

35 | 未雨绸缪：怎样通过静态与动态代码扫描保证代码质量？

2022-12-29 郑建勋 来自北京



天下无鱼

<https://shikey.com/>

《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 11:11 大小 10.22M



你好，我是郑建勋。

这节课让我们继续优化代码，让程序可配置化。然后通过静态与动态的代码扫描发现程序中存在的问题，让代码变得更加优雅。

micro 中间件

首先，让我们紧接上节课的 `go-micro` 框架，对代码进行优化，设置 `go-micro` 的中间件。如下，我们使用了 Go 函数闭包的特性，对请求进行了一层包装。中间件函数在接收到 GRPC 请求时，可以打印出请求的具体参数，方便我们排查问题。

复制代码

```
1 func logWrapper(log *zap.Logger) server.HandlerWrapper {  
2     return func(fn server.HandlerFunc) server.HandlerFunc {  
3         return func(ctx context.Context, req server.Request, rsp interface{}) error
```

```

4     log.Info("recieve request",
5         zap.String("method", req.Method()),
6         zap.String("Service", req.Service()),
7         zap.Reflect("request param:", req.Body()),
8     )
9     err := fn(ctx, req, rsp)
10    return err
11 }
12 }
13 }

```



接下来，使用 `micro.WrapHandler` 将中间件注入到 `micro.NewService` 中，这样就大功告成了。

 复制代码

```

1 service := micro.NewService(
2     ...
3     micro.WrapHandler(logWrapper(logger)),
4 )

```

当 GRPC 服务器收到请求之后，会打印出下面这样的请求信息。

 复制代码

```

1 {"level":"INFO","ts":"2022-11-28T00:29:28.287+0800","caller":"crawler/main.go:1

```

静态扫描

接下来，让我们用静态扫描把代码变得更优雅一些。

当前大多数公司采用的静态代码分析工具都是 `golangci-lint`。Linter 本来指的是一种分析源代码以此标记编程错误、代码缺陷和风格错误的工具，而 `golangci-lint` 就是集合多种 Linter 的工具。要查看 `golangci-lint` 支持的 Linter 列表，以及它启用 / 禁用了哪些 Linter，可以使用下面的命令：

 复制代码

```

1 > golangci-lint help linters

```

Go 语言定义了实现 Linter 的 API，它还提供了 golint 工具，golint 集成了几种常见的 Linter。在 [🔗 源码](#) 中，我们可以查看在标准库中如何实现典型的 Linter。



Linter 的实现原理是静态扫描代码的 AST（抽象语法树），Linter 的标准化意味着我们可以灵活实现自己的 Linters。不过，golangci-lint 里面其实已经集成了包括 golint 在内的众多 Linter，并且具有灵活的配置能力。所以如果你想自己写 Linter，我也建议你先了解一下 golangci-lint 现有的能力。

使用 golangci-lint 的第一步就是安装，不同环境下的安装方式你可以查看 [🔗 官方文档](#)。下面我来演示一下如何在本地使用 golangci-lint。最简单的方式就是执行下面的命令：

```
1 golangci-lint run
```

复制代码

它等价于：

```
1 golangci-lint run ./...
```

复制代码

我们也可以指定要分析的目录和文件：

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

复制代码

就像前面所说，golangci-lint 是众多 lint 的集合，要查看 golangci-lint 默认启动的 lint，可以运行下面的命令：

```
1 golangci-lint help linters
```

