



下载APP



加餐 | 聊聊Go 1.17版本的那些新特性

2021-12-17 Tony Bai

《Tony Bai · Go语言第一课》

课程介绍 >



讲述 : Tony Bai

时长 18:14 大小 16.70M



你好，我是 Tony Bai。

现在是 2021 年 12 月，万众期盼的潜力网红版本 Go 1.18 的开发已经冻结，Go 核心开发团队正在紧锣密鼓地修 bug。我们已经可以开始期待 2022 年的 2 月份，Go 1.18 将携带包括泛型语法的大批新特性赶来。不过当下我们不能“舍近求远”，今年 8 月中旬 Go 核心团队发布的 [Go 1.17 版本](#) 才是当下最具统治力的 Go 社区网红，它的影响力依旧处于巅峰。

根据我们在 [第 3 讲](#) 中提到的 Go 版本选择策略，我估计很多 Go 开发者都还没切换到 Go 1.17 版本，没有亲自体验过 Go 1.17 新特性带来的变化；还有一些 Go 开发者虽然已经升级到 Go 1.17 版本，但也仅限于对 Go 1.17 版本的基本使用，可能还不是很清楚 Go 1.17 版本中究竟有哪些新特性，以及这些新特性会带给他们哪些好处。



所以今天这讲，我们就来聊聊 Go 1.17 版本中的新特性，目的是让那些没用过 Go 1.17 版本，或者用过 Go 1.17 版本但还不知道它新特性变化的 Go 开发者，对 Go 1.17 有一个全面的了解。

Go 1.17 版本中的新特性很多，在这里我就不一一列举了，我仅挑几个有代表性的、重要的新特性和你好好聊聊。这里会包括新的语法特性、Go Module 机制变化，以及 Go 编译器与运行时方面的变化。


新的语法特性

在🔗第 2 讲学习 Go 语言设计哲学时，我们知道了 Go 语言的设计者们在语言设计之初，就**拒绝了走语言特性融合的道路**，选择了“做减法”，并致力于打造一门**简单**的编程语言。从诞生到现在，Go 语言自身语法特性变化很小，甚至可以用**屈指可数**来形容，因此新语法特性对于 Gopher 来说属于“稀缺品”。这也直接导致了每次 Go 新版本发布，我们都要先看看语法特性是否有变更，每个新加入语法特性都值得我们投入更多关注，去深入研究。

不出所料，Go 1.17 版本在语法特性方面仅仅做了一处增强，那就是**支持切片转换为数组指针**。下面我们详细来看一下。


支持将切片转换为数组指针

在🔗第 15 讲中，我们对 Go 中的切片做了系统全面的讲解。我们知道，通过数组切片化，我们可以将一个数组转换为切片。转换后，数组将成为转换后的切片的底层数组，通过切片，我们可以直接改变数组中的元素，就像下面代码这样：

 复制代码

```
1 a := [3]int{11, 12, 13}
2 b := a[:] // 通过切片化将数组a转换为切片b
3 b[1] += 10
4 fmt.Printf("%v\n", a) // [11 22 13]
```

但反过来就不行了。在 Go 1.17 版本之前，Go 并不支持将切片再转换回数组类型。当然，如果你非要这么做也不是没有办法，我们可以通过 `unsafe` 包以不安全的方式实现这样的转换，如下面代码所示：

 复制代码

```
1 b := []int{11, 12, 13}
2 var p = (*[3]int)(unsafe.Pointer(&b[0]))
3 p[1] += 10
4 fmt.Printf("%v\n", b) // [11 22 13]
```

但是 unsafe 包，正如其名，它的安全性没有得到编译器和 runtime 层的保证，只能由开发者自己保证，所以我建议 Gopher 们在通常情况下不要使用。


2009 年末，也就是 Go 语言宣布开源后不久，[@Roger Peppe](#)便提出一个 [issue](#)，希望 Go 核心团队考虑在语法层面补充从切片到数组的转换语法，同时希望这种转换以及转换后的数组在使用时的下标边界，能得到编译器和 runtime 的协助检查。**十二年后**这个 issue 终于被 Go 核心团队接受，并在 Go 1.17 版本加入到 Go 语法特性当中。

