



下载APP

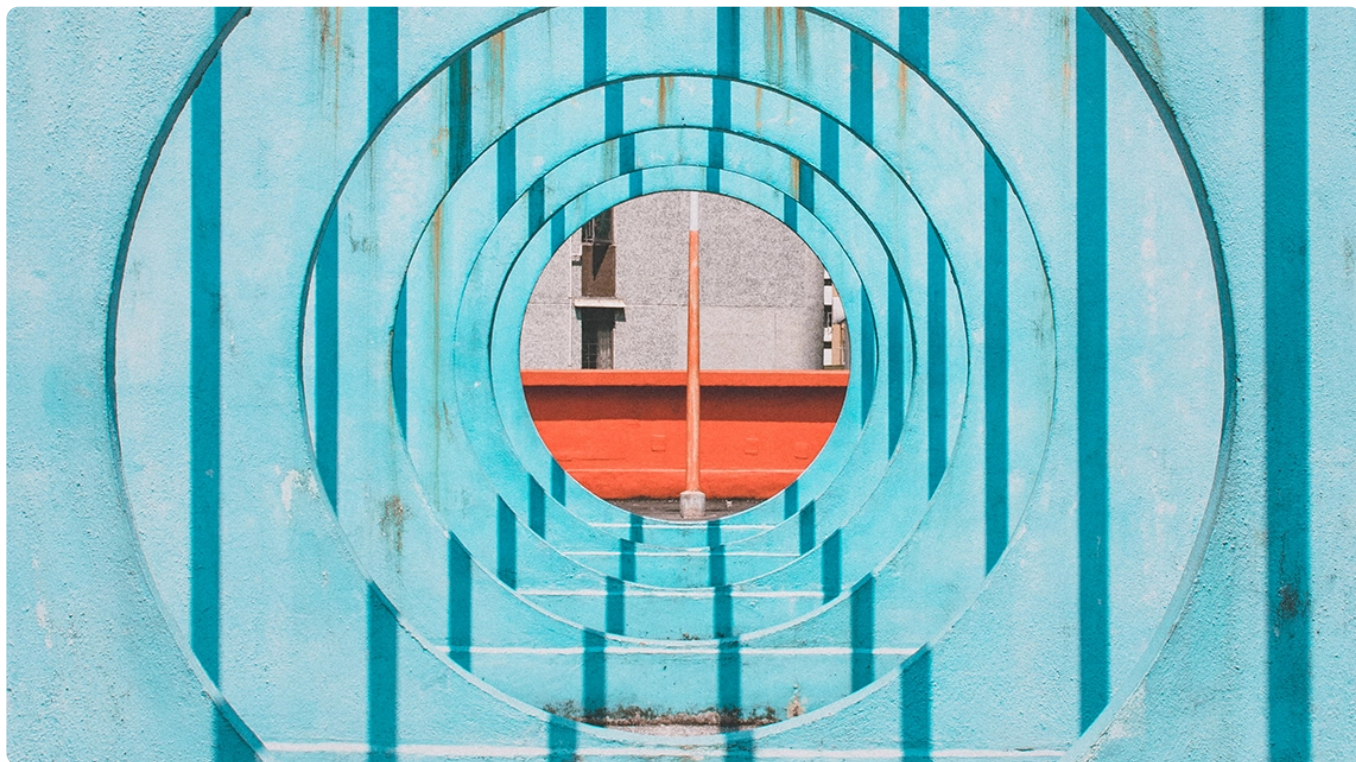


04 | 中间件：如何提高框架的可拓展性？

2021-09-20 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 16:15 大小 14.88M



你好，我是轩脉刃。

到目前为止我们已经完成了 Web 框架的基础部分，使用 net/http 启动了一个 Web 服务，并且定义了自己的 Context，可以控制请求超时。

之前在讲具体实现的时候，我们反复强调要注意代码的优化。那么如何优化呢？具体来说，很重要的一点就是封装。所以今天我们就回顾一下之前写的代码，看看如何通过封装来进一步提高代码扩展性。



在第二课，我们在业务文件夹中的 controller.go 的逻辑中设置了一个有超时时长的控制器：

```
1 func FooControllerHandler(c *framework.Context) error {
2     ...
3     // 在业务逻辑处理前，创建有定时器功能的 context
4     durationCtx, cancel := context.WithTimeout(c.BaseContext(), time.Duration(1*
5     defer cancel()
6
7     go func() {
8         ...
9         // 执行具体的业务逻辑
10
11         time.Sleep(10 * time.Second)
12         // ...
13
14         finish <- struct{}{}
15     }()
16     // 在业务逻辑处理后，操作输出逻辑...
17     select {
18         ...
19         case <-finish:
20             fmt.Println("finish")
21             ...
22     }
23     return nil
24 }
```

在正式执行业务逻辑之前，创建了一个具有定时器功能的 Context，然后开启一个 Goroutine 执行正式的业务逻辑，并且监听定时器和业务逻辑，哪个先完成，就先输出内容。

首先从代码功能分析，这个控制器像由两部分组成。

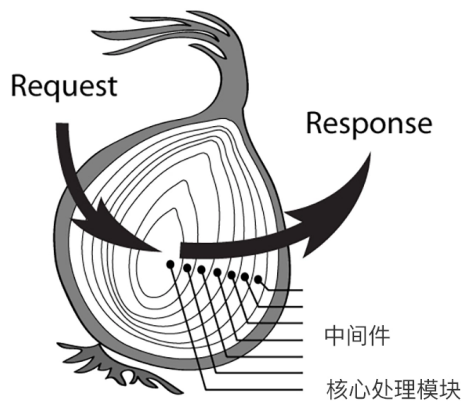
一部分是**业务逻辑**，也就是 time.Sleep 函数所代表的逻辑，在实际生产过程中，这里会有很重的业务逻辑代码；而另一部分是**非业务逻辑**，比如创建 Context、通道等待 finish 信号等。很明显，这个非业务逻辑是非常通用的需求，可能在多个控制器中都会使用到。

