
    })

    // Register a view engine on .html files inside the ./view/** directory.
    app.RegisterView(iris.HTML("./view", ".html"))

    // Register your route, as you normally do
    app.Handle("GET", "/", recordWhichContextForExample,
    func(ctx iris.Context) {
        // use the context's overridden HTML method.
        ctx.HTML("<h1> Hello from my custom context's HTML! </h1>")
    })

    // This will be executed by the
    // MyContext.Context embedded default context
    // when MyContext is not directly define the View function by itself.
    app.Handle("GET", "/hi/{firstname:alphabetical}",recordWhichContextForExampl
e,
```

```
    func(ctx iris.Context) {
        firstname := ctx.Values().GetString("firstname")

        ctx.ViewData("firstname", firstname)
        ctx.Gzip(true)

        ctx.View("hi.html")
    })

    app.Run(iris.Addr(":8080"))
}

// Should always print "($PATH) Handler is executing from 'MyContext'"
func recordWhichContextForExample(ctx iris.Context) {
    ctx.Application().Logger().Infof("(%s) Handler is executing from: '%s'",
        ctx.Path(), reflect.TypeOf(ctx).Elem().Name())

    ctx.Next()
}
```

Context方法

这里是 `iris.Context` 提供的完整的方法列表。

```
type (
    BodyDecoder interface {
        Decode(data []byte) error
    }

    Unmarshaler interface {
        Unmarshal(data []byte, outPtr interface{}) error
    }

    UnmarshalerFunc func(data []byte, outPtr interface{}) error
)

func (u UnmarshalerFunc) Unmarshal(data []byte, v interface{}) error {
    return u(data, v)
}
```

- `BodyDecoder` :
    - `BodyDecoder` 是一个接口，任何结构体都可以实现，以便于实现自定义读取JSON或者XML的 decode 行为。
- 一个简单的例子：

    ```
        type User struct { Username string }
        func (u *User) Decode(data []byte) error {
                return json.Unmarshal(data, u)
        }
    ```

    - `context.ReadJSON/ReadXML(&User{})` 将会调用 `User` 的 `Decode` 来解码请求体
    - 记住：这是完全可选的，默认的ReadJSON解码器是 `encoding/json` ，ReadXML解码器是 `encoding/xml`
- `Unmarshaler`
    - 这是一个接口，实现了可以反序列化任何类型的原始数据。

- 提示：任何值的指针实现了 `BodyDecoder` 将会覆写 `unmarshaler` 。

- `UnmarshalerFunc`
  - `Unmarahsler` 接口的快捷方式
  - 更多详情看 `Unmarshaler` 和 `BodyDecoder`

- `Unmarshal`
  - 解析 `x-encoded` 的数据，并将结构存储到 `v` 指向的指针中。
  - `Unmarshal` 使用与 `Marshal` 使用的相反的编码，必须为map，slice和指针。

- `Context` 是一个客户端在服务器的“中间人对象”。一个新的 `Context` 是从每一个连接的一个 `sync.Pool` 中获取的。 `Context` 是 Iris 的HTTP流上最重要的东西。开发者通过一个 `Context` 发送客户端请求的响应。开发者也从 `Context` 中获取客户端请求的信息。

```
    type Context interface {

        BeginRequest(http.ResponseWriter, *http.Request)

        EndRequest()

        ResetResponseWriter(ResponseWriter)

        Request() *http.Request

        ResetRequest(r *http.Request)

        SetCurrentRouteName(currentRouteName string)

        GetCurrentRoute() RouteReadOnly

        Do(Handlers)

        AddHandler(...Handler)

        SetHandlers(Handlers)

        Handlers() Handlers

        HandlerIndex(n int) (currentIndex int)
```

```
        Proceed(Handler) bool

        HandlerName() string

        HandlerFileLine() (file string, line int)

        RouteName() string

        Next()

        NextOr(handlers ...Handler) bool

        NextOrNotFound() bool

        NextHandler() Handler

        Skip()

        StopExecution()

        IsStopped() bool

        OnConnectionClose(fnGoroutine func()) bool

        OnClose(cb func())

        Params() *RequestParams

        Values() *memstore.Store

        Translate(format string, args ...interface{}) string

        Method() string

        Path() string

        RequestPath(escape bool) string

        Host() string

        Subdomain() (subdomain string)

        IsWWW() bool

        FullRequestURI() string
```

```
    RemoteAddr() string

    GetHeader(name string) string

    IsAjax() bool

    IsMobile() bool

    GetReferrer() Referrer

    Header(name string, value string)

    ContentType(cType string)

    GetContentType() string

    GetContentTypeRequested() string

    GetContentLength() int64

    StatusCode(statusCode int)

    GetStatusCode() int

    Redirect(urlToRedirect string, statusHeader ...int)

    URLParamExists(name string) bool

    URLParamDefault(name string, def string) string

    URLParam(name string) string

    URLParamTrim(name string) string

    URLParamEscape(name string) string

    URLParamInt(name string) (int, error)

    URLParamIntDefault(name string, def int) int

    URLParamInt32Default(name string, def int32) int32

    URLParamInt64(name string) (int64, error)

    URLParamInt64Default(name string, def int64) int64
```

```
	URLParamFloat64(name string) (float64, error)

	URLParamFloat64Default(name string, def float64) float64

	URLParamBool(name string) (bool, error)

	URLParams() map[string]string

	FormValueDefault(name string, def string) string

	FormValue(name string) string

	FormValues() map[string][]string

	PostValueDefault(name string, def string) string

	PostValue(name string) string

	PostValueTrim(name string) string

	PostValueInt(name string) (int, error)

	PostValueIntDefault(name string, def int) int

	PostValueInt64(name string) (int64, error)

	PostValueInt64Default(name string, def int64) int64

	PostValueFloat64(name string) (float64, error)

	PostValueFloat64Default(name string, def float64) float64

	PostValueBool(name string) (bool, error)

	PostValues(name string) []string

	FormFile(key string) (multipart.File, *multipart.FileHeader, error)

	UploadFormFiles(destDirectory string, before ...func(Context, *multipart.FileHeader)) (n int64, err error)

	NotFound()

	SetMaxRequestBodySize(limitOverBytes int64)
```

```
        GetBody() ([]byte, error)

        UnmarshalBody(outPtr interface{}, unmarshaler Unmarshaler) error

        ReadJSON(jsonObjectPtr interface{}) error

        ReadXML(xmlObjectPtr interface{}) error

        ReadForm(formObject interface{}) error

        ReadQuery(ptr interface{}) error

        Write(body []byte) (int, error)

        Writef(format string, args ...interface{}) (int, error)

        WriteString(body string) (int, error)

        SetLastModified(modtime time.Time)

        CheckIfModifiedSince(modtime time.Time) (bool, error)

        WriteNotModified()

        WriteWithExpiration(body []byte, modtime time.Time) (int, error)

        StreamWriter(writer func(w io.Writer) bool)

        ClientSupportsGzip() bool

        WriteGzip(b []byte) (int, error)

        TryWriteGzip(b []byte) (int, error)

        GzipResponseWriter() *GzipResponseWriter

        Gzip(enable bool)

        ViewLayout(layoutTmplFile string)

        ViewData(key string, value interface{})

        GetViewData() map[string]interface{}

        View(filename string, optionalViewModel ...interface{}) error
```

```
        Binary(data []byte) (int, error)

        Text(format string, args ...interface{}) (int, error)

        HTML(format string, args ...interface{}) (int, error)

        JSON(v interface{}, options ...JSON) (int, error)

        JSONP(v interface{}, options ...JSONP) (int, error)

        XML(v interface{}, options ...XML) (int, error)

        Markdown(markdownB []byte, options ...Markdown) (int, error)

        YAML(v interface{}) (int, error)

        ServeContent(content io.ReadSeeker, filename string, modtime time.Time,
 gzipCompression bool) error

        ServeFile(filename string, gzipCompression bool) error

        SendFile(filename string, destinationName string) error

        SetCookie(cookie *http.Cookie, options ...CookieOption)

        SetCookieKV(name, value string, options ...CookieOption)

        GetCookie(name string, options ...CookieOption) string

        RemoveCookie(name string, options ...CookieOption)

        VisitAllCookies(visitor func(name string, value string))

        MaxAge() int64

        Record()

        Recorder() *ResponseRecorder

        IsRecording() (*ResponseRecorder, bool)

        BeginTransaction(pipe func(t *Transaction))

        SkipTransactions()

        TransactionsSkipped() bool
```

```
        Exec(method, path string)

        RouteExists(method, path string) bool

        Application() Application

        String() string
    }
```

- `BeginRequest(http.ResponseWriter, *http.Request)`

```
- `BeginRequest` 对每个请求执行一次。
```

  - 它为新来的请求准备 `context` (新的或者从pool获取) 的字段。
  - 为了遵守 Iris 的流程，开发者应该：

```
1. 将处理器重置为 `nil`
2. 将值重置为空
3. 将session重置为 `nil`
4. 将响应writer重置为 `http.ResponseWriter`
5. 将请求重置为 `*http.Request`
```

  - 其他可选的步骤，视开发的应用程序类型而定

- `BeginRequest(http.ResponseWriter, *http.Request)`

```
- 当发送完响应后执行一次，当前的 `context` 无用或者释放。
```

  - 为了遵守 Iris 的流程，开发者应该：

    a. 刷新响应 writer 的结果

    b. 释放响应 writer
  - 其他可选的步骤，视开发的应用程序类型而定

- `ResponseWriter() ResponseWriter`

```
- 预期返回与 response writer 兼容的 `http.ResponseWriter`
```

- `ResetResponseWriter(ResponseWriter)`

```
- 升级或者修改 `Context` 的 `ResponseWriter`
```

- `Request() *http.Request`

```
- 按预期返回原始的 `*http.Request`
```

- `ResetRequest(r *http.Request)`

```
    - 设置 `Context` 的 `Request`
```

  - 通过标准的 `*http.Request` 的 `WithContext` 方法创建的新请求存储到 `iris.Context` 中很有用。
    - 当你出于某种愿意要对 `*http.Request` 完全覆写时，使用 `ResetRequest` 。
    - 记住，当你只想改变一些字段的时候，你可以使用 `Request()` ，它返回一个 Request 的指针，因此在没有完全覆写的情况下，改变也是用效的。
    - 用法：你使用原生的 http 处理器，它使用的是标准库 `context` 来替代 `iris.Context.Values` ，从而获取值。

```
r := ctx.Request()
    stdCtx := context.WithValue(r.Context(), key, val)
    ctx.ResetRequest(r.WithContext(stdCtx)).
```

- `SetCurrentRouteName(currentRouteName string)`

```
- `SetCurrentRouteName` 在内部设置 route 的名字， 目的是为了开发人员调用 `Ge
tCurrentRoute()` 函数时能找到正确的当前的 `只读` 路由。
```

  - 它被路由器初始化，如果你手动改变名字，除了通过 `GetCurrentRoute()` 函数你将会获取到别的 route 之外，没有什么其它影响。
  - 此外，要在该 context 执行不同路径的处理器，你应该使用 `Exec` 函数，或者通过 `SetHandlers/AddHandler` 函数改变头部信息。

- `GetCurrentRoute() RouteReadOnly`

```
- 返回被这个请求路径注册的只读的路由。
```

API版本控制

`versioning` 子包为你的 API 提供 `semver` 版本控制。它实现了书写在 `api-guidelines` 中所有的建议。

版本比较是通过 `go-version` 包比较的。它支持匹配像 `>= 1.0, < 3` 这种比较模式。

```
import (
    // [...]

