

03 | 路由：如何让请求更快寻找到目标函数？

2021-09-17 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >



讲述：叶剑峰
时长 22:15 大小 20.38M

▶

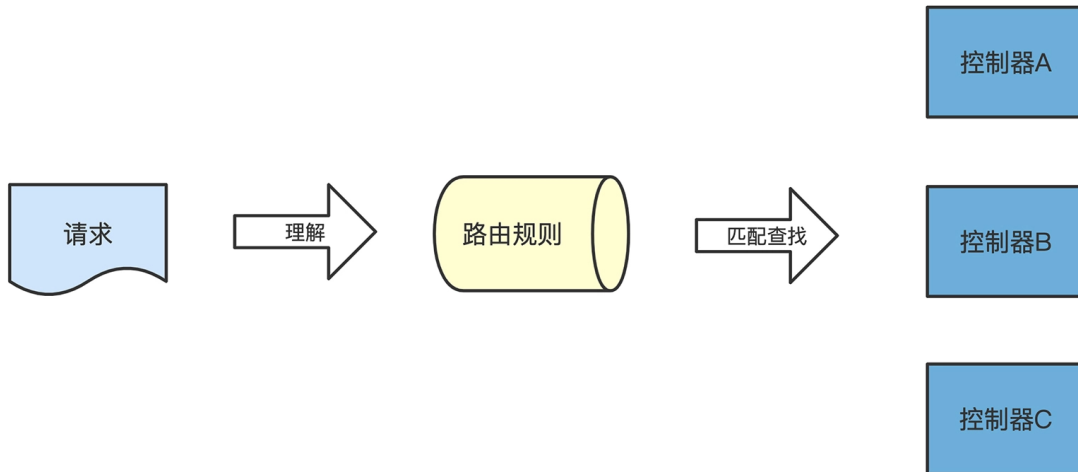
你好，我是轩脉刃。

上一讲，我们封装了框架的 Context，将请求结构 request 和返回结构 responseWriter 都封装在 Context 中。利用这个 Context，我们将控制器简化为带有一个参数的函数 FooControllerHandler，这个控制器函数的输入和输出都是固定的。在框架层面，我们也定义了对应关于控制器的方法结构 ControllerHandler 来代表这类控制器的函数。

每一个请求逻辑，都有一个控制器 ControllerHandler 与之对应。那么一个请求，如何找到指定的控制器呢？这就是今天要研究的内容：路由，我将带你理解路由，并且实现一个高效、易用的路由模块。

路由设计思路

相信你对路由是干啥的已经有大致了解，具体来说就是让 Web 服务器根据规则，理解 HTTP 请求中的信息，匹配查找出对应的控制器，再将请求传递给控制器执行业务逻辑，简单来说就是制定匹配规则。



但是就是这么简单的功能，**路由的设计感不同，可用性有天壤之别**。为什么这么说呢，我们带着这个问题，先来梳理一下制定路由规则需要的信息。

路由可以使用 HTTP 请求体中的哪些信息，得回顾我们第一节课讲 HTTP 的内容。

一个 HTTP 请求包含请求头和请求体。请求体内一般存放的是请求的业务数据，是基于具体控制业务需要的，所以，我们不会用来做路由。

而请求头中存放的是和请求状态有关的信息，比如 User-Agent 代表的是请求的浏览器信息，Accept 代表的是支持返回的文本类型。以下是一个标准请求头的示例：

📋 复制代码

```
1 GET /home.html HTTP/1.1
2 Host: developer.mozilla.org
3 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/201001
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Referer: https://developer.mozilla.org/testpage.html
```

每一行的信息和含义都是非常大的课题，也与今天要讲的内容无关，我们这里要关注的是 HTTP 请求的第一行，叫做 Request Line，由三个部分组成：**Method**、**Request-URI** 和 **HTTP-Version** ([RFC2616](#))。

HTTP 第一行 Request Line: GET /home.html HTTP/1.1

Method Request-URI HTTP-Version



Method 是 HTTP 的方法，标识对服务端资源的操作属性。它包含多个方法，每个方法都代表不同的操作属性。

```
1      Method      = "OPTIONS"           ; Section 9.2
2                  | "GET"               ; Section 9.3
3                  | "HEAD"              ; Section 9.4
4                  | "POST"              ; Section 9.5
5                  | "PUT"               ; Section 9.6
6                  | "DELETE"            ; Section 9.7
7                  | "TRACE"             ; Section 9.8
8                  | "CONNECT"           ; Section 9.9
9                  | extension-method
10     extension-method = token
```

[复制代码](#)

Request-URI 是请求路径，也就是浏览器请求地址中域名外的剩余部分。

<http://time.geekbang.org/serv/v3/product/infos?article=1>

Request-URI



HTTP-Version 是 HTTP 的协议版本，目前常见的有 1.0、1.1、2.0。

Web Service 在路由中使用的就是 Method 和 Request-URI 这两个部分。了解制定路由规则时，请求体中可以使用的元素之后，我们再回答刚才的问题，什么是路由的设计感。

这里说的设计感指的是：**框架设计者希望使用者如何用路由模块。**

如果框架支持 REST 风格的路由设计，那么使用者在写业务代码的时候，就倾向于设计 REST 风格的接口；如果框架支持前缀匹配，那么使用者在定制 URI 的时候，也会倾向于把同类型的 URI 归为一类。

这些设计想法通通会**体现在框架的路由规则上，最终影响框架使用者的研发习惯**，这个就是设计感。所以其实，设计感和框架设计者偏好的研发风格直接相关，也没有绝对的优劣。

这里你很容易走入误区，我要说明一下。很多同学认为设计感的好坏体现在路由规则的多少上，其实不是。

路由规则，是根据路由来查找控制器的逻辑，它本身就是一个框架需求。我们可以天马行空设想 100 条路由规则，并且全部实现它，也可以只设计 1、2 个最简单的路由规则。很

多或者很少的路由规则，都不会根本性影响使用者，所以，并不是衡量一个框架好坏的标准。

路由规则的需求

回到我们的框架，开头我们说过希望使用者高效、易用地使用路由模块，那出于这一点考虑，基本需求可以有哪些呢？

按照从简单到复杂排序，路由需求我整理成下面四点：

需求 1：HTTP 方法匹配

早期的 WebService 比较简单，HTTP 请求体中的 Request Line 或许只会使用到 Request-URI 部分，但是随着 REST 风格 WebService 的流行，为了让 URI 更具可读性，在现在的路由输入中，HTTP Method 也是很重要的一部分了，所以，我们框架也需要支持多种 HTTP Method，比如 GET、POST、PUT、DELETE。