所以，在 Go 1.17 版本中，我们可以像下面代码这样将一个切片转换为数组类型指针，不用再借助 unsafe 包的“黑魔法”了：

 复制代码

```
1 b := []int{11, 12, 13}
2 p := (*[3]int)(b) // 将切片转换为数组类型指针
3 p[1] = p[1] + 10
4 fmt.Printf("%v\n", b) // [11 22 13]
```

不过，这里你要注意的，Go 会通过运行时而不是编译器去对这类切片到数组指针的转换代码做检查，如果发现越界行为，就会触发运行时 panic。Go 运行时实施检查的一条原则就是“**转换后的数组长度不能大于原切片的长度**”，注意这里是切片的长度（len），而不是切片的容量（cap）。于是你会看到，下面的转换有些合法，有些非法：

 复制代码

```
1 var b = []int{11, 12, 13}
2 var p = (*[4]int)(b) // cannot convert slice with length 3 to pointer to array
3 var p = (*[0]int)(b) // ok, *p = []
4 var p = (*[1]int)(b) // ok, *p = [11]
5 var p = (*[2]int)(b) // ok, *p = [11, 12]
6 var p = (*[3]int)(b) // ok, *p = [11, 12, 13]
7 var p = (*[3]int)(b[:1]) // cannot convert slice with length 1 to pointer to a
```

另外，nil 切片或 cap 为 0 的 empty 切片都可以被转换为一个长度为 0 的数组指针，比如：

[📄 复制代码](#)

```
1 var b1 []int // nil切片
2 p1 := (*[0]int)(b1)
3 var b2 = []int{} // empty切片
4 p2 := (*[0]int)(b2)
```

说完了 Go 语法特性的变化后，我们再来看看 Go Module 构建模式在 Go 1.17 中的演进。

Go Module 构建模式的变化

自从 Go 1.11 版本引入 Go Module 构建模式以来，每个 Go 大版本发布时，Go Module 都会有不少的积极变化，Go 1.17 版本也不例外。

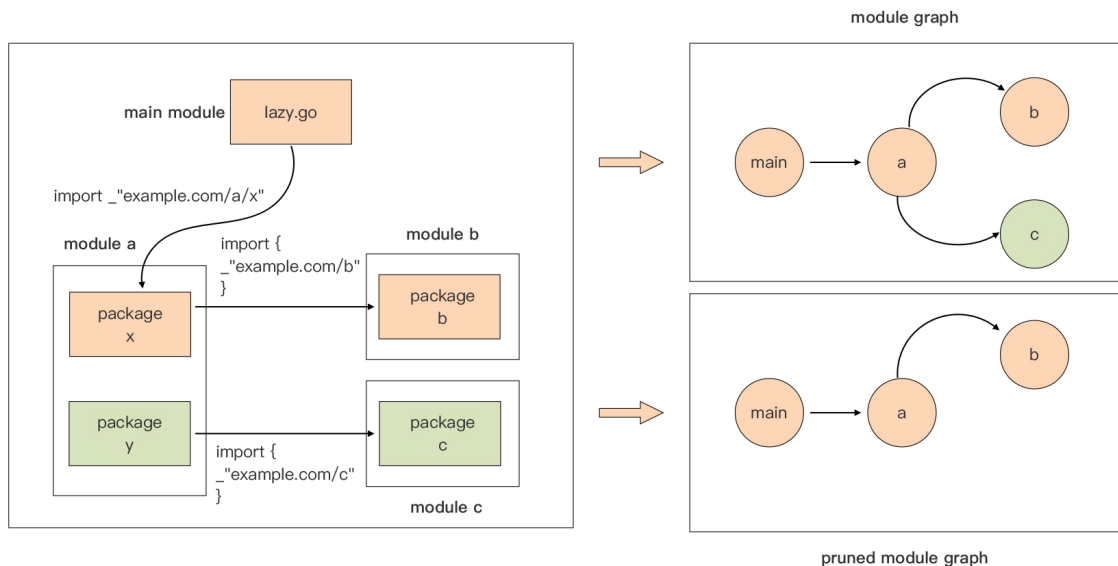
修剪的 module 依赖图

Go 1.17 版本中，Go Module 最重要的一个变化就是 **pruned module graph**，即**修剪的 module 依赖图**。要理解这个概念，我们先来讲什么是**完整 module 依赖图**。