    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/versioning"
)
```

特性

- 每个路由版本匹配，一个寻常的 Iris 处理器通过version => 处理器的映射来作 `switch-case` 匹配。
- 每个组的版本路由和弃用API
- 版本匹配类似 `>=1.0, <2.0` 和 `2.0.1` 等等形式
- 版本 `not found` 处理器(可以通过简单的添加版本来自定义。 `NotFound` ：在映射上自定义 `NotMatchVersionHandler`)
- 版本从 `Accept` 和 `Accept-Version` 头部取回(可以通过中间件自定义)
- 如果版本找到，响应具有 `X-API-Version` 头
- 通过 `Deprecated` 包装器，弃用了自定义 `X-API-Warn` ， `X-API-Deprecation-Data` ， `X-API-Deprecation-Info` 头部的选项

获取版本

当前请求的版本通过 `versioning.GetVersion(ctx)` 获得。
默认情况下 `GetVersion` 将尝试读取以下内容：

- `Accept header, i.e Accept: "application/json; version=1.0"`
- `Accept-Version header, i.e Accept-Version: "1.0"`

你也可以在中间件中通过使用 context 存储值来设置一个自定义的版本。例如：

```
func(ctx iris.Context) {
    ctx.Values().Set(versioning.Key, ctx.URLParamDefault("version", "1.0"))
    ctx.Next()
}
```

将版本与处理程序匹配

`versioning.NewMatcher(versioning.Map) iris.Handler` 创建一个简单的处理器，这个处理器决定基于请求的版本，哪个处理器需要被执行。

```
app := iris.New()

// middleware for all versions.
myMiddleware := func(ctx iris.Context) {
    // [...]
    ctx.Next()
}

myCustomNotVersionFound := func(ctx iris.Context) {
    ctx.StatusCode(404)
    ctx.Writef("%s version not found", versioning.GetVersion(ctx))
}

userAPI := app.Party("/api/user")
userAPI.Get("/", myMiddleware, versioning.NewMatcher(versioning.Map{
    "1.0":               sendHandler(v10Response),
    ">= 2, < 3":         sendHandler(v2Response),
    versioning.NotFound: myCustomNotVersionFound,
}))
```

弃用

使用 `versioning.Deprecated(handler iris.Handler, options versioning.DeprecationOptions) iris.Handler` 函数，你可以标记特定的处理器版本为被弃用的。

```
v10Handler := versioning.Deprecated(sendHandler(v10Response), versioning.DeprecationOptions{
    // if empty defaults to: "WARNING! You are using a deprecated version of this API."
    WarnMessage string
    DeprecationDate time.Time
```

```
    DeprecationInfo string
})
userAPI.Get("/", versioning.NewMatcher(versioning.Map{
    "1.0": v10Handler,
    // [...]
}))
```

这将会让处理器发送这些头到客户端：

- `"X-API-Warn": options.WarnMessage`
- `"X-API-Deprecation-Date": context.FormatTime(ctx, options.DeprecationDate))`
- `"X-API-Deprecation-Info": options.DeprecationInfo`

如果你不在意时间和日期，你可以使用 `versioning.DefaultDeprecationOptions` 替代选项。

通过版本分组路由

也可以按版本对路由进行分组。

使用 `versioning.NewGroup(version string) *versioning.Group` 函数你可以创建一个组来注册你的版本路由。 `versioning.RegisterGroups(r iris.Party, versionNotFoundHandler iris.Handler, groups ...*versioning.Group)` 必须在最后被调用，为了使用路由被注册到特定的 `Party` 中。

```
app := iris.New()

userAPI := app.Party("/api/user")
// [... static serving, middlewares and etc goes here].

userAPIV10 := versioning.NewGroup("1.0")
userAPIV10.Get("/", sendHandler(v10Response))

userAPIV2 := versioning.NewGroup(">= 2, < 3")
userAPIV2.Get("/", sendHandler(v2Response))
userAPIV2.Post("/", sendHandler(v2Response))
userAPIV2.Put("/other", sendHandler(v2Response))

versioning.RegisterGroups(userAPI, versioning.NotFoundHandler, userAPIV10, userA
PIV2)
```

> 使用上面我们学习的方法，一个中间件仅仅只能被注册到实际的 `iris.Party` ，即使用 `versioning.Match` 为了检测当 `x` 或者没有版本要求的时候你想使用什么

> `code/handler` 来执行。

弃用组

仅在你想要改变你的API消费者的组上调用 `Deprecated(versioning.DeprecationOptions)` ，这个指定的版本就被丢弃了。

```
userAPIV10 := versioning.NewGroup("1.0").Deprecated(versioning.DefaultDeprecatio
nOptions)
```

在内部的处理器中手动比较版本

```
// reports if the "version" is matching to the "is".
// the "is" can be a constraint like ">= 1, < 3".
If(version string, is string) bool
```

```
// same as `If` but expects a Context to read the requested version.
Match(ctx iris.Context, expectedVersion string) bool

app.Get("/api/user", func(ctx iris.Context) {
    if versioning.Match(ctx, ">= 2.2.3") {
        // [logic for >= 2.2.3 version of your handler goes here]
        return
    }
})
```

内容协商

有时候一个服务器应用程序需要在相同的 URI 上提供资源不同表示形式。当然这也可以手动实现，检查 `Accept` 请求头，设置推送的请求表单的文本。然而，当你的程序管理更多的资源和不同形式的时候，这将变得非常痛苦，你需要检查 `Accept-Charset` ， `Accept-Encoding` ，设置一些服务器端优先级，正确处理错误等等。

有一些 Go 的 web 框架已经艰难地实现了这个特性，但是它们不能正确地做以下的事情：

- 不处理 `accept-charset`
- 不处理 `accept-encoding`
- 不会发送 `RFC` 提出的一些错误状态码(`406 not acceptable`)。

但是我对我来说幸运的是，Iris 始终遵循最佳实践和Web标准。

基于：

- `https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation`
- `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept`
- `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Charset`
- `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Encoding`

实现于：

- `https://github.com/kataras/iris/pull/1316/commits/8ee0de51c593fe0483fbea38117c3c88e065f2ef`

示例

```
type testdata struct {
    Name string `json:"name" xml:"Name"`
    Age  int    `json:"age" xml:"Age"`
}
```

通过 `gzip` 编码算法将资源渲染为 `application/json` 或者 `text/html` 或者 `application/xml` 。

- 当客户端的 `Accept` 头部包含上述之一时，
- 如果接收的为空就声明为 `JSON` (首先声明)

- 当客户端的 `Accpet-Encoding` 头包含 `gzip` 或者为空时

```
    app.Get("/resource", func(ctx iris.Context) {
        data := testdata{
            Name: "test name",
            Age:  26,
        }

            ctx.Negotiation().JSON().XML().EncodingGzip()

        _, err := ctx.Negotiate(data)
        if err != nil {
            ctx.Writef("%v", err)
        }
    })
```

或者在一个中间件中定义他们，然后在最后的处理器中以 `nil` 参数调用 `Negotiate` 。

```
ctx.Negotiation().JSON(data).XML(data).Any("content for */*")
ctx.Negotiate(nil)

app.Get("/resource2", func(ctx iris.Context) {
    jsonAndXML := testdata{
        Name: "test name",
        Age:  26,
    }

    ctx.Negotiation().
        JSON(jsonAndXML).
        XML(jsonAndXML).
        HTML("<h1>Test Name</h1><h2>Age 26</h2>")

    ctx.Negotiate(nil)
})
```

文献资料

`Context.Negotiotion` 方法创建一次，返回“协商构造器”，以便构建适用于特定文本类型，字符和编码算法的服务器端的可用内容。

```
Context.Negotiation() *context.NegotiationBuilder
```

`Context.Negotiotion` 方法用于服务相同 URI 的不同表现形式的资源。当无法匹配 MIME 类型它返回 `context.ErrContentNotSupported` 。

```
Context.Negotiate(v interface{}) (int, error)
```

- `v` 接口可以是一个 `iris.N` 结构体类型值。
- `v` 接口可以是任何实现了 `context.ContentSelector` 接口的值。
- `v` 接口可以是任何实现了 `context.ContentNegotiator` 接口的值。
- `v` 接口可以是结构体 `(JSON、JSONP、XML、YAML)` ，或者 字符串 `(TEXT, HTML)` ，或者字节切片 `(Markdown, Binary)` ，或者匹配的 MIME 类型的字节切片。
- 如果 `v` 接口是 `nil` ，将会使用 `Context.Negotitation()` 构造器的内容来替代，除此之外， `v` 接口覆写了构造器的内容(服务器的 MIME 类型通过其注册的、支持的 MIME 列表进行检索)
- 通过 `Negotiation()` 返回的构造器 的 `MIME` ， `Text` ， `JSON` ， `XML` ， `HTML` 等等方法设置MIME类型优先级。
- 通过 `Negotiation()` 返回的构造器 的 `Charset` 方法设置字符串类型优先级。
- 通过 `Negotiation()` 返回的构造器 的 `Encoding` 方法设置编码算法优先级。
- 通过 `Negotiation()` 返回的构造器 的 `Accept` ， `Override` ， `XML` 等等方法修改 `Accept` 。

响应记录器

一个响应记录器是 Iris 特定的 `http.ResponseWriter` ，它记录了发送的响应体，状态码和响应头，你可以在任何这个路由的请求处理器链上的处理器内部操作它。

- 在发送数据前调用 `Context.Record`
- `Context.Recorder()` 返回一个 `ResponseRecorder` 。它的方法可以用来操作和找回响应。

`ResponseRecorder` 类型包含了标准的 Iris ResponseWriter 方法和下列的方法。

`Body` 返回目前为止 Writer 写入的响应体数据。不要使用它来编辑响应体。

```
Body() []byte
```

使用这个来清除响应体

```
ResetBody()
```

使用 `Write/Writef/WriteString` 流式写入， `SetBody/SetBodyString` 设置响应体。

```
Write(contents []byte) (int, error)

Writef(format string, a ...interface{}) (n int, err error)

WriteString(s string) (n int, err error)

SetBody(b []byte)

SetBodyString(s string)
```

在调用 `Context.Record` 之前，使用 `ResetHeaders` 重置响应头为原始状态。

```
ResetHeaders()
```

清除所有的头部信息。

```
ClearHeaders()
```

同时重置响应体，响应头和状态码。

```
Reset()
```

示例

在一个全局拦截器中记录操作日志。

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()

    // start record.
    app.Use(func(ctx iris.Context) {
        ctx.Record()
        ctx.Next()
    })

    // collect and "log".
    app.Done(func(ctx iris.Context) {
        body := ctx.Recorder().Body()

        // Should print success.
        app.Logger().Infof("sent: %s", string(body))
    })
}
```

注册路由：

```
app.Get("/save", func(ctx iris.Context) {
    ctx.WriteString("success")
    ctx.Next() // calls the Done middleware(s).
})
```

或者为了消除你的主处理器中对 `ctx.Next` 的需求，改变 Iris 处理器的执行规则，你可以如下所示：

```
// It applies per Party and its children,
// therefore, you can create a routes := app.Party("/path")
// and set middlewares, their rules and the routes there as well.
```

```
app.SetExecutionRules(iris.ExecutionRules{
    Done: iris.ExecutionOptions{Force: true},
})

// [The routes...]
app.Get("/save", func(ctx iris.Context) {
    ctx.WriteString("success")
})
```

HTTP Referer

HTTP referer (本来是 `referrer` 的拼写错误) 是一个可选的 HTTP 头部字段， 用于标记链接到请求资源的网页的地址(即 URI 或者 IRI)。通过检查 referrer，新的网页可以知道请求的来源。

Iris 使用 `Shopify's goreferrer` 包来实现 `Context.GetReferrer()` 方法。

`GetReferrer` 方法提取和返回 `Referer` 头的信息， `Referer` 通过 **https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy** 或者 URL 的 `referer` 查询参数(`query parameter`)指定。

```
GetReferrer() Referrer
```