而且考虑复用性，这里只是写了一个控制器，那如果有多个控制器呢，我们难道要为每个控制器都写上这么一段超时代码吗？那就非常冗余了。

所以，能不能设计一个机制，**将这些非业务逻辑代码抽象出来，封装好，提供接口给控制器使用**。这个机制的实现，就是我们今天要讲的中间件。

怎么实现这个中间件呢？我们再观察一下刚才的代码找找思路。

代码的组织顺序很清晰，先预处理请求，再处理业务逻辑，最后处理返回值，你发现没有这种顺序，其实很符合设计模式中的装饰器模式。装饰器模式，顾名思义，就是在核心处理模块的外层增加一个又一个的装饰，类似洋葱。



现在，抽象出中间件的思路是不是就很清晰了，把核心业务逻辑先封装起来，然后一层一层添加装饰，最终让所有请求正序一层层通过装饰器，进入核心处理模块，再反序退出装饰器。原理就是这么简单，不难理解，我们接着看该如何实现。

使用函数嵌套方式实现中间件

装饰器模式是一层一层的，所以具体实现其实也不难想到，就是使用函数嵌套。

首先，我们封装核心的业务逻辑。就是说，这个中间件的输入是一个核心的业务逻辑 `ControllerHandler`，输出也应该是一个 `ControllerHandler`。所以**对于一个超时控制器，我们可以定义一个中间件为 `TimeoutHandler`。**

在框架文件夹中，我们创建一个 `timeout.go` 文件来存放这个中间件。

复制代码

```
1 func TimeoutHandler(fun ControllerHandler, d time.Duration) ControllerHandler
2     // 使用函数回调
3     return func(c *Context) error {
```

```
4     finish := make(chan struct{}, 1)
5     panicChan := make(chan interface{}, 1)
6
7     // 执行业务逻辑前预操作：初始化超时 context
8     durationCtx, cancel := context.WithTimeout(c.BaseContext(), d)
9     defer cancel()
10
11     c.request.WithContext(durationCtx)
12
13     go func() {
14         defer func() {
15             if p := recover(); p != nil {
16                 panicChan <- p
17             }
18         }()
19         // 执行具体的业务逻辑
20         fun(c)
21
22         finish <- struct{}{}
23     }()
24     // 执行业务逻辑后操作
25     select {
26     case p := <-panicChan:
27         log.Println(p)
28         c.responseWriter.WriteHeader(500)
29     case <-finish:
30         fmt.Println("finish")
31     case <-durationCtx.Done():
32         c.SetHasTimeout()
33         c.responseWriter.Write([]byte("time out"))
34     }
35     return nil
36 }
37 }
38
```

仔细看下这段代码，中间件函数的返回值是一个匿名函数，这个匿名函数实现了 ControllerHandler 函数结构，参数为 Context，返回值为 error。

在这个匿名函数中，我们先创建了一个定时器 Context，然后开启一个 Goroutine，在 Goroutine 中执行具体的业务逻辑。这个 Goroutine 会在业务逻辑执行结束后，通过一个 finish 的 channel 来传递结束信号；也会在业务出现异常的时候，通过 panicChan 来传递异常信号。

而在业务逻辑之外的主 Goroutine 中，会同时进行多个信号的监听操作，包括结束信号、异常信号、超时信号，耗时最短的信号到达后，请求结束。这样，我们就完成了设置业务

超时的任务。

于是在业务文件夹 `route.go` 中，路由注册就可以修改为：

[复制代码](#)

```
1 // 在核心业务逻辑 UserLoginController 之外，封装一层 TimeoutHandler
2 core.Get("/user/login", framework.TimeoutHandler(UserLoginController, time.Sec
```

这种函数嵌套方式，让下层中间件是上层中间件的参数，通过一层层嵌套实现了中间件的装饰器模式。

但是你再想一步，就会发现，这样实现的中间件机制有两个问题：

1. **中间件是循环嵌套的**，当有多个中间件的时候，整个嵌套长度就会非常长，非常不优雅的，比如：

[复制代码](#)

```
1 TimeoutHandler(LogHandler(recoveryHandler(UserLoginController)))
```

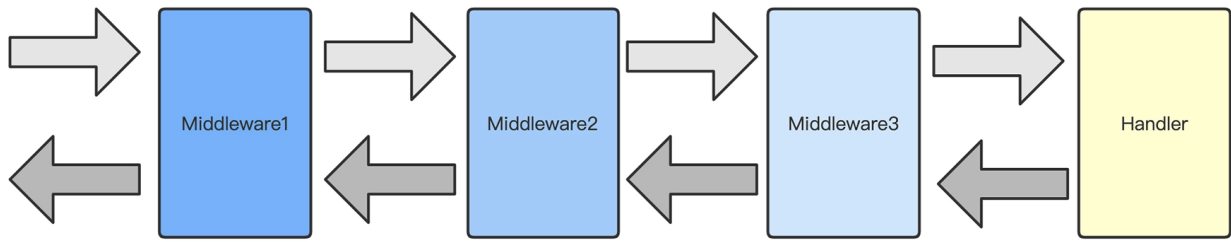
2. 刚才的实现，**只能为单个业务控制器设置中间件，不能批量设置**。上一课我们开发的路由是具有同前缀分组功能的（`IGroup`），需要批量为某个分组设置一个超时时长。