在 Go 1.17 之前的版本中，某个 module 的依赖图是由这个 module 的直接依赖以及所有间接依赖组成的。这样，无论某个间接依赖是否真正为原 module 的构建做出贡献，Go 命令在解决依赖时都会读取每个依赖的 go.mod，包括那些没有被真正使用到的 module，这样形成的 module 依赖图被称为**完整 module 依赖图 (complete module graph)**。

从 Go 1.17 的版本开始，Go 不再使用“完整 module 依赖图”，而是引入了 pruned module graph，也就是修剪的 module 依赖图。修剪的 module 依赖图就是在完整 module 依赖图的基础上，将那些对构建完全没有“贡献”的间接依赖 module 修剪掉后，剩余的依赖图。使用修剪后的 module 依赖图进行构建，有助于避免下载或阅读那些不必要的 go.mod 文件，这样 Go 命令可以不去获取那些不相关的依赖关系，从而在日常开发中节省时间。

这么说还是比较抽象，我们用下图中的例子来详细解释一下 module 依赖图修剪的原理。



上图中的例子来自于 Go 1.17 源码中的

src/cmd/go/testdata/script/mod_lazy_new_import.txt，通过执行 [txtar 工具](#)，我们可以将这个 txt 转换为 mod_lazy_new_import.txt 中描述的示例结构，转换命令为：

```
txtar -x <
```

```
$GOROOT/src/cmd/go/testdata/script/mod_lazy_new_import.txt.
```

在这个示例中，main module 中的 lazy.go 导入了 module a 的 package x，后者则导入了 module b 中的 package b。并且，module a 还有一个 package y，这个包导入了 module c 的 package c。通过 go mod graph 命令，我们可以得到 main module 的完整 module 依赖图，也就是上图的右上角的那张。

现在问题来了！package y 是因为自身是 module a 的一部分而被 main module 依赖的，它自己没有为 main module 的构建做出任何“代码级贡献”，同理，package y 所依赖的 module c 亦是如此。但是在 Go 1.17 之前的版本中，如果 Go 编译器找不到 module c，那么 main module 的构建也会失败，这会让开发者们觉得不够合理！

现在，我们直观地看一下在 Go 1.16.5 下，这个示例的 go.mod 是怎样的：

复制代码

```
1 module example.com/lazy
2
3 go 1.15
```

```
4 require example.com/a v0.1.0
5
6 replace (
7     example.com/a v0.1.0 => ./a
8     example.com/b v0.1.0 => ./b
9     example.com/c v0.1.0 => ./c1
10    example.com/c v0.2.0 => ./c2
11 )
12
```

我们只需要关注 `require` 块中的内容就可以了，下面的 `replace` 块主要是为了示例能找到各种依赖 `module` 而设置的。

我们知道，在 Go 1.16 及以前支持 Go Module 的版本建立的 Go Module 中，在 `go.mod` 经过 `go mod tidy` 后，`require` 块中保留的都是 main module 的直接依赖，[在某些情况下，也会记录 indirect 依赖](#)，这些依赖会在行尾用 `indirect` 指示符明示。但在这里，我们看不到 main module 的间接依赖以及它们的版本，我们可以用 `go mod graph` 来查看 module 依赖图：

[复制代码](#)

```
1 $go mod graph
2 example.com/lazy example.com/a@v0.1.0
3 example.com/a@v0.1.0 example.com/b@v0.1.0
4 example.com/a@v0.1.0 example.com/c@v0.1.0
```

这个 `go mod graph` 的输出，和我们在上面图中右上角画的 module graph 是一致的。此时，如果我们将 `replace` 中的第三行（`example.com/c v0.1.0 => ./c1` 这一行）删除，也就是让 Go 编译器找不到 module `c@v0.1.0`，那么我们构建 main module 时就会得到下面的错误提示：