`Referrer` 是这样的：

```
type (
    Referrer struct {
        Type       ReferrerType
        Label      string
        URL        string
        Subdomain  string
        Domain     string
        Tld        string
        Path       string
        Query      string
        GoogleType ReferrerGoogleSearchType
    }
```

`ReferrerType` 是 `Referrer.Type` 值(`indirect` ， `direct` ， `email` ， `search` ， `social`)的枚举。可以的类型有：

- ReferrerInvalid
- ReferrerIndirect
- ReferrerDirect
- ReferrerEmail
- ReferrerSearch
- ReferrerSocial

`GoogleType` 可以是下列之一：

- ReferrerNotGoogleSearch
- ReferrerGoogleOrganicSearch
- ReferrerGoogleAdwords

示例

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()

    app.Get("/", func(ctx iris.Context) {
        r := ctx.GetReferrer()
        switch r.Type {
        case iris.ReferrerSearch:
            ctx.Writef("Search %s: %s\n", r.Label, r.Query)
            ctx.Writef("Google: %s\n", r.GoogleType)
        case iris.ReferrerSocial:
            ctx.Writef("Social %s\n", r.Label)
        case iris.ReferrerIndirect:
            ctx.Writef("Indirect: %s\n", r.URL)
        }
    })

    app.Run(iris.Addr(":8080"))
}
```

`curl` :

```
curl http://localhost:8080?\
referer=https://twitter.com/Xinterio/status/1023566830974251008

curl http://localhost:8080?\
referer=https://www.google.com/search?q=Top+6+golang+web+frameworks\
&oq=Top+6+golang+web+frameworks
```

请求认证

Iris 通过它的 `jwt` 中间件，提供请求权限验证。这个章节你将会学习怎么在 Iris 中使用 JWT 的基础。

1. 使用下面的命令安装

```
$ go get github.com/iris-contrib/middleware/jwt
```

2. 使用 `jwt,New` 函数创建一个新的 jwt中间件。这个示例中通过 `token` url参数提取 token。经过身份验证的客户端应设计为使用签名令牌进行设置。默认的 jwt 中间件的行为是通过 `Authentication: Bearer $TOKEN` 头部来提取 token 的值。

   jwt 中间件有3个方法来验证 token：

   - 第一个方法是 `Serve` 方法，这是一个 `iris.Handler`
   - 第二个方法是 `CheckJWT(iris.Context) bool`
   - 第三个方法是 `Get(iris.Context) *jwt.Token` ，这是一个用于取回已验证的 token。

3. 要注册它，您只需在jwt j.Serve中间件之前添加特定的路由组，单个路由或全局路由即可。

```
app.Get("/secured", j.Serve, myAuthenticatedHandler)
```

4. 在一个处理器中生成一个token，接受一个用户的有效负载和响应已签名的token，然后可以通过客户端的请求头或者url参数发送。 `jwt.NewToken` ， `jwt.NewTokenWithClaims`

示例

```
import (
    "github.com/kataras/iris/v12"
    "github.com/iris-contrib/middleware/jwt"
)

func getTokenHandler(ctx iris.Context) {
    token := jwt.NewTokenWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
        "foo": "bar",
    })
```

```
    // Sign and get the complete encoded token as a string using the secret
    tokenString, _ := token.SignedString([]byte("My Secret"))

    ctx.HTML(`Token: ` + tokenString + `<br/><br/>
    <a href="/secured?token=` + tokenString + `">/secured?token=` + tokenString
 + `</a>`)
}

func myAuthenticatedHandler(ctx iris.Context) {
    user := ctx.Values().Get("jwt").(*jwt.Token)

    ctx.Writef("This is an authenticated request\n")
    ctx.Writef("Claim content:\n")

    foobar := user.Claims.(jwt.MapClaims)
    for key, value := range foobar {
        ctx.Writef("%s = %s", key, value)
    }
}

func main(){
    app := iris.New()

    j := jwt.New(jwt.Config{
        // Extract by "token" url parameter.
        Extractor: jwt.FromParameter("token"),

        ValidationKeyGetter: func(token *jwt.Token) (interface{}, error) {
            return []byte("My Secret"), nil
        },
        SigningMethod: jwt.SigningMethodHS256,
    })

    app.Get("/", getTokenHandler)
    app.Get("/secured", j.Serve, myAuthenticatedHandler)
    app.Run(iris.Addr(":8080"))
}
```

URL查询参数

Irir的 `Context` 有两个方法，返回的是我们已经在前面的章节中提到的 `net/http` 标准的 `http.ResponseWriter` 和 `http.Request` 。

- `Context.Request()`
- `Context.ResponseWriter()`

然而，除了 Iris Context 提供的独特的 Iris 特性和帮助，为了更易于开发，我们提供了一些现有 `net/http` 功能的包装器。

这里是完整的方法列表，在你处理 URL 查询字符串时可能帮助你。

```
// URLParam returns true if the url parameter exists, otherwise false.
URLParamExists(name string) bool

// URLParamDefault returns the get parameter from a request,
// if not found then "def" is returned.
URLParamDefault(name string, def string) string

// URLParam returns the get parameter from a request, if any.
URLParam(name string) string

// URLParamTrim returns the url query parameter with
// trailing white spaces removed from a request.
URLParamTrim(name string) string

// URLParamTrim returns the escaped url query parameter from a request.
URLParamEscape(name string) string

// URLParamInt returns the url query parameter as int value from a request,
// returns -1 and an error if parse failed.
URLParamInt(name string) (int, error)

// URLParamIntDefault returns the url query parameter as int value from a reques
t,
// if not found or parse failed then "def" is returned.
URLParamIntDefault(name string, def int) int

// URLParamInt32Default returns the url query parameter as int32 value from a re
quest,
// if not found or parse failed then "def" is returned.
URLParamInt32Default(name string, def int32) int32
```

```
// URLParamInt64 returns the url query parameter as int64 value from a request,
// returns -1 and an error if parse failed.
URLParamInt64(name string) (int64, error)

// URLParamInt64Default returns the url query parameter as int64 value from a re
quest,
// if not found or parse failed then "def" is returned.
URLParamInt64Default(name string, def int64) int64

// URLParamFloat64 returns the url query parameter as float64 value from a reque
st,
// returns -1 and an error if parse failed.
URLParamFloat64(name string) (float64, error)

// URLParamFloat64Default returns the url query parameter as float64 value from
 a request,
// if not found or parse failed then "def" is returned.
URLParamFloat64Default(name string, def float64) float64

// URLParamBool returns the url query parameter as boolean value from a request,
// returns an error if parse failed or not found.
URLParamBool(name string) (bool, error)

// URLParams returns a map of GET query parameters separated by comma if more th
an one
// it returns an empty map if nothing found.
URLParams() map[string]string
```

查询字符串参数通过使用已有的底层的 `request` 对象解析。这个请求响应一个匹配 `/welcome?firstname=Jane&lastname=Doe` 的 URL。

- `ctx.URLParam("lastname") == ctx.Request().URL.Query().Get("lastname")`

示例代码：

```
app.Get("/welcome", func(ctx iris.Context) {
    firstname := ctx.URLParamDefault("firstname", "Guest")
    lastname := ctx.URLParam("lastname")

    ctx.Writef("Hello %s %s", firstname, lastname)
})
```

表单

表单，post的数据和上传的文件可以使用下面的 Context 的方法获取。

```
// FormValueDefault 返回一个根据名字获取的form的值，
// 其中可能是URL的查询参数和POST或者PUT的数据，
// 如果没有找到返回 def 指定的值。
FormValueDefault(name string, def string) string

// FormValue 返回一个根据名字获取的form的值，
// 其中可能是URL的查询参数和POST或者PUT的数据，
FormValue(name string) string

//  FormValues 返回一个根据名字获取的form的值，
// 其中可能是URL的查询参数和POST或者PUT的数据，
// 默认的form的内存最大尺寸是32MB，
// 这个可以通过在 app.Run() 的第二个参数传入 iris.WithPostMaxMemory 配置器来改变这个大小。
// 记住：检查返回值是否为 nil 是有必要的！
FormValues() map[string][]string

// PostValueDefault 返回通过解析POST，PATCH或者PUT请求体参数，指定名字对应的值。
// 如果没有找到这个名字则返回 def 指定的默认值。
PostValueDefault(name string, def string) string

// PostValue 返回通过解析POST，PATCH或者PUT请求体参数，指定名字对应的值。
PostValue(name string) string

// PostValueTrim 返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的没有前后空格的值。
PostValueTrim(name string) string

// PostValueInt 返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的int的值。
// 如果没有找到name对应的值，则返回-1和一个非nil的错误。
PostValueInt(name string) (int, error)

// PostValueIntDefault  返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的int的值。
// 如果没有找到name对应的值，则 def 指定的默认值。
PostValueIntDefault(name string, def int) int

// PostValueInt64 返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的int64的值。
```

```
// 如果没有找到name对应的值，则返回-1和一个非nil的错误。
PostValueInt64(name string) (int64, error)

// PostValueInt64Default  返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的int64的值。
// 如果没有找到name对应的值，则 def 指定的默认值。
PostValueInt64Default(name string, def int64) int64

// PostValueFloat64 返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的float6464的值。
// 如果没有找到name对应的值，则返回-1和一个非nil的错误。
PostValueFloat64(name string) (float64, error)

// PostValueFloat64Default  返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的float64的值。
// 如果没有找到name对应的值，则 def 指定的默认值。
PostValueFloat64Default(name string, def float64) float64

// PostValueBool  返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的bool的值。
// 如果没有找到name对应的值，则返回false和一个非nil的错误。
PostValueBool(name string) (bool, error)

// PostValues  返回通过解析POST，PATCH或者PUT请求体参数，
// 指定名字对应的[]string的值。
// 默认的form的内存最大尺寸是32MB，
// 这个可以通过在 app.Run() 的第二个参数传入 iris.WithPostMaxMemory 配置器来改变这个大小。
// 记住：检查返回值是否为 nil 是有必要的！
PostValues(name string) []string

// FormFile 返回第一个从客户端上传的文件。
// 默认的form的内存最大尺寸是32MB，
// 这个可以通过在 app.Run() 的第二个参数传入 iris.WithPostMaxMemory 配置器来改变这个大小。
// 记住：检查返回值是否为 nil 是有必要的！
FormFile(key string) (multipart.File, *multipart.FileHeader, error)
```

Multipart/Urlencoded Form

```
func main() {
    app := iris.Default()

    app.Post("/form_post", func(ctx iris.Context) {
        message := ctx.FormValue("message")
```

```
        nick := ctx.FormValueDefault("nick", "anonymous")

        ctx.JSON(iris.Map{
            "status":  "posted",
            "message": message,
            "nick":    nick,
        })
    })

    app.Run(iris.Addr(":8080"))
}
```

另一个例子：query + post form

```
POST /post?id=1234&page=1 HTTP/1.1
Content-Type: application/x-www-form-urlencoded

name=manu&message=this_is_great
```

```
func main() {
    app := iris.Default()

    app.Post("/post", func(ctx iris.Context) {
        id := ctx.URLParam("id")
        page := ctx.URLParamDefault("page", "0")
        name := ctx.FormValue("name")
        message := ctx.FormValue("message")
        // or `ctx.PostValue` for POST, PUT & PATCH-only HTTP Methods.

        app.Logger().Infof("id: %s; page: %s; name: %s; message: %s",
            id, page, name, message)
    })

    app.Run(iris.Addr(":8080"))
}
```

```
id: 1234; page: 1; name: manu; message: this_is_great
```

上传文件

Context 提供了上传一个用于上传文件的助手(从请求的文件数据中保存文件到主机系统的硬盘上)。阅读下面的 `Context.UploadFormFiles` 方法。

