



下载APP



20 | 提效：实现调试模式加速开发效率（下）

2021-11-01 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 15:43 大小 14.40M



你好，我是轩脉刃。

上一节课，我们讨论了调试模式的整体设计思路和关键的技术难点 - 反向代理，最后定义了具体的命令设计，包括三个二级命令，能让我们调试前端 / 后端，或者同时调试。现在，大的框架都建立好了，但是其中的细节实现还没有讨论。成败在于细节，今天我们就撸起袖子开始实现它们。

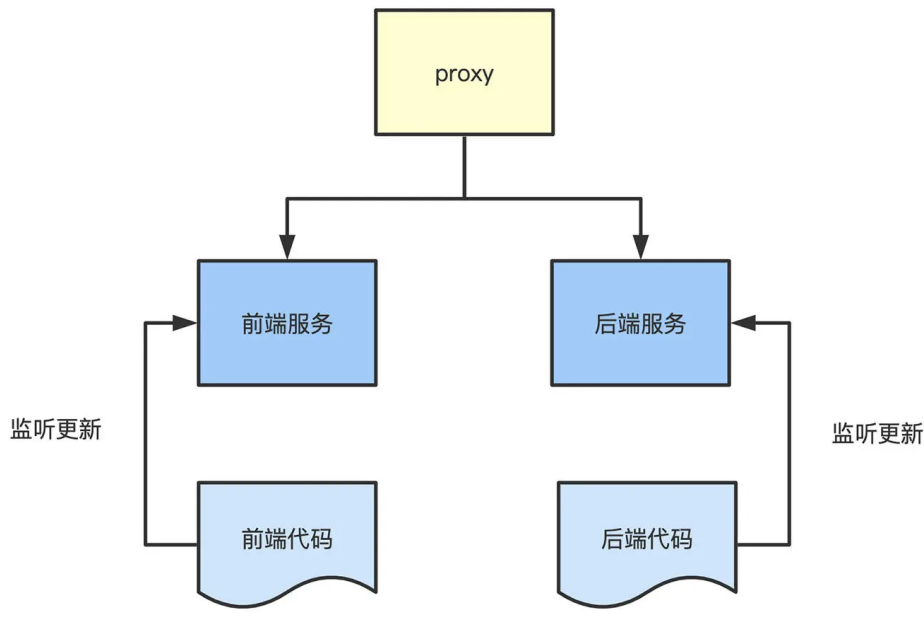
领资料



配置项的设计



简单回顾一下调试模式的架构设计。所有外部请求进入反向代理服务后，会由反向代理服务进行分发，前端请求分发到前端进程，后端请求分发到后端进程。



在这个设计中，前端服务启动的时候占用哪个端口？后端服务启动的时候占用哪个端口？反向代理服务 proxy 启动的时候占用哪个端口呢？这些都属于配置项，需要在设计之初就规划好，所以我们先设计配置项的具体实现。

由于调试模式配置项比较多，在 `framework/command/dev.go` 中，我们定义如下的配置结构 `devConfig` 来表示配置信息：

[复制代码](#)

```
1 // devConfig 代表调试模式的配置信息
2 type devConfig struct {
3
4     Port    string // 调试模式最终监听的端口，默认为8070
5
6     Backend struct { // 后端调试模式配置
7         RefreshTime int // 调试模式后端更新时间，如果文件变更，等待3s才进行一次更新，
8         Port        string // 后端监听端口，默认 8072
9         MonitorFolder string // 监听文件夹，默认为AppFolder
10    }
11
12    Frontend struct { // 前端调试模式配置
13        Port string // 前端启动端口，默认8071
14    }
15 }
```

这个结构可以说已经非常清晰了。结构根目录下的 Port 代表 proxy 的端口，而根目录下的 Backend 和 Frontend 分别代表后端和前端的配置。

其中，前端只需要配置一个端口 Port，而后端，我们除了配置端口 Port 之外，还另外多了两个配置，一个是监听的文件夹 MonitorFolder，另外一个为监听文件夹的变更时间 RefreshTime，这两个配置都是和后端监听文件夹相关的，具体如何使用，我们在后面写 proxy 的方法 monitorBackend 再详细说。

有了这个配置结构还不够，我们还要定义配置结构中每个值的赋值和默认值，在配置文件 app.yaml 中对应定义的配置字段如下：

[复制代码](#)

```
1 dev: # 调试模式
2   port: 8070 # 调试模式最终监听的端口，默认为8070
3   backend: # 后端调试模式配置
4     refresh_time: 3 # 调试模式后端更新时间，如果文件变更，等待3s才进行一次更新，能让频繁
5     port: 8072 # 后端监听端口，默认8072
6     monitor_folder: "" # 监听文件夹地址，为空或者不填默认为AppFolder
7   frontend: # 前端调试模式配置
8     port: 8071 # 前端监听端口，默认8071
```

之后如果在配置文件中有配置这些字段，就使用配置文件中的字段，否则的话，则使用默认配置。对应到代码上，我们可以在 framework/command/dev.go 中实现一个 initDevConfig。

实现思路也不难，参数只需要把服务容器传递进去就行了，在这个函数中，我们先定义好默认的配置，然后从容器中获取配置服务，通过配置服务，获取对应的配置文件的设置，如果配置文件有对应字段的话，就进行对应字段的配置。


[复制代码](#)

```
1 // 初始化配置文件
2 func initDevConfig(c framework.Container) *devConfig {
3   // 设置默认值
4   devConfig := &devConfig{
5     Port: "8087",
6     Backend: struct {
7       RefreshTime int
8       Port string
9       MonitorFolder string
```

```
10     }{
11         1,
12         "8072",
13         "",
14     },
15     Frontend: struct {
16         Port string
17     }{
18         "8071",
19     },
20 }
21 // 容器中获取配置服务
22 configer := c.MustMake(contract.ConfigKey).(contract.Config)
23 // 每个配置项进行检查
24 if configer.IsExist("app.dev.port") {
25     devConfig.Port = configer.GetString("app.dev.port")
26 }
27 if configer.IsExist("app.dev.backend.refresh_time") {
28     devConfig.Backend.RefreshTime = configer.GetInt("app.dev.backend.refre
29 }
30 if configer.IsExist("app.dev.backend.port") {
31     devConfig.Backend.Port = configer.GetString("app.dev.backend.port")
32 }
33
34 // monitorFolder 默认使用目录服务的AppFolder()
35 monitorFolder := configer.GetString("app.dev.backend.monitor_folder")
36 if monitorFolder == "" {
37     appService := c.MustMake(contract.AppKey).(contract.App)
38     devConfig.Backend.MonitorFolder = appService.AppFolder()
39 }
40 if configer.IsExist("app.dev.frontend.port") {
41     devConfig.Frontend.Port = configer.GetString("app.dev.frontend.port")
42 }
43 return devConfig
44 }
45
```