所以，我们要对刚才实现的简单中间件代码做一些改进。怎么做呢？

使用 pipeline 思想改造中间件

一层层嵌套不好用，如果我们将每个核心控制器所需要的中间件，使用一个数组链接（`Chain`）起来，形成一条流水线（`Pipeline`），就能完美解决这两个问题了。

请求流的流向如下图所示：



这个 Pipeline 模型和前面的洋葱模型不一样的点在于，**Middleware 不再以下一层的 ControllerHandler 为参数了，它只需要返回有自身中间件逻辑的 ControllerHandler。**

也就是在框架文件夹中的 timeout.go 中，我们将 Middleware 的形式从刚才的：

📄 复制代码

```
1 func TimeoutHandler(fun ControllerHandler, d time.Duration) ControllerHandler
2 // 使用函数回调
3 return func(c *Context) error {
4     //...
5 }
6 }
```

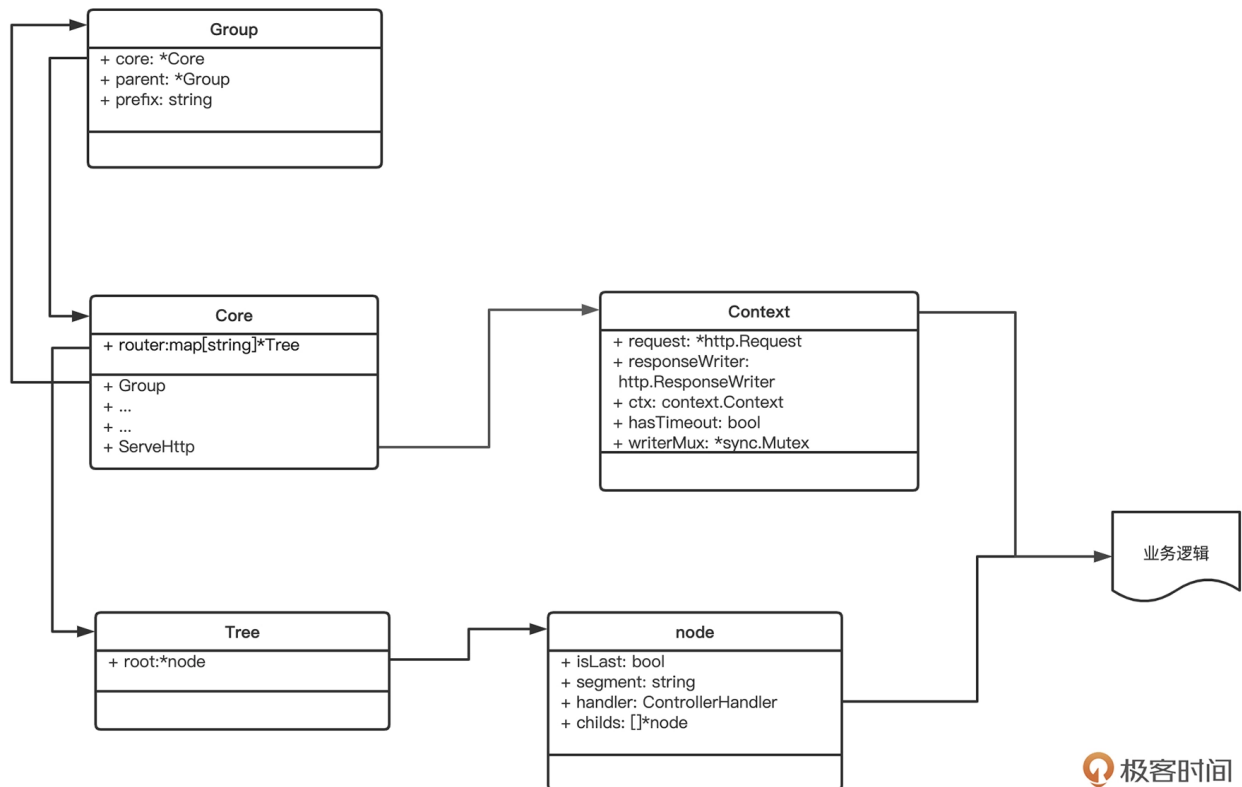
变成这样：

📄 复制代码

```
1 // 超时控制器参数中ControllerHandler结构已经去掉
2 func Timeout(d time.Duration) framework.ControllerHandler {
3     // 使用函数回调
4     return func(c *framework.Context) error {
5         //...
6     }
7 }
```

但是在中间件注册的回调函数中，如何调用下一个 `ControllerHandler` 呢？在回调函数中，只有 `framework.Context` 这个数据结构作为参数。

所以就需要我们在 `Context` 这个数据结构中想一些办法了。回顾下目前有的数据结构：`Core`、`Context`、`Tree`、`Node`、`Group`。



它们基本上都是以 `Core` 为中心，在 `Core` 中设置路由 `router`，实现了 `Tree` 结构，在 `Tree` 结构中包含路由节点 `node`；在注册路由的时候，将对应的业务核心处理逻辑 `handler`，放在 `node` 结构的 `handler` 属性中。

而 `Core` 中的 `ServeHttp` 方法会创建 `Context` 数据结构，然后 `ServeHttp` 方法再根据 `Request-URI` 查找指定 `node`，并且将 `Context` 结构和 `node` 中的控制器 `ControllerHandler` 结合起来执行具体的业务逻辑。

