



下载APP



## 08 | 自研or借力，集成Gin替换已有核心（上）

2021-09-29 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 16:49 大小 15.41M



你好，我是轩脉刃。

之前我们从零开始研发了一个有控制器、路由、中间件等组件的 Web 服务器框架，这个服务器框架可以说是麻雀虽小，但五脏俱全。上节课我们还介绍了目前最主流的一些框架 Gin、Iris 和 Beego。

这里不知道你有没有这些疑问，我们的框架和这些顶级的框架相比，差了什么呢？如何才能快速地把我们的框架可用性，和这些框架提升到同一个级别？我们做这个框架除了演每一个实现细节，它的优势是什么呢？



不妨带着这些问题，把我们新出炉的框架和 GitHub 上 star 数最高的  Gin 框架比一下，思考下之间的差距到底是什么。

## 和 Gin 对比

Gin 框架无疑是现在最火的框架，你能想出很多它的好处，但是在我看来，它之所以那么成功，**最主要的原因在于两点：细节和生态。**

其实框架之间的实现原理都差不多，但是生产级别的框架和我们写的示例级别的框架相比，差别就在于细节，这个细节是需要很多人、很多时间去不断打磨的。

如果你的 Golang 经验积累到一定时间，那肯定能很轻松实现一个示例级别的框架，但是往往是没有开源市场的，因为你的框架，在细节上的设计没有经过很多人验证，也没有经过在生产环境中的实战。这些都需要一个较为庞大的使用群体和较长时间才能慢慢打磨出来。

Gin 里面的每一行代码，基本都是在很长时间里，经过各个贡献者的打磨，才演变至今的。我们看 Gin 框架的代码优化和提交频率，从这一讲落笔算起（2021 年 8 月 9 日），最近的一次提交改进是 6 天前（2021 年 8 月 3 日），Gin 框架升级到了 v1.7.3 版本。

我们也可以统计一下 Gin 的 master 分支，从 2014 开始至今已经有 1475 次优化提交。这里的每一次优化提交，都是对 Gin 框架细节的再一次打磨。

## Recovery 的错误捕获

光放数字说服力不明显，我们直接比较代码，看看之前实现的各个逻辑在 Gin 框架中是如何实现的，你就可以感受到在细节上的差距了。

还记得在第四课的时候，我们实现了一个 Recovery 中间件么？放在框架 middleware 文件夹的 recovery.go 中：

[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    }
```

```
11 // 使用 next 执行具体的业务逻辑
12 c.Next()
13
14 return nil
15 }
16 }
```

这个中间件的作用是捕获协程中的函数异常。我们使用 `defer`、`recover` 函数，捕获了 `c.Next` 中抛出的异常，并且在 HTTP 请求中返回 500 内部错误的状态码。

乍看这段代码，是没有什么问题的。但是我们再仔细思考下是否有需要完善的细节？

首先是异常类型，我们原先认为，所有异常都可以通过状态码，直接将异常状态返回给调用方，但是这里是有问题的。**这里的异常，除了业务逻辑的异常，是不是也有可能是底层连接的异常？**

以底层连接中断异常为例，对于这种连接中断，我们是没办法通过设置 HTTP 状态码来让浏览器感知的，并且一旦中断，后续的所有业务逻辑都没有什么作用了。同时，如果我们持续给已经中断的连接发送请求，会在底层持续显示网络连接错误（broken pipe）。

所以在遇到这种底层连接异常的时候，应该直接中断后续逻辑。来看看 Gin 对于连接中断的捕获是怎么处理的。（你可以查看 Gin 的 Recovery [GitHub 地址](#)）

 复制代码

```
1 return func(c *Context) {
2     defer func() {
3         if err := recover(); err != nil {
4             // 判断是否是底层连接异常，如果是的话，则标记 brokenPipe
5             var brokenPipe bool
6             if ne, ok := err.(*net.OpError); ok {
7                 if se, ok := ne.Err.(*os.SyscallError); ok {
8                     if strings.Contains(strings.ToLower(se.Error()), "broken pipe") ||
9                         brokenPipe = true
10                }
11            }
12        }
13        ...
14
15
16        if brokenPipe {
17            // 如果有标记位，我们不能写任何的状态码
18        }
```