这里着重说一下 monitorFolder 这个配置的逻辑，如果配置文件中有定义这个配置的话，我们就使用配置文件的配置，否则我们就去目录服务中获取 AppFolder。其实这种有层次的配置方式，在配置服务那一节我们已经见过了，多使用这种配置方式能让框架可用性更高。

但是之前 [第 12 节课](#)，定义目录服务接口的时候，没有定义 App 的服务接口，所以我们得去稍微修改下目录服务接口 framework/contract/app.go，为其增加 AppFolder 这个目录接口：

 复制代码

```
1 // App 定义接口
2 type App interface {
3     ...
4
5     // AppFolder 定义业务代码所在的目录，用于监控文件变更使用
6     AppFolder() string
7     ...
8 }
```

同时修改其对应实现 framework/provider/app/service.go，增加这个 AppFolder 的实现：

 复制代码

```
1 // AppFolder 代表app目录
2 func (app *HadeApp) AppFolder() string {
3     if val, ok := app.configMap["app_folder"]; ok {
4         return val
5     }
6     return filepath.Join(app.BaseFolder(), "app")
7 }
```


到这里，配置结构 devConfig 及配置结构初始化方法 initDevConfig，就实现完成了。

具体实现

现在，来完成拼图的最后一个部分，回到 framework/command/dev.go 中，上节课只定义了 Proxy 结构，但是 Proxy 结构中的字段，我们没有讨论。

首先有了上面定义的 devConfig 结构之后，Proxy 的结构中，应该有一个字段保存这个 Proxy 的配置信息 devConfig。

其次，在 restart 前端或者后端的时候，由于**新进程和旧进程都使用一样的端口**，我们一定是先关闭旧的前端进程或者后端进程，才能启动新的前端或者后端进程。所以这里要记录一下前后端进程的进程 ID，设置了 backendPid 和 frontendPid 来存储进程 ID。

 复制代码

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

```
4    devConfig    *devConfig // 配置文件
5    backendPid   int         // 当前的backend服务的pid
6    frontendPid  int         // 当前的frontend服务的pid
    ,
```

下面我们就针对每个函数的具体实现——说明，这里把上节课定义的各个函数简单再列一下，如果你对它们的功能有点模糊了，可以再回顾一下第 19 课。

[复制代码](#)

```
1 // 初始化一个Proxy
2 func NewProxy(c framework.Container) *Proxy{}
3 // 重新启动一个proxy网关
4 func (p *Proxy) newProxyReverseProxy(frontend, backend *url.URL) *httputil.ReverseProxy {
5 // 启动前端服务
6 func (p *Proxy) restartFrontend() error{}
7 // 启动后端服务
8 func (p *Proxy) restartBackend() error {}
9 // 编译后端服务
10 func (p *Proxy) rebuildBackend() error {}
11 // 启动proxy
12 func (p *Proxy) startProxy(startFrontend, startBackend bool) error{}
13 // 监控后端服务源码文件的变更
14 func (p *Proxy) monitorBackend() error{}
```

newProxyReverseProxy

首先是 newProxyReverseProxy，它的核心逻辑就是创建 ReverseProxy，设置 Director、ModifyResponse、ErrorHandler 三个字段。但是我们在细节上要做一些补充。

首先，既然已经在 proxy 中存了前后端的 PID，那就可以知道当下前端服务或者后端服务是否已经启动了。如果只启动了前端服务，我们直接代理前端就好了；如果只启动后端服务，就直接代理后端。而**只有两个服务都启动了，我们才进行上一节课说的：先请求后端服务，遇到 404 了，再请求前端服务。**

同时稍微修改一下 director，对于前端一些固定的请求地址，比如 / 或者 /app.js，我们直接将这个地址固定请求前端。

[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     if p.frontendPid == 0 && p.backendPid == 0 {
4         fmt.Println("前端和后端服务都不存在")
5         return nil
6     }
7
8     // 后端服务存在
9     if p.frontendPid == 0 && p.backendPid != 0 {
10         return httputil.NewSingleHostReverseProxy(backend)
11     }
12
13     // 前端服务存在
14     if p.backendPid == 0 && p.frontendPid != 0 {
15         return httputil.NewSingleHostReverseProxy(frontend)
16     }
17
18     // 两个都有进程
19     // 先创建一个后端服务的directory
20     director := func(req *http.Request) {
21         if req.URL.Path == "/" || req.URL.Path == "/app.js" {
22             req.URL.Scheme = frontend.Scheme
23             req.URL.Host = frontend.Host
24         } else {
25             req.URL.Scheme = backend.Scheme
26             req.URL.Host = backend.Host
27         }
28     }
29
30     // 定义一个NotFoundErr
31     NotFoundErr := errors.New("response is 404, need to redirect")
32     return &httputil.ReverseProxy{
33         Director: director, // 先转发到后端服务
34         ModifyResponse: func(response *http.Response) error {
35             // 如果后端服务返回了404，我们返回NotFoundErr 会进入到errorHandler中
36             if response.StatusCode == 404 {
37                 return NotFoundErr
38             }
39             return nil
40         },
41         ErrorHandler: func(writer http.ResponseWriter, request *http.Request, er
42             // 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
43             if errors.Is(err, NotFoundErr) {
44                 httputil.NewSingleHostReverseProxy(frontend).ServeHTTP(writer, req
45             }
46         }}
47 }
```

rebuildBackend / restartBackend

下一个函数是 rebuildBackend。这个函数的作用是重新编译后端。

那如何编译后端呢？还记得第 18 课中为编译后端定义了命令行么？所以在“调试命令”中，我们只需要调用“编译命令”就行了。

编译前端 `./hade build frontend`

编译后端 `./hade build backend`

同时编译前后端 `./hade build all`

自编译 `./hade build self`

所以 `rebuildBackend` 这个函数，我们就是调用一次 `./hade build backend`。

 复制代码