结构都梳理清楚了，怎么改造成流水线呢？

我们可以将每个中间件构造出来的 `ControllerHandler` 和最终的业务逻辑的 `ControllerHandler` 结合在一起，成为一个 `ControllerHandler` 数组，也就是控制器链。

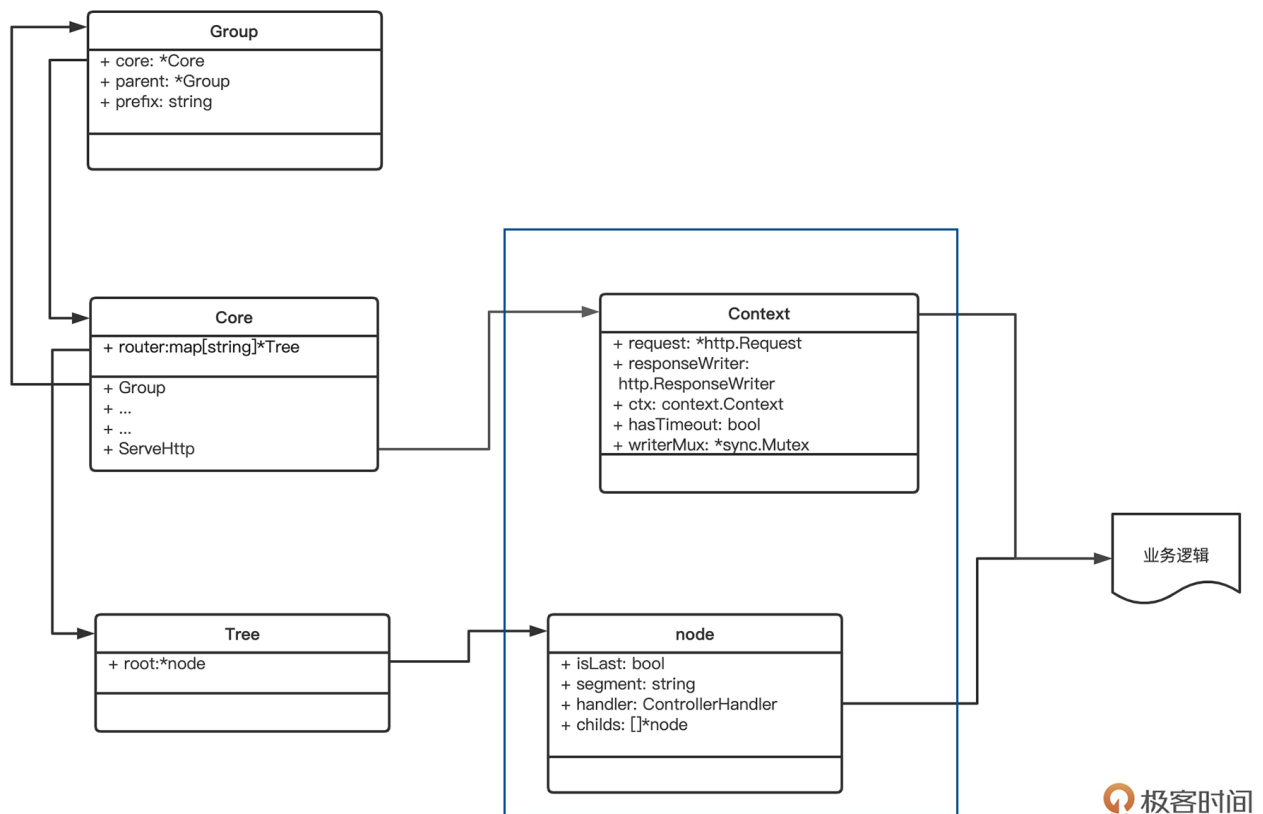
在最终执行业务代码的时候，能一个个调用控制器链路上的控制器。

这个想法其实是非常自然的，因为中间件中创造出来的 `ControllerHandler` 匿名函数，和最终的控制器业务逻辑 `ControllerHandler`，都是**同样的结构**，所以我们可以选用 **`Controllerhandler` 的数组**，来表示某个路由的业务逻辑。

对应到代码上，我们先搞清楚使用链路的方式，再看如何注册和构造链路。

如何使用控制器链路

首先，我们研究下如何使用这个控制器链路，即图中右边部分的改造。



第一步，我们需要修改路由节点 `node`。

在 `node` 节点中将原先的 `Handler`，替换为控制器链路 `Handlers`。这样在寻找路由节点的时候，就能找到对应的控制器链路了。修改框架文件夹中存放 `trie` 树的 `trie.go` 文件：

```
1 // 代表节点
2
```

复制代码


```
3 type node struct {
4     ...
5     handlers []ControllerHandler // 中间件+控制器
6     ...
7 }
```

第二步，我们修改 Context 结构。

由于我们上文提到，在中间件注册的回调函数中，只有 framework.Context 这个数据结构作为参数，所以在 Context 中也需要保存这个控制器链路 (handlers)，并且要记录下当前执行到了哪个控制器 (index)。修改框架文件夹的 context.go 文件：

[复制代码](#)

```
1 // Context代表当前请求上下文
2 type Context struct {
3     ...
4
5     // 当前请求的handler链条
6     handlers []ControllerHandler
7     index    int // 当前请求调用到调用链的哪个节点
8 }
```