复制代码

可以看到，golangci-lint 内置了数十个 lint：

```

VX:1131052403 ~ » golangci-lint help linters
Enabled by default linters:
deadcode: Finds unused code [fast: false, auto-fix: false]
errcheck: Errcheck is a program for checking for unchecked errors in go programs. These unchecked errors can be critical bugs in some cases [fast: false, auto-fix: false]
gosimple (megacheck): Linter for Go source code that specializes in simplifying a code [fast: false, auto-fix: false]
govet (vet, vetshadow): Vet examines Go source code and reports suspicious constructs, such as Printf calls whose arguments do not align with the format string [fast: false, auto-fix: false]
ineffassign: Detects when assignments to existing variables are not used [fast: true, auto-fix: false]
staticcheck (megacheck): Staticcheck is a go vet on steroids, applying a ton of static analysis checks [fast: false, auto-fix: false]
structcheck: Finds unused struct fields [fast: false, auto-fix: false]
typecheck: Like the front-end of a Go compiler, parses and type-checks Go code [fast: false, auto-fix: false]
unused (megacheck): Checks Go code for unused constants, variables, functions and types [fast: false, auto-fix: false]
varcheck: Finds unused global variables and constants [fast: false, auto-fix: false]

Disabled by default linters:
asciicheck: Simple linter to check that your code does not contain non-ASCII identifiers [fast: true, auto-fix: false]
bidichk: Checks for dangerous unicode character sequences [fast: true, auto-fix: false]
bodyclose: checks whether HTTP response body is closed successfully [fast: false, auto-fix: false]
containedctx: containedctx is a linter that detects struct contained context.Context field [fast: true, auto-fix: false]
contextcheck: check the function whether use a non-inherited context [fast: false, auto-fix: false]
cyclap: checks function and package cyclomatic complexity [fast: false, auto-fix: false]
decoder: check declaration order and count of types, constants, variables and functions [fast: true, auto-fix: false]
depguard: Go linter that checks if package imports are in a list of acceptable packages [fast: true, auto-fix: false]
dogsled: Checks assignments with too many blank identifiers (e.g. x, _, _, := f()) [fast: true, auto-fix: false]
dupl: Tool for code clone detection [fast: true, auto-fix: false]
durationcheck: check for two durations multiplied together [fast: false, auto-fix: false]
errchkjson: Checks types passed to the json encoding functions. Reports unsupported types and optionally reports occasions, where the check for the returned error can be omitted [fast: false, auto-fix: false]
errname: Checks that sentinel errors are prefixed with the `Err` and error types are suffixed with the `Error`. [fast: false, auto-fix: false]
errorlint: errorlint is a linter for that can be used to find code that will cause problems with the error wrapping scheme introduced in Go 1.13. [fast: false, auto-fix: false]
execinquery: execinquery is a linter about query string checker in Query function which reads your Go src files and warning it finds [fast: false, auto-fix: false]
exhaustive: check exhaustiveness of enum switch statements [fast: false, auto-fix: false]
exhaustivestruct [deprecated]: Checks if all struct's fields are initialized [fast: false, auto-fix: false]
exhauststruct: Checks if all structure fields are initialized [fast: false, auto-fix: false]
exportloopref: checks for pointers to enclosing loop variables [fast: false, auto-fix: false]

```

为了能够灵活地配置 `golangci-lint` 的功能，我们需要新建对应的配置文件。`golangci-lint` 会依次查找当前目录下的文件，实现启用或禁用指定的 Linter，并指定不同 Linter 的行为。具体的配置说明你也可以查看 [官方文档](#)。

- `.golangci.yml`
- `.golangci.yaml`
- `.golangci.toml`
- `.golangci.json`

现在让我们在项目中创建 `.golangci.yml` 文件，具体的配置如下：


复制代码

```

1 run:
2     tests: false
3     skip-dirs:
4         - vendor
5
6 linters-settings:
7     funlen:
8         # Checks the number of lines in a function.
9         # If lower than 0, disable the check.
10        # Default: 60
11        lines: 120
12        # Checks the number of statements in a function.
13        # If lower than 0, disable the check.
14        # Default: 40
15        statements: -1

```

```
16
17 # list all linters by run `golangci-lint help linters`
18 linters:
19     enable-all: true
20     disable:
21         # gochecknoglobals: Checks that no globals are present in Go code
22         - gochecknoglobals
23         # gochecknoinit: Checks that no init functions are present in Go code
24         - gochecknoinit
25         # Checks that errors returned from external packages are wrapped
26         - wrapcheck
27         # checks that the length of a variable's name matches its scope
28         - varnamelen
29         # Checks the struct tags.
30         - tagliatelle
31         # An analyzer to detect magic numbers.
32         - gomnd
33         ...
```