```
1 // rebuildBackend 重新编译后端
2 func (p *Proxy) rebuildBackend() error {
3     // 重新编译hade
4     cmdBuild := exec.Command("./hade", "build", "backend")
5     cmdBuild.Stdout = os.Stdout
6     cmdBuild.Stderr = os.Stderr
7     if err := cmdBuild.Start(); err == nil {
8         err = cmdBuild.Wait()
9         if err != nil {
10             return err
11         }
12     }
13     return nil
14 }
```

编译后端函数实现了，下面就是重启后端进程 `restartBackend`。


我们当然也会记得在 [第 12 章](#) 将启动 Web 服务变成一个命令 `./hade app start`。所以重启后端服务的步骤就是：

关闭旧进程（kill）

启动新进程（`./hade app start`）

但是这里有个小问题，之前启动进程的时候，进程端口是写死的。但是，现在需要固定启动的 App 的进程端口。所以要对 `./hade app start` 命令进行一些改造。


来修改 `framework/command/app.go`，我们增加一个 `appAddress` 地址，这个地址可以传递类似 `localhost:8888` 或者 `:8888` 这样的启动服务地址，并且在 `appStartCommand` 中使用这个 `appAddress`。

 复制代码

```
1 // app启动地址
2 var appAddress = ""
3
4 // initAppCommand 初始化app命令和其子命令
5 func initAppCommand() *cobra.Command {
6     // 设置启动地址
7     appStartCommand.Flags().StringVar(&appAddress, "address", ":8888", "设置appf
8
9     appCommand.AddCommand(appStartCommand)
10    return appCommand
11 }
12
13 // appStartCommand 启动一个Web服务
14 var appStartCommand = &cobra.Command{
15     Use:     "start",
16     Short:   "启动一个Web服务",
17     RunE:    func(c *cobra.Command, args []string) error {
18         ...
19         // 创建一个Server服务
20         server := &http.Server{
21             Handler: core,
22             Addr:     appAddress,
23         }
24         // 这个goroutine是启动服务的goroutine
25         go func() {
26             server.ListenAndServe()
27         }()
28         ...
29     },
30 }
```

这样，后端进程就可以通过命令 `./hade app start --address=:8888` 这样的方式，来指定端口启动服务了。

小问题解决之后，回到 `framework/command/dev.go`，我们实现 `restartBackend` 方法。先杀死旧的进程，再通过命令 `./hade app start` 带上参数 `address`，启动新的后端服务。启动之后，再将启动的进程 ID 存储到 `proxy` 结构的 `backendPid` 字段中：

 复制代码

```
1 // restartBackend 启动后端服务
2 func (p *Proxy) restartBackend() error {
3
4     // 杀死之前的进程
5     if p.backendPid != 0 {
6         syscall.Kill(p.backendPid, syscall.SIGKILL)
7         p.backendPid = 0
8     }
9
10    // 设置随机端口，真实后端的端口
11    port := p.devConfig.Backend.Port
12    hadeAddress := fmt.Sprintf(":" + port)
13    // 使用命令行启动后端进程
14    cmd := exec.Command("./hade", "app", "start", "--address="+hadeAddress)
15    cmd.Stdout = os.NewFile(0, os.DevNull)
16    cmd.Stderr = os.Stderr
17    fmt.Println("启动后端服务: ", "http://127.0.0.1:"+port)
18    err := cmd.Start()
19    if err != nil {
20        fmt.Println(err)
21    }
22    p.backendPid = cmd.Process.Pid
23    fmt.Println("后端服务pid:", p.backendPid)
24    return nil
25 }
```

restartFrontend


而重启前端服务的函数 `restartFrontend` 也是一样的逻辑，先关闭旧的前端进程，然后启动新的前端进程。这里同样也有一个问题，启动前端进程的命令是 `npm run dev`，我们怎么固定其端口呢？

在 Vue 中，我们可以通过  设置环境变量 `PORT`，来规定前端进程的启动端口。也就是让启动命令变为 `PORT=8071 npm run dev`，在 Golang 中启动一个命令，并为命令设置环境变量是这样设置的：

 复制代码

```
1 // 运行命令
2 cmd := exec.Command("npm", "run", "dev")
3 // 为默认的环境变量增加PORT=xxx的变量
4 cmd.Env = os.Environ()
5 cmd.Env = append(cmd.Env, fmt.Sprintf("%s%s", "PORT=", port))
```

所以启动前端服务的逻辑就如下，很简单，重点位置你可以看注释。


 复制代码

```
1 // 启动前端服务
2 func (p *Proxy) restartFrontend() error {
3     // 启动前端调试模式
4     // 先杀死旧进程
5     if p.frontendPid != 0 {
6         syscall.Kill(p.frontendPid, syscall.SIGKILL)
7         p.frontendPid = 0
8     }
9     // 否则开启npm run serve
10    port := p.devConfig.Frontend.Port
11    path, err := exec.LookPath("npm")
12    if err != nil {
13        return err
14    }
15    cmd := exec.Command(path, "run", "dev")
16    cmd.Env = os.Environ()
17    cmd.Env = append(cmd.Env, fmt.Sprintf("%s%s", "PORT=", port))
18    cmd.Stdout = os.NewFile(0, os.DevNull)
19    cmd.Stderr = os.Stderr
20    // 因为npm run serve 是控制台挂起模式，所以这里使用go routine启动
21    err = cmd.Start()
22    fmt.Println("启动前端服务：", "http://127.0.0.1:"+port)
23    if err != nil {
24        fmt.Println(err)
25    }
26    p.frontendPid = cmd.Process.Pid
27    fmt.Println("前端服务pid:", p.frontendPid)
28    return nil
29 }
```

startProxy

下面我们来实现 startProxy 方法，它有两个参数，表示在启动 Proxy 时是否要启动前端、后端服务。

这个方法的逻辑也并不复杂，步骤有第四步，先根据参数判断是否启动后端服务，根据参数判断是否启动前端服务，然后使用 newProxyReverseProxy 来创建新的 ReverseProxy，最后启动 Proxy 服务。在代码中也做了步骤说明了：

 复制代码


