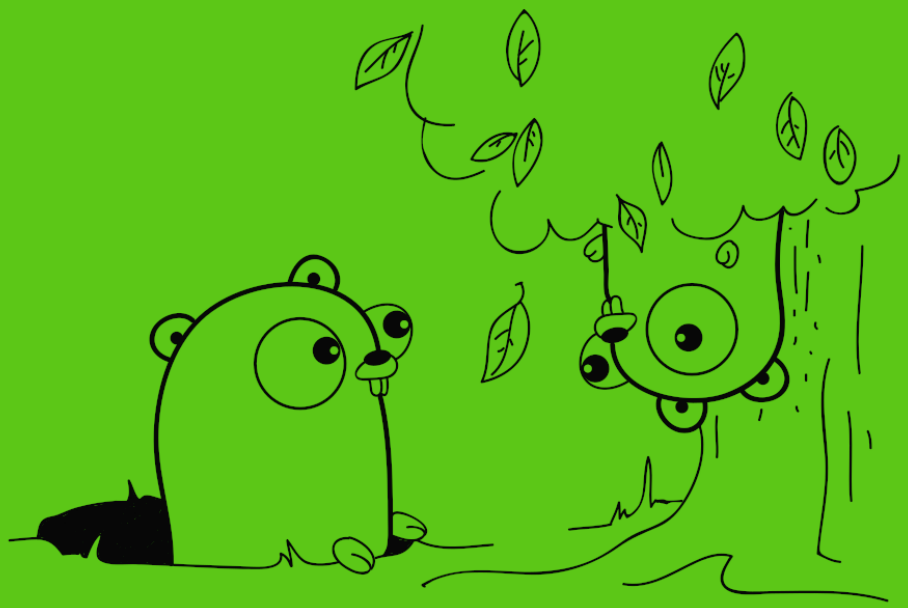
Go 细节和小技巧 101

Tapir Liu

Contents

Go细节与小技巧101

-= v1.19.b-rev-ff9b2d1-2022/11/15 =-



老貘 著

0.1 致谢

本书中有一些细节和小技巧从互联网搜集而来，也有一些是我自己发现的。我将尽量列出这些细节和小技巧的出处。但是我得承认对于某些细节来说很难做到这一点。

感谢 Olexandr Shalakhin 允许将其设计的生动的地鼠卡通图标 之一使用在本书的封面中。感谢 Renee French 女士设计了最初的可爱的地鼠卡通形象。

感谢下列开源软件和代码库的作者。这些开源软件和代码库在生成本书的电子书版本中帮了大忙。

- golang: https://go.dev/
- gomarkdown: https://github.com/gomarkdown/markdown
- goini: https://github.com/zieckey/goini
- go-epub: github.com/bmaupin/go-epub
- pandoc: https://pandoc.org
- calibre: https://calibre-ebook.com/
- GIMP: https://www.gimp.org

感谢对本书的翻译事业做出贡献的 Go 友：包括 ksco、dacapoday、iamazy、Darren 等。

感谢 ksco 对本书的初校。

Chapter 1

关于《Go 细节与小技巧 101》

本书搜集了很多 Go 语言编程中的小细节，并提供了若干 Go 编程中的小技巧。这些细节和技巧可以归集为以下几类：

- 语法和语义相关的；
- 类型转换相关的；
- 值比较相关的；
- 运行时相关的；
- 标准库相关的。

本书谈到的大多数细节是针对 Go 语言和官方工具链的，但也有一些是与语言无关的。

1.1 关于本书作者

本书作者为老貘，也就是《Go 101》一书的作者。老貘正计划完成更多《Go 101》系列丛书。敬请期待。

老貘曾（没准儿以后会重新成）为一个独立游戏开发者。他开发的游戏展示在这里。

你的赞赏是 Go 101 系列丛书不断改进和增容的动力：

1.2 反馈

欢迎各位在 Go 语言 101 项目（https://github.com/golang101/golang101）的问题列表中提交在本书中发现的各种问题和对这些问题的改正。

欢迎关注本书的微信公众号：Go 101（可在微信中搜索 go 101 或者 golang101，或者扫描下面的二维码进行关注）。

本书的 Twitter 帐号为 @Golang_101（中文 https://twitter.com/Golang_101）和 @go100and1（英文 https://twitter.com/go100and1）。

Chapter 2

语法和语义相关

2.1 零尺寸类型/值

不含非零尺寸字段的结构体类型的尺寸为零。一个长度为零或元素尺寸为零的数组类型的尺寸也是零。这些可以通过下面这个程序来验证。该程序打印出三个 0。

```
package main

import "unsafe"

type A [0][256]int

type S struct {
     x A
     y [1<<30]A
     z [1<<30]struct{}
}

type T [1<<30]S

func main() {
     var a A
     var s S
     var t T
     println(unsafe.Sizeof(a)) // 0
     println(unsafe.Sizeof(s)) // 0
     println(unsafe.Sizeof(t)) // 0
}
```

在 Go 中，内存尺寸通常用 int 值来表示。这意味着一个数组的最大可能长度是 MaxInt（在 64 位操作系统其值为 2^63-1）。但是官方标准 Go 编译器和运行时的实现中，元素尺寸不为零的数组的最大长度被用小得多的值限制住了。

举个例子：

```
var x [1<<63-1]struct{}     // okay
var y [2000000000+1]byte    // compilation error
var z = make([]byte, 1<<49) // panic: runtime error: makeslice: len out of range
```

2.2 零尺寸的值在内存中如何分配取决于具体使用的编译器

在目前官方的标准 Go 编译器实现（1.19）中，所有在堆上分配的局部零尺寸值都共享同一个地址。例如，下面打印了两次 false，然后打印了两次 true。

```
package main

var g *[0]int
var a, b [0]int

//go:noinline
func f() *[0]int {
	return new([0]int)
}

func main() {
	// x 和 y 被分配在栈上。
	var x, y, z, w [0]int
	// 使 z 和 w 逃逸到堆上。
	g = &z; g = &w
	println(&b == &a)  // false
	println(&x == &y)  // false
	println(&z == &w)  // true
	println(&z == f()) // true
}
```

请注意，上述程序的输出取决于所使用的特定编译器。对于未来的官方标准 Go 编译器版本，其输出结果可能会有所不同。

2.3 不要把一个零尺寸字段做为结构体类型的最后一个字段

在下面的代码中，类型 Tz 的尺寸要比类型 Ty 大。

```
package main

import "unsafe"

type Ty struct {
	_ [0]func()
	y int64
}

type Tz struct {
	z int64
	_ [0]func()
}

func main() {
	var y Ty
	var z Tz
	println(unsafe.Sizeof(y)) // 8
	println(unsafe.Sizeof(z)) // 16
```

```
}
```

为什么类型 Tz 的尺寸更大？

在当前官方标准 Go 运行时实现中，一个内存块只要被至少一个活动指针所引用，该内存块就不会被视为垃圾而回收。

一个可寻址结构体的所有字段都可以被取址。如果一个非零尺寸的结构体类型中最后一个字段的尺寸为零，那么当此结构体类型的一个值中最后一个字段的地址的时候，一个超出该结构体值内存块分配区的地址将被返回。此返回地址可能正指向另一个已分配的内存块。此另一个内存块与为非零尺寸结构体值分配的内存块紧密相连。只要此返回的地址仍被存储在一个活动指针值中，其此另一个内存块就不会被垃圾回收，这可能会导致内存泄漏。

为了避免这种内存泄露问题，官方标准 Go 编译器将确保在取非零尺寸结构体的最后一个字段的地址时候，绝不会返回超出为该结构体分配的内存块的地址。官方标准 Go 编译器通过在需要时在最后一个零尺寸字段之后填充一些字节来实现这一点。

因此，在上面的例子中，类型 Tz 的最后的（零）字段后填充至少有一个字节。这就是为什么类型 Tz 的尺寸比 Ty 大。

事实上，在 64 位操作系统上，在 Tz 的最后（零）字段后填充了 8 个字节。为了解释这个问题，我们应该知道官方标准编译器实现中的两个事实：

1. 一个结构体类型的地址对齐保证为其字段的最大地址对齐保证。
2. 一个类型的尺寸总是该类型的地址对齐保证的倍数。

第一个事实解释了为什么 Tz 类型的地址对齐保证是 8（这是内置的 int64 类型的地址对齐保证）。第二个事实解释了为什么 Tz 类型的尺寸是 16。

来源：https://github.com/golang/go/issues/9401

2.4 模拟其它一些语言中的 for i in 0..N 语法

我们可以使用 for-range 循环来模拟其它一些语言中的 for i in 0...N 循环。如下所示：

```
package main

const N = 8
var n = 8

func main() {
	for i := range [N]struct{}{} {
		println(i)
	}
	for i := range [N][0]int{} {
		println(i)
	}
	for i := range make([][0]int, n) {
		println(i)
	}
}
```

前两个循环的步数必须在写代码的时候就知道，而最后一个循环则没有这个要求。但是最后一个循环需要分配多分配一点点内存（在栈上，用于切片的头部）。

2.5 有多种创建切片的方式

例如，以下代码中的每个切片都是以不同方式创建的。

```
package main

func main() {
	var s0 = make([]int, 100)
	var s1 = []int{99: 0}
	var s2 = (&[100]int{})[:]
	var s3 = new([100]int)[:]
	// 100 100 100 100
	println(len(s0), len(s1), len(s2), len(s3))
}
```

2.6 for i, v = range aContiner 实际上是迭代 aContainer 的一个副本

例如，下面这个程序将打印 123 而不是 189。

```
package main

func main() {
	var a = [...]int{1, 2, 3}
	for i, n := range a {
		if i == 0 {
			a[1], a[2] = 8, 9
		}
		print(n)
	}
}
```

如果被遍历的容器是一个大的数组，那么复制成本就会很高。

有一个例外：如果 for-range 中的第二个迭代变量被省略或忽略，那么被遍历的容器将不会被复制（因为没有必要进行复制）。例如，在下面的两个循环中，数组 a 没有被复制。

```
func main() {
	var a = [...]int{1, 2, 3}
	for i := range a {
		print(i)
	}
	for i, _ := range a {
		print(i)
	}
}
```

在 Go 中，一个数组拥有其元素，但一个切片只是引用着其元素。在 Go 中，值的复制都是浅拷贝，复制一个值不会复制它所引用的值。所以复制一个切片不会复制其元素。这可以反映在下面的程序中。该程序打印出 189。

```
package main
```

```
func main() {
	var s = []int{1, 2, 3}
	for i, n := range s {
		if i == 0 {
			s[1], s[2] = 8, 9
		}
		print(n)
	}
}
```

2.7 在某些情形下，数组指针可以被当做数组使用

例如，下面的代码编译和运行都没有问题。

```
package main

func main() {
	var a = [128]int{3: 789}
	var pa = &a
	// 无需复制数来遍历数组元素。
	// 复制一个指针的成本很低。
	for i, v := range pa {
		_, _ = i, v
	}
	// 通过数组指针获取数组的长度和容量。
	_, _ = len(pa), cap(pa)
	// 通过数组指针访问数组元素。
	_ = pa[3]
	pa[3] = 555
	// 从数组指针派生切片。
	var _ []int = pa[:]
}
```

