



下载APP



## 09 | 自研or借力：集成Gin替换已有核心(下)

2021-10-01 叶剑峰

《手把手带你写一个Web框架》

课程介绍 &gt;

**讲述：叶剑峰**

时长 16:17 大小 14.93M



你好，我是轩脉刃。

在上一节课，比对了 Gin 框架和我们之前写的框架，明确框架设计的目标是，要设计出真正具有实用性的一个工业级框架，所以我们接下来会基于现有的比较成熟的 Gin 框架，并且好好利用其中间件的生态，站在巨人的肩膀，继续搭建自己的框架。

### 如何借力，讨论开源项目的许可协议

有的人可能就有点困惑了，这样借鉴其他框架或者其他库不是侵权行为吗？




要解答这个问题，我们得先搞清楚站在巨人肩膀是要做什么操作。借鉴和使用 Gin 框架作为我们的底层框架基本思路是，以复制的形式将 Gin 框架引入到我们的自研框架中，替换

和修改之前实现的几个核心模块。

我们后续会在这个以 Gin 为核心的新框架上，进行其余核心或者非核心框架模块的设计和开发，同时我们也需要找到比较好的方式，能将 Gin 生态中丰富的开源中间件进行复制和使用。

现在我们再来回答是否侵权的问题，首先得了解开源许可证，并且知道可以对 Gin 框架做些什么操作？

开源社区有非常多的开源项目，每个项目都需要有许可说明，包含：是否可以引用、是否允许修改、是否允许商用等。目前的开源许可证有非常多种，每个许可证都是一份使用这个开源项目需要遵守的协议，而主流的开源许可证在 OSI 组织（开放源代码促进会）都有  登记。最主流的开源许可证有 6 种：Apache、BSD、GPL、LGPL、MIT、Mozillia。

BSD 许可证、MIT 许可证和 Apache 许可证属于三个比较宽松的许可，都允许对源代码进行修改，且可以在闭源软件中使用，区别在于对新的修改，是否必须使用原先的许可证格式，以及修改后的软件是否能以原软件的名义进行销售等。

我们这里重点讲一下 Gin 框架使用的  MIT 开源许可证，这个许可证内容非常简单，对使用者的要求也最低。


允许被许可人使用、复制、修改、合并、出版发行、散布、再许可、售卖软件及软件副本。

唯一条件是在软件和软件副本中必须包含著作权声明和本许可声明。

所以如果软件用的是 MIT 许可证，不管你开发的项目是开源的，还是商业闭源的，都可以放心使用这个软件，只需要在软件中包含著作权声明和许可协议声明就行，且不要求新的文件必须使用 MIT 协议。

以使用了 MIT 协议的 Gin 框架为例，你可以在新项目中以引用或者复制的形式，使用 Gin 框架，也允许你在复制的 Gin 框架中进行修改，修改可以不用注明 Gin 的版权，但是非修改部分是需要包含版权声明的。

所以，如果我们使用复制形式使用 Gin 框架的话，需要在 Gin 框架每个文件的头部，增加上著作权和许可声明：

 复制代码

```
1 // Copyright 2014 Manu Martinez-Almeida. All rights reserved.  
2 // Use of this source code is governed by a MIT style  
3 // license that can be found in the LICENSE file.
```

## 如何将 Gin 迁移进入我们的框架

明确了 Gin 是允许被复制、修改和合并进入我们的框架，我们就可以开始规划迁移的具体方案了。

首先确定 Gin 的迁移版本，截止到目前（2021 年 8 月 14 日）最新的 Gin 版本为 1.7.3，所以我们将 Gin 的版本确定为 [1.7.3](#)。

在 Golang 中，要在一个项目中引入另外一个项目，一般有两种做法，一种做法是**把要引用的项目通过 go mod 引入到目标库中**，而另外一种做法则费劲的多，**使用复制源码的方式引入要引用的项目**。

go mod 是 Go 官方提供的第三方库管理工具。go mod 的使用方式是在代码方面，也就是在你要使用第三方库功能的时候，使用 import 关键字进行引用。它的好处是简单方便，我们直接使用官方提供的 go get 方法，就能将某个框架进行使用和升级。