```
1 // 启动proxy服务，并且根据参数启动前端服务或者后端服务
```

```
2 func (p *Proxy) startProxy(startFrontend, startBackend bool) error {
3     var backendURL, frontendURL *url.URL
4     var err error
5
6     // 启动后端
7     if startBackend {
8         if err := p.restartBackend(); err != nil {
9             return err
10        }
11    }
12    // 启动前端
13    if startFrontend {
14        if err := p.restartFrontend(); err != nil {
15            return err
16        }
17    }
18
19    if frontendURL, err = url.Parse(fmt.Sprintf("%s%s", "http://127.0.0.1:", p.
20        return err
21    }
22
23    if backendURL, err = url.Parse(fmt.Sprintf("%s%s", "http://127.0.0.1:", p.d
24        return err
25    }
26
27    // 设置反向代理
28    proxyReverse := p.newProxyReverseProxy(frontendURL, backendURL)
29    proxyServer := &http.Server{
30        Addr:    "127.0.0.1:" + p.devConfig.Port,
31        Handler: proxyReverse,
32    }
33
34    fmt.Println("代理服务启动:", "http://" + proxyServer.Addr)
35    // 启动proxy服务
36    err = proxyServer.ListenAndServe()
37    if err != nil {
38        fmt.Println(err)
39    }
40    return nil
41 }
```

monitorBackend

最后是一个 monitorBackend 方法，监控某个文件夹的变动，并且重新编译并且运行后端服务。

这个方法我们重点说一下，有些逻辑还是比较绕的。

首先，在前一节课说过了，可以使用  **fsnotify** 库对目录进行监控。那么对哪个目录进行监控呢？之前在配置 `devConfig` 中，定义了一个 `Backend.MonitorFolder` 目录，这个配置默认使用的是 `AppFolder` 目录。这个就是我们监控的目标目录。

其次，每次有变化的时候，都要进行一次编译后端服务、杀死旧进程、重启新进程么？

在开发过程中我们知道，每次调整一个逻辑的时候，是有可能短时间内重复修改、保存多个文件的，或者保存一个文件多次。而重新编译、重新启动进程的过程，又是有一定耗时的，如果每改一次就重来一次，可以想象这个体验是很差的。

能怎么优化这种体验呢？我们可以使用一种计时时间机制。

这个机制的逻辑就是，**每次有文件变动，并不立刻进行实质的操作，而是开启一个计时时间**，如果这个时间内，没有任何后续的文件变动了，那么在计时时间到了之后，我们再进行实质的操作。而如果在计时时间内，有任何更新的文件变动，我们就将计时时间机制重新开始计时。

这种机制能有一定概率保证，在“更新代码等待一段时间后”进行后端的重启服务。而这里的计时时间我们也变成一个配置，`devConfig` 里面的 `Backend.RefreshTime`，默认时长为 1s。

对应 `framework/command/dev.go` 的 `monitorBackend` 代码实现中，我们大致分为这么几步，**先创建 watcher，监听目标目录，有变动的时候开启计时时间机制，循环监听**：

目标目录变更事件，有事件更新计时机制；

计时机制到点事件，计时到点事件触发，代表有一个或多个目标目录变更已经存在，更新后端服务。

这里在监听目标目录的时候，我们需要监听 `AppFolder` 目录下的所有子目录及孙目录，所以这里需要用到递归 `filepath.Walk`，来递归一遍所有子目录及孙目录。如果是目录，就使用 `watcher.Add` 来将目录加入到监控列表中。

具体的代码逻辑可以看 `framework/command/dev.go` 中的 `monitorBackend`：

```
1 // monitorBackend 监听应用文件
2 func (p *Proxy) monitorBackend() error {
3     // 监听
4     watcher, err := fsnotify.NewWatcher()
5     if err != nil {
6         return err
7     }
8     defer watcher.Close()
9
10    // 开启监听目标文件夹
11    appFolder := p.devConfig.Backend.MonitorFolder
12    fmt.Println("监控文件夹:", appFolder)
13    // 监听所有子目录, 需要使用filepath.walk
14    filepath.Walk(appFolder, func(path string, info os.FileInfo, err error) error {
15        if info != nil && !info.IsDir() {
16            return nil
17        }
18        // 如果是隐藏的目录比如 . 或者 .. 则不用进行监控
19        if util.IsHiddenDirectory(path) {
20            return nil
21        }
22        return watcher.Add(path)
23    })
24
25    // 开启计时时间机制
26    refreshTime := p.devConfig.Backend.RefreshTime
27    t := time.NewTimer(time.Duration(refreshTime) * time.Second)
28    // 先停止计时器
29    t.Stop()
30    for {
31        select {
32        case <-t.C:
33            // 计时器时间到了, 代表之前有文件更新事件重置过计时器
34            // 即有文件更新
35            fmt.Println("...检测到文件更新, 重启服务开始...")
36            if err := p.rebuildBackend(); err != nil {
37                fmt.Println("重新编译失败:", err.Error())
38            } else {
39                if err := p.restartBackend(); err != nil {
40                    fmt.Println("重新启动失败:", err.Error())
41                }
42            }
43            fmt.Println("...检测到文件更新, 重启服务结束...")
44            // 停止计时器
45            t.Stop()
46        case _, ok := <-watcher.Events:
47            if !ok {
48                continue
49            }
50            // 有文件更新事件, 重置计时器
```



```
51         t.Reset(time.Duration(refreshTime) * time.Second)
52     case err, ok := <-watcher.Errors:
53         if !ok {
54             continue
55         }
56         // 如果有文件监听错误，则停止计时器
57         fmt.Println("监听文件夹错误：", err.Error())
58         t.Reset(time.Duration(refreshTime) * time.Second)
59     }
60 }
61 }
```

验证

到这里 Proxy 相关的逻辑和调试对应的命令行工具都开发完成了，下面我们来做一下对应的验证，一共三次验证，单独的前端、后端修改，以及同时对前后端的修改。

先修改一下 config/development/app.yaml，增加对应的调试模式配置：

[复制代码](#)