```
19         c.Error(err.(error)) // nolint: errcheck
20         c.Abort()
21     } else {
22         handle(c, err)
23     }
24 }
25 }()
26 c.Next()
}
```

这段代码先判断了底层抛出的异常是否是网络异常（`net.OpError`），如果是的话，再根据异常内容是否包含“broken pipe”或者“connection reset by peer”，来判断这个异常是否是连接中断的异常。如果是，就设置标记位，并且直接使用 `c.Abort()` 来中断后续的处理逻辑。

这个处理异常的逻辑可以说是非常细节了，**区分了网络连接错误的异常和普通的逻辑异常，并且进行了不同的处理逻辑**。这一点，可能是绝大多数的开发者都没有考虑到的。

## Recovery 的日志打印

我们再看下 Recovery 的日志打印部分，也非常能体现出 Gin 框架对细节的考量。

当然在第四讲我们只是完成了最基础的部分，没有考虑到打印 Recovery 捕获到的异常，不妨在这里先试想下，如果是自己实现，你会如何打印这个异常呢？如果你有想法了，我们再来对比看看 Gin 是如何实现这个异常信息打印的（[🔗 GitHub 地址](#)）。

首先，打印异常内容是一定得有的。这里直接使用 `logger.Printf` 就可以打印出来了。

```
1 logger.Printf("%s\n%s%s", err, ...)
```

[📄 复制代码](#)

其次，异常是由某个请求触发的，所以触发这个异常的请求内容，也是必要的调试信息，需要打印。

```
1 httpRequest, _ := httputil.DumpRequest(c.Request, false)
2 headers := strings.Split(string(httpRequest), "\r\n")
3 // 如果 header 头中有认证信息，隐藏，不打印。
```

[📄 复制代码](#)

```
4 for idx, header := range headers {
5     current := strings.Split(header, ":")
6     if current[0] == "Authorization" {
7         headers[idx] = current[0] + ": *"
8     }
9 }
10 headersToStr := strings.Join(headers, "\r\n")
```

分析一下这段代码，Gin 使用了一个 `DumpRequest` 来输出请求中的数据，包括了 HTTP header 头和 HTTP Body。

这里值得注意的是，为了安全考虑 Gin 还注意到了，**如果请求头中包含 Authorization 字段，即包含 HTTP 请求认证信息，在输出的地方会进行隐藏处理**，不会由于 panic 就把请求的认证信息输出在日志中。这一个细节，可能大多数开发者也都没有考虑到。

最后看堆栈信息打印，Gin 也有其特别的实现。我们打印堆栈一般是使用 [runtime](#) 库的 `Caller` 来打印：

[复制代码](#)

```
1 // 打印堆栈信息，是否有这个堆栈
2 func Caller(skip int) (pc uintptr, file string, line int, ok bool)
```


`caller` 方法返回的是堆栈函数所在的函数指针、文件名、函数所在行号信息。但是在使用过程中你就会发现，使用 `Caller` 是打印不出真实代码的。

比如下面这个例子，我们使用 `Caller` 方法，将文件名、行号、堆栈函数打印出来：

[复制代码](#)

```
1 // 在 prog.go 文件，main 库中调用 call 方法
2 func call(skip int) bool { //24 行
3     pc,file,line,ok := runtime.Caller(skip) //25 行
4     pcName := runtime.FuncForPC(pc).Name() //26 行
5     fmt.Println(fmt.Sprintf("%v %s %d %s",pc,file,line,pcName)) //27 行
6     return ok //28 行
7 } //29 行
```

打印出的第一层堆栈函数的信息：

 复制代码

```
1 4821380 /tmp/sandbox064396492/prog.go 25 main.call
```

这个堆栈信息并不友好，它告诉我们，第一层信息所在地址为 prog.go 文件的第 25 行，在 main 库的 call 函数里面。所以如果了解下第 25 行有什么内容，用这个堆栈信息去源码中进行文本查找，是做不到的。这个时候就非常希望信息能打印出具体的真实代码。

在 Gin 中，打印堆栈的时候就有这么一个逻辑：**先去本地查找是否有这个源代码文件，如果有的话，获取堆栈所在的代码行数，将这个代码行数直接打印到控制台中。**

 复制代码