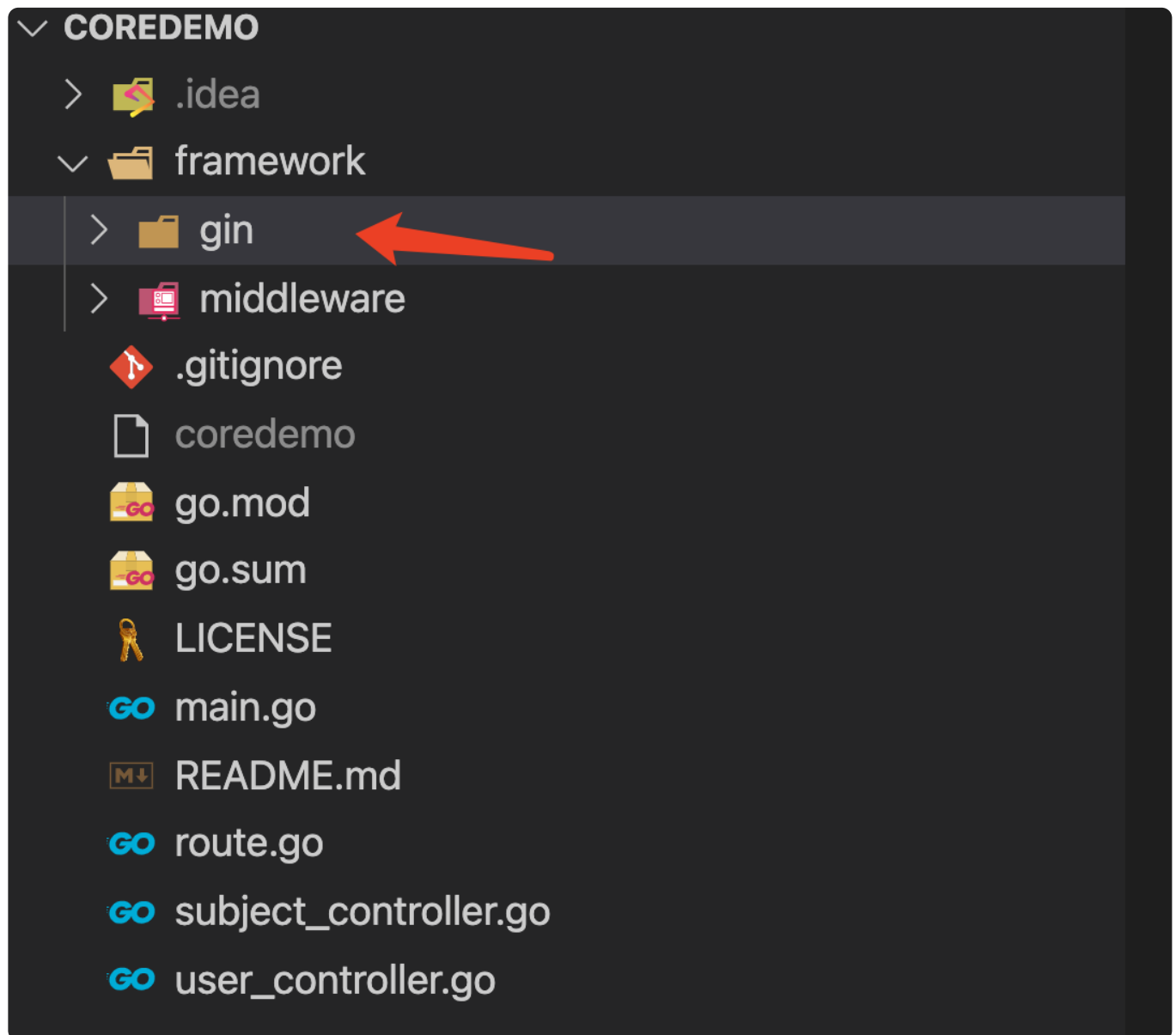
但是如果希望对第三方库进行源码级别的修改，go mod 的方式就会受到限制了。比如我们使用了 Gin 项目，想扩展项目中的 Context 这个结构，为其增加一个函数方法 A，这个需求用 go mod 的方式是做不到的。