[复制代码](#)

```
1 $go build
2 go: example.com/a@v0.1.0 requires
3     example.com/c@v0.1.0: missing go.sum entry; to add it:
4     go mod download example.com/c
```

现在我们将执行权限交给 Go 1.17 看看会怎样！

这个时候，我们需要对 go.mod 做一些修改，也就是将 go.mod 中的 go 1.15 改为 go 1.17，这样 Go 1.17 才能起到作用。接下来，我们执行 go mod tidy，让 Go 1.17 重新构建 go.mod：

[复制代码](#)

```
1 $go mod tidy
2 $cat go.mod
3
4 module example.com/lazy
5
6 go 1.17
7
8 require example.com/a v0.1.0
9
10 require example.com/b v0.1.0 // indirect
11
12 replace (
13     example.com/a v0.1.0 => ./a
14     example.com/b v0.1.0 => ./b
15     example.com/c v0.1.0 => ./c1
16     example.com/c v0.2.0 => ./c2
17 )
```

我们看到执行 go mod tidy 之后，go.mod 发生了变化：增加了一个 require 语句块，记录了 main module 的间接依赖，也就是 module b@v0.10。

现在，我们也同样将 go.mod replace 块中的第三行（example.com/c v0.1.0 => ./c1 这一行）删除，再来用 go 1.17 构建一次 main module。

这一次我们没有看到 Go 编译器的错误提示。也就是说在构建过程中，Go 编译器看到的 main module 依赖图中并没有 module c@v0.1.0。这是因为 module c 并没有为 main module 的构建提供“代码级贡献”，所以 Go 命令把它从 module 依赖图中剪除了。这一次，Go 编译器使用的真实的依赖图是上图右下角的那张。这种将那些对构建完全没有“贡献”的间接依赖 module 从构建时使用的依赖图中修剪掉的过程，就被称为 **module 依赖图修剪 (pruned module graph)**。

但 **module 依赖图修剪**也带来了一个副作用，那就是 **go.mod 文件 size 的变大**。因为从 Go 1.17 版本开始，每次调用 go mod tidy，Go 命令都会对 main module 的依赖做一次深度扫描（deepening scan），并将 main module 的所有直接和间接依赖都记录在

go.mod 中。考虑到依赖的内容较多，go 1.17 会将直接依赖和间接依赖分别放在多个不同的 require 块中。

所以，在 Go 1.17 版本中，go.mod 中存储了 main module 的所有依赖 module 列表，这似乎也是 Go 项目第一次有了项目依赖的完整列表。不知道会不会让你想起其他主流语言构架系统中的那个 lock 文件呢？虽然 go.mod 并不是 lock 文件，但有了完整依赖列表，至少我们可以像其他语言的 lock 文件那样，知晓当前 Go 项目所有依赖的精确版本了。

在讲解下一个重要变化之前，我还要提一点小变化，那就是**在 Go 1.17 版本中，go get 已经不再被用来安装某个命令的可执行文件了**。如果你依旧使用 go get 安装，Go 命令会提示错误。这也是很多同学在学习我们课程的入门篇时经常会问的一个问题。

新版本中，我们需要使用 go install 来安装，并且使用 go install 安装时还要用 @vx.y.z 明确要安装的命令的二进制文件的版本，或者是使用 @latest 来安装最新版本。

除了 Go 语法特性与 Go Module 有重要变化之外，Go 编译器的变化对 Go 程序的构建与运行影响同样十分巨大，我们接下来就来看一下 Go 1.17 在这方面的重要变化。

Go 编译器的变化

在 Go1.17 版本，Go 编译器的变化主要是在 AMD64 架构下实现了基于寄存器的调用惯例，以及新引入了 //go:build 形式的构建约束指示符。现在我们就来分析下这两点。

基于寄存器的调用惯例

Go 1.17 版本中，Go 编译器最大的变化是在 AMD64 架构下率先实现了 **从基于堆栈的调用惯例到基于寄存器的调用惯例的切换**。

所谓“调用惯例 (calling convention) ”，是指调用方和被调用方对于函数调用的一个明确的约定，包括函数参数与返回值的传递方式、传递顺序。只有双方都遵守同样的约定，函数才能被正确地调用和执行。如果不遵守这个约定，函数将无法正确执行。