```
1 // 打印具体的堆栈信息
2 func stack(skip int) []byte {
3     ...
4     // 循环从第 skip 层堆栈到最后一层
5     for i := skip; ; i++ {
6         pc, file, line, ok := runtime.Caller(i)
7         // 获取堆栈函数所在源文件
8         if file != lastFile {
9             data, err := ioutil.ReadFile(file)
10            if err != nil {
11                continue
12            }
13            lines = bytes.Split(data, []byte{'\n'})
14            lastFile = file
15        }
16        // 打印源代码的内容
17        fmt.Fprintf(buf, "\t%s: %s\n", function(pc), source(lines, line))
18    }
19    return buf.Bytes()
20 }
21
```

这样，打印出来的堆栈信息形如：

 复制代码

```
1 /Users/yejianfeng/Documents/gopath/pkg/mod/github.com/gin-gonic/gin@v1.7.2/con
2     (*Context).Next: c.handlers[c.index](c)
```



这个堆栈信息就友好多了，它告诉我们，这个堆栈函数发生在文件的 165 行，它的代码为 `c.handlers` 的 `c.index`，这行代码所在的父级别函数为 `(*Context).Next`。

最终，Gin 打印出来的 panic 信息形式如下：

[复制代码](#)

```
1 2021/08/15 14:18:57 [Recovery] 2021/08/15 - 14:18:57 panic recovered:
2 GET /first HTTP/1.1
3 Host: localhost:8080
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image
5 Accept-Encoding: gzip, deflate, br
6 ...
7 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
8
9
10
11 %!s(int=121321321)
12 /Users/yejianfeng/Documents/UGit/gindemo/main.go:19 (0x1394214)
13     main.func1: panic(121321321)
14 /Users/yejianfeng/Documents/gopath/pkg/mod/github.com/gin-gonic/gin@v1.7.2/con
15     (*Context).Next: c.handlers[c.index](c)
16 /Users/yejianfeng/Documents/gopath/pkg/mod/github.com/gin-gonic/gin@v1.7.2/rec
17     CustomRecoveryWithWriter.func1: c.Next()
18 /Users/yejianfeng/Documents/gopath/pkg/mod/github.com/gin-gonic/gin@v1.7.2/con