go mod 的方式提倡的是“使用第三方库”而不是“定制第三方库”。所以**对于很强的定制第三方库的需求，我们只能选择复制源码的方式**。

有的同学可能会觉得这种源码复制的方式有点奇怪，但是这种复制出来，再进行定制化的方式，其实在一些项目中是常常可以见到的。

比如 Gin 框架在使用 httprouter 的时候，由于要对其进行深度定制化和优化，所以直接将 httprouter 的代码以复制的方式引入到自己的项目中；上一讲简单提过，比较成功的开源项目 [macaron](#)，在项目的最初期也是参考另外一个项目 [martini](#)，将 martini 中的一些源码拷贝后进行优化。我们还能在这两个项目的某些代码文件中看到对第三方库的版权申明。

由于我们对 Gin 框架有深度定制改造的需求，所以接下来也采用源码复制的方式引入 Gin 框架。首先将 Gin1.7.3 的源码完整复制进入项目，存放在 framework 目录中创建一个 gin 目录。




复制之后需要做两步操作：

将 Gin 目录下的 go.mod 的内容放在项目目录下

既然我们以复制源码的方式引入了 Gin，那么 Gin 的地址就成为我们项目中的一部分，它就不是一个有 go.mod 的第三方库了，而 Gin 原本引入的第三方库，成为了我们项目所需要的第三方库。所以第一步需要将 Gin 目录下的 go.mod 和 go.sum 删除，并且将 go.mod 的内容复制到项目的根目录中。

将 Gin 中原有 Gin 库的引用地址，统一替换为当前项目的地址

go.mod 中的 module 代表了当前项目的模块名称，而在项目的 go.mod 中，我们将我们这个框架的模块名称从 “coredemo” 修改为 “github.com/gohade/hade”。在项目的 go.mod 文件：

 复制代码

```
1 module github.com/gohade/hade
2
3 go 1.15
4
5 require (
6     ...
7 )
8
```

其实这个 module 并不一定是 GitHub 上的一个项目地址，也可以自己定义的，我们之前用的 coredemo 就是一个自定义的字符串。但是开源项目的普遍做法是，将模块名定义为你的开源地址名称，这样能让开源项目使用者在导入包的时候，直接根据模块名查找到对应文件。

所有第三方引用在查询时，go.mod 中的当前模块名称为 github.com/gohade/hade，那么复制过来之后，对应的 Gin 框架的引用地址就是 github.com/gohade/hade/framework/gin，而 Gin 框架之前 go.mod 中的模块名称为 github.com/gin-gonic/gin。

所以我们这里**要做一次统一替换**，将之前 Gin 框架中的所有引用 github.com/gin-gonic/gin 的地方替换为 github.com/gohade/hade/framework/gin。

比如：

```
1 "github.com/gin-gonic/gin/binding" 替换为 "github.com/gohade/hade/framework/gin"
2 "github.com/gin-gonic/gin/render"   替换为 "github.com/gohade/hade/framework/gin"
```

 复制代码

这里就要借用 IDE 的强大替换工具了，进行一次批量替换。

做完上述两步的操作之后，我们的项目 `github.com/gohade/hade` 就包含了 Gin 1.3.7 了。下面就是重头戏了，需要思考如何将之前研发的定制化功能迁移到 Gin 上。