天下无鱼
<https://shikey.com/>

其中，`run.tests` 选项表明我们不扫描测试文件，`run.skip-dirs` 表示扫描特定的文件夹，`linters-settings` 选项用于设置特定 Linter 的具体行为。`funlen linter` 用于限制函数的行数，默认的限制是 60 行，在这里我们根据项目的规范，将其配置为了 120 行。Linter 的特性你可以根据自己项目和团队的要求动态配置。

另外，`linters.enable-all` 表示默认开启所有的 Linter，`linters.disable` 表示禁用指定的 Linter。存在这个设定是因为在 `golangci-lint` 中有众多的 Linter，但是有些 Linter 相互冲突，有些已经过时，还有些并不适合你当前的项目。例如，`gochecknoglobals` 禁止使用全局变量，但是有时候我们在项目中确实需要全局变量，这时候就要根据实际需求来调整了。

添加完配置文件之后，执行 `golangci-lint run` 可以看到静态扫描之后的众多警告，如下图所示：

engine/schedule.go:360:1: block should not end with a whitespace (or comment) (wsl)

```

}
^
storage/sqlstorage/sqlstorage.go:27:2: declarations should never be cuddled (wsl)
    var err error
    ^
storage/sqlstorage/sqlstorage.go:89:2: defer statements should only be cuddled with expressions on same variable (wsl)
    defer func() {
    ^
main.go:40:2: declarations should never be cuddled (wsl)
    var p proxy.ProxyFunc
    ^
main.go:56:2: if statements should only be cuddled with assignments used in the if statement itself (wsl)
    if storage, err = sqlstorage.New(
    ^
main.go:132:2: only one cuddle assignment allowed before defer statement (wsl)
    defer cancel()
    ^
collect/collect.go:62:47: non-wrapping format verb for fmt.Errorf. Use `%w` to format errors (errorlint)
        return nil, fmt.Errorf("get url failed:%v", err)
                        ^
collect/collect.go:52:13: Transport, CheckRedirect, Jar are missing in Client (exhaustivestruct)
    client := &http.Client{
            ^
collect/request.go:48:10: Data is missing in DataCell (exhaustivestruct)
    res := &storage.DataCell{}
        ^
collect/request.go:62:12: Reqesrts, Items are missing in ParseResult (exhaustivestruct)
    result := ParseResult{}
        ^

```

天下无鱼
https://shikey.com/

有很多 Linter 对提高代码的质量是非常有帮助的。例如在下面这个例子中，golangci-lint 会打印出文件、行号、不符合规范的位置以及原因。其中，第一行最后的 (golint) 表明问题是由 golint 这个 lint 静态扫描出来的。这里它提示我们应该将 sqlUrl 的命名修改为 sqlURL。

复制代码

```

1 sqlldb/option.go:9:2: struct field `sqlUrl` should be `sqlURL` (golint)
2     sqlUrl string

```

再举个例子，这里，wsl linter 要求我们在特定的场景下在 continue 前方空一行，这样可以方便阅读。

复制代码

```

1 engine/schedule.go:242:4: branch statements should not be cuddled if block has
2     continue

```

我将项目中所有的代码都根据 Linter 的提示进行了修改，完整的代码见 [v0.3.1](#)。

动态扫描

不过，有一些问题是很难通过静态扫描发现的，例如数据争用问题。数据争用是并发系统中最常见且最难调试的错误类型之一。在下面这个例子中，两个协程共同访问了全局变量 `count`，乍看之下可能没有问题，但是这个程序其实是存在数据争用的，`count` 的结果也是不明确的。

复制代码

```
1 // race.go
2 var count = 0
3 func add() {
4     count++
5 }
6 func main() {
7     go add()
8     go add()
9 }
```

`count++` 操作看起来是一条指令，但是对 CPU 来说，需要先读取 `count` 的值，执行 `+1` 操作，再将 `count` 的值写回内存。大部分人期望的操作可能是下面这样：`R ← 0` 代表读取到 0，`w → 1` 代表写入 `count` 为 1；协程 1 写入数据 1 后，协程 2 再写入，`count` 最后的值为 2。