需求 2：静态路由匹配

静态路由匹配是一个路由的基本功能，指的是路由规则中没有可变参数，即路由规则地址是固定的，与 Request-URI 完全匹配。

我们在第一讲中提到的 DefaultServerMux 这个路由器，从内部的 map 中直接根据 key 寻找 value，这种查找路由的方式就是静态路由匹配。

需求 3：批量通用前缀

因为业务模块的划分，我们会同时为某个业务模块注册一批路由，所以在路由注册过程中，为了路由的可读性，一般习惯统一定义这批路由的通用前缀。比如 /user/info、/user/login 都是以 /user 开头，很方便使用者了解页面所属模块。

所以如果路由有能力统一定义批量的通用前缀，那么在注册路由的过程中，会带来很大的便利。

需求 4：动态路由匹配

这个需求是针对需求 2 改进的，因为 URL 中某个字段或者某些字段并不是固定的，是按照一定规则（比如是数字）变化的。那么，我们希望路由也能够支持这个规则，将这个动态变化的路由 URL 匹配出来。所以我们需要，使用自己定义的路由来补充，只支持静态匹配的 DefaultServerMux 默认路由。

现在四个最基本的需求我们已经整理出来了，接下来通过一个例子来解释下，比如我们需要能够支持一个日志网站的这些功能：

功能	Request-URI	HTTP Method
用户登录	/user/login	POST
增加专题	/subject/add	POST
删除专题	/subject/1	DELETE
修改专题	/subject/1	PUT
查找专题	/subject/1	GET
获取专题列表	/subject/list	GET



接下来就是今天的重头戏了，要匹配这样的路由列表，路由规则定义代码怎么写呢？我把最终的使用代码贴在这里，你可以先看看，然后我们一步步实现，分析清楚每行代码背后的方法如何定义、为什么要这么定义。

[复制代码](#)

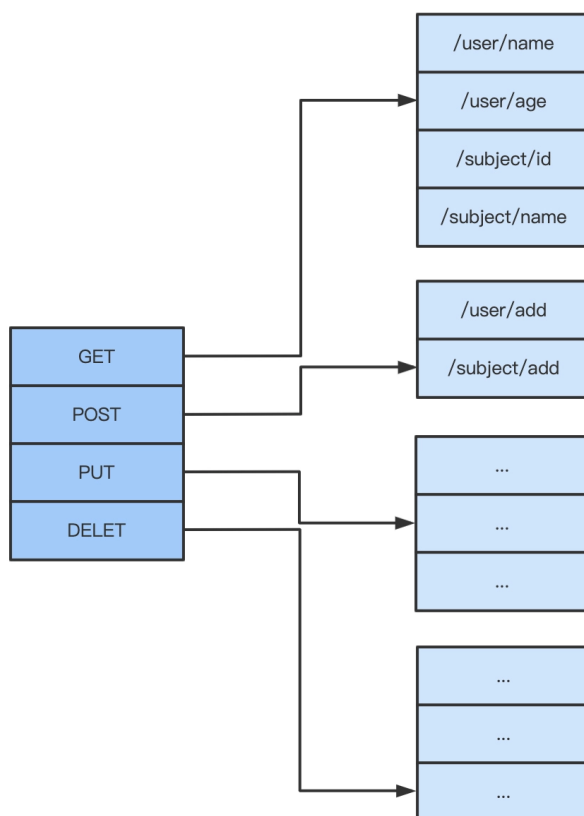
```
1 // 注册路由规则
2 func registerRouter(core *framework.Core) {
3     // 需求1+2:HTTP方法+静态路由匹配
4     core.Post("/user/login", UserLoginController)
5
6     // 需求3:批量通用前缀
7     subjectApi := core.Group("/subject")
8     {
9         subjectApi.Post("/add", SubjectAddController)
10        // 需求4:动态路由
11        subjectApi.Delete("/:id", SubjectDelController)
12        subjectApi.Put("/:id", SubjectUpdateController)
```

```
13     subjectApi.Get("/:id", SubjectGetController)
14     subjectApi.Get("/list/all", SubjectListController)
15 }
16 }
```

（这段代码会在最后补充到上节课中创建的业务目录中的路由文件 router.go。）

实现 HTTP 方法和静态路由匹配

我们首先看第一个需求和第二个需求。由于有两个待匹配的规则，Request-URI 和 Method，所以自然联想到可以使用两级哈希表来创建映射。



第一级 hash 是请求 Method，第二级 hash 是 Request-URI。

这个路由 map 我们会存放在第一讲定义的 Core 结构里（如下），并且在初始化 Core 结构的时候，初始化第一层 map。所以还是拉出 [03 分支](#)，来更新框架文件夹中的 core.go 文件：

```
1 // 框架核心结构
2
```

复制代码

```
3 type Core struct {
4 }
5
6 // 初始化框架核心结构
7 func NewCore() *Core {
8     return &Core{}
9 }
10
11 // 框架核心结构实现Handler接口
12 func (c *Core) ServeHTTP(response http.ResponseWriter, request *http.Request)
13     // TODO
14 }
```


接下来我们按框架使用者使用路由的顺序分成四步来完善这个结构：**定义路由 map、注册路由、匹配路由、填充 ServeHTTP 方法。**

首先，第一层 map 的每个 key 值都代表 Method，而且为了避免之后在匹配的时候，要转换一次大小写，我们将每个 key 都设置为大写。继续在框架文件夹中的 core.go 文件里写：

[复制代码](#)

```
1 // 框架核心结构
2 type Core struct {
3     router map[string]map[string]ControllerHandler // 二级map
4 }
5
6 // 初始化框架核心结构
7 func NewCore() *Core {
8     // 定义二级map
9     getRouter := map[string]ControllerHandler{}
10    postRouter := map[string]ControllerHandler{}
11    putRouter := map[string]ControllerHandler{}
12    deleteRouter := map[string]ControllerHandler{}
13
14    // 将二级map写入一级map
15    router := map[string]map[string]ControllerHandler{}
16    router["GET"] = getRouter
17    router["POST"] = postRouter
18    router["PUT"] = putRouter
19    router["DELETE"] = deleteRouter
20
21    return &Core{router: router}
22 }
```



下一步就是路由注册，我们将路由注册函数按照 Method 名拆分为 4 个方法：Get、Post、Put 和 Delete。

 复制代码

```
1 // 对应 Method = Get
2 func (c *Core) Get(url string, handler ControllerHandler) {
3     upperUrl := strings.ToUpper(url)
4     c.router["GET"][upperUrl] = handler
5 }
6
7 // 对应 Method = POST
8 func (c *Core) Post(url string, handler ControllerHandler) {
9     upperUrl := strings.ToUpper(url)
10    c.router["POST"][upperUrl] = handler
11 }
12
13 // 对应 Method = PUT
14 func (c *Core) Put(url string, handler ControllerHandler) {
15     upperUrl := strings.ToUpper(url)
16     c.router["PUT"][upperUrl] = handler
17 }
18
19 // 对应 Method = DELETE
20 func (c *Core) Delete(url string, handler ControllerHandler) {
21     upperUrl := strings.ToUpper(url)
22     c.router["DELETE"][upperUrl] = handler
23 }
```