如果第二个迭代变量被省略或忽略，则遍历一个 nil 数组指针将不会发生恐慌。例如，下面代码中的前两个循环都打印 01234，但最后一个循环会导致恐慌。

```
package main

func main() {
	var pa *[5]string

	// 打印出 01234
	for i := range pa {
		print(i)
	}

	// 打印出 01234
	for i, _ := range pa {
		print(i)
	}

	// 将产生一个恐慌。
```

```
    for _, v := range pa {
        _ = v
    }
}
```

2.8 有些函数调用在编译时被估值

在编译时估值的函数调用也被称为常量调用，因为其计算结果是常量值。

所有对 unsafe.Sizeof、unsafe.Offsetof 和 unsafe.Alignof 的调用都在编译时进行估值。

如果内置 len 和 cap 函数的一个调用的参数是一个常量字符串、一个数组或一个数组指针，并且参数表达式不包含通道读取操作或非常量函数调用，那么该调用将在编译时被估值。

在计算刚刚提到的常量函数调用时，只有涉及的实参类型是重要的（除了实参是常量字符串），即使对实参的估值可能在运行时引起恐慌。

例如，调用下面的代码中的 f 和 g 函数不会在运行时发生恐慌。

```
package main

import "unsafe"

func f() {
    var v *int64 = nil
    println(unsafe.Sizeof(*v)) // 8
}

func g() {
    var t *struct {s [][16]int} = nil
    println(len(t.s[99]))  // 16
}

func main() {
    f()
    g()
}
```

另一方面，调用下面所示的 f2 和 g2 函数中的任何一个都会在运行时引发恐慌。

```
func f2() {
    var v *int64 = nil
    _ = *v
}

func g2() {
    var t *struct {s [][16]int} = nil
    _ = t.s[99]
}
```

请注意，内置的 len 函数会在 for-range 循环中被隐式调用。了解这一点是理解为什么下面的代码中的前两个循环不会导致恐慌，而最后一个循环会导致恐慌的关键。

```
package main

type T struct {
     s []*[5]int
}

func main() {
       var t *T
       for i, _ := range t.s[99] { // 不引发恐慌
          print(i)
       }
       for i := range *t.s[99] { // 不引发恐慌
          print(i)
       }
       for i := range t.s { // 引发恐慌
          print(i)
       }
}
```

是的，隐式的 len(t.s[99]) 和 len(*t.s[99]) 调用是在编译时估值的。在估值过程中只有数组值 t.s[99] 的长度（这里是 5）是重要的。然而，隐式调用 len(t.s) 是在运行时估值的。所以当 t 是一个空指针时，它会导致恐慌。

上面已经提到，如果内置的 len 或 cap 函数的一个调用的实参表达式中包含通道读取操作或者非常量函数调用，则此调用不会在编译时刻被估值。例如，下面的代码就编译会出错。

```
var c chan int
var s []byte
const X = len([1]int{<-c})    // error: len(...) is not a constant
const Y = cap([1]int{len(s)}) // error: cap(...) is not a constant
```

在下面的代码中，表达式 imag(X) 是一个常量函数调用，但是表达式 imag(y) 是一个非常量（因为 X 是一个常量，但 y 是一个变量），所以表达式 len(A{imag(y)}) 不会在编译时刻估值，这就是为什么最后一行编译不通过的原因。而表达式 len(z) 不含有任何非常量函数调用，所以它将被认为是一个常量表达式并在编译时刻被估值。

```
const X = 1 + 2i
var y = 1 + 2i

type A [8]float64

var _ [len(A{imag(X)})]int    // 编译没问题
var z = A{imag(y)}
var _ [len(z)]int             // 编译没问题
// var _ [len(A{imag(y)})]int // 编译不通过
```

2.9 官方标准 Go 编译器不支持声明尺寸超过 2GB 的全局数组变量

例如，下面的程序编译时会报错误 main.x: symbol too large (2000000001 bytes > 2000000000 bytes) 。

```
package main

var x [2000000000+1]byte

func main() {}
```

分配在堆上的数组的尺寸可以大于 2GB。例如，在下面这个程序中，数组 x 和 y 均被分配在堆上，所以此程序编译没问题。

```
package main

var y *[2000000000+1]byte

func main() {
	var x [2000000000+1]byte
	y = &x
}
```

来源：https://github.com/golang/go/issues/17378

2.10 切片/数组/映射元素和结构体字段的可寻址性

以下是关于切片/数组/映射元素的可寻址性的一些事实：

- 一个切片的元素总是可寻址的，无论此切片本身是否可寻址。
- 可寻址数组值的元素也是可寻址的；不可寻址的数组值的元素也是不可寻址的。
- 映射值的元素总是不可寻址的。

和数组一样，可寻址结构体的字段也是可寻址的，不可寻址结构体的字段也是不可寻址的。

例如，在下面的代码中，函数 foo 中的取址操作都是非法的，而函数 bar 中的操作则是合法的。

```
type T struct {
	x int
}

func foo() {
	// 字面量是不可寻址的。
	_ = &([10]bool{}[1]) // error
	// 隐射元素是不可寻址的。
	var mi = map[int]int{1: 0}
	_ = &(mi[1]) // error
	var ma = map[int][10]bool{2: [10]bool{}}
	_ = &(ma[2][1]) // error
	_ = &(T{}.x)    // error
}

func bar() {
	var _ = &([]int{1: 0}[1]) // okay
	// 任何变量都是可寻址的。
	var a [10]bool
	_ = &(a[1]) // okay
```

```
	var t T
	_ = &(t.x) // okay
}
```

从不可寻址的数组中派生切片也是非法的。所以下面的代码也编译不成功。

```
var aSlice = [10]bool{}[:]
```

2.11 组合字面量是不可寻址的，但它们可以被取地址

组合字面量包括结构体/数组/切片/映射字面量。所有字面量都是不可寻址的。一般来说，不可寻址的值不能被取地址。然而，在 Go 中有一个例外（一个语法糖）：组合字面量可以被取地址。

例如，下面的代码编译没问题。

```
package main

type T struct {
	x int
}

func main() {
	// 所有这些取地址操作都是合法的。
	_ = &T{}
	_ = &[8]byte{}
	_ = &[]byte{7: 0}
	_ = &map[int]bool{}
}
```

请注意，索引运算符 [] 和属性选择运算符 . 的优先级都高于取址运算符 &。例如，下面代码中的两行都会编译报错。

```
	_ = &T{}.x // error
	_ = &[8]byte{}[1] // error
```

它们编译报错的原因是它们等价于下面的代码。

```
	_ = &(T{}.x) // error
	_ = &([8]byte{}[1]) // error
```

另一方面，以下两行编译是没有问题的。

```
	_ = (&T{}).x // okay
	_ = (&[8]byte{})[1] // okay
```

2.12 一行代码创建指向基本类型的非零值的指针的技巧

组合字面量可以被取地址，但是其它字面量不能被取地址。例如，下面的几行代码行都是非法的。

```
var pb = &true
var pi = &123
var pb = &"abc"
```

事实上，我们可以用单行但是略微繁琐一点的方式来实现类似的效果：

```
var pb = &(&[1]bool{true})[0]
var pi = &(&[1]int{9})[0]
var ps = &(&[1]string{"Go"})[0]

// 相比上面三行，这三行略简洁一点，但效率略低。
var pb2 = &([]bool{true})[0]
var pi2 = &([]int{9})[0]
var ps2 = &([]string{"Go"})[0]
```

这个技巧在填充一个大的结构体（通常用做配置变量）时很有用。例如，如果不使用这个技巧，代码可能需要写成：

```
var x = true

var cfg = mypkg.Config {
    ... // many other options

    // Three possible values: nil, &false, &true.
    OptionsX: &x,

    ...
}
```

如果此配置中有很多其它的选项，那么变量 x 的使用到它的声明的距离就会非常远。这并不是一个大问题，但在一定程度上损害了代码的可读性。

相反，我们可以使用下面的代码来避免这种远距离问题。

```
var cfg = mypkg.Config {
    ... // many other options

    // Three possible values: nil, &false, &true.
    OptionsX: &(&[1]bool{true})[0],

    ... // more options
}
```

一个更繁琐一点的解决方法是使用匿名函数调用：

```
var cfg = mypkg.Config {
    ...
    OptionsX: func() *bool {var x = true; return &x}(),
    ...
}
```

此技巧来自于 issue.

2.13 不可寻址的值是不可修改的，但映射元素（做为一个整体）可以被修改

上面的章节已经提到，映射元素是不可寻址的并且不能被取地址。一般来说，不可寻址的值也是不可修改的，但映射元素可以被修改，只不过每次修改都必须是一个整体修改。这意味着一个映射元素不能被部分地修改。

举个例子：

```
package main

type T struct {
	x int
}

func main() {
	var mt = map[int]T{1: T{x: 2}}
	var ma = map[int][3]bool{}
	mt[1] = T{x: 3}          // okay
	ma[1] = [3]bool{0: true} // okay

	// 下面两行均被视为局部修改而无法编译。
	mt[1].x = 3    // error
	ma[1][0] = true// error
}
```

2.14 创建映射的 make 调用中的第二个参数仅被看作是一个提示

每个非零的映射值都有一个内部底层数组用来存储其条目。随着越来越多的条目被放入该映射值，此底层数组会随着需要而逐渐增长。

一个的 make 调用将为其创建的映射值创建一个足够长的、能够容纳指定数量条目的底层数组（条目数量在未超越此指定数量时，此底层数组不会增容）。这个参数是可选的，它的默认值取决于编译器。

该参数可以是一个零，甚至是一个非常量的负数。例如，下面的代码运行正常（不会恐慌）。

```
	var n = -99
	var m = make(map[string]int, n)
```

注意此参数并不表示映射值的容量。一个映射值的容量理论上无穷大的。

来源：https://github.com/golang/go/issues/46909

2.15 使用映射来模拟集合

Go 支持内置的映射类型（map），但不支持集合类型（set）。我们可以使用映射类型来模拟集合类型。如果一个集合类型的元素类型 T 是可比较的。那么我们就可以使用 map[T]struct{} 类型来模拟集合类型。

```
package main

type Set map[int]struct{}

func (s Set) Put(x int) {
	s[x] = struct{}{}
}

func (s Set) Has(x int) (r bool) {
	_, r = s[x]
```

```
    return
}

func (s Set) Remove(x int) {
    delete(s, x)
}

func main() {
    var s = make(Set)
    s.Put(2)
    s.Put(3)
    println(len(s))   // 2
    println(s.Has(3)) // true
    println(s.Has(5)) // false
    s.Remove(3)
    println(len(s))   // 1
    println(s.Has(3)) // false
}
```

如果一个集合类型的元素类型 T 是不可比较的。我们可以使用 map 类型 map[*byte]T 来模拟集合类型。尽管此集合类型的功能会受限一些。

举个例子：

```
package main

type Set map[*byte]func()

func (s Set) Put(x func()) (remove func()) {
    key := new(byte)
    s[key] = x
    return func() {
        delete(s, key)
    }
}

func main() {
    var s = make(Set)
    remove1 := s.Put(func(){ println(111) })
    remove2 := s.Put(func(){ println(222) })
    for _, f := range s {
        f()
    }
    println(len(s))   // 2
    remove1()
    println(len(s))   // 1
    remove2()
    println(len(s))   // 0
}
```