## 如何迁移

首先，梳理下目前已经实现的模块：

Context：请求控制器，控制每个请求的超时等逻辑；

路由：让请求更快寻找目标函数，并且支持通配符、分组等方式制定路由规则；

中间件：能将通用逻辑转化为中间件，并串联中间件实现业务逻辑；

封装：提供易用的逻辑，把 request 和 response 封装在 Context 结构中；

重启：实现优雅关闭机制，让服务可以重启。


在 Gin 的框架中，Context、路由、中间件，都已经有了 Gin 自己的实现，而且我们从源码上分析了细节。

Context 方面，Gin 的实现基本和我们之前的实现是一致的。之前实现的 Core 数据结构对应 Gin 中的 Engine，Group 数据结构对应 Gin 的 Group 结构，Context 数据结构对应 Gin 的 Context 数据结构。

路由方面，Gin 的路由实现得比我们要好，这一点上一节课详细分析了 Gin 路由就不再赘述。

中间件方面，Gin 的中间件实现和我们之前的没有什么太大的差别，只有一点，我们定义的 Handler 为：

```
1 type ControllerHandler func(c *Context) error
```

 复制代码



而 Gin 定义的 Handler 为：

[复制代码](#)

```
1 type HandlerFunc func(*Context)
```

可以看到相比 Gin，我们多定义了一个 error 返回值。因为 Gin 的作者认为，中断一个请求返回一个 error 并没有什么用，**他希望中断一个请求的时候直接操作 Response**，比如设置返回状态码、设置返回错误信息，而不希望用 error 来进行返回，所以框架也不会用这个 error 来操作任何的返回信息。这一点我认为 Gin 框架的考量是有道理的，所以我们也沿用这种方式。

而对于 Request 和 Response 的封装，Gin 的实现比较简陋。Gin 对 Request 并没有以接口的方式，将 Request 支持哪些接口展示出来；并且在参数查询的时候，返回类型并不多，比如从 Form 中获取参数的系列方法，Gin 只实现了几个方法：

[复制代码](#)

```
1 PostForm
2 DefaultPostForm
3 GetPostForm
4 PostFormArray
5 GetPostFormArray
6 PostFormMap
7 GetPostFormMap
```

但是在我们定义的 Request 中，我们实现了按照不同类型获取参数的方法。

[复制代码](#)

```
1 // form表单中带的参数
2 DefaultFormInt(key string, def int) (int, bool)
3 DefaultFormInt64(key string, def int64) (int64, bool)
4 DefaultFormFloat64(key string, def float64) (float64, bool)
5 DefaultFormFloat32(key string, def float32) (float32, bool)
6 DefaultFormBool(key string, def bool) (bool, bool)
7 DefaultFormString(key string, def string) (string, bool)
8 DefaultFormStringSlice(key string, def []string) ([]string, bool)
9 DefaultFormFile(key string) (*multipart.FileHeader, error)
10 DefaultForm(key string) interface{}
```

而且在 Response 中，我们的设计是带有链式调用方法的，而 Gin 中没有。

[复制代码](#)

```
1 c.SetOkStatus().Json("ok, UserLoginController: " + foo)
```

这两点在使用者使用框架开发具体业务的时候会非常便利，所以我们将这些功能集成到 Gin 中，迁移这部分 Request 和 Response 的封装。

最后一个优雅关闭的逻辑，我们和 Gin 都是直接使用 HTTP 库的 `server.Shutdown` 实现的，不受 Gin 代码迁移的影响。

所以再明确下，context、路由、中间件、重启机制，我们都不需要迁移，唯一需要迁移的是对 Request 和 Response 的封装。

## 使用加法最小化和定制化我们的需求

对于 request 和 response 的封装，涉及易用性，我们希望能保留自己的定制化需求，同时又不希望影响 Gin 原有的代码逻辑。所以，可以用**加法最小化的方式**迁移这个封装。

为了尽量不侵入原有的文件，我们创建两个新的文件 `hade_request.go`、`hade_response.go` 来存放之前对 request 和 response 的封装。