```
1 dev: # 调试模式
2   port: 8070 # 调试模式最终监听的端口，默认为8070
3   backend: # 后端调试模式配置
4     refresh_time: 3 # 调试模式后端更新时间，如果文件变更，等待3s才进行一次更新，能让频繁
5     port: 8072 # 后端监听端口，默认8072
6     monitor_folder: "" # 监听文件夹地址，为空或者不填默认为AppFolder
7   frontend: # 前端调试模式配置
8     port: 8071 # 前端监听端口，默认8071
```

这里设置 refresh_time 为 3s，代表后续后端变更后 3s 后会触发重新编译。对我们的代码进行一次编译，不用 go build 了，可以使用自定义的 build 命令了。

```
~/Documents/UGit/coredemo ➜ geekbang/20 ●+ ➜ ./hade build self
编译 hade 成功
```

前端验证

首先验证前端调试模式。调用命令 ./hade dev front，可以看到如下的控制台信息：

```
~/Documents/UGit/coredemo > P geekbang/20 ➤ ./hade dev frontend
启动前端服务: http://127.0.0.1:8071
前端服务pid: 13750
代理服务启动: http://127.0.0.1:8070

> hade@1.0.0 dev /Users/yejianfeng/Documents/UGit/coredemo
> webpack-dev-server --inline --progress --config build/webpack.dev.conf.js

13% building modules 29/31 modules 2 active ...g/Documents/UGit/coredemo/src/App.vue{ parser: "babel" } is deprecated; we now treat it as { parser: "babel" }.
95% emitting
DONE Compiled successfully in 2444ms

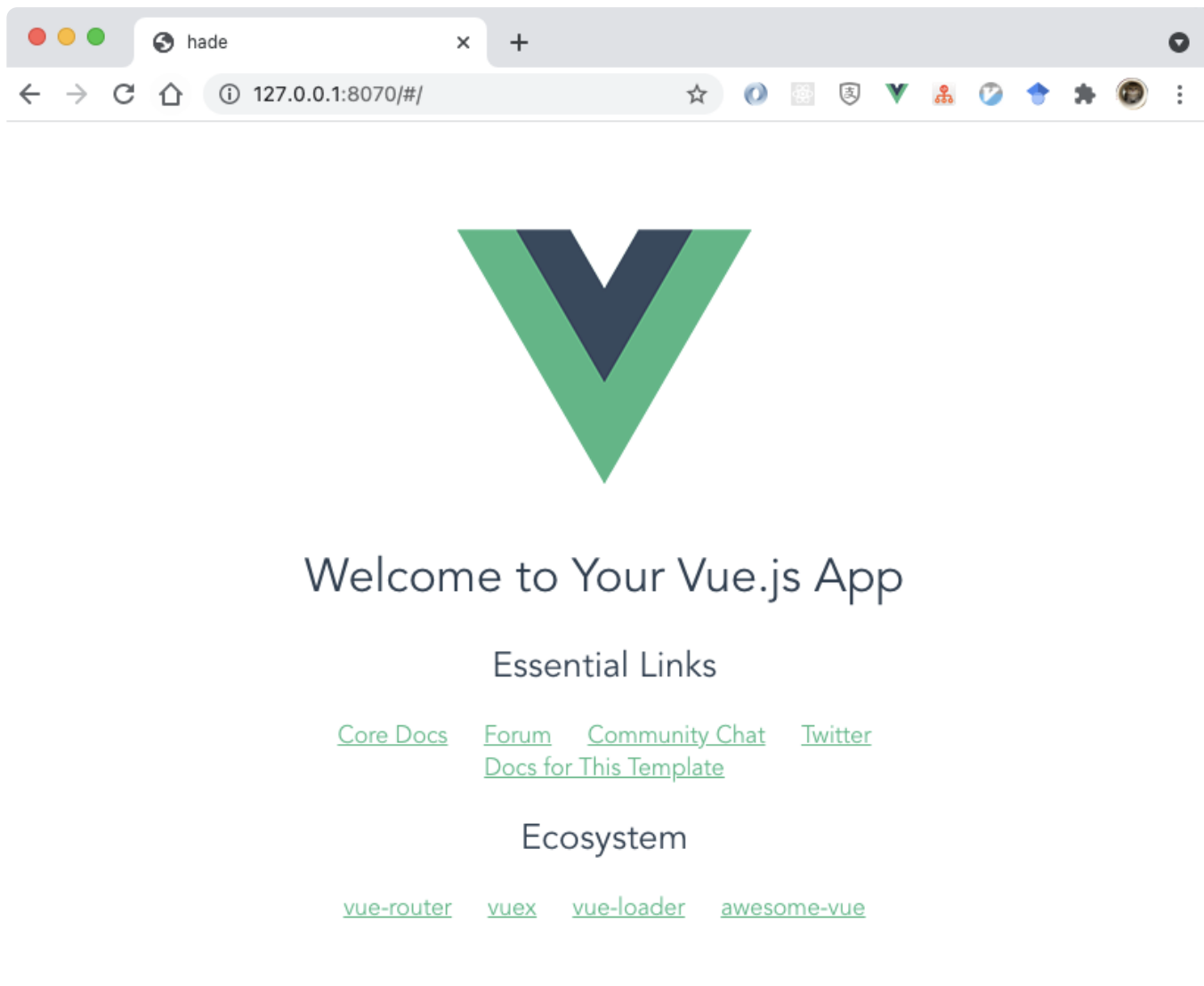
Your application is running here: http://localhost:8071
```

先是出现几行信息：


复制代码

- 1 启动前端服务： http://127.0.0.1:8071
- 2 前端服务pid: 13750
- 3 代理服务启动： http://127.0.0.1:8070

然后进入到了 Vue 的调试模式，从上述信息我们知道，代理服务启动在 8070 端口，使用浏览器打开 <http://127.0.0.1:8070> 看到了熟悉的 Vue 界面。




然后修改首页的前端组件，业务目录下 `src/components/HelloWorld.vue`，将其展示在首页的 `msg` 内容：

 复制代码