19     (*Context).Next: c.handlers[c.index](c)
20 ...
21 /usr/local/Cellar/go/1.15.5/libexec/src/net/http/server.go:1925 (0x12494ac)
22     (*conn).serve: serverHandler{c.server}.ServeHTTP(w, w.req)
23 /usr/local/Cellar/go/1.15.5/libexec/src/runtime/asm_amd64.s:1374 (0x106bb00)
24     goexit: BYTE    $0x90    // NOP
```

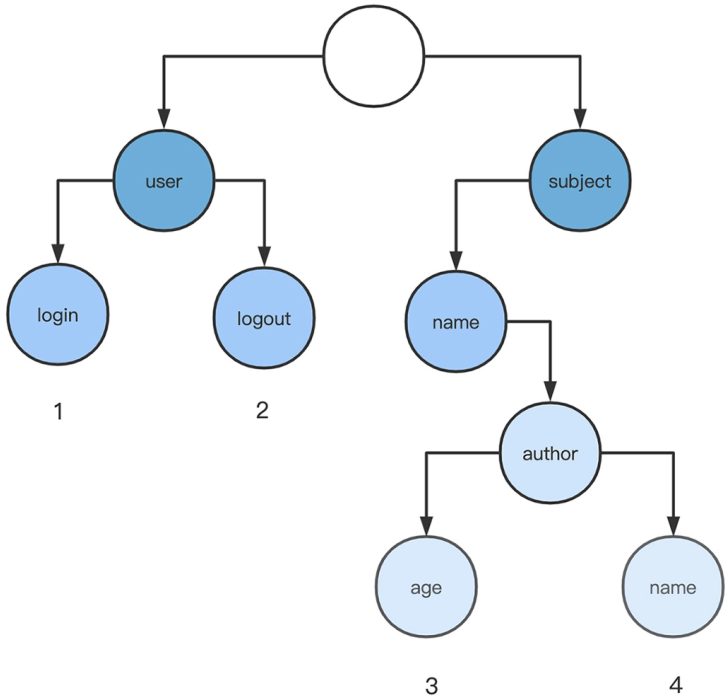
可以说这个调试信息就非常丰富了，对我们在实际工作中的调试会有非常大的帮助。但是这些丰富的细节都是在开源过程中，不断补充起来的。

## 路由对比

刚才只挑 Recovery 中间件的错误捕获和日志打印简单说了一下，我们可以再挑核心一些的功能，例如比较一下我们实现的路由和 Gin 的路由的区别。

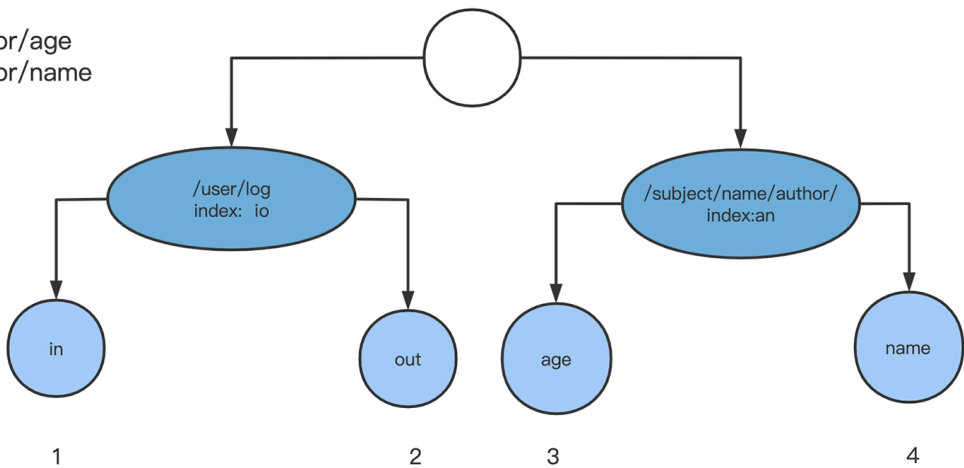
在第三节课使用 trie 树实现了一个路由，它的每一个节点是一个 segment。

- 1 /user/login
- 2 /user/logout
- 3 /subject/name/author/age
- 4 /subject/name/author/name



而 Gin 框架的路由选用的是一种压缩后的基数树（radix tree），它和我们之前实现的 trie 树相比最大的区别在于，它并不是把按照分隔符切割的 segment 作为一个节点，而是**把整个 URL 当作字符串，尽可能匹配相同的字符串作为公共前缀。**

- 1 /user/login
- 2 /user/logout
- 3 /subject/name/author/age
- 4 /subject/name/author/name





为什么 Gin 选了这个模型？其实 radix tree 和 trie 树相比，最大的区别就在于它节点的压缩比例最大化。直观比较上面两个图就能看得出来，对于 URL 比较长的路由规则，trie 树的节点数就比 radix tree 的节点数更多，整个数的层级也更深。

针对路由这种功能模块，创建路由树的频率远低于查找路由点频率，那么**减少节点层级，无异于能提高查找路由的效率，整体路由模块的性能也能得到提高**，所以 Gin 用 radix tree 是更好的选择。

另外在路由查找中，Gin 也有一些细节做得很好。首先，从父节点查找子节点，并不是像我们之前实现的那样，通过遍历所有子节点来查找是否有子节点。Gin 在每个 node 节点保留一个 indices 字符串，这个字符串的每个字符代表子节点的第一个字符：

在 Gin 源码的 [tree.go](#) 中可以看到。

[复制代码](#)

```
1 // node 节点
2 type node struct {
3     path      string
4     indices   string // 子节点的第一个字符
5     ...
6     children []*node // 子节点
7     ...
8 }
```

这个设计是为了加速子节点的查询效率。使用 Hash 查找的时间复杂度为  $O(1)$ ，而我们使用遍历子节点的方式进行查找的效率为  $O(n)$ 。

在拼接 indices 字符串的时候，这里 Gin 还有一个代码方面的细节值得注意，[在 tree.go](#) 中有一段这样的代码：

[复制代码](#)