第三步，来实现链条调用方式。

为了控制实现链条的逐步调用，我们为 Context 实现一个 Next 方法。这个 Next 方法每调用一次，就将这个控制器链路的调用控制器，往后移动一步。继续在框架文件夹中的 context.go 文件里写：

[复制代码](#)

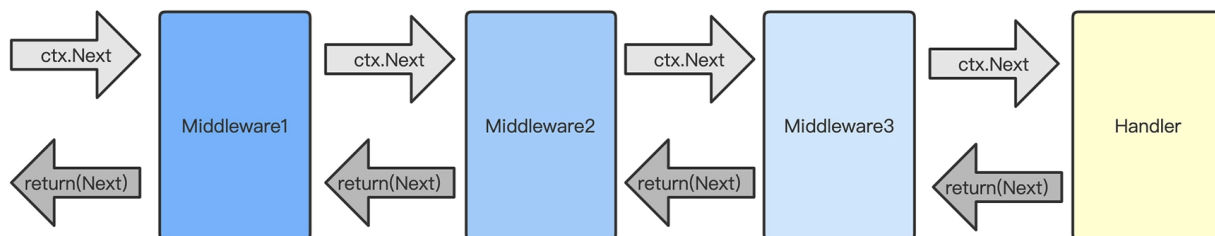
```
1 // 核心函数，调用context的下一个函数
2 func (ctx *Context) Next() error {
3     ctx.index++
4     if ctx.index < len(ctx.handlers) {
5         if err := ctx.handlers[ctx.index](ctx); err != nil {
6             return err
7         }
8     }
9     return nil
10 }
```

这里我再啰嗦一下，Next() 函数是整个链路执行的重点，要好好理解，它通过维护 Context 中的一个下标，来控制链路移动，这个下标表示当前调用 Next 要执行的控制器序列。

Next() 函数会在框架的两个地方被调用：

第一个是在此次请求处理的入口处，即 Core 的 ServeHttp；

第二个是在每个中间件的逻辑代码中，用于调用下个中间件。



极客时间


这里要注意，index 下标表示当前调用 Next 要执行的控制器序列，它的**初始值应该为 -1，每次调用都会自增 1**，这样才能保证第一次调用的时候 index 为 0，定位到控制器链条的下标为 0 的控制器，即第一个控制器。

在框架文件夹 context.go 的初始化 Context 函数中，代码如下：

复制代码


```
1 // NewContext 初始化一个Context
2 func NewContext(r *http.Request, w http.ResponseWriter) *Context {
3     return &Context{
4         ...
5         index:      -1,
6     }
7 }
```

被调用的第一个地方，在入口处调用的代码，写在框架文件夹中的 core.go 文件中：

 复制代码

```
1 // 所有请求都进入这个函数，这个函数负责路由分发
2 func (c *Core) ServeHTTP(response http.ResponseWriter, request *http.Request)
3
4 // 封装自定义context
5 ctx := NewContext(request, response)
6
7 // 寻找路由
8 handlers := c.FindRouteByRequest(request)
9 if handlers == nil {
10     // 如果没有找到，这里打印日志
11     ctx.Json(404, "not found")
12     return
13 }
14
15 // 设置context中的handlers字段
16 ctx.SetHandlers(handlers)
17
18 // 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
19 if err := ctx.Next(); err != nil {
20     ctx.Json(500, "inner error")
21     return
22 }
23 }
```

被调用的第二个位置在中间件中，每个中间件都通过调用 context.Next 来调用下一个中间件。所以我们可以创建 middleware 目录，其中创建一个 test.go 存放我们的测试中间件：