键（指针）类型的基类型不能是一个零尺寸的类型。否则，使用 new 函数创建出来的指针可能并不是唯一的（见前述一节）。

此集合的实现很简单，但它并非适用于所有场合。此技巧借鉴自 Tailscale 项目。

2.16 映射条目的迭代顺序是随机的

Go 内置的映射（map）值并不维护条目顺序。因此，当使用 for-range 循环遍历 一个映射值的条目时，条目的顺序是随机的（至少在某种程度上取决于所使用的编译器）。

多次运行下面的程序，我们会发现其输出可能是不同的。

```
package main

func main() {
	var m = map[int]int{3:3, 1:1, 2:2}
	for k, v := range m {
		print(k, v)
	}
}
```

但请注意，当使用 fmt 标准包中的各个打印函数打印映射值时，会对映射值的条目（按其键）进行排序。json 标准包的 Marshal 函数也是如此。

2.17 在迭代一个映射值的过程中放入此映射值的条目可能会在当前迭代过程中显示出来，也可能会被跳过

例如，下面这个程序运行输出是不固定的：

```
package main

var m = map[int]bool{0: true, 1: true}

func main() {
	for k, v := range m {
		m[len(m)] = true
		println(k, v)
	}
}
```

一些可能的输出：

```
$ go run main.go
0 true
1 true
2 true
3 true
$ go run main.go
0 true
1 true
$ go run main.go
0 true
1 true
2 true
$ go run main.go
1 true
2 true
3 true
```

```
4 true
5 true
6 true
7 true
0 true
```

请注意，如上所述，条目的迭代顺序是随机的（至少有几分）。

2.18 切片或数组组合字面量中的键必须为常量

例如，下面的代码编译不通过：

```
var m, n = 1, 2
var s = []string{m: "Go"} // error
var a = [3]int{n: 999}    // error
```

映射组合字面量中的键没有这个限制。

2.19 切片/数组/映射组合字面量的常量键不能重复

Go 白皮书明确规定切片/数组/映射组合字面量的常量键不能重复。

例如，所有的下面这些代码行都因为常量键重复而编译不通过。

```
var m = map[string]bool{"Go": true, "Go": false} // error
var s = []string{0: "Go", 0: "C"}                // error
var a = [3]int{2: 999, 2: 555}                   // error
```

请注意，映射条目中的非常量重复键会导致未定义行为。例如，下面的代码打印 1、2 或 3 是可以的。这些打印结果中的任何一个都没有违反 Go 白皮书。

```
package main

var a = 1
func main() {
     m := map[int]int{1: 1, a: 2, a: 3}
     println(m[1])
}
```

2.20 利用上一节中提到的事实进行编译时刻断言

如何在编译时断言一个常量布尔表达式为 true （或 false）？我们可以利用上一节中介绍的事实：在映射组合字面量中不允许有重复的常量键。

例如，下面的代码可以保证常量布尔表达式 aConstantBoolExpr 必须为 true。如果它不为 true，则此行代码就编译不通过。

```
var _ = map[bool]int{false: 0, aConstantBoolExpr: 1}
```

例如，下面的代码使用编译器断言一个常量字符串的长度为 32。

```
const S = "abcdefghijklmnopqrstuvwxyz123456"
var _ = map[bool]int{false: 0, len(S)==32: 1}
```

在这个技巧中，映射的元素类型可以是任意类型。

此技巧适用于官方标准 Go 编译器，但不适用于 gccgo（截至 10.2.1 20210110 版本）。在 gccgo 中存在一个 bug，它允许在映射组合字面量中出现重复的常量 bool 键。此 bug 将在 gccgo 12 中被修复。

来源：https://twitter.com/lukechampine/status/1026695476811390976

2.21 更多编译时断言的技巧

上一节中的编译时刻断言用例（断言一个常量字符串的长度为 32）还有一些其它实现方式，比如：

```
var _ = [1]int{len(S)-32: 0}
var _ = [1]int{}[len(S)-32]
```

下面是在编译时断言一个常量 N 不比另一个常量 M 小的技巧：

```
const _ uint = N-M
type _ [N-M]int
```

断言一个常量字符串不为空的技巧：

```
var _ = aStringConstant[0]
const _ = 1/len(aStringConstant)
```

最后一行的出处：https://groups.google.com/g/golang-nuts/c/w1-JQMaH7c4/m/qzBFSPImBgAJ

2.22 一个函数的返回结果可以在 return 语句执行后被修改

是的，一个 defer 函数调用可以修改调用它的函数的具名返回结果。例如，下面的程序打印的是 9 而不是 6。

```
package main

func triple(n int) (r int) {
	defer func() {
		r += n
	}()

	return n + n
}

func main() {
	println(triple(3)) // 9
}
```

2.23 一个延迟函数调用的实参和被其调用的函数值均是在注册此延迟函数调用时被估值的

在函数的退出阶段，被注册的延迟函数调用将按照它们的注册顺序逆序依个被执行。在执行这些延迟函数调用时，它们的实参和被它们调用的函数值将不在被重新估值。

例如，下面的程序打印的是 1，而不是 2 或 3。

```
package main

func main() {
	var f = func (x int) {
		println(x)
	}
	var n = 1
	defer f(n)
	f = func (x int) {
		println(3)
	}
	n = 2
}
```

下面的程序运行中并不会产生恐慌。它打印出 123。

```
package main

func main() {
	var f = func () {
		println(123)
	}
	defer f()
	f = nil
}
```

下面的程序打印出 123，然后恐慌。

```
package main

func main() {
	var f func () // nil
	defer f()
	println(123)
	f = func () {
	}
}
```

2.24 方法属主实参和其它普通实参一同被估值

所以，当一个延迟方法调用被注册时，它的属主实参（和其它普通实参）将被估值。在一个方法调用链 v.M1().M2() 中，方法调用 v.M1() 是 M2 方法调用的属主参数，所以方法调用 v.M1() 将在延迟调用 defer v.M1().M2() 被注册时估值（亦即执行）。

例如，下面的程序打印 132。

```
package main

type T struct{}

func (t T) M(n int) T {
	print(n)
```

```
	return t
}

func main() {
	var t T
	defer t.M(1).M(2)
	t.M(3)
}
```

下面的例子更自然一些。

```
import "sync"

type Counter struct{
	mu sync.Mutex
	n int
}

func (c *Counter) Lock() *Counter  {
	c.mu.Lock()
	return c
}

func (c *Counter) Unlock() *Counter  {
	c.mu.Unlock()
	return c
}

func (c *Counter) Add(x int) {
	defer c.Lock().Unlock()
	c.n += x
}
```

类似的用法还包括 `defer gl.PushMatrix().PopMatrix()` 和 `defer tag.Start(..).End()`。

2.25 如果一个非常量移位表达式的左操作数是类型不确定的，那么它的类型将被确定为此表达式的最终的假定类型

如果一个移位表达式有一个操作数不是常量，那么该移位表达式就是一个非常量表达式，其结果将在运行时计算结果。

目前（Go 1.19），类型不确定的整数必须是常量。因此，如果一个移位表达式是非常量，并且它的左操作数是类型不确定的，那么它的右操作数必须是一个非常量。

下面的程序打印出 002。原因是前两个移位表达式都是非常量，所以其中各自的类型不确定整数 1 的类型被推导为最终的假设类型 byte，所以每个 `1 << n` 表达式在运行时将被计算为 0（因为 256 溢出了 byte 类型的表示范围）。

而第三个移位表达式是一个常量表达式，所以它在编译时被计算。实际上，整个表达式 `(1<N)/128` 在编译时被计算为 2。

```
package main
```

```
func main() {
    var n = 8
    var x byte = 1 << n / 128
    print(x) // 0
    var y = byte(1 << n / 128)
    print(y) // 0

    const N = 8
    var z byte = 1 << N / 128
    println(z) // 2
}
```

为什么在上述情况下，一个类型不确定的整数的类型不会被推导为其默认类型 int？这可以用下面的例子来解释。如果下面的代码中的类型不确定的 1 被推导为 int 值，而不是 int64 值，那么移位操作将在 32 位架构（0）和 64 位架构（0x100000000）之间返回不同的结果，这可能会产生一些难以及时发现的 bug。

```
var n = 32
var y = int64(1 << n)
```

下面的移位表达式都不能编译，因为前三个类型不确定整数 1 都被推导为假设类型 float64 的值，最后一个被推导为假设类型 string 的值，而浮点值和字符串值不能被移位。

```
var n = 6
var x float64 = 1 << n  // error
var y = float64(1 << n) // error
var z = 1 << n + 1.0    // error
var w = string(1 << n)  // error
```

下面程序打印出 0 1：

```
package main

var n = 8
// 假定类型为 byte。
var x = 1 << n >> n + byte(0)
// 假定类型为 int16。
var y = 1 << n >> n + int16(0)

func main() {
    println(x, y) // 0 1
}
```

如果假定类型不存在，那么类型不确定的左操作数将被推断为其默认类型。所以下面的代码中类型不确定的 1 被推断为 int 值。变量 x 在 32 位架构上被初始化为 0（溢出），但在 64 位架构上被初始化为 0x100000000。对于一个合格的 Go 程序员来说，这并不奇怪。

```
var n = 32
var x = 1 << n // an int value
```

以下代码无法编译，因为以下代码中类型不确定的 1.0 被推断为 float64 值。

```
var n = 6
var y = 1.0 << n // error
```

2.26 aConstString[i] 和 aConstString[i:j] 是非常量，即使 aConstString，i 和 j 都是常量

例如，下面两行都通过不了编译：

```
const G = "Go"[0]       // error
const Go = "Golang"[:2] // error
```

下面两行代码就能编译通过：

```
var G = "Go"[0]
var Go = "Golang"[:2]
```

这是 Go 1.0 中的一个设计上的小瑕疵。遗憾的是，由于向后兼容的原因，这个事实很难改变。目前，下面的程序打印出 4 0，因为表达式 len(s[:]) 不是常量，而表达式 len(s) 是。

```
package main

const s = "Go101.org" // len(s) == 9
var a byte = 1 << len(s) / 128
var b byte = 1 << len(s[:]) / 128

func main() {
    println(a, b) // 4 0
}
```

来源：https://github.com/golang/go/issues/28591

2.27 目标类型为参数类型的转换结果总是非常量

比如，在下面的代码中，表达式 len(string(S)) 是一个常量，但表达式 len(T(S)) 不是常量。

```
package main

type MyString string
const S MyString = "Go101.org" // len(S) == 9

func foo() byte {
    var _ [len(string(S))]int // 编译没问题
    return 1 << len(string(S)) >> len(string(S))
}

func bar[T string]() byte {
    // var _ [len(T(S))]int // 编译不通过
    return 1 << len(T(S)) >> len(T(S))
}

func main() {
    println(foo()) // 1
    println(bar()) // 0
}
```

