



下载APP



16 | 代码检查：如何进行静态代码检查？

2021-07-01 孔令飞

《Go 语言项目开发实战》

课程介绍 >


**讲述：孔令飞**

时长 11:07 大小 10.19M



你好，我是孔令飞。上一讲中，我在讲代码开发的具体步骤时，提到了静态代码检查，今天我就来详细讲讲如何执行静态代码检查。

在做 Go 项目开发的过程中，我们肯定需要对 Go 代码做静态代码检查。虽然 Go 命令提供了 `go vet` 和 `go tool vet`，但是它们检查的内容还不够全面，我们需要一种更加强大的静态代码检查工具。

其实，Go 生态中有很多这样的工具，也不乏一些比较优秀的。今天我想给你介绍的 `golangci-lint`，是目前使用最多，也最受欢迎的静态代码检查工具，我们的 IAM 实战 ，也用到了它。

接下来, 我就从 golangci-lint 的优点、golangci-lint 提供的命令和选项、golangci-lint 的配置这三个方面来向你介绍下它。在你了解这些基础知识后, 我会带着你使用 golangci-lint 进行静态代码检查, 让你熟悉操作, 在这个基础上, 再把使用 golangci-lint 时总结的一些经验技巧分享给你。

为什么选择 golangci-lint 做静态代码检查?

选择 golangci-lint, 是因为它具有其他静态代码检查工具不具备的一些优点。在我看来, 它的核心优点至少有这些:

速度非常快: golangci-lint 是基于 gometalinter 开发的, 但是平均速度要比 gometalinter 快 5 倍。golangci-lint 速度快的原因有三个: 可以并行检查代码; 可以复用 go build 缓存; 会缓存分析结果。

可配置: 支持 YAML 格式的配置文件, 让检查更灵活, 更可控。

IDE 集成: 可以集成进多个主流的 IDE, 例如 VS Code、GNU Emacs、Sublime Text、Goland 等。

linter 聚合器: 1.41.1 版本的 golangci-lint 集成了 76 个 linter, 不需要再单独安装这 76 个 linter。并且 golangci-lint 还支持自定义 linter。

最小的误报数: golangci-lint 调整了所集成 linter 的默认设置, 大幅度减少了误报。

良好的输出: 输出的结果带有颜色、代码行号和 linter 标识, 易于查看和定位。

下图是一个 golangci-lint 的检查结果:

```
[colin@dev addlicense]$ golangci-lint run
addlicense.go:121: Function 'main' has too many statements (46 > 40) (funlen)
func main() {
addlicense.go:103: line is 141 characters (lll)
    checkonly = pflag.BoolP("check", "", false, "check only mode: verify presence of license headers and exit with non-zero code
    if missing")
addlicense.go:327: line is 146 characters (lll)
    case ".cc", ".cpp", ".cs", ".go", ".hh", ".hpp", ".java", ".m", ".mm", ".proto", ".rs", ".scala", ".swift", ".dart", ".groov
y", ".kt", ".kts":
addlicense.go:161:2: `if *licensef != ""` is deeply nested (complexity: 4) (nestif)
    if *licensef != "" {
        ^
addlicense.go:167:3: assignments should only be cuddled with other assignments (ws1)
    t, err = template.New("").Parse(string(d))
    ^
```

你可以看到, 输出的检查结果中包括如下信息:

检查出问题的源码文件、行号和错误行内容。

出问题的原因, 也就是打印出不符合检查规则的原因。

报错的 linter。


通过查看 golangci-lint 的输出结果，可以准确地定位到报错的位置，快速弄明白报错的原因，方便开发者修复。

除了上述优点之外，在我看来 golangci-lint 还有一个非常大的优点：**当前更新迭代速度很快，不断有新的 linter 被集成到 golangci-lint 中。**有这么全的 linter 为你的代码保驾护航，你在交付代码时肯定会更有自信。

目前，有很多公司 / 项目使用了 golangci-lint 工具作为静态代码检查工具，例如 Google、Facebook、Istio、Red Hat OpenShift 等。

golangci-lint 提供了哪些命令和选项？

在使用之前，首先需要**安装 golangci-lint**。golangci-lint 的安装方法也很简单，你只需要执行以下命令，就可以安装了。

 复制代码

```
1 $ go get github.com/golangci/golangci-lint/cmd/golangci-lint@v1.41.1
2 $ golangci-lint version # 输出 golangci-lint 版本号，说明安装成功
3 golangci-lint has version v1.39.0 built from (unknown, mod sum: "h1:aAUjdBxARw
```

