



下载APP

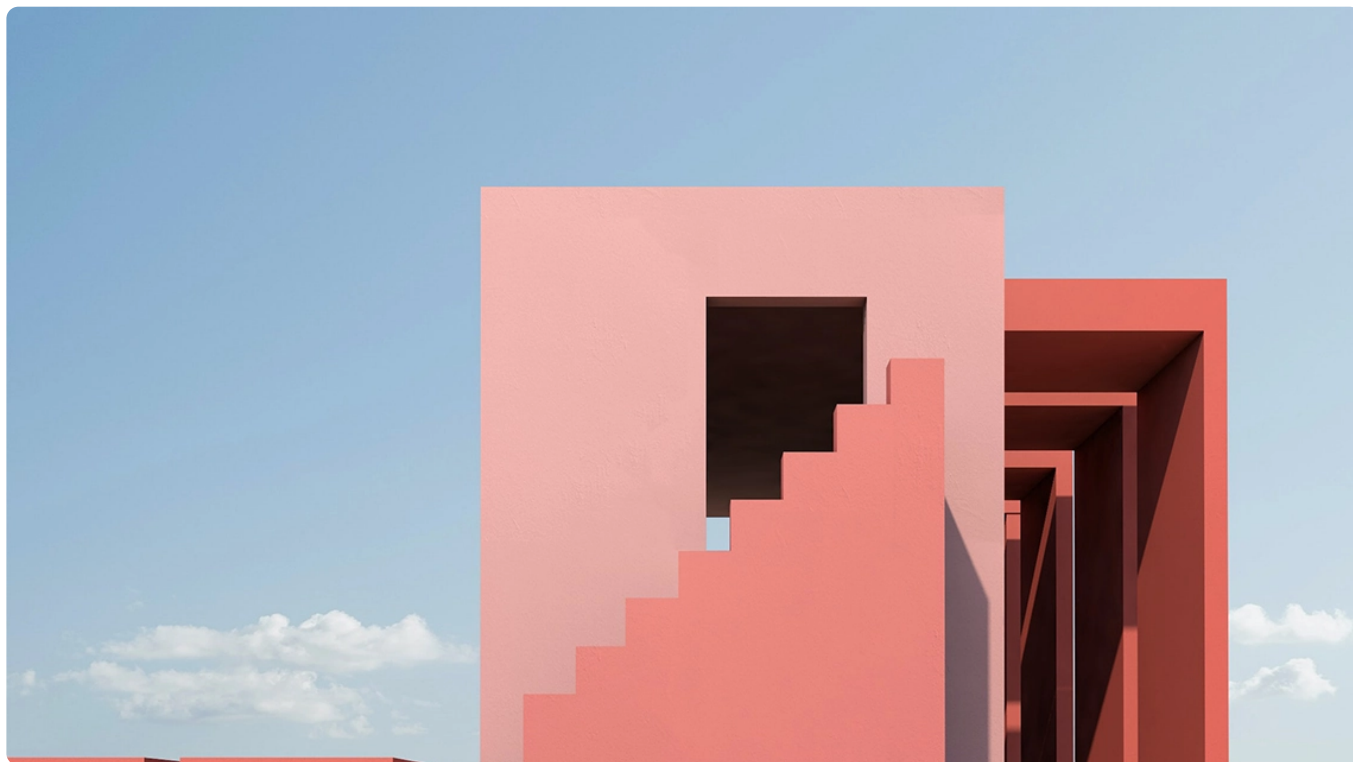


19 | 提效：实现调试模式加速开发效率（上）

2021-10-29 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 13:11 大小 12.08M



你好，我是轩脉刃。

上一节课我们把前端 Vue 融合进 hade 框架中，让框架能直接通过命令行启动包含前端和后端的一个应用，今天继续思考优化。

在使用 Vue 的时候，你一定使用过 `npm run dev` 这个命令，为前端开启调试模式，在这个模式下，只要你修改了 `src` 下的文件，编译器就会自动重新编译，并且更新浏览器页面上的渲染。这种调试模式，为开发者提高了不少开发效率。



那这种调试模式能否应用到 Golang 后端，让前后端都开启这种调试模式，来进一步提升我们开发应用的效率呢？接下来两节课，我们就来尝试实现这种调试模式。

方案思考和设计

先来思考下调试模式应该怎么设计？因为分为前端和后端，关于 Vue 前端，既然已经有了 `npm run dev` 这种调试模式，自然可以直接使用这种方式，要改的主要就是后端。