我们这里将 URL 全部转换为大写了，在后续匹配路由的时候，也要记得把匹配的 URL 进行大写转换，这样我们的路由就会是“大小写不敏感”的，对使用者的容错性就大大增加了。

注册完路由之后，如何匹配路由就是我们第三步需要做的事情了。首先我们实现匹配路由方法，这个匹配路由的逻辑我用注释写在代码中了。继续在框架文件夹中的 core.go 文件里写入：

 复制代码

```
1 // 匹配路由，如果没有匹配到，返回nil
2 func (c *Core) FindRouteByRequest(request *http.Request) ControllerHandler {
3     // uri 和 method 全部转换为大写，保证大小写不敏感
4     uri := request.URL.Path
5     method := request.Method
6     upperMethod := strings.ToUpper(method)
```

```
7  upperUri := strings.ToUpper(uri)
8
9  // 查找第一层map
10 if methodHandlers, ok := c.router[upperMethod]; ok {
11     // 查找第二层map
12     if handler, ok := methodHandlers[upperUri]; ok {
13         return handler
14     }
15 }
16 return nil
17 }
```

代码很容易看懂，匹配逻辑就是去二层哈希 map 中一层层匹配，先查找第一层匹配 Method，再查第二层匹配 Request-URI。

最后，我们就可以填充未实现的 ServeHTTP 方法了，所有请求都会进到这个函数中处理。（如果你有点模糊了，可以拿出第一节课中的思维导图，再巩固下 net/http 的核心逻辑。）继续在框架文件夹中的 core.go 文件里写：

[复制代码](#)

```
1 func (c *Core) ServeHTTP(response http.ResponseWriter, request *http.Request)
2
3 // 封装自定义context
4 ctx := NewContext(request, response)
5
6 // 寻找路由
7 router := c.FindRouteByRequest(request)
8 if router == nil {
9     // 如果没有找到，这里打印日志
10    ctx.Json(404, "not found")
11    return
12 }
13
14 // 调用路由函数，如果返回err 代表存在内部错误，返回500状态码
15 if err := router(ctx); err != nil {
16    ctx.Json(500, "inner error")
17    return
18 }
19 }
```


这个函数就把我们前面三讲的内容都串起来了。先封装第二讲创建的自定义 Context，然后使用 FindRouteByRequest 函数寻找我们需要的路由，如果没有找到路由，返回 404

状态码；如果找到了路由，就调用路由控制器，另外如果路由控制器出现内部错误，返回 500 状态码。

到这里，第一个和第二个需求就都完成了。

实现批量通用前缀

对于第三个需求，我们可以通过一个 Group 方法归拢路由前缀地址。修正在业务文件夹下的 route.go 文件，使用方法改成这样：

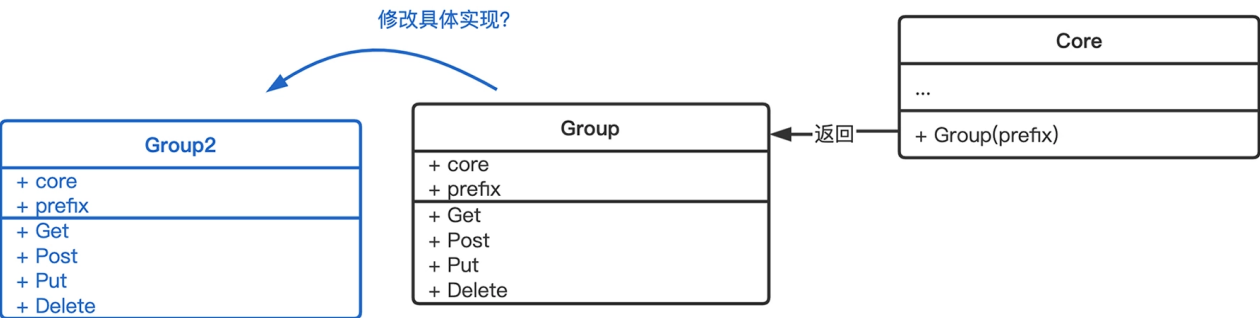
 复制代码

```
1 // 注册路由规则
2 func registerRouter(core *framework.Core) {
3     // 需求1+2:HTTP方法+静态路由匹配
4     core.Get("/user/login", UserLoginController)
5
6     // 需求3:批量通用前缀
7     subjectApi := core.Group("/subject")
8     {
9         subjectApi.Get("/list", SubjectListController)
10    }
11 }
```

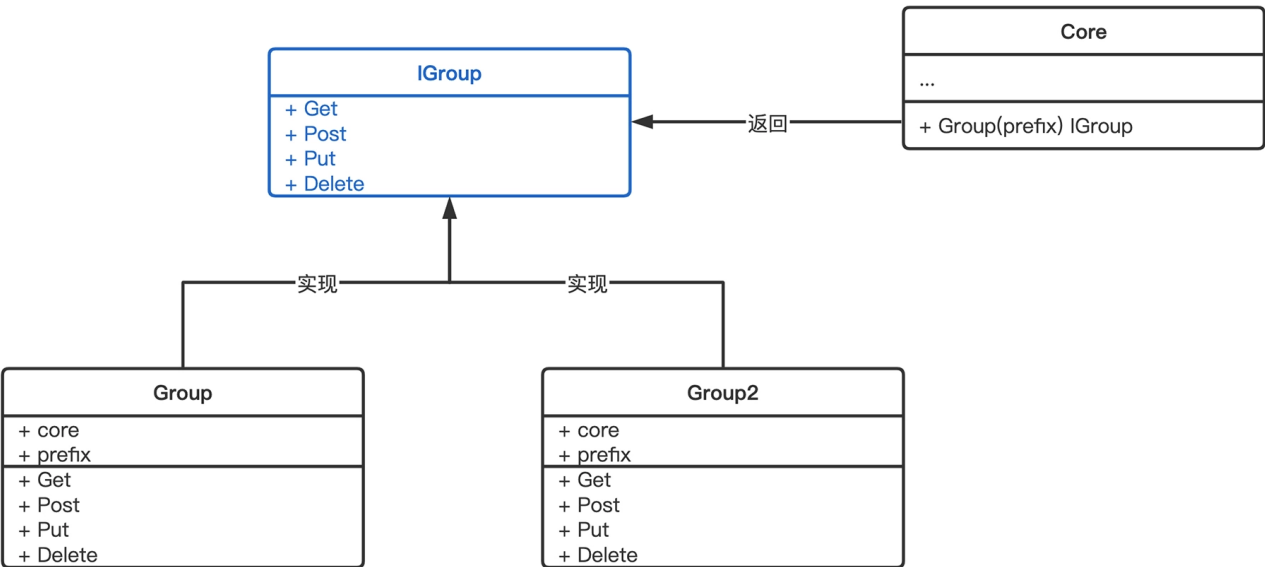
看下这个 Group 方法，它的参数是一个前缀字符串，返回值应该是包含 Get、Post、Put、Delete 方法的一个结构，我们给这个结构命名 Group，在其中实现各种方法。