这里注意，为了避免安装失败，强烈建议你安装 golangci-lint releases page 中的指定版本，例如 v1.41.1。

另外，还建议你定期更新 golangci-lint 的版本，因为该项目正在被积极开发并不断改进。

安装之后，就可以使用了。我们可以通过执行 golangci-lint -h 查看其用法，golangci-lint 支持的**子命令**见下表：

子命令	功能
cache	缓存控制，并打印缓存的信息
completion	输出bash/fish/powershell/zsh自动补全脚本
config	打印golangci-lint当前使用的配置文件路径
help	打印golangci-lint的帮助信息
linters	打印golangci-lint所支持的linter
run	执行golangci-lint对代码进行检查
version	打印golangci-lint的版本号

此外，golangci-lint 还支持一些**全局选项**。全局选项是指适用于所有子命令的选项，golangci-lint 支持的全局选项如下：

选项	功能
<code>--color</code>	是否打印带颜色的输出，有3个值：always, auto（默认），never
<code>j, --concurrency</code>	开启多少并发，默认：NumCPU
<code>--cpu-profile-path</code>	记录cpu性能数据到指定文件
<code>--mem-profile-path</code>	记录memory性能数据到指定文件
<code>--trace-path</code>	生成跟踪文件
<code>-h, --help</code>	输出golangci-lint的help信息
<code>-v, --verbose</code>	输出更多信息

接下来，我就详细介绍下 golangci-lint 支持的核心子命令：run、cache、completion、config、linters。

run 命令


run 命令执行 golangci-lint，对代码进行检查，是 golangci-lint 最为核心的一个命令。run 没有子命令，但有很多选项。run 命令的具体使用方法，我会在讲解如何执行静态代码检查的时候详细介绍。

cache 命令

cache 命令用来进行缓存控制，并打印缓存的信息。它包含两个子命令：

clean 用来清除 cache，当我们觉得 cache 的内容异常，或者 cache 占用空间过大时，可以通过 `golangci-lint cache clean` 清除 cache。

status 用来打印 cache 的状态，比如 cache 的存放目录和 cache 的大小，例如：


 复制代码

```
1 $ golangci-lint cache status
2 Dir: /home/colin/.cache/golangci-lint
3 Size: 773.4KiB
```

completion 命令


completion 命令包含 4 个子命令 bash、fish、powershell 和 zsh , 分别用来输出 bash、fish、powershell 和 zsh 的自动补全脚本。

下面是一个配置 bash 自动补全的示例 :

 复制代码

```
1 $ golangci-lint completion bash > ~/.golangci-lint.bash
2 $ echo "source '$HOME/.golangci-lint.bash'" >> ~/.bashrc
3 $ source ~/.bashrc
```

执行完上面的命令 , 键入如下命令 , 即可自动补全子命令 :


 复制代码

```
1 $ golangci-lint comp<TAB>
```

上面的命令行会自动补全为 golangci-lint completion 。

config 命令

config 命令可以打印 golangci-lint 当前使用的配置文件路径 , 例如 :

 复制代码

```
1 $ golangci-lint config path
2 .golangci.yaml
```

linters 命令

linters 命令可以打印出 golangci-lint 所支持的 linter，并将这些 linter 分成两类，分别是配置为启用的 linter 和配置为禁用的 linter，例如：

[复制代码](#)

```
1 $ golangci-lint linters
2 Enabled by your configuration linters:
3 ...
4 deadcode: Finds unused code [fast: true, auto-fix: false]
5 ...
6 Disabled by your configuration linters:
7 exportloopref: checks for pointers to enclosing loop variables [fast: true, au
8 ...
```

上面我介绍了 golangci-lint 提供的命令，接下来，我们再来看下 golangci-lint 的配置。

golangci-lint 配置

和其他 linter 相比，golangci-lint 一个非常大的优点是使用起来非常灵活，这要得益于它对自定义配置的支持。

golangci-lint 支持两种配置方式，分别是命令行选项和配置文件。如果 bool/string/int 的选项同时在命令行选项和配置文件中被指定，命令行的选项就会覆盖配置文件中的选项。如果是 slice 类型的选项，则命令行和配置中的配置会进行合并。

golangci-lint run 支持很多**命令行选项**，可通过 golangci-lint run -h 查看，这里选择一些比较重要的选项进行介绍，见下表：