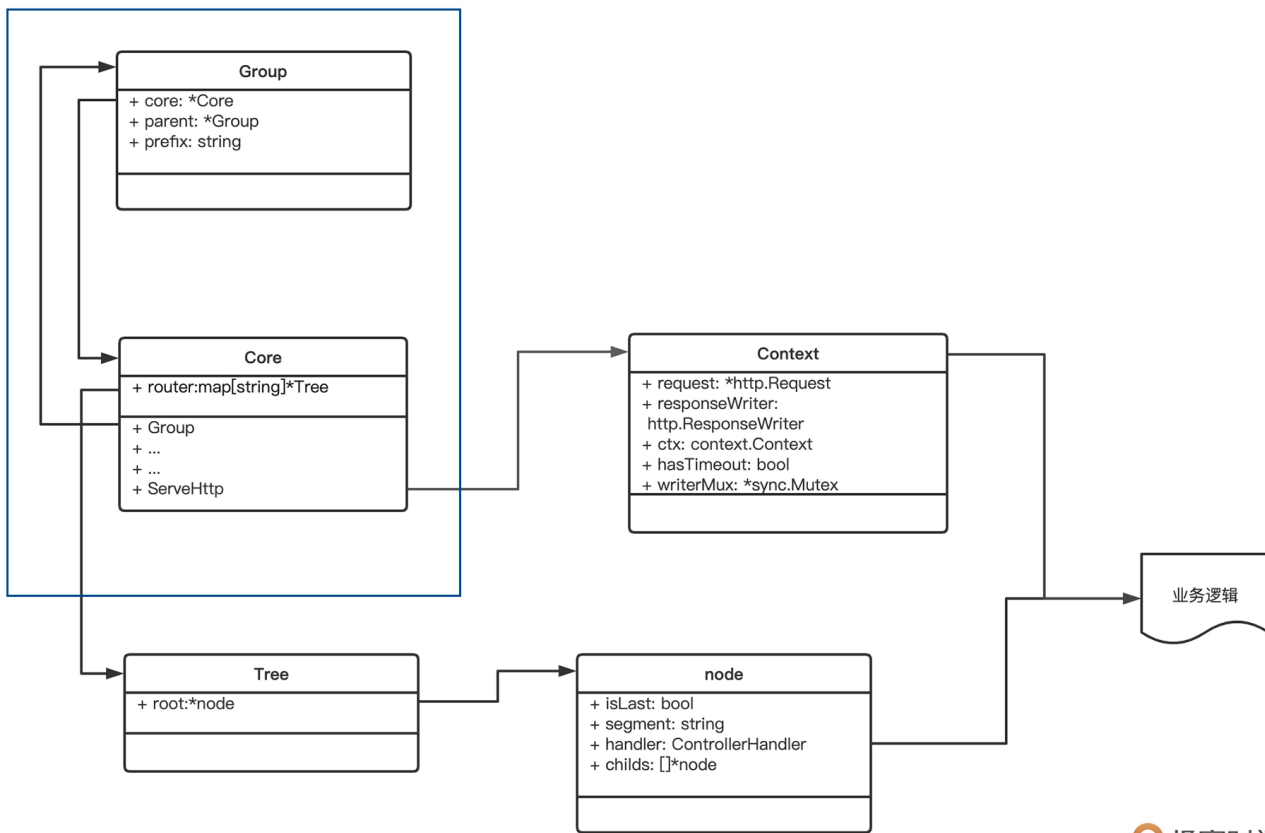
 复制代码

```
1 func Test1() framework.ControllerHandler {
2     // 使用函数回调
3     return func(c *framework.Context) error {
4         fmt.Println("middleware pre test1")
5         c.Next() // 调用Next往下调用，会自增ctx.index
6         fmt.Println("middleware post test1")
7         return nil
8     }
9 }
10
11 func Test2() framework.ControllerHandler {
12     // 使用函数回调
13     return func(c *framework.Context) error {
14         fmt.Println("middleware pre test2")
15     }
16 }
```

```
15     c.Next() // 调用Next往下调用，会自增contxt.index
16     fmt.Println("middleware post test2")
17     return nil
18 }
19 }
20
```

如何注册控制器链路

如何使用控制器链路，我们就讲完了，再看控制器链路如何注册，就是之前 UML 图的左边部分。




极客时间

很明显，现有的函数没有包含注册中间件逻辑，所以我们需要为 **Group** 和 **Core** 两个结构增加注册中间件入口，要设计两个地方：


Core 和 Group 单独设计一个 Use 函数，为其数据结构负责的路由批量设置中间件
为 Core 和 Group 注册单个路由的 Get / Post / Put / Delete 函数，设置中间件

先看下批量设置中间件的 Use 函数，我们在框架文件夹中的 core.go 修改：

 复制代码


```
1 // 注册中间件
2 func (c *Core) Use(middlewares ...ControllerHandler) {
3     c.middlewares = middlewares
4 }
5
6 // 注册中间件
7 func (g *Group) Use(middlewares ...ControllerHandler) {
8     g.middlewares = middlewares
9 }
10
```

注意下这里的参数，使用的是 Golang 的可变参数，**这个可变参数代表，我可以传递 0 ~ n 个 ControllerHandler 类型的参数**，这个设计会增加函数的易用性。它在业务文件夹中使用起来的形式是这样的，在 main.go 中：

 复制代码


```
1 // core中使用use注册中间件
2 core.Use(
3     middleware.Test1(),
4     middleware.Test2())
5
6 // group中使用use注册中间件
7 subjectApi := core.Group("/subject")
8 subjectApi.Use(middleware.Test3())
```

再看单个路由设置中间件的函数，我们也使用可变参数，改造注册路由的函数（Get /Post /Delete /Put），继续在框架文件夹中的 core.go 里修改：

 复制代码


```
1 // Core的Get方法进行改造
2 func (c *Core) Get(url string, handlers ...ControllerHandler) {
3     // 将core的middleware 和 handlers结合起来
4     allHandlers := append(c.middlewares, handlers...)
5     if err := c.router["GET"].AddRouter(url, allHandlers); err != nil {
6         log.Fatal("add router error: ", err)
7     }
8 }
9 ...
```

同时修改框架文件夹中的 group.go：

 复制代码

```
1 // 改造IGroup 的所有方法
2 type IGroup interface {
3     // 实现HttpMethod方法
4     Get(string, ...ControllerHandler)
5     Post(string, ...ControllerHandler)
6     Put(string, ...ControllerHandler)
7     Delete(string, ...ControllerHandler)
8     //..
9 }
10
11 // 改造Group的Get方法
12 func (g *Group) Get(uri string, handlers ...ControllerHandler) {
13     uri = g.getAbsolutePrefix() + uri
14     allHandlers := append(g.getMiddlewares(), handlers...)
15     g.core.Get(uri, allHandlers...)
16 }
17
18 ...
```

这样，回到业务文件夹中的 router.go，我们注册路由的使用方法就可以变成如下形式：

 复制代码

```
1 // 注册路由规则
2 func registerRouter(core *framework.Core) {
3     // 在core中使用middleware.Test3() 为单个路由增加中间件
4     core.Get("/user/login", middleware.Test3(), UserLoginController)
5
6     // 批量通用前缀
7     subjectApi := core.Group("/subject")
8     {
9         ...
10        // 在group中使用middleware.Test3() 为单个路由增加中间件
11        subjectApi.Get("/:id", middleware.Test3(), SubjectGetController)
12    }
13 }
```