```
UploadFormFiles(destDirectory string,
        before ...func(Context, *multipart.FileHeader)) (n int64, err error)
```

`UploadFromFile` 上载任何从客户端获取的文件到系统物理 `destDirctory` 位置。

第二个参数 `before` 给定可调用的函数，这些函数可以在保存到磁盘之前改变 `*miltipart.FileHeader` ，它可以用来基于当前请求改变文件的名字，并且所有的 `FileHeader` 的选项都可以改变。如果你不需要在保存文件到硬盘之前使用这个特性，你可以忽略这个参数。

请注意，它不会检查请求正文是否流式传输。

返回复制的长度的int64值， 和由于操作系统权限导致一个新文件无法创建的非 `nil` 的错误，或者由于没有文件获取而返回 `net/http.ErrMissingFile` 错误。

如果你想接收并接受文件，并且手动管理它们，你可以使用 `Context.FormFile` ，创建一个复制的函数，满足你的需求，下面是通用用法。

默认的form的内存限制是32MB，你通过在主配置时传递 `iris.WithPostMaxMemory` 配置器到 `app.Run` 的第二个参数来改变这个限制。

示例代码：

```go
func main() {
    app := iris.Default()
    app.Post("/upload", iris.LimitRequestBodySize(maxSize), func(ctx iris.Context) {
        //
        // UploadFormFiles
        // uploads any number of incoming files ("multiple" property on the form input).
        //

        // The second, optional, argument
        // can be used to change a file's name based on the request,
        // at this example we will showcase how to use it
        // by prefixing the uploaded file with the current user's ip.
        ctx.UploadFormFiles("./uploads", beforeSave)
    })

    app.Run(iris.Addr(":8080"))
}

func beforeSave(ctx iris.Context, file *multipart.FileHeader) {
    ip := ctx.RemoteAddr()
```

```
    // make sure you format the ip in a way
    // that can be used for a file name (simple case):
    ip = strings.Replace(ip, ".", "_", -1)
    ip = strings.Replace(ip, ":", "_", -1)

    // you can use the time.Now, to prefix or suffix the files
    // based on the current time as well, as an exercise.
    // i.e unixTime :=    time.Now().Unix()
    // prefix the Filename with the $IP-
    // no need for more actions, internal uploader will use this
    // name to save the file into the "./uploads" folder.
    file.Filename = ip + "-" + file.Filename
}
```

模型验证

Iris 没有内建的方法来验证请求数据，例如 `Models` 。然而，你并没有因此受到限制。在这个示例中，我们可以学习怎么使用 `go-playground/validator.v9` 来验证请求数据。

```
$ go get gopkg.in/go-playground/validator.v9
```

记住你需要在所有的字段设置相应的你想绑定的 `tag` 。例如，当你为 `JSON` 绑定时，设置 `json:"fieldname"` 。

```
package main

import (
    "fmt"

    "github.com/kataras/iris/v12"

    "gopkg.in/go-playground/validator.v9"
)

// User contains user information.
type User struct {
    FirstName      string     `json:"fname"`
    LastName       string     `json:"lname"`
    Age            uint8      `json:"age" validate:"gte=0,lte=130"`
    Email          string     `json:"email" validate:"required,email"`
    FavouriteColor string     `json:"favColor" validate:"hexcolor|rgb|rgba"`
    Addresses      []*Address `json:"addresses" validate:"required,dive,require
d"`
}

// Address houses a users address information.
type Address struct {
    Street string `json:"street" validate:"required"`
    City   string `json:"city" validate:"required"`
    Planet string `json:"planet" validate:"required"`
    Phone  string `json:"phone" validate:"required"`
}

// Use a single instance of Validate, it caches struct info.
var validate *validator.Validate

func main() {
```

```
    validate = validator.New()

    // Register validation for 'User'
    // NOTE: only have to register a non-pointer type for 'User', validator
    // internally dereferences during it's type checks.
    validate.RegisterStructValidation(UserStructLevelValidation, User{})

    app := iris.New()
    app.Post("/user", func(ctx iris.Context) {
        var user User
        if err := ctx.ReadJSON(&user); err != nil {
            // [handle error...]
        }

        // Returns InvalidValidationError for bad validation input,
        // nil or ValidationErrors ( []FieldError )
        err := validate.Struct(user)
        if err != nil {

            // This check is only needed when your code could produce
            // an invalid value for validation such as interface with nil
            // value most including myself do not usually have code like this.
            if _, ok := err.(*validator.InvalidValidationError); ok {
                ctx.StatusCode(iris.StatusInternalServerError)
                ctx.WriteString(err.Error())
                return
            }

            ctx.StatusCode(iris.StatusBadRequest)
            for _, err := range err.(validator.ValidationErrors) {
                fmt.Println()
                fmt.Println(err.Namespace())
                fmt.Println(err.Field())
                fmt.Println(err.StructNamespace())
                fmt.Println(err.StructField())
                fmt.Println(err.Tag())
                fmt.Println(err.ActualTag())
                fmt.Println(err.Kind())
                fmt.Println(err.Type())
                fmt.Println(err.Value())
                fmt.Println(err.Param())
                fmt.Println()
            }

            return
        }
```

```
        // [save user to database...]
    })

    app.Run(iris.Addr(":8080"))
}

func UserStructLevelValidation(sl validator.StructLevel) {
    user := sl.Current().Interface().(User)

    if len(user.FirstName) == 0 && len(user.LastName) == 0 {
        sl.ReportError(user.FirstName, "FirstName", "fname", "fnameorlname", "")
        sl.ReportError(user.LastName, "LastName", "lname", "fnameorlname", "")
    }
}
```

json表单的示例请求：

```
{
    "fname": "",
    "lname": "",
    "age": 45,
    "email": "mail@example.com",
    "favColor": "#000",
    "addresses": [{
        "street": "Eavesdown Docks",
        "planet": "Persphone",
        "phone": "none",
        "city": "Unknown"
    }]
}
```

缓存

有时缓存路由的静态内容是很重要的，因为这样可以使你的web程序性能更快，不会花时间在重构响应上。

这里有两个方式实现 HTTP 缓存。一个是在服务器端存储每个处理器的内容，另一个是检查请求头，然后发送 `304 not modified` ，以便让浏览器或者任何兼容的客户端自己来处理缓存。

服务器和客户端缓存的方式，Iris 都提供了，通过 `iris/cache` 子包来实现的，这个子包提供了 `iris.Cache` 和 `iris.Cache304` 中间件。

Cache

`Cache` 是一个中间件，为接下来的处理器提供了服务器端缓存功能，可以这样来使用： `app.Get("/", iris.Cache(time.Hour), aboutHandler)` 。

`Cache` 仅仅只需接收一个参数：缓存的生存时间。如果生存时间无效或者 <= 2秒，会从 `cache-control's maxage` 头部获取生存时间。

```
func Cache(expiration time.Duration) Handler
```

所有类型的响应都可以被缓存，模板，json，文本，任何类型。

使用它来达到服务器端缓存，查看 `Cache304` 获取另外的方法可能会更加适合你的需求。

有关更多选项和自定义，请使用kataras / iris / cache.Cache，它返回一个结构，您可以从中添加或删除“规则”。

NoCache

`NoCache` 是一个中间件，它重写了 `Cache-Control` ， `Pragma` 和 `Expires` 头部以便在浏览器使用前进和后退功能期间禁用缓存。

在 HTML 路由上使用这个中间件；即使在浏览器的“后退”和“前进”箭头按钮上也可以刷新页面。

```
func NoCache(Context)
```

查看 `StaticCache` 获取相反的行为。

StaticCache

`StaticCache` 返回一个中间件，通过向客户端发送 `Cache-Control` 和 `Expires` 头来缓存静态文件。它仅接受一个参数，是一个 `time.Duration` 类型的，用于计算有效期。

如果 `cacheDur` <= 0，则返回 `Nocache` 中间件来禁用浏览器在前进和后退行为时的缓存。

使用： `app.Use(iris.StaticCache(24 * time.Hour))` 或者 `app.Use(iris.StaticCache(-1))`

```
func StaticCache(cacheDur time.Duration)Handler
```

一个中间件，是一个简单的处理器，可以在其他处理器内部调用，例如：

```
cacheMiddleware := iris.StaticCache(...)

func(ctx iris.Context){
 cacheMiddleware(ctx)
 [...]
}
```

Cache304

`Cache304` 返回一个中间件，没当 `If-Modify-Since` 请求头(时间值) 在 time.Now() + 生存时间之前，就会发送 `StatusNotModified(304)` 。

所有兼容HTTP RCF的客户端(所有的浏览器和类似postman的工具)将会正确地处理缓存。

这个方法的唯一缺点就是将会发送一个 304 的状态码而不是 200，因此，如果您将其与其他微服务一起使用，则必须检查该状态码以及有效的响应。

开发人员可以通过手动查看系统目录的更改并根据文件修改日期使用 `ctx.WriteWithExpiration` （带有“ modtime”）来自由扩展此方法的行为，类似于 `HandleDir` （发送状态为OK（200）和浏览器磁盘缓存的方式， 304）。

```
func Cache304(expiresEvery time.Duration) Handler
```

文件服务

通过 `Party.HandleDir` 方法指定特定的目录(系统目录或者嵌入式应用)以获取静态文件。

`HandleDir` 注册一个处理程序，该处理程序使用文件系统的内容（物理或嵌入式）为HTTP请求提供服务。

- 第一个参数：路由路径
- 第二个参数：需要保存文件的系统或者嵌入式目录
- 第三个参数：不必要，目录选项，设置字段是可选的。

返回 `*Route`

```
HandleDir(requestPath, directory string, opts ...DirOptions) (getRoute *Route)
```

`DirOptions` 结构体是这样的：

```
type DirOptions struct {
    // 默认为 /index.html, 如果请求路径以 **/*/$IndexName 结尾，它会重定向到 **/*(/),
    // 然后另一个处理器会处理它，
    // 如果最后开发人员没有设法手动处理它，
    // 框架会自动注册一个名为index handler 的处理器来作为这个处理器
    IndexName string

   // 文件是否需要gzip压缩
    Gzip bool

    // 如果 IndexName没有找到，是否列出当前请求目录中的文件
    ShowList bool

    // 如果 ShowList 为 true，这个函数将会替代默认的列出当前请求目录中文件的函数。
    DirList func(ctx iris.Context, dirName string, dir http.File) error

    // 内嵌时
    Asset      func(name string) ([]byte, error)
    AssetInfo  func(name string) (os.FileInfo, error)
    AssetNames func() []string

    //  循环遍历每个找到的请求资源的可选验证器。
    AssetValidator func(ctx iris.Context, name string) bool
}
```

让我假设在你的可执行文件目录中有一个 `./assets` 文件夹，你想要处理 `http://localhost:8080/static/**/*` 路由下的文件。

```
app := iris.New()

app.HandleDir("/static", "./assets")

app.Run(iris.Addr(":8080"))
```

现在，如果您想将静态文件嵌入可执行文件内部以不依赖于系统目录，则可以使用 `go-bindata` 之类的工具将文件转换为程序内的[] byte。 让我们快速学习一下，以及Iris如何帮助服务这些数据。

安装 `go-bindata`

```
go get -u github.com/go-bindata/go-bindata/...
```

导航到你程序的目录，并且 `./assets` 子目录存在，然后执行：

```
$ go-bindata ./assets/...
```

上面创建了一个go文件，其中包含三个主要的函数： `Asset` ， `AssetInfo` ， `AssetNames` 。在 `iris.DirOptions` 中使用它们：