另外，如果一个 len 或者 cap 函数调用的实参的类型为一个参数类型，则此函数调用总是被视为一个非常量。比如，下面的程序打印出 1 0。

```
package main

const S = "Go"

func ord(x [8]int) byte {
	return 1 << len(x) >> len(x)
}

func gen[T [8]int](x T) byte {
	return 1 << len(x) >> len(x)
}

func main() {
	var x [8]int
	println(ord(x), gen(x)) // 1 0
}
```

2.28 操作数均为类型不确定值的二元运算的类型推导规则

如果一个二元操作（除移位外）的两个操作数均为类型不确定值，且它们的默认类型不同，那么此二元操作的结果仍为一个类型不确定值，此结果的默认类型为两个类型不确定操作数的默认类型中位于列表 [int, rune, float64, complex128] 的靠后者。

根据规则，下面的程序先打印 int32（即 rune），然后是 complex128，最后是 float64。

```
package main

import "fmt"

const A = 'A' // 65
const B = 66
const C = 67 + 0i
const One = B - A // 1
const Two = C - A // 2
const Three = B / 22.0

func main() {
	fmt.Printf("%T\n", One)   // int32
	fmt.Printf("%T\n", Two)   // complex128
	fmt.Printf("%T\n", Three) // float64
}
```

下面的程序打印出 01（在 64 位架构上）。因为类型不确定的常量 R 的类型被视为 rune（int32）。

```
package main

import "fmt"

const A = '\x61' // a rune literal
```

```
const B = 0x62   // default type is int
const R = B - A  // default type is rune

var n = 32

func main() {
     if R == 1 {
          fmt.Print(R << n >> n) // 0
          fmt.Print(1 << n >> n) // 1
     }
}
```

下面的程序打印 2 3.

```
package main

import "fmt"

const X = 3 / 2 * 2.
const Y = 3 / 2. * 2
var x, y int = X, Y

func main() {
     fmt.Println(x, y) // 2 3
}
```

2.29 一个类型不确定的常量整数可能会溢出其默认类型

以下代码中的两个常量声明是合法的。

```
const N = 1 << 200      // default type: int
const R = 'a' + 1 << 31 // default type: rune
```

类型化的值不得溢出其各自的类型。以下两个变量和两个常量的声明都是非法的。

```
const N int = 1 << 200
const R rune = 'a' + 1 << 31
var x = 1 << 200
var y = 'a' + 1 << 31
```

然而，以下这些都是合法的：

```
const N int = 1 << 200 >> 199
const R rune = 'a' + 1 << 31 - 'b'
var x = 1 << 200 >> 199
var y = 'a' + 1 << 31 - 'b'
```

2.30 在 switch 代码块中，default 分支的位置（如果存在的话）可以是任意的

例如，下面代码中的三个 switch 代码块都是合法的。

```
func foo(n int) {
     switch n {
```

```
    case 0: println("n == 0")
    case 1: println("n == 1")
    default: println("n >= 2")
    }

    switch n {
    default: println("n >= 2")
    case 0: println("n == 0")
    case 1: println("n == 1")
    }

    switch n {
    case 0: println("n == 0")
    default: println("n >= 2")
    case 1: println("n == 1")
    }
}
```

select 代码块中的 default 分支亦是如此。

2.31 switch 代码块中的常量情况表达可能是重复的，也可能不是，这取决于具体使用的编译器

目前，官方标准 Go 编译器和 gccgo 编译器都不允许使用重复的常量整数 case 表达式。例如，下面的代码编译失败。

```
    switch 123 {
    case 123:
    case 123: // error: duplicate case
    }
```

两种编译器都允许重复的常量布尔值 case 表达式。下面的代码能通过编译。

```
    switch false {
    case false:
    case false: // okay
    }
```

官方标准 Go 编译器不允许重复的常量字符串 case 表达式，但 gccgo 允许。

2.32 switch 表达式是可选的，它的默认值是为类型确定值 true （类型为内置类型 bool ）

例如，下面的程序将打印 True。

```
package main

var x, y = false, true

func main() {
    switch {
    case x: println("False")
```

```
    case y: println("True")
    }
}
```

但是下面的代码无法编译，因为 MyBool 类型的两个值（x 和 y）不能与 bool 值比较。

```
package main

type MyBool bool
var x, y MyBool = false, true

func main() {
    switch {
    case x: // error
    case y: // error
    }
}
```

2.33 Go 编译器会在代码中自动插入一些分号

让我们来看一个小程序：

```
package main

func foo() bool {
    return false
}

func main() {
    switch foo()
    {
    case false: println("False")
    case true: println("True")
    }
}
```

上面这个程序的输出是什么？让我们想一想。

~

~

~

False？不，它打印的是 True。很奇怪？函数 foo 难道不是总是返回 false 吗？

是的，函数 foo 总是返回 false，但在这里它和此问题无关。

编译器会自动为上面的代码插入一些分号，如下：

```
package main

func foo() bool {
    return false;
};
```

```
func main() {
     switch foo();
     {
     case false: println("False");
     case true: println("True");
     };
};
```

现在，这段代码清楚地表明 switch 表达式（true）被省略了。此 switch 代码块实际上等价于：

```
     switch foo(); true
     {
     case false: println("False");
     case true: println("True");
     };
```

这就是为什么此程序打印出 True 的原因。

关于详细的分号插入规则，请阅读这篇文章。

2.34 字节切片（和 rune 切片）到底是什么？

字节切片有两种解读：

1. 底层类型为 []byte 的切片类型称为字节切片类型。
2. 元素类型的底层类型为 byte 的切片类型称为字节切片类型。

第二种解释比第一种解释更宽泛。例如，在下面的代码中，类型 Tx 和 Ty 都适合第二种解读，但只有类型 Tx 适合第一种解读。

```
type Tx []byte
type MyByte byte
type Ty []MyByte
```

在 1.18 版本之前，官方标准 Go 编译器（gc）有时采用了第一种解读，而 gccgo 编译器总是采用的第二种解读。在 Go 中，一个字符串可以被转换为一个字节切片，反之亦然。下面的代码是否能编译成功，取决于采用哪种解读。所以下面代码中的 bar 函数使用 gc 编译器 1.17- 版本时将编译不通过。

```
type Tx []byte
type MyByte byte
type Ty []MyByte

var x Tx
var y Ty
var s = "Go"

func foo() {
     x = Tx(s)
     y = Ty(s)
     s = string(x)
}

func bar() {
```

```
	s = string(y) // error (当使用 gc 编译器 1.17- 时)
}
```

从 1.18 版本开始，gc 也完全采用了第二种解读，上面代码中的 bar 函数使用 gc 编译器 1.18+ 版本编译时没问题的。

Go 白皮书从 1.19 版本开始正式地采用了第二种解读方式。

请注意，在检查传递给内置函数 copy 和 append 的调用的实参时，编译器应该采用第一种解读。下面的代码中中的函数 g 使用 gccgo 编译是没问题的（这应该是一个 bug），但是使用 gc 编译器编译不通过（这是正确的实现）。

```
type Tx []byte
type MyByte byte
type Ty []MyByte

var x = make(Tx, 2)
var y = make(Ty, 2)
var s = "Go"

func f() {
	copy(x, s)
	_ = append(x, s...)
}

func g() {
	copy(y, s)          // error (当使用 gc 编译器时)
	_ = append(y, s...) // error (当使用 gc 编译器时)
}
```

码点（rune）切片的情况是相同的。

2.35 迭代变量在循环步之间是共享的

在下面的代码中，loop1 和 loop2 这两个函数彼此是不等同的。在 loop1 中，变量 v 在三个循环步之间共享，而在 loop2 中，每个循环步声明一个新的变量 v，这就是为什么 loop1 返回的结果中的三个元素都有相同的值。

```
package main

func loop1(s []int) []*int {
	r := make([]*int, len(s))
	for i, v := range s {
		r[i] = &v
	}
	return r
}

func loop2(s []int) []*int {
	r := make([]*int, len(s))
	for i := range s {
		v := s[i]
		r[i] = &v
	}
```

```
    return r
}

func printAll(s []*int) {
    for i := range s {
        print(*s[i])
    }
    println()
}

func main() {
    var s1 = []int{1, 2, 3}
    printAll( loop1(s1) ) // 333

    var s2 = []int{1, 2, 3}
    printAll( loop2(s2) ) // 123
}
```

出于同样的原因，下面代码中的第一个循环打印出 333，而第二个循环打印出 321。

```
package main

func main() {
    var s = []int{1, 2, 3}

    // 打印出 333
    for _, v := range s {
        defer func() {
            print(v)
        }()
    }

    // 打印出 321
    for _, v := range s {
        v := v
        defer func() {
            print(v)
        }()
    }
}
```

2.36 int、false、nil 等都不是关键字

它们是预声明的标识符，是可能被自定义的标识符所遮挡。

例如，下面这个奇怪的程序编译和运行都没问题。它打印出 false 和 123。

```
package main

var true = false
const byte = 123
type nil interface{}
func len(nil) int {
```

```
    return byte
}

func main() {
    var s = []bool{true, true, true}
    println(s[0])   // false
    println(len(s)) // 123
}
```

2.37 选择器碰撞

类型嵌入是 Go 的一个重要特征。通过类型嵌入，一个类型可以轻松地获得其它类型的字段和方法。

有时，并不是所有被嵌入类型的字段和方法都被嵌入其的类型获得。原因是促进了选择器（包括字段和方法）可能发生碰撞。

例如，在下面的代码中，类型 B 比类型 A 多嵌入了一个类型（T2）。但是它没有获得任何字段和方法。原因是 B.T1.m 和 B.T2.m 发生了碰撞，所以都没有得到提升。同样的情况也发生在 B.T1.n 和 B.T2.n 上。

```
package main

type T1 struct { m bool; n int }
type T2 struct { n int }
func (T2) m() {}

type A struct { T1 }
type B struct { T1; T2 }

func main() {
    var a A
    _ = a.m
    _ = a.n
    var b B
    _ = b.m // error: ambiguous selector
    _ = b.n // error: ambiguous selector
}
```

请注意，非导出的选择器（无论是字段还是方法）所处的代码包的引入路径是选择器的内在属性。两个来自两个不同包的同名的非导出选择器不会发生碰撞。

例如，在上面的例子中，如果 T1 和 T2 者两个类型分别在两个不同的 d 代码包中声明，那么类型 B 将获得 3 个字段和一个方法。

2.38 每个方法都对应着一个函数，其第一个参数是该方法的属主参数

例如，在下面的代码中。

- 类型 T 有一个方法 M1，对应的是函数 T.M1。
- 类型 *T 有两个方法：M1 和 M2，分别对应函数 (*T).M1 和 (*T).M2。

```
package main

type T struct {
	X int
}

func (t T) M1() int {
	return t.X
}

func (t *T) M2() int {
	return t.X
}

func main() {
	var t = T{X: 3}
	_ = T.M1(t)
	_ = (*T).M1(&t)
	_ = (*T).M2(&t)
}
```

2.39 方法选择器的正规化

Go 允许一些选择器的简化形式。

例如，在下面这个程序中，t1.M1 是 (*t1).M1 的简化形式，而 t2.M2 则是 (&t2).M2 的简化形式。在编译时，编译器将把简化的形式正规化为它们原来各自的完整形式。