对于后端 Golang 代码，Golang 本身并没有提供任何调试模式的方式进行代码调试，**只能通过 `go build` 编译出二进制文件，通过运行二进制文件再启动服务**。那我们如何实现刚才的想法，一旦修改代码源文件，就能重新编译运行呢？

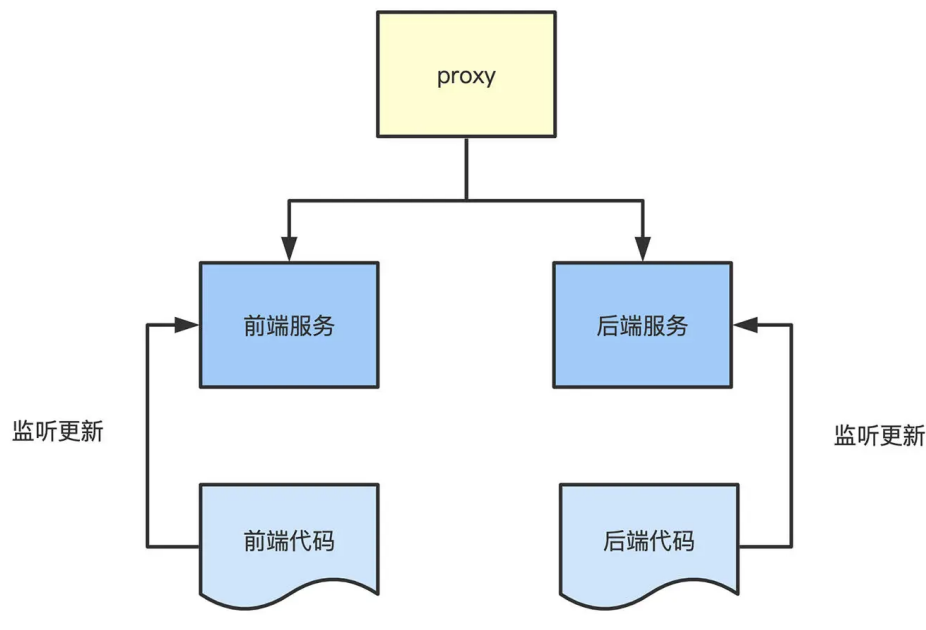
相信你一定很快想到了之前实现过配置文件的热更新。在第 16 章开发配置服务的时候，我们使用了 `fsnotify` 库，来对配置目录下的所有文件进行变更监听，一旦修改了配置目录下的文件，就重新更新内存中的配置文件 map。

那这里是否可以如法炮制，将 AppPath 目录下的文件也进行监听呢？一旦这个目录下的文件有了变更，就重新编译运行后端服务？

是的，原理可行，我们完全可以按照这种想法来构想一下。现在假设我们**监听了后端文件，能变更调试后端服务，也能通过 Vue 自带命令调试前端**，但这里又遇到难点了，如果需要前后端服务同时调试呢？

前端启动调试模式的方式和我们之前的编译方式完全不一样，它是直接启动一个端口来服务，并没有在 dist 下生成最终编译文件。这样，我们上一章设计的后端直接代理最终编译文件的方法就无法使用了。怎么办？

虽然过程不一样，但启动后的行为是差不多的。后端，实现了监听文件重新编译启动后，也是启动了一个进程来提供服务。思考到这里，自然而然，我们就想到**是否能在前端和后端服务的前面，设计一个反向代理 proxy 服务呢？**



让所有外部请求进入这个反向代理服务，然后由反向代理服务进行代理分发，前端请求分发到前端进程，后端请求分发到后端进程。

方案思路很流畅，我们来看如何实现。

实现技术难点分析

先攻坚最关键的技术难点，如何实现反向代理。

所谓反向代理，就是能将一个请求按照条件分发到不同的服务中去。在 Golang 中的 `net/http/httputil` 包中提供了 `ReverseProxy` 这么一个数据结构，它是实现整个反向代理的关键。