```
// [app := iris.New...]

app.HandleDir("/static", "./assets", iris.DirOptions {
    Asset: Asset,
    AssetInfo: AssetInfo,
    AssetNames: AssetNames,
    Gzip: false,
})
```

编译你的程序：

```
$ go build
```

`HandleDir` 物理目录和内嵌目录的所有标准，包括 `content-range` 。

然而，如果你仅仅想要一个处理器工作，而不注册路由，你可以使用 `iris.FileServer` 包级别函数替代。

`FileServer` 函数返回一个处理器，该处理器为来自特定系统，物理目录，嵌入式目录的文件提供服务。

- 第一个参数：目录
- 第一个参数：可选参数，调用者可选的配置

```
    iris.FileServer(directory string, options ...DirOptions)
```

使用：

```
handler := iris.FileServer("./assets", iris.DirOptions {
    ShowList: true, Gzip: true, IndexName: "index.html",
})
```

视图

Iris 通过它的视图引擎提供了 6 种开箱即用的模板解析器。当然开发者仍然可以使用各种go模板解析器作为 `Context.ResponseWriter()` ，只要实现了 `http.ResponseWriter` 和 `io.Writer` 。

Iris提出了一些默认的解析器不支持的通用规则和功能。例如，我们已经支持 `yield` ， `render` ， `render_r` ， `current` ， `urlpath` 模板函数，并通过中间件和嵌入式模板文件为所有的引擎布局和绑定。

为了使用一个模板引擎独特的功能，你需要通过阅读文档来学习模板引擎的特征和语法(点击下面)。选择最适合你app需要的。

内置的视图引擎：

- 标准 template/html： `iris.HTML(...)`
- django： `iris.Django(...)`
- handlebars： `iris.Handlebars(...)`
- amber： `iris.Amber(...)`
- pug(jade)： `iris.Pug(...)`
- jet： `iris.Jet(...)`

一个或者多个视图引擎可以注册到相同的应用中。使用 `RegisterView(ViewEngine)` 方法注册。

从 `./views` 目录中加载所有后缀为 `.html` 的模板，然后使用标准库 `html/template` 包来解析它们。

```
// [app := iris.New...]
tmpl := iris.HTML("./views", ".html")
app.RegisterView(tmpl)
```

在路由的处理器中用 `Context.View` 方法渲染或者执行一个视图。

```
ctx.View("hi.html")
```

在使用 `Context.View` 之前使用 `Context.ViewData` 方法绑定一个Go的键值对。

绑定 `{{.message}}` 为 `hello world`

```
ctx.ViewData("message", "Hello world!")
```

你有两种方法绑定一个go模型：

- 第一种

```
    ctx.ViewData("user", User{})

    // variable binding as {{.user.Name}}
```

- 第二种

```
    ctx.View("user-page.html", User{})

    // root binding as {{.Name}}
```

要添加一个模板函数， 请使用首选视图引擎的 `AddFunc` 方法。

```
//       func name, input arguments, render value
tmpl.AddFunc("greet", func(s string) string {
    return "Greetings " + s + "!"
})
```

要重新加载本地文件更改，请调用视图引擎的 `Reload` 方法。

```
tmpl.Reload(true)
```

使用嵌入式的模板并且不依赖本地文件系统，使用 `go-bindata` 外部工具，然后把 `Asset` 和 `AssetName` 函数传递到首选视图引擎的 `Binary` 方法。

```
tmpl.Binary(Asset, AssetNames)
```

示例代码，请阅读注释：

```
// file: main.go
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()

    // Parse all templates from the "./views" folder
    // where extension is ".html" and parse them
```

```
    // using the standard `html/template` package.
    tmpl := iris.HTML("./views", ".html")

    // Enable re-build on local template files changes.
    tmpl.Reload(true)

    // Default template funcs are:
    //
    // - {{ urlpath "myNamedRoute" "pathParameter_ifNeeded" }}
    // - {{ render "header.html" }}
    // and partial relative path to current page:
    // - {{ render_r "header.html" }}
    // - {{ yield }}
    // - {{ current }}
    // Register a custom template func:
    tmpl.AddFunc("greet", func(s string) string {
        return "Greetings " + s + "!"
    })

    // Register the view engine to the views,
    // this will load the templates.
    app.RegisterView(tmpl)

    // Method:    GET
    // Resource:  http://localhost:8080
    app.Get("/", func(ctx iris.Context) {
        // Bind: {{.message}} with "Hello world!"
        ctx.ViewData("message", "Hello world!")
        // Render template file: ./views/hi.html
        ctx.View("hi.html")
    })

    app.Run(iris.Addr(":8080"))
}
```

hi.html：

```
<!-- file: ./views/hi.html -->
<html>
<head>
    <title>Hi Page</title>
</head>
<body>
    <h1>{{.message}}</h1>
    <strong>{{greet "to you"}}</strong>
```

```
</body>
</html>
```

浏览器源码：

```
<html>
<head>
    <title>Hi Page</title>
</head>
<body>
    <h1>Hello world!</h1>
    <strong>Greetings to you!</strong>
</body>
</html>
```

Cookies

通过 Context 的请求实例可以地访问Cookies。 `ctx.Request()` 返回一个 `net/http.Request` 实例。

Iris 的 `Context` 提供了一个工具使得你可以更加容易地访问最常见的cookies用例，并且不需要任何你的自定义额外的代码，只需要使用 Request 的 cookie 方法就能满足。

设置cookie

`SetCookie` 方法添加一个cookie

```
SetCookie(cookie *http.Cookie, options ...CookieOption)
```

`options` 不是必需的，它们可以用来改变 cookie。稍后您将看到可用的选项，可以根据您的Web应用程序要求添加自定义选项，这也有助于避免在代码库中重复您的内容。

如果你也使用 `SetCookieKV` 方法，这个方法不需要导入 `net/http` 包。

```
SetCookieKV(name, value string, options ...CookieOption)
```

记住：通过 `SetCookieKV` 方法设置的cookie的默认有效期为365天。你可以使用 `CookieExpires` 这个cookie选项设置，或者使用 `kataras/iris/Context.SetCookieKVExpiration` 来全局设置。

`CookieOption` 这是一个 `func(*http.Cookie)` 类型的函数。

设置路径

```
CookiePath(path string) CookieOption
```

设置有效期

```
iris.CookieExpires(durFromNow time.Duration) CookieOption
```

HttpOnly

```
iris.CookieHTTPOnly(httpOnly bool) CookieOption
```

- 对于 `RemoveCookie` 和 `SetCookieKV` 来说 `HttpOnly` 字段默认为 `true`。

编码

当添加cookie时提供了编码功能。

接受一个 `CookieEncoder` 并把cookie的值设置为编码后的值。

`SetCookie` 和 `SetCookieKV` 会使用它。

```
iris.CookieEncode(encode CookieEncoder) CookieOption
```

解码

当获取cookie是提供了解码的功能。

接受一个 `CookieDecoder` ，在通过 `GetCookie` 返回cookie之前把cookie值解码。

`GetCookie` 时使用。

```
iris.CookieDecode(decode CookieDecoder) CookieOption
```

这里的 `CookieEncoder` 可以描述为：一个 CookieEncoder 编码cookie值。

- 接受cookie 的名字作为第一个参数
- 第二个参数为cookie的值的指针
- 返回的第一个值为编码后的值，当编码操作失败是返回空
- 第二个返回值为编码失败时的错误。

```
    CookieEncoder func(cookieName string, value interface{}) (string, error)
```

`CookieDecoder` 应该解码cookie值：

- 第一个参数为Cookie的名字
- 第二个参数是编码值，第三个参数是解码值的指针
- 返回的第一个值为解码值，当发生错误返回空
- 返回的第二个值为错误，当解码发生错误时返回不为空的错误

```
    CookieDecoder func(cookieName string, cookieValue string, v interface{}) error
```

异常不会被打印，因此你必须知道你所做的，记住：如果你使用 AES，它只支持键的大小为16，24或者32bytes。

获取cookie

`GetCookie` 通过cookie的名字返回cookie值，没有找到则返回空。

```
GetCookie(name string, options ...CookieOption) string
```

如果你想要获得除了值之外更多的信息，使用下面的方法：

```
cookie, err := ctx.Request().Cookie("name")
```

获取所有的cookie

`ctx.Request().Cookies()` 方法返回所有可用的请求cookie的切片。有时你想要改变他们，或者为它们每个都执行一个操作，最简单的方法就是通过 `VisitAllCookies` 方法。

```
VisitAllCookies(visitor func(name string, value string))
```

移除一个Cookie

`RemoveCookie` 方法删除对应名字和路径为"/"的 cookie。

Tip：通过 `iris.CookieCleanPath` 选项改变cookie的路径，例如： `RemoveCookie("nname", iris.CookieCleanPath)`

另外，请注意，默认行为是将其HttpOnly设置为true。 它根据网络标准执行cookie的删除。

```
RemoveCookie(name string, options ...CookieOption)
```

Sessions

当你使用一个应用程序，你打开它，做了一些改变，然后关闭它。这就像一个会话。计算机知道你是谁。它知道你什么时候开始这个程序，什么什么时候关闭。但是在互联网上，这是一个问题：web服务器不知道你是谁，也不知道你做什么，因为HTTP不能保持状态。

`Session` 变量通过存储要在多个页面上使用的用户信息（例如用户名，喜欢的颜色等）来解决此问题。默认情况下， `Session` 变量会一直存在，直到浏览器关闭。

因此， `Session` 变量保存有关一个用户的信息，并且可用于一个应用程序中的所有页面。

> Tip：如果你想要持久化存储，你可以存储数据到数据库中。

Iris 在 `iris/sessions` 子包中有自己的会话实现和会话管理。你只需导入这个包就能使用了。

一个会话通过 `Session` 对象的 `Start` 函数开始， `Session` 对象是通过 `New` 函数创建的，这个函数会返回一个 `Sessoin` 。

`Session` 变量通过 `Session.Set` 方法设置，通过 `Session.Get` 方法取回。使用 `Session.Delete` 删除一个变量。要删除整个会话并使之无效，请使用 `Session.Destroy` 方法。

session 管理器通过 `New` 方法创建的。

```
import "github.com/kataras/iris/v12/sessions"
sess := sessions.New(sessions.Config{Cookie: "cookieName", ...})
```

`Config` ：

```
Config struct {
        // session 的 cookie名字，例如："mysessionid"
        // 默认为 "irissessionid"
        Cookie string

        // 如果服务器通过 TLS 运行，CookieSecureTLS 设置为true，
        // 你需要把 session的cookie的 "Secure" 字段设置为 true。
        //
        // 记住：为了正常工作用户应该指定"Decode" 配置字段。
        // 建议：您不需要将此字段设置为true，只需在_examples文件夹中提供
        // example的第三方库（例如secure cookie）填充Encode和Decode字段即可。
        //
        // 默认为 false
        CookieSecureTLS bool
```

```
        // AllowReclaim 将允许在想用的请求处理器中清除然后重新开始一个session
        // 它所做的只是在“Destroy”时删除“Request”和“ ResponseWriter”的cook
ie，
        // 或者在“Start”时将新cookie添加到“请求”。
        //
        // 默认为 false
        AllowReclaim bool

        // 当cookie值不为nil时编码此cookie值(config.Cookie字段指定的值)。
        // 接受cookie的名字为第一个参数，第二个参数为服务器产生的session id。
        // 返回新的session id，如果发生错误，session id将置为空，这是无效的。
        //
        // 提示：错误将不会打印，因此你应该清楚你所做的。
        // 记住：如果你使用 AES，它仅支持key的大小为16,24，或者32 bytes。
        // 你要么提供准确的值或者从你键入的内容中得到key
        //
        // 默认为nil
        Encode func(cookieName string, value interface{}) (string, error)