此程序打印出 0 和 9，因为对 t1.X 的修改对 (*t1).M1 的估值结果没有影响。

```
package main

type T struct {
	X int
}

func (t T) M1() int {
	return t.X
}

func (t *T) M2() int {
	return t.X
}

func main() {
	var t1 = new(T)
	var f1 = t1.M1 // <=> (*t1).M1
	t1.X = 9
	println(f1()) // 0

	var t2 T
	var f2 = t2.M2 // <=> (&t2).M2
```

```
    t2.X = 9
    println(f2()) // 9
}
```

在下面的代码中，函数 foo 运行正常，但函数 bar 会产生恐慌。原因是 s.M 是 (*s.T).M 的简化形式。在编译时，编译器会将此简化形式规范化为原来的完整形式。在运行时，如果 s.T 是 nil，那么对 *s.T 的估值将导致一个恐慌。对 s.T 的两次修改对 *s.T 的估值结果没有影响。

```
package main

type T struct {
    X int
}

func (t T) M() int {
    return t.X
}

type S struct {
    *T
}

func foo() {
    var s = S{T: new(T)}
    var f = s.M // <=> (*s.T).M
    s.T = nil
    f()
}

func bar() {
    var s S
    var f = s.M // panic
    s.T = new(T)
    f()
}

func main() {
    foo()
    bar()
}
```

请注意，接口方法值和通过反射得到的方法值将被延迟扩展为提升的方法值。例如，在下面的程序中，对 s.T.X 的修改对通过反射和接口方式得到的方法值的返回值有影响。

```
package main

import "reflect"

type T struct {
    X int
}

func (t T) M() int {
```

```
    return t.X
}

type S struct {
    *T
}

func main() {
    var s = S{T: new(T)}
    var f = s.M // <=> (*s.T).M
    var g = reflect.ValueOf(&s).Elem().
        MethodByName("M").
        Interface().(func() int)
    var h = interface{M() int}(s).M
    s.T.X = 3
    println( f() ) // 0
    println( g() ) // 3
    println( h() ) // 3
}
```

来源：https://github.com/golang/go/issues/47863

但是，在当前版本（1.19 版本）的官方标准 Go 编译器的实现中存在一个 bug。官方标准 Go 编译器中一个优化会将一些接口方法值过度去虚拟化（de-virtualization），从而导致不正确的结果。比如，下面这个程序应该打印出 2 2，但是目前它却打印出 1 2。

```
package main

type I interface{ M() }

type T struct{
    x int
}

func (t T) M() {
    println(t.x)
}

func main() {
    var t = &T{x: 1}
    var i I = t

    var f = i.M
    defer f() // 2（正确）

    // i.M 将在编译时刻被（错误地）去虚拟化为 (*t).M。
    defer i.M() // 1（错误）

    t.x = 2
}
```

此 bug 将在官方 Go 工具链 1.20 版本中得到修复。

来源：https://github.com/golang/go/issues/52072

2.40 著名的 := 陷阱

让我们看一个简单的程序。

```
package main

import "fmt"
import "strconv"

func parseInt(s string) (int, error) {
     n, err := strconv.Atoi(s)
     if err != nil {
          fmt.Println("err:", err)
          b, err := strconv.ParseBool(s)
          if err != nil {
               return 0, err
          }
          fmt.Println("err:", err)

          if b {
               n = 123
          }
     }
     return n, err
}

func main() {
     fmt.Println(parseInt("true"))
}
```

我们知道函数调用 strconv.Atoi(s) 将返回一个非 nil 的错误，但函数调用 strconv.ParseBool(s) 将返回一个 nil 的错误。那么，调用 parseInt("true") 是否也会返回一个 nil 错误？答案是它将返回一个非 nil 的错误。下面是此程序的输出：

```
err: strconv.Atoi: parsing "true": invalid syntax
err: <nil>
123 strconv.Atoi: parsing "true": invalid syntax
```

等等，在 parseInt("true") 返回之前，err 变量不是在内部代码块中被再声明（re-declared，即修改），并且其值已经被修改为 nil 了吗？这是许多新的 Go 程序员，包括我在内，在刚刚开始使用 Go 语言时曾经遇到的困惑。

为什么函数调用 parseInt("true") 会返回一个非 nil 的错误？因为在内部代码块中声明的变量永远不会是外部代码块中声明的同名变量的再声明。这里，内部声明的 err 变量被初始化为 nil，这并不是对外部声明的 err 变量的修改。外层的 err 变量被设置（初始化）为一个非零值后就没有再被更改过。

Go 社区中有一种声音人为应该把 ... := ... 再声明语法形式从 Go 中删除。但这对 Go 来说，似乎是一个不小的改变。我个人认为，显式标记出再声明的变量是一个更为可行的解决方案。

2.41 官方标准 Go 编译器会报告一些（但并非所有）潜在的由上述 := 陷阱导致的 bug

比如，官方标准 Go 编译器不会报告上一节示例中的 bug，但是它将阻止下面这个很多人认为是合法的 Go 程序被编译成功。

```
package main

import "fmt"

func g() (int, error) {
     return 0, fmt.Errorf("not implemented")
}

func f() (err error) {
     if n, err := g(); err == nil { // line 10
          return                    // line 11
     } else {
          return fmt.Errorf("%w: %d", err, n)
     }
}

func main() {
     fmt.Println(f())
}
```

官方标准 Go 编译器编译此程序时的输出：

```
./main.go:11:3: result parameter err not in scope at return
     ./main.go:10:8: inner declaration of var err error
```

恕我直言，编译器不应该做 go vet 的工作。

2.42 一个 nil 标识符的含义取决于具体的环境

在 Go 中，许多种类的类型的零值都用预声明的 nil 标识符来表示，这些类型种类包括接口类型和一些非接口类型（指针、切片、映射、通道、函数）。

一个非接口值可以包裹（装箱）在一个接口值内，条件是前者的类型实现了后者的类型。非接口类型的零值（可能为 nil）也不是例外。

如果一个接口值内什么也没包裹，那么此接口值就是一个 nil 接口值。如果它包的是一个 nil 的非接口值，它肯定不是 nil 接口值。例如，下面的程序打印了两行 false（这是许多 Go 程序员新手常遇到的另一个常见的困惑），然后打印了一个 true。

```
package main

// 返回结果是一个包裹了 nil 指针的接口值。
func box(p *int) interface{} {
     return p
}

func main() {
     // 左 nil 被视为一个指针零值。
```

```
    // 右 nil 被视为一个接口零值。
    println(box(nil) == nil) // false
    var x interface{} = nil
    var y chan int = nil
    // 在比较前，y 被转换为一个接口值（包裹在一个接口值中）。
    println(x == y) // false

    // 此 nil 被视为一个通道值。
    println(nil == y) // true
}
```

2.43 在 Go 中，某些表达式的估值顺序未定义

在 Go 中，当对复杂表达式、赋值语句或返回语句中的表达式进行估值时，其中所有的函数调用（包括方法调用和通道操作）都按词法顺序从左到右进行估值。除此之外，Go 白皮书并没有定义非函数调用表达式之间以及函数调用表达式和非函数调用表达式之间的估值顺序。

例如，下面的程序打印出了两行不同的内容（使用 Go 工具链 v1.19）。在第一个多值赋值（再声明）语句中，表达式 a 在函数调用 f() 和 g() 之后被估值。但是在第二个多值赋值语句（普通变量声明）中，表达式 a 在函数调用 f() 和 g() 之前被估值。这两种情况都不违背 Go 白皮书。事实上，还有第三种合法可能：3 3 6（如果表达式 a 在函数调用 f() 和 g() 之间被估值）。

我们在实践中不应该写出这种不职业的代码。

```
package main

var a int

func f() int {
    a++
    return a
}

func g() int {
    a *= 2
    return a
}

func main() {
    {
        a = 2
        x, y, z := a, f(), g()
        println(x, y, z) // 6 3 6
    }
    {
        a = 2
        var x, y, z = a, f(), g()
        println(x, y, z) // 2 3 6
    }
}
```

下面是另一个不职业的例子，在这个例子中，CreateT 的调用返回一个 T 值，其 x 字段可能为 53（gccgo 版本 10.2.1-6），也可能为 50（gc 版本 1.19）。

```
package main

import (
	"errors"
	"fmt"
)

type T struct {
	x int
}

func validate(t *T) error {
	if t.x < 0 || t.x > 100 {
		return errors.New("T.x out if range")
	}
	t.x = t.x / 10 * 10
	return nil
}

func CreateT(v int) (T, error) {
	var t = T{x: v}
	return t, validate(&t)
}

func main() {
	var t, _ = CreateT(53)
	fmt.Println(t)
}
```

同样的原因，下面这个程序可能会恐慌退出（当前官方标准编译器 v1.19 的实现），也可能会正常退出。

```
package main

func main() {
	f := func(int) {}
	g := func() int {
		f = nil
		return 1
	}
	f(g())
}
```

2.44 Go 支持循环类型

例如，下面的类型声明都是合法的。

```
type S []S
type M map[int]M
type F func(F) F
```

```
type Ch chan Ch
type P *P
```

下面是一个使用上面最后一个声明的类型的例子。此例子编译和运行都没有问题。

```
package main

func main() {
     type P *P
     var pp = new(P)
     *pp = pp
     _ = ************pp
}
```

下面的这个程序编译和运行也都没有问题。

```
package main

type F func() F

func f() F {
  return f
}

func main() {
  f()()()()()()()()()
}
```

注意，标准 fmt 库包中的打印函数对循环容器类型支持不好。例如，下面的程序将崩溃退出（因为堆栈溢出）。

```
package main

import "fmt"

func main() {
     type S []S
     var s = make(S, 1)
     s[0] = s
     _ = s[0][0][0][0][0][0][0]
     fmt.Println(s) // 产生恐慌
}
```

2.45 几乎每种代码元素都可以被声明为空标识符 _

例如，下面的代码是合法的。

```
const _ = 123
var _ = false
type _ string
func _() {
_: // 一个跳转标签
     return
}
```

```
type T struct{
	_ []int
}
func (T) _() {}
```

包名称和接口方法名称不能为空标识符。

2.46 不使用内置 copy 函数复制切片元素

从 Go 1.17 开始，如果在写代码的时候知道被复制元素的数量，则我们可以使用一种新的方式来复制切片元素。下面的例子展示了这种方式。

```
package main

const N = 128
var x = []int{N-1: 789}

func main() {
	var y = make([]int, N)
	*(*[N]int)(y) = *(*[N]int)(x) // <=> copy(y, x)
	println(y[N-1]) // 789
}
```

但是请注意，当目标切片和源切片存在重叠元素时，对于官方工具链的某些版本（1.17 - 1.17.13、1.18 - 1.18.5 以及 1.19），此方式的实现中有一个 bug。

2.47 不完整的常量描述的自动补全规则中的一个细节

下面这个程序应该打印出啥？

```
package main

const X = 1

func main() {
	const (
		X = X + 1
		Y
	)
	println(X, Y)
}
```