协程1	协程2
<code>R ← 0</code>	
<code>w → 1</code>	
	<code>R ← 1</code>
	<code>w → 2</code>

极客时间

但是由于 `count++` 并不是一条原子指令，情况开始变得复杂。如果执行的流程如下所示，那么 `count` 最后的值为 1。

协程1	协程2
$R \leftarrow 0$	
	$R \leftarrow 0$
$w \rightarrow 1$	
	$w \rightarrow 1$

这两种情况告诉我们，当两个协程发生数据争用时，结果是不可预测的，这会导致很多奇怪的错误。

再举一个 Go 语言中经典的数据争用错误。如下伪代码所示，在 Hash 表中，存储了我们希望存储到 Redis 数据库中的 data 数据。但是在 Go 语言中使用 Range 时，变量 k 是一个堆上地址不变的对象，该地址存储的值会随着 Range 遍历而发生变化。

如果此时我们将变量 k 的地址放入协程 save，以此提供并发存储而不堵塞程序，那么最后的结果可能是，后面的数据会覆盖前面的数据，同时导致一些数据没有被存储，并且每一次完成存储的数据也是不明确的。

 复制代码

```
1 func save(g *data){
2     saveToRedis(g)
3 }
4 func main() {
5     var a map[int]data
6     for _, k := range a{
7         go save(&k)
8     }
9 }
```

数据争用可以说是高并发程序中最难排查的问题，原因在于它的结果是不明确的，而且可能只在在特定的条件下出错，这导致很难复现相同的错误，在测试阶段也不一定能测试出问题。

Go 1.1 后提供了强大的检查工具 race 来排查数据争用问题。如下所示，race 可以用在多个 Go 指令中。当检测器在程序中找到数据争用时，将打印报告。这个报告包含发生 race 冲突的协程栈，以及此时正在运行的协程栈。


```

1 $ go test -race mypkg
2 $ go run -race mysrc.go
3 $ go build -race mycmd
4 $ go install -race mypkg

```



如果对上面这个例子的 `race.go` 文件执行 `go run -race`，程序在运行时会直接报错，如下所示。从报错后输出的栈帧信息中可以看出发生冲突的具体位置。

```

1 » go run -race race.go
2 =====
3 WARNING: DATA RACE
4 Read at 0x00000115c1f8 by goroutine 7:
5 main.add()
6 bookcode/concurrency_control/race.go:5 +0x3a
7 Previous write at 0x00000115c1f8 by goroutine 6:
8 main.add()
9 bookcode/concurrency_control/race.go:5 +0x56

```

`Read at` 表明读取发生在 `race.go` 文件的第 5 行，而 `Previous write` 表明前一个写入也发生在 `race.go` 文件的第 5 行，这样我们就可以非常快速地发现并定位数据争用问题了。

不过，竞争检测也有一定成本，它因程序的不同而有所差异。对于典型的程序来说，内存使用量可能增加 5~10 倍，执行时间会增加 2~20 倍。同时，竞争检测器还会为当前每个 `defer` 和 `recover` 语句额外分配 8 字节，在 `Goroutine` 退出前，这些额外分配的字节不会被回收。这意味着，如果有一个长期运行的 `Goroutine`，而且定期有 `defer` 和 `recover` 调用，那么程序的内存使用量可能无限增长（有关 `race` 工具的原理你可以参考《Go 底层原理剖析》）。

配置文件

看完静态和动态的代码扫描，我们接着来让代码可配置化，这是我们项目一直没有实现的功能。很多人可能直接会书写 `JSON`、`TOML` 等配置文件并在程序启动时读取配置文件。不过一个优秀的处理配置的库要考虑更多内容。`go-micro` 的配置库提供了下面这几种能力。