选项	功能
<code>--print-issued-lines</code>	显示检查失败行所在的行号, 默认显示
<code>--print-linter-name</code>	显示检查失败行是由哪个linter引起的失败, 默认显示
<code>--timeout</code>	设置golanci-lint检查超时时间, 默认1分钟
<code>--tests</code>	是否检查*_test.go文件, 默认检查
<code>-c, --config PATH</code>	指定配置文件, 默认会从当前目录开始逐级往上查找.golangci.yaml、.golangci.json、.golangci.xml文件, 一直查找到根 (/) 目录, 如果找到则使用找到的文件作为配置文件
<code>--no-config</code>	不读取任何配置文件
<code>--skip-dirs</code>	设置需要忽略的文件夹, 支持正则表达式, 多个目录/正则, 用逗号隔开
<code>--skip-dirs-use-default</code>	使用预设的规则来忽略文件夹, 默认true

<code>--skip-files</code>	设置需要忽略的文件, 支持正则表达式, 多个目录/正则, 用逗号隔开
<code>-E, --enable</code>	使用指定的linter
<code>-D, --disable</code>	禁用指定的linter
<code>--disable-all</code>	禁用所有的linter
<code>--fast</code>	从启用的linter中, 选出支持快速检查的linter, 这些linter在第一次执行时, 需要缓存类型信息, 所以第一次检查并不快, 但后续的检查会很快
<code>-e, --exclude</code>	设置需要忽略的检查错误
<code>--exclude-use-default</code>	忽略预设的错误, 默认true
<code>--exclude-case-sensitive</code>	设置exclue规则时, 是否大小写敏感
<code>--max-issues-per-linter</code>	设置每个linter报告错误的最大错误数, 默认50
<code>--fix</code>	如果linter支持修复功能, 则fix发现的错误

此外, 我们还可以通过 `golangci-lint` **配置文件** 进行配置, 默认的配置文件名 为 `.golangci.yaml`、`.golangci.toml`、`.golangci.json`, 可以通过 `-c` 选项指定配置文件名。通过配置文件, 可以实现下面几类功能:

`golangci-lint` 本身的一些选项, 比如超时、并发, 是否检查 `*_test.go` 文件等。

配置需要忽略的文件和文件夹。

配置启用哪些 linter , 禁用哪些 linter。


配置输出格式。

golangci-lint 支持很多 linter , 其中有些 linter 支持一些配置项 , 这些配置项可以在配置文件中配置。

配置符合指定正则规则的文件可以忽略的 linter。

设置错误严重级别 , 像日志一样 , 检查错误也是有严重级别的。

更详细的配置内容 , 你可以参考 [Configuration](#)。另外 , 你也可以参考 IAM 项目的 golangci-lint 配置 [.golangci.yaml](#)。 .golangci.yaml 里面的一些配置 , 我建议你一定要设置 , 具体如下 :

 复制代码

```
1 run:
2   skip-dirs: # 设置要忽略的目录
3     - util
4     - .*~
5     - api/swagger/docs
6   skip-files: # 设置不需要检查的go源码文件, 支持正则匹配, 这里建议包括: _test.go
7     - ".*\\.my\\.go$"
8     - _test.go
9   linters-settings:
10    errcheck:
11      check-type-assertions: true # 这里建议设置为true, 如果确实不需要检查, 可以写成`nu
12      check-blank: false
13    gci:
14      # 将以`github.com/marmotedu/iam`开头的包放在第三方包后面
15      local-prefixes: github.com/marmotedu/iam
16    godox:
17      keywords: # 建议设置为BUG、FIXME、OPTIMIZE、HACK
18        - BUG
19        - FIXME
20        - OPTIMIZE
21        - HACK
22    goimports:
23      # 设置哪些包放在第三方包后面, 可以设置多个包, 逗号隔开
24      local-prefixes: github.com/marmotedu/iam
25    gomoddirectives: # 设置允许在go.mod中replace的包
26      replace-local: true
27      replace-allow-list:
28        - github.com/coreos/etcd
29        - google.golang.org/grpc
30        - github.com/marmotedu/api
```

```

31     - github.com/marmotedu/component-base
32     - github.com/marmotedu/marmotedu-sdk-go
33 gomodguard: # 下面是根据需要进行选择的包和版本, 建议设置
34     allowed:
35         modules:
36             - gorm.io/gorm
37             - gorm.io/driver/mysql
38             - k8s.io/klog
39         domains: # List of allowed module domains
40             - google.golang.org
41             - gopkg.in
42             - golang.org
43             - github.com
44             - go.uber.org
45     blocked:
46         modules:
47             - github.com/pkg/errors:
48                 recommendations:
49                     - github.com/marmotedu/errors
50                 reason: "`github.com/marmotedu/errors` is the log package used by
51 versions:
52             - github.com/MakeNowJust/heredoc:
53                 version: "> 2.0.9"
54                 reason: "use the latest version"
55     local_replace_directives: false
56 lll:
57     line-length: 240 # 这里可以设置为240, 240一般是够用的
58 importas: # 设置包的alias, 根据需要设置
59     jwt: github.com/appleboy/gin-jwt/v2
60     metav1: github.com/marmotedu/component-base/pkg/meta/v1