不多说废话，当使用官方标准编译器从 Go 1.18+ 版本时，此程序打印出 2 3，但是当使用官方标准编译器从 Go 1.17- 版本时，此程序打印出 2 2。换句话说，Go 1.17- 的官方标准编译器版本对自动补全的解读是错误的。Go 1.17- 的官方标准编译器版本在补全 Y 常量描述时，使用的是全局声明的 X（其实应该使用局部声明的 X）。

来源：https://github.com/golang/go/issues/49157

Chapter 3

类型转换相关

3.1 如果一个具名类型的底层类型是一个无名类型，那么此具名类型的值可以被隐式的转换为它的底层类型，反之亦然

下面的代码中声明的两个具名类型（Bytes 和 MyBytes）的底层类型都是 []byte。这两个具名类型的值的类型可以相互转换，但转换过程必须是显式的。然而，两个具名类型的值可以隐式地转换为它们的底层类型，[]byte，反之亦然。因为它们的底层类型是一个无名类型。

```
package main

type Bytes []byte
type MyBytes []byte

func f(bs []byte) {}
func g(bs Bytes) {}
func h(bs MyBytes) {}

func main() {
    var x []byte
    var y Bytes
    var z MyBytes

    f(y)
    f(z)

    g(x)
    g(z) // error: cannot use z (type MyBytes) as Bytes
    g(Bytes(z))

    h(x)
    h(y) // error: cannot use y (type Bytes) as MyBytes
    h(MyBytes(y))
}
```

3.2 如果两个具名指针类型的基类型共享相同的底层类型，则这两个类型的值可以间接转换为对方的类型

一般来说，如果两个指针类型的底层类型不同，则这两个指针类型的值不能相互转换。例如，下面代码中的 4 个类型转换都是非法的。

```
package main

type MyInt int
type IntPtr *int      // 底层类型为 *int
type MyIntPtr *MyInt // 底层类型为 *MyInt

func main() {
     var x IntPtr
     var y MyIntPtr
     x = IntPtr(y)        // error
     y = MyIntPtr(x)      // error
     var _ = (*int)(y)    // error
     var _ = (*MyInt)(x) // error
}
```

虽然上述 4 种类型转换无法直接实现，但是可以间接实现。这得益于以下的转换是合法的这一事实。

```
package main

type MyInt int

func main() {
     var x *int
     var y *MyInt
     x = (*int)(y)    // okay
     y = (*MyInt)(x) // okay
}
```

上述两种类型转换是合法的，原因是如果两个无名指针类型的基类型共享相同的底层类型，则这两个类型的值可以相互转换。在上面的例子中，x 和 y 的基类型是 int 和 MyInt，它们共享相同的底层类型 int，所以 x 和 y 可以相互转换为对方的类型。

受益于刚才提到的事实，IntPtr 和 MyIntPtr 的值也可以相互转换，不过这种转换必须是间接的，如下面的代码所示。

```
package main

type MyInt int
type IntPtr *int
type MyIntPtr *MyInt

func main() {
     var x IntPtr
     var y MyIntPtr
     x = IntPtr((*int)((*MyInt)(y)))      // okay
     y = MyIntPtr(((*MyInt))((*int)(x))) // okay
     var _ = (*int)((*MyInt)(y))          // okay
```

```
    var _ = (*MyInt)((*int)(x))          // okay
}
```

3.3 具名的双向通道类型的值不能直接转换为相同元素类型的具名单向通道类型，但是可以间接转换

一个例子：

```
package main

func main() {
    type C chan string
    type Cw chan<- string
    type Cr <-chan string

    var c C
    var w Cw
    var r Cr

    // 下面两行编译不通过。
    // w = Cw(c) // error
    // r = Cr(c) // error

    // 此行编译没问题。
    _ = (chan string)(c)

    // 这两行编译也没问题。
    w = Cw((chan string)(c)) // 间接转换
    r = Cr((chan string)(c)) // 间接转换

    _, _ = w, r
}
```

这样的转换在实践中很少用到，但多了解一些也并非坏事。

3.4 从字符串转换来的字节型切片的容量是未指定的

以下代码中 addPrefixes 函数的实现是不专业的。

```
package main

func addPrefixes(prefixStr string, bss [][]byte) {
    var prefix = []byte(prefixStr)
    println(len(prefix), cap(prefix))
    for i, bs := range bss {
        bss[i] = append(prefix, bs...)
    }
}

func main() {
    var bss = [][]byte {
```

```
            []byte("Java"),
            []byte("C++"),
            []byte("Go"),
            []byte("C"),
        }
        addPrefixes("> ", bss)
        println(string(bss[0])) // > Co+a
        println(string(bss[1])) // > Co+
        println(string(bss[2])) // > Co
        println(string(bss[3])) // > C
}
```

上面这个程序的输出如下（使用官方标准编译器 1.19 编译）：

```
2 8
> Co+a
> Co+
> Co
> C
```

输出的结果并不是符合我们的预期。为什么？因为转换 []byte("> ") 的结果切片的容量是 8（此容量实际上依赖于编译器的实现）。最终，bss 的所有元素都与此转换结果切片共享一些开头的字节。每个 append 调用都覆盖了转换结果切片中的一些字节。

为了修复问题，我们应该截取转换结果生成新切片，以使 bss 的元素间不共享字节。修复后的 addPrefixes 函数实现如下：

```
func addPrefixes(prefixStr string, bss [][]byte) {
        var prefix = []byte(prefixStr)
        prefix = prefix[:len(prefix):len(prefix)] // clip it
        for i, bs := range bss {
                bss[i] = append(prefix, bs...)
        }
}
```

然后输出就会变成预期的样子：

```
> Java
> C++
> Go
> C
```

Chapter 4

值比较相关

4.1 比较两个长度相等并且此长度在写代码的时候是已知的的切片

在 Go 中，切片是不可比较的。但是从 Go 1.17 开始，如果两个切片的元素是可比较的，在这两个切片的长度是相等的并且此长度在写代码的时候是已知的情况下，我们可以使用以下方式来比较两个切片。

```
package main

func main() {
	var x = []int{1, 2, 3, 4, 5}
	var y = []int{1, 2, 3, 4, 5}
	var z = []int{1, 2, 3, 4, 9}

	// 以下两行无法编译
	// _ = x == y
	// _ = x == z

	// 这两行编译没问题。
	println(*(*[5]int)(x) == *(*[5]int)(y)) // true
	println(*(*[5]int)(x) == *(*[5]int)(z)) // false
}
```

4.2 更多比较字节切片的方法

上面介绍的方法适用于元素类型为任何可比较类型的切片。它当然可以用于比较字节切片（长度相等并且在编码时已知）。当然我们也可以使用另外两种方式来比较字节切片 x 和 y ，即使这两个字节切片的长度在编译时是未知的。

- 第一种方式：bytes.Compare(x, y) == 0。
- 第二种方式：string(x) == string(y)。由于官方标准 Go 编译器对这样的比较做了特殊的优化，这种方式不会复制底层字节。实际上，bytes.Equal 函数就是使用这种方式来进行比较的。

这两种方式对两个操作数字节片的长度没有要求。

4.3 如果两个接口的动态类型为同一个不可比较类型，则比较两个接口将产生恐慌

例如，下面的程序打印三个 false，然后因为恐慌崩溃退出。

```
package main

func main() {
	var x interface{} = []int{1, 2}
	var y interface{} = map[string]int{}
	var z interface{} = func() {}

	// The lines all print false.
	println(x == y)
	println(x == z)
	println(x == nil)

	// Each of these line could produce a panic.
	println(x == x)
	println(y == y)
	println(z == z)
}
```

4.4 如何使一个结构体类型不可比较

很简单，只要在此结构体类型中放一个不可比较的字段即可。例如，下面的结构类型都是不可比较的。

```
type T1 struct {
	_ func ()
	x int
}

type T2 struct {
	_ []int
	y bool
}

type T3 struct {
	_ map[int]bool
	z string
}
```

为了避免上面所示的空标识符（_）字段浪费内存，它们的类型应该是零尺寸的类型。

例如，在下面的代码中，类型 Ty 的尺寸小于类型 Tx。

```
package main

import "unsafe"

type Tx struct {
	_ func()
```

```
    x int64
}

type Ty struct {
    _ [0]func()
    y int64
}

func main() {
    var x Tx
    var y Ty
    println(unsafe.Sizeof(x)) // 16
    println(unsafe.Sizeof(y)) // 8
}
```

另外，请尽量避免将零尺寸字段做为结构类型的最终字段。

4.5 数组值的比较是逐个元素进行比较的

当比较两个数组值时，它们的元素将被会一一逐个比较。一旦发现两个对应的元素不相等，整个比较就会停止并返回 false。整个比较也可能因比较两个接口时产生恐慌而停止。

例如，以下代码中的第一个比较结果为 false，但第二个比较导致恐慌。

```
package main

type T [2]interface{}

func main() {
    var a = T{1, func() {}}
    var b = T{2, func() {}}
    println(a == b) // false

    var c = T{2, func() {}}
    var d = T{2, func() {}}
    println(c == d) // 产生恐慌
}
```

4.6 结构体值的比较是逐个字段进行比较的

同样，当比较两个结构体值时，它们的字段将被逐个比较。一旦发现两个对应的字段不相等，整个比较就会停止并返回 false 结果。整个比较也可能会因比较两个接口值时产生恐慌而停止。

例如，以下代码中的第一个比较结果为 false，但第二个比较导致一个恐慌。

```
package main

type T struct {
    x interface{}
    y interface{}
}
```

```
func main() {
	var a = T{x: 1, y: func() {}}
	var b = T{x: 2, y: func() {}}
	println(a == b) // false

	var c = T{x: 2, y: func() {}}
	var d = T{x: 2, y: func() {}}
	println(c == d) // 产生恐慌
}
```

4.7 在结构体值的比较中，_ 字段将被忽略

例如，下面的程序打印 true。

```
package main

type T struct {
	_ int
	x string
}

func main() {
	var x = T{123, "Go"}
	var y = T{789, "Go"}
	println(x == y) // true
}
```

但请注意，如前面其中一节所示，如果一个结构类型包含一个不可比较类型的 _ 字段，那么该结构类型也是不可比较的。

4.8 NaN != NaN，Inf == Inf

在浮点计算中，在某些情况下，计算结果可能是无穷大（Inf）或 NaN（not a number）。例如，在下面的代码中，会产生一个 +Inf 和一个 NaN 值（是的，Inf 值乘以零的结果为一个 NaN 值）。

两个 +Inf（或 -Inf）值彼此相等，但两个 NaN 值总是不相等的。

```
package main

var a = 0.0
var x = 1 / a // +Inf
var y = x * a // NaN

func main() {
	println(x, y)   // +Inf NaN
	println(x == x) // true
	println(y == y) // false
}
```

由于 NaN 值彼此不相等，因此使用 NaN 键从映射中查找条目总是徒劳的。这可以从下面代码中得到证明。

```
package main

var a = 0.0
var x = 1 / a // +Inf
var y = x * a // NaN

func main() {
	var m = map[float64]int{}
	m[y] = 123
	m[y] = 456
	m[y] = 789
	q, ok := m[y]
	println(q, ok, len(m)) // 0 false 3
}
```