在这里我们暂停一下，看看有没有优化点。

这么设计直接返回 Group 结构，确实可以实现功能，但试想一下，随着框架发展，如果我们发现 Group 结构的具体实现并不符合我们的要求了，需要引入实现另一个 Group2 结构，该怎么办？直接修改 Group 结构的具体实现么？



其实更好的办法是使用接口来替代结构定义。在框架设计之初，我们要保证框架使用者，在最少改动中，就能流畅迁移到 Group2，这个时候，如果返回接口 IGroup，而不是直接返回 Group 结构，就不需要修改 core.Group 的定义了，只需要修改 core.Group 的具体实现，返回 Group2 就可以。



尽量使用接口来解耦合，是一种比较好的设计思路。

怎么实现呢，这里我们定义 IGroup 接口来作为 Group 方法的返回值。在框架文件夹下创建 [📄 group.go](#) 文件来存放分组相关的信息：

[📄 复制代码](#)

```
1 // IGroup 代表前缀分组
2 type IGroup interface {
3     Get(string, ControllerHandler)
4     Post(string, ControllerHandler)
5     Put(string, ControllerHandler)
6     Delete(string, ControllerHandler)
7 }
```

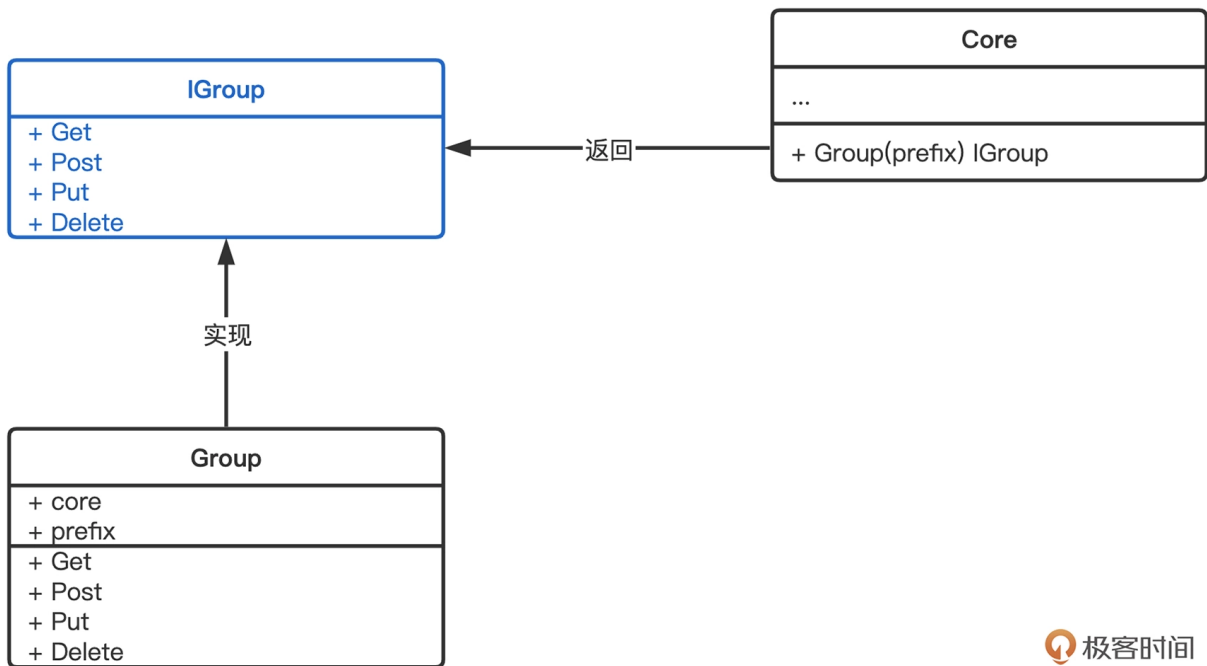
并且继续搭好 Group 结构代码来实现这个接口：

[📄 复制代码](#)

```
1 // Group struct 实现了IGroup
2 type Group struct {
3     core *Core
4     prefix string
5 }
6
7 // 初始化Group
8 func NewGroup(core *Core, prefix string) *Group {
9     return &Group{
10         core: core,
11         prefix: prefix,
12     }
13 }
14
15 // 实现Get方法
16 func (g *Group) Get(uri string, handler ControllerHandler) {
17     uri = g.prefix + uri
18     g.core.Get(uri, handler)
19 }
20
21 ....
22
23 // 从core中初始化这个Group
24 func (c *Core) Group(prefix string) IGroup {
25     return NewGroup(c, prefix)
26 }
```

这个 Group 结构包含自身的前缀地址和 Core 结构的指针。它的 Get、Put、Post、Delete 方法就是把这个 Group 结构的前缀地址和目标地址组合起来，作为 Core 的

Request-URI 地址。



极客时间

讲到这里，有的同学可能不以为然，觉得这不就是个人代码风格的问题吗。其实并不是，希望你能够意识到，这个选择并不仅仅是代码风格，而是关于框架设计、关于代码扩展性。


接口是一种协议，它忽略具体的实现，定义的是两个逻辑结构的交互，因为两个函数之间定义的是一种约定，不依赖具体的实现。

你可以这么判断：**如果你觉得这个模块是完整的，而且后续希望有扩展的可能性，那么就应该尽量使用接口来替代实现。**在代码中，多大程度使用接口进行逻辑结构的交互，是评价框架代码可扩展性的一个很好的标准。这种思维会贯穿在我们整个框架的设计中，后续我会时不时再提起的。

所以回到我们的路由，使用 IGroup 接口后，core.Group 这个方法返回的是一个约定，而不依赖具体的 Group 实现。

实现动态路由匹配

现在已经完成了前三个需求，下面我们考虑第四个需求，希望在写业务的时候能支持像下列这种动态路由：

 复制代码

```
1 func registerRouter(core *framework.Core) {
2     // 需求1+2:HTTP方法+静态路由匹配
3     core.Get("/user/login", UserLoginController)
4
5     // 需求3:批量通用前缀
6     subjectApi := core.Group("/subject")
7     {
8         // 需求4:动态路由
9         subjectApi.Delete("/:id", SubjectDelController)
10        subjectApi.Put("/:id", SubjectUpdateController)
11        subjectApi.Get("/:id", SubjectGetController)
12        subjectApi.Get("/list/all", SubjectListController)
13    }
14 }
```