```
1 <script>
2 export default {
3   name: 'HelloWorld',
4   data() {
5     return {
6       msg: 'Welcome to Your Vue.js App '
7     }
8   }
9 }
10 </script>
```

修改为：

 复制代码

```
1 <script>
2 export default {
3   name: 'HelloWorld',
4   data() {
5     return {
6       msg: 'Welcome to Hade Vue.js App '
7     }
8   }
9 }
10 </script>
```

现在你可以看到，前端自动更新：



Welcome to Hade Vue.js App

Essential Links

[Core Docs](#) [Forum](#) [Community Chat](#) [Twitter](#)
[Docs for This Template](#)

Ecosystem

[vue-router](#) [vuex](#) [vue-loader](#) [awesome-vue](#)

前端验证完成。下面验证后端调试模式。

后端验证

我们已经在业务代码 `app/http/module/demo/api.go` 中，定义了 `/demo/demo` 的路由，并且简单输出文字 `"this is demo"`。

复制代码

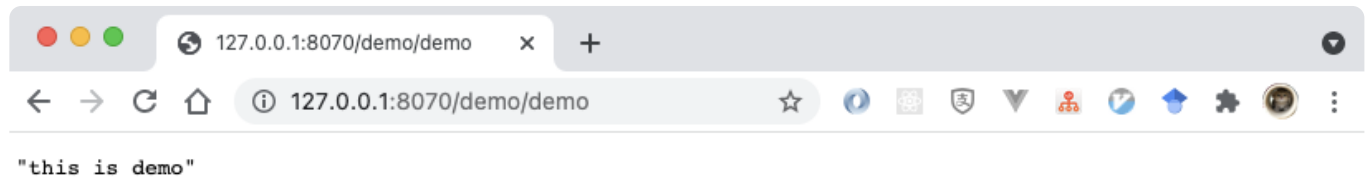
```
1 func Register(r *gin.Engine) error {
2     api := NewDemoApi()
3     ...
4     r.GET("/demo/demo", api.Demo)
5     ...
6     return nil
7 }
8
9 func (api *DemoApi) Demo(c *gin.Context) {
10    c.JSON(200, "this is demo")
}
```

```
11 }
```

使用命令 `./hade dev backend`，有如下输出，可以看到输出中已经把监控文件夹、后端服务端口、代理服务端口完整输出了：

```
~/Documents/UGit/coredemo geekbang/20 ➤ ./hade dev backend
启动后端服务： http://127.0.0.1:8072
监控文件夹： /Users/yejianfeng/Documents/UGit/coredemo/app
后端服务pid: 14595
代理服务启动： http://127.0.0.1:8070
```

访问代理服务 <http://127.0.0.1:8087/demo/demo>：



输出了后端接口内容。

同时在代码中修改下输出内容之后：

```
1 func (api *DemoApi) Demo(c *gin.Context) {
2     c.JSON(200, "this is demo for dev")
3 }
```

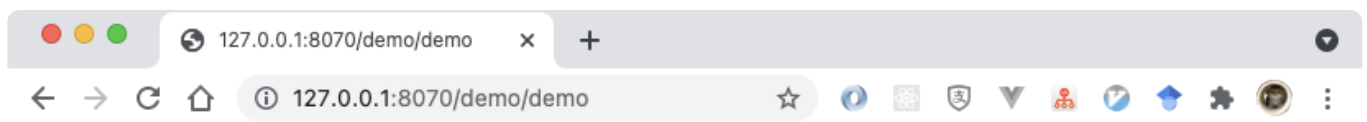
[复制代码](#)

在控制台中我们可以看到，等待了 3s 后（这里配置文件设置为 3s），在控制台看到如下输出：