事实上，NaN 值与任何值进行比较都会产生一个 false 结果：

```
package main

var a = 0.0
var y = 1 / a * a // NaN

func main() {
	println(y < y)  // false
	println(y == y) // false
	println(y > y)  // false

	println(y < a)  // false
	println(y == a) // false
	println(y > a)  // false
}
```

截止到 Go 1.19，NaN 键对应的条目无法被删除。因此，将键为 NaN 的一个条目放入映射就像将此条目放入黑洞一样无法再取出此条目，尽管 NaN 键对应的条目可以从 for-range 循环中获取：

```
package main

var a = 0.0
var y = 1 / a * a // NaN

func main() {
	var m = map[float64]int{}
	m[y] = 1
	m[y] = 2
	m[y] = 3
	delete(m, y)
	delete(m, y)
	delete(m, y)
	for k, v := range m {
		println(k, v)
	}
}
```

上面这个程序的（一个可能）输出：

```
NaN 3
NaN 1
NaN 2
```

当前有一个添加一个 clear 内置函数的提案。从提议的函数可以用来清除一个映射中的所有条目包括那些键为 NaN 的条目。此提案最快可能会在 Go 1.20 中被采纳。

4.9 如何防止将键含有 NaN 的条目放入一个映射

如果此映射的键类型为 float64，我们可以调用 math.IsNaN(key) 函数来检查键 key 是否为 NaN。如果调用返回 true，此条目不应该被放入映射。但是此方式不适用于键类型为含有浮点数的数组或者结构体类型。一个更为通用的方式是检查比较 key == key 的结果，如果结果为 false，则表明键 key 中含有 NaN，因此我们应该放弃将此条目放入映射。

4.10 使用 reflect.DeepEqual 函数的一些细节

如果传递给 reflect.DeepEqual 函数的一个调用的两个实参的类型不同，则此调用总是返回 false。

当使用 reflect.DeepEqual 函数比较两个不同的指针值（相同类型）时，实际比较的是它们所引用的值（仍然使用 reflect.DeepEqual 函数进行更深的比较）。

如果 reflect.DeepEqual 函数的一个调用的两个实参都处于循环引用链中，那么为了避免无限循环，此调用可能会返回 true。一个例子：

```
package main

import "reflect"

type Node struct{peer *Node}

func main() {
	var x, y, z Node
	x.peer = &x // form a cyclic reference chain
	y.peer = &z // form a cyclic reference chain
	z.peer = &y
	println(reflect.DeepEqual(&x, &y)) // true
}
```

当使用 reflect.DeepEqual 函数比较两个函数值时，只有当两个函数值的类型相同且均为 nil 时，返回结果才为 true。例如，下面的程序打印 true，然后打印出 false。

```
package main

import "reflect"

func main() {
	var x, y func()
	println(reflect.DeepEqual(x, y)) // true
	var z = func() {}
```

```
    println(reflect.DeepEqual(z, z)) // false
}
```

当使用 reflect.DeepEqual 函数比较两个类型相同且长度相等的切片值时，通常会一一比较它们的元素。但是，如果它们对应的第一个元素具有相同的地址，则快速返回 true 而不再比较它们的元素，即使它们的元素是不自等的（例如，非 nil 函数和 NaN 值）。

例如，下面的程序也打印 true 然后 false。

```
package main

import "reflect"

func main() {
    var f = func() {}
    var a = [2]func(){f, f}
    var x = a[:]
    var y = a[:]
    var z = []func(){f, f}
    println(reflect.DeepEqual(x, y)) // true
    println(reflect.DeepEqual(x, z)) // false
}
```

类似地，如果两个映射值共享一个底层哈希表，则使用 reflect.DeepEqual 函数对它们进行比较时，结果也是 true，即使此哈希表包含不自等的值。

```
package main

import (
    "math"
    "reflect"
)

func main() {
    nan := math.NaN()
    println(reflect.DeepEqual(nan, nan)) // false

    m1 := map[int]float64{1: nan}
    m2 := map[int]float64{1: nan}
    m3 := m1

    println(reflect.DeepEqual(m1, m1)) // true
    println(reflect.DeepEqual(m1, m2)) // false
    println(reflect.DeepEqual(m3, m3)) // true
}
```

4.11 bytes.Equal 和 reflect.DeepEqual 函数的返回结果可能不同

reflect.DeepEqual 函数认为 nil 切片和空切片不相等。但是，bytes.Equal 函数认为 nil 字节切片和空字节切片是相等的。这可以从下面的程序中得到证明。

```go
package main

import (
	"bytes"
	"reflect"
)

func main() {
	var x = []byte{}
	var y []byte
	println(bytes.Equal(x, y))       // true
	println(reflect.DeepEqual(x, y)) // false
}
```

4.12 一个使用类型别名做为嵌入字段的 bug

Go 1.9 引入了自定义类型别名声明。但是，随之也引入了一个 bug。直到官方 Go 工具链 1.18，此 bug 才得以修复。

此 bug 在下面这个程序中暴露无余。它应该打印 `false`，但是使用 1.18 之前的官方 Go 工具链版本时，它却打印出 `true`。

```go
package main

type Int = int

type A = struct{ int }
type B = struct{ Int }

func main() {
	var x, y interface{} = A{}, B{}
	println(x == y) // true（使用 Go toolchain 1.17-）
}
```

来源：https://github.com/golang/go/issues/24721

Chapter 5

运行时（runtime）相关

5.1 在官方标准编译器实现中，映射值的底层数组永远不会缩容

官方标准运行时在一个映射值中维护了一个底层数组，用来存储此映射值中的所有条目。随着越来越多的条目被放入此映射值中，它的底层数组也会随之扩容。但是它从不缩容。这意味着对于一个曾经存储了数百万条条目的映射值，即使从中删除了所有条目，它的底层数组依然保持着可以容纳数百万条条目的容量而无需再次扩容。

那么如何释放一个映射值的底层数组所占用的内存呢？只需把此映射值置为 nil，或者给它赋值一个新的映射值。

5.2 64 位整数的地址对齐问题

当对一个 64 位整型上进行 64 位原子操作时，此 64 位整型的地址必须是 8 字节对齐的。在 64 位体系架构中，64 位整型总是 8 字节对齐的，因此该条件在 64 位体系架构中总是满足。但是在 32 位体系架构中不总是这样。

sync/atomic 标准库文档指出，一个合格的 Go 编译器应当确保声明的变量中以及在开辟的结构体、数组和切片中的首个 64 位整数值是 8 字节对齐的。“开辟的”是什么意思？我们可以把“开辟的”值理解为一个声明的变量、一个由内置的 make 函数返回的值、或者是由内置 new 函数返回的指针的引用的值。

在下面的示例中，第一个 AddX 方法调用是安全的，因为 t.x 总是 8 字节对齐的（即使在 32 位体系架构中）。

然而，第二个 AddX 方法调用在 32 位体架构中却是不安全的。它可能会造成一个恐慌，因为 s.t.x 并不能保证总是 8 字节对齐的。

```
package main

import "sync/atomic"

type T struct {
    x uint64
}
```

```
func (t *T) AddX(dx uint64) {
    atomic.AddUint64(&t.x, dx)
}

type S struct {
    y int32
    t T
}

func main() {
    var t T
    t.AddX(1) // 安全的，即使在 32 位体系结构中

    var s S
    s.t.AddX(1) // 在 32 位体系结构中可能会产生恐慌
}
```

我们需要意识到的一点是，官方 Go 编译器（gc 以及 gccgo）保证在任何体系架构中，32 位或者 64 位整数总是 4 字节对齐的。实际上 sync.WaitGroup 类型的实现 曾经依赖过这一点。

sync.WaitGroup 需要两个字段。通常它们应该被定义为：

```
type WaitGroup struct {
   state uint64
   sema  uint32
}
```

其中，state 需要参与 64 位原子操作。然而在 32 位体系结构中，它的地址并不能保证是 8 字节对齐。因此，sync.WaitGroup 类型被定义成了

```
type WaitGroup struct {
    state1 [3]uint32
}
```

在运行时中，sync.WaitGroup 中的 state1 既可能是 4 字节对齐，也可能是 8 字节对齐。如果是 8 字节对齐，state1 中的前两个元素的组合可以看成是最初的 state 字段，第三个元素则被视为 sema 字段。否则，state1 的后两个元素的组合可以看成是最初的 state 字段，第一个元素则被视为 sema 字段。

5.3 利用 go vet 命令来检测不推荐的值复制

标准库中有一些类型的值不推荐被复制（比如标准库中的 sync 包中的类型）。当在代码中复制了这些类型的值的时候，官方的 go vet 命令会输出警告。我们称这种类型为 noCopy 类型。

目前，并没有禁止值复制的专门语法。go vet 通过检测指针类型 *T 是否拥有 Lock() 方法以及 Unlock() 方法来判断类型 T 是否是一个 noCopy 类型。

举个例子，go vet 命令将对下面代码中的赋值发出警告（在 Go 中，赋值是值复制）。

```
package main

type T struct{}
```

```
func (*T) Lock() {}
func (*T) Unlock() {}

func main() {
    var t T
    _ = t // warning: assignment copies lock value to _
}
```

带有 noCopy 类型的字段的结构体类型或者元素是 noCopy 类型的数组类型同样也是 noCopy 类型。举个例子：

```
package main

type T struct{}

func (*T) Lock() {}
func (*T) Unlock() {}

type S struct {
    t T
}

func main() {
    var s S
    _ = s // warning: assignment copies lock value to _

    var a [8]T
    _ = a // warning: assignment copies lock value to _
}
```

5.4 标准库中其它一些其值不应该被复制的类型

在标准库中，除了的 sync 库包中的类型，其它库包中的某些类型的值也不应该被复制，比如 bytes.Buffer 和 strings.Builder。

通常，如果一个值 x 引用着某些值，而这些被引用的值中至少有一个不应该被多个值引用，则值 x 不应该被拷贝。

5.5 一些零值在内存中可能包含非零字节

举个例子：

```
package main

import (
    "fmt"
    "reflect"
    u "unsafe"
)

var s = "abc"[0:0]
```

```
func main() {
     header := (*reflect.StringHeader)(u.Pointer(&s))
     if s == "" {
          fmt.Printf("%#v\n", *header)
     }
}
```

reflect.StringHeader 类型代表 string 类型的内部结构。

运行上面的程序，会得到类似如下的输出：

```
reflect.StringHeader{Data:0x4957ec, Len:0}
```

从输出中我们可以看到，空字符串 s 的 Data 字段并不是零值，但是 Go 运行时依然将 s 当做是一个空字符串。实际上，长度为 0 就足以证明它是一个空字符串。

5.6 一些值的内存地址可能会在运行中改变

在官方标准 Go 运行时实现中，协程的栈会在运行的过程中根据需要扩容或缩容。当栈大小发生改变时，分配在该栈上的值的地址也可能会发生改变。

举个例子，下面的程序很可能会打印出两个不同的地址。

```
package main

//go:noinline
func f(i int) byte {
     var a [1 << 12]byte
     return a[i]
}

func main() {
     var x int
     println(&x)
     f(100) // 使栈扩容
     println(&x)
}
```