        // 如果cookie值不为nil则对其解码
        // 第一个参数为cookie的名字(config.Cookie字段指定的值),
        // 第二个参数为客户端的cookie值(也就是被编码后的 session id),
        // 当操作失败时返回错误
        //
        // 提示：错误将不会打印，因此你应该清楚你所做的。
        // 记住：如果你使用 AES，它仅支持key的大小为16,24，或者32 bytes。=
        // 你要么提供准确的值或者从你键入的内容中得到key
        //
        // 默认为nil
        Decode func(cookieName string, cookieValue string, v interface{}) error

        // Defaults to nil.
        // Encoding 功能与 Encode和Decode类似，但是接受一个实例,
        // 这个实例实现了 "CookieEncoder" 接口(Encode和Decode方法)。
        //
        // 默认为nil
        Encoding Encoding

        // 指定cookie必须的生存时间(created_time.Add(Expires)),
        // 如果你想要在浏览器关闭时删除cookie，就设置为 -1。
        //  0 意味着没有过期时间(24年),
        // -1 意味着浏览器关闭时删除
        // >0 就是指定session 的cookie生存期(time.Duration类型)。
        Expires time.Duration
```

```
        // SessionIDGenerator 可以设置一个函数，用于返回一个唯一的session id。
        // 默认将使用uuid包来生成session id，但是开发者可以通过指定这个字段来改
变这个行为。
        SessionIDGenerator func(ctx context.Context) string

        // DisableSubdomainPersistence 设置为true时，
        // 将不允许你的子域名拥有访问session 的cookie的权利
        //
        // 默认为false
        DisableSubdomainPersistence bool
    }
```

`New` 返回一个 `Sessions` 的指针，并拥有这些方法：

```
// 为特定的请求创建或者取回一个已经存在的session。
Start(ctx iris.Context, cookieOptions ...iris.CookieOption)

// Handler 返回一个session中间件，用以注册到应用程序路由中。
Handler(cookieOptions ...iris.CookieOption) iris.Handler

// 移除session数据和相关cookie。
Destroy(ctx context.Context)

// 移除服务器端内存(和已注册的数据库)中的所有session。
// 客户端的session cookie将依然存在，但是它将会被下个请求重新设置
DestroyAll()

// DestroyByID 移除服务器端内存(和已注册的数据库)中的session 条目。
// 客户端的session cookie将依然存在，但是它将会被下个请求重新设置
// 使用这个很安全，即使你不确定这个id对应的session是否存在。
// 提示：sid应该是原始的数据(即从存储中获取)，不是已经解码的。
DestroyByID(sessID string)

// OnDestroy 注册一个或者多个移除监听者。
// 当一个服务端或者客户端(cookie)的session被移除，将会触发监听者。
// 记住如果一个监听者被阻塞，会话管理器将都会延迟，
// 在侦听器中使用goroutine避免这种情况。
OnDestroy(callback func(sid string))

// ShiftExpiration 通过session默认的超时配置，将会话的过期日期更改到新的日期
// 如果使用数据库保存session将会抛出 "ErrNotImplemented" 错误，因为数据库不支持
这个特性。
ShiftExpiration(ctx iris.Context,
         cookieOptions ...iris.CookieOption) error
```

```
// UpdateExpiration 通过 "expires" 这个超时值将session的超时日期改为新的日期。
// 如果更新一个不存在或者无效的session条目，将会返回 "ErrNotFound" 异常。
//  如果使用数据库保存session将会抛出 "ErrNotImplemented" 错误，因为数据库不支持这个特性。
UpdateExpiration(ctx iris.Context, expires time.Duration,
         cookieOptions ...iris.CookieOption) error

// UseDatabase 添加一个session数据库到session管理器中，
// 会话数据库没有写访问权
UseDatabase(db Database)
```

这里的 `CookieOption` 仅仅是一个 `func(*http.Cookie)` 函数，允许自定义配置cookie的属性。

`Start` 方法返回一个 `Session` 指针值，这个指针有自己的方法来为每个session工作。

```
func (ctx iris.Context) {
    session := sess.Start(ctx)

        // 返回session的id
        .ID() string

        // 如果这个session是在当前处理流程中创建，将返回true
        .IsNew() bool

        //  根据会话的键，填充相应的值
         .Set(key string, value interface{})

        // 根据会话的键，填充相应的值。
        // 与 Set 不同，当使用 Get 时，无法改变输出值
        //  一个不可变的条目只能通过 SetImmutable 改变，Set将不能工作
        // 如果条目是不可变的，这将是安全的。
        // 谨慎使用，它比 Set 慢
        .SetImmutable(key string, value interface{})

        // 获取所有值的一个备份
        .GetAll() map[string]interface{}

        // 返回这个session保存的值的总数
        .Len() int

        // 移除key对应的条目，如果有条目移除则返回true
        .Delete(key string) bool
```

```
        // 移除所有的session条目
        .Clear()

        // 返回key对应的值
        .Get(key string) interface{}

        // 与Get类似，但是返回的是值的字符串表示形式。
        // 如果不存在key，返回空字符串。
        .GetString(key string) string

        // 与Get类似，但是返回的是值的字符串表示形式。
        // 如果不存在key，返回defaultValue定义的默认值。
        .GetStringDefault(key string, defaultValue string) string

        // 与Get类似，但是返回的是值的int表示形式。
        // 如果不存在key，返回-1和一个非nil的错误
        .GetInt(key string) (int, error)

        // 与Get类似，但是返回的是值的int表示形式。
        // 如果不存在key，返回defaultValue定义的默认值。
        .GetIntDefault(key string, defaultValue int) int

        // 将保存的key的值+n,
        // 如果key不存在，则设置key的值为n
        // 返回增加后的值
        .Increment(key string, n int) (newValue int)

        // 将保存的key的值-n,
        // 如果key不存在，则设置key的值为n
        // 返回减少后的值
        .Decrement(key string, n int) (newValue int)

        // 与Get类似，但是返回的是值的int64表示形式。
        // 如果不存在key，返回-1和一个非nil的错误
        .GetInt64(key string) (int64, error)

        // 与Get类似，但是返回的是值的int64表示形式。
        // 如果不存在key，返回defaultValue定义的默认值。
        .GetInt64Default(key string, defaultValue int64) int64

        // 与Get类似，但是返回的是值的float32表示形式。
        // 如果不存在key，返回-1和一个非nil的错误
        .GetFloat32(key string) (float32, error)

        // 与Get类似，但是返回的是值的float32表示形式。
        // 如果不存在key，返回defaultValue定义的默认值。
```

```
        .GetFloat32Default(key string, defaultValue float32) float32

        // 与Get类似，但是返回的是值的float64表示形式。
        // 如果不存在key，返回-1和一个非nil的错误
        .GetFloat64(key string) (float64, error)

        // 与Get类似，但是返回的是值的float64表示形式。
        // 如果不存在key，返回defaultValue定义的默认值。
        .GetFloat64Default(key string, defaultValue float64) float64

        // 与Get类似，但是返回的是值的boolean表示形式。
        // 如果不存在key，返回-1和一个非nil的错误
        .GetBoolean(key string) (bool, error)

        // 与Get类似，但是返回的是值的boolean表示形式。
        // 如果不存在key，返回defaultValue定义的默认值。
        .GetBooleanDefault(key string, defaultValue bool) bool

        // 通过key设置一个即时信息。
        // 即时信息是为了保存一个信息到session中，这样同一个用户的多个请求都能获
取到这个信息。
        // 当这个信息展示给用户后就会被移除。
        // 即时信息通常用于与HTTP重定向组合，
        // 因为这种情况下是没有视图的，信息只能在重定向后展示给用户。
        //
        // 一条即时信息拥有它的key和内容。
        // 这是一个有关联的数组。
        // 名字是一个字符串：通常为"notice","sucess","error"，但是可以为任何stri
ng。
        // 内容通常是stirng。如果你想直接显示它，你可以放一些HTML标签在信息中。
        // 也可以放置一个数组或者数组：将会被序列化，然后以字符串类型保存。
        //
        // 即时信息可以是用哪个 SetFlash 方法设置。
        // 例如，你想要通知用户他的改变成功保存了，
        // 你可以在你的处理中添加下面一行：SetFlash("success", "data saved")
        // 在这个例子中我们使用key"success"，如果你想要定义更多即时信息，你可以
使用不同的key。
        .SetFlash(key string, value interface{})

        // 如果这个session有可用的即时信息将返回true
        .HasFlash() bool

        // GetFlashes 返回所有的即时信息的值，使用的是 map[string]interface{} 格
式。
        // 记住：这将导致在同一用户的下一个请求上删除所有当前的即时消息。
```

```
        .GetFlashes() map[string]interface{}
```

```
        // PeekFlash 返回key对应的暂存的即时信息。
        // 与GetFlash不同，这个信息在下个请求时可用，除非使用GetFlashes或者GetFlash
        .PeekFlash(key string) interface{}

        // GetFlash返回key对应的存储的即时信息，并且在下个请求中移除这个即时信息。
        // 检查即时信息我们使用 HashFlash() 方法，获得即时信息我们使用GetFlash()方法。
        // GetFlashes() 获取所有的信息。
        // 获取信息并从session删除，这意味着一条消息只能在提供给用户的第一个页面上显示。
        .GetFlash(key string) interface{}

        // 与GetFlash类似，但是返回的是string表示形式，
        // 如果key不存在，则返回空string
        .GetFlashString(key string) string

        // 与GetFlash类似，但是返回的是string表示形式，
        // 如果key不存在，则返回defaultValue指定的默认值
        .GetFlashStringDefault(key string, defaultValue string) string

        // 删除key这个即时信息
        .DeleteFlash(key string)

        // 移除所有的即时信息
        .ClearFlashes()

        // "摧毁"这个session，它移除session的值和所有即时信息。
        // session条目将会从服务器中移除，注册的session数据库也会被通知删除。
        // 记住这个方法不会移除客户端的cookie，如果附上新的session客户端的cookie将会被重置。
        // 使用会话管理器的 "Destroy(ctx)" 来移除cookie。
        .Destroy()
}
```

示例：

在这里例子中我们将允许通过验证的用户访问 `/secret` 的隐秘信息。为了有访问的权限，我们首先需要访问 `/login` 来获取一个可用的session 的 cookie，使它登录成功。另外也可以访问 `/logout` 撤销访问权限。

```
// sessions.go
package main

import (
    "github.com/kataras/iris/v12"

    "github.com/kataras/iris/v12/sessions"
)

var (
    cookieNameForSessionID = "mycookiesessionnameid"
    sess                   = sessions.New(sessions.Config{Cookie: cookieNameForS
essionID})
)

func secret(ctx iris.Context) {
    // Check if user is authenticated
    if auth, _ := sess.Start(ctx).GetBoolean("authenticated"); !auth {
        ctx.StatusCode(iris.StatusForbidden)
        return
    }

    // Print secret message
    ctx.WriteString("The cake is a lie!")
}

func login(ctx iris.Context) {
    session := sess.Start(ctx)

    // Authentication goes here
    // ...

    // Set user as authenticated
    session.Set("authenticated", true)
}

func logout(ctx iris.Context) {
    session := sess.Start(ctx)

    // Revoke users authentication
    session.Set("authenticated", false)
    // Or to remove the variable:
    session.Delete("authenticated")
    // Or destroy the whole session:
    session.Destroy()
```

```
}

func main() {
    app := iris.New()

    app.Get("/secret", secret)
    app.Get("/login", login)
    app.Get("/logout", logout)

    app.Run(iris.Addr(":8080"))
}
```

访问：

```
$ go run sessions.go

$ curl -s http://localhost:8080/secret
Forbidden

$ curl -s -I http://localhost:8080/login
Set-Cookie: mysessionid=MTQ4NzE5Mz...