我们使用命令 `go doc net/http/httputil.ReverseProxy` 看下这个数据结构的定义，每个字段的说明我详细写在代码注释里面了：

复制代码

```
1 // 反向代理
2 type ReverseProxy struct {
3     // Director这个函数传入的参数是新复制的一个请求，我们可以修改这个请求
4     // 比如修改请求的请求Host或者请求URL等
5     Director func(*http.Request)
6 }
```

```
7 // Transport 代表底层的连接池设置，比如连接最长保持多久等
8 // 如果不填的话，则使用默认的设置
9 Transport http.RoundTripper
10
11 // FlushInterval表示多久将下游的response的数据拷贝到proxy的response
12 FlushInterval time.Duration
13
14 // ErrorLog 表示错误日志打印的句柄
15 ErrorLog *log.Logger
16
17 // BufferPool表示将下游response拷贝到proxy的response的时候使用的缓冲池大小
18 BufferPool BufferPool
19
20 // ModifyResponse 函数表示，如果要将下游的response内容进行修改，再传递给proxy
21 // 的response，这个函数就可以进行设置，但是如果这个函数返回了error，则将response
22 // 传递进入ErrorHandler，否则使用默认设置
23 ModifyResponse func(*http.Response) error
24
25 // ErrorHandler 处理ModifyResponse返回的Error
26 ErrorHandler func(http.ResponseWriter, *http.Request, error)
27 }
```

这里我着重解释一下这次会使用到的三个字段 Director、ModifyResponse、ErrorHandler，Director 是必须填写的，而 ModifyResponse、ErrorHandler 是可选的。

Director 的参数是请求，表示如何对请求进行转发。最简单的，我们可以修改请求的目标 Host，将请求转发到后端的服务。具体如何使用，可以看 net/http/httputil 库带的 NewSingleHostReverseProxy 方法，它将请求转发给后端 target 地址的时候，直接将 request 的 scheme、host、path 都进行了替换。这个方法也是后面我们经常要用到的。

[复制代码](#)

```
1 // 将原先的请求转发到target地址
2 func NewSingleHostReverseProxy(target *url.URL) *ReverseProxy {
3     targetQuery := target.RawQuery
4     // 设置director
5     director := func(req *http.Request) {
6         // 将原先的request替换scheme, host, path.
7         req.URL.Scheme = target.Scheme
8         req.URL.Host = target.Host
9         req.URL.Path, req.URL.RawPath = joinURLPath(target, req.URL)
10        ...
11    }
12    return &ReverseProxy{Director: director}
13 }
```

其次是 **ModifyResponse 字段**，在下游 response 要拷贝给上游 proxy 的 response 的时候，会使用到它代表的函数，如果我们要对下游的返回数据进行修改，就可以设置这个字段。

[复制代码](#)

```
1 ModifyResponse func(*http.Response) error
```

这个字段的参数就只有一个，http.Response 指针，代表的是下游返回给上游的返回结构，我们可以对这个指针的内容进行操作。而返回值 error，代表操作的结果，如果在操作过程中出现错误，会返回 error。

返回的 error，就会进入**第三个字段函数 ErrorHandler** 中。

[复制代码](#)

```
1 ErrorHandler func(http.ResponseWriter, *http.Request, error)
```

ErrorHandler 有三个参数，responseWriter 是新 proxy 的 response 的写句柄，request 是 Director 修改后给下游的 request，而 error 则是 ModifyResponse 处理后的 error。

了解清楚这三个字段函数中每个参数和返回值是非常重要的，这样才能准确地使用这些字段。下面我们就活学活用这个 ReverseProxy。

使用 ReverseProxy 作为反向代理，那么对应的路由规则是什么样的呢？什么样的请求进入后端，什么样的请求进入前端呢？这里我们需要再思考下。

还记得么，在上一节课增加前端代码 Vue 进入 hade 框架中的时候，我们使用了一个中间件 static，来将请求按照规则进行分发：如果请求地址在 dist 目录中存在，返回对应的请求文件，而如果请求地址在 dist 目录中不存在，就什么都不做，进行后续的路由规则判定。

但是在调试模式下，并没有前端编译环境，那我们怎么判断这个请求是进入前端，还是进入后端呢？这里是一个比较难的点。

可以反过来做。一个请求到了，直接先请求一下后端服务，如果后端发现请求不存在，返回 404 Not Found 之后，我们再将请求再请求到前端服务，就可以完美解决这个问题。这里用到刚才学习的 ReverseProxy 结构里面的 Director。

在 Director 中，将请求设置为转发给后端服务。这样当后端服务查找到路由不存在，返回 404 的时候，我们是能在 ModifyResponse 中获取到后端返回的 StatusCode 的。之后再判断如果为 404，让 ModifyResponse 返回一个自定义的 NotFoundErr。