5.7 官方标准 Go 运行时在系统内存耗尽时表现得很糟糕

当系统内存被耗尽以至于需要进行内存交换，Go 运行时并不总是会让程序崩溃，而是几乎耗尽所有 CPU 资源，导致系统 UI 经常几乎不再响应。当这种情况发生后，常常需要重启系统来摆脱这种困境。

举个例子，有时在调试程序的过程中，如果我们意外的写下了如下代码片段并调试运行之，系统可能会卡住。

(警告：如果你想在机器上运行这段代码，请先保存你当前的工作文件!)

```
package main

var s = "1234567890"

func condition() bool {
     return true // 为了演示目的而简化
```

```
}

func main() {
     for condition() {
          s += s
          println(len(s))
     }
}
```

在调试时将循环步长设置成一个合理的足够大的数字是一个好主意，比如：

```
func main() {
     for range [10]struct{}{} {
          if condition() {
               break
          }

          s += s
          println(len(s))
     }
}
```

5.8 目前，一个 runtime.Goexit 调用会取消已经发生的恐慌

举个例子，下面这个程序运行时将正常退出。如果 runtime.Goexit() 代码行被删除掉，则此程序运行时将崩溃退出。

```
package main

import "runtime"

func wroker(c chan int) {
     defer close(c)
     defer runtime.Goexit()

     // ... 做一些其它工作

     panic("bye")
}

func main() {
     c := make(chan int)
     go wroker(c)
     <-c
}
```

未来的官方标准 Go 编译器可能会改变当前这种行为实现。

来源：https://github.com/golang/go/issues/35378

5.9 同一个协程中可能有多个恐慌共存

两个共存的恐慌必须存在于两个不同的函数调用深度中，并且新的恐慌必须存在于更深层次的函数调用中。如果深层的恐慌蔓延到浅层的函数调用中，且这里原来已经存在着另一个恐慌，则深层的恐慌会取代浅层的恐慌。

举个例子，对于下面的程序，在它运行的时候，将会有两个活跃的恐慌共存。之后，第二个恐慌会取代第一个恐慌，并最终被恢复。

```
package main

func main() {
     defer func() {
          println("Panic", recover().(int), " 被恢复了。")
     }()
     defer println(" 现在，恐慌 2 替代了恐慌 1。")
     defer func() {
          defer println(" 现在有两个恐慌共存。")
          panic(2)
     }()
     defer println(" 目前只有一个恐慌存在。")
     panic(1)
}
```

上面程序的输出如下：

```
目前只有一个恐慌存在。
现在有两个恐慌共存。
现在，恐慌 2 替代了恐慌 1。
Panic 2 被恢复了。
```

5.10 当前的 Go 语言白皮是（1.19 版本）并没有很好的解释 panic/recover 机制

根据当前的规范，下面代码中标记了 no-op 的行应该恢复恐慌 1，但是实际上它并没有。原因是只有最新产生的恐慌可以被恢复。

```
package main

import "fmt"

func main() {
     defer func() {
          fmt.Print(recover())
     }()
     defer func() {
          defer func() {
               fmt.Print(recover())
          }()
          defer recover() // no-op
          panic(2)
     }()
```

```
        panic(1)
}
```

上面的程序打印了 21。如果我们将“no-op”行改为非延迟调用，则取而代之打印的是 2<nil>。

请阅读详解恐慌和恢复原理一文来透彻地理解 Go 中的 panic/recover 机制。

Chapter 6

标准包相关

6.1 在 fmt.Errorf 调用中使用 %w 格式描述来构建错误链

当使用 fmt.Errorf 函数来包装一个更深的错误时，建议使用 %w 而不是 %s 格式描述，以避免丢失被包裹的错误信息。

例如，在下面的代码中，Bar 实现比 Foo 实现要好，因为调用者可以判断返回的错误是否是由是由某个特定的错误（这里是 ErrNotImpl）引发的。

```
package main

import (
	"errors"
	"fmt"
)

var ErrNotImpl = errors.New("not implemented yet")

func doSomething() error {
	return ErrNotImpl
}

func Foo() error {
	if err := doSomething(); err != nil {
		return fmt.Errorf("Foo: %s", err)
	}
	return nil
}

func Bar() error {
	if err := doSomething(); err != nil {
		return fmt.Errorf("Bar: %w", err)
	}
	return nil
}

func main() {
```

```
    println(errors.Is(Foo(), ErrNotImpl)) // false
    println(errors.Is(Bar(), ErrNotImpl)) // true
}
```

在用户代码中，我们应该尝试使用 errors.Is 函数，而不是使用直接比较来判断错误的起因。

6.2 fmt.Println、fmt.Print 和 print 函数之间的细微差异

fmt.Println 函数(和 println)将在任何两个相邻的参数之间输出一个空格。fmt.Print 函数只在两个相邻的参数都不是字符串才会这么做。print 函数永远不会在参数之间输出空格。

这可以通过以下代码来证明。

```
package main

import "fmt"

func main() {
    // 123 789 abc xyz
    println(123, 789, "abc", "xyz")
    // 123 789 abc xyz
    fmt.Println(123, 789, "abc", "xyz")
    // 123 789abcxyz
    fmt.Print(123, 789, "abc", "xyz")
    println()
    // 123789abcxyz
    print(123, 789, "abc", "xyz")
    println()
}
```

6.3 reflect.Type/Value.NumMethod 方法将统计接口的未导出的方法

对于非接口类型和值，reflect.Type.NumMethod 和 reflect.Value.NumMethod 方法不计算未导出的方法。但是对于接口类型和值，未导出的方法将会被计算在内。

这可以通过下面的代码来证明。

```
package main

import "reflect"

type I interface {
    m()
    M()
}

type T struct {}
func (T) m() {}
```

```
func (T) M() {}

func main() {
     var t T
     var i I = t
     var vt = reflect.ValueOf(t)
     var vi = reflect.ValueOf(&i).Elem()
     println(vt.NumMethod()) // 1
     println(vi.NumMethod()) // 2
}
```

6.4 如果两个切片的元素类型不同，则这两个切片的值不能被转换为对方的类型，但这个规则有一个漏洞

例如，下面代码中的两个转换都是非法的。

```
package main

type MyByte byte
var x []MyByte
var y []byte

func main() {
     x = []MyByte(y) // error
     y = []byte(x)   // error
}
```

这个规则有一个漏洞：如果一个切片的元素类型的底层类型是 byte（比如上例中显示的 MyByte 类型），那么我们可以使用 reflect.Value.Bytes 方法来此（字节）切片转换为 []byte。示例：

```
package main

import "reflect"

type MyByte byte
Value.Bytes
func main() {
     var x = make([]MyByte, 128)
     var y []byte
     y = reflect.ValueOf(x).Bytes()
     y[127] = 123
     println(x[127]) // 123
}
```

来源：https://github.com/golang/go/issues/24746

6.5 不要将 strings 和 bytes 标准库包中的 TrimLeft 函数误用为 TrimPrefix

TrimLeft 函数的第二个参数是一个码点集，第一个参数中包含在此码点集中的任何起始 Unicode 码点将被剪除，这与 TrimPrefix 函数有很大不同。

下面的程序显示了这两者的区别。

```
package main

import "strings"

func main() {
	var hw = "DoDoDo!"
	println(strings.TrimLeft(hw, "Do"))   // !
	println(strings.TrimPrefix(hw, "Do")) // DoDo!
}
```

TrimRight 和 TrimSuffix 函数也类似。

6.6 json.Unmarshal 函数接受不区分大小写的键匹配

例如，下面的程序打印 bar，而不是 foo。

```
package main

import (
	"encoding/json"
	"fmt"
)

type T struct {
	HTML string `json:"HTML"`
}

var s = `{"HTML": "foo", "html": "bar"}`

func main() {
	var t T
	if err := json.Unmarshal([]byte(s), &t); err != nil {
		fmt.Println(err)
		return
	}
	fmt.Println(t.HTML) // bar
}
```

json.Unmarshal 函数的文档提到：preferring an exact match but also accepting a case-insensitive match（倾向于精确匹配，但也接受不区分大小写的匹配）。所以我个人认为这是 json.Unmarshal 函数的一个 bug，但 Go 核心团队并不这么认为。

6.7 在结构体字段标签中，键值对中的空格将不会被忽略掉

例如，下面的程序将打印 {" foo":""}。Foo 字段的标签中的 omitempty 选项是一个拼写错误，它被视为与 omitempty 不同。而且 Foo 字段的标签键是 " foo"，而不是 "foo"。

```
package main

import (
    "encoding/json"
    "fmt"
)

type T struct {
    Foo string `json:" foo, omitempty"`
    Bar string `json:"bar,omitempty"`
}

func main() {
    var t T
    var s, _ = json.Marshal(t)
    fmt.Printf("%s", s) // {" foo":""}
}
```

6.8 如何尝试尽早运行一个自定义的 init 函数？

假设项目的模块是 x.y/app。添加一个 x.y/app/internal/init 包，并在 init 包中放置一个 init 函数。然后在 main 包中引入此 init 包。init 包将在一些核心包（比如 runtime 标准包）加载完成之后、但在其它包之前加载。

6.9 如何解决代码包的循环依赖问题？

Go 不支持循环的软件包依赖关系。如果包 foo 导入包 bar，那么包 bar 不能导入包 foo。

有时，我们可能会遇到这样的情况：两个包确实需要使用对方导出的标识符。我们应该如何处理这种情况呢？有两种方法来解决这个问题。

一种方法是将两个包合并成一个更大的包，这样循环依赖的问题就会消失。这种方法总是有效的。

另一种方法是把这两个包拆成更小的包，以消除循环依赖关系。这种方法并非总是有效。

6.10 在调用 os.Exit 函数后，已经注册的延迟函数调用将不会被执行

例如，以下代码中的延迟调用 cleanup() 完全没有意义。

```
func run() {
    defer cleanup()
```

```
    if err := doSomething(); err != nil {
        log.Println(err)
        os.Exit(1)
    }

    os.Exit(0)
}
```

请注意，log.Fatal 函数调用了 os.Exit 函数。所以在调用 log.Fatal 函数后，已注册的延迟调用也不会被执行。

6.11 如何让 main 函数返回一个退出代码？

没有办法做到这一点。Go 语法不支持这个。然而，我们可以用以下方式来模拟一个返回退出代码的 main 函数。

```
import "os"

func main() {
    os.Exit(realMain())
}

func realMain() int {
    ... // 做一些事情，必要时返回非零退出代码

    return 0
}
```

6.12 尽量不要使用导出变量

我们应该尽量避免从我们维护的包中导出变量（特别是 error 值）。标准包中有很多导出的 error 变量值，这实际上是一种不好的做法。我个人建议使用以下方式来声明 error 值。

```
package foo

type errType int

const (
    ErrA errType = iota
    ErrB
    ErrC
    errCount
)

func (e errType) Error() string {
    if e < 0 || e >= errCount {
        panic("invalid error number")
    }
    return errDescriptions[e]
}
```

```
var _ = [1]int{}[len(errDescriptions) - int(errCount)]

var errDescriptions = [...]string {
     ErrA: "error A",
     ErrB: "error B",
     ErrC: "error C",
}
```

通过这种方式，foo 包的用户不能修改此包中声明的错误值。