```

需要注意的是, `golangci-lint` 不建议使用 `enable-all: true` 选项, 为了尽可能使用最全的 linters, 我们可以使用以下配置:

[复制代码](#)


```

1 linters:
2     disable-all: true
3     enable: # enable下列出 <期望的所有linters>
4         - typecheck
5         - ...

```

<期望的所有linters> = <golangci-lint支持的所有linters> - <不期望执行的linters>, 我们可以通过执行以下命令来获取:

```
1 $ ./scripts/print_enable_linters.sh
2     - asciicheck
3     - bodyclose
4     - cyclop
5     - deadcode
6     - ...
```

 复制代码

将以上输出结果替换掉 `golangci.yaml` 配置文件中的 `linters.enable` 部分即可。


上面我们介绍了与 `golangci-lint` 相关的一些基础知识，接下来我就给你详细展示下，如何使用 `golangci-lint` 进行静态代码检查。

如何使用 `golangci-lint` 进行静态代码检查？

要对代码进行静态检查，只需要执行 `golangci-lint run` 命令即可。接下来，我会先给你介绍 5 种常见的 `golangci-lint` 使用方法。

1. 对当前目录及子目录下的所有 Go 文件进行静态代码检查：


```
1 $ golangci-lint run
```

 复制代码

命令等效于 `golangci-lint run ./...`。

2. 对指定的 Go 文件或者指定目录下的 Go 文件进行静态代码检查：


```
1 $ golangci-lint run dir1 dir2/... dir3/file1.go
```

 复制代码

这里需要你注意：上述命令不会检查 `dir1` 下子目录的 Go 文件，如果想递归地检查一个目录，需要在目录后面追加 `/...`，例如：`dir2/...`。

3. 根据指定配置文件，进行静态代码检查：

```
1 $ golangci-lint run -c .golangci.yaml ./...
```

 复制代码

4. 运行指定的 linter :

golangci-lint 可以在不指定任何配置文件的情况下运行, 这会运行默认启用的 linter, 你可以通过 `golangci-lint help linters` 查看它。

你可以传入参数 `-E/--enable` 来使某个 linter 可用, 也可以使用 `-D/--disable` 参数来使某个 linter 不可用。下面的示例仅仅启用了 `errcheck` linter :

```
1 $ golangci-lint run --no-config --disable-all -E errcheck ./...
```


 复制代码

这里你需要注意, 默认情况下, `golangci-lint` 会从当前目录一层层往上寻找配置文件名 `.golangci.yaml`、`.golangci.toml`、`.golangci.json` 直到根 (/) 目录。如果找到, 就以找到的配置文件作为本次运行的配置文件, 所以为了防止读取到未知的配置文件, 可以用 `--no-config` 参数使 `golangci-lint` 不读取任何配置文件。

5. 禁止运行指定的 linter :

如果我们想禁用某些 linter, 可以使用 `-D` 选项。

```
1 $ golangci-lint run --no-config -D godot,errcheck
```

 复制代码

在使用 `golangci-lint` 进行代码检查时, 可能会有很多误报。所谓的误报, 其实是我们希望 `golangci-lint` 的一些 linter 能够容忍某些 issue。那么如何尽可能减少误报呢? `golangci-lint` 也提供了一些途径, 我建议你使用下面这三种:

在命令行中添加 `-e` 参数, 或者在配置文件的 `issues.exclude` 部分设置要排除的检查错误。你也可以使用 `issues.exclude-rules` 来配置哪些文件忽略哪些 linter。