```
1     path = path[i:]
2     c := path[0]
3
4     ...
5     // 插入子节点
6     if c != ':' && c != '*' && n.nType != catchAll {
7         // []byte for proper unicode char conversion, see #65
8         // 将字符串拼接进入 indices
```

```
9     n.indices += bytesconv.BytesToString([]byte{c})
10    ...
11    n.addChild(child)
12    ...
```

将字符 `c` 拼接进入 `indices` 的时候，使用了一个 `bytesconv.BytesToString` 方法来将字符 `c` 转换为 `string`。你可以先想想，这里为什么不使用 `string` 关键字直接转换呢？

因为在 Golang 中，`string` 转化为 `byte` 数组，以及 `byte` 数组转换为 `string`，都是有内存消耗的。以 `byte` 数组使用关键字 `string` 转换为字符串为例，Golang 会先在内存空间中重新开辟一段字符串空间，然后将 `byte` 数组复制进入这个字符串空间，这样不管是在内存使用的效率，还是在 CPU 资源使用的效率上，都存在一定消耗。

而在 Gin 的第 [PR #2206](#) 号提交中，有开源贡献者就使用自研的 `bytesconv` 包，将 Gin 中的字符数组和 `string` 的转换统一做了一次修改。

[复制代码](#)

```
1 package bytesconv
2
3 // 字符串转化为字符数组，不需要创建新的内存
4 func StringToBytes(s string) []byte {
5     return *(*[]byte)(unsafe.Pointer(
6         &struct {
7             string
8             Cap int
9         }{s, len(s)},
10    ))
11 }
12
13 // 字符数组转换为字符串，不需要创建新的内存
14 func BytesToString(b []byte) string {
15     return *(*string)(unsafe.Pointer(&b))
16 }
```

`bytesconv` 包的原理就是，直接使用 `unsafe.Pointer` 来强制转换字符串或者字符数组的类型。

这些细节的修改，一定程度上减少了 Gin 包内路由的内存占用空间。类似的细节点还有很多，需要每一行代码琢磨过去，而且这里的每一个细节优化点都是，开源贡献者在生产环

境中长期使用 Gin 才提炼出来的。

不要小看这些看似非常细小的修改。因为细节往往是积少成多的，所有的这些细节逐渐优化、逐渐完善，才让 Gin 框架的实用度得到持久的提升，和其他框架的差距就逐渐体现出来了，这样才让 Gin 框架成了生产环境中首选的框架。

## 生态

聊完细节，再来看看生态。你肯定有点疑惑，为什么我会把生态这个点单独拿出来，难道生态不是由于框架的质量好而附带的生态繁荣吗？

其实不然。一个开源项目的成功，最重要的是两个事情，**第一个是质量，开源项目的代码质量是摆在第一位的，但是还有一个是不能被忽略的：生态的完善。**

一个好的开源框架项目，不仅仅只有代码，还需要围绕着核心代码打造文档、框架扩展、辅助命令等。这些代码周边的组件和功能的打造，虽然从难度上看，并没有核心代码那么重要，但是它是一个长期推进和完善的过程。

Gin 的整个生态环境是非常优质的，在开源中间件、社区上都能看到其优势。

从 2014 年至今 Gin 已有多个开源共享者为其共享了开源中间件，目前 [官方 GitHub](#) 组织收录的中间件有 23 个，非收录官方，但是在 [官方 README](#) 记录的也有 45 个。

这些中间件包含了 Web 开发各个方面的功能，比如提供跨域请求的 cors 中间件、提供本地缓存的 cache 中间件、集成了 pprof 工具的 pprof 中间件、自动生成全链路 trace 的 opengintracing 中间件等等。**如果你用一个自己的框架，就需要重建生态——开发，这是非常烦琐的，而且工作量巨大。**