回顾下第五节课封装的 request，定义的 `IRequest` 接口包含：通过地址 URL 获取参数的 `QueryXXX` 系列接口、通过路由匹配获取参数的 `ParamXXX` 系列接口、通过 Form 表单获取参数的 `FormXXX` 系列接口，以及一些基础接口。

**所以现在的目标是，要让 Gin 框架的 Context 也实现这些接口。**对比之前写的和 Gin 框架原有的实现方法，可以发现，接口存在下列三种情况：

1. Gin 框架中已经存在相同参数、相同返回值、相同接口名的接口
2. Gin 框架中不存在相同接口名的接口
3. Gin 框架中存在相同接口名，但是不同返回值的接口



第一种情况，由于 Gin 框架原先就已经有了相同的接口，所以不需要做任何迁移动作，Gin 的 Context 就已经具备了我们设计的封装。对第二种情况来说，由于 Gin 框架没有对应接口，我们把之前实现的接口原封不动迁移过来即可。

对于第三种情况则棘手一些。以 Gin 中已经有的 QueryXXX 系列接口为例，它的 QueryXXX 系列接口和我们想要的有一定差别，比如它的 Query 不支持多种数据类型的直接获取。怎么办？

**可以选择将 QueryXXX 系列接口重新命名**，又因为 Query 接口都带有一个默认值，所以我们将其重新命名为 DefaultQueryXXX。

经过上述三种情况的修改，IRequest 的定义修改为 (在框架目录的 framework/gin/hade\_request.go 中)：

[复制代码](#)

```
1 // 代表请求包含的方法
2 type IRequest interface {
3
4     // 请求地址url中带的参数
5     // 形如: foo.com?a=1&b=bar&c[]=bar
6     DefaultQueryInt(key string, def int) (int, bool)
7     DefaultQueryInt64(key string, def int64) (int64, bool)
8     DefaultQueryFloat64(key string, def float64) (float64, bool)
9     DefaultQueryFloat32(key string, def float32) (float32, bool)
10    DefaultQueryBool(key string, def bool) (bool, bool)
11    DefaultQueryString(key string, def string) (string, bool)
12    DefaultQueryStringSlice(key string, def []string) ([]string, bool)
13
14    // 路由匹配中带的参数
15    // 形如 /book/:id
16    DefaultParamInt(key string, def int) (int, bool)
17    DefaultParamInt64(key string, def int64) (int64, bool)
18    DefaultParamFloat64(key string, def float64) (float64, bool)
19    DefaultParamFloat32(key string, def float32) (float32, bool)
20    DefaultParamBool(key string, def bool) (bool, bool)
21    DefaultParamString(key string, def string) (string, bool)
22    DefaultParam(key string) interface{}
23
24    // form表单中带的参数
25    DefaultFormInt(key string, def int) (int, bool)
26    DefaultFormInt64(key string, def int64) (int64, bool)
27    DefaultFormFloat64(key string, def float64) (float64, bool)
28    DefaultFormFloat32(key string, def float32) (float32, bool)
29    DefaultFormBool(key string, def bool) (bool, bool)
```

```
30  DefaultFormString(key string, def string) (string, bool)
31  DefaultFormStringSlice(key string, def []string) ([]string, bool)
32  DefaultFormFile(key string) (*multipart.FileHeader, error)
33  DefaultForm(key string) interface{}
34
35  // json body
36  BindJson(obj interface{}) error
37
38  // xml body
39  BindXml(obj interface{}) error
40
41  // 其他格式
42  GetRawData() ([]byte, error)
43
44  // 基础信息
45  Uri() string
46  Method() string
47  Host() string
48  ClientIp() string
49
50  // header
51  Headers() map[string]string
52  Header(key string) (string, bool)
53
54  // cookie
55  Cookies() map[string]string
56  Cookie(key string) (string, bool)
57 }
```

IRequest 的封装就迁移完成了，对于我们封装的 IResponse 结构，也是同样的思路。

和 Gin 的 response 实现对比之后，我们发现由于设计了一个链式调用，很多方法的返回值使用 IResponse 接口本身，所以大部分定义的 IResponse 的接口，在 Gin 中都有同样接口名，但是返回值不同。所以我们可以用同样的方式来修改接口名。