一旦 ModifyResponse 返回了 Error，就会进入到 ErrorHandler 函数中，在这个函数中，我们判断一下参数中的 error 是否是之前定义的 NotFoundErr，如果是的话，就再用 NewSingleHostReverseProxy 来创建一个前端的 Proxy，将这个请求代理到前端服务中。

把这段实现的网关服务逻辑翻译成代码，在 framework/command/dev.go 中：

[复制代码](#)

```
1 // 重新启动一个proxy网关
2 func (p *Proxy) newProxyReverseProxy(frontend, backend *url.URL) *httputil.Rev
3 ...
4 // 先创建一个后端服务的directory
5 director := func(req *http.Request) {
6     req.URL.Scheme = backend.Scheme
7     req.URL.Host = backend.Host
8 }
9
10 // 定义一个NotFoundErr
11 NotFoundErr := errors.New("response is 404, need to redirect")
12
13 return &httputil.ReverseProxy{
14     Director: director, // 先转发到后端服务
15
16     ModifyResponse: func(response *http.Response) error {
17
18         // 如果后端服务返回了404，我们返回NotFoundErr 会进入到errorHandler中
19         if response.StatusCode == 404 {
20             return NotFoundErr
21         }
22         return nil
23     },
24
25     ErrorHandler: func(writer http.ResponseWriter, request *http.Request, err er
26
27     // 判断 Error 是否为NotFoundErr，是的话则进行前端服务的转发，重新修改writer
28     if errors.Is(err, NotFoundErr) {
```

```
29     httputil.NewSingleHostReverseProxy(frontend).ServeHTTP(writer, request)
30   }
31 }}
32 }
```

command 设计

思考清楚了技术难点，我们就可以开始设计命令了。这里为我们的框架重新定义一个 dev 一级命令，这个命令专门是调试模式，没有什么实际的作用，只是显示帮助信息。而它下面有三个二级命令：dev frontend 调试前端、dev backend 调试后端、dev all 前后端同时调试。

[复制代码](#)

```
1 ./hade dev //显示帮助信息
2 ./hade dev frontend // 调试前端
3 ./hade dev backend // 调试后端
4 ./hade dev all // 显示所有
```

在定义工具命令的时候，如果遇到有前端和后端的，我们应该统一在命令中使用关键字 frontend 和 backend 分别代表前后端，这样可以给使用者不断加深强调这两个关键字，这样我们在使用命令的时候，就能很快反应出前后端对应的命令了。

创建一个 framework/command/dev.go 来存放这个调试命令：

[复制代码](#)

```
1 // 初始化Dev命令
2 func initDevCommand() *cobra.Command {
3     devCommand.AddCommand(devBackendCommand)
4     devCommand.AddCommand(devFrontendCommand)
5     devCommand.AddCommand(devAllCommand)
6     return devCommand
7 }
8
9 // devCommand 为调试模式的一级命令
10 var devCommand = &cobra.Command{
11     Use:     "dev",
12     Short:   "调试模式",
13     RunE: func(c *cobra.Command, args []string) error {
14         c.Help()
15         return nil
16     },
17 }
```



```
18 }
19
20 // devBackendCommand 启动后端调试模式
21 var devBackendCommand = &cobra.Command{
22     Use:     "backend",
23     Short:   "启动后端调试模式",
24     RunE: func(c *cobra.Command, args []string) error {
25         ...
26     },
27 }
28
29 // devFrontendCommand 启动前端调试模式
30 var devFrontendCommand = &cobra.Command{
31     Use:     "frontend",
32     Short:   "前端调试模式",
33     RunE: func(c *cobra.Command, args []string) error {
34         ...
35     },
36 }
37
38 // 同时启动前端和后端调试
39 var devAllCommand = &cobra.Command{
40     Use:     "all",
41     Short:   "同时启动前端和后端调试",
42     RunE: func(c *cobra.Command, args []string) error {
43         ...
44     },
45 }
```