通过`run.skip-dirs`、`run.skip-files`或者`issues.exclude-rules`配置项, 来忽略指定目录下的所有 Go 文件, 或者指定的 Go 文件。

通过在 Go 源码文件中添加`// nolint`注释, 来忽略指定的代码行。

因为 `golangci-lint` 设置了很多 linters, 对于一个大型项目, 启用所有的 linter 会检查出很多问题, 并且每个项目对 linter 检查的粒度要求也不一样, 所以 `golangci-lint` 使用 **nolint 标记来开关某些检查项**, 不同位置的 nolint 标记效果也会不一样。接下来我想向你介绍 nolint 的几种用法。

1. 忽略某一行所有 linter 的检查

```
1 var bad_name int //nolint
```

[复制代码](#)

2. 忽略某一行指定 linter 的检查, 可以指定多个 linter, 用逗号, 隔开。

```
1 var bad_name int //nolint:golint,unused
```

[复制代码](#)


3. 忽略某个代码块的检查。

```
1 //nolint
2 func allIssuesInThisFunctionAreExcluded() *string {
3     // ...
4 }
5
6 //nolint:govet
7 var (
8     a int
9     b int
10 )
```

[复制代码](#)

4. 忽略某个文件的指定 linter 检查。

在 package xx 上面一行添加//nolint注释。

 复制代码

```
1 //nolint:unparam
2 package pkg
3 ...
```

在使用 nolint 的过程中，有 3 个地方需要你注意。

首先，如果启用了 nolintlint，你就需要在//nolint后面添加 nolint 的原因// xxxx。

其次，你使用的应该是//nolint而不是// nolint。因为根据 Go 的规范，需要程序读取的注释 // 后面不应该有空格。

最后，如果要忽略所有 linter，可以用//nolint；如果要忽略某个指定的 linter，可以用//nolint:<linter1>,<linter2>。


golangci-lint 使用技巧

我在使用 golangci-lint 时，总结了一些经验技巧，放在这里供你参考，希望能够帮助你更好地使用 golangci-lint。

技巧 1：第一次修改，可以按目录修改。

如果你第一次使用 golangci-lint 检查你的代码，一定会有很多错误。为了减轻修改的压力，可以按目录检查代码并修改。这样可以有效减少失败条数，减轻修改压力。

当然，如果错误太多，一时半会儿改不完，想以后慢慢修改或者干脆不修复存量的 issues，那么你可以使用 golangci-lint 的 --new-from-rev 选项，只检查新增的代码，例如：

 复制代码

```
1 $ golangci-lint run --new-from-rev=HEAD~1
```


技巧 2：按文件修改，减少文件切换次数，提高修改效率。

如果有很多检查错误，涉及很多文件，建议先修改一个文件，这样就不用来回切换文件。可以通过 `grep` 过滤出某个文件的检查失败项，例如：

[复制代码](#)

```
1 $ golangci-lint run ./...|grep pkg/storage/redis_cluster.go
2 pkg/storage/redis_cluster.go:16:2: "github.com/go-redis/redis/v7" imported but
3 pkg/storage/redis_cluster.go:82:28: undeclared name: `redis` (typecheck)
4 pkg/storage/redis_cluster.go:86:14: undeclared name: `redis` (typecheck)
5 ...
```

技巧 3：把 `linters-setting.lll.line-length` 设置得大一些。

在 Go 项目开发中，为了易于阅读代码，通常会将变量名 / 函数 / 常量等命名得有意义，这样很可能导致每行的代码长度过长，很容易超过 `lll` linter 设置的默认最大长度 80。这里建议将 `linters-setting.lll.line-length` 设置为 120/240。

技巧 4：尽可能多地使用 `golangci-lint` 提供的 linter。

`golangci-lint` 集成了很多 linters，可以通过如下命令查看：

[复制代码](#)

```
1 $ golangci-lint linters
2 Enabled by your configuration linters:
3 deadcode: Finds unused code [fast: true, auto-fix: false]
4 ...
5 varcheck: Finds unused global variables and constants [fast: true, auto-fix: f
6
7 Disabled by your configuration linters:
8 asciicheck: Simple linter to check that your code does not contain non-ASCII i
9 ...
10 wsl: Whitespace Linter - Forces you to use empty lines! [fast: true, auto-fix:
```

这些 linter 分为两类，一类是默认启用的，另一类是默认禁用的。每个 linter 都有两个属性：

`fast` : `true/false` , 如果为 `true` , 说明该 `linter` 可以缓存类型信息, 支持快速检查。因为第一次缓存了这些信息, 所以后续的运行会非常快。