因为大都返回 IResponse 接口，那么可以在所有接口名前面，加一个 I 字母作为区分。在 framework/gin/hade\_response.go 中：

 复制代码

```
1  // IResponse代表返回方法
2  type IResponse interface {
3      // Json输出
4      IJson(obj interface{}) IResponse
5
6      // Jsonp输出
7      IJsonp(obj interface{}) IResponse
```

```
8
9 //xml输出
10 IXml(obj interface{}) IResponse
11
12 // html输出
13 IHtml(template string, obj interface{}) IResponse
14
15 // string
16 IText(format string, values ...interface{}) IResponse
17
18 // 重定向
19 IRedirect(path string) IResponse
20
21 // header
22 ISetHeader(key string, val string) IResponse
23
24 // Cookie
25 ISetCookie(key string, val string, maxAge int, path, domain string, secure,
26
27 // 设置状态码
28 ISetStatus(code int) IResponse
29
30 // 设置200状态
31 ISetOkStatus() IResponse
32 }
```

现在 IRequest 和 IResponse 接口的修改已经完成了。

下面我们就应该迁移每个接口的具体实现。这里接口的实现比较多，就不一一赘述，Request 和 Response 我们分别举其中一个接口例子进行说明，其他的接口迁移可以具体参考 GitHub 仓库的 [geekbang/09 分支](https://github.com/geekbang/geekbang/tree/master/09)。

在 Request 中我们定义了一个 DefaultQueryInt 方法，是 Gin 框架中没有的，怎么迁移这个接口呢？首先将之前定义的 QueryInt 迁移过来，并重新命名为 DefaultQueryInt。

 复制代码

```
1 // 获取请求地址中所有参数
2 func (ctx *Context) QueryAll() map[string][]string {
3     if ctx.request != nil {
4         return map[string][]string(ctx.request.URL.Query())
5     }
6     return map[string][]string{}
7 }
8
9 // 获取Int类型的请求参数
```

```
10 func (ctx *Context) DefaultQueryInt(key string, def int) (int, bool) {
11     params := ctx.QueryAll()
12     if vals, ok := params[key]; ok {
13         if len(vals) > 0 {
14             // 使用cast库将string转换为Int
15             return cast.ToInt(vals[0]), true
16         }
17     }
18     return def, false
19 }
```

然后这里看 QueryAll 函数，其实是可以优化的。Gin 框架在获取 QueryAll 的时候，使用了一个 QueryCache，实现了在第一次获取参数的时候，用方法 initQueryCache 将 ctx.request.URL.Query() 缓存在内存中。

所以既然已经源码引入 Gin 框架了，我们也是可以使用这个方法优化 QueryAll() 方法的，先调用 initQueryCache，再直接从 queryCache 中返回参数：

[复制代码](#)

```
1 // 获取请求地址中所有参数
2 func (ctx *Context) QueryAll() map[string][]string {
3     ctx.initQueryCache()
4     return map[string][]string(ctx.queryCache)
5 }
```

这样 QueryAll 方法和 DefaultQueryInt 方法就都迁移改造完成了。

在 Response 中，我们没有需要优化的点，只要将代码迁移就行。比如原先定义的 Jsonp 方法：

[复制代码](#)

```
1 // Jsonp输出
2 func (ctx *Context) Jsonp(obj interface{}) IResponse {
3     ...
4 }
```

直接修改函数名为 IJsonp 即可：

```
1 // Jsonp输出
2 func (ctx *Context) IJsonp(obj interface{}) IResponse {
3     ...
4 }
```

[复制代码](#)

接口和实现都修改了，最后肯定还要对应修改下业务代码中之前定义的一些控制器。

第一个是控制器的参数，从 `framework.Context` 修改为 `gin.Context`，这里的 `gin` 是引用 `github.com/gohade/hade/framework/gin`，还有把之前定义的 `Handler` 的 `error` 返回值删除。