• 动态配置

大多数程序在初始化时会读取应用程序配置，之后就一直保持静态状态。如果需要更改配置，则需要重新启动应用程序，这有时候会显得比较繁琐。而动态配置通过监听配置的变化

化，实现了动态化的配置。

- **支持多种后端数据源**

它可以支持文件、flags、环境变量、甚至 etcd 等数据源获取源数据。



- **支持多种数据格式的解析**

它可以解析包括 JSON、TOML、YML 在内的多种数据源格式。

- **可合并**

它支持将多个后端数据源读取到的数据合并到一起进行处理。

- **安全性**

当配置文件不存在时，go-micro 的配置库支持返回默认的数据。

关于 go-micro 代码的设计你可以参考 [这篇文章](#)。

我们举一个简单的例子来说明 go-micro 配置库的使用方式。假设我们有配置文件 config.json:

复制代码

```
1 {
2   "hosts": {
3     "database": {
4       "address": "10.0.0.2",
5       "port": 3306
6     },
7     "cache": {
8       "address": "10.0.0.2",
9       "port": 6379
10    }
11  }
12 }
```

获取配置文件的实例代码如下:

复制代码

```
1 package main
2
3 import (
4   ...
5   "go-micro.dev/v4/config"
6   "go-micro.dev/v4/config/source/file"
7 )
```

```

8 func main() {
9
10 // 导入数据
11 err := config.Load(file.NewSource(
12     file.WithPath("config.json"),
13 ))
14 if err != nil {
15     fmt.Println(err)
16 }
17 type Host struct {
18     Address string `json:"address"`
19     Port    int    `json:"port"`
20 }
21
22 var host Host
23 // 获取hosts.database下的数据，并解析为host结构
24 config.Get("hosts", "database").Scan(&host)
25
26 fmt.Println(host)
27
28 w, err := config.Watch("hosts", "database")
29 if err != nil {
30     fmt.Println(err)
31 }
32
33 // 等待配置文件更新
34 v, err := w.Next()
35 if err != nil {
36     fmt.Println(err)
37 }
38
39 v.Scan(&host)
40 fmt.Println(host)
41 }

```



在这里，`config.Load` 用于导入某一个数据源中的 `config.json` 文件，`config.Get` 用于获得某一个层级下的数据，`Scan` 函数用于将数据解析到结构体中。`config.Watch` 函数用于监听指定的配置文件更新。

在项目中，我们使用 [TOML](#) 来作为配置文件。`TOML` 相比 `JSON` 文件的优势在于，能够书写注释，阅读起来相对清晰，但是它不适合表示一些复杂的层次结构。要想在项目中读取 `TOML` 数据并将其转化为类似 `JSON` 的层次结构，需要导入 [TOML 插件库](#) 并做额外的处理：

复制代码

```

1 enc := toml.NewEncoder()
2 cfg, err := config.NewConfig(config.WithReader(json.NewReader(reader.WithEncode

```

```
3 err = cfg.Load(file.NewSource(
4     file.WithPath("config.toml"),
5     source.WithEncoder(enc),
6 ))
```



之前我们有许多项目的配置是写死在代码中的，例如数据库的地址、etcd 的地址、GRPC 服务器的监听地址，以及超时时间、日志级别等等。现在我们需要将这些配置迁移到配置文件中，实现可配置化。

项目中配置文件的处理方法我这里就不再赘述了，具体你可以查看 [@v0.3.2 分支](#)。

复制代码

```
1 logLevel = "debug"
2
3 Tasks = [
4     {Name = "douban_book_list",WaitTime = 2,Reload = true,MaxDepth = 5,Fetcher
5     {Name = "xxx"},
6 ]
7
8 [fetcher]
9 timeout = 3000
10 proxy = ["<http://127.0.0.1:8888>", "<http://127.0.0.1:8888>"]
11
12 [storage]
13 sqlURL = "root:123456@tcp(127.0.0.1:3326)/crawler?charset=utf8"
14
15 [GRPCServer]
16 HTTPListenAddress = ":8080"
17 GRPCListenAddress = ":9090"
18 ID = "1"
19 RegistryAddress = ":2379"
20 RegisterTTL = 60
21 RegisterInterval = 15
22 ClientTimeOut    = 10
23 Name = "go.micro.server.worker"
```

Makefile

将配置文件准备好之后，我们就可以构建并运行程序了。在构建程序时，输入一长串的构建命令比较繁琐。为了解决这样的问题，我们可以把一些构建的脚本写入 Makefile 文件中。如下所示：