```
~/Documents/UGit/coredemo geekbang/20 ➤ ./hade dev backend
启动后端服务： http://127.0.0.1:8072
监控文件夹： /Users/yejianfeng/Documents/UGit/coredemo/app
后端服务pid: 14805
代理服务启动： http://127.0.0.1:8070
...检测到文件更新，重启服务开始...
编译hade成功
启动后端服务： http://127.0.0.1:8072
后端服务pid: 14903
...检测到文件更新，重启服务结束...
```

检测到文件更新，重启服务开启。

这个时候我们再刷新浏览器的接口，输出已经变化了。

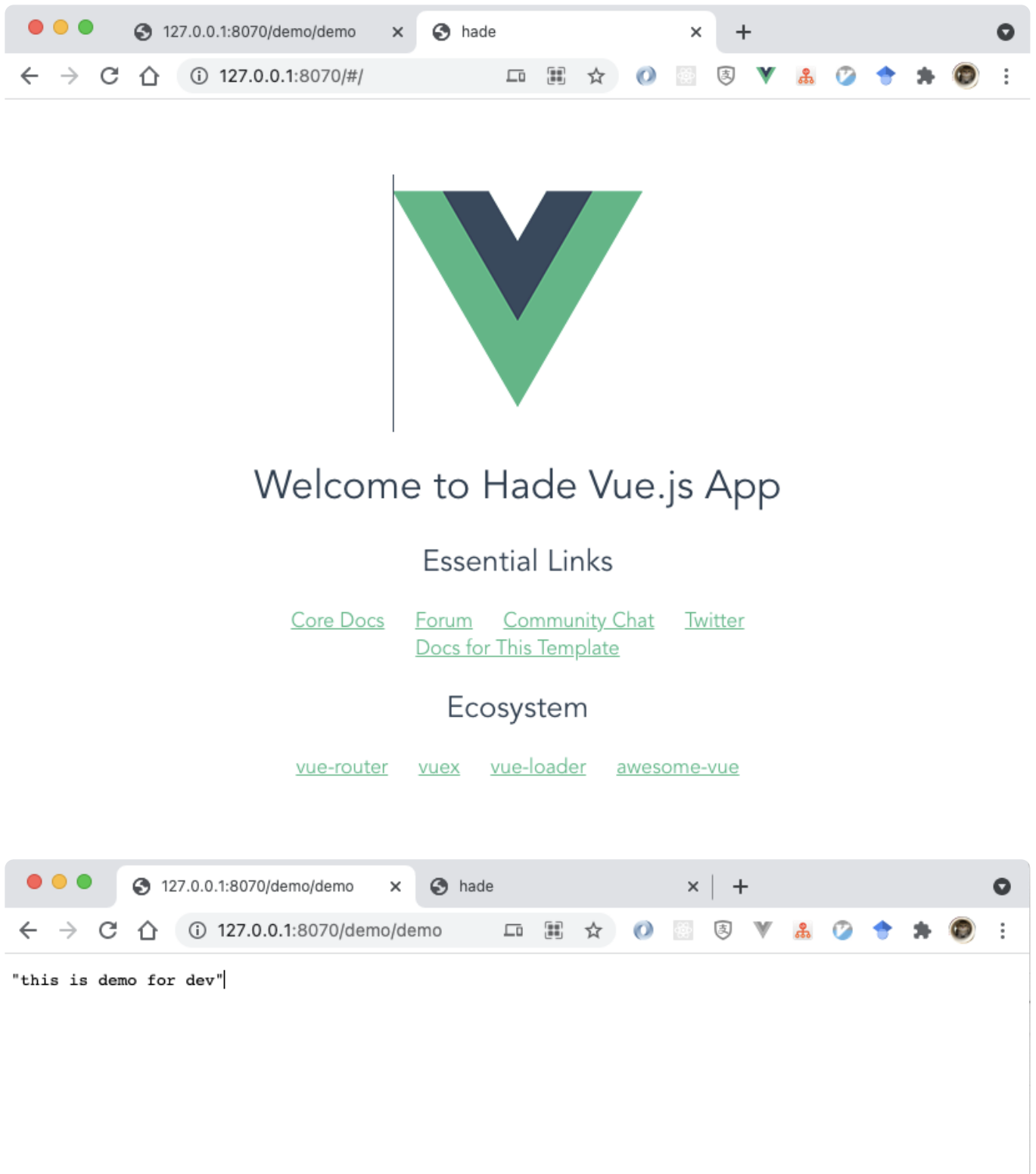


后端调试模式通过！

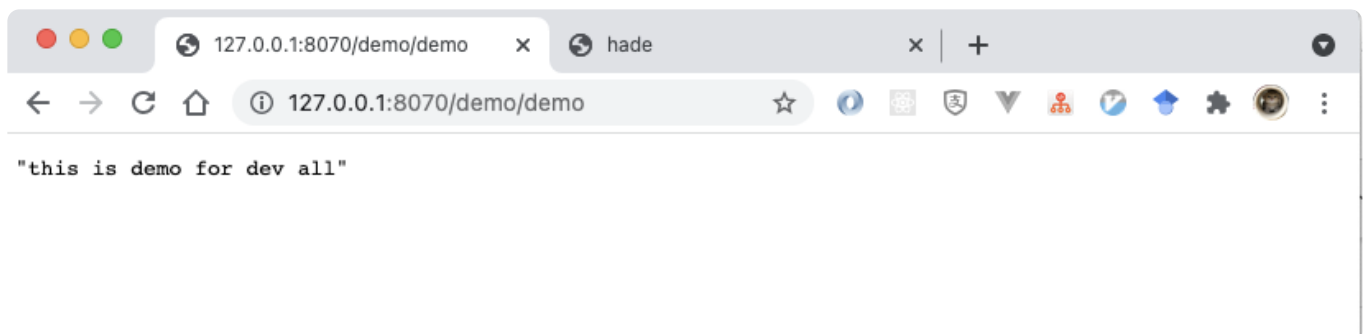
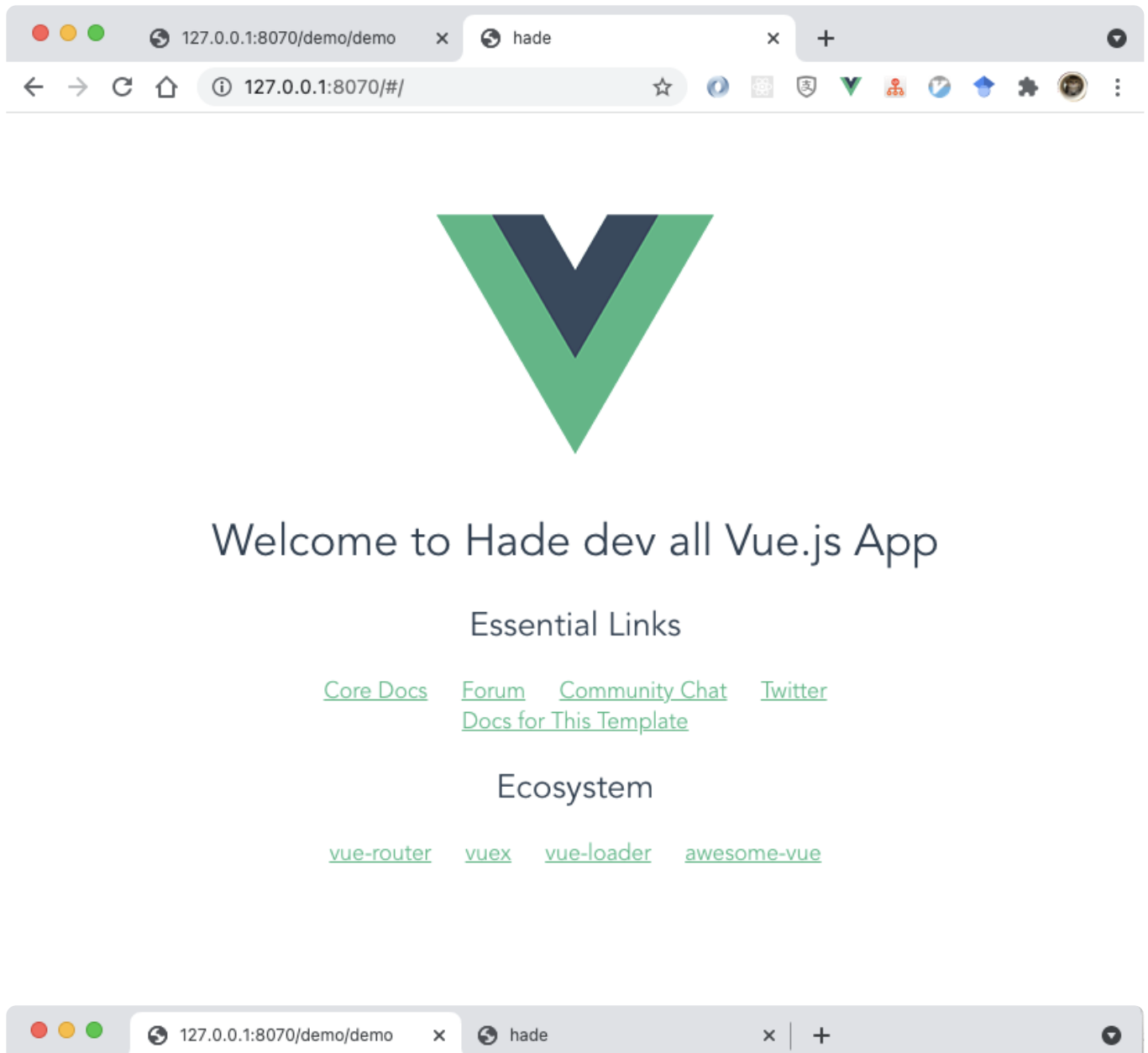
前后端验证

最后同时验证前端和后端，其实和前面单独验证的方法一样，只是启动命令换成了
`./hade dev all`

这里我们同时打开两个窗口，<http://127.0.0.1:8070/demo/demo>、
<http://127.0.0.1:8070/#/>，能同时看到前端和后端信息：

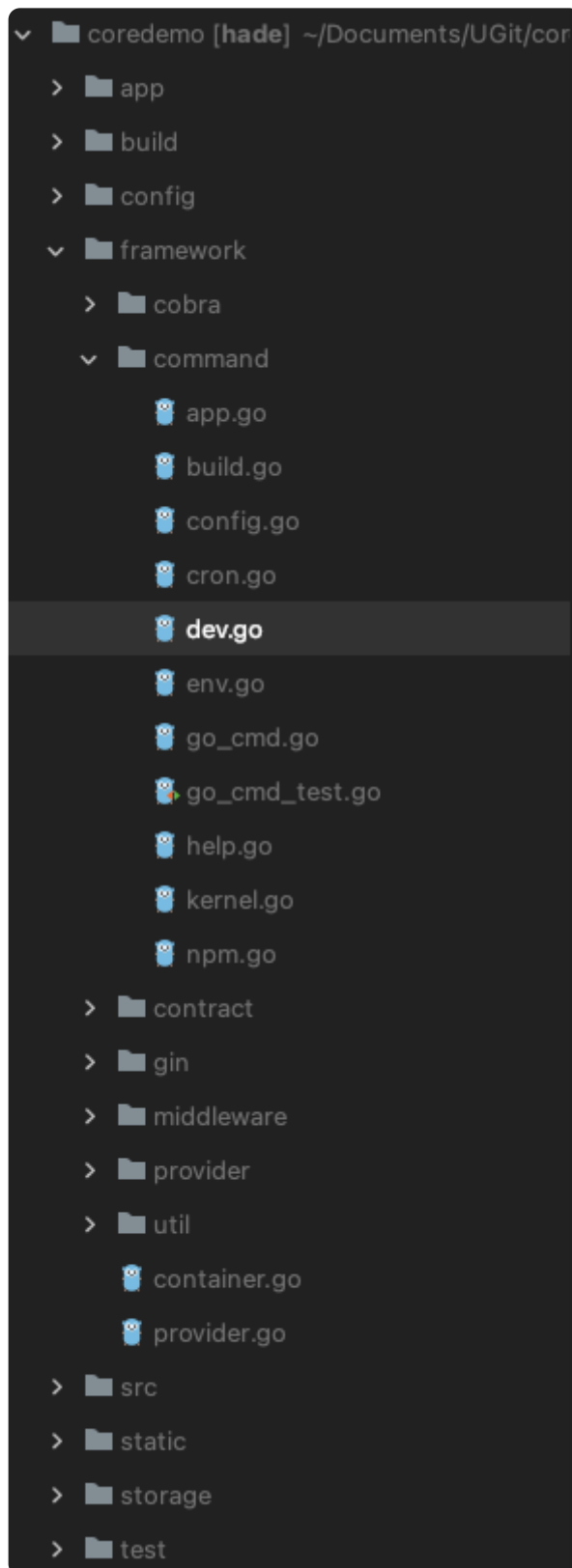


修改前端 msg 和修改后端内容后，变更生效：



到这里，前后端同时调试模式验证成功！