同时在 framework/command/kernel.go 中，我们加上 dev 的命令：

[复制代码](#)

```
1 // 框架核心命令
2 func AddKernelCommands(root *cobra.Command) {
3     ...
4     // dev 调试命令
5     root.AddCommand(initDevCommand())
6 }
```

proxy 类的设计

定义了 dev 命令的设计，我们再思考一下它如何实现。首先需要有一个结构来承担起调试模式所有的逻辑，这里定义为 Proxy 结构。proxy 结构和 proxy 结构对应的方法我们都存放在 framework/command/dev.go 中：


```
1 // Proxy 代表serve启动的服务器代理
2 type Proxy struct {
3     ...
4 }
```

[复制代码](#)

同时定义一个 NewProxy 方法来初始化这个 Proxy 结构：

```
1 func NewProxy(c framework.Container) *Proxy
```

[复制代码](#)

在初始化 proxy 的时候，需要容器中的一些服务，比如配置文件服务等，所以这里传递了一个容器的参数。

这个 proxy 结构应该有几个方法，按照代理分发的结构示意图，我们要**定义 proxy 服务需要的方法、前端服务需要的方法和后端服务需要的方法**。

针对 proxy 服务，首先需要定义我们在讲反向代理技术难点的时候提到的方法 newProxyReverseProxy，用来创建一个代理前后端的代理 ReverseProxy 结构。

```
1 func (p *Proxy) newProxyReverseProxy(frontend, backend *url.URL) *httputil.Rev
```

[复制代码](#)

其次，还需要一个启动 proxy 的方法 startProxy。它的传入参数就直接设置为两个 bool，代表是否要开启前端服务的代理、以及是否要开启后端服务的代理。

```
1 func (p *Proxy) startProxy(startFrontend, startBackend bool) error
```

[复制代码](#)

再来定义前后端服务的方法。明显要有一个方法能启动前端服务、也要有一个方法能启动后端服务：

```
1 func (p *Proxy) restartFrontend() error
2 func (p *Proxy) restartBackend() error
```

[复制代码](#)

这里注意一下，前端服务是直接使用 `npm run dev` 命令启动调试模式的，而后端服务是先进行 `go build` 再进行 `go run`，所以后端服务是需要进行编译的，所以我们还需要一个编译后端服务的方法：

[复制代码](#)

```
1 func (p *Proxy) rebuildBackend() error
```

同时，由于前端服务已经自己有了监控文件变更的逻辑，不需要我们再监控前端文件是否有变更了。而后端服务需要一个函数来监控源码文件的变更：

[复制代码](#)

```
1 func (p *Proxy) monitorBackend() error
```

这个监控文件我们设计为阻塞式的，在 `for` 循环中不断监控文件的变动，所以在调用的时候，如果不需要在这个函数中阻塞，可以开启一个 Goroutine 进行监听。


有了这些函数，我们就串联一下上面的 `command` 的设计。

首先前端调试模式，就非常简单，启动一个只带有前端的 `proxy` 就行：

[复制代码](#)


```
1 // devFrontendCommand 启动前端调试模式
2 var devFrontendCommand = &cobra.Command{
3     ...
4     RunE: func(c *cobra.Command, args []string) error {
5         // 启动前端服务
6         proxy := NewProxy(c.GetContainer())
7         return proxy.startProxy(true, false)
8     },
9 }
```

其次是后端调试模式，先启动一个 Goroutine 监听后端文件，再启动一个只有后端的 `proxy`：

 复制代码

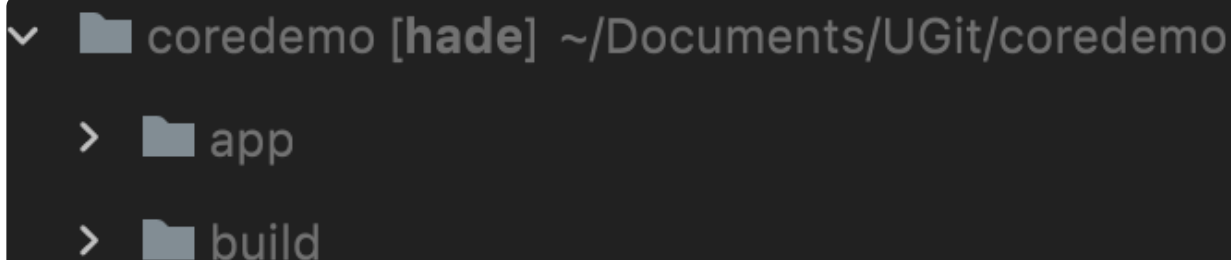
```
1 // devBackendCommand 启动后端调试模式
2 var devBackendCommand = &cobra.Command{
3     ...
4     RunE: func(c *cobra.Command, args []string) error {
5         proxy := NewProxy(c.GetContainer())
6         // 监听后端文件
7         go proxy.monitorBackend()
8         // 启动只有后端的proxy
9         if err := proxy.startProxy(false, true); err != nil {
10             return err
11         }
12         return nil
13     },
14 }
```