如何实现？我们继续看。

首先，你要知道的是，**一旦引入了动态路由匹配的规则，之前使用的哈希规则就无法使用了**。因为有通配符，在匹配 Request-URI 的时候，请求 URI 的某个字符或者某些字符是动态变化的，无法使用 URI 做为 key 来匹配。那么，我们就需要其他的算法来支持路由匹配。

如果你对算法比较熟悉，会联想到**这个问题本质是一个字符串匹配**，而字符串匹配，比较通用的高效方法就是字典树，也叫 trie 树。

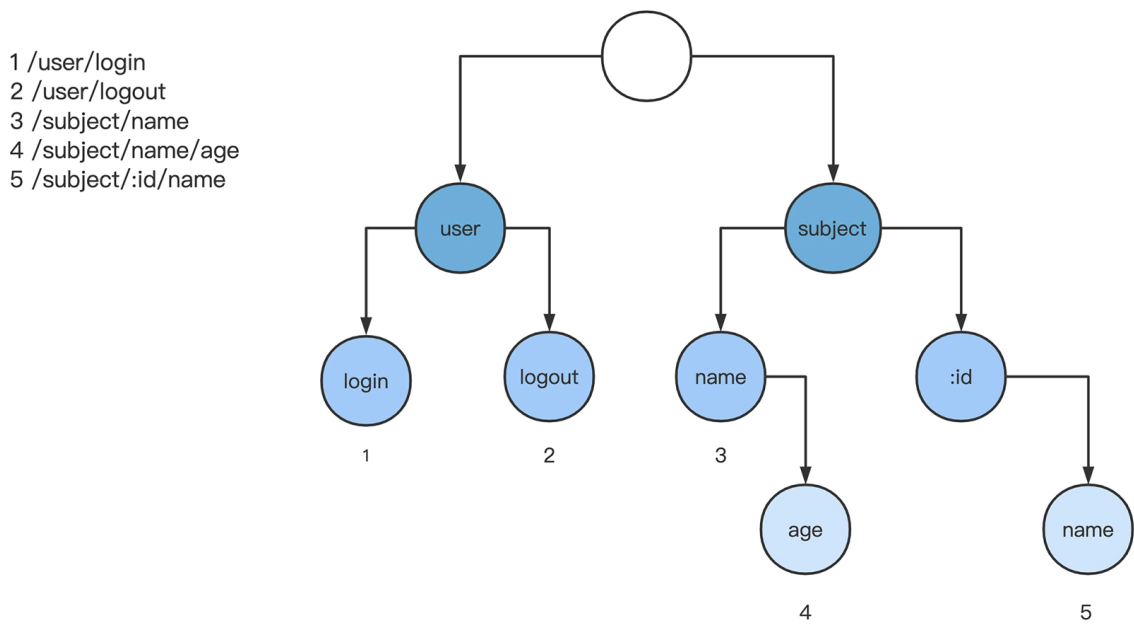
这里，我们先简单梳理下 trie 树的数据结构。trie 树不同于二叉树，它是多叉的树形结构，根节点一般是空字符串，而叶子节点保存的通常是字符串，一个节点的所有子孙节点都有相同的字符串前缀。

所以根据 trie 树的特性，我们结合前三条路由规则，可以构建出这样的结构：

 复制代码

```
1 1 /user/login
2 2 /user/logout
3 3 /subject/name
4 4 /subject/name/age
5 5 /subject/:id/name
```

画成图更清晰一些：



极客时间

这个 trie 树是按照路由地址的每个段 (segment) 来切分的，每个 segment 在 trie 树中都能找到对应节点，每个节点保存一个 segment。树中，每个叶子节点都代表一个 URI，对于中间节点来说，有的中间节点代表一个 URI（比如上图中的 /subject/name），而有的中间节点并不是一个 URI（因为没有路由规则对应这个 URI）。

现在分析清楚了，我们开始动手实现 trie 树。还是照旧先明确下可以分为几步：

1. 定义树和节点的数据结构
2. 编写函数：“增加路由规则”
3. 编写函数：“查找路由”
4. 将“增加路由规则”和“查找路由”添加到框架中

步骤非常清晰，好，废话不多说，我们一步一步来，首先定义对应的数据结构（node 和 tree）。先在框架文件夹下创建 tree.go 文件，存储 trie 树相关逻辑：

复制代码

```
1 // 代表树结构
2 type Tree struct {
3     root *node // 根节点
```



```
4 }
5
6 // 代表节点
7 type node struct {
8     isLast bool           // 代表这个节点是否可以成为最终的路由规则。该节点是否能成为
9     segment string        // uri中的字符串，代表这个节点表示的路由中某个段的字符串
10    handler ControllerHandler // 代表这个节点中包含的控制器，用于最终加载调用
11    childs []*node          // 代表这个节点下的子节点
12 }
```

Tree 结构中包含一个根节点，只是这个根节点是一个没有 segment 的空的根节点。

node 的结构定义了四个字段。childs 字段让 node 组成了一个树形结构，handler 是具体的业务控制器逻辑存放位置，segment 是树中的这个节点存放的内容，isLast 用于区别这个树中的节点是否有实际的路由含义。

有了数据结构后，第二步，我们就往 Tree 这个 trie 树结构中增加“路由规则”的逻辑。写之前，我们还是暂停一下想一想，会不会出现问题。**之前提过会存在通配符，那直接加规则其实是有可能冲突的。**比如：

```
1 /user/name
2 /user/:id
```

[复制代码](#)

这两个路由规则实际上就冲突了，如果请求地址是 /user/name，那么两个规则都匹配，无法确定哪个规则生效。所以在增加路由之前，我们需要判断这个路由规则是否已经在 trie 树中存在了。

这里，我们可以用 matchNode 方法，寻找某个路由在 trie 树中匹配的节点，如果有匹配节点，返回节点指针，否则返回 nil。**matchNode 方法的参数是一个 URI，返回值是指向 node 的指针，它的实现思路是使用函数递归**，我简单说明一下思路：

首先，将需要匹配的 URI 根据第一个分隔符 / 进行分割，只需要最多分割成为两个段。

如果只能分割成一个段，说明 URI 中没有分隔符了，这时候再检查下一级节点中是否有匹配这个段的节点就行。

如果分割成了两个段，我们用第一个段来检查下一个级节点中是否有匹配这个段的节点。

如果没有，说明这个路由规则在树中匹配不到。

如果下一级节点中有符合第一个分割段的（这里需要注意可能不止一个符合），我们就将所有符合的节点进行函数递归，重新应用于 `matchNode` 函数中，只不过这时候 `matchNode` 函数作用于子节点，参数变成了切割后的第二个段。