不管是通过批量注册中间件，还是单个注册中间件，最终都要汇总到路由节点 node 中，所以这里我们调用了上一节课最终增加路由的函数 Tree.AddRouter，把将这个请求对应的 Core 结构里的中间件和 Group 结构里的中间件，都聚合起来，成为最终路由节点的中间件。

聚合的逻辑在 `group.go` 和 `core.go` 中都有，实际上就是将 **Handler 和 Middleware 一起放在一个数组中**。

[复制代码](#)

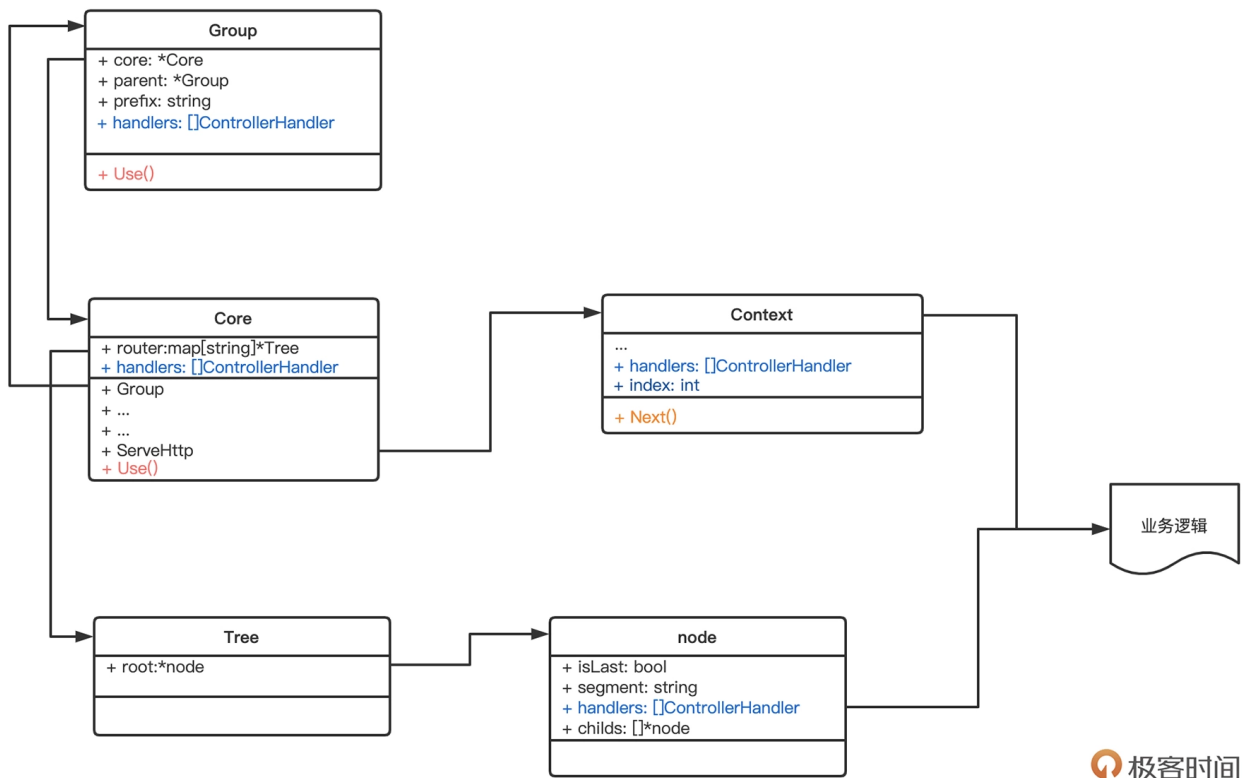
```
1 // 获取某个group的middleware
2 // 这里就是获取除了Get/Post/Put/Delete之外设置的middleware
3 func (g *Group) getMiddlewares() []ControllerHandler {
4     if g.parent == nil {
5         return g.middlewares
6     }
7
8     return append(g.parent.getMiddlewares(), g.middlewares...)
9 }
10
11 // 实现Get方法
12 func (g *Group) Get(uri string, handlers ...ControllerHandler) {
13     uri = g.getAbsolutePrefix() + uri
14     allHandlers := append(g.getMiddlewares(), handlers...)
15     g.core.Get(uri, allHandlers...)
16 }
```

在 `core.go` 文件夹里写：

[复制代码](#)

```
1 // 匹配GET 方法，增加路由规则
2 func (c *Core) Get(url string, handlers ...ControllerHandler) {
3     // 将core的middleware 和 handlers结合起来
4     allHandlers := append(c.middlewares, handlers...)
5     if err := c.router["GET"].AddRouter(url, allHandlers); err != nil {
6         log.Fatal("add router error: ", err)
7     }
8 }
9
```

到这里，我们使用 pipeline 思想对中间件的改造就完成了，最终的 UML 类图如下：



让我们简要回顾下改造过程。

第一步使用控制器链路，我们改造了 **node** 和 **Context** 两个数据结构。为 **node** 增加了 `handlers`，存放这个路由注册的所有中间件；**Context** 也增加了 `handlers`，在 `Core.ServeHttp` 的函数中，创建 **Context** 结构，寻找到请求对应的路由节点，然后把路由节点的 `handlers` 数组，复制到 **Context** 中的 `handlers`。

为了实现真正的链路调用，需要在框架的两个地方调用 **Context.Next()** 方法，一个是启动业务逻辑的地方，一个是每个中间件的调用。

第二步如何注册控制器链路，我们改造了 **Group** 和 **Core** 两个数据结构，为它们增加了注册中间件的入口，一处是批量增加中间件函数 `Use`，一处是在注册单个路由的 `Get / Post / Delete / Put` 方法中，为单个路由设置中间件。在设计入口的时候，我们使用了可变参数的设计，提高注册入口的可用性。