复制代码

```

1 VERSION := v1.0.0
2
3 LDFLAGS = -X "main.BuildTS=$(shell date -u '+%Y-%m-%d %I:%M:%S')"
4 LDFLAGS += -X "main.GitHash=$(shell git rev-parse HEAD)"
5 LDFLAGS += -X "main.GitBranch=$(shell git rev-parse --abbrev-ref HEAD)"
6 LDFLAGS += -X "main.Version=${VERSION}"
7
8 ifeq ($(gorace), 1)
9     BUILD_FLAGS--race
10 endif
11
12 build:
13     go build -ldflags '$(LDFLAGS)' $(BUILD_FLAGS) main.go
14
15 lint:
16     golangci-lint run ./...

```

天下无鱼
https://shikey.com/

其中，**build** 下的命令就是构建程序的命令。在这段命令中，**LDFLAGS** 为编译时的一些选项，我们在编译时注入了程序的版本号、分支、构建时间、**git commit** 号等信息。这些信息会注入到 **main.go** 中的全局变量中。在 **main.go** 中，我们还要进行一些配套的处理，用来打印一些程序的版本信息。

复制代码

```

1 // Version information.
2 var (
3     BuildTS    = "None"
4     GitHash    = "None"
5     GitBranch  = "None"
6     Version    = "None"
7 )
8
9 func GetVersion() string {
10     if GitHash != "" {
11         h := GitHash
12         if len(h) > 7 {
13             h = h[:7]
14         }
15         return fmt.Sprintf("%s-%s", Version, h)
16     }
17     return Version
18 }
19
20 // Printer print build version
21 func Printer() {
22     fmt.Println("Version: ", GetVersion())
23     fmt.Println("Git Branch: ", GitBranch)
24     fmt.Println("Git Commit: ", GitHash)
25     fmt.Println("Build Time (UTC): ", BuildTS)

```

```
26 }
27
28 var (
29     PrintVersion = flag.Bool("version", false, "print the version of this build")
30 )
31
32 func main(){
33     flag.Parse()
34     if *PrintVersion {
35         Printer()
36         os.Exit(0)
37     }
38 }
```



如下所示。当我们执行 `make build` 构建可运行程序，并传递 `-version` 运行参数时，就会打印出程序的版本信息了：

 复制代码

```
1 > make build
2 > ./main -version
3
4 Version:          v1.0.0-ed89d91
5 Git Branch:       master
6 Git Commit:       ed89d91d03834fe85b1ca7f74f0cca305b8e516a
7 Build Time (UTC): 2022-11-30 04:52:45
```

同时在 `Makefile` 中，`BUILD_FLAGS` 表示构建可执行文件的参数。当我们设置环境变量 `gorace=1` 时，`go build` 会将 `race` 工具编译到程序中。最后我们会看到完整的构建命令：

 复制代码

```
1 » export gorace=1
2 » make build
3 go build -ldflags '-X "main.BuildTS=2022-12-03 05:48:59" -X "main.GitHash=e73f1
```

总结

这节课，我们使用了静态与动态的代码扫描来发现代码的 `Bug` 和不太规范的代码，这可以帮助开发者遵循团队的编码规范，书写出更优雅的程序。

同时我们还看到了如何用 `go-micro` 的 `config` 库来更灵活地对配置文件进行管理。它不仅提供了配置化的能力，还实现了动态配置、配置合并的能力，在这个过程中，我们看到了功能全面的配置管理需要考虑的因素。




最后，通过书写 `Makefile` 文件，我们可以执行预先定义好的脚本，更快、更优雅地书写项目代码。

课后题

1. `golangci-lint` 中包含了众多的 `lint`，其中有些 `lint` 的功能是过时的，重复的。那么我们在项目中应该让哪些 `lint` 生效呢？
2. 配置文件、`JSON` 格式与 `TOML` 格式分别适用于哪一种场景？

欢迎你在留言区与我交流讨论，我们下节课见！

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 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 34 | 服务注册与监听：Worker节点与etcd交互

下一篇 36 | 测试的艺术：依赖注入、表格测试与压力测试

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。