好思路就讲完了，整个流程里，会频繁使用到“过滤下一层满足 `segment` 规则的子节点”，所以我们也用一个函数 `filterChildNodes` 将它封装起来。这个函数的逻辑就比较简单了：遍历下一层子节点，判断 `segment` 是否匹配传入的参数 `segment`。

在框架文件夹中的 `tree.go` 中，我们完成 `matchNode` 和 `filterChildNodes` 完整代码实现，放在这里了，具体逻辑我也加了详细的批注帮你理解。


[复制代码](#)

```
1 // 判断一个segment是否是通用segment，即以:开头
2 func isWildSegment(segment string) bool {
3     return strings.HasPrefix(segment, ":")
4 }
5
6 // 过滤下一层满足segment规则的子节点
7 func (n *node) filterChildNodes(segment string) []*node {
8     if len(n.chlds) == 0 {
9         return nil
10    }
11
12    // 如果segment是通配符，则所有下一层子节点都满足需求
13    if isWildSegment(segment) {
14        return n.chlds
15    }
16
17    nodes := make([]*node, 0, len(n.chlds))
18    // 过滤所有的下一层子节点
19    for _, cnode := range n.chlds {
20        if isWildSegment(cnode.segment) {
21            // 如果下一层子节点有通配符，则满足需求
22            nodes = append(nodes, cnode)
23        } else if cnode.segment == segment {
24            // 如果下一层子节点没有通配符，但是文本完全匹配，则满足需求
25            nodes = append(nodes, cnode)
26        }
27    }
28
29    return nodes
```

```
30 }
31
32 // 判断路由是否已经在节点的所有子节点树中存在于
33 func (n *node) matchNode(uri string) *node {
34     // 使用分隔符将uri切割为两个部分
35     segments := strings.SplitN(uri, "/", 2)
36     // 第一个部分用于匹配下一层子节点
37     segment := segments[0]
38     if !isWildSegment(segment) {
39         segment = strings.ToUpper(segment)
40     }
41     // 匹配符合的下一层子节点
42     cnodes := n.filterChildNodes(segment)
43     // 如果当前子节点没有一个符合，那么说明这个uri一定是之前不存在，直接返回nil
44     if cnodes == nil || len(cnodes) == 0 {
45         return nil
46     }
47
48     // 如果只有一个segment，则是最后一个标记
49     if len(segments) == 1 {
50         // 如果segment已经是最后一个节点，判断这些cnode是否有isLast标志
51         for _, tn := range cnodes {
52             if tn.isLast {
53                 return tn
54             }
55         }
56
57         // 都不是最后一个节点
58         return nil
59     }
60
61     // 如果有2个segment，递归每个子节点继续进行查找
62     for _, tn := range cnodes {
63         tnMatch := tn.matchNode(segments[1])
64         if tnMatch != nil {
65             return tnMatch
66         }
67     }
68     return nil
69 }
```

现在有了 `matchNode` 和 `filterChildNodes` 函数，我们就可以开始写第二步里最核心的增加路由的函数逻辑了。

首先，确认路由是否冲突。我们先检查要增加的路由规则是否在树中已经有可以匹配的节点了。如果有的话，代表当前待增加的路由和已有路由存在冲突，这里我们用到了刚刚定义的 `matchNode`。更新刚才框架文件夹中的 `tree.go` 文件：


 复制代码

```

1 // 增加路由节点
2 func (tree *Tree) AddRouter(uri string, handler ControllerHandler) error {
3     n := tree.root
4     // 确认路由是否冲突
5     if n.matchNode(uri) != nil {
6         return errors.New("route exist: " + uri)
7     }
8
9     ...
10 }
11

```

然后继续增加路由规则。我们增加路由的每个段时，先去树的每一层中匹配查找，如果已经有了符合这个段的节点，就不需要创建节点，继续匹配待增加路由的下个段；否则，需要创建一个新的节点用来代表这个段。这里，我们用到了定义的 filterChildNodes。

 复制代码

```

1 // 增加路由节点
2 /*
3 /book/list
4 /book/:id (冲突)
5 /book/:id/name
6 /book/:student/age
7 /:user/name(冲突)
8 /:user/name/:age
9 */
10 func (tree *Tree) AddRouter(uri string, handler ControllerHandler) error {
11     n := tree.root
12     if n.matchNode(uri) != nil {
13         return errors.New("route exist: " + uri)
14     }
15
16     segments := strings.Split(uri, "/")
17     // 对每个segment
18     for index, segment := range segments {
19
20         // 最终进入Node segment的字段
21         if !isWildSegment(segment) {
22             segment = strings.ToUpper(segment)
23         }
24         isLast := index == len(segments)-1
25
26         var objNode *node // 标记是否有合适的子节点
27
28         childNodes := n.filterChildNodes(segment)
29         // 如果有匹配的子节点

```

```

30     if len(childNodes) > 0 {
31         // 如果有segment相同的子节点，则选择这个子节点
32         for _, cnode := range childNodes {
33             if cnode.segment == segment {
34                 objNode = cnode
35                 break
36             }
37         }
38     }
39
40     if objNode == nil {
41         // 创建一个当前node的节点
42         cnode := newNode()
43         cnode.segment = segment
44         if isLast {
45             cnode.isLast = true
46             cnode.handler = handler
47         }
48         n.childds = append(n.childds, cnode)
49         objNode = cnode
50     }
51
52     n = objNode
53 }
54
55 return nil
56 }

```

到这里，第二步增加路由的规则逻辑已经有了，我们要开始第三步，编写“查找路由”的逻辑。这里你会发现，由于我们之前已经定义过 `matchNode`（匹配路由节点），所以这里只需要复用这个函数就行了。


[复制代码](#)

```

1 // 匹配uri
2 func (tree *Tree) FindHandler(uri string) ControllerHandler {
3     // 直接复用matchNode函数，uri是不带通配符的地址
4     matchNode := tree.root.matchNode(uri)
5     if matchNode == nil {
6         return nil
7     }
8     return matchNode.handler
9 }
10


```

前三步已经完成了，**最后一步，我们把“增加路由规则”和“查找路由”添加到框架中。**还记得吗，在静态路由匹配的时候，在 Core 中使用哈希定义的路由，这里将哈希替换为 trie 树。还是在框架文件夹中的 core.go 文件，找到对应位置作修改：

 复制代码

```
1 type Core struct {  
2     router map[string]*Tree // all routers  
3 }
```