第二个是修改里面的调用，因为现在的 `Response` 方法都带上了一个前缀 `I`。比如在业务目录下 `subject_controller.go` 中，把原先的 `SubjectListController`：

```
1 func SubjectListController(c *framework.Context) error {
2     c.SetOkStatus().Json("ok, SubjectListController")
3     return nil
4 }
```

[复制代码](#)

修改为：

```
1 func SubjectListController(c *gin.Context) {
2     c.ISetOkStatus().IJson("ok, SubjectListController")
3 }
```

[复制代码](#)

## 验证

所有修改完成之后，我们可以通过 `test` 来进行验证，调用 `go test ./...` 来运行 `Gin` 程序的所有测试用例，显示成功则表示我们的迁移成功。

```
~/Documents/UGit/coredemo  geekbang/09  go test ./...
?      github.com/gohade/hade [no test files]
ok     github.com/gohade/hade/framework/gin 0.967s
ok     github.com/gohade/hade/framework/gin/binding (cached)
?      github.com/gohade/hade/framework/gin/ginS [no test files]
ok     github.com/gohade/hade/framework/gin/internal/bytesconv (cached)
?      github.com/gohade/hade/framework/gin/internal/json [no test files]
ok     github.com/gohade/hade/framework/gin/render (cached)
?      github.com/gohade/hade/framework/middleware [no test files]
```

并且我们通过 `go build && ./hade` 可以看到熟悉的 gin 调试模式的输出：

```
~/Documents/UGit/coredemo  geekbang/09  go build && ./hade
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
- using env:   export GIN_MODE=release
- using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /user/login          --> main.UserLoginController (4 handlers)
[GIN-debug] DELETE /subject/:id         --> main.SubjectDelController (4 handlers)
[GIN-debug] PUT    /subject/:id         --> main.SubjectUpdateController (4 handlers)
[GIN-debug] GET    /subject/:id         --> main.SubjectGetController (5 handlers)
[GIN-debug] GET    /subject/list/all    --> main.SubjectListController (4 handlers)
[GIN-debug] GET    /subject/info/name   --> main.SubjectNameController (4 handlers)
```

## 小结

今天我们讨论几个开源项目的许可协议，比如 Gin 框架使用的 MIT 协议，在明确修改权限后，我们将 Gin 框架迁移进自己手写的 hade 框架，替换前面开发的 Context、路由等模块，为后续拓展做好准备。

在迁移的过程中，我们选择使用复制源码的方式进行替换，并且用了加法最小化的方法，尽量保留了我们的定制化接口。可能有的同学会觉得这种方式比较暴力，但是后续随着我们对框架的改造需求不断增加，这种方式会越来越体现出其优势。

## 思考题

我们的 hade 框架也希望用 MIT 协议进行开源，你知道如何改造来将它开源么？

欢迎在留言区分享你的思考。感谢你的收听，如果你觉得有收获，也欢迎你把今天的内容分享给你身边的朋友，邀他一起学习。我们下节课见。



分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 自研or借力，集成Gin替换已有核心（上）

下一篇 加餐 | 国庆特别放送：什么是业务架构，什么是基础架构？

## 精选留言 (2)

 写留言



鸭补一生如梦

2021-10-06

现在定制的是 gin 的 1.7.3 版本，那么后续 gin 升级了，尤其是频繁升级，我们如何快速及时的进行升级更新？  
或者说隔一段时间再更新？



广训

2021-10-01

DefaultQueryXXX 是不是没有实现 形如: foo.com?a=1&b=bar&c[]=bar 中 c 的获取？  
我看 默认就支持 c=1&c=2&c=3 能获取到 c 的 slice [1, 2, 3]  
但是 c[]=1&c[]=2&c[]=3 是获取不到的，除了自己解析不知道有没有更正常一点的方式  
展开 ∨