Gin 的 GitHub 官网的社区活跃度比较高，对于已有的一些问题，在官网的 issue 提出来立刻就会有人回复，在 Google 或者 Baidu 上搜索 Gin，也会有很多资料。所以对于工作中必然会出现的问题，我们可以在网络上很方便找寻到这个框架的解决办法。这也是 Gin 大受欢迎的原因之一。

其实除了 Gin 框架，我们可以从不少其他语言的框架中看到生态的重要性。比如前端的 Vue 框架，它不仅仅提供了最核心的 Vue 的框架代码，还提供了脚手架工具 Vue-CLI、

Vue 的 Chrome 插件、Vue 的官方论坛和聊天室、Vue 的示例文档 Cookbook。这些周边生态对 Vue 框架的成功起到了至关重要的作用。

## 站在巨人的肩膀才能做得更好

刚才分析了一些 Gin 框架的细节以及它强大的生态，相信已经回答了开头我们提出来的问题：和这些顶级的框架相比，我们做的到底差了什么？我们的框架在可用性上，能不能迅速提升到这些顶级框架的级别？很明显短时间是不可可能的。

讲了这么多，其实我想说：只有站在巨人的肩膀才能做得更好。

如果是为了学习，我们之前从零自己边造轮子边学是个好方法；**但是如果你的目标是工业使用，那从零开始就非常不明智了。**

因为在现在的技术氛围下，开源早已成为了共识。互联网的开源社区早就有我们需要的各个成形零件，我们要做的是，使用这些零件来继续开发。在 Golang Web 框架这个领域也是一样的，如果是想从头开始制造框架，除非你后面有很强大的团队在支撑，否则写出来的框架不仅没有市场，可能连实用性也会受到质疑。

**其实很多市面上的框架，也都是基于已有的轮子来再开发的。**就拿 Gin 框架本身来说吧，它的路由是基于 httprouter 这个项目来定制化修改的；再比如 Macaron 框架，它是基于 Martini 框架的设计实现的。它们都是在原有的开源项目基础上，按照自己的设计思路重新改造的，也都获得了成功。

而且从我的个人经验来看，那些从头开始所有框架功能都是由自己开发的同学，往往很难坚持下来。所以你现在是不是明白了，为什么在课程最开始会说，我们先从零搭建出框架的核心部分，然后基于 Gin 来做进一步拓展和完善。

因为我们希望通过这门课花大力气搭建出来的 Golang Web 框架，不只是一个示例级别的框架，而是真正能用到具体工作环境中的，要做到这一点，它得使用开源社区的各种优秀开源库，目的就是提升你解决问题的效率。

## 小结

今天通过对比 Gin 框架和我们之前设计的框架间的细节，展示了一个成熟的生产级别的框架与一个示例级别框架在细节上的距离。

**现代框架的理念不在于实现，而更多在于组合。**基于某些基础组件或者基础实现，不断按照自己或者公司的需求，进行二次改造和二次开发，从而打造出适合需求的形态。

比如 PHP 领域的 Laravel 框架，就是将各种底层组件、Symfony、Eloquent ORM、Monolog 等进行组装，而框架自身提供统一的组合调度方式；比如 Ruby 领域的 Rails 框架，整合了 Ruby 领域的各种优秀开源库，**而框架的重点在于如何整理组件库、如何提供更便捷的机制，让程序员迅速解决问题。**

所以，接下来我们设计框架的思路，就要从之前的从零开始造轮子，转换为站在巨人的肩膀上了，会借用 Gin 框架来继续实现。准备好，下一讲我们就要开始动手改造了。

## 思考题

在 Gin 框架中，也和我们第 5 讲一样提供了 Query 系列方法，你能看出 Gin 实现的 Query 系列方法和我们实现的有什么不同么？

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 4  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 07 | 目标：站在巨人肩膀，你的理想框架到底长什么样？

下一篇 09 | 自研or借力：集成Gin替换已有核心(下)

精选留言 (3)

写留言



王博

2021-09-29

前面的读了好几遍了，真的受益匪浅，期待后续  
展开



qinsi

2021-09-29

打印堆栈是对本地开发友好的功能，线上不会那么搞。因为通常部署的时候不会把源码一起部署上线，而且一个个去读取源文件开销也大  
展开



Geek\_5244fa

2021-09-29

众人拾柴火焰高，这个课程这么多人学习，可以一起做一个框架。