对应路由增加的方法，也从哈希的增加逻辑，替换为 trie 树的“增加路由规则”逻辑。同样更新 core.go 文件中的下列方法：

 复制代码

```
1 // 初始化Core结构  
2 func NewCore() *Core {  
3     // 初始化路由  
4     router := map[string]*Tree{}  
5     router["GET"] = NewTree()  
6     router["POST"] = NewTree()  
7     router["PUT"] = NewTree()  
8     router["DELETE"] = NewTree()  
9     return &Core{router: router}  
10 }  
11  
12  
13  
14 // 匹配GET 方法，增加路由规则  
15 func (c *Core) Get(url string, handler ControllerHandler) {  
16     if err := c.router["GET"].AddRouter(url, handler); err != nil {  
17         log.Fatal("add router error: ", err)  
18     }  
19 }  
20  
21 // 匹配POST 方法，增加路由规则  
22 func (c *Core) Post(url string, handler ControllerHandler) {  
23     if err := c.router["POST"].AddRouter(url, handler); err != nil {  
24         log.Fatal("add router error: ", err)  
25     }  
26 }  
27  
28 // 匹配PUT 方法，增加路由规则  
29 func (c *Core) Put(url string, handler ControllerHandler) {  
30     if err := c.router["PUT"].AddRouter(url, handler); err != nil {  
31         log.Fatal("add router error: ", err)  
32     }  
33 }
```

```
33 }
34
35 // 匹配DELETE 方法，增加路由规则
36 func (c *Core) Delete(url string, handler ControllerHandler) {
37     if err := c.router["DELETE"].AddRouter(url, handler); err != nil {
38         log.Fatal("add router error: ", err)
39     }
40 }
```

之前在 Core 中定义的匹配路由函数的实现逻辑，从哈希匹配修改为 trie 树匹配就可以了。继续更新 core.go 文件：

[复制代码](#)

```
1 // 匹配路由，如果没有匹配到，返回nil
2 func (c *Core) FindRouteByRequest(request *http.Request) ControllerHandler {
3     // uri 和 method 全部转换为大写，保证大小写不敏感
4     uri := request.URL.Path
5     method := request.Method
6     upperMethod := strings.ToUpper(method)
7
8     // 查找第一层map
9     if methodHandlers, ok := c.router[upperMethod]; ok {
10         return methodHandlers.FindHandler(uri)
11     }
12     return nil
13 }
14
```

动态匹配规则就改造完成了。

验证

现在，四个需求都已经实现了。我们验证一下：定义包含有静态路由、批量通用前缀、动态路由的路由规则，每个控制器我们就直接输出控制器的名字，然后启动服务。

这个时候我们就可以去修改业务文件夹下的路由文件 route.go：

[复制代码](#)

```
1 // 注册路由规则
2 func registerRouter(core *framework.Core) {
3     // 需求1+2:HTTP方法+静态路由匹配
4     core.Get("/user/login", UserLoginController)
5 }
```

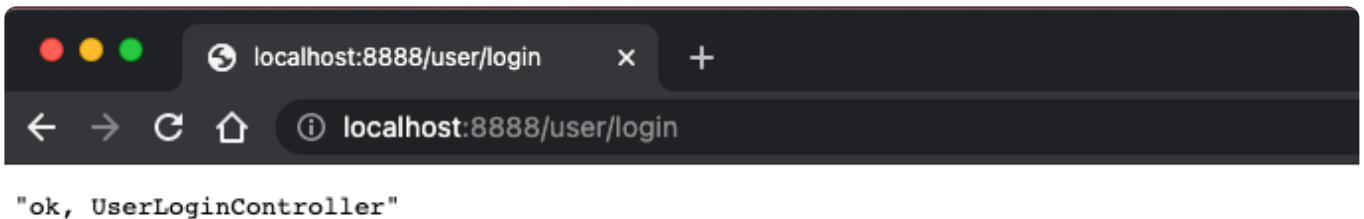
```
6 // 需求3:批量通用前缀
7 subjectApi := core.Group("/subject")
8 {
9     // 需求4:动态路由
10    subjectApi.Delete("/:id", SubjectDelController)
11    subjectApi.Put("/:id", SubjectUpdateController)
12    subjectApi.Get("/:id", SubjectGetController)
13    subjectApi.Get("/list/all", SubjectListController)
14 }
15 }
```

同时在业务文件夹下创建对应的业务控制器 `user_controller.go` 和 `subject_controller.go`。具体里面的逻辑代码就是打印出对应的控制器名字，比如

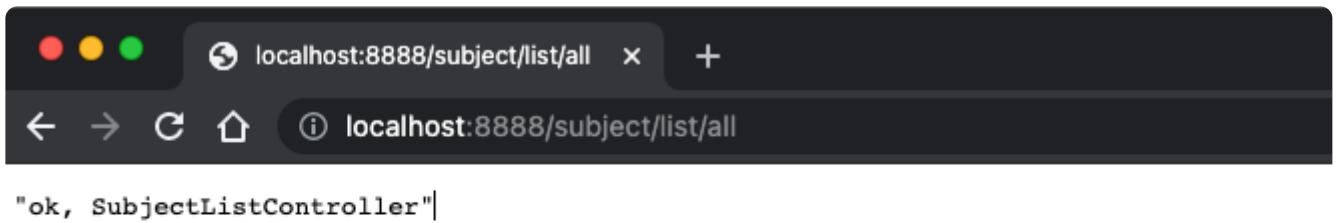
```
1 func UserLoginController(c *framework.Context) error {
2     // 打印控制器名字
3     c.Json(200, "ok, UserLoginController")
4     return nil
5 }
```

[复制代码](#)

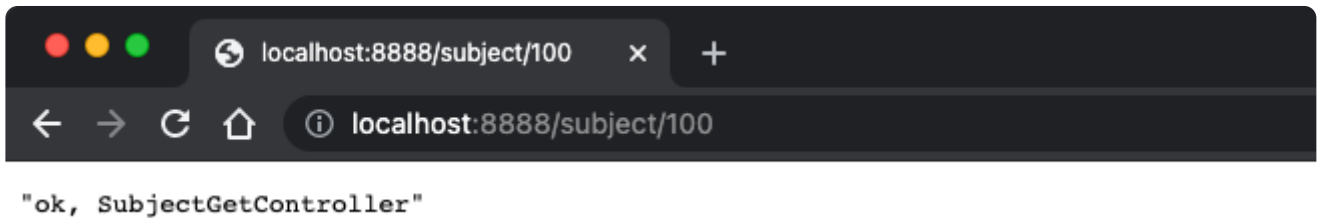
来看服务启动情况：访问地址 `/user/login` 匹配路由 `UserLoginContorller`。