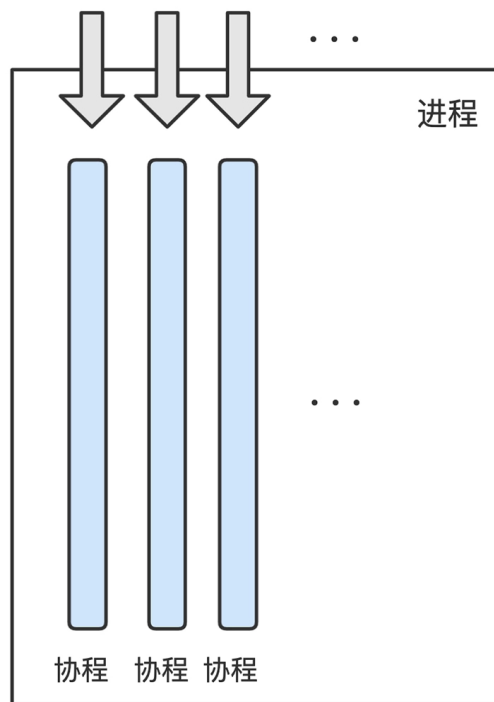
基本的中间件: Recovery

我们现在已经将中间件机制搭建并运行起来了，但是具体需要实现哪些中间件呢？这要根据不同需求进行不同的研发，是个长期话题。

这里我们演示一个最基本的中间件：Recovery。

中间件那么多，比如超时中间件、统计中间件、日志中间件，为什么我说 Recovery 是最基本的呢？给出我的想法之前，你可以先思考这个问题：在编写业务核心逻辑的时候，如果出现了一个 panic，而且在业务核心逻辑函数中未捕获处理，会发生什么？


我们还是基于第一节课讲的 net/http 的主流程逻辑来思考，关键结论有一点是，**每个 HTTP 连接都会开启一个 Goroutine 为其服务**，所以很明显，net/http 的进程模型是单进程、多协程。



在 Golang 的这种模型中，每个协程是独立且平等的，即使是创建子协程的父协程，在 Goroutine 中也无法管理子协程。所以，**每个协程需要自己保证不会外抛 panic**，一旦外抛 panic 了，整个进程就认为出现异常，会终止进程。


这一点搞清楚了，再看 Recovery 为什么必备就很简单。在 net/http 处理业务逻辑的协程中，要捕获在自己这个协程中抛出的 panic，就必须自己实现 Recovery 机制。

而 Recovery 中间件就是用来为每个协程增加 Recovery 机制的。我们在框架的 middleware 文件夹中增加 recovery.go 存放这个中间件：

 复制代码

```
1 // recovery机制，将协程中的函数异常进行捕获
2 func Recovery() framework.ControllerHandler {
3     // 使用函数回调
4     return func(c *framework.Context) error {
5         // 核心在增加这个recover机制，捕获c.Next()出现的panic
6         defer func() {
7             if err := recover(); err != nil {
8                 c.Json(500, err)
9             }
10        }()
11        // 使用next执行具体的业务逻辑
12        c.Next()
13
14        return nil
15    }
16 }
```

这个中间件就是在 `context.Next()` 之前设置了 `defer` 函数，这个函数的作用就是捕获 `c.Next()` 中抛出的异常 `panic`。之后在业务文件夹中的 `main.go`，我们就可以通过 `Core` 结构的 `Use` 方法，对所有的路由都设置这个中间件。

 复制代码

```
1 core.Use(middleware.Recovery())
```

今天所有代码的目录结构截图，我也贴在这里供你对比检查，代码放在 GitHub 上的 [🔗04分支](#)里。



```
🦉 context.go
🦉 controller.go
🦉 core.go
🦉 group.go
🦉 timeout.go
🦉 trie.go
🦉 trie_test.go
📄 .gitignore
🦉 controller.go
> 📄 go.mod
📄 LICENSE
🦉 main.go
📄 README.md
🦉 route.go
🦉 subject_controller.go
🦉 user_controller.go
```

小结

今天我们最终为自己的框架增加了中间件机制。中间件机制的本质就是装饰器模型，对核心的逻辑函数进行装饰、封装，所以一开始我们就使用函数嵌套的方式实现了中间件机制。

但是实现之后，我们发现函数嵌套的弊端：一是不优雅，二是无法批量设置中间件。所以我们引入了 **pipeline** 的思想，将所有中间件做成一个链条，通过这个链条的调用，来实现

中间件机制。

最后，我们选了最基础的 Recovery 中间件演示如何具体实现，一方面作为中间件机制的示例，另一方面，也在功能上为我们的框架增强了健壮性。

中间件机制是我们必须要掌握的机制，很多 Web 框架中都有这个逻辑。**在架构层面，中间件机制就相当于，在每个请求的横切面统一注入了一个逻辑。**这种统一处理的逻辑是非常有用的，比如统一打印日志、统一打点到统计系统、统一做权限登录验证等。

思考题

现在希望能对每个请求都进行请求时长统计，所以想写一个请求时长统计的中间件，在日志中输出请求 URI、请求耗时。不知道你如何实现呢？

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，也欢迎分享给你身边的朋友，邀请他一起学习~

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 3  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 03 | 路由：如何让请求更快寻找到目标函数？

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。