而前后端同时调试，则是先启动一个 Goroutine 监听后端文件，再同时启动监听前后端的 proxy：

 复制代码

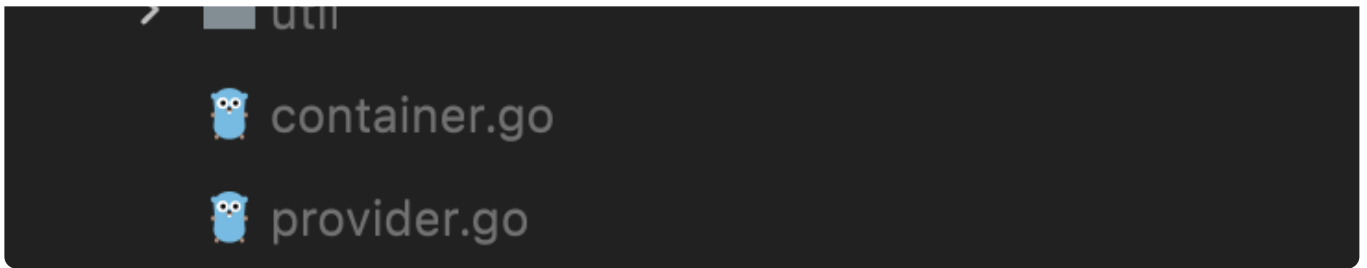
```
1 var devAllCommand = &cobra.Command{
2     ...
3     RunE: func(c *cobra.Command, args []string) error {
4         proxy := NewProxy(c.GetContainer())
5         // 监听后端文件
6         go proxy.monitorBackend()
7         // 启动前后端同时监听的proxy
8         if err := proxy.startProxy(true, true); err != nil {
9             return err
10        }
11        return nil
12    },
13 }
```

今天只在 framework/command/ 目录下增加了一个 dev.go 文件，代码地址在 [geekbang/19](https://github.com/geekbang/geekbang/tree/master/framework/command) 分支上。下节课我们继续完成调试模式的具体实现。



```
✓ core demo [hade] ~/Documents/UGit/core demo
> app
> build
```

```
> config
> dist
v framework
  > cobra
  v command
    owl app.go
    owl build.go
    owl config.go
    owl cron.go
    owl dev.go
    owl env.go
    owl go_cmd.go
    owl go_cmd_test.go
    owl help.go
    owl kernel.go
    owl npm.go
  > contract
  > gin
  > middleware
  > provider
  > util
```



小结

今天这节课最关键的点就在于 ReverseProxy 的运用。ReverseProxy 是 Golang 标准库提供的反向代理的实现方式。而反向代理，在实际业务开发过程中实际上是非常好用的。

比如我们在业务开发过程中很有可能会需要自研网关，来全局代理和监控所有的后端接口；又或者在拆分微服务的时候，需要有一个统一路由层来引导流量。这个 ReverseProxy 结构的熟练使用就是这些功能的核心关键。

今天我们为 hade 框架增加了调试模式，这个模式在很多 Golang 的框架中是没有的，算是我们的 hade 框架的一大特色了。大多数框架是依赖于日志进行编译调试。而 hade 框架之所以能提供这种方便的调试模式，也是依赖于我们前面已经实现的前后端一体、目录服务，配置服务等逻辑。

在实际工作中，特别在调试的时候，这种调试模式一定能为你带来很大的便利。

思考题

讲 ReverseProxy 时，我们的逻辑是先请求后端服务，如果后端服务出现 404，再请求前端。这里有两个问题你可以思考下：

1. 可不可以先请求 vue 的前端服务，如果前端服务出现 404，再请求后端呢？
2. 某些 vue 确定的请求地址，比如 `/app.js`，`/"`，是否可以不用走这个先后端服务、再前端服务的逻辑？如果可以，怎么做呢？

欢迎在留言区分享你的思考。感谢你的收听，如果觉得有收获，也欢迎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 0  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 18 | 一体化：前端和后端一定要项目分开吗？

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。