访问地址 `/subject/list/all` 匹配路由 `SubjectListController`。

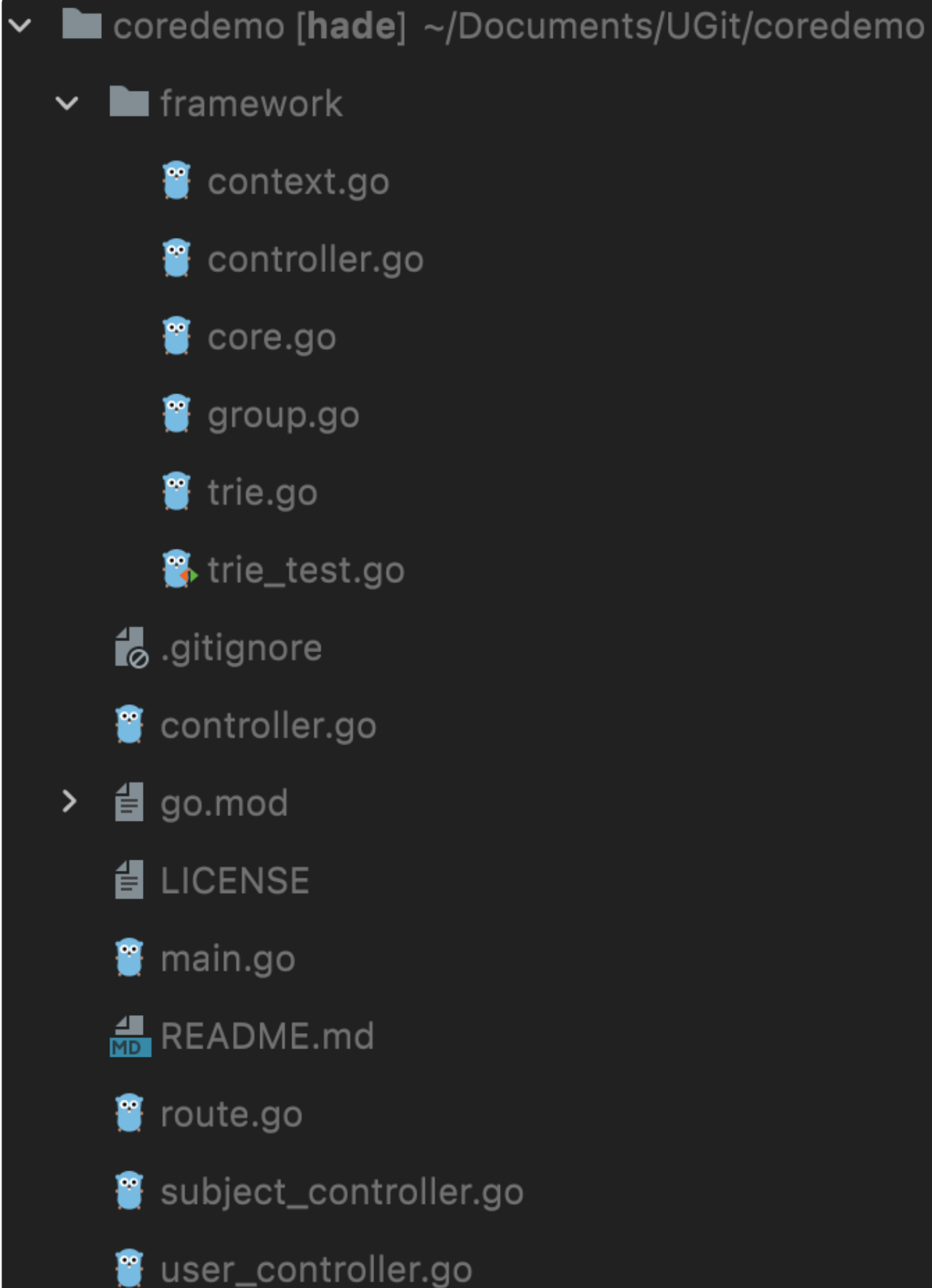


访问地址 /subject/100 匹配动态路由 SubjectGetController。



路由规则符合要求！

今天的文件及代码结构如下，新建的文件夹多一点你可以对照着 GitHub 再看看，代码地址在 [@geekbang/03](#) 分支上：



小结

在这一讲，我们一步步实现了满足四个需求的路由：HTTP 方法匹配、批量通用前缀、静态路由匹配和动态路由匹配。


我们使用 IGroup 结构和在 Core 中定义 key 为方法的路由，实现了 HTTP 方法匹配、批量通用前缀这两个需求，并且用哈希来实现静态路由匹配，之后我们使用 trie 树算法替代哈希算法，实现了动态路由匹配的需求。

所以，你有没有发现，**其实所谓的实现功能，写代码只是其中一小部分，如何思考、如何考虑容错性、扩展性和复用性，这个反而是更大的部分。**

以今天实现的路由这个功能为例，你是否考虑到了 URI 的容错性，在 Group 返回时候是否使用接口增加扩展性，在实现动态匹配的时候是否考虑函数复用性。我们要记住的是，思路比代码实现更重要。

思考题

光说不练假把式，毕竟我们是实战课，那针对第三个需求“批量通用前缀”，我们扩展一下变成：需要能多层嵌套通用前缀，这么定义路由：

 复制代码

```
1 // 注册路由规则
2 func registerRouter(core *framework.Core) {
3     // 静态路由+HTTP方法匹配
4     core.Get("/user/login", UserLoginController)
5
6     // 批量通用前缀
7     subjectApi := core.Group("/subject")
8     {
9         subjectInnerApi := subjectApi.Group("/info")
10        {
11            subjectInnerApi.Get("/name", SubjectNameController)
12        }
13    }
14 }
```

结合刚才说的考虑代码的设计感，你想一想如何实现呢？

欢迎在留言区分享你的思考。如果你觉得今天的内容对你有所帮助，也欢迎你把今天的内容分享给你身边的朋友，邀请他一起学习~

 赞 2  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 02 | Context：请求控制器，让每个请求都在掌控之中

下一篇 04 | 中间件：如何提高框架的可拓展性？

精选留言 (4)



半格式Hal

2021-09-19

老师好，两级映射哈希，“第一级 hash 是请求 Method，第二级 hash 是 Request-URL”。能不能第一级Request-URL，第二级Method的呢？会有什么问题吗？顺便说下这里Request打错了。

展开



qinsi

2021-09-18

/:user/name(冲突)
/:user/name/:age

这两个是不是写反了



好家庭

2021-09-17

为IGroup实现如下接口：
Group(string) IGroup

类似于builder设计模式，可以链式调用

作者回复: 你好，是的，这种方式支持链式调用，使用起来比较舒服。



**小然**

2021-09-17

文中的树的图结构是否问题呢，按照代码实际生成的每一颗trie树root节点下面第一个子节点实际上是segment为空的节点，然后在这个节点下才是各个一级路劲的子节点。是我理解错误吗？我带着文中的树图结构去看代码添加路由算法看起来有很大的偏差，脑袋里想象纠正，先在root节点下先加一个segment为空的节点就好理解了。

展开 ✓