$ curl -s --cookie "mysessionid=MTQ4NzE5Mz..." http://localhost:8080/secret
The cake is a lie!
```

Websockets

WebSocket是一种协议，可通过TCP连接启用双向持久通信通道。它用于聊天，股票报价，游戏等应用程序，以及您希望在Web应用程序中具有实时功能的任何位置。

当你需要直接使用套接字连接直接工作时，使用 WebSockets。例如，你可能需要在实时游戏中的最好的性能。

首先，阅读 `kataras/neffos(https://github.com/kataras/neffos/wiki)` 的介绍，掌握这个为 `net/http` 和 `iris` 构建的 websocket 库。

它在安装 iris 时就已经预装了，但是你也可以使用下面的命令单独安装：

```
$ go get github.com/kataras/neffos@latest
```

继续阅读怎么注册neffos websockets 服务器到你的 iris 程序中。

这里可以查看一系列全面的使用websockets的示例：

```
https://github.com/kataras/iris/tree/master/_examples/websocket
```

`iris/websocket` 子包仅包含iris 特定的为 `neffos websockets` 准备的迁移器(migrations，不知咋翻译)和助手。

例如，为了获得请求的 `context` 权限，你可以在处理器或者回调函数中调用 `websocket.GetContext(conn)` 来获取。

```
// GetContext从一个websocket连接中返回一个iris.Context
func GetContext(c *neffos.Conn) Context
```

使用 `websocket.Handler` 注册 websocket `neffos.Server` 到一个路由中。

```
// IDGenerator is an iris-specific IDGenerator for new connections.
type IDGenerator func(Context) string

// Handler returns an Iris handler to be served in a route of an Iris applicatio
n.
// Accepts the neffos websocket server as its first input argument
// and optionally an Iris-specific `IDGenerator` as its second one.
func Handler(s *neffos.Server, IDGenerator ...IDGenerator) Handler
```

使用：

```go
import (
    "github.com/kataras/neffos"
    "github.com/kataras/iris/v12/websocket"
)

// [...]

onChat := func(ns *neffos.NSConn, msg neffos.Message) error {
    ctx := websocket.GetContext(ns.Conn)
    // [...]
    return nil
}

app := iris.New()
ws := neffos.New(websocket.DefaultGorillaUpgrader, neffos.Namespaces{
    "default": neffos.Events {
        "chat": onChat,
    },
})
app.Get("/websocket_endpoint", websocket.Handler(ws))
```

---

websocket 控制器

Iris具有一种通过Go结构注册websocket事件的简单方法。 Websocket控制器是MVC功能的一部分。

Iris 提供了 `iris/mvc/Application.HandleWebsocket(v interface{}) *neffos.Struct` 来注册控制器到一个存在的 iris MVC程序中(提供功能齐全的依赖项注入容器，用于请求值和静态服务)。

```go
// HandleWebsocket handles a websocket specific controller.
// Its exported methods are the events.
// If a "Namespace" field or method exists then namespace is set,
// otherwise empty namespace will be used for this controller.
//
// Note that a websocket controller is registered and ran under
// a connection connected to a namespace
// and it cannot send HTTP responses on that state.
// However all static and dynamic dependencies behave as expected.
func (*mvc.Application) HandleWebsocket(controller interface{}) *neffos.Struct
```

我们来看看使用例子，我们想要通过我们的控制器方法绑定 `OnNamespaceConnected` ， `OnNamespaceDisconnect` 内置的实现和一个自定义的

`OnChat` 事件。

1. 我们创建一个控制器，声明 `NSConn` 类型字段为 `stateless` ，然后写我们需要的方法。

```
type websocketController struct {
        *neffos.NSConn `stateless:"true"`
        Namespace string
        Logger MyLoggerInterface
}

func (c *websocketController) OnNamespaceConnected(msg neffos.Message) error {
    return nil
}

func (c *websocketController) OnNamespaceDisconnect(msg neffos.Message) error {
    return nil
}

func (c *websocketController) OnChat(msg neffos.Message) error {
    return nil
}
```

Iris 足够聪明，通过 `Namespace string` 结构体字段规定的命名空间来注册控制器方法作为事件函数， 或者你可以创建一个控制器方法 `Namespace() string {return "default"}` ，或者使用 `HandleWebsocket` 的返回值的 `SetNamespace("default")` ，这取决于你。

2. 我们将MVC应用程序目标初始化为websocket端点，就像我们以前使用常规HTTP控制器进行HTTP路由一样。

```
```go
import (
        // [...]
        "github.com/kataras/iris/v12/mvc"
)
// [app := iris.New...]
mvcApp := mvc.New(app.Party("/websocket_endpoint"))
```
```

3. 注册我们的依赖项(如果有的话)

```
mvcApp.Register(
        &prefixedLogger{prefix: "DEV"},
)
```

4. 我们注册一个或者多个websocket 控制器，每个控制器匹配一个namespace(只需一个就足够了，因为在大多数情况下，您不需要更多，但这取决于您的应用程序的需求和要求)。

```
mvcApp.HandleWebsocket(&websocketController{Namespace: "default"})
```

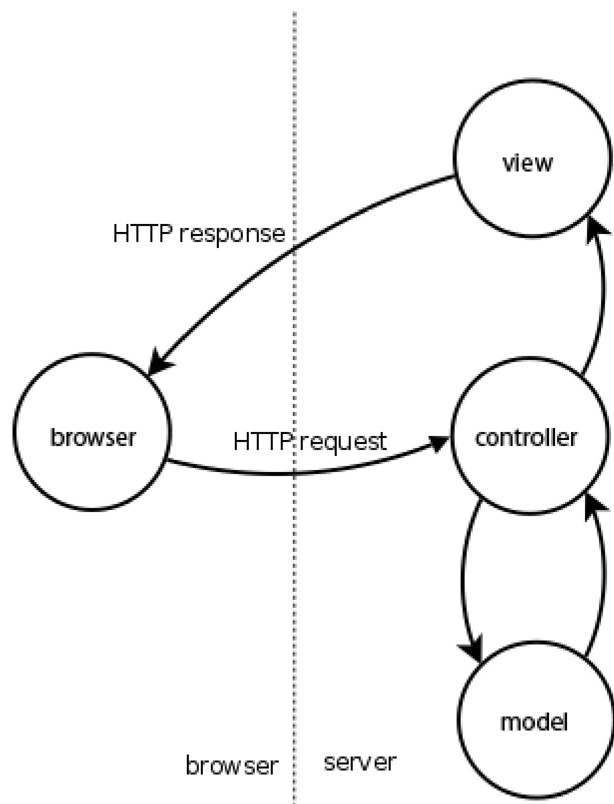
接下来，我们通过将mvc应用程序作为连接处理程序映射到websocket服务器来继续处理(一个websocket服务器可以在多个mvc应用上使用，通过 `neffos.JoinConnHandlers(mvcApp1, mvcApp2)`)。

```
websocketServer := neffos.New(websocket.DefaultGorillaUpgrader, mvcApp)
```

5. 最后一步是通过普通的 `.Get` 方法将该服务器注册到我们的端点。

```
mvcApp.Router.Get("/", websocket.Handler(websocketServer))
```

MVC



使用 iris MVC 来重用代码。

通过创建彼此独立的组件，开发人员可以在其他应用程序中快速轻松地重用组件。一个程序相同或者相似的视图可以被其他应用使用不同的数据重构，因为视图把数据展示给用户的做法都相似。

Iris对MVC（模型视图控制器）架构模式提供了一流的支持，在Go世界中其他任何地方都找不到这些东西。 您将必须导入 `iris/mvc` 子包。

```
import "github.com/kataras/iris/v12/mvc"
```

Iris web 框架支持请求数据，模型，持续性数据和最快的执行速度绑定。

如果您不熟悉后端Web开发，请先阅读有关MVC架构模式的文章，这是一个不错的开始。

特点

支持所有的 HTTP 状态码，例如，如果想要处理 `GET` 方法，控制器需要有一个名为 `Get()` 的函数，你可以在一个控制器中定义多个方法来处理请求。

通过每个控制器的 `BeforeActivation` 自定义事件回调，将自定义控制器的struct方法用作具有自定义路径（甚至带有 `regex` 参数化路径）的处理程序。 例：

```
import (
    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/mvc"
)

func main() {
    app := iris.New()
    mvc.Configure(app.Party("/root"), myMVC)
    app.Run(iris.Addr(":8080"))
}

func myMVC(app *mvc.Application) {
    // app.Register(...)
    // app.Router.Use/UseGlobal/Done(...)
    app.Handle(new(MyController))
}

type MyController struct {}

func (m *MyController) BeforeActivation(b mvc.BeforeActivation) {
    // b.Dependencies().Add/Remove
    // b.Router().Use/UseGlobal/Done
    // and any standard Router API call you already know

    // 1-> Method
    // 2-> Path
    // 3-> The controller's function name to be parsed as handler
    // 4-> Any handlers that should run before the MyCustomHandler
    b.Handle("GET", "/something/{id:long}", "MyCustomHandler", anyMiddleware...)
}

// GET: http://localhost:8080/root
func (m *MyController) Get() string {
    return "Hey"
}

// GET: http://localhost:8080/root/something/{id:long}
func (m *MyController) MyCustomHandler(id int64) string {
    return "MyCustomHandler says Hey"
}
```

通过为依赖项定义服务或者有一个 `Singleton` 控制器作用域， 在你的控制器结构体中持续性数据(在两个请求间分享数据)。

在控制器间分享依赖或者将它们注册到一个父MVC程序中，有能力在每个控制器的 `BeforeActivate` 可选事件回调函数中修改依赖，例如：

```
func(c *MyController) BeforeActivation(b mvc.BeforeActivation) {
        b.Dependencies().Add/Remove(...)
}
```

作为控制器的字段来访问 `Context` (无需手动绑定)，即 `Ctx iris.Context` 或者通过一个方法的输出参数，即 `func(ctx iris.Context, otherArguments)`

控制器结构体内部的模型(在方法函数中设置，并通过视图渲染)。你可以从一个控制器的方法中返回模型，或者在请求的声明周期中设置一个字段，在同一个请求的生命周期中的另一个方法中返回这个字段。

就像你以前使用的流程一样，MVC 程序有自己的 `Router` ，这是 `iris/router.Party` 类型的，标准的 iris api `Controllers` 可以被注册到任何 `Party` 中，包括子域名， `Party` 的开始和完成处理器与预期的一样工作。

可选的 `BeginRequest（ctx）` 函数，用于在方法执行之前执行任何初始化，这对调用中间件或许多方法使用相同的数据收集很有用。

可选的 `EndRequest（ctx）` 函数， 可在执行任何方法之后执行任何终结处理。

递归继承，例如 我们的mvc会话控制器示例具有 `Session * sessions.Session` 作为字段，由会话管理器的 `Start` 填充为MVC应用程序的动态依赖项： `mvcApp.Register(sessions.New(sessions.Config{Cookie:"iris_session_id"}).Start)`

通过控制器方法的输入参数访问动态路径参数，不需要绑定。当你使用 Iris 的默认语法从一个控制器中解析处理器，你需要定义方法的后缀为 `By` ，大写字母是新的子路径。例如：

如果 `mvc.New(app.Party("/user")).Handle(new(user.Controller))` ：

- `func(*Controller) Get() - GET:/user`
- `func(*Controller) Post() - POST:/user`
- `func(*Controller) GetLogin() - GET:/user/login`
- `func(*Controller) PostLogin() - POST:/user/login`
- `func(*Controller) GetProfileFollowers() - GET:/user/profile/followers`
- `func(*Controller) PostProfileFollowers() - POST:/user/profile/followers`
- `func(*Controller) GetBy(id int64) - GET:/user/{param:long}`

- `func(*Controller) PostBy(id int64) - POST:/user/{param:long}`

如果 `mvc.New(app.Party("/profile")).Handle(new(profile.Controller))` :

- `func(*Controller) GetBy(username string) - GET:/profile/{param:string}`

如果 `mvc.New(app.Party("/assets")).Handle(new(file.Controller))` :

- `func(*Controller) GetByWildard(path string) - GET:/assets/{param:path}`

方法函数接受的类型可以为: `int` , `int64` , `bool` 和 `string`

可选的响应输出参数，就像我们前面看到的一样：