`auto-fix` : `true/false` , 如果为 `true` 说明该 `linter` 支持自动修复发现的错误; 如果为 `false` 说明不支持自动修复。

如果配置了 `golangci-lint` 配置文件, 则可以通过命令 `golangci-lint help linters` 查看在当前配置下启用和禁用了哪些 `linter`。 `golangci-lint` 也支持自定义 `linter` 插件, 具体你可以参考: [🔗 New linters](#)。

在使用 `golangci-lint` 的时候, 我们要尽可能多的使用 `linter`。使用的 `linter` 越多, 说明检查越严格, 意味着代码越规范, 质量越高。如果时间和精力允许, 建议打开 `golangci-lint` 提供的所有 `linter`。

技巧 5 : 每次修改代码后, 都要执行 `golangci-lint`。

每次修改完代码后都要执行 `golangci-lint` , 一方面可以及时修改不规范的地方, 另一方面可以减少错误堆积, 减轻后面的修改压力。

技巧 6 : 建议在根目录下放一个通用的 `golangci-lint` 配置文件。

在 `/` 目录下存放通用的 `golangci-lint` 配置文件, 可以让你不用为每一个项目都配置 `golangci-lint`。当你需要为某个项目单独配置 `golangci-lint` 时, 只需在该项目根目录下增加一个项目级别的 `golangci-lint` 配置文件即可。

总结

Go 项目开发中, 对代码进行静态代码检查是必要的操作。当前有很多优秀的静态代码检查工具, 但 `golangci-lint` 因为具有检查速度快、可配置、少误报、内置了大量 `linter` 等优点, 成为了目前最受欢迎的静态代码检查工具。

`golangci-lint` 功能非常强大, 支持诸如 `run`、`cache`、`completion`、`linters` 等命令。其中最常用的是 `run` 命令, `run` 命令可以通过以下方式进行静态代码检查:

```
1 $ golangci-lint run # 对当前目录及子目录下的所有Go文件进行静态代码检查
```

[📄 复制代码](#)

```
2 $ golangci-lint run dir1 dir2/... dir3/file1.go # 对指定的Go文件或者指定目录下的Go文件进行静态代码检查
3 $ golangci-lint run -c .golangci.yaml ./... # 根据指定配置文件，进行静态代码检查
4 $ golangci-lint run --no-config --disable-all -E errcheck ./... # 运行指定的 errcheck
5 $ golangci-lint run --no-config -D godot,errcheck # 禁止运行指定的godot,errcheck
```

此外，golangci-lint 还支持 `//nolint`、`//nolint:golint,unused` 等方式来减少误报。

最后，我分享了一些自己使用 golangci-lint 时总结的经验。例如：第一次修改，可以按目录修改；按文件修改，减少文件切换次数，提高修改效率；尽可能多地使用 golangci-lint 提供的 linter。希望这些建议对你使用 golangci-lint 有一定帮助。

课后练习

1. 执行 `golangci-lint linters` 命令，查看 golangci-lint 支持哪些 linter，以及这些 linter 的作用。
2. 思考下，如何在 golangci-lint 中集成自定义的 linter。

如果遇到任何疑问，欢迎你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta订阅后你可得 **24元** 现金奖励

👍 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 15 | 研发流程实战：IAM项目是如何进行研发流程管理的？

下一篇 17 | API 文档：如何生成 Swagger API 文档？

更多课程推荐

容器实战高手课

在实战中深入理解容器技术的本质

李程远

eBay 总监级工程师
云平台架构师



涨价倒计时 ⏰

今日订阅 **¥69**，7月20日涨价至 **¥129**

精选留言 (2)

💬 写留言



helloworld

2021-07-01

我使用vim+ale+golangci-lint做Go的静态代码检查，编写代码的过程中保存文件时自动触发检查，第一时间在文件内就给出了错误或警告的提示信息，直接按提示修复即可，基本上用不着在命令行执行golangci-lint命令。而且配置了package级别的检查，不存在因单文件检查出现的误报。

作者回复: 在ide中集成，每个文件都检查，非常好的思路。但Ci的时间也需要用到golangci-lint。



👍 6



demon

2021-07-03

如果使用的IDE像goland这种，自身就带静态代码检查吧？

展开 ∨

作者回复: goland也是需要配置的。