今天的主要内容是创建调试模式的三个二级命令。完整的代码示例在 GitHub 上的 [@geekbang/20](#) 分支，欢迎比对查看。本节课我们只在命令文件中增加了一个 `framework/command/dev.go` 文件：



小结

今天我们具体实现了调试模式，其实了解了上节课对调试模式的设计之后，今天的内容主要是细节上的代码实现了，就是工作量。不过其中的实现细节，也是在工作中不断积累下来的，你可以多多体会。

比如 refresh_time 这个计时器窗口设计，在最初版本是没有的，在实际工作中，使用这个调试模式，遇到了频繁重建的困扰，才做了这个设计。总之，整个调试模式支持是非常赞的，它能让我们的 Web 开发效率提高了一个档次，希望你也能感同身受。

思考题

在回答同学们问题的时候，我发现有不少是其他语言转来 Go 的，不知道你的经历是怎样的，可以来聊一聊你在使用其他语言时，调试一个程序都是怎么调试的呢？有没有比较好的调试模式？

欢迎在留言区分享你的思考。感谢你的收听，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 19 | 提效：实现调试模式加速开发效率（上）

下一篇 21 | 自动化：DRY，如何自动化一切重复性劳动？（上）

11.11 全年底价

VIP 年卡限定 3 折

畅学 200 门课程 & 新课上线即解锁

超值拿下 ¥999

极客时间 VIP 年卡

365 天畅学

11.11 超值福

精选留言 (1)

写留言



qinsi
2021-11-02

实时重载(live reloading): 代码变动时重新编译和启动app。这个应该是文中的模式；

热重载(hot reloading): 代码变动时加载变动后的代码，不重启app，保留当前app的状态；

调试(debug): 通常牵涉到打断点，单步跟踪等。...

展开