Go 1.17 版本之前，Go 采用基于栈的调用约定，也就是说函数的参数与返回值都通过栈来传递，这种方式的优点是实现简单，不用担心底层 CPU 架构寄存器的差异，适合跨平台，

但缺点就是牺牲了一些性能。

我们都知道，寄存器的访问速度是要远高于内存的。所以，现在大多数平台上的大多数语言实现都使用基于寄存器的调用约定，通过寄存器而不是内存传递函数参数和返回结果，并指定一些寄存器为调用保存寄存器，允许函数在不同的调用中保持状态。Go 核心团队决定在 1.17 版本向这些语言看齐，并在 AMD64 架构下率先实现基于寄存器的调用惯例。

我们可以在 Go 1.17 的版本发布说明文档中看到，切换到基于寄存器的调用惯例后，一组有代表性的 Go 包和程序的基准测试显示，Go 程序的运行性能提高了约 5%，二进制文件大小典型减少约 2%。

那我们这里就来实测一下，看看是否真的能提升那么多。下面是一个使用多种方法进行字符串连接的 benchmark 测试源码：

[复制代码](#)

```
1 var sl []string = []string{
2     "Rob Pike ",
3     "Robert Griesemer ",
4     "Ken Thompson ",
5 }
6
7 func concatStringByOperator(sl []string) string {
8     var s string
9     for _, v := range sl {
10         s += v
11     }
12     return s
13 }
14
15 func concatStringByPrintf(sl []string) string {
16     var s string
17     for _, v := range sl {
18         s = fmt.Sprintf("%s%s", s, v)
19     }
20     return s
21 }
22
23 func concatStringByJoin(sl []string) string {
24     return strings.Join(sl, "")
25 }
26
27 func concatStringByStringsBuilder(sl []string) string {
28     var b strings.Builder
29     for _, v := range sl {
```

```
30     b.WriteString(v)
31 }
32 return b.String()
33 }
34
35 func concatStringByStringsBuilderWithInitSize(sl []string) string {
36     var b strings.Builder
37     b.Grow(64)
38     for _, v := range sl {
39         b.WriteString(v)
40     }
41     return b.String()
42 }
43
44 func concatStringByBytesBuffer(sl []string) string {
45     var b bytes.Buffer
46     for _, v := range sl {
47         b.WriteString(v)
48     }
49     return b.String()
50 }
51
52 func concatStringByBytesBufferWithInitSize(sl []string) string {
53     buf := make([]byte, 0, 64)
54     b := bytes.NewBuffer(buf)
55     for _, v := range sl {
56         b.WriteString(v)
57     }
58     return b.String()
59 }
60
61 func BenchmarkConcatStringByOperator(b *testing.B) {
62     for n := 0; n < b.N; n++ {
63         concatStringByOperator(sl)
64     }
65 }
66
67 func BenchmarkConcatStringByPrintf(b *testing.B) {
68     for n := 0; n < b.N; n++ {
69         concatStringByPrintf(sl)
70     }
71 }
72
73 func BenchmarkConcatStringByJoin(b *testing.B) {
74     for n := 0; n < b.N; n++ {
75         concatStringByJoin(sl)
76     }
77 }
78
79 func BenchmarkConcatStringByStringsBuilder(b *testing.B) {
80     for n := 0; n < b.N; n++ {
81         concatStringByStringsBuilder(sl)
```


```

82     }
83 }
84
85 func BenchmarkConcatStringByStringsBuilderWithInitSize(b *testing.B) {
86     for n := 0; n < b.N; n++ {
87         concatStringByStringsBuilderWithInitSize(sl)
88     }
89 }
90
91 func BenchmarkConcatStringByBytesBuffer(b *testing.B) {
92     for n := 0; n < b.N; n++ {
93         concatStringByBytesBuffer(sl)
94     }
95 }
96
97 func BenchmarkConcatStringByBytesBufferWithInitSize(b *testing.B) {
98     for n := 0; n < b.N; n++ {
99         concatStringByBytesBufferWithInitSize(sl)
100    }
101 }