```
func(c *ExampleController) Get() string |
                                (string, string) |
                                (string, int) |
                                int |
                                (int, string) |
                                (string, error) |
                                error |
                                (int, error) |
                                (any, bool) |
                                (customStruct, error) |
                                customStruct |
                                (customStruct, int) |
                                (customStruct, string) |
                                mvc.Result or (mvc.Result, error)
```

这里的 `mvc.Result` 是 `hero.Result` 的别名，就是这个接口：

```
type Result interface {
    // Dispatch should sends the response to the context's response writer.
    Dispatch(ctx iris.Context)
}
```

示例

这个例子相当于: `https://github.com/kataras/iris/blob/master/_examples/hello-world/main.go` 。

这看起来像多于的代码，不值得书写，但是记住，这个实例没有使用 Iris MVC 的模型，持续化或者视图引擎等特性，没有使用 `Session` ，这仅仅是为了学习，或许你在你的程序中不会

使用如此简单的控制器。

在这个实例中，在 `/hello` 路径下使用 MVC时，我的个人电脑的消耗是没20MB吞吐大于是2MB，这对于大多数应用程序都可以容忍，但是你可以选择iris中最适合你的，低级处理器的性能或高级控制器：易于维护，大型应用程序上的代码库较小。

仔细阅读注释

```go
package main

import (
    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/mvc"

    "github.com/kataras/iris/v12/middleware/logger"
    "github.com/kataras/iris/v12/middleware/recover"
)

func main() {
    app := iris.New()
    // Optionally, add two built'n handlers
    // that can recover from any http-relative panics
    // and log the requests to the terminal.
    app.Use(recover.New())
    app.Use(logger.New())

    // Serve a controller based on the root Router, "/".
    mvc.New(app).Handle(new(ExampleController))

    // http://localhost:8080
    // http://localhost:8080/ping
    // http://localhost:8080/hello
    // http://localhost:8080/custom_path
    app.Run(iris.Addr(":8080"))
}

// ExampleController serves the "/", "/ping" and "/hello".
type ExampleController struct{}

// Get serves
// Method:   GET
// Resource: http://localhost:8080
func (c *ExampleController) Get() mvc.Result {
    return mvc.Response{
        ContentType: "text/html",
        Text:        "<h1>Welcome</h1>",
```

```
    }
}

// GetPing serves
// Method:   GET
// Resource: http://localhost:8080/ping
func (c *ExampleController) GetPing() string {
    return "pong"
}

// GetHello serves
// Method:   GET
// Resource: http://localhost:8080/hello
func (c *ExampleController) GetHello() interface{} {
    return map[string]string{"message": "Hello Iris!"}
}

// BeforeActivation called once, before the controller adapted to the main appli
cation
// and of course before the server ran.
// After version 9 you can also add custom routes for a specific controller's me
thods.
// Here you can register custom method's handlers
// use the standard router with `ca.Router` to
// do something that you can do without mvc as well,
// and add dependencies that will be binded to
// a controller's fields or method function's input arguments.
func (c *ExampleController) BeforeActivation(b mvc.BeforeActivation) {
    anyMiddlewareHere := func(ctx iris.Context) {
        ctx.Application().Logger().Warnf("Inside /custom_path")
        ctx.Next()
    }

    b.Handle(
        "GET",
        "/custom_path",
        "CustomHandlerWithoutFollowingTheNamingGuide",
        anyMiddlewareHere,
    )

    // or even add a global middleware based on this controller's router,
    // which in this example is the root "/":
    // b.Router().Use(myMiddleware)
}

// CustomHandlerWithoutFollowingTheNamingGuide serves
```

```
// Method:   GET
// Resource: http://localhost:8080/custom_path
func (c *ExampleController) CustomHandlerWithoutFollowingTheNamingGuide() string
{
    return "hello from the custom handler without following the naming guide"
}

// GetUserBy serves
// Method:   GET
// Resource: http://localhost:8080/user/{username:string}
// By is a reserved "keyword" to tell the framework that you're going to
// bind path parameters in the function's input arguments, and it also
// helps to have "Get" and "GetBy" in the same controller.
//
// func (c *ExampleController) GetUserBy(username string) mvc.Result {
//     return mvc.View{
//         Name: "user/username.html",
//         Data: username,
//     }
// }

/* Can use more than one, the factory will make sure
that the correct http methods are being registered for each route
for this controller, uncomment these if you want:

func (c *ExampleController) Post() {}
func (c *ExampleController) Put() {}
func (c *ExampleController) Delete() {}
func (c *ExampleController) Connect() {}
func (c *ExampleController) Head() {}
func (c *ExampleController) Patch() {}
func (c *ExampleController) Options() {}
func (c *ExampleController) Trace() {}
*/

/*
func (c *ExampleController) All() {}
//        OR
func (c *ExampleController) Any() {}

func (c *ExampleController) BeforeActivation(b mvc.BeforeActivation) {
    // 1 -> the HTTP Method
    // 2 -> the route's path
    // 3 -> this controller's method name that should be handler for that route.
    b.Handle("GET", "/mypath/{param}", "DoIt", optionalMiddlewareHere...)
}
```

```
// After activation, all dependencies are set-ed - so read only access on them
// but still possible to add custom controller or simple standard handlers.
func (c *ExampleController) AfterActivation(a mvc.AfterActivation) {}
*/
```

在控制器中每个以HTTP方法(`Get` , `Post` , `Put` , `Delete` ...) 为前缀的函数，都作为一个 HTTP 端点。在上面的示例中，所有的函数都向响应写了一个字符串。注意每种方法之前的注释。

一个HTTP端点在web程序中是可定位的URL，例如 `http://localhost:8080/helloworld` ，结合使用的协议：HTTP，web服务器的网络定位(包括TCP端口)： `localhost:8080` 和 定位的URI： `/helloworld` 。

第一个注释指出这是一个HTTP GET方法，该方法通过在基本URL后面附加 `/helloworld` 来调用。第三条注释指定HTTP GET方法，该方法通过在URL后面附加 `/helloworld/welcome` 来调用。

控制器知道怎么处理 `GetBy` 上的 “name” 或者 `GetWelcomeBy` 上的 “name” 和 “numTimes”，因为 `By` 关键字，并且建立了没有样板的动态路由；第三个注释指定HTTP GET动态方法，该方法可以由任何以“ / helloworld / welcome”开头的URL调用，然后再加上两个路径部分，第一个可以接受任何值，第二个只能接受数字，例如： `http://localhost:8080/helloworld/welcome/golang/32719` ，除此以外， `404 Not Found HTTP Error` 将被发送到客户端。

> `https://github.com/kataras/iris/tree/master/_examples/mvc` 和 `https://github.com/kataras/iris/blob/master/mvc/controller_test.go` 通过简单的范式解释了特性，它们展示了如何利用 Iris MVC 的 Binder、模型等等...

> websocket 控制器请看前面的 `Websocket` 章节。

测试

Iris 为 `httpexpect` (一个web测试框架) 提供了丰富的支持。 `iris/httptest` 子包为 iris + httpexpect 提供了帮助。

如果你更喜欢 golang 的 `net/http/httptest` 标准库，你也可以使用它。Iris 尽可能与其它外部测试工具兼容。

基本认证

在第一个示例中，我们使用 `iris/httptest` 来测试权限验证。

1. `main.go` 文件像这样：

```go
package main

import (
	"github.com/kataras/iris/v12"
	"github.com/kataras/iris/v12/middleware/basicauth"
)

func newApp() *iris.Application {
	app := iris.New()

	authConfig := basicauth.Config{
		Users: map[string]string{"myusername": "mypassword"},
	}

	authentication := basicauth.New(authConfig)

	app.Get("/", func(ctx iris.Context) { ctx.Redirect("/admin") })

	needAuth := app.Party("/admin", authentication)
	{
		//http://localhost:8080/admin
		needAuth.Get("/", h)
		// http://localhost:8080/admin/profile
		needAuth.Get("/profile", h)

		// http://localhost:8080/admin/settings
		needAuth.Get("/settings", h)
	}
```

```
    return app
}

func h(ctx iris.Context) {
    username, password, _ := ctx.Request().BasicAuth()
    // third parameter ^ will be always true because the middleware
    // makes sure for that, otherwise this handler will not be executed.

    ctx.Writef("%s %s:%s", ctx.Path(), username, password)
}

func main() {
    app := newApp()
    app.Run(iris.Addr(":8080"))
}
```

2. 现在，创建一个 `main_test.go` 文件，然后复制下面的代码：

```
package main

import (
    "testing"

    "github.com/kataras/iris/v12/httptest"
)

func TestNewApp(t *testing.T) {
    app := newApp()
    e := httptest.New(t, app)

    // redirects to /admin without basic auth
    e.GET("/").Expect().Status(httptest.StatusUnauthorized)
    // without basic auth
    e.GET("/admin").Expect().Status(httptest.StatusUnauthorized)

    // with valid basic auth
    e.GET("/admin").WithBasicAuth("myusername", "mypassword").Expect().
        Status(httptest.StatusOK).Body().Equal("/admin myusername:mypassword")
    e.GET("/admin/profile").WithBasicAuth("myusername", "mypassword").Expect().
        Status(httptest.StatusOK).Body().Equal("/admin/profile myusername:mypass
word")
    e.GET("/admin/settings").WithBasicAuth("myusername", "mypassword").Expect().
        Status(httptest.StatusOK).Body().Equal("/admin/settings myusername:mypas
sword")
```

```
	// with invalid basic auth
	e.GET("/admin/settings").WithBasicAuth("invalidusername", "invalidpassword"
).
		Expect().Status(httptest.StatusUnauthorized)

}
```

3. 打开你的命令行然后执行：

```
$ go test -v
```

其他示例

```
package main

import (
	"fmt"
	"testing"

	"github.com/kataras/iris/v12/httptest"
)

func TestCookiesBasic(t *testing.T) {
	app := newApp()
	e := httptest.New(t, app, httptest.URL("http://example.com"))

	cookieName, cookieValue := "my_cookie_name", "my_cookie_value"

	// Test Set A Cookie.
	t1 := e.GET(fmt.Sprintf("/cookies/%s/%s", cookieName, cookieValue)).
		Expect().Status(httptest.StatusOK)
	// Validate cookie's existence, it should be available now.
	t1.Cookie(cookieName).Value().Equal(cookieValue)
	t1.Body().Contains(cookieValue)

	path := fmt.Sprintf("/cookies/%s", cookieName)

	// Test Retrieve A Cookie.
	t2 := e.GET(path).Expect().Status(httptest.StatusOK)
	t2.Body().Equal(cookieValue)

	// Test Remove A Cookie.
	t3 := e.DELETE(path).Expect().Status(httptest.StatusOK)
	t3.Body().Contains(cookieName)
```

```
    t4 := e.GET(path).Expect().Status(httptest.StatusOK)
    t4.Cookies().Empty()
    t4.Body().Empty()
}
```

```
$ go test -v -run=TestCookiesBasic$
```