```

我们使用 Go 1.16.5 和 Go 1.17 分别运行这个 Benchmark 示例，结果如下：

Go 1.16.5 :


 复制代码

```

1  $go test -bench .
2  goos: darwin
3  goarch: amd64
4  pkg: github.com/bigwhite/demo
5  cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
6  BenchmarkConcatStringByOperator-8          12132355      91.5
7  BenchmarkConcatStringByPrintf-8            2707862      445.1
8  BenchmarkConcatStringByJoin-8              24101215     50.8
9  BenchmarkConcatStringByStringsBuilder-8    11104750     124.4
10 BenchmarkConcatStringByStringsBuilderWithInitSize-8 24542085     48.2
11 BenchmarkConcatStringByBytesBuffer-8       14425054     77.7
12 BenchmarkConcatStringByBytesBufferWithInitSize-8 20863174     49.0
13 PASS
14 ok      github.com/bigwhite/demo    9.166s

```

Go 1.17 :

 复制代码

```

1  $go test -bench .

```

```

2 goos: darwin
3 goarch: amd64
4 pkg: github.com/bigwhite/demo
5 cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
6 BenchmarkConcatStringByOperator-8          13058850      89.4
7 BenchmarkConcatStringBySprintf-8           2889898      410.1
8 BenchmarkConcatStringByJoin-8              25469310      47.1
9 BenchmarkConcatStringByStringsBuilder-8     13064298      92.3
10 BenchmarkConcatStringByStringsBuilderWithInitSize-8 29780911      41.1
11 BenchmarkConcatStringByBytesBuffer-8       16900072      70.2
12 BenchmarkConcatStringByBytesBufferWithInitSize-8 27310650      43.9
13 PASS
14 ok      github.com/bigwhite/demo    9.198s

```

我们可以看到，相对于 Go 1.16.5 跑出的结果，Go 1.17 在每一个测试项上都有小幅的性能提升，有些性能提升甚至达到 10% 左右（以 BenchmarkConcatStringBySprintf 为例，它的性能提升为 $(445.1-410.1)/445.1=7.8\%$ ）。也就是说你的 Go 源码使用 Go 1.17 版本重新编译一下，就能获得大约 5% 的性能提升，这种新版本带来的性能的“自然提升”显然是广大 Gopher 乐意看到的。

我们再来看看编译后的 Go 二进制文件的 Size 变化。我们以一个自有的 1w 行左右代码的 Go 程序为例，分别用 Go 1.16.5 和 Go 1.17 进行编译，得到的结果如下：

[复制代码](#)

```

1 -rwxr-xr-x  1 tonybai  staff  7264432  8 13 18:31 myapp-go1.16.5*
2 -rwxr-xr-x  1 tonybai  staff  6934352  8 13 18:32 myapp-go1.17*

```

我们看到，Go 1.17 编译后的二进制文件大小相比 Go 1.16.5 版本减少了约 4%，比 Go 官方文档发布的平均效果还要好上一些。

而且，Go 1.17 发布说明也提到了：**改为基于寄存器的调用惯例后，绝大多数程序不会受到影响**。只有那些之前就已经违反 unsafe.Pointer 的使用规则的代码可能会受到影响，比如不遵守 unsafe 规则通过 unsafe.Pointer 访问函数参数，或者依赖一些像比较函数代码指针的未公开的行为。

//go:build 形式的构建约束指示符

此外，Go 编译器还在 Go 1.17 中引入了 //go:build 形式的构建约束指示符，以替代原先易错的 // +build 形式。

在 Go 1.17 之前，我们可以通过在源码文件头部放置 `// +build` 构建约束指示符来实现构建约束，但这种形式十分易错，并且它并不支持 `&&` 和 `||` 这样的直观的逻辑操作符，而是用逗号、空格替代，这里你可以看下原 `// +build` 形式构建约束指示符的用法及含义：

build tags	含义
<code>// +build tag1 tag2</code>	tag1 OR tag2
<code>// +build tag1,tag2</code>	tag1 AND tag2
<code>// +build !tag1</code>	NOT tag1
<code>// +build tag1
 // +build tag2</code>	多行build tag代表: tag1 AND tag2
<code>// +build tag1,tag2 tag3,!tag4</code>	复杂build tag: (tag1 AND tag2) OR (tag3 AND (NOT tag4))

但这种与程序员直觉“有悖”的形式让 Gopher 们十分痛苦，于是 Go 1.17 回归“正规正轨”，引入了 `//go:build` 形式的构建约束指示符。一方面，这可以与源文件中的其他指示符保持形式一致，比如 `//go:nosplit`、`//go:norace`、`//go:noinline`、`//go:generate` 等。

另一方面，新形式将支持 `&&` 和 `||` 逻辑操作符，这样的形式就是自解释的，这样，我们程序员就不需要再像上面那样列出一个表来解释每个指示符组合的含义了。新形式是这样的：

[复制代码](#)

```
1 //go:build linux && (386 || amd64 || arm || arm64 || mips64 || mips64le || ppc
2 //go:build linux && (mips64 || mips64le)
3 //go:build linux && (ppc64 || ppc64le)
4 //go:build linux && !386 && !arm
```

考虑到兼容性，Go 命令可以识别这两种形式的构建约束指示符，但推荐 Go 1.17 之后都用新引入的这种形式。

另外，`gofmt` 也可以兼容处理两种形式。它的处理原则是：如果一个源码文件只有 `// +build` 形式的指示符，`gofmt` 会把和它等价的 `//go:build` 行加入。否则，如果一个源文件中同时存在这两种形式的指示符行，那么 `//+build` 行的信息就会被 `//go:build` 行的信息所覆盖。

除了 `gofmt` 外，`go vet` 工具也会检测源文件中是否同时存在不同形式的、语义不一致的构建指示符，比如针对下面这段代码：

```
1 //go:build linux && !386 && !arm
2 // +build linux
3
4 package main
5
6 import "fmt"
7
8 func main() {
9     fmt.Println("hello, world")
10 }
```

[复制代码](#)

`go vet` 会提示如下问题：

```
1 ./buildtag.go:2:1: +build lines do not match //go:build condition
```

[复制代码](#)

小结

Go 1.17 版本的一些重要新特性就介绍到这里了，除了上面这些重要变化之外，Go 1.17 还有很多变更与改进，如果你还意犹未尽，建议你去认真读读 [Go 1.17 的发布说明文档](#)。

另外我还要多说一句，Go 1.17 版本的这些变更都是在 Go1 兼容性的承诺范围内的。也就是说，Go 1.17 版本秉持了 Go 语言开源以来各个版本的一贯原则：**向后兼容**，也就是即使你使用 Go 1.17 版本，也可以成功编译你十年前写下的 Go 代码。

读到这里，你是不是有一种要尽快切换到 Go 1.17 版本的冲动呢！赶快 [去 Go 官网下载 Go 1.17 的最新补丁版本](#)，开启你的 Go1.17 体验之旅吧。

思考题

在你阅读完 [Go 1.17 的发布说明文档](#) 之后，你会发现 Go 1.17 版本中的变化有很多，除了上面几个重要特性变化外，最让你受益或印象深刻的变化是哪一个呢？欢迎在留言区分

享。

欢迎你把这节课分享给更多对 Go 1.17 版本感兴趣的朋友。我是 Tony Bai，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 2

 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 加餐 | 他山之石：学习Go你还可以参考什么？

下一篇 期中测试 | 一起检验下你的学习成果吧

更多学习推荐

2021 最新大厂 Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

免费领取 



精选留言 (3)

 写留言



言己

2021-12-17

性能测试示例代码，golang 的字符串拼接原来有这么多方式。

**在下宝龙、**

2021-12-17

老师您好，我最近使用go mod tidy 发现并不能很好的生效，使用之后 go mod文件并没有增加出自己想要的包信息，给我的提示是这个，我尝试去理解并没有找到答案- =，我的go版本是1.17

To upgrade to the versions selected by go 1.16:

```
go mod tidy -go=1.16 && go mod tidy -go=1.17...
```

展开 ∨

**Calvin**

2021-12-17

老师后面可以加餐说下 go 1.18 的新“工作区模式”特性吗？感觉和泛型一样，它也会是一个重要的新特性。

展开 ∨

